Computable Functions

Chen Xu

June 5, 2020

Contents

1 Introduction

Computability is a branch of mathematics that emerged in the first half of the 1900s, developed by various mathematicians such as Alan Turing and Alonzo Church. In this paper, we will build a background in computability, taking asides when discussing interesting topics. This is based off of various sections of Rebecca Weber's book, *Computability Theory* [5]. We will demonstrate an application of computability theory by providing an outline as of the important steps of proving Hilbert's 10th problem, based on Davis's 1973 article, "Hilbert's 10'th problem is Unsolvable" [3]. The overall connection between Diophantine sets and computable functions will be built up, and conclude with the surprising fact that Diophantine equations are unsolvable in general.

Before proceeding, we now switch to a discussion of computability, to get build up background and intuition for how we should proceed.

2 Computability

We present relevant parts of Weber's reference on computability [5].

2.1 The Turing Machine

The Turing machine is probably the most well known idea related to computability, so no introduction of the topic can avoid it. It is the most direct way of defining computability, and is easy to see connections between it and how computers and computing machines work. The idea is that any machine has some states, and can write to some memory depending on its internal state. In theory, the most general machine could have an infinite number of states, and an infinite amount of memory — the only limiting factors would be the "code" or "instructions" that the machine had access to, and that the amount of time the machine took to ran should be finite.

The Turing machine encapsulates these ideas. Formally, a Turing machine is a finite set of 4-tuples $S \times \Sigma \times I \times S$, also known as "instructions", where S is the set of possible states the machine could be in, Σ is the alphabet of symbols that the machine uses, and I is an instruction to the machine, telling it where to move on the tape. We suppose that the machine is writing on some infinite "tape", from which it reads the symbols from Σ.

There are many choices of S, Σ, I that lead to Turing machines that can solve the same problems. In our case, for consistency, we will just consider $I = \{L, R\} \cup \Sigma$ where L denotes the instruction telling the machine to move one space left on the tape, R denotes the instruction to move one space right on the tape, $\cdot$ denotes the instruction to not move, and if an element of Σ is used, the machine will write that element to the tape. We will let $S = \mathbb{N}$, and $\Sigma = \{*, 0, 1\}$, and define the infinite tape to start as consisting of only the $*$ symbol before the machine has begun executing, unless a starting input is defined for the tape. We assume the first state of the machine is 0.

The machine executes as follows. It has some internal state $a \in S$, and reads a symbol, say $b \in \Sigma$. It then finds the first instruction (a, b, c, d) for some $c \in I$ and $d \in S$, and executes it by changing it's position on the tape by c and changing it's internal state to be d. This can best be seen with examples. We will denote the tape as $**\bullet *0101011*$ where the $\bullet$ represents where on the tape the machine is. In this paper, we will assume that the machine is reading the symbol to the right of the dot. Additionally, as the tape is infinite, we assume that the rest of the tape extends to the left and right with an infinite number of $*$ symbols.

Example 2.1. Suppose we have a machine defined by the following instructions:

$$(0, *, 0, 1)$$
$$(0, 0, R, 0)$$
$$(1, *, 0, 0)$$
$$(1, 0, L, 1)$$

At first, the tape is $*\bullet**$, with state 0. The only matching instruction is $(0, *, 0, 1)$, therefore the machine sets the value it's currently reading to 0, since the third entry of the tuple is 0, to get the new tape $*\bullet 0*$. The state of the machine then changes to 1.

Then, the execution of the machine would then continue as:

Tape	Machine State	Next Instruction to Execute
$*\bullet 0*$	1	$(1, 0, L, 1)$
$*\bullet *0*$	1	$(1, *, 0, 0)$
$*\bullet 00*$	0	$(0, 0, R, 0)$
$*0\bullet 0*$	0	$(0, 0, R, 0)$
$*00\bullet *$	0	$(0, *, 0, 1)$
$*00\bullet 0*$	1	$(1, 0, L, 1)$
$*0\bullet 00*$	1	$(1, 0, L, 1)$
$*\bullet 000*$	1	$(1, 0, L, 1)$
$*\bullet *000*$	1	$(1, *, 0, 0)$
$*\bullet 0000*$	0	$(0, 0, R, 0)$
$*0\bullet 000*$	0	$(0, 0, R, 0)$
...	...	...

As it can be seen, this machine gradually moves to the left when in state 0, until it reaches a $*$, and then after setting the tape to 0 at that spot, moves left until reaching a $*$ again, setting the tape to 0 at that spot. As a result, the machine runs forever, covering the entire tape in 0.

Example 2.2. Consider the machine defined by the instructions

$$(0, *, L, 0)$$
$$(0, 0, 1, 1)$$
$$(0, 1, 0, 3)$$
$$(2, 0, 1, 1)$$
$$(2, 1, 0, 3)$$
$$(3, 0, L, 2)$$
$$(2, *, 1, 1)$$

Then, supposing that the initial tape was $*1011 \bullet *$, the machine would execute the following instructions

Tape	Machine State	Next Instruction to Execute
$*1011 \bullet *$	0	$(0, *, L, 0)$
$*101 \bullet 1*$	0	$(0, 1, 0, 3)$
$*101 \bullet 0*$	3	$(3, 0, L, 2)$
$*10 \bullet 10*$	2	$(2, 1, 0, 3)$
$*10 \bullet 00*$	3	$(3, 0, L, 2)$
$*1 \bullet 000*$	2	$(2, 0, 1, 1)$

This machine is much more complicated to describe, but it's overall behavior is quite simple. The machine starts moving left until it finds a 0 or 1 symbol. If a 0 symbol is found, it changes it to a 1 and halts. If a 1 is found, it changes it to a 1, and sets all consecutive 1s to the left of it to 0, until it finds a non 1 symbol and sets it to a 1.

Although the machine has many instructions, and describing it's behavior is difficult, this machine can be more simply thought of as interpreting the numbers on the tape to the left of the machine as a binary number, and then incrementing the number by 1.

Note that this machine if thought of as incrementing a binary number, we can consider the evaluation of the machine given different starting tape configurations. For example, if the machine is run on a tape consisting of only $*$, the machine would stay in state 0 and run forever.

As the examples show, with the right interpretation, it seems that Turing machines might be able to implement certain mathematical functions, and depending on which instructions are used, may or may not halt. However, as constructing just a simple machine already required a significant number of instructions, for our analysis of computability, we will instead consider a different formulation of computability, namely *partial recursive functions*.

2.2 Primitive Recursive Functions

Partial recursive functions were originally introduced as primitive recursive functions. Although in the end, when we try to show that a result is not computable, we will not consider primitive recursive functions, it's interesting to consider the difficulties when defining computability. Furthermore, most common functions that are one would think are computable are captured by primitive recursive functions.

We also make use of the following notational choices. We use bold faced font to represent tuples, for example $\mathbf{x} \in \mathbb{N}^n$, and subscripts to represent entries in the tuple, e.g. $\mathbf{x} = (x_1, \ldots, x_n)$. If $f : \mathbb{N}^m \to \mathbb{N}$ where $m \geq n$, then we define

$$f(\mathbf{x}, y_1, \ldots, y_{m-n}) = f(x_1, \ldots, x_n, y_1, \ldots, y_{m-n})$$

Definition 2.3. As defined in Weber [5], the class of *primitive recursive functions* C is the smallest class of functions $f : \mathbb{N}^n \to \mathbb{N}$ for some $n \in \mathbb{N}$ with the following properties

1. The successor function is primitive recursive. $S(x) = x + 1$ is in C.
2. The constant function is primitive recursive. For any $n, m \in \mathbb{N}$ the function $M(x_1, x_2, \ldots, x_n) = m$ is in C.
3. The projection functions are primitive recursive: $U_i^n(x_1, x_2, \ldots, x_n) = x_i$, are in C.

4. The composition of primitive recursive functions is primitive recursive: If $f \in C$ and $g_i \in C$, where $g_i : \mathbb{N}^n \to \mathbb{N}$, then $f(g_1(\mathbf{x}), \dots g_m(\mathbf{x}))$ is also in C.

5. Functions can be primitively recursive. If $g, h \in C$, $g : \mathbb{N}^n \to \mathbb{N}$, and $h : \mathbb{N}^{n+2} \to \mathbb{N}$, then the function $f : \mathbb{N}^{n+1}$ defined by

$$f(\mathbf{x}, 0) = g(\mathbf{x})$$
$$f(\mathbf{x}, y+1) = h(\mathbf{x}, y, f(x_1 \dots, x_n, y))$$

is also primitively recursive.

The last property is particularly confusing, and it's not immediately obvious why these rules are related to computability. We provide some examples of primitive recursive functions.

Example 2.4. Addition is primitive recursive.

