mathematics

MDPI

Article

On Signifiable Computability: Part II: An Axiomatization of Signifiable Computation and Debugger Theorems

Vladimir A. Kulyukin

Department of Computer Science, Utah State University, Logan, UT 84322, USA; vladimir.kulyukin@usu.edu

Abstract: Signifiable computability aims to separate what is theoretically computable from what is computable through performable processes on computers with finite amounts of memory. Mathematical objects are signifiable in a formalism $\mathscr{L}$ on an alphabet $\mathscr{A}$ if they can be written as spatiotemporally finite texts in $\mathscr{L}$ on $\mathscr{A}$. In a previous article, we formalized the signification and reference of real numbers and showed that data structures representable as multidimensional matrices of discretely finite real numbers are signifiable. In this investigation, we continue to formulate our theory of signifiable computability by offering an axiomatization of signifiable computation on discretely finite real numbers. The axiomatization implies an ontology of functions on discretely finite real numbers that classifies them as signifiable, signifiably computable, and signifiably partially computable. Relative to $\mathscr{L}$ and $\mathscr{A}$, signification is performed with two formal systems: the Former $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ that forms texts in $\mathscr{L}$ on $\mathscr{A}$ and the Transformer $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ that transforms texts formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ into other texts in $\mathscr{L}$ on $\mathscr{A}$. Singifiable computation is defined relative to $\mathscr{L}$ on $\mathscr{A}$ as a finite sequence of signifiable program states, the first of which is generated by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ and each subsequent state is deterministically obtained from the previous one by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$. We define a debugger function to investigate signifiable computation on finite-memory devices and to prove two theorems, which we call the Debugger Theorems. The first theorem shows that, for a singifiably partially computable function signified by a program on a finite-memory device $\mathscr{D}$, the memory capacity of $\mathscr{D}$ is exceeded when running the program on signifiable discretely finite real numbers outside of the function's domain. The second theorem shows that there are functions signifiably computable in general that become partially signifiably computable when signified by programs on $\mathscr{D}$ insomuch as the memory capacity of $\mathscr{D}$ can be exceeded even when the programs are executed on some signifiable discretely finite real numbers in the domains of these functions.

Keywords: computability theory; recursion theory; signifiable computability; axiomatization; debugger theorems; discretely finite real numbers; number theory

MSC: 03D75; 03D80

Received: 11 February 2025
Revised: 1 March 2025
Accepted: 2 March 2025
Published: 11 March 2025

Citation: Kulyukin, V.A. On Signifiable Computability: Part II: An Axiomatization of Signifiable Computation and Debugger Theorems. *Mathematics* **2025**, *13*, 934. https://doi.org/10.3390/math13060934

1. Introduction

While general computability describes functions computable on devices without any memory limitations [1], or, equivalently, computable in principle [2], actual computability aims to characterize functions computable on devices with finite amounts of memory available for computation. We call such devices *finite-memory devices* (FMD) [3]. In a previous article [4], we began to formulate a theory of signifiable computability (or, equivalently, a signifiable computability theory (SCT)). The SCT separates general computability investigated in the classical computability theory (CCT) from actual computability when

computation occurs within a finite amount of memory, as is the the case in many areas of applied computer science. In signifiable computability, names of intuitive objects, such as numbers, functions, data types and structures (i.e., instances of data types), programs, and program states are treated as spatiotemporally finite texts constructed with formal languages (or, equivalently, *formalisms*) on specific alphabets. Thus, these texts *signify* (or, equivalently, *designate* or *name*) intuitive objects, such as numbers, functions, data types and structures, programs, and program states in a given universe of discourse described by a formalism $\mathscr{L}$ on an alphabet $\mathscr{A}$ (e.g., `Lisp` on `Unicode`). When the texts are stored within the available memory units on an FMD $\mathscr{D}$, they signify the corresponding objects in the finite memory of $\mathscr{D}$, thereby making the universe of discourse conditional not only on $\mathscr{L}$ and $\mathscr{A}$, but also on $\mathscr{D}$. E.g., a human programmer using `Lisp` on `Unicode` on a computer with 5 terabytes (TB) of combined RAM and hard disk space cannot signify an intuitive object on this FMD if the storage size of the signifying text (i.e., the name of the object in `Lisp` on `Unicode`) exceeds 5 TB.

In a previous investigation [4], we defined the concepts of signification and reference of real numbers and extended signification to number tuples and data types and structures. We showed that data structures representable as tuples of discretely finite real numbers are signifiable. The signification of multidimensional matrices was introduced to show that data structures representable as multidimensional matrices of discretely finite real numbers are also signifiable. We argued that, in order to be sufficiently expressive, $\mathscr{L}$ on $\mathscr{A}$ must be not only *decimal-sufficient* (i.e., should contain finitely many rules to signify discretely finite real numbers in the standard decimal notation or an equivalent thereof) but also *tuple-sufficient* (i.e., should have finitely many rules to signify sequences of such numbers).

In this investigation, we continue to formulate the SCT by offering an axiomatization of signifiable computation on discretely finite real numbers. We show how a sufficiently expressive $\mathscr{L}$ on $\mathscr{A}$ can be extended with instruction types and how these types can be signified with texts and arranged into sequences. Signified instructions in a text sequence can be executed (or, equivalently, *performed*) one-by-one in order to carry out the prescribed numerical computation. If the computation exists (i.e., the execution finishes), its result (or, equivalently, output) is signified with a text in $\mathscr{L}$ on $\mathscr{A}$. An independent party can verify the computation from (1) the input text; (2) the text of the instruction sequence; and (3) the text with the sequences of the numerals of the individual instructions used in the computation.

This investigation is motivated by the following limitations of the CCT reflected in its axioms: (1) what is computable in principle (general computability) is indistinguishable from what is actually computable on computers with finite amounts of memory (actual computability); (2) the investigation of computability properties is confined to functions on natural numbers; (3) the formation of programs is insufficiently axiomatized insomuch as it is unclear how programs are *formed* (or, equivalently, *generated* or *produced*); (4) the transformation of programs to obtain a computation in the form of a finite sequence of program states is insufficiently axiomatized in that there is no concept of error or cost; (5) interpretation of input and output texts is unresolved to the extent that no position is taken on whether interpretation (e.g., a denotational or operational semantics) is part of computation or not. While the SCT is grounded in the CCT, it extends the CCT, because it (1) separates general computability from actual; (2) extends the investigation of computability properties to functions on discretely finite real numbers; (3) makes actual computability properties of functions explicitly dependent on formalisms, alphabets, and FMDs on which those functions are signified and computed; (4) introduces error and cost into computation; (5) takes a position that interpretation is not part of computation and as such belongs to

metacomputability. As we formulate the SCT in this article, we reference, when appropriate, the CCT axioms summarized for the reader's convenience in the Appendix A.

The remainder of this article is organized as follows. In Section 2, we review the basic ontology of the CCT presented in [1,2] and the formalism in [4] that we continue to use in this article. In Section 3, we axiomatize the formation and transformation of texts. Signification is actualized with two formal systems: the Former $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ that generates texts in $\mathscr{L}$ on $\mathscr{A}$, and the Transformer $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ that transforms texts formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ into other texts in $\mathscr{L}$ on $\mathscr{A}$. In Section 4, we define formable and transformable instructions and axiomatize the formation and transformation of program instructions and formation of programs. In Section 5, we axiomatize program states and computations. Computation is defined relative to $\mathscr{L}$ on $\mathscr{A}$ as a finite sequence of signifiable program states, the first of which is generated by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ and each subsequent state is deterministically obtained from the previous one by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$. In Section 6, we axiomatize the transformation of programs and discuss whether it is possible for $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ to transform program states that are not generated by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$. In Section 7, we offer an ontology of functions on discretely finite real numbers that classifies functions as signifiable, signifiably computable, and signifiably partially computable. In Section 8, we introduce memory limitations on signifiable computation to characterize functions as signifiably computable and partially computable on FMDs. Sections 2–8 can be viewed as the preliminary, auxiliary steps to Section 9, where we present our main results. In particular, we define a debugger function to actualize signifiable computation on FMDs and use it to prove two theorems, which we call the Debugger Theorems, to investigate some of the effects of finite memory on signifiable computation. In Section 10, we discuss our results. In Section 11, we summarize our findings and outline the scope of the next part of our investigation of signifiable computability.

2. Prolegomena

The reader may skip this section on first reading and return to it as necessary. Here, we review the basic concepts of the CCT and the basic formal apparatus in [4] that we continue to use in this article.

The CCT ontology used in this article is based on [1,2] and includes the following concepts:

1. *function*—a mathematical object, i.e., a mapping from a given domain to a given range;
2. *alphabet*—a finite set of symbols (or, equivalently, *signs*) (e.g., $\mathscr{A} = \{\texttt{"0"}, \texttt{"1"}\}$);
3. *formalism*—a formal language $\mathscr{L}$ on a specific alphabet $\mathscr{A}$ defined with a finite set of syntactic rules used to *name* (or, equivalently, *signify* or *designate*) intuitive mathematical objects (e.g., numbers, functions, sequences, etc.);
4. *model*—a formal model to compute functions, such as a Turing machine (TM) (cf., e.g., Ch. 6 in [1]);
5. *program*—a signified finite sequence of instructions in $\mathscr{L}$ on $\mathscr{A}$ that specifies how a function named by the program can be computed, i.e., how a signified sequence of inputs in $\mathscr{L}$ on $\mathscr{A}$ can be mapped (or, equivalently, *transformed*) to a signified sequence of outputs by a computer capable of performing instructions in $\mathscr{L}$ on $\mathscr{A}$;
6. *computer*—a device or an agent, physical or abstract, that executes programs in $\mathscr{L}$ on $\mathscr{A}$ on given inputs according to a specific model;
7. *program state*—a state of a program in $\mathscr{L}$ on $\mathscr{A}$ signified by a spatiotemporally finite text in $\mathscr{L}$ on $\mathscr{A}$ that, at a minimum, signifies the value of every variable used by the program and the next instruction in the program for a computer to perform;
8. *computation*—a finite sequence of program states signified by a spatiotemporally finite text in $\mathscr{L}$ on $\mathscr{A}$ that a computer goes through in order to compute the signified values

of a function designated by a program in $\mathscr{L}$ on $\mathscr{A}$ on a signified input (or, equivalently, to transform a signified input to a signified output);

9. *class of functions*—a set of functions; in the CCT, the three most prominent classes are: *primitive recursive*; *computable*; and *partially computable*.

Finite automata (cf., e.g., [5]) such as finite state machines (FSA), stack machines (also known as push-down automata (PDA)), and TMs are *formal models* signifiable with different formalisms on different alphabets to compute different classes of functions. A formalism $\mathscr{L}$ is necessarily paired with an alphabet $\mathscr{A}$. E.g., a programming language, such as `Lisp`, `Perl`, `C/C++`, etc., can be defined on `Unicode` or `ASCII`. Computability is a property of functions. E.g., a primitive recursive function (e.g., $f(x) = x + 1$) is signifiable in different formalisms on different alphabets and is computable by an FSA, a PDA, or a TM. There are functions signifiable with context-free formalisms and computable by PDAs that are not computable by FSAs (e.g., $f(n) = \ddot{e}^n?^n$, $n > 0$, where $\ddot{e}^n$ and $?^n$ denote n consecutive occurences of ë and ?, respectively). There are functions signifiable with context-sensitive formalisms that are computable by TMs but not by PDAs (e.g., $f(n) = \ddot{e}^n?^n!^n$, $n > 0$) (cf., e.g., [6]). In formal logic, some texts in $\mathscr{L}$ on $\mathscr{A}$ are referred to as *names* and the named classes of those objects are called *extensions* (cf. p. 94 and pp. 268–269 in [7]). If a text signifies a unique object, it signifies a singleton.

The statements $S_1 \subset S_2$ and $S_1 \subseteq S_2$ mean that S_1 is a proper subset and a subset of the set S_2, respectively; $\emptyset$ denotes the empty set. $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Z}^-$, $\mathbb{Z}^+$, $\mathbb{Q}$, $\mathbb{R}$ respectively denote the infinite sets of natural numbers ($\mathbb{N} = \{0, 1, 2, \ldots\}$), of whole numbers ($\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$), of negative whole numbers ($\mathbb{Z}^- = \{\ldots, -2, -1\}$), of positive whole numbers ($\mathbb{Z}^+ = \{1, 2, 3, \ldots\}$), of rational numbers ($\mathbb{Q} = \{\frac{m}{n} | m \in \mathbb{Z}, n \in \mathbb{Z}^- \cup \mathbb{Z}^+\}$), and of real numbers that include the whole numbers, the rational numbers, and the irrational numbers. If $T_{i_1}, T_{i_2}, \ldots, T_{i_k}, k \in \mathbb{Z}^+$ are sets, then

$$T_{i_1} \times T_{i_2} \times \cdots \times T_{i_k} = \{(t_{i_1}, t_{i_2}, \ldots, t_{i_k}) | t_{i_1} \in T_{i_1}, t_{i_2} \in T_{i_2}, \ldots, t_{i_k} \in T_{i_k}\},$$

where $(t_{i_1}, t_{i_2}, \ldots t_{i_k})$ is called a *k-tuple*, i.e., a finite sequence of k elements.

A *finite-memory device* (FMD) $\mathscr{D}$ is a computational device with a finite number of memory units (or, equivalently, *memory cells*) available for computation. An FMD is not a model of computation insomuch as it can perform computations according to different models so long as each computation can be signified within its memory capacity. If $\mathfrak{D}$ is the set of all FMDs, then the function $\texttt{CCAP} : \mathfrak{D} \mapsto \mathbb{Z}^+$ maps each FMD to the number of its memory cells. Thus, each FMD has a strictly positive memory capacity. It should be noted that if a device $\mathscr{D} \in \mathfrak{D}$ is an FSA, a PDA, or a TM, then

$$\neg(\exists k)\{k \in \mathbb{Z}^+ \wedge \texttt{CCAP}(\mathscr{D}) = k\}.$$

On the other hand, if D is an FMD, then

$$(\exists k)\{k \in \mathbb{Z}^+ \wedge \texttt{CCAP}(\mathscr{D}) = k\}.$$

Thus, if $\mathscr{D}_1$ and $\mathscr{D}_2$ are two TMs, the predicates $\texttt{CCAP}(\mathscr{D}_1) = \texttt{CCAP}(\mathscr{D}_2)$, $\texttt{CCAP}(\mathscr{D}_1) < \texttt{CCAP}(\mathscr{D}_2)$, $\texttt{CCAP}(\mathscr{D}_1) > \texttt{CCAP}(\mathscr{D}_2)$ are, respectively, equivalent to $\infty = \infty$, $\infty < \infty$, $\infty > \infty$, of which the first statement is vacuously true, whereas the second and third statements have no immediate logical satisfaction without a reinterpretation of $<$ and $>$ (e.g., along the lines of Cantor's set theory, where $\aleph_0 < 2^{\aleph_0}$). On the other hand, if $\mathscr{D}_1$ and $\mathscr{D}_2$ are FMDs, then the three `CCAP` statements above can be assigned the Boolean values of *True* and *False* under the standard interpretation of the three comparison operators on natural numbers.

If $\mathscr{D}$ is an FSA, a PDA, or a TM, then $\mathscr{D}$ can be turned into an FMD if, and only if, one of the two conditions holds. First, $\texttt{CCAP}(\mathscr{D}) = k, k \in \mathbb{Z}^+$, i.e., an explicit upper bound in the form of a concrete positive integer is placed on the combined size of the the text that signifies the finite control of $\mathscr{D}$ and the size of the input tape. Second, $\mathscr{D}$ is signified in a specific formalism on a specific alphabet (e.g., `Lisp`, `Perl`, `C/C++` on `Unicode`) on a specific FMD (e.g., a computer with 5 TB of RAM and hard disk space). In the latter case, $\mathscr{D}$ is an FMD, because its memory capacity is bounded by the memory capacity of the device on which it runs, i.e., $\texttt{CCAP}(\mathscr{D}) \leq 5$ TB.

An *elementary sign* is that which can be written in exactly one memory cell. We enclose all elementary signs in pairs of matching `"` (e.g., `"0"`, `"1"`, `"+"`). The elementary sign `""` is the *empty sign*. An *alphabet* is an enumerably finite, non-empty set of non-empty elementary signs. A *text* is a spatiotemporally finite sequences of elementary signs of $\mathscr{A}$. We intentionally use the term *text* instead of the term *string* to avoid allusions to the modern string theory in physics or to the classical finite automata theory of theoretical computer science. All texts are also enclosed in matching pairs of `"`. An alphabet $\mathscr{A}$ is *decimal-sufficient* if, and only if, $\mathscr{A}$ contains the signs of the standard decimal notation or their isomorphic equivalents. $\mathscr{L}$ on $\mathscr{A}$ is *decimal-sufficient* if, and only if, it allows, via a finite set of rules, for the mechanical formation of texts on $\mathscr{A}$ in the characteristic-mantissa form of the standard decimal notation.

