

# From coinductive proofs to exact real arithmetic

Ulrich Berger

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

**Abstract.** We give a coinductive characterization 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 certified algorithms in exact real arithmetic.

*Key words:* Proof theory, program extraction, exact real number computation, coinduction.

## 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 [1, 2] or linear fractional transformations [3]) and proves their correctness using a certain proof method (for example coinduction [4–6]). 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 [7] 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 memoize computations which may improve performance in certain situations.

A crucial ingredient in the proofs (that we use for program extraction) is a coinductive definition of the notion of uniform continuity (u.c.). Although, classically, continuity and uniform continuity are equivalent for functions defined on a compact interval (we only consider such functions), 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 interval  $\mathbb{I} := [-1, 1] = \{x \in \mathbb{R} \mid |x| \leq 1\}$  and, for the purpose of this introduction, only

unary functions. However, later we will also look at functions of several variables where one has to deal with the non-trivial problem of deciding which of the input stream the next digit is to be consumed of. This choice can have a big influence on the performance of the program.

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

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

A function  $f : \mathbb{I} \rightarrow \mathbb{I}$  is *represented* by a stream transformer  $\hat{f} : \text{SDS} \rightarrow \text{SDS}$  if  $f \circ \sigma = \sigma \circ \hat{f}$ . The coinductive definition of uniform continuity allows us to extract from a constructive proof of the u.c. of a function  $f : \mathbb{I} \rightarrow \mathbb{I}$  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 memoized algorithms. These trees turn out to be a generalization of the data structure studied in [8], and the extracted program from the proof of closure under composition is a generalization of the tree composing program defined there.

In Section 2 we briefly review inductive and coinductive sets defined by monotone set operators. We give some simple examples, among them a coinductive characterization 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 characterization of real numbers is generalized to real functions, and closure under composition is proven. In Section 5 the positive effect of memoization is demonstrated by a case study on iterated logistic maps.

## 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 logic and computer science texts. For example in [9] inductive definitions are proof-theoretically analysed, and in [10] least and greatest fixed points are studied in the framework of the modal mu-calculus.

An operator  $\Phi: \mathcal{P}(U) \rightarrow \mathcal{P}(U)$  (where  $U$  is an arbitrary set and  $\mathcal{P}(U)$  is the powerset of  $U$ ) 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  $\Phi$ -*closed* (or a pre-fixed point of  $\Phi$ ) if  $\Phi(X) \subseteq X$ . Since  $\mathcal{P}(U)$  is a complete lattice,  $\Phi$  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  $\Phi$ -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  $\Phi$ , i.e.  $\Phi(\mu\Phi) = \mu\Phi$ . For monotone operators  $\Phi, \Psi: \mathcal{P}(U) \rightarrow \mathcal{P}(U)$  we define

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

It is easy to see that the operation  $\mu$  is *monotone*, i.e. if  $\Phi \subseteq \Psi$ , then  $\mu\Phi \subseteq \mu\Psi$ . Using monotonicity of  $\mu$  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  $\Phi$ -*coclosed* (or a post-fixed point of  $\Phi$ ) if  $X \subseteq \Phi(X)$ . By duality,  $\Phi$  has a largest fixed point  $\nu\Phi$  which can be defined as the largest  $\Phi$ -coclosed subset of  $U$ . Similarly, all other principles for induction have their coinductive counterparts. To summarise, we have the following principles:

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

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

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

Then  $\mu\Phi = \mathbb{N} = \{0, 1, 2, \dots\}$ . 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  $\forall x (X(x) \rightarrow X(x+1))$  (step), then  $\forall x \in \mathbb{N} X(x)$ . Strong induction weakens the step by restricting  $x$  to the natural numbers:  $\forall x \in \mathbb{N} (X(x) \rightarrow X(x+1))$ .

**Example (signed digits and the interval  $[-1, 1]$ )** For every signed digit  $d \in \text{SD}$  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  $\mathcal{J}_0: \mathcal{P}(\mathbb{R}) \rightarrow \mathcal{P}(\mathbb{R})$  by

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

and set  $C_0 := \nu\mathcal{J}_0$ . Since clearly  $\mathbb{I} \subseteq \mathcal{J}_0(\mathbb{I})$ , it follows, by coinduction, that  $\mathbb{I} \subseteq C_0$ . On the other hand  $C_0 \subseteq \mathbb{I}$ , by the fixed point property. Hence  $C_0 = \mathbb{I}$ . 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  $d \in \mathbb{SD}$  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 “ $\mathbb{I} \subseteq C_0$ ”, 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  $\sigma(a) = x$ .

