Logical Methods in Computer Science
Volume 20, Issue 2, 2024, pp. 17:1–17:55
https://lmcs.episciences.org/

Submitted Jun. 08, 2021
Published Jun. 24, 2024

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

SEWON PARK [a], FRANZ BRAUSSE [b], PIETER COLLINS [c], SUNYOUNG KIM [d], MICHAL KONEČNÝ [e], GYESIK LEE [f], NORBERT MÜLLER [g], EIKE NEUMANN [h], NORBERT PREINING [i], AND MARTIN ZIEGLER [a]

[a] School of Computing, KAIST, Daejeon, Republic of Korea
*e-mail address*: swelite@kaist.ac.kr, ziegler@kaist.ac.kr

[b] University of Manchester
*e-mail address*: franz.brausse@manchester.ac.uk

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

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

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

[f] Hankyong National University, Anseong, Republic of Korea
*e-mail address*: gslee@hknu.ac.kr

[g] Universität Trier, Trier, Germany
*e-mail address*: mueller@uni-trier.de

[h] Swansea University, Swansea, UK
*e-mail address*: neumaef1@gmail.com

[i] arXiv, Cornell Tech, Komatsu, Japan
*e-mail address*: norbert@preining.info

*Key words and phrases:* Computable Analysis, Reliable Numerics, Imperative Programming, Turing-Completeness, Decidability, Hoare Logic, Kleene Logic.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.46298/LMCS-20(2:17)2024

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 to be essential to real number computation, we use a Plotkin powerdomain and make 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 and model-complete. 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 specifications. Various examples demonstrate the practicality and convenience of our language and the extended Hoare logic.

## CONTENTS

## 1. Introduction

Real numbers in computable analysis are expressed and manipulated exactly without rounding errors by using infinite representations [Wei00, PER89]. For example, a real number $x$ is represented by an infinite sequence of integers $\phi_n$ such that $|x - \phi_n \cdot 2^{-n}| \leq 2^{-n}$ holds for all $n$. When we have *correctly* computed a real number $x$ under this framework, we automatically get access to the real number $x$ via any arbitrarily accurate precision. This property makes this approach promising for application domains where high-precision numerical results are required. Several implementations substantiate this approach of exact real number computation: AERN [KTD$^+$13], Ariadne [CGC$^+$20], Core2 [YYD$^+$10], Cdar [Bla], iRRAM [Mül01], and realLib [Lam07] to name a few. They provide their own data types of real numbers to programmers keeping complicated infinite representations in their back-ends. Abstracting away such details, the programmers can deal with real numbers as they were abstract algebraic entities closely resembling the classical mathematical structure of real numbers with the absence of rounding errors. According to this observation, it is expected that reasoning about programs' behaviours in this approach becomes more intuitive; cf. [KMP$^+$08, Mel08, BFM09, BM11].

Imperative programming is a ubiquitous style of expressing computation that is widely adopted in numerical computations admitting rich theories of precondition-postcondition-based program verification methodologies [Hoa69, AO19, Rey02, NMB06] and their successful applications such as [FP13]. Hence, to conveniently achieve provably correct arbitrary accurate numerical computations, it is desired to extend them with exact real number computation. To make such an extension rigorous, we first acquire a formal programming language with exact real computation 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 naïvely with the classical structure of real numbers as it is not identical, though resembles much thanks to exact operations, the computational structure of real numbers [Her99, Section 3]. For example, the real number comparison $x < y$ as a binary mapping is only partially computable causing non-termination when $x = y$ [Wei00, Theorem 4.1.16]. This property makes multi-valued functions and nondeterminism essential notions in exact real computation [Luc77]. We explain this in more detail later in Section 1.2.

The aim of this paper is to devise an imperative programming language ERC for exact real number computation that correctly deals with the partial and nondeterministic nature in the following sense. We formalize ERC's formal domain-theoretic denotational semantics in such a way that precisely the computable partial real functions and integer multivalued functions in computable analysis can be defined in ERC. Based on the formal semantics, we devise a sound Hoare-style logic of ERC specifications. The language ERC is expressive 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. For this purpose, we develop our language and theory relative to primitive operator extensions such that other frameworks providing richer sets of primitive operators can still benefit from ERC to analyze their real number computations.

### 1.1. Related Works.

In terms of formalizing programming languages for exact real number computation, this paper 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 the 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 instead such that we do not need to have a separate tool for reasoning programs' termination. Later in Remark 2.5, we discuss more about using a different powerdomain.

When we focus on imperative programming languages for exact real number computation, the Blum-Cucker-Shub-Smale model [BCSS98] aka real RAM [PS85, §1.4] captures real numbers as algebraic structure: It supposes tests as total, and renders transcendental functions uncomputable. The 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 iRRAM. Amongst further related works including [BC90, 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, a specification language, and reasoning principles.

### 1.2. **Kleenean-valued Comparison and Nondeterminism.**

An inevitable side-effect of using infinite representations for exact operations is that the ordinary order relation of real numbers as a Boolean-valued function becomes partial. Comparing $x < y$ does not terminate when $x = y$ whichever specific representation is used [Wei00, Theorem 4.1.16]. Instead of making an entire procedure fail when the same numbers are compared, some frameworks, including AERN and iRRAM, introduce the concept of Kleenean $\mathbb{K} := \{\mathit{true}, \mathit{false}, \mathit{unknown}\}$, a lazy extension of Boolean. The third value *unknown* denotes an explicit state of non-termination that is delayed until tested. Consider the following Kleenean-valued comparison

$$x \lessdot y := \begin{cases} \mathit{true} & \text{if } x < y, \\ \mathit{false} & \text{if } y < x, \\ \mathit{unknown} & \text{otherwise,} \end{cases} \tag{1.1}$$

as a replacement for the ordinary Boolean-valued comparison test. When $x = y$, the comparison $x \lessdot y$ evaluates to *unknown* instead of diverging immediately. Though *unknown* is an admissible value that can be assigned to Kleenean-typed variables when later it is required to test if the value is *true* or *false*, for example as the condition of a loop statement, it then causes non-termination.

A remark worth making here is that the third truth value *unknown* is only denotational in the sense that an expression having *unknown* value cannot be observed during a computation. Though we can reason that $x \lessdot y$ must be *unknown* when we know that $x = y$, we cannot observe $x \lessdot y$ having its value *unknown* and conclude $x = y$. In other words, we cannot write a program that branches differently on the three values of $\mathbb{K}$.

Due to the non-termination in comparisons, multi-valued functions and their nondeterministic computations become essential [Luc77] to compose a total but nondeterministic procedure from possibly non-terminating sub-procedures. Suppose we want to approximate $x < y$ with a tolerance factor $2^{-n}$ and consider the two Kleenean-valued comparisons $x \lessdot y + 2^{-n}$ and $y \lessdot x + 2^{-n}$. We can test their truths in parallel until one turns out to be *true*. Even in the case when one of them is *unknown*, hence the test faces non-termination, it is promised that the other comparison evaluates to *true*. Therefore, the parallel procedure can terminate safely and inform us if $x < y + 2^{-n}$ or $y < x + 2^{-n}$. When the two *true* conditions overlap $y - 2^{-n} < x < y + 2^{-n}$ it becomes nondeterministic as it will depend on how the real numbers and the parallel procedure are implemented which we abstract away. We model this nondeterministic computation by a multi-valued function whose value is $\{true, false\}$ at such inputs denoting the set of possible values of the nondeterministic computation.

As nondeterminism becomes essential, most exact real computation frameworks provide their users with various primitive constructs for nondeterminism. In iRRAM, an operator $\mathsf{choose}$ is provided 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

$$\mathsf{choose}(x \lessdot y + 2^{-n}, y \lessdot x + 2^{-n}) = 0.$$

1.3. **Main Contributions and Overviews.** In this paper, we define an imperative programming language *Exact Real Computation* (ERC) providing three primitive data types $\mathsf{K}$, $\mathsf{Z}$, and $\mathsf{R}$ that denote 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 comparison $\lessdot$ for real numbers replacing the ordinary total comparison and offering the nondeterminism construction $\mathsf{choose}(b_0, b_1, \ldots)$. We define the denotations of terms and commands to be functions to Plotkin powerdomains [Plo76] to model nondeterministic computations. The definitions of computable partial (multi-valued) functions are reviewed in Section 2.

Construction of limits, which is an essential feature that makes exact real computation more expressive than algebraic computations [Bra03], is available in a restricted way in ERC. 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, ERC supports limits to be computed at the level of "programs", another layer over commands representing (multi-valued) functions. A real-valued program is provided with access to an integer variable $p$ such that when the program computes (possibly multi-valued) $2^p$ approximations to a real number, 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 of equipping limits 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-valued function or a real program expressing a partial real function where limit operations are taken by default. Later in Remark 5.5, we discuss more about this design choice.

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

The base language without any extension $\mathrm{ERC}_0 := \mathrm{ERC}(\emptyset, \emptyset)$ does not provide integer multiplications or the naïve coercion $\mathbb{Z} \ni z \mapsto z \in \mathbb{R}$ as its term constructs. Instead, it provides the *precision embedding* $\imath : \mathbb{Z} \ni p \to 2^p \in \mathbb{R}$ as a way to convert an integer to real. Although this looks like a bit of restriction at first glance, since integer multiplication is expressible using a loop in our command language, we claim that our programming language itself is still expressive by showing its Turing-completeness in Theorem 5.3 and various examples in Section 5. This restriction and the strict hierarchy between ERC's term language and command language are motivated by the specification language of ERC which for $\mathrm{ERC}_0$'s terms is adequate and admits a decidable theory.

We introduce a first-order language over the three types (sorts) expanded with $(\mathcal{F}, \mathcal{G})$ as the specification language of ERC. We show that the first-order language can define the semantics of our nondeterministic term language. Moreover, for the default language $\mathrm{ERC}_0$, whose design is influenced by the structure suggested in [Dri85], the theory of the specification language becomes model-complete and decidable with regard to the canonical interpretation. Nonetheless, due to the fundamental trade-off, the specification language for $\mathrm{ERC}_0$ is not expressive enough to define the semantics of the command language.

**Remark 1.1.** Consider the following three desirable features:

i) a programming language being Turing-complete over integers, and thus able to realize an algorithm whose termination is co-r.e. hard
ii) a specification language sufficiently rich to express the termination of said algorithm
iii) a sound and complete r.e. deductive system of said specification language.

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

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

Based on the specification language, we extend the classical Hoare logic for ERC commands' total correctness

$$\{\phi\}\ S\ \{\psi\}$$

that says for any initial state satisfying the precondition $\phi$, all possible executions of the command $S$ regarding the nondeterminism in ERC terminate and each yields a state satisfying the postcondition $\psi$. The precondition and postconditions are formulae in the specification language. We prove the soundness of the extended Hoare logic with regard to the denotational semantics. We demonstrate the extended Hoare logic to prove the correctness of a root-finding program.

To summarize our main contributions:

- A choice of primitive operations over real numbers and their computable (multi-valued) semantics in Section 2.

- A small imperative programming language with three basic data types $\mathsf{R}$, $\mathsf{Z}$, and $\mathsf{K}$, and the primitive operations for exact (multi-valued) computations over the algebraic structure of $\mathbb{R}$, $\mathbb{Z}$, and $\mathbb{K}$ regarding the said semantics in Sections 3 and 4.
- 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 in Section 5
- Proof that this programming language is Turing-complete: those and only those real functions and integer multi-valued functions that can be realized in ERC are computable in the sense of computable analysis in Theorems 4.8 and 5.3.
- A many-sorted structure for the formal specification of ERC whose first-order theory is decidable and model-complete in the case of $\mathrm{ERC}_0$ in Section 6 and Theorem 6.2.
- An extended Hoare logic to enable formal verification of ERC programs' behaviours including their terminations in Section 6.3 and Theorem 6.11.
- An example correctness proof using the extended Hoare logic of the root-finding example program in Section 7.

This paper is structured as follows. In Section 2, we review the definitions of computable partial functions and computable partial multi-valued functions in computable analysis and the definition of Plotkin powerdomain over flat domains. We specify several computable partial functions and multi-valued functions that constitute our primitive operators. Section 3 defines the formal definition of our language syntax and typing rules. In Section 4, we define domain-theoretic denotational semantics using the Plotkin powerdomain and prove that for any well-typed terms, well-formed commands, and well-typed programs, their denotations are computable partial multi-valued functions (or partial real functions in the case of real programs). Section 5 defines some programming abbreviations and introduces various example programs which are: square root using Heron's method, 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-valued functions are realizable in ERC.

We introduce in Section 6 a three-sorted structure and its first-order language for specifying nondeterministic programs in ERC. We prove that its first-order theory is decidable and model-complete in the case of $\mathrm{ERC}_0$. Section 6.3 extends the classical Hoare logic from $\mathsf{Z}$ to ($\mathsf{K}$ and) $\mathsf{R}$ and prove its soundness. Section 7 demonstrates our verification method by proving the correctness of the aforementioned *tri*section program for root finding. We conclude this work with Section 8 suggesting future research including extending ERC from operating on real numbers to operating on functions to express operators and other higher-order data types.

## 2. Preliminaries

To devise sound mathematical semantics for unbounded loops, 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 we review later in this section) $\mathbb{P}(Y_\perp)$ on flat domains $Y_\perp$. An element of the domain $p \in \mathbb{P}(Y_\perp)$ is a subset of $Y \cup \{\perp\}$. Hence, a mapping $f : X \to \mathbb{P}(Y_\perp)$, which is a partial multi-valued function, models a nondeterministic computation whose return values are in

$Y$ where $f(x) \subseteq Y \cup \{\perp\}$ denotes the set of all possible return values of the computation at input $x$. When $f(x)$ contains $\perp$, it denotes the case where there is a nondeterministic branch that fails. In this sense, we say $f$ is partial and defined only at $x$ where $\perp \notin f(x)$.

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

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 or to powerdomains directly such that we do not have to handle many different computability notions at the same time. Remark 2.6 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 $\delta_X$ of it is a partial surjective function from the set of infinite sequences of integers to $X$. [1] 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.

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

$$\begin{aligned}
\delta_\mathbb{Z}(\varphi) = k &:\Leftrightarrow \varphi(n) = k \text{ for all } n \in \mathbb{N}, \\
\delta_\mathbb{R}(\varphi) = x &:\Leftrightarrow |x - \varphi(n) \cdot 2^{-n}| \leq 2^{-n} \text{ for all } n \in \mathbb{N}, \\
\delta_\mathbb{K}(0\#0 \cdots 0\#1\#1 \cdots) &= \mathit{true}, \\
\delta_\mathbb{K}(0\#0 \cdots 0\# -1\# -1 \cdots) &= \mathit{false}, \\
\delta_\mathbb{K}(0\#0 \cdots 0\#0 \cdots) &= \mathit{unknown}.
\end{aligned}$$

Here, $a\#\phi$ when $a \in \mathbb{Z}$ and $\phi \in \mathbb{Z}^\mathbb{N}$ denotes the infinite sequence where $a$ is prepended to $\phi$ as its new head. In a name of some $b \in \mathbb{K}$, the digit 0 acts as a token saying that we do not know yet if $b$ is *true* or *false*, the digit 1 acts as a token saying that we now know that $b$ is *true*, and the digit $-1$ acts as a token saying that we now know that $b$ is *false*. Note that for the name $0\#0\#0 \cdots$ of $b = \mathit{unknown}$, it is not possible to decide in a finite time if $b$ is *unknown* or if it will happen to be *false* or *true* in the future. In other words, the procedure deciding the value of $b$, if $b = \mathit{true}$ or *false*, diverges when $b = \mathit{unknown}$.

Though we present the specific representations and work with them in this paper, 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 $\mathbb{K}$:

$$\big\{\emptyset, \{\mathit{true}\}, \{\mathit{false}\}, \{\mathit{true}, \mathit{false}\}, \{\mathit{unknown}, \mathit{true}, \mathit{false}\}\big\} .$$

As we are only interested in computability in this paper, the underlying representations can be replaced with any other 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 $\mathrm{dom}(f) \coloneqq \{x \in X \mid f(x) \neq \perp\}$. When $X$ and $Y$ are represented by $\delta_X$ and $\delta_Y$, respectively, $f$ is said $(\delta_X, \delta_Y)$-***computable*** if there is a type-2 machine [Wei00, Chapter 2.1] that for all $x \in \mathrm{dom}(f)$ transforms each $\delta_X$-name of $x$ to a

[1] 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].

$\delta_Y$-name of $f(x)$.[2] In this case, we say the type-2 machine ***realizes*** $f$. We drop the prefix $(\delta_X, \delta_Y)$ if it is obvious or irrelevant which specific representations underlie. An important fact to note here is that the realizer of $f : X \to Y_\perp$ does not need to diverge in the case $f(x) = \perp$. The realizer on any name of such $x$ not in the domain can either diverge or even compute a meaningless result.

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 it. 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 $f : X_1 \times \cdots \times X_d \to Y_\perp$, its computability is defined by the existence of a type-2 machine with multiple inputs realizing it in a similar manner. Equivalently, it can be defined with regard to the product representation for its domain:

$$\delta_{X_1\times\cdots\times X_d}(\varphi) = (x_1, \cdots, x_d) \text{ if and only if } \delta_{X_j}(n \mapsto \varphi(d \times n + j - 1)) = x_j \text{ for each } j.$$

Again, other computably equivalent representations of products can replace this specific product representation.

The following are examples of computable partial and total functions:

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

(1) For any represented set $X$, its identity function $\mathrm{id} : X \to X$ 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 $\hat{\neg} : \mathbb{K} \to \mathbb{K}$, $\hat{\vee}, \hat{\wedge} : \mathbb{K} \times \mathbb{K} \to \mathbb{K}$ defined as the following tables are computable. Note that *unknown* stands for indeterminacy.

| $\hat{\wedge}$ | *true* | *false* | *unknown* |
|---|---|---|---|
| *true* | *true* | *false* | *unknown* |
| *false* | *false* | *false* | *false* |
| *unknown* | *unknown* | *false* | *unknown* |

| $\hat{\vee}$ | *true* | *false* | *unknown* |
|---|---|---|---|
| *true* | *true* | *true* | *true* |
| *false* | *true* | *false* | *unknown* |
| *unknown* | *true* | *unknown* | *unknown* |

| $\hat{\neg}$ | *true* | *false* | *unknown* |
|---|---|---|---|
| | *false* | *true* | *unknown* |

This justifies the name *Kleenean* for $\mathbb{K}$.
(4) The operators of the Presburger arithmetic $+ : \mathbb{Z}\times\mathbb{Z} \to \mathbb{Z}$, $- : \mathbb{Z} \to \mathbb{Z}$, and the $\mathbb{K}$-valued comparison tests $=, \leq : \mathbb{Z} \times \mathbb{Z} \to \mathbb{K}$ (which does not admit *unknown*) are computable.
(5) The field operators $+, \times : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, $- : \mathbb{R} \to \mathbb{R}$, and $^{-1} : \mathbb{R} \to \mathbb{R}_\perp$ where the multiplicative inversion is not defined at 0, i.e., $0^{-1} = \perp$, are computable. (Any realizer for the multiplicative inversion function will diverge on $0^{-1} = \perp$.)
(6) The exponentiation $\imath : \mathbb{Z} \ni p \mapsto 2^p \in \mathbb{R}$ called precision embedding is computable.

[2] It also can be defined equivalently using oracle machines [KC96].

(7) The $\mathbb{K}$-valued real number comparison $\lessdot : \mathbb{R}\times\mathbb{R} \to \mathbb{K}$ from Equation (1.1) is computable. It is realized by a procedure that given some names $\phi_x, \phi_y$ of $x, y \in \mathbb{R}$ computes the sequence of integers:

$$n \mapsto \begin{cases} 1 & \text{if } \phi_x(n) + 2 < \phi_y(n), \\ -1 & \text{if } \phi_y(n) + 2 < \phi_x(n), \\ 0 & \text{otherwise.} \end{cases}$$

We can easily verify that the resulting sequence is a name of $x \lessdot y$.

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

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

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

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

are computable.

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

2.2. **Computable Infinite Multi-valued functions.** Partial multi-valued functions are non-empty set-valued functions $f : X \to \mathcal{P}(Y_\perp)$ that model nondeterministic computations in the following sense. 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 nondeterministic branch for which the computation fails. We say a function of type $f : X \to \mathcal{P}(Y_\perp)$ a (*partial*) *multi-valued function* which is properly defined at $\mathrm{dom}(f) \coloneqq \{x \mid \perp \notin f(x)\}$. We also often call a partial multi-valued function *total* when its function values never contain $\perp$; i.e., $\mathrm{dom}(f) = X$.