A basic text formation operation in $\mathscr{L}$ on $\mathscr{A}$ is *concatenation*, denoted as $\bigoplus_{\mathscr{A},\mathscr{L}}$. The order of the sign concatenation in the formation of a text is arbitrarily assumed to be left to right, i.e., for $n \in \mathbb{Z}^+$,

$$s_{i_1}s_{i_2}\dots s_{i_n} \quad = \quad \Big(\dots\Big(\dots\Big(\Big(s_{i_1}\bigoplus_{\mathscr{A},\mathscr{L}} s_{i_2}\Big)\bigoplus_{\mathscr{A},\mathscr{L}} s_{i_3}\Big)\dots\bigoplus_{\mathscr{A},\mathscr{L}} s_{i_{n-1}}\Big)\bigoplus_{\mathscr{A},\mathscr{L}} s_{i_n}\Big).$$

Two enumerably infinite sets, denoted as

$$\mathscr{A}^+_{\mathscr{L}}; \mathscr{A}^*_{\mathscr{L}},$$

are associated with $\mathscr{L}$ on $\mathscr{A}$. The first set includes all signs of $\mathscr{A}$ and all non-empty texts in $\mathscr{L}$ on $\mathscr{A}$ formed according to the finite set of rules of $\mathscr{L}$. The second set is defined as

$$\mathscr{A}^*_{\mathscr{L}} = \{\texttt{""}\} \cup \mathscr{A}^+_{\mathscr{L}}.$$

For $t_i \in \mathscr{A}^+_{\mathscr{L}}$, $1 \leq i \leq l \in \mathbb{Z}^+$, $l > 1$, we let

$$\begin{gathered}\bigoplus_{\mathscr{A},\mathscr{L}}\Big(t_1, t_2, \dots, t_l\Big) = \bigoplus_{\mathscr{A},\mathscr{L}}\Big|_{i=1}^{l} t_i = \\ \bigoplus_{\mathscr{A},\mathscr{L}} t_1 \bigoplus_{\mathscr{A},\mathscr{L}} t_2 \bigoplus_{\mathscr{A},\mathscr{L}} \dots \bigoplus_{\mathscr{A},\mathscr{L}} t_l.\end{gathered} \tag{1}$$

If $t \in \mathscr{A}^+_{\mathscr{L}}$, then $|t|$ is the number of the elementary signs from $\mathscr{A}$ in t. If $s \in \mathscr{A}$, then $|s| = 1$ and $|\texttt{""}| = 0$. If $|t| = n \in \mathbb{Z}^+$, then s_1, s_2, …, s_n designate the consecutive elementary signs of $t = s_1 s_2 \dots s_n$ from the leftmost to the rightmost, where $s_i \in \mathscr{A}, 1 \leq i \leq n$.

If f is a function, $dom(f)$ and $ran(f)$ denote the domain and the range of f, respectively; $f : S \mapsto R$ abbreviates $dom(f) = S \wedge ran(f) = R$, where $\wedge$ denotes the logical *and*. E.g., if $f(n) = \text{ë}^n?^n!^n$, $n > 0$, then $dom(f) = \mathbb{Z}^+$ and $ran(f) = \{$ ë?!, ëë??!!, ëëë???!!!, …$\}$ or, equivalently, $f : \mathbb{N} \mapsto \{$ ë?!, ëë??!!, ëëë???!!!, …$\}$. If $f : T_{i_1} \times T_{i_2} \times \dots \times T_{i_k} \mapsto T_j$ and $(t_{i_1}, t_{i_2}, \dots, t_{i_k}) \in dom(f)$, then f is *defined on* $(t_{i_1}, t_{i_2}, \dots, t_{i_k})$ or, in symbols, $f(t_{i_1}, t_{i_2}, \dots, t_{i_k})\downarrow$ if, and only if, there exists $t_j \in T_j$ such that $f(t_{i_1}, t_{i_2}, \dots, t_{i_k}) = t_j$ or, equivalently, $(\exists\, t_j \in T_j)\ \{f(t_{i_1}, t_{i_2}, \dots, t_{i_k}) = t_j\}$, where $\exists$ designates the logical *existential quantifier*. If $\neg(\exists\, t_j \in T_j)\ \{f(t_{i_1}, t_{i_2}, \dots, t_{i_k}) = t_j\}$, where $\neg$ designates the logical *not*, then f is

undefined on $(t_{i_1}, t_{i_2}, \ldots, t_{i_k})$ or, in symbols, $f(t_{i_1}, t_{i_2}, \ldots, t_{i_k})\uparrow$. If $f : T_{i_1} \times T_{i_2} \times \cdots \times T_{i_k} \mapsto T_j$ and

$$(\forall\ (t_{i_1}, t_{i_2}, \ldots, t_{i_k}) \in T_{i_1} \times T_{i_2} \times \cdots \times T_{i_k})\ (\exists\ t_j \in T_j)\ \{f(t_{i_1}, t_{i_2}, \ldots, t_{i_k}) = t_j\},$$

where $\forall$ designates the logical *universal quantifier*, f is *total* on $T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$. If

$$(\exists\ (t_{i_1}, t_{i_2}, \ldots, t_{i_k}) \in T_{i_1} \times T_{i_2} \times \cdots \times T_{i_k})\ \{f(t_{i_1}, t_{i_2}, \ldots, t_{i_k})\uparrow\},$$

then f is *partial* on $T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$. It should be noted that while the notation $f : T_{i_1} \times T_{i_2} \times \cdots \times T_{i_k} \mapsto T_j$ is used for partial and total functions, total functions on $T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$ are defined for all $(t_{i_1}, t_{i_2}, \ldots, t_{i_k}) \in T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$, whereas partial functions on $T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$ may not be defined for some $(t_{i_1}, t_{i_2}, \ldots, t_{i_k}) \in T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$. Thus, every total function on $T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$ is partial on $T_{i_1} \times T_{i_2} \times \ldots \times T_{i_k}$, but not vice versa.

Let $r \in \mathbb{R}$. Then, r is *discretely finite* in $\mathscr{L}$ on $\mathscr{A}$ if, and only if, there exists

$$S_r = \left\{t | t \in \mathscr{A}^+_{\mathscr{L}}\right\} \neq \emptyset,$$

where each t signifies (or, equivalently, designates or names) r and no other number, or, in symbols,

$$r \leftarrow \{=\}_{\mathscr{A},\mathscr{L}} \rightarrow S_r. \tag{2}$$

If $t \in S_r$, then t signifies r in $\mathscr{L}$ on $\mathscr{A}$, or, in symbols,

$$t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r. \tag{3}$$

$\leftarrow \{=\}_{\mathscr{A},\mathscr{L}} \rightarrow$ and $\leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow$ are symmetric.

Let $r \in \mathbb{R}$ such that $r \leftarrow \{=\}_{\mathscr{A},\mathscr{L}} \rightarrow S_r \neq \emptyset$. Then, r is *finite-memory signifiable* (FM-signifiable) or, equivalently, FM-designatable in $\mathscr{L}$ on $\mathscr{A}$ on an FMD $\mathscr{D}$ if, and only if, there exists $S'_r \subseteq S_r$ such that $S'_r = \{t | \texttt{CCAP}(\mathscr{D}) \geq |t|\} \neq \emptyset$, in which case,

$$r \leftarrow \{=\}_{\mathscr{A},\mathscr{L},\mathscr{D}} \rightarrow S'_r \tag{4}$$

holds. If $t \in S'_r$, then t FM-signifies or, equivalently, FM-designates r in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$, or, in symbols,

$$t \leftarrow (=)_{\mathscr{A},\mathscr{L},\mathscr{D}} \rightarrow r. \tag{5}$$

$\leftarrow \{=\}_{\mathscr{A},\mathscr{L},\mathscr{D}} \rightarrow$ and $\leftarrow (=)_{\mathscr{A},\mathscr{L},\mathscr{D}} \rightarrow$ are symmetric. In general, the subscript $\mathscr{D}$ when added to the right of the subscripts $\mathscr{L}$ and $\mathscr{A}$ means that a relationship is defined not only relative to $\mathscr{L}$ and $\mathscr{A}$ but also to $\mathscr{D}$, and, in particular, relative to $\texttt{CCAP}(\mathscr{D})$.

A *numeral* is $t \in \mathscr{A}^+_{\mathscr{L}}$ that designates a real number. A decimal-sufficient alphabet $\mathscr{A}$ can be extended with the unique signs $\trianglelefteq$, $\trianglerighteq$, $\mp$ that designate the beginning of a tuple, the end of a tuple, and a number separator inside a tuple, respectively. E.g., let

$$\mathscr{A} = \{\texttt{"0"}, \texttt{"1"}, \texttt{"2"}, \texttt{"3"}, \texttt{"4"}, \texttt{"5"}, \texttt{"6"}, \texttt{"7"}, \texttt{"8"}, \texttt{"9"}, \texttt{"+"}, \texttt{"-"}, \texttt{"."}\}, \tag{6}$$

be a decimal-sufficient alphabet and let

$$\begin{aligned} \mathscr{B} &= \mathscr{A} \cup \mathscr{A}_\Delta; \\ \mathscr{A}_\Delta &= \{\texttt{"}\diamondsuit\texttt{"}, \texttt{"}|_a\texttt{"}, \texttt{"}_a|\texttt{"}, \texttt{";"}, \texttt{"}\triangledown\texttt{"}, \texttt{"}\square\texttt{"}, \texttt{"}\blacktriangleleft\texttt{"}, \texttt{"}\circledast\texttt{"}\}. \end{aligned} \tag{7}$$

$\mathscr{B}$ consists of the standard decimal notation signs in $\mathscr{A}$ and the signs in $\mathscr{A}_\Delta$ that can be used to signify number tuples, data types, and data instances. Specifically, in $\mathscr{B}$ in (7), we let $\trianglelefteq$ = "$|_a$", $\trianglerighteq$ = "$_a|$", and $\mp$ = ";". Alphabets that contain these unique signs are *tuple-sufficient*. Thus, $\mathscr{B}$ is tuple-sufficient. $\mathscr{L}$ on a tuple-sufficient $\mathscr{A}$ is *tuple-sufficient* if, and only if, $\mathscr{L}$ contains finitely many rules that allow for the signification of k-tuples of discretely finite real numbers on $\mathscr{A}$. For brevity, we will say that $\mathscr{L}$ on $\mathscr{A}$ is tuple-sufficient to mean that both $\mathscr{L}$ and $\mathscr{A}$ are tuple-sufficient. E.g., `Lisp`, `Perl`, `C/C++` on `Unicode` are tuple-sufficient.

Let $t_i \in \mathscr{A}^+_{\mathscr{L}}$, $1 \leq i \leq l \in \mathbb{Z}^+$, $l > 1$, $t_i \neq \trianglelefteq$, $t_i \neq \trianglerighteq$, $t_i \neq \mp$. Then,

$$\bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\left(t_1, \mp, t_2, \mp, \ldots, \mp, t_l\right)$$

is the *unmarked concatenation* of $t_1, \ldots, t_l$ in $\mathscr{L}$ on $\mathscr{A}$ and

$$\bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\left(\trianglelefteq, \bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\left(t_1, \mp, t_2, \mp, \ldots, \mp, t_l\right), \trianglerighteq\right)$$

is the *marked concatenation* of $t_1, \ldots, t_l$ in $\mathscr{L}$ on $\mathscr{A}$. E.g., in a tuple-sufficient $\mathscr{L}$ on $\mathscr{B}$, `"1;2;3"` is the unmarked concatenation of `"1"`, `"2"`, and `"3"`, whereas "$|_a$`1;2;3`$_a|$" is the marked concatenation of `"1"`, `"2"`, and `"3"`.

A k-tuple $\mathfrak{r} = (r_1, \ldots, r_k) \in \mathbb{R}^k$, $1 < k \in \mathbb{Z}^+$, is signifiable or, equivalently, designatable in $\mathscr{L}$ on $\mathscr{A}$ if, and only if, (a) $\mathfrak{r}$ is of type $\mathbb{T}$ and $\mathbb{T}$ is signifiable in $\mathscr{L}$ on $\mathscr{A}$, or, in symbols, $\mathbb{T} \Leftarrow\!\langle\rangle_{\mathscr{A},\mathscr{L}}\Rightarrow t_{\mathbb{T}} \in \mathscr{A}^+_{\mathscr{L}}$; and (b) $r_i \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_i$, $t_i \in \mathscr{A}^+_{\mathscr{L}}$, $1 \leq i \leq k$. If (a) and (b) hold, then $\mathfrak{r}$ is signifiable in $\mathscr{L}$ on $\mathscr{A}$ by

$$t_{\mathfrak{r}} = \bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\left(\trianglelefteq, \bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\left(t_1, \mp, t_2, \mp, \ldots, \mp, t_k\right), \trianglerighteq\right)$$

or, in symbols,

$$\mathfrak{r} \leftarrow \doteqdot_{\mathscr{A},\mathscr{L}} \rightarrow t_{\mathfrak{r}}.$$

If $\mathfrak{r}$ is not signifiable in $\mathscr{L}$ on $\mathscr{A}$, then $\mathfrak{r} \leftarrow \doteqdot_{\mathscr{A},\mathscr{L}} \rightarrow$ "$\leftarrow \doteqdot_{\mathscr{A},\mathscr{L}} \rightarrow$ is symmetric.

A k-tuple $\mathfrak{r} = (r_1, \ldots, r_k) \in \mathbb{R}^k$, $1 < k \in \mathbb{Z}^+$, is finite-memory signifiable (FM-signifiable) or, equivalently, FM-designatable in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$ if, and only if, (a) $\mathfrak{r} \leftarrow \doteqdot_{\mathscr{A},\mathscr{L}} \rightarrow t_{\mathfrak{r}} \in \mathscr{A}^+_{\mathscr{L}}$ and (b) $|t_{\mathfrak{r}}| \leq \mathtt{CCAP}(\mathscr{D})$. If (a) and (b) hold, then $\mathfrak{r} \leftarrow \doteqdot_{\mathscr{A},\mathscr{L},\mathscr{D}} \rightarrow t_{\mathfrak{r}}$. If $\mathfrak{r}$ is not FM-signifiable in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$, then $\mathfrak{r} \leftarrow \doteqdot_{\mathscr{A},\mathscr{L},\mathscr{D}} \rightarrow$ "$\leftarrow \doteqdot_{\mathscr{A},\mathscr{L},\mathscr{D}} \rightarrow$ is symmetric.

Let p_i be the i-th prime so that $p_1 = 2$, $p_2 = 3$, $p_3 = 5$, etc., and let

$$\pi(i) = p_i. \tag{8}$$

Let $(n_1, \ldots, n_k)$ be a k-tuple such that $n_i \in \mathbb{N}, 1 \leq i \leq k$. The Gödel number (G-number) of $(n_1, \ldots, n_k)$ is

$$[n_1, \ldots, n_k] = \prod_{i=1}^{k} \pi(i)^{n_i}. \tag{9}$$

Let $(z_1, \ldots, z_k)$ be such that $z_i \in \mathbb{Z}^+, 1 \leq i \leq k$. The j-shifted Gödel number (denoted as $\mathrm{G}_{\Delta=j}$, $j \in \mathbb{N}$) of this k-tuple is

$$[z_1, \ldots, z_k]_{\Delta=j} = \prod_{i=1}^{k} \pi(i+j)^{z_i}. \tag{10}$$

The G-number and $\mathrm{G}_{\Delta=j}$-number of $()$ are defined to be 1.

Let $\gamma_{\mathscr{A}} : \mathscr{A} \mapsto \mathbb{Z}^+$ be a 1–1 function that maps each sign of $\mathscr{A}$ to a unique odd prime, i.e.,

$$\gamma_{\mathscr{A}}(s_i) \quad = \quad \pi(i+1), 1 \leq i \leq l, l > 0. \tag{11}$$

E.g., for $\mathscr{B}$ in (7), we have

$$\begin{array}{llllllllllllllll}
\gamma_{\mathscr{B}}(\texttt{"0"}) & = & 3; & \gamma_{\mathscr{B}}(\texttt{"1"}) & = & 5; & \gamma_{\mathscr{B}}(\texttt{"2"}) & = & 7; & \gamma_{\mathscr{B}}(\texttt{"3"}) & = & 11; \\
\gamma_{\mathscr{B}}(\texttt{"4"}) & = & 13; & \gamma_{\mathscr{B}}(\texttt{"5"}) & = & 17; & \gamma_{\mathscr{B}}(\texttt{"6"}) & = & 19; & \gamma_{\mathscr{B}}(\texttt{"7"}) & = & 23; \\
\gamma_{\mathscr{B}}(\texttt{"8"}) & = & 29; & \gamma_{\mathscr{B}}(\texttt{"9"}) & = & 31; & \gamma_{\mathscr{B}}(\texttt{"+"}) & = & 37; & \gamma_{\mathscr{B}}(\texttt{"-"}) & = & 41; \\
\gamma_{\mathscr{B}}(\texttt{"."}) & = & 43; & \gamma_{\mathscr{B}}(\texttt{"}\diamondsuit\texttt{"}) & = & 47; & \gamma_{\mathscr{B}}(\texttt{"}|_a\texttt{"}) & = & 53; & \gamma_{\mathscr{B}}(\texttt{"}a|\texttt{"}) & = & 59; \\
\gamma_{\mathscr{B}}(\texttt{";"}) & = & 61; & \gamma_{\mathscr{B}}(\texttt{"}\triangledown\texttt{"}) & = & 67; & \gamma_{\mathscr{B}}(\texttt{"}\square\texttt{"}) & = & 71; & \gamma_{\mathscr{B}}(\texttt{" }\blacktriangleleft\texttt{ "}) & = & 73; \\
\gamma_{\mathscr{B}}(\texttt{" }\circledast\texttt{ "}) & = & 79. & & & & & & & & &
\end{array}$$

The 1–1 function in (12) maps $t \in \mathscr{A}^+_{\mathscr{L}}$ (i.e., $t = s_1 \ldots s_n$, $n \in \mathbb{Z}^+$, $s_j \in \mathscr{A}$, $1 \leq j \leq n$) to a unique positive integer and maps "" to 0 (cf. Theorem 4 in [4]).

$$\mathfrak{g}_{\mathscr{A},\mathscr{L}}(t) = \begin{cases} [\gamma_{\mathscr{A}}(s_1), \ldots, \gamma_{\mathscr{A}}(s_n)]_{\Delta=1} & \text{if } |t| > 0, \\ 0 & \text{if } |t| = 0. \end{cases} \tag{12}$$

3. Formation, Transformation, Metacomputability

A *programming formalism* $\mathscr{L}$ on $\mathscr{A}$ is a formalism by which one formal system generates texts in $\mathscr{L}$ on $\mathscr{A}$ that signify sequences of numerical instructions that another formal system can perform. Hence,

Definition 1 (Programming Formalism)**.** *Let $\mathscr{L}$ be tuple-sufficient on $\mathscr{A}$. Then, $\mathscr{L}$ on $\mathscr{A}$ is a programming formalism if it is used by two formal systems: the Former $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ and the Transformer $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$.*

A formal system in (1) is specified by a finite set of formal rules and a rule application mechanism to form texts or to transform texts into other texts. The rules and rule application mechanisms have no meaning inside the systems and, in that sense, are purely syntactic (cf. Ch. IV in [8]).

