Computability and Data Types

Newcomb Greenleaf
Department of Computer Science
Columbia University, New York, N. Y. 10027
newcomb@cs.columbia.edu

Abstract

The concept of a non-computable function has been valuable in showing absolute limits to our power to compute. But it is problematically oxymoronic, particular in computer science classes, since functions are generally understood algorithmically, as objects which can be computed. This paper presents an alternative explanation for functions usually regarded as non-computable, based on a more careful examination of the data types of domain and range. The limits of computation are still revealed, and additional algorithmic consequences are found. But the confusion inherent in the self-contradictory phrase "non-computable function" is avoided.

1. Background

Let **N** denote the set of positive integers (which we shall simply call **integers**). For concreteness we shall work mainly with functions from **N** to **N**, and denote the set of all such functions by $\mathbf{N}^{\mathbf{N}}$. Unless modified by "partial" the term **function** denotes a total function (defined for all $n \in \mathbf{N}$), and unless modified by "non-computable" it refers to something given by an algorithm. We use the simple and intuitive terms *decidable* and *enumerable* instead of the more traditional "recursive" and "recursively enumerable." A set of integers is **decidable** if there is an algorithm for deciding membership, and **enumerable** if there is an algorithm for listing its members.

The pioneering researches of Turing and Church in the1930s greatly enlarged our understanding of the nature of computation and functions. In particular, they provided three basic new insights:

1. **Universality.** There exist simple formal languages (such as Turing machines or recursive functions) in which, apparently, all functions can be expressed. This insight is known as the Church-Turing Thesis.

2. **Partiality.** The most simple concept is that of a partial function, a function defined only on a subset of the set of integers. To know that a function is total we must prove it to be so, and there is no general method of deciding whether a partial function is total.

3. **Undecidability.** The domain of a partial function need not be a decidable set (though it is necessarily enumerable).

The latter two insights are consequences of the undecidability of the halting problem. All three insights are fundamental to out understanding of computation, and seem well beyond controversy. There is a fourth fundamental consequence of the researches of Church and Turing which is problematic and controversial in computer science education, though that controversy is rarely mentioned in texts (however, see Minsky's footnote on p. 160 of [7]):

4. **Non-computability.** There exist non-computable functions. In fact, almost all functions (in the sense of cardinality) are non-computable.

Church and Turing deduced the existence of non-

computable functions from universality, which allows all (computable) partial functions to be listed (with the total functions being a non-decidable subset of this countable list) and from Cantor's theory of higher cardinality, which asserts on the basis of the diagonal method that there are "uncountably many" functions from N to N. Later, specific non-computable functions were found, the most interesting being the busy beaver function.

My paper at the last SIGCSE Symposium contained a critique of the argument that higher cardinality implies the existence of non-computable functions. Those remarks will be slightly amplified at the end of this paper. But our primary concern here is to give a new interpretation for specific non-computable functions, such as the busy beaver function. The heart of our interpretation consists of a careful analysis of the data types of the domain and range for such functions.

We see two principal advantages over the standard interpretation. First, the notion of a non-computable function is almost self-contradictory. We first tell our students that functions take an input and compute an output. But then we tell them that nevertheless *most* functions do not do this at all. What, one might ask, is the data type of a non-computable function? It is not satisfactory, in a computing course, to invoke the authority of Zermelo-Frankel set theory and Cantor's mysterious universe of higher infinities. An algorithmic explanation is needed. Since there are evident advantages of simplicity and unity in defining functions in terms of algorithms, we shall take the stand that functions are, by definition, computable, and then test those phenomena which are standardly taken as evidence for the existence of non-computable functions, to see if we need to yield any ground.

Second, the explanation which we shall give of the busy beaver function illuminates important algorithmic issues. In particular, it tells us interesting facts about complexity in the context of Turing machines.

Given a putative function f, we do not ask "Is it computable?" but rather "What are natural data types for the domain and range?" This question will often have more than one answer, and we will need to consider both restricted and expanded domain/range pairs. An attempt to pair an expanded domain with a restricted range will lead to the conclusion that the function in question is "non-computable."

