SEMANTICS, SPECIFICATION LOGIC, AND HOARE LOGIC OF EXACT REAL COMPUTATION

SEWON PARK, FRANZ BRAUSSE, PIETER COLLINS, SUNYOUNG KIM, MICHAL KONEČNÝ, GYESIK LEE, NORBERT MÜLLER, EIKE NEUMANN, NORBERT PREINING, AND MARTIN ZIEGLER

School of Computing, KAIST, Daejeon, Republic of Korea
e-mail address: swelite@kaist.ac.kr

University of Manchester
e-mail address: brausse@informatik.uni-trier.de

Maastricht University, Maastricht, The Netherlands
e-mail address: pieter.collins@maastrichtuniversity.nl

Yonsei University, Seoul, Republic of Korea
e-mail address: sy831@yonsei.ac.kr

Aston University, Birmingham, UK
e-mail address: m.konecny@aston.ac.uk

Hankyong National University, Ansan, Anseong, Republic of Korea
e-mail address: gslee@hknu.ac.kr

Universität Trier, Trier, Germany
e-mail address: mueller@uni-trier.de

Swansea University, Swansea, UK
e-mail address: neumaef1@gmail.com

Fujitsu Ltd., Japan
e-mail address: norbert@preining.info

KAIST, Daejeon, Republic of Korea
e-mail address: ziegler@kaist.ac.kr

Key words and phrases: Computable Analysis, Reliable Numerics, Imperative Programming, Turing-Completeness, Decidability, Hoare Logic, Kleene Logic.
ABSTRACT. We propose a simple imperative programming language, ERC, that features arbitrary real numbers as primitive data type, exactly. Equipped with a denotational semantics, ERC provides a formal programming language-theoretic foundation to the algorithmic processing of real numbers. In order to capture multi-valuedness, which is well-known essential to real number computation, we use a carefully designed Plotkin powerdomain. This makes our programming language semantics computable and complete: all and only real functions computable in computable analysis can be realized in ERC. The base programming language supports real arithmetic as well as implicit limits; expansions support additional primitive operations (such as a user-defined exponential function). By restricting integers to Presburger arithmetic and real coercion to the ‘precision’ embedding \( \mathbb{Z} \ni p \mapsto 2^p \in \mathbb{R} \), we arrive at a first-order theory which we prove to be decidable, model-complete, and expressive for the base programming language. Based on said logic as specification language for preconditions and postconditions, we extend Hoare logic to a sound (w.r.t. the denotational semantics) and expressive system for deriving correct total correctness predicates. Various examples demonstrate the practicality and convenience of our language and proof rules.

CONTENTS

1. Introduction 3
   1.1. Kleenean-valued Comparison and Nondeterminism 4
   1.2. Main Contributions and Overviews 5
2. Preliminaries 7
   2.1. Representations and Computable Partial Functions 8
   2.2. Computable (Infinite) Multi-functions 9
   2.3. Computable (Finite) Multi-functions and Plotkin Powerdomain 10
3. Formal Syntax and Typing Rules of Exact Real Computation 12
   3.1. Syntax 12
   3.2. Typing Rules 14
4. Denotational Semantics of Exact Real Computation 16
   4.1. Denotations of Terms 16
   4.2. Denotations of Commands 18
   4.3. Denotations of Programs 20
5. Programming in Exact Real Computation 21
   5.1. Programming Abbreviations 22
   5.2. Example Programs 23
   5.3. Turing-Completeness 30
6. Logic of Exact Real Computation 32
   6.1. Three-Sorted Structure with Decidable Theory 32
   6.2. Specification Language 35
   6.3. Hoare Logic 37
7. Example Formal Verification in Exact Real Computation 39
8. Conclusion and Future work 40
Acknowledgement 41
References 41
Appendix A. Proof of the Soundness of the Hoare Logic of ERC 44
1. Introduction

In computable analysis, real numbers are expressed and computed over exactly by using infinite representations [Wei00, PER89]. For example, a real number $x$ is represented by an infinite sequence of integers $x_n$ such that $|x - x_n \cdot 2^{-n}| \leq 2^{-n}$ holds for all $n$. That means, when we have correctly computed a real number $x$ we automatically get access to the real number $x$ via any arbitrarily accurate precision, making this approach well-suited for domains where high precision numerical results are required.

There are several implementations demonstrating the practicality of this approach for exact real number computation: AERN [KTD+13], Ariadne [CGC+20], Core2 [YYD+10], Cdar [Bla], iRRAM [Mül01], and realLib [Lam05]. They provide their own data types of real numbers to their users such that the users can deal with real numbers as they were abstract algebraic entities without being concerned with how the infinite representations are implemented in their back-ends. Together with the property that primitive operations are computed exactly without causing any rounding errors, real numbers that the users observe closely resemble the classical mathematical structure of real numbers. According to this observation, it is expected that reasoning programs’ behaviours in this approach becomes more intuitive as the accuracy of computations becomes not an interesting question anymore; cf. [KMP+08, Mel08, BFM09, BM11].

Imperative programming, a ubiquitous style of expressing computation that is widely adopted in numerical computation software, admits rich theories of precondition-postcondition-based program verification [Hoa69, AO19, Rey02, NMB06] and their applications including [FP13]. Hence, it is a natural desire to extend them with exact real number computation in order to conveniently achieve (provably) correct arbitrary accurate numerical computations.

To make such an extension rigorous, we first need to have a formal programming language on which the extended verification methodologies are based. However, extending an existing formal programming language with exact real number computation, which is a seemingly simple quest, fails when it is done naively with the classical model-theoretic structure of real numbers as it is not identical, though resembles much, to the computational structure of real numbers [Her99, Section 3]. For example, real number comparison $x < y$ is only partially computable causing non-termination when $x = y$ [Wei00, Theorem 4.1.16] resulting nondeterminism for multi-valued functions to become essential [Luc77]. (It is explained in more detail in Section 1.1.)

The aim of this paper is as follows. We formalize exact real number computation as an imperative programming language that correctly deals with the partial and nondeterministic nature. It means that the language’s formal (domain-theoretic denotational) semantics makes our language define precisely the computable partial real functions and integer multi-valued functions in computable analysis. Based on the formal programming language, we devise a specification logic for exact real number computation which is then used in our extended Hoare logic which is sound with regard to the denotational semantics. The language is rich enough to model core fragments of existing software and simple enough to stand as a design guideline for future exact real number computation software developments which wish to adopt the theory developed in this paper. We develop our language and theory relative to primitive operator extensions such that other frameworks that provide richer sets of primitive operators can still rely on our theoretical language and analyze their real number computations.
In terms of formalizing exact real number computation, this work is deeply influenced by the following related works. Real PCF [Esc96] is a pure functional language that is an extension of PCF [Plo77] with real numbers, computable real number operations, and a parallel conditional construct for nondeterminism. Its application in expressing integration demonstrated usefulness of formalizing exact real number computation [EE00]. In [MRE07], a domain-theoretic denotational semantics of a variant language is defined but using the Hoare powerdomain which cannot express termination. There, it is observed that nondeterminism in computable analysis becomes continuous only with the Hoare powerdomain. In this paper, we avoid the continuity problem and use the Plotkin powerdomain such that we do not need to have a separate tool for reasoning programs’ termination. (Later in Remark 2.6, we discuss about using different powerdomains.)

Speaking of imperative languages, Feasible real RAM [BH98] is a computable variant of real RAM where the infeasible total real comparison test [Bra03] is replaced with a nondeterministic approximation to it. Its operational semantics influenced the design of [Mül01]. Amongst further related works including [BC88, DGE13, ES14], WhileCC [TZ04], a simple imperative language extended with the computational structure of real numbers and a (countable) nondeterminism construct, which is suggested with its algebraic operational semantics, is close to our design. We proceed from there, suggest a different set of primitives, and further devise domain-theoretic computable denotational semantics, specification logic, and reasoning principles.

1.1. Kleenean-valued Comparison and Nondeterminism. As an inevitable side-effect of using infinite representations, the ordinary order relation of real numbers as a Boolean-valued function is only partially computable. More specifically, whichever infinite representation of reals underlies, the comparison test of real numbers \( x < y \) does not terminate when \( x = y \) if we want it to be correct at least in elsewhere [Wei00, Theorem 4.1.16]. That means dealing with possibly non-terminating Boolean expressions becomes essential, to the extent some frameworks, including AERN and iRRAM, introduce the concept of Kleenean, a lazy extension of Boolean, in their core design. (See [KST20b, Example 3] for an example formulation.) A variable of type Kleenean can store a non-terminating Boolean expression safely, deferring the non-termination to the moment when it is eagerly required to check if the stored value is true or false. Denoting the non-termination as unknown (standing for the third truth-value in Kleene three-value logic), an admissible value that variables can store, Kleenean-valued comparison test \( \preceq : \mathbb{R} \times \mathbb{R} \to \mathbb{K} \) \((\mathbb{K} := \{true, false, unknown\})\) defined by

\[
x < y := \begin{cases} 
true & \text{if } x < y, \\
false & \text{if } y < x, \\
unknown & \text{otherwise if } x = y,
\end{cases}
\]

replaces the partially computable ordinary Boolean-valued comparison test.

Due to the non-termination in comparisons, multi-valued functions (multi-functions in short) and nondeterministic computations realizing multi-functions become essential [Luc77] to compose a total but nondeterministic procedure from possibly non-terminating sub-procedures. For example, when we have two Kleenean-valued comparisons \( x < y + 2^{-n} \) and \( y < x + 2^{-n} \), which return unknown when \( x = y + 2^{-n} \) and \( y = x + 2^{-n} \) respectively, we can evaluate them in parallel until one returns true. And, if it was the first, we return true. Otherwise, we return false. As at least one of the two tests must evaluate
to true, the overall procedure safely approximates the partial test \( x < y \). However, when \( y - 2^{-n} < x < y + 2^{-n} \), the procedure is nondeterministic that it can be both true or false depending on the representations of the real numbers and the implementation of the parallel procedure which we abstract away. This nondeterministic computation is modelled by the multi-function whose value is \{true, false\} at such inputs. As nondeterministic computation becomes essential, exact real number computation frameworks provide their users with various primitive constructs for nondeterminism. For example, iRRAM offers an operator \texttt{choose} that receives multiple Kleenean expressions and returns the index (counting from 0) of some expression which evaluates to true. See that the above example can be expressed by \texttt{choose}(x \preceq y + 2^{-n}, y \preceq x + 2^{-n}) = 0.

For a formalization of exact real number computation software to be useful, it must correctly deal with the difference between the classical and computational structures of real numbers. It includes that real number comparison becomes partial Boolean-valued or total Kleenean-valued, and provide expressive enough functionality for generating interesting nondeterministic computations.

1.2. Main Contributions and Overviews. In this paper, we define an imperative programming language Exact Real Computation (ERC) providing three data types \( \mathbb{K} \), \( \mathbb{Z} \), and \( \mathbb{R} \) that ideally model the Kleene three-valued logic \( \mathbb{K} = \{true, false, unknown\} \), the set of integers \( \mathbb{Z} \), and the set of real numbers \( \mathbb{R} \), respectively. It is a simple imperative language with a strict hierarchy posed between its term language and command language where only the latter contains unbounded loops. Our language is carefully designed to ensure the computability (in the sense of computable analysis) of its domain-theoretic denotational semantics by adopting the Kleenean typed real number comparison \( \preceq \) replacing the total comparison and offering the nondeterminism construction \texttt{choose}(b_0, b_1, \ldots). We define the denotations of terms and commands to be functions to Plotkin powerdomains [Plo76] to model nondeterministic computations. The definition of computable partial (multi-)functions is reviewed in Section 2.

Construction of limits, which is an essential feature that brings ERC’s expressive power beyond algebraic computations [Bra03], is available in a restricted way. Instead of introducing an explicit operator for computing limits, which then requires the language to support constructions of nontrivial infinite sequences within its term language, we only let limits be computed at the level of “programs” another layer over commands representing (multi-)functions. More specifically, a real-valued program is provided with access to an integer variable \( p \) such that when the program computes (possibly multi-valued) \( 2^p \) approximation(s) to a real number for any \( p \), the single-valued real number, which is the limit of the program as \( p \to -\infty \), becomes the function value that of the program denotes. (A similar approach can be found in [TZ99, Section 9] and [TZ04, Definition 4.5.1].) Hence, in ERC, commands compose either an integer program expressing a partial integer multi-function or a real program expressing a partial real function where limit operations are taken. (Later in Remark 5.5, we discuss more about this design choice by introducing a straightforward way to ease this restriction which is to let the users decide whether to take the program-level limit operation.)

ERC is defined relative to primitive operator extensions such that it can be extended easily to other languages that provide further primitive operators. More formally, for each finite sets \( \mathcal{F} \) of computable partial real functions and \( \mathcal{G} \) of computable partial (finite) integer
multi-functions, we define a programming language \( \text{ERC}(\mathcal{F}, \mathcal{G}) \). We simply write \( \text{ERC} \) if the underlying extension sets are obvious or irrelevant.

Beyond modelling primitive operator extensions, the sets \( \mathcal{F} \) and \( \mathcal{G} \) provide a trick on reasoning user-defined function calls (but without recursive calls) from some richer exact real computation software. Instead of equipping \( \text{ERC} \) with a feature of calling programs within programs, which then makes \( \text{ERC} \)'s strict distinction between its term language and command language vague, in order to keep our term language as simple as possible, when the users want to model a program calling another program, they can use \( \text{ERC}(\mathcal{F}, \mathcal{G}) \) with \( \mathcal{F} \) or \( \mathcal{G} \) extended with the denotations of the programs.

We introduce a first-order logic over the three types (sorts) expanded with the (multi-)functions in \( (\mathcal{F}, \mathcal{G}) \) as a specification language which is used to specify the behaviours of \( \text{ERC} \) programs. We show that the first-order logic can define the semantics of the term language. Moreover, by not allowing integer multiplications and any integer-to-real coercion other than the \textit{precision embedding} \( \iota : \mathbb{Z} \ni p \rightarrow 2^p \in \mathbb{R} \) in our term language, we show that the specification language of our base language \( \text{ERC}_0 := \text{ERC}(\emptyset, \emptyset) \), whose design is influenced by the structure suggested in [Dri86], is model-complete and decidable with regard to its canonical interpretation. Although this looks like a bit of restriction at a first glance, since integer multiplication is expressible using a while loop in our command language, we claim that our programming language is still expressive enough by showing its Turing-completeness (in Theorem 5.3).

However, the specification language for \( \text{ERC}_0 \) is not expressive enough to define the semantics of the command language due to the fundamental trade-off:

\textbf{Remark 1.1.} Consider the following three desirable features:

\begin{itemize}
  \item [i)] a programming language being Turing-complete over integers, and thus able to realize an algorithm whose termination is co-r.e. hard
  \item [ii)] a logic sufficiently rich to express the termination of said algorithm
  \item [iii)] a sound and complete r.e. deductive system of said logic.
\end{itemize}

Obviously, not all three are simultaneously feasible. For instance integer WHILE programs / Peano arithmetic satisfy (i) / (ii), but not (iii) [Coo78, §6].

Though we propose choosing (i) and (iii) over (ii) as the base design aiming for further developments of automatic reasoning of exact real number computations using the decidability property, since our language is defined relative to primitive operator extensions, those who are not happy with this trade-off can take the other direction of the trade-off by adding, for example, the integer multiplication or the usual integer embedding \( \mathbb{Z} \ni z \mapsto z \in \mathbb{R} \) in \( (\mathcal{F}, \mathcal{G}) \). The other parts of this work including computable denotational semantics and sound verification rules remain valid.

To summarize our main contributions:

\begin{itemize}
  \item A choice of operations over real numbers and their computable semantics.
  \item A small imperative programming language with three basic data types \( \mathbb{R} \) and \( \mathbb{Z} \) and \( \mathbb{K} \) that fully coincide with \( \mathbb{R} \) and \( \mathbb{Z} \) and \( \mathbb{K} \) with respect to the said semantics.
  \item Examples demonstrating programming with these operations and semantics, such as: multi-valued integer rounding, determinant via Gaussian elimination with full pivoting (subject to full-rank promise), and root finding.
  \item Proof that this programming language is Turing-complete: those and only those real functions and integer multi-functions that can be realized in \( \text{ERC} \) are computable in the sense of computable analysis.
\end{itemize}
• A many-sorted structure for the formal specification of ERC whose first-order theory is decidable and model-complete in the case of ERC_0.
• An extended Hoare logic to enable formal verification of ERC programs’ behaviours including their terminations.

1.2.1. Overviews. In Section 2, we review the definition of computable partial (multi-)function in computable analysis and the definition of Plotkin powerdomain (over flat domains). In this section, we specify several computable partial (multi-)functions that constitute our primitive operators. Section 3 defines the formal definition of our language syntax which is a simple imperative language. There, we also define typing rules. In Section 4, we define domain-theoretic denotational semantics using Plotkin powerdomain and prove that for any well-typed terms, well-formed commands, and well-typed programs, their denotations are computable partial multi-functions (or partial real functions in the case of real programs) in Theorem 4.9. Section 5 defines some programming abbreviations and introduces various example programs: square root using Heron’s methods, exponential function via Taylor expansion and iterative, integer rounding, matrix determinants via Gaussian elimination, and root finding. Later in the section, based on the integer rounding example, we prove the Turing-completeness of ERC saying that any computable partial real functions and integer multi-functions are realizable in ERC (in Theorem 5.3).

We introduce in Section 6 a three-sorted structure and its first-order logic for rigorously specifying nondeterministic programs in ERC. Theorem 6.2 shows that its first-order theory is decidable and model-complete in the case of ERC_0. Section 6.3 extends classical Hoare logic from \( \mathbb{Z} \) to (\( \mathbb{K} \) and) \( \mathbb{R} \) and prove its soundness in Theorem 6.13. Section 7 demonstrates our verification method by proving the correctness of the aforementioned trisection program for root finding. We conclude this work with Section 8 suggesting future research, such as extending ERC from operating on real numbers in order to realize function to operating on functions in order to realize operators and other higher types.

2. Preliminaries

 ERC provides unbounded loops. To devise their sound mathematical semantics, we take a traditional domain-theoretic approach [Rey09, Chapter 2.4] and define the denotations of loops as the least fixed points of some continuous operators.

To model nondeterministic computations, we use the Plotkin powerdomain (which is reviewed later in this section) \( \mathbb{P}(Y_\bot) \) on flat domains \( Y_\bot \). An element of the domain \( p \in \mathbb{P}(Y_\bot) \) is a subset of \( Y \cup \{\bot\} \). Hence, a function \( f : X \to \mathbb{P}(Y_\bot) \), which is a multi-function, models a nondeterministic computation resulting values in \( Y \) where \( f(x) \subseteq Y \cup \{\bot\} \) denotes the set of all possible return values of the computation at input \( x \). When \( f(x) \) contains \( \bot \), it denotes the case where there is a nondeterministic branch that fails. In this sense, we say \( f \) is a partial multi-function properly defined only at \( x \) where \( \bot \not\in f(x) \).

In this section, we review the definitions of computable partial functions \( f : X \to Y_\bot \) and computable partial multi-functions \( f : X \to \mathbb{P}(Y_\bot) \). We specify some computable partial (multi-)functions that later constitute the set of primitive operators in ERC_0.

Instead of defining the computability of domain-theoretic functions by relating them with the ordinary computability definitions from computable analysis, we define the computability of functions to flat domains and powerdomains directly such that we do not have to handle
many different computability notions simultaneously. Remark 2.3 may help the readers who are already familiar with computable analysis to translate the settings back and forth.

2.1. Representations and Computable Partial Functions. For a set $X$, a representation of it $\delta_X$ is a partial surjective function from the set of infinite sequences of integers to $X$. (Recall that the domains of representations can be any Cantor or Baire space where type 2 computability theory can be built on; cf. [Wei00, Exercise 3.2.17].) When $\delta_X(\varphi) = x \in X$, we say $\varphi \in \mathbb{Z}^\mathbb{N}$ is a $\delta_X$-name of $x$. We often omit the prefix $\delta_X$ if it is obvious from the context or if it is irrelevant which specific representation underlies.

We work with the following representations of $\mathbb{Z}, \mathbb{R}, \mathbb{K}$:

- $\delta_{\mathbb{Z}}(\varphi) = k \iff \varphi(n) = k$ for all $n \in \mathbb{N}$.
- $\delta_{\mathbb{R}}(\varphi) = x \iff |x - \varphi(n) \cdot 2^{-n}| \leq 2^{-n}$ for all $n \in \mathbb{N}$.
- $\delta_X(00\cdot\cdot\cdot011\cdot\cdot\cdot) = true, \delta_X(00\cdot\cdot\cdot0(−1)(−1)\cdot\cdot\cdot) = false$, and $\delta_X(00\cdot\cdot\cdot\cdot\cdot) = unknown$. Though we present specific representations of the sets $\mathbb{Z}, \mathbb{R}, \mathbb{K}$, the theory of admissibility [Wei00, Chapter 3.2] ensures their universality with regard to the topologies: the discrete topology of $\mathbb{Z}$, the standard topology of $\mathbb{R}$, and the generalized Sierpinski topology of $\mathbb{K}$:

\[
\{ \emptyset, \{true\}, \{false\}, \{true, false\}, \{unknown, true, false\} \}.
\]

As we are yet only interested in computability, the underlying representations can be replaced with any computably equivalent representations.