Definition 2 (The Former)**.** *The formal text formation system, denoted as $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$, is a finite set of rules*

$$\mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} = \left\{ fr_1, \ldots, fr_k \middle| fr_j : {\mathscr{A}^*_{\mathscr{L}}}^m \mapsto \mathscr{A}^+_{\mathscr{L}} \right\}, 0 < j \leq k, k, m \in \mathbb{Z}^+,$$

and a rule application mechanism $M_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ to form texts in $\mathscr{A}^+_{\mathscr{L}}$.

Each formation rule $fr_j \in \mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ is assigned a unique name $t_{fr_j} \in \mathscr{A}^+_{\mathscr{L}}$ (i.e., signified with a unique t_{fr_j}). Since $\mathfrak{g}_{\mathscr{A},\mathscr{L}}(t_{fr_j})$ is a unique natural number, we have

$$(\exists t') \left\{ t' \in \mathscr{A}^+_{\mathscr{L}} \wedge t' \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow \mathfrak{g}_{\mathscr{A},\mathscr{L}}(t_{fr_j}) \right\}, \tag{13}$$

so that each formation rule is mapped to a unique natural number signifiable in $\mathscr{L}$ on $\mathscr{A}$, which we abbreviate as

$$fr_j \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_{fr_j}. \tag{14}$$

If $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ uses the rule sequence $fr_{i_1}, fr_{i_2}, \ldots, fr_{i_l}$, $l \in \mathbb{Z}^+$, to form $t \in \mathscr{A}^+_{\mathscr{L}}$ from "", then, by convention, the first argument of each rule takes the output of the previous rule and the first argument of fr_{i_1} is "", whence

$$t = fr_{i_l}\Big(fr_{i_{l-1}}\Big(\ldots fr_{i_2}\Big(fr_{i_1}\Big(t_{i_{1_1}},\ldots,t_{i_{1_m}}\Big),\ldots,t_{i_{2_m}}\Big)\ldots,t_{i_{l-1_m}}\Big)\ldots,t_{i_{l_m}}\Big),$$

where $t_{i_{j_1}} = \texttt{""}$ and $t_{i_{j_k}} \in \mathscr{A}^*_{\mathscr{L}}$, $1 < j \leq l, 1 < k \leq m \in \mathbb{Z}^+$. E.g., if $\mathscr{L}$ is tuple-sufficient on

$$\begin{aligned}\mathscr{A} \quad = \quad & \{\texttt{"0"},\texttt{"1"},\texttt{"2"},\texttt{"3"},\texttt{"4"},\texttt{"5"},\texttt{"6"},\texttt{"7"},\texttt{"8"},\texttt{"9"},\texttt{"+"},\texttt{"-"},\texttt{"."},\\ & \texttt{"}\diamondsuit\texttt{"},\texttt{"}|_a\texttt{"},\texttt{"}_a|\texttt{"},\texttt{";"},\texttt{"}\triangledown\texttt{"},\texttt{"}\square\texttt{"},\texttt{" }\blacktriangleleft\texttt{ "},\texttt{" }\circledast\texttt{ "},\texttt{"}\clubsuit\texttt{"}\}\end{aligned} \tag{15}$$

and $\mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ contains the rule for $\bigoplus_{\mathscr{A},\mathscr{L}}$, then

$$\texttt{"3.14"} = \bigoplus_{\mathscr{A},\mathscr{L}}\Big(\bigoplus_{\mathscr{A},\mathscr{L}}\Big(\bigoplus_{\mathscr{A},\mathscr{L}}\Big(\bigoplus_{\mathscr{A},\mathscr{L}}\Big(\texttt{""},\texttt{"3"}\Big),\texttt{"."}\Big),\texttt{"1"}\Big),\texttt{"4"}\Big).$$

$\mathscr{A}_{\oplus}$ designates the concatenation of the signs in $\mathscr{A}$ separated with $\mp \notin \mathscr{A}$. E.g., if $\mp = \texttt{"}\heartsuit\texttt{"}$, then, for $\mathscr{A}$ in (15), we have

$$\mathscr{A}_{\oplus} = \texttt{"0}\heartsuit\texttt{1}\heartsuit\texttt{2}\heartsuit\texttt{3}\heartsuit\texttt{4}\heartsuit\texttt{5}\heartsuit\texttt{6}\heartsuit\texttt{7}\heartsuit\texttt{8}\heartsuit\texttt{9}\heartsuit\texttt{+}\heartsuit\texttt{-}\heartsuit\texttt{.}\heartsuit\diamondsuit\heartsuit|_a\heartsuit_a|\heartsuit\texttt{;}\heartsuit\triangledown\heartsuit\square\heartsuit\texttt{ }\blacktriangleleft\texttt{ }\heartsuit\texttt{ }\circledast\texttt{ }\heartsuit\clubsuit\heartsuit\texttt{"}.$$

Definition 3 (Text Formation). *A mechanical formation or, simply, formation of $t \in \mathscr{A}^+_{\mathscr{L}}$ by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ is a finite sequence of formation rules $fr_{i_1}, fr_{i_2}, \ldots, fr_{i_l}$, $fr_{i_l} \in \mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ such that*

$$t = fr_{i_l}\Big(fr_{i_{l-1}}\Big(\ldots fr_{i_2}\Big(fr_{i_1}\Big(t_{i_{1_1}},\ldots,t_{i_{1_m}}\Big),\ldots,t_{i_{2_m}}\Big)\ldots,t_{i_{l-1_m}}\Big)\ldots,t_{i_{l_m}}\Big),$$

where

$$\begin{aligned}& fr_{i_j} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_{fr_{i_j}} \in \mathscr{A}^+_{\mathscr{L}};\\ & t_{i_{j_1}} = \texttt{""};\ t_{i_{j_k}} \in \mathscr{A}^*_{\mathscr{L}};\\ & 1 \leq j \leq l \in \mathbb{Z}^+;\ 1 < k \leq m \in \mathbb{Z}^+.\end{aligned}$$

Each rule $fr_j \in \mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ has access to $\mathscr{A}_{\oplus}$.

Definition 4 (Formation Signature). *Let $fr_{i_1}, fr_{i_2}, \ldots, fr_{i_l}$, $fr_{i_l} \in \mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$, $l \in \mathbb{Z}^+$, be a formation of $t \in \mathscr{A}^+_{\mathscr{L}}$ by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$, where, for $m \in \mathbb{Z}^+$,*

$$t = fr_{i_l}\Big(fr_{i_{l-1}}\Big(\ldots fr_{i_2}\Big(fr_{i_1}\Big(t_{i_{1_1}},\ldots,t_{i_{1_m}}\Big),\ldots,t_{i_{2_m}}\Big)\ldots,t_{i_{l-1_m}}\Big)\ldots,t_{i_{l_m}}\Big).$$

Then,

$$s = \bigoplus_{\mathscr{A},\mathscr{L}}\Big(\trianglelefteq, t_{fr_{i_l}}, \trianglelefteq, t_{fr_{i_{l-1}}}, \trianglelefteq, \ldots, t_{fr_{i_2}}, \trianglelefteq, t_{fr_{i_1}}, \trianglelefteq, \mathfrak{a}_{fr_{i_1}}, \trianglerighteq, \mathfrak{a}_{fr_{i_2}}, \trianglerighteq, \ldots, \mathfrak{a}_{fr_{i_{l-1}}}, \trianglerighteq, \mathfrak{a}_{fr_{i_l}}, \trianglerighteq, \trianglerighteq\Big),$$

where $fr_{i_j} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_{fr_{i_j}}$, $1 \leq j \leq l$, $\mathfrak{a}_{fr_{i_1}} = \bigoplus_{\mathscr{A},\mathscr{L}}\left(t_{i_{1_1}}, \mp, \ldots, \mp, t_{i_{1_m}}\right)$, $\mathfrak{a}_{fr_{i_2}} = \bigoplus_{\mathscr{A},\mathscr{L}}\left(t_{i_{2_1}}, \mp, \ldots, \mp, t_{i_{2_m}}\right)$, …, $\mathfrak{a}_{fr_{i_l}} = \bigoplus_{\mathscr{A},\mathscr{L}}\left(t_{i_{l_1}}, \mp, \ldots, \mp, t_{i_{l_m}}\right)$, for $t_{i_{j_k}}$, $1 \leq j \leq l$, $1 \leq k \leq m$, is a formation signature of $t \in \mathscr{A}^+_{\mathscr{L}}$ by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$, whence

$$(fr_{i_1}, fr_{i_2}, \ldots, fr_{i_l}) \leftarrow \div_{\mathscr{A},\mathscr{L}} \rightarrow s,$$

and $(fr_{i_1}, fr_{i_2}, \ldots, fr_{i_l})$ is a signed formation of $t \in \mathscr{A}^+_{\mathscr{L}}$ by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ designated by s.

E.g., let $\mathscr{L}$ be tuple-sufficient on $\mathscr{B}$ in (7), where $\trianglelefteq = \texttt{"}|_a\texttt{"}$, $\trianglerighteq = \texttt{"}|_a\texttt{"}$, $\mp = \texttt{";"}$, and

$$\bigoplus_{\mathscr{A},\mathscr{L}} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow \texttt{"}\clubsuit\texttt{"}.$$

Then, if we keep the matching pairs of " only in "" and "." for clarity and omit them elsewhere for brevity, then

$$s = \text{"}\clubsuit|_a\clubsuit|_a\clubsuit|_a\clubsuit|_a\text{""};3_a|;\text{"."}_a|;1_a|;4_a|\text{"}$$

designates the formation signature of "3.14" by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$.

Thus, any $t \in \mathscr{A}^+_{\mathscr{L}}$ formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ exists as a spatiotemporally finite entity constructed from a finite number of signs of $\mathscr{A}$ by finitely many rules signified in the formation signature of t. Hence, our first two axioms.

SCT Axiom 1. *A signed formation of $t \in \mathscr{A}^+_{\mathscr{L}}$ by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ takes a finite amount of physical time, and t occupies a finite amount of physical space.*

SCT Axiom 2. *If s designates a formation signature of t by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$, then, given $\mathscr{A}$, s, and $\mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$, another formal system can deterministically verify the formation of t by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ within a finite amount of physical time and space.*

No a priori knowledge is assumed on how time and space are physically measured (e.g., sun cycles, turns of a sand clock, nanoseconds, meters, centimeters, cubic meters, bits, bytes, megabytes, etc.). SCT Axioms 1 and 2 concretize CCT Axiom A1 (cf. Appendix A) by taking a definite position on the origin of texts that designate programs: (1) these texts are produced by one formal system for another formal system; (2) their production can be deterministically verified; and (3) the produced texts are spatiotemporally finite objects.

For $t \in \mathscr{A}^+_{\mathscr{L}}$, the predicate

$$\Vdash_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} (t) \tag{16}$$

holds if, and only if, t is formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ through a signed formation. We let $\Vdash_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} (\text{""})$ vacuously hold and define

$$\mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} = \left\{ t \in \mathscr{A}^+_{\mathscr{L}} \;\middle|\; \Vdash_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} (t) \right\} \tag{17}$$

to be the enumerably infinite set of texts formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$. E.g., if $\mathscr{A}$ is `Unicode` and $\mathscr{L}$ is `Lisp` and $\mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ includes the finitely many Common Lisp program formation rules in [9], then

$$\Vdash_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} (\texttt{"(defun f (n) (+ n 1))"}) \;\equiv\; \texttt{"(defun f (n) (+ n 1))"} \in \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}.$$

Definition 5 (The Transformer)**.** *The formal text transformation system, denoted as $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$, is a finite set of rules*

$$\mathfrak{R}_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} = \left\{ tr_1, \ldots tr_k | tr_j : \mathscr{A}^*_{\mathscr{L}} \mapsto \mathscr{A}^*_{\mathscr{A}} \right\}, 1 \leq j \leq k, k \in \mathbb{Z}^+,$$

and a deterministic rule application mechanism $M_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}$ to transform texts in $\mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ into texts in $\mathscr{A}^*_{\mathscr{L}}$.

Each transformation rule $tr_j \in \mathfrak{R}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ is assigned a unique name $w_{tr_j} \in \mathscr{A}^+_{\mathscr{L}}$ (i.e., signified with a unique w_{tr_j}). Since $\mathfrak{g}_{\mathscr{A},\mathscr{L}}(w_{fr_j})$ is a unique natural number,

$$(\exists t') \left\{ t' \in \mathscr{A}^+_{\mathscr{L}} \wedge t' \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow \mathfrak{g}_{\mathscr{A},\mathscr{L}}(w_{fr_j}) \right\} \tag{18}$$

so that each transformation rule is mapped to a unique natural number signifiable in $\mathscr{L}$ on $\mathscr{A}$, which we abbreviate as

$$tr_j \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow w_{fr_j}. \tag{19}$$

Associated with every tuple $(\mathscr{A},\mathscr{L})$ is a finite, non-empty set of error messages

$$\mathtt{ERR}_{\mathscr{A},\mathscr{L}} = \left\{ t \;\middle|\; t \in \mathscr{A}^{+}_{\mathscr{L}} \right\}, \mathtt{ERR}_{\mathscr{A},\mathscr{L}} \cap \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} = \emptyset, \tag{20}$$

that the transformation rules of $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ can output to signal that specific transformations are impossible. E.g., if $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ cannot verify the formation of $t \in \mathscr{A}^{+}_{\mathscr{L}}$ by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$, then $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ issues an error message.

Suppose that $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ uses the rule sequence $tr_{i_1}, tr_{i_2}, \ldots, tr_{i_l}$, $l \in \mathbb{Z}^{+}$, to transform $t \in \mathscr{A}^{+}_{\mathscr{L}}$ to $t' \in \mathscr{A}^{*}_{\mathscr{L}}$. By convention, the first argument of each subsequent rule takes the output of the previous rule and the first argument of tr_{i_1} is t, whence we have the following definition:

Definition 6 (Text Transformation)**.** *A mechanical transformation, or simply transformation, of $t \in \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ into $t' \in \mathscr{A}^{*}_{\mathscr{L}}$, $t' \notin \mathtt{ERR}_{\mathscr{A},\mathscr{L}}$, is a finite sequence of transformation rules $tr_{i_1}, tr_{i_2}, \ldots, tr_{i_l}$, $tr_{i_l} \in \mathfrak{R}_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}$, $tr_{i_l} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow w_{tr_{i_j}} \in \mathscr{A}^{+}_{\mathscr{L}}$, $1 \leq j \leq l \in \mathbb{Z}^{+}$, where*

$$t' = tr_{i_l}\Big(tr_{i_{l-1}}\Big(\ldots tr_{i_2}\Big(tr_{i_1}\Big(t\Big)\Big)\ldots\Big)\Big).$$

An example of a transformation rule is *substitution* which $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ can use to substitute $t_1 \in \mathscr{A}^{+}_{\mathscr{L}}$ for all occurrences of $t_2 \in \mathscr{A}^{+}_{\mathscr{L}}$ in $t_3 \in \mathscr{A}^{+}_{\mathscr{L}}$ so long as the texts satisfy a finite set of syntactic requirements (cf., e.g., Ch. 18, p. 558 in [1]; Ch. IX, § 44 in [8]; Ch. 14, pp. 402–403 in [9]); Ch. 4, p. 66 in [10], for various definitions of substitution in different formal systems).

Definition 7 (Transformation Signature)**.** *Let $tr_{i_1}, tr_{i_2}, \ldots, tr_{i_l}, tr_{i_l} \in \mathfrak{R}_{\hat{\hat{T}},\mathscr{A},\mathscr{L}}$, $l \in \mathbb{Z}^{+}$, be a transformation of $t \in \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$, where*

$$t' = tr_{i_l}\Big(tr_{i_{l-1}}\Big(\ldots tr_{i_2}\Big(tr_{i_1}\Big(t\Big)\Big)\ldots\Big)\Big).$$

Then,

$$s = \bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\Big(\trianglelefteq, w_{tr_{i_l}}, \trianglelefteq, w_{tr_{i_{l-1}}}, \trianglelefteq, \ldots, w_{tr_{i_2}}, \trianglelefteq, w_{tr_{i_1}}, \trianglelefteq, t, \trianglerighteq, \trianglerighteq, \ldots, \trianglerighteq, \trianglerighteq, \trianglerighteq\Big),$$

where $tr_{i_j} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow w_{tr_{i_j}}$, $1 \leq j \leq l$, is a transformation signature of the transformation of t into t' by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$, in which case

$$(tr_{i_1}, tr_{i_2}, \ldots, tr_{i_l}) \leftarrow\!\doteq_{\mathscr{A},\mathscr{L}}\!\rightarrow s$$

holds, and $(tr_{i_1}, tr_{i_2}, \ldots, tr_{i_l})$ is a signed transformation, designated by s, of t into t' by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$.

Let $t \in \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$, $t' \in \mathscr{A}^{*}_{\mathscr{L}}$, $t' \notin \mathtt{ERR}_{\mathscr{A},\mathscr{L}}$. The predicate

$$\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} (t, t') \tag{21}$$

holds if, and only if, there exists a signed transformation of t into t' by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$. The relation $\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} \left(t, t'\right)$ is clearly transitive, i.e., if

$$\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} \left(t, t'\right) \wedge \vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} \left(t', t''\right),$$

then

$$\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} \left(t, t''\right).$$

E.g., if $\mathscr{A}$ is `Unicode`, $\mathscr{L}$ is `Lisp`, and $\mathfrak{R}_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}$ is the standard transformation rules of Common Lisp applied deterministically by the rule application mechanism known as the *read-eval-print* loop (cf., e.g., Ch. 20 in [9]), then

$$\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} (\texttt{"(progn (defun g (n) (+ n 1)) (g 2.14))","3.14"})$$

holds, because $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ uses the `Lisp` `progn` construct that defines a sequence of Lisp operations to transform `"(defun g (n) (+ n 1)) (g 2.14))"` into a text of a procedure associated with the symbol named `"g"`, substitutes `"2.14"` for the variable named `"n"` in `"(+ n 1)"` to obtain `"(+ 2.14 1)"`, and then uses the transformation rules associated with the function named `"+"` to transform `"(+ 2.14 1)"` to `"3.14"`. One can easily verify this transformation in any `Lisp` system that implements the Common Lisp formation and transformation rules in [9]. E.g., if $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ is GNU CLISP 2.49.60+ on Linux 18.04 LTS, and $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ is used to form

```
"(progn (defun g (n) (+ n 1)) (g 2.14))"
```

on `Unicode`, then the following transformation is produced in the GNU CLISP 2.49.60+ Linux terminal, where `">"` is the CLISP prompt.

```
"> (progn (defun g (n) (+ n 1)) (g 2.14))
3.14"
```

Suppose $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ generates and signs

```
"(loop while (= 3.14 3.14) do (+ 13.7 13.7))"
```

for $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ to transform. $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ cannot transform the above text into another text designating a result, because the program, while formally flawless, contains an infinite loop. Consequently, although the choice and the application of a transformation rule by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ are mechanical and deterministic, and both actions take a finite amount of physical time, the transformation of a program into a text designating a result may not take a finite amount of physical time.