For a partial multi-valued 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 a type-2 machine that for all $x \in \mathrm{dom}(f)$ transforms each $\delta_X$-name of $x$ to a $\delta_Y$-name of some $y \in f(x)$. Note that the ***realizer***'s behaviour on non-valid inputs is not specified. It can either diverge or compute a meaningless value. 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 even for a fixed $x$, for its different names, the realizer can compute different $y, y' \in f(x)$ though the realizer itself is deterministic.

2.3. **Computable (Finite) Multi-valued functions and Plotkin Powerdomain.** Using the entire non-empty power-set $\mathcal{P}(Y_\perp)$ causes the same problem that motivated [Plo76, Section 2] 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 allowed as inputs there is no necessarily infinite integer multi-valued function (see Fact 2.4), we safely restrict to Plotkin powerdomain [Plo76]. For a set $X$, the powerdomain on its flat domain $X_\perp$ is defined by

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

endowed with Egli-Milner ordering:

$$p \sqsubseteq q \Leftrightarrow (\forall x \in p\, \exists y \in q.\, x \leq y) \wedge (\forall y \in q\, \exists x \in p.\, x \leq y)\ .$$

Here, $x \leq y$ is the order on the flat domain that holds if and only if $x = \perp$ or $x = y$. The constructed powerdomain for any flat domain $X_\perp$ is known to be a $\omega$-cpo with the least element $\{\perp\}$ [Rey09, Chapter 7.2]. We say a partial multi-valued function of type $f : X \to \mathbb{P}(Y_\perp) \subseteq \mathcal{P}(Y_\perp)$ *finite*.

For a function to a flat-domain $f : X \to Y_\perp$, let us write $f^\ddagger : X \to \mathbb{P}(Y_\perp)$ for the embedding $f^\ddagger(x) \coloneqq \{f(x)\}$. For a multivariate partial multi-valued 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

$$f^\dagger(p_1, \cdots, p_d) \coloneqq \bigcup_{x_1 \in p_1, \cdots, x_d \in p_d} \begin{cases} \{f(x_1, \cdots, x_d)\} & \text{if } \forall i.\ x_i \neq \perp, \\ \{\perp\} & \text{otherwise.} \end{cases}$$

Furthermore, 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 by

$$f^{\dagger_i}(x_1, \cdots, p, \cdots, x_d) \coloneqq \bigcup_{x_i \in p} \begin{cases} \{f(x_1, \cdots, x_i, \cdots x_d)\} & \text{if } x_i \neq \perp, \\ \{\perp\} & \text{otherwise.} \end{cases}$$

The embedding and extensions can be more naturally understood in the setting of category theory [BVS93, AJ95].

The composition of two partial multi-valued functions $f : X \to \mathbb{P}(Y_\perp)$ and $g : Y \to \mathbb{P}(Z_\perp)$ is expressed by $g^\dagger \circ f : X \to \mathbb{P}(Y)$. This composition is continuous in both arguments [Plo76, Section 6]. In other words, given $f, f_i : X \to \mathbb{P}(Y_\perp)$ and $g, g_i : Y \to \mathbb{P}(Z_\perp)$ where $f_i$ and $g_i$ are increasing chains with regard to the point-wise orderings, it holds that

$$\bigsqcup_{i \in \mathbb{N}} (g^\dagger \circ f_i) = g^\dagger \circ \bigsqcup_{i \in \mathbb{N}} f_i \quad \text{and} \quad \bigsqcup_{i \in \mathbb{N}} (g_i^\dagger \circ f) = \big(\bigsqcup_{i \in \mathbb{N}} g_i\big)^\dagger \circ f.$$

Partial composition is also continuous: Consider a multivariate multi-valued 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-valued functions $g_{i,k} : X \to \mathbb{P}((X_i)_\perp)$ in $k$. Then,

$$\bigsqcup_{k \in \mathbb{N}} f^{\dagger_i} \circ \langle g_1, \cdots, g_{i,k}, \cdots g_d \rangle = f^{\dagger_i} \circ \langle g_1, \cdots, \bigsqcup_{k \in \mathbb{N}} g_{i,k}, \cdots g_d \rangle\ .$$

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

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

$$\mathrm{Cond}(b, p, q) \coloneqq \begin{cases} p & \text{if } b = \mathit{true}, \\ q & \text{if } b = \mathit{false}, \\ \{\perp\} & \text{otherwise,} \end{cases}$$

a conditional combinator of $\mathbb{K}$. For any $b : X \to \mathbb{P}(\mathbb{K}_\perp)$ and $S : X \to \mathbb{P}(X_\perp)$, due to the continuity of compositions, the following operator can be seen as continuous:

$$\mathbf{W}(b, S)(f : X \to \mathbb{P}(X_\perp)) \coloneqq \mathrm{Cond}^{\dagger_1} \circ \langle b, f^\dagger \circ S, \mathrm{id}^\ddagger \rangle\ .$$

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

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

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

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

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

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

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

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

is a computable partial multi-valued function.

It is realized by the procedure which when it receives $\phi_0, \cdots, \phi_{n-1}$ as some names of $b_0, \cdots, b_{n-1}$, for each $p = 0, 1, \cdots$, checks $\phi_i(p)$ until it finds $\phi_i(p) = 1$ implying $b_i = \mathit{true}$. If the realizer finds such $i$, it returns $i\#i\#i\#\cdots$ as the name of the index $i \in \mathrm{choose}_n(b_0, \cdots, b_{n-1})$. Note that depending on the given names, the realizer can pick a different index. If there is no choosable index, the realizer will diverge.

(5) For any computable $f : X \to \mathbb{P}(\mathbb{K}_\perp)$ and $g, h : X \to \mathbb{P}(Y_\perp)$, the composite multi-valued function

$$\mathrm{Cond}^{\dagger_1} \circ \langle f, g, h\rangle : X \to \mathbb{P}(Y_\perp)$$

is computable.

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

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

with regard to the topology of $\mathbb{K}$; cmp. [Wei00, Theorem 2.3.8] and "parallel if" from PCF [Plo77]. For any computable $f : X \to \mathbb{P}(\mathbb{K}_\perp)$ and $g, h : X \to \mathbb{P}(\mathbb{R}_\perp)$, the composition $\mathrm{Kond}^{\dagger_1} \circ \langle f, g, h\rangle : X \to \mathbb{P}(\mathbb{R}_\perp)$ is a computable partial multi-valued function.

Here, we make a further explanation of this operation. Though this operation at first glance seems to violate the principle that we cannot make a case distinction on $b \in \mathbb{K}$, since this operation is continuous at $b = \mathit{unknown}$, it is not the case. The operation can be realized without testing $b = \mathit{unknown}$ or $p = q$. We describe one possible realizer of $\mathrm{Kond}^{\dagger_1} \circ \langle f, g, h\rangle$ to make it clearer.

For any $u \in X$, write $r \coloneqq f(u) \in \mathbb{P}(\mathbb{K}_\perp)$, $p \coloneqq g(u), q \coloneqq h(u) \in \mathbb{P}(\mathbb{R}_\perp)$ and assume $\perp \notin \mathrm{Kond}^{\dagger_1}(r, p, q)$. (Recall that we do not need to analyze how our realizer behaves when $\perp \in \mathrm{Kond}^{\dagger_1}(r, p, q)$.) Since $\perp$ is not in any of $p, q, r$, we are given names $\phi_b, \phi_x, \phi_y$ of $b, x, y$, respectively, where $b \in r$, $x \in p$, and $y \in q$. For some $z \in \mathrm{Kond}^{\dagger_1}(r, p, q)$ and its name $\phi$, the realizer needs to compute $n \mapsto \phi(n)$.

The realizer first tests if $\phi_x(n+2)$ and $\phi_y(n+2)$ differ by more than 2. In this case (i), since we can conclude that $x \neq y$, we test if $b = \mathit{true}$ or $b = \mathit{false}$ by inspecting $\phi_b(0), \phi_b(1), \cdots$. If $b = \mathit{true}$, since $x \in \mathrm{Kond}^{\dagger_1}(r,p,q)$, let $x$ be the $z$ and $\phi_x$ be the $\phi_z$ and let the realizer return $\phi_x(n)$ for the $\phi_z(n)$. We can let the realizer work similarly in the other case $b = \mathit{false}$. If $b$ was *unknown*, the realizer's inspecting procedure will diverge. However, this is against our assumption that $b = \mathit{unknown}$ and $p \neq q$ makes $\perp \in \mathrm{Kond}^{\dagger_1}(r,p,q)$.

For the other case (ii) where $\phi_x(n+2)$ and $\phi_y(n+2)$ are different by at most 2, from the two integers, the realizer can compute $k \in \mathbb{Z}$ where $|x - k \cdot 2^{-n}| \leq 2^{-n}$ and $|y - k \cdot 2^{-n}| \leq 2^{-n}$ hold. Without being concerned with $b$, the realizer returns the computed $k$ for $\phi_z(n)$ as a valid $n$'th entry of some names of both $x$ and $y$. There are some hypothetical cases to analyze. If $x = y$, then the realizer will execute this second case (ii) indefinitely for all $n$ and yield a name of $x$ (which can be different from $\phi_x$ or $\phi_y$). Regardless of $b$, even when $b = \mathit{unknown}$, since $x \in \mathrm{Kond}^{\dagger_1}(r,p,q)$ as long as $\perp \notin \mathrm{Kond}^{\dagger_1}(r,p,q)$, we can conclude that the realizer is computing a correct name. Otherwise, if $x \neq y$, after returning some finite entries of a name that works for both $x$ and $y$, the realizer will execute the first case (i) which is already analyzed to be correct.

Note that in the case $b = \mathit{unknown}$, $p, q$ being singleton is crucial. When we ease the condition to $p = q$ without forcing them to be a singleton, there is a case where the nondeterministically picked inputs $x \in p$ and $y \in q$ are different. In this case, at some point, the realizer executes the first case (i) and diverges by inspecting $b$.

Observe from the definitions that a realizer of a partial (multi-valued) function can be seen also as a realizer of some other partial (multi-valued) functions because (1) the out-of-domain behaviour of realizers is not specified and (2) for a partial multi-valued function, its realizer only has to compute some of the possible function values. Specifically:

**Fact 2.3.** If a partial function $f : X \to Y_\perp$ is computable, any domain restriction $g : X \to Y_\perp$ such that $\forall x.\ g(x) \leq f(x)$ is computable by the same realizers. Similarly, if a partial multi-valued function $f : X \to \mathcal{P}(Y_\perp)$ is computable, any domain restriction and function-value enlargement $g : X \to \mathcal{P}(Y_\perp)$ such that $\forall x.\ g(x) \sqsubseteq f(x)$ or $f(x) \subseteq g(x)$ is computable by the same realizers.

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 type-2 computable realizer of an integer partial multi-valued function $f : X_1 \times \cdots \times X_d \to \mathbb{P}(\mathbb{Z}_\perp)$ where $X_i = \mathbb{Z}$ or $\mathbb{R}$, for fixed inputs $(x_1, \cdots, x_d)$, the image of the realizer of the compact set of names is also compact as any type-2 computable realizer is continuous. That means even when $f(x_1, \cdots, x_d)$ is infinite, there are only finitely many different integers that its realizer actually computes; see [Bra95, Theorem 4.2].

**Fact 2.4.** Any computable multi-valued 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-valued function $g : X_1 \times \cdots \times X_d \to \mathbb{P}(\mathbb{Z}_\perp)$ such that $\forall (x_1, \cdots, x_d).\ g(x_1, \cdots, x_d) \subseteq f(x_1, \cdots, x_d)$ holds.

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

**Remark 2.5.** Note that $\mathrm{choose}_n : \mathbb{K}^n \to \mathbb{P}(\mathbb{Z}_\perp)$ is not (domain-theoretic) continuous when $\mathbb{K}$ is ordered by $b_1 \leq b_2$ iff $b_1 = b_2$ or $b_1 = \mathit{unknown}$. This is observed in [MRE07] and there it is suggested to use Hoare powerdomain instead. However, our approach distinguishes (domain-theoretic) continuity and computability of primitives: we do not use (domain-theoretic) continuity as a criterion for computability. We use continuity to define the semantics of loops whose computability is studied separately using the notion of type-2 computability of realizers. In other words, as long as we have a fact that $\mathrm{choose}_n : \mathbb{K}^n \to \mathbb{P}(\mathbb{Z}_\perp)$ is computable, there is no need to make it (domain-theoretic) continuous by giving an order on $\mathbb{K}$. This way we still can use Plotkin powerdomain in defining our semantics such that we do not need to have separate operational semantics to express termination as in [MRE07].

We conclude this section with a remark on how the formulations of computable partial functions and multi-valued functions in this paper are related to the traditional formulations in computable analysis.

**Remark 2.6.** In the traditional setting of computable analysis:

(1) The computability of a partial function presented by $f :\subseteq X \to Y$ is equivalent to the computability of $f_\perp : X \to Y_\perp$ defined by $x \mapsto f(x)$ if $x \in \mathrm{dom}(f)$ and $\perp$ if $x \notin \mathrm{dom}(f)$. Similarly, the computability of $f : X \to Y_\perp$ is equivalent to the computability of $f\restriction_{\mathrm{dom}(f)} :\subseteq X \to Y$. Here, $\restriction$ is a notation for the domain restriction and recall that $\mathrm{dom}(f) = \{x \mid f(x) \neq \perp\}$.

(2) The computability of a partial multi-valued function presented by $f :\subseteq X \rightrightarrows Y$, 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 $f\restriction_{\mathrm{dom}(f)} :\subseteq X \rightrightarrows Y$. Recall that $\mathrm{dom}(f) = \{x \mid \perp \notin f(x)\}$.

## 3. FORMAL SYNTAX AND TYPING RULES OF *Exact Real Computation*

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

**Convention 3.1.** Fix a (possibly empty) finite set $\mathcal{F}$ of computable partial 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 $\mathcal{G}$ of computable finite partial multi-valued functions to $\mathbb{P}(\mathbb{Z})$ 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:

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

See that ERC provides countably many data types: for each natural number $n \geq 1$, there is a data type $\mathsf{R}[n]$, 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 judged later by typing rules in Section 3.2, here we follow the following conventions: Write $m, \ell$ to denote terms which should be typed $\mathsf{Z}$, $u, v$ to denote terms which should be typed $\mathsf{R}$, and $b$ to denote terms which should be typed $\mathsf{K}$. Arbitrary terms are denoted by $t$. For some fixed countable set of variables $\mathcal{V}$, the term language of ERC is defined as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{t, m, \ell, u, v, b ::=} \\
\quad k & k \in \mathbb{Z};\ \mathsf{Z} \text{ literal} \\
\mid \mathit{true} \mid \mathit{false} \mid \mathit{unknown} & \mathsf{K} \text{ literal} \\
\mid [t_1, \cdots, t_n] & \text{array literal} \\
\mid x & x \in \mathcal{V};\ \text{variable} \\
\mid t[m] & \text{array access} \\
\mid t_1 + t_2 \mid -t & \text{addition and additive inversion} \\
\mid m \leq \ell \mid m = \ell & \text{integer comparison and equality test} \\
\mid u \times v \mid u^{-1} & \text{real multiplication and multiplicative inversion} \\
\mid u \lessdot v & \text{real comparison} \\
\mid 2^m & \text{precision embedding} \\
\mid \hat{\neg}\, b \mid b_1 \mathbin{\hat{\wedge}} b_2 \mid b_1 \mathbin{\hat{\vee}} b_2 & \text{Kleene logic} \\
\mid f(t_1, \cdots, t_d) & (f : X_1 \times \cdots \times X_d \to \mathbb{R}_\perp) \in \mathcal{F};\ \text{primitive function call} \\
\mid g(t_1, \cdots, t_d) & (g : X_1 \times \cdots \times X_d \to \mathbb{P}(\mathbb{Z}_\perp)) \in \mathcal{G}; \\
 & \quad \text{primitive multi-valued function call} \\
\mid \mathsf{choose}_n(b_0, b_1, \cdots, b_{n-1}) & n \in \mathbb{N};\ \text{multi-valued choice} \\
\mid b \mathrel{?} u : v & \text{continuous conditional}
\end{array}
$$

Note that we do not provide integer multiplication and any other coercion than $2^m$ unless they are provided through $\mathcal{F}, \mathcal{G}$. For each natural number $n$, $\mathsf{choose}_n$ is a distinct term construct. However, since the $n$ is determined syntactically by the number of the arguments, we may omit it and instead, simply refer to $\mathsf{choose}()$. We introduce $m = \ell$ as a term construct instead of as an abbreviation for $m \leq \ell \mathbin{\hat{\wedge}} m \leq \ell$ because their denotations which are defined later in Section 4.1 do not agree.

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

$$
\begin{array}{lll}
S & ::= \ \mathbf{skip} & \text{skip} \\
 & \mid x := t & \text{variable assignment} \\
 & \mid x[m] := t & \text{array assignment} \\
 & \mid \mathbf{let}\ x : \tau = t & \text{variable declaration} \\
 & \mid S_1;\ S_2 & \text{sequential composition} \\
 & \mid \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end} & \text{branching} \\
 & \mid \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{end} & \text{loop}
\end{array}
$$

The variable declaration **let** statement introduces a variable with limited scope in a loop or a conditional branch; see the related typing rules in Figure 2.

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 defined to be in either of the two forms:

| | |
|---|---|
| **input** $x_1 : \tau_1, x_2 : \tau_2, \cdots, x_n : \tau_n$ | **input** $x_1 : \tau_1, x_2 : \tau_2, \cdots, x_n : \tau_n$ |
| $S$ | $S$ |
| **return** $t$ | **return** $t$ **as** $p \to -\infty$ |

Here, $x_i$ and $p$ are variables varying over $\mathcal{V}$ and $\tau_i$ varies over $\{\mathsf{Z}, \mathsf{R}, \mathsf{R}[1], \mathsf{R}[2], \cdots\}$. The variable $p$ that is separated from the other input variables is the precision parameter. The intended behaviour of a program in the right kind is, taking $x_1, \cdots, x_n$ 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*** which is a mapping from a finite set of variables to their corresponding types. We use a list of assignments $\Gamma \coloneqq x_1 : \tau_1, x_2 : \tau_2, \cdots, x_n : \tau_n$ to represent the context mapping a variable $x_i$ 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 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.

The only construct that modifies a context is **let** $x : \tau = t$ (variable declaration). When it is executed under a context $\Gamma$, the command is well-formed if $x$ 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 $\Gamma, x : \tau$. However, when a context changes inside 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++).

Here we state a property of our typing rules that is needed later. We omit detailed proof as it can be seen directly by structural induction on the type judgement.