**Example (lists, streams and trees)** Let the Scott-domain  $D$  be defined by the recursive domain equation  $D = \{*\} + D \times D$  where “+” denotes the separated sum of domains (see [11] for information on domains). The elements of  $D$  are  $\perp$  (the obligatory least element),  $*$ , and  $(x, y)$  where  $x, y \in D$ . Define  $\text{Times}_* : \mathcal{P}(D) \rightarrow \mathcal{P}(D) \rightarrow \mathcal{P}(D)$  by

$$\text{Times}_*(X)(Y) := \{*\} \cup \{(x, y) \mid x \in X, y \in Y\}$$

Clearly,  $\text{Times}_*$  is monotone in both arguments. For a fixed set  $X \subseteq D$ ,  $\text{List}(X) := \mu(\text{Times}_*(X))$  ( $= \mu Y. \text{Times}_*(X)(Y)$ ) can be viewed as the set of *finite* lists of elements in  $X$  (viewing  $(\cdot, \cdot)$  as the “cons” operation), and  $\text{Stream}(X) := \nu(\text{Times}_*(X))$  ( $= \nu Y. \text{Times}_*(X)(Y)$ ) as the set of *finite or infinite* lists or *streams* of elements in  $X$ . Since  $\mu$  is monotone the operator  $\text{List} : \mathcal{P}(D) \rightarrow \mathcal{P}(D)$  is again monotone. Hence we can define  $\text{Tree} := \nu \text{List} \subseteq D$  which is the set of finitely branching wellfounded or non-wellfounded trees. On the other hand,  $\text{Tree}' := \mu \text{Stream}$  consist of all finitely or infinitely branching wellfounded trees. The point of this example is that the definition of  $\text{Tree}$  is similar to the characterization of uniformly continuous functions from  $\mathbb{I}^n$  to  $\mathbb{I}$  in Section 4, the similarity being the fact that it is a coinductive definition with an inductive definition in its body. The set  $C_0$  of the previous example corresponds to the case  $n = 0$  where the inner inductive definition is trivial.

### 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 realizability* [12]. The *extension* concerns the addition of inductive and coinductive predicates. Realizability for such predicates has been studied previously, in the slightly different context of **q**-realizability by Tatsuta [13]. 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, realizers do not depend on the individuals quantified over.

This is similar to the common uniform treatment of second-order variables [14]. 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 ‘canonical’ proofs. For example, the propositional skeleton of the canonical proof that 3 is a natural number records the three-fold application of the successor clause  $\forall x (\mathbb{N}(x) \rightarrow \mathbb{N}(x + 1))$  to the base clause  $\mathbb{N}(0)$ , hence it can be viewed as the unary representation of the number 3.

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

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

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

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

The data type corresponding to (2) is obtained by the following *type extraction* process:

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

Hence `Nat` is the least solution of the equation

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

In Haskell we can define this as

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

The comment “`-- data`” indicates that we intend to use the recursive data type `Nat` as an inductive data type (or initial algebra). This means that the “total”, or “legal” elements are inductively generated from `Zero` and `Succ`. The natural (domain-theoretic) semantics of `Nat` also contains, for example, an “infinite” element defined recursively by `infty = Succ infty` which is not total in the inductive interpretation of `Nat`. In a coinductive interpretation (usually indicated by the comment `-- codata`) `infty` would count as total. The fact that Haskell does not distinguish between the inductive and the coinductive interpretation is semantically justified by the so-called limit-colimit-coincidence of the domain-theoretical semantics [15].

