ERC is a programming language which abstracts exact real computation languages such as `iRRAM`, `AERN2`, $\cdots$. The purpose of the supplement is to formally construct ERC language and its verification rules thus that it can be implemented.

1 Syntax

ERC is an imperative language that has distinction between statements and terms. Data types of ERC are defined as follow:

$$\tau := \mathbf{R} \mid \mathbf{Z} \mid \mathbf{R}(n) \mid \mathbf{Z}(n)$$

$\mathbf{R}$ ($\mathbf{R}(n)$) represents real numbers (arrays of size n) and $\mathbf{Z}$ ($\mathbf{Z}(n)$) represents integer numbers (arrays of size n).

ERC supports modularization by introducing a set of special function symbols. It can be understood as a predefined set of ERC programs whose properties are known. We write $\mathcal{G}$ to be a set of integer valued multifunction symbols and $\mathcal{F}$ to be a set of real valued function symbols.

[I am confused: $\mathcal{F}$ is first a set of functions, then a function from string (?) to a list (?) of data types, then takes f (which is not a string) as argument!] [I want $\mathcal{F}$ to store information of assumed functions including its domain instead of introducing another symbol which stores the information of the assumed functions]

$\mathcal{G}$ ($\mathcal{F}$) is thought as a function from string to a list of data types. For example, if $\mathcal{F}(f) = \mathbf{R} :: \mathbf{R} :: \mathbf{R} :: \texttt{nil}$, it means that f is a predefined function symbol from $\mathbf{R} \times \mathbf{R} \times \mathbf{R}$ to $\mathbf{Z}$. The set $\mathcal{G}$ and $\mathcal{F}$ cannot be modified inside of any ERC program, hence it can be thought as constants. Function symbols stored in $\mathcal{G}$ and $\mathcal{F}$ not being duplicated is assumed.

Terms are not strictly separated by their types: we do not have **Rterm** and **Zterm** distinction; hence, both real addition and integer addition share their operator symbol $+$. The aim is to design ERC not to have any ambiguity in its type inference; given any context, it should be possible to decide a term's type. Type checking is introduced in the next section.

Though a term itself does not come with its type, in order to make it easier to be understood, we write z, z_i to denote terms which should be typed $\mathbf{Z}$, and x, y, x_i to denote terms which should be typed $\mathbf{R}$. Terms are defined inductively as follow:

$T, t, z, z_i, x, y, x_i :=$

$| \cdots -1, 0, 1 \cdots$ integer constants $| \cdots -1.0, 0.0, 1.0 \cdots$ real constants

$| v$ variable $| T[z]$ array access $| x > y$ real comparison

$| f(t)$ function application $| t_1 + t_2$ addition $| -t$ additive inversion

$| x * y$ multiplication $| /x$ multiplicative inversion $| \max(t_1, t_2)$ maximum

$| \neg z_1$ boolean negation $| z_1 \wedge z_2$ boolean conjunction

$| z_1 \vee z_2$ boolean disjunction $|$ select(z_0, z_1) multivalued select

$| z \ ? \ x : y$ conditional $| \iota(z)$ precision embedding

While a term represents values of a certain type, statements provide means of computation. Statements in ERC are constructed as follow:

$S, S_1, S_2 :=$

$| \epsilon$ Skip

$| v := t$ Variable Assignment

$| T[z] := t$ Array Assignment

$|$ **newvar** $v := t$ New Variable

$| S_1; \ S_2$ Sequence

$|$ **if** z **then** S_1 **else** S_2 Branching

$|$ **while** z **do** S Loop

Having data types, terms and statements defined, we can finally define what a program in ERC is:

$\boldsymbol{p} :=$

input $v_1 : \tau_1, v_2 : \tau_, \cdots, v_n : \tau_n$

S

return t

2 Type Checking

2.1 Type of Terms

Context is a mapping from a set of variables to their corresponding types; e.g., $\Gamma(x) = \mathbf{R}$. Well–typedness of a term t to τ under a context Γ is written as $\Gamma \vdash t : \tau$. The below shows ERC's type inference rules. Type checking, which is a function that tells whether a term t is well–typed and, if so, what type it has, under a context Γ is well–defined and computable.

$$\frac{}{\Gamma \vdash c : \mathbf{Z}} \qquad \frac{}{\Gamma \vdash c.0 : \mathbf{R}} \qquad \frac{v \in \mathrm{dom}(\Gamma) \quad \Gamma(v) = \tau}{\Gamma \vdash v : \tau}$$

$$\frac{\Gamma \vdash z : \mathbf{Z} \quad T \in \mathrm{dom}(\Gamma) \quad \Gamma(T) = \tau(n)}{\Gamma \vdash T[z] : \tau} \qquad \frac{\Gamma \vdash x, y : \mathbf{R}}{\Gamma \vdash x > y : \mathbf{Z}}$$

$$\frac{f \in \mathrm{dom}(G) \quad \Gamma \vdash t_i : \pi_i(G(f))}{\Gamma \vdash f(t_1, \cdots, t_n) : \mathbf{Z}} \qquad \frac{f \in \mathrm{dom}(F) \quad \Gamma \vdash t_i : \pi_i(F(f))}{\Gamma \vdash f(t_1, \cdots, t_n) : \mathbf{R}}$$

$$\frac{\tau := \mathbf{R} \text{ or } \mathbf{Z} \quad \Gamma \vdash t_1, t_2 : \tau}{\Gamma \vdash t_1 + t_2 : \tau} \qquad \frac{\tau := \mathbf{R} \text{ or } \mathbf{Z} \quad \Gamma \vdash t : \tau}{\Gamma \vdash -t : \tau} \qquad \frac{\Gamma \vdash x, y : \mathbf{R}}{\Gamma \vdash x * y : \mathbf{R}}$$

$$\frac{\Gamma \vdash x : \mathbf{R}}{\Gamma \vdash /x : \mathbf{R}} \qquad \frac{\tau := \mathbf{R} \text{ or } \mathbf{Z} \quad \Gamma \vdash t_1, t_2 : \tau}{\Gamma \vdash \max(t_1, t_2) : \tau} \qquad \frac{\Gamma \vdash z : \mathbf{Z}}{\Gamma \vdash \neg z : \mathbf{Z}} \qquad \frac{\Gamma \vdash z_1, z_2 : \mathbf{Z}}{\Gamma \vdash z_1 \wedge z_2 : \mathbf{Z}}$$

$$\frac{\Gamma \vdash z_1, z_2 : \mathbf{Z}}{\Gamma \vdash z_1 \vee z_2 : \mathbf{Z}} \qquad \frac{\Gamma \vdash z_1, z_2 : \mathbf{Z}}{\Gamma \vdash \mathrm{select}(z_0, z_1) : \mathbf{Z}} \qquad \frac{\Gamma \vdash z : \mathbf{Z} \quad \Gamma \vdash x, y : \mathbf{R}}{\Gamma \vdash (z \ ? \ x : y) : \mathbf{R}}$$

$$\frac{\Gamma \vdash z : \mathbf{Z}}{\Gamma \vdash \iota(z) : \mathbf{R}}$$

2.2 Well-typed Statements

Unlike terms, a statement in ERC may modify contexts. Let us denote a statement S under a context Γ being well–typed and yielding a new context Γ' as follow:

$$\Gamma \vdash S \triangleright \Gamma'$$

Well–typedness of a statement is defined with the inference rules as follow:

$$\frac{}{\Gamma \vdash \epsilon \triangleright \Gamma} \qquad \frac{\Gamma \vdash t : \tau \quad \Gamma(v) = \tau}{\Gamma \vdash v := t \triangleright \Gamma}$$

$$\frac{\Gamma \vdash z : \mathbf{Z} \quad \Gamma \vdash t : \tau \quad \Gamma(T) = \tau(n) \quad \tau = \mathbf{Z} \text{ or } \mathbf{R}}{\Gamma \vdash T[z] := t \triangleright \Gamma}$$

$$\frac{v \notin \mathrm{dom}(\Gamma) \quad \Gamma \vdash t : \tau}{\Gamma \vdash \textbf{newvar } v := t \ \triangleright \Gamma \cup t(v \mapsto \tau)} \qquad \frac{\Gamma \vdash S_1 \triangleright \Gamma_1 \quad \Gamma_1 \vdash S_2 \triangleright \Gamma_2}{\Gamma \vdash S_1; S_2 \triangleright \Gamma_2}$$

$$\frac{\Gamma \vdash z : \mathbf{Z} \quad \Gamma \vdash S_1 \triangleright \Gamma \quad \Gamma \vdash S_2 \triangleright \Gamma}{\Gamma \vdash \textbf{if } z \textbf{ then } S_1 \textbf{ else } S_2 \triangleright \Gamma} \qquad \frac{\Gamma \vdash z : \mathbf{Z} \quad \Gamma \vdash S \triangleright \Gamma}{\Gamma \vdash \textbf{while } z \textbf{ do } S \triangleright \Gamma}$$

Note that a new variable cannot be declared inside of a branching or a loop. Showing the type checking of a statement being well–defined and computable can be done by directly constructing it, using the recursion above.

2.3 Type of Program

An ERC program

$$\begin{aligned} \boldsymbol{p} := & \\ & \textbf{input } v_1 : \tau_1, v_2 : \tau, \cdots, v_n : \tau_n \\ & S \\ & \textbf{return } t \end{aligned}$$

is well–typed if,

$$\Gamma_0 \vdash S \triangleright \Gamma \quad \text{and} \quad \Gamma \vdash t : \tau$$

where $\Gamma_0 := \cup_i (v_i \mapsto \tau_i)$. We say that the ERC program $\boldsymbol{p}$ is a function from $\tau_1 \times \cdots \times \tau_n$ to τ.

