Computable Functions in the Theory of Algorithms: Collapsing Infinity to a Point

Paul Corazza, Ph.D.

ABOUT THE AUTHOR

Paul Corazza received his Bachelor of Arts degree in Western Philosophy from Maharishi International University in 1978 and his M.S. and Ph.D. degrees in Mathematics from Auburn University in 1981 and 1988, respectively. He was awarded a Van Vleck Assistant Professorship at University of Wisconsin for the years 1987–1990. He worked in the Mathematics Department at Maharishi International University in the years 1990–95. Following a career as a software engineer, he rejoined faculty at Maharishi University of Management in 2004 and currently serves a joint appointment in the Departments of Mathematics and Computer Science. Dr. Corazza has published more than a dozen papers in Set Theory, focused primarily on the quest for providing an axiomatic foundation for large cardinals based on a paradigm derived from Maharishi Vedic Science.

ABSTRACT

According to Maharishi Vedic Science, creation arises in the collapse of the infinite unbounded value of wholeness to a point, from which creation emerges in a sequential unfoldment. By analogy, the creative dynamics found in the field of computation arise from the collapse of a vast function field—whose elements are almost entirely indescribable—to a tiny packet of functions that are actually used: the polynomial time bounded functions. The class of functions that are actually used in Computer Science arises from a much vaster class of number-theoretic functions through a sequence of "collapses" to successively narrower function classes. The vast majority of number-theoretic functions are essentially indescribable. By contrast, it is possible to know in full detail the behavior of the functions belonging to the much more restricted class of definable functions. Restricting further to the class of computable functions, it becomes possible to describe the behavior of each function in terms of a computer program; that is, one can describe how to compute each function in this class. And the narrowest class of functions—the class of polynomial time bounded functions—consists of functions that can be computed in a feasible way, efficiently enough to be of practical value in real applications. We discuss how this sequence of collapses parallels the creative dynamics of pure consciousness itself, as described by Maharishi Vedic Science, as it brings forth creation through the collapse of the abstract, indescribable infinite value of wholeness to the concrete, specific point value within its nature.

Introduction

According to Maharishi Vedic Science, all that we see in this manifest world of existence arises because of the transformational dynamics of a single field, the field of pure consciousness, the Unified Field of all the Laws of Nature (see Maharishi, 1996, pp. 252, 504). These creative dynamics arise from a fundamental movement within pure consciousness, which repeats over and over again: the infinite expanded value of this field collapses to its own point value, and then emerges into infinite diversity. What we see in the form of sequential unfoldment of life in the manifest world is an expression of unseen self-referral dynamics at the level of pure consciousness. In this introduction to Algorithms, we examine how core notions in the field of computer science arise as a kind of collapse of expanded, noncomputable mathematical domains to a highly restricted domain in which

the usual forms of computation and computational analysis become possible.

The starting point for our investigation into these various classes of functions is the simple idea of a *computer program*. A computer program can be understood as a procedure for taking certain inputs—perhaps numbers, character strings, or other types—and transforming them in a sequential manner to produce well-defined output values. This simple idea is the core notion at the root of all of today's complex and remarkable software.

This idea of a computer program is a modern-day way of expressing the mathematical notion of a function. A function, like a computer program, acts on certain inputs and outputs well-defined values. A simple example is the function that acts on the natural numbers 0, 1, 2, . . . according to the following rule: The function transforms any natural number input to the natural number that is twice as big as the input. Formally, we express this rule by the formula $f(n) = 2 * n$. Thus, for example, $f(3) = 2 * 3 = 6$ and $f(7) = 2 * 7 = 14$ — the function transforms each input to an output value twice as big.

As we will discuss in more detail later, it is not true that every function defined on the natural numbers can be computed using a computer program. This insight leads to a remarkable discovery: the field of all functions on the natural numbers is a kind of wholeness, a kind of vast field, whose contents are fundamentally indescribable. To emphasize the point, we can say that, if one were to place all natural number functions in a bucket, and choose one from the bucket at random, the probability that this function's behavior could actually be represented in a computer program is 0! That is, there is zero probability that, for a function chosen at random from the bucket, a computer program could produce exactly the same outputs as the given function on given inputs. As we will show more formally, the functions that can be computed using computer programs represent an infinitesimal speck in the vast wholeness of all number-theoretic functions.

This phenomenon provides a striking analogy to the parallel journey from the field of manifest existence that we experience with the senses back to its source in the infinitely expanded value of intelligence itself. This infinitely expanded value is truly beyond description,

indeed, beyond the intellect itself. In the Bhagavad-Gita, Lord Krishna describes this field with the words (Maharishi, 1969),

> The senses, they say, are subtle; more subtle than the senses is mind; yet finer than mind is intellect; that which is beyond even the intellect is he.
>
> –Bhagavad-Gita, 3.42

In this paper, we describe the stages through which this fundamentally indescribable field of all functions as if "collapses" to the infinitesimal package of functions that we actually use in the field of computer science and in the context of software development.

We will describe the mechanics of singling out successively smaller collections of functions until we arrive at the class of functions that will be our primary concern—the class of polynomial time computable functions.

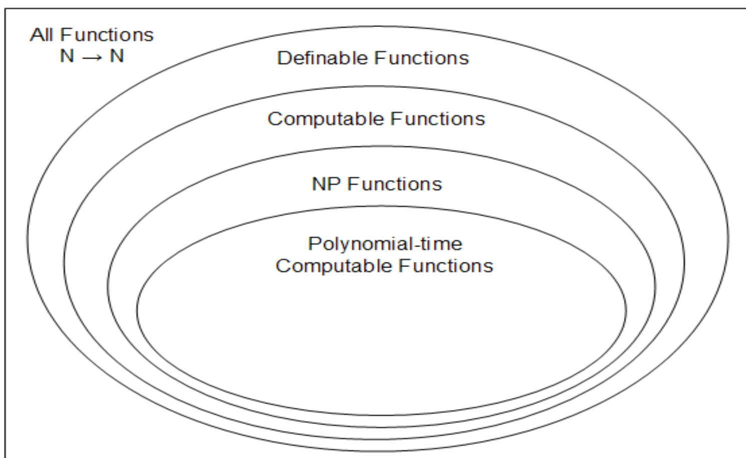
We will begin by discussing the class of *definable* functions—those functions that can be specified at all in a reasonable way. The class of definable functions can be said to emerge from the class of all number-theoretic functions in a way analogous to the emergence of the notion of "two" from "one"—existence and intelligence from oneness of consciousness as Samhita—as described in Maharishi's Vedic Mathematics (Maharishi, 1972, Lesson 8).[1] Each function can be viewed as a collection of ordered pairs, but when a function is definable, this means that there is, associated with the "existence" of the function as a set of ordered pairs, a *formula* that embodies the intelligence that underlies the operation of that function; the formula *defines* the function.

We will then discuss the computable functions—those functions whose input/output behavior can be captured by an algorithm or computer program. This class of functions is distinct from the class of definable functions in the sense that these functions are in fact *derivable*; by way of sequential steps of logic, each function in this class, with the unique features that characterize each, can be derived. By analogy, this quality of emergence by way of sequential unfoldment is characteristic of "three" emerging from "two" in Maharishi's Vedic Mathematics. In the togetherness of intelligence and existence, intelligence becomes

1 The emergence of two from one is also expressed elsewhere by Maharishi (Nader, 1995, p. 33) in terms of the sprouting of infinite silence and infinite dynamism from thstill as the emergence of unity and diversity (Maharishi, 1996, p. 345).

intelligent, becomes aware of its existence, by the very nature of intelligence as pure awareness. In other words, the togetherness of intelligence and existence results in a flow of intelligence. This flow is the flow of the knower toward the known, and introduces a third element: the process of knowing. In Sanskrit, knower, process of knowing, and object of knowledge have the names Rishi, Devata, and Chhandas, respectively (Maharishi, 1992, pp. 20–21). Being aspects of pure awareness, each of these is capable of being aware of any of the others; Rishi sees Devata and Chhandas; Devata sees Rishi and Chhandas; and so forth. In the act of knowing, the knower affects the known, and so, for example, the Chhandas that is seen by Rishi is now a slightly modified value of Chhandas, which in turn sees each of the other values, and in turn has an effect on each of them. In this way, all possible combinations and flavors of pure awareness sequentially unfold and give rise to all possible expressions. It is therefore at the level of the "three"—Rishi, Devata, Chhandas—that sequential unfoldment of the universe begins to take place. And, by analogy, we find that the distinguishing feature of the class of computable functions is their derivability in a sequence of logical steps or computations. Moreover, we will see that, using one of several formalizations of computable functions, all such functions can be shown to be derivable from just *three*.

Finally, we will discuss the class of polynomial time computable functions—those computable functions that can be implemented in a feasible way.



The concept of a function in this course will be intimately related to certain kinds of problems that we may wish to solve. Our analysis will show that certain kinds of problems are inherently unsolvable; others, though solvable in principle, are not known to be solvable in a feasible way; while still others are solvable, with solutions that can be implemented practically.

The course will focus primarily on the techniques for solving the problems that have feasible solutions, and demonstrating the degree of feasibility of the solutions. However, we will also discuss the applied value of problems with no known feasible solutions—particularly in the case of cryptography.

Definability vs. Computability

Recall that $N = N^1$ is the set $\{ 0, 1, 2, \ldots \}$ of natural numbers, $N \times N = N^2$ is the set $\{ (m, n) \mid m, n$ are natural numbers $\}$, and for each $k \geq 2$, N^k is the set $\{ (a_1, a_2, \ldots, a_k) \mid a_1, a_2, \ldots, a_k$ are natural numbers $\}$ of all k-tuples of natural numbers. Let A be a subset of N^k. A function $f : A \to N$ is an assignment of a unique natural number b to each k-tuple $(a_1, \ldots, a_k)$ in A; we write $f(a_1, \ldots, a_k) = b$. Such an f is called a *partial* function $N^k \to N$. If $A = N^k$, f is called a *total* function $N^k \to N$. Intuitively, if we can give a description of the input/output behavior of f using just natural number concepts, then f is said to be *definable* (we give a more precise definition below).

As a final warm-up definition, let us say that, given natural numbers a, b, *a divides b*, and write $a \mid b$, if there is a number c such that $b = ac$ (in other words, b is divisible by a with no remainder).

To a large extent, the significance of definability of a function is that it informs us, in a mathematical way, that it *exists*. This "mathematical way" begins with a mathematical formula; this formula represents the fundamental intelligence underlying the behavior of that function. A proof of the existence of the function depends critically on the presence of such a formula; the formula gives a precise description of how the function behaves, and without this information, there would be no way to provide a proof that such a function exists. Therefore, although there are in reality a vast number of number-theoretic functions, the collection of those that we can actually "get our hands on" is much narrower, and these are called the definable functions.

Interestingly, the definability of a function, though it informs us of its precise input/output behavior and is the basis for a proof of the existence of the function, does not necessarily provide us with a *procedure* for computing the values of the function; definability informs us about what the function is and does, but does not tell us *how* it does it (or how its behavior could be simulated computationally).

To illustrate the distinction between our ability to demonstrate the existence of a function using a defining formula on the one hand and being able to compute the values of the function on the other, we consider the following assertion:

Assertion. There is a partial function **gcd** : $\mathrm{N} \times \mathrm{N} \rightarrow \mathrm{N}$, which, on input (m, n), outputs the greatest number d that divides both m and n.

