Here I present an informal constructivist approach to mathematics. Probably no single idea is original, tho I hope the synthesis is illuminating. I write these notes primarily for myself, to further develop my thinking. Some of the eccentricities found below might be explained by my attempt to circumvent platonistic set theory.

Instead of HALT, I use the more familiar RETURN.

1

Computability textbooks sometimes work in a framework of platonistic set theory. For instance, Hartley Rogers is explicit on just this point, though he doesn't emphasize the platonism.

One says "if f halts on n." I object to this because we don't, in general *know* "if" this or that function returns on this or that input. The "halting problem" suggests to me that the very notion of "halts or not" is FUZZY.

We do know, for instance, that f **did** in fact return on n. We also can see obvious "infinite loops" in some programs. So we know that f *should not* ("will not") return on the input n.

We can be certain enough of the past and the future in cases that are therefore "easy." If we see an obvious non-terminating loop in the code and the code return anyway, we will doubt the hardware or the messenger, or even (finally) our perception or memory. Likewise, some programs obviously terminate. Perhaps they don't include loops.

The tricky cases will tend be large programs with loops and branches. Here I am thinking of "tape machines," which are equivalent to Turing machines. So we are talking about arbitrarily large and complex programs. Indeed, in a vague way we are talking about "all possible programs."

I suggest that an "obviously" (and finally informally) total program/-function gives us something like the maximum of objectivity. If "the halting problem is undecidable" then "whether" a program returns is not ideally objective. For the platonist, this is not a problem, because time is an illusion and the future is already present, even if we are blind to it. The future is already present enough to function as a truth-maker. The computation has already returned or not before that computation even begun. Indeed, the program returned or ran forever before pencils were invented, before we poor humans discovered the essence or concept

of a program.

I suggest that it is total computable functions that make such platonism plausible in the first place. A even number *stays even,* because generations come and go, performing equivalent intuitively computable determinations of parity.

I don't "officially" think that truth-as-correspondence is a coherent idea, but I have the usual strong beliefs about natural numbers. So I see why parts of math are "as objective as it gets." Those ultra-objective parts are, again, intuitive computable functions. The allure of formal systems is that they scoop logic itself, as much as will fit, into total recursive functions. I can check a proof like I "verify" (confirm) that $2 + 5 = 7$.

This is one reason that the study of computability is also a study of objectivity and meaning. "Returns or not" turns out to be an iffy concept. Though the same texts that convince me of this tend to speak of it platonistically. Which is to say presuppose the solidity of its meaning, which it's just that meaning which, for me, is put into question.

Of course this is related to the LEM. I prefer to think in terms of "we have a proof" or "we don't have a proof." We also have indirect proofs that convince us, for instance, that we should give up looking for a fraction that squares to 2. Of course people can and have formalized indirect proofs, but here I'm trying to dig all the way to the philosophical informal basis of decisions about whether to trust this or that formal system. In this context, I'm talking about real-world or final-vocabulary trust.

2

For convenience, I assume that *programs are non-negative integers and that all non-negative integers are programs.* This is easy to do with Turing-Markov "tape machines." Of course it generates work for the programmer, but it simplifies the presentation here.

In this paper, I define a *function* to be a program that we believe will return on its domain. For instance, we might write a program that returns two prime numbers that sum to its input, *if* that input is even and > 2. We might not bother with the program's response to odd inputs, because, for instance, it's a subroutine. But the typical case is the domain of type N.

Instead of finished pre-present sets, we can think of types. So $n \in N$ can be reinterpreted as $n : N$, which is to say that n has the type N

and should be valid input to a function of type $f : N \to Q$, for instance. Since any finite amount of information can be squeezed into a single integer, we can stick to programs of type $f : N \to N$.

To simplify the lingo, I use *program* for any partial recursive function and reserve *function* for what we believe to be a *total* recursive functions. All functions are programs — and themselves of type N — but not all programs are functions.

Here we come to a philosophical issue. In platonistic set theory, a function is a set of ordered pairs. The same function can be computed in different ways, *if* it is computable at all.

In an analogous situation, a real number might be defined as an equivalence class of Cauchy sequences of rational numbers. But in Bishop's constructive analysis, computable Cauchy sequences *are* real numbers. Two such real numbers can be proved equivalent, but they aren't equal. Their lack of equality is defensible, in my view, as a reminder of their concrete factuality as functions $f : N \to Q$. We might be convinced readily that a particular definitions of two real numbers are valid, while their equivalence is not so obvious.

