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

Dariusz Kalociński & Michał Wrocławski

April 19, 2020

Abstract

The class of all computable functions depends on the notation we choose for numbers. In this paper we consider the concept of a notation, 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 directions. We show that for every ordering isomorphic to the 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 our result to uniformly computable families of functions and to functions of other arities. Finally, we consider a weaker notion than computability, namely that of learnability, first investigated by Gold, and show that learnability of successor and ordering are equivalent, which redeems Benacerraf's claim at a higher level.

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 $\sigma : S \to \mathbb{N}$ which maps the 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 subset of $\mathbb{N}$ 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 [18] (see also [16, 17]), Michał Wrocławski considered the notion of a representation which generalises the concept of a notation in such a way that the function σ is only assumed to be surjective, not necessarily bijective (see: Definition 2). Somewhat similar questions have been considered by Michał Tomasz Godziszewski and Joel David Hamkins in [6], though the terminology utilised there is different and finite sets are considered rather than the whole set $\mathbb{N}$. Interesting computability results have also been proved by Konrad Zdanowski and presented at a conference [19] 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 [15]. 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 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. It should be noted that Shapiro's definition of a notation demands that the set of all numerals to which numbers are assigned must be computable—which is clearly not the case in one of examples given by Benacerraf in [2]. Obviously, this restriction is irrelevant: refutation of Benacerraf's claim by providing a notation in which it fails remains a refutation, no matter whether the set of numerals is computable or not. Apart from the above-mentioned difference, Shapiro's framework seems to be the best existing formalism to analyse Benacerraf's problem.

In Shapiro's terms, the Benacerraf's claim can be formulated as follows: in every notation, computability of the 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.

[1]However, note that if we allow a number to be represented by more than one numeral, this implication does not always hold—see [16] and [17].

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. In the finale, a completely novel idea is applied to notations for natural numbers—namely that of learnability which leads to Benacerraf's partial redemption.

The paper is structured as follows. Section 2 introduces necessary terminology. Section 3 is the technical part of the paper containing all the main results, spread throughout three subsections. In Section 3.1 we refute Benacerraf's claim. The following subsections provide various generalizations. In Section 3.2 we prove a theorem on nontrivial functions. Generalizations to ω-type orderings, collections of functions and functions of higher arities are provided in Section 3.3. In Section 3.4 the notion of learnability in notations is applied to Benacerraf's problem. Section 4 concludes the paper. Section 5 is an appendix which shows how to apply the method developed in Section 3.1 to prove that the computability of the ordering does not imply the computability of divisibility.

2 Preliminaries

In this section we start by defining all the necessary terminology. First, certain classical computability-theoretic notions are introduced. Next, we proceed to the definition of a notation for natural numbers and of the notions of computability and learnability in a notation.

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). Words are usually referred to by lowercase Greek letters $\alpha, \beta, \ldots,$ possibly with subscripts. 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 some 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 natural and well-researched extension of computable functions and relations is provided by the notion of learnable functions and relations.

Definition 1 (Gold [7]) *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 $n_1, \ldots, n_k \in \mathbb{N}$:*

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

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 $n_1, \ldots, n_k \in \mathbb{N}$:

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

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. 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 $\psi(0), \psi(1), \ldots$ 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, 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. By means of coding, the above notions (of a partial recursive function, decidability, 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, 14].

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.

Definition 2 (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.

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 is a difference between Shapiro's and Benacerraf's approach, as the latter does not postulate such a requirement [1, 2].

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 main idea behind the concept of a notation is that it precisely delineates numbers from their names. But the depth of this concept cannot be fully appreciated until we define what it means for a function or a relation to be computable in a notation. After all, we do not perform calculations directly on numbers but rather on numerals.

Definition 3 (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.

The notion of computability in a notation extends naturally to relations.

Definition 4 *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.