The **gcd** function is the function that, on input m, n, outputs the greatest common divisor of m and n. We give a proof of the existence of this function without giving any idea about how to compute its values. As we will see, the proof relies entirely on the definition of the function.

Theorem. Let $A = \{ (m, n) \mid \text{either } m \text{ or } n \text{ is not } 0 \}$. Then there is a function **gcd** : $A \rightarrow \mathrm{N}$ such that, on input (m, n), **gcd** outputs the greatest integer d that divides both m and n.

Proof. Given natural numbers m, n, not both 0, it is clear that 1 divides them both. Since at least one of them is not 0, it is also clear that there are fewer than $m + n$ divisors of both m and n. So there must be a largest divisor. □

The proof of the theorem establishes that the partial **gcd** function exists, but does not tell how to go about computing **gcd**(m, n) for particular values of m and n. As we will illustrate in a moment, the proof is founded on the precise definition of the function in question, and this definition can be represented as a formula. In fact, a formula $\psi(m, n, g)$ can be given informally here to illustrate the point:

$\psi(m, n, g)$: g divides m and n and, if h also divides m and n, then $h \leq g$.

The formula ψ defines the **gcd** function in the sense that for any natural numbers m, n, g, **gcd**$(m, n) = g$ if and only if $\psi(m, n, g)$ is true. Notice that ψ tells us the characteristic and defining features of **gcd** without telling us how to compute values procedurally. For instance, it doesn't tell us how to compute the **gcd** of 2342 and 54.

For some theoretical purposes in mathematics, just knowing that a certain function exists is all that is necessary. Usually though, it is desirable to come up with a procedure for computing the values of a function, once it has been demonstrated to be definable. "Coming up with a procedure" is another way of saying "finding an algorithm" that computes the function.

Typically, once one has a (definable) function in hand, if the input/output values of the function can indeed be computed at all, there will be many different algorithms that can be used for this purpose. We show here two different ways to compute values of the **gcd** function. The first approach is naturally called a *brute force* method that one might think of based on the existence proof just presented.

```
public static int gcd1(int m, int n) {
    if(m==0 && n==0) {
          while(true) { ; } //infinite loop
    }
    int gcd = 1;
    for(int i = 1; i < m + n; ++i) {
          if(divides(i,m) && divides(i,n)){
                gcd = i;
          }
    }
    return gcd;
}
public static boolean divides(int i, int n){
    return n % i == 0;
}
```

It turns out that there is a much more efficient way of computing **gcd**:

```
public static void gcd2(int m, int n) {
    if(m==0 && n==0) {
          while(true){ ; } //infinite loop
    }
    return computeGcd(m,n);
}
public static int computeGcd(int m, int n) {
    if(n==0) return m;
    return computeGcd(n, m % n);
}
```

As an example of the increased efficiency of the second approach, on a Windows XP laptop, computing **gcd**(123055509, 222535599) with the first algorithm requires more than 10 seconds, but with the second algorithm, less than a millisecond is necessary. This example motivates one of the main objectives of the course: to learn techniques for efficient algorithm design and for analyzing algorithm efficiency.

As a final observation about this example, we remark that the **gcd** function is indeed *partial* in the sense that it is not defined on all possible pairs (m, n) belonging to N × N. In particular, **gcd** is not defined on the input (0, 0). This is the reason why, in both programs above, we have put the function into an infinite loop when the input is (0, 0); this is the conventional way of indicating, in a programmatic way, that the input value does not belong to the domain of the function.

The Standard Model of Arithmetic

In order to make the concept of definability more precise, we take a short detour and define the standard model of arithmetic. As we will see, when we say that a function is definable, what we actually mean in this paper[2] is that it is definable in the standard model of arithmetic. Toward this goal, we give a brief introduction to the first-order logic of arithmetic (see Keisler and Robbins, 1996, or Enderton, 1972, for more on this topic).

[1]There are several ways to make the notion of definability precise. The way we discuss here is standard in computability theory, but in set theory a broader definition is used; in that context, one would say that a number-theoretic function is definable if it belongs to the constructible universe (again, there are other variants). We mention this here because our computation of the size of the set of definable functions depends on our definition of "definable." See the second footnote about this point later in the paper.

We begin with a language—the language of arithmetic—that provides special symbols for the binary operations of addition and multiplication, the unary successor function, and the constants **0** and **1**. In particular:

$$\mathcal{L} = \{\, \mathbf{s}, +, *, \mathbf{0}, \mathbf{1} \,\}$$

Our language $\mathcal{L}$ consists of *formal symbols*. The symbols are chosen to suggest a particular intended interpretation. For instance the symbol '+' suggests addition, even though, as a symbol belonging to $\mathcal{L}$, it has not yet been interpreted as addition (and could actually be interpreted to be virtually any operation).

Another part of the language is an infinite set of *variables*. Formally, these variables can be listed as $v_1, v_2, v_3, \ldots$, but for our purposes, it will be good enough to use boldface letters at the end of the alphabet, such as **x**, **y**, **z**.

We will now give rules for combining the various symbols (language elements and variables) to form *terms*, and then introduce some additional logical symbols that will allow us to create *formulas*. Once we have these terms and formulas, we will describe how they will be given meaning by interpreting them in the standard model of arithmetic. All this effort will then allow us to state precisely, in the next section, what it means for a function to be definable.

We give the following recursive definition of terms: A term t is one of the following:

(1) one of the constant symbols **0**, **1**;
(2) a variable;
(3) $\mathbf{s}(u)$, where u is another term;
(4) either $(u + v)$ or $(u * v)$, where u and v are other terms.

This recursion is legitimate because, in the recursive clauses (3) and (4), the term t must always end up being longer than any of its component terms.

Here are some examples of terms:

0
y

$\mathbf{s}(\mathbf{0})$
$((\mathbf{x} + 1) * \mathbf{y})$.

We now build up formulas. We begin with atomic formulas. An atomic formula is an expression of the form $(t = u)$ where t, u are terms. Here is an example of an atomic formula:

$$(\mathbf{s}(\mathbf{s}(\mathbf{0})) = (\mathbf{1} + \mathbf{1})).$$

We introduce the following symbols from mathematical logic: The symbol '$\wedge$' stands for 'and'; the symbol '$\vee$' stands for 'or'; the symbol '$\rightarrow$' stands for 'implies'; the symbol '$\neg$' stands for 'not'; '$\forall$' stands for 'for each'; and '$\exists$' stands for 'there exists.' $\forall$ and $\exists$ are called *quantifiers*. Starting from atomic formulas, we build up all formulas recursively as follows:

A formula ψ is one of the following:

(1) an atomic formula;

(2) $\neg\psi$, where ψ is a formula;

(3) $(\psi \wedge \theta)$, $(\psi \vee \theta)$, or $(\psi \rightarrow \theta)$, where ψ, θ are formulas;

(4) $\forall\mathbf{x}\psi$ or $\exists\mathbf{x}\psi$, where ψ is a formula and $\mathbf{x}$ is a variable.

The recursive steps (2)–(4) are legitimate because new formulas are longer than the formulas from which they are built. Here are some examples of formulas:

$((\mathbf{1} + \mathbf{0}) = \mathbf{1})$

$\neg(\mathbf{s}(\mathbf{1}) = \mathbf{0})$

$((\mathbf{0} = \mathbf{1}) \vee \neg(\mathbf{0} = \mathbf{1}))$

Now we give meaning to terms and formulas by interpreting the symbols we have defined so far in a model. The standard model of arithmetic, denoted $\mathcal{N}$, consists of a base set N (the set of natural numbers 0, 1, 2, . . .), together with the usual operations of addition (+), multiplication (∗), and a successor function $s : \mathrm{N} \rightarrow \mathrm{N}$ defined by $s(n)$

= $n + 1$. We think of the addition operation as an *interpretation* of the addition symbol **+** defined above. Likewise, ordinary multiplication is an interpretation of $\ast$; the successor function is an interpretation of **s**. And we will consider the two ordinary natural numbers 0 and 1 to be interpretations of the symbols **0** and **1**, respectively.

More formally, we write:

$\mathbf{0}^{\mathcal{N}} = 0$

$\mathbf{1}^{\mathcal{N}} = 1$

$\mathbf{+}^{\mathcal{N}} = +$

$\boldsymbol{\ast}^{\mathcal{N}} = \ast$

$\mathbf{s}^{\mathcal{N}} = s.$

Given a term t, we give an interpretation $t^{\mathcal{N}}$ of t. We wish to define an interpretation in such a way that the term is evaluated to a number. However, if the term happens to contain a variable, we cannot hope to determine the value. For instance, the term **s**(**x**) cannot be evaluated uniquely to a single number. Recall that **s** is a symbol whose standard interpretation is the successor function. So, it follows that, once we know which number the variable **x** is supposed to stand for, we can evaluate the term as "1 plus the value of **x**," but not until then. So, as part of our interpretation scheme, we define how terms are to be evaluated when particular numbers are used to replace the variables.

A convention we observe is that if the variables in the term t are among **x**, **y**, we will express t as $t(\mathbf{x}, \mathbf{y})$. (In the general case, we would use n variables instead of two.)

We can now finally define how terms are to be interpreted in $\mathcal{N}$. Given a term $t(\mathbf{x}, \mathbf{y})$, we seek to interpret t as a number, but, as we just observed, can do so only relative to an interpretation of the variables **x**, **y** as particular numbers. Thus we formally define the interpretation of $t(\mathbf{x}, \mathbf{y})$ at numbers a, b in $\mathcal{N}$, denoted $t^{\mathcal{N}}[a, b]$, by the following recursive clauses:

1. if $t(\mathbf{x}, \mathbf{y})$ is **x**, $t^{\mathcal{N}}[a, b] = \mathbf{x}^{\mathcal{N}}[a] = a$
2. if $t(\mathbf{x}, \mathbf{y})$ is **0**, $t^{\mathcal{N}}[a, b] = 0$

3. if $t(\mathbf{x}, \mathbf{y})$ is $\mathbf{1}$, $t^{\mathcal{N}}[a, b] = 1$
4. if $t(\mathbf{x}, \mathbf{y})$ is $u(\mathbf{x}, \mathbf{y}) + v(\mathbf{x}, \mathbf{y})$, then $t^{\mathcal{N}}[a, b] = u^{\mathcal{N}}[a, b] + v^{\mathcal{N}}[a, b]$
5. if $t(\mathbf{x}, \mathbf{y})$ is $u(\mathbf{x}, \mathbf{y}) * v(\mathbf{x}, \mathbf{y})$, then $t^{\mathcal{N}}[a, b] = u^{\mathcal{N}}[a, b] * v^{\mathcal{N}}[a, b]$ (variables suppressed)

This interpretation of terms defines a function {terms} → N by means of evaluation. For example, interpreting the term

$$(\mathbf{ss}(\mathbf{1} + \mathbf{s}(\mathbf{0})) * \mathbf{s}(\mathbf{x}))$$

at $a = 1$ (so that the variable **x** is replaced by the value $a = 1$) leads to the following steps, which result finally in a natural number:

- $(\mathbf{s}(\mathbf{s}(\mathbf{1} + \mathbf{s}(\mathbf{0}))) * \mathbf{s}(\mathbf{x}))^{\mathcal{N}}[1]$
- $s(s(1 + s(0))) * s(1)$
- $4 * 2$
- 8

Finally, we interpret formulas in the standard model $\mathcal{N}$. The result of this interpretation will be a boolean value—either true or false. We will explain this step somewhat informally, but accurately enough for our purposes. In order to perform the interpretation, we need to observe a distinction in the ways in which variables are used in formulas. To see the distinction, consider the following two formulas:

$$\phi : \ \forall \mathbf{x} \ (\mathbf{x} * \mathbf{1} = \mathbf{x})$$
$$\psi : \ \forall \mathbf{x} \ (\mathbf{x} * \mathbf{y} = \mathbf{x}).$$

Notice that the variable **x** in ϕ is "quantified" by $\forall$. Intuitively, ϕ says that, for every number, multiplying by 1 gives back the same number. Notice that the variable **x** in ψ is also "quantified" by $\forall$, but the variable **y** is not associated with a quantifier. Intuitively, ψ says that, for every number, multiplying by the (unspecified) number **y** will give back the original number.

