Successor, nontrivial functions and ordering. A study in computability and learnability in notations for natural numbers

Dariusz Kalociński & Michał Wrocławski

May 18, 2020

Abstract

The class of all computable functions and relations depends on the notation we choose for numbers. In this paper we consider the concept of a notation for natural numbers, as defined by Stewart Shapiro in [13], and prove certain formal results regarding it. In particular, we prove that computability of the ordering of numbers in a given notation does not imply computability of the successor—contrary to what Paul Benacerraf claimed in [1]. We generalise this result in various ways. We show that for every ordering isomorphic to the standard ordering of natural numbers and for every nontrivial (i.e., neither almost constant nor almost identity) unary number-theoretic function there exists a notation in which the ordering is computable but the function is not. This result strengthens one of the theorems proved by Shapiro in his paper. We further generalise this result to relations, other arities and uniformly computable families of functions or relations. Finally, we consider a weaker notion than computability, namely that of learnability, first investigated by Gold. We show that Benacerraf's claim is also false under such reinterpretation, except for a certain weak version.

Keywords computability, learnability, notations, natural numbers, successor, ordering

1 Introduction

In his paper [13], Stewart Shapiro considered the concept of a notation for natural numbers. According to Shapiro, a notation is a bijection from a recursive set of numerals to the set of natural numbers. He noted that computations are performed directly on numerals and only indirectly on numbers, and that the choice of a notation determines the class of functions that are computable. He proved a number of formal results regarding how the choice of a notation influences our ability to compute various functions. For instance, he noted that for every set of natural numbers there exists a notation in which its characteristic function is computable. Indeed, let A be any set of natural numbers. Then it suffices to take a notation in which the usual even numerals stand for numbers from A, and the usual odd numerals for numbers from outside A.

His other results are not as trivial. Shapiro proved that some functions are not computable in any notation, and that the only (unary) functions computable in every notation are almost constant and almost identity functions.

However, the key question discussed in Shapiro's paper was philosophical. According to Church-Turing thesis, a function is (intuitively) computable if and only if it is recursive (recursivity can be replaced by any of the equivalent models of computations). Shapiro noted that this thesis does not specify the notation used. By changing the notation, we can change which functions are computable, and which are not.

Shapiro believed that when we talk about a computable function, we mean a function computable in an acceptable notation. He tried to formulate a criterion which can be used to distinguish between acceptable and not acceptable notations. He argued that a notation is acceptable if and only if the successor is computable in it.

Various authors took part in the philosophical debate regarding this issue, most notably Michael Rescorla [10, 11], Brian Jack Copeland and Diane Proudfoot [4], as well as Paula Quinon [8]. Discussions about notations for numbers have been mainly philosophical, but there have also been some formal results. In his Ph.D. thesis [19] (see also [17, 18]), Michał Wrocławski considered the notion of a representation which generalises the concept of a notation in such a way that the function mapping numerals to natural numbers is only assumed to be surjective, not necessarily bijective (see: Definition 1). Some results were also obtained by Michał Tomasz Godziszewski and Joel David Hamkins in a somewhat related setting of quotient presentations of models of arithmetic [6].

Interesting computability results have also been proved by Konrad Zdanowski and presented at a conference 'Numbers and Truth' in 2012 but have not been published and are only available in the form of conference slides.

Philosophical arguments regarding notations have also heavily influenced discussions about interpretation of Tennenbaum's theorem [16]. Among the key participants of this debate are: Walter Dean [5], Paula Quinon and Konrad Zdanowski [9], Tim Button and Peter Smith [3].

But there is also another source of inspiration for our considerations here. In his famous paper [1], a structuralist philosopher Paul Benacerraf asked the question what natural numbers are. He noted that even inside the ZF set theory there are multiple ways of defining natural numbers, all of them equally valid, and he claimed that there is no way to say which of them is correct, i.e. which are the true natural numbers. What is more, he argued that this is not even a well-defined question. What matters is the mathematical structure, and talking about natural numbers „in themselves", without reference to their place within a structure, is meaningless. According to Benacerraf, any structure with a computable ω-type ordering can work as natural numbers. Later, in [2], he dropped the condition that the ordering ought to be computable and concluded that „any ω sequence would do after all".

Benacerraf uses the term „notation" multiple times throughout his paper, but he does not always clearly distinguish between natural numbers and a notation for them. However, philosophical subtleties aside, from a practical point of view the questions asked by Benacerraf and Shapiro seem to be very similar: which objects can be used as natural numbers in mathematics? Benacerraf (at least in his earlier paper) claimed that the key requirement for them is that their ordering should be computable. According to Shapiro, the successor function on them should be computable. One could ask, what is the relation between these two conditions. Benacerraf claimed that they are equivalent, but he did not provide any proof to back this statement. According to his words in [1]:

> This ability (to tell in a finite number of steps which of two numbers is greater) is connected with (both transitive and intransitive) counting, since its possibility is equivalent to the possibility of generating ("saying") the numbers in order of magnitude.

In this paper we wish to show, among other things, that this claim is false. Since Benacerraf did not develop any well-defined formal framework for his considerations, our proof is carried out within the framework proposed by Shapiro.

In Shapiro's terms, Benacerraf's claim can be formulated as follows: in every notation, computability of successor is equivalent to computability of the ordering. The left-to-right implication is an easy conclusion from Shapiro's considerations in [13], though it has not been explicitly stated there.[1] Our refutation concerns the implication in the opposite direction. What is important, this result turns out to be sufficient to disprove other plausible interpretations of the claim.

[1]However, note that if we allow a number to be represented by more than one numeral, this implication does not always hold—see [17] and [18].

We refute Benacerraf's claim using two different proofs. The first proof is shorter and easier, but the second one introduces a technique that allows us to go far beyond the successor function and obtain various generalizations, some of which are stronger versions of Shapiro's results. Finally, we consider several variants of Bernacerraf's claim in which the notion of learnability of functions is used instead of that of their recursiveness.

The paper is structured as follows. Section 2 introduces necessary notions from computability theory. Section 3 is devoted to the concept of notations as well as to plausible disagreements about it between Shapiro and Benacerraf. Section 4 is the technical part of the paper containing all the main results, spread throughout three subsections. In Section 4.1 we refute Benacerraf's claim. The following subsections provide various generalizations. In Section 4.2 we prove a theorem on nontrivial functions. Generalizations to ω-type orderings, collections of functions and functions of higher arities are provided in Section 4.3. In Section 4.4 the notion of learnability in notations is applied to Benacerraf's problem. Section 5 concludes the paper.

2 Computability-theoretic notions

Henceforth, $\mathbb{N}$ denotes the set of natural numbers. Numbers are referred to using lowercase Latin characters $a, b, \ldots, i, j, k, \ldots$, possibly with subscripts. Σ will denote a finite alphabet, and Σ^* the set of all words over Σ (i.e., finite sequences of elements of Σ). Lowercase Greek letters $\alpha, \beta, \ldots$, possibly with subscripts, usually refer to words. Sometimes we use uppercase Latin characters $A, B, C, \ldots$ to denote finite sequences (lists) of elements of a given set. If $A = (\alpha_i)_{i=1}^n, B = (\beta_i)_{i=1}^m$ are lists and α is an element, then $AB = (\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m)$ and $A\alpha = (\alpha_1, \ldots, \alpha_n, \alpha)$. Let E be a set. A characteristic function of $R \subseteq E^k$, denoted by χ_R, maps E onto $\{0, 1\}$ and is defined by $\chi_R(x_1, \ldots, x_k) = 1 \Leftrightarrow (x_1, \ldots, x_k) \in R$, for all $x_1, \ldots, x_k \in E$. We shall sometimes confuse a relation R with its characteristic function χ_R which makes writing $R(x_1, \ldots, x_k) = 1$ or $R(x_1, \ldots, x_k) = 0$ meaningful.

We fix a canonical enumeration $\psi_0, \psi_1, \ldots$ of all partial recursive functions from $\mathbb{N}$ to $\mathbb{N}$. We can assume that each e stands for a program P_e (in a Turing-complete programming language of your choice) computing a partial function ψ_e. A set $A \subseteq \mathbb{N}$ is recursive (decidable, computable) if there exists e such that $\psi_e = \chi_A$. A set $A \subseteq \mathbb{N}$ is computably enumerable (c.e.) if A is the domain of some partial recursive ψ_e. Equivalently, $A \subseteq \mathbb{N}$ is c.e. if $A = \emptyset$

or there exists a (total) recursive function ψ such that $A = \{\psi(n) : n \in \mathbb{N}\}$. Such a recursive function ψ exists for each nonempty c.e. set and is usually referred to as a recursive (or computable) enumeration of A (since one can effectively enumerate values $\psi(0), \psi(1), \dots$ which comprise the whole A). Existence of c.e. uncomputable sets is a folklore. Note that each infinite c.e. set A can be effectively enumerated without repetitions, i.e. there exists an injective recursive enumeration of A. Given a partial recursive function ψ_e and $t \in \mathbb{N}$, we write $\psi_{e,t}$ to denote the partial recursive function of x resulting from running P_e on x for at most t steps; if P_e happens to stop on x in $\leq t$ steps, then we write $\psi_{e,t}(x)\downarrow$ and the value of $\psi_{e,t}$ on x is $\psi_e(x)$, otherwise we write $\psi_{e,t}(x)\uparrow$ and the value of $\psi_{e,t}$ on x is considered undefined.