2. The Busy Beaver Function.

The busy beaver phenomenem concerns Turing machines (TMs) whose tape alphabet consists of a single non-blank symbol "*". A **beaver** is a TM which, when started on a blank tape, halts and computes an integer, known as its productivity. Two conventions are commonly used for what counts as the computation of an integer. The more restrictive requires that the machine halt on the leftmost * of a contiguous block on an otherwise blank tape. The less restrictive requires only that the machine halt and takes as productivity either the number of *'s on the tape or the number of steps of the computation. A k-state beaver is **busy** if, among all TMs with k states, it has greatest productivity. It does not matter which convention is taken, beavers turn out to be extremely busy. Already Rado had proved the following [8]:

Busy Beaver Theorem. Let f be any (total) Turing-computable function. Then there is an integer n such that for all integers $k \geq n$ there is a k-state beaver with productivity greater than $f(k)$.

An extremely careful proof is given in Chapter 4 of the text [1]. It will be sketched in the next section in a slightly different context. If we define the **busy beaver function** bb by taking $bb(k)$ to be the maximum productivity of any k-state beaver, then the theorem shows that bb grows faster than any Turing-computable function. Hence, under the Church-Turing Thesis, it appears that bb is a non-computable function. (I prefer the alternative interpretation given in the next section.)

But Rado's theorem gives no hint of the extraordinary complexity of computations performed by extremely small machines. While k-state busy beavers have been found for $k \leq 4$, computer searches are continually finding busier and busier 5-state beavers. In

recent months a 5-state machine which halts with 4,098 symbols on the tape after running for 23,554,760 steps has been announced![1] For descriptions of this work, we particularly recommend Brady's fascinating article [2] and the entertaining account in the *Scientific American* column of Dewdney [5].

3. Reinterpreting the Busy Beaver Function.

The busy beaver function *bb* becomes computable when its domain and range are properly defined. When the domain is taken to be N, the range will be the set of "weak integers," a superset of N. The standard proof then demonstrates that *bb* grows faster than any *integer-valued* function.

To determine the proper data type for $bb(k)$, consider what can be done by a suitable universal Turing machine. Given an input k, the UTM can first enumerate all k-state TMs. Then it can proceed to execute each k-state TM for longer and longer periods, starting each from a blank tape. Whenever a beaver is found, its productivity is placed on an output tape. We obtain $bb(k)$ as an enumerable set of integers, of cardinality bounded by the (very large) number of TMs with states $\{1..k\}$.

Hence we define a **weak integer** to be an enumerable set X of positive integers which contains at least one and at most B elements, for some integer B. Intuitively, a weak integer X is an approximation from below, and every element $x \in X$ establishes a lower bound. It is crucial to understand that while we are given a bound B on the number of elements in a weak integer X, we do not necessarily have any bound on the values of these elements. Clearly $bb(k)$ can be regarded as a weak integer.

Keeping in mind that weak integers approximate from below, it is natural to define equality and order on the collection of all weak integers as follows. For weak integers X and Y:

- $X \le Y$ means $(\forall x \in X)(\exists y \in Y)(x \le y)$
- $X = Y$ means $(X \le Y) \wedge (Y \le X)$
- $X < Y$ means $(\exists y \in Y)(\forall x \in X)(x < y)$

By associating each integer n with the singleton set $\{n\}$ the integers become a subset of the weak integers. Clearly a weak integer X equals an integer x if and only if x is the maximum element of X.

Hence there are really two busy beaver functions. Rather than extend the range to W, the set of weak integers, we can shrink the domain to D, the set of integers at which *bb* takes integer values (D contains at least $\{1..4\}$.

$$\begin{array}{ccc} \mathbf{N} & \xrightarrow{bb} & \mathbf{W} \\ \cup & & \cup \\ \mathbf{D} & \xrightarrow{bb} & \mathbf{N} \end{array}$$

In this context, the usual arguments [1] now prove:

The Busy Beaver Theorem. Let f be any total Turing-computable function from N to N. Then there is an integer n such that

$$bb(k) > f(k)$$

for all $k \ge n$.

That is, the busy beaver function grows faster than any total *integer-valued* function.