Based on our intuitive interpretations of ϕ and ψ, it seems clear that ϕ is true, but ψ may or may not be true, depending on which number the variable **y** stands for. For instance, if **y** is interpreted as the number 1, ψ would be true, but if **y** is interpreted as 2, ψ would be false.

The formula ϕ is an example of a *sentence*—a sentence is a formula in which no variable occurs free. This means that each occurrence of each variable is tied to a quantifier. The formula ψ is an example of a formula having a free variable $\mathbf{y}$. In order to interpret formulas that have free variables, we will need to supply numbers to fill in for these variables, so that a final value of true or false can be obtained.

To facilitate this plan, we will adopt the following convention: To denote a formula θ whose free variables are among $\mathbf{x}$, $\mathbf{y}$, we write $\theta(\mathbf{x}, \mathbf{y})$. If $\mathbf{z}$ also occurs in θ but does not occur free, we do not list $\mathbf{z}$ among the variables in the presentation of θ.

Now we can now interpret a formula $\phi(\mathbf{x}, \mathbf{y})$ in $\mathcal{N}$ as follows (to be formal, we should list n free variables, but for readability, we just use two). The result of interpreting $\phi(\mathbf{x}, \mathbf{y})$ in $\mathcal{N}$ at natural numbers a, b, denoted $\phi^{\mathcal{N}}[a, b]$, is defined recursively as follows:

1. if $\phi(\mathbf{x}, \mathbf{y})$ is atomic, of the form $t(\mathbf{x}, \mathbf{y}) = u(\mathbf{x}, \mathbf{y})$, then $\phi^{\mathcal{N}}[a, b]$ is true if and only if $t^{\mathcal{N}}[a, b] = u^{\mathcal{N}}[a, b]$;

2. if ϕ is $\neg\psi$, then $\phi^{\mathcal{N}}$ is true if and only if $\psi^{\mathcal{N}}$ is false;

3. if ϕ is $\psi \wedge \theta$, then $\phi^{\mathcal{N}}$ is true if and only if both $\psi^{\mathcal{N}}$ and $\phi^{\mathcal{N}}$ are true;

4. if ϕ is $\psi \vee \theta$, then $\phi^{\mathcal{N}}$ is true if and only if at least one of $\psi^{\mathcal{N}}$ and $\phi^{\mathcal{N}}$ is true;

5. if ϕ is $\psi \rightarrow \theta$, then $\phi^{\mathcal{N}}$ is false if and only if $\psi^{\mathcal{N}}$ is true and $\phi^{\mathcal{N}}$ is false;

6. if $\phi(\mathbf{y}, \mathbf{z})$ is $\forall x\psi(\mathbf{x}, \mathbf{y}, \mathbf{z})$, then $\phi^{\mathcal{N}}[b, c]$ is true if and only if $\psi^{\mathcal{N}}[a, b, c]$ is true for all a in N.

7. if $\phi(\mathbf{y}, \mathbf{z})$ is $\exists x\psi(\mathbf{x}, \mathbf{y}, \mathbf{z})$, then $\phi^{\mathcal{N}}[b, c]$ is true if and only if $\psi^{\mathcal{N}}[a, b, c]$ is true for at least one a in N.

Example. Consider the following formula:

$$\phi(\mathbf{x}, \mathbf{y}) : \qquad \exists \mathbf{z}\ (\mathbf{y} = \mathbf{x} * \mathbf{z}).$$

We evaluate $\phi^{\mathcal{N}}[3, 12]$ and $\phi^{\mathcal{N}}[2, 5]$:

$$\begin{aligned}\phi^{\mathcal{N}}[3, 12] \text{ is true} &\text{ iff } (\exists \mathbf{z}\ (\mathbf{y} = \mathbf{x} * \mathbf{z}))^{\mathcal{N}}[3, 12] \text{ is true}\\ &\text{ iff } (\mathbf{y} = \mathbf{x} * \mathbf{z})^{\mathcal{N}}[a, 3, 12] \text{ for some } a \text{ in N}\\ &\text{ iff } (12 = 3 * a)^{\mathcal{N}} \text{ for some } a \text{ in N}\\ &\text{ iff } 12 = 3 * a \text{ for some } a \text{ in N.}\end{aligned}$$

Clearly, the last line is true, since $12 = 3 * 4$. So the formula ϕ is true at 3, 12.

However:

$$\begin{aligned}\phi^{\mathcal{N}}[2, 5] \text{ is true} &\text{ iff } (\exists \mathbf{z}\ (\mathbf{y} = \mathbf{x} * \mathbf{z}))^{\mathcal{N}}[2, 5] \text{ is true}\\ &\text{ iff } (\mathbf{y} = \mathbf{x} * \mathbf{z})^{\mathcal{N}}[a, 2, 5] \text{ for some } a \text{ in N}\\ &\text{ iff } (5 = 2 * a)^{\mathcal{N}} \text{ for some } a \text{ in N}\\ &\text{ iff } 5 = 2 * a \text{ for some } a \text{ in N.}\end{aligned}$$

In the second case, since there is no integer a for which $5 = 2 * a$, we conclude that ϕ is false at 2, 5. These considerations lead to the observation that the formula ϕ is true in $\mathcal{N}$ at numbers b, c if and only if b divides c.

Defining Functions and Relations in the Standard Model

We can now explain what it means for a function to be definable in the standard model of arithmetic. We will give this explanation for functions of two variables, but generalizing to n variables is a straightforward exercise. Also, we explain what it means for two-place relations (which are a generalization of unary functions) to be definable in $\mathcal{N}$; in fact the latter will be a natural starting point for understanding definability of functions in $\mathcal{N}$.

Suppose R is a subset of N × N. (A familiar example is the "less than relation," expressed as ordered pairs: define R to consist of those pairs (a, b) for which $a < b$.) We say that R is *definable in* $\mathcal{N}$ if there is a formula $\phi(\mathbf{x}, \mathbf{y})$ such that, for all pairs of natural numbers (a, b), we have

$$(a, b) \text{ is a member of } R \text{ if and only if } \phi^{\mathcal{N}}[a, b] \text{ is true.}$$

Example. Let R be the set of all ordered pairs (a, b) for which it is true that a divides b. Let $\phi(\mathbf{x}, \mathbf{y})$ be the formula defined in the previous example, namely

$$\phi(\mathbf{x}, \mathbf{y}) : \quad \exists \mathbf{z}\ (\mathbf{y} = \mathbf{x} * \mathbf{z}).$$

Then, as the previous example explains, we have

$$(a, b) \text{ is a member of } R \text{ if and only if } \phi^{\mathcal{N}}[a, b] \text{ is true.}$$

Therefore, the "divides" relation is definable in $\mathcal{N}$.

Exercise. Let R be the set of all ordered pairs (a, b) for which $a \leq b$. Let $\phi(\mathbf{x}, \mathbf{y})$ be the formula defined by

$$\phi(\mathbf{x}, \mathbf{y}) : \quad \exists \mathbf{z}\ (\mathbf{y} = \mathbf{x} + \mathbf{z}).$$

Show that

$$(a, b) \in R \text{ if and only if } \phi^{\mathcal{N}}[a, b].$$

Therefore the "≤" relation is definable in $\mathcal{N}$.

Suppose $f : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is a partial function, with domain A. Then we say f is *definable in* $\mathcal{N}$ if there is a formula $\psi(\mathbf{x}, \mathbf{y}, \mathbf{z})$ such that, for all natural numbers a, b, c, we have

$$c = f(a, b) \text{ if and only if } \psi^{\mathcal{N}}[a, b, c].$$

Example. We show that the function **gcd** defined earlier is definable in $\mathcal{N}$. Since we already know that "divides" and "less than or equal to" are definable relations, we will use convenient symbols for these relations ('|' and '≤' respectively) in the formula we will build, with the understanding that these symbols could be replaced by the defining formulas in a straightforward way. We define $\psi(\mathbf{x}, \mathbf{y}, \mathbf{z})$ as follows:

$$\psi(\mathbf{x}, \mathbf{y}, \mathbf{z}) : \qquad (\mathbf{z} \mid \mathbf{x} \wedge \mathbf{z} \mid \mathbf{y} \wedge [\forall \mathbf{u}\ (\mathbf{u} \mid \mathbf{x} \wedge \mathbf{u} \mid \mathbf{y} \to \mathbf{u} \leq \mathbf{z})]).$$

Now, given (a, b, c), we verify that $c = \mathbf{gcd}(a, b)$ if and only if $\psi^{\mathcal{N}}[a, b, c]$.

$c = \mathbf{gcd}(a, b)$ iff $c|a$ and $c|b$ and c is the largest such number
iff $c|a$ and $c|b$ and for any d that divides both a and b, $d \leq c$
iff $c|a$ and $c|b$ and for each d in N ($d|a$ and $d|b$ implies $d \leq c$)
iff $\psi^{\mathcal{N}}[a, b, c]$.

This example demonstrates the pattern that we have for all definable functions: the input/output behavior of such a function—its "existence" as a set of ordered pairs—is captured in a formula which defines the

function relative to the standard model $\mathcal{N}$. It is this "intelligence" characteristic provided by such a formula that allows us to conceive of the function at all. Functions that have, beyond their bare existence as a set of ordered pairs, a formula that defines them form a specialized subcollection of the class of all number-theoretic functions—a class called the class of *definable functions*.

Is Every Function $\mathbf{N}^k \to \mathbf{N}$ Definable?

We show here that, although it's reasonably accurate to say that "any function you can think of" is definable in $\mathcal{N}$, in reality, nearly all functions $\mathrm{N}^k \to \mathrm{N}$ are not definable.[3] This observation should help provide perspective on the scope of our course. The finely honed tools that we will develop in our analysis of algorithms will be relevant only to a tiny sliver of the world of functions from $\mathrm{N} \to \mathrm{N}$.