Applying type extraction to (1) we see that a program extracted from a proof of this formula will have type `Nat`  $\rightarrow$  `1 + 1`. Identifying the two-element type `1 + 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 Zero      = True
parity (Succ x) = case parity x of {True -> False ; False -> True}
```

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 the quantifier  $\exists y$  in (1) 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 the fact that  $\mathbb{N}$  is closed under the successor operation in the proof. The extracted program is then

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

This example shows that we can get meaningful computational content despite ignoring the first-order part of a proof. Moreover, 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.

The reader might be puzzled by the fact that quantifiers are ignored in the program extraction process. Quantifiers are, of course, *not* ignored in the *specification* of the extracted program, i.e. in the definition of realizability. For example, the statement that the program  $p := \text{parity}$  realizes (1) is expressed by the formula

$$\forall n, x (n \mathbf{r} \mathbb{N}(x) \Rightarrow \exists y (p(n) = \text{True} \wedge x = 2y \vee p(n) = \text{False} \wedge x = 2y + 1))$$

where  $n$  ranges over  $\text{Nat}$ , i.e. the terms `Zero`, `Succ Zero`, `Succ(Succ Zero)`,  $\dots$ , and  $n \mathbf{r} \mathbb{N}(x)$  means that  $n$  realizes  $\mathbb{N}(x)$  which in this case amounts to  $x$  being the value of  $n$  in  $\mathbb{R}$ . The *Soundness Theorem* for realizability states that the program extracted from a proof realizes the proven formula (see e.g. [14], [13] for detailed proofs of soundness for related notions of realizability).

**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

$$C_0(x) \Leftrightarrow \exists d (\text{SD}(d) \wedge \mathbb{I}_d(x) \wedge C_0(2x - d)) \quad (3)$$

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

$$\text{SDS} = (\mathbf{1} + \mathbf{1} + \mathbf{1}) \times \text{SDS} \quad (4)$$

Identifying the type  $\mathbf{1} + \mathbf{1} + \mathbf{1}$  with  $\text{SD}$

```
data SD = N | Z | P      -- N = -1 , Z = 0, P = 1
```

we obtain that  $\text{SDS}$  is the type of infinite streams of signed digits, i.e. the largest fixed point of the type operator

```
type J0 alpha = (SD, alpha)
```

which corresponds to the set operator  $\mathcal{J}_0$  which  $C_0$  is the largest fixed point of. Therefore (choosing  $\text{Cons}$  as constructor name)

```
data SDS = Cons (J0 SDS)  -- codata
```

i.e.  $\text{SDS} = \text{Cons} (\text{SD}, \text{SDS})$ .

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

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

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

**Lemma 1.**

$$\forall x (\mathbb{I}(x) \wedge A(x) \Rightarrow C_0(x)) \quad (5)$$

*Proof.* We show  $\mathbb{I} \cap A \subseteq C_0$  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  $d \in \text{SD}$  such that  $x \in \mathbb{I}_d$  and  $2x - d \in \mathbb{I} \cap A$ . Since, clearly  $A(2x - d)$  holds for any  $d \in \text{SD}$ , 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  $|x - q| \leq 1/4$ . 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  $\text{Nat} \times \text{Nat} \times \text{Nat}$ , 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  $\text{Nat} \times \text{Nat} \times \text{Nat}$  from proofs that  $\mathbb{Q}$  is closed under these operations.) The type of the predicate  $A$  is  $\text{Nat} \rightarrow \text{Rational}$ . The program extracted from the proof of Lemma 1 is

```
cauchy2sd :: (Nat -> Rational) -> SDS
cauchy2sd = coitJ0 step
```

where `step` is the program extracted from the proof of  $\mathbb{I} \cap A \subseteq \mathcal{J}_0(\mathbb{I} \cap A)$ :

```
step :: (Nat -> Rational) -> J0(Nat -> Rational)
step f = (d,f') where
  q = f (Succ (Succ Zero))
  d = if q > 1/4 then P else if abs q <= 1/4 then Z else N
  f' n = 2 * f (Succ n) - sd2Rational d

sd2Rational :: SD -> Rational
sd2Rational d = case d of {N -> -1 ; Z -> 0 ; P -> 1}
```

The program `coitJ0` is a polymorphic “coiterator” realizing the coinduction scheme  $X \subseteq \mathcal{J}_0(X) \Rightarrow X \subseteq \nu \mathcal{J}_0$ :

```
coitJ0 :: (alpha -> J0 alpha) -> alpha -> SDS
coitJ0 s x = Cons (mapJ0 (coitJ0 s) (s x))

mapJ0 :: (alpha -> beta) -> J0 alpha -> J0 beta
mapJ0 f (d,x) = (d,f x)
```

An equivalent definition of `coitJ0` would be

```
coitJ0 s x = Cons (d,coitJ0 s y) where (d,y) = s x
```

We hope that this example and the way it was presented gives enough hints to understand how program extraction from coinductive proofs works in general. In the general case one has a coinductive predicate  $\nu\Phi$  defined from a positive (and therefore monotone) set operator  $\Phi$  ( $\mathcal{J}_0$  in our example), i.e.  $\Phi(X) = \{\mathbf{x} \mid A(X, \mathbf{x})\}$  where  $X$  occurs only positively (in the usual sense) in  $A$ .  $\Phi$  corresponds to a positive type operator  $\text{Phi}$  ( $\text{J0}$  in our example). Due to the positivity of  $\text{Phi}$  one can define `mapPhi` :: (alpha -> beta) -> Phi alpha -> Phi beta (by structural recursion on `Phi alpha`), and from that, recursively, the coiterator