**Lemma 3.2.** *Whenever $\Gamma \vdash S \triangleright \Gamma'$, the new context $\Gamma'$ is an extension of $\Gamma$ in the sense that there is a context $\Delta$ such that $\Gamma' = \Gamma, \Delta$.*

Figure 1: Typing rules for terms

$$\frac{}{\Gamma \vdash k : \mathsf{Z}} \qquad \frac{}{\Gamma \vdash \mathit{true} : \mathsf{K}} \qquad \frac{}{\Gamma \vdash \mathit{false} : \mathsf{K}} \qquad \frac{}{\Gamma \vdash \mathit{unknown} : \mathsf{K}}$$

$$\frac{\Gamma \vdash t_i : \mathsf{R} \quad (\text{for } i = 1, \cdots, n)}{\Gamma \vdash [t_1, \cdots, t_n] : \mathsf{R}[n]} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash m : \mathsf{Z} \quad \Gamma \vdash t : \mathsf{R}[n]}{\Gamma \vdash t[m] : \mathsf{R}}$$

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \odot t_2 : \tau'} \ (\odot, \tau, \tau') \in \mathbf{BinTy} \qquad \frac{\Gamma \vdash t : \tau}{\Gamma \vdash \star t : \tau'} \ (\star, \tau, \tau') \in \mathbf{UnTy}$$

$$\frac{(f : X_1 \times \cdots \times X_d \to \mathbb{R}_\perp) \in \mathcal{F} \quad \Gamma \vdash t_i : \langle X_i \rangle \quad (\text{for } i = 1, \cdots, d)}{\Gamma \vdash f(t_1, \cdots, t_d) : \mathsf{R}}$$

$$\frac{(g : X_1 \times \cdots \times X_d \to \mathbb{P}(\mathbb{Z}_\perp)) \in \mathcal{G} \quad \Gamma \vdash t_i : \langle X_i \rangle \quad (\text{for } i = 1, \cdots, d)}{\Gamma \vdash g(t_1, \cdots, t_d) : \mathsf{Z}}$$

$$\frac{\Gamma \vdash b_i : \mathsf{K} \quad (\text{for } i = 0, 1, \cdots, n-1)}{\Gamma \vdash \mathsf{choose}_n(b_0, b_1, \cdots, b_{n-1}) : \mathsf{Z}} \qquad \frac{\Gamma \vdash b : \mathsf{K} \quad \Gamma \vdash u : \mathsf{R} \quad \Gamma \vdash v : \mathsf{R}}{\Gamma \vdash (b \mathbin{?} u : v) : \mathsf{R}}$$

$$\begin{aligned}
\mathbf{BinTy} &\coloneqq \{(+, \mathsf{R}, \mathsf{R}), (+, \mathsf{Z}, \mathsf{Z}), (\leq, \mathsf{Z}, \mathsf{K}), (=, \mathsf{Z}, \mathsf{K}), (\lessdot, \mathsf{R}, \mathsf{K}), (\times, \mathsf{R}, \mathsf{R}), \\
&\qquad (\hat{\wedge}, \mathsf{K}, \mathsf{K}), (\hat{\vee}, \mathsf{K}, \mathsf{K})\} \\
\mathbf{UnTy} &\coloneqq \{(-, \mathsf{R}, \mathsf{R}), (-, \mathsf{Z}, \mathsf{Z}), (2^{\square}, \mathsf{Z}, \mathsf{R}), (\hat{\neg}, \mathsf{K}, \mathsf{K}), (\square^{-1}, \mathsf{R}, \mathsf{R})\} \\
\langle \mathbb{Z} \rangle &\coloneqq \mathsf{Z} \\
\langle \mathbb{R} \rangle &\coloneqq \mathsf{R} \\
\langle \mathbb{R}^n \rangle &\coloneqq \mathsf{R}[n] \quad (\text{for } n = 1, 2, \cdots)
\end{aligned}$$

Figure 2: Typing rules for commands

$$\frac{}{\Gamma \vdash \mathbf{skip} \triangleright \Gamma} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma(x) = \tau}{\Gamma \vdash x \coloneqq t \triangleright \Gamma} \qquad \frac{\Gamma \vdash m : \mathsf{Z} \quad \Gamma \vdash t : \mathsf{R} \quad \Gamma(x) = \mathsf{R}[n]}{\Gamma \vdash x[m] \coloneqq t \triangleright \Gamma}$$

$$\frac{x \notin \mathrm{dom}(\Gamma) \quad \Gamma \vdash t : \tau}{\Gamma \vdash \mathbf{let}\ x : \tau = t\ \triangleright \Gamma, x : \tau} \qquad \frac{\Gamma \vdash S_1 \triangleright \Gamma_1 \quad \Gamma_1 \vdash S_2 \triangleright \Gamma_2}{\Gamma \vdash S_1; S_2 \triangleright \Gamma_2}$$

$$\frac{\Gamma \vdash b : \mathsf{K} \quad \Gamma \vdash S_1 \triangleright \Gamma_1 \quad \Gamma \vdash S_2 \triangleright \Gamma_2}{\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ \triangleright \Gamma} \qquad \frac{\Gamma \vdash b : \mathsf{K} \quad \Gamma \vdash S \triangleright \Gamma'}{\Gamma \vdash \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{end}\ \triangleright \Gamma}$$

3.2.3. *Well-Typed Programs.* Consider the two different kinds of ERC programs:

$$P_{\mathsf{Z}} \coloneqq \begin{array}{l} \mathbf{input}\ x_1 : \tau_1, x_2 : \tau_2, \cdots, x_d : \tau_d \\ \quad S \\ \mathbf{return}\ t \end{array} \qquad P_{\mathsf{R}} \coloneqq \begin{array}{l} \mathbf{input}\ x_1 : \tau_1, x_2 : \tau_2, \cdots, x_d : \tau_d \\ \quad S \\ \mathbf{return}\ t\ \mathbf{as}\ p \to -\infty \end{array}$$

The first program $P_{\mathsf{Z}}$ has type $\tau_1 \times \cdots \times \tau_d \to \mathsf{Z}$ if there is a context $\Gamma'$ such that

$$x_1 : \tau_1, x_2 : \tau_2, \cdots, x_d : \tau_d \vdash S \triangleright \Gamma' \quad \text{and} \quad \Gamma' \vdash t : \mathsf{Z}\,.$$

Similarly, $P_{\mathsf{R}}$ has type $\tau_1 \times \cdots \times \tau_d \to \mathsf{R}$ if there is a context $\Gamma'$ such that

$$p : \mathsf{Z}, x_1 : \tau_1, \cdots, x_d : \tau_d \vdash S \triangleright \Gamma' \quad \text{and} \quad \Gamma' \vdash t : \mathsf{R}\,.$$

Note that $p : \mathsf{Z}$ is not regarded as an input variable and recall that $\tau_i$ varies over $\{\mathsf{Z}, \mathsf{R}, \mathsf{R}[1], \mathsf{R}[2], \cdots\}$ and cannot be $\mathsf{K}$.

We call an ERC program an integer program if it is of the first kind ($P_{\mathsf{Z}}$) and a real program if it is of the second kind ($P_{\mathsf{R}}$).

## 4. Denotational Semantics of *Exact Real Computation*

We define 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:

$$[\![\mathsf{K}]\!] = \mathbb{K}, \quad [\![\mathsf{Z}]\!] = \mathbb{Z}, \quad [\![\mathsf{R}]\!] = \mathbb{R}, \quad [\![\mathsf{R}[n]]\!] = \mathbb{R}^n.$$

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 $\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_d$, then

$$[\![\Gamma]\!] \coloneqq [\![\tau_1]\!] \times \cdots \times [\![\tau_d]\!]\,.$$

(3) Given a context $\Gamma$, an element $\sigma \in [\![\Gamma]\!]$ is called ***state*** which is a specific assignment of variables contained in the domain of the 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. For example, a well-typed term $\Gamma \vdash t : \mathsf{R}$ can evaluate to multiple values in $\mathbb{R}$ under a state $\sigma \in [\![\Gamma]\!]$. 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 $\mathbb{P}([\![\mathsf{R}]\!]_\perp)$.

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

$$\begin{aligned}
[\![\Gamma \vdash k : \mathsf{Z}]\!] &\coloneqq \ \sigma \mapsto \{k\} \\
[\![\Gamma \vdash \mathit{true} : \mathsf{K}]\!] &\coloneqq \ \sigma \mapsto \{\mathit{true}\} \\
[\![\Gamma \vdash \mathit{false} : \mathsf{K}]\!] &\coloneqq \ \sigma \mapsto \{\mathit{false}\} \\
[\![\Gamma \vdash \mathit{unknown} : \mathsf{K}]\!] &\coloneqq \ \sigma \mapsto \{\mathit{unknown}\} \\
[\![\Gamma \vdash [t_1, \cdots, t_n] : \mathsf{R}[n]]\!] &\coloneqq \ \mathrm{id}^{\ddagger\dagger} \circ \langle [\![\Gamma \vdash t_1 : \mathsf{R}]\!], \cdots, [\![\Gamma \vdash t_n : \mathsf{R}]\!] \rangle
\end{aligned}$$

$$
\begin{aligned}
[\![x_1:\tau_1,\cdots,x_i:\tau_i,\cdots,x_d:\tau_d\vdash x_i:\tau_i]\!] &\coloneqq \operatorname{proj}_i^{\ddagger}\\
[\![\Gamma\vdash t[m]:\mathsf{R}]\!] &\coloneqq \operatorname{proj}^{\ddagger^\dagger}\circ\langle[\![\Gamma\vdash t:\mathsf{R}[n]]\!],[\![\Gamma\vdash m:\mathsf{Z}]\!]\rangle\\
[\![\Gamma\vdash t_1\odot t_2:\tau']\!] &\coloneqq \odot^{\ddagger^\dagger}\circ\langle[\![\Gamma\vdash t_1:\tau]\!],[\![\Gamma\vdash t_2:\tau]\!]\rangle\\
&\phantom{\coloneqq}\ \text{for } (\odot,\tau,\tau')\in\textbf{BinTy}\\
[\![\Gamma\vdash\star t:\tau']\!] &\coloneqq \star^{\ddagger^\dagger}\circ[\![\Gamma\vdash t:\tau]\!]\\
&\phantom{\coloneqq}\ \text{for } (\star,\tau,\tau')\in\textbf{UnTy}\\
[\![\Gamma\vdash f(t_1,\ldots,t_d):\mathsf{R}]\!] &\coloneqq f^{\ddagger^\dagger}\circ\langle[\![\Gamma\vdash t_1:\langle X_1\rangle]\!],\cdots,[\![\Gamma\vdash t_d:\langle X_d\rangle]\!]\rangle\\
&\phantom{\coloneqq}\ \text{where } (f:X_1\times\cdots\times X_d\to\mathbb{R}_\perp)\in\mathcal{F}\\
[\![\Gamma\vdash g(t_1,\ldots,t_d):\mathsf{Z}]\!] &\coloneqq g^{\dagger}\circ\langle[\![\Gamma\vdash t_1:\langle X_1\rangle]\!],\cdots,[\![\Gamma\vdash t_d:\langle X_d\rangle]\!]\rangle\\
&\phantom{\coloneqq}\ \text{where } (g:X_1\times\cdots\times X_d\to\mathbb{P}(\mathbb{Z}_\perp))\in\mathcal{G}\\
[\![\Gamma\vdash \mathsf{choose}_n(b_0,\cdots,b_{n-1}):\mathsf{Z}]\!] &\coloneqq \operatorname{choose}_n^{\dagger}\circ\langle[\![\Gamma\vdash b_0:\mathsf{K}]\!],\cdots,[\![\Gamma\vdash b_{n-1}:\mathsf{K}]\!]\rangle\\
[\![\Gamma\vdash (b\ ?\ u:v):\mathsf{R}]\!] &\coloneqq \operatorname{Kond}^{\dagger_1}\circ\langle[\![\Gamma\vdash b:\mathsf{K}]\!],[\![\Gamma\vdash u:\mathsf{R}]\!],[\![\Gamma\vdash v:\mathsf{R}]\!]\rangle
\end{aligned}
$$

□

We often simply write $[\![t]\!]$ instead of $[\![\Gamma\vdash t:\tau]\!]$ 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 $x\in[\![t]\!]\,\sigma$.

Regarding the multi-valuedness, $\mathsf{choose}()$ and functions from $\mathcal{G}$ are the only atomic constructs that yield multi-valuedness. The denotations of all other constructs except the two and continuous conditionals are defined by the embedding of single-valued mappings. For the case of continuous conditionals, we can observe that $\operatorname{Kond}^{\dagger_1}(p,q,r)$ is a singleton when $p$, $q$, and $q$ are singletons. Hence, it does not generate multi-valuedness.

Another remark on the multi-valuedness is that the denotation of integer equality $m_1 = m_2$ does not coincide with that of $m_1\le m_2\mathbin{\hat\wedge} m_2\le m_1$. As an example, consider a state $\sigma$ where $[\![m_1]\!]\,\sigma=\{0,2\}$ and $[\![m_2]\!]\,\sigma=\{1,3\}$. Then, $[\![m_1\le m_2\mathbin{\hat\wedge} m_2\le m_1]\!]\,\sigma=\{\mathit{true},\mathit{false}\}$ whereas $[\![m_1=m_2]\!]\,\sigma=\{\mathit{false}\}$. This justifies introducing $m_1=m_2$ as a separate primitive.

Well-typedness in ERC does not prevent $\perp$: The sources of $\perp$ are (i) an array accessed by a wrong index (proj), (ii) the multiplicative inverse of 0 ($0^{-1}$), (iii) function calls to partial (multi-valued) functions at points out of their domains, (iv) a choice operation without any choosable argument (choose), and (v) a continuous conditional when the two arguments $u,v$ do not match in $\operatorname{Kond}(\mathit{unknown},u,v)$ as $u=v=\{x\}$ for some $x\in\mathbb{R}$. We emphasize here again that the computational meaning of $\perp$ is not non-termination but is more general non-specified behaviour; recall Section 2.

Though $\mathbb{P}(X_\perp)$ allows infinite sets (as long as they contain $\perp$), this case does not occur in 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.

The computability of the denotations follows directly from the computability of the primitive operations and of the compositions stated in Examples 2.1 and Examples 2.2:

**Lemma 4.3.** *The denotation of any well-typed term is a computable partial multi-valued function.*

4.2. **Denotations of Commands.** As the denotation of a well-typed term under a state is 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 4.4.** Recall the (multi-valued) functions specified in Examples 2.1 and Examples 2.2. Given a well-formed command $S$ such that $\Gamma \vdash S \triangleright \Gamma'$, we define its denotation to be a partial multi-valued function $[\![\Gamma \vdash S \triangleright \Gamma']\!] : [\![\Gamma]\!] \to \mathbb{P}([\![\Gamma']\!]_\perp)$ inductively as follows:

$$\begin{aligned}
[\![\Gamma \vdash \mathbf{skip} \triangleright \Gamma]\!] &\coloneqq \mathrm{id}^\ddagger \\
[\![\Gamma \vdash x_i \coloneqq t \triangleright \Gamma]\!] &\coloneqq \mathrm{update}_i^{\ddagger^\dagger} \circ \langle \mathrm{id}^\ddagger, [\![t]\!]\rangle \\
&\quad\ \text{where } \Gamma = x_1 : \tau_1, \cdots, x_i : \tau_i, \cdots, x_d : \tau_d \\
[\![\Gamma \vdash x_i[m] \coloneqq t \triangleright \Gamma]\!] &\coloneqq \mathrm{update}_i^{\ddagger^\dagger} \circ \langle \mathrm{id}^\ddagger, \mathrm{update}^{\ddagger^\dagger} \circ \langle \mathrm{proj}_i^\ddagger, [\![m]\!], [\![t]\!]\rangle\rangle \\
&\quad\ \text{where } \Gamma = x_1 : \tau_1, \cdots, x_i : \mathsf{R}[n], \cdots, x_d : \tau_d \\
[\![\Gamma \vdash \mathbf{let}\ x : \tau = t\ \triangleright \Gamma']\!] &\coloneqq \mathrm{id}^{\ddagger^\dagger} \circ \langle \mathrm{id}^\ddagger, [\![t]\!]\rangle \\
[\![\Gamma \vdash S_1; S_2 \triangleright \Gamma']\!] &\coloneqq [\![\Delta \vdash S_2 \triangleright \Gamma']\!]^\dagger \circ [\![\Gamma \vdash S_1 \triangleright \Delta]\!] \\
[\![\Gamma \vdash \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end} \triangleright \Gamma]\!] &\coloneqq \mathrm{Cond}^{\dagger_1} \circ \langle [\![b]\!], \restriction_\Gamma^{\ddagger^\dagger} \circ [\![\Gamma \vdash S_1 \triangleright \Gamma']\!], \restriction_\Gamma^{\ddagger^\dagger} \circ [\![\Gamma \vdash S_2 \triangleright \Gamma'']\!]\rangle \\
[\![\Gamma \vdash \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{end} \triangleright \Gamma]\!] &\coloneqq \mathbf{lfp}\big(\mathbf{W}([\![b]\!], \restriction_\Gamma^{\ddagger^\dagger} \circ [\![\Gamma \vdash S \triangleright \Gamma']\!])\big)
\end{aligned}$$

Here, $\restriction_\Gamma : [\![\Gamma, \Delta]\!] \to [\![\Gamma]\!]$ for any contexts $\Gamma, \Delta$ when $\Gamma = x_1 : \tau_1, \cdots, x_d : \tau_d$ is defined by

$$(x_1, \cdots, x_d, y_1, \cdots, y_m) \mapsto (x_1, \cdots, x_d). \qquad \square$$

We often write $[\![S]\!]$ to denote $[\![\Gamma \vdash S \triangleright \Gamma']\!]$ omitting $\Gamma$ and $\Gamma'$ when they are obvious or irrelevant.

When a term is assigned to a variable, an array variable, or a newly created variable as its initial value, the underlying state branches into multiple states for each single value in the denotation of the term. Instead of having a single state storing multi-values, we choose to have multiple states where each stores single-values.

The denotations of conditional statements and loops are well-defined due to Lemma 3.2 which justifies the post-composed restriction function $\restriction_\Gamma$ clearing the newly added local variables. Regarding loops and branches, note that their denotations contain *unknown* when their conditions evaluate to *unknown*. We could also consider replacing Cond with Kond for the semantics of conditionals. However, first, their domains do not match: Kond requires the branches to be in $\mathbb{P}(\mathbb{R}_\perp)$ whereas we want them to be in $\mathbb{P}([\![\Gamma]\!]_\perp)$. Though one could imagine extending Kond's domain, recalling how Kond is realized in Examples 2.2, this requires the resulting states of both branches to be inspected variable-wise. For this work, we decide to keep the ordinary branching which makes a decision only on its condition and diverges in the case the condition diverges by being *unknown* instead of further inspecting the two resulting states.

Unlike for terms, the multi-valued computability of loops needs more attention:

**Proposition 4.5.** *For a represented set $X$ and computable partial multi-valued functions $b : X \to \mathbb{P}(\mathbb{K}_\perp)$ and $S : X \to \mathbb{P}(X_\perp)$, the least fixed point*

$$\mathbf{lfp}\big(\mathbf{W}(b, S)\big) : X \to \mathbb{P}(X_\perp)$$

*as partial multi-valued function is computable uniformly to $b$ and $c$ in the sense that the realizer for $\mathbf{lfp}\big(\mathbf{W}(b, S)\big) : X \to \mathbb{P}(X_\perp)$ can be defined with function calls to the realizers of $b$ and $S$ for any $b$ and $S$.*

*Proof.* By the least fixed point theorem, the least fixed point is the limit of the chain

$$\mathcal{W}^{(0)}_{b,S} \coloneqq \sigma \mapsto \{\perp\} \quad \text{and} \quad \mathcal{W}^{(n+1)}_{b,S} \coloneqq \mathrm{Cond}^{\dagger_1} \circ \langle b, \mathcal{W}^{(n)}_{b,S} \circ S, \mathrm{id}^{\ddagger} \rangle$$

which, intuitively, represents the process of unrolling the loop. For any two $\sigma, \delta \in X$, $\delta \in \mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma$ if and only if there is a natural number $n$ where $\delta \in \mathcal{W}^{(n)}_{b,S}\sigma$. Furthermore, $\perp \in \mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma$ if and only if $\perp \in \mathcal{W}^{(n)}_{b,S}\sigma$ for all $n$.

Suppose $\tau_b$ realizes $b$ and $\tau_S$ realizes $S$. We consider the type-2 machine $\tau_{\tau_b,\tau_S}(\phi_\sigma)$ on its input $\phi_\sigma$ a name of some $\sigma \in X$ with the following description:

**input** $\phi_\sigma$
  **repeat**:
    $\phi_b \coloneqq \tau_b(\phi_\sigma)$
    **iterate** $i := 0, 1, 2, \cdots$ **until** $\phi_b(i) = -1$ **or** $\phi_b(i) = 1$
    **if** $\phi_b(i) = 1$:
      $\phi_\sigma \coloneqq \tau_S(\phi_\sigma)$
    **if** $\phi_b(i) = -1$:
      **return** $\phi_\sigma$

We need to prove that if $\perp \notin \mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma$, $\tau_{\tau_b,\tau_S}(\phi_\sigma)$ is defined and is a name of some $\delta \in \mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma$. (Recall that we do not need to reason about $\tau_{\tau_b,\tau_S}(\phi_\sigma)$ when $\perp \in \mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma$.)

