# Proofs, programs, processes

Ulrich Berger and Monika Seisenberger

Swansea University, Swansea, SA2 8PP, Wales, UK
{u.berger,m.seisenberger}@swansea.ac.uk

**Abstract.** We study a realisability interpretation for inductive and coinductive definitions and discuss its application to program extraction from proofs. A speciality of this interpretation is that realisers are given by terms that correspond directly to programs in a lazy functional programming language such as Haskell. Programs extracted from proofs using coinduction can be understood as perpetual processes producing infinite streams of data. Typical applications of such processes are computations in exact real arithmetic. As an example we show how to extract a program computing the average of two real numbers w.r.t. to the binary signed digit representation.

## 1 Introduction

The purpose of this paper is to provide a theoretical foundation for the extraction of programs from proofs involving inductive and coinductive definitions. Its title is motivated by the fact that programs extracted from proofs using coinduction can often be understood as perpetual processes producing infinite streams of data. Typical applications of such processes are computations in exact real arithmetic. An informal introduction into this subject focusing on intuitive explanations and illustrating examples, but refraining from a stringent formal development, was given in [Ber09a]. In the present paper we provide this formal development. We give a realisability interpretation of our theory of inductive and coinductive definitions with a *Soundness Theorem* stating that from a proof of a formula one can extract a functional program provably realising it.

The programming language where the realisers are taken from is formally represented by an ML-style polymorphically typed $\lambda$-calculus with full recursion. The fact that terms are typed and full recursion is available has two major advantages:

1. The realisation of induction and coinduction is simple and elegant.
2. Realisers can be directly understood as programs in a typed functional programming language with a lazy semantics such as, for example, Haskell.

Another aspect of our system which is of great practical importance is the fact that quantifiers are treated uniformly in the realisability interpretation. This entails that realisers never depend on variables of the logical system (which represents a piece of formalised mathematics) and do not produce output in that

language. Therefore, abstract mathematical objects do not need to be "constructivised", and it is possible to directly extract programs from abstract mathematical proofs. The uniform treatment of first-order quantifiers can be seen as a special case of the interpretations studied by Schwichtenberg [Sch08], Hernest and Oliva [HO08] and Ratiu and Trifonov [RT10], which allow for a fine control of the amount of computational information extracted from proofs.

We illustrate our interpretation by extracting the average function on the real interval $[-1, 1]$ from a proof. The average function (and other arithmetic operations) were implemented and verified before in [CDG06,Plu98]. We would like to stress that while in these papers the programs where "guessed" and verified afterwards, we are able to extract these programs from proofs. This saves ourselves the work to implement not only the algorithm, but also the underlying data structure (streams, in this case) as these are generated automatically through the extraction mechanism. Finally we get the correctness proof for free. So far, the extraction has been carried out "by hand". The implementation in the interactive proof system Minlog [BBS$^+$98,Min] of the extraction method extended to coinductive definitions is ongoing work.

Related interpretations of systems with coinductive definitions were studied by Tatsuta [Tat98], Miranda-Perea [MP05] and by the first author [Ber09b]. The latter paper defines realisability with respect to an untyped $\lambda$-calculus and gives an operational and denotational semantics of the calculus. It also discusses the main differences to the former two works. In the present paper we omit the semantics as it would be very similar to [Ber09b]. Instead, we concentrate on realisability and soundness where we significantly differ from and improve over the above cited work. To our knowledge, the application to the extraction of exact real number algorithms is new.

Due to lack of space we only give the main definitions and state the results. We intend to give a more complete account of our system with full proofs and more substantial applications in a future publication.

## 2 Inductive and and coinductive definitions

We fix a first-order language $\mathcal{L}$. *$\mathcal{L}$-terms*, $r, s, t \ldots$, are built from constants, first-order variables and function symbols as usual. *Formulas*, $A, B, C \ldots$, are $s = t$, $\mathcal{P}(\boldsymbol{t})$ where $\mathcal{P}$ is a predicate (see below), $A \wedge B$, $A \vee B$, $A \to B$, $\forall x\, A$, $\exists x\, A$. A *predicate* is either a predicate constant $P$, or a predicate variable $X$, or a comprehension term also written $\{\boldsymbol{x} \mid A\}$, or an inductive predicate $\mu X.\mathcal{P}$, or a coinductive predicate $\nu X.\mathcal{P}$ where $\mathcal{P}$ is a predicate of the same arity as the predicate variable $X$ and which is *strictly positive (s.p.)* in $X$, i.e. $X$ does not occur free in any premise of a subformula of $\mathcal{P}$ which is an implication. The application, $\mathcal{P}(\boldsymbol{t})$, of a predicate $\mathcal{P}$ to a list of terms $\boldsymbol{t}$ is a primitive syntactic construct, except when $\mathcal{P}$ is a comprehension term, $\mathcal{P} = \{\boldsymbol{x} \mid A\}$, in which case $\mathcal{P}(\boldsymbol{t})$ stands for $A[\boldsymbol{t}/\boldsymbol{x}]$. It will sometimes write $\boldsymbol{x} \in \mathcal{P}$ instead of $\mathcal{P}(\boldsymbol{x})$, $\mathcal{P} \subseteq \mathcal{Q}$ for $\forall \boldsymbol{x}\, (\mathcal{P}(\boldsymbol{x}) \to \mathcal{Q}(\boldsymbol{x}))$ and $\mathcal{P} \cap \mathcal{Q}$ for $\{\boldsymbol{x} \mid \mathcal{P}(\boldsymbol{x}) \wedge \mathcal{Q}(\boldsymbol{x})\}$ etc. We also write $\{t \mid A\}$ as an abbreviation for $\{x \mid \exists \boldsymbol{y}\, (x = t \wedge A)\}$ where $x$ is a fresh variable

