The Representational Foundations of Computation

Michael Rescorla

Abstract: Turing computation over a non-linguistic domain presupposes a notation for the domain. Accordingly, computability theory studies notations for various non-linguistic domains. It illuminates how different ways of representing a domain support different finite mechanical procedures over that domain. Formal definitions and theorems yield a principled classification of notations based upon their computational properties. To understand computability theory, we must recognize that representation is a key target of mathematical inquiry. We must also recognize that computability theory is an *intensional* enterprise: it studies entities as represented in certain ways, rather than entities detached from any means of representing them.

1. A COMPUTATIONAL PERSPECTIVE ON REPRESENTATION

Intuitively speaking, a function is *computable* when there exists a finite mechanical procedure that calculates the function's output for each input. In the 1930s, logicians proposed rigorous mathematical formalisms for studying computable functions. The most famous formalism is the *Turing machine*: an abstract mathematical model of an idealized computing device with unlimited time and storage capacity. The Turing machine and other computational formalisms gave birth to *computability theory*: the mathematical study of computability.

A Turing machine operates over strings of symbols drawn from a finite alphabet. These strings comprise a formal language. In some cases, we want to study computation over the formal language itself. For example, Hilbert's *Entscheidungsproblem* requests a uniform mechanical procedure that determines whether a given formula of first-order logic is valid. Items drawn from a formal language are *linguistic types*, whose *tokens* we can inscribe, concatenate, and manipulate [Parsons, 2008, pp. 34-36, pp. 160-164]. Most of modern mathematics concerns *non-linguistic entities*, such as natural numbers or real numbers. These entities do not have tokens that we can inscribe, concatenate, or manipulate [Parsons, 2008, pp. 37-39]. Thus, they do not comprise a formal language over which a Turing machine can directly operate. Nevertheless, we would like to extend the notion of Turing computability to subsume natural numbers, real numbers, and other non-linguistic domains.

Our interest in non-linguistic domains of computation ensures a central theoretical role for *representation*, as the following argument makes explicit:

> A Turing machine manipulates linguistic items, but we sometimes want to study computation over non-linguistic domain *X*. So there is a gap between the domain of items manipulated by the Turing machine and our desired domain of computation *X*. To bridge the gap, we must interpret linguistic items manipulated by the Turing machine as denoting items drawn from *X*. A Turing machine computes over *X* only if linguistic items manipulated by the Turing machine represent elements of *X*. Thus, any complete theory of computation must cite representational relations between linguistic items and non-linguistic items.

This argument, which I will call *the Gap Argument*, is quite robust. It does not assume any special conception of the 'symbols' comprising the Turing machine 'alphabet.' It does not

presuppose contentious details about the distinction between 'linguistic' and 'non-linguistic' domains. Computability theorists almost universally endorse the Gap Argument [Boolos and Jeffrey, 1980, p. 43; Davis, 1958, p. 9; Rogers, 1987, pp. 1-2, pp. 27-29; Weihrauch, 2000, p. 3, p. 13, p. 51], with varying degrees of explicitness.

My goal is to explore the central role that representation occupies within computability theory. Taking the Gap Argument as my starting point, I will highlight how representational notions shape the theory's most basic elements (Section 2). As I will explain, computability theorists investigate representation *from a computational perspective*. They assign prime importance to *notations* for representing entities. Sufficiently different notations support different notions of computability over a domain (Sections 2.2 and 2.3). Computability theory investigates the interplay between notation and computation, illuminating how different ways of representing a domain support different finite mechanical procedures over that domain. Formal definitions and theorems yield a principled classification of notations based upon their computational properties. To understand the aims and methods of computability theory, we must recognize that representation is a key target of mathematical inquiry (Section 3). We must also recognize that computability theory is an *intensional* enterprise (Sections 4 and 5): it studies entities as represented in certain ways, rather than entities detached from any means of representing them.

Notational conventions: I use the notation $f\colon X \to Y$ to signify that f is a *partial* function from X to Y, so that $f(x)$ may be defined only for some $x \in X$. I say that $x \in X$ belongs to f's domain, or $x \in dom(f)$, iff $f(x)$ is defined. I say that $f\colon X \to Y$ is *total* iff $\mathrm{dom}(f) = X$. 'Function' means 'partial function,' unless I explicitly specify that the function is total. '$f(x) = g(x)$' is true iff both sides are defined and equal to one another *or* neither side is defined.

2. COMPUTATION OVER NON-LINGUISTIC DOMAINS

A Turing machine contains a scanner that moves along a machine tape divided into cells. Depending on the scanner's state and the contents of its current cell location, the scanner either erases a symbol, inscribes a symbol, or moves to the left or right. Symbols belong to a fixed finite alphabet Σ that contains at least two elements. Let Σ^* be the set of finite strings over Σ. A Turing machine *halts* when it reaches a point where its instructions dictate no further action.

For $x, y \in \Sigma^*$, say that *Turing machine T yields output y on input x* iff

> If *T* begins computation with *x* inscribed on an otherwise blank machine tape and with the scanner located at the beginning of *x*, then *T* eventually halts with *y* inscribed on an otherwise blank machine tape.

Say that *Turing machine T computes* $\gamma: \Sigma^* \rightarrow \Sigma^*$ iff

> *T* yields output $\gamma(x)$ on any input $x \in dom(\gamma)$, and *T* yields no output on any input $x \notin dom(\gamma)$.

$\gamma: \Sigma^* \rightarrow \Sigma^*$ is *partial recursive* iff some Turing machine computes it. One can easily extend these definitions to functions with multiple arguments.

Computability theory is important partly because it illuminates computation over non-linguistic domains, such as the natural numbers and the real numbers. However, applying the Turing machine formalism to non-linguistic domains raises various complications. This section surveys several notable complications. Subsequent sections draw philosophical morals.

2.1. Relativity

Classical recursion theory emphasizes the intuitive concept *computable function from natural numbers to natural numbers*. If we want to elucidate this concept through the Turing machine

formalism, then the Gap Argument shows that we must introduce a semantic interpretation over strings. Let d: $\Sigma^* \rightarrow \mathbb{N}$ be a surjective function, where $\mathbb{N}$ is the natural numbers. If $d(x)$ is defined, then x is a *d-name* for $d(x)$.[1] Say that *Turing machine T computes f*: $\mathbb{N} \rightarrow \mathbb{N}$ *relative to d* iff there is a function γ: $\Sigma^* \rightarrow \Sigma^*$ such that T computes γ and

$$f(d(x) = d(\gamma(x)),$$

for all $x \in \Sigma^*$. Informally, T converts a d-name for any $x \in dom(f)$ into a d-name for $f(x)$. Pick any standard notation for the natural numbers. For example, let d_{binary}: $\Sigma^* \rightarrow \mathbb{N}$ be binary notation. Then f: $\mathbb{N} \rightarrow \mathbb{N}$ is *partial recursive* iff it is Turing-computable relative to d_{binary}. *Church's thesis* (sometimes called *the Church-Turing thesis*) states that a number-theoretic function is intuitively computable iff it is partial recursive.

One might try to avoid the Gap Argument by characterizing numerical computability through notions defined directly over the natural numbers, rather than linguistic intermediaries. To illustrate, say that a numerical function is *Kleene-computable* iff we can obtain it from the primitive recursive functions through function composition and application of the minimization operator μ. Kleene's Normal Form Theorem entails that a function is partial recursive iff it is Kleene-computable. We define Kleene-computability without mentioning symbol manipulation, so the Gap Argument does not apply.

However, few commentators would recommend Kleene-computability as a satisfactory foundation for computability theory. As Gödel [1934/1986, pp. 369-370] maintained, Turing analyzes not just *computability* but *computation* itself. Turing thereby connects formal

[1] Philosophers and computability theorists typically assume that Σ's elements are individuated *non-semantically*, through factors such as geometric shape. Under this assumption, the string language taken on its own is equally compatible with any arbitrary semantic interpretation d. In [Rescorla, forthcoming], I question whether the Turing formalism mandates an alphabet Σ whose elements are individuated non-semantically. For present purposes, I set these issues aside. I assume an alphabet Σ whose elements are individuated non-semantically.

mathematical theorizing with our pre-theoretic conception of computation. In contrast, a characterization through Kleene-computability does not even purport to analyze *computation*. That is why Turing's analysis supports Church's thesis while an analysis citing Kleene-computability does not. If we follow Turing by analyzing computation as symbol manipulation, then the Gap Argument gains a foothold.[2]

Classical recursion theory extends Turing computability beyond the natural numbers to other countable non-linguistic domains: integers; rational numbers; finite sets of natural numbers; and so on. In each case, the Gap Argument shows that we must introduce a semantic mapping from the string language to the desired domain.

A particularly important example: *computation over the partial recursive functions themselves*. Through Gödelization, we can code Turing machine programs using strings drawn from Σ^*. We then map each string to the partial recursive function computed by the corresponding program. This procedure yields a canonical total function $\varphi: \Sigma^* \rightarrow \mathbb{P}$, where $\mathbb{P}$ is the set of partial recursive string-theoretic functions $\gamma: \Sigma^* \rightarrow \Sigma^*$.[3] Abbreviate $\varphi(x)$ as φ_x. Thus, φ_x is the partial recursive function named by index x. Having introduced canonical names for partial recursive functions, we can study mechanical procedures that take those functions as inputs or outputs. For example, Turing proved the existence of a 'Universal Turing Machine' (UTM) that mimics any partial recursive function when provided with a name for that function:

> ***Enumeration (or UTM) theorem***: There exists a partial recursive function $U: \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ such that, for all $x, y \in \Sigma^*$, $U(x, y) = \varphi_x(y)$.

[2] The mathematical literature offers various alternative analyses of numerical computability, such as the lambda calculus [Church, 1936] or the equation calculus [Kleene, 1936]. Famously, all these alternative analyses are extensionally equivalent. Most analyses, although not all, assign a crucial role to symbol manipulation.

[3] Computability theorists typically index partial recursive functions by *numbers*, not *strings*. For mathematical purposes, the difference does not matter. From a philosophical perspective, it is more fitting to use strings as indices [Rogers, 1987, p. 28]. Ultimately, we must represent partial recursive functions by using concrete names. Using numbers as indices only postpones an inevitable appeal to concrete names and their representational properties.