The next axiom states that spatiotemporally finite transformations are verifiable. If a transformation and its signature are available, another formal system can independently and deterministically verify the transformation from its signature and the set of transformation rules.

SCT Axiom 3. *If* $\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} (t, t')$, *where* $t \in \mathscr{A}^{+}_{\mathscr{L}}$, $t' \in \mathscr{A}^{*}_{\mathscr{L}}$, *and* $t' \notin \texttt{ERR}_{\mathscr{A},\mathscr{L}}$, *holds and* s *designates the transformation signature of the transformation of* t *into* t' *by* $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$, *then, given* t, t', s, *and* $\mathfrak{R}_{\hat{\hat{T}},\mathscr{A},\mathscr{L}}$, *another formal system can deterministically verify the transformation of* t *into* t' *by* $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ *within finite amounts of physical time and space.*

SCT Axiom 3 extends the CCT axioms by introducing the concept of deterministic verifiability of transformation. We should note here that SCT Axioms 1, 2, and 3 take no position with respect to computing agents that use $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ and $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ or computing devices on which these systems are signified. Thus, neither $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ nor $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ are identical to the computing agent L in CCT Axioms A2, A4, A5, A9, and A10.

By *interpretation* of a program P in $\mathscr{L}$ on $\mathscr{A}$, we will mean the application of a denotational or operational semantics of $\mathscr{L}$ on $\mathscr{A}$ to P with respect to the input text t and, when applicable, to the output text t' (cf., e.g., Chapters 17, 18 in [1]; Ch. 2 in [10])). We will use the term *metacomputability* to refer to a denotational and operational semantics of $\mathscr{L}$ on $\mathscr{A}$ and the interpretation of programs with respect to given input and output texts and assume that metacomputability is captured by neither the formation nor the transformation rules. We will not tackle the question of whether metacomputability is algorithmic in nature and leave it outside of the scope of this article. Hence,

SCT Axiom 4. *Metacomputability occurs outside of* $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ *and* $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$.

4. Formation and Transformation of Instructions and Formation of Programs

Some texts formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ and transformed by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ designate instructions from a finite set of instruction types

$$\mathfrak{I}_{\mathscr{A},\mathscr{L}} = \{\tilde{\mathfrak{I}}_1, \tilde{\mathfrak{I}}_2, \ldots, \tilde{\mathfrak{I}}_n\}, n \in \mathbb{Z}^+. \tag{22}$$

The elements of $\mathfrak{I}_{\mathscr{A},\mathscr{L}}$ denote operations of a specific type, e.g., addition or subtraction of finitely many discretely finite real numbers. E.g., $\mathscr{L}$ is `Lisp` on `Unicode`, then

```
"(+ x1 y1)","(+ x2 y2)","(+ x3 y3)",...
```

designate the same numerical operation type of addition applied to numerals designated by specific variable names. If we switch $\mathscr{L}$ to `Perl` on `Unicode`, then we obtain

```
"$x1 + $y1","$x2 + $y2","$x3 + $y3",...
```

designating the same numerical operation type.

Definition 8 (Signifiable Instruction)**.** *If* $t \in \mathscr{A}^+_{\mathscr{L}}$ *and* $\mathfrak{i}_k \in \tilde{\mathfrak{I}}_j \in \mathfrak{I}_{\mathscr{A},\mathscr{L}}$*, then the predicate* $\bowtie_{\mathscr{A},\mathscr{L}} (t, \mathfrak{i}_k)$ *holds if, and only if,* t *signifies* $\mathfrak{i}_k$ *in* $\mathscr{L}$ *on* $\mathscr{A}$*, in which case* $\mathfrak{i}_k$ *is a signifiable or, equivalently, designatable instruction in* $\mathscr{L}$ *on* $\mathscr{A}$*.* $\bowtie_{\mathscr{A},\mathscr{L}}$ *is symmetric.*

E.g., let $x, y \in \mathbb{R}$ and `L` stand for `Lisp`, `P` for `Perl`, and `U` for `Unicode`. We have the following relations hold.

$$\begin{array}{lcl}
\bowtie_{\mathtt{U,L}} (x+y, \texttt{"(+ x y)"}) & \equiv & \bowtie_{\mathtt{U,L}} (\texttt{"(+ x y)"}, x+y) \\
\bowtie_{\mathtt{U,P}} (x+y, \texttt{"\$x + \$y;"}) & \equiv & \bowtie_{\mathtt{U,P}} (\texttt{"\$x + \$y;"}, x+y) \\
\bowtie_{\mathtt{U,L}} (x-y, \texttt{"(- x y)"}) & \equiv & \bowtie_{\mathtt{U,L}} (\texttt{"(- x y)"}, x-y) \\
\bowtie_{\mathtt{U,P}} (\texttt{"\$x - \$y;"}, x-y) & \equiv & \bowtie_{\mathtt{U,P}} (x-y, \texttt{"\$x - \$y;"}) \\
\bowtie_{\mathtt{U,L}} (xy, \texttt{"(* x y)"}) & \equiv & \bowtie_{\mathtt{U,L}} (\texttt{"(* x y)"}, xy) \\
\bowtie_{\mathtt{U,P}} (\texttt{"\$x * \$y;"}, xy) & \equiv & \bowtie_{\mathtt{U,P}} (\texttt{"\$x * \$y;"}, xy) \\
\bowtie_{\mathtt{U,L}} (x/y, \texttt{"(/ x y)"}) & \equiv & \bowtie_{\mathtt{U,L}} (\texttt{"(/ x y)"}, x/y) \\
\bowtie_{\mathtt{U,P}} (\texttt{"\$x / \$y;"}, x/y) & \equiv & \bowtie_{\mathtt{U,P}} (x/y, \texttt{"\$x / \$y;"})
\end{array}$$

The elements of $\mathfrak{I}_{\mathscr{A},\mathscr{L}}$ vary from alphabet to alphabet and from programming formalism to programming formalism due to the software-hardware duality principle of computer science: specific instructions are written to run on devices designed to perform those and only those instructions. E.g., while one formalism may restrict its set of instructions to addition and subtraction of natural numbers, another formalism may be designed for a device that adds, subtracts, multiplies, and divides discretely finite real numbers.

Definition 9 (Formable Instruction). *If* $\mathfrak{i}_k \in \tilde{\mathfrak{I}}_\mathfrak{j} \in \mathfrak{I}_{\mathscr{A},\mathscr{L}}$, *then*

$$\exists_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}(\mathfrak{i}_k, t) \quad \equiv \quad t \in \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} \quad \wedge \quad \bowtie_{\mathscr{A},\mathscr{L}}(t, \mathfrak{i}_k),$$

in which case $\mathfrak{i}_k$ *is an instruction in* $\mathscr{L}$ *on* $\mathscr{A}$ *formable or, equivalently, generatable by* $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$. $\exists_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ *is symmetric.*

Definition 10 (Transformable Instruction). *If* $\mathfrak{i}_k \in \tilde{\mathfrak{I}}_\mathfrak{j} \in \mathfrak{I}_{\mathscr{A},\mathscr{L}}$, *then*

$$\circlearrowleft_{\mathscr{A},\mathscr{L}}(\mathfrak{i}_k, t, t') \quad \equiv \quad \exists_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}(\mathfrak{i}_k, t) \quad \wedge \quad \vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}} \left(t, t'\right), t \in \mathscr{A}^+_\mathscr{L}, t' \notin \mathtt{ERR}_{\mathscr{A},\mathscr{L}},$$

in which case, $\mathfrak{i}_k$ *is an instruction in* $\mathscr{L}$ *on* $\mathscr{A}$ *transformable by* $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$.

The next axiom is rooted in the existence of general programming languages such as `Lisp` and `Perl`.

SCT Axiom 5. *There exists* $\mathscr{L}$ *on* $\mathscr{A}$ *in which the instruction types in Table A1 in the Appendix A are spatiotemporally finitely formable and transformable.*

SCT Axiom 5 agrees with CCT Axiom A9 in that the computational capacity of the computing agent is always finite, i.e., any agent, physical or abstract, can execute only a finite number of instruction types. However, SCT Axiom 5 differs from CCT Axiom A9, because it makes it explicit that the computational capacity of a computing agent is a characteristic of not just the computing agent, but also of $\mathscr{L}$ on $\mathscr{A}$. In this sense, SCT Axiom 5 offers yet another take on the software-hardware duality principle: programming formalisms are designed for computational devices and computational devices are constructed to perform computations prescribed by programs signified with specific formalisms. We will hereafter refer to a tuple-sufficient $\mathscr{L}$ on $\mathscr{A}$ that satisfies SCT Axiom 5 as *minimally adequate*. E.g., `Lisp`, `Perl`, and `C/C++ Unicode` are minimally adequate.

Definition 11 (Formable Program). *Let* $\mathscr{L}$ *on* $\mathscr{A}$ *be minimally adequate. Let* $m, n, q \in \mathbb{Z}^+$, $1 \leq j \leq n$. *If* $\mathfrak{z}_1 \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow 1, \ldots, \mathfrak{z}_n \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow n$ *and* $(\forall j)(\exists m)\left\{ \circlearrowleft_{\mathscr{A},\mathscr{L}}(\mathfrak{i}_m, t_{i_j}, t'_{i_j}) \wedge \mathfrak{i}_m \in \tilde{\mathfrak{I}}_\mathfrak{q} \wedge \tilde{\mathfrak{I}}_\mathfrak{q} \in \mathfrak{I}_{\mathscr{A},\mathscr{L}} \right\}$, *then*

$$P_k = \bigoplus\nolimits_{\mathscr{A},\mathscr{L}} \left(\trianglelefteq, \mathfrak{z}_1, \mp, t_{i_1}, \trianglerighteq, \mp, \ldots, \mp \trianglelefteq, \mathfrak{z}_n, \mp, t_{i_n} \trianglerighteq \right),$$

where $t_{i_j} \in \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$, *is a program formable by* $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$. *The function* $\mathtt{ni}(P_k) = n$ *denotes the number of the instructions in* P_k. *The predicate*

$$\succ_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}(P_k)$$

holds if, and only if, P_k *is formable by* $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$. *The set*

$$\mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} = \left\{ P_k \in \mathscr{A}^+_\mathscr{L} \middle| \succ_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}(P_k) \right\}$$

is the enumerably infinite set of programs in $\mathscr{L}$ *on* $\mathscr{A}$ *formable by* $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$.

Thus, P_k is formable by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ or, in symbols, $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ if, and only if, it is a finite concatenation of the enumerated formable and transformable instruction texts in $\mathscr{L}$ on

$\mathscr{A}$. E.g., P_j in Figure 1 is a `Lisp` program on `Unicode` formable by the GNU CLISP 2.49.60+ disassembler.

Definition 11 allows us to give a programmatic illustration of SCT Axiom 3 in Figure 2, where the bottom box contains a `Lisp` program on `Unicode` formed by the GNU CLISP 2.49.60+ disassembler that no `Lisp` transformer $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ whose $\mathfrak{R}_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}$ consists of the standard transformation rules of the Lisp read-eval-print loop can transform into a text on `Unicode` within a finite amount of physical time. However, each individual enumerated instruction text of P_k in Figure 2, is transformable by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ within a finite amount of physical time.

```
"(defun natural-number-p (n) (and (typep n 'integer) (>= n 0)))
 (deftype natural-number () '(satisfies natural-number-p))
 (defun f (n)
   (assert (typep n 'natural-number))
   (let ((n+1 (1+ n)))
     (assert (typep n+1 'natural-number))
     n+1))"
```

```
"1     (JMP L7)
 2     L2
 3     (CONST&PUSH 1)          ; (TYPEP N 'NATURAL-NUMBER)
 4     (CALL1&PUSH 2)          ; SYSTEM::ASSERT-ERROR-STRING
 5     (CALL1 3)               ; SYSTEM::SIMPLE-ASSERT-FAILED
 6     L7
 7     (LOAD&PUSH 1)
 8     (CALL1&JMPIFNOT 0 L2)   ; NATURAL-NUMBER-P
 9     (LOAD&INC&PUSH 1)
 10    (JMP L20)
 11    L15
 12    (CONST&PUSH 4)          ; (TYPEP N+1 'NATURAL-NUMBER)
 13    (CALL1&PUSH 2)          ; SYSTEM::ASSERT-ERROR-STRING
 14    (CALL1 3)               ; SYSTEM::SIMPLE-ASSERT-FAILED
 15    L20
 16    (LOAD&PUSH 0)
 17    (CALL1&JMPIFNOT 0 L15) ; NATURAL-NUMBER-P
 18    (POP)
 19    (SKIP&RET 2)"
```

Figure 1. (**Top box**) A definition in Lisp on Unicode of a strongly typed function $f(n) = n + 1$ on natural numbers, from which P_j in the bottom box is automatically formed by the GNU CLISP 2.49.60+ disassembler. (**Bottom box**) The program P_j in Lisp on Unicode is a concatenation of enumerated instruction texts generated from the definition of the function f in the top box by the GNU CLISP 2.49.60+ disassembler. The statements to the right of the semicolons, which are single line comment markers in Lisp, are comments automatically generated by the disassembler.

```
"(loop while (= 3.141592653 3.141592653) do (+ 13.7 13.7))"
```

```
"1   L0
 2   (T)
 3   (SKIP&RET 1)
 4   (CONST 0)       ; 27.4
 5   (SKIP&RET 1)
 6   (JMP L0)"
```

Figure 2. (**Top box**) A definition in Lisp on Unicode of an infinite loop, from which the Lisp program P_k in the bottom box is automatically formed by the GNU CLISP 2.49.60+ disassembler. (**Bottom box**) The program P_k in Lisp on Unicode is a concatenation of enumerated instruction texts formed from the definition of the loop in the top box by the GNU CLISP 2.49.60+ disassembler. The statements to the right of the semicolons, which are single line comment markers in Lisp, are comments automatically generated by the disassembler.

5. Program States and Computations

To transform $P_j \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ deterministically into an output text designating the result of the computation formalized in P_j on a singified input, $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ goes through a sequence of *states* of P_j. Hence,

Definition 12 (Program State). *Let $P_j \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$. Then, $\sigma \in \mathscr{A}^{+}_{\mathscr{L}}$ is a valid state of P_j if, and only if, $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ can determine from σ: (a) the numeral designating the number $1 \leq i \leq \mathtt{ni}(P_j)+1$ of the instruction in P_j to perform; (b) the numeral designating the total number of the instructions of P_j already performed; (c) the texts designating the type and value of every variable name in P_j, including the type and value of the unique output variable denoted by $\square_j$; (d) the text P_j; and (e) for every variable name in σ, $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ can assign exactly one type and exactly one value. If σ is a valid state, then P_j contained in σ is referred to as P_j of σ, while σ is referred to as σ of P_j.*

E.g., a valid state σ_{i_1} of P_j in Figure 1, constructed according to the Common Lisp syntactic conventions, is given in (23), where the symbol `"..."` in the right-hand side of the equality designates the text of the lines 2–18 and $\square_j =$ `"2"`, i.e., `"2"` is the numeral designating the number of the output register in this case.

$$\begin{aligned}
\sigma_{i_1} \;=\;& \texttt{"(1 0 (natural-number-p n) (n 12)"} \;\oplus_{\mathscr{A},\mathscr{L}}\\
& \texttt{"(natural-number-p 2) (2 0)"} \;\oplus_{\mathscr{A},\mathscr{L}}\\
& \texttt{"("} \;\oplus_{\mathscr{A},\mathscr{L}}\; P_j \;\oplus_{\mathscr{A},\mathscr{L}}\; \texttt{")"} \;\oplus_{\mathscr{A},\mathscr{L}}\; \texttt{")"}\\
=\;& \texttt{"(1 0 (natural-number-p n) (n 12)"} \;\oplus_{\mathscr{A},\mathscr{L}}\\
& \texttt{"(natural-number-p 2) (2 0)"} \;\oplus_{\mathscr{A},\mathscr{L}}\\
& \texttt{"(1 (JMP L7) ... 19 (SKIP\&RET 2)))"}\\
=\;& \texttt{"(1 0 (natural-number-p n) (n 12)}\\
& \texttt{(natural-number-p 2) (2 0)}\\
& \texttt{(1 (JMP L7) ... 19 (SKIP\&RET 2)))"}
\end{aligned} \tag{23}$$

$\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ mechanically obtains from σ_{i_1} in (23) the following elements:

1. `"1"` — designates the instruction number to perform;
2. `"0"` — designates the total number of instructions already performed;
3. `"(natural-number-p n)"` — designates the type of the input variable `"n"`;
4. `"(n 12)"` — designates that the current value of `"n"` is `"12"` that, according to the standard decimal notation in `Lisp` on `Unicode`, designates the natural number 12;

5. `"(natural-number-p 2)"`—designates the type specification of the output variable;
6. `"(2 0)"` — designates that the current value of the output variable named `"2"` is `"0"`;
7. `"(1 (JMP L7) ... 19 (SKIP&RET 2))"`— designates that the nineteen instructions of P_j are enumerated with `"1"`, `"2"`, ..., `"19"` that designate the natural numbers $1, 2, \ldots, 19$ in `Lisp` on `Unicode`.