3 Denotational Semantics

Well–typed terms, statements and programs have semantics which are mathematical meanings of the objects in the programming language.

Semantics of data types are $[\![\mathbf{R}]\!] = \mathbb{R}$ a set of real numbers, $[\![\mathbf{R}(n)]\!] = \mathbb{R}^n$ a set of real vectors of dimension n, $[\![\mathbf{Z}]\!] = \mathbb{Z}$ a set of integers and $[\![\mathbf{Z}(n)]\!] = \mathbb{Z}^n$ a set of integer vectors of dimension n,

Semantic of a context is a set of assignments of its defined variables into their values in proper types; for example, if $\Gamma = x \mapsto \mathbf{R}$, then $[\![\Gamma]\!] := \{x \mapsto w : w \in \mathbb{R}\}$. An element $\sigma \in [\![\Gamma]\!]$ of the semantic of a context is called state, which is a specific assignment of variables defined in Γ.

We use the Powerdomain discovered by Plotkin 1976 to a space of our semantics [**?**]: For any set A, $A_\perp$ is a poset where $\perp \sqsubseteq a$ for all $a \in A$ and any distinct elements of A not comparable. For any set A, we define $\mathcal{P}(A_\perp)$ a set of nonempty subsets of A with extra condition that for any infinite $B \in \mathcal{P}(A_\perp)$, $\perp \in B$. We say a member of $\mathcal{P}(A_\perp)$ is proper if it does not contain $\perp$. Egli–Milner ordering gives order in $\mathcal{P}(A_\perp)$ such that $p \sqsubseteq q$ if $\perp \in p$ and $p \subseteq q \cup \{\perp\}$, otherwise $p = q$; the ordering makes $\mathcal{P}(A_\perp)$ a domain .

3.1 Semantic of Terms

Considering the multivalue concept in ERC, a term's meaning under a state is a subset of a certain set; e.g., for a well–typed term $\Gamma \vdash t : \mathbf{Z}$, its semantic under a state σ is a subset of integers; semantic of a well–typed term is a function of the following type:

$$[\![\Gamma \vdash t : \tau]\!] : [\![\Gamma]\!] \to \mathcal{P}([\![\tau]\!]_\perp)$$

As is mentioned, semantic only is defined to well–typed terms. However, to ease describing, we often omit Γ, τ and simply write $[\![t]\!]$ instead of $[\![\Gamma \vdash t : \tau]\!]$. The semantic of well–typed terms is defined as follow:

$$[\![\Gamma \vdash c : \mathbf{Z}]\!]\sigma = \{c\}$$
$$[\![\Gamma \vdash c.0 : \mathbf{R}]\!]\sigma = \{c\}$$
$$[\![\Gamma \vdash v : \tau]\!]\sigma = \{\sigma(v)\}$$
$$[\![\Gamma \vdash T[z] : \tau]\!]\sigma = \bigcup_{n \in [\![z]\!]\sigma} \begin{cases} \{\pi_n(\sigma(T))\} & \text{if } 0 \le n < \dim(\Gamma(T)) \\ \{\bot\} & \text{else} \end{cases}$$
$$[\![\Gamma \vdash x > y : \mathbf{Z}]\!]\sigma = \bigcup_{x' \in [\![x]\!]\sigma \ y' \in [\![y]\!]\sigma} \begin{cases} \{1\} & \text{if } x' > y' \\ \{0\} & \text{if } x' < y' \\ \{\bot\} & \text{if } x' = y' \text{ or } x = \bot \text{ or } y = \bot \end{cases}$$
$$[\![\Gamma \vdash f(t_1, \cdots, t_n) : \mathbf{Z}]\!]\sigma = \bigcup_{w_i \in [\![t_i]\!]\sigma} \begin{cases} f(w_1, \cdots, w_n) & \text{if } w_i \neq \bot \text{ and } (w_1, \cdots, w_n) \in \mathrm{dom}(f) \\ \{\bot\} & \text{else} \end{cases}$$
$$[\![\Gamma \vdash f(t_1, \cdots, t_n) : \mathbf{R}]\!]\sigma = \bigcup_{w_i \in [\![t_i]\!]\sigma} \begin{cases} \{f(w_1, \cdots, w_n)\} & \text{if } w_i \neq \bot \text{ and } (w_1, \cdots, w_n) \in \mathrm{dom}(f) \\ \{\bot\} & \text{else} \end{cases}$$
$$[\![\Gamma \vdash t_1 + t_2 : \tau]\!]\sigma = \bigcup_{w_1 \in [\![t_1]\!]\sigma \ w_2 \in [\![t_2]\!]\sigma} \{w_1 \tilde{+} w_2\}$$
$$[\![\Gamma \vdash -t : \tau]\!]\sigma = \bigcup_{w \in [\![t]\!]\sigma} \{\tilde{-} w\}$$
$$[\![\Gamma \vdash x * y : \mathbf{R}]\!]\sigma = \bigcup_{w_1 \in [\![t_1]\!]\sigma \ w_2 \in [\![t_2]\!]\sigma} \{w_1 \tilde{\times} w_2\}$$

$$[\![\Gamma \vdash /x : \mathbf{R}]\!]\sigma = \bigcup_{w \in [\![t]\!]\sigma} \begin{cases} \{1/w\} & \text{if } w \neq \bot,\ w \neq 0 \\ \{\bot\} & \text{else} \end{cases}$$

$$[\![\Gamma \vdash \max(t_1, t_2) : \tau]\!]\sigma = \bigcup_{w_1 \in [\![t_1]\!]\sigma\ w_2 \in [\![t_2]\!]\sigma} \{\tilde{\max}(w_1, w_2)\}$$

$$[\![\Gamma \vdash \neg z : \mathbf{Z}]\!]\sigma = \bigcup_{w \in [\![z]\!]\sigma} \begin{cases} \{1\} & \text{if } w = 1 \\ \{0\} & \text{if } w \neq 0\ w \neq \bot \\ \{\bot\} & \text{else} \end{cases}$$

$$[\![\Gamma \vdash z_1 \wedge z_2 : \mathbf{Z}]\!]\sigma = \bigcup_{w_1 \in [\![z_1]\!]\sigma\ w_2 \in [\![z_2]\!]\sigma} \begin{cases} \{1\} & \text{if } w_1, w_2 = 1 \\ \{0\} & \text{if } w_1 = 0 \text{ or } w_2 = 0 \\ \{\bot\} & \text{else} \end{cases}$$

$$[\![\Gamma \vdash z_1 \vee z_2 : \mathbf{Z}]\!]\sigma = \bigcup_{w_1 \in [\![z_1]\!]\sigma\ w_2 \in [\![z_2]\!]\sigma} \begin{cases} \{1\} & \text{if } w_1 = 1 \text{ or } w_2 = 1 \\ \{0\} & \text{if } w_1, w_2 = 0 \\ \{\bot\} & \text{else} \end{cases}$$

$$[\![\Gamma \vdash \text{select}(z_0, z_1) : \mathbf{Z}]\!]\sigma = \bigcup_{b_0 \in [\![z_0]\!]\sigma\ b_1 \in [\![z_1]\!]\sigma} \begin{cases} \{0\} & \text{if } b_0 \neq \bot \\ \{1\} & \text{if } b_1 \neq \bot \\ \{\bot\} & \text{if } b_0 = b_1 = \bot \end{cases}$$

$$[\![\Gamma \vdash (z\ ?\ x : y) : \mathbf{R}]\!]\sigma = \bigcup_{b \in [\![z]\!]\sigma} \begin{cases} [\![x]\!]\sigma & \text{if } b \neq 0 \wedge b \neq \bot \\ [\![y]\!]\sigma & \text{if } b = 0 \\ [\![x]\!]\sigma \odot [\![y]\!]\sigma & \text{else.} \end{cases}$$

$$[\![\Gamma \vdash \iota(z) : \mathbf{R}]\!]\sigma = \bigcup_{w \in [\![z]\!]\sigma} \begin{cases} \{2^w\} & \text{if } w \neq \bot \\ \{\bot\} & \text{else} \end{cases}$$

For an operation *op*, we write $\tilde{op}$ which extends the co/–domain of *op* so that it returns $\bot$ when at least one of its arguments turn out to be $\bot$; otherwise, it remains the same. For two sets U, V the operation $\odot$ on those is defined as follow:

$$U \odot V := \begin{cases} U & \text{if } \bot \notin U = V\ \ |U| = 1 \\ \{\bot\} & \text{else.} \end{cases}$$

3.2 Semantic of Statements

Semantic of statements is a state transformer. Considering multivaluedness in ERC, we let the semantic of a well–typed statement to be a function from the set of states to the restricted powerset of the resulting states:

$$[\![\Gamma \vdash S \triangleright \Gamma']\!] : [\![\Gamma]\!] \to \mathcal{P}([\![\Gamma']\!]_\bot)$$