Kleene proved that one can computably incorporate arguments of partial recursive functions into indices for partial recursive functions:

> ***Parameterization (or s-m-n) theorem***: There exists a partial recursive function s: $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$ such that, for all $x, y, z \in \Sigma^*$, $\varphi_x(y, z) = \varphi_{s(x, y)}(z)$.

Indexing of partial recursive functions underlies most of recursion theory's basic results: Rice's theorem; the Kleene recursion theorem; and so on.[4]

In general, suppose we want to model Turing computations that take inputs drawn from countable domain X and yield outputs drawn from countable domain Y. Suppose that neither X nor Y contains strings drawn from a finite alphabet of symbols. Then we must introduce *notations* for X and Y. For present purposes, we construe notations as surjective functions d: $\Sigma^* \rightarrow X$ and e: $\Sigma^* \rightarrow Y$. We say that f: $X \rightarrow Y$ is *computable relative to d and e* iff there is a partial recursive γ: $\Sigma^* \rightarrow \Sigma^*$ such that $f(d(x)) = e(\gamma(x))$, for all $x \in \Sigma^*$. Informally, γ converts a d-name for each $x \in dom(f)$ into an e-name for $f(x)$.[5]

Our definitions reflect a crucial *relativity* inherent to Turing-computation over a non-linguistic domain:

> ***Relativity***: Turing computation over non-linguistic domains is relative to a notation. The same Turing machine T computes different non-linguistic functions, depending upon the semantic interpretation of strings manipulated by the Turing machine.

To illustrate, consider a machine T that doubles the number of stroke marks on the tape, and let $\underline{m}$ be a string of m strokes. If $\underline{m}$ denotes the number m, then T computes the function $f(n) = 2n$. If

---

[4] Similarly, one can introduce a canonical indexing for numerical partial recursive functions f: $\mathbb{N} \rightarrow \mathbb{N}$. The results mentioned in this paragraph are typically proved for that case, rather than for string-theoretic functions. But the proofs are essentially the same in either case.

[5] This definition captures a notion sometimes called *strong computability*. In contrast, *weak computability* demands only that $f(d(x)) = e(\gamma(x))$ for all $x \in dom(f \circ d)$. See [Weihrauch, 2000, pp. 53-54, p. 59]. Outside recursion theory, computability theorists assign more weight to weak computability than strong computability. For our purposes, the difference is not important.

$\underline{m}$ denotes the number m-1, then T computes $g(n) = 2n + 1$. Lacking a determinate interpretation of strings, we cannot treat a Turing machine as computing a determinate function over the natural numbers. Similarly, consider any UTM. Relative to our canonical indexing φ, the UTM computes the function $\Omega: \mathbb{P} \times \Sigma^* \rightarrow \Sigma^*$ that carries $\varphi_x \in \mathbb{P}$ and $y \in \Sigma^*$ to $\varphi_x(y)$, assuming $\varphi_x(y)$ is defined. Yet the UTM does not compute Ω relative to a suitably different notation for $\mathbb{P}$. As these examples illustrate, a change in notation typically alters the non-linguistic function computed by a Turing machine. Lacking a determinate semantic interpretation of strings, a Turing machine does not compute a determinate function over a non-linguistic domain.

2.2. Admissible versus deviant notations

There exist 'deviant' notations relative to which intuitively non-computable functions become Turing-computable [Copeland and Proudfoot, 2010; Montague, 1960; Shapiro, 1982]. In [Rescorla, 2007], I considered the following simple example. Let $A \subset \mathbb{N}$ have a non-recursive characteristic function. Let $A = \{x_0, x_1, x_2, \ldots\}$ and $\mathbb{N} \setminus A = \{y_0, y_1, y_2, \ldots\}$. Define $d_A: \Sigma^* \rightarrow \mathbb{N}$ by

$$d_A(\underline{n}) = \begin{cases} x_{n/2} & \text{if } n \text{ is even} \\ y_{(n-1)/2} & \text{if } n \text{ is odd} \end{cases}$$

A's characteristic function is Turing computable relative to d_A. Yet A's characteristic function is not recursive and hence (by Church's thesis) not intuitively computable.

Recursion theory textbooks invariably ignore deviant notations for $\mathbb{N}$. They select a single notation (such as d_{binary}) and develop recursion theory relative to that privileged notation. For many purposes, this procedure is entirely legitimate. However, if we desire a general theory of computation over natural numbers, then the procedure seems arbitrary and unsatisfying. Numerical computation can proceed relative to diverse reasonable notations. There is no clear

basis for privileging one specific notation as fundamental [Rescorla, 2007]. We want a general distinction between 'acceptable' notations, such as d_{binary}, and 'deviant' notations, such as d_A. A good account should demarcate those notations that are 'admissible' for numerical computation from those that are not. One does not explicate admissibility by giving examples of admissible notations, any more than one explicates wisdom by giving examples of wise people. We would like an analysis of *what it is* to be admissible. What do all admissible notations have in common that distinguishes them from deviant notations such as d_A?

Deviant semantic interpretations arise whenever we consider Turing computation over a non-linguistic countably infinite domain X:

> ***Deviancy***: There exist deviant semantic interpretations from strings to X. Relative to a deviant notation, Turing machines can 'compute' functions over X that are intuitively uncomputable.

To illustrate, consider the *Totality Problem*: is there a uniform mechanical procedure that decides whether a partial recursive function is total? One can prove that there is no Turing machine T such that:

> Given input x, T outputs $\underline{1}$ if φ_x is total and $\underline{0}$ if φ_x is not total,

where φ is our canonical indexing of $\mathbb{P}$. Based on this theorem, we conclude that the Totality Problem is undecidable. However, one can easily construct a notation η: $\Sigma^* \rightarrow \mathbb{P}$ such that, for some Turing machine T:

> Given input x, T outputs $\underline{1}$ if η_x is total and $\underline{0}$ if η_x is not total.

Should we conclude that the Totality Problem is decidable after all? Of course not, because the requisite notation η is highly deviant.

More generally, consider any countably infinite domain X. There exist uncountably many notations d: $\Sigma^* \rightarrow X$. For any notations d_1: $\Sigma^* \rightarrow X$ and d_2: $\Sigma^* \rightarrow X$, say that d_1 *is reducible to* d_2, or $d_1 \leq d_2$, iff there exists a partial recursive γ: $\Sigma^* \rightarrow \Sigma^*$ such that

$$d_1(x) = d_2(\gamma(x)),$$

for all $x \in dom(d_1)$. Informally, γ is a computable translation from d_1 to d_2: it carries each d_1-name to a co-referring d_2-name. Say that d_1 and d_2 are *equivalent* (or $d_1 \equiv d_2$) iff $d_1 \leq d_2$ and $d_2 \leq d_1$. We can partition the notations d: $\Sigma^* \rightarrow X$ into equivalence classes. There are countably many recursive γ: $\Sigma^* \rightarrow \Sigma^*$, so each equivalence class is countable. It follows that there are uncountably many equivalence classes. One can easily show that distinct equivalence classes induce distinct notions of *computable function from X to X*:

> If $\neg(d_1 \leq d_2)$, then there are functions computable relative to d_1 and d_1 that are not computable relative to d_1 and d_2.

For example, the identity mapping is computable relative d_1 and d_1 (when we use d_1-names for both inputs and outputs) but not relative to d_1 and d_2 (when we use d_1-names for inputs and d_2-names for outputs). Thus, one can define uncountably many concepts of computability over X. Of course, we do not have uncountably many *intuitive* concepts of computability over a domain. An intuitive notion of computability arises only when the relevant notations are 'admissible' rather than 'deviant.'

A theory of Turing computation over any non-linguistic countably infinite domain must address the distinction between admissible and deviant notations for the domain. Unfortunately, analyzing this distinction in satisfying terms is not so easy.

To illustrate, suppose one says that a notation is admissible iff there is an effective procedure that maps strings into values denoted by strings. On this analysis, admissibility

requires a computation that takes strings drawn from Σ^* and yields appropriate denotations drawn from X. Assuming that X is a non-linguistic domain, we need a notation to represent its elements. Which notations for X may we use when computing denotations of strings? Not just any notation will do. For example, many deviant notations will be incorrectly ruled admissible if we allow computations relative to d_A. But why should we disallow computations relative to d_A? Well, because d_A itself is deviant. Unfortunately, that answer presupposes the distinction between admissible and deviant notations. In other words, suppose we say:

> d: $\Sigma^* \rightarrow X$ is admissible iff there is an admissible notation e: $\Sigma^* \rightarrow X$ and a Turing machine T such that T carries each d-name to a co-referring e-name.

Then we assume that we have already demarcated the admissible notations for X, which renders our procedure circular.

To avoid circularity, we can demand that some effective procedure carry strings to denoted values *as represented by a fixed privileged notation*. More precisely:

(1) d: $\Sigma^* \rightarrow X$ is admissible iff there is a Turing machine T that carries each d-name to a co-referring d_X-name,

where d_X: $\Sigma^* \rightarrow X$ is some fixed privileged notation for X. We may rephrase (1) as follows:

(2) d: $\Sigma^* \rightarrow X$ is admissible iff $d \leq d_X$.

For example, we might explicate admissibility over the natural numbers by taking d_{binary} as our privileged notation:

(3) d: $\Sigma^* \rightarrow \mathbb{N}$ is admissible iff $d \leq d_{binary}$.

According to (3), a numerical notation is admissible just in case we can computably translate it into binary notation. (3) seems implicit in the practice of recursion theory, which dismisses without consideration all numerical notations not reducible to d_{binary}. Similarly, we might

explicate admissibility over the partial recursive functions by taking the canonical indexing φ as our privileged notation:

(4) $\pi: \Sigma^* \rightarrow \mathbb{P}$ is admissible iff $\pi \leq \varphi$.

According to (4), π is admissible just in case we can computably translate a π-name for a partial recursive function into a Turing machine program that computes the function.

(2) requires that we can computably translate an admissible notation into some fixed privileged notation. A more demanding definition requires that we can also computably translate back from the privileged notation:

(5) $d: \Sigma^* \rightarrow X$ is admissible iff $d \equiv d_X$.

