From coinductive proofs to exact real arithmetic

Ulrich Berger

University of Wales Swansea, Swansea, SA2 8PP, Wales UK
u.berger@swansea.ac.uk

Abstract. We give a coinductive characterisation of the set of continuous functions defined on a compact real interval, and extract certified programs that construct and combine exact real number algorithms with respect to the binary signed digit representation of real numbers. The data type corresponding to the coinductive definition of continuous functions consists of finitely branching non-wellfounded trees describing when the algorithm writes and reads digits. This is a pilot study in using proof-theoretic methods for obtaining certified algorithms in exact real arithmetic.

1 Introduction

Most of the recent work on exact real number computation describes algorithms for functions on certain exact representations of the reals (for example streams of signed digits [MRE07,GNSW07] or linear fractional transformations [EH02]) and proves their correctness using a certain proof method (for example coinduction [CDG06,Ber07,BH07]). Our work has a similar aim, and builds on the work cited above, but there are two important differences. The first is *methodological*: we do not 'guess' an algorithm and then verify it, instead we *extract* it from a proof, by some (once and for all) proven correct method. That this is possible in principle is well-known. Here we want to make the case that it is also feasible, and that interesting and nontrivial new algorithms can be obtained (see also [Sch08] for related work on program extraction in constructive analysis). The second difference is *algorithmic*: we do not represent a real function by a function on representations of reals, but by an infinite tree-like structure that contains not only information about the real function as a point map, but also and foremost information about the modulus of continuity. Since the representing tree is a pure data structure (without function component) a lazy programming language, like Haskell, will memoise computations which may improve performance in certain situations. A similar representation of stream transformers has been studied in [GHP06].

We show how to extract from constructive proofs tree structures that represent algorithms for continuous real functions defined on a compact interval w.r.t. the signed digit representation of real numbers. A crucial ingredient in the proofs is a coinductive definition of the notion of uniform continuity. Although, classically, continuity and uniform continuity coincide in our setting, it is a suitable constructive definition of *uniform* continuity which matters for our purpose.

For convenience, we consider as domain and range of our functions only the compact interval $\mathbb{I} := [-1,1] = \{x \in \mathbb{R} \mid |x| \leq 1\}$ and, for the purpose of this introduction, only unary functions. However, in the paper we will also look at functions of several variables, in which case one has to deal with the non-trivial question in which order the digits of the input streams are to be consumed in order to obtain optimal performance.

We let $\mathrm{SD} := \{-1,0,1\}$ be the set of *signed digits*. By SDS we denote the set of all infinite streams $a = a_0 : a_1 : a_2 : \ldots$ of signed digits $a_i \in \mathrm{SD}$. A signed digit stream $a \in \mathrm{SDS}$ represents the real number

$$\sigma(a) := \sum_{i \geq 0} a_i 2^{-(i+1)} \in \mathbb{I}$$

A function $f : \mathbb{I} \to \mathbb{I}$ is *represented* by a stream transformer $\hat{f} : \mathrm{SDS} \to \mathrm{SDS}$ if $f \circ \sigma = \sigma \circ \hat{f}$

We give a coinductive definition of uniform continuity (u.c.) that allows us to extract from a constructive proof of the u.c. of a function $f : \mathbb{I} \to \mathbb{I}$ an algorithm for a stream transformer $\hat{f}$ representing f. Furthermore we show directly and constructively that the coinductive notion of u.c. is closed under composition. The extracted algorithms are represented by finitely branching non-wellfounded trees which, if executed in a lazy programming language, give rise to memoised algorithms. These trees turn out to be a generalisation of the data structure studied in [GHP06], and the extracted program from the proof of closure under composition is a generalisation of the tree composing program defined in [GHP06].

In Section 2 we briefly review inductive and coinductive sets defined by monotone set operators. We give some simple examples, among them a coinductive characterisation of the real numbers in the interval $\mathbb{I}$. The method of program extraction from proofs involving induction and coinduction is discussed informally, but in some detail in Section 3. The earlier examples are continued and, for example, a program transforming fast Cauchy representations into signed digit representations is extracted from a coinductive proof. We also show how program extraction can be implemented in the functional programming language Haskell. As Haskell's syntax is very close to the usual mathematical notation for data and functions we hope that also readers not familiar with Haskell will be able to understand the code. In Section 4 the coinductive characterisation of real numbers is generalised to real functions, and closure under composition is proven. In Section 5 the positive effect of memoisation is demonstrated by a case study on iterated logistic maps. In Section 6 we extract a program for integration from a proof that has some similar to the programs described in [Sim98] and [Scr08], although it works on a different data structure.

Notations. By $\mathcal{P}(X)$ we denote the powerset of a set X, and by $X \to Y$ (sometimes also written Y^X) the set of all functions from X to Y. $X \to Y \to Z$ is shorthand for $X \to (Y \to Z)$. We will sometimes write $X(x)$ instead of $x \in X$ and $f\colon X \to Y$ instead of $f \in X \to Y$. If $f\colon X \to Y$ and Z is a set, then $f[Z] := \{f(x) \mid x \in X \cap Z\}$.

2 Induction and coinduction

We briefly discuss inductive and coinductive definitions as least and greatest fixed points of monotone set operators, and the corresponding induction and coinduction principles. The results in this section are standard and can be found in many mathematics and computer science texts. For example in [BS07] least and greatest fixed points are studied in the framework of the modal mu-calculus.

An operator $\Phi\colon \mathcal{P}(U) \to \mathcal{P}(U)$, where U is an arbitrary set, is *monotone* if for all $X, Y \subseteq U$

$$\text{if } X \subseteq Y, \text{ then } \Phi(X) \subseteq \Phi(Y)$$

A set $X \subseteq U$ is Φ-*closed* (or a pre-fixed point of Φ) if $\Phi(X) \subseteq X$. Since $\mathcal{P}(U)$ is a complete lattice, Φ has a least fixed point $\mu\Phi$ (Knaster-Tarski Theorem). For the sake of readability we will sometimes write $\mu X.\Phi(X)$ instead of $\mu\Phi$. $\mu\Phi$ can be defined as the least Φ-closed subset of U. Hence we have the *closure principle* for $\mu\Phi$, $\Phi(\mu\Phi) \subseteq \mu\Phi$ and the *induction principle* stating that for every $X \subseteq U$, if $\Phi(X) \subseteq X$, then $\mu\Phi \subseteq X$. It can easily be shown that $\mu\Phi$ is even a *fixed point* of Φ, i.e. $\Phi(\mu\Phi) = \mu\Phi$. For monotone operators $\Phi, \Psi\colon \mathcal{P}(U) \to \mathcal{P}(U)$ we define

$$\Phi \subseteq \Psi \;:\Leftrightarrow\; \forall X \subseteq U\, \Phi(X) \subseteq \Psi(X)$$

It is easy to see that the operation μ *is monotone*, i.e. if $\Phi \subseteq \Psi$, then $\mu\Phi \subseteq \mu\Psi$. Using monotonicity of μ one can easily prove, by induction, a principle, called *strong induction*. It says that, if $\Phi(X \cap \mu\Phi) \subseteq X$, then $\mu\Phi \subseteq X$.

