Electronic Communications of the EASST
Volume 23 (2009)

Proceedings of the
Ninth International Workshop on
Automated Verification of Critical Systems
(AVOCS 2009)

A coinductive approach to verified exact real number computation

Ulrich Berger and Sion Lloyd

15 pages

Guest Editor: Markus Roggenbach
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A coinductive approach to verified exact real number computation

Ulrich Berger and Sion Lloyd

University of Wales Swansea, Swansea, SA2 8PP, Wales UK

Abstract: We present an approach to verified programs for exact real number computation that is based on inductive and coinductive definitions and program extraction from proofs. We informally discuss the theoretical background of this method and give examples of extracted programs implementing the translation between the representation by fast converging rational Cauchy sequences and the signed binary digit representations of real numbers.

Keywords: Proof theory, program extraction, exact real number computation, coinduction

1 Introduction

In current implementations of main stream programming languages real numbers are represented in floating point format, and computation on real numbers is done with respect to this representation. As is well-known, rounding errors in floating point arithmetic may occur, and inevitably do so, due to the limited precision of floating point numbers on the one hand and the infinitary character of the real numbers on the other hand. The problem with this is not so much the fact that these rounding errors occur already in relatively simple computations, but that the user has no control over them. As a simple example consider the function

```
f(x) = 1+x-(x^2)*(x+1)*((1/x)-(1/(x+1)))     -- Haskell code
```

and the computations

```
*Main> f(10^4) :: Float
2.834961
*Main> f(10^4) :: Double
1.0000000006384653
*Main> f(10^9) :: Double
-149.21128177642822
```

Which of these results can we trust? Actually, in all three cases the correct result is 1.0, which can be easily seen by applying elementary school algebra to the expression defining the function f. The situation we encounter here is typical: in order to estimate the accuracy of a floating point computation one needs, in principle, always a mathematical analysis of the numerical stability of the problem which can be arbitrarily difficult. Increasing the precision cannot replace such an analysis because one does not know by how much the precision needs to be increased in order to obtain a required number of correct digits. In view of safety critical applications of numeric computation, for example autopilot systems for aircrafts, these problems can no longer be neglected, but require alternative approaches that have a sound mathematical and technological basis. Such approaches are currently promoted under the slogan “computing with exact real numbers”. In exact real number computation results are not necessarily exact, but they are guaranteed to be correct with any accuracy prescribed by the user. This means the user has full control over errors. Of course, it still can happen that a given accuracy cannot be obtained due to limited resources in time and space, but it will never happen that a result is delivered without information about its accuracy.

A further essential requirement in exact real number computation is that the correctness of a program has to be *proven* (in a stringent mathematical sense) in order to justify the user’s trust in it. The generation of provably correct programs in exact real number computation is the focus of this paper.

In traditional program verification one takes a program and applies a certain method to prove that it meets a given specification (see the seminal papers [Flo67, Hoa97, Dij75, Pnu77, Mil80] and systems supporting program verification, e.g. PVS [ORSS98], Isabelle [NPW02], Coq [Coq], KIV [BRS+00], ACL2 [KMM00], BLAST [Bez07]). Another approach is to develop or derive programs according to certain rules that preserve correctness, thus obtaining programs that are correct “by construction” [Dij97, Gri81, DJ78]. The method we are presenting can be seen as a rather radical instance of the latter approach. From a formal constructive proof of a mathematical statement A we extract a program that “realises” A. In general, the statement A does not need to be related to programming, but in specific cases A may be viewed as a specification of a computational problem that is solved by the extracted program. By a constructive proof we mean a proof that does not make use of the law of excluded middle or equivalent principles. Constructive reasoning is being adopted in Intuitionism [vH67, Hey56, Tro73], Constructive Mathematics [BB85] and Constructive Type Theory [ML84], and is implemented in a number of interactive proof systems, for example, NuPrL [Con86], Coq [Coq], Minlog [BBS+98] and Agda [Agd]. The logical basis of program extraction via realisability was laid by Kleene [Kle45] and Kreisel [Kre59] (for proof-theoretic purposes). It is an instance of what is known in Computer Science as the “Curry-Howard correspondence” or “proofs-as-programs” paradigm which is applicable to constructive proofs in a wide range of areas. In this paper we concentrate on constructive real analysis and we illustrate the method by some simple yet non-trivial examples. We want to make the case that it is not only possible in principle, but also feasible in practice to extract interesting programs from proofs (see also [Sch09] for related work on program extraction in constructive analysis).

An important principle for definitions and proofs we are using is *coinduction*. A coinductive definition can be viewed set-theoretically as the largest fixed-points of a monotone set operator or category-theoretically as the final coalgebra of a functor. Recently, coinductive definitions, coalgebras and coinductive proofs have become very popular for describing concurrent systems and cryptographic protocols [BS07, JR97, Mos99, Rut00, HW03]. Also, much of the recent work on exact real number computation uses coinduction to verify real number algorithms w.r.t. the representation of real numbers by infinite streams of signed digits and similar representations [EH02, ME07, GNSW07, CD06, Ber07, BH08]. In our paper we go one step further and extract these algorithms from proofs about coinductive characterisations of real numbers.

2 Induction and coinduction

We give a brief introduction to coinduction and the dual principle of induction. We begin with the latter as it is more familiar.