```
coitPhi :: (alpha -> Phi alpha) -> alpha -> Fix
coitPhi s x = Cons (mapPhi (coitPhi s) (s x))
```

where `Fix` is the largest fixed point of `Phi`:

```
data Fix = Cons (Phi Fix) -- codata
```

The program extracted from a coinductive proof of  $X \subseteq \nu\Phi$  is then `coitPhi step` where `step` :: alpha -> Phi alpha is the program extracted from the proof of  $X \subseteq \Phi(X)$  (alpha is the type corresponding to the predicate  $X$ ). For inductive proofs the construction is similar: One defines recursively an “iterator”

```
itPhi :: (Phi alpha -> alpha) -> Fix -> alpha
itPhi s (Cons z) = s (mapPhi (itPhi s) z)
```

(where the type `Fix` is now viewed as the *least* fixed point of `Phi`). The program extracted from an inductive proof of  $\mu\Phi \subseteq X$  is now `itPhi step` where `step :: Phi alpha -> alpha` is extracted from the proof of  $\Phi(X) \subseteq X$ .

The above sketched computational interpretations of induction and coinduction and more general recursive schemes can be derived from category-theoretic considerations using the initial algebra/final coalgebra interpretation of least and greatest fixed points (see for example [16–18]).

## 4 Coinductive definition of uniform continuity

For every  $n$  we define a set  $C_n \subseteq \mathbb{R}^{\mathbb{I}^n}$  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  $\text{SD}$ . Hence, for example,  $\exists d A(d)$  is shorthand for  $\exists d (\text{SD}(d) \wedge A(d))$ . We define average functions and their inverses

$$\begin{aligned} \text{av}_d: \mathbb{R} &\rightarrow \mathbb{R}, \quad \text{av}_d(x) := \frac{x+d}{2} \\ \text{va}_d: \mathbb{R} &\rightarrow \mathbb{R}, \quad \text{va}_d(x) := 2x-d \end{aligned}$$

Note that  $\text{av}_d[\mathbb{I}] = \mathbb{I}_d$  and hence  $f[\mathbb{I}] \subseteq \mathbb{I}_d$  iff  $(\text{va}_d \circ f)[\mathbb{I}] \subseteq \mathbb{I}$ . We also need extensions of the average functions to  $n$ -tuples

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

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

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

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

$$\exists d (f[\mathbb{I}^n] \subseteq \mathbb{I}_d \wedge X(\text{va}_d \circ f)) \Rightarrow \mathcal{J}_n(X)(f) \tag{6}$$

$$\exists i \forall d \mathcal{J}_n(X)(f \circ \text{av}_{i,d}) \Rightarrow \mathcal{J}_n(X)(f) \tag{7}$$

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

$$C_n = \nu \mathcal{J}_n = \nu X. \mu Y. \mathcal{K}_n(X)(Y) \tag{8}$$

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  $\text{SD} \times \alpha + \mathbb{N}_n \times \beta^3$ , where  $\mathbb{N}_n := \{1, \dots, n\}$ . Therefore, the type of  $\mathcal{J}_n(X)$  is  $\mu\beta.\text{SD} \times \alpha + \mathbb{N}_n \times \beta^3$  which is the type of finite ternary branching trees with indices  $i \in \mathbb{N}_n$  attached to the inner nodes and pairs  $(d, x) \in \text{SD} \times \alpha$  attached to the leaves. Consequently, the type of  $C_n$  is  $\nu\alpha.\mu\beta.\text{SD} \times \alpha + \mathbb{N}_n \times \beta^3$  which is the type of non-wellfounded trees obtained by infinitely often stacking the finite trees on top of each other, 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  $a_1, \dots, a_n$  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  $(i, (t_d)_{d \in \text{SD}})$  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  $C_n(f)$ , then  $C_n(f \circ \text{av}_{i,d})$ .*

*Proof.* We fix  $i \in \{1, \dots, n\}$  and  $d \in \text{SD}$  and set

$$D := \{f \circ \text{av}_{i,d} \mid C_n(f)\}$$

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

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