Definition 5 *Let (S, σ) be a notation. We say a function $f : \mathbb{N} \to \mathbb{N}$ 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 $\alpha \in S$:*

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

Definition 6 *Let (S, σ) be a notation. We say a relation $R \subseteq \mathbb{N}^k$ is learnable in (S, σ) if there exists a family relations $R_t \subseteq \mathbb{N}^k$, $t \in \mathbb{N}$, uniformly computable in (S, σ), such that for each $(\alpha_1, \ldots, \alpha_k) \in S$:*

$$R^\sigma(\alpha_1, \ldots, \alpha_k) = \lim_{t \to \infty} R_t^\sigma(\alpha_1, \ldots, \alpha_k). \tag{4}$$

3 Results

3.1 Successor

Theorem 1 *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:

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

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). 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 1. Our technique will be later generalised to show stronger results. The construction is a bit involved, but it is worth the effort.

Theorem 1 *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 computable sequence of numerals $(\alpha_i)_{i \in \mathbb{N}}$, without repetitions. The denotation of α_i, $i \in \mathbb{N}$, 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 $A_0, A_1, \dots$, 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' enter 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 $A_0, A_1, \dots$ 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.

$\blacksquare$

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. $\blacksquare$

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 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 $i + 1 \neq j$. ■

■

3.2 Nontrivial functions

A theorem similar to Theorem 1 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. Similarly, a function $f : \mathbb{N} \to \mathbb{N}$ is said to be almost identity, if $f(x) = x$ holds for all but finitely many x. A nontrivial function is a function that is neither almost constant nor almost identity. Clearly, the successor function is nontrivial. Theorem 2 is a generalisation of Theorem 1 to nontrivial functions. The generalization is not obvious—unlike the previous proof, an attempt to satisify 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 2 *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 1, with f in place of the successor. We start by outlining the main differences. Again, let $\varsigma_0, \varsigma_1, \ldots$ be a computable enumeration, without repetitions, of an infinite computable set S. Let $\psi_0, \psi_1, \ldots$ be a canonical enumeration of all partial recursive functions from S to S. For each program e we have the requirement

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

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 $f(i') = j'$.[2]

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 $\varsigma_0, \varsigma_1, \dots$ be a recursive enumeration (without repetitions) of a computable set S. Let $\psi_0, \psi_1, \dots$ 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 $p = 0, 1, \dots, k-1$, 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 $\max(i, j) = p$, where i, j are the positions of α, β.

Let $A := A_n$ be a temporary variable. We loop over the positions $p = 0, 1, \dots, k-1$ 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.

[2]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 1 but are necessary here, where f is an arbitrary nontrivial function.

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 $p, p+1, p'-1$ 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 $p = 0, 1, \dots, k-1$ 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 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): $f^\sigma(\alpha_1) \neq \beta$, $f^\sigma(\alpha_2) \neq \beta$, $\dots$, $f^\sigma(\alpha_m) \neq \beta$ 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 $f(i_1) \neq p'$, $\dots$, $f(i_m) \neq p'$: just take $p' = \max(p+1, f(i_1), \dots, f(i_m))$. 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 $f(p'') \neq q$.[3] 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 $f(p'') \neq p''$.[4] 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.

■

■

3.3 Other generalizations

An immediate observation is that in every notation the computability of each of the following relations: $<$, $>$, $\leq$, $\geq$ is equivalent, and each of them implies that $=$ is computable.

Below we present several ways of strengthening this theorem even further.

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

Proof.

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

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

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 $\prec$.

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 3.2 except for two differences. First, 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 3.2. 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: $i < j \iff \tau(\bar{i}) \prec \tau(\bar{j}) \iff \sigma(\alpha) \prec \sigma(\beta)$.

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 2 to functions of other arities. Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ and $g : \mathbb{N}^n \to \mathbb{N}$ be functions. We say that g is a 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 4 *For every $f : \mathbb{N}^n \to \mathbb{N}$ such that almost all of its substitutions are nontrivial, there exists a notation (S, σ) in which the ordering $<$ is computable, but f is not.*

Proof.