Consider an operator $\Phi : \mathscr{P}(U) \to \mathscr{P}(U)$, where U is a set and $\mathscr{P}(U)$ is the powerset of U, and assume that Φ is monotone, i.e. if $X \subseteq Y \subseteq U$, then $\Phi(X) \subseteq \Phi(Y)$. Since $\mathscr{P}(U)$ is a complete lattice w.r.t. set inclusion Φ has a least fixed point $\mu\Phi$, according to the Knaster-Tarski Theorem. In fact, $\mu\Phi$ is the least Φ-closed subset of U where a set $X \subseteq U$ is called Φ-closed if $\Phi(X) \subseteq X$. Hence we have the *closure principle* for $\mu\Phi$

$$\Phi(\mu\Phi) \subseteq \mu\Phi$$

as well as the *induction principle*

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

for all subsets X of U (one often says "induction on $\mu\Phi$"). In many cases Φ has a definition of the form $\Phi(X) = \{u \in U | A(u) \vee B(X,u)\}$ where $A(u)$ does not depend on X. Then the inclusion $\Phi(X) \subseteq X$ is equivalent to $\forall u \in U\ [(A(u) \Rightarrow X(u)) \vee (B(X,u) \Rightarrow X(u))]$ and the implications $A(u) \Rightarrow X(u)$ and $B(X,u) \Rightarrow X(u)$ are called induction *base* and *step* respectively, $B(X,u)$ is called *induction hypothesis*.

It is easy to see that μ, considered as an operation on monotone operators, is itself monotone, i.e. if $\Phi(X) \subseteq \Psi(X)$ for all $X \subseteq U$, then $\mu\Phi \subseteq \mu\Psi$. Indeed, by the induction principle it suffices to show $\Phi(\mu\Psi) \subseteq \mu\Psi$. But $\Phi(\mu\Psi) \subseteq \Psi(\mu\Psi) \subseteq \mu\Psi$. From the monotonicity of μ one can infer the following *strong induction* principle

$$\text{if } \Phi(X \cap \mu\Phi) \subseteq X \text{ then } \mu\Phi \subseteq X$$

For the proof of strong induction we assume $\Phi(X \cap \mu\Phi) \subseteq X$ which can be rewritten as $\Psi(X) \subseteq X$ where $\Psi(X) := \Phi(X \cap \mu\Phi)$. Clearly Ψ is a monotone operator. Thus $\mu\Psi \subseteq X$. Hence it is enough to show $\mu\Phi \subseteq \mu\Psi$. We prove this by induction on $\mu\Phi$. By the monotonicity result above we have the reverse inclusion $\mu\Psi \subseteq \mu\Phi$. Hence, $\Phi(\mu\Psi) = \Phi(\mu\Psi \cap \mu\Phi) = \Psi(\mu\Psi) \subseteq \mu\Psi$.

Example 1 (Natural Numbers) *Let $\mathbb{R}$ be the set of real numbers and define $\Phi : \mathscr{P}(\mathbb{R}) \to \mathscr{P}(\mathbb{R})$ by $\Phi(X) := \{0\} \cup \{y+1 | y \in X\}$. Then $\mu\Phi = \mathbb{N} = \{0,1,2,\dots\}$. The closure principle for $\mathbb{N}$ is equivalent to $\mathbb{N}(0) \wedge \forall x\, (\mathbb{N}(x) \to \mathbb{N}(x+1))$ while the induction principle for $\mathbb{N}$ is equivalent to $(X(0) \wedge \forall x\, (X(x) \to X(x+1))) \to \forall x\, (\mathbb{N}(x) \to X(x))$. The strong induction principle is similar, but with the step formula $\forall x\, (X(x) \to X(x+1))$ weakened to $\forall x \in \mathbb{N}\, (X(x) \to X(x+1))$.*

Now we turn our attention to coinduction which is dual to induction. For the same reason a monotone operator Φ has a least fixed point it has a greatest fixed point $\nu\Phi$. It is the largest Φ-coclosed subset of U where a set $X \subseteq U$ is called Φ-coclosed if $X \subseteq \Phi(X)$. Consequently, we have *coclosure*: $\nu\Phi \subseteq \Phi(\nu\Phi)$, and *coinduction*: if $X \subseteq \Phi(X)$ then $X \subseteq \nu\Phi$. With a similar argument as for μ one can show that ν is monotone and deduce from that a *strong coinduction* principle: if $X \subseteq \Phi(X \cup \nu\Phi)$ then $X \subseteq \nu\Phi$. We will see examples of coinduction in Sect. 3.

3 Cauchy and signed digit representations of real numbers

The primary objects of study in this paper are the real numbers in the compact interval

$$\mathbb{I} := [-1,1] = \{x \in \mathbb{R} \mid |x| \leq 1\}$$

Since real numbers are per-se abstract objects, it is not possible to compute with them directly: one has to refer to a specific representation. Two common representations of real numbers $x \in \mathbb{I}$ are

(1) Cauchy sequences $(q_n)_{n\in\mathbb{N}}$ where $q_n \in \mathbb{I}$ is rational with $|x - q_n| \leq 2^{-n}$ for all $n \in \mathbb{N}$,

(2) power series $x = \sum_{i\in\mathbb{N}} d_i 2^{-(i+1)}$ where $d_i \in \mathrm{SD} := \{0,1,-1\}$ (signed digits).

We consider the problem of producing a translation between the two representations that is formally proven to be correct. Note that in a traditional approach a quite complex formal system is required to deal with this problem: We need sorts for reals, rationals, natural numbers, digits and infinite sequences, and, in addition to the usual arithmetic operations, coercion functions between these sorts. Furthermore, the system must be able to express the recursive or iterative definitions of the higher-order functions translating between the two representations. On the other hand, the approach we are proposing and demonstrating here requires only one sort for real numbers with the usual first-order axioms for a real closed fields as well as the possibility to formalise inductive and coinductive definitions that were described in Sect. 2. In the following, all individual variables (denoted by lower case letters) range over real numbers.

We model the Cauchy representation (1) by the formula $A(x) := \forall n \in \mathbb{N} A_n(x)$ where $\mathbb{N}$ is defined as in Example 1 in Sect. 2 and

$$A_n(x) := \exists q \in \mathbb{Q}\,(|x| \leq 1 \wedge |x-q| \leq 2^{-n})$$

Here $\mathbb{Q}$ defines the rational numbers as a subset of the real numbers in the usual way with help of the predicate $\mathbb{N}$. E.g. $\mathbb{Q}(x) := \exists m,n,k \in \mathbb{N}\,(k > 0 \wedge xk = m-n)$ [1]. The formula $A(x)$ replaces the Cauchy representation in the sense that from a constructive proof of it one can extract a program implementing an infinite sequence $(q_n)_n \in \mathbb{N}$ satisfying (1). Note that this infinite sequence is not present in the formula $A(x)$. Details of how this extraction works in general will be given in Sect. 4.

