Uncomputability of non-trivial functions in certain notations for natural numbers with computable ordering

Dariusz Kalociński
Michał Wrocławski

February 19, 2020

Abstract

The class of all computable functions depends on the notation we choose for numbers. In this paper we consider the notion of a notation, as defined by Stewart Shapiro in [11] 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 successor — contrary to what Paul Benacerraf claimed in [1]. We then proceed to generalise this result and investigate which other functions can be uncomputable in a certain notation despite various orderings being computable. We show that for any ω-type ordering $\prec$ and any unary function on natural numbers f which is neither almost constant nor almost identity there exists a notation in which $\prec$ is computable but f is not. This result significantly strengthens one of theorems proved by Shapiro in his paper.

1 Introduction

In his paper [11] Stewart Shapiro considered the concept of a notation for natural numbers. He noted that computations are performed directly on numerals and only indirectly on numbers, and the choice of a notation determines the class of functions that are computable. He proved a number of formal results regarding how choice of a notation influences our ability to compute various functions. E.g. 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). He 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 successor is computable in it.

Various authors took part in the philosophical debate regarding this issue, most notably Michael Rescorla in [9] and [10], Brian Jack Copeland and Diane Proudfoot in [4] and Paula Quinon in [7]. Discussions about notations for numbers have mainly been philosophical, but there have also been some more formal results. In his Ph.D. thesis [15] Michał Wrocławski considered the notion of a representation which generalises the concept of a notation in such a way the function σ is only assumed to be surjective, not necessarily bijective (see: Definition 1), some of them have been published in [13] and [14]. 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 [16] 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 [12]. Among the key participants of this debate are: Walter Dean [5], Paula Quinon and Konrad Zdanowski [8], 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 a 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, 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 question asked by Benacerraf and Shapiro seems 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 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 which — as we believe — is a suitable way of thinking about Benacerraf's problem. What is more, it seems that our definition is even more restrictive in that it 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].

While it can be shown that computability of successor in a given notation implies computability of ordering[1], the implication in the opposite direction is not always true. In this paper, we give two different proofs of this statement. The first proof is shorter and easier, but the second one has an advantage that it can be generalised to prove much stronger Theorems 6 and 7. Finally, we show how a similar method can be used to prove a different result (Theorem 8).

2 Computability of functions in notations — formal results

In this section we start by defining all the necessary terminology, then we proceed to proving our theorems.

[1]It is an easy conclusion from Shapiro's considerations in [11], though it has not been explicitly stated there. However, note that if we allow a number to be represented by more than one numeral, this implication does not always hold — see [13] and [14].

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

This definition differs slightly from the one introduced by Shapiro who considered a function in the opposite direction, i.e. a function $d : \mathbb{N} \to S$. However, for our purposes it will be more convenient to utilise the function σ defined as above.

By the standard notation and standard numerals of $\mathbb{N}$ we are referring to the usual decimal notation and its numerals, i.e $\overline{0}$, $\overline{1}$, $\overline{2}$, ... When we are referring to numerals rather than numbers, we put bars over them. E.g. $\overline{0}$ is the standard decimal numeral for the number 0. However, in a non-standard notation it may represent a different number, or not be a valid numeral at all.

We need to define the notion of computability of a function in a given notation.

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

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

If f^σ is computable, then we shall say that f is computable in (S, σ).

If instead of a function f we consider a set A or a relation R, then its computability means computability of its characteristic function. Values of such functions $FALSE$ and $TRUE$ can be identified with numbers 0 and 1.

Definition 3 *Let (S, σ) be a notation for $\mathbb{N}$ 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.

Theorem 4 *There exists a notation (S, σ) of $\mathbb{N}$ in which the ordering $<$ is computable, but the successor function is not computable.*

Proof.

This proof is going to be carried out in two parts. First, we are going to construct a notation in which the set of all numerals is computably enumerable, but not computable. Then, we are going to show how to modify it to have the set of all numerals computable, while at the same preserving computability (or the lack thereof) of all the functions and relations fro the original notation.

Let $A \subset \mathbb{N}$ be a computably enumerable but not computable (in the standard notation) set of natural numbers. We first construct (T, τ) which satisfies all the requirements from the definition of a notation except one that T should be computable.