We utilise a very similar construction like previously. Since there are only finitely many trivial substitutions, we can enumerate all the numerals used in them: β_1, ... , β_t. We begin the sequence with $\beta_1, ..., \beta_t$. Then we perform the construction used for unary functions.

The only difference is that to each algorithm we assign a sequence of n new numerals $\alpha_1, ..., \alpha_n$ on which the algorithm is going to be carried out. Similarly like in the original case, for each condition we can only move the last numeral on the list that is included in this condition. Note that the requirement that no substitution of f is either almost constant, nor almost a projection ensures that this construction works.

■

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, it has many trivial substitutions. Substituting any argument for the second variable results in an almost constant function.

Theorem 5 *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 2. 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 $I = \{i : 0 \leq i < k\}$, for some $k \in \mathbb{N} \cup \{\infty\}$, is computable in (S, σ) under construction. 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 $i \in I$. (e, 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. ∎

3.4 Learnability

As Theorem 1 shows, Benacerraf's claim cannot be sustained. But perhaps the claim can be saved in a somewhat weaker form? A natural question to ask is what happens if we replace computability with a weaker notion. Do Theorems 1 and 2 still hold? There are many weaker notions sensible in the present context. Here, we consider a well-known formalization of learnable functions and relations. Suprisingly enough, replacing computability with learnability saves Benacerraf's claim.

Theorem 6 *Let (S, σ) be a notation. If $<$ is learnable in (S, σ) then the successor is learnable in (S, σ).*

Proof.

Suppose that $<$ is learnable in (S, σ). Let $(<_n)_{n \in \mathbb{N}} \subseteq \mathbb{N}^2$ be a family of relations, uniformly computable in (S, σ), such that for all $\alpha, \beta \in S$:

$$\alpha < \beta = \lim_{n \to \infty} \alpha <_n \beta.$$

Let $(p_n)_{n \in \mathbb{N}}$ be a computable enumeration without repetitions of all subsets of S such that $|p_n| = 2$. We write $p_n = \{\alpha_n, \beta_n\}$. On each such set we run consecutive algorithms checking if $\sigma(\alpha_n) <_n \sigma(\beta_n)$. From a certain point, they are always going to return the correct answer. We wish to create a set of conditions such that for every pair of distinct numerals α_n, β_n it contains either the condition $\sigma(\alpha_n) < \sigma(\beta_n)$ or $\sigma(\beta_n) < \sigma(\alpha_n)$.

We wish to construct inductively a sequence of functions $Succ_n$ whose limit is the successor function $Succ$. We begin with f_0 which is any function isomorphic to the successor function.

Suppose that the function f_n has already been constructed and we wish to construct f_{n+1}. We run the algorithm for $<_{n+1}$ on numerals from each of sets $p_1, ..., p_{n+1}$. If, supplied with numerals α_{n+1} and β_{n+1}, the algorithm returns 0, we add $\sigma(\beta_{n+1}) < \sigma(\alpha_{n+1})$ to the list of conditions. If it returns 1, we add $\sigma(\alpha_{n+1}) < \sigma(\beta_{n+1})$.

However, whenever we add a new condition, we remove the conditions which are conflicting with it. That is, if we add $\sigma(\alpha) < \sigma(\beta)$, but the condition $\sigma(\beta) < \sigma(\alpha)$ was already on the list, we remove the old condition.

In addition to that, if we add $\sigma(\alpha) < \sigma(\beta)$ while for some $\delta_1, ..., \delta_n$ the following conditions were added earlier: $\sigma(\beta) < \sigma(\delta_1)$, $\sigma(\delta_1) < \sigma(\delta_2)$, ... , $\sigma(\delta_{n-1}) < \sigma(\delta_n)$, $\sigma(\delta_n) < \sigma(\alpha)$, then we remove each of them.

Once this is done, rearrange the order of numerals to satisfy all conditions from the list. The function $Succ_{n+1}$ is constructed in a natural way based on this ordering of numerals.