Consider the indexed variants $\tau^{(n)}_{\tau_b,\tau_S}(\phi_\sigma)$ described by:

**input** $\phi_\sigma$
  **repeat**:
    **if** n = 0 **then diverge else** n := n − 1
    $\phi_b \coloneqq \tau_b(\phi_\sigma)$
    **iterate** $i := 0, 1, 2, \cdots$ **until** $\phi_b(i) = -1$ **or** $\phi_b(i) = 1$
    **if** $\phi_b(i) = 1$:
      $\phi_\sigma \coloneqq \tau_S(\phi_\sigma)$
    **if** $\phi_b(i) = -1$:
      **return** $\phi_\sigma$

which is almost identical to $\tau_{\tau_b,\tau_S}(\phi_\sigma)$ except that $\tau^{(n)}_{\tau_b,\tau_S}(\phi_\sigma)$ repeats the main iteration only at most $n$ times and diverges unless it returns until then. By induction on $n$, we can easily verify that for each natural number $n$, $\tau^{(n)}_{\tau_b,\tau_S}$ realizes $\mathcal{W}^{(n)}_{b,S}$.

Furthermore, it holds that as long as $\tau^{(n)}_{\tau_b,\tau_S}(\phi_\sigma)$ is defined, it coincides with $\tau_{\tau_b,\tau_S}(\phi_\sigma)$. Since $\perp \notin \mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma$, there exists $n$ where $\mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma = \mathcal{W}^{(n)}_{b,S}\sigma \not\ni \perp$ and $\tau^{(n)}_{\tau_b,\tau_S}(\phi_\sigma)$ is a name of some $\delta \in \mathcal{W}^{(n)}_{b,S}\sigma$. Therefore, by the above claim, $\tau_{\tau_b,\tau_S}(\phi_\sigma)$ coincides with $\tau^{(n)}_{\tau_b,\tau_S}(\phi_\sigma)$ which is a name of $\delta \in \mathbf{lfp}\big(\mathbf{W}(b, S)\big)\sigma$. □

In the lemma, we say "computable uniformly to $b$ and $c$" to denote that the realizer can be defined by function calls to the realizers of $b$ and $c$ without depending on any further specifics of (the realizers of) $b$ and $c$. We put a justification for this notion of uniformity. Type-2 computability provides a utm theorem stating that there exists an indexing of type-2 machines by $\mathbb{Z}^{\mathbb{N}}$ and a universal type-2 machine for that indexing. Hence, the above uniformity means that the realizer can be actually defined as a multivariate function that receives the indices for the realizers of $b$ and $c$ as its input where the function calls are replaced with compositions by the universal machine. In other words, the least fixed point of $\mathbf{W}(b, c)$ is computable uniformly in the following sense that the operation

$$\text{indices of the realizers of } b, c \text{ and a name of } \sigma \mapsto \text{a name of } \delta \in \mathbf{lfp}\big(\mathbf{W}(b, c)\big)\sigma$$

is computable for any $b, c, \sigma$. We can also make this notion formal by representing computable partial multi-valued functions and making the above operation a computable mapping from them using the same type-2 machine. However, since the cardinality of the set of computable partial multi-valued functions exceeds that of $\mathbb{Z}^{\mathbb{N}}$, it requires us to use a more general notion of representation, namely, multi-representation [Sch02] or assemblies over Kleene's second algebra.

Following up on Lemma 4.3 and Proposition 4.5, we establish the computability result for our command language:

**Lemma 4.6.** *The denotation of any well-formed command is a computable partial multi-valued function.*

*Proof.* The restriction operations are computable. Due to the computability of the primitives, compositions, and fixed points of loops, the denotations of well-formed commands are computable partial multi-valued functions. □

## 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 4.7.** The denotation of a well-typed integer program

$$\begin{aligned} P_{\mathsf{Z}} \coloneqq \quad & \mathbf{input}\ x_1 : \tau_1, x_2 : \tau_2, \cdots, x_d : \tau_d \\ & \quad S \\ & \mathbf{return}\ m \end{aligned}$$

is a partial integer multi-valued function

$$[\![P_{\mathsf{Z}}]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_d]\!] \to \mathbb{P}(\mathbb{Z}_\perp)$$

defined by $[\![P_{\mathsf{Z}}]\!] \coloneqq [\![m]\!]^\dagger \circ [\![S]\!]$.

The denotation of a well-typed real program

$$\begin{aligned} P_{\mathsf{R}} \coloneqq \quad & \mathbf{input}\ x_1 : \tau_1, x_2 : \tau_2, \cdots, x_d : \tau_d \\ & \quad S \\ & \mathbf{return}\ u\ \mathbf{as}\ p \to -\infty \end{aligned}$$

is a partial real function $[\![P_{\mathsf{R}}]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_d]\!] \to \mathbb{R}_\perp$ that is defined by

$$[\![P_{\mathsf{R}}]\!](x_1, \cdots, x_d) \coloneqq \begin{cases} x & \text{if } \forall p \in \mathbb{Z}.\ \forall y \in [\![t]\!]^\dagger \circ [\![S]\!](p, x_1, \cdots, x_d).\ y \neq \perp \wedge |y - x| \leq 2^p, \\ \perp & \text{otherwise.} \end{cases}$$

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

**Theorem 4.8.** *The denotation of a well-typed real* ERC *program is a computable partial real function. The denotation of a well-typed integer* ERC *program is a computable partial integer multi-valued function.*

*Proof.* The computability of terms, commands, compositions, and limit operations [Wei00, Theorem 4.3.7] yields the computability of the denotations of programs. □

When a real program on its input does not generate a converging sequence as its precision parameter $p$ heads to $-\infty$, its function value is denoted by $\perp$. This is the main motivation for us to make $\perp$ to represent not only divergence or non-termination but more general non-specified behaviour. When a sequence is not converging, operationally, there is no possible way to decide that the sequence is not converging. Hence, after reading some finite portion of the sequence, believing that the rest of the sequence converges, an approximation to the limit has to be made. That means even when we realize later that the sequence does not follow our expectations and decides not to converge, there can be an output printed already as the limit of the sequence which turns out to be completely meaningless as the sequence does not converge after all. Without giving this freedom of a procedure returning completely meaningless results on its non-valid inputs, limit operations cannot be realized.

Recall from Fact 2.3 that in computable analysis, a machine realizing a (multi-valued) function can be seen to realize some other (multi-valued) functions at the same time. Reflecting this property, we add one more layer to the denotations of programs:

**Definition 4.9.**

(1) A well-typed real program $P$ *realizes* a partial function $f : X_1 \times \cdots \times X_d \to \mathbb{R}_\perp$ 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 $x = \perp$ or $x = y$ for $x, y \in \mathbb{R}_\perp$.

(2) A well-typed integer program $P$ *realizes* a partial (possibly infinite) multi-valued function $f : X_1 \times \cdots \times X_d \to \mathcal{P}(\mathbb{Z}_\perp)$ if $\forall(x_1, \cdots, x_d).\ f(x_1, \cdots, x_d) \sqsubseteq [\![P]\!](x_1, \cdots, x_d)$ 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-valued functions, they can still realize *infinite* multi-valued functions. Directly from Fact 2.4, we get the computability of realizable partial (multi-valued) functions:

**Corollary 4.10.** *Realizable partial (multi-valued) functions are computable partial (multi-valued) functions.*

## 5. Programming in *Exact Real Computation*

In this section, we collect several examples of (multi-valued) functions that are realizable in ERC. They illustrate the purpose of ERC which is to allow and justify naïvely 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 $\mathsf{choose}$.

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. Heron's method, as an intuitive and efficient way to compute square roots of real numbers, is often expressed in other frameworks [Mül01, BMR18, KST20]. Through the example, the readers can easily compare ERC with the others; similarly, for Gauss elimination [TZ04, Section 2.2]. The example of realizing the exponential function shows that the base language $\mathrm{ERC}_0$ can realize transcendental functions; this leads to a more detailed discussion of our specification language later in Section 6. Integer rounding, on the other hand, shows that $\mathrm{ERC}_0$ can nondeterministically extract names of real numbers. This later forms a core routine in the proof of Turing-completeness in Section 5.3. Root finding, a computational version of the intermediate value theorem is often picked as an example in computable or constructive analysis. The example further demonstrates a trick of using the extension sets $\mathcal{F}$ and $\mathcal{G}$ to model function calls to arbitrary external functions. This example is chosen to be formally verified later in Section 7.

The examples have been implemented using a shallow embedding of ERC into Haskell which we develop on top of the AERN library for exact real number computation [KTD+13].[3] There, though internally real numbers and Kleeneans are represented by infinite sequences, they are hidden from the users where the users see them as abstract entities. Hence, without a nondeterminism monad, it provides the nondeterministic choose operator which can choose different indices for the same inputs whose internal sequences are different. In other words, the choose operator is deterministic about the internal sequences of the arguments which are inspected with some fixed interleaving procedure, but is nondeterministic when the internal sequences are abstracted away.

5.1. **Programming Abbreviations.** Several abbreviations are used in our example programs. While deferring their introductions to each example program, here we make some remarks for those used importantly throughout this paper.

5.1.1. *Extension by denotations.* Recall that the programming language $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$ is defined relative to the sets $\mathcal{F}$ and $\mathcal{G}$. 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 ERC with a feature of calling programs within programs, which removes ERC's strict barrier between its term language and command language, 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 $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$ with $\mathcal{F}$ or $\mathcal{G}$ extended with the denotations of the programs.

When we want to extend the set $\mathcal{F}$ with the denotation of a real program $P$ in $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$, we write $\mathrm{ERC}(\mathcal{F}\cup\{P\}, \mathcal{G})$ to denote $\mathrm{ERC}(\mathcal{F}\cup\{\llbracket P\rrbracket\}, \mathcal{G})$. Similarly, we write $\mathrm{ERC}(\mathcal{F}, \mathcal{G}\cup\{P\})$ when $P$ is an integer program in $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$. These notations are well-defined since the denotation of a real program is a computable partial real function and the denotation of an integer program is a computable partial integer multi-valued function. When the sets are extended by the denotation of a program $P$, we write $P(t_1, \cdots, t_d)$ for $\llbracket P\rrbracket\,(t_1, \cdots, t_d)$ in the extended term language.

[3]The implementation can be found in `https://github.com/michalkonecny/aern2/blob/master/aern2-erc/src/ERC/Examples.hs` .

5.1.2. *Variable scope.* ERC allows local variable creations inside conditional and loop statements. This mechanism for local variables can be used to define arbitrary variable scopes. For a well-formed command $\Gamma \vdash S \triangleright \Gamma'$, write

$$\{S\}$$

for

$$\Gamma \vdash \textbf{if } \textit{true} \textbf{ then } S \textbf{ else skip} \triangleright \Gamma$$

We can easily check:

$$\restriction^{\ddagger\dagger}_{\Gamma} \circ \llbracket \Gamma \vdash S \triangleright \Gamma' \rrbracket = \llbracket \Gamma \vdash \{S\} \triangleright \Gamma \rrbracket \,.$$

5.1.3. *Term Abbreviations.* We assume obvious term abbreviations such as $t_1/t_2$ for $t_1 \times t_2^{-1}$, $t_1 - t_2$ for $t_1 + (-t_2)$, and so on. For each integer constant $k$, we write $k$ to refer to the $|k|$ times repeated additions of either $2^0$ or $-2^0$ when we want it to represent the real number $k$.

5.1.4. *Command Abbreviations.* Though our default term language is restrictive without having integer multiplications or the ordinary integer-to-real coercion, the default command language can recover most of the restrictions. For instance, given two integer terms $m_1$ and $m_2$, though we cannot have its multiplication as a term, we can have a command that computes the multiplication and assigns the result into a variable say $y : \mathsf{Z}$:

$$
\begin{array}{l}
\{ \\
\quad \textbf{let } x_1 : \mathsf{Z} = m_1;\ \textbf{let } x_2 : \mathsf{Z} = m_2;\ \textbf{let } k : \mathsf{Z} = 0; \\
\quad y \coloneqq 0; \\
\quad \textbf{if } \ 0 \le x_2 \\
\quad \textbf{then } \ \textbf{while } k + 1 \le x_2 \ \textbf{ do } \ y \coloneqq y + x_1;\ k \coloneqq k + 1 \ \textbf{ end} \\
\quad \textbf{else } \ \ \textbf{while } x_2 + 1 \le k \ \textbf{ do } \ y \coloneqq y - x_1;\ k \coloneqq k - 1 \ \textbf{ end} \\
\quad \textbf{end} \\
\}
\end{array}
$$

The local variables $x_1$ and $x_2$ are used to choose values from possibly multi-valued $m_1$ and $m_2$ and to use them consistently throughout the computation. For a context $\Gamma$ such that $\Gamma \vdash m_1 : \mathsf{Z}$, $\Gamma \vdash m_2 : \mathsf{Z}$, and $\Gamma(y) = \mathsf{Z}$,

$$\Gamma \vdash y :\equiv m_1 \times m_2 \triangleright \Gamma$$

abbreviates the above well-formed command whose local variables are selected to not conflict with $\Gamma$. The notation $:\equiv$ is chosen to make it be distinguished from the usual assignment construct $\coloneqq$ to emphasize that $m_1 \times m_2$ is not a term. The following semantic equation can be verified:

$$\llbracket y :\equiv t_1 \times t_2 \rrbracket \, \sigma = \bigcup_{x_1 \in \llbracket t_1 \rrbracket \sigma \wedge x_2 \in \llbracket t_2 \rrbracket \sigma} \begin{cases} \{\bot\} & \text{if } x_1 = \bot \vee x_2 = \bot, \\ \{\sigma[y \mapsto x_1 \times x_2]\} & \text{otherwise.} \end{cases}$$

We can define various command abbreviations similarly for example

$$\Gamma \vdash y :\equiv u^m \triangleright \Gamma$$

when $\Gamma \vdash u : \mathsf{R}$, $\Gamma \vdash m : \mathsf{Z}$, and $\Gamma(y) = \mathsf{R}$ for the repeated multiplications satisfying

$$[\![ y :\equiv u^m ]\!]\,\sigma = \bigcup_{x_1 \in [\![u]\!]\sigma \wedge x_2 \in [\![m]\!]\sigma} \begin{cases} \{\bot\} & \text{if } x_1 = \bot \vee x_2 = \bot, \\ \{\sigma[y \mapsto x_1^{x_2}]\} & \text{otherwise.} \end{cases}$$

5.2. **Example Programs.** We annotate assertions next to commands to convey the behaviours, including the domains (preconditions), of the introduced programs. However, we remind the readers 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 $0 \le x$ up to any desired absolute error $2^p$, $p \in \mathbb{Z}$, by calculating a contracting sequence of upper and lower approximations $z_n := x/y_n \le \sqrt{x} \le y_n$ iteratively taking the average $y_{n+1} := (y_n + z_n)/2$. The $\mathrm{ERC}_0$ program `HeronSqrt` illustrated in Algorithm 1 realizes the square root function via Heron's method.

**Algorithm 1** `HeronSqrt` : $\mathsf{R} \to \mathsf{R}$

| | |
|---|---|
| **input** $x : \mathsf{R}$ | $//\ 0 \le x$ |
|  **let** $y : \mathsf{R} = 1$; **let** $z : \mathsf{R} = x/y$; | $//\ z \le \sqrt{x} \le y$ |
|  **while** $\mathsf{choose}\big(y - z \lessdot 2^p\ ,\ 2^{p-1} \lessdot y - z\big) = 1$ **do** | |
|   $y := (y + z)/2; \quad z := x/y$ | $//\ z \le \sqrt{x} \le y$ |
|  **end** | |
| **return** $y$ **as** $p \to -\infty$ | $//\ \lvert y - \sqrt{x}\rvert < 2^p$ |

Recall that the arguments to $\mathsf{choose}$ are indexed from 0, hence $\mathsf{choose}(b_0, b_1) = 1$ means that the second argument $b_1$ must evaluate to *true*. The ERC program employs the multi-valued $\mathsf{choose}()$ operation applied to two Kleenean-valued tests $y - z \lessdot 2^p$ and $2^{p-1} \lessdot y - z$ where at least one of the two must be *true* resulting in a total loop condition. Moreover, when the loop terminates, the first test (corresponding to return value 0) must (while the second test, corresponding to return value 1, 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

$$\exp(x) \;=\; \sum\nolimits_{j=0}^{\infty} x^j/j! \tag{5.1}$$

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 $|x| \le 1$ and positive $n \in \mathbb{N}$, the tail bound

$$\big|\sum\nolimits_{j>n} x^j/j!\big| \;\le\; \sum\nolimits_{j>n} |x|^j/j! \;\le\; \sum\nolimits_{j>n} 2^{-j+1} \;=\; 2^{-n+1} \tag{5.2}$$

justifies the straightforward algorithm as in the ERC program `pExp` in Algorithm 2. The program introduces a command abbreviation **let** $x_1, \cdots, x_d : \tau = t$ for **let** $x_1 : \tau =$

**Algorithm 2** $\texttt{pExp} : \mathsf{R} \to \mathsf{R}$

```
input x : R                                        // −1 ≤ x ≤ 1
    let j : Z = 1;  let j_r, 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
```

**Algorithm 3** $\texttt{posExp} : \mathsf{R} \to \mathsf{R}$

```
input x : R                                        // −1 ≤ x
    let z : R = pExp(1/2);
    let y : R = 1;
    while choose(x ⋖ 1, 1/2 ⋖ x) = 1 do
        y := y × z;
        x := x − 1/2
    end
return y × pExp(x) as p → −∞
```

$t; \cdots ;$ **let** $x_d : \tau = t$. In the program, the two variables $j : \mathsf{Z}$ and $j_r : \mathsf{R}$ retain identical values but have distinguished types, written as $j \equiv j_r$ in the assertions.

The following wrapper $\texttt{posExp}$ for $\texttt{pExp}$ in Algorithm 3, which is a program in $\mathrm{ERC}(\{\texttt{pExp}\}, \emptyset)$ removes the restriction $|x| \leq 1$ and instead works for all $-1 \leq x$. Furthermore, 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 $x \in \mathbb{R}$ using the continuous conditional by

$$0 \lessdot x \ ? \ \texttt{posExp}(x) : 1/\texttt{posExp}(-x).$$

5.2.3. *Exponential Function via Iteration.* $\texttt{pExp}$ employs unbounded sums, which leave the realm of the first-order language 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]:

$$\exp(x) \xleftarrow{n\to\infty} \left(1 + \tfrac{x}{n}\right)^n \ \leq \ \exp(x) \ \leq \ \left(1 + \tfrac{x}{n}\right)^{n+1} \xrightarrow{n\to\infty} \exp(x)$$

This suggests the iterative ERC program $\texttt{iExp}$ in Algorithm 4. It supposes $0 \leq x \leq 2$ but can be extended to the entire real line similar to Section 5.2.2. Note that the loop is guaranteed to terminate since $b - a$ converges to 0 as $n$ grows. The loop condition is

**Algorithm 4** $\texttt{iExp} : \mathsf{R} \to \mathsf{R}$

**input** $x : \mathsf{R}$ $\qquad // \ 0 \le x \le 2$
  **let** $n : \mathsf{Z} = 1$;
  **let** $n_r : \mathsf{R} = 1$;
  **let** $c : \mathsf{R} = 1 + x$; **let** $a : \mathsf{R} = c$; **let** $b : \mathsf{R} = a \times c$;
  **while** $\mathsf{choose}\big(b - a \lessdot 2^p, 2^{p-1} \lessdot b - a\big) = 1$ **do**
    $n \coloneqq n + n$;
    $n_r \coloneqq 2 \times n_r$;
    $c \coloneqq 1 + x/n_r$; $a :\equiv c^n$; $b \coloneqq a \times c$
  **end**
**return** $a$ **as** $p \to -\infty$

total: At least one of $b - a \lessdot 2^p$ and $2^{p-1} \lessdot b - a$ must be *true*. Furthermore, when the loop terminates, it must hold $b - a < 2^p$, hence $|a - \exp(x)| \le |b - a| \le 2^p$.

5.2.4. *Integer Rounding Multi-valued 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-valued function with *overlap*:

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

The ERC program $\texttt{Round}$ in Algorithm 5 realizes Round. It returns, given a real $x$, multi-valued integer $k$ such that $|x - k| < 1$.

