Bringing Mathematics Education Into the Algorithmic Age

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

Abstract

We began from the observation that most of our students find *algorithms* easy and natural and *proofs* difficult and obscure, and are totally unaware of the close relationship between algorithms and proofs. This observation led to the hypothesis that part of the problem lay in the fact that the students had been born into the algorithmic age, which their mathematics courses had largely yet to enter. This paper explores various ways in which mathematics courses can be made more algorithmic, both in style and in content. Particular attention will be paid to the term *non-computable function*, which will be seen as oxymoronic. An algorithmic explanation will be developed, particularly for the *busy beaver function*. We shall also give an algorithmic analysis of *Cantor's diagonal method*.

1. Learning Mathematics in the Algorithmic Age

The observations that follow are based on my own experience in teaching a variety of courses involving both theory and programming at Columbia University, on the comments (and complaints) of several of my colleagues about our students, and on numerous anecdotal reports from teachers at other institutions.

Our good students are very good algorithmists. They find the concept of an algorithm a natural one and delight in understanding the subtleties of intricate algorithms. When they write programs, they take pride not just in the correctness and robustness of their code, but also in its intelligibility and aesthetic appearance.

With a few exceptions, these same students are not good mathematicians. After four semesters of calculus they have only the foggiest notions of mathematical analysis. They find the notion of mathematical proof uninteresting and unintelligible. While they have seen proofs by mathematical induction in a variety of courses, they are still generally unable to do

even the simplest proofs by mathematical induction.

In view of this dichotomy, it is not surprising that they see no close connection between algorithms and proofs. They are capable of giving cogent informal arguments for the correctness of their programs, but they are generally skeptical and rather fearful of the subject of program verification (which, indeed, is little practiced at Columbia). Almost never have they been exposed to the idea that mathematics can be done algorithmically or constructively. They find this idea both intriguing and astonishing.

It seems natural to conjecture that the problem lies, in part, in the failure of instruction in mathematics to adjust and develop to living in the age of algorithms. If mathematics were presented more algorithmically, then it could build on the students' understanding of algorithms, and perhaps students would take to it with more ease and enthusiasm.

After a brief discussion of the algorithm concept and the way in which it has entered into the center of our thought, I will present several suggestions on ways in which mathematics instruction can be made more algorithmic, with some focus on the first course in computer theory.

2. The Algorithm Concept

The name "the Euclidean algorithm" may give the misleading impression that the concept of an algorithm has been securely and explicitly rooted in our mathematical thought for many centuries. In fact, the word **algorithm** acquired its modern meaning only half a century ago. Unabridged dictionaries of the 1930's define *algorithm* only as an "erroneous refashioning" of *algorism*. (*Algorists* used the ten digits for computing, instead of an abacus). This art was brought to Europe by the Latin translation of a text by al-Khwarizmi, the eponymous ninth century mathematician of Baghdad. According to the new **Oxford English Dictionary** of 1989, the first modern use of the word was in the classic 1938 number theory text of Hardy and Wright, where the nomenclature "the Euclidean algorithm" was introduced. While there are doubtless earlier occurrences missed by the OED, the term could not then have been in common parlance. Now it has become so central that Knuth has proposed *algorithmics* as the best name for the discipline of computer science [20].

I shall not attempt an extended discussion of the nature of algorithms. It is a subject to which philosophers have yet to pay sufficient attention. Like all foundational terms, the algorithm concept is difficult to define. A standard discussion occurs in Knuth's *Fundamental Algorithms* [19], pp. 1-9, where algorithms are characterized in terms of five properties: *finiteness*, *definiteness*, *input*, *output*, and *effectiveness*. Many interesting insights about the algorithm concept can be found in the papers presented at the symposium honoring al-Khwarizmi, held near the place of his birth on its 1200-th anniversary [15].

Associated with the algorithm concept is a powerful new language of *algorithms and data structures*. Since Euclid, mathematicians have used algorithms, but only recently have systematic languages for algorithms been developed. And only *very* recently has an evident quorum of mathematicians, through their programming experience, become fluent in higher-level algorithmic languages. A Pascal-like pseudo-code has become a new *lingua franca* and is used in many recent texts on discrete mathematics. There has been much discussion of the proper mathematics prerequisites for computing courses. Soon we may expect to see mathematics courses with a computer science prerequisite, since students master the language of algorithms through learning to program [17].

Of course, we have had the languages of Turing machines and recursive functions for half a century. While these precise formalisms have made their mark on our thought, they are at the level of assembly or machine language, and, as Martin-Lof and others have shown us, mathematics might best be regarded as a *very high level programming language* [22].

