Axiomatic Reals and Certified Efficient Exact Real Computation [⋆]

Michal Konečný[1], Sewon Park[2], and Holger Thies[3]

[1] Aston University, Birmingham, UK m.konecny@aston.ac.uk
[2] KAIST, Daejeon, Korea swelite@kaist.ac.kr
[3] Kyoto University, Kyoto, Japan thies.holger.5c@kyoto-u.ac.jp

Abstract. We introduce a new axiomatization of the constructive real numbers in a dependent type theory. Our main motivation is to provide a sound and simple to use backend for verifying algorithms for exact real number computation and the extraction of efficient certified programs from our proofs. We prove the soundness of our formalization with regards to the standard realizability interpretation from computable analysis. We further show how to relate our theory to a classical formalization of the reals to allow certain non-computational parts of correctness proofs to be non-constructive. We demonstrate the feasibility of our theory by implementing it in the Coq proof assistant and present several natural examples. From the examples we can automatically extract Haskell programs that use the exact real computation framework AERN for efficiently performing exact operations on real numbers. In experiments, the extracted programs behave similarly to native implementations in AERN in terms of running time.

Keywords: Constructive Real Numbers · Formal Proofs · Exact Real Number Computation · Program Extraction.

1 Introduction

Verifying the correctness of software is becoming increasingly important, in particular in safety critical application domains. Often, such programs need to interact in some way with the outside, physical world requiring numerical calculations over the real numbers and other uncountable mathematical entities. While proof assistants and formal methods are becoming more mature and are increasingly used in practical applications, verification of numerical programs remains extremely challenging [2]. One difficulty arises from the fact that in practice real

[⋆] Holger Thies is supported by JSPS KAKENHI Grant Number JP20K19744. Sewon Park is supported by the National Research Foundation of Korea (NRF) grants funded by the Korea government (No. NRF-2016K1A3A7A03950702, NRF-2017R1E1A1A03071032 (MSIT) & No. NRF-2017R1D1A1B05031658 (MOE)).
This project has received funding from the EU's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 731143.

numbers are commonly replaced by floating-point approximations, introducing rounding errors and uncertainties that pose additional problems for verification.

While there is active ongoing work on the verification of floating-point arithmetic [4, 15], we here consider a different approach known as *exact real computation.* In exact real computation, real numbers are basic entities that allow exact manipulation without rounding errors. Programs can output finite approximations up to any desired absolute precision. This is often realized by adding a datatype for reals and arithmetic operations on them as primitives in programming languages. Several implementations exist demonstrating the feasibility of the approach [16, 10, 1]. Although less efficient than optimized hardware-based floating-point calculations, implementations in exact real computation are by design more reliable than the former and are thus well-suited for situations where correctness is of high importance. Further, for efficient implementations there is often only a small overhead. However, subtilities of the semantics such as multivaluedness can still make writing correct programs difficult and stronger guarantees of correctness are highly desirable. One of the strongest such guarantees is a computer verified correctness proof e.g. in a proof assistant which however requires a sound model of the semantics. This poses some theoretical challenges as operations such as partial comparisons and multivalued branching are common in exact real computation and need to be computable [18].

Software packages for exact real computation often build on the theoretical framework of computable analysis and the theory of representations [25, 13]. In previous work [11] two of the authors of the present paper worked on verified exact real computation using the Incone library [23], which aims to directly formulate the model of computable analysis in Coq. Incone requires to define computational *realizers*, i.e. functions that work on low-level encodings of the reals. This low-level approach allows to directly execute the realizers inside of the proof assistant or to extract to simple Haskell or Ocaml code, but is less elegant and more labour-intensive than working with a high-level abstract implementation of a real number type. Further, there are already several sophisticated implementations of exact real computation in modern programming languages and those implementations have been tested on numerous examples and proven to be efficient and reliable. Instead of reimplementing and verifying basic real number operations, a more practical approach is to trust the implementation of a core of simple real number operations and to verify programs using those operations under the assumption that they are correctly implemented. This approach also provides a certain amount of independence of the concrete implementation of real numbers and thus allows to easily switch the underlying framework.

In the present work we therefore follow a different approach. We define a new constructive axiomatization that models the real numbers in a conceptually similar way as some mature implementations of exact real computation. We formally define our theory on top of a simple type theory inspired by the one used in Coq and prove its soundness with respect to the realizability interpretation used in computable analysis. We also give a theoretical foundation of relating proofs written over a classical theory of real numbers with our real numbers.

Of course, there are already several formalizations of real numbers and real analysis in most proof assistants (see e.g. [3] for an overview) and also large constructive implementations such as the C-CoRn library [6]. However, our axiomatization very closely models ideas used in computable analysis and practical implementations of exact real computation such as multivalued operations. We therefore think that it can be more appealing to people working in this area.

Our approach further allows to easily map the constructive real type, and its axiomatically defined basic operations such as arithmetic or limits, to corresponding types and operations in an exact real computation framework. Concretely, utilising this mapping and program extraction techniques, we obtain certified programs over an ERC implementation from correctness proofs.

We implemented the theory in the Coq proof assistant and extracted Haskell programs from our proofs using Coq code extraction. In the extracted programs, primitive operations on the reals are mapped to operations in the exact real computation framework AERN [10] which is written and maintained by one of the authors. Our first examples show that the extracted programs perform efficiently, having only a small overhead compared to hand-written implementations.

2 Computable Analysis and Exact Real Computation

In this section, we recap some essential concepts and limitations of computable analysis and exact real computation in order to justify our choice of axioms.

To compute over uncountable mathematical structures such as real numbers exactly, computable analysis takes *assemblies* over Kleene's second algebra (assemblies for short) as the basic data type [8, 22].[4] An assembly is a pair of a set A and a relation $\Vdash \subseteq \mathbb{N}^{\mathbb{N}} \times A$, which is surjective in that $\forall x \in A.\ \exists \varphi.\ \varphi \Vdash x$. We call $\varphi \in \mathbb{N}^{\mathbb{N}}$ a realizer of an abstract entity $x \in A$ if $\varphi \Vdash x$ holds. Given two assemblies of A and B, a function $f : A \to B$ is said to be computable if there is a computable function $\tau :\subseteq \mathbb{N}^{\mathbb{N}} \to \mathbb{N}^{\mathbb{N}}$ that tracks f, i.e. for any $x \in A$ and its realizer φ, $\tau(\varphi)$ is a realizer of $f(x)$.

For real numbers, there is a unique assembly (up to isomorphism in the category of assemblies $\mathsf{Asm}(\mathcal{K}_2)$) that makes the model-theoretic structure [7] of real numbers computable: (1) $0, 1 \in \mathbb{R}$ are computable, (2) field arithmetic is computable, (3) the order relation $<$ that is undefined at $\{(x, x) \mid x \in \mathbb{R}\}$ is computable, and (4) the limit operation defined at rapidly converging sequences is computable. An example is the Cauchy reals where φ is a realizer of $x \in \mathbb{R}$ if and only if φ encodes a sequence of rationals converging rapidly towards x. An assembly of reals satisfying the above computability conditions is called *effective*.