Applying this more demanding definition to the natural numbers yields:

(6) $d: \Sigma^* \rightarrow \mathbb{N}$ is admissible iff $d \equiv d_{binary}$.

(3) and (6) are extensionally equivalent, because $d \leq d_{binary}$ iff $d \equiv d_{binary}$. Applying the more demanding definition to the partial recursive functions yields:

(7) $\pi: \Sigma^* \rightarrow \mathbb{P}$ is admissible iff $\pi \equiv \varphi$,

which requires that one can computably recover a π-name for a partial recursive function from a Turing machine program that computes the function. (4) and (7) are not extensionally equivalent, because there are notations π such that $\pi \leq \varphi$ and $\neg(\varphi \leq \pi)$. One can show that the Parameterization theorem fails for any such π, i.e. the theorem fails if we substitute π for φ [Rogers, 1987, pp. 41-42]. Since the Parameterization theorem is needed for developing a fruitful recursion theory, π does not yield a useful notion of computation over $\mathbb{P}$. Accordingly, Rogers

[1987, p. 41] advocates (7) rather (4) as the proper definition of admissibility for partial recursive functions (although he uses the phrase 'acceptable' rather than 'admissible').[6]

Setting aside the differences between (2) and (5), each definition faces the same fundamental worry: it ties admissibility to a privileged notation arbitrarily chosen from among many equally worthy competitors [Rescorla, 2012]. Consider the special case of (3). I grant that (3) yields an extensionally adequate, non-circular characterization of admissible numerical notations. However, one can represent natural numbers using many legitimate notations besides d_{binary}. There is no principled basis for privileging d_{binary} over these alternative notations when elucidating admissibility. A good elucidation should instead isolate desirable features shared by d_{binary} and all the legitimate alternatives. Since (3) does not do so, it does not reveal *what it is* for numerical notations to be admissible. These worries become even more pronounced when we consider computation over an arbitrary domain. Definitions (2) and (5) offer no clue what desirable features are shared by the privileged notation d_X for domain X and the privileged notation d_Y for domain Y. The definitions prioritize certain privileged notations without explaining *why* those notations deserve privileged status.

An alternative strategy for elucidating admissibility is to isolate some 'intrinsic' property shared by all admissible notations. For example, we can easily prove the following result [Weihrauch, 2000, p. 73]: for any notation d: $\Sigma^* \rightarrow \mathbb{N}$,

[6] Some readers may insist that $\pi \leq \varphi$ suffices for admissibility. If this verdict were correct, then it would be an example of Multiplicity (discussed in Section 2.3), so it would actually strengthen my overall argument. However, I find the verdict implausible. Rogers [1987, p. 42] adduces a notation δ such that $\delta \leq \varphi$ and $\neg(\varphi \leq \delta)$ with the following property: there is a Turing machine that responds to input x with output $\underline{1}$ if $\delta_x(\underline{0})$ is defined and output $\underline{0}$ if $\delta_x(\underline{0})$ is undefined. In other words, we can 'decide' relative to δ whether a given partial recursive function is defined on input $\underline{0}$. We cannot decide this question relative to our canonical indexing φ, and normally that is taken as powerful evidence that the question is intuitively undecidable. Computation relative to δ does not seem to me to yield a mechanical procedure for deciding whether partial recursive functions are defined on input $\underline{0}$. At the very least, classifying δ as admissible would be a major revision to current mathematical practice.

$d \equiv d_{binary}$ iff the successor function is computable relative to d and $d \leq e$ for any notation e: $\Sigma^* \to \mathbb{N}$ relative to which the successor function is computable.

In a similar vein, Rogers [1987, pp. 41-42] proves that, for any notation π: $\Sigma^* \to \mathbb{P}$,

$\pi \equiv \varphi$ iff the Enumeration theorem and Parameterization theorems are true for π (i.e. the theorems remain true when one substitutes π for φ).

These 'intrinsic' characterizations of admissibility for $\mathbb{N}$ and $\mathbb{P}$ seem helpful. At least they avoid any appeal to an arbitrarily chosen privileged notation. However, neither characterization yields anything resembling an analysis of admissibility for the relevant domain. Neither characterization reveals why certain notations are suitable for computing over the relevant domain while others are not. Moreover, neither characterization even tries to characterize admissible notations over an arbitrary domain.

Can we provide a satisfying analysis of admissibility for arbitrary domains? Or, failing that, a satisfying analysis of admissibility for certain salient domains? For present purposes, we may leave these questions unanswered. What matters is simply that the distinction between admissible and deviant notations plays an important role within computability theory. The distinction arises whenever we model computation over a countably infinite non-linguistic domain. In certain cases (such as $\mathbb{P}$), the distinction is an explicit target of mathematical study.

2.3. Multiplicity

Most domains studied within classical recursion theory support only a single fruitful notion of computability. But some countably infinite domains X satisfy

Multiplicity: There exist non-equivalent notations that are both admissible for computation over X. These distinct notations yield distinct yet equally legitimate notions of Turing computation over X.

A good example is computation over finite subsets of $\mathbb{N}$ [Rogers, 1987, pp. 69-71]. There are at least three admissible ways of naming a finite $A \subset \mathbb{N}$:

d_{list}: We can name A by an index that codes A's elements.

d_{char}: Since A is finite, its characteristic function is recursive. We can name A by an index that codes A's characteristic function.

$d_{r.e.}$: Since A is recursive, it is *recursively enumerable* (i.e. it is the domain of a partial recursive function). We can name A by an index that codes some partial recursive function f with $dom(f) = A$.

All three notations figure prominently in recursion theory. One can show that $d_{list} \leq d_{char}$ and $d_{char} \leq d_{r.e.}$ but that $\neg(d_{char} \leq d_{list})$, $\neg(d_{r.e.} \leq d_{char})$, and $\neg(d_{r.e.} \leq d_{list})$. The three notations yield distinct notions of computability over finite subsets of $\mathbb{N}$. It seems pointless to debate which notion is the 'true' one. One must simply evaluate which notion is more fruitful in a given theoretical setting.

Multiplicity is relatively rare within classical recursion theory. It pervades *computable analysis*, which studies computation over the real numbers $\mathbb{R}$ and other uncountable structures.

Computable analysis originates with Turing's 1936 paper. Indeed, Turing's explicit concern in that paper is computation involving real numbers, rather than computation involving natural numbers. A basic challenge facing computable analysis is the cardinality disparity between our representational system and the represented domain: there are only countably many finite strings drawn from a finite alphabet, but the target domain is uncountable. The solution

employed in computable analysis, as in everyday life, is to employ 'infinitary names' for real numbers. A few notable examples:

> ***Base-n notation***: Decimal notation, $d_{base\text{-}10}$, is familiar to anyone with a grade school education. More generally, one can introduce a notation $d_{base\text{-}n}$ that uses natural number n as its base. Turing [1936] uses $d_{base\text{-}2}$.[7]
>
> ***Rational open interval notation*** ($d_{interval}$): Assume some canonical notation for rational numbers. Then we can name a real number x by listing names of all rational numbers a and b such that $a<x<b$. In a 1937 correction to the 1936 paper, Turing replaced $d_{base\text{-}2}$ with $d_{interval}$, crediting the latter notation to Brouwer.
>
> ***Rational lower or upper bounds*** ($d_{<}$ and $d_{>}$): We can name a real number x by listing names of all rational numbers $a<x$. Alternatively, we can name x by listing names of all rational numbers $a>x$.

There are many other viable notations, including notations inspired by the familiar Cauchy sequence and Dedekind cut constructions. Formally, we model all these notations as surjective functions d: $\Sigma^{\omega} \rightarrow \mathbb{R}$, where Σ^{ω} is the set of infinite strings over Σ.[8]

Turing computation normally operates over finite strings. Thus, one must alter the Turing formalism if one wants to model computation over infinitary names. A particularly appealing framework is Weihrauch's [2000] Type-2 Theory of Effectivity, which builds upon earlier work by Grzegorczyk [1955], Hauck [1973], Lacombe [1955], and others. Weihrauch introduces

---

[7] $d_{base\text{-}2}$ is non-injective, since every dyadic rational number has two names: one name terminates with infinitely many '0's, and the other name terminates with infinitely many '1's. Turing seems to have envisioned a slightly modified notation that includes only names of the first kind, so that each dyadic rational number has a unique name. See Gherardi [2011] for detailed analysis of Turing's approach to computable analysis.

[8] Weihrauch uses the term 'notation' for surjective d: $\Sigma^* \rightarrow X$ with X countable and the term 'representation' for surjective d: $\Sigma^{\omega} \rightarrow X$ with X uncountable. I use the term 'notation' for both kinds of mapping.

Turing-style machines that respond to infinite strings on a read-only input tape by progressively writing infinite strings onto a write-only output tape. He defines:

> $f: \mathbb{R} \to \mathbb{R}$ is computable relative to d iff there exists a machine T such that, when supplied with a d-name for $x \in dom(f)$ on the input tape, T progressively writes a d-name for $f(x)$ onto the output tape.

Intuitively: T can compute any finite initial portion of a d-name for $f(x)$, given a sufficiently long finite initial portion of a d-name for $x \in dom(f)$. Developing these ideas rigorously requires various technicalities that need not concern us. The important point is that computable analysis offers a rigorously defined notion of *Turing computability relative to d* for functions over $\mathbb{R}$.

One can also rigorously define reducibility $\leq$ over notations $d: \Sigma^{\omega} \to \mathbb{R}$ and $e: \Sigma^{\omega} \to \mathbb{R}$, similar to reducibility over notations for countable domains. Intuitively: $d \leq e$ iff some machine computes any finite initial portion of an e-name for $z \in \mathbb{R}$, given a sufficiently long finite initial portion of a d-name for z. As Turing in effect observed, $d_{base\text{-}n} \leq d_{interval}$ but $\neg(d_{interval} \leq d_{base\text{-}n})$.