There is a complex symbiotic relationship between *algorithm* and *proof*. In computer science proofs are used to verify algorithms. Indeed, any algorithm must be supported by some form of proof to be believed. Such proofs often consist of a very informal argument buttressed by testing of special cases, but many workers in program verification argue that a program should be a *proof that can be compiled* [2, 10].

On the other hand, mathematicians often use algorithms in their proofs, and many proofs are totally algorithmic, in that the triple

[assumption, proof, conclusion]

can be understood in terms of

[input data, algorithm, output data].

Such proofs are often known as *constructive*, a term which provokes endless unfortunate arguments about ontology.

An interesting discussion of the relation between algorithms and proofs occurs in Knuth's review [20] of Bishop's *Foundations of Constructive Analysis* [4]:

> The interesting thing about this book is that it reads essentially like ordinary mathematics, yet it is entirely algorithmic in nature if you look between the lines.

Knuth goes on to analyze the algorithmic nature of Bishop's proof of the Stone-Weierstrass theorem in great detail, even translating it into pseudocode, and noting that:

> I want to stress that the proof is essentially an algorithm; the algorithm takes any constructively given compact set X and continuous function f and tolerance ε as input, and it outputs a polynomial that approximates f to within ε on all points of X. Furthermore the algorithm operates on algorithms, since f is given by an algorithm of a certain type, and since real numbers are essentially algorithms themselves.

It seems that we are seeing the emergence of a new concept, of which proof and

algorithm are but two aspects. Michael Beeson recently put it nicely (in the context of a discussion of Prolog), [3]:

> The flow of information seems now to be logic, now to be computation. Like waves and particles, logic and computation are metaphors for different aspects of some underlying unity.

I have no good candidate for a name for this underlying unity (neither *verified algorithm* nor *constructive proof* does it justice). It should be remarked that the coming together of the concepts of algorithm and proof creates two tensions. The pull to make proofs more like algorithms is the subject of this article. I hope to discuss the reverse pull, which manifests in Prolog and logic programming generally, in a subsequent paper.

For our purposes, this superficial discussion of the relationship between algorithms and proofs will do. The important point is that the explicit notion of an algorithm has become central in mathematical thought only recently. While our students are at home with algorithms and algorithmic languages, instruction in mathematics is only beginning to adjust to the new reality. In particular, almost never do mathematics texts take advantage of the new algorithmic literacy to explain proofs in terms of algorithms.

3. Algorithmic Style

Through our experience in writing computer programs, we have gradually developed a distinctive and effective style for presenting algorithms. While this style has much in common with mathematics, it departs radically from traditional mathematical style in some ways. This is strikingly evident in the different approaches to names and symbols. Mathematicians have generally used single characters for symbols. There is good reason for this, since it leads to brevity and allows complex formulas to be written concisely. But this style demands of the reader that she remember the meanings of all of the symbols, which do not have mnemonic names, and my students generally are unable and/or unwilling to read material written in this style.

Computing also puts a high value on brevity. Good programming style dictates that procedures be fairly short, but this is not generally achieved by using single-character symbols. Rather, symbols are generally words or word fragments, chosen for mnemonic value. Brevity is achieved through procedural abstraction, by giving each procedure a relatively simple task, and using many procedures in a single program.

Good programming style also pays careful attention to the layout of the program on the page or screen. I regularly teach the Scheme dialect of LISP to novice programmers. A great deal of credit for the success of this enterprise goes to the pretty-printing mechanism of the editor which lays out the programs in a way which both illustrates the structure and is aesthetically pleasing. Here is the first substantial procedure which my students see, adapted

from [1], which uses Newton's method to compute the square root of a number x. The simple helping procedures `square` and `average` are defined elsewhere.

```
(define (sqrt x tolerance)
  (define (sqrt-iter guess)
    (define (good-enough?)
      (< (abs (- (square guess) x)) tolerance))
    (define (improve)
      (average guess (/ x guess)))
    (if (good-enough?)
        guess
        (sqrt-iter (improve))))
  (sqrt-iter 1))
```

While this is a challenging procedure for the second week of a first course, the combination of mnemonic names and pretty-printing (along with the simple syntax of Scheme) makes it accessible to students. Now consider how inaccessible the same procedure becomes when mnemonic names are replaced by single letter symbols:

```
(define (s x t)
  (define (i g)
    (define (g?)
      (< (abs (- (q g) x)) t))
    (define (p)
      (a g (/ x g)))
    (if (g?)
        g
        (i (p))))
  (i 1))
```

Had the students been presented with the latter version, which mimics the mathematical style of conciseness, their reaction would have been quite different. Instead of loving the course, they might have dropped it.