We say a function $f : \mathbb{N}^k \to \mathbb{N}$ is learnable if there exists a uniformly computable family of computable functions $\{f_t\}_{t\in\mathbb{N}}$ such that for every $n_1, \dots, n_k \in \mathbb{N}$:

$$f(n_1, \dots, n_k) = \lim_{t\to\infty} f_t(n_1, \dots, n_k). \tag{1}$$

We say a relation $R \subseteq \mathbb{N}^k$ is learnable if there exists a uniformly computable family of relations $\{R_t\}_{t\in\mathbb{N}}$ such that, for all $n_1, \dots, n_k \in \mathbb{N}$:

$$R(n_1, \dots, n_k) = \lim_{t\to\infty} R_t(n_1, \dots, n_k). \tag{2}$$

That is, a function is learnable if, for each given input, one can successively make effective guesses regarding its values in a way that the guesses finally settle down on the correct value. The notion of learnability is due to Gold [7].

An oracle program is a program which may use a special instruction to ask whether an element belongs to a given set, called the oracle. In a Turing machine formalism, this might be realised by introducing a one-way infinite read-only tape filled with zeros and ones, serving as a characteristic function of the oracle. If e is an oracle program and A is a set of natural numbers, ψ_e^A denotes a partial function computed by e with A supplied as the oracle. We say a partial function $f : \mathbb{N} \to \mathbb{N}$ is computable (recursive) in A, in symbols $f \leq A$, if there exists an oracle program e such that $f = \psi_e^A$. It is customary to start with oracle programs and to identify a canonical enumeration of all partial recursive functions $\psi_0, \psi_1, \dots$ with $\psi_0^\emptyset, \psi_1^\emptyset, \dots$. We say A is computable in B, in symbols $A \leq B$, if $A = \psi_e^B$, for some oracle program e. We say A is Turing equivalent to B, in symbols $A \equiv B$, if $A \leq B$ and $B \leq A$. Let $K = \{e : \psi_e(e)\downarrow\}$ be the halting problem. It is customary to write $\leq 0'$ instead $\leq K$ where $0'$ is the Turing degree of K, i.e. $0' = \{A \subseteq \mathbb{N} : A \equiv K\}$. The elements of $0'$ are Turing equivalent so it does not matter which one is supplied as the oracle.

The following classical result provides an important link between learnability and Turing reducibility.

Theorem 1 (Limit Lemma [14]) *Let $f : \mathbb{N} \to \mathbb{N}$ be a total function. f is learnable if and only if $f \leq 0'$. Similarly, a set $A \subseteq \mathbb{N}$ is learnable if and only if $A \leq 0'$.*

By means of coding, the above notions (of a partial recursive function, decidability, learnability, computable enumerability, recursive enumeration) can be extended to any finite arity and applied to other domains, in particular to a recursive $S \subseteq \Sigma^*$. For more information on computability theory, we refer the reader to textbooks [12, 15].

3 Notations for natural numbers

In the subsequent definitions we formalize the concept of a notation for natural numbers and of computability and learnability in a notation. Our exposition follows Shapiro [13], except that learnability has not been previously considered in this context. Along the way we highlight plausible differences in the usage of the term "notation" between Shapiro and Benacerraf.

Definition 1 (Shapiro [13]) *Let Σ be a finite alphabet. We shall call (S, σ) a notation for $\mathbb{N}$ if $S \subseteq \Sigma^*$ is computable and $\sigma : S \to \mathbb{N}$ is a bijection.*

Elements of S are referred to as numerals. We shall abbreviate „notation for $\mathbb{N}$" to „notation" since we do not consider notations for other domains. By the standard notation and standard numerals of $\mathbb{N}$ we refer to the usual decimal notation and its numerals, respectively. When we refer to numerals rather than numbers, we put bars over them: $\overline{n}$ is the standard decimal numeral for the number n. However, in a non-standard notation it may represent a different number, or not be a valid numeral at all.

The requirement of computability of the set of numerals is quite natural: we want to be able to mechanically distinguish numerals, i.e. strings which refer to some numbers, from strings which do not. Note that this constitutes a difference between Shapiro's and Benacerraf's approach, as the latter does not postulate such a requirement [1, 2]. When referring to numerals, Benacerraf often mentions that one should be able to effectively generate names for numbers in the order of their magnitude, but—even if we do not treat it as a part of his definition of a notation—this does not entail recursiveness of the set of all numerals but only its recursive enumerability.[2]

It is also worth noting that while for Shapiro numerals refer to the usual natural numbers, from Benacerraf's point of view, rooted in structuralist philosophy of mathematics, the exact nature of these objects is irrelevant. One could even identify notations in Benacerraf's paper with numerals without any reference to any other type of objects [1, pp. 71-72]. This, however, does not in any way undermine the validity of formal arguments contained in this article.

Let us proceed to the definitions of computability of functions and relations in notations for natural numbers.

Definition 2 (Shapiro [13]) *Let (S, σ) be a notation and let $f : \mathbb{N}^n \to \mathbb{N}$. $f^\sigma : S^n \to S$ shall denote such a function that for all $\alpha_1, ..., \alpha_n, \beta \in S$, the following condition is satisfied:*

$$f^\sigma(\alpha_1, ..., \alpha_n) = \beta \Leftrightarrow f(\sigma(\alpha_1), ..., \sigma(\alpha_n)) = \sigma(\beta).$$

We say that f is computable in (S, σ), if f^σ is computable.

Definition 3 *Let $\sigma : S \to \mathbb{N}$ be a bijection and let $R \subseteq \mathbb{N}^n$. Then $R^\sigma \subseteq S^n$ shall be defined in the following way:*

$$(\alpha_1, ..., \alpha_n) \in R^\sigma \Leftrightarrow (\sigma(\alpha_1), ..., \sigma(\alpha_n)) \in R,$$

for all $\alpha_1, ..., \alpha_n \in S$. We say that R is computable in (S, σ) if R^σ is computable.

For Shapiro, computability of a number-theoretic function (or relation) in a notation is understood as the existence of a program which acts on numerals

[2]This idea is connected to his notion of intransitive counting which is the process of generating numerals in the order of magnitude. Recall, for example the following fragment: "What is certain, and not immaterial, is that we will have to learn some recursive procedure for generating the *notation* in the proper order before we have learned to count transitively (...)" [1, p. 50]. Or this one: "If what we are generating is a notation, the most natural way for generating it is by giving recursive rules for getting the next element from any element you may have (...)" [1, p. 73].

and outputs numerals (or truth values, which might be identified with numerals denoting 0 and 1) such that the underlying referents agree with the function (relation) being computed. This way of understanding computability is also implicitly present in Benacerraf's considerations (see, e.g., [1, pp. 72-73]).

However, there is also another way in which computability of a function or relation is understood in Benacerraf's thought (see, e.g., [1, p. 51]). It concerns merely the elements of an ω-sequence which the numerals are mapped to. As mentioned earlier, this ω-sequence need not be identical to $\mathbb{N}$ with the usual order.

Notice that the computability of $<$ in a notation and the computability of the ordering between elements of the underlying ω-sequence are two different and nonequivalent notions. To make it more explicit, we would like to present an argument based on [2, p. 187–188]. Consider a sequence $a_0, a_1, a_2, \ldots$ of natural numbers such that the relation

$$n < m \Leftrightarrow \exists_{i,j}(a_i = n \wedge a_j = m \wedge i < j)$$

is nonrecursive. Now, map each standard numeral $\overline{n}$ to a_n. Clearly, $<$ is computable in this notation because given any two decimal numerals $\overline{n}, \overline{m}$ we have $a_n < a_m \Leftrightarrow n < m$. But the ordering $<$ is nonrecursive by definition.

It is worth emphasizing that in the above argument the notion of a notation is used in a loose sense. While throughout this article we use it to refer to notations of the whole set $\mathbb{N}$, this time we are considering a notation for a certain nonrecursive subset of $\mathbb{N}$.

Finally, we introduce the notion of learnability in notations. It will be applied later, in Section 4.4.

Definition 4 *Let (S, σ) be a notation. We say a function $f : \mathbb{N} \to \mathbb{N}$ is learnable in (S, σ) if f^σ is learnable. Similarly, a relation $R \subseteq \mathbb{N}^k$ is learnable in (S, σ) if R^σ is learnable.*

Let us slightly unpack this definition. Learnability in a notation makes use of the ordinary notion of learnability. f's learnability in (S, σ) means that f^σ is learnable, i.e. that, by Definition 1, there exists a uniformly computable family $\{g_t\}_{t\in\mathbb{N}}$ of functions $g_t : S \to S$ such that for each $\alpha \in S$:

$$f^\sigma(\alpha) = \lim_{t\to\infty} g_t(\alpha). \tag{3}$$

In other words, f is learnable in (S, σ) if there exists a family of number-theoretic functions $\{f_t\}_{t \in \mathbb{N}}$, uniformly computable in (S, σ), such that for each $\alpha \in S$:

$$f^{\sigma}(\alpha) = \lim_{t \to \infty} f_t^{\sigma}(\alpha). \tag{4}$$

Analogous remarks apply to learnability of relations.