Semantics of the statements except for the while loop are defined as follow:

$$[\![\Gamma \vdash \epsilon \triangleright \Gamma]\!]\sigma = \{\sigma\}$$

$$[\![\Gamma \vdash v := t \triangleright \Gamma]\!]\sigma = \bigcup_{w \in [\![t]\!]\sigma} \begin{cases} \{\sigma[v \mapsto w]\} & \text{if } w \neq \bot \\ \{\bot\} & \text{else} \end{cases}$$

$$[\![\Gamma \vdash T[z] := t \triangleright \Gamma]\!]\sigma = \bigcup_{n \in [\![z]\!]\sigma \ w \in [\![t]\!]\sigma} \begin{cases} \{\sigma[T \twoheadrightarrow_n w]\} & \text{if } 0 \leq n < d \text{ and } w \neq \bot \\ \{\bot\} & \text{else.} \end{cases}$$

$$[\![\Gamma \vdash \textbf{newvar } v := t \ \triangleright \Gamma']\!]\sigma = \bigcup_{w \in [\![t]\!]\sigma} \begin{cases} \{\sigma \cup (v \mapsto w)\} & \text{if } w \neq \bot \\ \{\bot\} & \text{else} \end{cases}$$

$$[\![\Gamma \vdash S_1; S_2 \triangleright \Gamma']\!]\sigma = \bigcup_{\delta \in [\![S_1]\!]\sigma} \begin{cases} [\![S_2]\!]\delta & \text{if } \delta \neq \bot \\ \{\bot\} & \text{else} \end{cases}$$

$$[\![\Gamma \vdash \textbf{if } z \textbf{ then } S_1 \textbf{ else } S_2 \triangleright \Gamma]\!]\sigma = \bigcup_{b \in [\![z]\!]\sigma} \begin{cases} [\![S_1]\!]\delta & \text{if } b \neq \bot,\ b \neq 0 \\ [\![S_2]\!]\delta & \text{if } b = 0 \\ \{\bot\} & \text{if } b = \bot \end{cases}$$

$\sigma[T \twoheadrightarrow_n w]$ is to substitute the n'th element of T in the state σ if n is in proper range of T's dimension, otherwise be $\bot$.

Defining the semantic of the while loop is a little subtle. However, luckily the domain that we are working on, $\mathcal{P}([\![\Gamma]\!]_\bot)$, is the famous powerdomain for bounded nondeterminism. Therefore, equipping point–wise ordering, with using the fixed point theorem, we can define the semantic of while loop; consider the recursive semantic equation of the while loop:

$$\begin{aligned} [\![\textbf{while } z \textbf{ do } S]\!]\sigma &= [\![\textbf{if } z \textbf{ then } S; (\textbf{while } z \textbf{ do } S) \textbf{ else } \epsilon]\!]\sigma \\ &= \bigcup_{b \in [\![z]\!]\sigma} \begin{cases} [\![S; (\textbf{while } z \textbf{ do } S)]\!]\sigma \\ \{\sigma\} \\ \{\bot\} \end{cases} \\ &= \bigcup_{b \in [\![z]\!]\sigma \ \delta \in [\![S]\!]\sigma} \begin{cases} [\![\textbf{while } z \textbf{ do } S]\!]\delta & \text{if } \delta \neq \bot\ b \neq \bot\ b \neq 0 \\ \{\sigma\} & \text{if } b = 0 \\ \{\bot\} & \text{else} \end{cases} \end{aligned}$$

Hence, we can define the operator of type $([\![\Gamma]\!] \to \mathcal{P}([\![\Gamma]\!]_\bot)) \to ([\![\Gamma]\!] \to \mathcal{P}([\![\Gamma]\!]_\bot))$ as follow:

$$\mathcal{F}_{z,S}(f) = \lambda\sigma. \bigcup_{b \in [\![z]\!]\sigma \ \delta \in [\![S]\!]\sigma} \begin{cases} f(\delta) & \text{if } \delta,\ b \neq \bot \text{ and } b \neq 0 \\ \{\sigma\} & \text{if } b = 0 \\ \{\bot\} & \text{else} \end{cases}$$

Proof of monotonicity and continuity of $\mathcal{F}_{z,S}$ will follow the proof used to define semantic of bounded nondeterminism in Dijkstra's guarded command [**?**]; (see its proof in the classic textbook of J. Reynold [**?**].)

The semantic of the while loop can be defined as the least fixed point of the operator:

$$[\![\textbf{while } z \textbf{ do } S]\!]\sigma = \mathcal{F}^{\infty}_{z,S}(\perp)$$

Or, consider the chain:

$$\omega_0 := \lambda\sigma.\ \perp \sqsubseteq \omega_1 \cdots \sqsubseteq \omega_{i+1} := \mathcal{F}_{z,S}(\omega_i) \sqsubseteq \cdots$$

where $[\![\textbf{while } z \textbf{ do } S]\!]\sigma = \bigsqcup_{\infty} \omega_i\sigma$.

3.3 Semantic of Programs

Having semantics of the statements defined, we can define semantics of the ERC programs. Recall that a ERC program is constructed with the following format:

$\boldsymbol{p} :=$

input $v_1 : \tau_1, v_2 : \tau, \cdots, v_n : \tau_n$

S

return t

A well–typed program $\boldsymbol{p}$ guarantees the following well–typedness: $\Gamma_0 \vdash S \triangleright \Gamma$ and $\Gamma \vdash t : \tau$ where $\Gamma_0 := \cup_i (v_i \mapsto \tau_i)$ with some τ. The semantic of the program $\boldsymbol{p}$ is a set–valued function

$$[\![\boldsymbol{p}]\!] : [\![\Gamma]\!] \to \mathcal{P}([\![\tau]\!]_{\perp})$$

such that

$$[\![\boldsymbol{p}]\!] := \lambda(x_1, \cdots, x_n).\ \{\sigma : \sigma \in [\![t]\!]\delta \wedge \delta \in [\![S]\!] \cup_i (v_i \mapsto x_i)\}$$

We say the program $[\![\boldsymbol{p}]\!]$ is total if does not contain $\perp$ for its any input values. We say the program $[\![\boldsymbol{p}]\!]$ is single–valued if for any of its input values, the output contains $\perp$ or is singleton.

4 Verification

Assertion is a logical predicate on a set of assignments $[\![\Gamma]\!]$. A language used to define assertions is called assertion language. In ERC, we let the first–order language on the two sorted structure (see Theorem **??**) to be the assertion language.

We let the function symbols to accompany with a finite list of tuples of predicates; for an example, if there is a function symbol $f : \tau_1 \times \cdots \times \tau_d \to \tau$ in either $\mathcal{F}$ or $\mathcal{G}$ loaded to the language, there exists $\{(\phi(f)_i, \psi(f)_i)\}$ where $\phi(f)_i$ is a predicate over $[\![\tau_1]\!] \times \cdots \times [\![\tau_d]\!]$ and $\psi(f)_i$ is a predicate defined over $[\![\tau]\!] \times [\![\tau_1]\!] \times \cdots \times [\![\tau_d]\!]$. The set of predicates defines [No!] the domain of the function and the function's properties on the domain:

$$f(x_1, \cdots, x_d) \downarrow v \leftarrow \exists k.\ \phi(f)_k(x_1, \cdots, x_d) \wedge \forall k.\ \phi(f)_k(x_1, \cdots, x_d) \to \psi(v, x_1, \cdots, x_d)$$

[(How) is v quantified over?] [v is the possible values of the function call; hence, should be universally quantified in the previous sentence: for all v x, $f(x)$ is defined to be v if and only if one of the domain predicates holds on x and x, v satisfies $\phi(x) \to \psi(v, x)$. If f were a multifunction we could say, instead, $f(x)$ is proper (doesn't contain $\perp$) and contains v, if and only if $\cdots$. I think you don't like 'if and only if' part; since specification of a program is usually looser than what the program really does. But I think without referring one concrete definition (or specification) of a function, we cannot prove anything: if the provided specification says f is defined when $x > 0$, then the prover should think $f(x) = \perp$ when $x \leq 0$. When the provided specification says $x > 0 \to v < 0$, then the prover should think $f(x)$ to be any value less than 0 but cannot be any nonnegative value.] The requirement is essential especially since we do not have a multivalued object in our assertion language. Consider we have a symbol representing an integer valued multi–function g. However, we cannot write $g(x)$ in our assertion language. The purpose is to write $\{\omega : P(\omega)\}$ instead, where P would be some predicate in our language. Whereas, for real valued single function, we leave the function symbol f in our assertion language. $\phi(f)_i$ will be used to check its domain and $\psi(f)_i$ will be loaded to the theory of our assertion system.

For examples, suppose we have a function symbol $\texttt{soft} : \mathbf{Z} \times \mathbf{R} \times \mathbf{R} \to \mathbf{Z}$ in $\mathcal{F}$. Then, there will be pairs of predicates $(x < y + 2^p, \texttt{soft}(p, x, y) = 1)$ and $(y < x + 2^p, \texttt{soft}(p, x, y) = 0)$. In our assertion language, when we wish express the values of the term $\texttt{soft}(x, y, z)$, we cannot refer to $\texttt{soft}$; instead, we write $\lambda\omega.\ (x < y + 2^p \vee y < x + 2^p) \wedge x < y + 2^p \to \omega = 1 \wedge y < x + 2^p \to \omega = 0$

On the other hand, consider we have a function symbol $\texttt{sqrt} : \mathbf{R} \to \mathbf{R}$ in $\mathcal{G}$. Then, there will be a pair of predicates $(x > 0, \texttt{sqrt}(x) \times \texttt{sqrt}(x) = x)$. In this case, since $\texttt{sqrt}$ is only a single–valued [No!] [But $\mathcal{F}$ only includes single–valued function? also, I want to keep function symbols as long as they're single–valued; because, then, we can say the Trisection program computes a root of the f] function, we keep the symbol $\texttt{sqrt}$ in our system with $x > 0 \to \texttt{sqrt}(x) \times \texttt{sqrt}(x) = x$ being an element of our theory.