**Algorithm 5** $\texttt{Round} : \mathsf{R} \to \mathsf{Z}$

**input** $x : \mathsf{R}$
  **let** $k : \mathsf{Z} = 0$;
  **while** $\mathsf{choose}(x \lessdot 1, 1/2 \lessdot x) = 1$ **do**
    $k \coloneqq k + 1$; $x \coloneqq x - 1$
  **end**;
  **while** $\mathsf{choose}(-1 \lessdot x, x \lessdot -1/2) = 1$ **do**
    $k \coloneqq k - 1$; $x \coloneqq x + 1$
  **end**
**return** $k$

Intuitively, the number of steps made by $\texttt{Round}$ is proportional to the value of the argument $x$, that is, exponential in its binary ($\approx$output) length because $\texttt{Round}$ essentially counts up to $x$.

Recovering the rounded integer bit-wise 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 Algorithm 6 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. There, the absolute value $|y|$ is an abbreviation for $0 \lessdot y \;?\; y :\; -y$.

**Algorithm 6** `BinRound` : R → Z

```
1:  input x : R
2:      let k, j, b : Z = 0;  let y : R = x;
3:      while choose(|y| ⋖ 1, 1/2 ⋖ |y|) = 1 do                      // x = y · 2^j
4:          j := j + 1;  y := y/2
5:      end;                                                          // |y| ⋖ 1
6:      while 1 ≤ j do                                  // x = (y + k) · 2^j ∧ |y| < 1
7:          y := y × 2;
8:          b := choose(y ⋖ 0, −1 ⋖ y ⋖ 1, 0 ⋖ y) − 1;
                                        // b =most-significant signed binary digit of y
9:          if  b = −1  then  y := y + 1  end; if  b = 1  then  y := y − 1 end;
10:         k := k + k + b;  j := j − 1
11:     end                                                           // |x − k| < 1
12: return k
```

Due to the multi-valuedness of the loop condition at Line 3, when the loop (Lines 3 to 5) exits, the second argument $1/2 \lessdot y$ may still be *true*, whereas the first $|y| \lessdot 1$ must be *true*. In the integer loop (Lines 6 to 11), multi-valuedness strikes only at Line 8 which employs $\mathsf{choose}()$ with trinary argument.

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

$$\det(A) \;=\; \sum\nolimits_{\pi} \operatorname{sign}(\pi) \cdot \prod\nolimits_{j=1}^{d} a_{j,\pi(j)} \tag{5.4}$$

where the sum ranges over all $d!$ permutations $\pi : \{1, \ldots, d\} \to \{1, \ldots, d\}$. Since ERC conveniently relieves the programmer from numerical issues like cancellation, this formula gives rise to a straightforward ERC program. However one executing 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, §2.3.3]. Following Turing [Tur48], apply Gaussian Elimination 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. 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.

We thus change the specification of the determinant (and of the LUPQ factorization[4] it builds on): with the promise for the argument matrix $A$ to have full rank—otherwise, its determinant will vanish, anyway.

$$\mathrm{Det}_d : \mathrm{GL}(\mathbb{R}^d) \ni A \mapsto \begin{cases} \det(A) & \text{if } \det(A) \neq 0, \\ \perp & \text{otherwise.} \end{cases} \tag{5.5}$$

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

$$\mathrm{Pivot}_d(A, k) \;=\; \big\{(i,j) \;:\; A \in \mathbb{R}^{d\times d} \wedge k \le i,j < d \wedge A_{i,j} \neq 0\big\}_{\{\perp\}} \;\in\; \mathbb{P}(\mathbb{Z}_\perp^2) \tag{5.6}$$

Here, $X_{\{\perp\}} \coloneqq X$ 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 $A[k \ldots d-1, k \ldots d-1] \coloneqq \big(A[i,j]\big)_{k \le i,j<d}$.

**Algorithm 7** $\texttt{Pivot}_d : \mathsf{R}[d \times d] \times \mathsf{Z} \to \mathsf{Z}$

$\quad$ **input** $A : \mathsf{R}[d \times d],\ k : \mathsf{Z}$
$\quad\quad$ **let** $i_0,\ j_0 : \mathsf{Z} = k$; **let** $x : \mathsf{R} = 0$;
$\quad\quad$ **for** $i : \mathsf{Z} = k$ **to** $d-1$ **do**
$\quad\quad\quad$ **for** $j : \mathsf{Z} = k$ **to** $d-1$ **do**
$\quad\quad\quad\quad$ $x \coloneqq \max\big(x, \mathrm{abs}(A[i,j])\big)$ **end end**;
$\quad\quad$ **for** $i : \mathsf{Z} = k$ **to** $d-1$ **do**
$\quad\quad\quad$ **for** $j : \mathsf{Z} = k$ **to** $d-1$ **do**
$\quad\quad\quad\quad$ **if** $\mathsf{choose}(\mathrm{abs}(A[i,j]) \lessdot x, x/2 \lessdot \mathrm{abs}(A[i,j])) = 1$ **then** $i_0 \coloneqq i; j_0 \coloneqq j$ **end**
$\quad\quad$ **end end**
$\quad$ **return** $(i_0, j_0)$

In the program $\texttt{Pivot}_d$ in Algorithm 7, we represent matrices with fixed dimensions using ERC's one-dimensional arrays. For each positive natural numbers $n$ and $m$, we write $A[n \times m]$, to express a $n \times m$-dimensional matrix, simulated by $A[k]$ where $k = n \times m$. Accessing an entry $A[i,j]_n$ is then a shorthand for $A[i + j + \cdots + j]$ denoting $A[i + n \cdot j]$. The repeated addition is only possible as $n$ is a fixed natural number constant, not a ERC term while $i, j$ are. The subscript $n$ is omitted for better readability when it is obvious. Similarly the pair $(i_0, j_0)$ of return values is thus understood as an abbreviation of the single integer $i_0 + d \cdot j_0$ where $d$ is the dimension of the input matrix.

The program introduces more abbreviations. For any integer terms $k, l$ and a command $S$, **for** $n : \mathsf{Z} = k$ **to** $l$ **do** $S$ **end** refers to

$$\textbf{let } n : \mathsf{Z} = k;\ \textbf{let } m : \mathsf{Z} = l;\ \textbf{while } (n \le m) \textbf{ do } S;\ n \coloneqq n + 1 \textbf{ end}$$

with $n, m$ not appearing in $S$. For a Kleenean typed term $b$ and a command $S$,

$$\textbf{if } b \textbf{ then } S \textbf{ end}$$

[4]LUPQ factorization $A \mapsto (L, U, P, Q)$ is not unique, hence a real matrix-tuple-valued *multi*-function.

is used as an abbreviation for

$$\textbf{if } b \textbf{ then } S \textbf{ else skip end}$$

Moreover, $\max(u, v)$ is a term abbreviation for $u \lessdot v \; ? \; v : u$.

Based on $\texttt{Pivot}_d$, the $\mathrm{ERC}(\emptyset, \{\texttt{Pivot}_d\})$ program `Det` in Algorithm 8 computes the non-zero determinant $\mathrm{Det}_d$ from Equation (5.5) via LUP decomposition with full pivoting. Note that the precision parameter $p \in \mathbb{Z}$ is present but ignored since the result gets computed exactly.

**Algorithm 8** $\texttt{Det} : \mathsf{R}[d \times d] \to \mathsf{R}$

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

Note that pivot search in Line 6 of program `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 $A \in \mathrm{GL}(\mathbb{R}^d)$.

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. We consider an algorithmic version of the Intermediate Value Theorem which is to find the unique root to a given continuous $f : [a; b] \to \mathbb{R}$ satisfying $f(a) < 0 < f(b)$. This case is commonly treated using Bisection: Determine the sign of $f(x)$ at the interval midpoint $x \coloneqq (a+b)/2$ and recurse to either $[a;x]$ or to $[x;b]$ accordingly. However, since equality is undecidable, the sign test fails in case $f(x)=0$. *Tri*section [Her96, p. 336] instead considers the signs of $f$ at both one third $x' \coloneqq (2a+b)/3$ and at two third $x'' \coloneqq (a+2b)/3$ 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 $\mathrm{Root} : f \mapsto x$ with $f(x)=0$ for $f$ satisfying the following first-order predicates:

$$\begin{aligned}
\mathrm{cont}(f,a,b) \quad &:\equiv \\
&\forall \epsilon > 0. \quad \exists \delta > 0. \quad \forall x, x'.\ a \le x \le x' \le x+\delta \le b \ \Rightarrow\ |f(x) - f(x')| \le \epsilon \\
\mathrm{uniq}(f,a,b) \quad &:\equiv \quad \mathrm{cont}(f,a,b) \wedge f(a)\cdot f(b) < 0 \wedge \exists! x.\ a < x < b \wedge f(x) = 0
\end{aligned}$$

The first condition $\mathrm{cont}(f,a,b)$ says $f$ is continuous in the usual sense on the interval $[a;b]$ and the second condition $\mathrm{uniq}(f,a,b)$ 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 `Trisection` in Algorithm 9 is annotated with precondition $\mathrm{uniq}(f,a,b)$ and postcondition $\mathrm{uniq}(f,a,b) \wedge |b-a| \le 2^p$, as well as loop invariant $\mathrm{uniq}(f,a,b)$. Formally, as ERC does not accept function arguments, the program is defined in $\mathrm{ERC}(\{f\},\emptyset)$. Given $p$, the postcondition guarantees that `Trisection` indeed returns an approximation to the root up to error $2^p$ when $\mathrm{uniq}(f,a,b)$ holds. Therefore, as $p \to -\infty$ the program returns the root of $f$. This program is chosen later in Section 7 to be formally proved for its correctness.

**Algorithm 9** `Trisection` : $\mathsf{R} \times \mathsf{R} \to \mathsf{R}$ in $\mathrm{ERC}(\{f\},\emptyset)$ for any $f : \mathbb{R} \to \mathbb{R}_\perp$

**input** $a : \mathsf{R}, b : \mathsf{R}$ // $\mathrm{uniq}(f,a,b)$
    **while** $\mathsf{choose}\big(b - a \lessdot 2^p,\ 2^{p-1} \lessdot b - a\big) = 1$ **do** // $\mathrm{uniq}(f,a,b) \wedge b - a > 2^{p-1}$
        **if** $\mathsf{choose}(f(b/3 + 2 \times a/3) \times f(b) \lessdot 0,\ f(a) \times f(2 \times b/3 + a/3) \lessdot 0) \ = \ 1$
        **then** $b \coloneqq 2 \times b/3 + a/3$
        **else** $a \coloneqq b/3 + 2 \times a/3$ **end**
    **end** // $\mathrm{uniq}(f,a,b) \wedge |b-a| \le 2^p$
**return** $a$ **as** $p \to -\infty$

5.3. **Turing-Completeness.** Algorithms 5 and 6 show the existence of a well-formed command which we abbreviate as

$$\Gamma \vdash z :\equiv \mathrm{round}(x) \triangleright \Gamma$$

when $\Gamma(x) = \mathsf{R}$ and $\Gamma(z) = \mathsf{Z}$ that assigns the multi-valued rounding of $x$ into $z$ having its denotation

$$[\![\Gamma \vdash z :\equiv \mathrm{round}(x) \triangleright \Gamma]\!]\,\sigma = \{\sigma[z \mapsto k] \mid \sigma(x) - 1 < k < \sigma(x) - 1\}.$$

Furthermore, though it is not available in the term language, the command language of ERC provides the naïve coercion from integers to reals: There exists a well-formed command which we write

$$\Gamma \vdash x :\equiv \mathrm{real}(z) \triangleright \Gamma$$

when $\Gamma(x) = \mathsf{R}$ and $\Gamma(z) = \mathsf{Z}$ that repeatedly add 1 or $-1$ in a loop such that

$$[\![\Gamma \vdash x :\equiv \mathrm{real}(z) \triangleright \Gamma]\!]\,\sigma = \{\sigma[x \mapsto \sigma(z)]\}.$$

ERC being an imperative language providing integers, arithmetical operations on them, compositions of commands, and general loops, it is type-1 Turing-complete over natural numbers:

**Proposition 5.1.** *For any type-1 computable (or equivalently $\mu$-recursive) $f : \mathbb{N}^d \rightharpoonup \mathbb{N}$, there exists a well-formed* ERC *command which we abbreviate as*

$$\Gamma \vdash y :\equiv f(x_1, \cdots, x_d) \triangleright \Gamma$$

*when $\Gamma(y) = \mathsf{Z}, \Gamma(x_i) = \mathsf{Z}$ for each $i$, such that for all $\sigma \in [\![\Gamma]\!]$ where $(\sigma(x_1), \cdots, \sigma(x_d)) \in \mathrm{dom}(f)$,*

$$[\![\Gamma \vdash y :\equiv f(x_1, \cdots, x_d) \triangleright \Gamma]\!]\,\sigma = \{\sigma[y \mapsto f(\sigma(x_1), \cdots, \sigma(x_d))]\}.$$

*Proof.* We can construct the command $S$ inductively on the $\mu$-recursiveness of $f$. □

From the fact that ERC is type-1 Turing-complete, we can automatically ensure many interesting operations on integers where some of which are necessary to build our (type-2) Turing-completeness result. Fix a standard numbering of finite sequences (including the empty sequence $\epsilon$) of integers $\zeta : \mathbb{N} \to \mathbb{Z}^*$ throughout this section. By the type-1 completeness, there are ERC commands for (1) constructing the code of the empty string $\epsilon$, (2) appending a new tail to a sequence from its code, (3) accessing the tail of a sequence from its code assuming that the sequence is not empty, and (4) obtaining the length of a sequence from its code. For any integer variables $y, x, x_1, x_2$ in $\Gamma$:

$$[\![\Gamma \vdash y :\equiv \epsilon \triangleright \Gamma]\!]\,\sigma = \{\sigma[y \mapsto \zeta^{-1}(\epsilon)]\}$$
$$[\![\Gamma \vdash y :\equiv x_1\#x_2 \triangleright \Gamma]\!]\,\sigma = \{\sigma[y \mapsto \zeta^{-1}(\zeta(\sigma(x_1))\#\sigma(x_2))]\}$$
$$[\![\Gamma \vdash y :\equiv \mathrm{tail}(x) \triangleright \Gamma]\!]\,\sigma = \{\sigma[y \mapsto m]\} \text{ if } \zeta(\sigma(x)) = a\#m \text{ for some } a \in \mathbb{Z}^*$$
$$[\![\Gamma \vdash y :\equiv \mathrm{length}(x) \triangleright \Gamma]\!]\,\sigma = \{\sigma[y \mapsto m]\} \text{ if } \zeta(\sigma(x)) \text{ has length } m$$

Note that the command for accessing the tail of a sequence $y :\equiv \mathrm{tail}(x)$ may not well-behave when the sequence that $x$ represents is the empty sequence. Here, $x\#y$ when $x \in \mathbb{Z}^*$ and $y \in \mathbb{Z}$ denotes the finite sequence of integers where $y$ is appended to $x$.

For the last building block of our main theorem, we recall the type-1 characterization of type-2 machines:

**Proposition 5.2.** *For any type-2 machine computing $\tau : \mathbb{Z}^\mathbb{N} \times \cdots \times \mathbb{Z}^\mathbb{N} \rightharpoonup \mathbb{Z}^\mathbb{N}$, there exists a total type-1 computable (with regard to the standard numbering of finite sequences)*

$\bar{\tau} : \mathbb{Z}^* \times \cdots \times \mathbb{Z}^* \to \mathbb{Z}^*$ *such that for any* $(\phi_1, \cdots, \phi_d) \in \mathrm{dom}(\tau)$*,*

$$(\bar{\tau}(\bar{\phi}_1^{(m)}, \cdots, \bar{\phi}_d^{(m)}))_{m \in \mathbb{N}}$$

*(which can contain the empty sequence) is a chain with regard to the prefix ordering and it converges to* $\tau(\phi_1, \cdots, \phi_d)$*. Here,* $\bar{\phi}^{(m)} \in \mathbb{Z}^*$ *denotes the length* $m$ *prefix of* $\phi \in \mathbb{Z}^{\mathbb{N}}$*.*

The type-1 $\bar{\tau}$ continuously approximates $\tau$ in the following sense. If $\phi \in \mathrm{dom}(\tau)$, $\bar{\tau}$ tries to approximate $\tau(\phi)$ using finite approximations of $\phi$. The function $\bar{\tau}$ may fail initially due to for example lack of input precision and return the empty sequence. However, it eventually returns longer and longer prefixes of $\tau(\phi)$ converging to $\tau(\phi)$ consistently (i.e. monotonically with regard to the prefix ordering).

*Proof.* Let us first consider the case where $\tau$ is univariate. As stated in [Wei00, Excercise 2.3.12], due to the relation between domain-computability and type-2 computability, for any partial $\tau$ computed by a type-2 machine, there exists a total function $T : \mathbb{Z}^* \to \mathbb{Z}^*$ which is monotone with regard to the prefix ordering, approximates $\tau$ in the sense that

$$\lim_{n \to \infty} T(\bar{\phi}^{(n)}) = \tau(\phi)$$

for each $\phi \in \mathrm{dom}(\tau)$, and the set

$$\{(u, v) \in \mathbb{Z}^* \times \mathbb{Z}^* \mid v \text{ is a prefix of } T(u)\}$$

is computably enumerable. Let $e : \mathbb{N} \to \mathbb{Z}^* \times \mathbb{Z}^*$ be one computable enumeration. Then, we can define $\bar{\tau}$ as follows: given $\bar{\phi}^{(n)}$, iterate $e(0), \cdots, e(n)$ to find $(u, v) \coloneqq e(i)$ where $u$ is a prefix of $\bar{\phi}^{(n)}$ and the length of $v$ is the longest amongst such. If such $(u, v)$ is located, return $v$. If there is no prefix of $\bar{\phi}^{(n)}$ found, return the empty sequence $\epsilon$. This computational procedure describes the computable function $\bar{\tau}$.

Now consider a computable $d$-variate $\tau$ and construct $\bar{\tau}$. As $\tau$ is computable, there exists a computable $\tau' : \mathbb{Z}^{\mathbb{N}} \rightharpoonup \mathbb{Z}^{\mathbb{N}}$ such that $\tau(\phi_1, \cdots, \phi_d) = \tau' \langle \phi_1, \cdots, \phi_d \rangle$ where $\phi \coloneqq \langle \phi_1, \cdots, \phi_d \rangle$ is defined by interleaving. By the above proof, there exists $\bar{\tau}' : \mathbb{Z}^* \to \mathbb{Z}^*$ for $\tau'$ satisfying the conditions. Given the prefixes $\bar{\phi}_1^{(n)}, \cdots, \bar{\phi}_d^{(n)}$, we construct $\bar{\phi}^{(n \cdot d)}$ by interleaving, feed it in $\tau'$, and let it be the return sequence of $\bar{\tau}(\bar{\phi}_1^{(n)}, \cdots, \bar{\phi}_d^{(n)})$. □

**Theorem 5.3.** *Recall the definitions of computable partial (mutli-valued) functions in Section 2 and* ERC *programs realizing them in Definition 4.9.*

(1) *Every computable partial real function from integers and reals can be realized in* $\mathrm{ERC}_0$*.*
(2) *Similarly, every computable partial integer multi-valued function from integers and reals can be realized in* $\mathrm{ERC}_0$*.*

As stated in Proposition 5.1, $\mathrm{ERC}_0$ can realize all $\mu$-recursive integer functions, and consequently also every computable single real number $r \in \mathbb{R}$, since the latter is defined as limit $r = \lim_n \phi(n)/2^n$ for $|r - \phi(n)/2^n| \leq 2^{-n}$. Note that also every oracle Turing machine computable integer function is oracle-recursive; see `http://math.stackexchange.com/questions/2778974`. This suggests realizing a type-2 oracle machine computing a real function $f : \mathbb{R} \to \mathbb{R}_\perp$ according to [Ko91, §2.3] in $\mathrm{ERC}_0$: by replacing any oracle query

"$\phi(n)$" for a $2^{-n}$–approximation $\phi(n)/2^n$ to the real argument $x \in \mathrm{dom}(f)$ by the $\mathrm{ERC}_0$ commands in Section 5.2.4 realizing $\mathrm{Round}(x \times 2^n)$. However said Round() is multi-valued, corresponding to a multi-valued oracle $\phi$; and for these, oracle Turing computation does not seem to be known as equivalent to oracle $\mu$-recursiveness. Instead, we present in the sequel a different proof of Theorem 5.3:

*Proof.* (1) Without loss of generality, suppose any computable partial $f : \mathbb{R}^{d_1} \times \mathbb{Z}^{d_2} \to \mathbb{R}_\perp$ where $d_1 + d_2 = d$. By definition, there is a type-2 machine computing $\tau : \mathbb{Z}^{\mathbb{N}} \times \cdots \times \mathbb{Z}^{\mathbb{N}} \rightharpoonup \mathbb{Z}^{\mathbb{N}}$ such that for each $(x_1, \cdots, x_d) \in \mathrm{dom}(f)$ and each $\phi_i$ a name of $x_i$, $\tau(\phi_1, \cdots, \phi_d)$ is defined and is a name of $f(x_1, \cdots x_d)$:

$$|f(x_1, \cdots, x_d) - \tau(\phi_1, \cdots, \phi_d)(n) \cdot 2^{-n}| \leq 2^{-n} \quad \text{holds for all } n.$$

By Propositions 5.1 and 5.2, there exists a well-formed command in ERC such that when $x'_i, y'$ are integer variables in $\Gamma$, for any $\sigma \in [\![\Gamma]\!]$ such that $\sigma(x'_i) = m_i^{(n)} \in \mathbb{N}$ a code for $\bar{\phi}_i^{(n)}$ (w.r.t. the fixed standard numbering $\xi$),

$$[\![\Gamma \vdash y' :\equiv \bar{\tau}(x'_1, \cdots, x'_d)]\!]\, \sigma = \{\sigma[y' \mapsto y^{(n)}]\}$$

holds for some $y^{(n)} \in \mathbb{N}$ which is a code for a finite, possibly empty, prefix of $\tau(\phi_1, \cdots, \phi_d)$.