Dual to inductive definitions are *coinductive definitions*. A subset X of U is called Φ-*coclosed* (or a post-fixed point of Φ) if $X \subseteq \Phi(X)$. By duality, Φ has a largest fixed point $\nu\Phi$ which can be defined as the largest Φ-coclosed subset of Φ. Similarly, all other principles for induction have their coinductive counterparts. To summarise, we have the following principles:

Fixed point	$\Phi(\mu\Phi) = \mu\Phi$ and $\Phi(\nu\Phi) = \nu\Phi$.
Monotonicity	if $\Phi \subseteq \Psi$, then $\mu\Phi \subseteq \mu\Psi$ and $\nu\Phi \subseteq \nu\Psi$.
Induction	if $\Phi(X) \subseteq X$, then $\mu\Phi \subseteq X$.
Strong induction	if $\Phi(X \cap \mu\Phi) \subseteq X$, then $\mu\Phi \subseteq X$.
Coinduction	if $X \subseteq \Phi(X)$, then $X \subseteq \nu\Phi$.
Strong coinduction	if $X \subseteq \Phi(X \cup \nu\Phi)$, then $X \subseteq \nu\Phi$.

Example (natural numbers) Define $\Phi : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{R})$ by

$$\Phi(X) := \{0\} \cup \{y + 1 \mid y \in X\}$$

Then $\mu\Phi = \mathbb{N} = \{0, 1, 2, \ldots\}$. We consider this as the *definition* of the natural numbers. The induction principle is logically equivalent to the usual zero-successor-induction on $\mathbb{N}$: if $X(0)$ (base) and $\forall x\,(X(x) \to X(x+1))$ (step), then $\forall x \in \mathbb{N}\, X(x)$. Strong induction weakens the step by restricting x to the natural numbers: $\forall x \in \mathbb{N}\,(X(x) \to X(x+1))$.

Example (signed digits and the interval $[-1,1]$) For every signed digit $d \in \mathrm{SD}$ we set $\mathbb{I}_d := [d/2-1/2, d/2+1/2] = \{x \in \mathbb{R} \mid |x-d/2| \leq 1/2\}$. Note that $\mathbb{I}$ is the union of the $\mathbb{I}_d$ and every sub interval of $\mathbb{I}$ of length $\leq 1/2$ is contained in some $\mathbb{I}_d$. We define an operator $\mathcal{J}_0 : \mathcal{P}(\mathbb{R}) \to \mathcal{P}(\mathbb{R})$ by

$$\mathcal{J}_0(X) := \{x \mid \exists d \in \mathrm{SD}\,(x \in \mathbb{I}_d \wedge 2x - d \in X)\}$$

and set $\mathrm{C}_0 := \nu\mathcal{J}_0$. Since clearly $\mathbb{I} \subseteq \mathcal{J}_0(\mathbb{I})$, it follows, by coinduction, that $\mathbb{I} \subseteq \mathrm{C}_0$. On the other hand $\mathrm{C}_0 \subseteq \mathbb{I}$, by the fixed point property. Hence $\mathrm{C}_0 = \mathbb{I}$. The point of this definition is, that the proof of "$\mathbb{I} \subseteq \mathcal{J}_0(\mathbb{I})$" has an interesting computational content: $x \in \mathbb{I}$ must be given in such a way that it is possible to find $d \in \mathrm{SD}$ such that $x \in \mathbb{I}_d$. This means that $d/2$ is a *first approximation* of x. The computational content of the proof of "$\mathbb{I} \subseteq \mathrm{C}_0$", roughly speaking, iterates the process of finding approximations to x ad infinitum, i.e. it computes a *signed digit representation* of x as explained in the introduction, that is, a stream a of signed digits with $\sigma(a) = x$.

3 Program extraction from proofs

In this section we briefly explain how we extract programs from proofs. Rather than giving a technical definition of the method and a rigorous correctness proof (which will be the subject of a separate paper) we explain it by means of simple examples, which hopefully provide a good intuition also for non-experts, and then make some general remarks concerning the computational content of induction and coinduction. The method of program extraction we are using is based on an extension and variation of Kreisel's *modified realisablity* [Kre59]. The *extension* concerns the addition of inductive and coinductive predicates. Realisability for such predicates has been studied previously, in the slightly different context of **q**-realisability by Tatsuta [Tat98]. The *variation* concerns the fact that we are treating the first-order part of the language (i.e. quantification over individuals) in a 'uniform' way, that is, realisers do not depend on individuals. This is similar to the common uniform treatment of second-order variables [Tro73]. The argument is that an arbitrary subset of a set is such an abstract (and even vague) entity so that one should not expect an algorithm to depend on it. With a similar argument one may argue that individuals of an abstract mathematical structure (reals, model of set-theory, etc.) are unsuitable as inputs for programs. But which data should a program then depend on? The answer is: on data defined by the 'propositional skeletons' of formulas and proofs.

Example (parity) Let us extract a program from a proof of

$$\forall x\,(\mathbb{N}(x) \Rightarrow \exists y\,(x = 2y \vee x = 2y+1)) \tag{1}$$

where the variable x ranges over real numbers and the predicate $\mathbb{N}$ is defined as in the example in Section 2, i.e. $\mathbb{N}$ is the least set of real numbers such that

$$\mathbb{N}(x) \Leftrightarrow x = 0 \vee \exists y\,(\mathbb{N}(y) \wedge x = y+1) \tag{2}$$

The data type corresponding to (2) is obtained by

- replacing $\mathbb{N}(t)$ by Nat (a name for the data type to be defined),
- replacing other atomic formulas by the unit or 'void' type $\mathbf{1}$,
- deleting all quantifiers,
- replacing $\vee$ by $+$ (disjoint sum) and $\wedge$ by $\times$ (cartesian product),
- carrying out obvious simplifications (e.g. replacing Nat $\times \mathbf{1}$ by ρ).