We model the signed digit representation (2) by a coinductive definition, motivated by the observation that if $x = \sum_{i\in\mathbb{N}} d_i 2^{-(i+1)}$, then $|x - d_0/2| \leq 1/2$ and $2x - d_0 = \sum_{i\in\mathbb{N}} d_{i+1} 2^{-(i+1)}$. Therefore, we set $\mathbb{I}_d(x) := |x - d_0/2| \leq 1/2$ and define C as the largest set (of real numbers) such that

$$\mathrm{C}(x) \Rightarrow \exists d \in \mathrm{SD}(\mathbb{I}_d(x) \wedge \mathrm{C}(2x-d))$$

More formally $\mathrm{C} := \nu\mathscr{J}$ where the set operator $\mathscr{J} : \mathscr{P}(\mathbb{R}) \to \mathscr{P}(\mathbb{R})$ is defined by $\mathscr{J}(X) := \{x \mid \exists d \in \mathrm{SD}(\mathbb{I}_d(x) \wedge X(2x-d))\}$. Now, the program extracted from a proof of $\mathrm{C}(x)$ will be an infinite stream of digits $d_i \in \mathrm{SD}$ such that (2) holds.

In order to extract a program that translates between the representations (1) and (2) it will be sufficient to prove constructively the equivalence of the formulas $\forall n \in \mathbb{N} A_n(x)$ and $\mathrm{C}(x)$.

[1] The bounded quantifiers we used are just shorthands: $\forall x \in X\, B(x)$ stands for $\forall x\,(X(x) \to B(x))$ and $\exists x \in X\, B(x)$ stands for $\exists x\,(X(x) \wedge B(x))$.

4 Program extraction: theory

In this section we briefly describe the program extraction process in general, giving explanations of the main ideas priority over complete and formal definitions and correctness proofs.

4.1 The programming language

The programming language that will be the target of the extraction process is a λ-calculus with constructors and pattern matching and (ML-)polymorphic recursive types. We let α range over type variables.

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

In the definition of terms we let x range over term variables and C over constructors. It is always assumed that a constructor is applied to the correct number of arguments as determined by its arity. We will only use the constructors Nil (nullary), Left, Right (unary), Pair (binary), and $\text{In}_{\text{fix}\,\alpha\,.\,\rho}$ (unary) for every fixed point type $\text{fix}\,\alpha\,.\,\rho$.

$$\text{Term} \ni M, N ::= x \mid C(\vec{M}) \mid \text{case}\, M\, \text{of}\{C_1(\vec{x_1}) \to N_1\, ; \dots ; C_n(\vec{x_n}) \to N_n\} \mid \lambda x.M \mid (M\,N) \mid \text{rec}\, x\,.\, M$$

In a pattern $C_i(\vec{x_i})$ of a case-term all variables in $\vec{x_i}$ must be different. The typing relation $\Gamma \vdash M : \rho$ is defined inductively as follows.

$$\Gamma, x : \rho \vdash x : \rho \qquad \Gamma \vdash \text{Nil} : \mathbf{1} \qquad \frac{\Gamma, x : \rho \vdash M : \rho}{\Gamma \vdash \text{rec}\, x\,.\, M : \rho}$$

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

$$\frac{\Gamma \vdash M : \rho \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \text{Pair}(M,N) : \rho \times \sigma} \qquad \frac{\Gamma \vdash M : \rho \times \sigma \qquad \Gamma, x_1 : \rho, x_2 : \sigma \vdash K : \tau}{\Gamma \vdash \text{case}\, M\, \text{of}\{\text{Pair}(x_1, x_2) \to K\} : \tau}$$

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

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

Let $\rho = \rho(\vec{\alpha}) = \text{fix}\, \alpha \,.\, \rho_0(\alpha, \vec{\alpha})$:

$$\frac{\Gamma \vdash M : \rho_0(\rho(\vec{\sigma}), \vec{\sigma})}{\Gamma \vdash \text{In}_\rho(M) : \rho(\vec{\sigma})} \qquad \frac{\Gamma \vdash M : \rho(\vec{\sigma}) \qquad \Gamma, x : \rho_0(\rho(\vec{\sigma}), \vec{\sigma}) \vdash K : \tau}{\Gamma \vdash \text{case}\, M\, \text{of}\{\text{In}_\rho(x) \to K\} : \tau}$$

Equipped with a lazy semantics, which is described in [Ber09b], this system can be viewed almost literally as a fragment of Haskell. For example, a type $\text{fix}\,\alpha\,.\,\rho$ where $\rho = \rho(\alpha, \beta)$ has no free type variables other than α and β, can be modelled in Haskell by the data declaration `data FixRho beta = InFixRho (Rho FixRho beta)` provided `Rho alpha beta` models ρ. A term $\text{rec}\, x\,.\, M$ is modelled by `let {x = M} in x`. In fact, in Sect. 5 we will present the extracted programs as Haskell code (however, for the general considerations in the remainder of this Section the system above is more convenient). It is easy to see that this system is ML-polymorphic: if $\vdash M : \rho(\vec{\alpha})$, then $\vdash M : \rho(\vec{\sigma})$ for arbitrary types $\vec{\sigma}$.