4.1 Translation Function

In a simple language, it is trivial how to translate a programming term into an assertion language; for an example, consider a programming term $x > y$ where x, y are variables. Then, the corresponding predicate which defines the set of states which yields the evaluation (semantic) of the term to be a boolean value $\texttt{true}$ would be $x > y$.

However, in ERC, the semantic of terms is quite subtle. Since a term t can be multivalued itself, the direct translation does not work: Its semantics is a *set*. Now we do not have a power object in the assertion language. Instead we identify the set of values with the properties of its elements, expressed as in first-order logic as predicate over the two-sorted structure of integers and real numbers. We construct a translation function which translates a well–typed term into a predicate in our assertion language which exactly defines the values

of the term's semantic.

We introduce two translation functions that are constructed simultaneously. $\mathcal{H}(t)(k)\sigma$ defines those values in $[\![t]\!]\sigma$ except for $\perp$ and $\mathcal{T}(t)(k)\sigma$ defines those values in a proper $[\![t]\!]\sigma$ only; otherwise $\{v : \mathcal{T}(t)(k)\sigma\}$. Hence, only $\mathcal{T}(\cdot)$ function will be used in the assertion language. However, $\mathcal{H}(\cdot)$ is required to construct the function: think it as a helper function.

Translation function is also only for well–typed terms. However, to ease writing, we write $\mathcal{T}(t)$ instead of $\mathcal{T}(\Gamma \vdash t : \tau)$; when Γ or τ appears, it refers to those that are omitted. To be precise, $\mathcal{T}(\Gamma \vdash t : \tau)$ is a predicate on $[\![t]\!] \times [\![\Gamma]\!]$. For example, $\mathcal{T}(\Gamma \vdash x : \mathbf{R}) = \lambda r.\lambda\sigma.\ r = \sigma(x)$. However, to make the description simpler, we leave $\sigma(\cdot)$ implicit. Hence, we will write $\mathcal{T}(\Gamma \vdash x : \mathbf{R}) = \lambda r.\ r = x$ instead. (Same for $\mathcal{H}(\cdot)$)

$$
\begin{aligned}
&\mathcal{H}(c) = \lambda\omega.\ \omega = c \\
&\mathcal{H}(c.0) = \lambda\omega.\ \omega = c \\
&\mathcal{H}(v) = \lambda\omega.\ \omega = v \\
&\mathcal{H}(T[z]) = \lambda\omega.\ \exists n.\ \omega = \pi_n(T) \\
&\mathcal{H}(x > y) = \\
&\quad \lambda\omega. \\
&\qquad \omega = 1 \wedge \exists\omega_1\ \omega_2.\ \mathcal{H}(x)(\omega_1) \wedge \mathcal{H}(y)(\omega_2) \wedge \omega_1 > \omega_2 \\
&\qquad \vee\ \omega = 0 \wedge \exists\omega_1\ \omega_2.\ \mathcal{H}(x)(\omega_1) \wedge \mathcal{H}(y)(\omega_2) \wedge \omega_2 < \omega_1 \\
&\mathcal{H}(f(t_1, \cdots, t_n) : \mathbf{R}) = \\
&\quad \lambda\omega. \\
&\qquad \exists\omega_1 \cdots \omega_n.\ \omega = f(\omega_1, \cdots, \omega_n) \wedge (\wedge_i \mathcal{H}(t_i)(\omega_i)) \wedge \vee_i (\phi(f)_i(\omega_1, \cdots, \omega_n)) \\
&\mathcal{H}(f(t_1, \cdots, t_n) : \mathbf{Z}) = \\
&\quad \lambda\omega. \\
&\qquad \exists\omega_1 \cdots \omega_n.\ (\wedge_i \mathcal{H}(t_i)(\omega_i)) \wedge (\wedge_i (\phi(f)_i \to \psi(f)_i)(\omega, \omega_1, \cdots, \omega_n)) \wedge \vee_i \phi(f)_i(\omega_1, \cdots, \omega_n) \\
&\mathcal{H}(t_1 + t_2) = \lambda\omega.\ \exists\omega_1\ \omega_2.\ \omega = \omega_1 + \omega_2 \wedge \mathcal{H}(t_1)(\omega_1) \wedge \mathcal{H}(t_2)(\omega_2) \\
&\mathcal{H}(-t) = \lambda\omega.\ \exists\omega'.\ \omega = -\omega' \wedge \mathcal{H}(t)(\omega') \\
&\mathcal{H}(x * y) = \lambda\omega.\ \exists\omega_1\ \omega_2.\ \omega = \omega_1 \times \omega_2 \wedge \mathcal{H}(t_1)(\omega_1) \wedge \mathcal{H}(t_2)(\omega_2) \\
&\mathcal{H}(/x) = \lambda\omega.\ \exists\omega' \neq 0.\ \omega \times \omega' = 1 \wedge \mathcal{H}(x)(\omega') \\
&\mathcal{H}(\max(t_1, t_2)) = \lambda\omega.\ \exists\omega_1\ \omega_2.\ \omega = \max(\omega_1, \omega_2) \wedge \mathcal{H}(t_1)(\omega_1) \wedge \mathcal{H}(t_2)(\omega_2) \\
&\mathcal{H}(\neg z) = \lambda\omega.\ (\omega = 1 \wedge \mathcal{H}(z)(0)) \vee (\omega = 0 \wedge \exists\omega' \neq 0.\ \mathcal{H}(z)(\omega')) \\
&\mathcal{H}(z_1 \wedge z_2) = \\
&\quad \lambda\omega. \\
&\qquad \omega = 0 \wedge (\mathcal{H}(z_1)(0) \vee \mathcal{H}(z_2)(0)) \\
&\qquad \vee\ \omega = 1 \wedge \exists\omega_1\ \omega_2.\ (\omega_1, \omega_2 \neq 0 \wedge \mathcal{H}(z_1)(\omega_1) \wedge \mathcal{H}(z_2)(\omega_2)) \\
&\mathcal{H}(z_1 \vee z_2) = \\
&\quad \lambda\omega. \\
&\qquad \omega = 1 \wedge \exists\omega_1\ \omega_2 \neq 0.\ \mathcal{H}(z_1)(\omega_1) \vee \mathcal{H}(z_2)(\omega_2)) \\
&\qquad \vee\ \omega = 0 \wedge (\mathcal{H}(z_1)(0) \wedge \mathcal{H}(z_2)(0)) \\
&\mathcal{H}(\text{select}(z_0, z_1)) = \lambda\omega.\ (\omega = 0 \wedge \exists\omega_0.\ \mathcal{H}(z_0)(\omega_0)) \vee (\omega = 1 \wedge \exists\omega_1.\ \mathcal{H}(z_1)(\omega_1)) \\
&\mathcal{H}(z\ ?\ x : y) = \\
&\quad \lambda\omega. \\
&\qquad \exists\omega_1 \neq 0\ \omega_2.\ \omega = \omega_2 \wedge \mathcal{H}(z)(\omega_1) \wedge \mathcal{H}(x)(\omega_2) \\
&\qquad \vee\ \exists\omega_3.\ \omega = \omega_3 \wedge \mathcal{H}(z)(0) \wedge \mathcal{H}(y)(\omega_3) \\
&\qquad \vee\ \neg(\exists\omega_1.\ \mathcal{T}(z)(\omega_1)) \wedge \exists\omega_1.\ \omega = \omega_1 \wedge \mathcal{T}(x)(\omega_1) \wedge \mathcal{T}(y)(\omega_1) \\
&\qquad\qquad\qquad\qquad \wedge \forall\omega_2.\ \mathcal{T}(x)(\omega_2) \vee \mathcal{T}(y)(\omega_2) \to \omega_1 = \omega_2
\end{aligned}
$$

$$\mathcal{H}(\iota(z)) = \lambda\omega.\ \exists\omega'.\ \omega = 2^{\omega'} \wedge \mathcal{H}(z)(\omega')$$