4 Results

4.1 Successor

The following result refutes Benacerraf's claim concerning the equivalence between computability of ordering and successor.

Theorem 2 *There exists a notation (S, σ) in which the ordering $<$ is computable, but the successor function is not computable.*

Proof.

This proof is carried out in two parts. First, we a construct a notation in which the set of numerals is c.e., but not computable (note a harmless misuse of terminology—a notation should have a computable set of numerals). Second, we show how to modify the notation to get a computable set of numerals, while at the same time preserving computability (or the lack thereof) of all the functions and relations for the original notation.

Let $A \subseteq \mathbb{N}$ be uncomputable and c.e. (in the standard notation). Let

$$T = \{(\overline{a}, \overline{0}) : a \in \mathbb{N}\} \cup \{(\overline{a}, \overline{1}) : a \in A\}.$$

Note that T is uncomputable and c.e. as well. Now, let τ assign numbers to numerals in a bijective way according to the following ordering:

$$\tau((\overline{a_1}, \overline{b_1})) < \tau((\overline{a_2}, \overline{b_2})) \Leftrightarrow a_1 < a_2 \lor (a_1 = a_2 \land b_1 < b_2).$$

We have constructed (T, τ) which satisfies all the requirements from the definition of a notation except one that T should be computable.

Observe that the ordering is computable in (T, τ): given $(\overline{a}_1, \overline{b}_1), (\overline{a}_2, \overline{b}_2) \in T$, simply compute the truth value of the right hand side of the above equivalence.

We want to show that the successor is not computable in (T, τ). Suppose to the contrary that it is. But then A turns out to be computable. To check whether $n \in A$, we just need to compute the successor of $(\overline{n}, \overline{0})$ in (T, τ). If it is $(\overline{n}, \overline{1})$, then it means that $n \in A$, otherwise $n \notin A$. This is a contradiction, hence the successor is not computable in (T, τ).

We are now going to convert (T, τ) into (S, σ) which is a valid notation. Let S be the standard set of decimal numerals. Let $(\alpha_n)_{n \in \mathbb{N}}$ be a computable enumeration of T without repetitions. Let σ be such that for every natural number n: $\sigma(\overline{n}) = \tau(\alpha_n)$. Note that S is trivially computable and that the class of all functions and relations computable in (S, σ) is exactly the same as in (T, τ). This is because if a function is computable in any of these notations, then to compute it in the other, we need to take any input in the latter notation, translate it to the former, then compute the value of the function and translate the result in the other direction.

Hence, in particular, for every $i, j \in \mathbb{N}$:

$$\sigma(\bar{i}) < \sigma(\bar{j}) \Leftrightarrow \tau(\alpha_i) < \tau(\alpha_j)$$

and

$$Succ(\sigma(\bar{i})) = Succ(\tau(\alpha_i)).$$

It follows that the ordering is computable in (S, σ), but successor is not—just like in (T, τ).

■

We now want to present another proof of Theorem 2. Our technique will be later generalised to show stronger results. The construction is a bit involved, but it is worth the effort.

Theorem 2 *There exists a notation (S, σ) in which the ordering $<$ is computable, but the successor function is not computable.*

Proof. We start with outlining the overall idea which organises the whole proof. We are going to construct an infinite sequence of numerals $(\alpha_i)_{i\in\mathbb{N}}$, without repetitions. The denotation of α_i, for $i \in \mathbb{N}$, will be determined by its position in the sequence, i.e. $\sigma(\alpha_i) = i$. This will ensure that $<$ is computable. The main challenge is that the construction of the sequence should guarantee that no program computes the successor. Since each program computing some unary partial function from numerals to numerals is a potential candidate for computing the successor function, we need to ensure that for each such a program e, the following requirement is satisfied:

$$\exists_\alpha[\psi_e(\alpha)\downarrow \implies \sigma(\alpha) + 1 \neq \sigma(\psi_e(\alpha))]. \qquad (\mathcal{R}_e)$$

This will be done by assigning to each program e a distinct numeral α and waiting for it to stop (with α as its input) in order to perform appropriate manipulations on the sequence to ensure that it satisfies the consequent of the above implication. The details are included below.

Construction Let $\varsigma_0, \varsigma_1, \ldots$ be a computable enumeration, without repetitions, of an infinite computable set S of numerals over a finite alphabet. Let $\psi_0, \psi_1, \ldots$ be a canonical enumeration of all partial recursive functions from S to S.

Stage 0. Let $A_0 = (\varsigma_0)$ and $L_0 = \{(0, \varsigma_0)\}$.

Stage $n+1$. We have A_n, L_n already defined, where A_n is list of numerals and L_n is a finite function that maps programs to numerals from A_n. We say that e requires attention at stage $n+1$ if, for some α, $(e, \alpha) \in L_n$ and $\psi_{e,n}(\alpha)\downarrow$. There are two possible cases:

(i) There exists a program e that requires attention at stage $n+1$.

(ii) No program requires attention at stage $n+1$.

Let us begin with the case (i). We must make sure that we take appropriate actions for each program e that requires our attention at stage $n+1$. Note that, given such a program e, we have $\psi_e(\alpha)\downarrow = \beta$, for some β. We do not want e to compute the successor in our notation, so we have to satisfy the condition $\sigma(\alpha) + 1 \neq \sigma(\beta)$. We do not know the ultimate positions of α and β (i.e. $\sigma(\alpha)$ and $\sigma(\beta)$), so the only thing we can do now, i.e. at stage $n+1$, is to make sure that β does not come directly after α in A_{n+1}. This justifies the following definition. We say that the requirement $\mathcal{R}_e$ is satisfied at stage t, if whenever e has required attention at a stage $s+1 \leq t$, with $\psi_{e,s}(\alpha)\downarrow = \beta$, then β does not come directly after α in A_t. Below we describe actions that are performed for each e that requires attention at stage $n+1$. These actions lead to the satisfaction of $\mathcal{R}_e$ at stage $n+1$.

Let $A := A_n$, $L := L_n$. A and L are temporary variables for computing A_{n+1} and L_{n+1}, respectively. For each program e that requires attention at stage $n+1$ we perform all the steps described below. Let β be such that $\psi_{e,n}(\alpha) = \beta$. Let $\beta' = \varsigma_t$, where t is the least index not utilised in $A \cup \{\beta\}$.

(1) Suppose β is already in A and it appears on this list directly behind α, i.e. we have $A = B\alpha\beta C$, for some lists B, C. We set $A := B\alpha\beta'\beta C$.

(2) Suppose β is already in A but it does not appear on this list directly behind α. One possibility is that we have $A = B\beta C\alpha D$ for some lists B, C, D. Second possibility is that $A = B\alpha C\beta D$, for some lists B, C, D, where C is nonempty. In either case, A remains unchanged.

(3) Suppose β does not appear in A and α is the last element there, i.e. we have $A = B\alpha$, for some list B. We set $A := A\beta'\beta$ which yields $A = B\alpha\beta'\beta$.

(4) Suppose β does not appear in A and α is not the last element there, i.e. we have $A = B\alpha C$, for some lists B, C, where C is nonempty. We set $A := A\beta$ which yields $A = B\alpha C\beta$, with C being nonempty.

Finally, set $L := L - \{(e, \alpha)\}$.

After performing the above steps for each e that requires attention at stage $n + 1$, we set $A_{n+1} = A$, $L_{n+1} = L$.

Now, let us consider the case (ii), i.e. when no program requires attention at stage $n+1$. Let $\alpha = \varsigma_t$ where t is the least index such that ς_t is not utilised in A_n and let e be the least program such that e has not appeared in L_k, for any $k \leq n$. Set $A_{n+1} = A_n\alpha$ and $L_{n+1} = L_n \cup \{(e, \alpha)\}$.

Thus we have constructed A_{n+1} and L_{n+1}. This ends the construction.

It remains to define the notation (S, σ) that results from the above construction. S has been fixed at the beginning of the proof. Let $\{A_n\}_{n\in\mathbb{N}}$ be a family of lists over a set E, i.e., for each n, $A_n \in \bigcup_{k=0}^{\infty} E^k$. Let $i \in \mathbb{N}$ and $e \in E$. We say that $\lim_{n\to\infty} A_n(i) = e$ if $\exists_{n_0}\forall_{n>n_0}$ $(|A_n| > i \wedge A_n(i) = e)$. If for some e we have $\lim_{n\to\infty} A_n(i) = e$, then the unique e for which this statements holds is denoted by $\lim_{n\to\infty} A_n(i)$, otherwise $\lim_{n\to\infty} A_n(i)$ is considered as undefined. Having this convention in mind, let $\{A_n\}_{n\in\mathbb{N}}$ be the family of lists over a set S obtained in the above construction. We define $\sigma : S \to \mathbb{N}$ as follows. For each $i \in \mathbb{N}$, let

$$g(i) = \lim_{n\to\infty} A_n(i). \tag{5}$$

We set $\sigma = g^{-1}$.

Verification

Lemma 1

(a) Each $\alpha \in S$ enters A_n at some stage n.

(b) For each e there exists α such that (e, α) enters L_n at some stage n.

