Computable Analysis for Verified Exact Real Computation

Michal Konečný
School of Engineering and Applied Science, Aston University, UK
m.konecny@aston.ac.uk

Florian Steinberg
Inria Saclay, France
fsteinberg@gmail.com

Holger Thies
Department of Informatics, Kyushu University, Japan
thies@inf.kyushu-u.ac.jp

Abstract

We use ideas from computable analysis to formalize exact real number computation in the Coq proof assistant. Our formalization is built on top of the Incone library, a Coq library for computable analysis. We use the theoretical framework that computable analysis provides to systematically generate target specifications for real number algorithms. First we give very simple algorithms that fulfill these specifications based on rational approximations. To provide more efficient algorithms, we develop alternate representations that utilize an existing formalization of floating-point algorithms and interval arithmetic in combination with methods used by software packages for exact real arithmetic that focus on execution speed. We also define a general framework to define real number algorithms independently of their concrete encoding and to prove them correct. Algorithms verified in our framework can be extracted to Haskell programs for efficient computation. The performance of the extracted code is comparable to programs produced using non-verified software packages. This is without the need to optimize the extracted code by hand.

As an example, we formalize an algorithm for the square root function based on the Heron method. The algorithm is parametric in the implementation of the real number datatype, not referring to any details of its implementation. Thus the same verified algorithm can be used with different real number representations. Since Boolean valued comparisons of real numbers are not decidable, our algorithms use basic operations that take values in the Kleeneans and Sierpinski space. We develop some of the theory of these spaces. To capture the semantics of non-sequential operations, such as the "parallel or", we use multivalued functions.

2012 ACM Subject Classification Theory of computation → Logic and verification; Mathematics of computing → Continuous mathematics

Keywords and phrases Computable Analysis, exact real computation, formal proofs, proof assistant, Coq

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2020.50

Supplementary Material The incone library at `https://github.com/FlorianSteinberg/incone`

Funding *Michal Konečný*: This project has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 731143.
Holger Thies: Supported by JSPS KAKENHI Grant Numbers JP18J10407 and JP20K19744 and by the Japan Society for the Promotion of Science (JSPS), Core-to-Core Program (A. Advanced Research Networks).

40th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2020).
Editors: Nitin Saxena and Sunil Simon; Article No. 50; pp. 50:1–50:18
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Computable analysis is a formal model for computation on real numbers and other spaces of interest in analysis [25, 9]. It extends classical computability theory from discrete structures to continuous ones. The model of computation used in computable analysis operates on properly infinite data while being realistic in the sense that proofs of computability specify algorithms that can in principle be implemented. Software for computation on the reals based on ideas from computable analysis is often labeled as *exact real computation* as such software allows to approximate real number outputs up to any desired precision. In practice, this can be realized in different ways and several implementations exist [22, 17, 3, 13]. In contrast to implementations using floating-point arithmetic, algorithms from computable analysis have sound compositional semantics and come with a mathematical correctness proof, making them well-suited for safety-critical applications. Proof assistants and formal methods are increasingly used to verify the correctness of such software and computable analysis goes well with this kind of verification.

In this work we present a new and fully formally verified implementation of exact real computation in Coq that makes use of Coq's code extraction features to generate efficient Haskell code for algorithms written and verified inside the proof assistant. The work builds on the Incone library, a formalization of ideas from computable analysis in Coq [24]. Implementations of exact real computation usually hide the internal details of the encoding from the user and instead provide a set of basic operations on real numbers that can be used to build more complicated algorithms. We follow this approach by defining a structure for basic operations on real numbers. Instantiating this structure means to explicitly give an encoding of the reals and algorithms for the basic operations and proving them correct. More complicated operations can then be defined using tools for composing functions that are available in the Incone library. Correctness proofs can be made independent of the concrete representation and different representations can be exchanged and compared easily. As algorithms verified in the proof assistant can be extracted to efficient Haskell code we hope that our work allows developers of exact real computation libraries to verify some particularly critical fragments and easily integrate the generated code into the library.

Computing with real numbers is central in many applications. It should therefore not be surprising that a treatment of the reals is available in most modern proof assistants [6]. In the Coq proof assistant in particular there exist a wide range of work covering the spectrum from purely mathematical and inherently non-computational [5, 2] to verification of approximate computations and concrete error bounds [7, 20].

In this work we treat the real numbers as a represented space. A represented space is an infinite data type that is both exact and fully computational but reasoned about using classical mathematics. Our work is by far not the first implementation of fully computational reals in a proof assistant, or even in Coq. A popular implementation is the C-CoRn library [12] which is based on constructive mathematics. Working constructively has the advantage that every proof has computational content. A constructive proof of an existential statement, for instance, gives rise to an algorithm to compute said object. The price to pay is that a constructive proof is harder to find and this extra effort may not be worthwhile in particular for proofs of correctness, where the computational content is of little to no interest. Most mathematicians and computer scientists distinguish formulation of algorithms from proving its correctness, and prefer the use of classical reasoning for the latter.

Our work and the Incone library are based on computable analysis which is a part of classical mathematics. For our implementation this means that we use the classical axiomatization of the reals from Coq's standard library for specification. Computational content is added in a second step through the use of encodings over certain spaces of functions and the formulation of algorithms on these. Thus, there is a clear separation between the formulation of an algorithm that operates only on computationally meaningful objects and its (possibly non-constructive) correctness proof that may involve purely mathematical objects such as abstract real numbers. We consider this a more pragmatic approach towards computational reasoning over mathematical structures and hope that it can be appealing to classically trained mathematicians and computer scientists. There are also some more practical advantages of our approach. Many Coq libraries are verified against the reals from the standard library and such libraries can easily be integrated into our development. For example, we rely on a Coq library for interval arithmetic [21, 20] to be able to imitate how the most efficient non-verified packages for exact real computation operate [22, 17].

While we consider the Coq formalization one of the main contributions of this work, we keep the presentation on a more informal mathematical level and only give a short overview of the implementation in Section 6. The interested reader can find all of the source code as part of the Incone library [23]. The parts of the library relevant for this paper are listed in Section 6 as well. A more exhaustive overview of the Incone library can be found in [24].

2 Computable analysis and the Incone library

Computable analysis gives computational meaning to abstract mathematical entities such as real numbers by use of encodings over Baire space $\mathbb{N}^{\mathbb{N}}$ called **representations** [19, 25]. To avoid an overload of coding, here and in our formal development we allow the use of arbitrary countable sets in place of the two copies of the natural numbers in Baire space. Let $\mathbf{Q}$ and $\mathbf{A}$ be two countable sets of **questions** and **answers** and let $\mathcal{B} := \mathbf{A}^{\mathbf{Q}}$ be the set of functions from questions to answers. A **representation** of a set X is a partial, surjective function $\delta \colon \subseteq \mathcal{B} \to X$. For $x \in X$, each $\varphi \in \mathcal{B}$ with $\delta(\varphi) = x$ is called a **name** of x and should be understood to provide on demand information about x by assigning a valid answer to each question about x. A **represented space** is a pair $\mathbf{X} := (X, \delta_{\mathbf{X}})$ of a set X and a representation $\delta_{\mathbf{X}}$ of X.

A standard example is the encoding of reals by rational approximations:

▶ **Example 1** ($\mathbb{R}_{\mathbb{Q}}$: Reals via rational approximations)**.** We denote by $\mathbb{R}_{\mathbb{Q}}$ the represented space of the real numbers together with the representation $\delta_{\mathbb{R}_{\mathbb{Q}}} \colon \subseteq \mathbb{Q}^{\mathbb{Q}} \to \mathbb{R}$ such that

$$\delta_{\mathbb{R}_{\mathbb{Q}}}(\varphi) = x \qquad \iff \qquad \forall \varepsilon > 0 \colon |x - \varphi(\varepsilon)| \leq \varepsilon.$$

While we do not make this a formal requirement, for all of our concrete examples there exist obvious and explicitly definable bijections of $\mathbf{Q}$ and $\mathbf{A}$ with the natural numbers. The skeptical reader can therefore always replace the questions and answers by natural numbers to regain the classical setting from computable analysis where $\mathcal{B}$ is only allowed to be the Baire space. Whenever we talk about computability, we assume that such bijections were fixed and refer to the well-established notion of computability of elements and of partial functions on Baire space. For instance, we call an element x of a represented space $\mathbf{X}$ **computable** if it has a name that is computable as an element of Baire space.