Since  $C_n = \mathcal{J}_n(C_n)$  it suffices to show  $\mathcal{J}_n(C_n) \subseteq E$ . We prove this by strong induction on  $\mathcal{J}_n(C_n)$ , i.e. we show  $\mathcal{K}_n(C_n)(E \cap \mathcal{J}_n(C_n)) \subseteq E$ . Induction base: Assume  $f[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$  and  $C_n(\text{va}_{d'} \circ f)$ . We need to show  $E(f)$ , i.e.  $\mathcal{J}_n(D \cup C_n)(f \circ \text{av}_{i,d})$ . By (6) it suffices to show  $(f \circ \text{av}_{i,d})[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$  and  $(D \cup C_n)(\text{va}_{d'} \circ f \circ \text{av}_{i,d})$ . We have  $(f \circ \text{av}_{i,d})[\mathbb{I}^n] = f[\text{av}_{i,d}[\mathbb{I}^n]] \subseteq f[\mathbb{I}^n] \subseteq \mathbb{I}_{d'}$ . Furthermore,  $D(\text{va}_{d'} \circ f \circ \text{av}_{i,d})$  holds by the assumption  $C_n(\text{va}_{d'} \circ f)$  and the definition of  $D$ . Induction step: Assume, as strong induction hypothesis,  $\forall d' (E \cap \mathcal{J}_n(C_n))(f \circ \text{av}_{i',d'})$ . We have to show  $E(f)$ , i.e.  $\mathcal{J}_n(D \cup C_n)(f \circ \text{av}_{i,d})$ . If  $i' = i$ , then the strong induction hypothesis implies  $\mathcal{J}_n(C_n)(f \circ \text{av}_{i,d})$  which, by the monotonicity of  $\mathcal{J}_n$ , in turn implies  $\mathcal{J}_n(D \cup C_n)(f \circ \text{av}_{i,d})$ , i.e.  $E(f)$ . If  $i' \neq i$ , then  $\forall d' \text{av}_{i',d'} \circ \text{av}_{i,d} = \text{av}_{i,d} \circ \text{av}_{i',d'}$  and therefore, since the strong induction hypothesis implies  $\forall d' E(f \circ \text{av}_{i',d'})$ , we have  $\forall d' \mathcal{J}_n(D \cup C_n)(f \circ \text{av}_{i,d} \circ \text{av}_{i',d'})$ . By (7) this implies  $\mathcal{J}_n(D \cup C_n)(f \circ \text{av}_{i,d})$ , i.e.  $E(f)$ .

**Proposition 1.** *Let  $f: \mathbb{I}^n \rightarrow \mathbb{R}$  and  $g_i: \mathbb{I}^m \rightarrow \mathbb{R}$ ,  $i = 1, \dots, n$ . If  $C_n(f)$  and  $C_m(g_1), \dots, C_m(g_n)$ , then  $C_m(f \circ (g_1, \dots, g_n))$ .*

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

$$D := \{f \circ (g_1, \dots, g_n) \mid C_n(f), C_m(g_1), \dots, C_m(g_n)\}$$

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

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

and  $C_m(\mathbf{g}) := C_m(g_1) \wedge \dots \wedge C_m(g_n)$ . Since  $C_n = \mathcal{J}_n(C_n)$  it suffices to show  $\mathcal{J}_n(C_n) \subseteq E$ . We do an induction on  $\mathcal{J}_n(C_n)$ , i.e. we show  $\mathcal{K}_n(C_n)(E) \subseteq E$ . Induction base: Assume  $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$ ,  $C_n(\text{va}_d \circ f)$  and  $C_m(\mathbf{g})$ . We have to show  $(f \circ \mathbf{g})[\mathbb{I}^m] \subseteq \mathbb{I}_d$  and  $\mathcal{J}_m(D)(f \circ \mathbf{g})$ . By (6) it suffices to show  $D(\text{va}_d \circ f \circ \mathbf{g})$ . But this holds by the definition of  $D$  and the assumption. Induction step: Assume, as induction hypothesis,  $\forall d E(f \circ \text{av}_{i,d})$ . We have to show  $E(f)$ , i.e.  $C_m \subseteq F$  where

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

Since  $C_m \subseteq \mathcal{J}_m(C_m)$  it suffices to show  $\mathcal{J}_m(C_m) \subseteq F$  which we do by a side induction on  $\mathcal{J}_m$ , i.e. we show  $\mathcal{K}_m(C_m)(F) \subseteq F$ . Side induction base: Assume  $g[\mathbb{I}^m] \subseteq \mathbb{I}_d$  and  $C_m(\text{va}_d \circ g)$  and  $C_m(\mathbf{g})$  where  $g = g_i$ . We have to show  $\mathcal{J}_m(D)(f \circ \mathbf{g})$ . Let  $\mathbf{g}'$  be obtained from  $\mathbf{g}$  by replacing  $g_i$  with  $\text{va}_d \circ g$ . By the main induction hypothesis we have  $\mathcal{J}_m(D)(f \circ \text{av}_{i,d} \circ \mathbf{g}')$ . But  $\text{av}_{i,d} \circ \mathbf{g}' = \mathbf{g}$ . Side induction step: Assume  $\forall d F(g \circ \text{av}_{j,d})$  (side induction hypothesis). We have to show  $F(g)$ . Assume  $C_m(\mathbf{g})$  where  $g = g_i$ . We have to show  $\mathcal{J}_m(D)(f \circ \mathbf{g})$ . By (7) it suffices to show  $\mathcal{J}_m(D)(f \circ \mathbf{g} \circ \text{av}_{j,d})$  for all  $d$ . Since the  $i$ -th element of  $\mathbf{g} \circ \text{av}_{j,d}$  is  $g \circ \text{av}_{j,d}$  and, by Lemma 2,  $C_m(\mathbf{g} \circ \text{av}_{j,d})$ , we can apply the side induction hypothesis.