An inevitable side-effect of this approach is partiality. Whichever realizability relation for reals we take, comparisons of real numbers are only partially computable [25, Theorem 4.1.16]. Let *Kleenean* $\mathbf{K}$ be the assembly of $\{\mathit{ff}, \mathit{tt}, \bot\}$ where an infinite sequence of zeros realizes $\bot$, an infinite sequence that starts

[4] Assemblies are generalizations of represented sets [13, 25] which are exactly the assemblies where the surjective relations are required to be partial surjective functions. The terminology *multi-representation* [20] may be more familiar to some readers.

with 1 after a finite prefix of zeros realizes ff, and an infinite sequence that starts with 2 after a finite prefix of zeros realizes tt (see e.g. [11, Example 3]). The assembly $\mathbf{K}$ can be seen as a generalization of the Booleans by adding an explicit state of divergence $\perp$. Comparison in any effective assembly of reals $\mathbf{R}$ is computable as a function $x < y = tt$ if $x < y$, ff if $y < x$, and $\perp$ if $x = y$.

Comparisons being partial, multivaluedness becomes an essential notion [14]. For two assemblies of A and B, a multivalued function $f : A \rightrightarrows B$, which is basically a nonempty set-valued function, is computable if there is a computable function that takes a realizer φ of $x \in A$ and computes a realizer of any $y \in f(x)$. An example is the multivalued soft comparison [5]:

$$x <_k y = \{tt \mid x < y + 2^k\} \cup \{\mathit{ff} \mid y < x + 2^k\}.$$

The above total multivalued function approximates the order relation. It is tracked by evaluating two partial comparisons $x < y + 2^k$ and $y < x + 2^k$ in parallel, returning tt if $x < y + 2^k = tt$, and ff if $y < x + 2^k = tt$. It is nondeterministic in the sense that for the same x and y but with different realizers, which of the tests terminates first may vary. Exact real number computation software such as [16, 10] further offer operators like $\mathbf{select} :\subseteq \mathbf{K} \times \mathbf{K} \rightrightarrows \mathbf{K}$ such that $\mathbf{select}(k_1, k_2) \ni tt$ iff $k_1 = tt$ and $\mathbf{select}(k_1, k_2) \ni \mathit{ff}$ iff $k_2 = tt$ as a primitive operation for generating multivaluedness.

3 Axiomatization

In this section we give an overview of the formalization and the axioms we introduce. For space reasons we omit most axioms in the main part but provide a complete list in Appendix A. For those axioms that we do introduce here, we also reference the corresponding entry from the appendix.

Our theory is formalized in a type theory similar to the one of Coq. More precisely, we work with a dependent type theory with basic types $0, 1, 2, \mathsf{N}, \mathsf{Z}$, and a universe of classical propositions. That is, we have an impredicative à la Russel universe Prop, closed under $\to, \wedge, \vee, \tilde{\exists}, \Pi$, where the law of excluded middle $\Pi(P : \mathsf{Prop}).\ P \vee \neg P$ holds (Axiom TT1) [17]. We assume that the identity types belong to Prop. Opposed to Prop, Type is an à la Russel universe of types (with an implicit type level) with type constructors $\to, \times, +, \Sigma, \Pi$. We further suppose propositional extensionality in Prop (Axiom TT2) and function extensionality (Axiom TT3). Based on this setting, we propose an axiomatization for the assemblies $\mathbf{K}, \mathbf{R}$ and computable multivalued functions from Section 2.

3.1 Kleenean and Multivalued Lifting

First, we assume that there is a type $\mathsf{K} : \mathsf{Type}$ of Kleeneans (Axiom K1) and that there are two *distinct* elements $\mathsf{true} : \mathsf{K}$ and $\mathsf{false} : \mathsf{K}$ (Axioms K2, K3 and K4). Let us define the abbreviation $\lceil t \rceil : \mathsf{Prop} :\equiv t = \mathsf{true}$. In many cases, we do not work directly with Kleenean. Instead, we call a proposition $P : \mathsf{Prop}$ semi-decidable (in its free variables) if there is a Kleenean t that identifies P:

$$\mathsf{semiDec}(P) :\equiv \Sigma(t : \mathsf{K}).\ P = \lceil t \rceil$$

Multivalued computations are axiomatized by a monad M (Axioms M1–M9) such that a mapping $f : A \to B$ expresses a singlevalued function and $f : A \to \mathsf{M}\ B$ expresses a multivalued function. We assume the monad structure: (1) there is a type constructor $\mathsf{M} : \mathsf{Type} \to \mathsf{Type}$, (2) there is a unit $\mathsf{unitM} : \Pi(A : \mathsf{Type}).\ A \to \mathsf{M}\ A$, (3) a multiplication $\mathsf{multM} : \Pi(A : \mathsf{Type}).\ \mathsf{M}(\mathsf{M}\ A) \to \mathsf{M}\ A$, (4) a function lifting $\mathsf{liftM} : \Pi(A, B : \mathsf{Type}).\ (A \to B) \to \mathsf{M}\ A \to \mathsf{M}\ B$, (5) and the corresponding coherence conditions.

Intuitively, the monad can be understood as the nonempty power-set monad. In this sense, we assume that there is a mapping

$$\mathsf{elimM} : \Pi(A : \mathsf{Type}).\ (\Pi(x, y : A).\ x = y) \to (\mathsf{M}\ A) \to A$$

which is an inverse of unitM (Axioms M10-M11).

For any sequence of types $P : \mathsf{N} \to \mathsf{Type}$, we assume that the map

$$\lambda(X : \mathsf{M}(\Pi(x : \mathsf{N}).\ P\ x)).\ \lambda(n : \mathsf{N}).\ \mathsf{liftM}\big(\lambda(f : \Pi(x : \mathsf{N}).\ P\ x).\ f\ n\big)\ X$$

which is of type $\mathsf{M}(\Pi(x : \mathsf{N}).\ P\ x) \to \Pi(x : \mathsf{N}).\ \mathsf{M}(P\ x)$ admits a section (Axioms M12-M13):

$$\omega\mathsf{lift}\ P : (\Pi(x : \mathsf{N}).\ \mathsf{M}(P\ x)) \to \mathsf{M}(\Pi(x : \mathsf{N}).\ P\ x)\ .$$

Intuitively, given a set of sequences S, the first map transforms it to a sequence of sets $(n \mapsto \bigcup_{f \in S}\{f(n)\})$. And, $\omega\mathsf{lift}$ is its section which transforms a sequence of sets f to a set of sequences $\{g \mid \forall n.\ g(n) \in f(n)\}$. This operation enables, for example, to interchange multivalued sequences of real numbers with sequences of multivalued real numbers.

The most important axiom we assume is multivalued branching (Axiom M14):

$$\mathsf{select} : \Pi(x, y : \mathsf{K}).\ (\lceil x \rceil \vee \lceil y \rceil) \to \mathsf{M}\big(\lceil x \rceil + \lceil y \rceil\big)\ .$$

The above axiom yields the following, which we use more frequently:

$$\mathsf{choose} : \Pi(P, Q : \mathsf{Prop}).\ P \vee Q \to \mathsf{semiDec}(P) \to \mathsf{semiDec}(Q) \to \mathsf{M}\big(P + Q\big)\ .$$

Namely, given two semi-decidable propositions and at least one of them holds classically, we can nondeterministically decide if P holds or Q holds.