Let:

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

Note that T is computably enumerable. Let $(\alpha_n)_{n\in\mathbb{N}}$ be a computable enumeration of T without any repetitions. Now let τ assign numbers to numerals in an injective way according to the following ordering:

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

Ordering is clearly computable in this notation (for the time being, we ignore that strictly speaking this is not a valid notation due to T not being computable). We want to show that successor is not computable in it. Suppose to the contrary that it is. Then A is also 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 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 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 $i, j \in \mathbb{N}$:

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

and

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

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 the same theorem. The method used in this proof will later be generalised to show other results.

Theorem 5 *There exists a notation (S, σ) of $\mathbb{N}$ in which the ordering $<$ is computable, but the successor function is not computable.*

Proof.

Informal description We are going to construct an infinite computable sequence of numerals $\alpha_0, \alpha_1, \ldots$, 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 construction of the sequence should guarantee that no algorithm computes the successor. Since each algorithm 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 algorithm e there is such numeral α that if e stops with α as its input, then its output β satisfies the following condition:

$$\sigma(\alpha) + 1 \neq \sigma(\beta).$$

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

Construction Let S be an infinite computable set of numerals over a finite alphabet and $(\zeta_n)_{n\in\mathbb{N}}$ — a computable enumeration of S. Let $(e_i)_{i\in\mathbb{N}}$ be a computable enumeration of all algorithms, each computing some partial function from S to S.

In the 0th stage of the construction let $A_0 = (\zeta_0)$ and $L_0 = ((e_0, \zeta_0))$ (it means that we shall run the 0th algorithm on numeral ζ_0).

Suppose we have already performed n stages of the construction. Let $A_n = (\alpha_0, \alpha_1, ..., \alpha_k)$ and let $L_n = ((e_0, \alpha_{i_0}), (e_1, \alpha_{i_1}), ..., (e_m, \alpha_{i_m}))$, where $0 \leq i_0, i_1, ..., i_m \leq k$.

There are two possible cases:

1. At least one algorithm e_l stops with its assigned numeral α_{i_l} supplied as the input in at most n steps.

2. None of the algorithms $e_0, ..., e_m$ stops in at most n steps with its assigned numeral supplied as the input.

In the first case, let e_l be an algorithm which stops with the corresponding numeral supplied as its input. Then let β be the numeral returned as the output and let $\beta' = \zeta_t$, where t is the least index not utilised in $A_n \cup \{\beta\}$. Since it is our aim to ensure that the algorithm under consideration is not an algorithm for a successor, we do one of the following depending on the situation:

1. if β is already in A_n and it appears on this list directly behind α, then let $A_{n+1} = (\alpha_0, ..., \alpha, \beta', \beta, ..., \alpha_k)$,

2. if β is already in A_n and but it does not appear on this list directly behind α, then let $A_{n+1} = A_n$,

3. if β does not appear in A_n and α is the last element there, then let $A_{n+1} = (\alpha_0, ..., \alpha_k, \beta', \beta)$,

4. if β does not appear in A_n and α is not the last element there, then let $A_{n+1} = (\alpha_0, ..., \alpha_k, \beta)$,

After that, let $L_{n+1} = L_n \setminus \{(e_l, \alpha)\}$. The above procedure needs to be performed for each such e_l.

In the second case, let β be the numeral with the least index not utilised in A_n and let e be the unused algorithm with the smallest index. Then add β at the end of A_n and (e, β) at the end of L_n. Thus we have constructed A_{n+1} and L_{n+1}. This ends the construction.

Verification Let:

$$A = \lim_{n\to\infty} A_n.$$

The limit in the above definition is understood in the following way:

$$\lim_{n\to\infty} A_n = A \Leftrightarrow \ \forall_i \, \exists_{n_0} \forall_{n>n_0} \ A_n(i) = A(i),$$

where $A_n(i)$ and $A(i)$ denote the i-th element of A_n and A, respectively.

We define σ as follows: $\sigma(A(i)) = i$, for all $i \in \mathbb{N}$.

This is a notation for $\mathbb{N}$.