Hence we obtain the definition of a data type Nat as the least solution of the equation

$$\mathrm{Nat} = \mathbf{1} + \mathrm{Nat}$$

In Haskell we could define this as

```
data Nat = Zero | Succ Nat
```

but it is more convenient to use the built-in data type of integers instead

```
type Nat = Int  -- (comment) we use only non-negative integers
```

Applying the same procedure to the formula (1) we see that a program extracted from a proof of it will have type Nat $\to \mathbf{1} + \mathbf{1}$. Identifying the two-element type $\mathbf{1} + \mathbf{1}$ with the Booleans we get the Haskell signature

```
parity :: Nat -> Bool
```

The definition of `parity` can be extracted from the obvious inductive proof of (1): For the base $x = 0$, we take $y = 0$ to get $x = 2y$. In the step, $x+1$, we have, by i.h. some y with $x = 2y \vee x = 2y + 1$. In the first case $x + 1 = 2y + 1$, in the second case $x + 1 = 2(y + 1)$. The Haskell program extracted from this proof is

```
parity :: Nat -> Bool
parity 0     = True
parity (x+1) = case parity x of {True -> False ; False -> True}
```

i.e. `parity (x+1) = not (parity x)`. If we wish to compute not only the parity, but as well the rounded down half of x (i.e. quotient and remainder), we just need to relativise in (1) the quantifier $\exists y$ to $\mathbb{N}$ (i.e. $\forall x\,(\mathbb{N}(x) \Rightarrow \exists y\,(\mathbb{N}(y) \wedge (x = 2y \vee x = 2y + 1)))$) and use in the proof the fact that $\mathbb{N}$ is closed under the $(+1)$ operation. The extracted program is then

```
parity1 :: Nat -> (Nat,Bool)
parity1 0     = (0,True)
parity1 (x+1) = case parity1 x of {(y,True) -> (y,False) ;
                                   (y,False) -> (y+1,True)}
```

This example shows that we can get meaningful computational content despite ignoring the first-order part of a proof, and we can fine tune the amount of computational information we extract from a proof by simple modifications. Note also that in the proofs we used arithmetic operations on the reals and their arithmetic laws without implementing or proving them. Since these laws can be written as equations (or conditional equations) their associated type is void.

Hence it is only their truth that matters, allowing us to treat them as ad-hoc axioms without bothering to derive them from basic axioms. Note that a formula that does not contain a disjunction has always a void type and can therefore be taken as an axiom as long as it is true.

Example (from Cauchy to signed digits) In the second example of Section 2 we defined the set C_0 as the largest set of real numbers such that

$$\mathrm{C}_0(x) \Leftrightarrow \exists d\,(\mathrm{SD}(d) \wedge \mathbb{I}_d(x) \wedge \mathrm{C}_0(2x-d)) \tag{3}$$

Since $\mathrm{SD}(d)$ is shorthand for $x = -1 \vee x = 0 \vee x = 1$, and $\mathbb{I}_d(x)$ is shorthand for $|x-d| \le 1/2$, the corresponding type is the largest solution of the equation

$$\mathrm{SDS} = (\mathbf{1} + \mathbf{1} + \mathbf{1}) \times \mathrm{SDS} \tag{4}$$

Identifying the type $\mathbf{1} + \mathbf{1} + \mathbf{1}$ with SD we obtain $\mathrm{SDS} = \mathrm{SD} \times \mathrm{SDS}$, i.e. SDS is the type of infinite streams of signed digits. We could model SDS in Haskell by a user defined data type (`data SDS = Cons SD SDS`), but we find it more convenient to use the built-in list data type, which contains finite and - Haskell having a lazy semantics - also infinite lists, i.e. streams. We also implement SDS by the built-in type of rational numbers (using only $-1, 0, 1$).

```
type SD  = Rational      -- use only -1,0,1
type SDS = [SD]          -- use as codata, i.e. largest fixed point
```

We wish to extract a program that computes a signed digit representation of $x \in \mathbb{I}$ from a fast rational Cauchy sequence converging to x. Set

$$\mathbb{Q}(x) := \exists n, m, k\,(\mathbb{N}(n) \wedge \mathbb{N}(m) \wedge \mathbb{N}(k) \wedge x = (n-m)/k)$$
$$A(x) := \forall n\,(\mathbb{N}(n) \Rightarrow \exists q\,(\mathbb{Q}(q) \wedge |x-q| \le 2^{-n}))$$

Constructively, $A(x)$ means that there is a fast Cauchy sequence of rational numbers converging to x.

Lemma 1.

$$\forall x\,(\mathbb{I}(x) \wedge A(x) \Rightarrow \mathrm{C}_0(x)) \tag{5}$$

Proof. We show $\mathbb{I} \cap A \subseteq \mathrm{C}_0$ by coinduction, i.e. we show $\mathbb{I} \cap A \subseteq \mathcal{J}_0(\mathbb{I} \cap A)$. Assume $\mathbb{I}(x)$ and $A(x)$. We have to show (constructively!) $\mathcal{J}_0(\mathbb{I} \cap A)(x)$, i.e. we need to find $d \in \mathrm{SD}$ such that $x \in \mathbb{I}_d$ and $2x - d \in \mathbb{I} \cap A$. Since, clearly $A(2x-d)$ holds for any $d \in \mathrm{SD}$, and $x \in \mathbb{I}_d$ holds iff $2x - d \in \mathbb{I}$, we only need to worry about x lying in $\mathbb{I}_d$. The assumption $A(x)$, used with $n = 2$, yields a rational q with $|x - q| \le 1/4$. It is easy to find (constructively!) a signed digit d such that $[q - 1/4, q + 1/4] \cap \mathbb{I} \subseteq \mathbb{I}_d$. For that d we have $x \in \mathbb{I}_d$.

The type corresponding to the predicate $\mathbb{Q}$ is $\mathrm{Nat} \times \mathrm{Nat} \times \mathrm{Nat}$, which we, however implement by Haskell's built-in rationals, since it is only the arithmetic operations on rational numbers that matter, whatever the representation. (It is possible - and instructive as an exercise - to extract implementations of the arithmetic operations on rational numbers w.r.t. the representation $\mathrm{Nat} \times \mathrm{Nat} \times \mathrm{Nat}$ from proofs that $\mathbb{Q}$ is closed under these operations.) The type of the predicate A is $\mathrm{Nat} \to \mathrm{Rational}$. The program extracted from the proof of Lemma 1 is

```
cauchy2sd :: (Nat -> Rational) -> SDS
cauchy2sd f = d : (cauchy2sd f') where (d,f') = step f
```

with the sub program

```
step :: (Nat -> Rational) -> (SD , Nat -> Rational)
step f = (d,f')  where
  q = f 2
  d = if q > 1/2 then 1 else if abs q <= 1/2 then 0 else (-1)
  f' n = 2 * f (n+1) - d
```

It should be clear how `step` is derived from the proof, but one might ask by which rules the recursive main program was extracted. We address this question next.