Proof. If $f(x, y) = x + y$, then we can construct f using the rules as follows. Using property 5, we can define f primitively recursively, such that

$$f(x, 0) = h(x, 0) = x$$

where $h(x_1, x_2) = x_1$. We can then use primitive recursion to calculate $f(x, y+1)$:

$$f(x, y+1) = g(x, y, f(x, y)) = S(f(x, y))$$

□

Example 2.5. Decrementing by 1 is primitive recursive. More precisely

$$f(x) = \begin{cases} 0 & x = 0 \\ x - 1 & \text{otherwise} \end{cases}$$

is primitive recursive.

Proof. The only way to decrement anything is to use primitive recursion. Applying primitive recursion, we get:

$$f(0) = M(0) = 0$$
$$f(x+1) = U_1^1(x) = x$$

□

Example 2.6. Subtraction is primitive recursive. Since the domain and range of all functions are natural numbers, we mean subtraction as in the function:

$$f(x, y) = x \dot{-} y = \begin{cases} x - y & x > y \\ 0 & \text{otherwise} \end{cases}$$

Proof. Let $d(x)$ be the decrement function defined in example 2.5. Using primitive recursion, we let $f(x, 0) = d(x) = \max\{0, x - 1\}$ and define

$$h(x, y, z) = d(U_3^3(x, y, z)) = \max\{z - 1, 0\}$$

as we know addition is primitive recursive by example 2.4. Working through the primitive recursion, we see that

$$f'(x, y+1) = h(x, y, f(x, y)) = d(\max\{x - y, 0\}) = \max\{x - y - 1, 0\}$$

which is the desired result. □

Example 2.7. The step function is primitive recursive. Given some number $a \in \mathbb{N}$, the function

$$f(x) = \begin{cases} 0 & x \leq a \\ 1 & \text{otherwise} \end{cases}$$

Similarly, the function

$$f'(x, y) = \begin{cases} 0 & x \leq y \\ 1 & \text{otherwise} \end{cases}$$

Proof. Using the subtraction function defined in example 2.6, we see that we can express f as

$$f(x) = (x \dot{-} a) \dot{-} ((x \dot{-} a) \dot{-} 1)$$

If $x \leq a$, then $x \dot{-} a = 0$ and $0 \dot{-} y = 0$ for all y. If $x > a$, then $(x \dot{-} a) = x - a \geq 1$, so $(x \dot{-} a) \dot{-} 1 = x - a - 1$. As $x - a > x - a - 1$, we get $f(x) = 1$ in this case.

We can use a similar construction for f':

$$f'(x, y) = (x \dot{-} y) \dot{-} ((x \dot{-} y) \dot{-} 1)$$

□

Example 2.8. The multiplication function is primitive recursive.

Proof. Let $f(x, y) = xy$. Using primitive recursion in the same way as the addition function, we can construct it as follows. First defining a base case

$$f(x, 0) = 0$$

as the constant functions are primitive recursive.

The recursive step, we can define

$$f(x, y + 1) = g(x, y, f(x, y)) = x + f(x, y)$$

as addition is primitive recursive, and we can compose primitively recursive functions. More explicitly

$$g(x, y, a) = \text{add}(U_1^3(x, y, a), U_3^3(x, y, a))$$

□

Example 2.9. The modulo function is primitive recursive.

Proof. Let $f(x, y) = x \bmod (y + 1)$. Note that

$$f(x, y) = \begin{cases} x & & x < y \\ \begin{cases} f(x - 1, y) + 1 & x \neq y \\ 0 & \text{otherwise} \end{cases} & & \text{otherwise} \end{cases}$$

We can represent functions of the form

$$g(x, a) = \begin{cases} g_1(x, a) & x < a \\ g_2(x, a) & x \geq a \end{cases}$$

by using the step function in example 2.7. If q is the step function, then

$$g(x, a) = q(x, a)g_1(x, a) + (1 \dot{-} q(x, a))g_2(x, a)$$

Additionally, the step function allows us to produce functions depending on the equality of two inputs. The function

$$h(x, y) = \begin{cases} 0 & x \neq y \\ 1 & \text{otherwise} \end{cases} = (1 \dot{-} q(x, y))(1 \dot{-} q(y, x))$$

can be used in a similar way as the step function. Thus, we can represent each of the cases required in the modulo function. □

As can be seen, primitive recursion is similar to looping in some sense, or tail recursion in programming. This is also very similar to the fold function used oftentimes in functional programming languages.

2.3 The Ackermann Function is not Primitive Recursive

Primitive recursive functions are not as powerful as Turing machines. Although they can emulate tail recursion, primitive recursion is not the same as full recursion. The canonical example is the Ackermann function. We use the definition provided by Weber [5]:

$$A(x,y) = \begin{cases} y+1 & x = 0 \\ A(x-1,1) & x > 0 \text{ and } y = 0 \\ A(x-1, A(x,y-1)) & \text{otherwise} \end{cases}$$

On a basic level, it is clear this function can be easily implemented in a program, and therefore implementable on a Turing machine. As a result, when discussing computable functions, we'd like our definition to allow for Ackermann functions, but it's not immediately clear why primitive recursion doesn't handle the Ackermann function either.

The proof is straightforward. The main idea is that the Ackermann function can be shown to be "larger" than every primitive recursive function. More formally, we need the following concept

Definition 2.10. A function $g : \mathbb{N}^2 \to \mathbb{N}$ *majorizes* another function $h : \mathbb{N}^n \to \mathbb{N}$ if there exists some $a \in \mathbb{N}$ such that for all $b \in \mathbb{N}$, if $\max\{x_1, x_2, \ldots, x_n\} < b$:

$$g(a,b) > h(x_1, x_2, \ldots, x_n)$$

Note that majorization is oftentimes defined more generally, but this is sufficient for this paper.

Additionally, we need a couple of simple but useful properties about the Ackermann function. None of them are particularly difficult to prove, and they are all fairly mechanical, so we do not provide proofs for all of them. The following lemmas are taken from [2].

Lemma 2.11. $A(x,y)$ is defined for all $x, y \in \mathbb{N}$.

This lemma will not be referenced explicitly, but is required for the rest of the analysis about Ackermann functions to be justified.

Proof. We use induction on x. By definition, $A(0,y)$ is defined for all y. Now, inductively assume that $A(x,y)$ is defined for all y. Then for any $y \neq 0$,

$$A(x+1,y) = A(x, A(x,y))$$

Since $A(x,y)$ is defined, we can say $z = A(x,y)$. $A(x,z)$ is defined for our inductive hypothesis. Thus $A(x,y)$ is defined for all x, y. □

Lemma 2.12. $A(1,y) = y + 2$

Proof. This can be shown by induction. For the base case, if $y = 0$

$$A(1,0) = A(0,1) = 2 = y + 2$$

Inductively assuming $A(1,y) = y + 2$, we note that

$$A(1,y+1) = A(0, A(1,y)) = A(0, y+2) = y + 3$$

□

Lemma 2.13. $A(2,y) = 2y + 3$

Proof. We induct on y. The base case:

$$A(2,0) = A(1,1) = 1 + 2 = 3$$

by lemma 2.12. Assuming the claim is true for $y - 1$

$$A(2,y) = A(1, A(2,y-1)) = A(1, 2y+1) = 2y + 3$$

by lemma 2.12 again. □

Lemma 2.14. $y < A(x, y)$

Proof. We proceed by induction on x. If $x = 0$, then $A(0, y) = y + 1 > y$. Now inductively assuming $A(x, y) > y$:

$$A(x + 1, y) > A(x, y) > y$$

□

Lemma 2.15. $A(x, y) < A(x, y + 1)$

Lemma 2.16. $A(x + 1, y) \leq A(x, y + 1)$

Lemma 2.17. $A(x, y) < A(x + 1, y)$

Lemma 2.18. $A(x, A(y, z)) < A(x + y + 2, z)$

Proof. Using lemmas 2.17 and 2.15 we repeatedly, we get that

$$A(x, A(y, z)) < A(x + y + 1, A(x + y, z)) = A(x + y + 1, z + 1) \leq A(x + y + 2, z)$$

where the last inequality follows by lemma 2.16 □

We now proceed with the proof, following [1].

Theorem 2.19. The Ackermann function majorizes every primitive recursive function.

Proof. We proceed by checking each way of constructing a primitive recursive function, and show that it is majorized by the Ackermann function. Throughout the proof, out of convenience, if $\mathbf{x} \in \mathbb{N}^n$, then we let $\widetilde{x} = \max\{x_1, \ldots, x_n\}$.