Let $\mathbf{X}$ and $\mathbf{X}'$ be represented spaces and $\mathcal{B} := \mathbf{A}^{\mathbf{Q}}$ denote the space of names of $\mathbf{X}$ and $\mathcal{B}' := \mathbf{A}'^{\mathbf{Q}'}$ that of $\mathbf{X}'$. We say that a function $f \colon \mathbf{X} \to \mathbf{X}'$ is realized by a partial operator $F \colon \subseteq \mathcal{B} \to \mathcal{B}'$ if for each name $\varphi \in \mathcal{B}$ of some $x \in \mathbf{X}$ the value $F(\varphi)$ is defined and a name of $f(x) \in \mathbf{X}'$. As f can be called **continuous** resp. **computable** if it can be realized by such an operator, it suffices to introduce these notions for the partial operators on Baire space. For continuity we use the standard topology that Baire space comes with. Thus,

$F\colon\subseteq B \to \mathcal{B}'$ is continuous if for each $q' \in \mathbf{Q}'$ its return value on a functional input φ is determined by a finite number of φ's values. Formally, we say F is continuous if

$$\forall \varphi \in \mathrm{dom}(F), q', \exists L \in \mathrm{seq}(\mathbf{Q}), \forall \psi \in \mathrm{dom}(F)\colon \varphi|_L = \psi|_L \Rightarrow F(\varphi)(q') = F(\psi)(q')$$

where $\mathrm{seq}(\mathbf{Q})$ denotes the set of finite words over $\mathbf{Q}$. Computability is defined using oracle Turing machines [14], but we refrain from stating this definition here and assume the reader to fill this gap or use his intuition. This intuition should include that computable operations can be partial but never discontinuous.

Different representations of the same set can be compared with regards to intertranslatability, that is by asking whether the identity function is computable if the source and target spaces are equipped with the different representations. If there are continuous resp. computable translations in both directions, the spaces are isomorphic and carry the same topological resp. computability structure.

2.1 Specification of algorithms with multifunctions

Usually each element of a represented space has many names. Thus, it may happen that an operator returns on input of each name of an element a name of a solution of a certain problem but for different names of the same input element returns names of different solutions. In this case the algorithm solves the problem but does not realize any function on the represented spaces. This is a situation that is regularly encountered in computable analysis and a popular tool for capturing the semantics of such algorithms are multivalued functions.

A multivalued function $\mathbf{f}\colon X \rightrightarrows Y$ assigns to each element $x \in X$ a possibly empty set of eligible return values $\mathbf{f}(x) \subseteq Y$. Those x for which $\mathbf{f}(x)$ is non-empty constitute the **domain** $\mathrm{dom}(\mathbf{f}) \subseteq X$ of $\mathbf{f}$. The multifunction is called **total** if its domain is all of X and **single-valued** if each value set has at most one element. A partial function can be considered a single-valued multi-function; this multifunction uniquely specifies the partial function and is total if and only if the partial function is.

A partial function f is said to **choose through** a multifunction $\mathbf{f}$ if on each $x \in \mathrm{dom}(\mathbf{f})$ it returns an eligible return value, i.e. $f(x)$ is defined and an element of $\mathbf{f}(x)$. Note that this allows for the domain of the partial function to be bigger than that of the multifunction. A multifunction should be considered a specification of all the partial functions that choose through it and this defines an important ordering on the multifunctions: A multifunction $\mathbf{f}$ is said to **tighten** another multifunction $\mathbf{g}$, in symbols $\mathbf{f} \prec \mathbf{g}$, if any partial function that is a choice for $\mathbf{f}$ is also a choice for $\mathbf{g}$. This can equivalently be formulated as

$$\mathbf{f} \prec \mathbf{g} \quad\iff\quad \mathrm{dom}(\mathbf{g}) \subseteq \mathrm{dom}(\mathbf{f}) \wedge \forall x \in \mathrm{dom}(\mathbf{g}), \mathbf{f}(x) \subseteq \mathbf{g}(x).$$

If $\mathbf{f}$ and $\mathbf{g}$ correspond to partial functions f and g then $\mathbf{f} \prec \mathbf{g}$ if and only if f is an extension of g and a partial function chooses through a multifunction if and only if the induced multifunction tightens it.

The multivalued functions from X to Y are in one-to-one correspondence with the relations but the natural operations on them differ from those on relations. For instance, for $\mathbf{f}\colon Y \rightrightarrows Z$ and $\mathbf{g}\colon X \rightrightarrows Y$ the composition as multivalued functions is defined as

$$\mathbf{f} \circ \mathbf{g}(x) := \{z \in Z \mid \mathbf{g}(x) \subseteq \mathrm{dom}(\mathbf{f}) \wedge \exists y \in \mathbf{g}(x)\colon z \in \mathbf{f}(y)\}.$$

This defines an associative operation that is asymmetric in contrast to the natural composition of relations which is symmetric. The multifunction composition has the advantage that it respects the interpretation as specifications. Namely, if the partial functions f and g

choose through $\mathbf{f}$ and $\mathbf{g}$ respectively, then their composition as partial functions chooses through the composition of $\mathbf{f}$ and $\mathbf{g}$ as multifunctions. The multifunction composition can be characterized as returning the minimal multifunction w.r.t. tightening such that this is true and not only respects being a choice function but more generally the tightening ordering.

A multifunction $\mathbf{f}\colon \mathbf{X} \rightrightarrows \mathbf{X}'$ between represented spaces $\mathbf{X}$ and $\mathbf{X}'$ is realized by a partial operator $F\colon\subseteq \mathcal{B} \to \mathcal{B}'$ if F chooses through $\delta_{\mathbf{X}'}^{-1} \circ \mathbf{f} \circ \delta_{\mathbf{X}}$. Such an $\mathbf{f}$ is called **continuous** or **computable** if it can be realized by an operator with that property. The above definition unfolds to the usual "a realizer translates each name of an element of the domain to a name of some eligible return value".

2.2 The Incone library

The Incone library formalizes ideas from computable analysis in the Coq proof assistant closely following the outline in the previous section. The equivalent of a represented space in Incone is called a **continuity space**. A continuity space $\mathbf{X}$ is defined as a record consisting of an abstract type X, a space $\mathcal{B}_{\mathbf{X}}$ of names that determines a countable inhabited type of questions $\mathbf{Q}_{\mathbf{X}}$ and a countable type of answers $\mathbf{A}_{\mathbf{X}}$ and finally a specification of a partial surjective function $\delta_{\mathbf{X}}\colon\subseteq \mathcal{B}_{\mathbf{X}} \to X$ referred to as representation.

A number of standard constructions on represented spaces are made available by Incone. For represented spaces $\mathbf{X}$ and $\mathbf{Y}$ there exists a represented space $\mathbf{X} \times \mathbf{Y}$ whose underlying set is the Cartesian product of the sets underlying $\mathbf{X}$ and $\mathbf{Y}$. Similarly, there exists a disjoint union $\mathbf{X} + \mathbf{Y}$ of spaces and a space $\mathbf{X}^{\omega}$ of infinite sequences in $\mathbf{X}$. There is also a space $\mathbf{Y}^{\mathbf{X}}$ of continuous functions, but while this is interesting for possible applications it is of lesser interest for the current paper. Details about these constructions and instructions for installation and use of Incone can be found in [24].

While Incone defines continuity as we presented it earlier, computability is not reflected in a definition but instead captured on the meta level via Coq's type/prop distinction. That is, an axiom-free definition of a realizer should be considered a certificate of computability of a function. While such a realizer is automatically continuous, a proof of this fact would proceed by induction on the structure of Coq terms and can clearly not be carried out internally. In principle it would be possible to extract continuity proofs using a tactic but for now the proofs have to be provided on a case by case basis by hand. Partiality is modeled using sigma types: A partial function takes as input not only an element of Baire space but also a proof that the element is contained in the domain which has to be specified beforehand. This means that the dependent type system of Coq gets involved in a meaningful way.