Computable analysis studies computational properties of notations $d: \Sigma^{\omega} \to \mathbb{R}$. For example, addition is not Turing-computable relative to $d_{base\text{-}10}$. To illustrate, suppose that the input base-10 names are '1.444…' and '1.555…' The sum has two base-10 names: '2.999…' and '3.000…'. One cannot determine from any finite initial portion of the input names that a corresponding output name should begin '2', since any finite initial portion is consistent with the sum being > 3. Nor can one determine from any finite initial portion of the input names that the output name should begin '3', since any finite initial portion is consistent with the denotation being < 3. Thus, no Turing-style machine can compute even the first digit of a $d_{base\text{-}10}$-name for $x+y$, given arbitrary finite initial portions of $d_{base\text{-}10}$-names for x and y.

In contrast, many familiar functions (e.g. addition, multiplication, exponentiation, trigonometric functions) are computable relative to $d_{interval}$. In fact, $d_{interval}$ is the unique notation (up to equivalence) that makes certain basic operations over $\mathbb{R}$ computable [Hertling, 1999]. On the other hand, $d_{interval}$ also has some disadvantages. Notably, the step function

$$step(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{otherwise} \end{cases}$$

is not computable relative to $d_{interval}$, since no finite set of open rational intervals containing $x = 0$ determines whether $step(x)$ should assume value 0 or 1. But *step* becomes computable if one represents outputs using $d_{>}$-names rather than $d_{interval}$-names.

As these examples illustrate, Multiplicity pervades computable analysis. There are many non-equivalent notations for $\mathbb{R}$, inducing distinct notions of computation over $\mathbb{R}$. It seems pointless to debate which notation yields 'true' computability over the reals. A notation may be more useful for certain mathematical purposes, but no one notation seems uniquely qualified above the others to deserve exclusive status.

A real number x is *computable relative to d* iff some Turing machine enumerates a d-name for x. Since there are countably many Turing machine programs, only countably many reals are computable relative to d. A real number is computable *simpliciter* iff it is computable relative to $d_{interval}$. $\mathbb{R}_c$ is the set of computable real numbers. Whenever $\mathbb{R}_c$ is the set of real numbers computable relative to d: $\Sigma^\omega \to \mathbb{R}$, one can easily transform d into a notation φ^d: $\Sigma^* \to \mathbb{R}_c$, where a φ^d-name for x encodes a Turing machine program that enumerates a d-name for x. Different notations d: $\Sigma^\omega \to \mathbb{R}$ can yield non-equivalent notations φ^d: $\Sigma^* \to \mathbb{R}_c$, inducing distinct notions of computability over $\mathbb{R}_c$. For example, $d_{interval}$ and $d_{base\text{-}10}$ yield non-equivalent notations

for $\mathbb{R}_c$. In this manner, computable analysis yields numerous examples of Multiplicity over a countable domain.

3. REPRESENTATION AS CENTRAL TO COMPUTABILITY THEORY

My survey of computability theory showcases the central role played by representation. Given the Gap Argument, we can study Turing computation over a non-linguistic domain only if we furnish a semantics for strings. Given Relativity, the non-linguistic function computed by a Turing machine depends upon our semantic interpretation. Some of the discipline's most basic concepts --- including *computation over* $\mathbb{N}$, *computation over* $\mathbb{P}$, and *computation over* $\mathbb{R}$ --- emerge only once we endow strings with representational import. Hence, many of the core phenomena studied by computability theory arise only once suitable representational relations are in place.

When the domain of computation is both infinite and non-linguistic, representation occupies an especially central role. Since Deviancy prevails, a satisfying account must address which notations are admissible for computation over the domain. What distinguishes acceptable and deviant notations? When Multiplicity prevails, further questions arise. How do various equivalence classes of notations relate to one another? What notion of computability does each equivalence class induce? Which such notions hold any interest? The representation relation becomes a central object of study. Reducibility serves as a vital tool, sorting notation systems into equivalence classes according to their computational properties. Thus, systematic investigation of notation systems is integral to mathematical theorizing about computation.

We may instructively compare computability theory with other developed branches of mathematics, such as number theory, analysis, set theory, algebra, and topology. Those fields

study a fixed mathematical structure (such as the natural numbers or the real numbers) or collection of mathematical structures (such as structures satisfying the axioms of a group or a topological space). One studies mathematical structures *in themselves*, without studying our representational access to those structures. Of course, one must represent a structure in order to study it. But number theory, algebra, analysis, topology, and set theory do not explicitly address our representational interface with the relevant structures. For example, number theory studies the natural numbers, without explicitly considering how we represent natural numbers. In contrast, computability theory studies human and machine computation over mathematical structures. This subject matter ensures a essential role for the representational interface between humans (or machines) and salient mathematical structures. Representation becomes an object of explicit study, in a way that it does not for most branches of mathematics.

3.1. Previous philosophical treatments of computation and representation

Many philosophers have emphasized the links between computation and representation. Indeed, philosophers sometimes overstate those links. Most dramatically, Fodor [1975] insists that *all* computation operates over entities with representational properties. He encapsulates his view through the slogan 'no computation without representation.' As Chalmers [2011] and Piccinini [2008] observe, Fodor's analysis seems implausible. Some computations do not require representation. A Turing machine might compute a string-theoretic function $\gamma: \Sigma^* \rightarrow \Sigma^*$. Strings manipulated by the machine need not have any semantic interpretation. For example, representational notions are irrelevant to Turing's discovery that the Entscheidungsproblem is unsolvable, because the Entscheidungsproblem concerns computation over strings drawn from a finite alphabet. A significant role for representation arises only when we consider computation

over non-linguistic domains. Only then do the Gap Argument and Relativity apply. Thus, a more accurate version of Fodor's slogan would be: 'no computation *over a non-linguistic domain* without representation.'

On the other hand, I think that Chalmers and Piccinini underplay the importance of representation to mathematical theorizing about computability. Chalmers writes: 'computations are specified syntactically, not semantically' [2011, p. 334], and 'the standard mathematical theory of computation, involving Turing machines and the like… seems to be largely nonsemantic' [2012, p. 218]. Similarly, Piccinini writes: 'representation does not seem to be presupposed by the notion of computation employed in computability theory and computer science' [2009, p. 519], and 'the whole mathematical theory of computation can be formulated without assigning any interpretation to the strings of symbols being computed' [2008, p. 212].

These passages omit much of what happens in the mathematical study of computability. Many computations of tremendous interest operate over non-linguistic domains. The Gap Argument and Relativity already show that we can describe those computations adequately only if we cite representational properties. If the domain is infinite, then Deviancy impels us to study computational properties of the representation relation. When Multiplicity prevails, comparative study of notation systems becomes even more essential. Mathematical theorizing is deeply concerned with computational phenomena that hinge upon representation, and it offers a systematic treatment of the requisite representational relations.

Let us distinguish between *computability theory in the narrow sense* and *computability in the broad sense*. *Computability theory in the narrow sense* studies computability over linguistic domains, such as strings drawn from a formal language. *Computability theory in the broad sense* studies computability *in general*, including computability over $\mathbb{N}$, $\mathbb{P}$, $\mathbb{R}$, and other non-linguistic

domains. One might object that, even if I have accurately described computability theory in the broad sense, Chalmers and Piccinini accurately describe computability theory in the narrow sense.

I happily agree that Chalmers and Piccinini accurately describe computability theory in the narrow sense. But I insist that computability theory in the narrow sense comprises a limited (albeit important) subset of overall mathematical research into computability. Mathematicians use computational tools to illuminate the representational relations that strings bear to natural numbers, partial recursive functions, real numbers, and other non-linguistic entities. One cannot define away this mathematical research by using the phrase 'computability theory' to mean 'computability theory in the narrow sense.'

In my opinion, it is far more natural to use the phrase 'computability theory' to denote the study of computability *in general*, rather than the study of computability *over linguistic domains*. I will therefore use the phrase 'computability theory' in the broad sense. However, my primary point here is not terminological. My primary point is that computation over certain important domains can be properly understood only when one elucidates computational properties of notations for those domains. By disregarding this point, Chalmers and Piccinini neglect vital computational phenomena.

Sieg's [2009] axiomatic treatment of computation likewise underplays representation. Sieg's axioms enshrine Turing-inspired constraints upon symbol manipulation by an idealized human computing agent. Sieg calls an agent satisfying those constraints a 'Turing computor.' In an implicit nod to the Gap Argument, Sieg writes: '[a] function **F** is (Turing) computable if and only if there is a Turing computor **M** whose computation results determine --- under a suitable encoding and decoding --- the values of **F** for any of its arguments' [2009, p. 599]. Sieg says

nothing about what makes an encoding 'suitable' even in the case of $\mathbb{N}$, let alone $\mathbb{P}$ or $\mathbb{R}$. He does not mention the mathematical literature on this topic. Sieg's axioms describe symbol manipulation in purely non-semantic terms, i.e. the axioms ignore any semantic relations between symbols and what symbols represent. Representation figures in his discussion as a casual afterthought. Reading Sieg, one would never guess how thoroughly representational notions pervade the mathematical study of computability. In particular, one would never guess the central role assigned to computational properties of notation systems.

By highlighting that central role, I have sought to fill a notable gap in the philosophical literature. I have showcased crucial aspects of mathematical practice heretofore underappreciated by philosophers.

3.2. The string-theoretic reinterpretation strategy

One might hope to construe recursion theory as concerned solely with computable functions over Σ^*. What would we lose by studying string-theoretic functions $\gamma: \Sigma^* \rightarrow \Sigma^*$ rather than numerical functions $f: \mathbb{N} \rightarrow \mathbb{N}$? Why bother studying computation over numbers when we can instead study computation over numerals? More generally, one might hope to construe the mathematical study of computability in non-semantic terms. Why not concentrate entirely on string manipulation, considered in detachment from any semantic interpretation of strings? In short, why not simply confine ourselves to computability theory in the narrow sense?

Let us call this approach *the string-theoretic reinterpretation strategy*. Note that the strategy is *revisionary* regarding mainstream mathematical practice. Since the 1930s, recursion theorists have studied computability over natural numbers *in addition to* computability over

strings. Thus, the string-theoretic reinterpretation strategy jettisons a fundamental, longstanding commitment of mathematical research into computability [Rescorla, 2007].