1. The successor function is majorized by the Ackermann function:

   $$S(x) = x + 1 < x + 2 = A(1, x)$$

   where the last equality follows by lemma 2.12

2. The constant functions are majorized by the Ackermann function Let $C(\mathbf{x}) = c$. As $g(x) = A(x, 0)$ is an increasing function by lemma 2.17, there exists some $r \in \mathbb{N}$ such that $g(r) > c$. Thus, $C(\mathbf{x}) < A(r, 0)$, and by lemma 2.15 we get that $A(r, 0) < A(r, \widetilde{x})$. Combining the inequalities, we get that $C(\mathbf{x}) < A(r, \widetilde{x})$, where r does not depend on $\mathbf{x}$.

3. The projection functions are majorized by the Ackermann function. If we let $U_i^n(\mathbf{x}) = x_i$, we note that

   $$U_i^n(x_1, \ldots, x_n) = x_i \leq \widetilde{x} < \widetilde{x} + 1 = A(0, \widetilde{x})$$

4. The composition of majorized functions is majorized by the Ackermann function. Let h, g_i be arbitrary primitive recursive functions that are majorized by the Ackermann function, and $\mathbf{x} \in \mathbb{N}^n$. Define

   $$f(\mathbf{x}) = h(g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$$

   Since g_i is majorized by A, there exists some r_i such that $g_i(\mathbf{x}) < A(r_i, \widetilde{x})$. Let $\mathbf{g}(\mathbf{x}) = (g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$. Since h is majorized by A, there exists s such that $h(\mathbf{g}) < A(s, \widetilde{g})$. Note that $\widetilde{g} < \max_i A(r_i, \widetilde{x})$, and we can use lemma 2.17 to conclude that $\widetilde{g} < A(\widetilde{r}, \widetilde{x})$. Using lemma 2.15, we get that $A(s, \widetilde{g}) < A(s, A(\widetilde{r}, \widetilde{x}))$. By lemma 2.18 we see that $A(s, A(\widetilde{r}, \widetilde{x})) < A(s + \widetilde{r} + 2, \widetilde{x})$. Since $s, \widetilde{r}$ both only depended on h and g_i, and not $\mathbf{x}$, we can conclude that f is majorized by A.

5. Primitive recursion of functions majorized by the Ackermann function results in a function still majorized by the Ackermann function. This section closely follows [2], but provides more detail.

   To see this, let g, h be some primitive recursive functions, and define f primitive recursively:

$$f(\mathbf{x}, 0) = g(\mathbf{x})$$
$$f(\mathbf{x}, y+1) = h(\mathbf{x}, y, f(\mathbf{x}, y))$$

   Since g, h are majorized by A, choose r, s such that $g(\mathbf{x}) < A(r, \widetilde{x})$ and $h(\mathbf{z}) < A(s, \widetilde{z})$. For any $x, y \in \mathbb{N}$, let $q = 1 + r + s$. We now show that $f(\mathbf{x}, y) < A(q, \widetilde{x} + y)$

   The result is trivial for $f(\mathbf{x}, 0)$, as

$$f(\mathbf{x}, 0) = g(\mathbf{x}) < A(r, \widetilde{x}) < A(q, \widetilde{x}) < A(q, \widetilde{x})$$

   where the last two inequalities follow by lemmas 2.17 and 2.15 respectively.

   We now proceed by induction, assuming that $f(\mathbf{x}, y) < A(d, \mathbf{x} + y)$. For $f(\mathbf{x}, y+1)$, we apply primitive recursion to get

$$f(\mathbf{x}, y+1) = h(\mathbf{x}, y, f(\mathbf{x}))$$

   and since h is majorized, we get that $f(\mathbf{x}, y+1) < A(s, \max\{\widetilde{x}, y, f(\mathbf{x}, y)\})$. Using lemma 2.14, we see that $\widetilde{x} + y < A(q, \widetilde{x} + y)$, and since $f(\mathbf{x}, y) < A(q, \widetilde{x} + y)$ by our inductive hypothesis, we have that $\max\{\widetilde{x}, y, f(\mathbf{x}, y)\} < A(q, \widetilde{x} + y)$. Thus we can use lemma 2.15 to get

$$f(\mathbf{x}, y+1) < A(s, A(q, \widetilde{x} + y))$$

   $q - 1 \geq s$, therefore applying lemma 2.17 and then the third case of the definition of the Ackermann function, we get

$$A(s, A(q, \widetilde{x} + y)) \leq A(q-1, A(q, \widetilde{x} + y)) = A(q, \widetilde{x} + y + 1)$$

   completing the induction.

   From this, it is straightforward to get the desired result. Using lemma 2.15 we can get the following inequality

$$f(\mathbf{x}, y) < A(q, \widetilde{x} + y) \leq A(q, 2\max\{\widetilde{x}, y\}) < A(q, 2\max\{\widetilde{x}, y\} + 3)$$

   and then using lemma 2.13 and 2.18 we get the desired result

$$= A(q, A(2, \max\{\widetilde{x}, y\})) < A(q+4, \max\{\widetilde{x}, y\})$$

   since q did not depend on $\mathbf{x}$ or y.

□

2.4 Partial Recursive Functions

However, we'd certainly like the Ackermann function to be in our class of computable functions — after all it's composed of functions that intuitively could be easily computed on a computer. The problem with primitive recursive functions is that the number of applications of primitive recursions is fixed, which in some sense corresponds to only being allowed to use a fixed number of loops in a program. However, the Ackermann function uses a number of recursions depending on the input. This can be fixed by adding the following rule

Definition 2.20. A *partial recursive function* (or in some sources, *general recursive function* or *μ-recursive functions*) is the smallest class of functions composed of primitive recursive functions, and also functions that can be composed with μ-recursion:

If $f(x_1, \ldots, x_n, y)$ is a partial recursive function, we are allowed to construct a new partial recursive function $g(x'_1, \ldots, x'_n) = y'$ where y' is the smallest y' such that $f(x'_1, \ldots, x'_n, y') = 0$.

The name "partial" recursive is a reference to the fact that if partial recursive functions may not be defined for some elements in the domain, because of μ-recursion.

Example 2.21. Division is partial recursive.

Proof. Division is primitive recursive so this is overkill, but we use this as an example of μ-recursion. As we know $f(x, y) = xy$ is partial recursive, we can define $g(x, y) = \lceil \frac{x}{y} \rceil = z$ where z is the smallest value such that $f(z, y) > x$, where we can check $f(z, y) > x$ using the step function. □

Example 2.22. The function $g(x) = y$ where y is the smallest value such that $f(x, y) = x + y = 0$ is not defined when $x \neq 0$. This is because the domain of all primitive recursive functions is $\mathbb{N}$, therefore when $x > 0$, then $x + y = 0$ implies $y < 0$, which means $y \notin \mathbb{N}$.

Like in Weber [5], we will use in this paper $f(x) \uparrow$ to denote when $f(x)$ is not defined, or does not "halt" on an input x, and $f(x) \downarrow$ to indicate f does halt. As this turns out to be useful in the future, we introduce the following

Definition 2.23. A function is said to be *total* if it halt's on all inputs in it's domain.

Note that all primitive recursive functions are total.

Surprisingly, this addition of μ-recursion to primitive recursive functions makes this class equivalent to Turing machines in some sense, as will be elaborated on in the following theorem.

Theorem 2.24. The class of partial recursive functions are equivalent to the class of functions that a Turing machine could emulate.

Proof. A real proof of this statement would require explicitly constructing Turing machines that can emulate each of the ways to construct partial recursive functions, e.g. the successor function, projection functions, constant functions, primitive recursion, etc. We've shown a version of the successor function can be represented by a Turing machine in example 2.2. Additionally, formalizing this is tedious, which isn't surprising since the construction of simple Turing machines were already somewhat complicated, so we will just provide an interesting idea relating to the proof.

In particular, as Weber [5] mentions in her description of this proof, one particularly interesting method when showing that a function represented by a Turing machine can be emulated by partial recursive functions is to represent the Turing machine as the set of binary digits to the left and right of the machine's current position. Then, instructions moving the machine left and right correspond to dividing the binary number to the left and right by 2 or multiplying by 2 respectively. Division can be represented as a primitive recursive function fairly straightforwardly. □

One issue that theorem 2.24 introduces is if and how computability should be defined on sets that aren't natural numbers. It seems reasonable that the function $f(x) = -x$ defined in $\mathbb{Z}$ should be computable, as a Turing machine representing signed integers isn't difficult to conceive of. We address this idea by considering the following term

Definition 2.25. A set S is *effectively countable* if there exists a computable function with computable inverse between S and $\mathbb{N}$. The computable function mapping $S \to \mathbb{N}$ is may be referred to as a *coding function.*

This notion helps expand our notion of what computable functions can be defined on, but doesn't resolve it. However, this does provide methods for us to analyze if subsets of $\mathbb{N}$ and $\mathbb{N}^n$ are effectively countable.

Effective countability is useful in particular as if S is effectively countable, we can easily define computable functions operating on S by first using a computable function to map S to $\mathbb{N}$, then using a computable function on $\mathbb{N}$.