Notice that for every initial segment of $\mathbb{N}$ consisting of numbers $0, \ldots, k$ the position of numerals representing each of these numbers will be fixed. That is because once we have performed the algorithm for $<$ on each pair consisting only of these numerals, no further occasion can arise to change their order. At this point the successor function for arguments $0, ..., k-1$ will be fixed. Since this observation holds for any n it follows that $\lim_{n\to\infty} Succ_n = Succ$. ■

Since computability entails learnability, we have

Corollary 1 *For every notation in which $<$ is computable, the successor is learnable.*

Moreover, the exact analogue of Theorem 2 is false.

Corollary 2 *It is not the case that for every nontrivial function f there exists a notation in which $<$ is computable but f is not learnable.*

Proof. Just choose f to be the successor function. ■

Finally, let us observe that the implication in the other direction also holds which, together with Theorem 6, makes the notions of learnability of the ordering and of the successor equivalent.

Theorem 7 *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 suffiently large n, we always have $d_n(\alpha, \beta) < d_n(\beta, \alpha)$, as desired. ■

4 Conclusions

In this paper we have shown several results regarding the relationship between computability of nontrivial functions and computability of the ordering in notations for natural numbers. Nontrivial functions are functions that are neither almost constant nor almost identity. Our main result (Theorem 3) 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 Bernacerraf'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. This is because, as we have shown (Theorems 1 and 2), the condition of decidability of the ordering on natural numbers is not sufficient for us to be able to compute all the functions we need. In other words, if we consider Benacerraf's criticism of his earlier position expressed in [2] to be valid, then we are going to reject his earlier argument. If, on the other hand, we believe that he was initially right to demand that the ordering of natural numbers should be decidable, then it seems that we should also expect successor to be computable. However, it not always is, even with decidable ordering—hence Benacerraf's original position was wrong anyway.

Surprisingly enough, Benacerraf's claim is redeemed at a higher level although such a possibility certainly was not considered by the author. If we replace computability with a weaker notion, here referred to as learnability (originally dubbed by Gold [7] *recursiveness in the limit*), and transfer it to the framework of notations, the equivalence asserted by Benacerraf becomes a fact (Theorems 6 and 7).

Further questions of a similar kind can be asked. Many of them have been answered in [16], [17] and [18]. In particular, in [17] several hypothesis have been suggested regarding generalisation of these results, and answers to some special cases have been given. The general question is: consider all notations[5] 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.

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

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

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

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

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

[17] 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.

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

[19] Konrad Zdanowski. On Notation Systems for Natural Numbers and Polynomial Time Computations. Unpublished slides from the conference 'Numbers and Truth', 2012.

5 Appendix

Theorem 8 *There exists a notation (S, σ) of $\mathbb{N}$ in which the ordering $<$ is computable, but divisibility $|$ is not.*

Proof.
Since the argument is very similar to previous cases, we only sketch it, without going into details.

Let S be a computable set of numerals and $(\zeta_n)_{n\in\mathbb{N}}$ — a computable enumeration of all elements of S without repetitions. Let $(e_n)_{n\in\mathbb{N}}$ be a computable enumeration of all algorithms on S which, given two numerals as input, return a logical value TRUE or FALSE as output. We shall provide an algorithm for the construction of an infinite sequence of numerals from S. During this construction, to each index of an algorithm we shall assign a pair of numerals on which this algorithm is going to be performed. The limit sequence obtained after infinitely many steps of this construction shall contain exactly one occurrence of every numeral from S. To each numeral we shall assign a natural number — the order of numbers will be the same as the order of numerals in the sequence.

Construction In the 0th stage of the construction let $A_0 = (\zeta_0, \zeta_1, \zeta_2, \zeta_3)$ and let $L_0 = ((e_0, \zeta_2, \zeta_3))$ (it means that we shall run the 0th algorithm on a pair of numerals ζ_2 and ζ_3). We are also going to construct a list of conditions which need to be satisfied at every stage. This list is empty at the beginning.