Thus, by Definition 12, P_j in Figure 1 is the program of σ_{i_i} in (23) and σ_{i_1} is a valid state of P_j.

Definition 12 and valid program states like the state in (23) imply the existence of several predicates and functions that we now proceed to define to subsequently formalize the concept of transformation of programs in $\mathfrak{P}_{\hat{F}_{\mathscr{A},\mathscr{L}}}$.

The function in (24) (`VLDS` abbreviates *Valid State*) returns `"1"`, if σ is a valid program state, and `"0"` otherwise.

$$\begin{aligned} &\mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) : \mathscr{A}^{+} \mapsto \{\mathtt{"0"},\mathtt{"1"}\}; \\ &\mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = \begin{cases} \mathtt{"1"} & \text{if } \sigma \text{ is a valid program state;} \\ \mathtt{"0"} & \text{otherwise,} \end{cases} \end{aligned} \tag{24}$$

where `"1"` is arbitrarily interpreted as *True* and `"0"` as *False*. E.g., if σ_1 is a valid program state and σ_2 is not, then

$$\begin{aligned} \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma_1) &\leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow 1; \\ \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma_2) &\leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow 0, \end{aligned} \tag{25}$$

which can be, respectively, abbreviated as

$$\begin{aligned} \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma_1) &= 1; \\ \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma_2) &= 0. \end{aligned} \tag{26}$$

The function in (27) (`PROG` abbreviates *Program*) returns the text of the program of σ, if the latter is valid, and t_{err} otherwise.

$$\begin{aligned} &\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma) : \mathscr{A}^{+}_{\mathscr{L}} \mapsto \mathscr{A}^{+}_{\mathscr{L}} \cup \{t_{err}\}; \\ &\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma) = \begin{cases} t & \text{if } \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1; \\ t_{err} & \text{if } \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 0, \end{cases} \end{aligned} \tag{27}$$

where t is the program of σ, $t_{err} \in \mathtt{ERR}_{\mathscr{A},\mathscr{L}}$. For example, if σ_{i_1} is given in (23) and P_j is given in Figure 1, then $\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}) = P_j$.

The function in (28) (`NPIS` abbreviates *Number of Program Instructions*) returns the numeral designating the number of the instructions in $\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$ if σ is valid and t_{err} otherwise.

$$\begin{aligned} &\mathtt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma) : \mathscr{A}^{+}_{\mathscr{L}} \mapsto \mathscr{A}^{+}_{\mathscr{L}} \cup \{t_{err}\}; \\ &\mathtt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma) = \begin{cases} t & \text{if } \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1; \\ t_{err} & \text{if } \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 0, \end{cases} \\ &\mathtt{where}\ t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow n \in \mathbb{Z}^{+}, t_{err} \in \mathtt{ERR}_{\mathscr{A},\mathscr{L}}. \end{aligned} \tag{28}$$

E.g., for σ_{i_1} in (23), $\texttt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}) = \texttt{"19"} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow 19$. If, for some valid σ, $\texttt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}) = t \in \mathscr{A}_{\mathscr{L}}^{+}$ and $t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow z \in \mathbb{Z}^{+}$, we abbreviate $\texttt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma) = t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow z$ to $\texttt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma) = z = \texttt{ni}(\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma))$ (cf. (11)).

The function in (29) (CINS abbreviates *Current Instruction*) returns the numeral designating the number of the instruction to execute in $\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid, and t_{err} otherwise.

$$\texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma) : \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}\};$$

$$\texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma) = \begin{cases} t & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 0, \end{cases} \tag{29}$$

$$\texttt{where } t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow n \in \mathbb{Z}^{+}, t_{err} \in \texttt{ERR}_{\mathscr{A},\mathscr{L}}.$$

E.g., for σ_{i_1} in (23), $\texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}) = \texttt{"1"} \leftarrow (=) \rightarrow 1 \in \mathbb{Z}^{+}$ or, more succinctly, $\texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1$.

The function in (30) is a side-effect function that sets the value returned by $\texttt{CINS}_{\mathscr{A},\mathscr{L}}$ to the numeral t.

$$\texttt{SETCINS}_{\mathscr{A},\mathscr{L}}(\sigma, t) : \mathscr{A}_{\mathscr{L}}^{+} \times \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}\};$$

$$\texttt{SETCINS}_{\mathscr{A},\mathscr{L}}(\sigma, t) = \begin{cases} \sigma' & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1 \wedge \\ & \quad t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow z \wedge \\ & \quad 1 \leq z \leq \texttt{ni}(\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)) + 1; \\ t_{err} & \text{otherwise.} \end{cases} \tag{30}$$

$$\texttt{where } \texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma') = t, t_{err} \in \texttt{ERR}_{\mathscr{A},\mathscr{L}}.$$

The function in (31) (TOTI abbreviates *Total Instructions*) returns the numeral designating the total number of the transformed (or, equivalently, performed or executed) instruction texts in $\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$ if σ is valid and t_{err} otherwise.

$$\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma) : \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}\};$$

$$\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma) = \begin{cases} t & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 0, \end{cases} \tag{31}$$

$$\texttt{where } t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow n \in \mathbb{Z}^{+}, t_{err} \in \texttt{ERR}_{\mathscr{A},\mathscr{L}}.$$

E.g., for σ_{i_1} in (23), $\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}) = \texttt{"0"} \leftarrow (=) \rightarrow 0$. If, for some valid σ, $\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma) = t \in \mathscr{A}_{\mathscr{L}}^{+}$ and $t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow z \in \mathbb{Z}^{+}$, we abbreviate $\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma) = t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow z$ to $\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma) = z$.

The next function in (32) (VVRS abbreviates *Values of Variables*) returns a marked concatenation designating a tuple of 2-tuples, each of which designates a variable and its numerical value. Toward that end, we let σ be valid and let $\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$ contain $l > 0$ variables, including the unique output variable designated by $\square_j$. We let $\mathbf{x}_i \in \mathscr{A}_{\mathscr{L}}^{+}$, $1 \leq i \leq l$, designate the i-th variable and $\mathbf{z}_i \in \mathscr{A}_{\mathscr{L}}^{+}$, $1 \leq i \leq l$, designate the variable's numerical value, i.e., $\mathbf{z}_i \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r \in \mathbb{R}$, $1 \leq i \leq l$. We let $\trianglelefteq, \trianglerighteq, \mp \in \mathscr{A}_{\mathscr{L}}^{+}$ designate the beginning of a tuple, the end of a tuple, and the separator sign, respectively. With these conventions in place, the function in (32) returns the text designating the tuple of 2-tuples, each of which

designates a variable and its numerical value in $\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$, when σ is valid, and t_{err} otherwise.

$$\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma) : \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}\};$$

$$\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma) = \begin{cases} t & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 0, \end{cases} \tag{32}$$

$$\texttt{where } t = \bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\left(\trianglelefteq, t_1, \mp, t_2, \mp, \ldots, \mp, t_l, \trianglerighteq\right), t_{err} \in \texttt{ERR}_{\mathscr{A},\mathscr{L}},$$
$$t_i = \bigoplus\nolimits_{\mathscr{A},\mathscr{L}}\left(\trianglelefteq, \mathbf{x}_i, \mp, \mathbf{z}_i \trianglerighteq\right), 1 \leq i \leq l, l > 0.$$

E.g., if we recall that in `Lisp` $\trianglelefteq$ = `"("`, $\trianglerighteq$ = `")"`, and $\mp$ is any non-empty sequence of white space Unicode characters, then $\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1})$ = `"((`n `2) (`$\square_j$ `0))"`.

The function in (33) (`OUTV` abbreviates *Output Value*) returns the numeral designating the value of the output variable of $\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid, and t_{err} otherwise.

$$\texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma) : \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}\};$$

$$\texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma) = \begin{cases} t & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 0, \end{cases} \tag{33}$$

$$\texttt{where } t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r \in \mathbb{R}, t_{err} \in \texttt{ERR}_{\mathscr{A},\mathscr{L}}.$$

E.g., $\texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1})$ = `"0"` $\leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow 0$. If, for some valid σ, $\texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma) = t \in \mathscr{A}_{\mathscr{L}}^{+}$ and $t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r \in \mathbb{R}$, we abbreviate $\texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma) = t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r$ to $\texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma) = r$. $\texttt{OUTV}_{\mathscr{A},\mathscr{L}}$ takes the output of $\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma)$ (i.e., $\texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma))$) and returns the numeral designating the value of the output variable in the tuple so long as σ is valid.

The function in (34) (`VV` abbreviates *Variable Value*) returns the text designating the value of the i-th variable in $\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma)$ and t_{err} otherwise.

$$\texttt{VV}_{\mathscr{A},\mathscr{L}}(\sigma, t) : \mathscr{A}_{\mathscr{L}}^{+} \times \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}\};$$

$$\texttt{VV}_{\mathscr{A},\mathscr{L}}(\sigma, t) = \begin{cases} t' & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 1; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) = 0, \end{cases} \tag{34}$$

$$\texttt{where } t \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow i, 1 \leq i \leq l, t_{err} \in \texttt{ERR}_{\mathscr{A},\mathscr{L}}, t' \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r \in \mathbb{R}.$$

We now define a useful variant of $\texttt{VV}_{\mathscr{A},\mathscr{L}}$ to obtain the numerals designating the values of the input variables for any program that takes $k \in \mathbb{Z}^{+}$ input texts. If $\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma)$ contain $l > 0$ variable-value tuples, of which the first $0 < k < l$ tuples, by convention, designate the names and values of the input variables. The function in (35) (`IVV` abbreviates *Input Variable Value*) returns the numeral designating the value of the i-th input variable in $\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid, and t_{err} otherwise.

$$\texttt{IVV}_{\mathscr{A},\mathscr{L}}(\sigma,t):\mathscr{A}^{+}_{\mathscr{L}}\times\mathscr{A}^{+}_{\mathscr{L}}\mapsto\mathscr{A}^{+}_{\mathscr{L}}\cup\{t_{err}\};$$

$$\texttt{IVV}_{\mathscr{A},\mathscr{L}}(\sigma,t)=\begin{cases}t' & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma)=1;\\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma)=0,\end{cases}\tag{35}$$

$$\texttt{where } t\leftarrow(=)_{\mathscr{A},\mathscr{L}}\rightarrow i,1\leq i\leq l,t_{err}\in\texttt{ERR}_{\mathscr{A},\mathscr{L}},t'\leftarrow(=)_{\mathscr{A},\mathscr{L}}\rightarrow r\in\mathbb{R}.$$

The function in (36) (NXTS abbreviates *Next State*) returns the next valid program state of $\texttt{PROG}(\sigma)_{\mathscr{A},\mathscr{L}}$ if σ is valid and an error message otherwise.

$$\texttt{NXTS}_{\mathscr{A},\mathscr{L}}(\sigma):\mathscr{A}^{+}_{\mathscr{L}}\mapsto\mathscr{A}^{+}_{\mathscr{L}}\cup\{t_{err}\};$$

$$\texttt{NXTS}_{\mathscr{A},\mathscr{L}}(\sigma)=\begin{cases}tr_j(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma)=1;\\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma)=0,\end{cases}\tag{36}$$

$$t_{err}\in\texttt{ERR}_{\mathscr{A},\mathscr{L}},tr_j\in\mathfrak{R}_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}.$$

This function looks at the current instruction and deterministically constructs the next valid state σ' from the type of the instruction and the variables used in the instruction in the current program state σ so long as that latter is valid, or, equivalently, it transforms σ into σ'. The choice of the transformation rule tr_j is determined by σ. Since the next valid state $tr_j(\sigma)=\sigma'$ (the notation $tr_j(\sigma)=\sigma'$ denotes the application of tr_j to σ to compute σ') necessarily depends on the specifics of $\mathscr{L}$ on $\mathscr{A}$, we offer an intuitive and abstract outline of how $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ uses $\texttt{NXTS}_{\mathscr{A},\mathscr{L}}$ to transform σ to σ'.

$\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ transforms the text of the instruction whose number is designated by $\texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma)$ according to its type (cf. Table A1) and then (1) updates, if necessary, the value numerals of the variable names involved in this instruction, i.e., updates the text returned by $\texttt{VVRS}_{\mathscr{A},\mathscr{L}}(\sigma)$; (2) updates the numeral designating the number of the instruction to execute in σ'; and (3) updates the numeral designating the total number of the instructions (or, to be more exact, instruction texts) of $\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$ transformed thus far to the numeral designating the value of the total number incremented by 1. Per Definition 6, if σ is valid, then we define $\texttt{NXTS}^{0}_{\mathscr{A},\mathscr{L}}(\sigma)=\sigma$ and, for $l\in\mathbb{Z}^{+}$, define

$$\texttt{NXTS}^{l}_{\mathscr{A},\mathscr{L}}(\sigma)=tr_{i_l}\Big(tr_{i_{l-1}}\Big(\ldots tr_{i_2}\Big(tr_{i_1}\Big(\sigma\Big)\Big)\ldots\Big)\Big)=\sigma',tr_{i_j}\in\mathfrak{R}_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}},1\leq j\leq l.\tag{37}$$

Thus, we have

Lemma 1. *Let $\mathscr{L}$ on $\mathscr{A}$ be minimally adequate and let $\mathscr{A}$ contain the signs of the standard decimal notation. If $\texttt{NXTS}_{\mathscr{A},\mathscr{L}}(\sigma)=\sigma'\neq t_{err}$, then $\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma)\leftarrow(=)_{\mathscr{A},\mathscr{L}}\rightarrow n$ and $\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma')\leftarrow(=)_{\mathscr{A},\mathscr{L}}\rightarrow n+1$, $n\in\mathbb{N}$.*

Let us assume that a minimally adequate $\mathscr{L}$ on $\mathscr{A}$, like many modern programming languages, has at least four types of numerical instructions on the numerals in the standard decimal notation: addition, multiplication, subtraction, and division applicable to finitely many discretely finite real numbers. Then, if the current instruction is addition, the numerals designating the values of two or more variable names are added and the numeral of the sum is assigned to a variable name, the numeral value of which is accordingly updated in σ'. The numerals designating the values of the variable names that are not affected by the instruction text, remain in σ' exactly as they are in σ. The transformation

of the multiplication and subtraction instruction types is performed analogously. The transformation of the division instructions is performed with the convention that the division by zero (or, to be more exact, by the numeral designating 0 in $\mathscr{L}$ on $\mathscr{A}$) results in an error message.

If $\mathscr{L}$ on $\mathscr{A}$ has the assignment instruction type whereby the value of a variable is set to a value, then the numeral value of that variable's name is accordingly updated in σ'. If $\mathscr{L}$ has a GOTO/JUMP instruction type and the current instruction prescribes a jump to a label (e.g., `"(JMP L7)"` in P_j in Figure 1 prescribes a jump to the label `"L7"`), then no updates of the variable numeral values are performed, and the numeral of the next instruction to execute in σ' is set to the numeral of the program line with that label. If multiple instruction texts of $\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$ have the same label, the numeral of the instruction to execute in σ' is set to the numeral designating the smallest such number. If the current instruction calls for a jump to a label that does not exist in $\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$, then the numeral designating the number of the instruction to execute in σ' is set to the numeral designating $\mathtt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma)+1$. The transformation of a dispatch to a numeral designating an instruction number is performed analogously to the transformation of a dispatch to a label. Thus,

Lemma 2. *If* $\mathtt{NXTS}_{\mathscr{A},\mathscr{L}}(\sigma)=\sigma' \neq t_{err}$, *then* $\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)=\mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma')$.

Associated with the execution of $P \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ are two states: a program start state and a program end state. The predicate in (38) (`STARTS` abbreviates *Start State*) returns `"1"` if σ is a valid program start state and `"0"` otherwise.

$$\mathtt{STARTS}_{\mathscr{A},\mathscr{L}}(\sigma): \mathscr{A}^{+} \mapsto \{\texttt{"0"},\texttt{"1"}\};$$

$$\mathtt{STARTS}_{\mathscr{A},\mathscr{L}}(\sigma)=\begin{cases}\texttt{"1"} & \text{if } \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma)=1 \ \wedge \ \mathtt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma)=1 \ \wedge \\ & \quad \mathtt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma)=0; \\ \texttt{"0"} & \text{otherwise.}\end{cases} \tag{38}$$

The predicate in (39) (`ENDS` abbreviates *End State*) returns `"1"` if σ is a valid end state and `"0"` otherwise.

$$\mathtt{ENDS}_{\mathscr{A},\mathscr{L}}(\sigma): \mathscr{A}^{+}_{\mathscr{L}} \mapsto \{\texttt{"0"},\texttt{"1"}\};$$

$$\mathtt{ENDS}_{\mathscr{A},\mathscr{L}}(\sigma)=\begin{cases}\texttt{"1"} & \text{if } \mathtt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma)=\mathtt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma)+1; \\ \texttt{"0"} & \text{otherwise.}\end{cases} \tag{39}$$

Suppose that $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ starts in a valid start state $\sigma_{i_1} \in \mathfrak{S}_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}$ such that $P_k = \mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1})$, and then uses a sequence of its transformation rules through $\mathtt{NXTS}_{\mathscr{A},\mathscr{L}}$ (cf. Equation (37)) to transform σ_{i_1} into σ_{i_2}, and so on, thus generating a finite sequence of states $\sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_l}$, $1<l\in\mathbb{Z}^{+}$, where σ_{i_l} and σ_{i_l} are valid start and end states of P_k, respectively, then $\mathtt{NXTS}^{l-1}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1})=\sigma_{i_l}$. Hence,

Definition 13 (Signifiable Computation). *A signifiable computation of* $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ *is a finite sequence of valid program states* $\sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_l}$, $1\leq j<l$, $1<l\in\mathbb{Z}^{+}$, $\sigma_{i_1}\in\mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$, *such that*

$$\begin{array}{lll}\mathtt{STARTS}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1})=1 & \wedge & \mathtt{ENDS}_{\mathscr{A},\mathscr{L}}(\sigma_{i_l})=1 \quad \wedge \\ \mathtt{NXTS}_{\mathscr{A},\mathscr{L}}(\sigma_{i_j})=\sigma_{i_{j+1}} & \wedge & \mathtt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma_{i_l})=P_k.\end{array}$$

The predicate

$$\texttt{SGNCOMP}_{\mathscr{A},\mathscr{L}}(P_k,(\sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_l}))$$

holds if, and only if, $\sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_l}$ *is a signifiable computation of* P_k *in* $\mathscr{L}$ *on* $\mathscr{A}$.