To sum up, we assume a "tape machine" programming language such that all programs are of type N and all $n : N$ are programs. We judge some of these programs to be functions, which is to say we expect them to always return for inputs in their domain. The domain D is best thought of as a subtype of N and not a set. One reason to do this is that we might not be sure whether to trust that a program p is a function. We might determine its domain experimentally. If the program returns on n, then we declare $n : D$. Intuitively, we are talking about "creative" or "temporal" sets. A "constructive set" is an *ongoing list of our results so far.*

3

We use the standard function notation $f(n)$ for the application of program f to input n. In terms of our tape machine, we place f_0 at the beginning of the input string and start the program. Depending on little choices about how exactly to define the tape machine, the computation either halts when no production applies to the state string or when a production includes an explicit RETURN signifier. In any case, it is *intuitively decidable* that a computation has returned or halted. Indeed, each step of the computation is informally or intuitively checkable.

If application of the program f starts on n and the result on the tape is r when the computation returns, then we write $f(n) = r$. If it's informally obvious to us that the application of f to n will never return, then we write $f(n) = \infty$. We could use $f(n) \uparrow$ for this, of course, but let's be finitists and make ∞ a bad thing. We write $f(n) < \infty$ if f returns and we don't care what it returns.

Now we can ask some strange questions, using our simplified situation in which every $n : N$ is at least a program.

Is there a function $f : N \to N$ such that $f(g) = 1$ if $g(g) < \infty$? And $f(g) = 0$ if $g(g) = \infty$? In other words, can we find a function that reliably decides whether *any* given procedure halts on *itself* ? Note that $g(g)$ involves the use of g as code in the first instance and as data in the second. So we are asking $f(g)$ to tell us whether or not $g(g) < \infty$, whether or not g returns on g.

More philosophy: I framed the question as *is* there such a function, as if I want to know something about *a pre-existent Platonic realm.* I did this accidentally, echoing various computability textbooks. But see my initial remarks on a non-Platonist approach.

I ask then instead: *should I bother trying to invent a function that could do that ?* This is bit like asking whether I should keep searching for $m : N$ and $n : N$ with $n \neq 0$ such that $m^2 = 2n^2$. Should I keep searching, in other words, for a fraction that squares to 2 ?

Before leaping to the more familiar argument against such a project, we can consider applying f to itself. Presumably f will be a complicated program. How do we know that f never falls into an infinite loop ? Of course we also want f to give us a correct output when it does halt. So let's consider $f(f)$.

If $f(f) = 0$, then f is lying about itself. If $f(f) = 1$, then this makes sense. This is what we hope for.

But how long do we wait for f to return ? The purpose of f is to tell us whether or not to bother waiting for *any* given function, fed its own code as date, to return. So we depend on f to tell us about whether to wait on the computation $f(f)$ to return. So we have to wait on f to see if we have to wait on f. So we have to wait on f.

Given that we hypothesize or daydream about a finite f that can answer questions about programs that are arbitrarily larger than f, we might assume that f itself is quite complicated and hard to debug. So this "waiting for f to tell us whether we should wait on f" is a practical and

not just a theoretical issue.

4

We can easily encode two integers as a single integer and stick with one-input functions. The less psychedelic function f that we might hope for would tell us whether $g(n) < \infty$. If we want to avoid talking about this extra complexity, then we can limit our interest to programs that ignore their inputs. For instance, we could write a program g that "obviously" returns iff a counterexample to goldbach's conjecture is discovered. Indeed, it returns the first even number > 2 that can't be written as the sum of primes. We'll assume that g erases its inputs and begins its search.

Our hypothetical desire function f will give us $f(g) = 1$ if g returns on every input and $f(g) = 0$ if g never returns. Of course this "returns or not" talk is subject to the criticism above. Indeed, this consideration suggests to me that Goldbach's conjecture is not "already true or false." Instead it's in a third state, which is something like **unknown** or **undecided.** The other two states are **confirmed** and **rejected.**

If $f(g) = 0$, then f is telling us that Goldbach's conjecture is "true." If $f(g) = 1$, then f is telling us that Goldbach's conjecture is "false." How long should we wait for the computation $f(g)$ to return ? We've hypothetically transformed one waiting game into another, except we assumed that f is indeed a function. So we trust that f "will" terminate. The argument that convinces us that f is indeed a function is therefore "more powerful" than an argument for Goldbach's conjecture.

Such a function f would transform every possibly infinite search into an unbounded but at least finite search. But I should note that "unbounded but at least finite" has its own issues. What we would really really like is a bounded search. "In just ≤ 10000 steps, my function f will settle the Goldbach conjecture. Indeed *every* infinite search is now reduced to a bounded finite search." The program f is "just a number," so we would have a "divine number" in our hands. A finite piece of information would allow us to answer an infinity of infinite questions.