The program `cauchy2sd` is an instance of the scheme of *corecursion* or *guarded recursion*. We briefly discuss how this scheme, and more generally, any realisers of inductive and coinductive definitions, can be derived from general category-theoretic considerations. Let Φ be a set operator defined by

$$\Phi(X) = \{x \mid A(x, X)\}$$

such that the predicate variable occurs only strictly positively in the formula $A(x, X)$ and hence Φ is monotone (the condition of strict positivity could be weakened, but it is sufficiently general for our purpose and easy to work with). Therefore, we can form the inductive predicate $\mu X.\Phi(X)$. We assign to $\mu X.\Phi(X)$ a type $\mu\alpha.\varphi(\alpha)$, where α is a type variable assigned to the predicate variable X and $\varphi(\alpha)$ is the type (recursively) assigned to the predicate $\Phi(X)$. We write $\mu\Phi$ instead of $\mu X.\Phi(X)$ (as before), but also $\mu\varphi$ instead of $\mu\alpha.\varphi(\alpha)$. The fixed point principle, $\Phi(\mu\Phi) \subseteq \mu\Phi$ (closure) and $\mu\Phi \subseteq \Phi(\mu\Phi)$ (coclosure), as well as the induction principle, $\Phi(X) \subseteq X \Rightarrow \mu\Phi \subseteq X$, which we discussed in Section 2 are 'realised' by the constants

$$\begin{aligned} \mathbf{In}_\varphi &: \varphi(\mu\varphi) \to \mu\varphi \\ \mathbf{Out}_\varphi &: \mu\varphi \to \varphi(\mu\varphi) \\ \mathbf{It}_\varphi &: (\varphi(\alpha) \to \alpha) \to \mu\varphi \to \alpha \end{aligned}$$

The category-theoretic explanation is that $\varphi(\alpha)$ is functorial in α, because α occurs only strict positively, and hence, in a suitable category of types, has an *initial algebra* which is given by $\mathbf{In}_\varphi : \varphi(\mu\varphi) \to \mu\varphi$. The 'iterator' $\mathbf{It}_\varphi$ (sometimes also called "fold") witnesses the initiality of the algebra $\mathbf{In}_\varphi : \varphi(\mu\varphi) \to \mu\varphi$, i.e. for any other α-algebra $s : \varphi(\alpha) \to \alpha$, $\mathbf{It}_\varphi\, s : \mu\varphi \to \alpha$ is the unique mediating algebra morphism. That $\mathbf{It}_\varphi\, s$ is an algebra morphism means that the equation

$$(\mathbf{It}_\varphi\, s) \circ \mathbf{In}_\varphi = s \circ (\mathbf{map}_\varphi\, (\mathbf{It}_\varphi\, s))$$

holds for all φ-algebras $s : \varphi(\alpha) \to \alpha$. Here, $\mathbf{map}_\varphi : (\alpha \to \beta) \to \varphi(\alpha) \to \varphi(\beta)$ is the action of the functor φ on morphisms, which can be defined by structural

recursion on the built-up of $\varphi(\alpha)$. Since in the category of types composition of morphisms is defined pointwise we can rewrite this as

$$\mathbf{It}_\varphi\, s\, (\mathbf{In}_\varphi\, x) = s\, (\mathbf{map}_\varphi\, (\mathbf{It}_\varphi\, s)\, x) \tag{6}$$

Uniqueness means that this equation actually *defines* $\mathbf{It}_\varphi$. The inverse of $\mathbf{In}_\varphi$, $\mathbf{Out}_\varphi : \mu\varphi \to \varphi(\mu\varphi)$ could be defined from $\mathbf{In}_\varphi$ and $\mathbf{It}_\varphi$ (even extracted, since coclosure can be proven from closure and induction), but it is convenient to add it as a primitive. Modelling this in Haskell is straightforward. We define $\mu\varphi$ as a recursive type:

```
data Mu = In (Phi Mu)
```

Hence the constructor `In :: Phi Mu -> Mu` comes for free and the destructor `Out :: Mu -> Phi Mu` can be defined by pattern matching `Out (In x) = x`. Finally, we use equation (6) for a recursive definition of $\mathbf{It}_\varphi$:

```
it :: (Phi alpha -> alpha) -> Mu -> alpha
it s (In x) = s (mapPhi it s x)
```

For coinduction the story is similar. We assign to $\nu X.\Phi(X)$ the type $\nu\alpha.\varphi(\alpha)$ and realise fixed point principle and coinduction, $\Phi(\nu\Phi) \subseteq \nu\Phi$, $\nu\Phi \subseteq \Phi(\nu\Phi)$, and $X \subseteq \Phi(X) \Rightarrow X \subseteq \nu\Phi$, by

$$\begin{aligned}\mathbf{In}_\varphi &: \varphi(\nu\varphi) \to \nu\varphi\\ \mathbf{Out}_\varphi &: \nu\varphi \to \varphi(\nu\varphi)\\ \mathbf{Coit}_\varphi &: (\alpha \to \varphi(\alpha)) \to \alpha \to \nu\varphi\end{aligned}$$

Now $\mathbf{Out}_\varphi : \mu\varphi \to \varphi(\mu\varphi)$ is the *final coalgebra* of α, and its finality is witnessed by the 'coiterator' $\mathbf{Coit}_\varphi$ (also called "unfold"), i.e. for any other α-coalgebra $s : \alpha \to \varphi(\alpha)$, $\mathbf{Coit}_\varphi\, s : \alpha \to \nu\varphi$ is the unique mediating coalgebra morphism satisfying

$$\mathbf{Out}_\varphi\, (\mathbf{Coit}_\varphi\, s\, t) = \mathbf{map}_\varphi\, (\mathbf{Coit}_\varphi\, s)\, (s\, t) \tag{7}$$

Because Haskell does not distinguish between least and largest fixed points, the implementation of $\nu\Phi$ is the same as for $\mu\Phi$, namely `data Nu = In (Phi Nu)`, with constructor `In :: Phi Nu -> Nu` and destructor `Out :: Nu -> Phi Nu` defined by `Out (In x) = x`. In order to use Equation (7) as a recursive definition of $\mathbf{Coit}_\varphi$ we apply the constructor $\mathbf{In}_\varphi$ to both sides and obtain

```
coit :: (alpha -> Phi alpha) -> alpha -> Nu
coit s x = In (mapPhi (coit s) (s x))
```

Realisers of strong induction and strong coinductions can be defined by similar recursive procedures. The operational semantics of these and more general operators have been studied e.g. in [Geu92] and [AMU05].

In the case of our example about transforming fast Cauchy sequences into signed digit streams we have $\varphi(\alpha) = \mathrm{SD} \times \alpha$ and $\mathbf{map}_\varphi(f)(d, x) = (d, f(x))$. Hence $\mathbf{Coit}_\varphi(s)(x) = \mathbf{In}_\varphi(d, \mathbf{Coit}_\varphi(s)(y))$ where $(d, y) = s(x)$, and the extracted

program of the proof of (5) is $\mathbf{Coit}_\varphi$(step) where step is the previously defined sub program of `cauchy2sd`. If we, as done before, implement $\nu\varphi$ by `[SD]`, then $\mathbf{In}_\varphi$ is to be implemented by `inPhi (d,x) = d:x` and altogether we obtain

```
coit :: (alpha -> (SD , alpha)) -> alpha -> SDS
coit s x = d : coit s y  where (d,y) = s x

cauchy2sd :: (Nat -> Rational) -> SDS
cauchy2sd = coit step
```

which is clearly equivalent to what we had before.