The set of numerals S is computable by definition. Every numeral from S is included in the sequence A because the second of the two cases happens infinitely many times. Suppose to the contrary that it happens only finitely many times. Then after a certain point, in each step we would always need to follow instructions for the first case which include, among others, removing a certain element from L_n. It follows that after finitely many steps L_n would be empty and in the following step we would need to follow instructions from the second case (no algorithm would be able to stop because the list of algorithms would be empty). This leads to a contradiction.

We also need to show that A is a well-defined sequence of numerals, i.e. that for each numeral in this sequence there are only finitely many numerals in front of it. That is the case because we always add new numerals at the end of the sequence unless an algorithm e with input α stops and returns β as the output while β has already been on the list A_n directly behind α. However, only finitely many numerals can be added this way to the sequence in front of any given numeral. This is because every numeral can be assigned to only one algorithm and no algorithm is assigned to new numerals added to this sequence via method described in the first case.

In (S, σ) the ordering is computable because we are able to generate approximations of A by performing the algorithm described above. We have also ensured that once two numerals appear on the list, their order is never reversed. Thus, in order to compare numbers represented by two numerals, we perform the algorithm until both these numerals appear on the list and then we check which of them comes earlier.

However, the successor function is not computable in (S, σ) because by construction of A we have ensured that none of the algorithms computes this function.

■

A similar theorem 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. Clearly, the successor function is a member of the class of neither almost constant nor almost identity functions. Theorem 6 is a generalisation of Theorem 5 to functions from this class.

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 6 *For every $f : \mathbb{N} \to \mathbb{N}$ which is neither almost constant, nor almost identity, there exists a notation (S, σ) for $\mathbb{N}$ 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 function that is neither almost constant nor almost identity.

We construct a computable infinite sequence of numerals (without repetitions), each denoting (via σ) its position in the sequence. Thus we ensure that $<$ is computable in this notation. Along the way, we discard each algorithm that could possibly compute f by performing appropriate manipulations on the sequence if an algorithm stops with the numeral assigned to it and supplied as the input. We begin with an incomplete description of the construction to help the reader grasp how it is structured.

Outline of the construction Let $\zeta_0, \zeta_1, \ldots$ be a computable enumeration, without repetitions, of an infinite computable set S of numerals over a finite alphabet. Let $\varphi_0, \varphi_1, \ldots$ be a canonical enumeration of all partial recursive functions from S to S.[2]

The construction proceeds in stages. At each stage n we define three finite objects A_n, L_n and C_n. A_ns ultimately define the entire representation (S, σ) whereas L_ns and C_ns help us in ensuring that no algorithm computes f in (S, σ).

For each n, A_n will always be a finite list of numerals (with no repetitions). Eventually, $A_0, A_1, \ldots$ will be a non-decreasing in length (i.e. $|A_n| \leq |A_{n+1}|$, for all n) and unbounded in length (i.e. for all $l \in \mathbb{N}$ there exists $n \in \mathbb{N}$ such that $|A_n| \geq l$) sequence of lists of numerals. Each numeral $\alpha \in S$ will at some stage n enter the list A_n and remain in all subsequent lists A_k,

[2]Such a enumeration can be obtained, for example, by setting $\varphi_n = \lambda x[c\psi_n c^{-1}]$, where $\psi_0, \psi_1, \ldots$ is any canonical enumeration of partial recursive functions from $\mathbb{N}$ to $\mathbb{N}$ and $c : \mathbb{N} \to S$ is a computable bijection.

$k > n$. Once a numeral enters a list, it may happen that its position in subsequent lists changes. However, our construction will be crafted in way that each numeral eventually reaches its final and unique position, i.e. for each numeral α there exists a position i such that $A_n(i) = \alpha$ for all but finitely many n. If we define $\lim_n A_n^\alpha$ to be that position, we can define our notation (S, σ) by setting $\sigma(\alpha) = \lim_n A_n^\alpha$.