One nice thing about primitive recursive functions is the following

Lemma 2.26. Any primitive recursive function between $\mathbb{N}^n$ and $\mathbb{N}$ that is a bijection has a computable inverse.

Proof. Let $f : \mathbb{N}^n \to \mathbb{N}$ be a bijective primitive recursive function. Let

$$g_n(y, x_1, \ldots, x_n) = \begin{cases} 0 & f(\mathbf{x}) = y \\ 1 & \text{otherwise} \end{cases}$$

We know g_n is primitive recursive as example 2.9 demonstrated that defining a function dependent on the equality of two inputs is primitive recursive. Now we can define functions using μ-recursion. Let $g_k(y, x_1, \ldots, x_k) = z_k$ be the smallest value z_k such that $g_{k+1}(y, x_1, \ldots, x_k, z_k) = 0$. Importantly, we know z_k must exist, since as f has an inverse, there exists a unique $a_1, \ldots, a_n$ such that $f(\mathbf{a}) = y$. Therefore, as $\mathbf{a}$ is unique, $g_{n-1}(y, a_1, \ldots, a_{n-1}) = a_n$, and using induction, we deduce that $g_k(y, a_1, \ldots, a_k) = a_{k+1}$ by the uniqueness of $\mathbf{a}$ and y. Note that the $(g_0, \ldots, g_{n-1})$ form an inverse for f, and are partial recursive. □

This allows us to show several useful sets are effectively countable.

Theorem 2.27. The set $\mathbb{N}^n$ is effectively countable.

Proof. First, we show that the set $\mathbb{N}^2$ is effectively countable. We consider the function

$$f(x, y) = \frac{(x + y + 1)(x + y)}{2} + y$$

Clearly f is primitive recursive. Note that since $\frac{n(n+1)}{2}$ is an increasing function of n, there exists a unique n such that $\frac{n(n+1)}{2} \leq z < \frac{(n+1)(n+2)}{2}$. The, given n, there exists a unique y such that $z + y = \frac{(n+1)(n+2)}{2}$, and $x = n + 1 - y$. Since a unique x, y can be chosen for any z such that $f(x, y) = z$, a bijection exists. Using lemma 2.26, we can get a computable inverse.

Now, we note that for any effectively countable sets S, U, $S \times U$ is also effectively countable as we can choose computable functions $g : S \to \mathbb{N}$ and $h : U \to \mathbb{N}$, to get a computable function from $S \times U \to \mathbb{N}^2$. As $f : \mathbb{N}^2 \to \mathbb{N}$, we get $f \circ (g, h) : S \times U \to \mathbb{N}$, and the inverse is $(g^{-1}, h^{-1}) \circ f^{-1}$, which is clearly computable. Therefore, $(\mathbb{N} \times \mathbb{N}) \times \cdots \times \mathbb{N}$ is also effectively countable, so $\mathbb{N}^n$ is effectively countable. □

Remark 2.28. Using theorem 2.27, we note that any computable function $f : \mathbb{N}^n \to \mathbb{N}$ can be decomposed into $g \circ h$ where $g : \mathbb{N} \to \mathbb{N}$, and $h : \mathbb{N}^n \to \mathbb{N}$ is the coding function for $\mathbb{N}^n$. In this case, $g = f \circ h^{-1}$. As a result, we only need to study computable functions of one variable.

Theorem 2.29. The set of tuples of any length are effectively countable. More specifically

$$S = \bigcup_{k=1}^{\infty} \mathbb{N}^k$$

is effectively countable.

Proof. The appropriate computable function mapping $S \to \mathbb{N}$ is

$$f(x_1, \ldots, x_k) = 2^{x_1} + 2^{x_1 + x_2 + 1} + \cdots + 2^{x_1 + \cdots + x_k + k - 1}$$

If we think about the binary representation of $f(\mathbf{x})$, we note that there are exactly k ones. Therefore, a k'-tuple cannot map to the same value as any k-tuple when $k \neq k'$, so f is injective. Additionally, if $x_i \neq x_i'$, then the $2^{x_1 + \cdots + x_i + i - 1}$ terms will differ, and since that's the i'th smallest power of 2 for both $f(\mathbf{x})$ and $f(\mathbf{x}')$, that means $f(\mathbf{x}) \neq f(\mathbf{x}')$. Therefore, f is injective.

Now we show f is surjective. For any $z \in \mathbb{N}$, z has a binary representation, say with $z = \sum_{i=1}^{k} 2^{a_i}$. If we let $B\mathbf{x} = \mathbf{a}$, where B represents the system of equations

$$\begin{aligned} x_1 &= a_1 \\ x_1 + x_2 &= a_2 - 1 \\ &\cdots \\ x_1 + \cdots + x_k &= a_k - k + 1 \end{aligned}$$

then B is lower triangular, and therefore a unique solution for $\mathbf{x}$ exists. □

The set of tuples of any length is extremely useful, because it provides a direction for us to show that the set of Turing machines is effectively countable.

Theorem 2.30. The set of all Turing machines is effectively countable. More precisely, the set

$$W = \bigcup_{k \geq 0} (S \times \Sigma \times I \times S)^k$$

is effectively countable.

Proof. We do not prove this formally, but instead provide a sketch. $S \times \Sigma \times I \times S$ is effectively countable, since $S = \mathbb{N}$ and $\Sigma \times I$ is a finite set. As a result, there exists a natural computable function with computable bijection between $S \times \Sigma \times I \times S$ with $\mathbb{N}^2 \cup \cdots \cup \mathbb{N}^2$. We can then use the coding function for this set, call the function f, and given a k-tuple of instructions, we map it to a $\mathbb{N}^k$ by applying f to each instruction. Then, we effectively have a computable function and computable inverse between W and $\bigcup_{k \geq 0} \mathbb{N}^k$, and we can apply theorem 2.29. □

Let σ be the function mapping Turing machines to $\mathbb{N}$. Since σ^{-1} is computable, we can construct a partial recursive function $f : \mathbb{N} \to \mathbb{N}$ that given some index n, computes the Turing machine associated with n using σ^{-1}, and then executes it, which is possible by theorem 2.24. We call f a universal Turing machine:

Definition 2.31. A *universal Turing machine* is a Turing machine that can simulate any other Turing machine.

Additionally, as we will oftentimes want to refer to Turing machines as elements of $\mathbb{N}$

Definition 2.32. The *index* of a partial recursive function f will be used to denote a number $i \in \mathbb{N}$ that is the result of the coding function on a Turing machine that computes the same function as f. We will then use φ_i to denote the function represented by the Turing machine with index i.

One useful thing to keep in mind about indexes of Turing machines is that they are not unique.

Lemma 2.33. (Padding lemma) If the Turing machine M is encoded by $i \in \mathbb{N}$, there exists a larger j that encodes the same function.

Proof. Note that the Turing machine M consists of some finite number of instructions. Suppose the maximum state value used by the instructions is q, then we could add the instruction $(q+1, *, R, q+1)$ to M to get a different machine and different index, but it would represent the same function as M never enters $q+1$. □

Now, perhaps the most famous result of computability theory is the halting problem. In the language we've been using, this is captured by the following ideas.

Definition 2.34. A set is *computable* if it's characteristic function is computable. For example, S is computable if the function

$$f(x) = \begin{cases} 1 & x \in S \\ 0 & x \notin S \end{cases}$$

is computable.

Theorem 2.35. The *halting set* is not computable. The halting set is

$$K = \{e : \varphi_e(e) \downarrow\}$$

or the set of functions that halt on it's index.

Proof. Let f be the characteristic function of K. Suppose for sake of contradiction that f were computable. Note that f must be total if it were computable. Define

$$g(e) = \begin{cases} \varphi_e(e) + 1 & f(e) = 1 \\ 1 & f(e) = 0 \end{cases}$$

f being computable would imply that g is also computable, since equality of a computable function is computable and $\varphi_e(e)$ is computable if $f(e) = 1$. However, then we note that g being computable means there must have some index e' such that $\varphi_{e'} = g$. Note that g is defined on all inputs, and therefore is total. Now consider $g(e')$. $f(e') = 1$ thus $\varphi_{e'}(e') \downarrow = g(e') \downarrow$, so $g(e') = \varphi_{e'}(e') + 1 = g(e') + 1$, which is a contradiction. □

3 Hilbert's 10th Problem

We now try to apply the ideas we've built up discussing computability theory. The structure and all of the results in this section can be attributed to Davis's proof of Hilbert's 10th problem [3]. First, we introduce Diophantine equations.

Definition 3.1. A *Diophantine equation* refers to a polynomial with a finite number of variables that only has integer coefficients. For some Diophantine equation $P(x_1, x_2, \ldots, x_m) = 0$, we say that $(x_1, x_2, \ldots, x_m)$ is a solution of P if each of the $x_i \in \mathbb{N}$ and the polynomial is satisfied.