In order to prove that the extraction method sketched above yields correct programs one defines a notion of realisability relating programs and formulas and shows a *Soundness Theorem* stating that every extracted program realises the formula it proves (see e.g. [Tro73], [Tat98]for detailed proofs of soundness for related notions of realisability).

4 Coinductive definition of uniform continuity

For every n we define a set $\mathrm{C}_n \subseteq \mathbb{R}^{\mathbb{I}^n}$ for which we will later show that it coincides with the set of uniformly continuous functions from $\mathbb{I}^n$ to $\mathbb{I}$.

In the following we let n, m, k, l, i range over $\mathbb{N}$, p, q over $\mathbb{Q}$, x, y, z over $\mathbb{R}$, and d, e over SD. Hence, for example, $\exists d\, A(d)$ is shorthand for $\exists d\,(\mathrm{SD}(d) \wedge A(d))$. We define average functions and their inverses

$$\mathrm{av}_d\colon \mathbb{R} \to \mathbb{R},\ \ \mathrm{av}_d(x) := \frac{x+d}{2}$$
$$\mathrm{va}_d\colon \mathbb{R} \to \mathbb{R},\ \ \mathrm{va}_d(x) := 2x - d$$

Note that $\mathrm{av}_d[\mathbb{I}] = \mathbb{I}_d := \{x \mid |x - d/2| \leq 1/2\} = \mathbb{I}_{d/2,1}$ and hence $f[\mathbb{I}] \subseteq \mathbb{I}_d$ iff $(\mathrm{va}_d \circ f)[\mathbb{I}] \subseteq \mathbb{I}$. We also need extensions of the average functions to n-tuples

$$\mathrm{av}_{i,d}(x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_n) := (x_1, \ldots, x_{i-1}, \mathrm{av}_d(x_i), x_{i+1}, \ldots, x_n)$$

We define an operator $\mathcal{K}_n\colon \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n})$ by

$$\mathcal{K}_n(X)(Y) := \{f \mid \exists d\,(f[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge X(\mathrm{va}_d \circ f)) \vee \exists i\, \forall d\, Y(f \circ \mathrm{av}_{i,d})\}$$

Since $\mathcal{K}_n$ is strictly positive in both arguments, we can define an operator $\mathcal{J}_n\colon \mathcal{P}(\mathbb{R}^{\mathbb{I}^n}) \to \mathcal{P}(\mathbb{R}^{\mathbb{I}^n})$ by

$$\mathcal{J}_n(X) := \mu(\mathcal{K}_n(X)) = \mu Y.\mathcal{K}_n(X)(Y)$$

Hence, $\mathcal{J}_n(X)$ is the set inductively defined by the following two rules:

$$\exists d\,(f[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge X(\mathrm{va}_d \circ f)) \Rightarrow \mathcal{J}_n(X)(f) \tag{8}$$

$$\exists i\, \forall d\, \mathcal{J}_n(X)(f \circ \mathrm{av}_{i,d}) \Rightarrow \mathcal{J}_n(X)(f) \tag{9}$$

Since, as mentioned in Section 2, the operation μ is monotone, $\mathcal{J}_n$ is monotone as well. Therefore, we can define C_n as the largest fixed point of $\mathcal{J}_n$,

$$\mathrm{C}_n = \nu\mathcal{J}_n = \nu X.\mu Y.\mathcal{K}_n(X)(Y) \tag{10}$$

Note that for $n = 0$ the second argument Y of $\mathcal{K}_n$ becomes a dummy variable, and therefore $\mathcal{J}_0$ and C_0 are the same as in the corresponding example in Section 2.

The type corresponding to the formula $\mathcal{K}_n(X)(Y)$ is $\mathrm{SD} \times \alpha + \mathbb{N}_n \times \mathrm{SD}^3 \times \beta$. where $\mathbb{N}_n := \{1, ..., n\}$. Therefore, the type of $\mathcal{J}_n(X)$ is $\mu\beta.\mathrm{SD} \times \alpha + \mathbb{N}_n \times \mathrm{SD}^3 \times \beta$ which is the type of finite ternary branching trees with indices $i \in \mathbb{N}_n$ attached to the inner nodes and pairs $(d, x) \in \mathrm{SD} \times \alpha$ attached to the leaves. Consequently, the type of C_n is $\nu\alpha.\mu\beta.\mathrm{SD}\times\alpha+\mathbb{N}_n\times\mathrm{SD}^3\times\beta$ which is the type of non-wellfounded trees obtained by infinitely often stacking the finite trees on top of each others, i.e. replacing in a finite tree each x in a leaf by another finite tree and repeating ad-infinitum the process in the substituted trees. Alternatively, the elements of this type can be described as non-wellfounded trees without leaves such that

1. each node is either a
   writing node labelled with a signed digit and with one subtree, or a
   reading node labelled with an index $i \in \mathbb{N}_n$ and with three subtrees;
2. each path has infinitely many writing nodes.

The interpretation of such a tree as a stream transformer is easy. Given n signed digit streams $a_1, \ldots, a_n$ as inputs, run through the tree and output a signed digit stream as follows:

1. At a writing node (d, t) output d and continue with the subtree t.
2. At a reading node $(i, (t_d)_{d\in\mathrm{SD}})$ continue with t_d, where d is the head of a_i, and replace a_i by its tail.

This interpretation corresponds to the extracted program of a special case of Proposition 1 below which shows that the predicates C_n are closed under composition.

Lemma 2. *If* $\mathrm{C}_n(f)$*, then* $\mathrm{C}_n(f \circ \mathrm{av}_{i,d})$*.*

Proof. We fix $i \in \{1, \ldots, n\}$ and $d \in \mathrm{SDS}$ and set

$$D := \{f \circ \mathrm{av}_{i,d} \mid \mathrm{C}_n(f)\}$$

We show $D \subseteq \mathrm{C}_n$ by strong coinduction, i.e. we show $D \subseteq \mathcal{J}_n(D \cup \mathrm{C}_n)$, i.e. $\mathrm{C}_n \subseteq E$ where

$$E := \{f \mid \mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})\}$$

Since $\mathrm{C}_n = \mathcal{J}_n(\mathrm{C}_n)$ it suffices to show $\mathcal{J}_n(\mathrm{C}_n) \subseteq E$. We prove this by strong induction on $\mathcal{J}_n(\mathrm{C}_n)$, i.e. we show $\mathcal{K}_n(\mathrm{C}_n)(E \cap \mathcal{J}_n(\mathrm{C}_n)) \subseteq E$. Induction base: Assume $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$ and $\mathrm{C}_n(\mathrm{va}_{d'} \circ f)$. We need to show $E(f)$, i.e. $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$. By (8) it suffices to show $(f \circ \mathrm{av}_{i,d})[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$ and $(D \cup \mathrm{C}_n)(\mathrm{va}_{d'} \circ f \circ \mathrm{av}_{i,d})$.