Proof: Let f be a total, integer-valued function which is computed by a Turing machine T with n_T states. It is no loss of generality to assume that the function f is strictly increasing. While we don't generally know the busiest beaver, we do know that beavers can be very busy indeed, and this easily allows us to find a beaver M with n_M states that computes a number m larger than $n_M + n_T$. The composite machine MT, which has $n = n_T + n_M$ states, computes $f(m)$ from an empty tape, so $bb(n) \ge f(m) > f(n)$. Further, it is easily seen that this can be done for all $k \ge n$. *Q. E. D.*

This theorem confirms our intuition that the complexity of a computation is incomparably more sensitively linked to the size of the machine than to the size of the input. There can be no universal total machine

[1]Note that these are 5-tuple machines, which simultaneously print and move. Other authors, like [1], work with 4-tuple machines which can either move or print (but not both at once). A 5-tuple machine with 5 states will generally convert to a 4-tuple machine with 8 or 9 states.

which computes all (total) functions from **N** to **N**; no single machine can keep up with a sequence of ever larger machines. Note that, since we do have a universal machine for partial functions, this implies the unsolvability of the halting problem, for if we could decide the halting problem then we could carry out a brute force computation of $bb(k)$ as an integer by running all machines with k states which halt when started on a blank tape.

The theorem also shows that we can obtain faster growing functions by relaxing the data type of the range. Functions to the weak integers can grow faster than functions to the integers. Hence the weak integers cannot be identified with the integers, and this requires that we use intuitionistic rather than classical logic. Of course there are other good algorithmic reasons for preferring such a logic [4, 3]. For a general discussion of intuitionistic extensions, see [9].

4. The Cantor Diagonal Algorithm

We shall consider the Cantor diagonal method (which we will call the diagonal algorithm) in the context of the set $\mathbf{N}^{\mathbf{N}}$ of all integer valued functions. The algorithm takes as input a sequence of such functions $\{F_k(n)\}$ and produces as output a function G different from each F_k. It was Cantor's genius to notice that this is achieved if we simply "go down the diagonal" and construct G by a simple rule such as

$$G(n) = F_n(n) + 1.$$

As long as the functions F_k are total, this procedure is wholly algorithmic, and one implication is similar to that drawn from busy beavers: there does not exist a universal total machine. A single fixed Turing machine which takes two integer inputs k and n cannot, by fixing one input, imitate the behavior of an arbitrary machine which takes one integer input, when all functions are required to be total. (This is hardly surprising, since Cantor also showed that two integers are really no better than one.)

The diagonal algorithm is commonly used to point to the existence of non-computable functions in two different ways. It is used to directly construct specific non-computable functions. Used indirectly, it is the source of the theory of infinite cardinal numbers, which seems to imply that almost all functions are non-computable. We shall examine the direct argument here and briefly consider cardinality in the next section. The direct construction of a non-computable function by the diagonal algorithm is carried out with great care in Chapter 5 of [1]. The basic idea is very simple. The collection of all Turing machines which compute partial functions is arranged in a list $\{F_k\}$ so that $F_k(n)$ represents the value computed by the k-th TM when given input n. However, because the functions are partial, we must modify the diagonal algorithm:

$$G(n) = \begin{cases} 1 & \text{if } F_n(n) \text{ is undefined} \\ F_n(n) + 1 & \text{otherwise} \end{cases}$$

Certainly G is a function distinct from every Turing computable function, and it is very tempting to say that $G(n)$ is an integer. Since it is an integer if $F_n(n)$ is defined or is undefined, we might conclude that logically it *must* be an integer. But it is certainly not an integer in any computational sense, since we have no general way of finding its value (indeed, here lurks another proof of the undecidability of the halting problem). So the question remains, what is the data type of $G(n)$? Again, we must describe the proper superset of **N**. Warning: this definition may seem artificial and even paradoxical to those unused to intuitionistic logic. But the artificiality really resides in the application of the diagonal algorithm to a sequence of partial functions.

A **pseudo-integer** is a set X of integers satisfying:

- X contains at most one integer (i.e. if $x \in X$ and $y \in X$, then $x = y$),
- X is non-empty (in the sense that it is contradictory that $X = \emptyset$).