3 Finite spaces and operations on multifunctions

Besides allowing for computation on spaces of continuum cardinality, the methods of computable analysis can be used to operate on non-discrete finite spaces.

▶ **Example 2** (Sierpinski space)**.** Consider the two-element set $\{\top_{\mathbb{S}}, \bot_{\mathbb{S}}\}$ with the following representation: Let the questions and answers be given by $\mathbf{Q} := \mathbb{N}$ and $\mathbf{A} := \mathbb{B} = \{\mathrm{true}, \mathrm{false}\}$ and as representation use the total function $\delta_{\mathbb{S}}$ such that

$$\delta_{\mathbb{S}}(\varphi) = \top_{\mathbb{S}} \qquad \iff \qquad \exists n, \varphi(n) = \mathrm{true}.$$

The represented space $\mathbb{S} := (\{\top_{\mathbb{S}}, \bot_{\mathbb{S}}\}, \delta_{\mathbb{S}})$ is called **Sierpinski space**. The elements of Sierpinski space denote convergence and divergence, respectively: For any kind of computational process with a meaningful notion of basic computational steps, we can obtain a name of an element of Sierpinski space by saying $\varphi(n) = \text{true}$ if the computation terminates within the first n steps and $\varphi(n) = \text{false}$ otherwise. This reflects the interpretation of $\top_{\mathbb{S}}$ and $\bot_{\mathbb{S}}$: we produce a name of $\top_{\mathbb{S}}$ if and only if we started out with a terminating computation.

▶ **Example 3** (Kleeneans). Another finite space that is important in computable analysis is the three-point set $\{\text{true}_{\mathbb{K}}, \text{false}_{\mathbb{K}}, \bot_{\mathbb{K}}\}$ with names of type $\mathbb{N} \to \text{opt}\,\mathbb{B}$ and representation

$$\delta_{\mathbb{K}}(\varphi) = \begin{cases} b_{\mathbb{K}} & \text{if } \exists n, \varphi(n) = \text{Some}\, b \wedge \forall m < n, \varphi(m) = \text{None} \\ \bot_{\mathbb{K}} & \text{otherwise.} \end{cases}$$

Here, $\text{opt}\,\mathbf{A}$ is the disjoint union of $\mathbf{A}$ with a single new element and for each $a \in \mathbf{A}$ we use $\text{Some}\, a$ for the corresponding element of $\text{opt}\,\mathbf{A}$ and None for the new element. This space denoted by $\mathbb{K}$ is known as the **Kleeneans** as it models Kleene's three-valued logic [4].

A continuously realizable multifunction need not have any partial continuous choice function. As example of such behavior let us consider a version of the parallel or.

▶ **Example 4** (The `which` function). Consider the multifunction $\texttt{which}\colon \mathbb{S} \times \mathbb{S} \rightrightarrows \mathbb{K}$ such that

$$\texttt{which}(s_{\text{false}}, s_{\text{true}}) := \begin{cases} \{b_{\mathbb{K}} \mid s_b = \top_{\mathbb{S}}\} & \text{if this set is non-empty} \\ \{\bot_{\mathbb{K}}\} & \text{otherwise,} \end{cases}$$

This means that the `which` function specifies the correct answers to the question which of the input processes terminates. It has many applications including one later in this paper.

A realizer of `which` can be defined from the projections π_i that get names of the components from a name of a pair via

$$F(\varphi)(n) := \begin{cases} \text{Some false} & \text{if } (\pi_0\varphi)(n) = \text{true} \\ \text{Some true} & \text{if } (\pi_1\varphi)(n) = \text{true} \\ \text{None} & \text{otherwise.} \end{cases}$$

The case distinction above is overlapping and we have to add that if more than one of the conditions are satisfied we choose the top-most option. This corresponds to a non-canonical choice and reordering the overlapping cases in the case distinction gives another valid realizer. Either of the realizers is clearly continuous and even computable and thus, the multivalued function `which` is computable and in particular continuous. However, both realizers return both names of $\text{true}_{\mathbb{K}}$ and $\text{false}_{\mathbb{K}}$ on input of two converging processes and switch between the return values depending on the names of these inputs. This is no coincidence, one may verify that no singlevalued choice function for `which` is continuously realizable.

The element $\bot_{\mathbb{K}}$ of the Kleeneans stands for being undefined and the case distinction in the definition of `which` can be understood as extending a (partial) multivalued function in a canonical way to a total multivalued function. The use of such extensions is standard in more order-oriented models of computation [1]. In general, such an extension embodies a stricter specification than the non-extended version, as a realizer for the latter may behave arbitrarily on elements outside its domain, while a realizer for the former has to guarantee divergence. As the `which` function is computable, there is no difference in this case. Generally, there can only be a difference if the domain of the non-extended function is sufficiently complicated.

3.1 Operations on multifunctions and multivalued branching

Given $\mathbf{f}\colon \mathbf{X} \rightrightarrows \mathbf{Y}$ and $\mathbf{f}'\colon \mathbf{X}' \rightrightarrows \mathbf{Y}'$ consider the multifunction $\mathbf{f} \times \mathbf{f}'\colon \mathbf{X} \times \mathbf{X}' \rightrightarrows \mathbf{Y} \times \mathbf{Y}'$ that on input of a pair returns the Cartesian product of the value sets, i.e.

$$(\mathbf{f} \times \mathbf{f}')(x, x') := \mathbf{f}(x) \times \mathbf{f}'(x').$$

As the Cartesian product is empty if one of the value sets is empty, $\mathbf{f}\times\mathbf{f}'$ should be understood as the parallelization of $\mathbf{f}$ and $\mathbf{f}'$. That is, if computable realizers of $\mathbf{f}$ and $\mathbf{f}'$ are given, a computable realizer for $\mathbf{f}\times\mathbf{f}'$ can be specified by running the realizers for $\mathbf{f}$ and $\mathbf{f}'$ in parallel and returning something once both computations have come to an end.

To appropriately capture multivalued branching we need a similar operation for sums. Given $\mathbf{f}\colon \mathbf{X} \rightrightarrows \mathbf{Y}$ and $\mathbf{f}'\colon \mathbf{X}' \rightrightarrows \mathbf{Y}'$ define a multifunction $\mathbf{f}+\mathbf{f}'\colon \mathbf{X}+\mathbf{X}' \rightrightarrows \mathbf{Y}+\mathbf{Y}'$ by

$$(\mathbf{f}+\mathbf{f}')(p) := \begin{cases} \{\operatorname{inl} y \mid y \in \mathbf{f}(x)\} & \text{if } p = \operatorname{inl} x \\ \{\operatorname{inr} y' \mid y' \in \mathbf{f}'(x')\} & \text{if } p = \operatorname{inr} x'. \end{cases}$$

Now, while $\mathbf{f}\times\mathbf{f}'$ corresponds to parallel execution, $\mathbf{f}+\mathbf{f}'$ corresponds to selective execution.