When we have a function to a flat domain $f : X \to Y_\perp$, we consider it as a partial function which is properly defined at $\text{dom}(f) := \{x \in X \mid f(x) \neq \perp\}$. When $X$ and $Y$ are represented by $\delta_X$ and $\delta_Y$, respectively, $f$ is $(\delta_X, \delta_Y)$-computable if there is a (function) oracle machine $\hat{X}$ such that for any $x \in \text{dom}(f)$, when it is equipped with a $\delta_X$-name of $x$ as the oracle, there is a $\delta_Y$-name of $f(x)$ such that the machine, given $n \in \mathbb{N}$ as its input, computes the $n$th entry of the $\delta_Y$-name; see [KC96] for an example description of such oracle machines used in type-2 computability. (It also can be defined equivalently via type-2 machines instead of oracle machines; cf. [Wei00, Chapter 2.1].) We drop the prefix $(\delta_X, \delta_Y)$- if it is obvious or irrelevant which specific representations underlie.

We often call a function $f : X \to Y_\perp$ partial to emphasize that $\perp$ may be a function value. Often we call $f$ which does not admit $\perp$ as its function value total when we want to emphasize the fact. To make our notation more informative, let us write $f : X \to Y$ in the case $f : X \to Y_\perp$ is total.

For a multivariate function $f : X_1 \times \cdots \times X_d \to Y_\perp$, its computability is defined by a machine with $d$ oracles. Or, equivalently, it is defined with regard to the product representation on its domain: $\delta_{X_1 \times \cdots \times X_d}(\varphi) = (x_1, \cdots, x_d)$ if and only if $\delta_{X_j}(n \mapsto \varphi(d \times n + j - 1)) = x_j$ for each $j$. Again, other computably equivalent representations of products can replace.

The followings are examples of computable (partial) functions:

**Examples 2.1** (Primitive partial operations).

1. For any represented set $X$, its identity function $\text{id} : X \to X$ is computable.
2. The constants $true, false, unknown \in \mathbb{K}$ and $k \in \mathbb{Z}$ are computable as constant functions from any represented set.
3. The logic operations for Kleene three-valued logic $\hat{\lor}, \hat{\land} : \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ whose definitions are
are computable. Note that unknown stands for indeterminacy.

(4) The operators of Presburger arithmetic $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$, $- : \mathbb{Z} \to \mathbb{Z}$, and the $\mathbb{K}$-valued order comparisons $=, \leq : \mathbb{Z} \times \mathbb{Z} \to \mathbb{K}$ are computable.

(5) The field operators $+, \times : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $- : \mathbb{R} \to \mathbb{R}$, and $-^{-1} : \mathbb{R} \to \mathbb{R}^{-1}$ where the multiplicative inversion is not defined at 0, i.e., $0^{-1} = \perp$, are computable.

(6) The precision embedding $i : \mathbb{Z} \ni p \mapsto 2^p \in \mathbb{R}$ is computable.

(7) The $\mathbb{K}$-valued real number comparison $< : \mathbb{R} \times \mathbb{R} \to \mathbb{K}$ from Equation 1.1 is computable.

(8) A partial projection map $\text{proj} : X^d \times \mathbb{Z} \to X_{\perp}$ defined by

$$\text{proj}((x_0, \cdots, x_{d-1}), n) := \begin{cases} x_n & \text{if } 0 \leq n < d, \\ \perp & \text{otherwise}, \end{cases}$$

and a partial updating map $\text{update} : X^d \times \mathbb{Z} \times X \to (X^d)_{\perp}$ defined by

$$\text{update}((x_0, \cdots, x_{d-1}), n, x) := \begin{cases} (x_0, \cdots, x_{n-1}, x, x_{n+1}, \cdots, x_{d-1}) & \text{if } 0 \leq n < d, \\ \perp & \text{otherwise}, \end{cases}$$

are computable.

(9) A total projection map $\text{proj}_i : X_1 \times \cdots \times X_d \to X_i$ defined by $\text{proj}_i(x_1, \cdots, x_d) := x_i$ and a total updating map $\text{update}_i : X_1 \times \cdots \times X_d \times X \to X_1 \times \cdots \times X_d$ defined by $\text{update}_i((x_1, \cdots, x_d), y) := (x_1, \cdots, x_{i-1}, y, x_{i+1}, \cdots, x_d)$ updating the $i$'th entry $x_i$ with $y$, are computable.

2.2. Computable (Infinite) Multi-functions. Partial multi-functions, which are non-empty set-valued functions $f : X \to \mathcal{P}(Y_{\perp})$, model nondeterministic computations; for an input $x$, $f(x)$ is the set of all possible nondeterministic values that the computation modelled by $f$ can yield, including $\perp$ if there is a computation that fails. We say a function of type $f : X \to \mathcal{P}(Y_{\perp})$ a (partial) multi-function which is properly defined at $\text{dom}(f) := \{ x | \perp \notin f(x) \}$. We also often say a partial multi-function total when its function values never contain $\perp$; i.e., $\text{dom}(f) = X$.

For a partial multi-function $f : X \to \mathcal{P}(Y_{\perp})$, when the underlying sets $X, Y$ are represented by $\delta_X$ and $\delta_Y$ respectively, we say that $f$ is $(\delta_X, \delta_Y)$-computable if there is an oracle machine such that when a $\delta_X$-name of any $x \in \text{dom}(f)$ is equipped as the oracle, there is some $y \in f(x)$ and its $\delta_Y$-name such that on the machine's input $n$, it returns the $n$'th entry of the $\delta_Y$-name of $y$. (Again, type-2 machines can replace in the definition.) We omit the prefix $(\delta_X, \delta_Y)$- if it is not necessary to explicitly specify the underlying representations. Note that nondeterminism happens due to the fact that when a different name of the same $x$ is given, it is allowed to compute different $y' \in f(x)$.
2.3. Computable (Finite) Multi-functions and Plotkin Powerdomain. However, using the entire non-empty power-set \( \mathcal{P}(Y_\perp) \) causes the same problem that motivated \cite[Section 2]{plotkin76} that it does not give us a nice order theoretic property in defining denotational semantics. Reminding that ERC only provides finite nondeterminism, and that as long as only real numbers and integers are possible inputs there is no necessarily infinite integer multi-function (see Fact 2.5), we safely restrict to Plotkin powerdomain \cite{plotkin76}. For any set \( X \), the powerdomain on its flat domain \( X_\perp \) is defined by

\[
\mathbb{P}(X_\perp) := \{ p \subseteq X \cup \{ \perp \} \mid p \neq \emptyset, \text{ and } p \text{ is finite or contains } \perp \} \subseteq \mathcal{P}(X_\perp)
\]

endowed with Egli-Milner ordering:

\[
p \subseteq q \iff (\forall x \in p \exists y \in q. x \leq y) \land (\forall y \in q \exists x \in p. x \leq y).
\]

Recall that \( x \leq y \) if and only if \( x = \perp \) or \( x = y \) for \( x, y \in X_\perp \). The constructed powerdomain for any flat domain \( X_\perp \) is known to be a \( \omega \text{-}cpo \) with the least element \{ \perp \} \cite[Chapter 7.2]{rey09}.

For a function to a flat-domain \( f : X \to Y_\perp \), let us write \( f^\dagger : X \to \mathbb{P}(Y_\perp) \) for the embedding \( f^\dagger(x) := \{ f(x) \} \). For a multivariate partial multi-function \( f : X_1 \times \cdots \times X_d \to \mathbb{P}(Y_\perp) \), its extension \( f^\dagger : \mathbb{P}((X_1)_\perp) \times \cdots \times \mathbb{P}((X_d)_\perp) \to \mathbb{P}(Y_\perp) \) is defined by

\[
f^\dagger(p_1, \ldots, p_d) := \bigcup_{x_1 \in p_1, \ldots, x_d \in p_d} \begin{cases} \{ f(x_1, \ldots, x_d) \} & \text{if } \forall i. x_i \neq \perp, \\ \{ \perp \} & \text{otherwise.} \end{cases}
\]

And, the extension by one argument \( f^\dagger_i : X_1 \times \cdots \times \mathbb{P}((X_i)_\perp) \times \cdots \times X_d \to \mathbb{P}(Y_\perp) \) is defined

\[
f^\dagger_i(x_1, \ldots, p_i, \ldots, x_d) := \bigcup_{x_i \in p_i} \begin{cases} \{ f(x_1, \ldots, x_i, \ldots, x_d) \} & \text{if } x_i \neq \perp, \\ \{ \perp \} & \text{otherwise.} \end{cases}
\]

Though the embedding and extensions are more naturally understood in the setting of category theory \cite{bvs93, aj95}, we define them separately in this independent context.

The composition of two partial multi-functions \( f : X \to \mathbb{P}(Y_\perp) \) and \( g : Y \to \mathbb{P}(Z_\perp) \) is expressed by \( g^\dagger \circ f : X \to \mathbb{P}(Z_\perp) \). Recall that this composition is continuous in both arguments \cite[Section 6]{plotkin76}. In other words, given \( f_i, f_i : X \to \mathbb{P}(Y_\perp) \) and \( g_i : Y \to \mathbb{P}(Z_\perp) \) where \( f_i \) and \( g_i \) are increasing chains with regard to the point-wise orderings, it holds that

\[
\bigcup_{i \in \mathbb{N}} (g_i^\dagger \circ f_i) = g^\dagger \circ \bigcup_{i \in \mathbb{N}} f_i \quad \text{and} \quad \bigcup_{i \in \mathbb{N}} (g_i^\dagger \circ f) = (\bigcup_{i \in \mathbb{N}} g_i)^\dagger \circ f.
\]

Also, the partial composition is continuous: Consider a multivariate multi-function \( f : X_1 \times \cdots \times X_d \to \mathbb{P}(Y_\perp) \), a list of functions \( g_j : X \to X_j \) for \( j \neq i \), and a chain of multi-functions \( g_{i,k} : X \to \mathbb{P}((X_i)_\perp) \) in \( k \). Then,

\[
\bigcup_{k \in \mathbb{N}} f^\dagger_i \circ (g_1, \ldots, g_i, \ldots, g_d) = f^\dagger_i \circ (g_1, \ldots, \bigcup_{k \in \mathbb{N}} g_{i,k}, \ldots, g_d).
\]

Here, for a finite collection of functions \( f_1, \ldots, f_d \) sharing their domains, \( \langle f_1, \ldots, f_d \rangle \) denotes \( x \mapsto (f_1(x), \ldots, f_d(x)) \).

For each set \( X \), define \( \text{Cond} : \mathbb{K} \times \mathbb{P}(X_\perp) \times \mathbb{P}(X_\perp) \to \mathbb{P}(X_\perp) \) by

\[
\text{Cond}(b, p, q) := \begin{cases} p & \text{if } b = \text{true}, \\ q & \text{if } b = \text{false}, \\ \{ \perp \} & \text{otherwise,} \end{cases}
\]
a conditional combinator of $\mathbb{K}$. For any $b : X \to \mathcal{P}(X)_{\perp}$ and $S : X \to \mathcal{P}(X)_{\perp}$, due to the continuity of compositions, the following operator can be seen continuous:

$$W(b, S) (f : X \to \mathcal{P}(X)_{\perp}) := \text{Cond}^\dagger \circ \langle b, f^\dagger \circ S, \text{id}^\dagger \rangle .$$

The continuity ensures the existence of the least fixed point $\text{lfp}(W(b, S)) : X \to \mathcal{P}(X)_{\perp}$ which is used to define the denotations of loops later in Section 4.

The followings are examples of computable (finite) multi-functions:

**Examples 2.2** (Primitive multi-valued operations).

1. When $f : X \to Y_{\perp}$ is a computable partial/total function, its embedding $f^\dagger : X \to \mathcal{P}(Y)_{\perp}$ is a computable partial/total multi-function.

2. When $f : X \to \mathcal{P}(Y)_{\perp}$ and $g : Y \to \mathcal{P}(Z)_{\perp}$ are computable partial multi-functions, their composition $g^\dagger \circ f : X \to \mathcal{P}(Z)_{\perp}$ is a computable partial multi-function.

3. When $f_i : X \to \mathcal{P}(X)_{\perp}$ and $g : Y_1 \times \cdots \times Y_d \to Z$ are computable multi-functions, their composition $g^\dagger \circ \langle f_1, \ldots, f_d \rangle : X \to \mathcal{P}(Z)_{\perp}$ is a computable multi-function.

4. For each $n \in \mathbb{N}$, the nondeterministic selection function choose$_n : \mathbb{K}^n \to \mathcal{P}(Z)_{\perp}$ defined by

$$\text{choose}_n(b_0, \ldots, b_{n-1}) := \begin{cases} \{i \mid b_i = \text{true}\} & \text{if there is } i \text{ s.t. } b_i = \text{true}, \\ \{\perp\} & \text{otherwise}, \end{cases}$$

is a computable partial multi-function.

5. For any computable $f : X \to \mathcal{P}(Y)_{\perp}$ and $g, h : X \to \mathcal{P}(Y)_{\perp}$, the composite multi-function

$$\text{Cond}^\dagger \circ \langle f, g, h \rangle : X \to \mathcal{P}(Y)_{\perp}$$

is computable.

6. Define Kond : $\mathbb{K} \times \mathcal{P}(\mathbb{R})_{\perp} \times \mathcal{P}(\mathbb{R})_{\perp} \to \mathcal{P}(\mathbb{R})_{\perp}$, a continuous extension of Cond by

$$\text{Kond}(b, p, q) := \begin{cases} p & \text{if } b = \text{true}, \\ q & \text{if } b = \text{false}, \\ \{x\} & \text{if } b = \text{unknown} \land p = q = \{x\}, \\ \{\perp\} & \text{otherwise}, \end{cases}$$

with regard to the topology of $\mathbb{K}$; cmp. [Wei00, Theorem 2.3.8] and “parallel if” from PCF [Plo77]. For any computable $f : X \to \mathcal{P}(Y)_{\perp}$ and $g, h : X \to \mathcal{P}(Y)_{\perp}$, the composition $\text{Kond}^\dagger \circ \langle f, g, h \rangle : X \to \mathcal{P}(Y)_{\perp}$ is a computable partial multi-function.

The formulations of computable partial (multi-)functions in this paper differ from how they are traditionally dealt with. The following remark states how they are related.

**Remark 2.3.** In the traditional setting of computable analysis:

1. The computability of a partial function presented by $f : \sqsubseteq X \to Y$ is equivalent to the computability of $f_{\perp} : X \to Y_{\perp}$ defined by $x \mapsto f(x)$ if $x \in \text{dom}(f)$ and $\perp$ if $x \not\in \text{dom}(f)$. Similarly, the computability of $f : X \to Y_{\perp}$ is equivalent to the computability of $f \upharpoonright_{\text{dom}(f)} : \sqsubseteq X \to Y$. Here, $\upharpoonright$ is a notation for the domain restriction and recall that $\text{dom}(f) = \{x \mid f(x) \neq \perp\}$.

2. The computability of a partial multi-function presented by $f : \sqsubseteq X \rightharpoonup Y$, which is a partial non-empty set-valued function, is equivalent to the computability of $f_{\perp} : X \to \mathcal{P}(Y)_{\perp}$. Similarly, the computability of $f : X \to \mathcal{P}(Y)_{\perp}$ is equivalent to the computability of $f \upharpoonright_{\text{dom}(f)} : \sqsubseteq X \rightharpoonup Y$. Recall that $\text{dom}(f) = \{x \mid \perp \notin f(x)\}$.
Observe from the definitions that a machine computing a partial (multi-)function can be seen to compute some other partial (multi-)functions as (1) for a partial (multi-)function, the out-of-domain behaviour of its realizer is not specified and (2) for a partial multi-function, its realizer only has to compute some of the possible function values. Specifically:

**Fact 2.4.** If a partial function $f : X \to Y_\perp$ is computable, any domain restriction of it is computable; $g : X \to Y_\perp$ such that $\forall x. g(x) \leq f(x)$ is computable by the same machine computing $f$. And, if a multi-function $f : X \to \mathcal{P}(Y_\perp)$ is computable, any domain restriction of it and function value enlargement, i.e. $g : X \to \mathcal{P}(Y_\perp)$ such that $\forall x. g(x) \subseteq f(x)$ or $f(x) \subseteq g(x)$ is computable by the same machine computing $f$.

For any real number or integer, the set of its names is compact with regard to the standard Baire space topology on $\mathbb{Z}^\mathbb{N}$. Hence, for any realizer, an oracle machine, computing an integer partial multi-function $f : X_1 \times \cdots \times X_d \to \mathcal{P}(\mathbb{Z}_\perp)$ where $X_i = \mathbb{Z}$ or $\mathbb{R}$, for fixed inputs $(x_1, \ldots, x_d)$, when we range over their names, the set of machine outputs is also compact as machines’ semantics is continuous. That means even when $f(x_1, \ldots, x_d)$ is infinite, there are only finitely many different integers that the machine actually computes; see [Bra95, Theorem 4.2].

**Fact 2.5.** Any computable multi-function $f : X_1 \times \cdots \times X_d \to \mathcal{P}(\mathbb{Z}_\perp)$ where $X_i = \mathbb{Z}$ or $\mathbb{R}$, admits finite refinement in the sense that there is a computable (finite) multi-function $g : X_1 \times \cdots \times X_d \to \mathcal{P}(\mathbb{Z}_\perp)$ such that $\forall(x_1, \ldots, x_d). g(x_1, \ldots, x_d) \subseteq f(x_1, \ldots, x_d)$ holds.

In addition to the above justification of our use of Plotkin powerdomain, the following is a note on the continuity issue for nondeterminism:

**Remark 2.6.** Note that choose$_n : \mathbb{K}^n \to \mathcal{P}(\mathbb{Z}_\perp)$ is not (domain-theoretic) continuous when $\mathbb{K}$ is ordered by $b_1 \leq b_2$ iff $b_1 = b_2$ or $b_1 = \text{unknown}$. It is observed in [MRE07] and there it is suggested Hoare powerdomain be used instead. However, our approach distinguishes (domain-theoretic) continuity and computability of primitives: we do not use (domain-theoretic) continuity as a criterion for computability. As long as we have a fact that choose$_n : \mathbb{K}^n \to \mathcal{P}(\mathbb{Z}_\perp)$ is computable, there is no need to make it (domain-theoretic) continuous by giving an order on $\mathbb{K}$. This way we still use Plotkin powerdomain in defining our semantics such that we do not need to have separate operational semantics to express termination as in [MRE07].

### 3. Formal Syntax and Typing Rules of Exact Real Computation

In this section, we provide the formal syntax and typing rules of ERC($\mathcal{F}, \mathcal{G}$) for any $\mathcal{F}$ and $\mathcal{G}$ according to the above convention:

**Convention 3.1.** For the sequel fix a (possibly empty) finite set $\mathcal{F}$ of computable functions to $\mathbb{R}_\perp$ from products of $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{R}^n$ for any $n \geq 1$, as well as a (possibly empty) finite set $\mathcal{F}$ of computable multi-functions to $\mathcal{P}(\mathbb{Z})$ from products of $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{R}^n$ for any $n \geq 1$.

#### 3.1. Syntax

ERC is an imperative programming language comprising of the following axiomatized constituents: data types (Section 3.1.1), terms (Section 3.1.2), commands (Section 3.1.3), and programs (Section 3.1.4).
3.1.1. **Data Types.** The data types that ERC provides are as follow:

\[ \tau ::= K \mid Z \mid R \mid R[n] \text{ for each } n = 1, 2, \cdots \]

See that ERC provides countably many data types: for each natural number \( n \geq 1 \), there is a data type \( R[n] \), which represents the set of arrays of real numbers of fixed length \( n \). Note \( n \) here is a meta-level expression which is not a ERC “term” from Section 3.1.2.

3.1.2. **Terms.** Although the type of a term is not determined syntactically, we here follow the following conventions for convenience: Write \( z, z_i \) to denote terms which should be typed \( Z \), \( x, y, x_i, y_i \) to denote terms which should be typed \( R \), and \( b, b_i \) to denote terms which should be typed \( K \). Arbitrary terms are denoted by \( t, t_i \). For some fixed countable set of variables \( \mathcal{V} \), the term language of ERC is defined as follows.

\[
\begin{align*}
t, t_i, z, z_i, x, y, b, b_i &::= \\
&k \quad k \in \mathbb{Z}; \text{ \( Z \) literal} \\
&| \text{ true} | \text{ false} | \text{ unknown} \quad \text{\( K \) literal} \\
&[t_1, \cdots, t_n] \quad \text{array literal} \\
&v \quad v \in \mathcal{V}; \text{ variable} \\
&t[z] \quad \text{array access} \\
&t_1 + t_2 \mid -t \quad \text{addition and additive inversion} \\
&z_1 \leq z_2 \mid z_1 = z_2 \quad \text{integer comparison and equality test} \\
&x \times y \mid x^{-1} \quad \text{real multiplication and multiplicative inversion} \\
&x < y \quad \text{real comparison} \\
&\iota(z) \quad \text{precision embedding} \\
&\ll b \mid b_1 \hat{\lor} b_2 \mid b_1 \hat{\land} b_2 \quad \text{Kleene logic} \\
&f(t_1, \cdots, t_d) \quad (f : X_1 \times \cdots \times X_d \to \mathbb{R}_\bot) \in \mathcal{F}; \text{ primitive function call} \\
&g(t_1, \cdots, t_d) \quad (g : X_1 \times \cdots \times X_d \to \mathbb{P}(\mathbb{Z}_\bot)) \in \mathcal{G}; \text{ primitive multi-function call} \\
&\text{choose}_{n'}(b_0, b_1, \cdots, b_{n-1}) \quad n \in \mathbb{N}; \text{ multi-valued choice} \\
&b \ ? \ x : y \quad \text{continuous conditional}
\end{align*}
\]

Note that we do not provide integer multiplication and any other coercion than \( \iota \) unless they are provided through \( \mathcal{F}, \mathcal{G} \). Though for each natural number \( n \), \( \text{choose}_n \) is a distinct term construct, since the \( n \) is determined syntactically by the number of the arguments, we may omit it and instead simply refer to \( \text{choose}() \). We introduce \( z_1 = z_2 \) as a term construct instead of as an abbreviation for \( z_1 \leq z_2 \land z_2 \leq z_1 \); see Remark 4.3(7).