and $\boldsymbol{y} = \mathrm{FV}(t) \cap \mathrm{FV}(A)$, as well as $f(\mathcal{P})$ for $\{f(x) \mid x \in \mathcal{P}\}$. Furthermore, we introduce *operators* $\Phi := \lambda \boldsymbol{X}.\mathcal{P}$, and write $\Phi(\boldsymbol{\mathcal{Q}})$ for the predicate $\mathcal{P}[\boldsymbol{\mathcal{Q}}/\boldsymbol{X}]$ where the latter is the usual substitution of the predicates $\boldsymbol{\mathcal{Q}}$ for the predicate variables $\boldsymbol{X}$. $\Phi$ is called a *s.p. operator* if $\mathcal{P}$ is s.p. in $X$. In this case we also write $\mu\Phi$ and $\nu\Phi$ for $\mu X.\mathcal{P}$ and $\nu X.\mathcal{P}$. A formula, predicate, or operator is called *non-computational*, if it contains neither free predicate variables nor the propositional connective $\vee$. Otherwise it is called *computational*.

The *proof rules* are the usual ones of intuitionistic predicate calculus with equality augmented by rules expressing that $\mu\Phi$ and $\nu\Phi$ are the least and greatest fixed points of the operator $\Phi$. As is well-known, the fixed point property can be replaced by appropriate inclusions. Hence we stipulate the axioms

$$\begin{array}{llll} \text{Closure} & \Phi(\mu\Phi) \subseteq \mu\Phi & \text{Induction} & \Phi(\mathcal{Q}) \subseteq \mathcal{Q} \to \mu\Phi \subseteq \mathcal{Q} \\ \text{Coclosure} & \nu\Phi \subseteq \Phi(\nu\Phi) & \text{Coinduction} & \mathcal{Q} \subseteq \Phi(\mathcal{Q}) \to \mathcal{Q} \subseteq \nu\Phi \end{array}$$

for all s.p. operators $\Phi$ and predicates $\mathcal{Q}$. In addition we allow any axioms expressible by non-computational formulas that hold in the intended model. We write $\Gamma \vdash A$ if $A$ is derivable from assumptions in $\Gamma$ in this system. If $A$ is derivable without assumptions we write $\vdash A$, or even just $A$. We define falsity as $\bot := \mu X.X$ where $X$ is a propositional variable (i.e. a 0-ary predicate variable). From the induction axiom for $\bot$ follows directly $\bot \to A$ for every formula $A$. As a running example we use the first-order language of the ordered real numbers. As axioms we adopt any non-computational formulas that are true in the structure of real numbers, e.g. the axioms of a real closed field where anti-symmetry and linearity of the order are expressed non-computationally, e.g. by $\forall x, y\,((y \not< x \wedge x \not< y) \leftrightarrow x = y)$. All sets we define in the following are subsets of the set of real numbers. We define the set $\mathbb{N}$ of natural numbers as usual inductively by

$$\mathbb{N} := \mu X.\{0\} \cup \{x + 1 \mid X(x)\}$$

Note that, since $\forall x, y \in \mathbb{N}\,(x < y \vee x = y \vee y < x)$ is (easily) provable, equality of natural number is decidable. Next we define coinductively a constructive analogue of the closed interval $\mathbb{I} := [-1, 1] \subseteq \mathbb{R}$. Let $\mathrm{SD} := \{0, 1, -1\}$ be the set of signed binary digits. We define coinductively

$$\mathrm{C}_0 := \nu X.\{(i + x)/2 \in \mathbb{I} \mid \mathrm{SD}(i) \wedge X(x)\}$$

It is easy to see that, classically, $\mathrm{C}_0$ coincides with $\mathbb{I}$. The point is that from a constructive proof of $\mathrm{C}_0(x)$ we can extract a program computing an infinite signed digit representation of $x$.