Next let us formulate branching over multivalued predicates. Consider the function $\operatorname{if}_{\mathbf{X}}\colon \mathbb{B}\times\mathbf{X} \to \mathbf{X}+\mathbf{X}$ defined by

$$\operatorname{if}_{\mathbf{X}}(b,x) := \begin{cases} \operatorname{inl} x & \text{if } b = \text{true} \\ \operatorname{inr} x & \text{if } b = \text{false}. \end{cases}$$

Branching over the values of a function $b\colon \mathbf{X} \to \mathbb{B}$ given $f_0, f_1\colon \mathbf{X} \to \mathbf{Y}$ can be expressed using the $\times$ and $+$ operations and the $\operatorname{if}_{\mathbf{X}}$ function:

$$\text{if } b(x) \text{ then } f_1(x) \text{ else } f_0(x) = (\nabla \circ (f_1 + f_0) \circ \operatorname{if}_{\mathbf{X}} \circ (b \times \mathrm{id}) \circ \Delta)(x),$$

where $\Delta(x) := (x,x)$ is the diagonal mapping and $\nabla\colon \mathbf{Y}+\mathbf{Y} \to \mathbf{Y}$ is the backwards diagonal that returns y on both of the inputs $\operatorname{inl} y$ and $\operatorname{inr} y$. Replacing the functions b, f_0 and f_1 by multifunctions is what we use as semantics for multivalued branching. The use of a sum reflects that only one of the if-statement branches should be evaluated. That is: if $b(x) = \{\text{true}\}$ the eligible return-values are $f_1(x)$ even if $f_0(x)$ is empty, but if $b(x) = \{\text{true}, \text{false}\}$ the eligible return-values of the if-statement are empty if either of $f_0(x)$ and $f_1(x)$ is empty and $f_0(x) \cup f_1(x)$ otherwise. This is the behaviour one would expect from combining realizers.

4 Representations for computation on the reals

The represented space $\mathbb{R}_{\mathbb{Q}}$ from Section 2 is widely considered to provide the “correct” computability structure on the reals and is sometimes even used as a benchmark representation in works that reason about complexity in computable analysis. It provides an easy to understand question and answer structure that gives concrete meaning to the realizers. For the sake of automatically obtaining efficient algorithms carrying out a large number of arithmetic operations, on the other hand, other representations are superior. Such efficient representations should clearly reproduce the computability structure of $\mathbb{R}_{\mathbb{Q}}$.

Our goal is to provide a framework to define operations on real numbers without explicitly referring to implementation details while still allowing to replace the representations used and take advantage of some of their properties for improved performance. We therefore specify a set of operations that are convenient as building-blocks for higher-level operations. This can be seen as a computational axiomatization of the real numbers. Working relative to such an axiomatization allows to recompile the same algorithms for a new representation once these building-blocks, i.e. the axioms have been instantiated natively. Programs obtained this way can take better advantage of the details of the new representation than programs that just translate back and forth.

Other formal developments of real numbers such as the C-CoRn library use a constructive axiomatization. As the setting of our and prior work on real computation is fairly different, we chose to not directly reuse any of the constructive axiomatizations that can be found in the literature [11]. Instead we used work from computable analysis such as [8, 10] and efficient non-verified software packages like iRRAM and AERN as guideline for choosing appropriate basic operations. We ended up requiring the following to be implemented:

- Arithmetic operations (addition, multiplication, subtraction and division),
- The efficient limit $\lim_{\text{eff}} : \subseteq \mathbb{R}^\omega \to \mathbb{R}$, that maps any sequence $(x_i) \in \mathbb{R}^\omega$ that is **efficiently Cauchy**, i.e. such that for all i and j, $|x_i - x_j| \leq 2^{-i} + 2^{-j}$, to its limit $\lim(x_i)$.
- The function $\text{FtoR} : \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$, $(m, e) \mapsto m \cdot 2^{-e}$ that embeds the dyadic rational numbers, or arbitrary-precision floating-point numbers, into $\mathbb{R}$.
- Rational approximation $\text{approx} \colon \mathbb{R} \times \mathbb{Q} \rightrightarrows \mathbb{Q}$, where $\text{approx}(x, \varepsilon) := \{q \mid |x - q| \leq \varepsilon\}$.
- The Kleenean comparison function $<_{\mathbb{K}}$ of type $\mathbb{R} \times \mathbb{R} \to \mathbb{K}$, defined from the Boolean comparison $<$ on the reals by

$$x <_{\mathbb{K}} y := \begin{cases} (x < y)_{\mathbb{K}} & \text{if } x \neq y \\ \bot_{\mathbb{K}} & \text{otherwise.} \end{cases}$$

- A clean-up function that realizes the identity function $\text{id} \colon \mathbb{R} \to \mathbb{R}$. This function can always be instantiated with the identity function on the corresponding name spaces but in concrete cases it can be very useful as an optional performance enhancer that translates names to simpler names for the same object.

An equivalent formulation of the fourth item requires the availability of a translation to the rational representation from Example 1. The second and third item together are sufficient to define a translation in the other direction, so that any representation for which the above are instantiated is equivalent to the rational representation.

For the space $\mathbb{R}_{\mathbb{Q}}$ from Example 1 we instantiated the above basic operations straightforwardly. This does not lead to satisfactory performance and there are several reasons for the inefficiency; one of these we addressed by providing a clean-up function: Iterated multiplication of rational numbers leads to huge numerators and denominators and this is exactly what happens if realizers are implemented using multiplication of rationals and then composed in a naive way. Efficiency can be recovered by replacing exact operations on rational numbers by rounded operations. Note that the direct use of rounded rational operations in the implementation of arithmetic operations would undermine the main advantage of the rational representation, namely, that the approximations have a nice mathematical structure. Instead, we round a rational name only when the clean-up operation is explicitly called.

4.1 The interval reals and their arithmetic operations

There are more problems with the rational representation that make it difficult to optimize in applications. Approximations to the same real number may be required by different parts of an algorithm with different precision leading to extensive re-evaluation and the backwards propagation of errors requires building computation trees and results in blowup of time and space consumption if not done carefully. While these problems are in principle solvable, we decided to mostly use the rational representation for handling input and output and to translate to a representation based on sequences of intervals with dyadic endpoints [21]. Such a representation is commonly used in software packages for exact real arithmetic such as iRRAM [22]. The developers of C-CoRn made a similar switch in a fully constructive setting, also for performance reasons [12].

The **dyadic** numbers are the rational numbers of the form $\frac{z}{2^n}$ for some $z \in \mathbb{Z}$ and $n \in \mathbb{N}$. Let $\mathbb{ID}$ be the set of all closed intervals with dyadic endpoints together with the infinite interval $I_\infty := (-\infty, \infty)$. For an interval $I \in \mathbb{ID}$ let $|I|$ denote the diameter of I, i.e. $|[a,b]| := b - a$ and $|I_\infty| = \infty$. To define the represented space $\mathbb{R}_{\mathbb{ID}}$ of Interval reals use $\mathbf{Q}_{\mathbb{R}_{\mathbb{ID}}} := \mathbb{N}$, $\mathbf{A}_{\mathbb{R}_{\mathbb{ID}}} := \mathbb{ID}$ and the representation $\delta_{\mathbb{R}_{\mathbb{ID}}} : \subseteq \mathcal{B}_{\mathbb{R}_{\mathbb{ID}}} \to \mathbb{R}$ uniquely specified by

$$\delta_{\mathbb{R}_{\mathbb{ID}}}(I_n) = x \qquad \iff \qquad x \in \bigcap_{n\in\mathbb{N}} I_n \text{ and } \lim_{n\to\infty} |I_n| = 0.$$

That is, a sequence of intervals $(I_n)_{n\in\mathbb{N}}$ is a name for $x \in \mathbb{R}_{\mathbb{ID}}$ if x is contained in each interval and the diameter of the intervals approaches zero when n goes to infinity. In particular $\bigcap_{n\in\mathbb{N}} I_n = \{x\}$ and we call the interval with index n the n-th approximation of x. We do not require the diameter to decrease monotonically as this would complicate operations and deteriorate performance.

In the formal development we made use of an existing formalization of interval arithmetic in Coq known as the Coq-interval library [20]. The library provides interval versions for many standard functions and in particular for arithmetic operations. For example, for any two intervals $I, J \in \mathbb{ID}$ and any precision $n \in \mathbb{N}$, the Coq-interval function add returns an interval add(n,I,J) such that for all real numbers x, y with $x \in I$ and $y \in J$, $x+y \in$ add(n,I,J). The new endpoints are obtained by using arbitrary-precision floating-point operations with different rounding modes to compute the upper and lower interval bounds. The parameter n determines the bits used for the mantissa.