We have $(f\circ \mathrm{av}_{i,d})[\mathbb{I}^n] = f[\mathrm{av}_{i,d}[\mathbb{I}^n]] \subseteq f[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$. Furthermore, $D(\mathrm{va}_{d'}\circ f\circ \mathrm{av}_{i,d})$ holds by the assumption $\mathrm{C}_n(\mathrm{va}_d \circ f)$ and the definition of D. Induction step: Assume, as strong induction hypothesis, $\forall d'\,(E \cap \mathcal{J}_n(\mathrm{C}_n))(f \circ \mathrm{av}_{i',d'})$. We have to show $E(f)$, i.e. $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$. If $i' = i$, then the strong induction hypothesis implies $\mathcal{J}_n(\mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$ which, by the monotonicity of $\mathcal{J}_n$, in turn implies $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$, i.e. $E(f)$. If $i' \neq i$, then $\forall d'\,\mathrm{av}_{i',d'} \circ \mathrm{av}_{i,d} = \mathrm{av}_{i,d}\circ\mathrm{av}_{i',d'}$ and therefore, since the strong induction hypothesis implies $\forall d'\,E(f\circ \mathrm{av}_{i',d'})$, we have $\forall d'\,\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d} \circ \mathrm{av}_{i',d'})$. By (9) this implies $\mathcal{J}_n(D \cup \mathrm{C}_n)(f \circ \mathrm{av}_{i,d})$, i.e. $E(f)$.

Proposition 1. *Let* $f\colon \mathbb{I}^n \to \mathbb{R}$ *and* $g_i\colon \mathbb{I}^m \to \mathbb{R}$, $i = 1, \ldots, n$. *If* $\mathrm{C}_n(f)$ *and* $\mathrm{C}_m(g_1), \ldots, \mathrm{C}_m(g_n)$, *then* $\mathrm{C}_m(f \circ (g_1, \ldots, g_n))$.

Proof. We prove the proposition by coinduction, i.e. we set

$$D := \{f \circ (g_1, \ldots, g_n) \mid \mathrm{C}_n(f),\ \mathrm{C}_m(g_1),\ \ldots,\ \mathrm{C}_m(g_n)\}$$

and show that $D \subseteq \mathcal{J}_m(D)$, i.e. $\mathrm{C}_n \subseteq E$ where

$$E := \{f \in \mathbb{R}^{\mathbb{I}^n} \mid \forall \boldsymbol{g}\,(\mathrm{C}_m(\boldsymbol{g}) \Rightarrow \mathcal{J}_m(D)(f \circ \boldsymbol{g}))\}$$

and $\mathrm{C}_m(\boldsymbol{g}) := \mathrm{C}_m(g_1) \wedge \ldots \wedge \mathrm{C}_m(g_n)$. Since $\mathrm{C}_n = \mathcal{J}_n(\mathrm{C}_n)$ it suffices to show $\mathcal{J}_n(\mathrm{C}_n) \subseteq E$. We do an induction on $\mathcal{J}_n(\mathrm{C}_n)$, i.e. we show $\mathcal{K}_n(\mathrm{C}_n)(E) \subseteq E$. Induction base: Assume $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$, $\mathrm{C}_n(\mathrm{va}_d \circ f)$ and $\mathrm{C}_m(\boldsymbol{g})$. We have to show $(f \circ g)[\mathbb{I}^m] \subseteq \mathbb{I}_d$ and $\mathcal{J}_m(D)(f \circ \boldsymbol{g}))$. By (8) it suffices to show $D(\mathrm{va}_d \circ f \circ \boldsymbol{g})$. But this holds by the definition of D and the assumption. Induction step: Assume, as induction hypothesis, $\forall d\,E(f \circ \mathrm{av}_{i,d})$. We have to show $E(f)$, i.e. $\mathrm{C}_m \subseteq F$ where

$$F := \{g \in \mathbb{R}^{\mathbb{I}^m} \mid \forall \boldsymbol{g}\,(g = g_i \wedge \mathrm{C}_m(\boldsymbol{g}) \Rightarrow \mathcal{J}_m(D)(f \circ \boldsymbol{g}))\}$$

Since $\mathrm{C}_m \subseteq \mathcal{J}_m(\mathrm{C}_m)$ it suffices to show $\mathcal{J}_m(\mathrm{C}_m) \subseteq F$ which we do by a side induction on $\mathcal{J}_m$, i.e. we show $\mathcal{K}_m(\mathrm{C}_m)(F) \subseteq F$. Side induction base: Assume $g[\mathbb{I}^m] \subseteq \mathbb{I}_d$ and $\mathrm{C}_m(\mathrm{va}_d \circ g)$ and $\mathrm{C}_m(\boldsymbol{g})$ where $g = g_i$. We have to show $\mathcal{J}_m(D)(f \circ \boldsymbol{g})$. Let $\boldsymbol{g}'$ be obtained from $\boldsymbol{g}$ by replacing g_i with $\mathrm{va}_d \circ g$. By the main induction hypothesis we have $\mathcal{J}_m(D)(f \circ \mathrm{av}_{i,d} \circ \boldsymbol{g}')$. But $\mathrm{av}_{i,d} \circ \boldsymbol{g}' = \boldsymbol{g}$. Side induction step: Assume $\forall d\,F(g \circ \mathrm{av}_{j,d})$ (side induction hypothesis). We have to show $F(g)$. Assume $\mathrm{C}_m(\boldsymbol{g})$ where $g = g_i$. We have to show $\mathcal{J}_m(D)(f \circ \boldsymbol{g})$. By (9) it suffices to show $\mathcal{J}_m(D)(f \circ \boldsymbol{g} \circ \mathrm{av}_{j,d})$ for all d. Since the i-th element of $\boldsymbol{g} \circ \mathrm{av}_{j,d}$ is $g \circ \mathrm{av}_{j,d}$ and, by Lemma 2, $\mathrm{C}_m(\boldsymbol{g} \circ \mathrm{av}_{j,d})$, we can apply the side induction hypothesis.

The program extracted from Proposition 1 composes trees. The special case where $m = 0$ interprets a tree in C_n as an n-ary stream transformer. The special case $n = m = 1$ was treated in [GHP06], however, without applications to exact real number computation. In [GHP06], the program was 'guessed' and then verified, whereas we extracted the program from a proving making verification unnecessary.

Of course, one can reduce Proposition 1 to the case $m = n = 1$, by coding n streams of single digits into one stream of n-tuples of digits. But this would lead to less efficient programs, since it would mean that in each reading step *all* inputs are read, even those that might not be needed (for example, the function $f(x, y) = x/2 + y/100$ certainly should read x more often than y).

5 Digital systems

Now we introduce digital systems which are a convenient tool for obtaining implementations of certain families of u.c. functions.