A glaring example of the failure of mathematics instruction to make the subject algorithmic occurs in the elementary differential calculus, which is still generally presented as a large collection of derivative formulas. Of course, these formulas are intended to be used as the base cases and recursive operations of a grand recursive *derivative algorithm*, which, for want of a proper language, is not made explicit (and therefore never really verified).

The derivative algorithms can be cast in two different forms. If the chain rule is made an explicit recursive operation, then there will be a large number of base case formulas, such as:

$$D(\sin x) = \cos x$$

If the chain rule is built into most derivative formulas, then the sine formula will appear as a recursive operation, corresponding to the formula

$$D(\sin u) = (\cos u) * Du$$

and there will be two base cases which tell us that the derivative of a constant is 0 and the derivative of the identity function is 1. The student of calculus is expected not merely to learn

the various formulas, but primarily to understand the operation of the recursive algorithm, which never appears explicitly.

We can hope that tomorrow's calculus texts will express the derivative algorithm explicitly in a (formal or informal) algorithmic language, as is already done in programming texts such as [1]. Here is a simple implementation of the derivative algorithm in the Scheme programming language (any programming language supporting recursion would do), adapted from [1], which takes the derivative of an expression `exp` with respect to `var`. Note that the first two cases are the only base cases. The various constructors, selectors, and predicates, such as `make-sum`, `multiplicand`, and `constant?` are defined elsewhere. This simple version handles only sums, products and sines:

```
(define (deriv exp var)
  (cond ((constant? exp) 0)
        ((variable? exp)
         (if (same-variable? exp var) 1 0))
        ((sum? exp)
         (make-sum (deriv (addend exp) var)
                   (deriv (augend exp) var)))
        ((product? exp)
         (make-sum
          (make-product (multiplier exp)
                        (deriv (multiplicand exp) var))
          (make-product (deriv (multiplier exp) var)
                        (multiplicand exp))))
        ((sine? exp)
         (make-product (deriv (arg exp) var)
                       (make-cosine (arg exp))))
        (else
         (error "Unknown expression type -- DERIV" exp))))
```

When the derivative algorithm assumes its rightful place as the *primary explicit structure* of differential calculus, there is a welcome gain in clarity, and also a subtle but profound shift of meaning. The proofs of the various derivative formulas are now part of the *verification* of the algorithm. They prove the *correctness* of an algorithm rather than the *truth* of a theorem. These considerations are amplified in [25].

We can summarize this discussion by enunciating

Four Marks of Algorithmic Style:

- **Mnemonics**: Use mnemonic names for symbols, and achieve brevity through abstraction.
- **Explicitness**: Make the algorithms explicit. Identify input and output and check the effectiveness of the algorithm. When appropriate, pay attention to questions of efficiency and feasibility.
- **Language**: Use an appropriate algorithmic language, which can be pseudocode rather than a formal programming language.
- **Primacy**: Make algorithms the primary structures when appropriate. For instance, the chapter of a calculus text dealing with differentiation could be called

The Derivative Algorithm.

We will illustrate these principles further by considering an example from the first course in computing theory. This course is traditionally called "Computability," a name which I feel is highly inappropriate, for reasons to appear below. A far better title would be "Languages, Grammars, and Machines."

I have taught Computability on numerous occasions at Columbia. The first time I used the excellent traditional text of Lewis and Papadimitriou [21]. While I found the book to be clear and comprehensive, my students found it almost completely unreadable. They would open it only when problems were assigned, and then would skim backwards from the problems to see if they could find a relevant formula. Now this is what students have been doing with mathematics texts for many years, but it is not optimal for learning. I then heard of the text by Daniel Cohen, which takes a very different approach [9]. At first I was skeptical, but after giving it a try became an enthusiastic booster of Cohen's text. True, Cohen takes 800 pages to cover what Lewis and Papadimitriou do in less than half of their 400 pages. But students found the book a joy to read, and in reading it they acquired a much better understanding than they had when I had used a traditional text, and they went on to distinguish themselves in more advanced courses using traditional texts.

The text of Lewis and Papadimitriou is written in a traditional mathematical style and displays none of the marks of algorithmic style. If we examine Cohen's book in light of the four marks, we see that he is very careful to use well-chosen mnemonic names. Indeed, the book contain virtually no traditional-looking mathematical formulas which students new to the subject find so opaque. He always presents his proofs as algorithms and continually points this out. But he does not use any special language for algorithms, preferring to present them in lengthy English descriptions. Of course, this allows the instructor to ask students to summarize various algorithms in pseudocode. And while the algorithms are explicit, they are embedded within the proofs of theorems, which remain the primary structures. So Cohen exhibits two of the four marks to a high degree.