$$\mathcal{T}(c) = \lambda\omega.\ \omega = c$$
$$\mathcal{T}(c.0) = \lambda\omega.\ \omega = c$$
$$\mathcal{T}(v) = \lambda\omega.\ \omega = v$$
$$\mathcal{T}(T[z]) = \lambda\omega.\ (\exists n.\ \omega = \pi_n(T) \wedge \mathcal{T}(z)(n)) \wedge (\forall n.\ \mathcal{T}(z)(n) \to 0 \leq n < \dim(T))$$
$$\mathcal{T}(x > y) =$$
$$\lambda\omega.$$
$$\omega = 1 \wedge \exists\omega_1\ \omega_2.\ \mathcal{T}(x)(\omega_1) \wedge \mathcal{T}(y)(\omega_2) \wedge \omega_1 > \omega_2$$
$$\vee\ \omega = 0 \wedge \exists\omega_1\ \omega_2.\ \mathcal{T}(x)(\omega_1) \wedge \mathcal{T}(y)(\omega_2) \wedge \omega_2 < \omega_1$$
$$\wedge\ \forall\omega_1\ \omega_2.\ \mathcal{T}(x)(\omega_1) \wedge \mathcal{T}(y)(\omega_2) \to \omega_1 \neq \omega_2$$
$$\mathcal{T}(f(t_1, \cdots, t_n) : \mathbf{R}) =$$
$$\lambda\omega.$$
$$\exists\omega_1 \cdots \omega_n.\ \omega = f(\omega_1, \cdots, \omega_n) \wedge (\wedge_i \mathcal{T}(t_i)(\omega_i))$$
$$\wedge\ \forall\omega_1 \cdots \omega_n.\ (\wedge_i \mathcal{T}(t_i)(\omega_i)) \to \vee_i \phi(f)_i(\omega_1, \cdots, \omega_n)$$
$$\mathcal{T}(f(t_1, \cdots, t_n) : \mathbf{Z}) =$$
$$\lambda\omega.$$
$$\exists\omega_1 \cdots \omega_n.\ (\wedge_i \mathcal{T}(t_i)(\omega_i)) \wedge (\wedge_i (\phi(f)_i \to \psi(f)_i)(\omega, \omega_1, \cdots, \omega_n))$$
$$\wedge\ \forall\omega_1 \cdots \omega_n.\ (\wedge_i \mathcal{T}(t_i)(\omega_i)) \to \vee_i \phi(f)_i(\omega_1, \cdots, \omega_n)$$
$$\mathcal{T}(t_1 + t_2) = \lambda\omega.\ \exists\omega_1\ \omega_2.\ \omega = \omega_1 + \omega_2 \wedge \mathcal{T}(t_1)(\omega_1) \wedge \mathcal{T}(t_2)(\omega_2)$$
$$\mathcal{T}(-t) = \lambda\omega.\ \exists\omega'.\ \omega = -\omega' \wedge \mathcal{T}(t)(\omega')$$
$$\mathcal{T}(x * y) = \lambda\omega.\ \exists\omega_1\ \omega_2.\ \omega = \omega_1 \times \omega_2 \wedge \mathcal{T}(t_1)(\omega_1) \wedge \mathcal{T}(t_2)(\omega_2)$$
$$\mathcal{T}(/x) = \lambda\omega.\ \exists\omega' \neq 0.\ \omega \times \omega' = 1 \wedge \mathcal{T}(x)(\omega') \wedge \forall\omega'.\ \mathcal{T}(x)(\omega') \to \omega' \neq 0$$
$$\mathcal{T}(\max(t_1, t_2)) = \lambda\omega.\ \exists\omega_1\ \omega_2.\ \omega = \max(\omega_1, \omega_2) \wedge \mathcal{T}(t_1)(\omega_1) \wedge \mathcal{T}(t_2)(\omega_2)$$
$$\mathcal{T}(\neg z) = \lambda\omega.\ (\omega = 1 \wedge \mathcal{T}(z)(0)) \vee (\omega = 0 \wedge \exists\omega' \neq 0.\ \mathcal{T}(z)(\omega'))$$
$$\mathcal{T}(z_1 \wedge z_2) =$$
$$\lambda\omega.$$
$$\omega = 0 \wedge (\mathcal{T}(z_1)(0) \vee \mathcal{T}(z_2)(0))$$
$$\wedge \Big(\forall\omega_1.\ \mathcal{T}(z_1)(\omega_1) \to \omega_1 = 0 \wedge \forall\omega_2.\ \mathcal{T}(z_2)(\omega_2) \to \omega_2 = 0$$
$$\vee\ \exists\omega_1\ \omega_2.\ \mathcal{T}(z_1)(\omega_1) \wedge \mathcal{T}(z_2)(\omega_2)\Big)$$
$$\vee\ \omega = 1 \wedge \exists\omega_1\ \omega_2.\ (\omega_1, \omega_2 \neq 0 \wedge \mathcal{T}(z_1)(\omega_1) \wedge \mathcal{T}(z_2)(\omega_2))$$

$$
\begin{aligned}
&\mathcal{T}(z_1 \vee z_2) = \\
&\quad \lambda\omega. \\
&\qquad \omega = 1 \wedge \exists\omega_1\ \omega_2.\ (\omega_1, \omega_2 \neq 0 \wedge \mathcal{T}(z_1)(\omega_1) \vee \mathcal{T}(z_2)(\omega_2)) \\
&\qquad\qquad \wedge \Big(\forall\omega_1.\ \mathcal{T}(z_1)(\omega_1) \to \omega_1 \neq 0 \wedge \forall\omega_2.\ \mathcal{T}(z_2)(\omega_2) \to \omega_2 \neq 0 \\
&\qquad\qquad\qquad \vee \exists\omega_1\ \omega_2.\ \mathcal{T}(z_1)(\omega_1) \wedge \mathcal{T}(z_2)(\omega_2)\Big) \\
&\qquad \vee \omega = 0 \wedge (\mathcal{T}(z_1)(0) \wedge \mathcal{T}(z_2)(0)) \\
&\mathcal{T}(\mathrm{select}(z_0, z_1)) = \\
&\quad \lambda\omega. \\
&\qquad (\omega = 0 \wedge \exists\omega_0.\ \mathcal{H}(z_0)(\omega_0)) \vee (\omega = 1 \wedge \exists\omega_1.\ \mathcal{H}(z_1)(\omega_1)) \\
&\qquad \wedge \exists k.\ \mathcal{T}(z_0)(k) \vee \mathcal{T}(z_1)(k) \\
&\mathcal{T}(z\ ?\ x : y) = \\
&\quad \lambda\omega. \\
&\qquad \exists\omega_1 \neq 0.\ \mathcal{T}(z)(\omega_1) \to \exists\omega_2.\ \mathcal{T}(x)(\omega_2) \\
&\qquad \wedge \mathcal{T}(z)(0) \to \exists\omega_3.\ \mathcal{T}(y)(\omega_3) \\
&\qquad \wedge \quad \exists\omega_1 \neq 0\ \omega_2.\ \omega = \omega_2 \wedge \mathcal{T}(z)(\omega_1) \wedge \mathcal{T}(x)(\omega_2) \\
&\qquad\quad \vee \exists\omega_3.\ \omega = \omega_3 \wedge \mathcal{T}(z)(0) \wedge \mathcal{T}(y)(\omega_3) \\
&\qquad\quad \vee \neg(\exists\omega_1.\ \mathcal{T}(z)(\omega_1)) \wedge \exists\omega_1.\ \omega = \omega_1 \wedge \mathcal{T}(x)(\omega_1) \wedge \mathcal{T}(y)(\omega_1) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \wedge \forall\omega_2.\ \mathcal{T}(x)(\omega_2) \vee \mathcal{T}(y)(\omega_2) \to \omega_1 = \omega_2 \\
&\mathcal{T}(\iota(z)) = \lambda\omega.\ \exists\omega'.\ \omega = 2^{\omega'} \wedge \mathcal{T}(z)(\omega')
\end{aligned}
$$

Theorem 1. $\{v : \mathcal{H}(\Gamma \vdash t : \tau)(v)\sigma\} = [\![\Gamma \vdash t : \tau]\!]\sigma \setminus \{\bot\}$. *And,* $\{v : \mathcal{T}(\Gamma \vdash t : \tau)(v)\sigma\} = \varnothing$ *if and only if* $\bot \in [\![\Gamma \vdash t : \tau]\!]\sigma$. *If* $\{v : \mathcal{T}(\Gamma \vdash t : \tau)(v)\sigma\} \neq \varnothing$, *then it coincides with* $[\![\Gamma \vdash t : \tau]\!]\sigma$

Proof. It can be proved with induction on t. Skipping the trivial cases, let us see $\mathcal{H}(z\ ?\ x : y)$, $\mathcal{T}(x > y)$, and $\mathcal{T}(\mathrm{select}(z_0, z_2))$.

– [*conditional*] Induction hypothesis says $\mathcal{H}(z)(\omega_1) \leftrightarrow \omega_1 \in [\![z]\!]$, $\mathcal{H}(x)(\omega_2) \leftrightarrow \omega_2 \in [\![x]\!]$ and $\mathcal{H}(z)(\omega_3) \leftrightarrow \omega_3 \in [\![z]\!]$ for all $\omega_{1,2,3}$. See that $\mathcal{H}(z\ ?\ x : y)(\omega)$ if and only if either

1. $\exists\omega_1 \neq 0.\ \mathcal{H}(z)(\omega_1) \wedge \mathcal{H}(x)(\omega)$
2. $\mathcal{H}(z)(0) \wedge \mathcal{H}(y)(\omega)$
3. $\neg(\exists\omega'.\ \mathcal{T}(z)(\omega')) \wedge \mathcal{T}(x)(\omega) \wedge \mathcal{T}(y)(\omega) \wedge \forall\omega'.\ \mathcal{T}(x)(\omega') \vee \mathcal{T}(y)(\omega') \to \omega' = \omega$

Consider the last case: $\neg(\exists\omega'.\ \mathcal{T}(z)(\omega'))$ if and only if $\bot \in [\![z]\!]\sigma$. $\mathcal{T}(x)(\omega) \wedge \mathcal{T}(y)(\omega) \wedge \forall\omega'.\ \mathcal{T}(x)(\omega') \vee \mathcal{T}(y)(\omega') \to \omega' = \omega$ if and only if $[\![x]\!]\sigma = [\![y]\!]\sigma = \{\omega\}$; hence, $\omega \in [\![z\ ?\ x : y]\!]\sigma$. Hence the above three conditions are equivalent to:

1. $\exists\omega_1 \neq 0.\ \omega_1 \in [\![z]\!]\sigma \wedge \omega \in [\![x]\!]\sigma$
2. $0 \in [\![z]\!]\sigma \wedge \omega \in [\![y]\!]$

3. $\bot \in [\![z]\!]\sigma \wedge [\![x]\!]\sigma = [\![y]\!]\sigma = \{\omega\}$

Hence, $\mathcal{H}(z\ ?\ x : y)(\omega)$ if and only if $\omega \in [\![z\ ?\ x : y]\!]\sigma$

– [*real comparison*] See that $\bot \in [\![x > y]\!]\sigma$ if and only if either $\bot \in [\![x]\!]\sigma$ or $\bot \in [\![y]\!]\sigma$ or $\exists z.\ z \in [\![x]\!]\sigma \wedge z \in [\![y]\!]\sigma$. If either $\bot \in [\![x]\!]\sigma$ or $\bot \in [\![y]\!]\sigma$, by the induction hypothesis, $\{v : \mathcal{T}(x)(v)\} = \varnothing$ or $\{v : \mathcal{T}(y)(v)\} = \varnothing$. Either case $\{v : \mathcal{T}(x > y)(v)\} = \varnothing$. Suppose $\exists z.\ z \in [\![x]\!] \wedge z \in [\![y]\!]$. Then the last clause, $z \neq z$ cannot be satisfied. Hence, $\{v : \mathcal{T}(x > y)(v)\} = \varnothing$.