Let $(A, <)$ be a wellfounded relation. A *digital system* is a family $\mathcal{F} = (f_\alpha : \mathbb{I}^n \to \mathbb{I})_{\alpha\in A}$ such that for all $\alpha \in A$

$$\exists d\,(f_\alpha[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge \exists \beta < \alpha\, f_\beta = \mathrm{va}_d \circ f_\alpha) \vee \exists i\, \forall d\, \exists \beta\, f_\beta = f_\alpha \circ \mathrm{av}_{i,d}$$

When convenient we identify the family $\mathcal{F}$ with the set $\{f_\alpha \mid \alpha \in A\}$.

Proposition 2. *If $\mathcal{F}$ is a digital system, then $\mathcal{F} \subseteq \mathrm{C}_n$.*

Proof. Let $\mathcal{F}$ be a digital system. We show $\mathcal{F} \subseteq \mathrm{C}_n$ by coinduction. Hence, we have to show $\mathcal{J}_n(\mathcal{F})(f_\alpha)$ for all $\alpha \in A$. But, looking at the definition of $\mathcal{J}_n(\mathcal{F})$ and the properties of a digital system, this follows immediately by wellfounded $<$-induction on α. (Wellfounded induction can be realised by a simple recursive procedure.)

Example (linear affine functions) For $\boldsymbol{u}, v \in \mathbb{Q}^{n+1}$ define $f_{\boldsymbol{u},v}\colon \mathbb{I}^n \to \mathbb{R}$ by

$$f_{\boldsymbol{u},v}(\boldsymbol{x}) := u_1x_1 + \ldots + u_nx_n + v$$

Clearly, $f_{\boldsymbol{u},v}[\mathbb{I}^n] = [v-|\boldsymbol{u}|, v+|\boldsymbol{u}|]$ where $|\boldsymbol{u}| := |u_1|+\ldots+|u_n|$. Hence $f_{\boldsymbol{u},v}[\mathbb{I}^n] \subseteq \mathbb{I}$ iff $|\boldsymbol{u}| + |v| \le 1$, and if $|\boldsymbol{u}| \le 1/4$, then $f_{\boldsymbol{u},v}[\mathbb{I}^n] \subseteq \mathbb{I}_d$ for some d. Furthermore, $f_{\boldsymbol{u},v} \circ \mathrm{av}_{i,d} = f_{\boldsymbol{u}',v'}$ where $\boldsymbol{u}'$ is like $\boldsymbol{u}$ except that the i-th component is halved, and $v' = v + u_i d/2$. Hence, if i was chosen such that $|u_i| \ge |\boldsymbol{u}|/n$, then $|u'| \le q|u|$ where $q := 1 - 1/(2n) < 1$. Therefore, we set $A := \{\boldsymbol{u}, v \in \mathbb{Q}^{n+1} \mid |\boldsymbol{u}| + |v| \le 1\}$ and define a wellfounded relation $<$ on A by

$$\boldsymbol{u}', v' < \boldsymbol{u}, v \quad :\Leftrightarrow \quad |\boldsymbol{u}| \ge 1/4 \wedge |\boldsymbol{u}'| \le q|\boldsymbol{u}|$$

From the above it follows that $\mathrm{Pol}_{1,n} := (f_{\boldsymbol{u},v})_{\boldsymbol{u},v\in A}$ is a digital system. Hence $\mathrm{Pol}_{1,n} \subseteq \mathrm{C}_n$, by Proposition 2. Program extraction gives us a program that assigns to each tuple of rationals $\boldsymbol{u}, w \in A$ a digit implementation of $f_{\boldsymbol{u},w}$.

Remark. In [Kon04] it has been shown that the linear affine transformations are exactly the functions that can be represented by a finite automaton. The trees computed by our program generate these automata, simply because for the computation of the tree for $f_{\boldsymbol{u},v}$ only finitely many other indices $\boldsymbol{u}', v'$ are used, and Haskell will construct the tree by connecting these indices by pointers.

Example (iterated logistic map) With a similar proof as for the linear affine maps one can show that all polynomials of degree 2 with rational coefficients mapping $\mathbb{I}$ to $\mathbb{I}$ are in C_1. In particular the function logistic map (transformed to $\mathbb{I}$), defined by $f_a(x) = a(1-x^2) - 1$ is in C_1 for each rational number $a \in [0, 2]$. Exact computation of iterations of the logistic map on $[0, 1]$ were studied in [Bla05] and [Plu98]. Our extracted programs are able to compute 100 binary digits of $f_a^{500}(q)$ for arbitrary choices of a, q in a few minutes. This compares

favourably with the experiments in [Plu98] which are based on the binary signed digit representation as well. In addition, when one carries out this computation for a sequence of values q that are close together, then the memoising effect of the tree representation kicks in and one observes a speed up of computation of a factor ≥ 2 compared to the non-memoised computation.

Proposition 3. $\mathrm{C}_n(f)$ *iff* f *is uniformly continuous and* $f[\mathbb{I}^n] \subseteq \mathbb{I}$.

Proof. We only show the "if" part. We use Proposition 2. For every u.c. function $f\colon \mathbb{I}^n \to \mathbb{I}$ there exists a monotone continuous function $m_f : \mathbb{Q} \to \mathbb{Q}$ such that $m_f(0) = 0$ and for every $\boldsymbol{p} \in \mathbb{I}^n$ and $\delta > 0$ there exists $q \in \mathbb{I}$ with $f[\mathrm{B}_\delta(\boldsymbol{p})] \subseteq \mathrm{B}_{m_f(\delta)}(q)$. Let A be the set of pairs (f, m_f) where f and m_f are as above. Let $|(f, m_f)|$ be the least k such that $m_f(2^{-k}) \leq 1/4$ and define a wellfounded relation $<$ on A by $(f', m'_f) < (f, m_f) :\Leftrightarrow |(f', m'_f)| < |(f, m_f)|$. It is easy to see that the family $(f)_{(f,m_f)\in A}$ is a digital system.

The proof of Proposition 3 provides a program that computes for any u.c. function from $\mathbb{I}^n$ to $\mathbb{I}$ implementing it. The efficiency of this program depends on the details of how uniform continuity is defined constructively.

6 Integration

For a continuous function $f : \mathbb{I} \to \mathbb{R}$ we set $\int f := \int_{-1}^{1} f = \int_{-1}^{1} f(x)\,\mathrm{d}x$. In order to compute the integral of a function $f \in \mathrm{C}_1$ we only need the following two basic facts (which have no computational content).

Lemma 3. *(a)* $\int f = \frac{1}{2}\int(\mathrm{va}_d \circ f) + d$. *(b)* $\int f = \frac{1}{2}(\int(f \circ \mathrm{va}_{-1}) + \int(f \circ \mathrm{va}_1))$.

Proposition 4. *If* $\mathrm{C}_1(f)$, *then* $\forall k\, \exists p\, |\int f - p| \leq 2^{1-k}$.