Proof. Suppose this is not the case. Let $\alpha = \varsigma_t$, where t is the least such that ς_t does not enter A_n at any stage n. Let s be the least stage such that for every $i < t$, ς_i enters A_n at some stage $n \leq s$. Observe that if α never enters the lists $A_0, A_1, \ldots$, then the condition (ii) is not satisfied at any stage $n + 1 \geq s$ (for if (ii) was satisfied at some stage $n + 1 \geq s$, then, by the construction, α would enter A_{n+1}). Hence, for every A_{n+1} with $n + 1 \geq s$, we always have (i) satisfied, i.e. some program e requires attention at stage $n + 1$. However, actions performed in response to (i) at stage $n + 1$, always lead to a shrinkage of L_n, i.e. we have $L_n \supset L_{n+1}$. Since, L_s is finite, there must be a number k such that $L_{s+k} = \emptyset$. For the least such k, the condition (i) does not hold at stage $s + k + 1$ but (ii) holds, so—as prescribed by the construction—α enters A_{s+k+1} which is a contradiction. This proves Lemma 1 (a).

The proof of Lemma 1(b) is similar. Suppose the contrary, choose the least e such that e never enters any L_n, and the least s such that for every $e' < e$, e' enters L_n at some stage $\leq n$. Then continue the proof with e instead α and L_n instead A_n. ∎

Lemma 2 *For each $i \in \mathbb{N}$, $\lim_{n\to\infty} A_n(i)$ is defined.*

Proof. We proceed by induction on i. Let $i \in \mathbb{N}$ be such that, for all $j < i$, $\lim_{n\to\infty} A_n(j)$ is defined. Let s be a stage such that for all subsequent stages $t \geq s$ we have $A_t(j) = \lim_{n\to\infty} A_n(j)$, for $j < i$. We want to show that the numeral at position j in the lists $A_0, A_1, \ldots$ finally settles down.

It might be the case that $|A_s| = i$, so there is nothing in A_s at position i. However, by Lemma 1 and the observation that the lists A_n never get shorter as we proceed with n, their lengths finally become $> i$. Hence, without loss of generality, we may assume that $|A_s| > i$.

Observe that the numeral at position i can change at a stage $t + 1 \geq s$ only because of the action performed in response to (i) with $A_t = A\alpha\beta B$ such that $A_t(i) = \beta$. This is because (2) does not affect the lists A_n while (3) and (4) only extend the existing list so that the position i is not affected. On the other hand, there can be no actions in response to (1) with $A_t = A\alpha\beta C$ and $A_t(j) = \beta$, for $j < i$, because this would violate the inductive assumption—something new would have been inserted at position j.

One possibility is that the numeral at position i never changes at stages $t + 1 \geq s$. If this is so, the proof is complete. Otherwise, at some stage $t+1 \geq s$, we perform an action in response to (1): we have $A_t = A\alpha\beta C$ with

$A_t(i) = \beta$ and thus $A_{t+1}(i) = \beta'$. Now it remains to observe that β' is the final value at position i. Suppose it is not a final value. Then again some β'' is inserted at position i in response to (1) at a later stage $u+1$. But this implies that throughout the construction there are at least two programs, say e and e', that are attached to α at some stages $\leq t$ and $\leq u$, respectively (i.e. both (e, α) and (e', α) enter the sets $L_0, L_1, \dots$), which then require attention at stages $t+1$ and $u+1$, respectively, and lead to two changes at position i that we have just described. However, this is not possible, since, by the case (ii), for each program the construction picks out a fresh numeral, so two programs cannot be attached to the same α. Therefore, β' is the final value at position i.

■

Lemma 3 (S, σ) *is a notation.*

Proof. The set of numerals S is computable by definition. It remains to show that $\sigma : S \to \mathbb{N}$ is a bijection. Recall that $\sigma = g^{-1}$, where $g(i) = \lim_{n\to\infty} A_n(i)$, for all $i \in \mathbb{N}$. By Lemma 2, g is a function. By Lemma 1(a), g is onto. It is easy to see that there are no repetitions in the lists A_n: whenever an element is added to A_n, it is always a fresh one. Therefore g is one-one. Hence, g is a bijection, and thus $\sigma = g^{-1}$ is a bijection as well. ■

Lemma 4 $<$ *is computable in* (S, σ).

Proof. Observe that the construction is recursive. The program for computing $<$ in (S, σ) works as follows. Given $\alpha, \beta \in S$ as input, proceed with the construction until you find out a stage n such that both α and β appear on the list A_n (which must happen by Lemma 1(a)) and then check which of them comes earlier.

To prove that the above program computes $<$ in (S, σ), it remains show that once two numerals appear on the lists A_n, their order is never reversed. To see this, note that all modifications to the lists A_n are performed in response to conditions (1), (2), (3) and (4). These modifications are of two types: extension (new elements are added at the end of the list) and insertion (a new element is added between the elements of the current list). Neither of them changes the order between numerals in the lists. The conclusion follows.

■

Lemma 5 *The successor is not computable in* (S, σ).

Proof. It suffices to show that $\mathcal{R}_e$ holds for every program e.

First, we show that every requirement $\mathcal{R}_e$ is satisfied at every stage. Let e be any program. By Lemma 1(b), let s be the stage at which (e, α) enters L_s. There are two cases: $\psi_e(\alpha) \downarrow$ or $\psi_e(\alpha) \uparrow$. If $\psi_e(\alpha) \uparrow$ then e never requires attention and thus $\mathcal{R}_e$ is always satisfied. Suppose that $\psi_e(\alpha) \downarrow = \beta$. Then for some $n + 1 \geq s$, e requires attention at stage $n + 1$. The construction guarantees that $\mathcal{R}_e$ is satisfied at stage $n + 1$, i.e. β does not come directly after α in A_{n+1}. It suffices to observe that β does not come directly after α at any subsequent stage because elements in the lists A_n are never reversed nor withdrawn.

Now we show that $\mathcal{R}_e$ holds, for each e. Let e be a program. By Lemma 1(b), choose α such that (e, α) enters some L_n. Suppose $\psi_e(\alpha) \downarrow = \beta$. By Lemma 3, let $\sigma(\alpha) = i$, $\sigma(\beta) = j$. Let s be such that for all $t \geq s$, $A_t(i) = \alpha$, $A_t(j) = \beta$ (by Lemma 2). Since β does not come directly after α in the lists A_t, for $t \geq s$, we have $i + 1 \neq j$. ■

■

4.2 Nontrivial unary functions

A theorem similar to Theorem 2 can be obtained for a wide class of number-theoretic functions, with the successor being just one of its instances. We say that a function $f : \mathbb{N} \to \mathbb{N}$ is almost constant if there exists $y \in \mathbb{N}$ such that $f(x) = y$ holds for all but finitely many $x \in \mathbb{N}$. Similarly, a function $f : \mathbb{N}^n \to \mathbb{N}$ is said to be almost identity if $f(x) = x$ holds for all but finitely many tuples $x \in \mathbb{N}$. A unary function is said to be nontrivial if it is neither almost constant nor almost identity. Clearly, the successor function is nontrivial.

Theorem 3 is a generalisation of Theorem 2 to nontrivial functions. The generalization is not obvious—unlike the previous proof, an attempt to satisfy one requirement can spoil our previous attempts of satisfying some other requirements. We need to find the way out of this. As we shall see, the solution bears some resemblance to the finite injury priority method.

On a side note, this result also strengthens one of Shapiro's theorems according to which a function is computable in any notation if and only if this function is either almost constant or almost identity. It follows from our result that not only is every such function computable in a certain notation, but it can be also assumed that it is computable in a certain notation in which ordering is also computable.

Theorem 3 *For every nontrivial $f : \mathbb{N} \to \mathbb{N}$ there exists a notation (S, σ) such that the ordering $<$ is computable in (S, σ), but f is not.*

Proof. If f is not computable in the standard notation, then this theorem is trivially true. We only need to show that it is true otherwise.

Let $f : \mathbb{N} \to \mathbb{N}$ be a computable nontrivial function.

The overall idea is the same as in the second proof of Theorem 2, with f in place of the successor. We start by outlining the main differences. Again, let $\varsigma_0, \varsigma_1, \dots$ be a computable enumeration, without repetitions, of an infinite computable set S. Let $\psi_0, \psi_1, \dots$ be a canonical enumeration of all partial recursive functions from S to S. For each program e we have the requirement

$$\exists_\alpha[\psi_e(\alpha)\downarrow \implies f^\sigma(\alpha) \neq \psi_e(\alpha)]. \tag{$\mathcal{R}_e$}$$

At each stage n we define three finite objects A_n, L_n and C_n. A_n and L_n are understood in the same way as previously. C_n is a new ingredient. When e requires attention at stage $n+1$ while being assigned to α, (e, α) is withdrawn from L_n, as previously, but we put $f^\sigma(\alpha) \neq \beta$ into C_n. If $f^\sigma(\alpha) \neq \beta \in C_n$, we say that $f^\sigma(\alpha) \neq \beta$ is active at stage n. We say that a condition $f^\sigma(\alpha) \neq \beta$ is satisfied at stage n, if there exist i, j such that $A_n(i) = \alpha$, $A_n(j) = \beta$ and $f(i) \neq j$. Now, we say the requirement $\mathcal{R}_e$ is satisfied at stage n, if whenever e has required attention at a stage $k+1 \leq n$, with $\psi_{e,k}(\alpha)\downarrow= \beta$, then the condition $f^\sigma(\alpha) \neq \beta$ is satisfied at stage n.