To highlight the difference between Cohen's text and that of Lewis and Papadimitriou, I will present the beginning of a crucial proof from each text. The proof involves showing that any for deterministic finite automaton (DFA) there exists an equivalent regular expression. Lewis and Papadimitriou start like this:

> Let $M = (K,\Sigma, \delta, s, F)$ be a deterministic finite automaton; we need to show that there is a regular language R such that $R = L(M)$. We represent $L(M)$ as the union of many (but a finite number of) simple languages. Let $K = \{q_1, \cdots, q_n\}$, $s = q_1$. For $i, j = 1, \cdots, n$ and $k = 1, \cdots, n+1$, we let $R(i,j,k)$ be the set of all strings in Σ^* which drive M from q_i to q_j without passing through any state numbered k or greater.

The proof is off to an economical start. It will be completed in less than a page. But because of the density of non-mnemonic notation, and the complete lack of an algorithmic framework, few beginning students will even attempt to read it, let alone come to understand it.

In marked contrast, Cohen starts off as follows:

> The proof of this part will be by constructive algorithm. This means that we present a procedure that starts out with a transition graph and ends up with a regular expression that defines the same language. To be acceptable as a method of proof, any algorithm must satisfy two criteria. It must work for every conceivable TG, and it must guarantee to finish its job in a finite time (in a finite number of steps). For the purposes of theorem-proving alone, it does not have to be a good algorithm (quick, least storage used, etc.). It just has to work in every case.

This is hardly an economical start! But while the proof proper has yet to begin, a proper foundation has been laid. The proof itself will occupy twelve or so pages (with many illustrations). But my students read it with enthusiasm. I often get reports that they have stayed up half the night reading it and running hand simulations of the algorithm. And they retain from their reading the ability to execute the algorithm on simple machines.

4. Are All Functions Computable?

I remarked above that the name "Computability" seems to be inappropriate for the first course in computing theory. Traditionally, a high point of this course has been the demonstration of the existence of "non-computable functions." By accepting uncritically the traditional mathematical account of these phenomena, we introduce radically non-algorithmic ideas into the heart of our curriculum.

Indeed, the very concept of a non-computable function is problematic. First we explain functions to students in computational terms. A *function* from X to Y takes an element of X as input and delivers an element of Y as output. Then, a few pages or weeks later, we turn around and announce to the students that, in fact, almost all functions are non-computable! But what is a non-computable function? But if a function does not correspond to an algorithm, what can it be? There is in this context no higher court corresponding to the set theory of logical mathematics. Indeed, one thing we can say is that, for our students, the term *non-computable function* is an oxymoron which undermines their algorithmic understanding of functions.[1] 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. Our strategy will be the following. Given a putative function f, we do not ask

Is f computable?

---

[1]Following Umberto Eco we might imagine a **College of Oxymoronics** in which a course in *Non-Computable Functions* would take its place in the curriculum along side such offerings as *Tradition in Revolution*, *Democratic Oligarchy*, *Parmenidean Dynamics*, *Heraclitean Statics*, *Boolean Eristic*, and *Tautological Dialectics* [14].

but rather

What are the **proper data types** for the domain and range of f?

The latter question may have more than one natural answer, and we can consider both restricted and expanded domain/range pairs. Only if you attempt to pair an expanded domain for f with a restricted range will you come to the conclusion that f is "non-computable."

The usual argument given for the existence of non-computable functions involves Cantor's diagonal algorithm. Let us look at a typical example of how this argument is usually presented. As we will see, it is, from an algorithmic standpoint, badly flawed. I chose the following excerpt from a review by Ian Stewart not because it is unusual, but because it states the received view so well [27].

> A real number is computable if its binary expansion can be the output of a computer program. It is a theorem--at first sight surprising, but true, and even easy--that 'almost all' real numbers are not computable. For example, Turing used an argument that goes back to Georg Cantor to prove the existence of at least one non-computable real number.
>
> The proof is based on the idea that all possible computer programs, each of which must be represented by a finite sequence of digits, can be listed in order. To do this, interpret the program's defining sequence as a whole number, expressed in binary, and arrange these numbers in increasing numerical order. Now assign to each program in this list its output data, a real number expressed in binary. Run down the diagonal of this table of numbers, changing the nth digit in the nth number. The new diagonal is a number that is not on the list, which therefore corresponds to the output of no computer program whatsoever.