Even if we were to replace computation over numbers with computation over strings, the string-theoretic reinterpretation strategy would be untenable for recursion theory as a whole. As we have seen, indexing of partial recursive functions is a central plank of recursion theory, underlying most of the field's basic results: the Enumeration theorem; the Parameterization theorem; undecidability of the Totality Problem; Kleene's recursion theorem; and so on. These theorems involve computations that take partial recursive functions as inputs or outputs. In that sense, recursion theory presupposes a representational mapping from strings to partial recursive functions. Crucially, the presupposition persists *even if we ignore number-theoretic computation and focus instead on string-theoretic computation*. Suppose we eschew all reference to numerical functions f: $\mathbb{N} \rightarrow \mathbb{N}$, instead developing recursion theory for string-theoretic functions γ: $\Sigma^* \rightarrow \Sigma^*$. We still want to prove standard recursion-theoretic results such as the Enumeration and Parameterization Theorems. We must therefore introduce an indexing scheme φ: $\Sigma^* \rightarrow \mathbb{P}$.

If we attend solely to string manipulation, then we cannot express basic results such as the Enumeration and Parameterization theorems. Stating those results requires us to cite a mapping from Σ^* to $\mathbb{P}$. Only when we cite this mapping can we *fully* describe key properties of the relevant computations --- such as that some computation converts inputs x and y into the value (if any) that φ_x assumes on input y. The mapping φ: $\Sigma^* \rightarrow \mathbb{P}$ reflects a semantic interpretation on which strings denote partial recursive functions.

The string-theoretic reinterpretation strategy seems equally untenable when applied to computable analysis. A top research priority in that field is to compare different ways of representing real numbers. More specifically, researchers compare diverse notions of

computability induced by diverse notations d: $\Sigma^{\omega} \rightarrow \mathbb{R}$. If we were to focus exclusively on computation over infinite strings, then we would abandon all this mathematical research. Computable analysis would cease to exist in anything resembling its current form. We could still describe the same underlying manipulations of infinite strings, but many elementary discoveries would no longer expressible. For example, we could no longer say that addition and multiplication over $\mathbb{R}$ are uncomputable relative to base-10 notation. Stating these results requires explicit comparison of notation systems.

3.3. Formal apparatus versus intuitive grounding

Readers may protest that we can formulate the definitions and theorems of computability theory *even construed in the broad sense* without deploying semantic notions. After all, how does semantics enter computability theory? Through surjective functions d: $\Sigma^* \rightarrow X$ or d: $\Sigma^{\omega} \rightarrow X$. We can model these functions as sets of ordered pairs. One might insist that we can pursue computability theory *even construed in the broad sense* by citing appropriate set-theoretic entities, without any official talk about 'representation,' 'semantic interpretation,' 'naming,' or the like. Representational locutions are a heuristic crutch that we discard when formulating computability theory more rigorously.

I concede that the formal definitions and theorems of computability theory do not cite representation. The official definitions only mention set-theoretic functions d: $\Sigma^* \rightarrow X$ and d: $\Sigma^{\omega} \rightarrow X$, without explicitly mentioning representational relations between linguistic and non-linguistic items. Nevertheless, I will now argue that representational concepts play an indispensable role within computability theory.

As many authors have stressed [Gödel, 1972/1990, p. 275, fn. 5; Rogers, 1987, pp. 1-20; Soare, 1996], computability theory is fundamentally concerned with *intuitive* notions of computation over various structures, including $\mathbb{N}$, $\mathbb{R}$, and so on. What makes the formal theory so important are its links to our intuitive notions. As Kripke puts it: 'independently of any idea of intuitive computability, one can state the formal definitions of the theory… However, without the idea of intuitive computability, the entire motivation of the theory would be lost' [2011, p. 344]. Even though one can state computability theory's formal definitions and theorems without mentioning intuitive computability, one fully understands the *significance* of these definitions and theorems only when one relates them to intuitive computability.

Classical recursion theory provides a clear illustration. We begin with an intuitive concept (*computable number-theoretic function*), and we then isolate a corresponding formal definition (*partial recursive function*). Church's thesis asserts that the intuitive concept and the formal definition are extensionally equivalent. The formal definition commands so much interest precisely due to this putative extensional equivalence. When we prove a theorem involving the formal definition, we feel confident that our theorem illuminates intuitive computability over $\mathbb{N}$.

The links between intuitive concepts and formal definitions are less straightforward but no less important for other domains. For example, we have not one but numerous intuitive notions of computation over $\mathbb{R}$, corresponding to non-equivalent notations for real numbers. Computable analysis elucidates these informal notions through formal definitions. It reveals that certain informal notions (e.g. *computability relative to base-10 notation*) are not as mathematically fruitful as we might have hoped, thereby leading us to prioritize others (e.g. *computability relative to rational open interval notation*). Thus, the relations between formal apparatus and informal grounding are typically more dynamic than the special case of $\mathbb{N}$ might

suggest. What matters at present is that those relations are fundamental to computability theory. The field's definitions and theorems merit such intense interest because they purportedly explicate, refine, or illuminate various intuitive notions of computability. Absent such ties to our intuitive notions, the definitions and theorems would articulate a formal mathematical structure lacking the profound significance that computability theory in fact enjoys.

Representational concepts play a key role in connecting formal mathematical theorizing with our intuitive conception. Admittedly, we can define the formal notion *f*: $X \rightarrow Y$ *is computable relative to d and e* while treating *d* and *e* as purely set-theoretic functions, without any appeal to representation. But our choice of definition reflects our conviction, codified by the Gap Argument, that string manipulation implements computation over a non-linguistic domain only if the strings represent non-linguistic items. The surjective functions *d*: $\Sigma^* \rightarrow X$ and *d*: $\Sigma^\omega \rightarrow X$ serve as formal mathematical counterparts to representational relations between strings and non-linguistic items. We regard the formal notion *f*: $X \rightarrow Y$ *is computable relative to d and e* as a good formal proxy for our pre-theoretic notion of computability only because we regard the functions *d* and *e* as good formal proxies for pre-theoretic representational relations between strings and non-linguistic items. Until we connect the formal apparatus to our pre-theoretic concept of representation, we cannot link it to our pre-theoretic concept of computation over a non-linguistic domain.

Representational concepts also shape *which* surjective functions *d*: $\Sigma^* \rightarrow X$ and *d*: $\Sigma^\omega \rightarrow X$ computability theorists investigate. There are uncountably many equivalence classes of functions *d*. Mathematical inquiry centers upon a relatively small number of equivalence classes. How do mathematicians choose which equivalence classes to study? By reflecting upon standard representational schemes, such as d_{binary}. Turing-computability relative to d_{binary} is a more

important formal notion than Turing-computability relative to the deviant notation d_A, partly because the former notation aligns much more closely with our normal way of representing natural numbers for computational purposes.

To illustrate the crucial importance of representation, consider the Totality Problem. There is no Turing machine *T* such that

Given input *x*, *T* outputs $\underline{1}$ if φ_x is total and $\underline{0}$ if φ_x is not total,

where $\varphi: \Sigma^* \rightarrow \mathbb{P}$ is our canonical notation for partial recursive functions. We can state and prove this theorem without deploying intuitive notions of computation or representation. But the theorem is so significant only because it illuminates an intuitive notion of computation over the partial recursive functions. Based on the theorem, we conclude that the Totality Problem is 'undecidable,' i.e. there is no mechanical procedure for deciding whether a partial recursive function is total. Our conclusion involves the pre-theoretic notion of *mechanical procedure*. It reflects our conviction that φ is a good formal proxy for all representational schemes one might legitimately employ when computing over partial recursive functions. We presume that any admissible notation for $\mathbb{P}$ is computably intertranslatable with φ. Only given this presumption does the formal theorem entail that the Totality Problem is 'undecidable.'

The interconnections between formal apparatus, intuitive computability, and representation become particularly salient when Multiplicity prevails. Turing's original discussion already highlights the interconnections. His 1936 paper uses $d_{base\text{-}2}$. In the 1937 correction, he argues that $d_{base\text{-}2}$ notation does not let us compute an operation that we would like to compute (roughly, computation of a real number *r* from arbitrarily tight rational upper and

lower bounds for r).[9] He recommends that we replace $d_{base\text{-}2}$ with $d_{interval}$, which renders the desired operation computable. So Turing motivates a shift in his formal theory by reflecting upon intuitive computability, and he implements the shift by considering rival schemes for representing real numbers. Subsequent research within computable analysis employs similar methodology. Researchers routinely assess and revise formal definitions by adducing pre-theoretic notions of computation and representation. Many key results (e.g. uncomputability of addition relative to $d_{base\text{-}10}$) are so interesting precisely because they illuminate which ways of representing real numbers support which mechanical operations.

In summary, representational concepts crucially inform the aims and methods of computability theory. Talk about 'representation,' 'notational systems,' and 'names' is no mere heuristic gloss. Computability theorists advance their formal definitions and theorems so as to illuminate computations that presuppose representational relations between linguistic and non-linguistic items. To explain why this particular formal apparatus merits special attention, and to articulate how the formal apparatus bears upon our intuitive conception of computability, we must consider the relevant representational relations.

Over the past century, philosophers, logicians, and mathematicians have often discussed whether set theory can provide a unifying framework for mathematics. Researchers widely agree that one can model all 'normal' mathematics within a sufficiently powerful set theoretic system, such as ZFC. Quine concludes that 'all mathematical truth can be seen as truth of set theory' [1966, p. 31]. However, some authors worry that formalization with set theory distorts important mathematical phenomena. Our discussion bolsters this worry *for the special case of computability theory*. Granted, set theory can model the formal mathematical structures studied

[9] As Gherardi [2011] notes, Turing's argument is unsatisfactory. The argument presupposes Turing's modified injective version of $d_{base\text{-}2}$ (see note 7), rather than normal non-injective $d_{base\text{-}2}$. Despite this expository flaw, it is clear that Turing perceived the computational deficiencies of $d_{base\text{-}2}$.

by computability theory. But formalization within set theory omits crucial ties between those structures and our intuitive starting point. The formal apparatus illuminates how different ways of representing a domain support different mechanical operations over that domain. It thereby illuminates which functions over the domain are intuitively computable. If we restrict ourselves to the language of set theory, we cannot describe these ties between formal apparatus and pre-theoretic notions. We cannot even state Church's thesis. Formalization within set theory does not capture the full scientific content of computability theory.[10]

3.4. Worries about mathematical representation

Philosophers have extensively debated how we are able to represent mathematical entities and even *whether* we succeed in doing so. I want to clarify my analysis of computability theory by briefly engaging with these debates.