4.2 The object language

The object language $\mathscr{L}$ which is used to formalise the proofs we want to extract programs from is a first-order language extended by predicate variables and the possibility to form least and greatest fixed points of strictly positive (and hence monotone) operators. *Terms*, $r, s, t \ldots$, are built from constants, first-order variables and function symbols as usual. *Formulas*, $A, B, C \ldots$, are $s = t$, $\mathscr{P}(\vec{t})$ where $\mathscr{P}$ is a predicate (predicates are defined 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 $\{\vec{x} \mid A\}$ where A is a formula and $\vec{x}$ is a vector of first-order variables, or an inductive predicate $\mu X . \mathscr{P}$, or a coinductive predicate $\nu X . \mathscr{P}$ where $\mathscr{P}$ is a predicate of the same arity as the predicate variable X and which is *strictly positive* in X, i.e. X does not occur free in any premise of a subformula of $\mathscr{P}$ which is an implication. The application, $\mathscr{P}(\vec{t})$, of a predicate $\mathscr{P}$ to a list of terms $\vec{t}$ is a primitive syntactic construct, except when $\mathscr{P}$ is a comprehension term, $\mathscr{P} = \{\vec{x} \mid A\}$, in which case $\mathscr{P}(\vec{t})$ stands for $A[\vec{t}/\vec{x}]$. We will frequently use common abbreviations such as $\mathscr{P} \subseteq \mathscr{Q} := \forall \vec{x}\, (\mathscr{P}(\vec{x}) \to \mathscr{Q}(\vec{x}))$, $\{\vec{t} \mid A\} := \{\vec{x} \mid \exists \vec{y}\, (\vec{x} = \vec{t} \wedge A)\}$ where $\vec{y}$ are the variables occurring free in A or t, $f(\vec{\mathscr{P}}) := \{f(\vec{x}) \mid \mathscr{P}_1(x_1) \wedge \ldots \wedge \mathscr{P}_n(x_n)\}$, and so on.

The *proof rules* for $\mathscr{L}$ are the usual ones for intuitionistic predicate calculus with equality [Hey56, Kle45, Tro73], plus the axiom schemes for inductive and coinductive predicates that were discussed in Sect. 2. In addition we allow any axioms expressible by non-computational formulas that hold in the intended model. Falsity can be defined as $\perp := \mu X . X$ where X is a 0-ary predicate variable (i.e. a propositional variable). From the induction axiom for $\perp$ it follows immediately $\perp \to A$ for every formula A.

For our examples it will be sufficient to have only one sort for real numbers in $\mathscr{L}$ (a many-sorted language would be possible as well) together with the usual algebraic equations and inequations for the operations on real numbers.

4.3 Realisability

The first step in program extraction is to assign to every $\mathscr{L}$-formula A a type $\tau(A)$, the type of potential realisers of A. If A contains neither predicate variables nor the logical connective $\vee$ (disjunction), then we call it *non-computational* (otherwise *computational*) and set $\tau(A) = \mathbf{1}$ (= `()` in Haskell). Otherwise, $\tau(\mathscr{P}(\vec{t})) = \tau(\mathscr{P})$ (for predicates $\mathscr{P}$ the type $\tau(\mathscr{P})$ is defined below), $\tau(A \wedge B) = \tau(A) \times \tau(B)$, $\tau(A \vee B) = \tau(A) + \tau(B)$, $\tau(A \to B) = \tau(A) \to \tau(B)$, $\tau(\forall x A) = \tau(\exists x A) = \tau(A)$, For predicates $\mathscr{P}$ we define $\tau(\mathscr{P})$ by $\tau(X) = \alpha_X$ where α_X is a type variable assigned in a one-to-one fashion to the predicate variable X, $\tau(\{\vec{x} \mid A\}) = \tau(A)$, and $\tau(\mu X . \mathscr{P}) = \tau(\nu X . \mathscr{P}) = \mathrm{fix}\, \alpha_X \,.\, \tau(\mathscr{P})$.

As one can see, the mapping τ wipes out all first-order content of a formula (first-order terms and quantifiers), hence the type $\tau(A)$ can be viewed as the “propositional skeleton” of the formula A. This is necessarily so, since the sorts in our first order language ($\mathbb{R}$ in our example in Sect. 3) have no counterpart in our programming language.

The next step is to define for every formula A and every program term M of type $\tau(A)$ what it means for M to *realise* A, formally $M\, \mathbf{r}\, A$. The latter is a formula in the language $\mathbf{r}(\mathscr{L})$ which is obtained by adding to $\mathscr{L}$ a sort for program terms and extending all other constructions concerning formulas and proofs mutatis mutandis. The $\mathbf{r}(\mathscr{L})$-formula $M\, \mathbf{r}\, A$ is in fact shorthand for

$\mathbf{r}(A)(M)$ where the $\mathbf{r}(\mathscr{L})$-predicate $\mathbf{r}(A)$ is defined by structural recursion on A, relative to a fixed one-to-one mapping from $\mathscr{L}$-predicate variables X to $\mathbf{r}(\mathscr{L})$-predicate variables $\tilde{X}$ with one extra argument place for program terms. 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 $\mathbf{r}(\mathscr{P})$ for every predicate $\mathscr{P}$, where $\mathbf{r}(\mathscr{P})$ has one extra argument place for program terms. If A is non-computational, then $\mathbf{r}(A) = \{() \mid A\}$. If $\mathscr{P}$ is non-computational, then $\mathbf{r}(\mathscr{P}) = \{((),\vec{x}) \mid \mathscr{P}(\vec{x})\}$. In all other cases: For a non-computational formula A we set $M\,\mathbf{r}\,A := M = \mathrm{Nil} \wedge A$, and

$$\begin{array}{rcl@{\qquad}rcl}
\mathbf{r}(\mathscr{P}(\vec{t})) & = & \{x \mid \mathbf{r}(\mathscr{P})(x,\vec{t})\} & \mathbf{r}(A \to B) & = & \{f \mid f(\mathbf{r}(A)) \subseteq \mathbf{r}(B)\} \\
\mathbf{r}(A \vee B) & = & \mathrm{inl}(\mathbf{r}(A)) \cup \mathrm{inl}(\mathbf{r}(B)) & \mathbf{r}(A \wedge B) & = & \mathrm{Pair}(\mathbf{r}(A),\mathbf{r}(B)) \\
\mathbf{r}(\exists y A) & = & \{x \mid \exists y\,(\mathbf{r}(A)(x))\} & \mathbf{r}(\forall y A) & = & \{x \mid \forall y\,(\mathbf{r}(A)(x))\} \\
\mathbf{r}(X) & = & \tilde{X} & \mathbf{r}(\{\vec{y} \mid A\}) & = & \{(x,\vec{y}) \mid \mathbf{r}(A)(x)\} \\
\mathbf{r}(\mu X.\mathscr{P}) & = & \mu\tilde{X}.\mathbf{r}(\mathscr{P}) & \mathbf{r}(\nu X.\mathscr{P}) & = & \nu\tilde{X}.\mathbf{r}(\mathscr{P})
\end{array}$$

We see that quantifiers and the quantified variable, although ignored by the program M and its type, of course do play a role in the definition of realisability, i.e. the *specification* of the program.