Note that Stewart expresses himself in a rather algorithmic fashion, using words which imply feasible actions, like *arrange*, *assign*, *run down*, and *change*. But in fact what is proposed is completely non-algorithmic. We can indeed arrange all possible computer programs producing binary sequences in an infinite list (more precisely, we can write a program which will generate as much of this list as desired), but these programs will be partial functions which may produce a binary sequence but may, at some of the places in the sequence, compute forever without producing a 0 or a 1. We cannot change the value in such a case because there is no value to change. But we cannot be sure that there is no value, because perhaps a value will be found if we just let the program run for a longer time. The undecidability of the halting problem bars the way.

But perhaps Stewart means only that we should arrange in a list only those programs which correspond to *total* functions. Then the diagonal algorithm will work without a hitch, but a problem remains: without an oracle we have no way of extracting from the list of all programs those which correspond to total functions, and hence cannot arrange the latter in a list. Again the halting problem halts our progress.

The pioneering researches of Turing, Church, and others showed that the functions defined by Turing machines (or equivalent formalisms) are typically partial, and their domains are typically undecidable (because of the undecidability of the *halting problem*). They also concluded that functions are typically non-computable, on the grounds that Turing machines can be enumerated, while functions cannot. Later, specific examples of non-computable functions were found, most notably the *busy beaver function*. We will present another interpretation of the busy beaver phenomenon, based on careful attention to the data types of domain and range, in which the busy beaver function is indeed computable. Then we will consider the Cantor diagonal algorithm and questions of cardinality.

Note that I 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 or Turing machine for deciding membership, and *enumerable* if there is an algorithm or Turing machine for listing its members.

5. The Busy Beaver Function

The busy beaver phenomenon concerns Turing machines (TMs) whose tape alphabet consists of a single non-blank symbol, say "*". A *beaver* is a TM which, when started on a blank tape, halts and computes an integer, known as its *productivity*. Several conventions are commonly used for what counts as the computation of an integer. The most restrictive requires that the machine halt on the leftmost * of a contiguous block on an otherwise blank tape. The less restrictive require only that the machine halt and take as productivity either the number of *'s on the tape, the number of steps of the computation, or some other measure of the complexity 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 [26]:

> **Rado's 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 [7]. 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 might appear that *bb* is a non-computable function.

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. Recently a 5-state machine which halts with 4,098 symbols on the tape after running for 23,554,760 steps has been announced! Note that these are 5-tuple machines, which simultaneously print

and move. Other authors, like [7], 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. For further discussion of busy beavers, we particularly recommend Brady's fascinating article [8] and the entertaining account in the *Scientific American* column of Dewdney [13].

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** which we shall define shortly. Rado's Theorem then demonstrates that *bb* grows faster than any *integer-valued* function.

To determine the proper data type for $bb(k)$, simply attempt to compute it by brute force. First list all k-state machines (the number of such machines is staggeringly large, so this is, of course, feasible only for very small k. Then run all of the k-state machines in parallel. Whenever one of them halts, determine its productivity and include it in the output. We obtain an enumerable set of integers, of cardinality bounded by the (very large) number of k-state TMs. But the process of generating this enumerable set will not halt, even though the set has bounded cardinality. Many of the TMs will never halt, and while we can weed out many obvious non-halters, there will be others whose status we will be unable to decide, so we will have to keep them running. We are led to the following definition.

Definition. A **weak integer** X is an enumerable set of positive integers which contains at least one and at most B elements, for some integer B.

Intuitively, a weak integer X represents 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. For we do not generally have access to the entire list, but only to the algorithm or TM which enumerates it. So, in concrete terms, a weak integer is an algorithm or Turing machine which enumerates a set of integers of bounded cardinality. Given k, the function *bb* produces such a TM $bb(k)$ by the finite process outlined above.

The situation here is analogous to that of real numbers. When we speak of a real number x, we generally have in mind a Cauchy sequence of rationals. But when an algorithm produces a real number x which is not rational, what it actually delivers as x is an algorithm for computing as much of the Cauchy sequence as we may wish to see [20]. Similarly, when we speak of a weak integer X, we have in mind an enumerable set of integers. But when an algorithm produces a weak integer which is not an integer, what it actually delivers is an algorithm or Turing machine for enumerating as much of that set as we may wish to see.

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:

- $X \leq Y$ means $(\forall x \in X)(\exists y \in Y)(x \leq y)$

- $X = Y$ means $(X \leq Y) \wedge (Y \leq X)$
- $X < Y$ 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 the set of weak integers, we could shrink the domain to **D**, the set of integers at which *bb* takes integer values (**D** contains at least $\{1..4\}$).

We can now deduce from Rado's Theorem:

> **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 $k \geq n$. That is, the busy beaver function grows faster than any total *integer-valued* function.

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 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 interpretation requires that we use intuitionistic rather than classical logic. Using classical logic, it can be shown that every set of integers of bounded cardinality contains a greatest element, and hence every weak integer equals an integer. The weak integers form an *intuitionistic extension* of **N** [28].