The 10th problem, then as originally stated by a translation of Hilbert's original work [4] is as follows:

> Given a diophantine equation with any number of unknown quantities and with rational integral numerical coefficients : *To devise a process according to which it can be determined by a finite number of operations whether the equation is solvable in rational integers*

3.1 Background on Diophantine Related Ideas

The way Davis [3] goes about proving this is to first consider two related ideas to Diophantine equations. We will only consider Diophantine equations with solutions that are positive. This is without loss of generality, because by Lagrange's four-square theorem, any positive integer can be represented by the sum of four integer squares. Thus, the polynomial $P(x_1, \ldots, x_n) = 0$ has positive solutions if and only if the polynomial $Q(\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{d}) = P(a_1^2 + b_1^2 + c_1^2 + d_1^2, \ldots)$ has integer solutions.

Definition 3.2. A set S is *Diophantine*, or called a *Diophantine set* if it's only composed of n-tuples $(x_1, \ldots x_n)$ such that each x_i is a positive integer and there exists a polynomial P such that for any $(x_1, \ldots, x_n) \in S$, there exists a corresponding m-tuple $(y_1, \ldots, y_m)$ such that each y_i is a positive integer and

$$P(x_1, \ldots x_n, y_1, \ldots y_m) = 0$$

where n, m, P only depend on S.

Throughout this paper, when trying to show that a given set is Diophantine, I will call P the *associated polynomial* of S, although that terminology is nonstandard.

We now provide some informative examples of Diophantine sets. Unless otherwise mentioned, every variable used is assumed to be a positive integer. We start with examples from Davis [3].

Example 3.3. The set of positive even numbers is Diophantine.

Proof. This can be seen by considering the polynomial $P(x, y) = x - 2y$. If $x \in S$, $x = 2j$ for some j, therefore we can choose $y = j$ and get that $P(x, y) = x - 2j = 0$. Similarly, if given some x, we can choose some y such that $x - 2y = 0$. In other words, $x = 2y$ for some $y \in \mathbb{N}$, which means x is even. □

Example 3.4. The set of pairs of positive integers (x_1, x_2) such that $x_1 \leq x_2$ is Diophantine.

Proof. In this case, we consider the polynomial $P(x_1, x_2, y_1) = x_1 - x_2 + y_1 - 1$. If $x_1 \leq x_2$, then $x_2 - x_1 \geq 0$, so we can choose $y_1 - 1 = x_2 - x_1$ to satisfy the equation. Conversely if the equation is satisfied, then as $y_1 - 1 \geq 0$, we know that $x_1 - x_2 + y_1 - 1 = 0$ implies that $x_1 = x_2 - y_1 + 1 \leq x_2$, so the ordering relation is preserved. □

Example 3.5. Any finite set of numbers $S = \{s_1, \ldots, s_n\}$ is Diophantine.

Proof. For finite sets, we can encode the information about the set inside the polynomial. Thus, the appropriate polynomial is $P(x) = (x - s_1)^2(x - s_2)^2 \dots (x - s_n)^2$. Clearly, the only zeros of P are when $x \in S$. □

Example 3.6. If we have multiple Diophantine sets $S_1, \dots S_n$, then their intersection and union are Diophantine.

Proof. Suppose the corresponding polynomial to S_i is P_i. Then, the corresponding polynomial to $S_1 \cap \dots \cap S_n$ would be $P = P_1^2 + P_2^2 + \dots + P_n^2$, as P is only satisfied if all of $P_i = 0$.

The corresponding polynomial to $S_1 \cup \dots \cup S_n$ would be $P = P_1 P_2 \dots P_n$, since P is satisfied if any of $P_i = 0$. □

There are parallels between constructing Diophantine sets and defining sets with predicate logic. Suppose we define sets using predicate logic, for example for some $a \in \mathbb{N}$.

$$S = \{(x_1, x_2) : -a \leq x_2 - x_1 \wedge x_2 - x_1 \leq a\}$$

We can encode the "and" restriction by by considering the intersection of sets $S_1 = \{(x_1, x_2) : -a \leq x_2 - x_1\}$, and $S_2 = \{(x_1, x_2) : x_2 - x_1 \leq a\}$, and get $S = S_1 \cap S_2$, and similarly we can encode "or" operations with unions. Additionally, we can encode "not" conditions via set complement.

The following example demonstrates the parallel more explicitly

Example 3.7. The following set is Diophantine

$$S = \{(x_1, x_2) : (\exists y_1, y_2) : x_1 y_1 = x_2 \wedge y_1 \neq 1 \wedge y_1 = y_2^2\}$$

Proof. We note that it's straightforward to implement each restriction individually as polynomials. The first restriction already is a polynomial, so we let $P_1(x_1, x_2, y_1, y_2) = x_1 y_1 - x_2$. The second restriction can be resolved by noticing $y_1 \neq 1$ implies $(y_1 - 1)^2 > 0$, therefore we can introduce a third variable and note that

$$(\exists y_3) : (y_1 - 1)^2 - 1 - y_3 = 0 \iff y_1 \neq 1$$

so we can define a polynomial $P_2(y_1, y_3) = (y_1 - 1)^2 - 1 - y_3$ to represent the restriction. The third restriction is also already a polynomial, so it clearly is satisfied if and only if $P_3(y_1, y_2) = y_1 - y_2^2 = 0$

As mentioned before, one of the techniques we can use to take the logical and of all the restrictions is by adding the corresponding polynomials, letting us conclude that the associated polynomial for S is $P(x_1, x_2, y_1, y_2, y_3) = P_1(x_1, x_2, y_1, y_2)^2 + P_2(y_1, y_3)^2 + P_3(y_1, y_2)^2$. As $(x_1, x_2) \in S$ if and only if there exists a y_1, y_2 satisfying the required relationships, which is equivalent to there existing a y_1, y_2, y_3 satisfying $P(x_1, x_2, y_1, y_2, y_3) = 0$, we see S is Diophantine. □

However, Diophantine sets are not always so easy to understand or analyze. Instead, we analyze Diophantine *functions*. We also provide the definition of the graph of a function, in case the reader has forgotten

Definition 3.8. The *graph* of a function $f(\mathbf{x})$ is the set

$$\{(x_1, \dots, x_n, f(\mathbf{x})) : \mathbf{x} \in \text{dom} f\}$$

where $\text{dom} f$ indicates the domain of f.

Definition 3.9. A *Diophantine function* $f(x_1, x_2, \dots, x_n) = y$ is a function mapping natural numbers to natural numbers, such that the graph of the function is a Diophantine set. In other words, the set

$$S = \{(x_1, \dots, x_n, f(\mathbf{x})) : \mathbf{x} \in \mathbb{N}^n\}$$

is Diophantine.

Now we proceed with a couple examples of Diophantine functions to get some intuition

Example 3.10. The addition function, $f(x_1, x_2) = x_1 + x_2$ is Diophantine.

Proof. By considering the polynomial $P(x_1, x_2, y) = x_1 + x_2 - y$, we note that since $x_1 + x_2 - y = 0$ implies $x_1 + x_2 = y$, clearly the graph of f is Diophantine. The other direction also follows quickly. □

This analysis also lets us realize that given any polynomial Q, the function $g(x_1, \ldots, x_n) = Q(x_1, x_2, \ldots, x_n)$ is Diophantine. We can easily justify this by noting that the polynomial $P(x_1, \ldots, x_n, y) = Q(x_1, \ldots, x_n) - y$ is the associated polynomial of the graph of g.

Example 3.11. For some $a \in \mathbb{N}$, the step function

$$f(x) = \begin{cases} 0 & x \leq a \\ 1 & x > a \end{cases}$$

is Diophantine.

This shouldn't be too surprising, as the ordering relation $x \leq y$ can be captured by a Diophantine set.

Proof. We split up the graph of f into two sets, $S_1 = \{(x, 0) : x \leq a\}$ and $S_2 = \{(x, 1) : x > a\}$. We first note that we can rewrite S_1 as $\{(x_1, x_2) : x_1 \leq a \wedge x_2 = 0\}$. Then, using the idea from example 3.7, we note that we can break up the associated polynomial into two parts, one representing $x_1 \leq a$ and the other representing $x_2 = 0$ via the polynomial $P(x_1, x_2, y_1) = (a - x_1 - y_1)^2 + x_2^2$. We do essentially the same thing for S_2, and get that the associated polynomial is $P_2(x_1, x_2, y_2) = (x_1 - a - y_2 - 1)^2 + (x_2 - 1)^2$, so S_2 is Diophantine as well. Thus, as S is the union of Diophantine sets, S is also Diophantine. □

3.2 Fundamental Functions to the Analysis

The following Diophantine functions are central to the analysis, and so we'll introduce them as theorems.