The program extracted from Prop. 1 composes trees (unfortunately there is not enough space to show the extracted Haskell code of this and later examples). 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 [8], however, without applications to exact real number computation. The program was ‘guessed’ and then verified, whereas we are able to extract the program from a proof making verification unnecessary. Of course, one could 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  $\mathcal{F} = (f_\alpha : \mathbb{I}^n \rightarrow \mathbb{I})_{\alpha \in A}$  such that for all  $\alpha \in A$

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

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  $\mathcal{F} \subseteq C_n$ .*

*Proof.* Let  $\mathcal{F}$  be a digital system. We show  $\mathcal{F} \subseteq C_n$  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  $\alpha$ .

Since wellfounded induction can be realized by a simple recursive procedure we can extract from the proof of Prop. 2 a program that transforms a (realization of) a digital system into a family of trees realizing its members.

**Example (linear affine functions)** For  $\mathbf{u}, v \in \mathbb{Q}^{n+1}$  define  $f_{\mathbf{u},v} : \mathbb{I}^n \rightarrow \mathbb{R}$  by

$$f_{\mathbf{u},v}(\mathbf{x}) := u_1 x_1 + \dots + u_n x_n + v$$

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

$$\mathbf{u}', v' < \mathbf{u}, v \quad :\Leftrightarrow \quad |\mathbf{u}| \geq 1/4 \wedge |\mathbf{u}'| \leq q|\mathbf{u}|$$

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

*Remark.* In [19] 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  $f_{\mathbf{u},v}$  only finitely many other indices  $\mathbf{u}', v'$  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 [20] and [21]. 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 [21] 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 memoizing effect of the tree representation kicks in and one observes a speed up of computation of a factor  $\geq 2$  compared to the non-memoized computation.

An important application of digital systems is the following proof that the predicate  $C_n$  precisely captures uniform continuity. We work with the maximum norm on  $\mathbb{I}^n$  and set  $B_\delta(\mathbf{p}) := \{\mathbf{x} \in \mathbb{I}^n \mid |\mathbf{x} - \mathbf{p}| \leq \delta\}$  for  $\mathbf{p} \in \mathbb{I}^n$ . We also set  $Q := \mathbb{I} \cap \mathbb{Q}$  and let  $\delta, \epsilon$  range over positive rational numbers. Furthermore, we set

$$\text{Box}(\delta, \epsilon, f) := \forall \mathbf{p} \in Q^n \exists q \in Q (f[B_\delta(\mathbf{p})] \subseteq B_\epsilon(q))$$

It is easy to see that  $f: \mathbb{I}^n \rightarrow \mathbb{R}$  is uniformly continuous with  $f[\mathbb{I}^n] \subseteq \mathbb{I}$  iff

$$\forall \delta \exists \epsilon \text{Box}(\delta, \epsilon, f) \tag{9}$$

**Proposition 3.** *For any function  $f: \mathbb{I}^n \rightarrow \mathbb{R}$ ,  $C_n(f)$  iff  $f$  is uniformly continuous and  $f[\mathbb{I}^n] \subseteq \mathbb{I}$ .*

*Proof.* We have to show that  $C_n(f)$  holds iff (9) holds.

For the “if” part we use Prop. 2. Let  $A$  be the set of triples  $(f, m, [d_1, \dots, d_k])$  such that  $f$  satisfies (9),  $\text{Box}(2^{-m}, 1/4, f)$  holds, and  $d_1, \dots, d_k \in \text{SD}$  with  $k < n$  (hence in the case  $n = 1$  the list  $[d_1, \dots, d_k]$  is always empty). Define a well-founded relation  $<$  on  $A$  by