Using the Coq-interval functions, realizers of the arithmetic operations can be defined in a pointwise manner. The realizer for addition is e.g. defined as the function that maps $(I_n)_{n\in\mathbb{N}}, (J_n)_{n\in\mathbb{N}}$ and a question $n \in \mathbb{N}$ to add(n, I_n, J_n). Here, and in other realizer definitions we round the n-th approximation to n mantissa digits to make the computational effort for different arithmetic operations on approximations with identical indices comparable. That these realizers return sequences of intervals each containing the correct result can be concluded from the inclusion property of the interval operations already proven in the interval library. Showing that the produced interval sequence converges requires bounds on the diameters that are not included in Coq-interval as they are not of particular interest for interval computation. We derive the error bounds from the theorems for the basic multiple precision floating-point operations from the Flocq library [7]. These operations use relative error bounds and we need bounds on the absolute error, which makes the proofs more complicated than one might first expect. The bounds usually depend not only on the diameter of the intervals but also on the values of the end-points.

4.2 The efficient limit operator and name cleanup

As compared to $\mathbb{R}_{\mathbb{Q}}$, an implementation of the limit operator on $\mathbb{R}_{\mathbb{ID}}$ is more complicated. Recall that one has to transform a name of some efficiently Cauchy sequence $(x_j) \subseteq \mathbb{R}$ to a name of its limit x. That is, given a sequence of sequences of intervals $(I_{i,j})_{i,j\in\mathbb{N}}$ such that for each $j \in \mathbb{N}$, x_j is contained in each $I_{i,j}$ and $|I_{i,j}| \to 0$ for $i \to \infty$ the goal is to return a sequence $(J_i)_{i\in\mathbb{N}}$ such that $x \in J_i$ for all i and $|J_i| \to 0$ for $i \to \infty$. The double-sequence $(I_{i,j})$ can be thought of as an infinite matrix where each column contains a name, and intuitively it should be possible to find a name of the limit by traversing this matrix diagonally. However, it is at least necessary to slightly enlarge each interval to ensure that the limit is contained. But still after that, naively using the diagonal does not guarantee convergence. There are several strategies to search through the intervals and extract a name of the limit. In our implementation, we use a simple strategy known as *vertical search*. To get the n-th approximation, we choose the $(n+1)$-st element of the sequence, do an unbounded search for an interval of size less than $2^{-(n+1)}$ and extend it by $2^{-(n+1)}$. An advantage of this strategy is that it returns names with quickly converging intervals, resetting any precision loss incurred in other operations.

On concrete examples one quickly notices that computing a limit at low precisions tends to return useless results and yet takes a long time. This is because iterated use of arithmetic operations leads to intervals with large diameter and endpoints with big integer parts. We avoid this using the heuristic that the diameter of an interval should never be bigger than $1/2$ so that at least the integer part of intermediate results is correct. This can be forced using a clean-up function that replaces any interval whose diameter is too large with the infinite interval. As the interval operations barely do any computation if one of the input intervals is infinite, this leads to a considerable speedup at low precision. Another cause for performance issues is the functional nature of names: Function values are not cached automatically leading to extensive reevaluation. As the questions are natural numbers in unary, there exists a simple solution: we internally replace the names by elements of a coinductive datatype of streams that are treated as lazy lists in evaluation.

5 A verified parametric square root algorithm

As a case study on how the basic operations can be used to define other operations on real numbers, let us study the example of the square root function in some detail. By the square root function we mean the partial function from reals to reals whose domain is $[0, \infty)$ and whose return value on input of $x \geq 0$ is $\sqrt{x}$. This function is a popular example as it being continuous but not analytic in 0 is a challenge in providing a good algorithm to compute it. We aim to recover this function in a compositional way from the basic operations listed in Section 4. A computable realizer for the square root function can then be extracted almost automatically by composing the realizers of the relevant basic operations independently from the exact implementation of the data-type of real numbers.

5.1 Square root approximation using Heron's method

A well known and efficient way to approximate the square root of a real number x is the Heron iteration inductively defined by $x_0 := 1$ and $x_{i+1} = \frac{1}{2}\left(x_i + \frac{x}{x_i}\right)$. Let the function heron: $\mathbb{R} \to \mathbb{R}^\omega$ be defined by $\mathrm{heron}(x)_i := x_{\lceil \log_2 i \rceil}$. This function can be defined from our basic operations and returns an efficiently convergent sequence:

▶ **Lemma 5.** $|\mathrm{heron}(x)_i - \sqrt{x}| \leq 2^{-i}$, *whenever* $\frac{1}{4} \leq x \leq 2$.

Proof Sketch. It is well-known that (x_i) converges quadratically to $\sqrt{x}$ in the above interval. This means $|x_i - \sqrt{x}| \leq 2^{-2^i}$ and thus heron returns an efficiently convergent sequence. ◀

Thus, the square root of some $x \in [\frac{1}{4}, 2]$ can be approximated using heron and the efficient limit. We aim to extend the scope of this algorithm from the bounded interval $[\frac{1}{4}, 2]$ to all of $[0, \infty)$. Our strategy is to handle 0 as a special case and scale strictly positive numbers to end up in the interval, apply the method above and then rescale the result appropriately. The following Lemma follows directly from Lemma 5.

▶ **Lemma 6.** *Let* $p \in \mathbb{Z}$ *such that* $4^{-p}x \in [\frac{1}{4}, 2]$*, then* $|2^p\mathrm{heron}(4^{-p}x)_{i+p} - \sqrt{x}| \leq 2^{-i}$.

Let us call such a p a **scale** for x. Note that a scale exists if and only if $x > 0$ and there always exists more than one possible choice in that case. Since $\mathbb{Z}$ is discrete and $\mathbb{R}$ connected, the semantics of an algorithm extracting an appropriate p from x are necessarily multivalued.

The treatment of the special case 0 requires branching. If $x \leq 2^{-2i}$ then 0 is already an approximation of the square root with error at most 2^{-i}. Boolean-valued comparisons on the reals are discontinuous and therefore not computable. We may only use the Kleenean

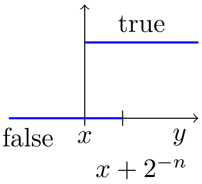

Figure 1 $\mathrm{sc}(n, x, y)$ plotted over y for fixed x and n.

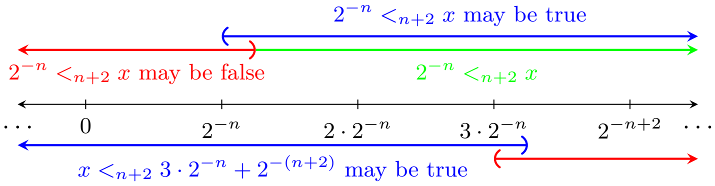
$2^{-n} <_{n+2} x$ may be true
$2^{-n} <_{n+2} x$ may be false
$2^{-n} <_{n+2} x$
$\cdots$
0
2^{-n}
$2 \cdot 2^{-n}$
$3 \cdot 2^{-n}$
2^{-n+2}
$\cdots$
$x <_{n+2} 3 \cdot 2^{-n} + 2^{-(n+2)}$ may be true

Figure 2 $-n \in \mathrm{mag}(x)$ if both inequality tests may be true. For $x \in (0, 1)$ there is a number $n \in \mathbb{N}$ such that both tests have true as the only valid value.

comparison $\mathbb{R} \times \mathbb{R} \to \mathbb{K}$ that we included in our basic operations. Luckily, we do not need exact comparison but only need to know if either $x > 0$ or $x \leq 2^{-2i}$ and such a test can be implemented from the Kleenean comparison. To disregard the controlled divergence and define a total function in the end, we also go through multivaluedness.

For each of the previous two paragraphs let us develop some general purpose tools that may also be useful in other applications. For the branching needed around zero we use soft comparisons and for obtaining p we use a multivalued magnitude function, both of which we implement using the basic operations.