3.1.3. **Commands.** ERC provides strict distinction between its term language and command language where loops belong only to the command language. Commands in ERC are inductively constructed as follow.

\[
\begin{align*}
S, S_i &::= \text{skip} \\
&| v := t \quad \text{variable assignment} \\
&| v[z] := t \quad \text{array assignment}
\end{align*}
\]
let $v : \tau = t$

$S_1; S_2$

if $b$ then $S_1$ else $S_2$ end

while $b$ do $S$ end

Although we use the identifier `let`, it is not the usual ‘let-in binding’ for creating a local variable.

3.1.4. Programs. Having data types, terms, and commands defined, we can finally define what a program in ERC is. A program in ERC is in one of the following forms:

\[
\begin{align*}
\text{input } & v_1 : \tau_1, v_2 : \tau_2, \cdots, v_n : \tau_n \\
\text{return } & t
\end{align*}
\]

Here, $v_i$ and $p$ are variables varying over $\mathcal{V}$ and $\tau_i$ varies over \{\Z, \R, \R[1], \R[2], \cdots\}. The variable $p$ that is separated from the other input variables is the precision parameter which should not be misunderstood as an input variable. The intended behaviour of a program in the right kind is, taking $v_1, \cdots, v_n$ as inputs, to compute the limit of the command $S$ and the return term $t$ as $p$ goes to $-\infty$. This is formally defined later in Section 4.

3.2. Typing Rules. Here we define well-typedness of terms (Section 3.2.1), well-formedness of commands (Section 3.2.2), and well-typedness of programs (Section 3.2.3) under a context.

A context is a mapping from a finite set of variables to their corresponding types represented by a list of assignments $\Gamma := v_1 : \tau_1, v_2 : \tau_2, \cdots, v_n : \tau_n$ denoting the context mapping variable $v_i$ to its data type $\tau_i$ for $i = 1, \ldots, n$.

3.2.1. Well-Typed Terms. Well-typedness of a term $t$ in ERC to $\tau$ under a context $\Gamma$ is written as $\Gamma \vdash t : \tau$. Figure 1 shows ERC's type inference rules.

Note that $+$ and $-$ are used both in integer operations and real number operations.

3.2.2. Well-formed Commands. Unlike terms, a command in ERC may modify contexts. Let us denote a command $S$ under a context $\Gamma$ being well-formed and yielding a new context $\Gamma'$ as $\Gamma \vdash S \triangleright \Gamma'$. The Well-formedness of commands is defined by the inference rules in Figure 2.

See that the only construct that modifies a context is `let $v : \tau = t$` (variable declaration). When it is executed under a context $\Gamma$, the command is well-formed if $v$ is not already included in $\Gamma$ and the type of the initializing term $t$ is of the declared type $\tau$. After the execution, we get the new context $\Gamma, v : \tau$.

Note also that even if a context changes inside of a branch or a loop, it gets restored once the block is finished. In other words, the variables created inside of a branch or a loop only survive locally (as common for example in C++).
3.2.3. Well-Typed Programs. Consider the two different kinds of ERC programs:

\[ P_Z := \text{input } \tau_1, \tau_2, \ldots, \tau_d \quad \text{return } t \quad P_R := \text{input } \tau_1, \tau_2, \ldots, \tau_d \quad \text{return } t \text{ as } p \rightarrow -\infty \]

The first program \( P_Z \) has type \( \tau_1 \times \cdots \times \tau_d \rightarrow Z \) if there is a context \( \Gamma' \) such that

\[ \Gamma' \vdash t : \tau \]

and

\[ \vdash z : Z \quad \Gamma' \vdash t : R[n] \]

\[ \Gamma \vdash t[z] : R \]

Figure 1: Typing rules for terms

| Rule | Description |
|------|-------------|
| \( \Gamma \vdash k : Z \) | \( \Gamma \vdash true : K \) | \( \Gamma \vdash false : K \) | \( \Gamma \vdash unknown : K \) |
| \( \Gamma \vdash t_i : R \) \ (for \( i = 1, \ldots, n \)) | \( \Gamma \vdash \text{choose}_n(b_0, b_1, \ldots, b_{n-1}) : Z \) | \( \Gamma \vdash \text{false} : K \) | \( \Gamma \vdash \text{true} : K \) |
| \( \Gamma \vdash t_1 \circ t_2 : \tau \) \ (\( \circ, \tau, \tau' \) \in BinTy) | \( \Gamma \vdash t_1 \odot t_2 : \tau \) \ (\( \odot, \tau, \tau' \) \in UnTy) | \( \Gamma \vdash z : Z \) | \( \Gamma \vdash t : R[n] \) |
| \( \Gamma \vdash f : (X_1 \times \cdots \times X_d \rightarrow \mathbb{R}_+) \in F \) | \( \Gamma \vdash f\langle \langle X_i \rangle \rangle : \text{for } i = 1, \ldots, d \) | \( \Gamma \vdash \text{let } v : \tau = t \vdash \Gamma \) | \( \Gamma \vdash (\forall x : \tau) \in \text{UnTy} \) |
| \( \Gamma \vdash g : (X_1 \times \cdots \times X_d \rightarrow \mathbb{R}_+) \in G \) | \( \Gamma \vdash g\langle \langle X_i \rangle \rangle : \text{for } i = 1, \ldots, d \) | \( \Gamma \vdash x \vdash \Gamma \) | \( \Gamma \vdash x^{-1} : R \) |

Figure 2: Typing rules for commands

| Rule | Description |
|------|-------------|
| \( \Gamma \vdash \text{skip} \supset \Gamma \) | \( \Gamma \vdash v := t \supset \Gamma \) | \( \Gamma \vdash z : Z \) | \( \Gamma \vdash t : R \) | \( \Gamma \vdash t[z] := t \supset \Gamma \) |
| \( \Gamma \vdash \text{if } b \text{ then } S_1 \text{ else } S_2 \supset \Gamma \) | \( \Gamma \vdash b : K \) | \( \Gamma \vdash S_1 \supset \Gamma \) | \( \Gamma \vdash S_2 \supset \Gamma \) | \( \Gamma \vdash \text{while } b \text{ do } S \text{ end} \supset \Gamma \) |
Similarly, $P_R$ has type $\tau_1 \times \cdots \times \tau_d \to R$ if there is a context $\Gamma'$ such that

$$p : Z, v_1 : \tau_1, \ldots, v_d : \tau_d \vdash S' \Gamma' \quad \text{and} \quad \Gamma' \vdash t : R.$$ 

Note once again here that $p : Z$ is not regarded as an input variable. Recall also that $\tau_i$ varies over $\{Z, R, R[1], R[2], \ldots\}$ and cannot be $k$.

We call an ERC program an integer program if it is of the first kind ($P_Z$) and a real program if it is of the second kind ($P_R$).

4. Denotational Semantics of Exact Real Computation

We define a multi-valued semantics for well-typed terms (Section 4.1), well-formed commands (Section 4.2), and well-typed programs (Section 4.3). In this semantics, the objects of ERC are assigned mathematical meanings that are arguably (i) closest possible to the intuition of real numbers as entities to be operated on exactly while simultaneously featuring (ii) Turing-completeness.

We start with denotations for data types and the definition of states.

**Definition 4.1.**

(1) Data types are interpreted as intended:

$$[K] = K, \quad [Z] = Z, \quad [R] = R, \quad [R[n]] = R^n.$$ 

Recall that $n$ above is a natural number greater than 0 which is not to be confused with an integer typed term in ERC.

(2) Contexts are interpreted as sets of assignments. Formally, given a context $\Gamma = v_1 : \tau_1, \ldots, v_n : \tau_d$, then

$$[\Gamma] := [\tau_1] \times \cdots \times [\tau_d].$$

(3) Given a context $\Gamma$, an element $\sigma \in [\Gamma]$ is called state. That is, states are specific assignments of variables contained in the domain of a context.

4.1. Denotations of Terms. A (well-typed) term's meaning under a state is, considering the nondeterminism in ERC, the set of all possible values that the term can evaluate to. For example, a well-typed term $\Gamma \vdash t : R$ can evaluate to multiple values under a state $\sigma$: Its denotation contains a subset of $R$. Moreover, if 0 is among these values, then the compound term $t^{-1}$ could be undefined—in addition to its defined values derived from non-zero values of $t$. This is reflected by including $\perp$ in the denotation, which is thus an element of $\mathbb{P}([R]_\perp)$:

**Definition 4.2.** Recall the (multi-)functions specified in Example 2.1 and 2.2. Given a well-typed term $t$ such that $\Gamma \vdash t : \tau$, we define its denotation as a partial multi-function $[\Gamma \vdash t : \tau] : [\Gamma] \to \mathbb{P}([\tau]_\perp)$ inductively as follows:

$$[\Gamma \vdash k : Z] := \sigma \mapsto \{k\}$$
$$[\Gamma \vdash \text{true} : K] := \sigma \mapsto \{\text{true}\}$$
$$[\Gamma \vdash \text{false} : K] := \sigma \mapsto \{\text{false}\}$$
$$[\Gamma \vdash \text{unknown} : K] := \sigma \mapsto \{\text{unknown}\}$$
$$[\Gamma \vdash [t_1, \ldots, t_n] : R[n]] := \text{id}_{\perp}^t \circ ([\Gamma \vdash t_1 : R], \ldots, [\Gamma \vdash t_n : R])$$
\[
\begin{align*}
\forall v_1 : \tau_1, \ldots, v_i : \tau_i, \ldots, v_d : \tau_d \vdash v_i : \tau_i & := \text{proj}_i^d \\
\llbracket \Gamma \vdash t[z] : \mathbb{R} \rrbracket & := \text{proj}_i^d \circ \langle \llbracket \Gamma \vdash t : \mathbb{R} [n] \rrbracket, \llbracket \Gamma \vdash z : \mathbb{Z} \rrbracket \rangle \\
\llbracket \Gamma \vdash t_1 \circ t_2 : \tau \rrbracket & := \circ^t \circ \langle \llbracket \Gamma \vdash t_1 : \tau' \rrbracket, \llbracket \Gamma \vdash t_2 : \tau' \rrbracket \rangle \\
& \quad \text{for } (\circ, \tau, \tau') \in \text{BinTy} \\
\llbracket \Gamma \vdash t \times \tau : \tau' \rrbracket & := \ast^t \circ \llbracket \Gamma \vdash t : \tau' \rrbracket \\
& \quad \text{for } (\ast, \tau, \tau') \in \text{UnTy} \\
\llbracket \Gamma \vdash x^{-1} : \mathbb{R} \rrbracket & := (-1)^t \circ \llbracket \Gamma \vdash x : \mathbb{R} \rrbracket \\
\llbracket \Gamma \vdash f(t_1, \ldots, t_d) : \mathbb{R} \rrbracket & := f^t \circ \langle \llbracket \Gamma \vdash t_1 : (X_1) \rrbracket, \ldots, \llbracket \Gamma \vdash t_d : (X_d) \rrbracket \rangle \\
& \quad \text{where } (f : X_1 \times \cdots \times X_d \to \mathbb{R}_+) \in \mathcal{F} \\
\llbracket \Gamma \vdash g(t_1, \ldots, t_d) : \mathbb{Z} \rrbracket & := g^t \circ \langle \llbracket \Gamma \vdash t_1 : (X_1) \rrbracket, \ldots, \llbracket \Gamma \vdash t_d : (X_d) \rrbracket \rangle \\
& \quad \text{where } (g : X_1 \times \cdots \times X_d \to \mathbb{P}(\mathbb{Z}_\bot)) \in \mathcal{G} \\
\llbracket \Gamma \vdash \text{choose}_n(b_0, \ldots, b_{n-1}) : \mathbb{Z} \rrbracket & := \text{choose}_n^t \circ \langle \llbracket \Gamma \vdash b_0 : \mathbb{K} \rrbracket, \ldots, \llbracket \Gamma \vdash b_{n-1} : \mathbb{K} \rrbracket \rangle \\
\llbracket \Gamma \vdash (b ? x : y) : \mathbb{R} \rrbracket & := \text{Kond}^t \circ \langle \llbracket \Gamma \vdash b : \mathbb{K} \rrbracket, \llbracket \Gamma \vdash x : \mathbb{R} \rrbracket, \llbracket \Gamma \vdash y : \mathbb{R} \rrbracket \rangle 
\end{align*}
\]

We often simply write \([t] \sigma\) instead of \([\Gamma \vdash t : \tau] \sigma\) and \([\Gamma \vdash t : \tau] \sigma\), respectively, omitting \(\Gamma\) and \(\tau\) when they are obvious or irrelevant. We also synonymously say \(t\) evaluates to \(x\) under \(\sigma\), or \(t\) has/contains the element \(x \in [t] \sigma\).

We add some explanation:

\textbf{Remark 4.3.}

1. Note that \text{choose()} and functions from \(\mathcal{G}\) constitute atomic constructs that yield multi-valuedness, all other atomic constructs do not generate any multi-valuedness.
2. Although the denotations of terms are multi-valued, every variable only stores one specific value: in an assignment possibly selected non-deterministically, see Section 4.2.
3. The denotations of arithmetical operations are defined naturally by lifting the intended mathematical functions with regard to the powerdomain. (Recall Section 2.)
4. Well-typedness in ERC does not prevent \(\bot\): The sources of the ill-definition \(\bot\) are (i) an array accessed by a wrong index (\text{proj}), (ii) the multiplicative inverse of 0 \((0^{-1})\), (iii) function calls to partial (multi-)functions at out of their domains, (iv) a choice operation without any choosable argument (\text{choose}), and (v) a continuous conditional when the two arguments \(x, y\) do not match in \text{Kond}(\text{unknown}, x, y).
5. The denotation of \([t_1, \cdots, t_n] \含有 \(\bot\) if at least one of the denotations of the arguments contains \(\bot\). This is what the postcomposed id\(^t\) does.
6. The denotation of comparison \(\langle x \prec y \rangle\) between real terms \(x, y\) formalizes the return value \text{true} if the value of \(x\) is smaller than the value \(y\), return value \text{false} if the value of \(x\) is bigger than the value \(y\), and 'return' value \text{unknown} when their values are equal: indicating constructive indefiniteness. Independently, the comparison is ill-defined \((\bot)\) if any of its sub-terms is.
(7) Due to multi-valuedness, the denotation of integer equality \(z_1 = z_2\) is subtly different from that of \(z_1 \leq z_2 \wedge z_2 \leq z_1\): Consider a state \(\sigma\) where \([z_1]\) \(\sigma = \{0, 2\}\) and \([z_2]\) \(\sigma = \{1, 3\}\). Then, \([z_1 \leq z_2 \wedge z_2 \leq z_1]\) \(\sigma = \{\text{true, false}\}\) whereas \([z_1 = z_2]\) \(\sigma = \{\text{false}\}\).

(8) Though \(\mathcal{P}(X_{\perp})\) allows infinite sets (as long as they contain \(\perp\)), the case does not occur in the denotational semantics of our term language. The denotation of any well-typed term at any state is finite unless the case is provided by functions in \(\mathcal{G}\). However, we still use the powerdomain for our term language to use the same setting with the denotational semantics of our command language where indeed \(\perp\)-containing infinite sets arise.

See that our denotational semantics uses only the constructions that are already specified to be yielding computable partial multi-functions from Section 2:

**Lemma 4.4.** The denotation of any well-typed term is computable. In other words, for any well-typed term \(\Gamma \vdash t : \tau\), the function \([\Gamma \vdash t : \tau] : [\Gamma] \rightarrow \mathcal{P}([\tau]_{\perp})\) is a computable partial multi-function.

**Proof.** It is direct from the computability of the primitive operations and of the compositions stated in Example 2.1 and Example 2.2.

### 4.2. Denotations of Commands.

As the denotation of a well-typed term under a state was the set of all possible values that the term may evaluate to, considering multi-valuedness in ERC, the denotation of a well-formed command under a state is the set of all possible resulting states that the command may result in. Hence, we let a well-typed command denote a function from the set of states to the restricted power-set of the resulting states:

**Definition 4.5.** Recall the (multi-)functions specified in Example 2.1 and 2.2. Given a well-formed command \(S\) such that \(\Gamma \vdash S \triangleright \Gamma\), we define its denotation to be a partial multi-function \([\Gamma \vdash S \triangleright \Gamma'] : [\Gamma] \rightarrow \mathcal{P}([\Gamma']_{\perp})\) inductively as follows:

\[
\begin{align*}
[\Gamma \vdash \text{skip} \triangleright \Gamma] & := \text{id}^t \\
[\Gamma \vdash v_i := t \triangleright \Gamma] & := \text{update}^t_{\pi} \circ (\text{id}^t, \{i\}) \\
& \quad \text{where } \Gamma = v_1 : \tau_1, \ldots, v_i : \tau_i, \ldots, v_d : \tau_d \\
[\Gamma \vdash v_i[z] := t \triangleright \Gamma] & := \text{update}^t_{\pi} \circ \langle \text{id}^t, \text{update}^t_i \circ \langle \text{proj}^t_i, [z], [t]\rangle \rangle \\
& \quad \text{where } \Gamma = v_1 : \tau_1, \ldots, v_i : \tau_i, \ldots, v_d : \tau_d \\
[\Gamma \vdash \text{let } v : \tau = t \triangleright \Gamma'] & := \text{id}^t_{\pi} \odot (\text{id}^t, [t]) \\
[\Gamma \vdash S_1 ; S_2 \triangleright \Gamma'] & := [\Delta \triangleright S_2 \triangleright \Gamma'] \circ [\Gamma \vdash S_1 \triangleright \Delta] \\
[\Gamma \vdash \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ end} \triangleright \Gamma] & := \text{Cond}^t_{\pi} \odot \langle [b], [i]^{t^t} \circ [\Gamma \vdash S_1 \triangleright \Gamma'], [i]^{t^t} \circ [\Gamma \vdash S_2 \triangleright \Gamma']\rangle \\
[\Gamma \vdash \text{while } b \text{ do } S \text{ end} \triangleright \Gamma] & := \text{lfp}([b], [i]^{t^t} \circ [\Gamma \vdash S \triangleright \Gamma'])
\end{align*}
\]

Here, \([\cdot] : [\Gamma, \Delta] \rightarrow [\Gamma]\) for any contexts \(\Gamma, \Delta\) when \(\Gamma = x_1 : \tau_1, \ldots, x_d : \tau_d\) is defined by

\[
(x_1, \ldots, x_d, y_1, \ldots, y_m) \mapsto (x_1, \ldots, x_d).
\]

Again, we simply write \([S]\) and \([S]_{\sigma}\) instead of \([\Gamma \vdash S \triangleright \Gamma']\) and \([\Gamma \vdash S \triangleright \Gamma']\), respectively, omitting \(\Gamma\) and \(\Gamma'\) when they are obvious or irrelevant.
Remark 4.6.

(1) Regarding assignment “\(v_i := t\)”: Note that instead of having a single state where \(v_i\) stores multiple values, it generates multiple states where each \(v_i\) stores different single values. For each value in \([t]\) \(\sigma\) for some state \(\sigma\), the command creates \(\sigma\) where its value at \(u_i\) updated with the value unless it is \(\bot\).

(2) Regarding assignment “\(v_z := t\)”: It generates \(\bot\) when the index \(z\) is not acceptable. For each value in the denotation of \(z\) at a state \(\sigma\), if it is either negative or larger than the dimension of \(v\), it creates \(\bot\).

(3) “if \(b\) then \(S_1\) else \(S_2\) end”: It naturally extends the conditional statement. Though new variables can be created in \(S_1\) and \(S_2\), they get removed by the restriction operator \(\rhd\) at the end.

(4) “while \(b\) do \(S\) end”: By the least fixed point theorem, the least fixed point is the limit of the chain:

\[
W_{b,S}^{(0)} := \sigma \mapsto \{\bot\} \quad \text{and} \quad W_{b,S}^{(n+1)} := \text{Cond}^{\rhd_1} \circ ([b], W_{b,S}^{(n)} \circ ([\rhd \circ [S]], \text{id}^{\rhd})).
\]

The sequence, intuitively, represents the process of unrolling the loop.

For any two states \(\sigma, \delta \in [\Gamma]\), \(\delta\) is in \([\Gamma \vdash \text{while } b \text{ do } S \text{ end} \rhd \Gamma] \sigma\) if and only if there is a natural number \(n\) where \(\delta \in W_{b,S}^{(n)} \sigma\). Hence, instead of working with the subtle fixed point for the denotation of a while loop, one can do reasoning on \(W_{b,S}^{(n)}\) when an appropriate \(n\) is found.