If $\{v : \mathcal{T}(x > y)(v)\} = \varnothing$, using tautological equivalence yields two cases: $\exists \omega_1\ \omega_2.\ \mathcal{T}(T)(x) \wedge \mathcal{T}(T)(y) \wedge x = y$ or $\forall v.\ \mathcal{T}(x)(v) \vee \mathcal{T}(y)(v)$. By the induction hypothesis, $\exists z.\ z \in [\![x]\!]\sigma \wedge z \in [\![y]\!]\sigma$ or $\bot \in [\![x]\!]\sigma \vee \bot \in [\![y]\!]\sigma$. Hence, $\bot \in [\![x > y]\!]\sigma$.

Suppose $\mathcal{T}(x > y)(0)$ Since $\mathcal{T}(x > y)(0) \to \exists \omega_1\ \omega_2.\ \mathcal{T}(x)(\omega_1) \wedge \mathcal{T}(y)\omega_2) \wedge \omega_2 < \omega_1$, by the induction hypothesis, we have $\exists \omega_1\ \omega_2.\ \in [\![x]\!]\sigma \wedge [\![y]\!]\sigma \wedge \omega_2 < \omega_1$. Hence, $1 \in [\![x > y]\!]\sigma$. The other case can be done similarly.

See that $1 \in [\![x > y]\!]\sigma$ if and only if $\exists x'\ y'.\ x' \in [\![x]\!]\sigma \wedge y' \in [\![y]\!]\sigma \wedge x' > y'$. Hence, by induction hypothesis, $\exists \omega_1\ \omega_2.\ \mathcal{T}(x)(\omega_1) \wedge \mathcal{T}(y)(\omega_2) \wedge \omega_1 > \omega_2$. Hence, $\mathcal{T}(x > y)(1)$ holds. The other case can be done similarly.

– [*select*] $\bot \in [\![\text{select}(z_0, z_1)]\!]\sigma$ if and only if $\bot \in [\![z_0]\!]\sigma \wedge \bot \in [\![z_1]\!]\sigma$. Then, by the induction hypothesis, $\{v : \mathcal{T}(z_0)(v)\} = \{v : \mathcal{T}(z_1)(v)\} = \varnothing$. Then, $\exists \omega'.\ \omega = 2^{\omega'} \wedge \mathcal{T}(z)(\omega')$ cannot be satisfied, hence, $\{v : \mathcal{T}(\text{select}(z_0, z_1))(v)\} = \varnothing$.

If $\{v : \mathcal{T}(\text{select}(z_0, z_1))(v)\} = \varnothing$, then $\{v : \mathcal{T}(z_0)(v)\} = \{v : \mathcal{T}(z_1)(v)\} = \varnothing$. Hence, $\bot \in [\![z_0]\!]\sigma \wedge \bot \in [\![z_1]\!]\sigma$ and $\bot \in [\![\text{select}(z_0, z_1)]\!]\sigma$.

$\bot \notin [\![\text{select}(z_0, z_1)]\!]\sigma$ and $1 \in [\![\text{select}(z_0, z_1)]\!]\sigma$ if and only if $\exists k.\ \mathcal{T}(z_0)(k) \vee \mathcal{T}(z_1)(k)$ and $[\![z_1]\!]\sigma \neq \{\bot\}$ includes some non bottom element. Hence, $\exists k.\ \mathcal{H}(z_1)(k)$; therefore, $\mathcal{T}(\text{select}(z_0, z_1))(1)$.

Suppose $\mathcal{T}(\text{select}(z_0, z_1))(1)$. Then, $\mathcal{H}(z_1)(1)$ hence, $\exists k \neq \bot.\ k \in [\![z_1]\!]\sigma$. Therefore, $1 \in [\![\text{select}(z_0, z_1)]\!]\sigma$. The same can be done for the case 0.
[*conditional*]

□

4.2 Hoare Logic

Hoare triple of a well–typed statement is defined as follow:

$$[P]\ \ \Gamma \vdash S \triangleright \Gamma'\ [Q]$$

where P is a predicate on $[\![\Gamma]\!]$ and Q is a predicate on $[\![\Gamma']\!]$. The Hoare triple is a proposition saying that for all state satisfying P, execution (semantic) of S yields a proper set of states which each element satisfies Q:

$$[P]\ \ \Gamma \vdash S \triangleright \Gamma'\ [Q]\ := \tag{1}$$
$$\forall \sigma. P\ \sigma \to (\bot \notin [\![\Gamma \vdash S \triangleright \Gamma']\!]\sigma \wedge \forall \delta \in [\![\Gamma \vdash S \triangleright \Gamma']\!]\sigma.\ Q\ \delta)$$

[What is $[\![\Gamma \vdash S \triangleright \Gamma']\!]$?] [Its the semantic of a well–typed statement; see Section 3.2] The following inference rules can be introduced:

$$\frac{}{[P]\ \Gamma \vdash \epsilon \triangleright \Gamma\ [P]}$$

$$\frac{}{[\exists w.\ \mathcal{T}(t)(w) \wedge \forall w.\ \mathcal{T}(t)(w) \to P[v \mapsto w]]\ \Gamma \vdash v := t\ \triangleright \Gamma\ [P]}$$

$$\frac{}{\begin{array}{c}[\exists w\ n.\ \mathcal{T}(t)(w) \wedge \mathcal{T}(z)(n) \wedge \forall n.\ \mathcal{T}(z)(n) \to 0 \leq n < \\ d \wedge \forall w\ n.\ \mathcal{T}(t)(w) \wedge \mathcal{T}(z)(n) \to P[T \twoheadrightarrow_n w]]\ \ \Gamma \vdash T[z] := t\ \triangleright \Gamma\ [P]\end{array}}$$

$$\frac{}{[\exists w.\ \mathcal{T}(t)(w) \wedge \forall w.\ \mathcal{T}(t)(w) \to P[v \mapsto w]]\ \Gamma \vdash \textbf{newvar}\ v := t\ \triangleright \Gamma'\ [P]}$$

$$\frac{[P]\ \Gamma \vdash S_1 \triangleright \Gamma_1\ [Q] \qquad [Q]\ \Gamma_1 \vdash S_2 \triangleright \Gamma_2\ [R]}{[P]\ \Gamma \vdash S_1; S_2 \triangleright \Gamma_2\ [R]}$$

$$\frac{[P \wedge \exists b \neq 0.\ \mathcal{T}(z)(b)]\ \Gamma \vdash S_1 \triangleright \Gamma\ [Q] \qquad [P \wedge \mathcal{T}(z)(0)]\ \Gamma \vdash S_1 \triangleright \Gamma\ [Q]}{[P \wedge \exists k.\ \mathcal{T}(z)(k)]\ \Gamma \vdash \textbf{if}\ z\ \textbf{then}\ S_1\ \textbf{else}\ S_2 \triangleright \Gamma\ [Q]}$$

$$\frac{\begin{array}{c}\exists c_0 \forall c\ [\exists k \neq 0. \mathcal{T}(z)(k) \wedge I \wedge V = c]\ \Gamma \vdash S \triangleright \Gamma\ [I \wedge V \leq c - c_0] \\ I \to \exists k. \mathcal{T}(z)(k) \quad I \wedge V \leq 0 \to \forall k.\ \mathcal{T}(z)(k) \to k = 0\end{array}}{[I]\ \Gamma \vdash \textbf{while}\ z\ \textbf{do}\ S \triangleright \Gamma\ [I \wedge h(z)(0)]}$$

$$\frac{P \to P'\ [P']\ \Gamma \vdash S \triangleright \Gamma'\ [Q'] \qquad Q' \to Q}{[P]\ \Gamma \vdash S \triangleright \Gamma'\ [Q]}$$

[Let us also add rules about function calls to integer multifunction $g \in \mathcal{G}$ and real function $f \in \mathcal{F}$ with specification according to the beginning of Section 4 here and Definition 7 in the main text.] [function call is a term constructor hence doesn't belong here; Hoare triple says how a statement changes states. Function call should be dealt in the semantic of terms section.]

Theorem 2. *The introduced inference rules are sound; for each inference rule, Equation 1 is satisfied.*

Proof. It can be proved using induction on statements. The other proofs are trivial except for the conditional and loop statements.