For any two types A, B, we write $f : A \rightrightarrows B$ to denote $f : A \to \mathsf{M}\ B$ and $\overline{\Sigma}(x : A).\ P(x)$ for $\mathsf{M}\ \Sigma(x : A).\ P(x)$ (multivalued functions and existences).

Example 1. For any proposition P, suppose both $\mathsf{semiDec}(P)$ and $\mathsf{semiDec}(\neg P)$ hold. As $P \vee \neg P$ holds by the classical law of excluded middle, we have $\mathsf{M}(P + \neg P)$ by applying choose. As it is provable that $P + \neg P$ is subsingleton, using elimM, we have $P + \neg P$, the decidability of the proposition P.

3.2 Real Numbers

We assume real numbers by declaring that there is a type $\mathsf{R} : \mathsf{Type}$ for real numbers (Axiom R1) and axiomatizing its model-theoretic structure. There are distinct constants $0 : \mathsf{R}$ and $1 : \mathsf{R}$, (infix) binary operators $+, \times : \mathsf{R} \to \mathsf{R} \to \mathsf{R}$, a unary operator $- : \mathsf{R} \to \mathsf{R}$, a term $/ : \Pi(x : \mathsf{R}).\ x \neq 0 \to \mathsf{R}$, and a (infix) binary predicate $<: \mathsf{R} \to \mathsf{R} \to \mathsf{Prop}$ (Axioms R2-R8). We assume the properties of the structure classically in a safe way that does not damage constructivity (Axioms R11-R27). For example, trichotomy (Axiom R22) is assumed as a term of type

$$\Pi(x, y : \mathsf{R}).\ x < y \vee x = y \vee y < x\ .$$

However, an inhabitant of the type $\Pi(x, y : \mathsf{R}).\ (x < y) + (x = y) + (y < x)$ is not posed anywhere.

In addition to the axioms in Prop, we assume $\Pi(x, y : \mathsf{R}).\ \mathsf{semiDec}(x < y)$ (Axiom R9). Namely, for any two real numbers its order, as a classical proposition, is semi-decidable.

Example 2. Using the classical trichotomy, we can construct a term of type

$$\Pi(x, y, \epsilon : \mathsf{R}).\ 0 < \epsilon \to x < y + \epsilon \vee y < x + \epsilon.$$

Since the inequalities are semi-decidable, using choose, the multivalued version of the approximate splitting lemma [21, Lemma 1.23]

$$\mathsf{mSplit} : \Pi(x, y, \epsilon : \mathsf{R}).\ 0 < \epsilon \rightrightarrows \big((x < y + \epsilon) + (y < x + \epsilon)\big)$$

is obtainable, which roughly says, for any real numbers x, y, ϵ, when ϵ is positive, we can nondeterminstically decide if $x < y + \epsilon$ or $y < x + \epsilon$.

The set of classical axioms living in Prop includes the completeness of the set of real numbers (Axiom R27). For its constructive counterpart (Axiom R10), for any predicate $P : \mathsf{R} \to \mathsf{Prop}$ such that $p : \tilde{\exists}!(z : \mathsf{R}).\ P\ z$ holds, we assume

$$\mathsf{lim}\ P\ p : \Pi(n : \mathsf{N}).\ \Sigma(e : \mathsf{R}).(\tilde{\exists}(a : \mathsf{R}).\ P\ a \wedge -2^{-n} < e - a < 2^{-n})) \to \Sigma(a : \mathsf{R}).\ P\ a.$$

Here, for any $n : \mathsf{N}$, $2^{-n} : \mathsf{R}$ is constructed by recursive division of $1 + 1$ on 1 and $\tilde{\exists}!(a : A).\ P\ a$ stands for $\tilde{\exists}(a : A).\ P\ a \wedge \Pi(b : A).\ P\ b \to a = b$. Note that P can be seen as a data that classically defines a real number. The axiom says that when we have a procedure that computes a 2^{-n} approximation to the real number for each n, we have the real number constructively.

Example 3. In many cases, we compute an approximation of a real number using multivalued computation. Using elimM and $\omega\mathsf{lift}$, we can define

$$\overline{\mathsf{lim}}\ P\ p : \Pi(n : \mathsf{N}).\ \overline{\Sigma}(e : \mathsf{R}).(\tilde{\exists}(a : \mathsf{R}).\ P\ a \wedge -2^{-n} < e - a < 2^{-n})) \to \Sigma(a : \mathsf{R}).\ P\ a.$$

where $P : \mathsf{R} \to \mathsf{Prop}$ and $p : \tilde{\exists}!(z : \mathsf{R}).\ P\ z$. Namely, when we have a procedure that computes a *multivalued* approximation to a real number, the procedure itself gets converted to the real number.

3.3 Soundness by Realizabiltiy

To prove soundness of the set of axioms, we extend the standard realizability interpretation of extensional dependent type theories to the category of assemblies over Kleene's second algebra with computable morhpisms $\mathsf{Asm}(\mathcal{K}_2)$ [19, § 4 and § 5]. That is, to each type constant $A : \mathsf{Type}$ we axiomatize, we designate an assembly $[\![A : \mathsf{Type}]\!]$ and to each axiomatic term constant $c : A$, we assign a morphism $[\![c : A]\!] : \mathbf{1} \to [\![A : \mathsf{Type}]\!]$ in $\mathsf{Asm}(\mathcal{K}_2)$ where $\mathbf{1}$ is a terminal object.

In consequence, by extending the interpretation, we not only prove soundness of the axiomatization but also argue that a closed term in our type theory automatically gives a construction of a computable function in the sense of computable analysis. For example, suppose we have a proof of the statement

$$\Pi(x : \mathsf{R}).\ P\ x \rightrightarrows \Sigma(y : \mathsf{R}).\ Q\ x\ y$$

where $P : \mathsf{R} \to \mathsf{Prop}$ and $Q : \mathsf{R} \to \mathsf{R} \to \mathsf{Prop}$. The interpretation of the proof is a computable partial multifunction $f :\subseteq \mathbb{R} \rightrightarrows \mathbb{R}$ where for any $x \in \mathbb{R}$ such that $[\![P]\!](x) = \mathbf{1}$, $f(x)$ is well-defined and for any $y \in f(x)$, $[\![Q]\!](x, y) = \mathbf{1}$.

For our axioms, we interpret K to the Kleenean assembly $\mathbf{K}$ and R to any effective assembly of real numbers $\mathbf{R}$. Mapping the axiomatic constants properly, e.g., `true` to tt and `false` to $f\!f$, validates most of the axioms.

In order to interpret the multivaluedness, we specify the endofunctor $\mathbf{M} : \mathsf{Asm}(\mathcal{K}_2) \to \mathsf{Asm}(\mathcal{K}_2)$ such that for an assembly $\mathbf{A}$, $\mathbf{M}\ \mathbf{A}$ is an assembly of the set of nonempty subsets of $\mathbf{A}$ whose realization relation $\Vdash$ is defined by

$$\varphi \Vdash_{\mathbf{M}\ \mathbf{A}} S \quad :\Leftrightarrow \quad \exists x.\ x \in S \wedge \varphi \Vdash_{\mathbf{A}} x\ .$$

In words, φ realizes a nonempty subset S of $\mathbf{A}$ if φ realized an element x of S in the original $\mathbf{A}$. Note that for any assemblies $\mathbf{A}, \mathbf{B}$, a multifunction $f : \mathbf{A} \rightrightarrows \mathbf{B}$ is computable if and only if it appears as a morhpism $f : \mathbf{A} \to \mathbf{M}\ \mathbf{B}$.