At each stage n, L_n is a finite set of pairs $(e, \alpha) \in \mathbb{N} \times S$. If we have $(e, \alpha) \in L_n$, we say that the algorithm e, or a partial recursive function φ_e, is assigned to α at stage n. If e is assigned to α at stage n, it remains assigned to it at subsequent stages $t > n$ until we find out that $\varphi_{e,t}(\alpha) \downarrow = \beta$, for some $\beta \in S$. Once that happens, say at stage k, we put $f^\sigma(\alpha) \neq \beta$ into C_k. We say that $f^\sigma(\alpha) \neq \beta$ is active at stage n if $f^\sigma(\alpha) \neq \beta \in C_n$. Now, given an active condition $f^\sigma(\alpha) \neq \beta$ at stage n, we say that $f^\sigma(\alpha) \neq \beta$ is satisfied at stage n if there exist $i, j \in \mathbb{N}$ such that $A_n(i) = \alpha$, $A_n(j) = \beta$ and $f(i) \neq j$. The idea is that, since we know that $\varphi_e(\alpha) \downarrow = \beta$, we wish to modify the representation under construction so that $f(\sigma(\alpha)) \neq \sigma(\beta)$. Since the denotation of a numeral is identified with its (ultimate) position in lists A_n, we have to move α and β to the positions i, j such that $f(i) \neq j$.

Note that once a condition has been satisfied some stage, it is possible that our future attempts of satisfying some other conditions will ruin our previous work. This follows from the fact satisfying a condition $f^\sigma(\alpha) \neq \beta$ at stage n usually implies changing the position of α or β in the list A_n which in turn implies that all numerals that come after α or β will change their positions as well. However, if for some α', β' coming after α, β in A_n there is a condition $f^\sigma(\alpha') \neq \beta'$ that has been satisfied at some stage $< n$ then there is no guarantee that $f^\sigma(\alpha') \neq \beta'$ is still satisfied at stage n since for the new positions i', j' of α', β' respectively, it might be that $f(i') = j'$.

The above reasoning shows that once a condition 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, i.e. for each $f^\sigma(\alpha) \neq \beta$ in $\bigcup_n C_n$ we want to have a stage s such that for all $t \geq s$, $f^\sigma(\alpha) \neq \beta$ is satisfied at stage t. As we shall see in the main construction and its verification, there is a desirable way of satisfying all active conditions at subsequent stages that leads to the ultimate satisfaction of all conditions.

Construction In the 0th stage of the construction let $A_0 = (\zeta_0)$ and $L_0 = ((e_0, \zeta_0))$ (it means that we shall run the 0th algorithm with the numeral ζ_0 taken as input). The list of conditions is empty at the beginning.

Suppose we have already performed n stages of the construction. Let $A_n = (\alpha_0, \alpha_1, ..., \alpha_k)$ and let $L_n = ((e_0, \alpha_{i_0}), (e_1, \alpha_{i_1}), ..., (e_m, \alpha_{i_m}))$, where $0 \leq i_0, i_1, ..., i_m \leq k$.

There are two possible cases:

1. At least one algorithm e_l stops with its assigned numeral α_{i_l} taken as input in at most n steps.

2. None of the algorithms stops in at most n steps with its assigned numeral taken as input.

In the first case, for every (e_l, α) from the list such that an algorithm stops with its corresponding numeral as the input, let β be the numeral returned as the output.

Also, for each new algorithm that stops, add the following condition to the list:

$$f^\sigma(\alpha) \neq \beta.$$

We say that numerals α and β appear in this condition.

After that, remove each such (e_l, α) from L_n.

We are going to ensure that all conditions from the list are satisfied in each step. Every time we do this, we go through all numerals in A_n, one by one, in the same order in which they are situated there. If numerals α and β appear in a certain condition, we modify the sequence A_n only when we reach the later one of them (according to their position in A_n). We shall satisfy each condition by adding new numerals to the sequence (if necessary), either behind both α and β, or directly in front of the one that comes later. This will help us avoid a situation in which we again spoil a condition which has already been satisfied.

For each of the conditions, we ensure that it is satisfied in the following way:

1. if $\alpha = \beta$ is on the i-th position, then we need to find such $i' > k$ that $f(i') \neq i'$. Then move α to such position and fill all positions between i and i' with unused numerals. This is possible since f is not an almost identity function.