## 3 Programs

We now introduce an extended $\lambda$-calculus which we will use in Sect. 4 as our programming language, i.e. language of realisers, as well as a typing discipline

which will facilitate the definition of map operators, iterators and coiterators as realisers of monotonicity, induction and coinduction. Program terms (which we will simply call "terms" in the following) are given by the grammar

$$M, N, K, L ::= x \mid \lambda x.M \mid M\,N \mid \mathrm{rec}\,x\,.\,M \mid C(M_1, \ldots, M_n) \mid \\ \mathrm{case}\,M\,\mathrm{of}\{C_1(\boldsymbol{x_1}) \to R_1\,;\,\ldots\,;\,C_n(\boldsymbol{x_n}) \to R_n\}$$

where $x$ ranges over a set of variables, $C$ ranges over a set of constructors (each with a fixed arity) and in $\mathrm{case}\,M\,\mathrm{of}\{C_1(\boldsymbol{x_1}) \to R_1\,;\,\ldots\,;\,C_n(\boldsymbol{x_n}) \to R_n\}$ all constructors $C_i$ are distinct and each $\boldsymbol{x_i}$ is a vector of distinct variables. We axiomatise this calculus by the equations

$$(\lambda x.M)N = M[N/x] \qquad \mathrm{rec}\,x\,.\,M = M[\mathrm{rec}\,x\,.\,M/x]$$
$$\mathrm{case}\,C_i(\boldsymbol{K})\,\mathrm{of}\{\ldots;C_i(\boldsymbol{x_1}) \to R_i\,;\,\ldots\} = R_i[\boldsymbol{K}/\boldsymbol{x_i}]$$

We write $\vdash M = N$ if the equation $M = N$ can be derived from these axioms by the usual rules of equational logic.

The typing we introduce now serves two purposes. First, types are used as indices for terms realising monotonicity, induction and coinduction. Second, we will show that extracted programs are typeable and hence are valid programs in typed functional programming languages such as Haskell or ML. Types are constructed from type variables $\alpha, \beta, \ldots \in \mathrm{TVar}$ according to the grammar

$$\mathrm{Type} \ni \rho, \sigma, \tau ::= \alpha \mid \rho \to \sigma \mid \mathbf{1} \mid \rho \times \sigma \mid \rho + \sigma \mid \mathrm{fix}\,\alpha.\rho$$

We consider the instance of our term language determined by the constructors Nil (nullary), Left, Right (unary), Pair (binary), and $\mathrm{In}_{\mathrm{fix}\,\alpha.\rho}$ (unary) for every fixed point type $\mathrm{fix}\,\alpha.\rho$. We inductively define the relation $\Gamma \vdash M : \rho$ (term $M$ is of type $\rho$ in environment $\Gamma$.)

*Variable, recursion and λ-calculus rules.*

$$\Gamma, x : \rho \vdash x : \rho \qquad \frac{\Gamma, x : \tau \vdash M : \tau}{\Gamma \vdash \mathrm{rec}\,x\,.\,M : \tau}$$

$$\frac{\Gamma, x : \rho \vdash M : \sigma}{\Gamma \vdash \lambda x.M : \rho \to \sigma} \qquad \frac{\Gamma \vdash M : \rho \to \sigma \qquad \Gamma \vdash N : \rho}{\Gamma \vdash M\,N : \sigma}$$

*Constructor rules.*

$$\Gamma \vdash \mathrm{Nil} : \mathbf{1} \qquad \frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \mathrm{Pair}(M, N) : \rho \times \sigma}$$

$$\frac{\Gamma \vdash M : \rho}{\Gamma \vdash \mathrm{Left}(M) : \rho + \sigma} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \mathrm{Right}(M) : \rho + \sigma}$$

$$\frac{\Gamma \vdash M : \rho_0(\rho(\boldsymbol{\sigma}), \boldsymbol{\sigma})}{\Gamma \vdash \mathrm{In}_\rho(M) : \rho(\boldsymbol{\sigma})} \quad \text{where } \rho = \rho(\boldsymbol{\alpha}) = \mathrm{fix}\,\alpha.\rho_0(\alpha, \boldsymbol{\alpha}).$$

*Elimination rules for constructors.* The constructor rules for each type constructor determine one elimination rule. For example:

$$\frac{\Gamma \vdash M : \rho + \sigma \qquad \Gamma, x_1 : \rho \vdash L : \tau \qquad \Gamma, x_2 : \sigma \vdash R : \tau}{\Gamma \vdash \mathrm{case}\, M\, \mathrm{of}\{\mathrm{Left}(x_1) \to L\,;\mathrm{Right}(x_2) \to R\} : \tau}$$

$$\frac{\Gamma \vdash M : \rho(\boldsymbol{\sigma}) \qquad \Gamma, x : \rho_0(\rho(\boldsymbol{\sigma}), \boldsymbol{\sigma}) \vdash N : \tau}{\Gamma \vdash \mathrm{case}\, M\, \mathrm{of}\{\mathrm{In}_\rho(x) \to N\} : \tau} \quad (\rho = \rho(\boldsymbol{\alpha}) = \mathrm{fix}\,\alpha.\rho_0(\alpha, \boldsymbol{\alpha}))$$