Proof. We show by induction on k

$$\forall k\, \forall f\, (\mathrm{C}_1(f) \Rightarrow \exists p\, |\int f - p| \leq 2^{1-k})$$

$k = 0$: Since $\mathrm{C}_1(f)$ implies $f[\mathbb{I}] \subseteq \mathbb{I}$, we have $\int f \leq 2$. Hence set $p := 0$.
$k + 1$: $\mathrm{C}_1(f)$ implies $f[\mathbb{I}] \subseteq \mathbb{I}$ and $\mathcal{J}_1(\mathrm{C}_1)(f)$. Hence it suffices to show

$$\forall f\, (\mathcal{J}_1(\mathrm{C}_1)(f) \Rightarrow \exists p\, |\int f - p| \leq 2^{-k})$$

by a side induction on $\mathcal{J}_1(\mathrm{C}_1)(f)$. If $\mathrm{C}_1(\mathrm{va}_d \circ f)$, then, by the main induction hypothesis, $|\int(\mathrm{va}_d \circ f) - p| \leq 2^{1-k}$ for some p. By Lemma 3 (a) it follows $|\int f - (\frac{p}{2} + d)| = \frac{1}{2}|\int(\mathrm{va}_d \circ f) - p| \leq 2^{-k}$. If $\forall d\, \mathcal{J}_1(\mathrm{C}_1)(f)$, then, by the side induction hypothesis, there are p and q such that $|\int(f \circ \mathrm{va}_{-1}) - p| \leq 2^{-k}$ and $|\int(f \circ \mathrm{va}_1) - q| \leq 2^{-k}$. By Lemma 3 (b) it follows $|\int f - \frac{1}{2}(p + q)| = \frac{1}{2}|\int(f \circ \mathrm{va}_{-1}) + \int(f \circ \mathrm{va}_1) - (p + q)| \leq \frac{1}{2}(|\int(f \circ \mathrm{va}_{-1}) - p| + |\int(f \circ \mathrm{va}_1) - q|) \leq 2^{-k}$.

The program extracted from this proof allows to compute arbitrary good approximations to the integral of a function in C_1, and, in combination with Lemma 1, to compute a signed digit representation of the integral.

Remark. The proof of Proposition 4 is based on the same idea as the programs for integration developed in [Sim98] and [Scr08]. However, we with a different data structure. While in [Sim98,Scr08] the input is given as a stream transforming *function* for which a modulus of uniform continuity has to be computed by an expensive bar-recursive process, our program gets a *tree* as input that has the necessary information on the modulus of u.c. built-in (we see when we can write).

7 Conclusion

We presented a method for extracting from coinductive proofs tree-like data structures that code exact lazy algorithms for real functions. The extraction method is based on a variant of modified realisability that strictly separates the (abstract) mathematical model the proof is about from the data types the extracted program is dealing with. The latter are determined solely by the propositional structure of formulas and proofs. This has the advantage that the abstract mathematical structures do not need to be 'constructivised'. In addition, formulas that do not contain disjunctions are computationally meaningless and can therefore be taken as axioms as long as they are true. This enormously reduces the burden of formalisation and turns - in our opinion - program extraction into a realistic method for the development of nontrivial certified algorithms.

Further work. Currently, we are adapting the existing implementation of program extraction in the Minlog proof system [BBS$^+$98] to our setting. We are also extending this work to more general situations where the interval $\mathbb{I}$ and the maps av_d are replaced by an arbitrary bounded metric space with a system of contractions (see [Scr08] for related work), or even to the non-metric case (for example higher types). These extensions will facilitate the extraction of efficient programs for e.g. analytic functions, parametrised integrals, and set-valued functions.

References

[AMU05] A. Abel, R. Matthes, and T. Uustalu. Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci*, 333:3–66, 2005.

[BBS$^+$98] H. Benl, U. Berger, H. Schwichtenberg, M. Seisenberger, and W. Zuber. Proof theory at work: Program development in the Minlog system. In W. Bibel and P.H. Schmitt, editors, *Automated Deduction – A Basis for Applications*, volume II of *Applied Logic Series*, pages 41–71. Kluwer, Dordrecht, 1998.

[Ber07] Y. Bertot. Affine functions and series with co-inductive real numbers. *Mathematical Structures in Computer Science*, 17:37–63, 2007.

[BH07] U. Berger and T. Hou. Coinduction for exact real number computation. *Theory of Computing Systems*, 00:00–00, 2007. To appear.

[Bla05] J. Blanck. Efficient exact computation of iterated maps. *Journal of Logic and Algebraic Programming*, 64:41–59, 2005.

[BS07] J. Bradfield and C. Stirling. Modal mu-calculi. In P. Blackburn, J. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 721–756. Elsevier, 2007.

[CDG06] A. Ciaffaglione and P. Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theor. Comput. Sci*, 351:39–51, 2006.

[EH02] A. Edalat and R. Heckmann. Computing with real numbers: I. The LFT Approach to Real Number Computation; II. A Domain Framework for Computational Geometry. In G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, editors, *Applied Semantics - Lecture Notes from the International Summer School, Caminha, Portugal*, pages 193–267. Springer, 2002.

[Geu92] H. Geuvers. Inductive and coinductive types with iteration and recursion. In B. Nordström, K. Pettersson, and G. Plotkin, editors, *Informal Proceedings 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.

[GHP06] N. Ghani, P. Hancock, and D. Pattinson. Continuous functions on final coalgebras. volume 164 of *Electron. Notes Theor. Comput. Sci.*, 2006.

[GNSW07] H. Geuvers, M. Niqui, B. Spitters, and F. Wiedijk. Constructive analysis, types and exact real numbers (overview article). *Mathematical Structures in Computer Science*, 17(1):3–36, 2007.

[Kon04] M. Konečný. Real functions incrementally computable by finite automata. *Theoretical Computer Science*, 315(1):109–133, 2004.

[Kre59] G. Kreisel. Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics*, pages 101–128, 1959.

[MRE07] J. R. Marcial-Romero and M. H. Escardo. Semantics of a sequential language for exact real-number computation. *Theor. Comput. Sci*, 379(1-2):120–141, 2007.

[Plu98] D. Plume. A calculator for exact real number computation, 1998.

[Sch08] H. Schwichtenberg. Realizability interpretation of proofs in constructive analysis. To appear in ToCS, 2008.

[Scr08] A. Scriven. A functional algorithm for exact real integration with invariant measures. *Electron. Notes Theor. Comput. Sci.*, 218:337–353, 2008.

[Sim98] A. Simpson. Lazy functional algorithms for exact real functionals. In *Mathematical Foundations of Computer Science*, volume 1450 of *LNCS*, pages 456–464, 1998.

[Tat98] M. Tatsuta. Realizability of monotone coinductive definitions and its application to program synthesis. In R. Parikh, editor, *Mathematics of Program Construction*, volume 1422 of *Lecture Notes in Mathematics*, pages 338–364. Springer, 1998.

[Tro73] A.S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, volume 344 of *LNM*. Springer, 1973.