The endofunctor $\mathbf{M}$ is a monad whose unit is $\eta_{\mathbf{A}} : x \mapsto \{x\}$, multiplication is $\mu_{\mathbf{A}} : S \mapsto \bigcup_{T \in S} T$, and its action on morphisms is $\mathbf{M}(f) : S \mapsto \bigcup_{x \in S}\{f(x)\}$.

When $\mathbf{A}$ is sub-singleton, $\mathbf{M}\ \mathbf{A}$ is isomorphic to $\mathbf{A}$. And, for any sequence of assemblies $(\mathbf{A}_i)_{i \in \mathbb{N}}$, there is a mapping $\Pi_{i \in \mathbb{N}} \mathbf{M}(\mathbf{A}_i) \to \mathbf{M}(\Pi_{i \in \mathbb{N}} \mathbf{A}_i)$ that collects all sections of $f \in \Pi_{i \in \mathbb{N}} \mathbf{M}(\mathbf{A}_i)$. The axioms of multivalue types are validated by mapping the monad structure of M to the monad structure of $\mathbf{M}$ and mapping `select` to **select**. Discussions thus far conclude the soundness of our axioms:

Lemma 1. *The axiomatization is sound admitting a realizability interpretation.*

4 Relating Classical Analysis

Although our axiomatization is constructive, in some cases we allow a certain amount of classical reasoning to prove non-computational properties. For example, in terms of program extraction (c.f. Section 5) we often want to prove a statement of the form $\Pi(x : \mathsf{R}).\ \Sigma(y : \mathsf{R}).\ P\ x\ y$ where $P : \mathsf{R} \to \mathsf{R} \to \mathsf{Prop}$. To do this, we assume any $x : \mathsf{R}$, provide an explicit $y : \mathsf{R}$ and prove that $P\ x\ y$ holds. $P\ x\ y : \mathsf{Prop}$ is a classical statement and thus admits nonconstructive proofs.

As mentioned in the introduction, most proof assistants already provide formalizations of classical reals and some theory upon them. Instead of rebuilding all this theory on top of our axiomatization, in the above situation it would be more practical to have a way to carefully apply classical results to our type without breaking constructivity.

More concretely, let us assume a Coq-like dependent type theory that already provides a rich theory of classical analysis through a type $\tilde{\mathsf{R}}$. Here, by classical analysis, we mean that classical statements such as $\Pi(x : \tilde{\mathsf{R}}).\ x > 0 + \neg(x > 0)$ hold in the type theory. We want to embed our axiomatization and apply theorems proven over the classical theory to our formalization while separating the constructive part and the classical part of the type theory correctly so that realizability results like those from Section 3.3 still hold.

Even though the type theory provides classical types and terms, it stays fully constructive for the terms that do not access the classical axioms. That means, a term in the type theory can be formally interpreted into two different models. We have two type judgements $\tilde{\vdash}\ t : A$ saying that t of type A may rely on classical axioms and $\vdash t' : A'$ saying that t' of type A' is free from any classical axioms. When $\tilde{\vdash}\ t : A$, we interpret it in the category of sets Set and when $\vdash t : A$, we interpret it in $\mathsf{Asm}(\mathcal{K}_2)$. For example, $\tilde{\vdash}\ t : \Pi(x : \tilde{\mathsf{R}}).\ x > 0 + \neg(x > 0)$ is derivable for some t, but $\vdash t : \Pi(x : \tilde{\mathsf{R}}).\ x > 0 + \neg(x > 0)$ is not for the same t.

The goal is to correctly relate the two type judgements. One way is obvious: when $\vdash t : A$ is derivable, then so is $\tilde{\vdash}\ t : A$.[5] However, we are more interested in the other direction, i.e. how we can get a constructively well-typed term from classical well-typedness.

Recall that Set is a reflective subcategory of $\mathsf{Asm}(\mathcal{K}_2)$ by the forgetful functor $\boldsymbol{\Gamma} : \mathsf{Asm}(\mathcal{K}_2) \to \mathsf{Set}$ and its right adjoint $\boldsymbol{\nabla} : \mathsf{Set} \to \mathsf{Asm}(\mathcal{K}_2)$ where for any set A, $\boldsymbol{\nabla} A$ is the assembly of A with the trivial realization relation [24, Theorem 1.5.2].

For each type A, define

$$\nabla A :\equiv \Sigma(P : A \to \mathsf{Prop}).\ \tilde{\exists}!(x : A).\ P\ x\ .$$

See that for any type A, $\llbracket \tilde{\vdash}\ \nabla A : \mathsf{Type} \rrbracket$ is isomorphic to $\llbracket \tilde{\vdash}\ A : \mathsf{Type} \rrbracket$ in Set and $\llbracket \vdash \nabla A : \mathsf{Type} \rrbracket$ is isomorphic to $\boldsymbol{\nabla\Gamma} \llbracket \vdash A : \mathsf{Type} \rrbracket$ in $\mathsf{Asm}(\mathcal{K}_2)$. It can be understood as a functor that erases all the computational structure of A while keeping its set-theoretic structure.

We add the type judgement rule:

$$\frac{\tilde{\vdash}\ t : \nabla A}{\vdash t : \nabla A}\ (\textsc{Relate})$$

saying that when t is a classically constructed term of type ∇A, we judge t also as a constructive term of type ∇A. See that this judgement rule is validated in our

[5] However, this is no longer true if we assumed counter-classical axioms such as the continuity principle.

interpretation. When $\tilde{\vdash} t : A$, we have a function $[\![\tilde{\vdash} t : \nabla A]\!] : \{*\} \to [\![\tilde{\vdash} A : \mathsf{Type}]\!]$ in Set. Hence, we interpret $[\![\vdash t : \nabla A]\!]$ to be $\boldsymbol{\nabla}[\![\tilde{\vdash} t : A]\!] : \boldsymbol{\nabla}\{*\} \to \boldsymbol{\nabla}[\![\tilde{\vdash} A : \mathsf{Type}]\!]$ in $\mathsf{Asm}(\mathcal{K}_2)$. Note that $\boldsymbol{\nabla}\{*\} \simeq \mathbf{1}$.

It is provable in the type theory using the assumptions of Prop being the type of classical propositions admitting propositional extensionality that ∇ is an idempotent monad where its unit $\mathsf{unit}_\nabla : \Pi(A : \mathsf{Type}).\ A \to \nabla A$ on ∇A, is an equivalence with the inverse being the multiplication. Moreover, it holds that $\mathsf{unit}_\nabla\ \mathsf{Prop} : \mathsf{Prop} \to \nabla\mathsf{Prop}$ is an equivalence. That means, given a mapping $f : A_1 \to A_2 \to \cdots \to A_d$, there is a naturally defined lifting $f^{\dagger_\nabla} \colon \nabla A_1 \to \nabla A_2 \to \cdots \to \nabla A_d$ and given a predicate $P : A_1 \to A_2 \to \cdots \to \mathsf{Prop}$, there is $P^{\dagger_\nabla} \colon \nabla A_1 \to \nabla A_2 \cdots \to \mathsf{Prop}$.