$$(f', m', [d'_1, \dots, d'_{k'}]) < (f, m, [d_1, \dots, d_k]) :\Leftrightarrow m' < m \vee (m' = m \wedge k' > k)$$

For  $\mathbf{d} = [d_1, \dots, d_k]$ , where  $k < n$ , set  $\text{av}_{\mathbf{d}} := \text{av}_{1,d_1} \circ \dots \circ \text{av}_{k,d_k}$ , i.p.  $\text{av}_{[]} is the identity function. We show that  $\mathcal{F} := (f \circ \text{av}_{\mathbf{d}})_{(f,m,\mathbf{d}) \in A}$  is a digital system (this is sufficient, because  $f \circ \text{av}_{[]} = f$ ).$

Let  $\alpha := (f, m, [d_1, \dots, d_k]) \in A$ .

*Case  $m = 0$ , i.e.  $\text{Box}(1, 1/4, f)$ .* We show that the left disjunct in the definition of a digital system holds. We have  $f[\mathbb{I}^n] = f[B_1(\mathbf{0})] \subseteq B_{1/4}(q)$  for some  $q \in Q$ . If  $|q| \leq 1/4$ , choose  $d := 0$ , if  $q > 1/4$ , choose  $d := 1$ , if  $q < -1/4$  choose  $d := -1$ . Then clearly  $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$ , and  $g := \text{va}_d \circ f$  is uniformly continuous and maps  $\mathbb{I}^n$  into  $\mathbb{I}$ . Hence  $(g, m', []) \in A$  for some  $m'$ .

*Case  $m > 0$ .* We show that the right disjunct in the definition of a digital system holds. Choose  $i := k + 1$ . Let  $d \in \text{SD}$ . If  $k + 1 < n$ , then  $\beta := (f, m, [d_1, \dots, d_k, d]) < \alpha$  and  $f \circ \text{av}_{[d_1, \dots, d_k, d]} = (f \circ \text{av}_{[d_1, \dots, d_k]}) \circ \text{av}_{i,d}$ . If  $k + 1 = n$ , then for  $g := f \circ \text{av}_{[d_1, \dots, d_k, d]}$  we have  $\beta := (g, m - 1, []) \in A$  because  $\text{av}_{[d_1, \dots, d_k, d]}$  is a contraction with contraction factor  $1/2$ . Clearly,  $\beta < \alpha$ . Furthermore,  $g \circ \text{av}_{[]} = g = (f \circ \text{av}_{[d_1, \dots, d_k]}) \circ \text{av}_{i,d}$ .

For the “only if” part we assume  $C_n(f)$ . Set

$$E_k := \{f : \mathbb{I}^n \rightarrow \mathbb{R} \mid \exists \delta \text{Box}(\delta, 2^{-k}, f)\}$$

For proving (9) it obviously suffices to show  $\forall k (f \in E_k)$ . Hence, it suffices to show  $C_n \subseteq E_k$  for all  $k$ . We proceed by induction on  $k$ .

*Base,  $k = 0$ :* Since  $B_1(0) = \mathbb{I}$ , we clearly have  $\text{Box}(1, 2^0, f)$  for all  $f \in C_n$ .

*Step,  $k \rightarrow k + 1$ :* Since  $C_n = \mathcal{J}_n(C_n)$  it suffices to show  $\mathcal{J}_n(C_n) \subseteq E_{k+1}$ . We prove this by side induction on  $\mathcal{J}_n(C_n)$ , i.e. we show  $\mathcal{K}_n(C_n)(E_{k+1}) \subseteq E_{k+1}$ .

*Side induction base:* Assume  $f[\mathbb{I}^n] \subseteq \mathbb{I}_d$  and  $C_n(\text{va}_d \circ f)$ . By the main induction hypothesis,  $\text{Box}(\delta, 2^{-k}, \text{va}_d \circ f)$  for some  $\delta$ . Hence, clearly,  $\text{Box}(\delta, 2^{-(k+1)}, f)$ . *Side induction step:* Assume, as side induction hypothesis,  $\text{Box}(\delta_d, 2^{-(k+1)}, f \circ \text{av}_{i,d})$  for all  $d \in \text{SD}$ . Setting  $\delta = \min\{\delta_d \mid d \in \text{SD}\}$ , we clearly have  $\text{Box}(\delta, 2^{-(k+1)}, f)$ .

*Remark.* The proof of the “if” direction computes a tree for every u.c. function, however, usually not a very good one, since if some input needs to be read, then *all* inputs are read. Hence, for particular families of u.c. functions one should *not* use this proof, but rather design a special digital system that reads inputs only when necessary (as done in the case of the linear affine functions).

## 6 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 realizability 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 formalization 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 [22] to our setting. We are also extending this work to more general situations where the interval  $\mathbb{I}$  and the maps  $\text{av}_d$  are replaced by an arbitrary bounded metric space with a system of contractions (see [23] 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

1. Marcial-Romero, J.R., Escardo, M.H.: Semantics of a sequential language for exact real-number computation. *Theor. Comput. Sci.* **379** (2007) 120–141