We claim that the real program in $\mathrm{ERC}_0$ in Algorithm 10 realizes $f$. On its input variables $x_1, \cdots, x_d$, the program prepares integer variables $x'_1, \cdots, x'_d$ to let each $x'_i$ stores a code of a finite prefix (of say length $n$) of a name of $x_i$. Starting from the empty sequences ($n = 0$), the program keeps appending $x'_i$ and computes $\bar{\tau}(x'_1, \cdots, x'_d)$ in $y'$ which is a finite prefix (of say length $\ell$) of some name of $f(x_1, \cdots, x_d)$. The program iterates until the length $\ell$ of the obtained prefix $y'$ becomes enough to obtain a $2^{-p}$ approximation to $f(x_1, \cdots, x_d)$.

The variable $n$ increases as the number of iterations. Each state that appears at the beginning of the $n$'th iteration satisfies that

$$x'_i \text{ is (a code for) the length } n \text{ prefix of some name } \phi_i \text{ of } x_i \quad \text{and}$$

$$y' \text{ is (a code for) the length } \ell \text{ prefix of } \tau(\phi_1, \cdots, \phi_d) \text{ for some name } \phi_i \text{ of } x_i$$

irrelevant to the multi-valuedness of the integer rounding. (Note that though due to multi-valuedness, $\phi_i$ in each annotated assertion does not need to be consistent.)

Suppose for now that the loop is total. When the loop exits, due to $\ell - 1 \geq -p$ and $\ell > 0$ the tail of $y'$, which is $y$, is the $\ell - 1$'th (counting from 0) entry of $\tau(\phi_1, \cdots, \phi_d)$ for some names $\phi_i$ of $x_i$. Hence, it holds that $|f(x_1, \cdots, x_d) - y \times 2^{-(\ell-1)}| \leq 2^{-(\ell-1)} \leq 2^p$ where $y \times 2^{-(\ell-1)}$ is the return value. We conclude that the real program indeed realizes $f$.

To prove that the loop is total, we show that for any natural number $k$, there is a natural number $n(k)$ such that for any names $\phi_i$ of $x_i$, $\bar{\tau}(\phi_1^{(n(k))}, \cdots, \phi_d^{(n(k))})$ has length greater than or equal to $k$. If this is true, the loop terminates after at most $n(-p)$ (or $n(1)$ if $p$ is non-negative) iterations.

Recall that $A := \{(\phi_1, \cdots, \phi_d) \mid \phi_i \text{ is a name of } x_i\}$ is compact with regard to the standard topology of Baire space. For each natural number $n$, define $A_n := \{(\phi_1, \cdots, \phi_d) \mid \phi_i \text{ is a name of } x_i \wedge \bar{\tau}(\phi_1^{(n)}, \cdots, \phi_d^{(n)}) \text{ has length greater than or equal to } k\}$ which is the set

**Algorithm 10** A real program in $\text{ERC}_0$ realizing $f$ in the proof of Theorem 5.3

```
input x_1 : R, ⋯ , x_{d_1} : R, x_{d_1+1} : Z, ⋯ , x_d : Z
    let x'_1, ⋯ , x'_d, y', y'' :≡ ε;
    let ℓ, t, n : Z = 0;
    while ℓ ≤ −p ∨̂ ℓ ≤ 0 do
        // x'_i is a code for the length n prefix of some name φ_i of x_i
        // y' is a code for the length ℓ prefix of τ(φ_1, ⋯ , φ_d) for some name φ_i of x_i
        // append x'_i:
        t :≡ round(x_1 × 2^n);
        x'_1 :≡ x'_1#t;
        ⋮
        t :≡ round(x_{d_1} × 2^n);
        x'_{d_1} :≡ x'_{d_1}#t;
        x'_{d_1+1} :≡ x'_{d_1+1}#x_{d_1+1}; ⋯ ; x'_d :≡ x'_d#x_d;
        n := n + 1;
        // x'_i is a code for the length n prefix of some name φ_i of x_i
        y' :≡ τ̄(x'_1, ⋯ , x'_d);
        ℓ :≡ length(y');
        // y' is a code for the length ℓ prefix of τ(φ_1, ⋯ , φ_d) for some names φ_i of x_i
    end;
    // y' is a code for a length ℓ prefix of τ(φ_1, ⋯ , φ_d) for some names φ_i of x_i and
    // ℓ > 0 and ℓ − 1 ≥ −p
    y : R = 0;
    y'' :≡ tail(y');                          // ℓ > 0 ensures tail(y') does not fail
    y :≡ real(y'');                           // |f(x_1, ⋯ , x_d) − y × 2^{−(ℓ−1)}| ≤ 2^{−(ℓ−1)} ≤ 2^p
return y × 2^{−(ℓ−1)} as p → −∞
```

of names whose length $n$ prefix is enough to make $\bar{\tau}$ returns length $k$ prefix of $\tau$. Note that $A_n$ is open and by the convergence of $\bar{\tau}$, $(A_n)_{n\in\mathbb{N}}$ forms a countable open cover of $A$. The compactness gives us a finite subcover of $A_n$ and we can choose the maximum index of the finite subcover to be $n(k)$.
(2) The integer multi-valued function case can be proved similarly except that now we only need to iterate until $\ell > 0$ and return $\mathrm{tail}(y')$. □

**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 with 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-valued functions but not a real multi-valued 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-valued functions is to ease our typing rule such that we let the program

$$\begin{aligned} P \coloneqq\ & \textbf{input } x_1 : \tau_1, x_2 : \tau_2, \cdots, x_d : \tau_d \\ & \quad S \\ & \textbf{return } t \end{aligned}$$

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

$$x_1 : \tau_1, x_2 : \tau_2, \cdots, x_d : \tau_d \vdash S \triangleright \Gamma' \quad \text{and} \quad \Gamma' \vdash t : \mathsf{R}\,.$$

We let its denotation be $[\![P]\!] \coloneqq [\![t]\!]^\dagger \circ [\![S]\!] : [\![\tau_1]\!] \times \cdots \times [\![\tau_d]\!] \to \mathbb{P}(\mathbb{R}_\perp)$ as it was an integer program. From Lemmas 4.3 and 4.6, it can be seen immediately that the denotations of such real programs are computable multi-valued functions. We can also extend the notion of realization in Definition 4.9 accordingly.

However, this extension does not carry the completeness in Theorem 5.3 over. Recall that the denotations of programs are finite multi-valued functions. Hence, a computable infinite multi-valued function is realizable in ERC only if it admits point-wise finite refinement that is computable. In the integer case, it was feasible as any infinite integer multi-valued function admits point-wise finite refinement. (Recall Fact 2.4.) However, this does not work for real multi-valued functions. That means when we extend ERC with real multi-valued 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.

## 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 which we use to rigorously specify (multi-valued) functions with real arguments in Section 6.2. It is carefully designed to be 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.

**Definition 6.1.** The *Structure of* $\mathrm{ERC}_0$ is the three-sorted structure $\mathcal{S}_0$ combining the Kleene Algebra $(\mathbb{K}, \mathit{false}, \mathit{true}, \mathit{unknown}, \hat{\wedge}, \hat{\vee}, \hat{\neg})$ with Presburger Arithmetic $(\mathbb{Z}, 0, 1, +, -, \leq, 2\mathbb{Z}, 3\mathbb{Z}, 4\mathbb{Z}, \ldots)$ and ordered field $(\mathbb{R}, 0, 1, +, -, \times, <)$. Available inter-sort functions are the *binary precision* embedding $\imath : \mathbb{Z} \ni p \mapsto 2^p \in \mathbb{R}$ and its partial half-inverse $\lfloor \log_2 \circ \operatorname{abs} \rfloor$ :

$\mathbb{R} \setminus \{0\} \to \mathbb{Z}$. Here $k\mathbb{Z}$ denotes the predicate on $\mathbb{Z}$ which is *true* precisely for all integer multiples of $k \in \mathbb{N}$.

The *(specification) language of* $\mathrm{ERC}_0$ is the first-order language $\mathcal{L}_0$ over the signature of the structure $\mathcal{S}_0$; and the *theory of* $\mathrm{ERC}_0$ is the first-order theory $\mathcal{T}_0$ containing true statements of $\mathcal{L}_0$ in $\mathcal{S}_0$.

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 the classical order relation of real numbers and is not to be confused with anything computational. Another important note is that the symbol $\perp$ in our denotational semantics does not appear in the structure of ERC.

Theorem 6.2(a) shows that the theory $\mathcal{T}_0$ is decidable. This applies for example to preconditions and postconditions or loop invariants of $\mathrm{ERC}_0$ programs; see Lemma 6.4 and Remark 6.9. 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* $\mathcal{T}_0$ *of* $\mathrm{ERC}_0$ *is decidable.*
(b) $\mathcal{T}_0$ *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* $\mathcal{S}_0$*:*
- *expanding with integer multiplication*
- *expanding with the unary predicate* $\mathbb{Z}$ *on* $\mathbb{R}$*, or with the standard coercion* $\mathbb{Z} \hookrightarrow \mathbb{R}$
- *replacing the binary precision embedding* $\imath$ *with its unary counterpart* $\jmath : \mathbb{N}_+ \ni n \mapsto 1/n \in \mathbb{R}$
- *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 $\jmath$. 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 [Dri85] extends classical Tarski-Seidenberg quantifier elimination from the first-order theory of real-closed fields to the expanded structure

$$(\mathbb{R}, 0, 1, +, -, \times, <, 2^{k\mathbb{Z}} : k \in \mathbb{N}, 2^{\lfloor \log_2 \circ \, \mathrm{abs} \rfloor}) \tag{6.1}$$

with axiomatized additional predicates $2^{k\mathbb{Z}}$, $k \in \mathbb{N}$, and truncation function to binary powers $2^{\lfloor \log_2 \circ \, \mathrm{abs} \rfloor}$, 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* [Bés02]:

- Replace quantifiers over Skolem integers with real quantifiers subject to the predicate $2^{k\mathbb{Z}}$ for $k := 1$;
- Consider $\imath : \mathbb{Z} \to \mathbb{R}$ as the restricted identity $\mathrm{id}_{2^{\mathbb{Z}}}$ 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 $2^{\lfloor \log_2 \circ \, \mathrm{abs} \rfloor}$).

To translate this back to some equivalent $\psi$ over the two-sorted structure, while re-introducing only one type of quantifier, observe that for real $x$:

$$\begin{aligned} x \in 2^{k\mathbb{Z}} \quad &\Leftrightarrow \quad \exists z \in \mathbb{Z}.\ z \in k\mathbb{Z} \ \wedge \ x = \imath(z); \\ x \notin 2^{k\mathbb{Z}} \quad &\Leftrightarrow \quad \exists z \in \mathbb{Z}.\ z \in k\mathbb{Z} \ \wedge \ \imath(z) < x < \imath(z + k)\,. \end{aligned}$$

Similarly, replace real binary truncation $2^{\lfloor \log_2 \circ \, \mathrm{abs}(x) \rfloor}$ with "$\imath(z)$" for some/every $z \in \mathbb{Z}$ s.t. $\imath(z) \le |x| < \imath(z) + 1$ in case $x > 0$, with 0 otherwise.

Since the Kleene Algebra $\mathbb{K}$ as the third sort is finite, it does not affect decidability. □

6.2. **Specification Language.** We define a specification language for each $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$ by expanding the structure of $\mathrm{ERC}_0$ as follows:

**Definition 6.3.** The *structure of* $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$ is the expansion $\mathcal{S}(\mathcal{F}, \mathcal{G})$ of $\mathcal{S}_0$ with the graphs of $f \in \mathcal{F}$ and $g \in \mathcal{G}$ as new relations:

- $\mathrm{Graph}_f := \{(x_1, \cdots, x_d; y) \mid f(x_1, \cdots, x_d) = y \neq \bot\}$ and
- $\mathrm{Graph}_g := \{(x_1, \cdots, x_d; y) \mid y \in g(x_1, \cdots, x_d) \wedge \bot \notin g(x_1, \cdots, x_d)\}$.

The *(specification) language of* $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$, which we write $\mathcal{L}(\mathcal{F}, \mathcal{G})$ is the first-order language $\mathcal{L}_0$ extended with the new relation symbols. The *theory of* $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$, which we write $\mathcal{T}(\mathcal{F}, \mathcal{G})$, is the sentences in the language that is true in $\mathcal{S}(\mathcal{F}, \mathcal{G})$.

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 (multivalued) functions, we may still have their function applications as appropriate abbreviations.

The following lemma shows that our specification logic is adequate for the term language of ERC:

**Lemma 6.4.** *For each well-typed term* $\Gamma \vdash t : \tau$ *in* ERC*, its denotation* $[\![\Gamma \vdash t : \tau]\!] : [\![\Gamma]\!] \to \mathbb{P}([\![\tau]\!]_\bot)$ *is definable in* $\mathcal{L}$ *in the sense that there is a formula* $\psi(x_1, \cdots, x_d; y)$ *defining the graph of the denotation (as in Definition 6.3). We represent a fixed length array, of say length* $d$*, with* $d$ *variables* $x = (x_{(0)}, \cdots, x_{(d-1)})$ *in an obvious way.*

*Proof.* Let $X, Y_i, Z \in \{[\![\tau]\!] \mid \tau \text{ is a data type}\}$. Suppose $f_i : X \to \mathbb{P}((Y_i)_\perp)$ and $g : Y_1 \times \cdots \times Y_d \to \mathbb{P}(Z_\perp)$ are definable partial multi-valued functions where $\psi_i(x; y_i)$ defines $f_i$ and $\psi(y_1, \cdots, y_d; z)$ defines $g$. Then, its composition $g^\dagger \circ \langle f_1, \cdots, f_d\rangle : X \to \mathbb{P}(Z_\perp)$ is definable by

$$\psi(x; z) :\equiv \exists y_1, \cdots, y_d.\ \psi_1(x; y_1) \wedge \cdots \wedge \psi_d(x; y_d) \wedge \psi(y_1, \cdots, y_d; z)\ .$$

Furthermore, if a partial function $f : X \to Z_\perp$ is definable by $\psi(x; z)$, its embedding $f^\ddagger$ 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, (2) Kond, and (3) proj:

(1) See that $\mathrm{choose}_n : \mathbb{K}^n \to \mathbb{Z}$ is definable by

$$\psi(x_1, \cdots, x_n; y) :\equiv (y = 0 \wedge x_1 = \mathit{true}) \vee \cdots \vee (y = n-1 \wedge x_{n-1} = \mathit{true})\ .$$

(2) Suppose $\psi_b(x; y)$ defines $b : X \to \mathbb{P}(\mathbb{K}_\perp)$, $\psi_f(x; y)$ defines $f : X \to \mathbb{P}(\mathbb{R}_\perp)$, and $\psi_g(x; y)$ defines $g : X \to \mathbb{P}(\mathbb{R}_\perp)$. Then, the composition $\mathrm{Kond}^{\dagger_1} \circ \langle b, f, g\rangle$ is definable by

$$\begin{aligned}\psi(x; z) :\equiv & \big(\psi_b(x; \mathit{true}) \Rightarrow \exists y.\ \psi_f(x; y)\big) \\ & \wedge\big(\psi_b(x; \mathit{false}) \Rightarrow \exists y.\ \psi_g(x; y)\big) \\ & \wedge\big(\psi_b(x; \mathit{unknown}) \\ & \quad \Rightarrow \big(\exists y.\ \psi_f(x; y) \wedge \psi_g(x; y) \wedge \forall y_1, y_2.\ \psi_f(x; y_1) \wedge \psi_g(x; y_2) \Rightarrow y_1 = y_2\big)\big) \\ & \wedge \big((\psi_b(x; \mathit{true}) \wedge \psi_f(x; z)) \vee (\psi_b(x; \mathit{false}) \wedge \psi_g(x; z)) \vee (\psi_b(x; \mathit{unknown}) \wedge \psi_f(x; z))\big)\ .\end{aligned}$$

(3) The partial projection map $\mathrm{proj} : \mathbb{R}^d \times \mathbb{Z} \to \mathbb{R}_\perp$ is defined by

$$\psi(x, k; y) :\equiv 0 \le k < d \wedge (k = 0 \Rightarrow y = x_{(0)}) \wedge \cdots \wedge (k = d-1 \Rightarrow y = x_{(d-1)})\ .$$

Note that when $k$ is an index out of range, there is no $y$ satisfying $\psi(x, k; y)$ since $\mathrm{proj}(x, k, y) = \perp$. $\square$

**Definition 6.5.** For a well-typed term $\Gamma \vdash t : \tau$ in ERC, let us write $(\!|\Gamma \vdash t : \tau|\!)$ for the formula defining $[\![\Gamma \vdash t : \tau]\!]$ according to Lemma 6.4.

We write simply $(\!|t|\!)$ for $(\!|\Gamma \vdash t : \tau|\!)$ when the particular context $\Gamma$ is obvious or irrelevant. We let the free variables representing the input values in $(\!|t|\!)$ be synchronized with $\Gamma$ and omit them in the notation; i.e., $(\!|t|\!)(y) := (\!|x_1 : \tau_1, \cdots, x_d : \tau_d \vdash t : \tau|\!)(x_1, \cdots, x_d; y)$.

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

**Example 6.6.** The restricted exponential function $\exp : I = [0; 1] \to \mathbb{R}$ is uniquely characterized by the following formula:

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

$$\forall x, y \in I.\ |\exp(x) - \exp(y)| \le 3 \cdot |x - y|$$

$$\exp(1) \ = \ \lim_n (1 + 1/n)^n \ = \ \sum\nolimits_n 1/n! \tag{6.3}$$

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 Equation (6.3) *exceeds* the first-order language 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. Propositional formulae in the specification language 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 $\mathcal{S}\big(\{\exp\}, \emptyset\big)$ 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. Nonetheless, 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 6.10. Since our language is simple imperative, we adopt the following notion of *total correctness specification*:

**Definition 6.7.** For a well-typed command $S$ in $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$ with $\Gamma \vdash S \triangleright \Gamma'$, a (total correctness) *specification*

$$\Gamma \vDash \big\{\phi\big\}\; S \;\big\{\psi\big\} \triangleright \Gamma'$$

is defined by

$$\forall \sigma \in [\![\Gamma]\!]\,.\; \bot \notin [\![S]\!]\,\sigma \wedge \forall \delta \in [\![S]\!]\,\sigma.\; \psi(\delta).$$

Here, $\phi$, $\psi$ are formulae in the specification language of $\mathrm{ERC}(\mathcal{F}, \mathcal{G})$. 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 $\sigma \in [\![\Gamma]\!]$ that satisfies $\phi$, (i) $\bot \notin [\![S]\!]\,\sigma$ and (ii) any $\delta \in [\![S]\!]\,\sigma$ satisfies $\psi$.