6. The 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 the implication is similar to one 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 (which 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 as in the argument of Stewart given above. Used indirectly, it is the source of the theory of infinite cardinal numbers, which seems to imply that almost all functions are non-computable. The direct construction of a non-computable function by the diagonal algorithm is carried out with great care in Chapter 5 of [7]. 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. Note that while we have a list of all Turing machines which compute partial functions, we have no way of extracting the list of those machines which compute total functions, because of the undecidability of the halting problem. 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 total function which is 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. Using classical logic, we could conclude that logically $G(n)$ *must* be an integer. On the other hand, 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 arises, what is the data type of $G(n)$? Again, we must describe the proper superset of N. {*Warning*: this definition will 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.}

Definition. A **singleton 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 $X = \varnothing$).

A similar definition is found in [11].

The function G, defined above by the diagonal algorithm, is a (computable) function from N to the set of singleton integers. If we identify each integer m with the singleton set $\{m\}$, then the singleton integers become an (intuitionistic) extension of N. If we require 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 subset, while it has empty complement, cannot be identified with N (again a paradoxical situation for those unused to the algorithmic niceties of intuitionistic logic). We should emphasize that, unlike weak integers, singleton integers are not in general enumerable. Indeed, an enumerable singleton integer is very close to being an ordinary integer and can be proved equal to an ordinary integer if *Markov's principle* is assumed (see [5], p. 63). While Markov's principle is often considered constructive or very weakly non-constructive, it has the effect of erasing the algorithmically significant distinction between algorithms which are merely known not to run forever and those for which there exists an upper bound on runtime.

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 standardly connected with non-computability can better be understood in purely algorithmic terms. The usual approach simply lumps together the busy beaver function and the function output by the diagonal algorithm as "non-computable functions." Our approach distinguishes them in terms of the very different data types of their natural ranges.

7. Cardinality as shape

Cantor argued that the diagonal algorithm showed that the set $\mathbf{N}^{\mathbf{N}}$ contained *more* elements 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.[2] Cantor did indeed show that there is a fundamental difference between the sets $\mathbf{N}^{\mathbf{N}}$ 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 $\mathbf{N}^{\mathbf{N}}$ uncountable, it might better be called *productive*, because there are very powerful methods for producing elements of $\mathbf{N}^{\mathbf{N}}$, in particular for producing an element outside of any given sequence in $\mathbf{N}^{\mathbf{N}}$ [16].

Definition. A set X is **productive** if, for every function $f: \mathbf{N} \to X$ there exists an $x \in X$ outside of $f(\mathbf{N})$.

Cantor's Theorem. The set $\mathbf{N}^{\mathbf{N}}$ is productive.

This use of the term *productive*, taken from recursive function theory, grounds our

[2]This point was perhaps first made by Wittgenstein, who wrote of the diagonal algorithm: "Our suspicion ought always to be aroused when a proof proves more than its means allow it. Something of this sort might be called a 'puffed-up proof'." ([29], p. 56)

understanding in algorithmic reality rather than idealistic fantasy.[3] Given any sequence of elements of $\mathbf{N}^{\mathbf{N}}$, we can find an element of $\mathbf{N}^{\mathbf{N}}$ outside the sequence, not because there are more functions than integers, but because of the structure of $\mathbf{N}^{\mathbf{N}}$.

Of course it follows from the diagonal algorithm that there is no surjection from $\mathbf{N}$ to $\mathbf{N}^{\mathbf{N}}$. Let $\mathbf{N}^{\mathbf{N}}_{par}$ denote the set of all partial binary functions on $\mathbf{N}$, with intensional equality. Then, under the assumptions of the Church-Turing Thesis, the set of all partial functions can be represented by the set of Turing machines, which can be listed, so there is indeed a bijection from $\mathbf{N}$ to $\mathbf{N}^{\mathbf{N}}_{par}$. Since $\mathbf{N}^{\mathbf{N}}$ is a subset of $\mathbf{N}^{\mathbf{N}}_{par}$, this might be considered as evidence that $\mathbf{N}$ is *larger* than $\mathbf{N}^{\mathbf{N}}$, 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 $\mathbf{N}^{\mathbf{N}}$ plays an essential role in the diagonal algorithm.

If we want a simple metaphor for differences such as that between $\mathbf{N}$ and $\mathbf{N}^{\mathbf{N}}$, *shape* might be better than *size*. It is then perfectly intelligible to maintain that

$$\mathbf{N} \subset \mathbf{N}^{\mathbf{N}} \subset \mathbf{N}^{\mathbf{N}}_{par}$$

where $\mathbf{N}$ and $\mathbf{N}^{\mathbf{N}}_{par}$ have the same shape, which is distinct from that of $\mathbf{N}^{\mathbf{N}}$.