Theorem 3.12. The "pair functions" are Diophantine. In other words, there exists Diophantine functions p, l, r such that for all $x, y \in \mathbb{N}$, we get the following properties

$$l(p(x, y)) = x \tag{3.13}$$
$$r(p(x, y)) = y \tag{3.14}$$
$$p(l(x), r(x)) = x \tag{3.15}$$
$$l(x) \leq x \tag{3.16}$$
$$r(x) \leq x \tag{3.17}$$

We choose the letters l, r, as the functions extract the "left" and "right" elements of the pair respectively.

Proof. One way to think about how to prove this is to enumerate all of the pairs of integers, first listing the pairs (a, b) that sum to 0, then to 1, etc. Then, we use $p(a, b)$ as an index into this list. Now we do it formally.

We use triangle numbers to show this. Let $T(n) = 1 + \cdots + n = \frac{n(n+1)}{2}$ denote the n'th triangle number. Since the triangle numbers are increasing, for any number z, we can find $T(n) < z \leq T(n+1)$. Since this n is unique given z, we can then define $z = T(n) + y$ for some positive integer $1 \leq y \leq T(n+1) - T(n) = n + 1$.

Intuitively, the triangular part $T(n)$ of z represents the number of pairs that had sum n, and the pair z represents has sum $n + 1$. Then, are $n + 1$ choices for y in the pair (x, y), therefore $1 \leq y \leq n + 1$.

Returning back to the analysis, since $n + 2 > y$, we can choose $x = n + 2 - y$ and substitute it into the definition of z to get $z = T(x + y - 2) + y$. Since $T(x + y - 2) + y$ is a polynomial, the function

$$p(x, y) = T(x + y - 2) + y$$

is Diophantine. As n, y were unique given z, x, y are also unique given z, so defining a function $l(p(x, y)) = x$ and $r(p(x, y)) = y$ is well defined. This definition of l, r satisfies equations (3.13) and (3.14) by definition.

We show that l, r are Diophantine as well. We use the associated polynomial to the graph of p. Letting $z = p(x, y)$

$$z - T(x + y - 2) - y = z - \frac{(x + y - 2)(x + y - 1)}{2} - y = 0 \tag{3.18}$$

multiplying out by 2:

$$2z - (x + y - 2)(x + y - 1) - y = 0 \tag{3.19}$$

Let $P(x, y, z) = 2z - (x + y - 2)(x + y - 1) - y$. Thus, the function l, r can be represented as

$$l(z) = x \iff (\exists y > 0) : P(x, y, z) = 0$$
$$r(z) = y \iff (\exists x > 0) : P(x, y, z) = 0$$

Hence, the associated polynomial for l would be $Q_1(z, l(z), y) = P(l(z), y, z)$ and for r would be $Q_2(z, r(z), x) = P(x, r(z), z)$

Now to show showing (3.15). Note that given arbitrary z, there exists x, y such that $z = p(x, y)$ as just shown, so $p(l(z), r(z)) = p(l(p(x, y)), r(p(x, y))) = p(x, y) = z$.

Equations (3.16) and (3.17) follow from the fact that $p(x, y) \geq y$ and $p(x, y) \geq x$ since $T(n) \geq n$ for all n. Then we realize $l(p(x, y)) = x \leq p(x, y)$ and $r(p(x, y)) = y \leq p(x, y)$. □

One other extremely useful Diophantine function is one produced by the sequence number theorem. As stated by Davis [3]

Theorem 3.20. There exists a function $S(i, u)$ such that

1. $S(i, u) \leq u$
2. For any sequence of numbers $a_1, \ldots, a_n$, there exists a corresponding u such that $S(i, u) = a_i$

Proof. Let l, r be the left and right functions defined in theorem 3.12.

$$S(i, u) = w$$

where $1 \leq w \leq 1 + ir(u)$ and $w \equiv l(u) \mod 1 + ir(u)$. Clearly such a w must exist.

First, we show S is Diophantine. Thinking about the requirements for w , we know that if $u = p(x, y), x = l(u), y = r(u)$, the only way for this system to be satisfied is using the polynomial (3.19) described in theorem 3.12. Let $P_1(x, y, u) = 2u - (x + y - 2)(x + y - 1) - y$ be that polynomial. Additionally, to guarantee that $w \equiv l(u) \equiv x \mod 1 + iy$, we note by the definition of modulus that this is equivalent to

$$(\exists b_1 > 0) : P_2(x, y, w, b_1) \coloneqq w - x - (b_1 - 1)(1 + iy) = 0$$

Finally, as $1 \leq w$ is already ensured as we require Diophantine sets and Diophantine functions to only deal with positive integers, we only need to restrict $w \leq 1 + iy$. We've already seen how to construct a polynomial representing this type of restriction in example 3.4:

$$(\exists b_2 > 0) : P_3(y, w, b_2) \coloneqq 1 + iy - w - (b_2 - 1) = 0$$

Ensuring that all of these polynomials are satisfied can be done by using the trick introduced in example 3.6 and used in example 3.7, allowing us to conclude that the associated polynomial to the graph of S is

$$P(x, y, w, b_1, b_2) = P_1(x, y, u)^2 + P_2(x, y, w, b_1)^2 + P_3(y, w, b_2)^2$$

This proves item 1.

To prove item 2, we note that there exists some m such that $m > \max\{a_1, \ldots, a_n\}$. By the Chinese Remainder Theorem, we know there exists some number v such that

$$\begin{aligned} v &\equiv a_1 \mod 1 + m \\ v &\equiv a_2 \mod 1 + 2m \\ &\cdots \\ v &\equiv a_n \mod 1 + nm \end{aligned}$$

We can then choose $u = p(v, m)$, and note that $S(i, u) = a_i$ as required. □

3.3 The Exponential Function

Another extremely useful Diophantine function is the exponential function x^y. The construction is very mechanical and does not provide too much intuition , but I don't think it's obvious how an exponential function could be constructed using Diophantine equations. Thus, we build up a couple lemmas around Pell equations to provide intuition about how an exponential function might be constructed.

Definition 3.21. The *Pell equation* is the following polynomial:

$$x^2 - dy^2 = 1 \tag{3.22}$$

where $d = a^2 - 1$ for some $a > 1$.

We provide a couple of the useful many lemmas that Davis [3] uses in his construction of the exponential function.

Lemma 3.23. For some d, if x, y are integers such that

$$1 < x + y\sqrt{d} < a + \sqrt{d} \tag{3.24}$$

then they do not satisfy equation (3.22).

Proof. Suppose for sake of contradiction that x, y satisfy equation (3.22). Then

$$(x + y\sqrt{d})(x - y\sqrt{d}) = x^2 - dy^2 = 1 = a^2 - d = (a + \sqrt{d})(a - \sqrt{d})$$

Since $x + y\sqrt{d} > 1$, then $x - y\sqrt{d} < 1$ as their product is equal to 1. Also, as $a + \sqrt{d} = (x + y\sqrt{d})\frac{x - y\sqrt{d}}{a - \sqrt{d}}$, we can use equation (3.24) to conclude that $\frac{x - y\sqrt{d}}{a - \sqrt{d}} > 1$ and get that $x - y\sqrt{d} > a - \sqrt{d}$. Taking the negative of both inequalities and combining them, we get that

$$-1 < -x + y\sqrt{d} < -a + \sqrt{d}$$

and then adding to equation (3.24), we get

$$0 < 2y\sqrt{d} < 2\sqrt{d}$$

and conclude that $0 < y < 1$. But as y is an integer, this is a contradiction. □

Lemma 3.25. For some d, let x_1, y_1 and x_2, y_2 satisfy equation (3.22). If we define x, y such that $x + y\sqrt{d} = (x_1 + y_1\sqrt{d})(x_2 + y_2\sqrt{d})$ then x, y satisfy equation (3.22).

Proof. We can note that conjugation is a ring homomorphism in $\mathbb{Z}[\sqrt{d}]$, and then immediately deduce that

$$x - y\sqrt{d} = (x_1 - y_1\sqrt{d})(x_2 - y_2\sqrt{d})$$

like in Davis [3]. It's not too hard to work out manually however, by noticing $x = x_1x_2 + dy_1y_2$ and $y = x_1y_2 + y_1x_2$, and then expanding the product

$$(x_1 - y_1\sqrt{d})(x_2 - y_2\sqrt{d}) = x_1x_2 + y_1y_2 - (x_1y_2 + x_2y_1)\sqrt{d} = x - y\sqrt{d}$$

We can then directly check equation (3.22)

$$\begin{aligned} x^2 - dy^2 = (x - y\sqrt{d})(x + y\sqrt{d}) &= (x_1 - y_1\sqrt{d})(x_1 + y_1\sqrt{d})(x_2 - y_2\sqrt{d})(x_2 + y_2\sqrt{d}) \\ &= (x_1^2 - dy_1^2)(x_2^2 - dy_2^2) = 1 \end{aligned}$$

□

Now, for the primary lemma about Pell equations.