As a preparation for the realisability interpretation in Sect. 4 we define map-terms, iterators and coiterators that will be used as realisers for the monotonicity of strictly positive predicate transformers, induction and coinduction. The following definition refers to a fixed one-to-one assignment of variables $f_\alpha$ to type variables $\alpha$. For every list of type variables $\boldsymbol{\alpha}$ and every type $\rho$ which is s.p. in $\boldsymbol{\alpha}$ we define the term $\mathbf{map}_{\boldsymbol{\alpha};\rho} := \lambda f_{\alpha_1}, \ldots, f_{\alpha_n}.\, \mathbf{Map}_{\boldsymbol{\alpha};\rho}$ where

$$\begin{aligned}
\mathbf{Map}_{\boldsymbol{\alpha};\alpha_i} &= f_i, \qquad \mathbf{Map}_{\boldsymbol{\alpha};\rho} = \lambda x.\, x, \text{ if no } \alpha_i \text{ occurs in } \rho \\
\mathbf{Map}_{\boldsymbol{\alpha};\rho+\sigma} &= \lambda x.\mathrm{case}\, x\, \mathrm{of}\{\mathrm{Left}(y) \to \mathrm{Left}(\mathbf{Map}_{\boldsymbol{\alpha};\rho} y)\,; \\
&\qquad\qquad \mathrm{Right}(z) \to \mathrm{Right}(\mathbf{Map}_{\boldsymbol{\alpha};\sigma} z)\} \\
\mathbf{Map}_{\boldsymbol{\alpha};\rho\times\sigma} &= \lambda x.\, \mathrm{case}\, x\, \mathrm{of}\{\mathrm{Pair}(y, z) \to \mathrm{Pair}(\mathbf{Map}_{\boldsymbol{\alpha};\rho} y, \mathbf{Map}_{\boldsymbol{\alpha};\sigma} z)\} \\
\mathbf{Map}_{\boldsymbol{\alpha};\rho\to\sigma} &= \lambda x.\, \lambda y.\, \mathbf{Map}_{\boldsymbol{\alpha};\sigma}(x\, y) \\
\mathbf{Map}_{\boldsymbol{\alpha};\mathrm{fix}\,\alpha.\rho} &= \mathrm{rec}\, f_\alpha\, .\, \lambda x.\, \mathrm{case}\, x\, \mathrm{of}\{\mathrm{In}_{\mathrm{fix}\,\alpha.\rho}(y) \to \mathrm{In}_{\mathrm{fix}\,\alpha.\rho}(\mathbf{Map}_{\boldsymbol{\alpha},\alpha;\rho} y)\}
\end{aligned}$$

A type is called *regular* if in its construction the clause $\mathrm{fix}\,\alpha.\rho$ is applied only if $\rho$ is s.p. in $\alpha$. In the following all mentioned types are assumed to be regular.

**Lemma 1 (Typing of map).** *Let $\rho = \rho(\boldsymbol{\alpha})$ be s.p. in $\boldsymbol{\alpha}$. Then*

$$\vdash \mathbf{map}_{\boldsymbol{\alpha};\rho} : (\boldsymbol{\sigma} \to \boldsymbol{\tau}) \to \rho(\boldsymbol{\sigma}) \to \rho(\boldsymbol{\tau})$$

For every (regular) type $\mathrm{fix}\,\alpha.\rho$ we define iterator and coiterator by

$$\begin{aligned}
\mathbf{It}_{\mathrm{fix}\,\alpha.\rho} &:= \lambda s.\, \mathrm{rec}\, f\, .\, \lambda x.\, \mathrm{case}\, x\, \mathrm{of}\{\mathrm{In}_{\mathrm{fix}\,\alpha.\rho}(y) \to s(\mathbf{map}_{\alpha;\rho} f y)\} \\
\mathbf{Coit}_{\mathrm{fix}\,\alpha.\rho} &:= \lambda s.\mathrm{rec}\, f\, .\, \lambda x.\mathrm{In}_{\mathrm{fix}\,\alpha.\rho}(\mathbf{map}_{\alpha;\rho} f(s\, x))
\end{aligned}$$

**Lemma 2 (Typing of iterator and coiterator).** *For all types $\sigma, \boldsymbol{\sigma}$*

$$\begin{aligned}
&\vdash \mathbf{It}_{\mathrm{fix}\,\alpha.\rho} : (\rho(\sigma, \boldsymbol{\sigma}) \to \sigma) \to \mathrm{fix}\,\alpha.\rho(\boldsymbol{\sigma}) \to \sigma \\
&\vdash \mathbf{Coit}_{\mathrm{fix}\,\alpha.\rho} : (\sigma \to \rho(\sigma, \boldsymbol{\sigma})) \to \sigma \to \mathrm{fix}\,\alpha.\rho(\boldsymbol{\sigma})
\end{aligned}$$

In the following let $\rho = \mathrm{fix}\,\alpha.\rho_0(\alpha)$. We set $\mathbf{in}_\rho := \lambda y.\, \mathrm{In}_\rho(y)$ and $\mathbf{out}_\rho := \lambda x.\, \mathrm{case}\, x\, \mathrm{of}\{\mathrm{In}_\rho(y) \to y\}$. Note that $\vdash \mathbf{in}_\rho : \rho_0(\rho) \to \rho$ and $\vdash \mathbf{out}_\rho : \rho \to \rho_0(\rho)$, i.e. $\mathbf{in}_\rho$ resp. $\mathbf{out}_\rho$ is an algebra resp. coalgebra for the functor $\alpha \mapsto \rho_0(\alpha)$. The next lemma states that the iterator resp. coiterator witnesses the initiality resp. finality of $\mathbf{in}_\rho$ resp. $\mathbf{out}_\rho$. ($M \circ N := \lambda z.M(Nz)$, $z$ fresh.)