2. if α is in in A_n on the i-th position, β is already on the j-th position and $i < j$, then add new numerals directly in front of β to push β to such position $j' > k$ that $f(i) \neq j'$. This is possible since f is not an almost constant function.

3. if α is in in A_n on the i-th position, β is already on the j-th position and $i > j$, then add new numerals directly in front of α to push α to such position $i' > k$ that $f(i') \neq j$. This is possible since f is not an almost constant function.

4. if α is in in A_n on the i-th position and β does not appear in A_n at all, then put β on such position $j > k$ that $f(i) \neq j$ and fill all empty positions between α and β with unused numerals. This is possible since f is not an almost constant function.

Once all conditions have been satisfied, we have obtained a new sequence which shall be called A_{n+1}.

In the second case, let β be the numeral with the smallest index not utilised in A_n and let e be the unused algorithm with the smallest index. Add β at the end of A_n and (e, β) at the end of the L_n. Thus we have constructed A_{n+1} and L_{n+1}. This ends the construction.

Verification Let:

$$A = \lim_{n \to \infty} A_n.$$

The limit in the above definition is understood as in the previous proof. We define σ as follows: $\sigma(A(i)) = i$, for all $i \in \mathbb{N}$. This is a notation for $\mathbb{N}$.

The set of numerals S is computable by definition. Every numeral from S is included in the sequence A because in each step we add at least one numeral to the sequence.

Also, A is a well-defined sequence of numerals, i.e. that for each numeral in this sequence there are only finitely many numerals in front of it. That is the case because, as we are going to show, all numerals from each initial segment of sequence A have their positions fixed after a certain number of steps.

Note that each numeral is added to the sequence A either as the input, or the output of a certain algorithm, or as a 'filler' numeral put between other numerals to push them to certain positions within the sequence. We want to

show that, once a numeral λ has been added to the sequence, only finitely many numerals can later be added in front of λ.

Let λ' be a numeral added to A when λ is already there. If λ' is added as the input or the output of a certain algorithm, then it must be added behind λ. The only situation when we need to add new numerals in front of λ is when they are filler numerals — but for each λ this can only be the case finitely many times. Note that whenever we add such filler numerals with α as the input and β as the output, we do so either behind both α and β or directly in front of the one which comes later (in terms of position in the sequence). Hence we only need to add filler numerals in front of λ if there is α in front of λ which was added to the sequence prior to λ and was assigned as the input to one of the algorithms. But there can only be finitely many such numerals α.

Therefore, at a certain stage of the construction, each numeral occupies a fixed position in the sequence which does not change at any later stage. This also guarantees that all conditions from the list are satisfied in (S, σ).

The ordering is computable in (S, σ) because we are able to generate approximations of A by performing the algorithm described above. We have also ensured that once two numerals appear on the list, their order is never reversed. Thus, in order to compare numbers represented by two numerals, we perform the algorithm until both these numerals appear on the list and then we check which of them comes earlier.

However, f is not computable in (S, σ) because by construction of A we have ensured that none of the algorithms computes this function.

■

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

This theorem can be strengthened even further.

Theorem 7 *For every $f : \mathbb{N} \to \mathbb{N}$ which is neither almost constant, nor almost identity, 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 computable.*

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 $\prec$.

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 the previous theorem, the only differences being that this time the position of each numeral in the constructed sequence determines its denotation according to the ordering $\prec$, and that all the computations are carried out within (T, τ), rather then within the standard notation.

■

We would like to present another theorem proved using a similar method:

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

Proof.

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 This procedure generates a sequence containing all numerals because in each step we add to it two numeral 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.

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.

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).

■

3 Summary

In this paper we have shown several result regarding relations between computability of various functions in different notations for natural numbers. Our main result is that computability of any ω-type ordering does not imply computability of any unary function which is not almost identity or almost constant.

Further questions of a similar kind can be asked. Many of them have been answered in [13], [14] and [15]. In particular, in [14] 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[3] 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.

[3]Or, even more generally, representations, as defied in section 1 of this paper. As a reminder for the reader: a representation, unlike a notation, allows the function σ to be non-injective

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.

[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] 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.

[8] 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.

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

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

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

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

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

[14] 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.

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

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