We assume the map $\mathsf{relator} : \mathsf{R} \to \nabla\tilde{\mathsf{R}}$ to relate our axiomatic real numbers with classical analysis (Axiom ∇1). Its interpretation in Set is the identity map $[\![\tilde{\vdash}\ \mathsf{relator} : \mathsf{R} \to \nabla\tilde{\mathsf{R}}]\!] : \mathbb{R} \ni x \mapsto x \in \mathbb{R}$. We assume enough axioms that characterize the mapping (Axiom ∇1-∇10). For example, $\mathsf{relator}\ 0 = \mathsf{unit}_\nabla\ \tilde{\mathsf{R}}\ 0$ (Axiom ∇4), $\Pi(x, y : \mathsf{R}).\ \mathsf{relator}(x+y) = (\mathsf{relator}\ x) +^{\dagger_\nabla} (\mathsf{relator}\ y)$ (Axiom ∇6), $\Pi(x, y : \mathsf{R}).\ (x < y) = (\mathsf{relator}\ x) <^{\dagger_\nabla} (\mathsf{relator}\ y)$ (Axiom ∇10), and so on.

Example 4. Suppose from the theory of classical analysis, we have a term f saying that for any positive real number, there is a square root:

$$\tilde{\vdash} f : \Pi(x : \tilde{\mathsf{R}}).\ 0 < x \to \Sigma(y : \tilde{\mathsf{R}}).\ x = y \times y$$

From the judgement rule (Relate), we have

$$\vdash \mathsf{unit}_\nabla f : \nabla\Pi(x : \tilde{\mathsf{R}}).\ 0 < x \to \Sigma(y : \tilde{\mathsf{R}}).\ x = y \times y$$

in the constructive part of the type theory. Transferring ∇, we get

$$\vdash (\mathsf{unit}_\nabla f)' : \Pi(x : \nabla\tilde{\mathsf{R}}).\ (\mathsf{unit}_\nabla 0) <^{\dagger_\nabla} x \to \Sigma(y : \nabla\tilde{\mathsf{R}}).\ x = y \times^{\dagger_\nabla} y$$

Using the axioms of the relator, we can obtain a term of type

$$\vdash \Pi(x : \mathsf{R}).\ 0 < x \to \tilde{\exists}(y : \mathsf{R}).\ x = y \times y : \mathsf{Prop}.$$

Thus, we can transport a classical proof of the existence of square root based on $\tilde{\mathsf{R}}$ to a constructive proof of the classical existence of square root based on R.

5 Implementation and Examples

We implemented the above theory in the Coq proof assistant.[6] From a correctness proof in our implementation, we can extract Haskell code that uses the AERN library to perform basic real number arithmetic operations. For this, we introduce several extraction rules replacing operations on the constructive reals with the corresponding AERN function. The extracted code requires only minor mechanical editing, such as adding import statements (c.f. Appendix B for details of the extraction process).

Let us present the main features of our implementation by giving some examples of operations on real numbers.

[6] The source code can be found on https://github.com/holgerthies/coq-aern

```
Lemma Realmax : forall x y, {z | W_M x y z}.
Proof.
  intros.
  apply mslimit.
  + (* max is single valued predicate *) ...
  + (* construct limit *)
        intros.
        apply (mjoin (x>y - prec n)
                     (y > x - prec n)).
        ++ intros [c1|c2].
           +++ (* when x>y-2^n *)
         exists x. ...
           +++ (* when x<y+2^n *)
         exists y. ...
        ++ apply M_split.
           apply prec_pos.
Defined.
```

```
realmax :: CReal -> CReal -> CReal
realmax x y =
  mslimit (\n ->
    mjoin (\h -> case h of {
                  True -> x;
                  False -> y})
      (m_split x y (((creal 0.5)^) n)))
```

Fig. 1. Outline of a Coq proof and corresponding extracted Haskell code

5.1 Maximization

A simple example for an operation that requires multivaluedness in its definition is the maximization operator that takes to real numbers x and y and returns their maximum. We can define it by the following Coq statement.[7]

```
forall x y, {z | (x > y -> z = x) /\ (x = y -> z = x) /\ (x < y -> z = y)}.
```

The statement can be proven by applying the limit operator defined in Example 3. That is, we have to show that there exists exactly one `z: Real` for which the condition in the above statement holds and that for each `n : nat` we can construct a `e : Real` multivaluedly that approximates `z` up to error 2^{-n}. The first part can be easily concluded from the axioms over the `Real` type. The approximation can be constructed by concurrently testing whether $x > y - 2^{-n}$ or $x < y + 2^{-n}$, i.e. by multivalued branching from Example 2. In the first case, x can be used as the desired approximation and in the second case y.

Extracting code from this proof yields a maximization operator in AERN. Fig. 1 shows parts of the Coq proof and the extracted Haskell code.

5.2 Intermediate Value Theorem (IVT)

A classical example from computable analysis (see e.g. [25, Chapter 6.3]) is finding the zero of a continuous, real valued function $f : [0, 1] \to \mathbb{R}$ with $f(0) < 0$ and $f(1) > 0$ under the assumption that there is exactly one zero in the interval (i.e. a constructive version of the intermediate value theorem from analysis).

More precisely, we prove the following statement in Coq.

```
forall (f : Real -> Real),
    continuous f -> uniq f 0 1 -> {z | 0<z<1 /\ f z = 0}.
```

[7] For sake of presentation, we applied some slight, non-essential simplifications to the Coq statements in this section compared to the original source code.

Here, `continuous` is defined using the usual ϵ-δ-criterion and `uniq f a b` is the statement that f has exactly one zero in the interval $[a, b]$. The statement can be proven using the trisection method which is similar to the classical bisection method but avoids uncomputable comparison to 0. That is we inductively define sequences a_i, b_i with $f(a_i) * f(b_i) < 0$ and $b_i - a_i \leq (2/3)^i$. In each step we let $a'_i := (2a_i + b_i)/3$, $b'_i := (a_i + 2b_i)/3$ and in parallel check if $f(a'_i) * f(b_i) < 0$ or $f(a_i) * f(b'_i) < 0$. In the first case we set $a_{i+1} := a'_i$, $b_{i+1} := b_i$, in the second case $a_{i+1} := a_i$, $b_{i+1} := b'_i$. As at least one of the inequalities is true by the assumptions, this selection can be done using the multivalued `choose` operator from Section 3.1. The zero can then be defined using the limit operator. Again, we can extract an AERN program from the proof. The extracted program is an implementation of a root finding algorithm using the above algorithm.

5.3 Classical proofs and a fast square root algorithm

As a final example let us look at how to use the relator operation defined in Section 4 to prove facts about our constructive real type using classical results from the Coq standard library. We follow an example from [11] that implements the Heron method to compute the square root of a real number in the Incone library. The proof is interesting as it is mostly classical and makes use of some of the theory and external libraries for classical analysis that are already available for Coq. Making use of this huge repertoire on theory already formalized in Coq vastly simplifies the proof. We repeated the example using our new implementation and compared it to the implementation in Incone.