**Lemma 3 (Initiality and finality).**

*(a)* $\vdash \mathbf{It}_\rho s \circ \mathbf{in}_\rho = s \circ \mathbf{map}_{\alpha;\rho_0(\alpha)}(\mathbf{It}_\rho s)$
*(b)* $\vdash \mathbf{out}_\rho \circ \mathbf{Coit}_\rho s = \mathbf{map}_{\alpha;\rho_0(\alpha)}(\mathbf{Coit}_\rho s) \circ s$

## 4 Realisability

We now introduce a formalised realisability interpretation of the theory of inductive and coinductive definitions of Sect. 2. To this end we need a system that can talk about mathematical objects *and* realisers. Therefore we extend our first-order language $\mathcal{L}$ to a language $\mathbf{r}(\mathcal{L})$ by adding a new sort for program terms. All logical operations including inductive and coinductive definitions, as well as axioms and rules for $\mathcal{L}$ including closure, induction, coclosure and coinduction and the rules for equality, are extended mutatis mutandis for $\mathbf{r}(\mathcal{L})$. In addition, we have as extra axioms the equations given in Sect. 3.

We assign to every $\mathcal{L}$-formula $A$ a unary $\mathbf{r}(\mathcal{L})$-predicate $\mathbf{r}(A)$ on program terms. Intuitively, $\mathbf{r}(A)(a)$, sometimes also written $a\,\mathbf{r}\,A$, states that $a$ "realises" $A$. The predicate $\mathbf{r}(A)$ is defined relative to a fixed one-to-one mapping from $\mathcal{L}$-predicate variables $X$ to $\mathbf{r}(\mathcal{L})$-predicate variables $\tilde{X}$ with one extra argument place for program terms. The definition of $\mathbf{r}(A)$ is such that if the formula $A$ has the free predicate variables $X_1, \ldots, X_n$, then the predicate $\mathbf{r}(A)$ has the free predicate variables $\tilde{X}_1, \ldots, \tilde{X}_n$. Simultaneously with $\mathbf{r}(A)$ we define a predicate $\mathbf{r}(\mathcal{P})$ for every predicate $\mathcal{P}$, where $\mathbf{r}(\mathcal{P})$ has one extra argument place for domain elements. We also define regular types $\tau(A)$ and $\tau(\mathcal{P})$ relative to a fixed assignment of a type variable $\alpha_X$ to each predicate variable $X$.

If $A$ and $\mathcal{P}$ are non-computational:

$$\begin{array}{llll} \mathbf{r}(A) & = \{\mathrm{Nil} \mid A\} & \tau(A) & = \mathbf{1} \\ \mathbf{r}(\mathcal{P}) & = \{(\mathrm{Nil}, \boldsymbol{x}) \mid \mathcal{P}(\boldsymbol{x})\} & \tau(\mathcal{P}) & = \mathbf{1} \end{array}$$

If $A$ is non-computational but $B$ is:

$$\begin{array}{llll} \mathbf{r}(A \wedge B) & = \mathbf{r}(B \wedge A) = \{x \mid A \wedge \mathbf{r}(B)(x)\} & \tau(A \wedge B) & = \tau(B \wedge A) = \tau(B) \\ \mathbf{r}(A \to B) & = \{x \mid A \to \mathbf{r}(B)(x)\} & \tau(A \to B) & = \tau(B) \end{array}$$

In all other cases:

$$\begin{array}{llll} \mathbf{r}(\mathcal{P}(\boldsymbol{t})) & = \{x \mid \mathbf{r}(\mathcal{P})(x, \boldsymbol{t})\} & \tau(\mathcal{P}(\boldsymbol{t})) & = \tau(\mathcal{P}) \\ \mathbf{r}(A \wedge B) & = \mathrm{Pair}(\mathbf{r}(A), \mathbf{r}(B)) & \tau(A \wedge B) & = \tau(A) \times \tau(B) \\ \mathbf{r}(A \vee B) & = \mathrm{Left}(\mathbf{r}(A)) \cup \mathrm{Right}(\mathbf{r}(B)) & \tau(A \vee B) & = \tau(A) + \tau(B) \\ \mathbf{r}(A \to B) & = \{f \mid f(\mathbf{r}(A)) \subseteq \mathbf{r}(B)\} & \tau(A \to B) & = \tau(A) \to \tau(B) \\ \mathbf{r}(\forall y\, A) & = \{x \mid \forall y\, (\mathbf{r}(A)(x))\} & \tau(\forall y\, A) & = \tau(A) \\ \mathbf{r}(\exists y\, A) & = \{x \mid \exists y\, (\mathbf{r}(A)(x))\} & \tau(\exists y\, A) & = \tau(A) \\ \mathbf{r}(\{\boldsymbol{x} \mid A\}) & = \{(y, \boldsymbol{x}) \mid \mathbf{r}(A)(y)\} & \tau(\{\boldsymbol{x} \mid A\}) & = \tau(A) \\ \mathbf{r}(X) & = \tilde{X} & \tau(X) & = \alpha_X \\ \mathbf{r}(\mu X.\mathcal{P}) & = \mu\tilde{X}.\{(\mathrm{In}(y), x) \mid \mathbf{r}(\mathcal{P})(y, x)\} & \tau(\mu X.\mathcal{P}) & = \mathrm{fix}\,\alpha_X.\tau(\mathcal{P}) \\ \mathbf{r}(\nu X.\mathcal{P}) & = \nu\tilde{X}.\{(\mathrm{In}(y), x) \mid \mathbf{r}(\mathcal{P})(y, x)\} & \tau(\nu X.\mathcal{P}) & = \mathrm{fix}\,\alpha_X.\tau(\mathcal{P}) \end{array}$$

where in the last two equations $\mathrm{In} := \mathrm{In}_{\mathrm{fix}\,\alpha_X.\tau(\mathcal{P})}$.

For example, $\tau(\mathbb{N}) = \mathrm{fix}\,\alpha.\mathbf{1} + \alpha$, the usual recursive definition of the data type of unary natural numbers. Its canonical inhabitants are the numerals $\underline{k} := \mathrm{inr}^k(\mathrm{inl}(\mathrm{Nil}))$ $(k \in \mathbb{N})$ where $\mathrm{inl}(x) := \mathrm{In}(\mathrm{Left}(x))$ and $\mathrm{inr}(x) := \mathrm{In}(\mathrm{Right}(x))$. Realisability for $\mathbb{N}$, $\mathbf{r}(\mathbb{N})$, is the least relation such that

$$\mathbf{r}(\mathbb{N}) = \{(\mathrm{inl}(\mathrm{Nil}), 0)\} \cup \{(\mathrm{inr}(n), x+1) \mid \mathbf{r}(\mathbb{N})(n, x)\}$$

Hence, we have for a term $d$ and $k \in \mathbb{R}$ that $d\,\mathbf{r}\,\mathbb{N}(k)$ holds iff $k$ is a natural number and $d = \underline{k}$, i.e. $d$ is a unary representation of $k$.

Regarding $\mathrm{C}_0$, if we identify the set SD with the type $\mathbf{1} + \mathbf{1} + \mathbf{1}$ and every $i \in \mathrm{SD}$ with its corresponding program term, then $\tau(\mathrm{C}_0) = \mathrm{fix}\,\alpha.\mathrm{SD} \times \alpha$, the type of infinite streams of signed digits, and

$$\mathbf{r}(\mathrm{C}_0) = \nu\tilde{X}.\{(\mathrm{Pair}(i, a), (i+x)/2) \mid i \in \mathrm{SD} \wedge |(i+x)/2| \leq 1 \wedge \tilde{X}(a, x)\}$$

It is easy to see that $\mathbf{r}(\mathrm{C}_0)(a, x)$ means that the signed digit stream $a = a_0, a_1, \ldots$ represents $x$ i.e. $x = \Sigma_{i=0}^{\infty} 2^{-(i+1)} * a_i$.

**Lemma 4 (Map).** *Let $\Phi = \lambda X.\mathcal{P}$ be a (strictly positive) operator in the language $\mathcal{L}$, $\alpha := \alpha_X$, and $\rho := \tau(\mathcal{P})$. Then $\mathbf{map}_{\alpha;\rho}$ realises the monotonicity of $\Phi$, that is*

$$\mathbf{map}_{\alpha;\rho}\,\mathbf{r}\,(\mathcal{P} \subseteq \mathcal{Q} \to \Phi(\mathcal{P}) \subseteq \Phi(\mathcal{Q}))$$

*for all $\mathcal{L}$-predicates $\mathcal{P}$ and $\mathcal{Q}$.*

The previous lemmas are the essential facts needed to prove:

**Theorem 1 (Soundness).** *From a closed derivation of a formula $A$ one can extract a program term $M$ such that $\mathbf{r}(A)(M)$ and $M\colon\tau(A)$ are derivable.*

From the Soundness Theorem one can easily conclude that the program extracted from the proof of a *data formula*, i.e. a formula without free predicate variables and such that every subformula of the form $A \to B$ or $\nu\Phi(\boldsymbol{t})$ is non-computational, "evaluates" to a data term, i.e. a closed term built from constructors only, which realises $A$. For details about evaluating terms w.r.t. a call-by-name semantics see [Ber09b].

## 5 Example: extraction of the average function

As an example we extract from a proof that the predicate $\mathrm{C}_0$ is closed under averages a program computing the average of two real numbers in the interval $\mathbb{I}$ w.r.t. the signed digit representation.