For convenience, we restrict ourselves to the case $k = 1$: we show that there are functions $\mathrm{N} \to \mathrm{N}$ that are not definable. Remember that every definable function $\mathrm{N} \to \mathrm{N}$ is defined by a formula $\phi(\mathbf{x}, \mathbf{y})$. ϕ itself is an expression involving finitely many symbols. We could arrange all formulas in a sequence, according to length. First we list all the length-1 formulas, then the length-2 formulas, and so forth.[4] (To be more careful in handling the fact that we have infinitely many variables in our language, we can devise the list more carefully as follows: List all length-1 formulas that use only variable v_1 first. Then list all length-1 and length-2 formulas that use variables only among v_1, v_2. Then list all length-1, length-2, and length-3 formulas that use variables only among v_1, v_2, v_3. And so forth.) In this way, we have a list

$$\phi_0, \phi_1, \phi_2, \ldots$$

of all formulas that define functions $\mathrm{N} \to \mathrm{N}$. Replacing each ϕ_i by the function that it defines gives us the list:

[2] As was discussed in the first footnote, this point about the size of the set of definable functions depends on our definition of "definable." If instead we take "definable" to mean "constructible" (as a set theorist might do), the thread of reasoning still makes the same basic point but in a slightly different way. One shows in this case that one of the following two assertions must be true: (a) The set of definable functions is a smaller infinity than the set of all functions. (b) The set of computable functions is a smaller infinity than the set of definable functions. Either way, the set of computable functions always ends up being a tiny speck in the very large set of all number-theoretic functions.

4 For any m, a *length-m formula* is a formula that has exactly m symbols.

$$f_0, f_1, f_2, \ldots.$$

Now we have a list of all the definable functions N → N. The question is, does this list exhaust all the functions from N → N? If we come up with a function not on this list, the answer must be "no."

Consider the following function $g : N \to N$:

$$\text{for each } k,\ g(k) = f_k(k) + 1.$$

Is g in the list? Notice that $g \neq f_0$ because g disagrees with f_0 at 0: Namely, $g(0) = f_0(0) + 1 \neq f_0(0)$. But the same can be said for each k: $g \neq f_k$ because g disagrees with f_k at k, since $g(k) = f_k(k) + 1 \neq f_k(k)$.

We have found a function g not on the list. Therefore, not every function N→ N is definable. But we have shown much more. In fact we have shown that, no matter what list of functions N → N one comes up with,

$$h_0, h_1, h_2, \ldots,$$

there will always be a function g, defined exactly as above, that does not lie on the list. This means that the collection of all functions N → N simply cannot be arranged in a list! The collection of all such functions is too big to be put in a list. This fact demonstrates that the size of the set of all functions N → N is a higher order of infinity than the set of natural numbers (see Jech, 1978, for more on higher orders of infinity).

The point of all these esoteric considerations is that the collection of all functions that are definable represents only a miniscule fraction of the vast collection of all functions N → N.

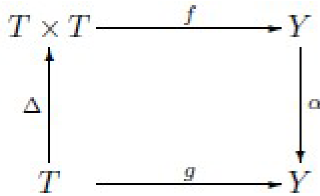
Diagonalization and Fixed Points

The result proved in the previous section uses a technique that appears over and over in the theory of computability. We wished to show that some function existed outside a particular class of functions. The technique for building such a function is called *diagonalization*—we built g by examining each f_k and forcing g to disagree with it. One says that one builds g by "diagonalizing out" of the class.

Diagonalization provides a rich analogy to the process of transcending. While "diagonalizing out" describes the phenomenon of "going beyond," another process that arises in computability theory describes

another aspect of transcending equally well: The process of finding fixed points. If $\alpha : Y \to Y$ is a function, a *fixed point* for α is a $y \in Y$ such that $\alpha(y) = y$. Finding fixed points involves locating the self-referral dynamics inherent in a process.

Both the process of diagonalization and of locating fixed points have been largely captured in a striking abstract diagram (Yanofsky, 2003):



The following two theorems (actually two variants of a single theorem), which give significance to the diagram, are the following:

Diagonalization Theorem. Suppose $f: T \times T \to Y$ and $\alpha : Y \to Y$ are functions. Define $\Delta : T \to T \times T$ by $\Delta(t) = (t, t)$. Then if α has no fixed point, there is a function $g : T \to Y$ that is not represented by f—that is, there is no t_0 for which $f(t, t_0) = g(t)$ for each t. Moreover, g can be defined by $\alpha \circ f \circ \Delta$.

Fixed Point Theorem. Suppose $f: T \times T \to Y$ and $\alpha : Y \to Y$ are functions. Define $\Delta : T \to T \times T$ by $\Delta(t) = (t, t)$. If $\alpha \circ f \circ \Delta$ can be represented by f (that is, for some t_0, $f(t, t_0) = \alpha(f(\Delta(t)))$ for every t), then α must have a fixed point.

By way of review, if $f: A \to B$ and $g : B \to C$ are functions, then the composition of g with f, denoted $g \circ f$, is defined by

$$\text{dom } g \circ f = A$$
$$g \circ f(a) = g(f(a)).$$

The diagram asserts that $\alpha \circ f \circ \Delta = g$: Following the bottom arrow g from T to Y is the same as following the arrow Δ from T to $T \times T$, followed by the arrow f from $T \times T$ to Y, followed by the arrow α from Y to Y. For more on commutative diagrams, see (Pierce, 1991).

As an example, we use the Diagonalization Theorem to reprove the result in the previous section: Let $f_0, f_1, f_2, \ldots$ be a list of all the definable functions (or for that matter, any list of functions $N \to N$), as described earlier. Define $F : N \times N \to N$ by

$$F(m, n) = f_n(m).$$

Define $\alpha : N \to N$ by $\alpha(k) = k + 1$. (The connection to the diagram is that we have set $T = N$ and $Y = N$.) The definition of α makes it clear that α has no fixed point. So the theorem applies, and we can find a g that is not represented by F; in particular, $g = \alpha \circ F \circ \Delta$ does the job. This means that, for each choice of n, the function f_n does not agree with g. But this is precisely what we were seeking: a function g that disagrees with every one of the functions f_n on the list. (Notice how the composition $\alpha \circ F \circ \Delta$ reveals the mechanics of diagonalization in this case: Requiring Δ to precede F forces us to consider the values $F(n, n)$; further applying α makes sure that $\alpha(F(n, n))$ always differs from $F(n, n)$, and so, in particular, that $g(n)$ is different from $f_n(n)$.)

Computable Functions

Prior to the 1950s, there was considerable interest in giving a mathematical characterization of those functions $N \to N$ (or $N^k \to N^m$) whose input/output behavior could be captured by an algorithm. An inherent difficulty in making this determination was that there were many views about the definition of "algorithm" (see Rogers, 1988, and Bell & Machover, 1977).

A natural, modern-day answer to the question is: A partial function $f: N \to N$ is computable if and only if there is a Java program which, on input n, either fails to halt (in case n is not in the domain of f) or halts with output $f(n)$.

For example, the function f defined by $f(n) = 2n$ is computable because the following Java method demonstrates that it is:

```
int f(int n) {

     return n + n;
}
```

For this criterion for computability to be workable, we need to introduce one abstraction in our concept of a "Java program"—we must assume that the program can be run on a computer with "as much memory as necessary." Otherwise, the program given above for $2n$ will fail to compute values (correctly, or at all) once the input integer exceeds memory limits. Later in this section, we will make the necessary assumptions more precise. Historically, many classes of functions have been proposed as candidates for the "class of all computable functions." The definitions of these classes have varied widely. The following is a partial list (see Bell & Machover, 1977; Rogers, 1988; and Barendregt 1984).

(1) *The class of partial functions computable by a Turing machine.* Turing machines are extremely simple structures that behave like low-level programs—like assembly code for an extremely simple processor. Input and output are handled by a tape containing infinitely many cells which may be blank or contain any character from a predetermined finite alphabet—however, at any time, all but finitely many of the cells are blank. The tape head is used both for reading and writing characters on the tape. When the machine begins to run, it reads the current value of the tape and checks its instruction set to determine what to do next. Its next step depends upon the value read and its current state (it may be in any of finitely many states). The tape head can move one cell to the left or right, erase a value, or write a value. In taking such an action, it may enter a new state, or remain in its present state. On a given input, the machine may halt, but on other inputs it may not. A partial function $f: N \to N$ is *Turing computable* if there is a Turing machine M which, when started on the first of n consecutive cells containing the symbol '1' (and the rest of the tape blank), the machine eventually halts with head on the first of $f(n)$ consecutive 1s (and the rest of the tape is blank) if n is in the domain of f, or else never halts. Similarly, a partial function $f: N^2 \to N$ is Turing computable if there is a Turing machine M which, when started on the leftmost 1 on a tape having m consecutive 1s followed by a blank followed by n consecutive 1s (and the rest of the tape blank), eventually halts with head on the first of $f(m, n)$ consecutive 1s (and the rest of the tape is blank) if (m, n) is in the domain of f, or else never halts.

(2) *The class of partial functions computable by a nondeterministic Turing machine.* A *nondeterministic Turing machine* is just like a Turing machine, except that, when reading a value on the tape, and when in a particular state, its next step is not, in some cases, uniquely determined—it may do any of a number of things. In this case, f is considered computable if there is a nondeterministic machine M which, when started on the first 1 of n consecutive 1s (and the rest of the tape blank), has an execution path that will cause M to eventually halt with tape head on the first 1 of $f(n)$ consecutive 1s (and the rest of the tape blank). A similar description provides a definition for a partial function $N^2 \to N$ to be computable by a nondeterministic Turing machine.

(3) *The class of λ-definable partial functions.* Church devised a formal system called the *λ-calculus* in an effort to build a foundation for all of mathematics based on the concept of a function. The formal system consists of an infinite set of variables $v_1, v_2, \ldots$, together with λ-terms defined recursively by:

(a) Any variable is a term;

(b) If M and N are terms, (MN) is a term;

(c) If M is a term and x is a variable, $(\lambda x.M)$ is a term.

The λ-notation is a way of specifying a function. Stepping outside the formal system for a moment, one can use the λ-notation to specify the function that takes n to $2n$ by writing

$$\lambda n.2n.$$

("$\lambda n.2n$ is the function that takes argument n to $2n$.")

Clause (c) says, intuitively speaking, that if M is a term, the "function" that takes x to M is also a term. Multiple applications of clause (c) are typically written in abbreviated form. For instance $\lambda x.\lambda y.M$ is written $\lambda xy.M$. The number of parentheses involved in writing down terms according to rules (b) and (c) tends to become unmanageable, so there is an "association to the left" convention: It is understood that the expression MNP is shorthand for $((MN)P)$. If x is a variable, x^2 denotes the term xx, x^3 denotes $x(xx)$, x^4 denotes $x(x(xx))$, and so forth.

λ-terms can often be "reduced" to simpler terms. Many of these reductions follow our intuitive expectation. For instance, the term $(\lambda x.y)M$ reduces to simply y. The intuitive reason is that $\lambda x.y$ is the function that takes any x to constant value y. If this function is applied to the input M, the output is simply y. We write $(\lambda x.y)M \to y$. As an exercise, the reader is invited to verify intuitively why $(\lambda x.x(xy))M \to M(My)$.