The Heron method is an approximation scheme for the square root of a real $x \in \mathbb{R}$ by the sequence inductively defined by $x_0 := 1$ and $x_{i+1} := \frac{1}{2}\left(x_i + \frac{x}{x_i}\right)$. In this work we only consider a restricted version where $\frac{1}{4} \leq x \leq 2$. In this interval, the sequence converges quadratically to $\sqrt{x}$, i.e. $|x_i - \sqrt{x}| \leq 2^{-2^i}$. This restricted version can be expanded to all non-negative reals (see the aforementioned work on Incone) but we omit it here as it provides little new insights.

We prove the following statement in Coq.

```
forall x, (/ 4) <= x -> (x <= 2) -> {y | 0 <= y /\ y * y = x}.
```

The Coq standard library already contains a (non-constructive) definition of a function `sqrt` and proves many of its properties. To prove our statement, we construct a real number y by applying the limit operator to the sequence defined by the Heron iteration scheme. We then relate it to the classical real number `sqrt`(x) and use the characteristics of `sqrt` to show the condition. All necessary properties to show that the relation holds are again proven purely classical using tools from the standard library and other libraries building upon it.

The proof is very similar to the one in Incone and we could reuse large parts of it without major adaptions. It should be noted though, that Incone additionally requires to prove the existence of a realizer in the sense of computable analysis which adds an additional layer of complexity that is not required with our axiomatic approach and the new proof therefore becomes significantly simpler.

5.4 Performance measurements

Since our axiomatization of constructive reals is built on a datatype similar to that used by AERN, we expect the performance of the extracted programs to be similar to that of hand-written AERN code. The measurements summarized below are consistent with our hypothesis.[8] iRRAM is known to be one of the most efficient implementations of exact real computation and thus we also included hand-written iRRAM versions for calibration. For IVT, the hand-written AERN code features tail recursion and the iRRAM code updates `a` and `b` in-place.

Benchmark		Average execution time (s)			
Formula	Accuracy	Extracted	Hand-written	Native	iRRAM
$\max(0, \pi - \pi)$	10^6 bits	3.5	3.8	3.8	1.59
$\sqrt{2}$	10^6 bits	0.72	0.70	0.40	0.62
$\sqrt{\sqrt{2}}$	10^6 bits	1.52	1.38	0.85	1.15
$x - 0.5 = 0$	10^3 bits	1.44	0.32	—	0.03
$x(2 - x) - 0.5 = 0$	10^3 bits	2.02	0.35	—	0.04
$\sqrt{x + 0.5} - 1 = 0$	10^3 bits	12.9	2.35	—	0.29

6 Conclusion and Future Work

We presented a new axiomatization of constructive reals in a type theory and proved its soundness with respect to the standard realizability interpretation from computable analysis. We implemented our theory in Coq and used Coq's code extraction features to generate efficient Haskell programs for exact real computation based on the AERN library.

We think our new axiomatization is particularly well-suited for verifying exact real computation programs built on top of the theory of computable analysis. Nevertheless, we plan to more thoroughly compare our implementation with other implementations of constructive reals in Coq and other proof assistants in the future. In particular, we plan to take a deeper look at the C-CoRn library and how it differs from our implementation. Relating to other constructive formalization would also allow execution directly in the proof assistant.

From a more practical point of view, we plan to extend our implementation by other important operations on real numbers such as trigonometric and exponential functions and mathematical constants such as π and e. Such extensions should be straight-forward and we do not expect any major difficulties in their implementation. Maybe more interestingly, we also plan to extend to more complicated operations such as solution operators for ordinary or partial differential equations by applying recent ideas from real complexity theory [9, 12].

[8] Benchmarks were run 10 times on a Lenovo T440p laptop with Intel i7-4710MQ CPU and 16GB RAM, OS Ubuntu 18.04, compiled using Haskell Stackage LTS 17.2.

References

1. Balluchi, A., Casagrande, A., Collins, P., Ferrari, A., Villa, T., Sangiovanni-Vincentelli, A.L.: Ariadne: a framework for reachability analysis of hybrid automata. In: In: Proceedings of the International Syposium on Mathematical Theory of Networks and Systems. (2006)
2. Boldo, S., Filliâtre, J.C., Melquiond, G.: Combining coq and gappa for certifying floating-point programs. In: Carette, J., Dixon, L., Coen, C.S., Watt, S.M. (eds.) Intelligent Computer Mathematics. pp. 59–74. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
3. Boldo, S., Lelay, C., Melquiond, G.: Formalization of real analysis: A survey of proof assistants and libraries. Mathematical Structures in Computer Science **26**(7), 1196–1233 (2016), http://hal.inria.fr/hal-00806920
4. Boldo, S., Melquiond, G.: Flocq: A unified library for proving floating-point algorithms in coq. In: 2011 IEEE 20th Symposium on Computer Arithmetic. pp. 243–252. IEEE (2011)
5. Brattka, V., Hertling, P.: Feasible real random access machines. Journal of Complexity **14**(4), 490–526 (1998). https://doi.org/https://doi.org/10.1006/jcom.1998.0488, https://www.sciencedirect.com/science/article/pii/S0885064X98904885
6. Cruz-Filipe, L., Geuvers, H., Wiedijk, F.: C-CoRN, the constructive Coq repository at Nijmegen. In: International Conference on Mathematical Knowledge Management. pp. 88–103. Springer (2004)
7. Hertling, P.: A real number structure that is effectively categorical. Math. Log. Q. **45**, 147–182 (1999). https://doi.org/10.1002/malq.19990450202, https://doi.org/10.1002/malq.19990450202
8. Hofmann, M.: On the interpretation of type theory in locally cartesian closed categories. In: Pacholski, L., Tiuryn, J. (eds.) Computer Science Logic. pp. 427–441. Springer Berlin Heidelberg, Berlin, Heidelberg (1995)
9. Kawamura, A., Steinberg, F., Thies, H.: Parameterized complexity for uniform operators on multidimensional analytic functions and ODE solving. In: International Workshop on Logic, Language, Information, and Computation. pp. 223–236. Springer (2018)
10. Konecnỳ, M.: AERN-Real: Arbitrary-precision interval arithmetic for approximating exact real numbers (2008)
11. Konečnỳ, M., Steinberg, F., Thies, H.: Computable analysis for verified exact real computation. In: 40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2020)
12. Koswara, I., Selivanova, S., Ziegler, M.: Computational complexity of real powering and improved solving linear differential equations. In: International Computer Science Symposium in Russia. pp. 215–227. Springer (2019)
13. Kreitz, C., Weihrauch, K.: Theory of representations. Theoretical computer science **38**, 35–53 (1985)
14. Luckhardt, H.: A fundamental effect in computations on real numbers. Theoretical Computer Science **5**(3), 321 – 324 (1977). https://doi.org/https://doi.org/10.1016/0304-3975(77)90048-2, http://www.sciencedirect.com/science/article/pii/0304397577900482
15. Melquiond, G.: Proving bounds on real-valued functions with computations. In: International Joint Conference on Automated Reasoning. pp. 2–17. Springer (2008)