Finally, we sketch how to extract from a proof of a formula A a program term M realising A. Assuming the proof is given in a natural deduction system the extraction process is straightforward and follows in most cases the usual pattern of the Curry-Howard correspondence: Any non-computational axiom has the trivial program Nil as extracted program. The introduction and elimination rules for conjunction, disjunction and implication correspond to pairing, projection, injections into a disjoint sum, pattern matching, lambda-abstraction, and application, respectively. The $\forall$-introduction rule, $\forall$-elimination rule, and the $\exists$-introduction rule are ignored, i.e. the extracted program of the conclusion is identical to the one of the premise. The $\exists$-elimination rule corresponds to application, more precisely, if the proofs of the premises $\exists x A$ and $\forall x\,(A \to B)$ (where x is not free in B) have extracted programs $M : \tau(A)$ and $N : \tau(A) \to \tau(B)$, respectively, then the extracted program for the conclusion B is simply the application $M N : \tau(B)$. The extracted programs of closure, $\Phi(\mu\Phi) \subseteq \mu\Phi$, and induction, $(\Phi(X) \subseteq X) \Rightarrow \Phi(\mu\Phi) \subseteq X$, are $\mathbf{in}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda x.\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(x)$ and

$$\mathbf{it}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda s.\mathrm{rec}\,f\,.\,\lambda x.\mathrm{case}\,x\,\mathrm{of}\{\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(y) \to s(\mathbf{map}_{\alpha,\rho} f\,y)\}$$

where it is assumed that $\tau(\Phi(X)) = \rho(\alpha)$. The term $\mathbf{map}_{\alpha,\rho}$ has type $(\alpha \to \beta) \to \rho(\alpha) \to \rho(\beta)$ and can be defined by induction on $\rho(\alpha)$. For coclosure, $\nu\Phi \subseteq \Phi(\nu\Phi)$, and coinduction, $(X \subseteq \Phi(X)) \Rightarrow X \subseteq \Phi(\nu\Phi)$, the extracted programs are $\mathbf{out}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda x.\mathrm{case}\,x\,\mathrm{of}\{\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(y) \to y\}$ and

$$\mathbf{coit}_{\mathrm{fix}\,\alpha\,.\,\rho} := \lambda s.\mathrm{rec}\,f\,.\,\lambda x.\mathrm{In}_{\mathrm{fix}\,\alpha\,.\,\rho}(\mathbf{map}_{\alpha,\rho} f\,(s x))$$

We have the typings

$$\begin{array}{ll}
\vdash \mathbf{in}_{\mathrm{fix}\,\alpha\,.\,\rho} : \rho(\mathrm{fix}\,\alpha\,.\,\rho) \to \mathrm{fix}\,\alpha\,.\,\rho & \vdash \mathbf{it}_{\mathrm{fix}\,\alpha\,.\,\rho} : (\rho(\alpha) \to \alpha) \to (\mathrm{fix}\,\alpha\,.\,\rho) \to \alpha \\
\vdash \mathbf{out}_{\mathrm{fix}\,\alpha\,.\,\rho} : (\mathrm{fix}\,\alpha\,.\,\rho) \to \rho(\mathrm{fix}\,\alpha\,.\,\rho) & \vdash \mathbf{coit}_{\mathrm{fix}\,\alpha\,.\,\rho} : (\alpha \to \rho(\alpha)) \to \alpha \to \mathrm{fix}\,\alpha\,.\,\rho
\end{array}$$

The Soundness Theorem, stating that the program extracted from a proof does indeed realise the proven formula, is shown in [Ber09b]. Soundness Theorems for similar systems can be found in [Tat98] and Miranda-Perea [MP05].

5 Program extraction: applications

In this Section we apply the program extraction procedure described in Sect. 4 to a proof of the equivalence of the real number representations described in Sect. 3. Below we give a fairly detailed constructive proof of the equivalence which can be easily formalised in the object system described in Sect. 4.2, and from which we then extract the program.

Before we do that we discuss a simple example that demonstrates the fact that it is indeed crucial for program extraction that proofs are constructive and no axioms other than the axioms for inductive and coinductive definitions and true non-computational axioms are used. The formula $\forall x,y\,(x \leq y \vee y > x)$, although true in the real numbers, must not be used as an axiom because it is computational. We cannot prove it constructively either, because otherwise we could extract a (closed) program $M : \mathbf{Boole}$ realising it, where $\mathbf{Boole} := \mathbf{1}+\mathbf{1}$. This would mean that the formula, setting $\mathbf{t} := \text{Left}(\text{Nil})$ and $\mathbf{f} := \text{Right}(\text{Nil})$,

$$\forall x,y\,((M = \mathbf{t} \wedge x \leq y) \vee (M = \mathbf{f} \wedge y > x))$$

holds. Since M is closed, either $M = \mathbf{t}$ or $M = \mathbf{f}$. In the first case it would follow $\forall x,y\,(x \leq y)$ in the second case $\forall x,y\,(x > y)$ which are both false statements (similar unprovability results were among Kleene's and Kreisel's original motivations for studying realisability). Of course, we can prove constructively the relativised formula $\forall x,y \in \mathbb{N}\,(x \leq y \vee y > x)$ (by induction). The extracted program is a decision procedure for the ordering of natural numbers. Since $\mathbf{Nat} := \tau(\mathbb{N}(x)) = \text{fix}\,\alpha\,.\,\mathbf{1}+\alpha$ the program works with the unary representation of natural numbers, more precisely, its type is $\mathbf{Nat} \to \mathbf{Nat} \to \mathbf{Boole}$. Similarly, we can prove constructively $\forall x,y \in \mathbb{Q}\,(x \leq y \vee y > x)$ and extract a program deciding the ordering on rational numbers.