Intuitively, the λ-terms that have no "free" variables correspond to functions with actual input/output behavior. For instance, the λ-term $\lambda x.xx$ *binds* the variable x; this makes it possible to uniquely determine an output from a given input. In this case, still speaking intuitively, $\lambda x.xx$ is the function that, for any input a, outputs aa. By contrast, the λ-term mentioned in the previous paragraph—$\lambda x.x(xy)$—has the free variable y, so any computation using this term will still leave the value of y undetermined. λ-terms that have no free variable are called *closed terms*. As was mentioned in the introduction, it can be shown that all closed terms can be derived from an initial set of just *three* closed terms—usually denoted I, K, S. For completeness of the presentation, we give their definitions here:

$$
\begin{aligned}
I &: \quad \lambda x.x \\
K &: \quad \lambda xy.x \\
S &: \quad \lambda xyz.xz(yz).
\end{aligned}
$$

Natural numbers are represented in the λ-calculus in the following way: the natural number n is represented by the λ-*numeral* $\lambda xy.x^n y$. In the special case of $n = 0$, the definition asserts that 0 is represented by the term $\lambda xy.y$. A shorthand notation for the λ-numeral for n is $\overline{n}$.

We can now define what it means for a number-theoretic function to be computable in the sense of the λ-calculus. We do this for total functions; a bit of additional development would be needed to handle the case of partial functions. A total function $f : \mathrm{N} \times \mathrm{N} \to \mathrm{N}$ is said to be λ-*definable* if there is a λ-term X such that for each $m, n \in N$,

$$X\,\overline{m}\,\overline{n} \to \overline{f(m,n)}.$$

(This definition generalizes to arbitrarily many variables in the obvious way.)

As we will indicate shortly, the λ-definable functions turn out to be precisely the computable functions, the functions whose behavior is computable with an algorithm. The fact that all λ-definable functions (and in fact, all closed terms) are derivable from just three is one of many striking analogies between the way in which the self-referral computational dynamics of pure consciousness find a direct parallel in the foundational dynamics of the λ-calculus. According to Maharishi Vedic Science, the sequential unfoldment of the Veda, and, ultimately, all of manifest existence, arises from self-referral dynamics in which pure awareness interacts with itself, simultaneously playing the roles of knower (Rishi), process of knowing (Devata), and known (Chhandas). In the λ-calculus context, the three fundamental closed terms play a similar role: The term I, known as the *identity combinator,* represents pure silence, the witness, the Rishi value, since its action under application is to leave its argument unchanged. The term K acts by "forgetting" its second argument; this "hiding" influence corresponds naturally to Chhandas. And the term S has a strongly dynamic flavor as it distributes its third argument z across the other two arguments, and so corresponds naturally to Devata. Moreover, all three can be seen as the "play" of a single term X interacting with itself (the definition of X is rather involved, but can be expressed succinctly using K and S as follows: $X = \lambda z.zKSK$).

(4) *The class of partial recursive functions.* One begins the definition of this class by specifying a *Base Set* of functions, all of which are obviously computable. Then one specifies a collection $\mathcal{O}$ of fundamental operations for building new functions from old ones. The final class consists of all those functions that can be obtained by finitely many applications of these operations to the functions in the Base Set. For this discussion, we will consider functions $\mathbf{N}^k \rightarrow \mathbf{N}$, for all $k \geq 1$.

Base Set. The Base Set $\mathcal{B}$ consists of the following functions:

1. The *successor function*: The function $f : \mathbf{N} \rightarrow \mathbf{N}$ defined by $f(n) = n + 1$.
2. All *constant functions*: For each n, the function $f : \mathbf{N}^k \rightarrow \mathbf{N}$ defined by $f(x_1, x_2, \ldots, x_k) = n$.

3. All *projection functions*: For each i, with $1 \leq i \leq k$, the function $f : N^k \rightarrow N$ defined by $f(x_1, x_2, \ldots, x_k) = x_i$.

Recursive Operations. The recursive operations $\mathcal{O}$ will be applied to the Base Set to build up a large class of functions. Here we define these operations on arbitrary functions and then specify how partial recursive functions are to be built up later on.

1. *Composition*. Given functions $g_1, g_2, \ldots, g_m : N^n \rightarrow N$ and a function $h : N^m \rightarrow N$, the composition $h \circ \langle g_1, g_2, \ldots, g_m \rangle$ is the function $f : N^n \rightarrow N$ defined by

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), g_2(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)).$$

For the case $m = 1$, we write $h \circ g_1$ rather than $h \circ \langle g_1 \rangle$. For the case in which any or all of the g_i are only partially defined, we adopt the convention that $f(x_1, \ldots, x_n)$ is undefined unless $g_1(x_1, \ldots, x_n)$ is defined for all i.

2. *Primitive Recursion*. Given functions $g : N^{n-1} \rightarrow N$ and $h : N^{n-1} \rightarrow N$, the function $f = \mathbf{pr}(g, h) : N^n \rightarrow N$, obtained by primitive recursion from g and h, is defined inductively as follows:

$$f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n)$$
$$f(x_1 + 1, x_2, \ldots, x_n) = h(x_1, f(x_1, x_2, \ldots, x_n), x_2, \ldots, x_n).$$

When $n = 1$, we take g to be a constant (rather than a function). If g is only partially defined, $f(0, x_2, \ldots, x_n)$ will be defined only when $g(x_2, \ldots, x_n)$ is defined, and similarly in the definition of $f(x_1 + 1, x_2, \ldots, x_n)$.

3. *Unbounded Search*. Suppose $g : N^{n+1} \rightarrow N$ is a partial function. Then $f = \mathbf{us}(g) : N^n \rightarrow N$ is defined by

$$f(x_1, \ldots, x_n) = \text{least } y \text{ such that } g(x_1, \ldots, x_n, y) = 0 \text{ and } (x_1, \ldots, x_n, z) \text{ is in the domain of } g \text{ for each } z \leq y.$$

Let us say that a function $f : \mathrm{N}^n \to \mathrm{N}$ is *derivable* from $\mathcal{B}$ and $\mathcal{O}$ if there are functions $f_1, \ldots, f_m$, with $f = f_m$, having the following properties: For each $k \leq m$, f_k is obtained in one of the following ways:

a. f_k belongs to $\mathcal{B}$;
b. for some $i_1, \ldots, i_r$, j all less than k, $f_k = f_j \circ \langle f_{i_1}, \ldots, f_{i_r} \rangle$;
c. for some i, j both less than k, $f_k = \mathbf{pr}(f_i, f_j)$;
d. for some $j < k$, $f_k = \mathbf{us}(f_j)$.

As a simple example, we give a derivation of the function $f(x_1, x_2) = x_1 + x_2$ from $\mathcal{B}$, $\mathcal{O}$:

- f_1 is the successor function.
- $f_2 : \mathrm{N} \to \mathrm{N}$ is defined by $f_2(x) = x$.
- $f_3 : \mathrm{N}^3 \to \mathrm{N}$ is defined by $f_3(x_1, x_2, x_3) = x_2$.
- $f_4 : \mathrm{N}^3 \to \mathrm{N}$ is obtained by composition: $f_4 = f_1 \circ f_3$. Therefore, $f_4(x_1, x_2, x_3) = x_2 + 1$.
- $f_5 : \mathrm{N}^2 \to \mathrm{N}$ is obtained by primitive recursion: $f_5 = \mathbf{pr}(f_2, f_4)$. Therefore,
  $f_5(0, x) = f_2(x) = x$
  $f_5(1, x) = f_4(0, f_5(0, x), x) = x + 1$
  $f_5(2, x) = f_4(1, f_5(1, x), x) = x + 1 + 1 = x + 2$
  $f_5(n + 1, x) = f_4(n, f_5(n, x), x) = x + n + 1$.
  It follows that $f = f_5$, and so f has been derived from $\mathcal{B}$ and $\mathcal{O}$.

Finally, the class of all partial recursive functions is defined to be the set of all functions that are derivable from $\mathcal{B}$ and $\mathcal{O}$.

It may not be obvious that some of the functions belonging to the class of partial recursive functions are only partially defined, since the base set of functions includes only total functions. Partially defined functions do arise, and they do so by virtue of applications of unbounded search. As an example, we show how the partially defined function

$$f(x) = \begin{cases} 0 & \text{if } x \geq 1 \\ \infty & \text{if } x = 0 \end{cases}$$

is derivable.

- f_1 is the constant function with one argument having value 1 (i.e. $f_1(x) = 1$).
- f_2 is the constant function with three arguments having value 0 (i.e. $f_2(x, y, z) = 0$).
- $f_3 : \mathrm{N}^2 \rightarrow \mathrm{N}$ is obtained by primitive recursion: $f_3 = \mathbf{pr}(f_1, f_2)$. Therefore,
  $f_3(0, y) = f_1(y) = 1$;
  $f_3(x + 1, y) = f_2(x, f_3(x, y), y) = 0$.
- $f_4 : \mathrm{N} \rightarrow \mathrm{N}$ is obtained by unbounded search: $f_4 = \mathbf{us}(f_3)$.

Notice that, by the definition of unbounded search, $f_4(0) = 1$, but for all $x > 0, f_4(x) = 0$. It follows that $f = f_4$, and so the partially defined function f has been derived from $\mathcal{B}$ and $\mathcal{O}$.

(5) *The class of partial functions that are weakly representable in Peano Arithmetic*. Peano Arithmetic (PA) consists of a set of sentences in the language of first-order arithmetic (described earlier). PA is an example of an axiomatic theory for arithmetic. As such, it provides a way of systematically deriving sentences that are true in the standard model $\mathcal{N}$. PA begins with a set of obviously true (in $\mathcal{N}$) sentences as its set of axioms, and then allows one to derive one theorem (true sentence) after another using a single rule of inference. Kurt Gödel showed that, although nearly every true sentence of arithmetic that would ever arise in mathematical practice is derivable from PA, PA is not rich enough to allow one to derive all the true sentences.[5]

The following is a list of axioms for Peano Arithmetic (see Keisler & Robbins, 1996).

[3] Moreover, he showed that there is no recursive axiomatic theory of arithmetic from which every true sentence of arithmetic can be derived. An axiomatic theory is said to be *recursive* if there is a computer program that can, for any formula in the language given as input, correctly answer the question: Is this formula one of the axioms of the theory? Intuitively, this means that we have a clear idea of what the axioms are to begin with. Without this requirement, one could start with all kinds of axioms that may well permit a derivation of every true sentence in arithmetic, but would be so complicated that it would be impossible to obtain proofs from the axioms, since one would never know which sentences are axioms and which are not. A simple example of this phenomenon occurs if one takes as one's set of axioms all true sentences of arithmetic!

Axioms of Peano Arithmetic

1. Every axiom of first-order logic
2. $\mathbf{s}(\mathbf{0}) = \mathbf{1}$
3. $\forall \mathbf{x} \neg(\mathbf{s}(\mathbf{x}) = \mathbf{0})$
4. $\forall \mathbf{x} \forall \mathbf{y}(\mathbf{s}(\mathbf{x}) = \mathbf{s}(\mathbf{y}) \rightarrow \mathbf{x} = \mathbf{y})$
5. $\forall \mathbf{x}(\mathbf{x} + \mathbf{0} = \mathbf{x})$
6. $\forall \mathbf{x} \forall \mathbf{y}(\mathbf{x} + \mathbf{s}(\mathbf{y}) = \mathbf{s}(\mathbf{x} + \mathbf{y}))$
7. $\forall \mathbf{x}\ (\mathbf{x} * \mathbf{0} = \mathbf{0})$
8. $\forall \mathbf{x} \forall \mathbf{y}(\mathbf{x} * \mathbf{s}(\mathbf{y}) = (\mathbf{x} * \mathbf{y}) + \mathbf{x})$
9. For each formula $\phi(\mathbf{x}, \mathbf{y})$, the following is an axiom:
   $\forall \mathbf{y}(\phi(\mathbf{0}, \mathbf{y}) \wedge \forall \mathbf{x}\ [\phi(\mathbf{x}, \mathbf{y}) \rightarrow \phi(\mathbf{s}(\mathbf{x}), \mathbf{y})] \rightarrow \forall \mathbf{x}\ \phi(\mathbf{x}, \mathbf{y})$.