16. Müller, N.T.: The iRRAM: Exact arithmetic in C++. In: International Workshop on Computability and Complexity in Analysis. pp. 222–252. Springer (2000)
17. Palmgren, E.: On universes in type theory. Twenty five years of constructive type theory pp. 191–204 (1998)
18. Park, S., Brauße, F., Collins, P., Kim, S., Konečnỳ, M., Lee, G., Müller, N., Neumann, E., Preining, N., Ziegler, M.: Foundation of computer (algebra) analysis systems: Semantics, logic, programming, verification. arXiv e-prints pp. arXiv–1608 (2016)
19. Reus, B.: Realizability models for type theories. Electronic Notes in Theoretical Computer Science **23**(1), 128–158 (1999)
20. Schröder, M.: Effectivity in spaces with admissible multirepresentations. Mathematical Logic Quarterly: Mathematical Logic Quarterly **48**(S1), 78–90 (2002)
21. Schwichtenberg, H.: Constructive analysis with witnesses. Proof Technology and Computation. Natio Science Series pp. 323–354 (2006)
22. Seely, R.A.G.: Locally cartesian closed categories and type theory. Mathematical Proceedings of the Cambridge Philosophical Society **95**(1), 33–48 (1984). https://doi.org/10.1017/S0305004100061284
23. Steinberg, F., Thery, L., Thies, H.: Computable analysis and notions of continuity in coq. arXiv preprint arXiv:1904.13203 (2019)
24. Van Oosten, J.: Realizability: an introduction to its categorical side. Elsevier (2008)
25. Weihrauch, K.: Computable analysis. Springer, Berlin (2000)

A Full list of axioms

Here we list all our axioms, grouped by the files in our implementation.
`Base.v` defines our base type theory, making it extensional and Prop classical:

TT1 $\Pi(P : \mathsf{Prop}).\ P \vee \neg P$
TT2 $\Pi(P, Q : \mathsf{Prop}).\ (P \to Q) \to (Q \to P) \to P = Q$
TT3 $\Pi(A, B : \mathsf{Type}).\ \Pi(f, g : A \to B).\ (\Pi(x : A).\ f\ x = g\ x) \to f = g$

`Kleene.v` axiomatizes the type of Kleeneans and the multivalued monad.

K1 $\mathsf{K} : \mathsf{Type}$
K2 $\mathsf{true} : \mathsf{K}$
K3 $\mathsf{false} : \mathsf{K}$
K4 $\mathsf{true} \neq \mathsf{false}$
K5 $\hat{\neg} : \mathsf{K} \to \mathsf{K}$
K6 $\hat{\vee} \colon \mathsf{K} \to \mathsf{K} \to \mathsf{K}$
K7 $\hat{\wedge} \colon \mathsf{K} \to \mathsf{K} \to \mathsf{K}$
Define $\lceil k : \mathsf{K} \rceil :\equiv k = \mathsf{true}$, $\lfloor k : \mathsf{K} \rfloor :\equiv k = \mathsf{false}$, and $(k : \mathsf{K})\!\downarrow :\equiv \lceil k \rceil \vee \lfloor k \rfloor$.
K8 $x\!\downarrow \to \lceil x \rceil + \lfloor x \rfloor$
Kleene logic operations:
K9 $\lceil \hat{\neg} x \rceil = \lfloor x \rfloor$ and $\lfloor \hat{\neg} x \rfloor = \lceil x \rceil$
K10 $\lceil x \mathbin{\hat{\wedge}} y \rceil = \lceil x \rceil \wedge \lceil y \rceil$ and $\lfloor x \mathbin{\hat{\wedge}} y \rfloor = \lfloor x \rfloor \vee \lfloor y \rfloor$
K11 $\lceil x \mathbin{\hat{\vee}} y \rceil = \lceil x \rceil \vee \lceil y \rceil$ and $\lfloor x \mathbin{\hat{\vee}} y \rfloor = \lfloor x \rfloor \wedge \lfloor y \rfloor$

The monad structure:

M1 $\mathsf{M} : \mathsf{Type} \to \mathsf{Type}$

M2 $\mathsf{unitM} : \Pi(A : \mathsf{Type}).\ A \to \mathsf{M}\ A$

M3 $\mathsf{multM} : \Pi(A : \mathsf{Type}).\ \mathsf{M}\ (\mathsf{M}\ A) \to \mathsf{M}\ A$

M4 $\mathsf{liftM} : \Pi(A, B : \mathsf{Type}).\ (A \to B) \to (\mathsf{M}\ A \to \mathsf{M}\ B)$

unitM and multM are natural transformations:

M5 $\Pi(A, B : \mathsf{Type}).\ \Pi(f : A \to B).\ \Pi(x : A).\ \mathsf{liftM}\ A\ B\ f(\mathsf{unitM}\ A\ x) = \mathsf{unitM}, B\ (f\ x)$

M6 $\Pi(A, B : \mathsf{Type}).\ \Pi(f : A \to B).\ \Pi(x : \mathsf{M}\ (\mathsf{M}\ A).$
$\mathsf{multM}\ B((\mathsf{liftM}\ (\mathsf{M}\ A)\ (\mathsf{M}\ B)\ (\mathsf{liftM}\ A\ B\ f))\ x) = (\mathsf{liftM}\ A\ B\ f)\ (\mathsf{multM}\ A\ x)$

The coherence conditions:

M7 $\Pi(A : \mathsf{Type}).\ \Pi(x : \mathsf{M}\ A).\ \mathsf{multM}\ A\ (\mathsf{unitM}\ (\mathsf{M}\ A)\ x) = x$

M8 $\Pi(A : \mathsf{Type}).\ \Pi(x : \mathsf{M}\ A).\ \mathsf{multM}\ A\ (\mathsf{liftM}A\ (\mathsf{M}\ A)\ (\mathsf{unitM}\ A)\ x) = x$

M9 $\Pi(A : \mathsf{Type}).\ \Pi(x : \mathsf{M}\ (\mathsf{M}\ (\mathsf{M}\ A))).\ \mathsf{multM}\ A\ (\mathsf{multM}\ (\mathsf{M}\ A)\ x) =$
$\mathsf{multM}\ A\ (\mathsf{liftM}\ (\mathsf{M}\ (\mathsf{M}\ A))(\mathsf{M}\ A)(\mathsf{multM}\ A)\ x)$

M10 $\mathsf{elimM} : \Pi(A : \mathsf{Type}).\ (\Pi(x, y : A).\ x = y) \to A \to \mathsf{M}\ A$

M11 $\Pi(A : \mathsf{Type}). \Pi(p : \big(\Pi(x, y : A). x = y\big)). \Pi(a : \mathsf{M}\ A). \mathsf{unitM}\ A\ \big(\mathsf{elimM}\ A\ p\ a\big) = a$

M12 $\omega\mathsf{lift} : \Pi(P : \mathsf{N} \to \mathsf{Type}).\ \big(\Pi(x : \mathsf{N}).\ \mathsf{M}\ P(x)\big) \to \mathsf{M}\big(\Pi(x : \mathsf{N}).\ P(x)\big)$