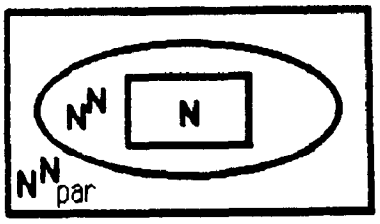
We should emphasize that X is not assumed to be enumerable. Let **P** denote the set of all pseudo-integers. If we identify each integer m with the singleton set $\{m\}$, then $\mathbf{N} \subset \mathbf{P}$, and G, as defined by the diagonal algorithm, is a (computable) function from **N** to **P**. If we want G to take integer values, then the domain must be restricted to the set of integers n for which $F_n(n)$ is either defined or

undefined. This set, while it has empty complement, cannot be identified with N (again a paradoxical situation for those unused to the algorithmic niceties of intuitionistic logic).

5. Cardinality as Shape.

Cantor argued that the diagonal algorithm showed that the set N^N was larger than the set N of natural numbers. It is this last step which introduces into mathematics a supposed universe of non-algorithmic functions. We might well wonder how so simple an algorithm could transcend the computable. Cantor did indeed show that there is a fundamental difference between the sets N^N and N, but this difference can be understood not as a quantitative difference, but as a difference of quality or structure. Rather than call the set N^N uncountable, it might better be called **productive**, because there are very powerful methods for producing elements of N^N, in particular for producing an element outside of any given sequence in N^N [6]. This use of the term productive, taken from recursive function theory, grounds our understanding in algorithmic reality rather than idealistic fantasy.

It follows, of course, that there is no surjection from N to N^N. Let N^N_{par} denote the set of all partial binary functions on N, with intensional equality. Then, under the assumptions of the Church-Turing Thesis, there is indeed a bijection from N to N^N_{par}. Since N^N is a subset of N^N_{par}, this might be considered as evidence that N is *larger* than N^N, were one inclined to make a quantitative comparison between them. Cantor, and most mathematicians after him, considered sets as "mere collections of elements," which could differ only in quantity. We do not find this position algorithmically intelligible, since the extra structure of N^N plays an essential role in the diagonal algorithm.



Perhaps shape is a better metaphor than size for the difference between N and N^N which is revealed by the diagonal algorithm. We naturally have $N \subset N^N \subset N^N_{par}$, but N and N^N_{par} have the same shape, which is distinct from that of N^N.

6. Conclusions.

It is often felt that the existence of non-computable functions shows that mathematics necessarily transcends the algorithmic. I have tried to show that this is not so, that the phenomena commonly connected with non-computability can better be understood in purely algorithmic terms.

Thanks to Matthew Kamerman for his insistent curiosity about busy beavers and for suggesting shape as a good metaphor for cardinality.

References

1. Boolos, G. S. and R. C. Jeffrey. *Computability and Logic*. Cambridge University Press, 1980.

2. Brady, A. H. The busy beaver game and the meaning of life. In *The Universal Turing Machine: a Half-Century Survey*, Herken, R., Ed., Oxford University Press, 1988.

3. Clarke, M. R. B. and D. M Gabbay. An intuitionistic basis for non-monotonic reasoning. In *Non-Standard Logics for Automated Reasoning*, Smets, P. et. al, Ed., Academic Press, 1988.

4. Constable, R. L. et al. *Implementing Mathematics with the Nuperl Proof Development System*. Prentice-Hall, 1986.

5. Dewdney, A. K. "Computer Recreations". *Scientific American 252* (April, 1984), 20-30. Reprinted in *The Armchair Universe*, Freeman, 1988, 160-171..

6. Greenleaf, N. Liberal constructive set theory. In *Constructive Mathematics*, Richman, F., Ed., Springer Lecture Notes in Mathematics, Vol. 873, 1981.

7. Minsky, M. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

8. Rado, T. "On non-computable functions". *Bell Sys. Tech. Journal* (1962), 887-884.

9. Troelstra, A. S. "Intuitionistic extensions of the reals". *Nieuw Arch. Wisk. 28* (1980), 63-113.