The idea about how to satisfy $\mathcal{R}_e$ is as follows. Once we know that $\psi_e(\alpha)\downarrow= \beta$, we wish to modify the notation under construction so that $f(\sigma(\alpha)) \neq \sigma(\beta)$. Since the denotation of a numeral is identified with its (ultimate) position in the lists A_n, we have to move α and β to positions i, j such that $f(i) \neq j$.

Note that once a requirement has been satisfied at some stage, our future attempts to satisfy some other requirements can destroy our previous work. Suppose that we have satisfied a requirement $\mathcal{R}_e$ by moving α and β to the desired positions i, j such that $f(i) \neq j$. Suppose also that later a different requirement needs attention which leads to relocation of numerals that come *before* α, β in the lists A_n. Then, clearly, the positions of α, β change to some other positions i', j' for which we might have $f(i') = j'$.[3]

[3]Note this is not an issue for the successor function: once we assure that β does not come directly after α, inserting elements before α or β cannot change this. This is why no preventive measures are necessary in the proof of Theorem 2 but are necessary here, where f is an arbitrary nontrivial function.

The above reasoning shows that once a condition $f^\sigma(\alpha) \neq \beta$ becomes active it should remain active later on. Therefore, as the construction proceeds, we have more and more active conditions. Crucially, we want them all to be ultimately satisfied. As we will see in the main construction and its verification, there is a way to satisfy all active conditions at any given stage.

Construction Let $\varsigma_0, \varsigma_1, \ldots$ be a recursive enumeration (without repetitions) of a computable set S. Let $\psi_0, \psi_1, \ldots$ be a canonical enumeration of all partial recursive functions from S to S.

Stage 0. Let $A_0 = (\varsigma_0)$, $L_0 = \{(0, \varsigma_0)\}$, $C_0 = \emptyset$.

Stage $n+1$. We have A_n, L_n, C_n already defined, where A_n is list of numerals, L_n is a finite function that maps programs to numerals from A_n, and C_n is a finite set of conditions of the form $f^\sigma(\alpha) \neq \beta$.

As previously, there are two possible cases:

(i) Some program e requires attention at stage $n+1$, i.e. there exists e, α such that $(e, \alpha) \in L_n$ and $\psi_{e,n}(\alpha)\downarrow$.

(ii) No program requires attention at stage $n+1$.

Let us first consider the case (i). Let $L := L_n$, and $C := C_n$ be temporary variables. For each e that requires attention at stage $n+1$ with $(e, \alpha) \in L_n$ and $\beta = \psi_{e,n}(\alpha)$, make $f^\sigma(\alpha) \neq \beta$ an active condition (i.e. put it into C). After that, remove each such (e, α) from L. Set $L_{n+1} = L$ and $C_{n+1} = C$.

We are going to ensure that all active conditions, i.e. those from C_{n+1}, are satisfied at stage $n+1$. Let $|A_n| = k$. For each position $p = 0, 1, \ldots, k-1$, for each active condition $f^\sigma(\alpha) \neq \beta \in C_{n+1}$, we say that $f^\sigma(\alpha) \neq \beta$ requires action at position p if $\max(i, j) = p$ and $f(i) = j$, where i, j are the positions of α, β.

Let $A := A_n$ be a temporary variable. We loop over the positions $p = 0, 1, \ldots, k-1$ and for each value of p we perform the actions described below. We emphasise that all modifications are made to a list stored in the temporary variable A.

If no condition requires action at position p, we do nothing. If precisely one condition requires action at position p we do the following.

(1) If $\alpha = \beta$ and $A(i) = \alpha$, then we need to find $i' \geq i$ such that $f(i') \neq i'$. Then move α to such position and fill all positions from i to $i'-1$ with fresh numerals.

(2) If $A(i) = \alpha$, $A(j) = \beta$ and $i < j$ then add fresh numerals directly in front of β to push β to a position $j' > j$ that $f(i) \neq j'$.

(3) If $A(i) = \alpha$, $A(j) = \beta$ and $i > j$, then add fresh numerals directly in front of α to push α to a position $i' > i$ such that $f(i') \neq j$.

(4) If $A(i) = \alpha$ and β does not appear in A, then put β on a position $j > i$ that $f(i) \neq j$ and fill all empty positions between α and β with fresh numerals.

Now, suppose that more than one condition requires action at position p. Let $f^\sigma(\alpha) \neq \beta$ be such a condition. There are four possibilities:

(a) α, β are different and appear in A at distinct positions i, j with $i < j = p$.

(b) α, β are different and appear in A at distinct positions i, j with $p = j < i$.

(c) α, β are identical and appear in A at position p.

(d) α, β are distinct, α appears in A at position p and β does not appear in A.

Observe that there can be at most one condition satisfying (b), (c) or (d) and requiring action at position p (because L_n is one-one and there are no repetitions in A). Other such conditions, must be of type (a). However, there can be many conditions satisfying (a) (at most p). If we have a condition of type (d), then we first satisfy all other conditions and (d) afterwards by applying (4). Conditions of type (a) and at most one condition of type (b) or (c) are satisfied all at once by moving the numeral at position p to a new distant position p', with $p, p+1, \ldots, p'-1$ filled with fresh numerals.

Once the whole A_n has been scanned and the above actions have been performed where necessary, we set $A_{n+1} = A$.

Finally, let us consider the case (ii). Let $\alpha = \varsigma_t$ where t is the least index such that ς_t is not utilised in A_n and let e be the least program such that e has not appeared in L_k, for any $k \leq n$. Set $A_{n+1} = A_n\alpha$, $L_{n+1} = L_n \cup \{(e, \alpha)\}$ and $C_{n+1} = C_n$. This ends the construction.

As previously, we define $\sigma = g^{-1}$ where g is defined by (5).

Verification The proof of Lemma 1 remains unchanged.

Lemma 2 *For each $i \in \mathbb{N}$, $\lim_{n\to\infty} A_n(i)$ is defined.*

Proof. Here, some modifications are necessary. Again, we proceed by induction. Fix i and assume that for all $j < i$, $\lim_{n\to\infty} A_n(j)$ is defined. Let s be a stage such that $A_t(j) = \lim_{n\to\infty} A_n(j)$, for $j < i$ and all $t \geq s$. Without loss of generality, we may assume that $|A_s| > i$.

If the numeral at position i never changes at stages $t \geq s$, the proof is complete. Otherwise, let $t+1 \geq s$ be a stage at which we modify the list A (which is initially equal to A_t) at position i and place there some numeral β. The modification of the numeral at position i happens while looping over the positions $p = 0, 1, \ldots, k-1$ of A_t and performing the prescribed actions on A for the conditions requiring action at position p. Note that by the inductive assumption the numeral at position i is not affected while scanning the positions $< i$. The method of scanning A_t and modifying A also guarantees that the numeral at position i is not affected while scanning the positions $> i$. Hence, the numeral at position i changes while scanning i. Hence β, when landing at position i, is fresh.

Since β is a fresh numeral inserted in response to (i), it does not appear in any L_n. It follows that no program e requires attention at a future stage $n+1$ with $(e, \beta) \in L_n$, $\psi_{e,n}(\beta)\downarrow$. Therefore, in the future, no new condition of the form $f^\sigma(\beta) \neq \gamma$ (for some γ) will become active. However, the numeral at position i can change in the future but this is possible only if there will be active conditions $f^\sigma(\alpha) \neq \beta$ with α at position $\leq i$. Let $(e_1, \alpha_1), \ldots, (e_r, \alpha_r)$ be all pairs in L_t such that the position of α_l in A_t is $\leq i$, for $1 \leq l \leq r$. If some (e', δ) enters the lists L_n in the future, then the position of δ will be $> i$. Hence, $(e_1, \alpha_1), \ldots, (e_r, \alpha_r)$ are all such pairs. By now it should be clear that the numeral at position i will change finitely many times in the future. At a sufficiently late stage, all programs among $e_1, \ldots, e_r$ that should stop will stop on their corresponding inputs and will give rise to active conditions of the form $f^\sigma(\alpha) \neq \beta$ with α at position $\leq i$. They will soon be fulfilled once and for all and later no changes will affect position i.

■

The proofs of Lemmas 3 and 4 remain unchanged. To repeat the proof of Lemma 4 we need

Lemma 6 *Every requirement $\mathcal{R}_e$ is satisfied at all stages.*

Proof. Let e be a program. By Lemma 1(b), a pair (e, α), for some α, at some point enters the lists L_n. If $\psi_e(\alpha)\uparrow$, $\mathcal{R}_e$ is always satisfied. Suppose $\psi_e(\alpha)\downarrow = \beta$. Then at some stage $n+1$, e requires attention, so $f^\sigma(\alpha) \neq \beta$ becomes an active condition. It suffices to show that, for every stage, all conditions active at that stage are satisfied. Let $n+1$ be a stage. In this paragraph, by active conditions we mean the elements of C_{n+1}.