Though new variables can be created in \(S\), at the end of each iteration, they get removed by the restriction operator \(\rhd\).

Following up on Lemma 4.4, we establish the computability result for our command language:

**Lemma 4.7.** The denotation of any well-formed command is a computable partial multifunction. In other words, for any well-formed command \(\Gamma \vdash S \rhd \Delta\), its denotation \([\Gamma \vdash S \rhd \Delta] : [\Gamma] \to \mathcal{P}(\{\bot\})\) is computable.

**Proof.** The restriction operations are computable. Due to the computability of the primitives and the compositions, we only need to show that for any computable \(b : X \to \mathcal{P}(\{\bot\})\) and \(S : X \to \mathcal{P}(X)\), the least fixed point \(\text{lfp}(W_{b,S})\) is computable.

Due to Fact 2.4, we only need to consider an arbitrary \(\sigma\) where \(\bot \notin \text{lfp}(W_{b,S})(\sigma)\). Note that from Remark 4.6(4) this happens only when there is \(n \in \mathbb{N}\) such that \(\bot \notin W_{b,S}^{(n)}(\sigma)\).

Consider the following intuitive description of a type-2 machine \(\tau\) that computes on its input \(\varphi_{\sigma}\), a name of \(\sigma\), with accesses to the other machines \(\tau_b\) computing \(b\) and \(\tau_S\) computing \(S\):

1. Repeat:
2. \(\varphi_b := \tau_b(\sigma)\)
3. Iterate \(n := 0, 1, 2, \cdots\) until \(\varphi_b(n) = -1\) or 1
4. If \(\varphi_b(n) = 1\):
5. \(\varphi_{\sigma} := \tau_S(\sigma)\)
6. If \(\varphi_b(n) = -1:\)
7. return \(\varphi_{\sigma}\)
The machine $\tau$ may diverge when at line (2) when $\tau_b(\varphi_\sigma)$ diverges, at line (3) when $\varphi_b$ does not contain either 1 or $-1$, and at line (5) when $\tau_S(\varphi_\sigma)$ diverges. The claim is that this machine $\tau$ realizes $\text{lfp}(W(b, S))$.

Consider another machine $\tau_i$ for each $i$ as the machine with the following line added at the beginning of each iteration: “if $i = 0$, diverge. $i := i - 1$”. Note that $\tau_i$ simulates the first $i$ iterations of the loop of $\tau$ then diverges unless the simulation returns within the $i$ iterations. We can easily see that by induction on $i$, $\tau_i$ computes $W_{b,S}^{(i)}$ for each $i$. Note also that when $\tau_i(\varphi_\sigma)$ does not fail, it prints the same stream that $\tau(\varphi_\sigma)$ prints.

Hence, for an initial $\sigma$ and its name $\varphi_\sigma$ such that $\text{lfp}(W(b, S))(\sigma) = W_{b,S}^{(n)}(\sigma) \neq \bot$, $\tau_n(\varphi_\sigma)$ prints a name of $y \in W_{b,S}^{(n)}(\sigma)$ which is what $\tau(\varphi_\sigma)$ prints. 

\begin{proof}

4.3. Denotations of Programs. Having defined the meanings of terms and commands, we are now ready to define denotations of well-typed ERC programs.

Definition 4.8. The denotation of the well-typed integer program

$$P_z := \text{input } v_1 : \tau_1, v_2 : \tau_2, \ldots, v_d : \tau_d \quad S \quad \text{return } t$$

is a partial integer multi-function

$$[P_z] : [\tau_1] \times \cdots \times [\tau_d] \to \mathbb{P}(\mathbb{Z}_\bot)$$

defined by $[P_z] := [t]^\dagger \circ [S]$.

And, the denotation of the well-typed real program

$$P_r := \text{input } v_1 : \tau_1, v_2 : \tau_2, \ldots, v_d : \tau_d \quad S \quad \text{return } t \text{ as } p \to -\infty$$

is a partial real function $[P_r] : [\tau_1] \times \cdots \times [\tau_d] \to \mathbb{R}_\bot$ that is defined by

$$[P_r] (x_1, \ldots, x_d) := \begin{cases} x & \text{if } \forall p \in \mathbb{Z}, \forall y \in [t]^\dagger \circ [S] (p, x_1, \ldots, x_d), y \neq \bot \land |y - x| \leq 2^p, \\ \bot & \text{otherwise.} \end{cases}$$

In words, a real program $P_r$ denotes a single-valued function $f$ whose function value $f(x_1, \ldots, x_d)$ is the limit of the multi-valued sequence $[t]^\dagger \circ [S] (p, x_1, \ldots, x_d)$ as $p \to -\infty$ in the sense that for each $i \in \mathbb{N}$, whichever $y_i$ is picked from $[t]^\dagger \circ [S] (-i, x_1, \ldots, x_d)$ nondeterministically, $y_i$ is a $2^{-i}$ approximation to $f(x_1, \ldots, x_d)$ and not $\bot$. The obligated rate of convergence $2^p$ is chosen to make it coincide with the precision embedding $\iota : \mathbb{Z} \ni p \mapsto 2^p \in \mathbb{R}$.

Theorem 4.9. For any well-typed ERC program, its denotation is either a computable partial real function or a computable partial integer multi-function.

Proof. The computability of terms, commands, and compositions directly yields the computability of the denotations of integer programs.

For $P_r$, note that $[t]^\dagger \circ [S] : \mathbb{Z} \times [\tau_1] \times \cdots \times [\tau_d] \to \mathbb{P}(\mathbb{R}_\bot)$ is a computable multi-function. We provide an informal description of a machine computing $[P_r]$ as follows. Suppose it
receives names $\varphi_1, \cdots, \varphi_d$ of $x_1, \cdots, x_d$. For each $i$'th entry of its output tape, we let the machine to simulate a machine computing $[t]^{\dagger} \circ [S]$ on the name of $(-(i + 2), x_1, \cdots, x_d)$ constructed by the name of $-(i + 2) \in \mathbb{Z}$ and $\varphi_d, \cdots, \varphi_d$. The simulation either returns an infinite sequence or does not terminate. When the simulation succeeds resulting in an infinite sequence $\varphi$, we let our machine print the closest integer of $\varphi(i + 2)/4$ at the $i$'th output entry. Of course, it does wait forever if the simulation runs forever. (Note that getting the closet integer here only requires discrete rational arithmetic which is different from rounding real numbers discussed in Section 5.2.4.)

We only need to check that for any $(x_1, \cdots, x_d)$ and $x \in \mathbb{R}$ st.

$$\forall p \in \mathbb{Z}. \forall y \in [t^{\dagger}] \circ [S] (p, x_1, \cdots, x_d). y \neq \bot \land |y - x| \leq 2^p$$

holds, the $i$'th entry what the above machine prints is $z \in \mathbb{Z}$ st. $|x - z \cdot 2^{-i}| \leq 2^{-i}$ holds. For the $i$'th entry, it simulates $[t]^{\dagger} \circ [S] (- (i + 2), x_1, \cdots, x_d)$. By the assumption that $\bot$ is not in it, the simulation must succeed and return an infinite sequence $\varphi$ which is a name of some $y \in [t^{\dagger}] \circ [S] (- (i + 2), x_1, \cdots, x_d)$ st. $|y - x| \leq 2^{-(i+2)}$ holds. Recall that since $z$ is obtained by rounding $\varphi(i + 2)/4$, it holds that $|z - \varphi(i + 2)| \leq 2^{-1}$. And, since $\varphi$ is a name of $y$, it holds that $|y - \varphi(i + 2) \cdot 2^{-(i+2)}| \leq 2^{-(i+2)}$. Therefore, we have $|x - z \cdot 2^{-i}| \leq |x - y| + |y - \varphi(i + 2) \cdot 2^{-(i+2)}| + 2^{-i}|\varphi(i + 2) \cdot 2^{-(i+2)} - z| \leq 2^{-(i+2)} + 2^{-(i+2)} + 2^{-(i+1)} \leq 2^{-i}$.

Recall from Fact 2.4 that in computable analysis, a machine computing a (multi-)function can be seen to compute some other (multi-)functions at the same time. Therefore, reflecting this property, we add one more layer on the denotations of programs:

**Definition 4.10.**

1. A well-typed real program $P$ realizes a partial function $f : X_1 \times \cdots \times X_d \rightarrow \mathbb{R}_\bot$ if $\\forall (x_1, \cdots, x_d). f(x_1, \cdots, x_d) \leq [P] (x_1, \cdots, x_d)$ holds. Recall that $x \leq y$ if and only if $x = \bot$ or $x = y$ for $x, y \in \mathbb{R}_\bot$.

2. A well-typed integer program $P$ realizes a partial (possibly infinite) multi-function $f : X_1 \times \cdots \times X_d \rightarrow \mathcal{P}(\mathbb{Z}_\bot)$ if $\forall (x_1, \cdots, x_d). f(x_1, \cdots, x_d) \subseteq [P] (x_1, \cdots, x_d)$ or $[P] (x_1, \cdots, x_d) \subseteq f(x_1, \cdots, x_d)$ holds.

Note that though the denotations of integer programs are finite multi-functions, it is defined such that they can realize infinite multi-functions. Directly from Fact 2.5, we get the computability of realizable partial (multi-)functions:

**Corollary 4.11.** Realizable partial (multi-)functions are computable.

5. PROGRAMMING IN Exact Real Computation

Based on programming abbreviations defined in Section 5.1, in Section 5.2 we collect several examples of (multi-)functions that are realizable in ERC. They illustrate the purpose of ERC which is to allow and justify naively implementing numerical algorithms, with computable analysis as the rigorous but hidden theoretical foundation. They demonstrate how (naturally) the classical algorithms can be modified to incorporate the Kleenean-valued comparisons and the nondeterministic choose.

The examples are square root using Heron’s method, exponential function via two different ways, integer rounding, matrix determinants via Gauss elimination, and root finding,
For a Kleenean typed term

For a real typed term

For an integer

For any real typed terms

For any integer terms

For an integer typed term

For any two term

Definition 5.1.

are used for example programs:

Programming Abbreviations. Throughout this paper, the following abbreviations are used for example programs:

5.1. Programming Abbreviations. Throughout this paper, the following abbreviations are used for example programs:

Definition 5.1.

(1) For any two term $t_1$ and $t_2$, we write $t_1 - t_2$ as an abbreviation for $t_1 + -t_2$, $t_1/t_2$ as an abbreviation for $t_1 \times t_2^{-1}$, $t_1 \geq t_2$ as an abbreviation for $t_2 \leq t_1$, $t_1 > t_2$ as an abbreviation for $t_2 < t_1$, $t_1 > t_2$ as an abbreviation of $-(t_2 \geq t_1)$, and $t_1 < t_2$ as an abbreviation of $t_2 > t_1$. For any term $t$, we write $2^t$ as an abbreviation for $i(t)$.

(2) For an integer typed term $j$ and a constant integer $d \in \mathbb{Z}$, $d \cdot j$ and $j \cdot d$ both mean $j + j + \cdots + j$ in case $d \geq 0$, $(-j) \cdot (-d)$ otherwise.

(3) For an integer $k \in \mathbb{Z}$, when it is used as a real typed term, it is an abbreviation for $2^0 + \cdots + 2^0$, the $k$-times repeated addition of $2^0$, when $k > 0$, $-(-k)$ when $k < 0$, and $2^0 - 2^0$ when $k = 0$.

(4) For any real typed terms $x, y, z$, max($x, y$) is an abbreviation for $y \leq x ? x : y$ and abs($z$) is an abbreviation for $0 < z ? z : -z$.

(5) For any integer terms $k, l$ and a command $S$, for $n : Z = k$ to $l$ do $S$ end is short for

let $n : Z = k$; let $m : Z = l$; while $(n \leq m)$ do $S$; $n := n + 1$ end

with $n, m$ not modified in $S$.

(6) For a real typed term $x$ and an integer typed term $j$, $x^j$ is short for the result $y$ of the following command:

let $y : R = 1$; let $x' : R = x$; let $z : Z = j$;
for $n : Z = 1$ to $z$ do $y := y \times x'$ end;
for $n : Z = 1$ to $-z$ do $y := y/x'$ end

(7) For a Kleenean typed term $b$ and a command $S$, if $b$ then $S$ end is an abbreviation for

if $b$ then $S$ else skip end

Recall that skip is an instruction for doing nothing.
Arrays in ERC are one-dimensional of constant size with indices starting at 0. We implicitly simulate and identify a two-dimensional array $A = (A[i,j])_{i,j}$, having first dimension of size $d > i \geq 0$, with the one-dimensional array $A[i + j \cdot d]$; similarly for higher dimensions.

Recall that the programming language $\text{ERC}(\mathcal{F}, \mathcal{G})$ is defined relative to the sets of functions $\mathcal{F}$ and $\mathcal{G}$. When we want to extend the set $\mathcal{F}$ with the denotation of a real program $P$ (in $\text{ERC}(\mathcal{F}, \mathcal{G})$), we write $\text{ERC}(\mathcal{F} \cup \{P\}, \mathcal{G})$ for $\text{ERC}(\mathcal{F} \cup \{[P]\}, \mathcal{G})$. Similarly, when we have an integer program $P$ in $\text{ERC}(\mathcal{F}, \mathcal{G})$, we denote the extension with its denotation by $\text{ERC}(\mathcal{F}, \mathcal{G} \cup \{P\})$. These notations are well-defined due to the fact that the denotation of a real program is a computable partial real function from a product of either $\mathbb{Z}$ or $\mathbb{R}$ and the denotation of an integer program is a computable integer multi-function from a product of either $\mathbb{Z}$ or $\mathbb{R}$. Recall Convention 3.1 and Theorem 4.9.

When the sets are extended by the denotation of a program $P$, let us write $P(t_1, \ldots, t_d)$ for $J_P^K(t_1, \ldots, t_d)$ in the extended term language.

5.2. Example Programs. In this subsection, we introduce some ERC programs. We annotate assertions next to commands in order to convey the behaviours, including the domains (preconditions), of the introduced programs. (we do not impose any obligation to programs on their behaviours when the preconditions are not met.) However, note that the annotations are not part of ERC.

5.2.1. Square Root Function via Heron’s Method. Heron’s method approximates the square root of a given real number $x \geq 0$ up to any desired absolute error $2^p$, $p \in \mathbb{Z}$. To this end calculate a contracting sequence of upper and lower approximations $y_n \geq \sqrt{x} \geq z_n$ by iteratively taking the average $y_{n+1} := (y_n + z_n)/2$.

The ERC program $\text{HeronSqrt}$ illustrated in Figure 3 realizes the square root function via Heron’s method. Recall that the arguments to $\text{choose}$ are indexed from 0, so $\text{choose}(b_0, b_1) = 1$ means that the second argument $b_1$ must evaluate to $true$.

```
input x : R
let y : R = 1; let z : R = x/y;
while choose(2^p \geq y - z , y - z \geq 2^{p-1}) = 1 do
  y := (y + z)/2; z := x/y
end
return y as p \rightarrow -\infty
```

The ERC program employs the multi-valued $\text{choose}()$ operation applied to two Kleenean-valued tests “$2^p \geq y - z$” and “$y - z \geq 2^{p-1}$”. Note that at least one of both inequalities “$2^p \geq y - z$” and “$y - z \geq 2^{p-1}$” is always $true$ (and in particular not $unknown$), resulting in a total condition for the $\text{while}$ loop. And when the loop terminates, the first test (corresponding to return value 0) must (while the second test, corresponding to return value
may or may not) be true: guaranteed to return an approximation \(y\) to \(\sqrt{x}\) up to absolute error \(2^p\).

5.2.2. Exponential Function via Taylor Expansion. The exponential function has a globally converging Taylor expansion

\[
\exp(x) = \sum_{j=0}^{\infty} \frac{x^j}{j!}
\]  

A ERC real program computing it must return, given a dedicated precision parameter \(p \in \mathbb{Z}\) and argument \(x \in \mathbb{R}\), an approximation to \(\exp(x)\) up to error \(2^p\). For \(|x| \leq 1\) and positive \(n \in \mathbb{N}\), the tail bound

\[
\left| \sum_{j>n} \frac{x^j}{j!} \right| \leq \sum_{j>n} \left| x \right|^j \frac{1}{j!} \leq \sum_{j>n} 2^{-j+1} = 2^{-n+1}
\]

justifies the straightforward algorithm as in Figure 4.

\begin{figure}
\centering
\begin{verbatim}
input x : R          // -1 ≤ x ≤ 1
let j : Z = 1;      let j_r : R = 1 let f : R = 1;      // j ≡ j_r, f ≡ j!
let y : R = 1;      let z : R = x;                        // z ≡ x^j
while j ≤ −p + 1 do
  y := y + z/f;
  j := j + 1;
  j_r := j_r + 1;
  z := z × x;
  f := f × j_r;
end
return y as p → −∞          // |y − exp(x)| ≤ 2^p
\end{verbatim}
\caption{pExp : R → R}
\end{figure}

Observe that the two variables \(j : \mathbb{Z}\) and \(j_r : \mathbb{R}\) in Figure 5 retain identical values but have carefully distinguished types.

The following wrapper \(\text{posExp}\) for \(\text{pExp}\) in Figure 5, which is a program in \(\text{ERC}(\{\text{pExp}\}, \emptyset)\) (Recall Definition 5.1 (9)), removes the restriction \(|x| \leq 1\) and instead works for all \(x \geq −1\).

\begin{figure}
\centering
\begin{verbatim}
input x : R          // -1 ≤ x
let z : R = pExp(1/2);
let y : R = 1;
while choose(x < 1, x > 1/2) = 1 do
  y := y × z;
  x := x − 1/2
end
return y × pExp(x) as p → −∞
\end{verbatim}
\caption{posExp : R → R}
\end{figure}
Like Heron’s method, the loop employs “choose()” with overlapping conditions $x < 1$ and $x > 1/2$ in order to guarantee totality. Also, since for $x < 0$ it holds that $\exp(x) = 1/\exp(-x)$, and $\exp(y) = 1/\exp(y)$ at $y = 0$, we can construct a procedure that computes $\exp(x)$ for all $x \in \mathbb{R}$ simply by ‘$x > 0$? $\text{posExp}(x) : 1/\text{posExp}(-x)$’.

5.2.3. Exponential Function via Iteration. \texttt{pExp} in Section 5.2.2 employs unbounded sums, which leave the realm of the first-order logic considered in Section 6 for formal specification and verification purposes. Here we consider an alternative, iterative approach to realize the exponential function in ERC. To this end recall for every $x \in [0; 2]$ it holds [Mit70, §3.6.3]:

$$\exp(x) \xrightarrow{n \to \infty} (1 + \frac{x}{n})^n \leq \exp(x) \leq (1 + \frac{x}{n})^{n+1} \xrightarrow{n \to \infty} \exp(x)$$

This suggests the following iterative ERC program:

\texttt{Figure 6: } \texttt{iExp : R \to R}

\begin{verbatim}
input x : R
  // 0 \leq x \leq 2
  let n : Z = 1;
  let nr : R = 1;
  let c : R = 1 + x; let a : R = c; let b : R = a \times c;
  while choose(2^p \geq b - a, b - a \geq 2^{p-1}) = 1 do
    n := n + n;
    nr := 2 \times nr;
    c := 1 + x/nr; a := c^n; b := a \times c
  end
  return a as p \to -\infty
\end{verbatim}

Note that, similarly to Figure 4, variables $n \equiv nr$ of different types are maintained to coincide in value. Moreover “$a := c^n$” is to be understood as abbreviating a loop of $n$ real multiplications. (Recall Definition 5.1.)

The program in Figure 6 supposes $0 \leq x \leq 2$, but can be extended to the entire real line similar to Section 5.2.2. Note that the \texttt{while} loop is guaranteed to terminate since $b - a$ converges to 0 as $n$ grows. The \texttt{while} condition is total: At least one of ‘$2^p \geq b - a$’ and ‘$b - a \geq 2^{p-1}$’ is always true. And when the \texttt{while} loop terminates, then it must hold $2^p > b - a$, hence $|a - \exp(x)| \leq |b - a| \leq 2^p$.

5.2.4. Integer Rounding Multi-function. All real-to-integer rounding functions (up, down, to nearest) are discontinuous and therefore not computable. We thus relax the specification and instead consider the following multi-function with ‘overlap’:

$$\texttt{Round : } \mathbb{R} \ni x \mapsto \{k \in \mathbb{Z} \mid x - 1 < k < x + 1\} \subseteq \mathbb{Z}$$ (5.3)

The ERC program \texttt{Round} in Figure 7 realizes \texttt{Round}. It returns, given a (not necessarily positive) real $x$, some integer $k$ such that $|x - k| < 1$. 

Intuitively, the ‘number of steps made’ by Round are proportional to the value of the argument $x$, that is, exponential in its binary (≈output) length: because Round essentially counts up to $x$.