A formula which we can safely assume as an axiom is $\forall x\,(A(x) \to \mathbb{I}(x))$. Clearly, this formula is true and it has the trivial realiser $\lambda f.\text{Nil}$.

5.1 Proofs

The following Lemma takes care of some simple technicalities in the main proof.

Lemma 1 *(a)* $\text{SD} \subseteq \mathbb{Q}$.

(b) $\forall q \in \mathbb{Q}\,\exists d \in \text{SD}\,[q-1/4, q+1/4] \cap \mathbb{I} \subseteq \mathbb{I}_d$.

(c) $\forall p \in \mathbb{Q}\,\exists q \in \mathbb{Q} \cap \mathbb{I}\,\forall x \in \mathbb{I}\,|x-q| \leq |x-p|$.

Proof. Part (a) is obvious. For part (b) we do a case analysis on the position of q. For $q < -\frac{1}{4}$ choose $d = -1$, for $-\frac{1}{4} \leq q \leq \frac{1}{4}$ choose $d = 0$ and for $\frac{1}{4} < q$ choose $d = 1$. It is easy to see that in all cases d is as required. For part (c) let $p \in \mathbb{Q}$ be given. Depending on whether $p < -1$ or $|p| \leq 1$ or $p > 1$ we set $q := -1$ or $q := p$ or $q := 1$. The required property obviously holds. $\square$

Proposition 1 $\forall x\,(\text{C}(x) \Leftrightarrow A(x))$.

Proof. For the implication from left to right we prove $\forall n \in \mathbb{N}(\text{C}(x) \Rightarrow A_n(x))$ by induction on $\mathbb{N}$. Base $n = 0$: If $\text{C}(x)$, then clearly $x \in \mathbb{I}$, i.e. $|x| \leq 2^0$. Hence we can take $q := 0$ to satisfy $A_0(x)$.

Step: The induction hypothesis is $\forall x(\mathrm{C}(x) \Rightarrow \exists q|x-q| \leq 2^{-n})$. We have to show $\forall x(\mathrm{C}(x) \Rightarrow \exists q|x-q| \leq 2^{-(n+1)})$. Assume $\mathrm{C}(x)$. By the coclosure principle for C we have $\mathbb{I}_d(x)$ and $\mathrm{C}(2x-d)$ for some $d \in \mathrm{SD}$. Set $x' := 2x-d$. By induction hypothesis, $|x'-q| \leq 2^{-n}$ for some q. Hence $\frac{|x'-q|}{2} \leq \frac{2^{-(n+1)}}{2}$, i.e. $\frac{|2x-d-q|}{2} \leq 2^{-(n+1)}$, i.e. $|x-\frac{d}{2}-\frac{q}{2}| \leq 2^{-(n+1)}$, i.e. $|x-\frac{d+q}{2}| \leq 2^{-(n+1)}$, so we may take $q' := \frac{d+q}{2}$ using Lemma 1 (a).

For the implication from right to left we need to show $A \subseteq \mathrm{C}$. By applying coinduction it is sufficient to show $A \subseteq \mathscr{J}(A)$, i.e.

$$\forall x(A(x)) \Rightarrow \exists d \in \mathrm{SD}\,(\mathbb{I}_d(x) \wedge A(2x-d)))$$

Assume $A(x)$. Then $\mathbb{I}(x)$ by the axiom discussed above. We have to find $d \in \mathrm{SD}$ such that

$$\mathbb{I}_d(x) \wedge \forall n \in \mathbb{N}\, \exists q \in \mathbb{Q} \cap \mathbb{I}\, |(2x-d)-q| \leq 2^{-n}$$

Using the assumption $A(x)$ with $n=2$, we obtain a $q \in \mathbb{Q} \cap \mathbb{I}$ such that $|x-q| \leq \frac{1}{4}$. According to Lemma 1 (b) there is some $d \in \mathrm{SD}$ such that $[q-1/4, q+1/4] \cap \mathbb{I} \subseteq \mathbb{I}_d$. Now let $n \in \mathbb{N}$. We have to find $q \in \mathbb{Q} \cap \mathbb{I}$ such that $|(2x-d)-q| \leq 2^{-n}$. We use the assumption $A(x)$ with $n+1$ and obtain $q' \in \mathbb{Q} \cap \mathbb{I}$ such that $|x-q'| \leq 2^{-(n+1)}$. Since $x \in \mathbb{I}_d$ and because of Lemma 1 (c) we may assume without loss of generality that $q' \in \mathbb{I}_d$. Hence $q := 2q'-d \in \mathbb{I}$ and $|(2x-d)-q| = 2|x-q'| \leq 2^{-n}$. □

5.2 Programs

We begin our program extraction by declaring the types of the predicates SD and $\mathbb{Q}$. We do not bother deriving them pedantically following Sect. 4, but define them in Haskell in a convenient way. For $\mathbb{Q}$ it is most convenient to use the build-in type of exact rational numbers.

```
data SD = N | Z | P deriving Show
type Rat = Rational
```

Next we extract programs from Lemma 1. Again, because the proofs are so simple, we do the extraction in an intuitive and non-pedantic way using build-in operations on the rational numbers. We also apply, here and in the following, some simplifications to types and extracted programs. For example, if B is computational, but A is not, we set $\tau(A \wedge B) = \tau(A \to B) = \tau(B)$ (instead of $\mathbf{1} \times \tau(B)$ and $\mathbf{1} \to \tau(B)$) and adjust realisability and program extraction accordingly.

```
lema :: SD -> Rat
lema = \d-> case d of {N -> -1; Z -> 0; P -> 1}

lemb :: Rat -> SD
lemb q | q < -1/4   = N
       | q > 1/4    = P
       | otherwise = Z

lemc :: Rat -> Rat
lemc q | q < -1     = -1
       | q > 1      = 1
       | otherwise = q
```