5.2 From Kleenean comparisons to soft comparison

Kleenean-valued comparisons are easy to implement but often inconvenient to use for implementation of total functions as they feature explicit divergence. A popular version of real number comparison trades off divergence for multivaluedness and is known as ε-test or soft comparison in numerics. For simplicity, we restrict to ε of the form 2^{-n} and consider the multi-valued soft comparison $sc\colon \mathbb{N} \times \mathbb{R} \times \mathbb{R} \rightrightarrows \mathbb{B}$ specified by

$$\mathrm{true} \in \mathrm{sc}(n, x, y) \Leftrightarrow x < y \quad \text{and} \quad \mathrm{false} \in \mathrm{sc}(n, x, y) \Leftrightarrow y < x + 2^{-n}. \tag{1}$$

This multifunction is total and properly multivalued as there is an interval of size 2^{-n} where both cases overlap (see Figure 1).

Multivaluedness makes moving from prefix to infix notation more complicated. We fix the following conventions: $x <_n y$ without any additions means $\mathrm{sc}(n, x, y) = \{\mathrm{true}\}$, for $\mathrm{true} \in \mathrm{sc}(n, x, y)$ we state "$x <_n y$ may be true" and for $\mathrm{false} \in \mathrm{sc}(n, x, y)$ we write "$x <_n y$ may be false". This is illustrated by means of an example at the top of Figure 2. For functions $f_0, f_1 : \mathbb{R} \rightrightarrows \mathbb{R}$ expressions such as "if $x <_n 0$ then $f_1(x)$ else $f_0(x)$" are meaningful as soft comparison is a multivalued predicate and branching works as explained in Section 3.1.

Soft comparison can be implemented using the Kleenean comparison and the `which` function from Example 4. We state this in the next Lemma. We use slightly sloppy notation as we identify the Booleans with a subspace of the Kleeneans, which in turn we consider elements of Sierpinski space by identifying $\mathrm{true}_{\mathbb{K}}$ with $\top_{\mathbb{S}}$ and everything else with $\bot_{\mathbb{S}}$.

▶ **Lemma 7.** *Soft comparison can be expressed from the* `which` *function and* $<_{\mathbb{K}}$ *via*

$$\mathrm{sc}(n, x, y) = \mathtt{which}(x <_{\mathbb{K}} y, y <_{\mathbb{K}} x + 2^{-n}).$$

Proof. No matter x and y we always have either $x <_{\mathbb{K}} y = \mathrm{true}_{\mathbb{K}}$ or $y <_{\mathbb{K}} x + 2^{-n} = \mathrm{true}_{\mathbb{K}}$. Thus the `which` function on the above input always returns a Boolean. Further, true is a valid return value if and only if $x < y$ and false is a valid return value if and only if $y < x+2^{-n}$. ◀

5.3 The magnitude function for scaling

Recall that for computation of the value of the square root of a strictly positive real via rescaling, we needed to find an integer p such that $x2^p$ is from a bounded interval and that such an integer cannot be found algorithmically without introducing multivaluedness. We thus implement the multifunction $\operatorname{mag}\colon \mathbb{R} \rightrightarrows \mathbb{Z}$ that extracts the magnitude of x in the sense that $z \in \operatorname{mag}(x) \Leftrightarrow 2^z < x < 2^{z+2}$. Such a z exists whenever $x > 0$, i.e., the domain of magnitude are the positive real numbers.

Let us first argue that we may restrict to the case that $0 < x < 1$.

▶ **Lemma 8.** *The function* mag *can be recovered from its restriction to* $(0,1)$ *as*

$$\operatorname{mag}(x) = \textit{if } x <_1 2 \textit{ then } \operatorname{mag}|_{(0,1)}(x/2) + 1 \textit{ else } - \operatorname{mag}|_{(0,1)}(1/x) - 2.$$

Proof. In the first case $x < 2$ and therefore $x/2 \in (0,1)$. In the second case $x > 3/2$ and therefore $1/x \in (0,1)$. That the bounds are correct can be checked easily. ◀

Thus, assume $0 < x < 1$ in the following.

▶ **Lemma 9.** *There always exists an* $n \in \mathbb{N}$ *such that* $2^{-n} <_{n+2} x$ *and* $x <_{n+2} 3{\cdot}2^{-n}{+}2^{-(n+2)}$*, i.e.* true *is the only possible value for both conditions. Moreover, for any* $n \in \mathbb{N}$

$$2^{-n} <_{n+2} x \textit{ may be } \mathrm{true} \wedge x <_{n+2} 3 \cdot 2^{-n} + 2^{-(n+2)} \textit{ may be } \mathrm{true} \quad \Longrightarrow \quad -n \in \operatorname{mag}(x).$$

Proof Sketch. $2^{-n} + 2^{-(n+2)} \leq x$ implies $2^{-n} <_{n+2} x$ and $x \leq 3 \cdot 2^{-n}$ implies $x <_{n+2} 3{\cdot}2^{-n}{+}2^{-(n+2)}$ (see Figure 2) and both these inequalities hold for instance for $n = -\lceil \log_2(\frac{x}{3}) \rceil$. Further, the two soft comparisons may only be true if $2^{-n} < x < 3 \cdot 2^{-n} + 2^{-(n+2)} < 2^{-n+2}$ and thus $-n \in \operatorname{mag}(x)$. ◀

In particular, a linear search for the first n such that the equation holds implemented using the realizers for the soft-comparison will always terminate and give a realizer for $\operatorname{mag}|_{(0,1)}$, which in turn can be used to implement the full magnitude function via multivalued branching.

5.4 Defining the square root function

For any $x > 0$ and $m \in \operatorname{mag}(x)$, $\lceil \frac{m+1}{2} \rceil$ is a scale for x in the sense of Lemma 6. Thus, we finally have all tools necessary to define the square root function. We define an approximation function $\operatorname{sqapprox}\colon \mathbb{R} \rightrightarrows \mathbb{R}^\omega$ with domain $[0,\infty)$ by

$$\operatorname{sqapprox}(x)_i := \text{if } x <_{2i+1} 2^{-2i} \text{ then } 0 \text{ else } 2^p \operatorname{heron}(4^{-p}x)_{i+p} \text{ where } p \text{ is a scale for } x.$$

Correctness is given by the following Lemma.

▶ **Lemma 10.** $\lim_{\mathrm{eff}} \circ \operatorname{sqapprox}$ *tightens the square root function.*

Proof. It suffices to show that for each $i \in \mathbb{N}$, $\operatorname{sqapprox}(x)_i$ is a 2^{-i} approximation of $\sqrt{x}$. If $x <_{2i+1} 2^{-2i}$ may be true then $x < 2^{-2i}$ and 0 is a 2^{-i} approximation of $\sqrt{x}$. If $x <_{2i+1} 2^{-2i}$ may be false then $2^{-(2i+1)} \leq x$ and thus $x \in \operatorname{dom}(\operatorname{mag})$ and we can apply Lemma 6. ◀

This means that any realizer for $\lim_{\mathrm{eff}} \circ \operatorname{sqapprox}$ is also a realizer for the square root function and can thus be defined only by using realizers for basic operations.

6 Implementation

All results of this paper have been formally verified in the Coq proof assistant. The implementation is part of the Incone library. It is in the development branch and will be featured in a future release. An overview of the library and instructions on how to get started can be found in [24]. The content of the current work can be found in a folder for examples about real numbers in the development branch of the library [23]. The real number structure from Section 4, the treatment of interval reals and the interval representation are each given their own files in that folder. The error bound estimates for operations from the Coq-interval library needed for the interval representation have been exported to a separate file so that the file sizes remain manageable. The content of Section 5 is separated into a file for the soft comparison, one for the magnitude function and finally one where the square root function is implemented. The finite spaces from Section 3 have been integrated into Incone and can be found in the folder for constructions on continuity spaces under the name “hyperspaces”.

Our development uses a fairly small set of axioms, namely those used in the axiomatic formalization of the reals, the law of excluded middle, functional extensionality and some choice principles. The reasoning is usually divided into two parts, where the first is coming by with mathematics and the second part is to define realizers and prove them correct. We carefully define the realizers such that they do not rely on the non-constructive axioms of the reals and actually correspond to executable programs.

As a concrete example, let us consider some parts of the formalization of the square root algorithm from Section 5. While the more difficult part and most of the content of the sqrt.v file constitutes the extension to the whole real line, for simplicity we here only consider the restriction to the interval $[\frac{1}{4}, 2]$ where the Heron method converges quadratically.