Recovering the rounded integer bitwise via binary search seems exponentially more efficient, but fails due to lack of continuity: Extracting any digit of ‘the’ binary expansion (or one of the at most two possible ones) of a given real number is uncomputable [Tur37]. Instead, BinRound in Figure 8 realizes the idea that some signed-digit expansion [Wei00, Definition 7.2.4] of a given real number $x \in \mathbb{R}$ can be determined computably:

Loop invariants have been included into the code as informal comments in order to convey partial correctness with respect to the specified postcondition; separate work will expand these for formal verification. Due to multi-valuedness of the test, after the real number while loop (lines 3 to 5) has ended, the second argument “$y \geq 1/2$” may still be true, whereas the first “$|y| < 1$” must be true; but always at least one of both is valid, thus guaranteeing total correctness. In the integer loop (lines 6 to 10), multi-valuedness ‘strikes’ only at line 8 which
employs \texttt{choose()} with trinary argument. “\( y - b \)” in line 9 is to be understood as abbreviation of the following code:

```
if \( b = -1 \) then \( y := y + 1 \) end; if \( b = 1 \) then \( y := y - 1 \) end
```

5.2.5. **Determinant Function via Gaussian Elimination.** The determinant of a \( d \times d \) matrix \( A = (a_{ij})_{i,j} \) is given by Leibniz’ formula

\[
\det(A) = \sum_\pi \text{sign}(\pi) \cdot \prod_{j=1}^d a_{j,\pi(j)}
\]

where the sum ranges over all \( d! \) permutations \( \pi : \{1, \ldots, d\} \rightarrow \{1, \ldots, d\} \). Since ERC conveniently relieves the programmer from numerical issues like cancellation, this formula gives rise to a straight-forward ERC program. However one executing a number of arithmetic operations exponential in \( d \).

Common numerical approaches therefore transform \( A \) to triangular form, whose determinant is simply the product of its diagonal elements \[PTVF07, \S 2.3.3\]. More formally, following Turing \[Tur48\], apply Gaussian Elimination in order to determine a \( LU \) factorization with full pivoting \( P \cdot A \cdot Q = L \cdot U \) of \( A \), where \( P, Q \) denote permutation matrices and \( L \) and \( U \) are lower and upper triangular matrices, respectively. Error propagation through Gaussian Elimination with its variants are non-trivial topics in numerical analysis \[TB97, \S IV\] that hamper rigorous specification when implemented in floating point arithmetic. Exact arithmetic in ERC on the other hand eliminates the need for such considerations—except for pivot search, which involves real comparisons that are only partially defined. In Gauss’ original algorithm, such search either (i) returns the index of a non-zero entry in the given sub-matrix or (ii) asserts that said sub-matrix is identically zero. By iterating this process, Gaussian Elimination determines the rank \( k \in \mathbb{N} \) of the original matrix \( A \in \mathbb{R}^{d \times d} \)—which depends discontinuously on \( A \)'s entries and hence cannot be computed.

**Remark 5.2.** We thus change the specification of the determinant (and of the \( LUPQ \) factorization\(^1\) it builds on): with the promise for the argument matrix \( A \) to have full rank—otherwise its determinant will vanish, anyway.

\[
\text{Det}_d : \text{GL}(\mathbb{R}^d) \ni A \mapsto \begin{cases} 
\det(A) & \text{if } \det(A) \neq 0, \\
\bot & \text{otherwise.}
\end{cases}
\]  

(5.5)

We also relax pivot search (within the lower-right submatrix) to become a partial multi-function:

\[
\text{Pivot}_d(A,k) = \{(i,j) : A \in \mathbb{R}^{d \times d} \land k \leq i,j < d \land A_{i,j} \neq 0\}_{\{\perp\}} \in \mathbb{P}(\mathbb{Z}_+^2)
\]

(5.6)

Here, \( X_{\{\perp\}} \) := \( X \) if \( X \neq \emptyset \) and \( \{\perp\} \) if \( X = \emptyset \). So the argument is a real \( d \times d \) matrix \( A \) and an integer \( k \), indicating that a pivot is to be sought for in the \( (d-k) \times (d-k) \) non-zero sub-matrix \( A[k \ldots d-1, k \ldots d-1] := (A[i,j])_{k \leq i,j < d} \).

The pair \( (i_0, j_0) \) of return values here is thus understood as abbreviation of the single integer \( i_0 + j_0 \cdot d \), similarly to the two-dimensional array access \( A[i,j] \) as abbreviation of \( A[i + j \cdot d] \). Recall Definition 5.1(8).

Based on \( \text{Pivot}_d \), the ERC program in Figure 10 computes the non-zero determinant \( \text{Det}_d \) from Equation (5.5) via \( LUP \) decomposition with full pivoting. Note that the precision

\(^1\)LUPQ factorization \( A \mapsto (L, U, P, Q) \) is not unique, hence a real matrix-tuple-valued multi-function.
Figure 9: \( \text{Pivot}_d : \mathbb{R}[d \times d] \times \mathbb{Z} \rightarrow \mathbb{Z} \)

1: \textbf{input} \( A : \mathbb{R}[d \times d] \), \( k : \mathbb{Z} \)
2: \text{let} \( i_0 : Z = k \); \text{let} \( j_0 : Z = k \); \text{let} \( x : R = 0 \);
3: \text{for} \( i : Z = k \) to \( d - 1 \) \text{do}
4: \quad \text{for} \( j : Z = k \) to \( d - 1 \) \text{do}
5: \quad \quad \text{x := max}(x, \text{abs}(A[i,j])) \text{ end end;}
6: \quad \text{for} \( i : Z = k \) to \( d - 1 \) \text{do}
7: \quad \quad \text{if} \( \text{choose}(|A[i,j]| < x, |A[i,j]| > x/2) = 1 \) then \( i_0 := i; j_0 := j \) end
8: \quad \text{end end}
9: \text{return} \((i_0, j_0)\)

parameter \( p \in \mathbb{Z} \) is present but ignored since the result gets computed exactly. Again, formally speaking the program \( \text{Det}_d \) is a program in ERC(\( \emptyset \), \{\text{Pivot}_d\}).

Figure 10: \( \text{Det} : \mathbb{R}[d \times d] \rightarrow \mathbb{R} \)

1: \textbf{input} \( A : \mathbb{R}[d \times d] \) \quad // \( A \) invertible, \( p \) ignored
2: \text{let} \( i : Z = 0 \); \text{let} \( j : Z = 0 \); \text{let} \( k : Z = 0 \);
3: \text{let} \( p_i : Z = 0 \); \text{let} \( p_j : Z = 0 \); \text{let} \( \text{det} : R = 1 \); \quad // ret.val
4: \text{for} \( k := 0 \) to \( d - 2 \) \text{do}
5: \quad \text{// Convert \( A[k..d-1,k..d-1] \) to reduced row echelon form:}
6: \quad \text{(p_i, p_j) := Pivot(A, k); } \quad // p_i, p_j \geq k \text{ s.t. } A[p_i, p_j] \neq 0.
7: \quad \text{det := det} \times A[p_i, p_j];
8: \quad \text{for} \( j := 0 \) to \( d - 1 \) \text{do swap}(A[k,j], A[p_i,j]) \text{ end;}
\quad \text{// Exchange rows \# k and \# p_i.}
9: \quad \text{if} \( k \neq p_i \) \text{then det := -det; } \quad // \text{flip sign}
10: \quad \text{for} \( i := 0 \) to \( d - 1 \) \text{do swap}(A[i,k], A[i,p_j]) \text{ end;}
\quad \text{// Exchange columns \# k and \# p_j.}
11: \quad \text{if} \( k \neq p_j \) \text{then det := -det; } \quad // \text{flip sign}
12: \quad \text{for} \( j := k + 1 \) to \( d - 1 \) \text{do } // \text{Scale row \# k by } 1/A[k,k] \text{ and}
13: \quad \quad \text{A}[k,j] := A[k,j]/A[k,k]; \quad // \text{and subtract the } A[i,k]-\text{fold from}
14: \quad \quad \text{for} \( i := k + 1 \) to \( d - 1 \) \text{do } // \text{from rows \# i = k + 1...d - 1.}
15: \quad \quad \quad \text{do } A[i,j] := A[i,j] - A[i,k] \times A[k,j] \text{ end}
16: \quad \quad \text{end; A}[k,k] := 1; \quad \text{for} \( i := k + 1 \) to \( d - 1 \) \text{do } A[i,k] := 0 \text{ end}
17: \quad \text{end;}
18: \quad \text{det := det} \times A[d - 1, d - 1];
19: \text{return \( \text{det as } p \rightarrow -\infty \)}

Note that pivot search in line 6 of program \( \text{Det} \) is guaranteed to succeed in that the \( (d-k) \times (d-k) \) submatrix \( A[k\ldots d-1, k\ldots d-1] \) under consideration will indeed contain
at least one — usually non-unique — non-zero element due to the promise/restriction that $A \in \text{GL}(\mathbb{R}^d)$ according to Remark 5.2.

5.2.6. Simple Unique Root Finding Functional. The problem of finding (i.e. approximating) a root to a given real function $f$ occurs ubiquitously in numerics under various hypotheses. Here we consider algorithmic versions of the Intermediate Value Theorem, that is, finding a root to a given continuous function $f : [a; b] \rightarrow \mathbb{R}$ satisfying $f(a) < 0 < f(b)$. This case is commonly treated using Bisection: Determine the sign of $f(x)$ at the interval mid point $x := (a + b)/2$ and recurse to either $[a; x]$ or to $[x; b]$ accordingly. However, since equality is undecidable, the sign test fails in case $f(x) = 0$. Trisection [Her96, p. 336] instead considers the signs of $f$ at both one third $x' := (2a + b)/3$ and at two third $x'' := (a + 2b)/3$ of the interval, in parallel; and recurses to either $[a; x']$ or to $[x''; b]$ accordingly: Now at most one of the two sign tests at $f(x')$ and $f(x'')$ can fail, provided that $f$’s root is unique and $f(a) \cdot f(b) < 0$. These hypotheses also avoid common counterexamples like [Spe59] or [Wei00, Theorem 6.3.2]. To summarize, we consider the (single-valued) root finding problem Root : $f \mapsto x$ with $f(x) = 0$ for $f$ satisfying the following first-order predicates:

\[
\begin{align*}
\text{cont}(f,a,b) &:= \forall \epsilon > 0. \exists \delta > 0. \forall x, x'. a \leq x \leq x' \leq x + \delta \leq b \Rightarrow |f(x) - f(x')| \leq \epsilon \\
\text{uniq}(f,a,b) &:= \text{cont}(f,a,b) \land f(a) \cdot f(b) < 0 \land \exists x. a < x < b \land f(x) = 0
\end{align*}
\]

See that cont$(f,a,b)$ says $f$ is continuous in the usual sense on the interval $[a; b]$ and uniq$(f,a,b)$ says $f$ is continuous in the interval $[a; b]$, admits a unique root in the interval $(a; b)$, and the signs of $f(a)$ and $f(b)$ are different.

The ERC program in Section 11 annotated with precondition uniq$(f,a,b)$ and postcondition uniq$(f,a,b) \land |b-a| \leq 2^p$, as well as loop invariant uniq$(f,a,b)$. Formally speaking, as ERC does not accept function arguments, the program is defined in ERC($\{f\}$, $\emptyset$).

Given $p$, the postcondition guarantees that Trisection indeed returns an approximation to the root up to error $2^p$ when uniq$(f,a,b)$ holds.

Figure 11: Trisection : $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ in ERC($\{f\}$, $\emptyset$) for any $f : \mathbb{R} \rightarrow \mathbb{R}_-$

\begin{verbatim}
1: input a : R, b : R                        // uniq(f, a, b)
2: while choose(2^p > b - a, b - a > 2^p-1) = 1 do // uniq(f, a, b) \land b - a > 2^{p-1}
3:   if choose(0 > f(b/3 + 2 * a/3) \times f(b), 0 > f(a) \times f(2 \times b/3 + a/3)) = 1
4:   then b := b/3 + 2 * a/3
5:   else a := b/3 + 2 * a/3  end
6: end  // uniq(f, a, b) \land |b - a| \leq 2^p
7: return a as p \rightarrow -\infty
\end{verbatim}

Recall the abbreviations $b/3$ and $2 \times a/3$ form Definition 5.1.
This program is chosen later in Section 7 to be formally proved for its correctness.
5.3. **Turing-Completeness.** Extending and building on the above examples of programming in ERC, we now establish that ERC is expressive enough to deal with any computable real functions and integer multi-functions. In other words, it is Turing-complete in the following sense.

**Theorem 5.3.**

1. Every computable partial real function \( f \) from integers and reals can be realized in ERC\(_0\).
2. Similarly, every computable partial integer multi-function \( g \) from integers and reals can be realized in ERC\(_0\).

**Proof.** Note that though ERC\(_0\) does not provide integer multiplications in its term language, as its command language provides loops, when we restrict ERC\(_0\) with integers only, without real numbers or nondeterminism, the expressive power of the command language of ERC\(_0\) is equivalent to the ordinary Turing machine.

Our command language, being powerful enough to simulate Turing machines, can also implement finite integer map structures; we can have integer variables which encode a finite pairs \((a_1 : b_1, \cdots, a_n : b_n)\) with the functionality of searching by key \(a_i\) and adding key-value pairs \((a_i : b_i)\). Also note that by repeated addition or subtraction, the ordinary integer embedding \(\mathbb{Z} \ni z \mapsto z \in \mathbb{R}\) is expressible through the command language.

By definition, for a computable partial function \( f : X_1 \times \cdots \times X_d \to \mathbb{R}_{\bot} \), there is an oracle machine such that when it is equipped with oracles \(\varphi_1, \cdots, \varphi_d\), which are names of \(x_1, \cdots, x_d\) s.t. \(f(x_1, \cdots, x_d) \neq \bot\), there is a name \(\varphi\) of the function value s.t. \(\forall i \cdot |f(x_1, \cdots, x_d) - \varphi(i) \cdot 2^{-i}| \leq 2^{-i}\) such that when the machine receives an integer \(m\) as its input, returns \(\varphi(m)\).

We can modify the oracle machine to define an ERC\(_0\) program with input variables \(x_1, \cdots, x_d\) and a precision parameter \(p\). In the beginning, let an integer variable \(m\) which is recognized as the input of the oracle machine be initialized with \(-p\) by let \(m : Z = -p\). For each real and real array input variable, we create an empty map.

For the rest of the program, we simulate the oracle machine with the initialized variable \(m\). For each oracle query asking for the \(n\)th entry of its \(i\)th oracle, which is meant to be a name of the \(i\)th input \(x_i\), we make the following replacement:

- If \(x_i\) is an integer variable, we replace (the answer of) the query with \(x_i\).
- If \(x_i\) is a real variable, (1) check if \((n : z)\) is in the map of \(x_i\) for some \(z\), (2) if so replace (the answer of) the query with \(z\), and (3) otherwise perform the multi-valued rounding on \(x_i \times 2^n\) (from Section 5.2.4) and store the pair \((n, z)\) where \(z\) is the resulting term of the rounding, in the map of \(x_i\). (Since a map is essentially variables, at this moment, it creates multiple states for each possible rounding of \(x_i \times 2^n\).) Then, perform (1) above.
- If \(x_i\) is an array \(R[k]\) variable, (1) check if \((n : z)\) is in the map of \(x_i\) for some \(z\), (2) if so replace (the answer of) the query with \(z\), and (3) otherwise, we first using the command language, find the indices \(0 \leq j < k, 0 \leq l\) such that \(k \times l + j = n\), (note that this is the decoding of the index embedding for product representation in Section 2) and (4) perform the multi-valued rounding on \(x_i[j] \times 2^l\) and store the pair \((n, z)\) where \(z\) is the resulting term of the rounding, in the map of \(x_i\). (Again, since a map is essentially variables, at this moment, it creates multiple states for each possible rounding of \(x_i[j] \times 2^l\)) Then, perform (1) above.

When the oracle machine terminates, when an integer term \(t\) is about to be returned, first embed the integer term \(t\) to reals \(t'\) using our command language, then return \(t' \times 2^p\).
Observe that for any input \(x_i\), the replacement of accessing the \(n\)th entry of the corresponding oracle as above correctly but nondeterministically provides the \(n\)th entry of names of \(x_i\). Thanks to the maps, once an oracle is accessed \(\varphi_i(n)\), the future access with the same query gets consistent.

For example, when a query is made for the \(n\)th entry of \(x_i\) for the first time, it may create two different states \(\{\sigma_1, \sigma_2\}\) where in \(\sigma_1\), the map for \(x_i\) stores \((n : \varphi_i(n))\) and in \(\sigma_2\), the map for \(x_i\) stores \((n : \varphi'_i(n))\) such that \(\varphi_i\) and \(\varphi'_i\) are different names of \(x_i\). However, for either \(\sigma_1\) or \(\sigma_2\), when the same query is made, the answer is consistent in the sense that for \(\sigma_1\), it accesses \(\varphi_i(n)\) and for \(\sigma_2\), it accesses \(\varphi'_i(n)\).

Note that the representations we use are defined in a way that the entries of names are independent in the sense that for any \(x \in \mathbb{Z}, \mathbb{R}\), or \(\mathbb{R}^k\), and for any partial function \(h : \subseteq \mathbb{N} \to \mathbb{Z}\) such that for all \(j \in \text{dom} h\), there exists a name \(\varphi_j\) of \(x\) such that \(h(j) = \varphi_j(j)\), there is a name \(\varphi\) of \(x\) such that \(\forall j \in \text{dom} h. h(j) = \varphi(j)\). In words, for any \(x\), choosing a name \(\varphi\) of \(x\), the choices of each entry of \(\varphi\) are independent.

Hence, the replacement results in multiple states \(\Sigma\) where in each \(\sigma \in \Sigma\) and each real or real array input \(x\), there is a name \(\varphi\) of \(x\) where the map of \(x\) stores a segment of \(\varphi\). It means there are names \(\varphi_1, \ldots, \varphi_d\) of \(x_1, \ldots, x_d\) such that the computation leading to \(\sigma\) is equivalent to the oracle machine equipped with \(\varphi_1, \ldots, \varphi_d\) as its oracles.

Therefore, the denotation of the original integer return term \(t\) is a subset of the oracle machine's return values over all possible oracles (names) \(\varphi_1, \ldots, \varphi_d\) for \(x_1, \ldots, x_d\): a set of integer \(z\) such that \(z \cdot 2^p\) approximates the function value by \(2^p\). (Recall that we make \(m := -p\) in the beginning of the program.) Therefore, after embedding it in reals and multiplying it by \(2^p\) at the end, the denotation of the return term is a set of real numbers that are \(2^p\) approximations to the function value. Therefore, the denotation of the program at any \((x_1, \ldots, x_d)\) when \(f(x_1, \ldots, x_d) \neq \bot\) is \(f(x_1, \ldots, x_d)\); the program realizes \(f\) according to Definition 4.10.

The integer multi-function case can be proved identically except for that we do not need to get bothered with the type casting at the end.

\[\square\]

**Remark 5.4.** The completeness of WhileCC*’s algebraic semantics [TZ04] is obtained in a similar way: using nondeterminism to obtain discrete approximation then proceed ordinary computation. However, their countable choice construct is not necessary for our case as we only consider real numbers; using the total order of reals, we can rely on our finite choice construct to extract names.

ERC defines integer multi-functions but not a real multi-function. Real programs are forced to take limit operations and compute single-valued real numbers. This is our design choice that can be chosen differently. Here, we state a possible alternative and explain our choice:

**Remark 5.5.** One straightforward way to allow real multi-functions is to ease our typing rule such that we let the program

\[
P := \text{input } v_1 : \tau_1, v_2 : \tau_2, \ldots, v_d : \tau_d
\]

\[
S
\]

\[
\text{return } t
\]

have type \(\tau_1 \times \cdots \tau_d \to \mathbb{R}\) if there is a context \(\Gamma'\) such that

\[
v_1 : \tau_1, v_2 : \tau_2, \ldots, v_d : \tau_d \vdash S \Gamma' \quad \text{and} \quad \Gamma' \vdash t : \mathbb{R}.
\]
And, we let its denotation be $[P] := \{[t]^1 \circ [S] : [\tau_1] \times \cdots \times [\tau_d] \rightarrow \mathcal{P}(\mathbb{R}_\perp)\}$ as it was an integer programs. From Lemma 4.4 and 4.7, it can be seen immediately that the denotations of such real programs are computable multi-functions. We can also extend the notion of “realization” in Definition 4.10 accordingly.

However, this extension does not carry the completeness in Theorem 5.3 over. Recall that the denotations of programs are finite multi-functions. Hence, a computable infinite multi-function is realizable in ERC only if it admits point-wise finite refinement that is computable. In the integer case, it was fine as any infinite integer multi-function admits point-wise finite refinement. (Recall Fact 2.5.) However, it does not hold for real multi-functions. Therefore, when we extend ERC with real multi-functions as above, we end up with a restricted completeness theorem. As easing the restriction as above is straightforward, we choose to keep the restriction and achieve a more natural completeness theorem.

Recall that the ordinary (single-valued) limit operation was essential in having completeness for (single-valued) real functions. Multi-valued limit is still under debate [FB18, Kon18] and we leave it as future work to equip multi-valued limit operation in ERC and make it complete even for partial computable multi-valued real functions.

6. Logic of Exact Real Computation

In this section, we devise a specification language and based on the language devise an extended Hoare logic for formal verification of ERC programs. Section 6.1 proposes a three-sorted structure for rigorously specifying (multi-)functions with real arguments in Section 6.2. It is carefully designed rich enough to allow arguing about computations in ERC yet restricted such as to assert logical decidability as a guarantee to formal program verification; recall Remark 1.1. Section 6.3 extends the classical Hoare logic which is sound with regard to our denotational semantics.