Now we declare the data type $\tau(\mathbb{N})$:

```
type NAT alpha = Either () alpha
data Nat = ConsNat (NAT Nat)
```

For later use we define the successor operation and numerals on `Nat`:

```
suc :: Nat -> Nat
suc n = ConsNat (Right n)

num :: Int -> Nat
num n = if n <= 0 then ConsNat (Left ()) else suc (num (n-1))
```

The program `suc` can be extracted from a proof that $\mathbb{N}$ is closed under successor. Now we move on to a more interesting part, the extracted programs of the axioms for the inductive predicate $\mathbb{N}$ and the coinductive predicate C. Below, `Sds` stands for "signed digit stream".

```
mapNAT :: (alpha -> beta) -> NAT alpha -> NAT beta
mapNAT = \f-> \z->
  case z of {Left u -> Left () ; Right x -> Right (f x)}

itNat :: (NAT alpha -> alpha) -> Nat -> alpha
itNat = \step->
 let {f = \n-> case n of {ConsNat z -> step (mapNAT f z)}} in f

type SDS alpha = (SD,alpha)
data Sds = ConsSds (SDS Sds)

mapSDS :: (alpha -> beta) -> SDS alpha -> SDS beta
mapSDS = \f-> \z-> case z of {(d,x) -> (d,f x)}

coitSds :: (alpha -> SDS alpha) -> alpha -> Sds
coitSds = \f-> \x-> ConsSds (mapSDS (coitSds f) (f x))
```

The type of the formula $A(x)$ is

```
type Approx = Nat -> Rat
```

Finally, the programs extracted from Prop. 1 are

```
proplr :: Sds -> Approx
proplr = \a-> \n-> itNat proplrStep n a

proplrStep :: NAT (Sds -> Rat) -> Sds -> Rat
proplrStep = \z-> \a-> case z of
 {Left u -> 0; Right ih -> case a of
                {ConsSds (d,a') -> (lema d + ih a')/2}}
```

```
proprl :: Approx -> Sds
proprl = coitSds proprlStep

proprlStep :: Approx -> SDS Approx
proprlStep = \f->
  let d = lemb (f (num 2))
  in (d, \n-> lemc (2 * f (suc n) - lema d))
```

5.3 Results

As an example, we compute the signed digit representation of $\frac{1}{\sqrt{e}} \in \mathbb{I}$. Since

$$\frac{1}{\sqrt{e}} = e^{-\frac{1}{2}} = \sum_{i=0}^{\infty} \frac{(-\frac{1}{2})^i}{i!}$$

and the series converges at an exponential rate we define

$$e_n := \sum_{i=0}^{n} \frac{(-\frac{1}{2})^i}{i!} \in \mathbb{Q}$$

and obtain an infinite sequence $e = (e_n)_{n\in\mathbb{N}}$ realising the formula $A(\frac{1}{\sqrt{e}})$. Feeding e into the program `proplr` we obtain a realiser of $\mathrm{C}(\frac{1}{\sqrt{e}})$, i.e. a signed digit representation of $\frac{1}{\sqrt{e}}$. In order to display the stream in the usual Haskell format we coerce `Sds` into `[SD]`.

```
e :: Approx
e n = sum [((-1/2)^i) / (fromIntegral (product [1..i]))
           | i <- [0..(fromIntegral n')]]
  where n' = nat2Int n

nat2Int :: Nat -> Int
nat2Int (ConsNat (Left ())) = 0
nat2Int (ConsNat (Right n)) = 1 + nat2Int n

sds2Stream :: Sds -> [SD]
sds2Stream (ConsSds (d,a)) = d : sds2Stream a

sde :: [SD]
sde = sds2Stream (proprl e)

*Main> take 100 sde
[P,Z,P,Z,N,P,Z,N,P,N,Z,Z,P,N,P,Z,N,Z,P,N,P,Z,Z,Z,Z,Z,N,Z,Z,P,
 Z,N,P,Z,Z,Z,Z,N,Z,P,N,P,Z,N,Z,Z,Z,Z,P,N,Z,P,Z,Z,Z,Z,Z,Z,Z,N,
 Z,P,Z,Z,Z,N,P,N,P,Z,N,Z,P,Z,Z,N,P,N,P,Z,N,P,N,Z,Z,P,Z,N,P,N,
 P,N,P,N,P,N,Z,P,Z,N]
```

We can check that this stream is correct by computing the corresponding floating point approximation and comparing it with the result obtained by floating point arithmetic (ironically, relying on the correctness of the latter).

```
list2Double :: [SD] -> Double
list2Double  = foldr av 0   where
    av d x = ((fromRational (lema d))+x)/2

*Main> exp (-0.5)
0.6065306597126334
*Main> list2Double (take 100 sde)
0.6065306597126334
```

6 Conclusion

We presented a method for extracting certified programs from constructive proofs. The method is based on a variant of realizability that strictly separates the (abstract) mathematical model from the data types the extracted program is dealing with. The latter are determined completely 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.

We used the problem of translating between different representations of real numbers as an example to illustrate the method in general, and to show the use of inductive and coinductive definitions in program extraction.

Currently, we are working on an extension of this work to the situation where not only real numbers, but also real functions are coinductively represented and where the underlying domain is extended to arbitrary separable metric spaces or even more general structures [Ber09b, Ber09a].

An important aspect of the program extraction method is the ability to import existing certified software. This is partly accounted for by the possibility to add statements as axioms for which programs provably realising them are given. But as least as important is the possibility to include existing trusted data structures (large integers exact rationals, etc.) together with efficient (and certified) implementations of basic operations. It is possible to extend our approach in this respect, for example, using a general theory of realisability based on equilogical spaces and assemblies [BBS04].