Hence,

Lemma 3. *Let* $\texttt{SGNCOMP}_{\mathscr{A},\mathscr{L}}(P_k,(\sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_l}))$. *Then,* $\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}(\sigma_{i_j},\sigma_{i_{j+1}}), 1 \leq j < l \in \mathbb{Z}^+$, *and* $\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}(\sigma_{i_1},\sigma_{i_l})$.

6. Transformation of Programs

Definition 14 (Transformation of Programs)**.** *Let* $l \in \mathbb{Z}^+$, $1 \leq j \leq n \in \mathbb{Z}^+$, $t_j \in \mathscr{A}^+_{\mathscr{L}}$, $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$, *and* $\sigma_{i_1} \in \mathfrak{S}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$. *Then,* P_k *is transformable by* $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ *from* σ_{i_1}, *which is denoted as* $\prec_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}(P_k,\sigma_{i_1})$, *if, and only if,* $(\exists\ \sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_n})\ \ \texttt{SGNCOMP}_{\mathscr{A},\mathscr{L}}(P_k,(\sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_n}))$.

The next lemma follows from Definition 14 and Lemmas 2 and 3.

Lemma 4. $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ *and* $\texttt{SGNCOMP}_{\mathscr{A},\mathscr{L}}(P_k,(\sigma_{i_1},\sigma_{i_2},\ldots,\sigma_{i_l}))$, $1 < l \in \mathbb{Z}^+$. *Then, the following statements are equivalent, for* $1 \leq j < l$*:*

1. $\prec_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}(P_k,\sigma_{i_1})$;
2. $\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}(\sigma_{i_j},\sigma_{i_j+1})$;
3. $\vdash_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}(\sigma_{i_1},\sigma_{i_l})$;
4. $\texttt{NXTS}^{l-1}(\sigma_{i_1}) = \sigma_{i_l} \neq t_{err}$.

A question naturally arises whether it is possible for $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ to transform, in finitely many steps, a valid start state σ_{i_1} of a program P_k into another valid state of P_k when P_k is not formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$. In other words, can $\prec_{\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}}(P_k,\sigma_{i_j})$ hold if $\texttt{PROG}(\sigma_{i_j}) = P_k$ but $P_k \notin \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$? Definition 11 implies a negative answer, because for the program to be transformable by $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ from a valid start state σ_{i_1}, every enumerated instruction text in P_k must be transformable. $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ and $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ are co-dependent in that $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ cannot place in a program an instruction text that $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ cannot transform.

E.g., consider a `Lisp` program by Steele in Figure 3. This text is formally impeccable insomuch as it is formed according to the `Lisp` syntactic conventions. However, the program is not transformable unless one supplies the definitions of the functions `stroke`, `stuck`, `make-oar`, `winded`, and `dream`, all of which, in turn, can be converted by the Lisp disassembler into finite sequences of instruction texts that $\hat{\hat{T}}_{\texttt{Unicode,Lisp}}$ can transform with the rules of the read-eval-print loop from a valid start state for this program. For example, if $\hat{\hat{T}}_{\texttt{Unicode,Lisp}}$ is GNU CLISP 2.49.60+, then the attempted transformation of `"(life)"` results in

$$t_{err} = \texttt{"*** - EVAL: undefined function ROW"} \in \texttt{ERR}_{\texttt{Unicode,Lisp}},$$

whence

$$\neg \prec_{\texttt{Unicode,Lisp}} (\texttt{"(life)"},\sigma_{i_1}).$$

```
"(defun gently (oar)
   (stroke oar :force 0.5)
   (when (stuck oar) (throw 'crab nil)))

 (defun row (boat stroken-fn stream &key count)
   (let ((oar (make-oar boat stream)))
     (loop repeat count do (funcall stroke-fn oar))))

 (defun life ()
   (catch 'crab
     (catch 'breath
       (unwind-protect
         (row "your boat" #'gently *query-io* :count 3)
         (when (winded) (throw 'breath nil)))
       (loop repeat 4 do (set-mode :merry))
       (dream))))"
```

```
"1     (CONST&PUSH 0)  ; LIFE
 2     (CALL1 1)       ; SYSTEM::REMOVE-OLD-DEFINITIONS
 3     (CONST&PUSH 0)  ; LIFE
 4     (CONST&PUSH 2)  ; #<COMPILED-FUNCTION LIFE>
 5     (CALLS2 156)    ; SYSTEM::%PUTD
 6     (CONST 0)       ; LIFE
 7     (SKIP&RET 1)"
```

Figure 3. (**Top box**) A Lisp program by Guy Steele on p. 191 in [9] to illustrate the operational semantics of catch and throw with unwind-protect. (**Bottom box**) The program generated from the definition of the Lisp function $life$ in the top box by the GNU CLISP 2.49.60+ disassembler. The statements to the right of the semicolons, which are single line comment markers in Lisp, are comments automatically generated by the disassembler.

The above observations lead to the next axiom.

SCT Axiom 6. *There exists a minimally adequate $\mathscr{L}$ on $\mathscr{A}$, where any $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ is reducible to an enumerated sequence of instruction texts from a finite set of transformable instruction types that are irreducible to any other instruction types.*

SCT Axiom 6 does not appear to have a direct equivalent among the CCT axioms. It is related to CCT Axiom A9 insomuch as it reflects the fact that the computational capacity of any computing agent is finite. This axiom reflects a common fact of modern compiler theory: all higher level constructs (e.g., function definitions and macros) of programming languages are reduced to a finite set of irreducible (also known as *primitive*) instructions (cf., e.g., Ch. 3 in [1]). Therefore, for any $\mathscr{L}$ on $\mathscr{A}$ compliant with SCT Axiom 6, we can assume that a $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ is a concatenation of enumerated instruction texts, each of which designates an irreducible instruction type. We will hereafter assume that $\mathscr{L}$ on $\mathscr{A}$ is minimally adequate and satisfies SCT Axiom 6.

7. Signifiable Computability of Functions

To characterize functions that are signifiably computable in principle, we remove memory limitations imposed on computation by an FMD $\mathscr{D}$, which is formalized in our next definition.

Definition 15 (Signifiably Computable Function). *A total function $f : \mathbb{R}^m \mapsto \mathbb{R}$ is signifiably computable in $\mathscr{L}$ on $\mathscr{A}$ if, and only if, there exists $P_k \in \mathfrak{P}_{\hat{F}_{\mathscr{A},\mathscr{L}}}$ and, for $1 < n \in \mathbb{Z}^+$, $1 \leq j \leq m \in \mathbb{Z}^+$, $(r_1, \ldots, r_m) \in \mathbb{R}^m$,*

$$(\forall\ (r_1, \ldots, r_m) \in dom(f))\ f(r_1, \ldots, r_m) = r$$

if, and only if,

$$\begin{array}{ll} (\exists\ (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n}))\ \mathtt{SGNCOMP}_{\mathscr{A},\mathscr{L}}(P_k, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n})) & \wedge \\ \mathtt{IVV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}, t_{i_j}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r_j & \wedge \\ j \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_{i_j} & \wedge \\ \mathtt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_n}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r. & \end{array}$$

The predicate $\mathtt{SGNCF}_{\mathscr{A},\mathscr{L}}(f, P_k)$ holds if, and only if, f is signifiably computable in $\mathscr{L}$ on $\mathscr{A}$ and P_k is a program that signifies f in $\mathscr{L}$ on $\mathscr{A}$.

We proceed to characterize functions that are signifiably partially computable in principle.

Definition 16 (Signifiably Partially Computable Function). *A partial function $f : \mathbb{R}^m \mapsto \mathbb{R}$ is signifiably partially computable in $\mathscr{L}$ on $\mathscr{A}$ if, and only if, there exists $P_k \in \mathfrak{P}_{\hat{F}_{\mathscr{A},\mathscr{L}}}$ and, for $1 < n \in \mathbb{Z}^+$, $1 \leq j \leq m \in \mathbb{Z}^+$, $(r_1, \ldots, r_m) \in \mathbb{R}^m$,*

$$(r_1, \ldots, r_m) \in dom(f) \quad \wedge \quad f(r_1, \ldots, r_m) = r$$

if, and only if,

$$\begin{array}{ll} (\exists\ (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n}))\ \ \mathtt{SGNCOMP}_{\mathscr{A},\mathscr{L}}(P_k, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n})) & \wedge \\ \mathtt{IVV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}, t_{i_j}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r_j & \wedge \\ j \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_{i_j} & \wedge \\ \mathtt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_n}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r. & \end{array}$$

The predicate $\mathtt{SGNPCF}_{\mathscr{A},\mathscr{L}}(f, P_k)$ holds if, and only if, f is signifiably partially computable in $\mathscr{L}$ on $\mathscr{A}$ and P_k is a program that signifies f in $\mathscr{L}$ on $\mathscr{A}$.

Definitions 15 and 16 imply

Lemma 5. *If $\mathtt{SGNCF}_{\mathscr{A},\mathscr{L}}(f, P_k)$, then $\mathtt{SGNPCF}_{\mathscr{A},\mathscr{L}}(f, P_k)$.*

Definitions 15 and 16 and Lemma 5 broaden the scope of CCT Axiom A0 by making the properties *computable* and *partially computable* apply to functions on discretely finite real numbers. The next definition states that signifiable functions are those functions that can be signified with texts in $\mathscr{L}$ on $\mathscr{A}$.

Definition 17 (Signifiable Function). *A function f is signifiable in $\mathscr{L}$ on $\mathscr{A}$, which is denoted as $\mathtt{SGNF}_{\mathscr{A},\mathscr{L}}(f)$, if, and only if, there exists $t \in \mathscr{A}^+_{\mathscr{L}}$ such that*

$$f \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t.$$

If $t \leq \mathtt{CCAP}(\mathscr{D})$, for some FMD $\mathscr{D}$, then f is signifiable in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$, which is denoted as $\mathtt{SGNF}_{\mathscr{A},\mathscr{L},\mathscr{D}}(f)$.

E.g., $f : \mathbb{N} \mapsto \mathbb{N}$, $f(x) = x + 1$, then

$\mathtt{SGNF}_{\mathtt{Unicode},\mathtt{Lisp}}(f)$ where $t \in \mathtt{Unicode}^{+}_{\mathtt{Lisp}}$ that signifies f is given in Figure 1.

Lemma 6. *There exist $\mathscr{L}$ on $\mathscr{A}$ and a function f such that*

$$\mathtt{SGNF}_{\mathscr{A},\mathscr{L}}(f) \wedge \left(\forall\, P_k \in \mathfrak{P}_{\hat{\mathtt{F}}_{\mathscr{A},\mathscr{L}}}\right) \neg \mathtt{SGNCF}_{\mathscr{A},\mathscr{L}}(f, P_k).$$

Proof. Let $x \in \mathscr{A}^{+}_{\mathscr{L}}$ such that $x \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r \in \mathbb{R}$. Let $s \in \mathscr{A}^{+}_{\mathscr{L}}$ such that $s \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow 1$. Let

$$f(x, P) : \mathscr{A}^{+}_{\mathscr{L}} \times \mathfrak{P}_{\hat{\mathtt{F}}_{\mathscr{A},\mathscr{L}}} \mapsto \{\mathtt{"0"}, \mathtt{"1"}\};$$

$$f(x, P) = \begin{cases} \mathtt{"1"} & \text{if } \mathtt{SGNCOMP}_{\mathscr{A},\mathscr{L}}(P, (\sigma_{i_1}, \ldots, \sigma_{i_n})); \\ \mathtt{"0"} & \text{otherwise,} \end{cases} \tag{40}$$

where $\mathtt{IVV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}, s) = x$ and $x \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r$, $r \in \mathbb{R}$. Take $\mathscr{A} = \mathtt{Unicode}$, $\mathscr{L} = \mathtt{Lisp}$, and consider the following $t \in \mathtt{Unicode}^{+}_{\mathtt{Lisp}}$

```
"(defun f (x p) (if (halts-p p x) 1 0))",
```

where `"(halts-p p x)"` returns the text in $\mathtt{Unicode}^{+}_{\mathtt{Lisp}}$ designating the Boolean value *True* if a `Lisp` funcallable object designated by $\mathtt{p} \in \mathtt{Unicode}^{+}_{\mathtt{Lisp}}$ halts on the real number designated by $x \in \mathtt{Unicode}^{+}_{\mathtt{Lisp}}$. Since $t \leftarrow (=)_{\mathtt{Unicode},\mathtt{Lisp}} \rightarrow f$, we have

$$\mathtt{SGNF}_{\mathtt{Unicode},\mathtt{Lisp}}(f).$$

However, since the predicate `halts-p` is not signifiably computable in `Lisp` on `Unicode`,

$$\neg(\exists P_k) P_k \in \mathfrak{P}_{\hat{\mathtt{F}}_{\mathtt{Unicode},\mathtt{Lisp}}} \wedge \mathtt{SGNCF}_{\mathtt{Unicode},\mathtt{Lisp}}(f, P_k).$$

□

8. Signifiable Computability of Functions on Finite Memory Devices

We now proceed to modify some of the definitions in Section 5 for situations when functions signifiable by programs in $\mathscr{L}$ on $\mathscr{A}$ are computed on an FMD $\mathscr{D}$. We associate with every triplet $\mathscr{L}, \mathscr{A}, \mathscr{D}$ a non-empty set of error messages

$$\mathtt{ERR}_{\mathscr{A},\mathscr{L},\mathscr{D}} = \{t_{cap}\} \cup \{t \in \mathtt{ERR}_{\mathscr{A},\mathscr{L}} | \mathtt{CCAP}(\mathscr{D}) \geq |t|\}, \tag{41}$$

where $t_{cap} \in \mathscr{A}^{+}_{\mathscr{L}}$, $|t_{cap}| \leq \mathtt{CCAP}(\mathscr{D})$, $t_{cap} \notin \mathtt{ERR}_{\mathscr{A},\mathscr{L}}$, is a unique error message whose size does not exceed the memory capacity of $\mathscr{D}$. An important difference between the functions and predicates in Section 5 and the functions and predicates in this section is that the latter return t_{cap} when the memory capacity of $\mathscr{D}$ is exceeded.

The function in (42) returns $\mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma)$ if the size of σ does not exceed the memory capacity of $\mathscr{D}$. Otherwise, the function returns t_{cap}.

$$\mathtt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}^{+}_{\mathscr{L}} \mapsto \{\mathtt{"0"}, \mathtt{"1"}, t_{cap}\};$$

$$\mathtt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \mathtt{VLDS}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } |\sigma| \leq \mathtt{CCAP}(\mathscr{D}); \\ t_{cap} & \text{otherwise.} \end{cases} \tag{42}$$

The function in (43) returns $\texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is a valid program state and the size of σ does not exceed the memory capacity of $\mathscr{D}$. Otherwise, t_{err} or t_{cap} are returned.

$$\texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}, t_{cap}\};$$

$$\texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \texttt{PROG}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"0"}; \\ t_{cap} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = t_{cap}. \end{cases} \tag{43}$$

The function in (44) returns $\texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid and the size of σ does not exceed the memory capacity of $\mathscr{D}$. Otherwise, t_{err} or t_{cap} are returned.

$$\texttt{CINS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}^{+} \mapsto \mathscr{A}^{+} \cup \{t_{err}, t_{cap}\};$$

$$\texttt{CINS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \texttt{CINS}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"0"}; \\ t_{cap} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = t_{cap}. \end{cases} \tag{44}$$

The function in (45) returns $\texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid and the size of σ does not exceed the memory capacity of $\mathscr{D}$. Otherwise, t_{err} or t_{cap} are returned.

$$\texttt{TOTI}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}, t_{cap}\};$$

$$\texttt{TOTI}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \texttt{TOTI}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"0"}; \\ t_{cap} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = t_{cap}. \end{cases} \tag{45}$$

The function in (46) $\texttt{STARTS}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid and the size of σ does not exceed the memory capacity of $\mathscr{D}$. Otherwise, the function returns t_{err} or t_{cap}.

$$\texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}^{+} \mapsto \{\texttt{"0"}, \texttt{"1"}, t_{err}, t_{cap}\};$$

$$\texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \texttt{STARTS}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"0"}; \\ t_{cap} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = t_{cap}. \end{cases} \tag{46}$$

The function in (47) returns $\texttt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid and the size of σ does not exceed $\texttt{CCAP}(\mathscr{D})$. Otherwise, t_{err} or t_{cap} is returned.

$$\texttt{NPIS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}_{\mathscr{L}}^{+} \mapsto \mathscr{A}_{\mathscr{L}}^{+} \cup \{t_{err}, t_{cap}\};$$

$$\texttt{NPIS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \texttt{NPIS}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"0"}; \\ t_{cap} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = t_{cap}. \end{cases} \tag{47}$$

The function in (48) returns $\texttt{ENDS}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is a valid program state and the size of σ does not exceed the memory capacity of $\mathscr{D}$. Otherwise, it returns t_{err} or t_{cap}.

$$\texttt{ENDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}^{+} \mapsto \{\texttt{"0"}, \texttt{"1"}, t_{err}, t_{cap}\};$$

$$\texttt{ENDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \texttt{ENDS}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"0"}; \\ t_{cap} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = t_{cap}. \end{cases} \tag{48}$$

The function in (49) returns $\texttt{NXTS}_{\mathscr{A},\mathscr{L}}(\sigma)$, if σ is valid and the size of σ does not exceed the memory capacity of $\mathscr{D}$. Otherwise, t_{err} or t_{cap} are returned.

$$\texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) : \mathscr{A}^{+}_{\mathscr{L}} \mapsto \mathscr{A}^{+}_{\mathscr{L}} \cup \{t_{err}, t_{cap}\};$$

$$\texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \begin{cases} \texttt{NXTS}_{\mathscr{A},\mathscr{L}}(\sigma) & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}; \\ t_{err} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"0"}; \\ t_{cap} & \text{if } \texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = t_{cap}. \end{cases} \tag{49}$$

We now define memory-dependent signifiable computation on discretely finte real numbers in (18). This definition is similar to its memory-independent counterpart in (13), but is defined in terms of memory-dependent functions and predicates defined above in this section.