The Coq standard library already defines a function sqrt for the square root built on the axiomatization of the reals and proves some of its properties. Assume we have fixed some representation of the reals and denote the spaces of questions and answers by $\mathbf{Q}$ and $\mathbf{A}$, respectively. An algorithm implementing the square root function thus takes and returns elements of $\mathbf{A}^{\mathbf{Q}}$. In a first approximation such an algorithm may be represented by a Coq function of type sqrt_rlzr: (Q -> A) -> (Q -> A). For this function to actually correspond to an algorithm, its definition should not involve incomputable axioms. The correctness of the algorithm is guaranteed by a specification Lemma of the form:

```
Lemma sqrt_rlzr_spec: sqrt_rlzr \realizes sqrt.
```

The notation \realizes is part of the Incone library and means that sqrt_rlzr(φ) is name of sqrt(x) whenever φ is a name of x. Incone defines several such notations making the formal statements look very similar to the informal mathematical statements.

Unfortunately, the situation is usually more complicated as the realizer may need to be partial. Some algorithm can diverge if the input is not a valid name of a real number. In Coq, all functions are total but partial functions can be modeled using dependent types. A partial function takes as input a pair consisting of the actual input and a proof that this input is contained in its domain. Note that Coq's sqrt function itself is not partial. For the restriction, as the realized function does not carry computational information, we take a different approach to partiality here and instead of using the dependent type system we move to a relational specification right away. That is, we replace the function sqrt by its induced multifunction F2MF sqrt which can be restricted by adding a domain condition. Finally we may bundle the realizing function with its correctness proof so that result to be found in Incone actually takes the following form:

```
Lemma sqrt_rlzr_exists :
          {f : partial_function | f \solves (F2MF sqrt)|_[/4,2]}.
```

The partial function itself can be retrieved by (sval sqrt_rlzr_exists) and its correctness proof by (svalP sqrt_rlzr_exists). The function definition can be extracted to Haskell code and then be executed. The terms usually fail to reduce internally due to the use of non-computational real numbers in the specification part that entangled with the definitional part through the use of partial functions and sigma types.

For broad applicability we do not only work with a specific representation but define a structure of computable_reals that can be instantiated with different representations. This structure closely resembles the informal description in Section 4 and serves as an intermediate level for real number operations. It contains a representation for the reals and partial functions realizing the basic operations together with correctness proofs. Other operations can be defined by composition, product, sums and branching on the basic operations. For instance, the square root function in our implementation has a parameter Rc of type computable_reals and returns for each instance of this structure an executable program. The actual algorithm is based on Heron's method and closely follows the outline in Section 5. Let us list the definition and specification Lemma of the sqrt_approx function from the paper as an example:

```
Fixpoint sqrt_approx x0 n x :=
  match n with
  | 0 => x0
  | S n' => let x' := sqrt_approx x0 n' x in (x' + x / x') / 2
  end.

Lemma sqrt_approx_correct x n:
       /4 <= x <= 2 -> Rabs (sqrt_approx 1 n x - sqrt x) <= /2^(2^n).
```

The Incone library equips the space of infinite sequences with the structure of a represented space again. An algorithm to find the limit of an efficiently convergent sequence as operation lim_eff: Rc$^\omega$ -> Rc is part of the computable_reals structure. As outlined in Section 5 one can use the iteration above to obtain a function heron: Rc -> Rc$^\omega$ that returns an efficiently convergent sequence for certain real numbers. The composition of heron with the limit operator returns the sqrt function:

```
Lemma sqrt_as_lim :
   (lim_eff \o heron) \tightens (F2MF sqrt)|_[/4,2]
```

Once this specification is available, a realizer of the right hand side can be obtained from the realizers of the operations on the left hand side by compositionality. The realizer of the heron function is obtained by piecing together the realizers for the arithmetic operations.

6.1 Executability and code extraction

As the definitions of the domains of the partial functions we use as realizers involve the non-computational real numbers, the corresponding functions can not be executed Coq internally using term reduction. These problems can be worked around for instance by using a fuel-based approach [18], but in our current implementation this method leads to considerably worse running times and we therefore refrain from giving details here. An additional drawback is that extra information about the realizers of the basic operations is needed and additional work is necessary for propagating this information through operations such as composition and taking fix points.

As the main purpose of our implementation is not to do computation inside Coq but to provide an easy to use interface for developers in exact real computation frameworks to define and verify their algorithms, we focus on using Coq's code extraction features to generate efficient code instead of direct execution inside of Coq. We hope that the extracted programs can be integrated into other developments for parts where particularly strong correctness guarantees are needed.

Coq's code extraction feature can be used to generate executable programs. However, the performance of the extracted programs depends on how the extraction is done exactly. For instance, if the basic operations on integers are translated from their Coq implementation that is targeted towards simple proofs instead of efficiency, the performance will suffer. It is possible to instruct Coq to extract these operations to native implementations in Haskell instead. We have extracted all arithmetic operations and comparisons on integer types such as `Z`, `nat` or `positive` to the corresponding operations on arbitrary-sized integers in Haskell. Some other operations such as shifting and taking integer logarithms that turned out to be particularly slow were also replaced by more efficient implementations available in Haskell. Of course, these Haskell operations are not formally verified and each modification increases the size of the trusted core and the risk for errors in the final program. As the set of operations we trust for the extraction is quite small, we believe this risk to be manageable.

The replacement of functions by streams discussed in Section 4.2 can either be done directly in Coq or in Haskell by adding some instructions for the extraction. The replacement in Haskell performed slightly better in experiments and we used it as the default option.

7 Conclusion and Future work

On paper the approach of computable analysis is very much in accordance with the spirit of Coq. In principle it should be possible to parameterize the theories over an abstract type so that the classical treatment of real numbers and similar structures is hidden in the propositional layer. In practice there are a lot of additional hurdles in maintaining a clean mathematical presentation, executability and reuseability of existing work. Much of the existent infrastructure for computation on the reals such as the Flocq and Interval libraries are specified against the classical axiomatization of real numbers from the standard library. This axiomatization of the real numbers states classical properties, such as decidability of equality, as global facts and makes maintaining executability challenging.

Currently, we have only implemented a few basic operations on real numbers, mostly to demonstrate that our framework indeed can be used for efficient computation. Adding further operations such as trigonometric functions should not be too difficult. An interesting direction for future work is to extend the computation on real numbers to operators on real functions such as integration and ODE solving. The tools contained in the Incone library can already be used to automatically generate a representation for real functions from a representation for real numbers. Ideas from real number complexity theory [16, 15] suggest that the use of specialized representations over this generic function representation might yield even better results.

References

1 S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, Oxford, 1994.

2 Reynald Affeldt, Cyril Cohen, and Damien Rouhling. Formalization techniques for asymptotic reasoning in classical analysis. *Journal of Formalized Reasoning*, 11(1):43–76, 2018. doi:10.6092/issn.1972-5787/8124.

3 Andrea Balluchi, Alberto Casagrande, Pieter Collins, Alberto Ferrari, Tiziano Villa, and Alberto L Sangiovanni-Vincentelli. Ariadne: a framework for reachability analysis of hybrid automata. In *In: Proceedings of the International Syposium on Mathematical Theory of Networks and Systems.*, 2006.

4 Merrie Bergmann. *An introduction to many-valued and fuzzy logic: semantics, algebras, and derivation systems.* Cambridge University Press, 2008.

5 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for Coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

6 Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Formalization of real analysis: A survey of proof assistants and libraries. *Mathematical Structures in Computer Science*, 26(7):1196–1233, 2016. URL: http://hal.inria.fr/hal-00806920.

7 Sylvie Boldo and Guillaume Melquiond. Flocq: A unified library for proving floating-point algorithms in coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*, pages 243–252. IEEE, 2011.

8 Vasco Brattka and Peter Hertling. Feasible real random access machines. *J. Complexity*, 14(4):490–526, 1998. doi:10.1006/jcom.1998.0488.