6.1. Three-Sorted Structure with Decidable Theory. ERC natively supports three basic data types and operations: $\mathbb{Z}$ coinciding with the Presburger structure $\mathbb{Z}$, $\mathbb{R}$ coinciding with the ordered field $\mathbb{R}$, and $\mathbb{K}$ coinciding with the three-valued Kleene logic $\mathbb{K} = \{true, false, unknown\}$ extending Boolean $\mathbb{B} = \{true, false\}$.

**Definition 6.1.** The Structure of $\text{ERC}_0$ (without any extension) is the three-sorted structure $\mathcal{S}_0$ combining the Kleene Algebra $(\mathbb{K}, false, true, unknown, \land, \lor, \neg)$ with Presburger Arithmetic $(\mathbb{Z}, 0, 1, +, -, \leq, 2\mathbb{Z}, 3\mathbb{Z}, 4\mathbb{Z}, \ldots)$ and ordered field $(\mathbb{R}, 0, 1, +, -, \times, <)$. They are connected via the binary precision embedding $i : \mathbb{Z} \ni p \mapsto 2^p \in \mathbb{R}$ and its partial half-inverse $[\log_2 \circ \text{abs}] : \mathbb{R} \setminus \{0\} \rightarrow \mathbb{Z}$. Here $k\mathbb{Z}$ denotes the predicate on $\mathbb{Z}$ which is true precisely for all integer multiples of $k \in \mathbb{N}$.

The logic of ERC$_0$ is the first-order language $L_0$ of the structure $\mathcal{S}_0$; the theory of ERC$_0$ is the complete first-order theory $T_0$ of the structure $\mathcal{S}_0$.

We take the inequality on $\mathbb{Z}$ as non-strict $\leq$, but that on $\mathbb{R}$ strict $<$ as in [Mar17]. This classical predicate $< \subseteq \mathbb{R} \times \mathbb{R}$ is not to be confused with the computational, Kleenean-valued comparison $\ll : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{K}$. The symbol $\perp$ which represents ill-definiteness in our denotational semantics is not an element of the structure of ERC.

Theorem 6.2(a) shows that the theory of our base language $T_0$ is decidable: every first-order sentence about $\mathcal{S}_0$ can be formally either verified or refuted. This applies for
example to pre/post conditions or loop invariants of ‘plain’ ERC₀ programs; see Remarks 6.6 and 6.10. This is a significant advantage of ERC, compared to traditional programming languages for discrete data: Classical While programs over integers with multiplication for instance do suffer from Gödel undecidability [Coo78, §6].

**Theorem 6.2.**

a) The Theory \( T_0 \) of ERC₀ is decidable.
b) \( T_0 \) is also ‘model complete’ in that it admits elimination of quantifiers up to one (by choice either existential or universal) block ranging over integers.
c) Each of the following expansions destroys the decidability of the first-order theory of the structure \( S_0 \):

- expanding with integer multiplication
- expanding with the unary predicate \( \mathbb{Z} \) on \( \mathbb{R} \), or with ‘type casting’ \( \mathbb{Z} \hookrightarrow \mathbb{R} \)
- replacing the binary precision embedding \( i \) with its unary counterpart \( j : \mathbb{N}^+ \ni n \mapsto 1/n \in \mathbb{R} \)
- expanding simultaneously with the real exponential and sine function and with transcendental constants \( \pi \) and \( \ln 2 \)

**Proof.** c) Including integer multiplication recovers Peano arithmetic and Gödel undecidability via Robinson’s Theorem [Rob59]. A unary predicate \( \mathbb{Z} \) on \( \mathbb{R} \) allows to express integer multiplication via the reals; similarly for (any total extension of) the unary precision embedding \( j \). Finally, the real transcendental functions and constants make the theory undecidable according to Richardson’s Theorem [Ric68].

a)+b) A celebrated result of van den Dries [Dri86] extends classical Tarski-Seidenberg quantifier elimination from the first-order theory of real-closed fields to the expanded structure

\[
(\mathbb{R}, 0, 1, +, -, \times, <, 2^{k\mathbb{Z}} : k \in \mathbb{N}, 2^{[\log_2 \circ \text{abs}]})
\]  

with axiomatized additional predicates \( 2^{k\mathbb{Z}} \), \( k \in \mathbb{N} \), and truncation function to binary powers \( 2^{[\log_2 \circ \text{abs}]} \), see also [AY07].

Note that both the real-closed field \((\mathbb{R}, 0, 1, +, -, \times, <)\) and Presburger Arithmetic can be embedded into the expanded structure from Equation (6.1); the latter interpreted as its multiplicative variant \((2^{\mathbb{Z}}, 1, 2, \times, <, 2^{k\mathbb{Z}} : k \in \mathbb{N})\) is called *Skolem Arithmetic* [Bes02]:

- Replace quantifiers over Skolem integers with real quantifiers subject to the predicate \( 2^{k\mathbb{Z}} \) for \( k := 1 \);
- Consider \( i : \mathbb{Z} \to \mathbb{R} \) as the restricted identity \( \text{id}_{2\mathbb{Z}} \) in \( \mathbb{R} \).

Then every formula \( \varphi \) with or without parameters in our two-sorted structure translates signature by signature to an equivalent one \( \tilde{\varphi} \) over the expanded theory where quantifiers can be eliminated, yielding equivalent decidable \( \tilde{\psi} \) (which may involve binary truncation \( 2^{[\log_2 \circ \text{abs}]} \)).

To translate this back to some equivalent \( \psi \) over the two-sorted structure, while reintroducing only one type of quantifiers, observe that for real \( x \):

\[
\begin{align*}
x \in 2^{k\mathbb{Z}} & \iff \exists z \in \mathbb{Z}. z \in k\mathbb{Z} \land x = i(z); \\
x \notin 2^{k\mathbb{Z}} & \iff \exists z \in \mathbb{Z}. z \in k\mathbb{Z} \land i(z) < x < i(z + k).
\end{align*}
\]

Similarly, replace real binary truncation \( 2^{[\log_2 \circ \text{abs}(x)]} \) with “\( i(z) \)” for some/every \( z \in \mathbb{Z} \) s.t. \( i(z) \leq |x| < i(z) + 1 \) in case \( x > 0 \), with 0 otherwise.

The Kleene Algebra \( K \) as the third sort is finite and does not affect decidability.  

\( \square \)
We want to benefit from these nice properties of \( S_0 \) by using it in formally specifying the behaviour of ERC programs. For the purpose, there is a need to deal with the array variables from ERC_0 in \( S_0 \). We take the following straightforward translation, to express \( \mathbb{R}^n \) sorted variable with \( n \) different \( \mathbb{R} \) sorted variables, implicitly:

**Remark 6.3.** Consider \( S_0 \) expanded with, for each \( n = 1, 2, 3, \ldots \):

- a sort \( \mathbb{R}^n \),
- a \( n \)-ary function \( \text{id} : \mathbb{R} \times \cdots \times \mathbb{R} \to \mathbb{R}^n \) such that \( \text{id}(x_0, \cdots, x_{n-1}) = (x_0, \cdots, x_{n-1}) \), and
- the following predicates
  
  \[
  \text{proj} := \{(x_0, \cdots, x_{n-1}), k, y) \mid x_k = y\} \subseteq \mathbb{R}^n \times \mathbb{Z} \times \mathbb{R},
  \]
  
  \[
  \text{update} := \{(x_0, \cdots, x_{n-1}), k, y, (z_0, \cdots, z_{n-1})\} \mid z_k = y \land \forall i. i \neq k \Rightarrow x_i = z_i
  \]
  
\[
\subset \mathbb{R}^n \times \mathbb{Z} \times \mathbb{R} \times \mathbb{R}^n.
\]

Observe that these are the graphs of the partial projection and updating from Example 2.1 discarding inputs yielding \( \bot \).

We translate each finite set of sorted variables in the extended logic as follows:

- We translate the empty set to the empty set; \([\emptyset] := \emptyset\).
- For a variable \( v \in \mathcal{V} \) and a finite set of sorted variables \( \Gamma \), we define \([v : s, \Gamma] := v : s, [\Gamma]\) for each \( s \in \{\mathbb{K}, \mathbb{Z}, \mathbb{R}\}\)
- and \([v : \mathbb{R}^n, \Gamma] := v_0 : \mathbb{R}, \cdots, v_{n-1} : \mathbb{R}, [\Gamma]\) where \( v_0, \cdots, v_{n-1} \in \mathcal{V} \) are fresh variables that do not appear in \([\Gamma]\).

See that the set of variable assignments \([\Gamma]\) for any \( \Gamma \) in this extended structure is isomorphic to the \([\Gamma]\) in the original structure. Let \([\cdot] : [\Gamma] \to ([\Gamma])\) denote the isomorphism.

For each \( \mathbb{R}^n \) sorted term \( t \) with free variables \( \Gamma \), there are \( n \) terms \([t_0, \cdots, t_{n-1}]\) of sort \( \mathbb{R} \) with free variables \( \Gamma \) such that for each variable assignment \( \sigma \in [\Gamma] \), the interpretation \([t] \sigma\) of \( t \) is \(([[t_0]] \sigma, \cdots, [[t_{n-1}]] \sigma))

To see this, we only need to consider the case when \( t \) is a \( \mathbb{R}^n \) sorted variable or a term of the form \( \text{id}(x_0, \cdots, x_{n-1}) \). When \( t \) is a variable \( v \) of sort \( \mathbb{R}^n \) in \( \Gamma \), in \([\Gamma]\) we have \( v_0, \cdots, v_{n-1} \) for the desired components. When \( t \) is of the form \( \text{id}(x_0, \cdots, x_{n-1}) \) where \( x_i \) is a term of sort \( \mathbb{R} \), we have \( x_0, \cdots, x_{n-1} \) for the desired components.

Consider any well-formed proposition \( P \) with free variables \( \Gamma \) in the extended language.

We can translate it to be a well-formed proposition \([P]\) in \( L_0 \) with free variables in \([\Gamma]\) as follows:

- \([P] := P\) when \( P \) does not use any \( \mathbb{R}^n \) sort.
- \([t_1 = t_2] := [t_1]_0 = [t_2]_0 \land \cdots \land [t_1]_{n-1} = [t_2]_{n-1}\) when \( t_1, t_2 \) are terms of sort \( \mathbb{R}^n \).
- \([\text{proj}(x, k, y)] := (k = 0 \Rightarrow y = [x]_0) \land \cdots \land (k = n - 1 \Rightarrow y = [x]_{n-1})\) when \( x \) is of sort \( \mathbb{R}^n \).
- \([\text{update}(x, k, y, z)] := (k = 0 \Rightarrow [z]_0 = y \land [z]_1 = [x]_1 \land \cdots \land [z]_{n-1} = [x]_{n-1}) \land \cdots \land (k = d - 1 \Rightarrow [z]_0 = [x]_0 \land [z]_1 = [x]_1 \land \cdots \land [z]_{n-1} = y)\) when \( x, z \) are of sort \( \mathbb{R}^n \).
- \([P \lor Q] := [P] \lor [Q], [P \land Q] := [P] \land [Q], \) and \([P \Rightarrow Q] := [P] \Rightarrow [Q]\).
- \([\forall v. P] := \forall v. [P] \) when \( v \) is of sort \( \mathbb{K}, \mathbb{Z} \) or \( \mathbb{R} \). When \( v \) is of sort \( \mathbb{R}^n \), \([\forall v. P] := \forall v_0, \cdots, v_{n-1}. [P]\) where \( v_0, \cdots, v_{n-1} \) are variables of sort \( \mathbb{R} \) that are added in translating variable \( v \) in \([P]\).
- \([\exists v. P] := \exists v. [P] \) when \( v \) is of sort \( \mathbb{K}, \mathbb{Z} \) or \( \mathbb{R} \). When \( v \) is of sort \( \mathbb{R}^n \), \([\exists v. P] := \exists v_0, \cdots, v_{n-1}. [P]\) where \( v_0, \cdots, v_{n-1} \) are variables of sort \( \mathbb{R} \) that are added in translating variable \( v \) in \([P]\).
By structural induction on propositions, we can see that for any proposition \( P \) with free variables \( \Gamma, \sigma \in [\Gamma] \) validates \( P \) if and only if \( [\sigma] \in [\Gamma] \) validates \( [P] \).

Taking this translation implicitly, we use \( \mathbb{R}^n \) sorted variables, partial projections, and updating.

6.2. Specification Language. Since we can express fixed length arrays in the three-sorted structure, we can express (multi-)functions that appear in the primitive extensions \( F, G \).

(Recall that the domains of (multi-)functions there can be a product of \( \mathbb{R}^n \).) We define a specification language for each ERC, \( \mathcal{F}, \mathcal{G} \) by expanding the structure of ERC as follows:

**Definition 6.4.** The Structure of ERC, \( \mathcal{F}, \mathcal{G} \) is the expansion \( \mathcal{S}(\mathcal{F}, \mathcal{G}) \) of \( S_0 \) with the partial real functions \( f \in \mathcal{F} \) of various arities and with the partial integer multi-functions \( g \in \mathcal{G} \), also of various arities. More formally, the expansion is done with the graphs:

- \( \{(x_1, \cdots, x_d; y) \mid f(x_1, \cdots, x_d) = y \neq \bot\} \) for each \( f \in \mathcal{F} \) and
- \( \{(x_1, \cdots, x_d; y) \mid y \in g(x_1, \cdots, x_d) \land \bot \notin g(x_1, \cdots, x_d)\} \) for each \( g \in \mathcal{G} \).

The Logic of ERC, \( \mathcal{F}, \mathcal{G} \) is the first-order language \( \mathcal{L}(\mathcal{F}, \mathcal{G}) \) of \( \mathcal{S}(\mathcal{F}, \mathcal{G}) \) and the Theory of ERC, \( \mathcal{F}, \mathcal{G} \) is the complete first-order theory \( \mathcal{T}(\mathcal{F}, \mathcal{G}) \) of \( \mathcal{S}(\mathcal{F}, \mathcal{G}) \).

Let us write simply \( \mathcal{S}, \mathcal{T}, \) and \( \mathcal{L} \) when the underlying \( \mathcal{F} \) and \( \mathcal{G} \) are obvious or not so relevant. Though, formally, we only add “graphs” of the (multi-)functions, we may still have their function application in the term language as an appropriate abbreviation where the last argument of the predicate for the graph is quantified over.

We take the logic of ERC, \( \mathcal{F}, \mathcal{G} \) to formally specify ERC, \( \mathcal{F}, \mathcal{G} \) programs. The following lemma shows that our specification logic are ‘adequate’ for the term language of ERC.

**Definition 6.5.** Consider any data types \( \tau_i, \tau \) in ERC. A partial function \( f : [\tau_1] \times \cdots \times [\tau_d] \rightarrow [\tau] \) is definable in the logic of ERC if there is a formula \( \psi(x_1, \cdots, x_d; y) \) defining the graph (as in Definition 6.4) of \( f \) in the structure of ERC. Similarly, a partial multi-function \( g : [\tau_1] \times \cdots \times [\tau_d] \rightarrow \mathcal{P}(\tau) \) is definable in the logic of ERC if there is a formula \( \psi(x_1, \cdots, x_n; y) \) defining the graph of \( g \).

**Lemma 6.6.** For each well-typed term \( \Gamma \vdash t : \tau \) in ERC, its denotation \( [\Gamma \vdash t : \tau] : [\Gamma] \rightarrow \mathcal{P}(\tau) \) is definable in the logic of ERC.

**Proof.** Let \( X, Y_i, Z \in \{[\tau] \mid \tau \) is a data type \}. Suppose \( f_i : X \rightarrow \mathcal{P}(\langle Y_i \rangle) \) and \( g : Y_1 \times \cdots \times Y_d \rightarrow \mathcal{P}(Z) \) are definable partial multi-functions where \( \psi_i(x; y_i) \) defines \( f_i \) and \( \psi(y_1, \cdots, y_d; z) \) defines \( g \). Then, its composition \( g \circ f \) is definable by \( \psi(x; z) := \psi(y_1, \cdots, y_d; \psi(x; y_1) \land \cdots \land \psi_d(x; y_d) \land \psi(y_1, \cdots, y_d; z)) \).

Also, if a partial function \( f : X \rightarrow Z \) is definable by \( \psi(x; z) \), its embedding \( f^\dagger \) is definable by the same \( \psi \).

Hence, we only need to check the nontrivial cases, the definability of primitive functions that are absent in the structure ERC: (1) choose and (2) Kond:

(1) See that choose \( n : \mathbb{K}^n \rightarrow \mathbb{Z} \) is definable by \( \psi(x_1, \cdots, x_n; y) := (y = 0 \land x_1 = \text{true}) \lor \cdots \lor (y = n - 1 \land x_{n-1} = \text{true}) \).
(2) Suppose \( \psi_b(x; y) \) defines \( b : X \to \mathbb{P}(\mathbb{R}_+) \), \( \psi_f(x; y) \) defines \( f : X \to \mathbb{P}(\mathbb{R}_+) \), and \( \psi_g(x; y) \) defines \( g : X \to \mathbb{P}(\mathbb{R}_+) \). Then, the composition \( \text{Kond}^{1 \circ} (b, f, g) \) is definable by

\[
\psi(x; z) \equiv (\psi_b(x; \text{true}) \Rightarrow \exists y. \psi_f(x; y)) \\
\wedge (\psi_b(x; \text{false}) \Rightarrow \exists y. \psi_g(x; y)) \\
\wedge (\psi_b(x; \text{unknown}) \\
\Rightarrow (\exists y. \psi_f(x; y) \wedge \psi_g(x; y) \wedge \forall y_1, y_2. \psi_f(x; y_1) \wedge \psi_g(x; y_2) \Rightarrow y_1 = y_2)) \\
\wedge ((\psi_b(x; \text{true}) \wedge \psi_f(x; z)) \vee (\psi_b(x; \text{false}) \wedge \psi_g(x; z)) \vee (\psi_b(x; \text{unknown}) \wedge \psi_f(x; z)))
\]

\[\square\]

**Definition 6.7.** For a well-typed term \( \Gamma \vdash t : \tau \) in ERC, let us write \( \langle \Gamma \vdash t : \tau \rangle \) for the formula defining \( \lceil \Gamma \vdash t : \tau \rceil \) according to Lemma 6.6.

Let us write simply \( \langle t \rangle \) for \( \langle \Gamma \vdash t : \tau \rangle \) when the particular context \( \Gamma \) is obvious or irrelevant. Also, we let the free variables representing the input values in \( \langle t \rangle \) be synchronized with \( \Gamma \) and omit them in the notation; i.e., \( \langle t \rangle (y) := \langle x_1 : \tau_1, \ldots, x_d : \tau_d \vdash t : \tau \rangle(x_1, \ldots, x_d; y) \).

Note however that when we go beyond the term language of \( \text{ERC}_0 \), there are real functions that can be defined in the programming language but not in our logic:

**Example 6.8.** The restricted exponential function \( \exp : I = [0; 1] \to \mathbb{R} \) is uniquely characterized by the following formula:

\[
\forall x, y \in I. \quad x + y \in I \Rightarrow \exp(x + y) = \exp(x) \cdot \exp(y) \tag{6.2}
\]

\[
\forall x, y \in I. \ |\exp(x) - \exp(y)| \leq 3 \cdot |x - y|
\]

\[
\exp(1) = \lim_{n \to \infty} (1 + 1/n)^n = \sum_{n=0}^{\infty} 1/n! \tag{6.3}
\]

The first line is the well-known functional equation, and the second one captures Lipschitz-continuity.

Although \( \exp \) can be realized as in Section 5.2.2, the defining Formula (6.3) **exceeds** the logic of ERC by involving the transcendental constant \( e \), which cannot be characterized algebraically. \( \square \)

In other words, our logic is not expressive enough for the command language of ERC. More generally, propositional formulas in the Logic of ERC can only define semi-algebraic subsets of Euclidean space; and, according to Tarski-Seidenberg, real quantification does not increase the expressive power. Integer quantification can define countable unions of semi-algebraic subsets, but no more according to Theorem 6.2(b). According to Lindemann-Weierstrass, the graph of \( \exp : [0; 1] \to \mathbb{R} \) is no countable union of semi-algebraic Euclidean sets, hence impossible to define in \( \mathcal{S}_0 \).

The expansion \( \mathcal{S}(\{\exp\}, \emptyset) \) on the other hand makes \( \exp \) trivial to define—but its first-order theory may violate decidability. Such trade-offs are unavoidable according to Remark 1.1.

On the other hand, specification and formal verification may suffice with less than definability of the function under consideration: applications tend to be interested in solutions that satisfy given algebraic properties expressible in ERC—such as the exponential functional Equation 6.2—but do not necessarily make them unique, particularly transcendental or multi-valued ones.
6.3. **Hoare Logic.** Hoare logic is a well-known tool for formally proving the total correctness of a program and agreement with the problem specification. The following considerations are guided by [Rey09, §3], adapted and extended to ERC with its three-sorted structure and multi-valued semantics. Both complicate matters since, for instance, a real guard variable in a while loop may strictly decrease during each iteration yet remain bounded forever; furthermore merely evaluating the loop condition can cause lack of termination when real equality occurs; see Remark 6.12 below. Since our language is simple imperative, we adopt the following notion of “(total correctness) specification”:

**Definition 6.9.** For a well-typed command $S$ in $\text{ERC}(\mathcal{F}, \mathcal{G})$ with $\Gamma \vdash S \triangleright \Gamma'$, a (total correctness) specification is of the following form:

$$\Gamma \models \{\phi\} S \{\psi\} \triangleright \Gamma'$$

where $\phi, \psi$ are formulae in the logic of $\text{ERC}(\mathcal{F}, \mathcal{G})$. In the precondition $\phi$, only the variables in $\Gamma$ appear free and in the postcondition $\psi$, only the variables in $\Gamma'$ appear free. The notation says, for any $\sigma \in \mathcal{I}_\Gamma$ which validates $\phi$ (i) $\perp \not\in \mathcal{I}_S \sigma$ and (ii) any $\delta \in \mathcal{I}_S \sigma$ validates $\psi$.