M13 $\Pi(P : \mathsf{N} \to \mathsf{Type}).\ \Pi(f : \big(\Pi(x : \mathsf{N}).\ \mathsf{M}\ P(x)\big)).\ f\ n = \lambda(n : \mathsf{N}).\ \mathsf{liftM}\big(\lambda(f : \big(\Pi(x : \mathsf{N}).\ P(x)\big)).\ f\ n\big)\ (\omega\mathsf{lift}\ P\ f)$

M14 $\mathsf{select} : \Pi(x, y : \mathsf{K}).\ (\lceil x \rceil \vee \lceil y \rceil) \to \mathsf{M}\ \big(\lceil x \rceil + \lceil y \rceil\big)$

`RealAxioms.v` axiomatizes the real numbers:

The structure of real numbers:

R1 $\mathsf{R} : \mathsf{Type}$

R2 $0 : \mathsf{R}$

R3 $1 : \mathsf{R}$

R4 $+ : \mathsf{R} \to \mathsf{R} \to \mathsf{R}$

R5 $\times : \mathsf{R} \to \mathsf{R} \to \mathsf{R}$

R6 $- : \mathsf{R} \to \mathsf{R}$

R7 $/ : \Pi(x : \mathsf{R}).\ x \neq 0 \to \mathsf{R}$

R8 $<: \mathsf{R} \to \mathsf{R} \to \mathsf{Prop}$

Semi-decidability of comparison tests:

R9 $\Pi(x, y : \mathsf{R}).\ \mathsf{semiDec}(x < y)$

Constructive completeness:

R10 $\Pi(P : \mathsf{R} \to \mathsf{Prop}).\ (\tilde{\exists}!(x : \mathsf{R}).\ P\ x) \to (\Pi(n : \mathsf{N}).\ \Sigma(x : \mathsf{R}).\ \tilde{\exists}(\tilde{x} : \mathsf{R}).\ P\ x \wedge -2^{-n} < x - \tilde{x} < 2^{-n}) \to \Sigma(x : \mathsf{R}).\ P\ x$

Classical axioms in Prop:

R11 $\Pi(x, y : \mathsf{R}).\ x + y = y + x$

R12 $\Pi(x, y, z : \mathsf{R}).\ (x + y) + z = x + (y + z)$

R13 $\Pi(x : \mathsf{R}).\ x + -x = 0$

R14 $\Pi(x : \mathsf{R}).\ 0 + x = x$

R15 $\Pi(x, y : \mathsf{R}).\ x \times y = y \times x$

R16 $\Pi(x, y, z : \mathsf{R}).\ (x \times y) \times z = x \times (y \times z)$

R17 $\Pi(x : \mathsf{R}).\ \Pi(p : x \neq 0).\ (/\ x\ p) \times x = 1$

R18 $\Pi(x : \mathsf{R}).\ 1 \times x = x$

R19 $\Pi(x, y, z : \mathsf{R}).\ x \times (y + z) = x \times y + x \times z$

R20 $1 \neq 0$

R21 $1 > 0$

R22 $\Pi(x, y : \mathsf{R}).\ x < y \lor x = y \lor x > y$

R23 $\Pi(x, y : \mathsf{R}).\ x < y \to \neg(y < x)$

R24 $\Pi(x, z, y : \mathsf{R}).\ x < y \to y < z \to x < z$

R25 $\Pi(x, y, z : \mathsf{R}).\ y < z \to x + y < x + z$

R26 $\Pi(x, y, z : \mathsf{R}).\ 0 < x \to y < z \to x \times y < x \times z$

For each $P : \mathsf{R} \to \mathsf{Prop}$ and $x : \mathsf{R}$, define $P < x :\equiv \Pi(y : \mathsf{R}).\ P\ y \to y \leq x$.

R27 $\Pi(P : \mathsf{R} \to \mathsf{Prop}).\ (\tilde{\exists}(x : \mathsf{R}).\ P\ x) \to (\tilde{\exists}(x : \mathsf{R}).\ P \leq x) \to \tilde{\exists}(x : \mathsf{R}).\ P \leq x \land \Pi(y : \mathsf{R}).\ P \leq y \to x \leq y$.

`Nabla.v` defines the idempotent monad ∇ and `RealCoqReal.v` axiomatizes the relator:

∇1 $\mathsf{relator} : \mathsf{R} \to \nabla\tilde{\mathsf{R}}$

∇2 $\Pi(x, y : \mathsf{R}).\ \mathsf{relator}\ x = \mathsf{relator}\ y \to x = y$

∇3 $\Pi(y : \nabla\tilde{\mathsf{R}}).\ \tilde{\exists}(x : \mathsf{R}).\ y = \mathsf{relator}\ x$

∇4 $\mathsf{relator}\ 0 = \mathsf{unit}_{\triangledown}\ \tilde{\mathsf{R}}\ 0$

∇5 $\mathsf{relator}\ 1 = \mathsf{unit}_{\triangledown}\ \tilde{\mathsf{R}}\ 1$

∇6 $\Pi(x, y : \mathsf{R}).\ \mathsf{relator}\ (x + y) = (\mathsf{relator}\ x) +^{\dagger\triangledown} (\mathsf{relator}\ y)$

∇7 $\Pi(x, y : \mathsf{R}).\ \mathsf{relator}\ (x \times y) = (\mathsf{relator}\ x) \times_{\triangledown} (\mathsf{relator}\ y)$

∇8 $\Pi(x : \mathsf{R}).\ \mathsf{relator}\ (-x) =-^{\dagger\triangledown} (\mathsf{relator}\ x)$

∇9 $\Pi(x : \mathsf{R}).\ \Pi(p : x \neq 0).\ \mathsf{relator}\ (/x\ p) =/^{\dagger\triangledown} (\mathsf{relator}\ x)$

∇10 $\Pi(x, y : \mathsf{R}).\ (x < y) = (\mathsf{relator}\ x) <^{\dagger\triangledown} (\mathsf{relator}\ y)$

B Code extraction

We have two modes for extraction, defined in files `Extract.v` and `ExtractMB.v`. Here are selected key mappings defined in these files:

Coq	AERN (`Extract.v`)	AERN (`ExtractMB.v`)
`Real`	`CReal`	`WithCurrentPrecision (CN MPBall) p`
`Real0`	`0`	`0`
`Realplus`	`(+)`	`(+)`
`limit`	`limit`	`limit`
`choose`	`select`	`select`
`Realltb`	`(<)`	`(<)`
`K`	`CKleenean`	`CN Kleenean`
`sumbool`	`Bool`	`CN Bool`
`M`	type identity	type identity
`unitM`	`id`	`id`
`Nat.log2`	`(integer . integerLog2)`	`(integer . integerLog2)`

Note that the monad M does not appear in the extracted programs. Multivaluedness is intrinsic thanks to redundancy in the underlying representations.

The AERN comparison `<` returns a (lazy) Kleenean for real numbers.

The type `CN MPBall` is the interval type underpinning `CReal`. `ExtractMB.v` produces programs that execute a bit like iRRAM programs, converging to `CReal` computation with increasing precision `p`. This mode gives an efficient implementation of the `choose` operator. We used it in our benchmarks in Section 5.4.

Running the extracted code requires a few simple mechanical modifications, which are specified in files `Extract.v` and `ExtractMB.v`.