9 Vasco Brattka, Peter Hertling, and Klaus Weihrauch. A Tutorial on Computable Analysis. In S. Barry Cooper, Benedikt Löwe, and Andrea Sorbi, editors, *New Computational Paradigms: Changing Conceptions of What is Computable*, pages 425–491. Springer, 2008.

10 Franz Brauße, Pieter Collins, Johannes Kanig, SunYoung Kim, Michal Konečnỳ, Gyesik Lee, Norbert Müller, Eike Neumann, Sewon Park, Norbert Preining, et al. Semantics, logic, and verification of" exact real computation". *arXiv preprint arXiv:1608.05787*, 2016.

11 Alberto Ciaffaglione and Pietro Di Gianantonio. A certified, corecursive implementation of exact real numbers. *Theoretical Computer Science*, 351(1):39–51, 2006. Real Numbers and Computers. doi:10.1016/j.tcs.2005.09.061.

12 Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-CoRN, the constructive Coq repository at Nijmegen. In *International Conference on Mathematical Knowledge Management*, pages 88–103. Springer, 2004.

13 Martín Hötzel Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162(1):79–115, 1996.

14 Akitoshi Kawamura. *Computational complexity in analysis and geometry.* University of Toronto, 2011.

15 Akitoshi Kawamura, Norbert Th. Müller, Carsten Rösnick, and Martin Ziegler. Computational Benefit of Smoothness. *Journal of Complexity*, 2015. doi:10.1016/j.jco.2015.05.001.

16 Akitoshi Kawamura, Florian Steinberg, and Holger Thies. Parameterized complexity for uniform operators on multidimensional analytic functions and ODE solving. In *International Workshop on Logic, Language, Information, and Computation*, pages 223–236. Springer, 2018.

17 Michal Konecnỳ. AERN-Real: Arbitrary-precision interval arithmetic for approximating exact real numbers, 2008.

18 Michal Konečný, Florian Steinberg, and Holger Thies. Continuous and Monotone Machines. In Javier Esparza and Daniel Kráľ, editors, *45th International Symposium on Mathematical Foundations of Computer Science (MFCS 2020)*, volume 170 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 56:1–56:16, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2020.56.

19 Christoph Kreitz and Klaus Weihrauch. Theory of representations. *Theoretical computer science*, 38:35–53, 1985.

20 Guillaume Melquiond. Proving bounds on real-valued functions with computations. In *International Joint Conference on Automated Reasoning*, pages 2–17. Springer, 2008.

21 R.E. Moore, R.B. Kearfott, and M.J. Cloud. *Introduction to Interval Analysis.* SIAM e-books. Society for Industrial and Applied Mathematics (SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104), 2009.

22 Norbert Th. Müller. The iRRAM: Exact arithmetic in C++. In *Computability and complexity in analysis. 4th international workshop, CCA 2000. Swansea, GB, September 17–19, 2000. Selected papers*, pages 222–252. Berlin: Springer, 2001.

23 Florian Steinberg. The INCONE library. `https://github.com/FlorianSteinberg/incone`, 2019. release v1.0.

24 Florian Steinberg, Laurent Thery, and Holger Thies. Computable analysis and notions of continuity in coq. *arXiv preprint arXiv:1904.13203*, 2019.

25 Klaus Weihrauch. *Computable Analysis.* Springer, Berlin/Heidelberg, 2000.

A Experimental results

To show that our implementation indeed gives a feasible implementation of exact real computation we did a small experimental study where we compared the running times to approximate some simple functions using our implementation to an implementation in the C-CoRn library and non-verified implementations using exact real arithmetic packages. While the experiments show that our implementation is not yet optimal, the difference in running time was only by a small factor and we think that it could be further reduced by optimizing our representation.

For all experiments we extracted Haskell code from the specification in COQ (version 8.9.0) using the code extraction mechanism. Apart from the simple optimizations for the code extraction mentioned above we did not do any additional changes to increase performance. In particular we did not change the extracted code except for adding a few includes of standard Haskell libraries in the beginning of the file. The Haskell code was compiled with GHC version 8.8.1 and profiling options turned on. The running times were taken from the total time written in the Time and Allocation Profiling Report generated by Haskell. All experiments were done on a Macbook Pro 2015 model with 16 GB RAM and 2.2 GHz Intel Core i7 processor. We tried the experiments with both the rational and interval representation for real numbers, however as expected the (non-optimized) rational representation performed very poorly and we thus focus on the results for the interval representation.

We also compared the running time to computing the same problem with the C-CoRn library (using the same code extraction techniques) and a (non-verified) C++ implementation using the IRRAM framework. These comparisons have to be taken with a grain of salt as many details of the implementations differ. For instance, our implementation outputs the result as a rational number giving numerator and denominator while IRRAM and C-CoRn output decimal approximations.

The first experiment is to compute iterations of the logistic map $x_{n+1} = rx_n(1 - x_n)$ for $x_0 = 0.5$ and $r = 3.75$. The logistic map is often used as a benchmark problem in exact real computation as it exhibits chaotic behavior, i.e., a slight change in the initial condition leads to completely different values at later iterations. In particular, computations using standard floating-point methods quickly diverge from the correct solution. While it may be argued that computing the exact values is not of any practical relevance, it is a popular example for where floating point computations fail completely, while exact methods can quickly produce correct results.

In this experiment we output an approximation of the result after several iterations of the logistic map with error less than 10^{-1000} (i.e. approximately 1000 decimal digits). In our experiments our implementation performed quite well (see Table 3a) and was only a

N	Incone	iRRAM
100	0.02	0
500	0.1	0.01
1000	0.18	0.01
5000	0.94	0.27
10000	3.03	0.83
20000	12.02	4.32
50000	67.23	38.4

(a) Approximating 1000 digits of the N-th iteration of the logistic map.

n	$\sqrt{5}$	$\sqrt{\frac{5}{32}}$
10	0.12	0.08
100	0.33	0.23
500	1.05	0.64
1000	1.78	1.12
2000	2.74	1.76
5000	6.33	3.61
10000	8.51	5.13
50000	18.02	10.33
100000	27.93	15.98
500000	91.1	48.72

(b) Computing n digits of the square root.

Figure 3 Running times (seconds) for the different experiments.

factor $2 - 5$ slower than the iRRAM implementation. A straightforward implementation in C-CoRn did not give good results as evaluating x_n twice in the iteration rule leads to exponential growth and therefore already computing more than a few iterations takes a very long time. However, this is probably just due to our naive implementation and it might be possible to do a more clever implementation in C-CoRn that caches the intermediate values.

Our second experiment was to compute some square roots, i.e., compute the square root of a given rational number and output an approximation with a certain error bound. We give the results for $\sqrt{5}$ and $\sqrt{\frac{5}{32}}$ as representatives for numbers that are scaled down resp. up in our algorithm. Other numbers performed mostly similarly, however as computing the magnitude uses a linear search for very large numbers the running time gets significantly worse. Here, while our algorithm is still usable, its performance was far worse than both the iRRAM and C-CoRn versions. For example iRRAM could still compute 500000 digits in less than 0.01 seconds. The C-CoRn version was nearly as fast as the iRRAM version for up to 10000 digits. For higher precision it got significantly slower and for 500000 digits even performed worse than our implementation. The performance log shows that this is not a bug in C-CoRn but due to some integer operation being extracted to a sub-optimal implementation. As C-CoRn is made for execution inside of Coq and not optimized for Haskell code extraction, it is quite hard to compare these numbers.

Our implementation has similar issues when using the interval library. The Coq interval library is built for fast execution inside of Coq, however that makes the extracted code quite complicated and many operations could be implemented much more efficiently in Haskell. Moving to a simpler implementation of interval arithmetic should therefore lead to a drastic improvement.

As the performance hugely depends on factors that have mostly to do with code extraction, it is questionable how valuable a thorough performance comparison of the different frameworks is. We think the main take-away message from this experimental study should be that while possibly not as fast as some of the alternatives, our simple implementation still performs reasonably well and can be used to compute approximations up to very high precision.