The method of program extraction including inductive definitions is implemented for example in the Minlog system [BBS$^+$98]. Carrying out in Minlog case studies involving coinduction as presented here requires an appropriate extension of the system. This is work in progress.

Bibliography

[Agd] Agda. http://wiki.portal.chalmers.se/agda/.

[BB85] E. Bishop, D. Bridges. *Constructive Analysis*. Grundlehren der mathematischen Wissenschaften 279. Springer, Berlin, Heidelberg, NewYork, Tokyo, 1985.

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

[BBS04] A. Bauer, L. Birkedal, D. S. Scott. Equilogical spaces. *Theor. Comput. Sci.* 315(1):35–59, 2004. doi:http://dx.doi.org/10.1016/j.tcs.2003.11.012

[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 Grädel and Kahle (eds.), *Computer Science Logic*. LNCS 5771, pp. 132–146. Springer, 2009.

[Ber09b] U. Berger. Realisability and Adequacy for (Co)induction. In Bauer et al. (eds.), *6th Int'l Conf. on Computability and Complexity in Analysis*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, 2009. http://drops.dagstuhl.de/opus/volltexte/2009/2258

[Bez07] M. Bezem. The Software Model Checker Blast. *Journal on Software Tools for Technology Transfer* 9(5-6):505–525, 2007.

[BH08] U. Berger, T. Hou. Coinduction for Exact Real Number Computation. *Theory of Computing Systems* 43:394–409, 2008. doi:DOI: 10.1007/s00224-007-9017-6

[BRS$^+$00] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, A. Programmiermethodik. Formal system development with KIV. In *Fundamental Approaches to Software Engineering, number 1783 in LNCS*. Pp. 363–366. Springer, 2000.

[BS07] J. Bradfield, C. Stirling. Modal mu-calculi. In Blackburn et al. (eds.), *Handbook of Modal Logic*. Studies in Logic and Practical Reasoning 3, pp. 721–756. Elsevier, 2007.

[CD06] A. Ciaffaglione, P. Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theor. Comput. Sci.* 351:39–51, 2006.

[Con86] R. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice–Hall, New Jersey, 1986.

[Coq] The Coq Proof Assistant. http://coq.inria.fr/.

[Dij75] E. Dijkstra. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Comm. ACM* 18:453–457, 1975.

[Dij97] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[DJ78] B. Dines, C. Jones. *The Vienna Development Method: The Meta-Language*. LNCS 61. Springer, Berlin, Heidelberg, New York, 1978.

[EH02] A. Edalat, R. Heckmann. Computing with Real Numbers: I. The LFT Approach to Real Number Computation; II. A Domain Framework for Computational Geometry. In Barthe et al. (eds.), *Applied Semantics - Lecture Notes from the International Summer School, Caminha, Portugal*. Pp. 193–267. Springer, 2002.

[Flo67] R. Floyd. Assigning meaning to Programs. In *Mathematical Aspects of Computer Science*. Pp. 19–32. American Mathematical Society, 1967.

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

[Gri81] D. Gries. *The Science of Programming*. Springer, 1981.

[vH67] J. van Heijenoort (ed.). *From Frege to Gödel. A Source Book in Mathematical Logic 1879–1931*. Harvard University Press, Cambridge, MA., 1967. Reprinted 1970.

[Hey56] A. Heyting. *Intuitionism: An Introduction*. North-Holland, Amsterdam. Third Revised Edition 1971, 1956.

[Hoa97] T. Hoare. An Axiomatic Basis for Computer Programming. *Comm. ACM* 12:567–580, 1997.

[HW03] J. Hughes, M. Warnier. The Coinductive Approach to Verifying Cryptographic Protocols. In Wirsing et al. (eds.), *Recent Trends in Algebraic Development Techniques*. LNCS 2755, pp. 268–283. Springer, Berlin, 2003.

[JR97] B. Jacobs, J. Rutten. A Tutorial on (Co)Algebras and (Co)Induction. *EATCS Bulletin* 62:222–259, 1997.

[Kle45] S. C. Kleene. On the interpretation of intuitionistic number theory. *Jour. Symb. Logic* 10:109–124, 1945.

[KMM00] M. Kaufmann, P. Manolios, J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

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

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[ME07] J. R. Marcial-Romero, M. H. Escardo. Semantics of a sequential language for exact real-number computation. *Theor. Comput. Sci.* 379(1-2):120–141, 2007.

[Mil80] R. Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[MP05] F. Miranda-Perea. Realizability for Monotone Clausular (Co)Inductive Definitions. *Electr. Notes in Theoret. Comput. Sci.* 123:179–193, 2005.

[Mos99] L. S. Moss. Coalgebraic Logic. *Annals of Pure and Applied Logic* 96, 1999.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[ORSS98] S. Owre, J. Rushby, N. Shankar, D. Stringer-Calvert. PVS: An Experience Report. In Hutter et al. (eds.), *Applied Formal Methods—FM-Trends 98*. Lecture Notes in Computer Science 1641, pp. 338–345. Springer-Verlag, Boppard, Germany, oct 1998.
http://www.csl.sri.com/papers/fmtrends98/

[Pnu77] A. Pnueli. The Temporal Logic of Programs. In *In Proc. 18th IEEE Symposium on Foundation of Computer Science*. LNCS 902, pp. 350–364. IEEE, 1977.

[Rut00] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science* 249(1):3–80, 2000.

[Sch09] H. Schwichtenberg. Realizability interpretation of proofs in constructive analysis. Theory Comput. Sys., to appear, 2009.

[Tat98] M. Tatsuta. Realizability of Monotone Coinductive Definitions and Its Application to Program Synthesis. In Parikh (ed.), *Mathematics of Program Construction*. Lecture Notes in Mathematics 1422, pp. 338–364. Springer, 1998.

[Tro73] A. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. Lecture Notes in Mathematics 344. Springer, 1973.