Recall that, in the construction, we set a variable $A := A_n$, we loop over all positions $p = 0, 1, \ldots, k-1$ in A_n, and for each value of p, we perform actions for active conditions that require action at position p. Let p be a currently scanned position in A_n and suppose that, for all $r < p$, all active conditions that have required action at position r are satisfied by A. Our claim is that performing actions prescribed by the construction when p is the currently scanned position, leads to a (potential) modification of A such that all active conditions that require action at positions $r \leq p$ are satisfied by the modified A. In the two following paragraphs we show that the updated A satisfies all conditions that require action at position p.

If no condition requires action at position p, we do nothing to A so the claim follows. If there is precisely one condition requiring action at position p, then we perform one of the actions (1), (2), (3) or (4). Note that (1) is feasible because f is not an almost identity function. Feasibility of other cases follows from the fact that f is not an almost constant function. After performing the prescribed action on A, the condition that has just required action at position p is clearly satisfied by A.

Now, suppose that more than one condition requires action at position p. As we noted in the construction, there can by many such conditions of type (a) but at most one condition of other types. If we have a condition of type (d), we first deal with other conditions and afterwards we satisfy type (d) using (4). Let us show that conditions of type (a) and at most one condition of type (b) or (c) can be satisfied at once. Let us have m conditions of type (a): $f^\sigma(\alpha_1) \neq \beta$, $f^\sigma(\alpha_2) \neq \beta$, $\ldots$, $f^\sigma(\alpha_m) \neq \beta$ with i_j being the position of α_j in A. p is the position of β. Clearly, we can find a position $p' > p$ such that $f(i_1) \neq p'$, $\ldots$, $f(i_m) \neq p'$: just take $p' = \max(p+1, f(i_1), \ldots, f(i_m))$. Now, let us have m conditions as above and one additional condition of type (b) or (c). For sufficiently large p'', i.e. for $p'' \geq p'$, we always have $f(i_1) \neq p''$, $\ldots$, $f(i_m) \neq p''$. If the additional condition is of type (b), i.e. $f^\sigma(\beta) \neq \alpha$, with α at position $q < p$, then, since f is not almost constant, for some $p'' > p'$

we must have $f(p'') \neq q$.[4] On the other hand, if the additional condition is of type (c), i.e. $f^\sigma(\beta) \neq \beta$, then, since f is not almost identity, for some $p'' > p'$ we must have $f(p'') \neq p''$.[5] Anyway, we just find an appropriate p'', move β to position p'', and fill the positions $p, p+1, \ldots, p''-1$ with fresh numerals. This clearly satisfies all the m conditions of type (a) and one condition of type (b) or (c). Hence, we have shown that if there are more than one condition requiring action at position p, we have them all satisfied after modifying A as described in the construction.

Note that the modifications of A performed when p is the currently scanned position do not affect positions $< p$. Therefore, all active conditions that have required action at position $r < p$ remain satisfied by the modified A. Therefore, A satisfies all conditions that require attention at positions $\leq p$.

Note that every active condition requires action at some position $p = 0, 1, \ldots, k-1$. Hence, once looping over the positions $p = 0, 1, \ldots, k-1$ and performing the prescribed actions is finished, the variable A stores a list that satisfies all active conditions. The construction sets $A_{n+1} = A$ so, clearly, all active conditions at stage $n+1$ are satisfied at that stage.

■

■

4.3 Other generalizations

An immediate observation is that in every notation the computability of each of the following relations: $<$, $>$, $\leq$, $\geq$. Below we present several ways of strengthening this theorem even further.

Theorem 4 *For every nontrivial $f : \mathbb{N} \to \mathbb{N}$, and every ω-type ordering $\prec$ there exists a notation (S, σ) such that $\prec$ is computable in (S, σ), but f is not.*

Proof.

We construct the following notation (T, τ):

T is the standard set of numerals. The function τ assigns to subsequent numerals $\overline{0}$, $\overline{1}$, $\overline{2}$, ... natural numbers, starting from the smallest one, according to their position in ordering $\prec$.

[4]For the record, f is not almost constant if $\forall_q \forall_{p'} \exists_{p''} (p'' > p' \wedge f(p'') \neq q)$.

[5]For the record, f is not almost identity if $\forall_{p'} \exists_{p''} (p'' \geq p' \wedge f(p'') \neq p'')$.

Obviously, $\prec$ is computable in (T, τ), since for any $m, n \in \mathbb{N}$:

$$\overline{m} \prec^{\tau} \overline{n} \Leftrightarrow m < n.$$

If f is not computable in (T, τ), then let $(S, \sigma) = (T, \tau)$. Otherwise, we perform the construction from Section 4.2 except for two differences. First, we define (S, σ) by setting $S = T$ and, for each $i \in \mathbb{N}$:

$$\sigma(\lim_{n\to\infty} A_n(i)) = \tau(\bar{i}). \tag{6}$$

The second difference concerns the way we find appropriate positions for numerals when attempting to satisfy active conditions. Previously, we relied on computing the values of f for natural numbers. By f's nontriviality we were able to find positions i, j such that $f(i) \neq j$ to satisfy an active condition $f^{\sigma}(\alpha) \neq \beta$ by placing α, β at positions i, j. This was possible because f was computable in the standard notation. Now, f may not be computable in the standard sense. However, by assumption, f is computable in (T, τ) so we have at our disposal a program computing f^{τ}. Let us consider an active condition $f^{\sigma}(\alpha) \neq \beta$. Now, we say that a condition $f^{\sigma}(\alpha) \neq \beta$ is satisfied at stage n, if there exist i, j such that $A_n(i) = \alpha$, $A_n(j) = \beta$ and $f^{\tau}(\bar{i}) \neq \bar{j}$. By f's nontriviality we are able to find $\bar{i}, \bar{j}$ such that $f^{\tau}(\bar{i}) \neq \bar{j}$ and place α, β at positions i, j.

To give an example, consider the case (2) in the construction from Section 4.2. We have α at position i and β at position j with $i < j$: we insert fresh numerals in front of β and move it to a position $j' > j$ such that $f^{\tau}(\bar{i}) \neq \overline{j'}$.

Clearly, $\prec$ is computable in (S, σ). Recall that there is a program which, given any $\alpha, \beta \in S$, answers whether the construction ultimately places α before β. This program computes $\prec$ in (S, σ) because, by the definition of τ and (6), for the ultimate positions i, j of α, β:

$$i < j \Leftrightarrow \tau(\bar{i}) \prec \tau(\bar{j}) \Leftrightarrow \sigma(\alpha) \prec \sigma(\beta).$$

Finally, let us show that f is not computable in (S, σ). Suppose it is. Let e be a program computing it. At some point of the construction we discover that $\psi_e(\alpha) = \beta$ which gives rise to an active condition $f^{\sigma}(\alpha) \neq \beta$. Let i, j be the ultimate positions of α, β. By (suitably adapted) Lemma 6, we know that $f^{\tau}(\bar{i}) \neq \bar{j}$. The following are equivalent: $f^{\tau}(\bar{i}) \neq \bar{j}$, $f(\tau(\bar{i})) \neq \tau(\bar{j})$, $f(\sigma(\alpha)) \neq \sigma(\beta)$, $f^{\sigma}(\alpha) \neq \beta$. Hence, e does not compute f in (S, σ). A contradiction. ∎

We proceed to generalization of Theorem 3 to functions of other arities. We say that a function $f : \mathbb{N}^n \to \mathbb{N}$ is almost constant if there exists $y \in \mathbb{N}$ such that $f(x_1, \ldots, x_n) = y$ holds for all but finitely many tuples $x_1, \ldots, x_n$. Similarly, a function $f : \mathbb{N}^n \to \mathbb{N}$ is said to be almost projection if there exists i such that $1 \leq i \leq n$ and $f(x_1, \ldots, x_n) = x_i$ holds for all but finitely many tuples $x_1, \ldots, x_n$. A nontrivial function is a function that is neither almost constant nor almost projection. Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ be functions. We say that g is is substitution of f if and only if g can be obtained from f by substituting some number a for one of variables in f.

Theorem 5 *Let $k > 1$ and let $f : \mathbb{N}^k \to \mathbb{N}$ be such that for some i, $1 \leq i \leq k$, almost all substitutions for the i-th variable in f are nontrivial. Then there exists a notation (S, σ) in which the ordering $<$ is computable, but f is not.*

Proof. Let $f : \mathbb{N}^k \to \mathbb{N}$ be as above. Choose i such that almost all substitutions for the i-th variable in f are nontrivial. Let a be the least natural number such that, for each $b > a$, substituting b for the i-th variable in f results in a nontrivial function.

We utilise a construction similar to the one from Theorem 3. At stage 0 we put $A_0 = \varsigma_0\varsigma_1 \ldots \varsigma_a$ (i.e., we begin the sequence with $\varsigma_0, \varsigma_1, \ldots, \varsigma_a$) and $L_0 = \emptyset$ (i.e., we do not start testing any programs). Since programs are always assigned fresh numerals and numerals change their positions only if they are assigned to some program, at every stage s we will have $A_s = \varsigma_0\varsigma_1 \ldots \varsigma_a B$, for some list of numerals B. Hence, programs will always be tested on numerals which denote numbers $> t$.