Lemma 3.26. Let x_n, y_n be defined by $(a+\sqrt{d})^n$. Then all positive solutions to equation (3.22) have solutions x, y such that for some n, $x = x_n, y = y_n$.

Proof. It's clear that x_n, y_n satisfy equation (3.22) since $(a, 1)$ is a solution, using lemma 3.25, $x_2 + y_2\sqrt{d} = (a+\sqrt{d})^n$ is a solution, and we can use induction on n to conclude (x_n, y_n) are also solutions.

$x \geq 1$, therefore $x+y\sqrt{d} \geq 1$. Therefore, there exists a $n \geq 0$ such that $(a+\sqrt{d})^n \leq x+y\sqrt{d} < (a+\sqrt{d})^{n+1}$ since $a+\sqrt{d} > 1$ and the exponential function is increasing. Rewriting the inequality using x_n, y_n

$$x_n + y_n\sqrt{d} = (a+\sqrt{d})^n \leq x + y\sqrt{d} < (a+\sqrt{d})^{n+1} = (a+\sqrt{d})^n(a+\sqrt{d}) = (x_n + y_n\sqrt{d})(a+\sqrt{d})$$

We note if $x_n + y_n\sqrt{d} = x + y\sqrt{d}$ the result is trivial as x_n, y_n are a solution to the Pell equation, so we consider when $x_n+y_n\sqrt{<}x+y\sqrt{d}$. $(x_n+y_n\sqrt{d})(x_n-y_n\sqrt{d}) = 1$, and $x_n+y_n\sqrt{d} \geq 1$, therefore $x_n-y_n\sqrt{d} > 0$. Multiplying both sides

$$(x_n + y_n\sqrt{d})(x_n - y_n\sqrt{d}) = 1 < (x + y\sqrt{d})(x_n - y_n\sqrt{d}) < (x_n + y_n\sqrt{d})(x_n - y_n\sqrt{d})(a+\sqrt{d}) = a+\sqrt{d}$$

However, by lemma 3.23, this inequality is impossible, as no there does not exist any integers $1 < c_1+c_2\sqrt{d} < a+\sqrt{d}$. □

The main idea from lemma 3.26 is that solutions to the Pell equations grow as fast as an exponential function. As a result, it is possible with many Pell equations to construct a system of polynomials that is only solvable if x, y, x^y are used, which can lead to the development of the exponential function. Refer to sections 2 and 3 of Davis [3] for the full construction.

3.4 Connecting Diophantine Functions to Computability

Before continuing, we slightly formalize our notion of using predicate logic to construct Diophantine sets, and extend the language.

Remark 3.27. Assuming $\mathbf{x} \in \mathbb{Z}^n_{>0}$ and $\mathbf{y} \in \mathbb{Z}^m_{>0}$ as always, we have shown (at least implicitly) that

$$S_1 = \{\mathbf{x} : (\exists \mathbf{y}) P(\mathbf{x}, \mathbf{y}) = 0\}$$
$$S_2 = \{\mathbf{x} : (\exists \mathbf{y}) P_1(\mathbf{x}, \mathbf{y}) = 0 \wedge \cdots \wedge P_k(\mathbf{x}, \mathbf{y}) = 0\}$$
$$S_3 = \{\mathbf{x} : (\exists \mathbf{y}) P_1(\mathbf{x}, \mathbf{y}) = 0 \vee \cdots \vee P_k(\mathbf{x}, \mathbf{y}) = 0\}$$

are all valid ways to construct Diophantine sets. Additionally as example 3.7 showed, if $R_i(\mathbf{x})$ is a restriction or logical statement, for example $R_1(x_1, x_2) = x_1 < x_2$, and if $R_i(\mathbf{x})$ is true if and only if some polynomial $(\exists \mathbf{y}) Q_i(\mathbf{x}, \mathbf{y}) = 0$, then we can also define sets via

$$S_4 = \{\mathbf{x} : R_1(\mathbf{x}) \wedge \cdots \wedge R_k(\mathbf{x})\}$$

We now show that bounded quantifiers can also be used to construct Diophantine sets.

Definition 3.28. The *bounded quantifiers* are the quantifiers $(\forall y)_{\leq x}$ or $(\exists y)_{\leq x}$. For some restriction $R(x)$

$$(\forall y)_{\leq x} R(x) \equiv (\forall y)(y > x \vee R(x))$$
$$(\exists y)_{\leq x} R(x) \equiv (\exists y)(y \leq x \wedge R(x))$$

In other words, $(\forall y)_{\leq x}$ is read "for all y less than x", and $(\exists y)_{\leq x}$ is read "there exists a y less than x such that..."

Theorem 3.29. Bounded quantifiers can be used to construct Diophantine sets. If $P(z, u, \mathbf{x}, \mathbf{y})$ be a polynomial, then the sets

$$S_1 = \{(x_1, \ldots, x_n, z) : (\forall u)_{\leq z} (\exists \mathbf{y}) P(z, u, \mathbf{x}, \mathbf{y}) = 0\}$$
$$S_2 = \{(x_1, \ldots, x_n, z) : (\exists u)_{\leq z} (\exists \mathbf{y}) P(z, u, \mathbf{x}, \mathbf{y}) = 0\}$$

are Diophantine.

Proof. The bounded "there exists" quantifier is easy to prove. Using our definition, we can rewrite the definition for S_2 as

$$S_2 = \{(x_1, \ldots, x_n, z) : (\exists u, \mathbf{y}) u \leq z \wedge P(z, u, \mathbf{x}, \mathbf{y}) = 0\}$$

but we already have a way to construct a polynomial $Q(u, z)$ such that $Q(u, z) = 0 \iff u \leq z$ as mentioned in the proof for example 3.4. Therefore, we can replace that condition with $Q(u, z) = 0$. Then, we have a polynomial in the form discussed in remark 3.27, and so we can conclude S_2 is Diophantine.

The proof for the $(\forall u)_{\leq z}$ is significantly more complicated, so I will just summarize it. There exists an appropriate polynomial involving the factorial function that allows you to convert the quantifier to a string of quantifiers of the form $(\exists \mathbf{t}) P(t) = 0 \wedge \ldots$. As the right side consists of Diophantine predicates, the left must also describe a Diophantine set. The resulting predicate consists of over 10 logical clauses. □

The above theorem makes defining Diophantine sets significantly easier.

Example 3.30. The set of primes is Diophantine.

Proof. We know p is prime if and only if

$$(\forall d_1, d_2)_{\leq p} p > 1 \wedge (d_1 d_2 \neq p \vee d_1 = 1 \vee d_2 = 1)$$

Since we only used logical or's, and we can represent the not equal to relation with a polynomial as done in example 3.7, all the we can combine polynomials to find a polynomial P such that $P = 0$ if and only if this logical expression is satisfied. Thus, P is the associated polynomial for the set of primes, concluding the proof. □

Now we start showing useful to show the connection to computability.

Lemma 3.31. The function S derived from the sequence number theorem 3.20 is partial recursive.

Proof. We note that S is composed of the modulo and pair functions. The modulo function was shown to be primitive recursive in example 2.9. Additionally, the pair functions are just the coding functions for $\mathbb{N}^2$, and a trivial computable bijection exists between $\mathbb{N}^2$ and $\mathbb{Z}^2_{>0}$. Since S is composed of computable functions, S is computable. □

Lemma 3.32. If a function is Diophantine, then it is partial recursive.

Proof. If f is Diophantine, we know it's graph is Diophantine. Thus, there exists some associated polynomial P to the graph of f where

$$f(\mathbf{x}) = y \iff (\exists a_1, \ldots, a_m) P(\mathbf{x}, y, \mathbf{a})$$

Recall, since for any sequence $y, a_1, \ldots, a_m$, there exists a u such that $S(i + 1i, u) = a_i$ and $S(1, u) = y$ by theorem 3.20. As a result, we get the equivalence

$$(\exists a_1, \ldots, a_m) P(\mathbf{x}, y, \mathbf{a}) = 0 \iff (\exists u) P(\mathbf{x}, S(1, u), S(2, u), \ldots, S(m + 1, u)) = 0$$

As a result, since P is a partial recursive function, being composed of additions, subtractions, and multiplications which were shown to be in examples 2.4, 2.6, and 2.8. We can then define using μ-recursion $g(\mathbf{x}) = u$, where g is partial recursive, and then get our original function f via

$$f(\mathbf{x}) = S(1, g(\mathbf{x}))$$

□

Lemma 3.33. If a function is partial recursive, it is Diophantine.