Note that the definition of our specification does not publish any obligation to $S$ when the precondition $\phi$ is not met; see Fact 2.4.

The purpose of (Hoare) logic is to replace ‘semantic’ arguments with formal proofs: sequences of purely syntactic manipulations, starting with the axioms and following certain inference rules, that for example, a computer can verify. Classical Hoare logic contains one exception:

**Remark 6.10.** The rule of consequence for precondition-strengthening and postcondition-weakening

$$\Gamma \models \{\phi\} S \{\psi\} \triangleright \Gamma' \quad \frac{\phi \Rightarrow \phi' \text{ and } \psi' \Rightarrow \psi}{\Gamma' \models \{\phi\} S \{\psi\} \triangleright \Gamma'}$$

depends on the semantic side-conditions $\phi \Rightarrow \phi'$ and $\psi' \Rightarrow \psi$ which may or may not be feasible to verify algorithmically. Over integers, algorithmic verifiability can fail according to Gödel [Coo78, §6], but not in the logic of $\text{ERC}_0$ according to Theorem 6.2(a).

A side condition is also present in the rule for while loop termination.

**Definition 6.11.** A Hoare triple for $\text{ERC}(\mathcal{F}, \mathcal{G})$ is of the form $\Gamma \vdash \{\phi\} S \{\psi\} \triangleright \Gamma'$ where $S$ is a well-typed command in $\text{ERC}(\mathcal{F}, \mathcal{G})$ such that $\Gamma \vdash S \triangleright \Gamma'$, and $\phi, \psi$ are formulae in the logic of $\text{ERC}(\mathcal{F}, \mathcal{G})$ such that only the variables in $\Gamma$ are free in $\phi$ and only the variables in $\Gamma'$ are free in $\psi$. Hoare logic of $\text{ERC}(\mathcal{F}, \mathcal{G})$ is a formal system which consists of the inference rules and axioms for constructing Hoare triples defined in Figure 12.

**Remark 6.12.**

1. In the axiom for assignments, recall from Definition 6.7 that the precondition $\exists w. \{\emptyset\}(w)$ ensures that the denotation of $t$ is well-defined. And, $\forall w. \{\emptyset\}(w) \Rightarrow \psi[w/v]$ says that for each value in the denotation of $t$, $\psi$ holds when we replace the variable $v$ with the value.
2. In the while loop case,
   a. the formula $I$ is the loop invariant and the term $V$ is the loop variant. The term $L$ is some invariant quantity that bounds by how much $V$ decreases in each iteration.
   b. $\xi, \xi'$ are ghost variables that do not appear in $\Gamma$. They can be understood as meta-level universally quantified variables.
Figure 12: The Hoare logic of $\mathcal{ERC}(\mathcal{F}, \mathcal{G})$

\[
\begin{align*}
\Gamma \vdash \{\phi\} S \{\psi\} \triangleright \Gamma' & \quad \phi \Rightarrow \phi' \text{ and } \psi \Rightarrow \psi' \quad \Gamma \vdash \{\psi\} \text{ skip } \{\psi\} \triangleright \\
\Gamma \vdash \{\exists w. ([t](w) \land \forall w. ([t](w) \Rightarrow \psi[w/v]) \} v := t \{\psi\} \triangleright \\
\Gamma \vdash \{\text{ArrPre}(\psi, v, z, t)\} v[z] := t \{\psi\} \triangleright \\
\Gamma \vdash \{\phi\} S_1 \{\theta\} \triangleright \Gamma_1 \quad \Gamma_1 \vdash \{\theta\} S_2 \{\psi\} \triangleright \Gamma_2 & \quad \Gamma \vdash \{\phi \land ([b](\text{true}))\} S_1 \{\psi\} \triangleright \Gamma_1 \quad \Gamma \vdash \{\phi \land ([b](\text{false}))\} \quad S_2 \{\psi\} \triangleright \Gamma_2 \\
\Gamma \vdash \{\phi \land ([b](\text{true}) \lor ([b](\text{false})) \land \neg([b](\text{unknown}))\} \quad \text{if } b \text{ then } S_1 \text{ else } S_2 \text{ end } \{\psi\} \triangleright \\
\Gamma' \vdash \{([b](\text{true}) \land I \land V = \xi \land L = \xi')\} \quad S \{I \land V \leq \xi - \xi' \land L = \xi'\} \triangleright \Gamma'' & \quad \Gamma \vdash \{I\} \text{ while } b \text{ do } S \text{ end } \{I \land ([b](\text{false}))\} \triangleright \\
\end{align*}
\]

In the rule for loops, $\Gamma' = \Gamma, \xi, \xi' : R$. And, the rule has the side-conditions:

\[
I \land [b](\text{true}) \Rightarrow L > 0 ,
\]

\[
I \Rightarrow ([b](\text{true}) \lor ([b](\text{false})) \land \neg([b](\text{unknown})) ,
\]

\[
I \land V \leq 0 \Rightarrow \forall k. ([b](k) \Rightarrow k = \text{false} , \text{ and}
\]

\[
\xi, \xi' \text{ do not appear free in } I, V, L.
\]

In the case of array assignment $\text{ArrPre}(P, v, z, t)$ is defined as follows:

\[
\text{ArrPre}(P, v, z, t) := (\exists w, m. ([b](w) \land [z](m)) \\
\land \forall m. ([z](m) \Rightarrow 0 \leq m < d) \\
\land \forall w, m. ([b](w) \land [z](m) \land \forall v'. \text{ update}(v, m, w, v') \Rightarrow P[v'/v])
\]

assuming $\Gamma \vdash v[z] := t \triangleright \Gamma$ and $\Gamma \vdash v : R[d]$.

(c) The side condition says (i) each loop decreases $V$ by some positive invariant quantity $L$; (ii) as long as $I$ holds, the evaluation of $b$ is either $\text{true}$ or $\text{false}$ (but not $\text{unknown}$); and (iii) when $V$ is negative, it is guaranteed that the evaluation of $b$ is $\text{false}$.

(3) In the rule of if conditionals, the precondition $([b](\text{true}) \lor ([b](\text{false})) \land \neg([b](\text{unknown}))$ says that the evaluation of $b$ is either $\text{true}$ or $\text{false}$ (but not $\text{unknown}$).

\begin{theorem}
The Hoare logic of $\mathcal{ERC}(\mathcal{F}, \mathcal{G})$ is sound; i.e., for any Hoare triple $\Gamma \vdash \{\phi\} S \{\psi\} \triangleright \Gamma'$, it holds that $\Gamma \vdash \{\phi\} S \{\psi\} \triangleright \Gamma'$.
\end{theorem}

\begin{proof}
See Appendix A.
\end{proof}
7. Example Formal Verification in Exact Real Computation

The present section picks up from Section 5.2.6 to illustrate formal verification in ERC. To emphasize, our purpose here is not to actually establish correctness of the long-known Trisection method, but to demonstrate the extended Hoare logic from Section 6.3 using a toy example. Since Trisection relies on the Intermediate Value Theorem, any correctness proof must make full use of real (as opposed to, say, floating point, rational, or algebraic) numbers.

Let us define some abbreviations such that the algorithm in Figure 11, which is a program in ERC(\{f\}, \emptyset) for some continuous f becomes of the form input a : R, b : R S return a as p \to -\infty.

\[
\begin{align*}
\tilde{t}_1 & := 2^p > b - a, \quad \tilde{t}_2 := b - a > 2^{p-1} \\
t_1 & := 0 > f(b/3 + 2 \times a/3) \times f(b), \quad t_2 := 0 > f(a) \times f(2 \times b/3 + a/3) \\
b_1 & := \text{choose}(\tilde{t}_1, \tilde{t}_2) = 1, \quad b_2 := \text{choose}(t_1, t_2) = 1 \\
S & := \text{while } b_1 \text{ do } S_1 \text{ end} \\
S_1 & := \text{if } b_2 \text{ then } S_2 \text{ else } S_3 \text{ end} \\
S_2 & := b := 2 \times b/3 + a/3 \\
S_3 & := a := b/3 + 2 \times a/3
\end{align*}
\]

We work with the logic of \(T(\{f\}, \emptyset)\) which contains sentences saying that \(f\) is a continuous function. We want to verify that the program’s denotation at its inputs \(a, b\) which isolate a root of \(f\) uniquely with a sign change:

\[\text{uniq}(f, a, b) := \text{cont}(f, a, b) \land f(a) \land f(b) < 0 \land \exists x. a < x < b \land f(x) = 0.\]

Considering the limit taken at the end of every real programs, we need to prove the specification:

\[\Gamma \vdash \left\{ p = p' \land \text{uniq}(f, a, b) \right\} S \left\{ \exists z. f(z) = 0 \land a < z < b \land |a - z| \leq 2^p \right\} \Gamma'\]

where \(\Gamma = p, p' : Z, a, b : R\) and \(\Gamma' = p, p' : Z, a, b : R\). The ghost variable \(p'\) captures the initial value that the variable \(p\) stores, considering that the value \(p\) stores may throughout the computation (though it does not in this specific example.)

To simplify our presentation, let us write \(t_1, t_2, \tilde{t}_1, \tilde{t}_2\) as valid formulae in our specification language where their occurrences of \(<\) are implicitly replaced with \(\leq\). See that \(\{b_1\}(\text{true}) \leftrightarrow t_2, \{b_1\}(\text{false}) \leftrightarrow \tilde{t}_1, \sim \{b_1\}(\text{unknown})\), \(\{b_2\}(\text{true}) \leftrightarrow t_2, \{b_2\}(\text{false}) \leftrightarrow \tilde{t}_1, \sim \{b_2\}(\text{unknown})\) hold. Let us define \(I := p = p' \land \text{uniq}(f, a, b)\) as a candidate for the loop invariant, \(V := b - a - 2^{p-1}\) as a candidate for the loop variant, \(L := 2^{p-2}\) as a candidate for a lower bound decrements, \(\tilde{P} := \tilde{t}_2 \land I \land V = \xi \land L = \xi'\), and \(\tilde{Q} := I \land V \leq \xi - \xi' \land L = \xi'\) in our specification language with variables \(\xi, \xi'\) of the sort \(R\). Let \(\Delta := p, p' : Z, a, b, \xi, \xi' : R\).

From the axiom for assignments, we have the triples:

\[
\begin{align*}
\Delta \vdash \{ \exists \omega. \{a/3 + 2 \times b/3\}(\omega) \land \forall \omega. \{a/3 + 2 \times b/3\}(\omega) \Rightarrow \tilde{Q}[\omega/b]\} & S_2 \{ \tilde{Q} \} \triangleright \Delta \\
\Delta \vdash \{ \exists \omega. \{2 \times a/3 + b/3\}(\omega) \land \forall \omega. \{2 \times a/3 + b/3\}(\omega) \Rightarrow \tilde{Q}[\omega/a]\} & S_3 \{ \tilde{Q} \} \triangleright \Delta
\end{align*}
\]

See that we can apply the rule of precondition weakening to get the following triples derived:

\[
\Delta \vdash \{ \tilde{Q}[(a/3 + 2 \times b/3)/b] \} C_2 \{ \tilde{Q} \} \triangleright \Delta, \quad \Delta \vdash \{ \tilde{Q}[(2 \times a/3 + b/3)/a] \} C_3 \{ \tilde{Q} \} \triangleright \Delta.
\]

When we unwrap abbreviations, we can verify the following equivalences:

\[
\tilde{Q}[(2 \times a/3 + b/3)/a] \iff p = p' \land \text{uniq}(f, (2 \times a/3 + b/3), b) \\
\land b - (2 \times a/3 + b/3) - 2^{p-1} \leq \xi - \xi'
\]
\[2^{p-2} = \xi'
\]
\[\bar{P} \land t_1 \iff p = p' \land \text{uniq}(f, a, b) \land f(b/3 + 2 \times a/3) \times f(b) < 0
\]
\[\land 2^{p-1} < b - a \land b - a - 2^{p-1} = \xi
\]
\[\land 2^{p-2} = \xi'
\]

Due to intermediate value theorem, if an interval \((a; b)\) contains a root of \(f\) uniquely, and if \(f(x) \times f(y) < 0\) for \(a \leq x < y \leq b\), then \((x; y)\) also contains the root of \(f\) uniquely. Hence, \(\bar{P} \land t_1 \Rightarrow \bar{Q}[(2 \times a/3 + b/3)/a]\) holds. And, similarly, \(\bar{P} \land t_2 \Rightarrow \bar{Q}[(a/3 + 2 \times b/3)/b]\) holds.

Therefore, by the rule of precondition strengthening on the triples of \(S_2, S_3\), and the rule for conditionals, we get the triple:
\[\Delta \vdash \{\bar{t}_2 \land I \land (V = \xi) \land L = \xi'\} \ S_1 \ \{I \land (V \leq \xi \land L = \xi')\} \triangleright \Delta
\]
The side-conditions of the rule for while loops are quite trivial. Hence, assuming that they are proven, we apply the rule of while loops, and we get the following triple:
\[\Gamma \vdash \{I\} \ S \ \{I \land \bar{t}_2\} \triangleright \Gamma'
\]
Using the rule of pre/postcondition strengthening/weakening, we can get the originally desired specification.

8. Conclusion and Future work

We have formalized exact real number computation as an imperative programming language. Its domain-theoretic denotational semantics, based on the Plotkin powerdomain modelling multi-valued computations, is carefully designed to be computable and complete, matching the intuition of operating on continuous data exactly without rounding errors and in agreement with (proofs in) calculus. This enables a natural approach to formal program verification by adding real number axioms to Hoare logic.

The following considerations naturally suggest future further investigations:

- **Computational Cost:** After the design of an algorithm comes its analysis, in terms of computational cost as quantitative indicator of its practical performance. For realistic predictions, real complexity theory [Ko91, Wei03] employs bit-cost, as opposed to unit cost common in algebraic complexity theory [BCS97]. In [BH98, Definition 2.4] it is suggested that a logarithmic cost measure where each operation is supposed to take time according to the binary length of the integer (part of the real) to be processed. More accurate predictions take into account the precision parameter \(p\) for real programs and for real number comparisons “\(x \gg y\)” the logarithm of the difference \(|x - y|\).

- **Full Mixed Data Types:** ERC as introduced here formalizes computing with data types \(\mathbb{Z}, \mathbb{R}, \text{and } K\) as counterparts to mathematical \(\mathbb{Z}, \mathbb{R}, \mathbb{K}\) where \(K\) is permitted only for expressions and local variables. A future extension will include also (multi-)functions with \(K\) type arguments and return values as well as arrays and a dedicated limit operator.

- **Multi-valued Real Functions:** The present version of ERC formalizes computing mappings from reals to integers in the multi-valued sense because any single-valued, and necessarily continuous [Wei00, Theorems 4.3.1+3.2.11], function with connected domain and discrete range must be constant. On the other hand, computing real values is deliberately restricting to the single-valued case. Defining approximate computation of real multi-valued functions is delicate and still under exploration [FB18, Kon18].
• **Functionals and Operators:** The algorithm in Figure 11 mimics a functional that receives a continuous function $f$ as argument, accessible by point-wise blackbox evaluation. Extending ERC with function arguments enriched with quantitative continuity information, such as a modulus of continuity [KC12], is necessary to extend Theorem 5.3 (Turing-completeness of ERC) from functions to functionals.

• **Automated Formal Verification:** The decidability of the theory of the base language ERC$_0$ according to Theorem 6.2 includes its complete (and actually elegant) axiomatization: Guaranteed to yield convenient automated formal verification. This includes formalizing ERC itself in a formal system for example in the Coq proof assistant [KST20a, STT21] and further development of annotated ERC and verification condition extraction mechanisms.

• **Continuous Data Types beyond the Reals:** Real computability theory has been extended to topological $T_0$ spaces, real complexity theory to co-Polish spaces [KSZ16, Sch04]. Current and future works similarly extend ERC to continuous data types beyond real numbers/functions, such as the Grassmannian, tensors [SAL+18] and groups [SZ18]. As it is hinted in Remark 5.4, this may require extending our language with countable nondeterminism for the extended language to be complete as well.

**Acknowledgement**

This work was supported by the National Research Foundation of Korea (grants NRF-2017R1E1A1A03071032 and NRF-2017R1D1A1B05031658), by the International Research & Development Program of the Korean Ministry of Science and ICT (grant NRF-2016K1A3A7A03950702), by the European Union’s Horizon 2020 research and innovation program under the Marie Skłodowska-Curie grant agreement No 731143, and by the German Research Foundation (DFG) Grant MU 1801/5-1.

We thank Andrej Bauer, Cyril Cohen, Jeehoon Kang, Johannes Kanig, Alex Simpson, and Hongseok Yang for seminal discussions. Anonymous referees have provided valuable feedback to an earlier version.

**References**

[AJ95] Samson Abramsky and Achim Jung. Domain theory. In *Handbook of Logic in Computer Science*, volume III, pages 1–168. Oxford University Press, Oxford, United Kingdom, 1995.

[AO19] Krzysztof R Apt and Ernst-Rüdiger Olderog. Fifty years of hoare’s logic. *Formal Aspects of Computing*, 31(6):751–807, 2019.

[AY07] Jeremy Avigad and Yimu Yin. Quantifier elimination for the reals with a predicate for the powers of two. *Theor. Comput. Sci.*, 370(1-3):48–59, 2007. doi:10.1016/j.tcs.2006.10.005.

[BC88] Hans-Juergen Karl Hermann Boehm and Robert Cartwright. *Exact real arithmetic: Formulating real numbers as functions*. Rice University, Department of Computer Science, 1988.

[BCS97] Peter Bürgisser, Michael Clausen, and Mohammad Amin Shokrollahi. *Algebraic complexity theory*, volume 315 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag Berlin Heidelberg, Berlin, Heidelberg, 1997.

[Bés02] Alexis Bès. A survey of arithmetical definability. *Soc. Math. Belgique*, A tribute to Maurice Boffa:1–54, 2002.

[BFM09] Sylvie Boldo, Jean-Christophe Filliâtre, and Guillaume Melquiond. Combining coq and gappa for certifying floating-point programs. In Jacques Carette, Lucas Dixon, Claudio Sacerdoti Coen, and Stephen M. Watt, editors, *Intelligent Computer Mathematics*, pages 59–74, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[BH98] Vasco Brattka and Peter Hertling. Feasible real random access machines. *Journal of Complexity*, 14(4):490–526, 1998.
[Bla] Jens Blanck. cdar – Reals based on Centred Dyadic Approximations. https://github.com/jensblanck/cdar.

[BM11] 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.

[Bra95] Vasco Brattka. Computable selection in analysis. In Proc. of the Workshop on Computability and Complexity in Analysis, Informatik Berichte, FernUniversitat Hagen, volume 190, pages 125–138, 1995.

[Bra03] Vasco Brattka. The emperor’s new recursiveness: The epigraph of the exponential function in two models of computability. In Masami Ito and Teruo Imaoka, editors, Words, Languages & Combinatorics III, pages 63–72, Singapore, 2003. World Scientific Publishing. ICWLC 2000, Kyoto,Japan, March 14–18, 2000.

[BVS93] Stephen Brookes and Kathryn Van Stone. Monads and comonads in intensional semantics. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1993.

[CGC++20] Pieter Collins, Luca Geretti, Alberto Casagrande, Ivan Zapreev, and Sanja Zivanovic. Ariadne. http://www.ariadne-cps.org/, 2005-20.

[Coo78] Stephen A Cook. Soundness and completeness of an axiom system for program verification. SIAM Journal on Computing, 7(1):70–90, 1978.

[DGE13] Pietro Di Gianantonio and Abbas Edalat. A language for differentiable functions. In Frank Pfenning, editor, Foundations of Software Science and Computation Structures, pages 337–352, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Dri86] Lou van den Dries. The field of reals with a predicate for the powers of two. Manuscripta Mathematica, 54:187–196, 1986. URL: http://eudml.org/doc/155108.

[EE00] Abbas Edalat and Martin Hötzel Escardó. Integration in real pcf. Information and Computation, 160(1-2):128–166, 2000.

[ES14] Martin Hötzel Escardó and Alex Simpson. Abstract datatypes for real numbers in type theory. In Gilles Dowek, editor, Rewriting and Typed Lambda Calculi, pages 208–223, Cham, 2014. Springer International Publishing.

[Esc96] Martin Hötzel Escardó. PCF extended with real numbers. Theoretical Computer Science, 162:79–115, 1996.

[FB18] Robert Rettinger Franz Brausse, Norbert Müller. Intensionality and multi-valued limits. In Proc. 15th Internat. Conf. on Computability and Complexity in Analysis (CCA), page 11, 2018.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, Programming Languages and Systems, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[Her96] Peter Hertling. Topological complexity with continuous operations. Journal of Complexity, 12:315–338, 1996. doi:10.1006/jcom.1996.0021.

[Her99] Peter Hertling. A real number structure that is effectively categorical. Mathematical Logic Quarterly, 45(2):147–182, 1999.