[*conditional*] The precondition $(P \wedge \exists k.\ \mathcal{T}(z)(k))\ \sigma$ ensures $\bot \notin [\![z]\!]\sigma$ finite. If $\exists k \neq 0.\ \{0, k\} \subseteq [\![z]\!]\sigma$, then $[\![\textbf{if}\ \cdots]\!]\sigma = [\![S_1]\!]\sigma \cup [\![S_2]\!]\sigma$. If $\exists k \neq 0.\ \{0, k\} \subseteq [\![z]\!]\sigma$, then $\exists k \neq 0.\ \mathcal{T}(z)(k)$ and $\mathcal{T}(z)(0)$. By the premises, for all $\delta \in [\![S_1]\!]\sigma \cup [\![S_2]\!]\sigma$, and the case distinction, $Q(\delta)$ holds. For the cases $0 \notin [\![z]\!]\sigma$ and $\{0\} = [\![z]\!]\sigma$, can be done similarly.

[*loop*] The loop variant V can be replaced to an original integer version $V' :=$

$\lceil V/c_0 \rceil$:

$$\forall c \, [\exists k \neq 0. \mathcal{T}(z)(k) \wedge I \wedge V' = c] \; \Gamma \vdash S \triangleright \Gamma \, [I \wedge V' < c]$$
$$I \wedge V' \leq 0 \rightarrow (\forall k. \, \mathcal{T}(z)(k) \rightarrow k = 0)$$

Also, the real loop variant can be restored from any loop invariant; hence, let us consider the integer loop variant.

Recall the chain $\omega_0 \sqsubseteq \omega_1 \sqsubseteq \cdots$ where $[\![\Gamma \vdash \textbf{while } z \textbf{ do } S \triangleright \Gamma']\!] = \bigsqcup_\infty \omega_i$. We need to prove that for any σ with the precondition $I(\sigma)$ the following holds:

$$\bot \notin (\sqcup_\infty \omega_i)\sigma \wedge \forall \delta \in (\sqcup_\infty \omega_i)\sigma. \, (I \wedge h(z)(0))\delta$$

Point–wise ordering on $[\![\Gamma]\!] \rightarrow \mathcal{P}[\![\Gamma]\!]_\bot$ ensures that $\omega_1\sigma \sqsubseteq \omega_2\sigma \sqsubseteq \cdots$ is a chain; hence, $(\bigsqcup_\infty \omega_i)\sigma = \bigsqcup_\infty (\omega_i\sigma)$. Due to the ordering on $\mathcal{P}[\![\Gamma]\!]_\bot$, $\bot \notin \bigsqcup_\infty (\omega_i\sigma)$ if and only if there is some n such that $\bot \notin \omega_n\sigma$ finite and $\omega_n\sigma = \omega_{n+1}\sigma = \cdots$. The loop variant will yield the bound; we need to prove the following:

$$\forall \sigma \; c. \; I \; \sigma \wedge V' \; \sigma \leq c \rightarrow \bot \notin \omega_{c+1}\sigma \wedge \forall \delta \in \omega_{c+1}\sigma. \, (I \wedge h(z)(0))\delta \qquad (2)$$

If $c = 0$, due to the premises, $\forall k. \; \mathcal{T}(z)(k)\sigma \rightarrow k = 0$ and $I \; \sigma$ ensures $\exists k. \, \mathcal{T}(z)(k)\sigma$. Therefore, $[\![z]\!]\sigma = \{0\}$. Hence, $\omega_1(\sigma) = \{\sigma\}$ and $\bigsqcup_\infty (\omega_j\sigma) = \{\sigma\}$.

Now, let us assume Equation 2 and prove the following proposition:

$$\bot \notin \omega_{c+2}\sigma \wedge \forall \delta \in \omega_{c+2}\sigma. \, (I \wedge h(z)(0))\delta$$

assuming $I \; \sigma \wedge V' \; \sigma \leq c+1$ for some σ. Due to the premises, we have $\bot \notin [\![z]\!]\sigma$ finite. Let us consider the case $\{0, k\} \subseteq [\![z]\!]\sigma$ for some nonzero k. The premise about the loop body ensures the following (with $c := c + 1$):

$$\bot \notin [\![S]\!]\sigma \wedge \forall \delta \in [\![S]\!]\sigma. \, (I \wedge V \leq c)\delta$$

Since, $\bot \notin [\![S]\!]\sigma$ finite, using $\omega_{c+2} := \mathcal{F}_{z, \, S}\omega_{c+1}$, we have the following:

$$\omega_{c+2} = \{\sigma\} \cup \bigcup_{\delta \in [\![S]\!]\sigma} \omega_{c+1}\delta$$

Using the induction hypothesis on any $\delta \in [\![S]\!]\sigma$, we have

$$\bot \notin \omega_{c+1}\delta \wedge \forall \delta' \in \omega_{c+1}. \, (I \wedge h(z)(0))\delta'$$

Hence, $\bot \notin \omega_{c+2}\sigma$ finite and all its element, including σ, satisfies $I \wedge h(z)(0)$.

The case $0 \notin [\![z]\!]\sigma$ can be done similarly and the other case $\{0\} = [\![z]\!]\sigma$ can be done exactly as the case $c = 0$. $\square$

4.3 Verification Condition

Verification condition of a well–typed statement $\Gamma \vdash S \triangleright \Gamma'$ annotated with a predicates $Q : Pred([\![\Gamma']\!])$ is a predicate $vc(\Gamma \vdash S \triangleright \Gamma', Q) : Pred([\![\Gamma]\!])$, which defines a set of states such that

$$\forall \sigma \in [\![\Gamma]\!].\ vc(\Gamma \vdash S \triangleright \Gamma', Q)(\sigma) \to \bot \notin [\![S]\!]\sigma \wedge \forall \delta \in [\![S]\!]\sigma.\ Q(\delta).$$

Verification condition is defined inductively as follow:

$$\begin{aligned}
&vc(\epsilon, Q) = Q \\
&vc(v := t, Q) = \exists w.\ \mathcal{T}(t)(w) \wedge \forall w.\ \mathcal{T}(t)(w) \to Q[v \mapsto w] \\
&vc(T[z] := t, Q) = (\exists n\ w.\ \mathcal{T}(z)(n) \wedge \mathcal{T}(t)(w)) \wedge \\
&\qquad\qquad \wedge (\forall n.\ \mathcal{T}(z)(n) \to 0 \leq n < d) \\
&\qquad\qquad \wedge (\forall n\ w.\ \mathcal{T}(z)(n) \wedge \mathcal{T}(t)(w) \to Q[T \mapsto T \twoheadrightarrow_n w]) \\
&vc(\textbf{newvar}\ v := t, Q) = \exists w.\ \mathcal{T}(t)(w) \wedge \forall w.\ \mathcal{T}(t)(w) \to Q[T \mapsto w] \\
&vc(S_1; S_2, Q) = vc(S_1, wp(S_2, Q)) \\
&vc(\textbf{if}\ z\ \textbf{then}\ S_1\ \textbf{else}\ S_2, Q) = \\
&\qquad \exists k.\mathcal{T}(z)(k) \wedge (\exists k \neq 0.\ \mathcal{T}(z)(k) \to vc(S_1, Q)) \wedge (\mathcal{T}(z)(0) \to vc(S_2, Q)) \\
&vc(\textbf{while}\ z\ \textbf{do}\ S, Q) = \\
&\qquad I \wedge \\
&\qquad \forall\ \mathbf{v}. \\
&\qquad\quad (I \wedge \mathcal{T}(z)(0) \to Q) \\
&\qquad\quad \wedge I \to (\exists k.\mathcal{T}(z)(k)) \\
&\qquad\quad \wedge \exists c_0.\forall c.\ (I \wedge (\exists k \neq 0.\ \mathcal{T}(z)(k)) \wedge (V = c) \to vc(S, I \wedge V \leq c - c_0)) \\
&\qquad\quad \wedge V \leq 0 \to (\forall v.\ \mathcal{T}(z)(v) \to v = 0)
\end{aligned}$$

Above $\mathbf{v}$ represents the set of variables that gets assigned in S. the first clause I denotes that the loop invariant I should hold at the beginning of the loop; other clauses are conditions that should be satisfied throughout any stages of the loop.

Theorem 3. *The introduced verification condition is correct*

Proof. The definition is induced from Hoare logic inference rules; hence, the proof is analogous to the proof of Theorem 2. □

Having a well–typed statement S, proving $P \to vc(S, Q)$ yields $[\![P]\!]\ S\ [\![Q]\!]$. Note that the other way wont hold as vc is not the weakest precondition.

4.4 Program Verification

Recall that a ERC program is in the following format:

$\boldsymbol{p} :=$

input $v_1 : \tau_1, v_2 : \tau, \cdots, v_n : \tau_n$

S

return t

Considering that the program $\boldsymbol{p}$ is well–typed, we have $\Gamma_0 \vdash S \triangleright \Gamma$ and $\Gamma \vdash t : \tau$ where $\Gamma_0 = \cup_i (v_i \to \tau_i)$ with some τ.

Precondition of a program is a predicate that defines a set of input values of the input variables. Postcondition should be a predicate on the input values of the input variables and their return value. Hence, $Pre : Pred(\llbracket \Gamma_0 \rrbracket)$ and $Post : Pred(\llbracket \tau \rrbracket \times \llbracket \Gamma_0 \rrbracket)$.

Now, let us consider which predicate should hold at the end of the statement in order to guarantee $Post$. One thing to make sure is that the semantic of the returning term to be proper and all of its elements to satisfy $Post$:

$$\exists r.\ \mathcal{T}(t)(r) \wedge \forall r,\ \mathcal{T}(t)(r) \to Post(r, \cdots)$$

However, the above modification is wrong since the variables in the postcondition should refer to the initial values of the input variables, not the values that the input variables are storing at the moment.