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

**Definition 6.8.** A *Hoare triple* for ERC$(\mathcal{F},\mathcal{G})$ is of the form $\Gamma \vdash \{\phi\}\ S\ \{\psi\} \triangleright \Gamma'$ where $S$ is a well-typed command in ERC$(\mathcal{F},\mathcal{G})$ such that $\Gamma \vdash S \triangleright \Gamma'$, and $\phi,\psi$ are formulae in the specification language of 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 ERC$(\mathcal{F},\mathcal{G})$ is a formal system which consists of the inference rules and axioms for constructing Hoare triples defined in Figure 3. □

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

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

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 our specification language of $\mathrm{ERC}_0$ according to Theorem 6.2(a).

**Remark 6.10.**

(1) In the axiom for assignments, recall from Definition 6.5 that the precondition $\exists w.\ (\!|t|\!)(w)$ ensures that the denotation of $t$ is well-defined. And, $\forall w.\ (\!|t|\!)(w) \Rightarrow \psi[w/x]$ says that for each value in the denotation of $t$, $\psi$ holds when we replace the variable $x$ 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.
   (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 *true* or *false* (but not *unknown*); and (iii) when $V$ is negative, it is guaranteed that the evaluation of $b$ is *false*.

(3) In the rule of **if** conditionals, the precondition $((\!|b|\!)(\mathit{true}) \vee (\!|b|\!)(\mathit{false})) \wedge \neg(\!|b|\!)(\mathit{unknown})$ says that the evaluation of $b$ is either *true* or *false* (but not *unknown*).

(4) In the rule of array assignments, $\mathrm{update}(x,i,w,y)$ says $x = (x_{(0)},\cdots,x_{(d-1)})$ is updated to $y = (y_{(0)},\cdots,y_{(d-1)})$ by assigning $w$ at $0 \le i < d$. □

**Theorem 6.11.** *The Hoare logic of* ERC$(\mathcal{F},\mathcal{G})$ *is sound; i.e., for any Hoare triple* $\Gamma \vdash \{\phi\}\ S\ \{\psi\} \triangleright \Gamma'$, *it holds that* $\Gamma \models \{\phi\}\ S\ \{\psi\} \triangleright \Gamma'$.

*Proof.* See Appendix A. □

Figure 3: The Hoare logic of ERC($\mathcal{F},\mathcal{G}$)

$$\frac{\Gamma \vdash \left\{\phi'\right\}\ S\ \left\{\psi'\right\} \triangleright \Gamma'}{\Gamma \vdash \left\{\phi\right\}\ S\ \left\{\psi\right\} \triangleright \Gamma'}\ \phi \Rightarrow \phi' \text{ and } \psi' \Rightarrow \psi \qquad\qquad \frac{}{\Gamma \vdash \left\{\psi\right\}\ \mathbf{skip}\ \left\{\psi\right\} \triangleright \Gamma}$$

$$\frac{}{\Gamma \vdash \left\{\exists w.\ (\!|t|\!)(w) \wedge \forall w.\ (\!|t|\!)(w) \Rightarrow \psi[w/x]\ \right\}\ x \coloneqq t\ \ \{\psi\}\ \triangleright \Gamma}$$

$$\frac{}{\Gamma \vdash\ \{\mathsf{ArrPre}(\psi, x, m, t)\}\ \ x[m] \coloneqq t\ \ \{\psi\}\ \triangleright \Gamma}$$

$$\frac{}{\Gamma \vdash \left\{\exists w.\ (\!|t|\!)(w) \wedge \forall w.\ (\!|t|\!)(w) \Rightarrow \psi[w/x]\right\}\ \mathbf{let}\ x : \tau = t\ \left\{\psi\right\} \triangleright \Gamma, x : \tau}$$

$$\frac{\Gamma \vdash\ \{\phi\}\ \ S_1\ \ \{\theta\}\ \triangleright \Gamma_1 \quad \Gamma_1 \vdash \left\{\theta\right\}\ S_2\ \ \{\psi\}\ \triangleright \Gamma_2}{\Gamma \vdash \left\{\phi\right\}\ S_1; S_2\ \left\{\psi\right\} \triangleright \Gamma_2}$$

$$\frac{\Gamma \vdash\ \{\phi \wedge\ (\!|b|\!)(\mathit{true})\}\ \ S_1\ \ \{\psi\}\ \triangleright \Gamma_1 \quad \Gamma \vdash\ \{\phi \wedge (\!|b|\!)(\mathit{false})\}\ \ S_2\ \ \{\psi\}\ \triangleright \Gamma_2}{\Gamma \vdash\ \{\phi \wedge ((\!|b|\!)(\mathit{true}) \vee (\!|b|\!)(\mathit{false})) \wedge \neg(\!|b|\!)(\mathit{unknown})\}\ \ \mathbf{if}\ b\ \mathbf{then}\ S_1\ \mathbf{else}\ S_2\ \mathbf{end}\ \ \{\psi\}\ \triangleright \Gamma}$$

$$\frac{\Gamma, \xi, \xi' : \mathsf{R} \vdash\ \{(\!|b|\!)(\mathit{true}) \wedge I \wedge V = \xi \wedge L = \xi'\}\ \ S\ \ \{I \wedge V \le \xi - \xi' \wedge L = \xi'\}\ \triangleright \Gamma''}{\Gamma \vdash\ \{I\}\ \ \mathbf{while}\ b\ \mathbf{do}\ S\ \mathbf{end}\ \ \{I \wedge (\!|b|\!)(\mathit{false})\}\ \triangleright \Gamma}$$

The rule for loops has the side conditions:

$$I \wedge (\!|b|\!)(\mathit{true}) \Rightarrow L > 0\ ,$$
$$I \Rightarrow ((\!|b|\!)(\mathit{true}) \vee (\!|b|\!)(\mathit{false})) \wedge \neg(\!|b|\!)(\mathit{unknown})\ ,$$
$$I \wedge V \le 0 \Rightarrow \forall k.\ (\!|b|\!)(k) \Rightarrow k = \mathit{false}\ ,\ \text{and}$$
$$\xi, \xi' \text{ do not appear free in } I, V, L\ .$$

where $L$ and $V$ are real-valued.

In the case of array assignment $\mathsf{ArrPre}(\psi, x, m, t)$ is defined as follows:

$$\begin{aligned}\mathsf{ArrPre}(\psi, x, m, t) :\equiv{} & \big(\exists i, w.\ (\!|m|\!)(i) \wedge (\!|t|\!)(w)\big) \\ & \wedge\ \forall i.\ ((\!|m|\!)(i) \Rightarrow 0 \le i < d) \\ & \wedge\ \forall i, w.\ ((\!|m|\!)(i) \wedge (\!|t|\!)(w) \wedge \forall y.\ \mathrm{update}(x, i, w, y) \Rightarrow \psi[y/x])\end{aligned}$$

assuming $\Gamma \vdash x[m] \coloneqq t\ \triangleright \Gamma$ and $\Gamma \vdash x : \mathsf{R}[d]$ where

$$\begin{aligned}\mathrm{update}(x, i, w, y) :\equiv{} & 0 \le i < d \wedge ((y_{(0)} = x_{(0)} \wedge i \ne 0) \vee (y_{(0)} = w \wedge i = 0)) \\ & \wedge \cdots \wedge ((y_{(d-1)} = x_{(d-1)} \wedge i \ne d-1) \vee (y_{(d-1)} = w \wedge i = d-1))\end{aligned}$$

## 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 floating point, rational, or algebraic) numbers.

Let us define some abbreviations such that the program in Algorithm 9 becomes of the form

$$\textbf{input } a : \mathsf{R}, b : \mathsf{R} \; S \textbf{ return } a \textbf{ as } p \to -\infty.$$

$$\begin{aligned}
\tilde{t}_1 \; &:\equiv \; b - a \lessdot 2^p \;, \;\; \tilde{t}_2 \coloneqq \; 2^{p-1} \lessdot b - a \\
t_1 \; &:\equiv \; f(b/3 + 2 \times a/3) \times f(b) \lessdot 0 \;, \;\; t_2 :\equiv f(a) \times f(2 \times b/3 + a/3) \lessdot 0 \\
b_1 \; &:\equiv \; \mathsf{choose}(\tilde{t}_1, \tilde{t}_2) = 1, \;\; b_2 \coloneqq \; \mathsf{choose}(t_1, t_2) = 1 \\
S \; &:\equiv \; \textbf{while } b_1 \textbf{ do } S_1 \textbf{ end} \\
S_1 \; &:\equiv \; \textbf{if } b_2 \textbf{ then } S_2 \textbf{ else } S_3 \textbf{ end} \\
S_2 \; &:\equiv \; b \coloneqq 2 \times b/3 + a/3 \\
S_3 \; &:\equiv \; a \coloneqq b/3 + 2 \times a/3
\end{aligned}$$

We work with $\mathcal{T}(\{f\}, \emptyset)$ 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:

$$\mathrm{uniq}(f, a, b) :\equiv \mathrm{cont}(f, a, b) \wedge f(a) \cdot f(b) < 0 \wedge \exists! x.\; a < x < b \wedge f(x) = 0 \;.$$

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

$$\Gamma \vDash \big\{p = p' \wedge \mathrm{uniq}(f, a, b)\big\} \; S \; \big\{\exists! z.\; f(z) = 0 \wedge a < z < b \wedge |a - z| \leq 2^{p'}\big\} \triangleright \Gamma'$$

where $\Gamma = p, p' : \mathsf{Z}, a, b : \mathsf{R}$ and $\Gamma' = p, p' : \mathsf{Z}, a, b : \mathsf{R}$. 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 $\lessdot$ are implicitly replaced with $<$. See that $(\!|b_1|\!)(\mathit{true}) \Leftrightarrow \tilde{t}_2$, $(\!|b_1|\!)(\mathit{false}) \Leftrightarrow \tilde{t}_1$, $\neg(\!|b_1|\!)(\mathit{unknown})$, $(\!|b_2|\!)(\mathit{true}) \Leftrightarrow t_2$, $(\!|b_2|\!)(\mathit{false}) \Leftrightarrow t_1$, and $\neg(\!|b_2|\!)(\mathit{unknown})$ hold. Let us define $I \coloneqq p = p' \wedge \mathrm{uniq}(f, a, b)$ as a candidate for the loop invariant, $V \coloneqq b - a - 2^{p-1}$ as a candidate for the loop variant, $L \coloneqq 2^{p-2}$ be a candidate for a lower bound decrements, $\tilde{P} \coloneqq \tilde{t}_2 \wedge I \wedge V = \xi \wedge L = \xi'$, and $\tilde{Q} \coloneqq I \wedge V \leq \xi - \xi' \wedge L = \xi'$ in our specification language with variables $\xi, \xi'$ of the sort $\mathbb{R}$. Let $\Delta \coloneqq p, p' : \mathsf{Z}, a, b, \xi, \xi' : \mathsf{R}$.

From the axiom for assignments, we have the triples:

$$\Delta \vdash \big\{\exists \omega.\; (\!|a/3 + 2 \times b/3|\!)(\omega) \wedge \forall \omega.\; (\!|a/3 + 2 \times b/3|\!)(\omega) \Rightarrow \tilde{Q}[\omega/b]\big\} \; S_2 \; \big\{\tilde{Q}\big\} \triangleright \Delta$$

$$\Delta \vdash \big\{\exists \omega.\; (\!|2 \times a/3 + b/3|\!)(\omega) \wedge \forall \omega.\; (\!|2 \times a/3 + .b/3|\!)(\omega) \Rightarrow \tilde{Q}[\omega/a]\big\} \; S_3 \; \big\{\tilde{Q}\big\} \triangleright \Delta$$

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

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

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

$$\begin{aligned} \tilde{Q}[(2 \times a/3 + b/3)/a] \;&\Leftrightarrow\; p = p' \wedge \mathrm{uniq}(f, (2 \times a/3 + b/3), b) \\ &\quad\wedge\;\; b - (2 \times a/3 + b/3) - 2^{p-1} \leq \xi - \xi' \\ &\quad\wedge\;\; 2^{p-2} = \xi' \\ \tilde{P} \wedge t_1 \;&\Leftrightarrow\; p = p' \wedge \mathrm{uniq}(f, a, b) \wedge f(b/3 + 2 \times a/3) \times f(b) < 0 \\ &\quad\wedge\;\; 2^{p-1} < b - a \wedge b - a - 2^{p-1} = \xi \\ &\quad\wedge\;\; 2^{p-2} = \xi' \end{aligned}$$

Due to the 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, $\tilde{P} \wedge t_1 \Rightarrow \tilde{Q}[(2 \times a/3 + b/3)/a]$ holds. And, similarly, $\tilde{P} \wedge t_2 \Rightarrow \tilde{Q}[(a/3 + 2 \times b/3)/b]$ 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:

$$\Delta \vdash \left\{\tilde{t}_2 \wedge I \wedge \big(V = \xi\big) \wedge L = \xi'\right\} S_1 \left\{I \wedge (V \leq \xi - \xi') \wedge L = \xi'\right\} \triangleright \Delta$$

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:

$$\Gamma \vdash \left\{I\right\} S \left\{I \wedge \tilde{t}_2\right\} \triangleright \Gamma'$$

Using the rule of pre/postcondition strengthening/weakening, we can get the originally desired specification.

## 8. Conclusion and Future work

We have formalized an imperative programming language for exact real number computation. 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 a 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 "$x \lessdot y$" the logarithm of the difference $|x - y|$.

- **Full Mixed Data Types:** ERC as introduced here formalizes computing with data types $\mathsf{Z}$, $\mathsf{R}$, and $\mathsf{K}$ as counterparts to mathematical $\mathbb{Z}, \mathbb{R}, \mathbb{K}$ where $\mathsf{K}$ is permitted only for expressions and local variables. A future extension will include also (multi-)functions with $\mathsf{K}$ type arguments and return values as well as arrays and a dedicated limit operator.
- **Explicit Limit Operator:** Constructing real numbers as the limits of converging sequences is done only at the level of programs in ERC. It was our design choice to separate the term language of ERC with loops and of course limits. A direction of extending our framework to allow constructing a limit within user programs as a term has been also investigated [BPS20] where the authors suggest an expression-based programming language called Clerical that provides an explicit limit operator. Its prototype implementation can be found in [BPS17].
- **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 restricted to the single-valued case. Defining approximate computation of real multi-valued functions is delicate and still under exploration [BMR18, Kon18].
- **Function*al*s and Operators:** The program in Algorithm 9 mimics a function*al* that receives a continuous function $f$ as an argument, accessible by point-wise black box 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 $\mathrm{ERC}_0$ according to Theorem 6.2 includes its complete (and 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 [KST20, 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 $\mathrm{T}_0$ 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 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. We would like to thank Holger Thies for constructive comments on this work and proofreading.

A preliminary version of this work was submitted as a part of the first author's PhD thesis [Par21].

## 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. doi:10.1007/s00165-019-00501-3.

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

[BC90] Hans Boehm and Robert Cartwright. Exact Real Arithmetic: Formulating real numbers as functions. In *Research Topics in Functional Programming*, pages 43—64. Addison-Wesley Longman Publishing Co., Inc., USA, 1990.

[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.

[BCSS98] Lenore Blum, Felipe Cucker, Michael Shub, and Steven Smale. *Complexity and real computation*. Springer, New York, NY, 1998. doi:10.1007/978-1-4612-0701-6.

[Bés02] Alexis Bés. 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. doi:10.1007/978-3-642-02614-0_10.

[BH98] Vasco Brattka and Peter Hertling. Feasible real random access machines. *Journal of Complexity*, 14(4):490–526, 1998. doi:10.1006/jcom.1998.0488.

[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. doi:10.1109/ARITH.2011.40.

[BMR18] Franz Brausse, Norbert Müller, and Robert Rettinger. Intensionality and multi-valued limits. In *Proc. 15th International Conference on Computability and Complexity in Analysis (CCA)*, page 11, 2018.

[BPS17] Andrej Bauer, Sewon Park, and Alex Simpson. Clerical. https://github.com/andrejbauer/clerical, 2017.

[BPS20] Andrej Bauer, Sewon Park, and Alex Simpson. Clerical: an imperative language for verified real-number computation. In *Proc. 17th International Conference on Computability and Complexity in Analysis (CCA)*, pages 22–23, 2020.

[Bra95] Vasco Brattka. Computable selection in analysis. In *Proc. of the Workshop on Computability and Complexity in Analysis, Informatik Berichte, FernUniversit at 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. doi:10.1142/9789812704979_0005.

[BVS93] Stephen Brookes and Kathryn Van Stone. Monads and comonads in intensional semantics. Technical report, Carnegie-Mellon University Pittsburgh PA Department 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. doi:10.1137/0207005.

[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. doi:10.1007/978-3-642-37075-5_22.

[Dri85] Lou van den Dries. The field of reals with a predicate for the powers of two. *Manuscripta mathematica*, 54:187–195, 1985. doi:10.1007/BF01171706.

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

[ES14] Martín 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. doi:10.1007/978-3-319-08918-8_15.

[Esc96] Martín Hötzel Escardó. PCF extended with real numbers. *Theoretical Computer Science*, 162:79–115, 1996. doi:10.1016/0304-3975(95)00250-2.

[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. doi:10.1007/978-3-642-37036-6_8.

[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. doi:10.1002/malq.19990450202.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, oct 1969. 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. doi:10.1137/S0097539794263452.

[KC12] Akitoshi 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. *Computational Geometry*, 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 Konečný. Verified exact real limit computation. In *Proc. 15th International Conference on Computability and Complexity in Analysis (CCA)*, pages 9–10, 2018.

[KST20] Michal Konečný, Florian Steinberg, and Holger Thies. Computable analysis for verified exact real computation. In Nitin Saxena and Sunil Simon, editors, *40th IARCS Annual Conference on*

*Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2020, December 14-18, 2020, BITS Pilani, K K Birla Goa Campus, Goa, India (Virtual Conference)*, volume 182 of *LIPIcs*, pages 50:1–50:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.FSTTCS.2020.50.

[KSZ16] 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. doi:10.1145/2933575.2935311.

[KTD+13] Michal Konečný, Walid Taha, Jan Duracz, Adam Duracz, and Aaron Ames. Enclosing the behavior of a hybrid system up to and beyond a zeno point. In *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, pages 120–125, 2013. doi:10.1109/CPSNA.2013.6614258.

[Lam07] Branimir Lambov. RealLib: An efficient implementation of exact real arithmetic. *Mathematical Structures in Computer Science*, 17:81–98, 2007. doi:10.1017/S0960129506005822.

[Luc77] Horst Luckhardt. A fundamental effect in computations on real numbers. *Theoretical Computer Science*, 5(3):321–324, 1977. doi:10.1016/0304-3975(77)90048-2.

[Mar17] David Marker. *Model Theory of Fields*. Lecture Notes in Logic. Cambridge University Press, Cambridge, United Kingdom, 2017. doi:10.1017/9781316716991.

[Mel08] 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. doi:10.1007/978-3-540-71070-7_2.

[Mit70] Dragoslav S. Mitrinović. *Analytic Inequalities*, volume 165 of *Die Grundlehren der mathematischen Wissenschaften*. Springer, 1970. doi:10.1007/978-3-642-99970-3.

[MRE07] J. Raymundo Marcial-Romero and Martín H. Escardó. Semantics of a sequential language for exact real-number computation. *Theoretical Computer Science*, 379(1):120–141, 2007. doi:10.1016/j.tcs.2007.01.021.

[Mül01] 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. doi:10.1007/3-540-45335-0_14.

[NMB06] 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. doi:10.1145/1160074.1159812.

[Par21] Sewon Park. *Continuous Abstract Data Types for Verified Computation*. PhD thesis, KAIST, School of Computing, Daejeon, South Korea, 2021.

[PER89] Marian B. Pour-El and J. Ian Richards. *Computability in Analysis and Physics*. Perspectives in Mathematical Logic. Springer, Berlin, 1989. doi:10.1017/9781316717325.

[Plo76] Gordon D. Plotkin. A powerdomain construction. *SIAM Journal on Computing*, 5(3):452–487, 1976. doi:10.1137/0205035.

[Plo77] Gordon D. Plotkin. LCF considered as a programming language. *Theoretical computer science*, 5(3):223–255, 1977. doi:10.1016/0304-3975(77)90044-5.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry*. Monographs in Computer Science. Springer, 1985. doi:10.1007/978-1-4612-1098-6.

[PTVF07] 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.

[Rey02] 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. doi:10.1109/LICS.2002.1029817.