[Ho69] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM, 12(10):576–580, oct 1969. URL: https://doi.org/10.1145/363235.363259, doi:10.1145/363235.363259.

[KC96] Bruce M. Kapron and Steven A. Cook. A new characterization of type-2 feasibility. SIAM Journal on Computing, 25(1):117–132, 1996.

[KC12] Akitoishi Kawamura and Stephen Cook. Complexity theory for operators in analysis. ACM Trans. Comput. Theory, 4(2):5:1–5:24, May 2012. doi:10.1145/2189778.2189780.

[KMP++08] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee-Keng Yap. Classroom examples of robustness problems in geometric computations. Comput. Geom., 40(1):61–78, 2008. doi:10.1016/j.comgeo.2007.06.003.

[Ko91] Ker-I Ko. Complexity Theory of Real Functions. Progress in Theoretical Computer Science. Birkhäuser, Boston, 1991.

[Kon18] Michal Konecný. Verified exact real limit computation. In Proc. 15th Internat. Conf. on Computability and Complexity in Analysis (CCA), pages 9–10, 2018.

[KST20a] Michal Konecný, Florian Steinberg, and Holger Thies. Computable analysis for verified exact real computation. In Nitin Saxena and Sunil Simon, editors, 40th IARCS Annual Conference on
Michal Konečný, Florian Steinberg, and Holger Thies. 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.

Akitoshi Kawamura, Florian Steinberg, and Martin Ziegler. Complexity theory of (functions on) compact metric spaces. In Proc. 31st Ann. Symposium on Logic in Computer Science, pages 837–846, New York, NY, United States, 2016. ACM/IEEE, Association for Computing Machinery.

M. Konecny, W. Taha, J. Duracz, A. Duracz, and A. Ames. Enclosing the behavior of a hybrid system up to and beyond a zeno point. In Proc. 1st IEEE Int. Conf. on Cyber-Physical Systems, Networks, and Applications, pages 120–125, New York, NY, United States, 2013. Association for Computing Machinery.

Branimir Lambov. Reallib: an efficient implementation of exact real arithmetic. In CCA 2005 - Second International Conference on Computability and Complexity in Analysis, August 25-29, 2005, Kyoto, Japan, pages 169–175, 2005.

Horst Luckhardt. A fundamental effect in computations on real numbers. Theoretical Computer Science, 5(3):321–324, 1977.

David Marker. Model Theory of Fields. Lecture Notes in Logic. Cambridge University Press, Cambridge, United Kingdom, 2017. doi:10.1017/9781316716991.

Guillaume Melquiond. Proving bounds on real-valued functions with computations. In Alessandro Armando, Peter Baumgartner, and Gilles Dowek, editors, Automated Reasoning, pages 2–17, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

Dragoslav S. Mitrinović. Analytic Inequalities, volume 165 of Die Grundlehren der mathematischen Wissenschaften. Springer, 1970.

J Raymundo Marcial-Romero and Martin H Escardo. Semantics of a sequential language for exact real-number computation. Theoretical Computer Science, 379(1-2):120–141, 2007.

Norbert Th. Müller. The iRRAM: Exact arithmetic in C++. In Jens Blanck, Vasco Brattka, and Peter Hertling, editors, Computability and Complexity in Analysis, volume 2064 of Lecture Notes in Computer Science, pages 222–252, Berlin, 2001. Springer. 4th International Workshop, CCA 2000, Swansea, UK, September 2000.

Aleksandar Nanevski, Greg Morrisett, and Lars Birkedal. Polymorphism and separation in hoare type theory. In Proceedings of the eleventh ACM SIGPLAN international conference on Functional programming, pages 62–73, 2006.

Marian B. Pour-El and J. Ian Richards. Computability in Analysis and Physics. Perspectives in Mathematical Logic. Springer, Berlin, 1989.

Gordon D. Plotkin. A powerdomain construction. SIAM J. Comput., 5(3):452–487, 1976.

Gordon D. Plotkin. Lcf considered as a programming language. Theoretical computer science, 5(3):223–255, 1977.

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. Numerical recipes: The art of scientific computing. Cambridge university press, Cambridge, United Kingdom, 3 edition, 2007.

John C Reynolds. Separation logic: A logic for shared mutable data structures. In Proceedings 17th Annual IEEE Symposium on Logic in Computer Science, pages 55–74. IEEE, 2002.

John C Reynolds. Theories of programming languages. Cambridge University Press, Cambridge, United Kingdom, 2009.

Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. Journal of Symbolic Logic, 33(4):514–520, 1968.

Julia Robinson. The undecidability of algebraic rings and fields. Proceedings of the American Mathematical Society, 10:950–957, 1959.

Joonhyung Shin, Namjo Ahn, Seungwoo Lee, Chansu Park, and Martin Ziegler. Towards tensor calculus in exact real computation. 21st Japan-Korea Joint Workshop on Algorithms and Computation, August 2018.
Appendix A. Proof of the Soundness of the Hoare Logic of ERC

We start the proof with the lemma:

Lemma A.1. For a well-typed command $\Gamma \vdash \textbf{while } b \textbf{ do } S \textbf{ end } \triangleright \Gamma$, define the sequences of set-valued functions on $[\Gamma]$:

- $B^0_{b,S}(\sigma) \equiv \{ \sigma \}$
- $C^0_{b,S}(\sigma) \equiv \emptyset$
- $B^{n+1}_{b,S}(\sigma) \equiv \bigcup_{\delta \in B^n_{b,S}(\sigma)} \bigcup_{i \in [0]} \bigcup_{\delta' \in \Gamma^i \circ S} \begin{cases} \{ \delta' \} & \text{if } l = \text{true} \land \delta' \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases}$
- $C^{n+1}_{b,S}(\sigma) \equiv C^n_{b,S}(\sigma) \cup \bigcup_{\delta \in B^n_{b,S}(\sigma)} \bigcup_{i \in [0]} \bigcup_{\delta' \in \Gamma^i \circ S} \begin{cases} \{ \delta \} & \text{if } l = \text{false}, \\ \emptyset & \text{if } l = \text{true} \land \delta' \neq \bot, \\ \{ \bot \} & \text{otherwise.} \end{cases}$

Then, for all $n \in \mathbb{N}$, it holds that $W(n)(\sigma) = C^n_{b,S}(\sigma) \cup \{ \bot \ | \ \exists x. \ x \in B^n_{b,S}(\sigma) \}$.

Intuitively, $B^n_{b,S}(\sigma)$ is the set of states that requires further execution after running the while loop on $\sigma$ for $n$ times, and $C^n_{b,S}(\sigma)$ is the set of states that have escaped from the loop (either because false has been evaluated or $\bot$ has occurred) during running the loop for $n$ times.

Proof. Let us drop the subscripts $b, S$ for the convenience in the presentation and write $S$ for $[\Gamma] \circ [S]$. 
We first prove the following alternative characterization of the sequence of sets:

\[ B^{n+1}(\sigma) = \bigcup_{\ell \in [b]_\sigma} \bigcup_{\delta \in S_\sigma} \begin{cases} B^n(\delta) & \text{if } \ell = true \land \delta \neq \bot, \\ \emptyset & \text{otherwise}. \end{cases} \]

It is trivial when \( n = 0 \). Now, suppose the equation holds for all \( \sigma \) and for all \( n \) up to \( m \). Then the following derivation shows that the characterization is valid for \( n = m + 1 \) as well.

\[ B^{m+2}(\sigma) = \bigcup_{\gamma \in B^{m+1}(\sigma)} \bigcup_{\ell' \in [b]_\gamma} \bigcup_{\delta' \in S_\gamma} \begin{cases} \{\delta\} & \text{if } \ell = true \land \delta \neq \bot, \\ \emptyset & \text{otherwise}. \end{cases} \]

We now show the following characterization:

\[ C^{m+1}(\sigma) = \bigcup_{\ell \in [b]_\sigma} \bigcup_{\delta \in S_\sigma} \begin{cases} C^m(\delta) & \text{if } \ell = true \land \delta \neq \bot, \\ \{\sigma\} & \text{if } \ell = false, \\ \{\bot\} & \text{otherwise}. \end{cases} \]

It is easy to show that the equation holds for \( n = 0 \). Now, assume the equation holds for all \( n \) up to \( m \). Then,

\[ C^{m+2}(\sigma) = C^{m+1}(\sigma) \cup \bigcup_{\delta \in B^{m+1}(\sigma)} \bigcup_{\ell' \in [b]_\sigma} \bigcup_{\delta' \in S_\delta} \begin{cases} \{\delta\} & \text{if } \ell' = false, \\ \emptyset & \text{if } \ell' = true \land \delta' \neq \bot, \\ \{\bot\} & \text{otherwise}. \end{cases} \]

\[ = C^{m+1}(\sigma) \cup \bigcup_{\delta \in B^{m+1}(\sigma)} \bigcup_{\ell' \in [b]_\delta} \bigcup_{\delta' \in S_\delta} \begin{cases} B^m(\gamma) & \text{if } \ell = true \land \gamma \neq \bot, \\ \emptyset & \text{if } \ell' = true \land \delta' \neq \bot, \\ \{\bot\} & \text{otherwise}. \end{cases} \]

\[ = C^{m+1}(\sigma) \cup \bigcup_{\ell \in [b]_\sigma} \bigcup_{\gamma \in S_\sigma} \begin{cases} \{\delta\} & \text{if } \ell' = false, \\ \emptyset & \text{if } \ell' = true \land \gamma \neq \bot, \\ \{\bot\} & \text{otherwise}. \end{cases} \]
Then, (Conditional):

We can prove the statement by checking the soundness of each rule. Proof. W

Now, we proceed to the proof of Theorem 6.13: 

Now, using the suggested characterization, we prove $W^{(n)}(\sigma) = C_{b,S}^{m}(\sigma) \cup \{\bot \mid \exists x. x \in B_{b,S}(\sigma)\}$ for all $n \in \mathbb{N}$. When $n = 0$, both are $\{\bot\}$. Suppose the equation holds for $n = m$. Then,

$W^{(n+1)}(\sigma) = \bigcup_{\ell \in [b]_\sigma} \left\{ \begin{array}{ll} W^{(n)}(\delta) & \text{if } \ell = true \land \delta \neq \bot, \\
\{\sigma\} & \text{if } \ell = false, \\
\{\bot\} & \text{otherwise.} \end{array} \right.$

$= \bigcup_{\ell \in [b]_\sigma} \left\{ \begin{array}{ll} C^m(\delta) & \text{if } \ell = true \land \delta \neq \bot, \\
\{\sigma\} & \text{if } \ell = false, \\
\{\bot\} & \text{otherwise.} \end{array} \right. \bigcup \bigcup_{\ell \in [b]_\sigma} \left\{ \begin{array}{ll} \{\bot|\exists \gamma. \gamma \in B^m(\delta)\} & \text{if } \ell = true \land \delta \neq \bot, \\
\emptyset & \text{if } \ell = false, \\
\emptyset & \text{otherwise.} \end{array} \right.$

$= C^{m+1}(\sigma) \cup \left\{ \begin{array}{ll} \emptyset & \text{if } \ell = true \land \delta \neq \bot, \\
\{\bot|\exists \gamma. \gamma \in B^{m+1}(\sigma)\} & \text{otherwise.} \end{array} \right.$

We now proceed to the proof of Theorem 6.13:

Proof. We can prove the statement by checking the soundness of each rule.

(1) (Assignment):

Consider any state $\sigma$ which validates $\exists w. (L)(w) \land \forall w. (L)(w) \Rightarrow \psi[w/v]$. Then, $\bot \not\in [t]_\sigma$ and for any $w \in [t]_\sigma$, $\psi[w/v]$ holds.

Now, see that $[w := t]_\sigma = \bigcup_{w \in [t]_\sigma} \{\sigma[v \mapsto w]\}$ since $\bot \not\in [t]_\sigma$ and for all $w \in [t]_\sigma$, $\sigma[v \mapsto w]$ validates $\psi$.

(2) The rule variable declarations and the rule of array assignments can be verified in very similar manner as above and the rules of pre/postcondition strengthening/weakening, skip, and sequential compositions can be verified quite trivially.

(3) (Conditional):

Consider any state $\sigma$ which validates $\phi \land (\langle b \rangle(true) \lor \langle b \rangle(false)) \land \neg \langle b \rangle(unknown)$. Then, $[b]_\sigma = \{true, false\}, \{true\}, \text{ or } \{false\}$. Let us check the three cases:

(a) when $[b]_\sigma = \{true, false\}$:
Then, \( \sigma \) validates \( \phi \land \langle b \rangle (\text{true}) \) and \( \phi \land \langle b \rangle (\text{false}) \). Therefore, (i) \( \perp \not\in [S_1] \sigma \), (ii) for all \( \delta \in [S_1] \sigma \) it holds that \( \delta \Vdash \psi \), (iii) \( \perp \not\in [S_2] \sigma \), and (iv) for all \( \delta \in [S_2] \sigma \) it holds that \( \delta \Vdash \psi \).

Since \( \perp \not\in [b] \sigma \) and \( \text{unknown} \not\in [b] \sigma \), the denotation becomes \( [\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ end}] \sigma = \varepsilon^{†} \big( [S_1] \sigma \big) \cup \varepsilon^{†} \big( [S_2] \sigma \big) \). Since the restriction operator does not create \( \perp \), the denotation does not contain \( \perp \). Also, since \( \psi \) only consists of free variables that are in \( \Gamma \), each resulting state after the restriction still validates \( \psi \).

(b) when \( [b] \sigma = \{ \text{true} \} \):

Then, \( \sigma \) validates \( \phi \land \langle b \rangle (\text{true}) \). Hence, (i) \( \perp \not\in [S_1] \sigma \), (ii) each \( \delta \in [S_1] \sigma \) validates \( \psi \). Since \( [b] \sigma = \{ \text{true} \} \), the denotation becomes \( [\text{if } b \text{ then } S_1 \text{ else } S_2 \text{ end}] \sigma = \varepsilon^{†} \big( [S_1] \sigma \big) \). Again, since the restriction does not create \( \perp \) and \( \perp \not\in [S_1] \sigma \), \( \perp \) is not in the denotation. And, since \( \psi \) only consists of free variables that are in \( \Gamma \), each resulting state after the restriction still validates \( \psi \).

(c) when \( [b] \sigma = \{ \text{false} \} \), it can be done very similarly to the above item.

(4) (Loop):

Consider any state \( \sigma \) that validates \( I \). Then, by the side-conditions, it also validates \( (\langle b \rangle (\text{true}) \lor \langle b \rangle (\text{false})) \land \langle \text{unknown} \rangle \). Hence, \( [b] \sigma = \{ \text{true}, \text{false} \}, \{ \text{true} \}, \) or \( \{ \text{false} \} \) for any state \( \sigma \) that validates \( I \). Now, we fix a state \( \sigma \) which validates \( I \) hence satisfies the precondition.

The core part of the proof is the statement: for any natural number \( n \), it holds that (i) \( \perp \not\in B_{b,S}^{n} (\sigma) \), (ii) \( \perp \not\in C_{b,S}^{n} (\sigma) \), (iii) all \( \delta \in B_{b,S}^{n} (\sigma) \) or \( C_{b,S}^{n} (\sigma) \) validates \( I \), and (iv) all \( \delta \in C_{b,S}^{n} (\sigma) \) validates \( \langle b \rangle (\text{false}) \).

At the moment, suppose that the above statement is true. Then, all we have to show is that \( B_{b,S}^{n} (\sigma) \) becomes empty as \( m \in \mathbb{N} \) increases. Let us define \( \ell_n := \max \{ V(\delta) \mid \delta \in B_{b,S}^{n} (\sigma) \} \) and show that \( \ell_n \) is strictly decreasing by some quantity that is bounded below, as \( n \) increases. See that if it holds, there will be some \( m \) that for all \( \delta \in B_{b,S}^{m} (\sigma) \), \( \varepsilon = \{ \text{false} \} \) and hence \( B_{b,S}^{m+1} (\sigma) = \emptyset \).

In order to prove it, we take the two steps:

(a) If \( B_{b,S}^{1} (\sigma) \neq \emptyset \), then for all \( n \in \mathbb{N} \) and for all \( \delta \in B_{b,S}^{n} (\sigma) \), it holds that \( L(\delta) = L(\sigma) > 0 \). In this case, let us write \( \ell_0 = L(\sigma) \).

(b) If \( B_{b,S}^{m+1} (\sigma) \neq \emptyset \), it holds that \( \ell_{m+1} \leq \ell_m - \ell_0 \).

Now, we prove each statement:

(a) \( B_{b,S}^{1} (\sigma) \neq \emptyset \) only if \( \text{true} \in [b] \sigma \) and there is some non-bottom \( \delta \in S \). Therefore, by the side-condition, \( L(\sigma) > 0 \).

Suppose any \( \delta \in B_{b,S}^{m+1} (\sigma) \) for any \( m \in \mathbb{N} \). See that it happens only if there is \( \delta' \in B_{b,S}^{m} (\sigma) \) such that \( \text{true} \in [b] \delta' \) and \( \delta \in S \delta' \). Together with Item (iii), \( \delta' \) validates \( I \) and \( \langle b \rangle (\text{true}) \). Let us define \( \delta' \equiv \delta' \cup (\xi \mapsto V(\delta') \cup \xi' \mapsto L(\delta')) \). Since \( \delta' \) validates the precondition in the premise, we have that for any \( \delta \in S \delta' \), \( \delta \) validates \( I \) and \( V \leq \xi - \xi' \) and \( L = \xi' \). Hence, \( L(\delta) = L(\delta') \). Since \( \xi', \xi \) are ghost variables, \( L(\delta) = L(\delta') \). In conclusion, for any \( \delta \in B_{b,S}^{m+1} (\sigma) \), the quantity \( L(\delta) \) is identical to the quantity \( L(\delta') \) for some \( \delta' \in B_{b,S}^{m} (\sigma) \). Since, \( B_{b,S}^{0} (\sigma) = \{ \sigma \} \), we conclude that they are all identical to \( L(\sigma) \).

(b) Suppose any \( \delta \in B_{b,S}^{m+1} (\sigma) \) for any \( m \in \mathbb{N} \). See that it happens only if there is \( \delta' \in B_{b,S}^{m} (\sigma) \) such that \( \text{true} \in [b] \delta' \) and \( \delta \in S \delta' \). Together with Item (iii), \( \delta' \)
validates $I$ and $\llbracket b \rrbracket(\text{true})$. Consider $\hat{\delta}' := \delta' \cup (\xi \mapsto V(\delta') \cup \xi' \mapsto L(\delta'))$ which validates the precondition of the premise. Hence, $\delta \cup (\xi \mapsto V(\delta') \cup \xi' \mapsto L(\delta'))$ validates the postcondition. Hence, $V(\delta) \leq V(\delta') - L(\delta) = \ell_0$. Hence, $\ell_{m+1} \leq \ell_m - \ell_0$.

Now, we need to prove the aforementioned statement on $B_{b,S}^n$ and $C_{b,S}^n$.

(a) (Base case): Recall that $B_{b,S}^0(\sigma) = \{\sigma\} \neq \{\bot\}$ and $C_{b,S}^0(\sigma) = \emptyset$. Hence, the four conditions are all satisfied.

(b) (Induction step): Recall $B_{b,S}^{n+1}(\sigma) \equiv \bigcup_{\delta \in B_{b,S}^n(\sigma)} \bigcup_{l \in \llbracket b \rrbracket \delta} \{\delta'\} \text{ if } l = \text{true} \land \delta' \neq \bot \bigcup_{\delta' \in S_{\delta}} \emptyset \text{ otherwise}$.

Since all $\delta \in B_{b,S}^n(\sigma)$ validates $I$, $\llbracket b \rrbracket \delta = \{\text{true}\}, \{\text{false}\}$, or $\{\text{true}, \text{false}\}$. In the case of $\text{true} \in \llbracket b \rrbracket \delta$, $\delta'$ validates the precondition of the premise. Hence, for all $\delta' \in S_{\delta}$, $\delta'$ is not $\bot$ and also validates $I$. The case of $\llbracket b \rrbracket \delta = \{\text{false}\}$ is not of interest.

Recall $C_{b,S}^{n+1}(\sigma) \equiv C_{b,S}^n(\sigma) \cup \bigcup_{\delta \in B_{b,S}^n(\sigma)} \bigcup_{l \in \llbracket b \rrbracket \sigma} \{\delta\} \text{ if } l = \text{false} \bigcup_{\delta' \in S_{\delta}} \{\bot\} \text{ otherwise}$.

Since all $\gamma \in C_{b,S}^n(\sigma)$ validates $I$ and $\llbracket b \rrbracket(\text{false})$, we only need to care the rightmost part of the construction. Since all $\delta \in B_{b,S}^n(\sigma)$ validates $I$, by the side-condition, $\text{unknown}$ and $\bot$ are not in $\llbracket b \rrbracket \delta$. The $\delta$ is added to $C_{b,S}^{n+1}(\sigma)$ only if $\text{false} \in \llbracket b \rrbracket \delta$. Therefore, $\delta$ validates both $I$ and $\llbracket b \rrbracket(\text{false})$.

Also, in the case of $\text{true} \in \llbracket b \rrbracket \delta$, since $\delta$ validates the precondition in the premise, $\bot \notin S_{\delta}$. Therefore, $\bot \notin C_{b,S}^{n+1}$.