2. Geuvers, H., Niqui, M., Spitters, B., Wiedijk, F.: Constructive analysis, types and exact real numbers (overview article). *Math. Struct. Comput. Sci.* **17** (2007) 3–36
3. Edalat, A., Heckmann, R.: Computing with real numbers: I. The LFT approach to real number computation; II. A domain framework for computational geometry. In Barthe, G., Dybjer, P., Pinto, L., Saraiva, J., eds.: *Applied Semantics - Lecture Notes from the International Summer School, Caminha, Portugal*. Springer (2002) 193–267

4. Ciaffaglione, A., Di Gianantonio, P.: A certified, corecursive implementation of exact real numbers. *Theor. Comput. Sci.* **351** (2006) 39–51
5. Bertot, Y.: Affine functions and series with co-inductive real numbers. *Math. Struct. Comput. Sci.* **17** (2007) 37–63
6. Berger, U., Hou, T.: Coinduction for exact real number computation. *Theory Comput. Sys.*, to appear (2009)
7. Schwichtenberg, H.: Realizability interpretation of proofs in constructive analysis. *Theory Comput. Sys.*, to appear (2009)
8. Ghani, N., Hancock, P., Pattinson, D.: Continuous functions on final coalgebras. *Electr. Notes in Theoret. Comp. Sci.* **164** (2006)
9. Buchholz, W., Feferman, F., Pohlers, W., Sieg, W.: Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies. Volume 897 of *Lecture Notes in Mathematics*. Springer, Berlin (1981)
10. Bradfield, J., Stirling, C.: Modal  $\mu$ -calculi. In Blackburn, P., van Benthem, J., Wolter, F., eds.: *Handbook of Modal Logic*. Volume 3 of *Studies in Logic and Practical Reasoning*. Elsevier (2007) 721–756
11. Gierz, G., Hofmann, K., Keimel, K., Lawson, J., Mislove, M., Scott, D.: Continuous Lattices and Domains. Volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press (2003)
12. Kreisel, G.: Interpretation of analysis by means of constructive functionals of finite types. *Constructivity in Mathematics* (1959) 101–128
13. Tatsuta, M.: Realizability of monotone coinductive definitions and its application to program synthesis. In Parikh, R., ed.: *Mathematics of Program Construction*. Volume 1422 of *Lecture Notes in Mathematics*., Springer (1998) 338–364
14. Troelstra, A.: *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Volume 344 of *Lecture Notes in Mathematics*. Springer (1973)
15. Abramsky, S., Jung, A.: Domain theory. In Abramsky, S., Gabbay, D.M., Maibaum, T.S.E., eds.: *Handb. Logic Comput. Sci.* Volume 3. Clarendon Press (1994) 1–168
16. Malcolm, G.: Data structures and program transformation. *Science of Computer Programming* **14** (1990) 255–279
17. Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. *Theor. Comput. Sci.* **333** (2005) 3–66
18. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. *Information and Computation* **204** (2006) 437–468
19. Konečný, M.: Real functions incrementally computable by finite automata. *Theor. Comput. Sci.* **315** (2004) 109–133
20. Blanck, J.: Efficient exact computation of iterated maps. *Journal of Logic and Algebraic Programming* **64** (2005) 41–59
21. Plume, D.: *A Calculator for Exact Real Number Computation*. PhD thesis, University of Edinburgh (1998)
22. Benl, H., Berger, U., Schwichtenberg, H., Seisenberger, M., Zuber, W.: Proof theory at work: Program development in the Minlog system. In Bibel, W., Schmitt, P., eds.: *Automated Deduction – A Basis for Applications*. Volume II of *Applied Logic Series*. Kluwer, Dordrecht (1998) 41–71
23. Scriven, A.: A functional algorithm for exact real integration with invariant measures. *Electron. Notes Theor. Comput. Sci.* **218** (2008) 337–353