The axioms indicated in Axiom 1 are the axioms from which all theorems of first-order logic—which must hold true in every model of the language of arithmetic—can be derived (we do not list these here).

Axiom 9 is the Principle of Mathematical Induction. It says that if a formula is true at 0, and if, whenever it is true at some n it can be shown to be true at $n + 1$, then the formula must be true for all natural numbers.

Suppose σ is a sentence in the language of arithmetic. We say that σ is *provable from PA*, and we write PA $\vdash \sigma$, if there are sentences $\psi_1, \ldots, \psi_n$, with $\sigma = \psi_n$, such that for each $m \leq n$, ψ_m is obtained in one of the following ways:

1. (*axiom*) ψ_m is one of the axioms;
2. (*modus ponens*) there are j, k both less than m such that ψ_k is of the form $\psi_j \rightarrow \psi_m$.

Modus ponens says that, in our derivation list of sentences, if we find ψ_j somewhere in our derivation, and somewhere else we find $\psi_j \rightarrow \psi_m$, then it is legitimate for the sentence ψ_m to be included as part of the derivation. This is the only way that new sentences are derived from other sentences, and is called *inference by modus ponens*. Modus ponens says, in essence, that if you know the sentences p and $p \rightarrow q$ are true, you may conclude that q is true too.

We can now define weak representability. As a matter of notation, let us specify some terms in arithmetic that will be useful. We will write $\mathbf{s}^2(\mathbf{0})$ for $\mathbf{s}(\mathbf{s}(\mathbf{0}))$, $\mathbf{s}^3(\mathbf{0})$ for $\mathbf{s}(\mathbf{s}(\mathbf{s}(\mathbf{0})))$, and, in general, $\mathbf{s}^{n+1}(\mathbf{0})$ for $\mathbf{s}(\mathbf{s}^n(\mathbf{0}))$. $\mathbf{s}^n(\mathbf{0})$ provides us with a natural way of representing the natural number n in the formal language of arithmetic. Now, suppose $f : \mathrm{N}^2 \to \mathrm{N}$ is a partial function. (We can provide a similar definition for any $f : \mathrm{N}^k \to \mathrm{N}$.) We say that f is *weakly representable in PA* if there is a formula $\phi(\mathbf{x}, \mathbf{y}, \mathbf{z})$ such that, for all natural numbers a, b, c,

$$c = f(a, b) \text{ iff PA} \vdash \phi(\mathbf{s}^a(\mathbf{0}), \mathbf{s}^b(\mathbf{0}), \mathbf{s}^c(\mathbf{0})).$$

Example. We give an example of a proof from PA. One of the true sentences of arithmetic—a well-known and popular fact—is that 1 + 1 = 2. Formulating this sentence in the formal language of arithmetic gives us:

$$\mathbf{1} + \mathbf{1} = \mathbf{s}^2(\mathbf{0}).$$

For practice, let us verify that this sentence is indeed true in $\mathcal{N}$. First, the term $\mathbf{1} + \mathbf{1}$ evaluates to 2, because

$$(\mathbf{1} + \mathbf{1})^{\mathcal{N}} = \mathbf{1}^{\mathcal{N}} + \mathbf{1}^{\mathcal{N}} = 1 + 1 = 2.$$

But $\mathbf{s}^2(\mathbf{0}) = \mathbf{s}(\mathbf{s}(\mathbf{0}))$ also evaluates to 2:

$$(\mathbf{s}(\mathbf{s}(\mathbf{0})))^{\mathcal{N}} = \mathbf{s}^{\mathcal{N}}(\mathbf{s}^{\mathcal{N}}(\mathbf{0}^{\mathcal{N}})) = s(s(0)) = 2.$$

Therefore,

$$(\mathbf{1} + \mathbf{1} = \mathbf{s}^2(\mathbf{0}))^{\mathcal{N}} \text{ is true.}$$

We now write a formal proof of this sentence from PA. In actual mathematical practice, proofs from any theory (PA, or any other) are written in a much more abbreviated and readable form, but can, in principle, always be transformed to a format like the following. We use a table format in order to make evident the reason for each step in the proof. The proof will demonstrate that

$$\text{PA} \vdash \mathbf{1} + \mathbf{1} = \mathbf{s}(\mathbf{s}(\mathbf{0})).$$

Statement	Reason
(1) $\forall\mathbf{x}(\mathbf{x} + \mathbf{0} = \mathbf{x})$	Axiom 5
(2) $\forall\mathbf{x}(\mathbf{x} + \mathbf{0} = \mathbf{x}) \to \mathbf{1} + \mathbf{0} = \mathbf{1}$	Axiom 1
(3) $\mathbf{1} + \mathbf{0} = \mathbf{1}$	Modus ponens, (1), (2)

Statement	Reason
(4) $\forall \mathbf{x} \forall \mathbf{y}(\mathbf{x} + \mathbf{s}(\mathbf{y}) = \mathbf{s}(\mathbf{x} + \mathbf{y}))$	Axiom 6
(5) $\forall \mathbf{x} \forall \mathbf{y}(\mathbf{x} + \mathbf{s}(\mathbf{y}) = \mathbf{s}(\mathbf{x} + \mathbf{y})) \rightarrow \mathbf{1} + \mathbf{s}(\mathbf{0}) = \mathbf{s}(\mathbf{1} + \mathbf{0})$	Axiom 1
(6) $\mathbf{1} + \mathbf{s}(\mathbf{0}) = \mathbf{s}(\mathbf{1} + \mathbf{0})$	Modus ponens, (4), (5)
(7) $\mathbf{1} + \mathbf{0} = \mathbf{1} \rightarrow \mathbf{s}(\mathbf{1} + \mathbf{0}) = \mathbf{s}(\mathbf{1})$	Axiom 1
(8) $\mathbf{s}(\mathbf{1} + \mathbf{0}) = \mathbf{s}(\mathbf{1})$	Modus ponens, (3), (7)
(9) $\mathbf{1} + \mathbf{s}(\mathbf{0}) = \mathbf{s}(\mathbf{1} + \mathbf{0}) \rightarrow \mathbf{1} + \mathbf{s}(\mathbf{0}) = \mathbf{s}(\mathbf{1})$	Axiom 1, (8)
(10) $\mathbf{1} + \mathbf{s}(\mathbf{0}) = \mathbf{s}(\mathbf{1})$	Modus ponens, (6), (9)
(11) $\mathbf{s}(\mathbf{0}) = \mathbf{1}$	Axiom 2
(12) $\mathbf{1} + \mathbf{s}(\mathbf{0}) = \mathbf{s}(\mathbf{1}) \rightarrow \mathbf{1} + \mathbf{1} = \mathbf{s}(\mathbf{1})$	Axiom 1, (11)
(13) $\mathbf{1} + \mathbf{1} = \mathbf{s}(\mathbf{1})$	Modus ponens, (10), (12)
(14) $\mathbf{s}(\mathbf{0}) = \mathbf{1} \rightarrow \mathbf{s}(\mathbf{s}(\mathbf{0})) = \mathbf{s}(\mathbf{1})$	Axiom 1
(15) $\mathbf{s}(\mathbf{s}(\mathbf{0})) = \mathbf{s}(\mathbf{1})$	Modus ponens, (11), (14)
(16) $\mathbf{1} + \mathbf{1} = \mathbf{s}(\mathbf{1}) \rightarrow \mathbf{1} + \mathbf{1} = \mathbf{s}(\mathbf{s}(\mathbf{0}))$	Axiom 1, (15)
(17) $\mathbf{1} + \mathbf{1} = \mathbf{s}(\mathbf{s}(\mathbf{0}))$	Modus ponens, (13), (16)

Having examined briefly each of five classes of functions, let us observe that all are obtained in significantly different ways, and in some cases, their relationship to the notion of computability seems remote. One of the remarkable discoveries in mathematical logic is that these five classes are all identical! We state this fact as a theorem below. Since we would like to include among these classes the class of all partial functions that are computable by a Java program, we take a moment to be more precise about this latter notion.

Suppose $f : N^k \rightarrow N$ is a partial function. We declare that f is *computable by a Java program* if, given a computer, supporting operating system, and JVM in which there is no limit on memory, we can in principle (ignoring time constraints) write a Java method m_f (which can be executed by instantiating its enclosing class and making a method call with appropriate arguments) that accepts a BigInteger array as its argument, has a BigInteger return value, and behaves in the following way: If $(a_1, a_2, \ldots, a_k)$ is in the domain of f, then, on input consist-

ing of the BigInteger array of $a_1, a_2, \ldots, a_k$, m_f returns (in principle, ignoring time constraints) the BigInteger version of $f(a_1, a_2, \ldots, a_k)$. If $(a_1, a_2, \ldots, a_k)$ is not in the domain of f, then, with input consisting of the BigInteger array of $a_1, a_2, \ldots, a_k$, m_f never returns a value (this may occur because m_f has entered an infinite loop, because an Exception has been thrown, or any other reason). Such a Java method m_f will be called a *regular Java program* (even though it is just a Java method), or the *regular Java program that computes f*.

As a simple example of a regular Java program that demonstrates computability, we show that $f(n) = 2n$ is computable:

```
BigInteger f(BigInteger[] n) {
    return n[0].add(n[0]);
}
```

We can now state our theorem (see Bell & Machover, 1977):

Unification Theorem. The following classes of functions $N^k \rightarrow N$, $k \geq 1$, are identical:

1. The class of Turing-computable partial functions.
2. The class of nondeterministic Turing-computable partial functions.
3. The class of λ-definable partial functions.
4. The class of partial recursive functions.
5. The class of partial functions that are weakly representable in Peano Arithmetic.
6. The class of partial functions that are computable by a Java program.

On the basis of evidence similar to our Unification Theorem, Church proposed (1936) that the class of all partial functions whose input/output behavior could be captured by an algorithm was also identical to each of the classes mentioned above. This proposal has come to be known as *Church's Thesis*. Since no definition of algorithm has acquired universal assent, Church's Thesis cannot actually be proved or disproved. Nonetheless, the Unification Theorem (and other even stronger results of this kind) provides compelling evidence for the truth of Church's Thesis. By now, it has achieved nearly universal acceptance

by the mathematical and computer science community, and we will take it to be true in this course.

One general feature of all the classes of functions mentioned here is that the notion of computability of a function seems always to require that it exhibit a *sequence of steps* from a well-defined starting point, in order to arrive at a "computation." This is apparent from the versions of computable function that involve some notion of a computing machine, such as Turing machines or Java programs: To compute the value of a function on any input, either the machine goes into an infinite loop (when the input is not in the domain of the function), or it performs a finite sequence of steps to arrive at an output. In the case of the classes of partial recursive functions and λ-definable functions, the derivations involved have a different flavor, but still involve finite sequences of steps. A function is partial recursive if it can be derived from a simple set of functions via finitely many applications of fundamental computable operations. In the case of λ-definable functions, all are derivable through reductions from the three basic closed terms (see above), though the steps are restricted by the need to operate on natural number inputs rather than on arbitrary domains. The sequential flow is seen in the form of finite-length proofs from Peano Arithmetic for the other two classes. Therefore, in general, a distinguishing characteristic of the class of computable functions is that they arise in the context of finite sequential unfoldment, expressed in various ways.