Suppose that we have already performed the first n stages of the construction, $n \geq 0$. Let $A_n = (\alpha_0, \alpha_1, ..., \alpha_k)$ and let

$$L_n = ((e_0, \alpha_{i_0}, \beta_{i_0}), (e_1, \alpha_{i_1}, \beta_{i_1}), ..., (e_m, \alpha_{i_m}, \beta_{i_m})),$$

where $0 \leq i_0, i_1, ..., i_m \leq k$.

In the $n + 1$th stage do the following (in order to ensure that none of the algorithms computes characteristic function of divisibility):

1. Take an unused (i.e. one that has never been in L_i for any $i \leq n$) algorithm e with the smallest index and take two unused numerals with the least indices λ_1 and λ_2 and assign these numerals to that algorithm.

2. Let A_{n+1} be equal to A_n with λ_1 and λ_2 added at the end.

3. Let L_{n+1} be equal to L_n with $(e, \lambda_1, \lambda_2)$ added at the end.

4. Perform the first $n + 1$ steps of all algorithms from L_{n+1} with their respective numerals as the input.

5. For each algorithm e_l working on numerals α_{i_l} and β_{i_l} that stops during these steps:

   (a) If e_l returns $TRUE$, then do nothing (because for sufficiently large numbers, it cannot be the case that one of the subsequent natural numbers is a divisor and the other one is its multiplicity).

   (b) If e_l returns $FALSE$, then we add $\alpha_{i_l} | \beta_{i_l}$ to the list of conditions.

   Then remove $(e_l, \alpha_{i_l}, \beta i_l)$ from L_{n+1}.

Now (if it is necessary) we are going to modify A_{n+1} in such a way as to ensure that each condition from the list is satisfied. To achieve this, we go through all numerals in A_{n+1}, one by one, in the same order in which they are situated there. If numerals α_{i_l} and β_{i_l} appear in a certain condition, we modify the sequence A_{n+1} when we reach β_{i_l}. The modification consists in adding sufficiently many new numerals directly in front of β_{i_l} to push it to a position in the sequence whose index is a multiple of the index of the position of α_{i_l}. Note that due to the construction of A_{n+1}, β_{i_l} always appears behind α_{i_l}. The placement of new filler numerals described here will help us avoid a situation in which we we spoil a condition which has already been satisfied.

This ends the construction.

Verification

Lemma 1 *a Each $\alpha \in S$ enters A_n at some stage n.*

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

Proof. Each numeral is added to a certain A_n because in each step we add to it two numerals with least unused indices and the procedure consists of infinitely many steps. Also, no repetitions of numerals are possible because in each step only unused numerals are added.

For an analogous reason, for each algorithm e, $(e, \alpha, \beta$ is added to L_n for certain $\alpha, beta$. ■

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

Proof.

This is a well-defined sequence, since position of each numeral in A is fixed at a certain stage of the construction. We are going to prove this now. Suppose that a numeral λ has been added to the sequence at a certain stage. Note that all numerals added later are going to be placed behind λ, except for those added to satisfy one of the conditions from the list. This, however, can only occur finitely many times, and each time we add finitely many filler numerals. This is because all numerals added to A_n this way must be placed directly in front of a numeral β_{i_l} assigned to a certain algorithm as its second argument and it is possible only if β_{i_l} was added to A_n before λ was. Therefore A is a well-defined sequence.

It follows from the above that all conditions from the list are going to be satisfied in A.

■

Lemma 3 *(S, σ) is a notation.*

The proof of this lemma is the same as in the previous theorems.

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

Proof.

To compare numbers represented by two numerals, we perform the algorithm described above until both numbers are in the sequence and then check which of them comes first. This is a correct method because once two numerals have been added to the sequence in a specific order, their order cannot be reversed (even if at a later step some additional numerals are added between them).

■

Lemma 5 $|$ *is not computable in* (S, σ).

Proof.

This is the case because we have ensured that none of the algorithms computes this function in (S, σ).

■

■