Definition 18 (Signifiable Computation on Finite Memory)**.** *A signifiable computation of $P_k \in \mathfrak{P}_{\hat{F}_{\mathscr{A},\mathscr{L}}}$ on an FMD $\mathscr{D}$ is a finite sequence of valid program states $\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_l}$, $1 \leq j < l$, $1 < l \in \mathbb{Z}^{+}$, such that*

$$\begin{array}{lcll} \sigma_{i_1} \in \mathfrak{S}_{\hat{F}_{\mathscr{A},\mathscr{L}}} & & & \wedge \\ \texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_l}) & = & \texttt{"1"} & \wedge \\ \texttt{ENDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_l}) & = & \texttt{"1"} & \wedge \\ \texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) & = & \sigma_{i_{j+1}} & \wedge \\ \texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_l}) & = & P_k. & \end{array}$$

The predicate

$$\texttt{SGNCOMP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_l}))$$

holds if, and only if, $\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_l}$ is a signifiable computation of P_k in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$.

Definition 19 (Signifiably Computable Function on Finite Memory)**.** *A total function $f : \mathbb{R}^m \mapsto \mathbb{R}$ is signifiably computable in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$ if, and only if, there exists $P_k \in \mathfrak{P}_{\hat{F}_{\mathscr{A},\mathscr{L}}}$ and, for $1 < n \in \mathbb{Z}^{+}$, $1 \leq j \leq m \in \mathbb{Z}^{+}$, $(r_1, \ldots, r_m) \in \mathbb{R}^m$,*

$$(\forall\ (r_1, \ldots, r_m) \in dom(f))\ f(r_1, \ldots, r_m) = r$$

if, and only if,

$$\begin{array}{l} (\exists\ (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n}))\ \texttt{SGNCOMP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n})) \quad \wedge \\ (\sigma_{i_k}, \sigma_{i_{k+1}}) \leftarrow \dot{-}_{\mathscr{A},\mathscr{L}} \rightarrow s_{k,k+1} \quad \wedge \\ |s_{k,k+1}| \leq \texttt{CCAP}(\mathscr{D}) \quad \wedge \\ \texttt{IVV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}, t_{i_j}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r_j \quad \wedge \\ j \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_{i_j} \quad \wedge \\ \texttt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_n}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r. \end{array}$$

The predicate

$$\texttt{SGNCF}_{\mathscr{A},\mathscr{L},\mathscr{D}}(f, P_k)$$

holds if, and only if, f is signifiably computable in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$ and P_k is a program that signifies f in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$.

The next definition formalizes signifiably computable partial functions when the memory available for computation is confined to the memory capacity of $\mathscr{D}$ on which the signifiable computation is performed.

Definition 20 (Signifiably Partially Computable Function on Finite Memory)**.** *A partial function $f : \mathbb{R}^m \mapsto \mathbb{R}$ is signifiably partially computable in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$ if, and only if, there exists $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ and, for $1 < n \in \mathbb{Z}^+$, $1 \leq k < n$, $1 \leq j \leq m \in \mathbb{Z}^+$, $(r_1, \ldots, r_m) \in \mathbb{R}^m$,*

$$(r_1, \ldots, r_m) \in dom(f) \quad \wedge \quad f(r_1, \ldots, r_m) = r$$

if, and only if,

$$\begin{array}{ll}
(\exists\, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n}))\ \mathtt{SGNCOMP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n})) & \wedge \\
(\sigma_{i_k}, \sigma_{i_{k+1}}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow s_{k,k+1} & \wedge \\
|s_{k,k+1}| \leq \mathtt{CCAP}(\mathscr{D}) & \wedge \\
\mathtt{IVV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_1}, t_{i_j}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r_j & \wedge \\
j \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow t_{i_j} & \wedge \\
\mathtt{OUTV}_{\mathscr{A},\mathscr{L}}(\sigma_{i_n}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r. &
\end{array}$$

The predicate

$$\mathtt{SGNPCF}_{\mathscr{A},\mathscr{L},\mathscr{D}}(f, P_k)$$

holds if, and only if, f is signifiably partially computable in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$ and P_k is a program that signifies f in $\mathscr{L}$ on $\mathscr{A}$ on $\mathscr{D}$.

The above definitions imply that $\mathscr{D}$ can hold $s_{k,k+1}$ in its memory. In other words, the text $s_{k,k+1}$ that signifies two consecutive valid program states in a signifiable computation does not exceed the memory capacity of $\mathscr{D}$. We will hereafter state that such an FMD has a sufficiently large memory capacity, whence the next definition.

Definition 21 (Sufficient Memory Capacity)**.** *Let $\mathscr{D}$ be an FMD and $P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$. Then, $\mathscr{D}$ has a sufficient memory capacity relative to P_k if, for any sequence of states such that*

$$\mathtt{SGNCOMP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_l})),$$

$$(\sigma_{i_k}, \sigma_{i_{k+1}}) \leftarrow \dot{=}_{\mathscr{A},\mathscr{L}} \rightarrow s_{k,k+1} \ \wedge\ |s_{k,k+1}| \leq \mathtt{CCAP}(\mathscr{D}), 1 \leq k < l \in \mathbb{Z}^+.$$

The predicate

$$\mathtt{SMCAP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k)$$

holds if, and only if, $\mathscr{D}$ has a sufficient memory capacity relative to P_k.

We now axiomatize the signifiable computability of $\mathtt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$.

SCT Axiom 7.

$$(\exists P_k)\left\{P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}} \wedge \mathtt{SMCAP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k) \wedge \mathtt{SGNCF}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\mathtt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}, P_k)\right\}.$$

SCT Axiom 7 concretizes CCT Axiom A3 by formalizing a facility (i.e., $\mathtt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$) to move a computation forward step-by-step when a step is identical to a valid program state.

SCT Axiom 7 also broadens CCT Axiom A8 by stating that there is an explicit finite bound (i.e., $\texttt{CCAP}(\mathscr{D})$) on the amount of memory available for signifiable computation whenever signifiable computation is carried out by a computing agent (e.g., an operating system or a virtual machine) on $\mathscr{D}$.

Since the functions $\texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (43), $\texttt{CINS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (44), $\texttt{TOTI}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (45), $\texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (46), $\texttt{NPIS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (47), $\texttt{ENDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (48), $\texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (49), and $\texttt{SGNCOMP}_{\mathscr{A},\mathscr{L},\mathscr{D}}$ (18) are defined in terms of $\texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}$, we have

Definition 22 (Sufficiently Powerful Finite Memory Device)**.** *An FMD $\mathscr{D}$ is sufficiently powerful relative to $\mathscr{L}$ on $\mathscr{A}$ if, for any f, such that*

$$\begin{aligned} f \quad \in \quad & \{\texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}, \texttt{CINS}_{\mathscr{A},\mathscr{L},\mathscr{D}}, \texttt{TOTI}_{\mathscr{A},\mathscr{L},\mathscr{D}}, \texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}, \\ & \texttt{NPIS}_{\mathscr{A},\mathscr{L},\mathscr{D}}, \texttt{ENDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}, \texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}, \texttt{SGNCOMP}_{\mathscr{A},\mathscr{L},\mathscr{D}}\}, \end{aligned}$$

$$(\exists P_k)\Big\{P_k \in \mathfrak{P}_{\hat{F}_{\mathscr{A},\mathscr{L}}} \wedge \texttt{SMCAP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k) \wedge \texttt{SGNCF}_{\mathscr{A},\mathscr{L},\mathscr{D}}(f, P_k)\Big\}.$$

9. Main Results

We now present the two theorems, which we refer to as the *Debugger Theorems*, that constitute the main results of this investigation. The theorems are so called, because they characterize some properties of the debugger (DBGR) function defined as

$$\texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, s) : \mathscr{A}^+ \times \mathscr{A}^+ \mapsto \mathscr{A}^+ \cup \{t_{cap}, t_{err}\};$$

$$\texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, s) = \begin{cases} & \texttt{STARTS}_{\mathscr{D},\mathscr{A},\mathscr{L}}(\sigma) = \texttt{"1"} \quad \wedge \\ \texttt{NXTS}^{n-1}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) & \text{if } (\forall \sigma_{i_j}, \sigma_{i_{j+1}})\{\texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) \neq \sigma_{i_{j+1}} \quad \vee \\ & \{\sigma_{i_{j+1}} \neq t_{err} \ \wedge \ \sigma_{i_{j+1}} \neq t_{cap} \ \}\}; \\ t_{err} & \text{if } (\exists \sigma_{i_j})\texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) = t_{err}; \\ t_{cap} & \text{if } (\exists \sigma_{i_j})\texttt{NXTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) = t_{cap}, \end{cases} \tag{50}$$

where $\sigma = \sigma_{i_1}, s \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow n \in \mathbb{Z}^+, n > 1, 1 \leq j < n$.

We define a debuggable FMD relative to $\mathscr{L}$ on $\mathscr{A}$ as

Definition 23 (Debuggable Finite Memory Device)**.** *An FMD $\mathscr{D}$ is debuggable relative to $\mathscr{L}$ on $\mathscr{A}$ if*

$$(\exists P_k)\Big\{P_k \in \mathfrak{P}_{\hat{F}_{\mathscr{A},\mathscr{L}}} \wedge \texttt{SMCAP}_{\mathscr{A},\mathscr{L},\mathscr{D}}(P_k) \wedge \texttt{SGNCF}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}, P_k)\Big\}.$$

We now prove

Lemma 7. *Let $\mathscr{D}$ be an FMD debuggable relative to $\mathscr{L}$ on $\mathscr{A}$. Let $s \in \mathscr{A}^+_{\mathscr{L}}$ such that, for $1 < n \in \mathbb{Z}^+$,*

$$s \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow n \wedge \texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}.$$

Then,

$$\texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, s) = t_{cap}$$

if, and only if, for $1 \leq j < n$,

$$\texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) = \texttt{"1"} \wedge \texttt{NXT}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) = t_{cap}.$$

Proof. We omit the subscripts $\mathscr{A}, \mathscr{L}, \mathscr{D}$ for brevity and let $\sigma = \sigma_{i_1}$. If $\texttt{STARTS}(\sigma_{i_1}) = \texttt{"1"}$ and, for $1 \leq j < n$, $\texttt{VLDS}(\sigma_{i_j}) = \texttt{"1"}$ and $\texttt{NXT}(\sigma_{i_j}) = t_{cap}$, then $\texttt{DBGR}(\sigma_{i_1}, s) = t_{cap}$, for $s \leftarrow (=) \rightarrow j+1 = n$. Conversely, assume that $\texttt{DBGR}(\sigma_{i_1}, s) = t_{cap}$. Since $\texttt{STARTS}(\sigma_{i_1}) = \texttt{"1"}$,

$$\{\texttt{NXT}(\sigma_{i_1}) = \sigma_{i_2} \wedge \texttt{VLDS}(\sigma_{i_2}) = \texttt{"1"}\} \vee \texttt{NXT}(\sigma_{i_1}) = t_{cap}.$$

If $\texttt{NXT}(\sigma_{i_1}) = t_{cap}$, then $j = 1$ and $n = 2$. Otherwise, we proceed from $j = 2$ and examine each consecutive pair $(\sigma_{i_j}, \sigma_{i_{j+1}})$ in the same fashion as the pair $(\sigma_{i_1}, \sigma_{i_2})$ to find $2 \leq j < n$ and $j+1 = n \leftarrow (=) \rightarrow s$ such that $\texttt{VLDS}(\sigma_{i_j}) = \texttt{"1"}$ and $\texttt{NXT}(\sigma_{i_j}) = t_{cap}$. □

Corollary 7.1. *Let* $\mathscr{D}$ *be an FMD debuggable relative to* $\mathscr{L}$ *on* $\mathscr{A}$. *Let* $s \in \mathscr{A}^+_{\mathscr{L}}$ *such that, for* $1 < n \in \mathbb{Z}^+$,

$$s \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow n \wedge \texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"}.$$

Then,

$$\texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, s) = t_{err}$$

if, and only if, for $1 \leq j < n$,

$$\texttt{VLDS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) = \texttt{"1"} \wedge \texttt{NXT}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma_{i_j}) = t_{err}.$$

We now prove

Theorem 1. *Let* $\mathscr{D}$ *be an FMD debuggable relative to* $\mathscr{L}$ *and* $\mathscr{A}$. *Let* $f : \mathbb{R}^m \mapsto \mathbb{R}$ *such that*

$$\texttt{SGNPCF}_{\mathscr{A},\mathscr{L},\mathscr{D}}(f, P_k) \wedge P_k \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}.$$

Then, if

$$\begin{array}{ll}
\sigma \in \mathscr{A}^+_{\mathscr{L}} & \wedge \\
(r_1, \ldots, r_m) \notin dom(f) & \wedge \\
\texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"} & \wedge \\
\texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = P_k & \wedge \\
\texttt{IVV}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, t_{i_j}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r_j & \wedge \\
t_{i_j} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow j & \wedge \\
1 \leq j \leq m \in \mathbb{Z}^+ & \wedge \\
\neg(\exists\, s)\ \texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, s) = t_{err}, &
\end{array}$$

then

$$(\exists n \in \mathbb{Z}^+)\Big\{1 < n \wedge n \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow s \wedge \texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D},}(\sigma, s) = t_{cap}\Big\}.$$

Proof. We omit the subscripts $\mathscr{A}, \mathscr{L}, \mathscr{D}$ for brevity. Let $\sigma = \sigma_{i_1}$ be such that $\texttt{STARTS}(\sigma_{i_1}) = \texttt{"1"}$, $\texttt{PROG}(\sigma_{i_1}) = P_k$, and $\texttt{IVV}(\sigma_{i_1}, t_{i_j}) \leftarrow (=) \rightarrow r_j$, $1 \leq j \leq m \in \mathbb{Z}^+$, $t_{i_j} \in \mathscr{A}^+$. Let $P_w \in \mathfrak{P}_{\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}}$ be such that $\texttt{SGNCF}(\texttt{DBGR}, P_w)$.

Since $(r_1, \ldots, r_m) \notin dom(f)$, we have

$$\neg(\exists n > 1)\ \texttt{SGNCOMP}(P_k, (\sigma_{i_1}, \sigma_{i_2}, \ldots, \sigma_{i_n})),$$

whence

$$\neg(\exists n > 1)\ \Big\{\texttt{NXTS}^{n-1}(\sigma_{i_1}) = \sigma' \wedge \texttt{ENDS}(\sigma') = \texttt{"1"} \wedge \texttt{OUTV}(\sigma') \leftarrow (=) \rightarrow f(r_1, \ldots, r_m)\Big\}.$$

We observe that if, for some σ, $\texttt{VLDS}(\sigma) = \texttt{"1"}$, $\texttt{TOTI}(\sigma) = t \in \mathscr{A}^{+}_{\mathscr{L}}$, $t \leftarrow (=) \rightarrow z \in \mathbb{Z}^{+}$, $\texttt{NXT}(\sigma) = \sigma'$, and $\texttt{VLDS}(\sigma') = \texttt{"1"}$, then, by Lemma 1,

$$\texttt{TOTI}(\sigma') = t' \in \mathscr{A}^{+}_{\mathscr{L}} \wedge t' \leftarrow (=) \rightarrow z' \in \mathbb{Z}^{+} \wedge z' = z + 1.$$

Let $\texttt{NXT}(\sigma_{i_1}) = \sigma_{i_2}$. If $|\sigma_{i_2}| > \texttt{CCAP}(\mathscr{D})$, then $n = 2$. Otherwise, we run P_w to compute $\texttt{DBGR}(\sigma_{i_1}, s_3)$, $s_3 \leftarrow (=) \rightarrow 3$, whereby P_w computes

$$\texttt{NXT}(\texttt{NXT}(\sigma_{i_1})) = \texttt{NXT}(\sigma_{i_2}) = \sigma_{i_3}.$$

If $|\sigma_{i_3}| > \texttt{CCAP}(\mathscr{D})$, then $n = 3$. Otherwise, we continue incrementing the natural number corresponding to the previous state counter numeral by 1 and running P_w on σ_{i_1} and

$$s_4 \leftarrow (=) \rightarrow 4, s_5 \leftarrow (=) \rightarrow 5, \ldots,$$

(i.e., the numerals corresponding to the natural numbers $4, 5, \ldots$). After a finite number of such iterations, we necessarily reach σ and σ' such that

$$\texttt{NXT}(\sigma) = \sigma' \wedge \texttt{VLDS}(\sigma) = \texttt{VLDS}(\sigma') = \texttt{"1"} \wedge |\sigma| < |\sigma'|,$$

because in the standard decimal notation the numeral encoding $\texttt{TOTI}(\sigma')$ contains more digit signs than the numeral encoding $\texttt{TOTI}(\sigma)$, whence

$$(\exists n > 3)\Big\{s \leftarrow (=) \rightarrow n \ \wedge \ \texttt{DBGR}(\sigma_{i_1}, s_n) = t_{cap}\Big\}.$$

□

We now prove

Theorem 2. *Let $\mathscr{D}$ be an FMD debuggable relative to $\mathscr{L}$ and $\mathscr{A}$. Then, there exists $f : \mathbb{R}^m \mapsto \mathbb{R}$ such that $\texttt{SGNCF}_{\mathscr{A},\mathscr{L}}(f, P_k)$, $P_k \in \mathfrak{P}_{\hat{\mathrm{F}}_{\mathscr{A},\mathscr{L}}}$, and, for some $(r_1, \ldots, r_m) \in dom(f)$, if*

$$\begin{array}{ll}
\sigma \in \mathscr{A}^{+}_{\mathscr{L}} & \wedge \\
\texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"} & \wedge \\
\texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = P_k & \wedge \\
\texttt{IVV}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, t_{i_j}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow r_j & \wedge \\
t_{i_j} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow j & \wedge \\
1 \leq j \leq m \in \mathbb{Z}^{+} & \wedge \\
\neg(\exists\, s)\ \texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, s) = t_{err}, &
\end{array}$$