**Lemma 5.** *If $x, y \in \mathrm{C}_0$, then $\frac{x+y}{2} \in \mathrm{C}_0$.*

*Proof.* Set $X := \{\frac{x+y}{2} \mid x, y \in \mathrm{C}_0\}$. In order to show $X \subseteq \mathrm{C}_0$, it suffices to find a set $Y \subseteq \mathbb{R}$ such that $X \subseteq Y$ and $Y \subseteq \Phi(Y)$ (hence $Y \subseteq \mathrm{C}_0$, by coinduction). Setting $\mathrm{SD}_2 := \{-2, -1, 0, 1, 2\}$ we define

$$Y := \{\frac{x+y+i}{4} \mid x, y \in \mathrm{C}_0, i \in \mathrm{SD}_2\}$$

*Proof of "$X \subseteq Y$"*: Let $x, y \in \mathrm{C}_0$. We have to show that $z := \frac{x+y}{2} \in Y$. By the coclosure axiom, $x = \frac{x'+d}{2}$, $y = \frac{y'+e}{2}$, for some $d, e \in \mathrm{SD}$ and $x', y' \in \mathrm{C}_0$. Hence $z = \frac{x'+y'+d+e}{4} \in Y$.

*Proof of "$Y \subseteq \Phi(Y)$"*: Let $x, y \in \mathrm{C}_0$ and $i \in \mathrm{SD}_2$. We have to show that $z := \frac{x+y+i}{4} \in \Phi(Y)$. By the coclosure axiom, $x = \frac{x'+d'}{2}$ and $y = \frac{y'+e'}{2}$, for some $d', e' \in \mathrm{SD}$ and $x', y' \in \mathrm{C}_0$. Hence $z = \frac{x'+y'+d'+e'+2i}{8}$. Since $z \in \mathbb{I}$, it suffices to find $d \in \mathrm{SD}$ and $\tilde{z} \in Y$ such that $z = \frac{\tilde{z}+d}{2}$. By the definition of $Y$ this means that we must find $\tilde{x}, \tilde{y} \in \mathrm{C}_0$ and $j \in \mathrm{SD}_2$ such that $\tilde{z} = \frac{\tilde{x}+\tilde{y}+j}{4}$, i.e. $z = \frac{\tilde{x}+\tilde{y}+j+4d}{8}$. Choosing $\tilde{x} := x'$ and $\tilde{y} := y'$ we are left with the equation $d' + e' + 2i = j + 4d$ which is clearly solvable with suitable $d \in \mathrm{SD}$ and $j \in \mathrm{SD}_2$.

Applying our method of program extraction to this proof one obtains the following Haskell program `average`. It takes two infinite streams and first reads the first digits $d$ and $e$ on both of them; this corresponds to the proof of "$X \subseteq Y$". The functional `aux` recursively calls itself; this corresponds to the use of the coinduction principle with coiteration as realiser. The remainder of the program links to the computational content of the proof of "$Y \subseteq \Phi(Y)$" in an obvious way.

```
type SD = Int   -- -1, 0, 1 only
type SDS = [SD] -- infinite streams only
type SD2 = Int  -- -2, -1, 0, 1, 2 only

average :: SDS -> SDS -> SDS
average (d:ds) (e:es) = aux (d+e) ds es  where

   aux :: SD2 -> SDS -> SDS -> SDS
   aux i (d':ds) (e':es) = d : aux j ds es  where

      k = d'+e'+2*i

      d | abs k <= 2  = 0
        | k > 2       = 1
        | otherwise   = -1

      j = k-4*d
```

As a demo we run the extracted program with inputs `[1,0,1,0,0,0,...]`$= \frac{5}{8}$ and `[1,1,0,0,0,...]` $= \frac{3}{4}$. Looking at the first 10 digits of the computed stream

```
Main> take 10 (average ([1,0,1]++[0,0..]) ([1,1]++[0,0..]))
[1,1,0,-1,0,0,0,0,0,0]
```

one sees that the result is correct, as $(\frac{5}{8} + \frac{3}{4})/2 = \frac{11}{16} = \frac{1}{2} + \frac{1}{4} - \frac{1}{16}$.

## 6 Conclusion and further work

In our opinion, one of the main advantages of program extraction over the traditional specify-implement-verify method is that it is possible to carry out proofs in a very simple formal system. Neither complicated data types (lists, streams, trees, function types, etc.) nor programming constructs (recursion, lambda-abstraction) need to be formalised by the user; these are all generated by the realisability interpretation automatically.