To make sure the variables in $Post$ refer to the initial values, we can alter the variables:

$$\exists r.\ \mathcal{T}(t)(r) \wedge \forall r,\ \mathcal{T}(t)(r) \to Post[v_i \mapsto v_i'](r)$$

We can compute the verification condition on the above predicate. Since v_i' refers to the initial values and the variables v_i in $uv(\mathrm{cot})$, now, also refer the the initial values, we can reset those altered variable back; program verification of the program $\boldsymbol{p}$ is to prove the following:

$pv(P, \boldsymbol{p}, Q) :=$

$$P \to vc(S, \exists r.\ \mathcal{T}(t)(r) \wedge \forall r,\ \mathcal{T}(t)(r) \to Q(r)[v_i \mapsto v_i'])[v_i' \mapsto v_i]$$

5 Example: Trisection

Trisection, which is also introduced in the main text, is a rigorous algorithm finding the root in the unit interval of a continuous function where it is guaranteed that the function f is (i) continuous, (ii) has a sign change $f(0) < 0 < f(1)$ and (iii) its root is unique.

Since, we do not have a function typed variable in the language, we put the function in our set of the special function symbols. Hence, we let $\mathcal{F} := \{f \mapsto \mathbf{R}\}$. We put the three conditions into our theory.

Trisection algorithm can be expressed as a program in ERC as follow:

Input$(n : \mathbf{Z})$
  newvar $a := 0.0$;
  newvar $b := 1.0$;
  while $\text{select}(\text{test}(\iota(n) > b - a), \text{test}(b - a > \iota(n-1)))$
  do
    if $\text{select}(f((a + 2.0 * b) * /3.0) > 0.0, 0.0 > f((2.0 * a + b) * /3.0))$
    then $a := (2.0 * a + b) * /3.0)$
    else $b := (a + 2.0 * b) * /3.0$
return a

It is not hard to see that the program is well–typed with its return type **R**.

5.1 Semantic of Terms

The current construction of the semantic translation function introduces many quantifiers. As an example, even for a simple term like $(2.0*a+b)*/3.0$, directly evaluating the translation yields the following long predicate:

$$
\begin{array}{l}
\mathcal{T}((2.0 * a + b) * /3.0) = \\
\lambda\omega. \\
\quad \exists v_1\ v_2.\ \omega = v_1 \times v_2 \wedge \\
\quad\quad (\exists v_3\ v_4.\ v_1 = v_3 + v_4 \wedge \\
\quad\quad\quad (\exists v_6\ v_7.\ v_3 = v_6 \times v_7 \wedge v_6 = 2 \wedge v_7 = a \wedge v_4 = b \wedge \\
\quad\quad\quad\quad (\exists v_5.\ 1 = v_2 \times v_5 \wedge v_5 = 3)))
\end{array}
$$

However, we can note that the quantifiers are trivial; we can reduce a predicate $\exists x.\ x = a \wedge b$ to $b[x \mapsto a]$. Hence, we can reduce most of quantifiers that are related to variables and constants; since the introduced quantifiers are mostly related to multivaluedness and undefinedness.

Translated semantics of the terms in Trisection can be simplified as follow:

$\mathcal{T}(0.0) =$

$\lambda\omega.\ \omega = 0$

$\mathcal{T}(1.0) =$

$\lambda\omega.\ \omega = 1$

$\mathcal{T}(\text{select}(\text{test}(\iota(n) > b - a), \text{test}(b - a > \iota(n-1)))) =$

$\lambda\omega.\ (\omega = 0 \wedge 2^n > b - a) \vee (\omega = 1 \wedge b - a > 2^{n-1})$

$\mathcal{T}(\text{select}(f((a + 2.0 * b) * /3.0) > 0.0, 0.0 > f((2.0 * a + b) * /3.0))) =$

$$\lambda\omega.\ \left(\omega = 0 \wedge f\left(\frac{a + 2 \cdot b}{3}\right) > 0\right) \vee \left(\omega = 1 \wedge f\left(\frac{2 \cdot a + b}{3}\right) < 0\right)$$

$\mathcal{T}((2.0 * a + b) * /3.0))$

$$\lambda\omega.\ \omega = \frac{2 \cdot a + b}{3}$$

$\mathcal{T}((a + 2.0 * b) * /3.0))$

$$\lambda\omega.\ \omega = \frac{a + 2 \cdot b}{3}$$

5.2 Verification

We want the program to compute 2^n approximation of the root. Hence, we let our pre and postcondition to be as follow:

$$\begin{aligned} Pre &:= True, \\ Post &:= \lambda r.\ \exists u.\ f(u) = 0 < 0 \leq u < 1 \wedge approx(u, r, n). \end{aligned}$$

$Post$ is a predicate on the return value r and the input value n. $approx(u, r, n) := -2^n < r - u < 2^n$ is an abbreviation.

Let the loop invariant to be $I := uniq(0, 1)$ and the loop variant to be $V = b - a - 2^{n-1}$ where $uniq(a, b) := f(a) < 0 < f(b) \wedge 0 < a < b < 1 \wedge \exists!\ z.\ f(z) = 0 \wedge a < z < b$. Notice that $uniq(0, 1)$ is the one of the conditions added to our theory.

With $Post$, the postcondition of the *program*, we apply the verification condition function on the following predicate:

$$\begin{aligned} Post' &:= \exists r.\ \mathcal{T}(t)(r) \wedge \forall r,\ \mathcal{T}(t)(r) \rightarrow Post[v_i \mapsto v_i'](r) \\ &= \exists r.\ r = a \wedge \forall r.\ r = a \rightarrow (\exists u.\ f(u) = 0 \wedge 0 \leq u \leq 1 \wedge |u - r| < 2^{n'}) \\ &= (\exists u.\ f(u) = 0 \wedge 0 \leq u \leq 1 \wedge |u - a| < 2^{n'}) \end{aligned}$$

Abbreviating $S := \textbf{if} \cdots \textbf{then} \cdots \textbf{else} \cdots$ in the program, applying the verification condition function on the postcondition yields the following five condi-

tions to be proved:

$$uniq(0,1) \tag{3}$$

$$\forall \alpha\ \beta.$$

$$\big(uniq(\alpha,\beta) \wedge (2^n < \beta - \alpha) \rightarrow Post'[a \mapsto \alpha]\big)[n' \mapsto n] \tag{4}$$

$$uniq(\alpha,\beta) \rightarrow (2^n > \beta - \alpha \vee \beta - \alpha > 2^{n-1}) \tag{5}$$

$$\exists c_0.\ \forall c.\ uniq(\alpha,\beta) \wedge \beta - \alpha > 2^{n-1} \wedge (\beta - \alpha - 2^{n-1}) = c) \rightarrow \tag{6}$$
$$vc(S, uniq(a,b) \wedge b - a \leq c - c_0)[a \mapsto \alpha, b \mapsto \beta]$$

$$(\beta - \alpha - 2^{n-1} \leq 0) \rightarrow \tag{7}$$
$$(\forall \omega.\ (\omega = 0 \wedge 2^n < \beta - \alpha) \vee (\omega = 1 \wedge \beta - \alpha > 2^{n-1}) \rightarrow \omega = 0)$$

Proposition 3 is exactly the proposition which we have added to our theory. $uniq(\alpha,\beta)$ assures that there is an unique root in $[\alpha,\beta]$ and $2^n < \beta - \alpha$ ensures that the distance between α and the root is less than 2^n; hence, Proposition 4 can be proven. Having $2^n > 2^{n-1} > 0$ ensures $\forall x.\ 2^n > x \vee x > 2^{n-1}$; hence, Proposition 5 can be proven as well. Proposition 7 can be easily proven by case distinction.

Now, let us see what happens inside the loop (Proposition 6). See that $vc(S, uniq(a,b) \wedge b - a \leq c - c_0)[a \mapsto \alpha, b \mapsto \beta]$ can be evaluated to be the following propositions quantified by $\forall c.\ \exists c_0.$:

$$f\left(\frac{\alpha + 2\cdot\beta}{3}\right) > 0 \vee f\left(\frac{2\cdot\alpha + \beta}{3}\right) < 0$$

$$f\left(\frac{2\cdot\alpha + \beta}{3}\right) < 0 \rightarrow uniq((2\cdot\alpha+\beta)/3, \beta) \wedge \frac{2}{3}(\beta - \alpha) \leq c - c_0$$

$$f\left(\frac{\alpha + 2\cdot\beta}{3}\right) > 0 \rightarrow uniq(\alpha, (\alpha+2\cdot\beta)/3) \wedge \frac{2}{3}(\beta - \alpha) \leq c - c_0$$

Letting $c_0 := 2^{n-3}$, together with Intermediate Value Theorem and the uniqueness of the root in $[\alpha,\beta]$, we can ensure the three propositions.

6 Implementation: `erc-vc-extract`

`erc-vc-extract`, a verification condition extractor is implemented in `ocaml` using `yacc` parser generator. Given a pre/postcondition and loop in/variant annotated ERC program, `erc-vc-extract` generates a `Coq` file which contains the extracted verification condition of the annotated ERC program; a user can prove the statements in `Coq` to verify the input ERC program.

`erc-vc-extract` reduces the trivial quantifiers (see Section 5.1) and produces a text file includes how the quantifiers are reduced for each term.

The source code of `erc-vc-extract` can be found in `http://erc.realcomputation.asia/wc-extract`. As a prototype of the development, annotated Trisection program `trisection.erc` was written, and ran by the extractor. The produced

`Coq` file `trisection.v` contains more or less the same propositions mentioned in Section 5.2, and the verification conditions are proved in `Coq`; proof of it is contained in `trisection_proved.v` file. The example codes and proofs are also accessible in the provided url.