Modern mathematics routinely postulates numbers, functions, sets, and other mathematical entities. At first blush, anyone who accepts modern mathematics is committed to the existence of these entities. Some philosophers disagree. They recommend that we interpret (or reinterpret) mathematical discourse in nominalist terms that avoid all apparent reference to mathematical entities. As many critics have noted, it is unclear whether one can develop a satisfying nominalist construal of mathematical discourse. In any case, I assume the existence of natural numbers, real numbers, partial recursive functions, etc.

Some philosophers grant the existence of mathematical entities while denying that we achieve determinate reference to them. Consider *true arithmetic*: the theory containing all true sentences of the language of first-order arithmetic. The *standard model* of true arithmetic is

[10] Kripke [2013, p. 95] makes a similar point, focusing on the 'easy half' of Church's thesis (that all partial recursive functions are intuitively computable).

given by the familiar natural number sequence: 0, 1, 2, … There also exist models of true arithmetic that are not isomorphic to the standard model. Thus, first-order arithmetical truth does not privilege the standard model as the intended interpretation of arithmetical discourse. But then what makes the standard model 'intended'? Perhaps mathematical practice does not fix any determinate structure as 'the' natural numbers. Similar worries arise for other domains, including real analysis. The standard model of first-order analysis has an uncountable domain, corresponding to the familiar real number line. The Löwenheim-Skolem theorem also supplies a non-standard countable model. But then what makes the standard model 'intended'? Perhaps mathematical practice does not fix any determinate structure as 'the' real numbers. *Model-theoretic skepticism* along these lines traces back to Skolem [1922/1967]. Putnam [1980] espouses model-theoretic skepticism regarding set theory. In certain passages [1979, p. 22; 1981, p. 67], he appears to espouse model-theoretic skepticism regarding arithmetic.

Model-theoretic skepticism may seem to endanger my analysis of computability theory. I have claimed that computability theorists study computation over the natural numbers, the real numbers, and other mathematical structures. But what makes it the case that one is computing over the 'intended' structure, rather than a non-isomorphic structure? Perhaps mathematical practice does not fix any determinate structure as the one over which a human or machine computes. For example, what ensures that some Turing machine is computing over the standard natural numbers? One might worry that we can just as well reinterpret recursion theory as concerning Turing-computation over a non-isomorphic model [Araújo and Carnielli, 2012; Dean, 2014].

I think that we may safely dismiss such worries. Existing arguments for model-theoretic skepticism are unconvincing [Bays, 2001]. Most such arguments assume a premise along the following lines:

> The only constraint on a good interpretation of mathematical discourse is that it make certain sentences true.

This premise, coupled with the existence of non-standard models, entails that one cannot 'latch onto' a determinate interpretation. Fortunately, we need not grant the premise [Lewis, 1984]. We need not concede that good interpretation of mathematical discourse is constrained *only* by the requirement that certain sentences come out true. Many additional factors may constrain good interpretation of mathematical discourse, including mathematical factors [Gaifman, 2004; Halbach and Horsten, 2005], physical factors [Field, 2001, pp. 332-360], cognitive factors [Peacocke, 1998], and communicative factors [Parsons, 2008, pp. 279-293]. For all the model-theoretic skeptic has argued, such factors may help privilege a determinate interpretation.[11]

I therefore assume that model-theoretic skepticism is mistaken. I assume that arithmetic concerns a unique structure (up to isomorphism). Similarly for real analysis. This paper is directed towards readers who share my assumptions. Given my assumptions, we may safely assume that humans and machines compute over determinate structures, including the natural numbers and the real numbers. Computability theory studies these computations, along with representational relations that make the computations possible.

Even if one rejects model-theoretic skepticism, pressing questions remain concerning our ability to represent determinate mathematical structures. How do we 'latch onto' the standard

[11] Putnam [1980] dismisses such additional constraints as 'just more theory,' i.e. just more sentences subject to non-standard interpretation. Most discussants hold that Putnam's 'just more theory' response blurs the vital distinction between *describing* why some model is the intended interpretation and *adding more sentences* that should come out true under any good interpretation. See [Bays, 2001] for discussion, with references to the literature.

model of arithmetic or analysis, rather than some non-standard model? If first-order arithmetical truth does not fix a unique interpretation for arithmetical discourse, then what does? Such questions are especially puzzling in light of the fact, emphasized by Benacerraf [1973], that causal interaction with mathematical entities seems impossible. The questions merit thorough investigation. But I think we should investigate them under the working assumption that model-theoretic skepticism is false. Moreover, one need not answer these questions in order to conduct fruitful philosophical inquiry premised on the falsity of model-theoretic skepticism.

Recently, some writers have suggested that computational considerations can illuminate reference to the natural numbers [Halbach and Horsten, 2005; Horsten, 2012]. They cite Tennenbaum's theorem that non-standard models of arithmetic render addition non-recursive. They conclude that the standard model is privileged over non-isomorphic models. Sometimes the goal behind this computationalist agenda is to rebut model-theoretic skepticism about arithmetic. Sometimes the goal is more modest: to illuminate mathematical reference *under the assumption that model-theoretic skepticism is false*. Button and Smith [2012] and Dean [2014] critique the computationalist agenda. I take no stand in these debates. I do not pursue the computationalist agenda, nor do I criticize it. My topic is not whether computational considerations help select a determinate interpretation for arithmetical discourse. I simply assume that relevant mathematical vocabulary has a determinate interpretation (up to isomorphism).

4. COMPUTABILITY THEORY: AN INTENSIONAL ENTERPRISE

To study computation over a non-linguistic domain *X*, computability theorists adduce linguistic items that *name* elements of *X*. They develop a computability theory for names, and they then transfer this theory from names to objects named. Thus, the field's typical methodology is to

consider computations over *representations* of mathematical entities. In that sense, computability theory adopts an *intensional* viewpoint. By an 'intensional' viewpoint, I mean one that takes into account how objects are represented.

Within classical recursion theory, we can usually discount differences among notation systems. All admissible notations over $\mathbb{N}$ yield the same notion of computability. We can therefore develop an *extensional* theory of computation over $\mathbb{N}$, that is, a theory that ignores how natural numbers are represented [Rogers, 1987, p. 10]. When Multiplicity prevails, a purely extensional theory is not viable. We must explicitly relativize to notational systems. Even when Multiplicity does not prevail, so that a purely extensional theory becomes possible, our extensional theory rests upon an explanatorily prior theory of computation over names. Inevitably, then, computability theory assigns explanatory priority to the intensional viewpoint over the extensional viewpoint. We cannot disentangle the study of computation over a non-linguistic domain from the study of how one represents the domain.[12]

Exclusive focus on $\mathbb{N}$ encourages neglect of computability theory's intensional aspects. Domains where Multiplicity prevails offer a salutary corrective, reminding us that computation over a non-linguistic domain always presupposes an invidious distinction among notation systems that represent the domain.

When we study computation over a non-linguistic domain, we are studying traits that adhere in the first instance to objects as represented in suitable ways, not to objects in themselves. One computes an output *as represented in a suitable way* from an input *as represented in a suitable way*. Hence, our core subject matter mandates an intensional viewpoint.

[12] Intensional aspects of computability theory are widely recognized among computer scientists and mathematicians [Abramsky, 2013; Feferman, 2013]. They receive much less attention from philosophers, Dean [2014] and Shapiro [2000] being notable exceptions. Shapiro's discussion of intensionality emphasizes *representations of the function computed*, whereas I emphasize *representations of the function's inputs and outputs*.

This mandate differentiates computability theory from most other branches of mathematics, including number theory, analysis, set theory, algebra, and topology.

5. 'A FRANKLY INEQUALITARIAN ATTITUDE'

Let us explore the intensional aspects of computability theory by considering *quantification into referentially opaque contexts*.[13]

A referentially opaque context arises when substitution of co-referring expressions does not preserve truth-value. *Propositional attitude attributions* generate opaque contexts: even though Mark Twain is Samuel Clemens, one can believe that Mark Twain is famous without believing that Samuel Clemens is famous. Quine [1966; 1981] compares the sentences:

(1) Ralph believes that $(\exists x)(x$ is a spy).

(2) $(\exists x)$(Ralph believes that x is a spy).

There is a clear intuitive difference: (1) attributes a belief that spies exist, while (2) attributes suspicion that a specific individual is a spy. Now suppose Ralph believes that $(\exists x)(x$ is a spy) and also believes that no two spies are the same height. He concludes that the shortest spy is a spy. Yet (2) may be false. Apparently, then, one can infer (2) only from certain sentences of the form

(3) Ralph believes that a is a spy.

(2) is true only if Ralph attributes spyhood to an individual *as represented in a suitable way*. In Quine's [1966, p. 184] words, quantifying into opaque contexts presupposes a 'frankly inequalitarian attitude toward the various ways of specifying' an object.[14]

---

[13] Shapiro [2000, p. 46] also mentions the connection with quantification into opaque contexts, although he pursues the connection from a rather different angle.

[14] Kaplan [1969] attempts to articulate the requisite 'frankly inequalitarian' attitude. He proposes that the inference from (3) to (2) is licensed just in case '*a*' is *vivid* and '*a*' is a name *of* its denotation for Ralph. Unfortunately, Kaplan leaves the crucial *vivid*/*non-vivid* distinction fairly obscure.

Following Hintikka [1962], Quine eventually [1981, pp. 120-122] decides that the inference from (3) to (2) is licensed only when Ralph *knows which* entity '*a*' designates. Quine argues that the notion *knowing which* is highly interest-relative. He concludes that the difference between (1) and (2), which initially seems so sharp, is distressingly context-sensitive. He despairs of finding any systematic criterion that dictates when (2) is true. Detecting no firm scientific basis for the 'frankly inequalitarian' attitude presupposed by (2), he recommends that we purge (2) and kindred locutions from rigorous scientific theorizing.