On the basis of the results of this paper one can now begin to formalise parts of constructive analysis and other branches of mathematics where inductive and coinductive definitions are used (or can be used), with the aim of extracting nontrivial certified programs. Currently, we are investigating a generalisation of the predicate $\mathrm{C}_0 \subseteq \mathbb{R}$ (one of our running examples) to predicates $\mathrm{C}_n \subseteq \mathbb{R}^{\mathbb{I}^n}$ characterising the (constructively) uniformly continuous function from $\mathbb{I}^n$ to $\mathbb{I}$ [Ber09a]. For $n = 1$ the definition is

$$\mathrm{C}_1 := \nu F.\mu G.\{f \in \mathbb{I}^{\mathbb{I}} \mid \exists i \in \mathrm{SD}\, \exists f'\, (f = \mathrm{av}_i \circ f' \wedge F(f')) \ \vee \bigwedge_{i \in \mathrm{SD}} G(f \circ \mathrm{av}_i)\}$$

where $F$ and $G$ range over subsets of $\mathbb{R}^{\mathbb{I}}$ and $\mathrm{av}_i(x) := (i + x)/2$. To see the analogy with $\mathrm{C}_0$ it is useful to rewrite the definition of the latter equivalently as

$$\mathrm{C}_0 := \nu X.\{x \in \mathbb{I} \mid \exists i \in \mathrm{SD}\, \exists x'\, (x = \mathrm{av}_i(x') \wedge X(x'))\}$$

The predicate $\mathrm{C}_0$ characterises real numbers in $\mathbb{I}$ as objects perpetually emitting digits. A continuous function $f : \mathbb{I} \to \mathbb{I}$, which can be viewed as a real number in $\mathbb{I}$ that depends on an input in $\mathbb{I}$, perpetually emits digits as well, but before an emission can take place $f$ may have to gain information about the input by absorbing finitely many digits from it in order to decide which digit to emit. The absorption part is formalised in $\mathrm{C}_1$ by the inner "$\mu G \ldots G(f \circ \mathrm{av}_i)$". The data type associated with $\mathrm{C}_1$ is

$$\tau(\mathrm{C}_1) = \nu\alpha.\mu\beta.\mathrm{SD} \times \alpha + \beta^3$$

which is the type of non-wellfounded trees with two kinds of nodes: one kind labelled by a signed digit and one child (emitting a digit), the other kind without label and three children (absorbing a digit). The fact that $\beta$ is quantified by $\mu$ means that only those trees are legal members of $\tau(\mathrm{C}_1)$ that have on each path infinitely many emitting nodes. A similar type of trees has been studied independently in [GHP06], however, not in the context of analysis and realisability. The definition of $\mathrm{C}_1$ is motivated by earlier works on the development and verification of exact real number algorithms based on the signed digit representation of real numbers [MRE07,GNSW07,EH02] some of which make use of coinductive methods [CDG06,Ber07,BH08,Niq08].

Based on the characterisation of uniformly continuous functions by the predicates $\mathrm{C}_n$ implementations of elementary arithmetic functions have been extracted [Ber09a]. Further work in progress is the automatization of this form of program extraction in the Minlog proof system [BBS+98], and the study of integration and analytic functions based on this approach.

## References


[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. *Math. Struct. Comput. Sci.*, 17:37–63, 2007.

[Ber09a] U. Berger. From coinductive proofs to exact real arithmetic. In E. Grädel and R. Kahle, editors, *Computer Science Logic*, volume 5771 of *LNCS*, pages 132–146. Springer, 2009.

[Ber09b] U. Berger. Realisability and adequacy for (co)induction. In Andrej Bauer, Peter Hertling, and Ker-I Ko, editors, *6th Int'l Conf. on Computability and Complexity in Analysis (Ljubljana)*, Dagstuhl, Germany, 2009. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[BH08] U. Berger and T. Hou. Coinduction for exact real number computation. *Theory of Computing Systems*, 43:394–409, 2008.

[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.

[GHP06] N. Ghani, P. Hancock, and D. Pattinson. Continuous functions on final coalgebras. *Electr. Notes in Theoret. Comput. Sci.*, 164, 2006.

[GNSW07] H. Geuvers, M. Niqui, B. Spitters, and F. Wiedijk. Constructive analysis, types and exact real numbers. *Math. Struct. Comput. Sci.*, 17(1):3–36, 2007.

[HO08] M. D. Hernest and P. Oliva. Hybrid functional interpretations. In A. Beckmann, C. Dimitracopoulos, and B. Löwe, editors, *CiE 2008: Logic and Theory of Algorithms*, volume 5028 of *LNCS*, pages 251–260. Springer, 2008.

[Min] The Minlog System. http://www.mathematik.uni-muenchen.de/∼minlog/.

[MP05] F. Miranda-Perea. Realizability for monotone clausular (co)inductive definitions. *Electr. Notes in Theoret. Comput. Sci.*, 123:179–193, 2005.

[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.

[Niq08] M. Niqui. Coinductive formal reasoning in exact real arithmetic. *Logical Methods in Computer Science*, 4(3:6):1–40, September 2008.

[Plu98] D. Plume. *A Calculator for Exact Real Number Computation.* PhD thesis, University of Edinburgh, 1998.

[RT10] D. Ratiu and T. Trifonov. Exploring the computational content of the infinite pigeonhole principle. To appear in Journal of Logic and Computation, 2010.

[Sch08] H. Schwichtenberg. Realizability interpretation of proofs in constructive analysis. *Theory of Computing Systems*, 43(3-4):583–602, 2008.

[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.