We say that e requires attention at stage n if there exist $\beta_1, \ldots, \beta_k$ such that $(e, \beta_1, \ldots, \beta_k) \in L_n$ and $\psi_{e,n}(\beta_1, \ldots, \beta_k) \downarrow$. If no program requires attention at stage n, we pick a fresh program e, fresh numerals $\alpha_1, \ldots, \alpha_k$ and set $A_{n+1} = A_n\alpha_1 \ldots \alpha_k$, $C_{n+1} = C_n$; the difference is that we set $L_{n+1} = L_n \cup \{(e, \beta_1, \ldots, \beta_k)\}$, where $\beta_1, \ldots, \beta_k$ is a permutation of $\alpha_1, \ldots, \alpha_k$ such that $\alpha_k \neq \beta_i$. This modification is combined with the original idea that for each condition of the form $f^\sigma(\beta_1, \ldots, \beta_k) \neq \beta$ from C_n (with $\beta_1, \ldots, \beta_k$ being a permutation of $\alpha_1, \ldots, \alpha_k$ which were appended originally to the sequence in this order) we can only move the last numeral on the list A_n that is included in this condition (which is α_k). By moving it (to the right), we do not change the position of β_i (which is equal to α_j, for some $j < k$). Since β_i denotes a number which substituted for the i-th variable gives us a nontrivial function (see the last remark of the previous paragraph), we can satisfy the condition by changing the position of α_k (i.e. the argument which is not the i-th one).

■

In the above theorem it is not sufficient to assume that f itself is not trivial (i.e. that it is neither almost constant, nor almost a projection). For, suppose that $f : \mathbb{N}^2 \to \mathbb{N}$ is defined in the following way: $f(a,b) = 1$ if and only if $a < b$, else $f(a,b) = 0$. Computability of f obviously implies that of $<$, hence the above theorem does not work for such function f. However, while f is not trivial, any substitution of f results in an almost constant function.

Theorem 6 *For each uniformly computable family of nontrivial functions there exists a notation (S,σ) in which $<$ is computable but no function from this family is computable in (S,σ).*

Proof. It is a straightforward modification of the proof of Theorem 3. The only difference is that we must assure that no function from a fixed uniformly computable family of nontrivial functions $\{f_i\}_{i\in I}$, where $I = \{i : 0 \le i < k\}$, for some $k \in \mathbb{N} \cup \{\infty\}$, is computable in (S,σ). For each program $e \in \mathbb{N}$ and index $i \in I$ we have the following requirement:

$$\exists_\alpha[\psi_e(\alpha)\downarrow \implies f_i^\sigma(\alpha) \neq \psi_e(\alpha)]. \qquad (\mathcal{R}_{e,i})$$

Instead adding pairs (e,α) to L_n, we add triples (e,i,α), where $i \in I$. Now, attention is required by pairs (e,i) (for the same reason as previously). If (e,i) requires attention with $\psi_{e,n}(\alpha)\downarrow\beta$, we withdraw (e,i,α) from L_n (as previously) and add a condition $f_i^\sigma(\alpha) \neq \beta$. The rest of the proof remains virtually the same. ■

4.4 Learnability

As can be seen from Theorem 2, Benacerraf's claim, translated to Shapiro's framework, cannot be sustained. However, the question can be asked what happens if we replace computability with a weaker notion. Do Theorems 2 and 3 still hold? Here, we consider this question with regard to the concept of learnable functions and relations. It turns out that replacing computability with learnability does not help—such reinterpretation of Benacerraf's claim is false as well, unless we consider its weakest version stated below.

We show that notations constructed in the proofs of Theorem 2 are not enough to refute the following weaker version of Benacerraf's claim.

Theorem 7 *If (S, σ) is a notation and $<$ is computable in (S, σ) then the successor is learnable in (S, σ).*

Proof.

Suppose that $(\alpha_n)_{n \in \mathbb{N}}$ is a recursive enumeration of S and that $<$ is computable in (S, σ). Let $\{p_n\}_{n \in \mathbb{N}}$ be an enumeration of all subsets of S such that $|p_n| = 2$.

Then $Succ^\sigma$ is the limit of sequence of recursive functions defined as follows:

$$Succ^\sigma_0(\alpha_n) = \alpha_{n+1}.$$

This function corresponds to the following ordering of elements of S: $\alpha_0, \alpha_1, \ldots$. In subsequent steps we are going supply different inputs to the algorithm for $<$ and modify the ordering of numerals and the successor function accordingly. We are also going to remember all the answers we got in order to ensure that all of them are satisfied by the sequence obtained.

Once we have constructed $Succ^\sigma_t$ we obtain $Succ^\sigma_{t+1}$ in the following way: let $p_t = \{\alpha_i, \alpha_j\}$. We ask if $\alpha_i <^\sigma \alpha_j$. We either add $\alpha_i <^\sigma \alpha_j$ or $\alpha_j <^\sigma \alpha_i$ to the list of conditions which need to be satisfied. If the answer corresponds to the order of numerals according to $Succ^\sigma_t$, then $Succ^\sigma_{t+1} = Succ^\sigma_t$.

If, however, the answer does not correspond to the current state of the sequence, we consider all the numerals the positions of which with regard to α_i and α_j have already been established by the algorithm for $<$. Then move α_i and α_j within the sequence to ensure that all conditions are satisfied. Define $Succ^\sigma_{t+1}$ accordingly.

At a certain point, when we have checked the order of all numerals representing the initial segment of natural numbers, each of their positions will be fixed. It can however still be the case that there are some additional numerals between them. But since there are only finitely many of them, after a certain number of steps the algorithm will determine the order of them as well. Hence the algorithms for successor will give correct answers when asked about this initial segment of natural numbers.

$\blacksquare$

Here we consider another weaker version of Benacerraf's claim. This version is false. We present a counterexample obtained by relativization of the construction from the second proof of Theorem 2.

Theorem 8 *There exists a notation (S, σ) in which $<$ is learnable but the successor is not.*

Proof. By Definition 4 and Theorem 1, R is learnable in (S, σ) iff $R^\sigma \leq 0'$. Similarly, a function $f : \mathbb{N} \to \mathbb{N}$ is learnable in (S, σ) iff $f^\sigma \leq 0'$. We repeat the construction from the second proof of Theorem 2 with the exception that now the construction is not recursive but rather recursive in $0'$. Instead of using $\psi_0, \psi_1, \ldots$ as the enumeration of unary partial recursive functions, we use the enumeration $\psi_0^K, \psi_1^K, \ldots$. This enumeration consists of all partial recursive in $0'$ functions from S to S which are potential candidates to compute (in $0'$) the successor. Because we can use the $0'$ oracle, we can effectively (in $0'$) establish the truth-value of any statement of the form $\psi_{e,n}^K(\alpha)\downarrow$. Accordingly, each occurrence of $\psi_{e,n}$ or ψ_e in the proof of Theorem 2 should be replaced with $\psi_{e,n}^K$ and ψ_e^K. For example, for each oracle program e, we have the requirement:

$$\exists_\alpha [\psi_e^K(\alpha)\downarrow \implies \sigma(\alpha) + 1 \neq \sigma(\psi_e^K(\alpha))]. \qquad (\mathcal{R}_e)$$

Constructed this way, the notation (S, σ) is such that $<^\sigma$ is computable in $0'$ and $Succ^\sigma$ is not computable in $0'$. Hence, by Theorem 1 and Definition 4, $<$ is learnable in (S, σ) but $Succ$ is not learnable in (S, σ). ■

Note that, by Theorem 7, the ordering is not computable in the notation obtained in Theorem 8.

Theorem 9 *Let (S, σ) be a notation. If the successor is learnable in (S, σ) then $<$ is learnable in (S, σ).*

Proof. Let the successor $Succ$ be learnable in (S, σ). Let $(Succ_n)_{n \in \mathbb{N}}$ be a family of functions, uniformly computable in (S, σ), such that $\lim_{n\to\infty} Succ_n = Succ$. We define $(<_n)_{n\in\mathbb{N}}$ in stages $n = 0, 1, \ldots$. Let $\alpha, \beta \in S$.

Stage 0. Let $d_0(\alpha, \beta) = \infty, d_0(\beta, \alpha) = \infty$. Return 0.

Stage $n + 1$. If $\alpha = \beta$, return 0. Suppose $\alpha \neq \beta$. Compute the lists $A_n = \alpha_0, \alpha_1, \ldots, \alpha_n$ and $B_n = \beta_0, \ldots, \beta_n$, where $\alpha_0 = \alpha$ and $\beta_0 = \beta$, $\alpha_{k+1} = Succ_n(\alpha_k)$ and $\beta_{k+1} = Succ_n(\beta_k)$. Let $d_{n+1}(\alpha, \beta)$ be the index of the first occurrence of β in A_n, or ∞, if β does not occur in A_n. Similarly, let $d_{n+1}(\beta, \alpha)$ be the index of the first occurrence of α in B_n, or ∞, if α does not occur in B_n. Return the truth value of the expression $d_{n+1}(\alpha, \beta) < d_{n+1}(\beta, \alpha)$.