Kripke deploys computability theory to rebut Quine. Restricting attention to computation over $\mathbb{N}$, Kripke writes [2011, p. 261]:

> A computable function is a function *f* such that for each given *n*, if you put in a particular number, the value *f*(*n*) can be computed. And what does that mean? That given the definition of the function and an argument *n*, you can know, by computation, what the value *f*(*n*) is. This would hardly make sense if all ways, even mathematical ways, of designating a number, were on a par. For then every function would be computable, since the value of *f* for a given *n* could simply be 'computed' as *f*(*n*)! To say otherwise would be to adopt an 'inequalitarian' attitude towards different ways of designating a number, supposedly a sin.

Kripke's example is a special case of the general phenomenon discussed in Section 2.2. If we define a notation d_f: $\Sigma^* \rightarrow \mathbb{N}$ that maps $\underline{n}$ to the *n*th element in the sequence:

0, 1, *f*(0), 2, *f*(1), *f*(*f*(0)), 3, *f*(2), *f*(*f*(1)), *f*(*f*(*f*(0))), …

then *f* is Turing-computable relative to d_f.

I agree with Kripke that computability theory poses a serious challenge to Quine's position. The central dichotomy of the entire discipline --- *computable function* versus *non-*

computable function --- presupposes an invidious distinction between admissible and deviant notations. Thus, computability theory requires us to privilege certain ways of representing entities over alternative ways of representing those same entities. Computability theory requires the 'frankly inequalitarian attitude' decried by Quine.

According to Kripke, 'the notion of computability is best seen as having a procedure for knowing which number is the value of the function' [2011, p. 344]. He also suggests that one knows which number is *f*'s value when one computes that $f(n) = m$, for some numeral '*m*' in a canonical notation system (such as Arabic decimal notation). In response, Burge complains that computation resulting in a long Arabic decimal numeral '*m*' does not normally suffice for 'knowing which' number is $f(n)$: 'One needs to do some figuring, calculating, grouping, or simplifying of a thirty-seven-figure numerical name to grasp which number it names' [2007, p. 73]. These worries ramify when one generalizes to other domains of computation. For example, does one 'know which' computable real number *r* is at issue simply from grasping a $\varphi^{base\text{-}10}$ name for *r*? No matter how well one understands the name, one may have little idea which base-10 decimal the name encodes. Yet computability relative to $\varphi^{base\text{-}10}$ is a non-deviant kind of computability.

Kripke might reply that *in some sense* one 'knows which' value is at issue even from a very long Arabic decimal numeral or a $\varphi^{base\text{-}10}$ name. Perhaps so. Overall, though, I think it more fruitful to formulate Kripke's anti-Quinean perspective without relying upon the 'know which' locution. I now attempt to do so.

5.1. Opacity and computability

Intuitively, a mechanical procedure *P* computes function *f* just in case *P* yields the correct output *f*(*x*) given any input *x*. In this spirit, Soare describes an *algorithm* for computing *f* as 'a finite set of instructions which, given an input *x*, yields after a finite number of steps an output $y = f(x)$' [1987, p. 8]. More formally, we may write:

(4) Mechanical procedure *P* computes function *f* iff $(\forall x)(\forall y)(\, f(x) = y \leftrightarrow P$ yields output *y* on input *x*).[15]

As Kripke's discussion highlights, we must interpret (4) carefully so as to avoid trivializing the notion of computability. For any total numerical function *f*, there is a mechanical procedure P_f such that, for all *n*,

(5) P_f yields output $f(n)$ on input *n*;

namely, the procedure that replaces an input numeral '*n*' with output symbol '$f(n)$'. P_f yields a unique output for each input, so we have

(6) P_f yields output *y* on input $n \rightarrow f(n) = y$.

Applying Leibniz's law to (5), we infer

(7) $f(n) = y \rightarrow P_f$ yields output *y* on input *n*.

From (4), (6), and (7), we infer that P_f computes *f*. Since *f* is an arbitrary total numerical function, we have collapsed the distinction between computability and non-computability.

How can we avoid this disastrous result? Intuitively, the reason why P_f does not compute *f* is that P_f represents *f*'s output using the unsuitable term '$f(n)$'. A procedure that computes *f* must represent inputs and outputs using a suitable canonical notation. If we want to preserve (4) while honoring these intuitions, then the most promising strategy is to regard the *y* position in

(8) *P* yields output *y* on input *x*

[15] My formulation expresses an intuitive analogue to *strong* computability (see note 5).

as opaque. We can then say that $f(n) = m$ while denying that

(9) P_f yields output m on input n,

where 'm' is a numeral in a canonical notation system (e.g. Arabic decimal notation). Perhaps there is also a reading on which (8) is transparent rather than opaque. On that putative reading, (9) is true. But no such reading is relevant to computability theory. Instead, we require a reading on which (9) is false. We must construe (8) as an *intensional transitive*, akin to locutions such as 'admires,' 'seeks,' and so on. Just as there is a natural reading on which it may be that

Ralph admires Mark Twain.

Ralph does not admire Samuel Clemens.

Mark Twain = Samuel Clemens.

so our desired reading supports

P_f yields output $f(n)$ on input n.

P_f does not yield output m on input n.

$f(n) = m$.

Since substitution of co-referring terms does not preserve truth-value, the inference from (5) to (7) is blocked.[16] We thereby preserve a robust distinction between computability and non-computability. The price we pay is quantifying into an opaque context.[17]

The quantifier in (4) is universal, not existential. For an example involving existential quantification, consider:

---

[16] We can likewise say that, even though (5) is true, the open sentence 'P_f yields output y on input x' is not true under a variable assignment that assigns n to 'x' and $f(n)$ ($= m$) to 'y'. Standard Tarskian semantics then entails that '$(\forall x)(\forall y)($ $f(x) = y \leftrightarrow P_f$ yields output y on input x)' is not true.

[17] One might propose that 'yields' is a disguised quotational device, so that a more proper rendering of *P yields output m on input n* would be *P yields output 'm' on input 'n'*, or perhaps *P yields output 'm' with denotation m on input 'n' with denotation n.* According to this proposal, (4) quantifies into quotation marks. Since quantification into quotation marks does not seem intelligible, the proposal impugns (4)'s intelligibility. I respond that, although there may be a reading on which 'yields' involves disguised quotation, there is also a reading on which it does not. For example, there is a reading on which *P yields output 15 on input 39* is true even though *P* operates over binary numerals rather than base-10 numerals. By adopting such a reading, we can make good sense of (4).

(10) Ralph uses mechanical procedure *P* to compute *f*'s value on input *n*.

Statements of this kind figure prominently in the pre-theoretic discourse that gives rise to computability theory. We may naturally regiment (10) as

(11) (∃*y*)(Ralph uses mechanical procedure *P* to compute that *f* has output *y* on input *n*).

Perhaps there is a reading on which (11) is true no matter how Ralph represents *f*'s output. On this putative reading,

Ralph uses mechanical procedure P_f to compute that *f* assumes output *f*(*n*) on input *n*.

entails

Ralph uses mechanical procedure P_f to compute *f*'s value on input *n*.

No such reading is relevant to computability theory. Instead, we require a reading on which (11) is true if Ralph represents *f*'s output using an admissible notation (e.g. binary notation) but false if Ralph represents *f*'s output using a deviant notation (e.g. d_f). On the requisite reading, there is a huge difference between (11) and

(12) Ralph uses mechanical procedure *P* to compute that (∃*y*)(*f* has output *y* on input *n*).

(12) attributes a generic realization that *f* attains *some* output on input *n*, while (11) attributes a suitable grasp of the specific numerical output. Quine might regard the distinction between (11) and (12) as too context-sensitive or interest-relative for serious scientific purposes. Nevertheless, a distinction along those lines underlies computability theory.[18]

The opacity in (4) and (11) would repay considerable study. What is the semantics for these locutions? How should one formulate suitable quantifier rules? I have not attempted to

[18] In the unpublished Whitehead lectures 'Logicism, Wittgenstein, and *De Re* Beliefs about Numbers,' Kripke suggests that computability theory enshrines a distinction between *de dicto* and *de re* beliefs about numbers: e.g. *believing that f has some output on input n* versus *believing* of *some particular number that it is f's output on input n.* Authors use the *de dicto*/*de re* terminology in so many different ways that I have thought it best to avoid this terminology when developing my position.

answer such questions. I have only undertaken the fairly modest task of showing that the relevant contexts are opaque.

5.2. Intensional aspects of computability theory

When elucidating basic aspects of computability, we find ourselves drawn to the linguistic device castigated by Quine: quantification into an opaque context. Admittedly, one can state computability theory's formal definitions and theorems without employing locutions such as (4) or (11). However, as I argued in Section 3.3, fully understanding the formal definitions and theorems requires that we relate them to our pre-theoretic conception of computation. (4), (11), and kindred locutions naturally arise when we articulate how the formal theory bears upon our intuitive conception. Evidently, Quine overlooked the extent to which mathematical science embraces the intensional phenomena he deemed so problematic.

We can avoid quantifying into opaque contexts by introducing a quantifier over names. We can define *mechanical procedure P computes function f relative to notation d*, and we can say that *f* is computable iff some mechanical procedure computes *f* relative to an admissible notation. This alternative procedure only accentuates the need for distinguishing between admissible and deviant notations. We can honor the distinction either *implicitly* (by quantifying into an opaque context) or *explicitly* (by overtly privileging certain notations over others). Either way, we adopt a 'frankly inequalitarian attitude toward the various ways of specifying' objects in the domain of computation. Such an attitude is needed to preserve the distinction between computable and non-computable functions.

To illustrate, suppose we want to say that some total number-theoretic function *f* is not intuitively computable. We do not capture this fact in full generality if we only mention a single

privileged notation for $\mathbb{N}$, because some mechanical procedures operate relative to other admissible notations. To attain a suitably general formulation, we might say:

(13) There is no mechanical procedure *P* and no admissible notation *d* for the natural numbers such that $(\forall x)(\forall y)(\ f(d(x)) = d(y) \leftrightarrow P$ yields output *y* on input *x*),

where quantifiers range over numerals. Alternatively, we might avoid explicit talk of notations:

(14) There is no mechanical procedure *P* such that $(\forall x)(\forall y)(\ f(x) = y \leftrightarrow P$ yields output *y* on input *x*),

where quantifiers range over natural numbers. As I have argued, (14) comes out true only if we construe it as quantifying into an opaque context. Whether we choose (13) or (14), we draw an invidious distinction between notations. We do so either *explicitly* (by overtly mentioning admissibility) or *implicitly* (by quantifying into an opaque context). Either way, we must consider how natural numbers are represented. When expressing with full generality that *f* is not intuitively computable, we find ourselves adopting an intensional viewpoint.