8. Algorithmic logics

We have seen that if we are to take the stand that all functions are computable, we must distinguish the ordinary integers from such intuitionistic extensions as the weak integers and the singleton integers. This requires that we adopt a logic such as intuitionistic logic which allows us to make such distinctions.

Brouwer claimed that the Law of Excluded Middle was not algorithmically valid and developed intuitionistic logic as a subset of classical logic. However, there is another way to view the difference between the two logics. Since intuitionistic logic allows us to make more distinctions, it must itself contain additional distinctions. Most basically, intuitionistic logic distinguishes between a statement and its double negation, and distinguishes the constructive disjunction $A \vee B$ from the classical disjunction $\neg(\neg A \wedge \neg B)$. It is this richness of distinctions which allows the distinction between the integers and the weak integers or singleton integers.

---

[3]The fantasy here is not the theory of sets, as elegantly elaborated in the concrete confines of Zermelo-Fraenkel set theory, but rather the notion that, somewhere, there are really "more" elements of $\mathbf{N}^{\mathbf{N}}$ than of $\mathbf{N}$. It is worth remembering that Cantor lobbied the Vatican to recognize that the higher cardinals pointed the way to God [12, 18].

Looking at algebras rather than sets of truths, we can see classical logic as a *quotient* or collapsing of intuitionistic logic. From the algorithmic standpoint, classical logic does not make enough distinctions. In the logical paradigm for mathematics, the traditional test for correctness is *consistency*, and while this has an appealing simplicity, it turns out not to be sufficient for the needs of algorithms. As Errett Bishop forcefully stated: *meaningful distinctions deserve to be maintained* [6]. Similar ideas were recently expressed in [23]:

> A common misconception among mathematicians is to think of intuitionistic mathematics as "mathematics without the law of the excluded middle" (the law asserting that every statement is either true or false. From this point of view, intuitionistic mathematics is a proper subset of ordinary mathematics, and doing your mathematics intuitionistically is like doing it with your hands tied behind your back.
>
> Another more realistic viewpoint is to regard intuitionistic logic, and the mathematics based on that logic, as the logic of sets with some structure, rather than of bare sets. Traditional examples are sets growing in time (as in Kripke semantics), or sets with some recursive structure (as in Kleene's realizability interpretation), or sets continuously varying over some fixed parameter space. Universes of such sets are perfectly suitable for developing mathematics, but one is often *forced* to use intuitionistic logic.
>
> . . .
>
> In the first half of this century, the interest in intuitionistic logic and mathematics was mainly of a philosophical and foundational nature. More recently, it has become apparent that intuitionistic logic or some variant thereof is often the right logic to use in theories of computing.

Of course, from an algorithmic standpoint, there can be no such thing as a "bare" set, except possibly finite sets. Any set carries a structure by virtue of its construction. The weak integers and singleton integers can be considered as further examples of sets "growing in time."

What logic should we teach our students? Traditionally, classical logic has been the only logic used. Indeed, students have often been taught that "to prove something, assume it it false and try to derive a contradiction." From an algorithmic standpoint there could hardly be worse general advice. When intuitionistic logic has been mentioned, it has often been in the pejorative category of a "deviant logic." But intuitionistic logic makes it natural to think of proofs as programs, and on this grounds should be preferred. We have found that some sort of natural deduction system provides an excellent way to introduce students to logic with an algorithmic flavor.

Ultimately we would, of course, like our students to be literate in both classical and intuitionistic logic. Experience has shown that those who thinking classically have a very difficult time understanding intuitionistic thought. The law of excluded middle is made part

of one's subconscious thought and even questioning it seems strange. On the other hand, those who start with intuitionistic logic have no difficulty in adding excluded middle when appropriate (for example for decidable propositions). This seems to me to be a very strong argument for starting with intuitionistic logic.

An analogous situation holds in programming languages. Ultimately, of course, programmers will want to be able to use non-local control mechanisms, but experience has shown that those who start using GOTO as novice programmers develop bad habits which are very hard to erase. This analogy between local control and intuitionistic logic and non-local control and classical logic actually runs quite deep, as shown in other papers at this symposium [24].

The algorithmic spirit is diverse. We have a multitude of programming languages, programming paradigms, operating systems, etc., and this is recognized as a healthy situation in which the diversity leads to evolutionary progress (even though our students may complain about having to learn more than one programming language).