Let us verify that the construction yields a uniformly computable family of relations converging to $<$. Without loss of generality, assume $\alpha < \beta$. Let $d = \beta - \alpha$. Note that $d = \lim_{n\to\infty} d_n(\alpha, \beta)$, i.e. $d_n(\gamma, \delta)$ approximates the number of steps required to reach β from α by successive applications of

$Succ$. Let $B = \beta_1, \beta_2, \ldots, \beta_d$ be numerals where β_i is the ith iteration of $Succ$ starting from β. Clearly, for sufficiently large n, α does not occur at positions $1, 2, \ldots, d$ in B_n which implies $d_n(\beta, \alpha) > d$. Therefore, for sufficiently large n, we always have $d_n(\alpha, \beta) < d_n(\beta, \alpha)$, as desired. ■

Theorem 10 *There exists a notation (S, σ) in which the successor is learnable but $<$ is not computable.*

Proof. S is the standard set of numerals. Let $A = \{\alpha_n\}_{n \in \mathbb{N}}$ be a recursively enumerable but not recursive set of natural numbers. We define σ as follows:

$$\sigma(\overline{2n}) = \begin{cases} 2n & \text{if } n \in A, \\ 2n+1 & \text{if } n \notin A, \end{cases}$$

$$\sigma(\overline{2n+1}) = \begin{cases} 2n & \text{if } n \notin A, \\ 2n+1 & \text{if } n \in A. \end{cases}$$

The idea behind this construction is such that if $n \notin A$, then the interpretation of $\overline{2n}$ and $\overline{2n+1}$ is standard, otherwise it is reversed.

Successor is learnable in (S, σ) since $Succ^\sigma$ is the limit of a recursive sequence of functions $Succ^\sigma_t$ defined as follows:

$$Succ^\sigma_t(\overline{2n}) = \overline{2n+1} \text{ if } n \text{ is not among } \alpha_0, \ldots, \alpha_t,$$

$$Succ^\sigma_t(\overline{2n}) = \overline{2n+2} \text{ if } n \text{ is among } \alpha_0, \ldots, \alpha_t \text{ but } n+1 \text{ is not,}$$

$$Succ^\sigma_t(\overline{2n}) = \overline{2n+3} \text{ if both } n \text{ and } n+1 \text{ are among } \alpha_0, \ldots, \alpha_t,$$

$$Succ^\sigma_t(\overline{2n+1}) = \overline{2n} \text{ if } n \text{ is among } \alpha_0, \ldots, \alpha_t,$$

$$Succ^\sigma_t(\overline{2n+1}) = \overline{2n+2} \text{ if neither } n \text{ nor } n+1 \text{ are among } \alpha_0, \ldots, \alpha_t,$$

$$Succ^\sigma_t(\overline{2n+1}) = \overline{2n+3} \text{ if } n \text{ is not among } \alpha_0, \ldots, \alpha_t \text{ but } n+1 \text{ is.}$$

However, $<$ is not computable in (S, σ). If it were, then for A would be decidable since for every $n \in \mathbb{N}$:

$$\overline{2n} <^\sigma \overline{2n+1} \Leftrightarrow n \notin A.$$

■

5 Conclusions

In this paper we have shown several results regarding the relationship between computability of successor, nontrivial functions and ordering in notations for natural numbers. Nontrivial functions are functions that are neither almost constant nor almost identity. One of the main results (Theorem 4) states that computability of any ω-type ordering does not imply computability of any unary nontrivial function.

The results are also of philosophical significance. They show that Benacerraf's original view, expressed in [1], stating the equivalence between the computability of the ordering and successor, cannot be sustained, regardless of whether we share his later criticism of it formulated in [2], or not. The result refuting Benacerraf's claim is obtained in Shapiro's framework of notations for natural numbers.

Theorem 2 *There exists a notation (S, σ) in which the ordering $<$ is computable, but the successor function is not computable.*

Obviously, one should be careful in drawing conclusions from the formalism in which the claim was not natively expressed. It turns out, however, that regardless of which route we take in interpreting Benacerraf's position, his claim cannot be maintained due to Theorem 2.

If we assume that the concept of a notation utilised by Benacerraf is akin to that of Shapiro, then our arguments are sufficient to refute his claim. Obviously, if we decided to weaken the definition of this notion, then it would not invalidate our refutation of Benacerraf's claim in any way.

If, however, we are more convinced by the view that Benacerraf's claim concerns merely objects that serve as referents for numerals, then we also have suitable tools at our disposal. In such case, the computability of the ordering and of the successor is understood as computability of the ordering relation between these objects and the successor operation which maps each object to the next one in the order, while the nature of such objects is entirely irrelevant. The proofs of Theorem 2 show how one can obtain an ω-type ordering that refutes Benacerraf's claim under this interpretation as well.

Refutation of Benacerraf's claim can be used to criticise his argument for the computability of ordering from [1]. The argument from [1] purported to show that the ordering of numbers should be decidable because it would guarantee that numbers exhibit desirable computational properties. Benacerraf's original view was that the ability to say numbers in order of magnitude (intransitive counting) is essential if we want to treat any sort of objects as fulfilling the role of natural numbers. Since recursiveness of intransitive counting is equivalent to successor's computability, Theorem 2 clearly invalides this kind of argument: decidability of ordering is not enough to ensure that successor is computable.

We also showed that Benacerraf's claim regarding the equivalence between ordering and successor fails if we replace computability with a weaker notion, here referred to as learnability, originally dubbed by Gold [7] *recursiveness in the limit* (see Theorem 8). For similar reasons as before, this modified claim cannot be used to argue for the learnability of the ordering as this condition cannot guarantee successor's learnability. However, computability of the ordering entails learnability of successor. This, clearly, is not enough to support Benacerraf's argument, yet it seems an interesting result that a certain weak reinterpretation of his claim is correct.

Further questions of a similar kind can be asked. Many of them have been answered in [17], [18] and [19]. In particular, in [18] several hypotheses have been suggested regarding generalisation of these results, and answers to some special cases have been given. The general question is: consider all notations[6] in which all functions (or relations) from a certain class are computable. What is the class of all functions (or relations) computable in each of these notations? In general case, this question until now has not been answered.

References

[1] Paul Benacerraf. What Numbers Could Not Be. *Philosophical Review*, 74(1):47–73, 1965.

[2] Paul Benacerraf. Recantation or Any Old ω-sequence Would Do After All. *Philosophia Mathematica*, 4(2):184–189, 1996.

[3] Tim Button and Peter Smith. The Philosophical Significance of Tennenbaum's Theorem. *Philosophia Mathematica*, 20(1):114–121, 2012.

[4] Brian Jack Copeland and Diane Proudfoot. Deviant Encodings and Turing's Analysis of Computability. *Studies in History and Philosophy of Science Part A*, 41(3):247–252, 2010.

[6]Possibly such that the function σ is not bijective, unlike in considerations in this paper.

[5] Walter Dean. Models and Recursivity. Available at `https://pdfs.semanticscholar.org/692a/f690a97981d992983abc92b4cfc4385b31c4.pdf`, 2002.

[6] Michał Tomasz Godziszewski and Joel David Hamkins. Computable Quotient Presentations of Models of Arithmetic and Set Theory. *arXiv e-prints*, page arXiv:1702.08350, 2017.

[7] E. Mark Gold. Limiting Recursion. *Journal of Symbolic Logic*, 30(1):28–48, 1965.

[8] Paula Quinon. A Taxonomy of Deviant Encodings. In Dirk Nowotka Florin Manea, Russell G. Miller, editor, *14th Conference on Computability in Europe, CiE 2018, Lecture Notes in Computer Science*, volume 10936 LNCS, pages 338–348. Springer Verlag, Kiel, 2018.

[9] Paula Quinon and Konrad Zdanowski. The Intended Model of Arithmetic. An Argument From Tennenbaum's Theorem. In S. Barry Cooper, Thomas F. Kent, Benedikt Löwe, and Andrea Sorbi, editors, *Computation and Logic in the Real World, CiE 2007, Local Procedings*. Sienna, 2007.

[10] Michael Rescorla. Church's Thesis and the Conceptual Analysis of computability. *Notre Dame Journal of Formal Logic*, 48(2):253–280, 2007.

[11] Michael Rescorla. Copeland and Proudfoot on Computability. *Studies in History and Philosophy of Science Part A*, 43(1):199–202, 2012.

[12] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, Cambridge, MA, USA, 1987.

[13] Stewart Shapiro. Acceptable Notation. *Notre Dame Journal of Formal Logic*, 23(1):14–20, 01 1982.

[14] Joseph R Shoenfield. On degrees of unsolvability. *Annals of mathematics*, 69:644–653, 1959.

[15] Robert I. Soare. *Recursively Enumerable Sets and Degrees*. Springer-Verlag New York, Inc., New York, NY, USA, 1987.

[16] Stanley Tennenbaum. Non-Archimedean Models for Arithmetic. In *Notices of the American Mathematical Society 6*. 1959.

[17] Michał Wrocławski. Representing Numbers. *Filozofia Nauki*, 26(4):57–73, 2018.

[18] Michał Wrocławski. Representations of Natural Numbers and Computability of Various Functions. In Florin Manea, Barnaby Martin, Daniel Paulusma, and Giuseppe Primiero, editors, *15th Conference on Computability in Europe, CiE 2019, Lecture Notes in Computer Science*. Springer Verlag, 2019.

[19] Michał Wrocławski. *Representations of Numbers and their Computational Properties*. PhD thesis, 2019.