As we discussed in the introduction, sequential unfoldment, in the dynamics of pure consciousness according to Maharishi Vedic Science, begins with the emergence of "three" from "two"; with the emergence of Rishi, Devata, and Chhandas. For this reason, while we associated the class of definable functions with the emergence of existence and intelligence, or "two," from the unified, Samhita, value of consciousness, we associate the class of computable functions with the emergence of the flow of intelligence in the form of Rishi, Devata, and Chhandas, which gives rise to the sequential unfoldment of creation.

Is Every Definable Function Computable?

In this section, we show that the class of computable functions is a proper subset of the class of definable functions. This result shows that the requirement on a function that it should be possible to compute its

input/output behavior in finite-length computations is more stringent and demanding than the bare assertion that such a function exists (in the sense that it can be defined in the standard model). To say it another way, knowing the behavior of an existing function does not suffice to tell one how to compute its values. After showing that every computable function is indeed definable (as might be expected), we move on to an important example of a function that is definable but not computable (for a complete proof, see Keisler and Robbins, 1996).

Computable Implies Definable Theorem. Every partial function $N^k \to N$ that is computable is also definable in $\mathcal{N}$.

Partial Proof. We give a proof here for total computable functions; the proof for partial functions is considerably more difficult. Suppose $f : N^2 \to N$ is a computable total function (the more general case in which k is arbitrary is similar). By part (5) of the Unification Theorem, there is a formula $\phi(\mathbf{x}, \mathbf{y}, \mathbf{z})$ that weakly represents f in PA; namely, for all natural numbers a, b, c, we have

$$(*) \qquad c = f(a, b) \text{ iff } \mathrm{PA} \vdash \phi(\mathbf{s}^a(\mathbf{0}), \mathbf{s}^b(\mathbf{0}), \mathbf{s}^c(\mathbf{0})).$$

Recall that every sentence derivable from PA is true in $\mathcal{N}$. Also notice that for any natural number e, $\mathbf{s}^e(\mathbf{0})^{\mathcal{N}} = e$. Therefore, if $c = f(a, b)$, we have by $(*)$ that $\mathrm{PA} \vdash \phi(\mathbf{s}^a(\mathbf{0}), \mathbf{s}^b(\mathbf{0}), \mathbf{s}^c(\mathbf{0}))$. It follows that $\phi(\mathbf{s}^a(\mathbf{0}), \mathbf{s}^b(\mathbf{0}), \mathbf{s}^c(\mathbf{0}))^{\mathcal{N}}$ is true, and so $\phi^{\mathcal{N}}[a, b, c]$ is true.

On the other hand, suppose $c \neq f(a, b)$. Then, since f is total, there is some $d \neq c$ such that $f(a, b) = d$, and again we have

$$\mathrm{PA} \vdash \phi(\mathbf{s}^a(\mathbf{0}), \mathbf{s}^b(\mathbf{0}), \mathbf{s}^d(\mathbf{0})),$$

and again

$$\phi^{\mathcal{N}}[a, b, d] \text{ is true.}$$

Since $c \neq d$, it follows that

$$\phi^{\mathcal{N}}[a, b, c] \text{ is false.}$$

We have therefore shown that

$$c = f(a, b) \text{ iff } \phi^{\mathcal{N}}[a, b, c] \text{ is true,}$$

and so f is definable. □

In order to give our example of a definable function that is not computable, we develop an encoding scheme that will allow us to associate to each regular Java program an integer code, from which we will be able to reconstruct the program. We make the assumption that there are fewer than 500 distinct characters that ever occur in a regular Java program. Therefore, each character can be represented as a bit string of length 9 (since $2^9 = 512$). For the moment, let us assume that some assignment of characters to length-9 bit strings has been made. The first step of encoding a regular Java program is to minimize whitespace by removing all line breaks and reducing multiple consecutive occurrences of a blank space to a single blank space, and then to translate all characters to length-9 bit strings, and concatenating. The result will be a long bit string. We prepend the character '1' to this long bit string so that we can view this string as a binary number (the initial 1 takes care of the cases in which the long bit string happens to begin with one or more 0s).

Example. We show how to encode a simple regular Java program. There is nothing special about the way we have chosen to assign bit strings to characters. Codes have not been given for all possible characters—we have just given here enough codes to illustrate this example.

```
BigInteger f(BigInteger[] n) {
    return n[0].add(n[0]);
}
```

Char	Code	Char	Code	Char	Code
a	000000001	d	000000010	e	000000011
f	000000100	g	000000101	i	000000110
n	000000111	r	000001000	t	000001001
u	000001010	B	000001011	I	000001100
.	000001101	{	000001110	}	000001111
(	000010000	)	000010001	;	000010010
⟨space⟩	000010011	[	000010100	]	000010101

Char	Code	Char	Code	Char	Code
0	000010110				

The following bit string (including the prepended '1') represents the regular Java program above:

```
1000001011 000000110 000000101 000001100 000000111 000001001 000000011
 000000101 000000011 000001000 000010011 000000100 000010000 000001011
 000000110 000000101 000001100 000000111 000001001 000000011 000000101
 000000011 000001000 000010100 000010101 000010011 000000111 000010001
 000001110 000001000 000000011 000001001 000001010 000001000 000000111
 000010011 000000111 000010100 000010110 000010101 000001101 000000001
 000000010 000000010 000010000 000000111 000010100 000010110 000010101
 000010001 000010010 000001111.
```

We shall call the bit string code for a regular Java program P the *Gödel number of P,* and denote it #*P*. Notice that the code table can be used to reconstruct P from #*P*. Another thing to notice is that, even though the Java program we have given in the example is quite short, the bit string is rather long. This suggests that "most" (binary) integers will not be codes for any Java program. However, it is not hard to see that one could write a regular Java program IsGodel that accepts an integer (by convention, this means a BigInteger type) and outputs 1 if the binary equivalent of the integer is a Gödel number, or 0 if not. Such a program would have to embed the reconstructed Java method into a test class and attempt to compile; it would verify that that the method had an appropriate return value and an input argument of type BigInteger[].

Since all the Gödel numbers of regular Java programs are integers, they can be listed in an ordered sequence:

$$g_0 < g_1 < g_2 < \dots .$$

We will now arrange, once and for all, the regular Java programs in a sequence—programs will be arranged according to the order of their Gödel numbers. The sequence of programs

$$P_0, P_1, P_2, \dots$$

is devised so that for each integer e, P_e is the Java program whose Gödel number is g_e. Let us now make one further observation: There is a regu-

lar Java program Gen which, given any integer e, generates the code for program P_e. The program works like this: It examines the natural numbers 0, 1, 2, . . . one by one, in order, seeking Gödel numbers (it will make use of IsGodel). Each time it finds a Gödel number, it adds a tally to an initially empty list. When the number of tallies finally becomes e, this means that the program has located the Gödel number g_e. It will then reconstruct the program P_e from g_e (we mentioned above how this can be done). We will say that the number e is the *index* for the Java program P_e.

Now observe that, by the conventions we have adopted, every regular Java program computes, in one way or another, a function $N \to N$, another function $N^2 \to N$, another function $N^3 \to N$, etc. Although we may create such a program with a function $N \to N$ in mind, nonetheless, the other types of functions are implicitly defined as well. As an example, consider the code given above for computing the function $f : N \to N$ given by $f(n) = 2n$. But if we pass in a 2-element array of integers, the program will still return the result of adding the 0th array element to itself.

In other words, the program computes the function $g : N \times N \to N$ defined by $g(m, n) = m + m$. With these points in mind, we will denote the function $N^n \to N$ computed by P_e by $\phi_e^{(n)}$. The superscript will be omitted when the number of arguments of the function is clear from the context. Our earlier discussion shows that the n-argument computable function $\phi_e^{(n)}$ can be computed from e: First compute P_e from e; then for any n-ary argument $(a_1, \ldots, a_n)$, run P_e on these arguments and output the result. We will say that e is the *index* of $\phi_e^{(n)}$.

Summarizing the points above, we have the following (see Rogers, 1988, or Keisler & Robbins, 1996):

Enumeration Theorem.

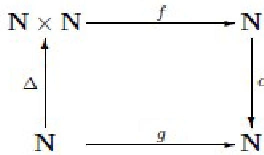
1. Every regular Java program P has a Gödel number $\#P$, which is a binary integer (sometimes identified with its base 10 equivalent).
2. There is a regular Java program IsGodel that accepts one natural number for input, and returns 1 if that number is a Gödel number, 0 otherwise.

3. The regular Java programs can be effectively arranged in a sequence; that is, we can write the regular Java programs in a sequence $P_0, P_1, \ldots, P_e, \ldots$ so that one can, with a regular Java program, generate the Java code for program P_e from the integer e.
4. For each $n \geq 1$, each of the n-argument computable functions $\phi_0^{(n)}, \phi_1^{(n)}, \ldots, \phi_e^{(n)}, \ldots$ can be computed from its index by a regular Java program.

Example: The Halting Problem. We define a function $H(x, y)$: $N^2 \rightarrow N$ as follows:

$$H(e,n) = \begin{cases} 1 & \text{if program } P_e \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

The function H tells whether a given program will halt on a given input. H is a well-defined function and we may conclude, intuitively at least, that H is definable. We will show more formally later that it is indeed definable in $\mathcal{N}$. We show here, however, that there is no algorithm that computes the values of H. We will use the Diagonalization Theorem to establish this result.



First, we create a Java program A that does the following: On any input other than 1, the program outputs the number 1. On input 1, A goes into an infinite loop. For this example, we denote by α the 1-argument function computed by A. α is the partial function $N \rightarrow N$ that is undefined at 1, but has constant value 1 for all other input values. Notice that α does not have a fixed point. Let $f(x, y) = H(y, x)$. By the Diagonalization Theorem, the function $g = \alpha \circ f \circ \Delta$ is not represented by f—that is, for each e, there is an n such that $g(n) \neq f(n, e) = H(e, n)$. Notice that Δ and α are computable.

Now let us suppose that H is computable. It follows that both f and g are also computable. Let P_e be a program that computes g. We arrive

at an absurdity in trying to determine the value of $H(e, e)$. There are two possibilities: either $H(e, e) = 1$ or $H(e, e) = 0$. We consider each separately.

If $H(e, e) = 1$, then by definition of H, P_e must halt on input e. But by definition of the program P_e, because $H(e, e) = 1$, P_e must go into an infinite loop on input e, and therefore does not halt on input e.

If $H(e, e) = 0$, then by definition of H, P_e does not halt on input e. However, by the definition of P_e, since $H(e, e) \neq 1$, P_e must halt on input e.

Either way, the conclusion is absurd. This demonstrates that the assumption that H is computable must be false.

Notice that the reasoning given above doesn't go through if we do not assume H is computable. When we make this assumption, we can find a program index e (the index for g) that leads to a paradox. But without this assumption, no such paradoxical e arises. Therefore, as a function, H is entirely legitimate; the only issue is that its behavior is not computable.