Mathematics, on the other hand, has been characterized by conformity. Physicists may still be seeking the "grand unified theory," but mathematicians seemed to have found theirs. For the past half century adherence to Zermelo-Frankel set theory and classical logic as the foundations of mathematics has been virtually unquestioned. While this unity has some advantages, it has also led to a stifling of fresh inquiry into the nature of mathematics.

Following the algorithmic spirit, we might hope that in the future mathematics will become more diverse, that students will be presented with a genuine choice among a range of foundational outlooks, and that they will learn, in particular, to view mathematics algorithmically.

References

1. Abelson, H. and Sussman, G. J. *Structure and Interpretation of Computer Programs*. M. I. T. Press, 1984.

2. Bates, J. L. and Constable, R. L. "Proofs as programs". *ACM Transactions on Programming Languages and Systems 7* (1985), 113-136.

3. Beeson, M. Computerizing mathematics: logic and computation. In Herken, R., Ed., *The Universal Turing Machine: a Half-Century Survey*, Oxford University Press, 1988, pp. 191-225.

4. Bishop, E. *Foundations of Constructive Analysis*. McGraw-Hill, 1967. [5] is a new edition.

5. Bishop, E. and Bridges, D. *Constructive Analysis*. Springer-Verlag, 1985. Revised edition of [4].

6. Bishop, E. *Contemporary Mathematics*. Volume 39: Schizophrenia in Contemporary Mathematics. In Rosenblatt, M., Ed., *Errett Bishop: Reflections on Him and His Research*, American Mathematical Society, 1985. Originally distributed as the American Mathematical Society Colloquium Lectures in 1973.

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

8. Brady, A. H. The busy beaver game and the meaning of life. In Herken, R., Ed., *The Universal Turing Machine: a Half-Century Survey*, Oxford University Press, 1988, pp. 259-278.

9. Cohen, D. I. A. *Introduction to Computer Theory*. John Wiley and Sons, 1986.

10. Constable, R. L. "Programs as proofs: a synopsis". *Information Processing Letters 16* (1983), 105-112.

11. Dalen, D. van. Singleton Reals. In *Logic Colloquium '80*, North-Holland, 1982, pp. 83-94.

12. Dauben, J., W. *Georg Cantor: His Mathematics and Philosophy of the Infinite*. Harvard University Press, 1979.

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

14. Eco, Umberto. *Foucault's Pendulum*. Harcourt Brace Jovanovich, 1989.

15. Ershov, A. P., and Knuth, D. E. (Ed.) *Algorithms in Modern Mathematics and Computer Science*. Springer Lecture Notes in Computer Science, 1981.

16. Greenleaf, N. Liberal constructive set theory. In Richman, F., Ed., *Constructive Mathematics*, Springer Lecture Notes in Mathematics, Vol. 873, 1981, pp. 213-240.

17. Greenleaf, N. "Algorithms and proofs: mathematics in the computing curriculum". *ACM SIGCSE Bulletin 21* (1989), 268-272.

18. Hallett, M. *Cantorian Set Theory and Limitation of Size*. Oxford University Press, 1984.

19. Knuth, D. E. *The Art of Computer Programming*. Volume 1:*Fundamental Algorithms*. Addison-Wesley, 1973.

20. Knuth, D. E. Algorithms in modern mathematics and computer science. In *Algorithms in Modern Mathematics and Computer Science*, Springer Lecture Notes in Computer Science, Vol 122, 1981, pp. 82-99. Revised (from ALGOL to Pascal) and reprinted in *American Mathematical Monthly* 92 (1985), 170-181.

21. Lewis, H. and Papadimitriou, C. *Elements of the Theory of Computation*. Prentice-Hall, 1981.

22. Martin-Lof, P. Constructive mathematics and computer programming. In *Sixth International Congress for Logic, Methodology, and Philosophy of Science*, North-Holland, 1982. Reprinted in *Mathematical Logic and Programming Languages*, Hoare, C. A. R. and Shepherdson, J. C. (eds.), Prentice-Hall, 1986.

23. Moerdijk, I. "Review of *Mathematical Intuitionism. Introduction to Proof Theory*". *Bull. Amer. Math. Society 22* (1990), 301-304.

24. Murthy, C. Classical Proofs as Programs: How, What, and Why. These Proceedings , 1991.

25. Myers, J. P., Jr. "The central role of mathematical logic in computer science". *ACM SIGCSE Bulletin 22* (1990), 22-26.

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

27. Stewart, I. "The ultimate in undecidability". *Nature 332* (10 March 1988), 115-116.

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

29. Wittgenstein, L. *Remarks on the Foundations of Mathematics*. Basil Blackwell, 1956. Translated by G. E. M. Anscombe.