Some readers may worry that formulations such as (13) and (14) veer dangerously close to vicious circularity. My formulations rely, either explicitly or implicitly, upon an unexplicated notion of 'admissible notation.' It may therefore seem that I am assuming some unexplicated notion of *computability* for notation systems. Assuming such a notion would surely be illegitimate when we are trying analyze numerical computability in non-circular terms.

I reply that I am not trying to analyze numerical computability in non-circular terms. In fact, I have argued elsewhere that there is no evident way to provide a non-circular conceptual analysis of numerical computability, precisely because we do not know how to analyze the notion 'admissible notation' in satisfying non-circular terms [Rescorla, 2007]. I do not advance (13) and (14) as contributions to a *non-circular analysis* of computability. I advance them as

theoretically central and *extensionally correct* statements about computability. That we find ourselves assigning a central role to such locutions demonstrates the ineliminably intensional nature of computability theory.

6. CONCLUSION

Computability theory offers a rigorous study of the interplay between ways of representing a domain and finite mechanical operations over that domain. It embeds reciprocal explanatory relations between computation and representation: we need representational notions to understand computation over non-linguistic domains; and computational tools illuminate key properties of representational systems. By further exploring these reciprocal explanatory relations, we may hope to gain substantial insight into representation as well as computation. Just as philosophical reflection upon set theory has illuminated fundamental ontological and epistemological questions surrounding abstract entities, philosophical reflection upon computability theory can illuminate fundamental questions surrounding representation and intensionality. We have only begun to mine computability theory for its philosophical payoff.

ACKNOWLEDGMENTS

I am grateful to audience members at the SoCal PhilMath + PhilLogic + FOM 2013 workshop, especially D. A. Martin and Kai Wehmeier, where I presented an earlier version of this material. I also thank Gualtiero Piccinini, Nathan Salmon, and an anonymous referee for this journal for comments that improved the paper.

REFERENCES

Abramsky, S. [2013]: 'Two puzzles about computation', in S. B. Cooper and J. van Leeuwen, eds, *Alan Turing: His Work and Impact*, pp. 53-56. Waltham: Elsevier.

Araújo, A., and Carnielli, W. [2012]: 'Non-standard numbers: a semantic obstacle for modeling arithmetical reasoning', *Logic Journal of IGPL* **20**, 477-485.

Bays, T. [2001]: 'On Putnam and his models', *The Journal of Philosophy* **98**, 331-350.

Benacerraf, P. [1973]: 'Mathematical truth', *The Journal of Philosophy* **70**, 661-679.

Boolos, G. and Jeffrey. R. [1980]: *Computability and Logic*, 2nd ed. Cambridge: Cambridge University Press.

Burge, T. [2007]: *Foundations of Mind*. Oxford: Oxford University Press.

Button, T., and Smith, P. [2012]: 'The philosophical significance of Tennenbaum's theorem', *Philosophia Mathematica* **20**, 114-121.

Chalmers, D. [2011]: 'A computational foundation for the study of cognition', *Journal of Cognitive Science* **12**, 323-357.

---. [2012]. 'The varieties of computation: a reply', *The Journal of Cognitive Science* **13**, 211-248.

Church, A. [1936]: 'An unsolvable problem of elementary number theory', *American Journal of Mathematics* **58**, 345-363.

Copeland, J., and Proudfoot, D. [2010]: 'Deviant encodings and Turing's analysis of computability', *Studies in History and Philosophy of Science A* **41**, 247-252.

Davis, M. [1958]: *Computability and Unsolvability*. New York: McGraw-Hill.

Dean, W. [2014]: 'Models and computability', *Philosophia Mathematica* **22**, 143-166.

Feferman, S. [2013]: ‘About and around computing over the reals’, in B. J. Copeland, C. Posy, and O. Shagrir, eds, *Computability: Turing, Gödel, Church, and Beyond*, pp. 55-76. Cambridge: MIT Press.

Field, H. [2001]: *Truth and the Absence of Fact*. Oxford: Clarendon Press.

Gaifman, H. [2004]: ‘Non-standard models in a broader Perspective’, in A. Enayat and R. Kossak, eds, *Non-standard Models of Arithmetic and Set Theory*, pp. 1-22. New York: American Mathematical Society.

Gherardi, G. [2011]: ‘Alan Turing and foundations of computable analysis’, *The Bulletin of Symbolic Logic* **17**, 394-430.

Gödel, K. [1934/1986]: ‘On undecidable propositions of formal mathematical systems’, rpt. in S. Feferman, J. Dawson, S. Kleene, G. Moore, R. Solovay, and J. Heijenoort, eds, *Collected Works*, vol. 1, pp. 346-372, Oxford: Oxford University Press.

---. [1972/1990]: ‘On an extension of finitary mathematics which has not yet been used’, rpt. in S. Feferman, J. Dawson, S. Kleene, G. Moore, R. Solovay, and J. Heijenoort, eds, *Collected Works*, vol. 2, pp. 271-280. Oxford: Oxford University Press.

Grzegorczyk, A. [1955]: ‘Computable functionals’, *Fundamenta Mathematica* **42**, 168-202.

Halbach, V., and Horsten, L. [2005]: ‘Computational structuralism’, *Philosophia Mathematica* **13**, 174-186.

Hauck, J. [1973]: ‘Berechenbare reelle Funktionen’, *Zeitschrift für Mathematische Logik und Grundlangen der Mathematik* **19**, 121-140.

Hertling, P. [1999]: ‘A real number structure that is effectively categorical’, *Mathematical Logic Quarterly* **45**, 147-182.

Hintikka, J. [1962]: *Knowledge and Belief*. Ithaca: Cornell University Press.

Horsten, L. [2012]: 'Vom Zählen zu den Zahlen: on the relation between computation and arithmetical structuralism', *Philosophia Mathematica* **20**, 275-288.

Kaplan, D. [1969]: 'Quantifying In', in D. Davidson and J. Hintikka, eds, *Words and Objections*, pp. 206-242. Dordrecht: Reidel.

Kleene, S. [1936]: 'General recursive functions of natural numbers', *Mathematische Annalen* **112**, 727-742.

Kripke, S. [2011]: *Philosophical Troubles*. Oxford: Oxford University Press.

---. [2013]: 'The Church-Turing "thesis" as a special corollary.' In B. J. Copeland, C. Posy, and O. Shagrir, eds, *Computability: Turing, Gödel, Church, and Beyond*, pp. 77-104. Cambridge: MIT Press.

Lacombe, D. [1955]: 'Extension de la notion de fonction récursive aux fonctions d'une ou plusieurs variables réelles III', *Comptes Rendus Académie des Sciences Paris* **241**, 151-153.

Lewis, D. [1984]: 'Putnam's Paradox', *Australasian Journal of Philosophy* **62**, 221-236.

Montague, R. [1960]: 'Towards a general theory of computability', *Synthese* **12**, 429-438.

Parsons, C. [2008]: *Mathematical Thought and its Objects*. Cambridge: Cambridge University Press.

Peacocke, C. [1998]: 'The concept of a natural number', *Australasian Journal of Philosophy* **76**, 105-109.

Piccinini, G. [2008]: 'Computation without representation', *Philosophical Studies* **137**, 205-241.

---. [2009]: 'Computationalism in the philosophy of mind.' *Philosophy Compass* **4**, 512-532.

Putnam, H. [1979]: *Mathematics, Matter, and Method: Philosophical Papers, vol. 1*. Cambridge: Cambridge University Press.

---. [1980]: ‘Models and reality’, *The Journal of Symbolic Logic* **45**, 464-482.

---. [1981]: *Reason, Truth, and History*. Cambridge: Cambridge University Press.

Quine, W. V. [1966]: *The Ways of Paradox*. Cambridge: Harvard University Press.

---. [1981]: *Theories and Things*. Cambridge: Belknap Press of Harvard University Press.

Rescorla, M. [2007]: ‘Church’s thesis and the conceptual analysis of computability’, *Notre Dame Journal of Formal Logic* **48**, 253-280.

---. [2012]: ‘Copeland and Proudfoot on computability’, *Studies in History and Philosophy of Science A* **43**, 199-202.

---. [Forthcoming]: ‘From Ockham to Turing --- and back again’, in A. Bokulich and J. Floyd, eds, *Turing 100: Philosophical Explorations of the Legacy of Alan Turing*. Springer.

Rogers, H. [1987]: *The Theory of Recursive Functions and Effective Computability*. Cambridge: MIT Press.

Shapiro, S. [1982]: ‘Acceptable notation’, *Notre Dame Journal of Formal Logic* **23**, 14-20.

---. [2000]: ‘Effectiveness’, in eds. J. van Benthem, G. Heinzmann, M. Rebuschi, and H. Visser, eds, *The Age of Alternative Logics*, pp. 37-50. Berlin: Springer.

Sieg, W. 2009. ‘On computability’, in A. Irvine, ed, *Philosophy of Mathematics*, pp. 535-630. Burlington: Elsevier.

Skolem, T. [1922/1967]: ‘Some remarks on axiomatized set theory,’ trans. S. Bauer-Mengelberg, rpt. in J. van Heijenoort, ed, *From Frege to Gödel*, pp. 290-301. Cambridge: Harvard University Press.

Soare, R. [1987]: *Recursively Enumerable Sets and Degrees*. New York: Springer-Verlag.

---. [1996]: ‘Computability and recursion’, *Bulletin of Symbolic Logic* **2**, 284-321.

Turing, A. [1936]: ‘On computable numbers, with an application the

Entscheidungsproblem', *Proceedings of the London Mathematical Society* **42**, 230-265.

---. [1937]: 'On computable numbers, with an application to the Entscheidungsproblem: a correction.' *Proceedings of the London Mathematical Society* **43**, 544-546.

Weihrauch, K. [2000]: *Computable Analysis: An Introduction*. Berlin: Springer.