[Rey09] John C Reynolds. *Theories of programming languages*. Cambridge University Press, Cambridge, United Kingdom, 2009. doi:10.1017/CBO9780511626364.

[Ric68] Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. *Journal of Symbolic Logic*, 33(4):514–520, 1968. doi:10.2307/2271358.

[Rob59] Julia Robinson. The undecidability of algebraic rings and fields. *Proceedings of the American Mathematical Society*, 10:950–957, 1959. doi:10.2307/2033628.

[SAL+18] 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.

[Sch02] Matthias Schröder. Effectivity in spaces with admissible multirepresentations. *Mathematical Logic Quarterly*, 48(S1):78–90, 2002.

[Sch04] Matthias Schröder. Spaces allowing type-2 complexity theory revisited. *Mathematical Logic Quarterly*, 50(4,5):443–459, 2004. doi:10.1002/malq.200310111.

[Spe59] Ernst Specker. Der Satz vom Maximum in der rekursiven Analysis. In A. Heyting, editor, *Constructivity in mathematics*, Studies in Logic and the Foundations of Mathematics, pages 254–265, Amsterdam, 1959. North-Holland. Proc. Colloq., Amsterdam, Aug. 26–31, 1957.

[STT21] Florian Steinberg, Laurent Thery, and Holger Thies. Computable analysis and notions of continuity in Coq. *Logical Methods in Computer Science*, Volume 17, Issue 2, May 2021. doi:10.23638/LMCS-17(2:16)2021.

[SZ18] Dongseong Seon and Martin Ziegler. Computing Haar averages. In *Proc. 15th International Conference on Computability and Complexity in Analysis (CCA)*, pages 43–44, 2018.

[Tur37] Alan M. Turing. On computable numbers, with an application to the "Entscheidungsproblem". *Proceedings of the London Mathematical Society*, 42(2):230–265, 1937.

[Tur48] Alan M. Turing. Rounding off errors in matrix processes. *Quarterly Journal of Mechanical and Applied Math*, 1(1):287–308, 1948.

[TZ99] J.V. Tucker and J.I. Zucker. Computation by 'While' programs on topological partial algebras. *Theoretical Computer Science*, 219:379–420, 1999. doi:10.1016/S0304-3975(98)00297-7.

[TZ04] J.V. Tucker and J.I. Zucker. Abstract versus concrete computation on metric partial algebras. *ACM Transactions on Computational Logic*, 5(4):611–668, 2004. doi:10.1145/1024922.1024924.

[Wei00] Klaus Weihrauch. *Computable Analysis*. Springer, Berlin, 2000. doi:10.1007/978-3-642-56999-9.

[Wei03] Klaus Weihrauch. Computational complexity on computable metric spaces. *Mathematical Logic Quarterly*, 49(1):3–21, 2003. doi:10.1002/malq.200310001.

[YYD+10] Jihun Yu, Chee Yap, Zilin Du, Sylvain Pion, and Hervé Brönnimann. The design of Core 2: A library for exact numeric computation in geometry and algebra. In Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama, editors, *Mathematical Software – ICMS 2010*, pages 121–141, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-15582-6_24.

## 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* $[\![\Gamma]\!]$*:*

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

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

*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 $\mathbf{S}$ for $\upharpoonright_{\Gamma}^{\ddagger\dagger} \circ [\![S]\!]$.

We first prove the following alternative characterization of the sequence of sets:

$$B^{n+1}(\sigma) = \bigcup_{\substack{\ell \in [\![b]\!]\sigma \\ \delta \in \mathbf{S}\sigma}} \begin{cases} B^n(\delta) & \text{if } \ell = \textit{true} \wedge \delta \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases}$$

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.

$$\begin{aligned}
B^{m+2}(\sigma) &= \bigcup_{\gamma \in B^{m+1}(\sigma)} \bigcup_{\substack{\ell \in [\![b]\!]\gamma \\ \delta \in \mathbf{S}\gamma}} \begin{cases} \{\delta\} & \text{if } \ell = \textit{true} \wedge \delta \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases} \\
&= \bigcup_{\gamma \in \bigcup_{\substack{\ell' \in [\![b]\!]\sigma \\ \delta' \in \mathbf{S}\sigma}} \begin{cases} B^m(\delta') & \text{if } \ell' = \textit{true} \wedge \delta' \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases}} \bigcup_{\substack{\ell \in [\![b]\!]\gamma \\ \delta \in \mathbf{S}\gamma}} \begin{cases} \{\delta\} & \text{if } \ell = \textit{true} \wedge \delta \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases} \\
&= \bigcup_{\substack{\ell' \in [\![b]\!]\sigma \\ \delta' \in \mathbf{S}\sigma}} \begin{cases} \bigcup_{\gamma \in B^m(\delta')} \bigcup_{\substack{\ell \in [\![b]\!]\gamma \\ \delta \in \mathbf{S}\gamma}} \begin{cases} \{\delta\} & \text{if } \ell = \textit{true} \wedge \delta \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases} & \text{if } \ell' = \textit{true} \wedge \delta' \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases} \\
&= \bigcup_{\substack{\ell' \in [\![b]\!]\sigma \\ \delta' \in \mathbf{S}\sigma}} \begin{cases} B^{m+1}(\delta') & \text{if } \ell' = \textit{true} \wedge \delta' \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases}
\end{aligned}$$

We now show the following characterization:

$$C^{n+1}(\sigma) = \bigcup_{\substack{\ell \in [\![b]\!]\sigma \\ \delta \in \mathbf{S}\sigma}} \begin{cases} C^n(\delta) & \text{if } \ell = \mathit{true} \wedge \delta \neq \bot, \\ \{\sigma\} & \text{if } \ell = \mathit{false}, \\ \{\bot\} & \text{otherwise.} \end{cases}$$

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

$$\begin{aligned}
C^{m+2}(\sigma) &= C^{m+1}(\sigma) \cup \bigcup_{\delta \in B^{m+1}(\sigma)} \bigcup_{\substack{\ell' \in [\![b]\!]\delta \\ \delta' \in \mathbf{S}\delta}} \begin{cases} \{\delta\} & \text{if } \ell' = \mathit{false}, \\ \emptyset & \text{if } \ell' = \mathit{true} \wedge \delta' \neq \bot, \\ \{\bot\} & \text{otherwise.} \end{cases} \\
&= C^{m+1}(\sigma) \cup \\
&\qquad \bigcup_{\delta \in \bigcup_{\substack{\ell \in [\![b]\!]\sigma \\ \gamma \in \mathbf{S}\sigma}} \begin{cases} B^m(\gamma) & \text{if } \ell = \mathit{true} \wedge \gamma \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases}} \bigcup_{\substack{\ell' \in [\![b]\!]\delta \\ \delta' \in \mathbf{S}\delta}} \begin{cases} \{\delta\} & \text{if } \ell' = \mathit{false}, \\ \emptyset & \text{if } \ell' = \mathit{true} \wedge \delta' \neq \bot, \\ \{\bot\} & \text{otherwise.} \end{cases} \\
&= C^{m+1}(\sigma) \cup \\
&\qquad \bigcup_{\substack{\ell \in [\![b]\!]\sigma \\ \gamma \in \mathbf{S}\sigma}} \begin{cases} \bigcup_{\delta \in B^m(\gamma)} \bigcup_{\substack{\ell \in [\![b]\!]\delta \\ \delta' \in \mathbf{S}\delta}} \begin{cases} \{\delta\} & \text{if } \ell' = \mathit{false} \\ \emptyset & \text{if } \ell' = \mathit{true} \wedge \delta' \neq \bot, \\ \{\bot\} & \text{otherwise.} \end{cases} & \text{if } \ell = \mathit{true} \wedge \gamma \neq \bot, \\ \emptyset & \text{otherwise.} \end{cases} \\
&= \bigcup_{\substack{\ell \in [\![b]\!]\sigma \\ \gamma \in \mathbf{S}\sigma}} \begin{cases} C^m(\gamma) \cup & \text{if } \ell = \mathit{true} \wedge \gamma \neq \bot, \\ \quad \bigcup_{\delta \in B^m(\gamma)} \bigcup_{\substack{\ell \in [\![b]\!]\delta \\ \delta' \in \mathbf{S}\delta}} \begin{cases} \{\delta\} & \text{if } \ell' = \mathit{false} \\ \emptyset & \text{if } \ell' = \mathit{true} \wedge \delta' \neq \bot, \\ \{\bot\} & \text{otherwise.} \end{cases} & \\ \{\sigma\} & \text{if } \ell = \mathit{false}, \\ \{\bot\} & \text{otherwise.} \end{cases} \\
&= \bigcup_{\substack{\ell \in [\![b]\!]\sigma \\ \gamma \in \mathbf{S}\sigma}} \begin{cases} C^{m+1}(\gamma) & \text{if } \ell = \mathit{true} \wedge \gamma \neq \bot, \\ \{\sigma\} & \text{if } \ell = \mathit{false}, \\ \{\bot\} & \text{otherwise.} \end{cases}
\end{aligned}$$

Using the suggested characterization, we prove $\mathcal{W}^{(n)}(\sigma) = C^n_{b,S}(\sigma) \cup \{\bot \mid \exists x.\ x \in B^n_{b,S}(\sigma)\}$ for all $n \in \mathbb{N}$. When $n = 0$, both are $\{\bot\}$. Suppose the equation holds for $n = m$.

Then,

$$\begin{aligned}
\mathcal{W}^{(n+1)}(\sigma) &= \bigcup_{\substack{\ell\in[\![b]\!]\sigma\\ \delta\in\mathbf{S}\sigma}} \begin{cases} \mathcal{W}^{(n)}(\delta) & \text{if } \ell = \mathit{true} \wedge \delta \neq \bot,\\ \{\sigma\} & \text{if } \ell = \mathit{false},\\ \{\bot\} & \text{otherwise.}\end{cases}\\
&= \bigcup_{\substack{\ell\in[\![b]\!]\sigma\\ \delta\in\mathbf{S}\sigma}} \begin{cases} C^{m}(\delta) & \text{if } \ell = \mathit{true} \wedge \delta \neq \bot,\\ \{\sigma\} & \text{if } \ell = \mathit{false},\\ \{\bot\} & \text{otherwise.}\end{cases}\\
&\qquad \cup \bigcup_{\substack{\ell\in[\![b]\!]\sigma\\ \delta\in\mathbf{S}\sigma}} \begin{cases} \{\bot | \exists\gamma.\ \gamma\in B^{m}(\delta)\} & \text{if } \ell = \mathit{true} \wedge \delta \neq \bot\\ \emptyset & \text{if } \ell = \mathit{false},\\ \emptyset & \text{otherwise.}\end{cases}\\
&= C^{m+1}(\sigma) \cup \left\{ \bot \;\middle|\; \exists\gamma.\ \gamma \in \bigcup_{\substack{\ell\in[\![b]\!]\sigma\\ \delta\in\mathbf{S}\sigma}} \begin{cases} B^{m}(\delta) & \text{if } \ell = \mathit{true} \wedge \delta \neq \bot,\\ \emptyset & \text{otherwise.}\end{cases} \right\}\\
&= C^{m+1}(\sigma) \cup \{\bot \mid \exists\gamma.\ \gamma\in B^{m+1}(\sigma)\}
\end{aligned}$$

□

We now proceed to the proof of Theorem 6.11:

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

(1) (Assignment):

Consider any state $\sigma$ which satisfies $\exists w.\ (\!|t|\!)(w) \wedge \forall w.\ (\!|t|\!)(w) \Rightarrow \psi[w/x]$. Then, $\bot \notin [\![t]\!]\,\sigma$ and for any $w \in [\![t]\!]\,\sigma$, $\psi[w/x]$ holds.

Now, see that $[\![x \coloneqq t]\!]\,\sigma = \bigcup_{w\in[\![t]\!]\sigma}\{\sigma[x \mapsto w]\}$ since $\bot \notin [\![t]\!]\,\sigma$ and for all $w \in [\![t]\!]\,\sigma$, $\sigma[x \mapsto w]$ satisfies $\psi$.

(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 trivially.

(3) (Conditional):

Consider any state $\sigma$ which satisfies $\phi \wedge ((\!|b|\!)(\mathit{true}) \vee (\!|b|\!)(\mathit{false})) \wedge \neg(\!|b|\!)(\mathit{unknown})$. Then, $[\![b]\!]\,\sigma = \{\mathit{true}, \mathit{false}\}, \{\mathit{true}\}$, or $\{\mathit{false}\}$. Let us check the three cases:

(a) when $[\![b]\!]\,\sigma = \{\mathit{true}, \mathit{false}\}$:

Then, $\sigma$ satisfies $\phi \wedge (\!|b|\!)(\mathit{true})$ and $\phi \wedge (\!|b|\!)(\mathit{false})$. Therefore, (i) $\bot \notin [\![S_1]\!]\,\sigma$, (ii) for all $\delta \in [\![S_1]\!]\,\sigma$ it holds that $\delta \vDash \psi$, (iii) $\bot \notin [\![S_2]\!]\,\sigma$, and (iv) for all $\delta \in [\![S_2]\!]\,\sigma$ it holds that $\delta \vDash \psi$.

Since $\bot \notin [\![b]\!]\,\sigma$ and $\mathit{unknown} \notin [\![b]\!]\,\sigma$, the denotation becomes

$$[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}]\!]\,\sigma = \upharpoonright^{\ddagger\dagger}_{\Gamma} ([\![S_1]\!]\,\sigma) \cup \upharpoonright^{\ddagger\dagger}_{\Gamma} ([\![S_2]\!]\,\sigma).$$

Since the restriction operator does not create $\bot$, the denotation does not contain $\bot$. Also, since $\psi$ only consists of free variables that are in $\Gamma$, each resulting state after the restriction still satisfies $\psi$.

(b) when $[\![b]\!]\,\sigma = \{\mathit{true}\}$:
Then, $\sigma$ satisfies $\phi \wedge (\!|b|\!)(\mathit{true})$. Hence, (i) $\bot \notin [\![S_1]\!]\,\sigma$, (ii) each $\delta \in [\![S_1]\!]\,\sigma$ satisfies $\psi$. Since $[\![b]\!]\,\sigma = \{\mathit{true}\}$, the denotation becomes

$$[\![\textbf{if } b \textbf{ then } S_1 \textbf{ else } S_2 \textbf{ end}]\!]\,\sigma = \upharpoonright_{\Gamma}^{\ddagger\dagger} ([\![S_1]\!]\,\sigma).$$

Again, since the restriction does not create $\bot$ and $\bot \notin [\![S_1]\!]\,\sigma$, $\bot$ is not in the denotation. Since $\psi$ only consists of free variables that are in $\Gamma$, each resulting state after the restriction still satisfies $\psi$

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

(4) (Loop):

Consider any state $\sigma$ that satisfies $I$. Then, by the side-conditions, it also satisfies $((\!|b|\!)(\mathit{true}) \vee (\!|b|\!)(\mathit{false})) \wedge \neg(\!|b|\!)(\mathit{unknown})$. Hence, $[\![b]\!]\,\sigma = \{\mathit{true}, \mathit{false}\}, \{\mathit{true}\}$, or $\{\mathit{false}\}$ for any state $\sigma$ that satisfies $I$. Now, we fix a state $\sigma$ which satisfies $I$ hence satisfies the precondition.

The core part of the proof is the statement: for any natural number $n$, it holds that (i) $\bot \notin B^n_{b,S}(\sigma)$, (ii) $\bot \notin C^n_{b,S}(\sigma)$, (iii) all $\delta$ in $B^n_{b,S}(\sigma)$ or $C^n_{b,S}(\sigma)$ satisfies $I$, and (iv) all $\delta$ in $C^n_{b,S}(\sigma)$ satisfies $(\!|b|\!)(\mathit{false})$.

At the moment, suppose that the above statement is true. Then, all we have to show is that $B^m_{b,S}(\sigma)$ becomes empty as $m \in \mathbb{N}$ increases. Let us define $\ell_n \coloneqq \max\{V(\delta) \mid \delta \in B^n_{b,S}(\sigma)\}$ 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 $\delta \in B^m_{b,S}(\sigma)$, $[\![b]\!]\,\delta = \{\mathit{false}\}$ and hence $B^{m+1}_{b,S}(\sigma) = \emptyset$.

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

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

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

Now, we prove each statement:

(a) $B^1_{b,S}(\sigma) \neq \emptyset$ only if $\mathit{true} \in [\![b]\!]\,\sigma$ and there is some non-bottom $\delta \in \mathbf{S}$. Therefore, by the side-condition, $L(\sigma) > 0$.
Suppose any $\delta \in B^{m+1}_{b,S}(\sigma)$ for any $m \in \mathbb{N}$. See that it happens only if there is $\delta' \in B^m_{b,S}(\sigma)$ such that $\mathit{true} \in [\![b]\!]\,\delta'$ and $\delta \in \mathbf{S}\delta'$. Together with Item (iii), $\delta'$ satisfies $I$ and $(\!|b|\!)(\mathit{true})$. Let us define $\hat{\delta}' \coloneqq \delta' \cup (\xi \mapsto V(\delta') \cup \xi' \mapsto L(\delta')$. Since $\hat{\delta}'$ satisfies the precondition in the premise, we have that for any $\hat{\delta} \in \mathbf{S}\hat{\delta}'$, $\hat{\delta}$ satisfies $I$ and $V \leq \xi - \xi'$ and $L = \xi'$. Hence, $L(\hat{\delta}) = L(\delta')$. Since $\xi', \xi$ are ghost variables, $L(\hat{\delta}) = L(\delta) = L(\delta')$. In conclusion, for any $\delta \in B^{m+1}_{b,S}(\sigma)$, the quantity $L(\delta)$ is identical to the quantity $L(\delta')$ for some $\delta' \in B^m_{b,S}(\sigma)$. Since, $B^0_{b,S}(\sigma) = \{\sigma\}$, we conclude that they are all identical to $L(\sigma)$.

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

Now, we need to prove the aforementioned statement on $B^m_{b,S}$ and $C^m_{b,S}$:

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

(b) (Induction step): Recall $B^{n+1}_{b,S}(\sigma) :\equiv \bigcup_{\delta \in B^n_{b,S}(\sigma)} \bigcup_{\substack{l \in [\![b]\!]\delta \\ \delta' \in \mathbf{S}\delta}} \begin{cases} \{\delta'\} & \text{if } l = \mathit{true} \wedge \delta' \neq \bot \\ \emptyset & \text{otherwise.} \end{cases}$.

Since all $\delta \in B^n_{b,S}(\sigma)$ satisfies $I$, $[\![b]\!]\,\delta = \{\mathit{true}\}, \{\mathit{false}\}$, or $\{\mathit{true}, \mathit{false}\}$. In the case of $\mathit{true} \in [\![b]\!]\,\delta$, $\delta$ satisfies the precondition of the premise. Hence, for all $\delta' \in \mathbf{S}\delta$, $\delta'$ is not $\bot$ and also satisfies $I$. The case of $[\![b]\!]\,\delta = \{\mathit{false}\}$ is not of interest.

Recall $C^{n+1}_{b,S}(\sigma) :\equiv C^n_{b,S}(\sigma) \cup \bigcup_{\delta \in B^n_{b,S}(\sigma)} \bigcup_{\substack{l \in [\![b]\!]\sigma \\ \delta' \in \mathbf{S}\delta}} \begin{cases} \{\delta\} & \text{if } l = \mathit{false} \\ \emptyset & \text{if } l = \mathit{true} \wedge \delta' \neq \bot \\ \{\bot\} & \text{otherwise.} \end{cases}$.

Since all $\gamma \in C^n_{b,S}(\sigma)$ satisfies $I$ and $(\!|b|\!)(\mathit{false})$, we only need to care the rightmost part of the construction. Since all $\delta \in B^n_{b,S}(\sigma)$ satisfies $I$, by the side-condition, *unknown* and $\bot$ are not in $[\![b]\!]\,\delta$. The $\delta$ is added to $C^{n+1}(\sigma)$ only if $\mathit{false} \in [\![b]\!]\,\delta$. Therefore, $\delta$ satisfies both $I$ and $(\!|b|\!)(\mathit{false})$.

Also, in the case of $\mathit{true} \in [\![b]\!]\,\delta$, since $\delta$ satisfies the precondition in the premise, $\bot \notin \mathbf{S}\delta$. Therefore, $\bot \notin C^{n+1}_{b,S}$. $\square$