Then,

$$(\exists m \in \mathbb{Z}^{+})(\forall n \in \mathbb{Z}^{+})\Big\{m \leq n \wedge n \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow s \wedge \texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D},}(\sigma, s) = t_{cap}\Big\}.$$

Proof. We omit the subscripts $\mathscr{A}$, $\mathscr{L}$, $\mathscr{D}$ for brevity. Let $f : \mathbb{N} \mapsto \mathbb{Z}^{+}$ such that $f(x) = x + 1$. Then, $\texttt{SGNCF}(f, P_k)$, $P_k \in \mathfrak{P}_{\hat{\mathrm{F}}_{\mathscr{A},\mathscr{L}}}$ (cf., e.g., P_j in Figure 1). Let $z \in \mathbb{Z}^{+}$ be the smallest positive integer such that $z \leftarrow (=) \rightarrow w$ and $|w| > \texttt{CCAP}(\mathscr{D})$. Let

$$\begin{array}{ll}
\sigma \in \mathscr{A}^{+}_{\mathscr{L}} & \wedge \\
\texttt{STARTS}(\sigma) = \texttt{"1"} & \wedge \\
\texttt{PROG}(\sigma) = P_k & \wedge \\
\texttt{IVV}(\sigma, t_{i_1}) \leftarrow (=) \rightarrow 0 & \wedge \\
t_{i_1} \leftarrow (=) \rightarrow 1. &
\end{array}$$

Since $\texttt{STARTS}(\sigma) = \texttt{"1"}$, then, by (50),

$$(\exists m)\Big\{1 \leq m \leq z \ \wedge \ m \leftarrow (=) \rightarrow v \ \wedge \ \texttt{DBGR}(\sigma, v) = t_{cap}\Big\}.$$

□

Corollary 2.1. *Let $\mathscr{D}$ be an FMD debuggable relative to $\mathscr{L}$ and $\mathscr{A}$. Let f in Theorem (2) be such that if $(r_1, \ldots, r_m), (r'_1, \ldots, r'_m) \in dom(f)$ and $r_1 < r'_1, \ldots, r_m < r'_m$, then $f(r_1, \ldots, r_m) < f(r'_1, \ldots, r'_m)$. Then, for some $(y_1, \ldots, y_n) \in dom(f)$, if*

$$\begin{array}{ll}
\sigma \in \mathscr{A}^{+}_{\mathscr{L}} & \wedge \\
\texttt{STARTS}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = \texttt{"1"} & \wedge \\
\texttt{PROG}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma) = P_k & \wedge \\
\texttt{IVV}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, t_{i_j}) \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow y_j & \wedge \\
t_{i_j} \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow j & \wedge \\
1 \leq j \leq m \in \mathbb{Z}^{+} & \wedge \\
\neg(\exists\, s)\ \texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D}}(\sigma, s) = t_{err}, &
\end{array}$$

then

$$(\exists m \in \mathbb{Z}^{+})(\forall n \in \mathbb{Z}^{+})\Big\{m \leq n \wedge n \leftarrow (=)_{\mathscr{A},\mathscr{L}} \rightarrow v \wedge \texttt{DBGR}_{\mathscr{A},\mathscr{L},\mathscr{D},}(\sigma, v) = t_{cap}\Big\}.$$

10. Discussion

The proposed axiomatization broadens the scope of CCT Axiom A0 in that it makes the properties *computable* and *partially computable* apply to functions on discretely finite real numbers (cf. Definitions 15 and 16 and Lemma 5). The ontology of functions on discretely finite real numbers implied by the proposed axiomatization distinguishes three classes of functions: signifiable, signifiably computable, and signifiably partially computable. Per Lemma 5, signifiably computable functions are signifiably partially computable, but not vice versa. Per Lemma 6, there are signifiable functions that are not signifiably partially computable.

The proposed axiomatization characterizes signifiable computation on discretely finite real numbers in terms of two formal systems: the Former $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ that generates texts in $\mathscr{L}$ in $\mathscr{A}$ and the Transformer $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ that transforms texts formed by $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ into other texts in $\mathscr{L}$ on $\mathscr{A}$. These systems are co-dependent on each other and cannot exist in isolation. Neither system is involved in the interpretation of the results insomuch as such interpretation is outside of their scope. Both systems can be signified as programs in minimally adequate formalisms and those programs can be executed by different computing agents, such as human programmers or automated program synthesizers and debuggers. Consequently, $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ and $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ are not identical to the computing agent L of CCT Axioms A4, A5, A9 and A10. All texts produced by these two formal systems are spatiotemporally finite. While $\hat{\hat{F}}_{\mathscr{A},\mathscr{L}}$ always takes a finite amount of physical time, regardless of how time is measured, to form a text, $\hat{\hat{T}}_{\mathscr{A},\mathscr{L}}$ is not guaranteed to take a finite amount of physical time to transform a text into another text. The formation and transformation of texts can be signed by both systems and verified by third parties.

Programs that use continuous, random or analogue methods are evidently inconsistent with CCT Axioms A4 and A5. However, they are consistent with the proposed axiomatization in the following sense. If such programs are signified in a minimally adequate formalism, then their formation and transformation can be deterministically verified at least on some inputs in the domains of functions that those programs designate. Furthermore, if those programs are executed on an FMD debuggable relative to the said formalism, then,

per the two Debugger Theorems, the programs can be effectively debugged on all signifiable inputs. It should be noted that this debugging is effective, because on any signifiable input, the program that signifies the debugger function in (50) on the said FMD returns either the output signifying a discretely finite real number, an error message, or a message that the computation cannot be carried forward, because the FMD's memory capacity is exceeded. This integration of error into computation makes the proposed axiomatization different from the CCT axiomatization.

SCT Axiom 5 of the proposed axiomatization concretizes CCT Axiom A9, because it states that the computational capacity of a computing agent is a characteristic of not just the computing agent but also of the programming formalism. SCT Axiom 7 of the proposed axiomatization also concretizes CCT Axiom A8 by stating that there is an explicit finite bound on the memory available for signifiable computation whenever the computation is carried out by a computing agent on an FMD. Furthermore, the Debugger Theorems (cf. Section 9) indicate that CCT Axioms A6, A7, A8, and A10 do not apply to actual computability, i.e., situations when the signification of computation is performed on FMDs.

The concept of Debuggable FMD (cf. Definition 23) also bears on CCT Axiom A9 and distinguishes the proposed axiomatization from the CCT axiomatization. Debuggable FMDs are FMDs on which the debugger function (cf. Equation (50)) can be signified by a program in a minimally adequate $\mathscr{L}$ on $\mathscr{A}$. Intuitively, debuggable FMDs are computational devices with finite amounts of memory available for computation that can be effectively debugged. Thus, the computational capacity of a computing agent is limited not only by a finite number of instruction types the agent can perform but also by a concrete finite amount of memory available for signifiable computation.

A fundamental difference between the proposed axiomatization and the CCT axiomatization is the explicit integration of cost into computation through the function `TOTI` (cf. Equations (31) and (45)). Cost is different from the standard Big-O analysis of computer science (cf., e.g., Ch. 2 in [11]). The Big-O analysis is based on two assumptions: (1) the number of steps can be known ahead of computation as a function of the size of the input, and (2) constant factors do not matter (e.g., $f(n) = kn \in O(n)$, for $k \in \mathbb{Z}^+$, even when $k = 10^{58}$, i.e., the estimated number of photons in the observable universe). A consequence of the second assumption, almost never explicitly stated, is that the energy is available on demand to carry forward the prescribed computation. The function `TOTI` is not based on either assumption. It simply states that the cost of computation increases by one abstract unit (e.g., one unit of energy required to obtain a result, one unit of hardware maintenance cost, one unit of economic unavailability of computed results for decision making, etc.) with every signified instruction performed by a computing agent. To put it differently, per Lemma 1, the cost of signifiable computation monotonically increases by one abstract unit with every new signifiable program state.

Our approach to typed instructions (cf., e.g., Equation (22) and Definition 8) is inspired by the approach originally proposed by Church in [12]. However, our end objective is different. We do not aim to integrate λ-calculus (or any other formalism) into Russell's hierarchical theory of logical types (cf., e.g., [13,14]). Rather, we aim to formally characterize what is computable in principle and what is computable with performable processes on computational devices with finite amounts of memory.

The usage of logically quantified variables in our formalism makes no ontological commitment that assertions containing the said variables imply that the ranges of the variables actually exist [15] or that designating texts serve as senses of names [16]. While our formal apparatus (cf. Section 2) relies on the axiom of choice of the Zermelo–Fraenkel set theory, the proposed axioms, definitions, lemmas, theorems, and implications thereof should not be viewed as arguments for or against replacing the Zermelo–Fraenkel set theory

with the axiom of choice with the modern type theory as the foundation of mathematics (cf., e.g., [17]). The alternative foundations of mathematics are beyond the scope of our article.

Finally, we make no claim that the proposed axiomatization has no alternatives. Rather, it is our hope that this investigation will motivate other computability theory and recursion theory researchers to take a closer look at the differences between general and actual computabilities and to seek alternative axiomatizations of computability to gain deeper insights into the theory and practice of computer science.

11. Conclusions

In our previous investigation [4], we formalized the signification and reference of real numbers and showed that discretely finite structures representable as tuples of discretely finite numbers are signifiable on finite memory devices. In this investigation, we offered an axiomatization of signifiable computation on discretely finite real numbers and proved two theorems to investigate the computability properties of signifiably and partially signifiably computable functions on finite memory devices. For the next part of our investigation of significable computability, which we intend to cover in our next article, we plan to further investigate connections between signifiable computability and the theory of primitive recursive functions (cf., e.g., [18,19]) and, if and when possible, to connect signifiable computability to formal approaches to symbolic AI [10].

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study.

Conflicts of Interest: The author declares no conflicts of interest.

Appendix A

Below are the 11 axioms of the classical computability theory (CCT) based on Chapters 1–3 in [2]. CCT Axiom A0 is based on footnote 2 on p. 2 in [2] that states that the concepts *function* and *partial function* are restricted to mappings on non-negative integers. The ordinal numbers in the CCT Axioms A1–A10 correspond to the ordinals in the 10-item list of statements or questions to which affirmative or negative answers are given in Section 1.1 in [2], pp. 2–5. The assumptions reflected in the CCT axioms below are apparent in Chapters 2–8 in [1], another classical text on computability and recursion theory.

CCT Axiom A0. *Functions map natural numbers to natural numbers.*

CCT Axiom A1. *An algorithm is given as a set of instructions of finite size.*

CCT Axiom A2. *There exists a computing agent that can react to the instructions and carry out the computations.*

CCT Axiom A3. *There are facilities for making, storing, and retrieving steps in a computation.*

CCT Axiom A4. *If P is the set of instructions in an algorithm and L is a computing agent, then, given an input to P, L reacts to P by performing the computation in a discrete, stepwise fashion without using continuous methods or analogue devices.*

CCT Axiom A5. *L reacts to P by performing a computation deterministically, without random methods or devices.*

CCT Axiom A6. *There is no fixed finite bound on the size of inputs.*

CCT Axiom A7. *There is no fixed finite bound on the size of a set of instructions P.*

CCT Axiom A8. *There is no fixed finite bound on the amount of memory storage available to L to carry out a computation in P.*

CCT Axiom A9. *There is a fixed finite bound on the capacity of the computing agent L.*

CCT Axiom A10. *There is no fixed finite bound on the length of the computation performed by L on P; it is only required that the computation terminate after some finite number of steps.*

Table A1. Instruction types in $\mathfrak{I}_{\mathscr{A},\mathscr{L}}$ spatiotemporally finitely formable and transformable in programming formalisms that satisfy SCT Axiom 5. The subscripts $\mathscr{A}$ and $\mathscr{L}$ in the right column are omitted for brevity.

Instruction Type	Semantics
$\texttt{SET}(v,t)$, $v \in \mathscr{A}^+$, $t \in \mathscr{A}^*$	set the value of the variable named v to t
$\texttt{ADD}(t_1,\ldots,t_k)$, $t_i \in \mathscr{A}^+$, $1 \le i \le k$, $k > 1$	$t_i \leftarrow (\approx) \rightarrow r_i \in \mathbb{R} \wedge$ $\exists(\sum_{i=1}^{k} r_i, \mathrm{t}) \wedge \circlearrowleft (\sum_{i=1}^{k} r_i, t, t') \wedge$ $t' \leftarrow (\approx) \rightarrow \sum_{i=1}^{k} r_i$
$\texttt{SUB}(t_1,\ldots,t_k)$, $t_i \in \mathscr{A}^+$, $1 \le i \le k$, $k > 1$	$t_i \leftarrow (\approx) \rightarrow r_i \in \mathbb{R} \wedge$ $\exists(r_1 - r_2 - \ldots - r_k, t) \wedge$ $\circlearrowleft (r_1 - r_2 \ldots - r_k, t, t') \wedge$ $t' \leftarrow (\approx) \rightarrow r_1 - \ldots - r_k$
$\texttt{MUL}(t_1,\ldots,t_k)$, $t_i \in \mathscr{A}^+$, $1 \le i \le k$, $k > 1$	$t_i \leftarrow (\approx) \rightarrow r_i \in \mathbb{R} \wedge$ $\exists(\prod_{i=1}^{k} r_i, t) \wedge$ $\circlearrowleft (\prod_{i=1}^{k} r_i, t, t') \wedge$ $t' \leftarrow (\approx) \rightarrow \prod_{i=1}^{k} r_i$
$\texttt{DIV}(t_1,t_2)$, $t_i \in \mathscr{A}^+$, $1 \le i \le 2$	$t_1 \leftarrow (\approx) \rightarrow r_1 \in \mathbb{R} \wedge$ $t_2 \leftarrow (\approx) \rightarrow r_2 \in \mathbb{R} \wedge r_2 \neq 0 \wedge$ $\exists\left(\frac{r_1}{r_2}, t\right) \wedge \circlearrowleft \left(\frac{r_1}{r_2}, t, t'\right) \wedge$ $t' \leftarrow (\approx) \rightarrow \frac{r_1}{r_2}$
$\texttt{CONC}(t_1,\ldots,t_k)$, $t_i \in \mathscr{A}^+$, $1 \le i \le k$, $k > 1$	$\exists\left(\bigoplus\Big\vert_{i=1}^{k} t_i, t\right) \wedge$ $\circlearrowleft \left(\bigoplus\Big\vert_{i=1}^{k} t_i, t, t'\right) \wedge$ $t' = \bigoplus_{\mathscr{A}}\Big\vert_{i=1}^{k} t_i$
$\texttt{SUBST}(t,v_1,t_1,\ldots,v_k,t_k)$, $t, t_i, v_i \in \mathscr{A}^+$, $1 \le i \le k$, $k > 1$	return the text obtained by substituting in t t_1 for v_1, …, t_k for v_k.

References

1. Davis, M.; Sigal, R.; Weyuker, E. *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd ed.; Harcourt, Brace & Company: Boston, MA, USA, 1994.
2. Rogers, H., Jr. *Theory of Recursive Functions and Effective Computability*; The MIT Press: Cambridge, MA, USA, 1988.
3. Kulyukin, V. On Correspondences between feedforward artificial neural networks on finite memory automata and classes of primitive recursive functions. *Mathematics* **2023**, *11*, 2620. [CrossRef]
4. Kulyukin, V.A. On Signifiable Computability: Part I: Signification of Real Numbers, Sequences, and Types. *Mathematics* **2024**, *12*, 2881. [CrossRef]
5. Hopcroft, J.E.; Ullman, J.D. *Introduction to Automata Theory, Languages, and Computation*; Narosa Publishing Hourse: New Delhi, India, 2002.

6. Kuroda, S.Y. Classes of languages and linear-bounded automata. *Inf. Control* **1964**, *7*, 207–223. [CrossRef]
7. Quine, W.V. *Methods of Logic*; Harvard University Press: Cambridge, MA, USA, 1982.
8. Kleene, S.C. *Introduction to Metamathematics*; D. Van Nostrand: New York, NY, USA, 1952.
9. Steele, G.L. *Common Lisp: The Language*, 2nd ed.; Digital Press: Bedform, MA, USA, 1990.
10. Genesereth, M.R.; Nilsson, N.J. *Logical Foundations of Artificial Intelligence*; Morgan Kaufmann: Los Altos, CA, USA, 1987.
11. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L. *Introduction to Algorithms*; MIT Press: Cambridge, MA, USA, 1993.
12. Church, A. A formulation of the simple theory of types. *J. Symb. Log.* **1940**, 5, 56–68. [CrossRef]
13. Russell, B. Mathematical logic as based on the theory of types. *Am. J. Math.* **1908**, 30, 222–262. [CrossRef]
14. Kamareddine, F.; Laan, T.; Nederpelt, R. Types in logic and mathematics before 1940. *Bull. Symb. Log.* **2002**, *8*, 185–245. [CrossRef]
15. Church, A. Ontological commitment. *J. Philos.* **1958**, 55, 1008–1014. [CrossRef]
16. Church, A. A revised formulation of the logic of sense and denotation. Alternative (1). *Nous* **1993**, 27, 141–157.
17. Altenkirch, T. Should type theory replace set theory as the foundation of mathematics? *Glob. Philos.* **2023**, *33*, 21. [CrossRef]
18. Petersen, U. Induction and primitive recursion in a resource conscious logic—With a new suggestion of how to assign a measure of complexity to primitive recursive functions. *Dilemmata Jahrb. Asfpg* **2008**, 3, 49–106.
19. Paolini, L.; Piccolo, M.; Roversi, L. A class of recursive permutations which is primitive recursive complete. *Theor. Comput. Sci.* **2020**, *813*, 218–233. [CrossRef]