The definition of H given above suggests another function that is both computable and extremely useful. Let us define the partial function $U(x, y)$ by

$$U(e,n) = \begin{cases} \phi_e(n) & \text{if } x \text{ is in the domain of } \phi_e \\ \infty & \text{otherwise} \end{cases}$$

We have already discussed an algorithm for computing U: By part (4) of the Enumeration Theorem, there is a program which, on input e, n, outputs the value that is returned from P_e when running on input n, if P_e halts on input n; otherwise, it does not halt. U is called the *universal function for 1-argument functions*. There are similar universal functions for n-argument functions, for every n.

Using U, we can show more formally that the halting function H is definable in $\mathcal{N}$: Since U is computable, it is also definable by a formula $\phi(\mathbf{x}, \mathbf{y}, \mathbf{z})$. One can now show that H is definable in $\mathcal{N}$ by the following formula $\psi(\mathbf{x}, \mathbf{y}, \mathbf{u})$:

$$\psi(\mathbf{x}, \mathbf{y}, \mathbf{u}) : \ (\exists \mathbf{z}\ \phi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \rightarrow \mathbf{u} = \mathbf{1}) \wedge (\neg\exists \mathbf{z}\ \phi(\mathbf{x}, \mathbf{y}, \mathbf{z}) \rightarrow \mathbf{u} = \mathbf{0}).$$

The formula says that if $U(x, y)$ has a value z, then the value computed by $H(x, y)$ will be 1, whereas if $U(x, y)$ is undefined, then the

value computed by $H(x, y)$ will be 0. This correctly specifies H, and one proves fairly easily that

$$c = H(a, b) \text{ iff } \psi^N[a, b, c].$$

The Complexity of Algorithms

Once we know that a function $f : \mathrm{N}^k \rightarrow \mathrm{N}$ is computable, we can ask, "Can the input/output behavior of f be captured by an efficient algorithm?" Earlier, we considered two algorithms that implemented the function **gcd** and observed that one of them was considerably more efficient than the other. What is it that causes one algorithm to "run faster" than another? The basic measure of algorithm efficiency is the relationship between the size of the input and the number of steps required to arrive at the output. Since different inputs of size n may require a different number of steps, depending on the particulars of the input, we are most often concerned with the performance in the worst case.

Definition. The worst-case complexity of an algorithm A is a function $w : \mathrm{N} \rightarrow \mathrm{N}$ defined as follows: For each n, $w(n)$ denotes the maximum number of steps performed by A in computing (and returning) output when the size of the input is n. In other words, $w(n)$ equals the number of steps required to process an input of size n in the worst case.

A familiar example can be found among sorting algorithms. We start with a list of comparable objects (say, integers) and we ask our algorithm to produce a sorted list. The algorithm's sorting speed may depend on the initial arrangement of integers. For instance, for many sorting algorithms, if the initial list is already sorted, the output will be returned more quickly than if the initial list is ordered in some other way, say in reverse order or randomly.

In practice, it has been found that algorithms with a worst-case complexity that is not bounded by a polynomial, or indeed by a function of the form cn^k for some constants c, k, run too slowly to be useful, unless inputs are guaranteed to be of small size. In particular, if the only known algorithms for a function have a running time that is exponential in the size of the input, implementation of the function is viewed as infeasible. (See Cormen, 2001, for a full treatment of these topics.)

Definition. An algorithm is *polynomial-bounded* if there is a polynomial $p(n)$ such that for any input of size n, the algorithm returns output in fewer than $p(n)$ steps. The class of polynomial-bounded algorithms is denoted P.

In this course, we will focus on polynomial-bounded algorithms, and develop analytical tools for improving their efficiency and investigating their degree of complexity. As a secondary topic, we will take a look at some functions whose only known algorithms have exponential complexity. There is a vast collection of such algorithms. Perhaps surprisingly, some of these are known to be "more intractable" than others. Functions that admit no feasible algorithm sometimes do have a tractable feature that can be exploited.

Usually, the algorithms we study will be concerned with producing a "solution" to a problem—producing a sorted list, finding a shortest path through some data structure, optimizing a cost. Typically, then, one thinks of the class P as representing the class of all *problems* that admit a polynomial-bounded solution. Many of the "hard" problems, which do not belong to P, are known to have a special feature that makes them more tractable: if a solution to the problem is given, the number of steps required to *check that the solution is correct* is polynomial bounded. Such problems are called *nondeterministically polynomial bounded*. The class of all such algorithms is denoted NP.

It should be apparent that every problem in P also belongs to NP; however the converse inclusion is not known to be true. As we have said, the only known algorithms for many of the problems in NP are exponentially slow, but this does not eliminate the possibility that one day someone will discover a polynomial-bounded algorithm that solves the problem. It is widely believed, however, that the classes P and NP are different. One striking feature of the class NP is the existence of *NP-complete problems*—these are problems with the remarkable property that if a polynomial-bounded algorithm for the problem is ever found, there will automatically be polynomial-bounded algorithms for all problems in NP! We will discuss this remarkable phenomenon at the end of the course.

We close this section of the paper with a short list of some well-known NP-complete problems.

Subset Sum. The inputs are positive integers C, $m_1, \ldots, m_k$. The problem is: Among subsets of $\{m_1, \ldots, m_k\}$ having sum at most C, what is the largest subset sum?

CNF-Satisfiability. Consider the following language. We have a list of boolean symbols p, q, r, ... We understand these to be variables that can be assigned a value of either true or false. We have the usual connectives and ('$\wedge$'), or ('$\vee$'), and not ('$\neg$'). (For this discussion, we omit the connective "implies.") A boolean combination of boolean symbols is obtained by applying the following rules:

1. any boolean symbol is a boolean combination;
2. if A is a boolean combination, so is $\neg A$;
3. if A and B are boolean combinations, so are $A \wedge B$ and $A \vee B$.

We are interested in determining the truth value of a boolean combination, given an assignment of truth values to the boolean symbols. For instance, if we make the assignment

$$p = \text{true},\ q = \text{false},$$

what is the truth value of $\neg p \vee q$? Since p is true, $\neg p$ is false, and since q is also false, we conclude that the whole statement is false.

The problem we want to solve is this: Given a boolean combination, is there a truth assignment for the boolean symbols for which the boolean combination evaluates to true?

To make the problem more tractable, theorists observed early on that any boolean combination is equivalent to (i.e. has the same truth values as) a boolean combination in *conjunctive normal form*. A boolean combination is in conjunctive normal form if it consists of one or more Boolean combinations connected by "ands," like this:

$$A \wedge B \wedge C \wedge D \wedge \ldots.$$

Each of these component boolean combinations contains no "and" connective, and must be in the following form:

$$X \vee Y \vee Z \vee \ldots.$$

Each of the components in this latter boolean combination is either a boolean symbol or the negation of a boolean symbol.

Here is an example of a boolean combination in conjunctive normal form:

$$(p \vee q \vee \neg r) \wedge (\neg s \vee \neg p \vee t) \wedge (\neg p \vee s).$$

Finally, then, the CNF-Satisfiability Problem is the following: The input is a set of boolean symbols and a CNF boolean combination. The problem is to find a truth assignment for the boolean symbols for which the CNF boolean combination evaluates to true (or the algorithm outputs false if no such assignment exists).

Knapsack. Suppose we have a knapsack of capacity C (a positive integer) and n objects with sizes $s_1, \ldots, s_n$ and "profits" $p_1, \ldots, p_n$ (all of which are positive integers). The problem is to find the largest total profit of any subset of the object for which the sum of the sizes does not exceed C.

Traveling Salesman Problem. Suppose we have n cities. Find the shortest path that one can follow which passes through each city once and only once.

Conclusion

We have shown in this article how the class of functions that are actually used for purposes of developing software and that rest at the heart of a highly creative and rapidly developing industry represent a mere speck in a much vaster wholeness consisting of all number-theoretic functions. This fact mirrors the dynamics of the unfoldment of creation itself, as described in Maharishi Vedic Science: The infinite dynamism and creativity at the basis of the sequential unfoldment of manifest existence arises in the collapse of the unbounded silent aspect of wholeness to a point. In computer science, the unbounded value of wholeness corresponds to the class of all possible number-theoretic functions, whereas the "point" corresponds to the extremely meager and narrowly defined class of polynomial-time computable functions.

In examining the stages of "collapse" from all functions to the polynomial-time-bounded computable functions, we have also observed another theme of unfoldment that parallels the dynamics of pure con-

sciousness: Just as wholeness moves from the state of Samhita, unified consciousness or "one," to the state of "two," existence and intelligence, to the state of "three" as Rishi, Devata, Chhandas, from which all possible transformations of Samhita sequentially unfold as creation, so likewise does the "togetherness of all functions" move to the class of definable functions which display the duality of existence (each function exists as a set of ordered pairs) and intelligence (each definable function's behavior is captured in a formula), to the class of computable functions, which are characterized by being specifiable via finite-length derivations (and in the λ-calculus formulation, all derivations arise from *three* fundamental closed terms, which in turn arise from the self-interacting dynamics of the single X combinator). Moreover, this class of computable functions lies at the basis of all modern-day creative applications of computability.

These parallels serve to bring into focus the hidden dynamics of pure intelligence in the workings of the modern-day theory of computation. It suggests that much of the great power and creativity that infuses the business of building software is due to a fundamental collapse within the very structure of mathematical transformation, from an unbounded field of all possibilities to a "point value"—the tiny packet of algorithms that contains only the most efficiently computable types of transformations, which lend themselves to application in the concrete world of software development.

References

Barendregt, H.P. (1984). *The lambda calculus: Its syntax and semantics*. Amsterdam: North-Holland.

Bell, J., Machover, M. (1977). *A course in mathematical logic*. Amsterdam: North-Holland.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics, 58*, 345–363.

Cormen, T.H. (2001). *Introduction to algorithms*. Cambridge, Massachusetts: The MIT Press.

Enderton, H.B. (1972). *A mathematical introduction to logic*. California: Academic Press.

Jech, T. (1978). *Set theory*. New York: Academic Press.

Keisler, H.J., Robbins, J. (1996). *Mathematical logic and computability*. New York: McGraw-Hill.

Maharishi Mahesh Yogi (1969). *On the Bhagavad-Gita: A new translation and commentary, chapters 1–6, with Sanskrit text*. Baltimore: Penguin Press.

Maharishi Mahesh Yogi (1972). *The Science of Creative Intelligence: Knowledge and experience* (Lessons 1–33) [Syllabus of videotaped course]. Los Angeles: Maharishi International University Press.

Maharishi Mahesh Yogi (1992). *Constitution of the universe: The source of all order and harmony displayed throughout the universe*. Fairfield, Iowa: Maharishi International University Press.

Maharishi Mahesh Yogi (1996). *Maharishi's Absolute Theory of Defence*. Holland: Maharishi Vedic University Press.

Nader, T. (1995). *Human physiology: Expression of Veda and the Vedic literature*. Vlodrop, The Netherlands: Maharishi Vedic University Press.

Pierce, B.C. (1991). *Basic category theory for computer scientists.* Cambridge, Massachusetts: The MIT Press.

Rogers Jr., H. (1988). *Theory of recursive functions and effective computability.* Cambridge Massachusetts: The MIT Press. (First published 1967, Academic Press.)

Yanofsky, N.S. (2003). A universal approach to self-referential paradoxes, incompleteness, and fixed points. *The Bulletin of Symbolic Logic, 9(3)*, 362–386.