Proof. We just show that all ways to construct partial recursive functions result in Diophantine functions. We consider partial recursive functions as functions from $\mathbb{Z}_{>0} \to \mathbb{Z}_{>0}$ in this section. We go through the methods in order

1. The successor function is Diophantine. We've shown that addition is Diophantine in example 3.10, we can adjust the instances of x_2 with 1 to get the successor function.

2. The constant function is Diophantine. They satisfy the polynomial

$$C(\mathbf{x}) = y \iff y - c = 0$$

where c is the constant.

3. The projection functions are Diophantine. We can show this easily with the associated polynomial

$$U_i^n(x_1, \ldots, x_n) = y \iff x_i - y = 0$$

4. The composition of Diophantine partial recursive functions is Diophantine. Let

$$f(\mathbf{x}) = h(g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$$

Then, we see that

$$y = f(\mathbf{x}) \iff (\exists t_1, \ldots, t_m)(t_1 = g_1(\mathbf{x}) \wedge \ldots t_m = g_m(\mathbf{x}) \wedge y = h(t_1, \ldots, t_m)) \tag{3.34}$$

We note that if a function f' is Diophantine, its graph is Diophantine, and therefore there exists a corresponding polynomial P' such that $f'(\mathbf{x}) = y \iff (\exists \mathbf{t})P'(\mathbf{x}, y, \mathbf{t}) = 0$. We let G_i be the corresponding polynomial for g_i and P be the corresponding polynomial for h. Then, we have that the right side of the equivalence (3.34) is true if and only if

$$(\exists t_1, \ldots, t_m)((\exists \mathbf{w}_1)G_1(\mathbf{x}, t_1, \mathbf{w}_1) = 0 \wedge \cdots \wedge (\exists \mathbf{w}_m)G_m(\mathbf{x}, t_m, \mathbf{w}_m) = 0 \wedge (\exists \mathbf{w}_{m+1})P(\mathbf{t}, y, \mathbf{w}_{m+1}) = 0)$$

As this expression consists only of Diophantine predicates, the polynomial $y = f(\mathbf{x})$ is Diophantine.

5. Primitive recursion of Diophantine partial recursive functions is Diophantine.

$$f(\mathbf{x}, 1) = g(\mathbf{x}) \tag{3.35}$$
$$f(\mathbf{x}, y + 1) = h(\mathbf{x}, y, f(\mathbf{x}, y)) \tag{3.36}$$

We develop a logical predicate that is satisfied if and only if $z = f(\mathbf{x}, y)$. The strategy is to first keep track of all of the values $f(\mathbf{x}, i)$ for $1 \leq i \leq y$. We do that using the sequence number theorem.

Given $\mathbf{x}, y, z$, we know that there exists u such that $S(i, u) = f(\mathbf{x}, i)$ for all $1 \leq i \leq y$ by theorem 3.20. $S(i, u) = f(\mathbf{x}, i)$ if and only if $S(i, u)$ satisfies the construction of f. In other words, to equation (3.35) is satisfied if and only if

$$f(\mathbf{x}, 1) = S(1, u) = g(\mathbf{x})$$

and equation (3.36) is satisfied if and only if

$$f(\mathbf{x}, y + 1) = S(y + 1, u) = h(\mathbf{x}, y, S(y, x))$$

Additionally, we note for any Diophantine equations f_1, f_2, we can always represent the expression

$$(\exists \mathbf{x})f_1(\mathbf{x}) = f_2(\mathbf{x})$$

or any related expression since the above is true if and only if $(\exists \mathbf{x}, v)v = f_1(\mathbf{x}) \wedge v = f_2(\mathbf{x})$, and we can represent predicates of the form $v = f_i$ since f_i is Diophantine.

Combining all of these expressions, we get the logical predicate Davis [3] presents

$$\begin{aligned} z = f(\mathbf{x}, y) \iff & (\exists u)[S(1, u) = g(\mathbf{x}) \wedge \\ & (\forall t)_{\leq y}(t = y \vee S(t + 1, u) = h(\mathbf{x}, y, S(t, u)) \wedge z = S(y, u)] \end{aligned}$$

which is representable by a Diophantine equation since it is composed of predicates (such as the bounded quantifier in theorem 3.29) which are representable by Diophantine equations.

6. μ-recursion of Diophantine functions produces Diophantine functions. We adjust our definition of μ-recursion, replacing 0s with 1s where appropriate. Let f, g be Diophantine functions. Define

$$f(\mathbf{x}) = y$$

where y is the smallest value such that $g(\mathbf{x}, y) = 1$. If y is the smallest values such that the equality holds, that means that $(\forall t)_{\leq y-1} g(\mathbf{x}, y) \neq 1$, assuming $y > 1$. Thus, it can be deduced that

$$y = f(\mathbf{x}) \iff (\forall t)_y (t = y \vee g(\mathbf{x}, t) \neq 1) \wedge g(\mathbf{x}, y) = 1$$

□

Now, we get the central theorems connecting Diophantine equations and computability by combining lemmas 3.32 and 3.33:

Theorem 3.37. A function is Diophantine if and only if it is partial recursive.

Theorem 3.38. The set of polynomials is effectively countable.

Proof. By theorem 2.29 we know that $S = \bigcup_{k\geq 1} \mathbb{N}^k$ is effectively countable. We can start from 1 by defining subtracting 1 from the coding function for $\bigcup_{k\geq 0} \mathbb{N}^k$, which keeps it computable. Let f be the coding function for $\bigcup_{k\geq 0} \mathbb{N}^k$. Let $H = \bigcup_{j\geq 0} S^j$. Note we can create a computable bijection between H and $\bigcup_{j\geq 0} \mathbb{N}^j$ by creating a function $h(s_1, \ldots, s_j) = f(f(s_1) - 1, \ldots, f(s_j) - 1)$. As f has a computable inverse, and $f - 1$ has a computable inverse when considered as a function from $\mathbb{Z}_{>0} \to \mathbb{N}$. Additionally, a computable function with computable inverse between H and the set of polynomials can be constructed by mapping to each $(s_1, \ldots, s_j) \in H$ to the polynomial represented by the sum of the terms represented by s_i. If $(n_{i,1}, \ldots, n_{i,k}) = s_i$, then the term would be $g^{-1}(n_{i,1}) x_{n_{i,2}} \ldots x_{n_{i,k}}$, where g is the coding function from $\mathbb{Z}$ to $\mathbb{N}$. This was mentioned to be computable, which shouldn't be surprising, but it's difficult to formalize. □

Note that since the set of polynomials is effectively countable, we can use it to index the set of Diophantine equations, by their associated polynomials. As we have a procedure to create a partial recursive function, and therefore Turing machine generating that recursive function by lemma 3.33, this indexing can be used in the proof of theorem 2.35.

Now it immediately follows that Diophantine equations cannot be solvable by any algorithm, as if it were possible, there would exist a Turing machine that could execute the algorithm, say M. But we would be able to determine if an arbitrary Diophantine function with polynomial with index e (which is possible by theorem 3.38), f halts ($f(x)\downarrow$) by checking if there existed a y such that $f(x) = y$ using M, checking if the polynomial associated to the graph of f had a solution with x, y and potentially other variables. In particular, as theorem 3.32 implied, we could construct a partial recursive function equal to f, by theorem 2.24, we can then construct the appropriate Turing machine representing f. Then, we can evaluate if $f(e)\downarrow$. Note that $f(e)\downarrow \iff \varphi_e(e)\downarrow$. However, this would imply the halting set is computable, which we know is a contradiction, by theorem 2.35. Thus, we conclude there does not exist a computable method to solve Diophantine equations.

References

[1] CWoo. *Ackermann function is not primitive recursive*. `http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm`. Accessed: 2020-06-02. 2013.

[2] CWoo. *Properties of Ackermann function*. `https://web.archive.org/web/20130509224602/http://planetmath.org/propertiesofackermannfunction`. Accessed: 2020-06-02. Mar. 2013.

[3] Martin Davis. "Hilbert's Tenth Problem is Unsolvable". In: *The American Mathematical Monthly* 80.3 (Mar. 1973), p. 233. ISSN: 00029890. DOI: `10.2307/2318447`. URL: `https://www.jstor.org/stable/2318447?origin=crossref` (visited on 05/25/2020).

[4] David Hilbert. "Mathematical problems". In: *Bulletin of the American Mathematical Society* 8.10 (July 1, 1902), pp. 437–480. ISSN: 0002-9904. DOI: 10.1090/S0002-9904-1902-00923-3. URL: http://www.ams.org/journal-getitem?pii=S0002-9904-1902-00923-3 (visited on 05/30/2020).

[5] Rebecca Weber. *Computability theory*. Providence, R.I: American Mathematical Society, 2012. ISBN: 978-0-8218-7392-2.