Learning Objectives:

- See more examples of uncomputable functions that are not as tied to computation.
- See Godel's incompleteness theorem - a result that shook the world of mathematics in the early 20th century.

12

Is every theorem provable?

> *"Take any definite unsolved problem, such as ... the existence of an infinite number of prime numbers of the form* $2^n + 1$*. However unapproachable these problems may seem to us and however helpless we stand before them, we have, nevertheless, the firm conviction that their solution must follow by a finite number of purely logical processes..."*
> *"...This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus."*, David Hilbert, 1900.

12.1 *Unsolvability of Diophantine equations*

The problems of the previous lecture, while natural and important, still intimately involved NAND++ programs or other computing mechanisms in their definitions. One could perhaps hope that as long as we steer clear of functions whose inputs are themselves programs, we can avoid the "curse of uncomputability". Alas, we have no such luck.

Many of the functions people wanted to compute over the years involved solving equations. These have a much longer history than mechanical computers. The Babilonians already knew how to solve some quadratic equations in 2000BC, and the formula for all quadratics appears in the Bakshali Manuscript that was composed in India around the 3rd century. During the Renaissance, Italian mathematicians discovered generalization of these formulas for cubic and

quartic (degrees 3 and 4) equations. Many of the greatest minds of the 17th and 18th century, including Euler, Lagrange, Leibnitz and Gauss worked on the problem of finding such a formula for *quintic* equations to no avail, until in the 19th century Ruffini, Abel and Galois showed that no such formula exists, along the way giving birth to *group theory*.

However, the fact that there is no closed-form formula does not mean we can not solve such equations. People have been solving higher degree equations numerically for ages. The Chinese manuscript Jiuzhang Suanshu from the first century mentions such approaches. Solving polynomial equations is by no means restricted only to ancient history or to students' homeworks. The gradient descent method is the workhorse powering many of the machine learning tools that have revolutionized Computer Science over the last several years.

But there are some equations that we simply do not know how to solve *by any means*. For example, it took more than 200 years until people succeeded in proving that the equation $a^{11} + b^{11} = c^{11}$ has no solution in integers.[1] The notorious difficulty of so called *Diophantine equations* (i.e., finding *integer* roots of a polynomial) motivated the mathematician David Hilbert in 1900 to include the question of finding a general procedure for solving such equations in his famous list of twenty-three open problems for mathematics of the 20th century. I don't think Hilbert doubted that such a procedure exists. After all, the whole history of mathematics up to this point involved the discovery of ever more powerful methods, and even impossibility results such as the inability to trisect an angle with a straightedge and compass, or the non-existence of an algebraic formula for qunitic equations, merely pointed out to the need to use more general methods.

[1] This is a special case of what's known as "Fermat's Last Theorem" which states that $a^n + b^n = c^n$ has no solution in integers for $n > 2$. This was conjectured in 1637 by Pierre de Fermat but only proven by Andrew Wiles in 1991. The case $n = 11$ (along with all other so called "regular prime exponents") was established by Kummer in 1850.

Alas, this turned out not to be the case for Diophantine equations: in 1970, Yuri Matiyasevich, building on a decades long line of work by Martin Davis, Hilary Putnam and Julia Robinson, showed that there is simply *no method* to solve such equations in general:

Theorem 12.1 — MRDP Theorem. Let $SOLVE : \{0,1\}^* \to \{0,1\}^*$ be the function that takes as input a string describing a 100-variable polynomial with integer coefficients $P(x_0, \ldots, x_{99})$ and outputs either $(z_0, \ldots, z_{99}) \in \mathbb{N}^{100}$ s.t. $P(z_0, \ldots, z_{99}) = 0$ or the string `no solution` if no P does not have non-negative integer roots. [2] Then $SOLVE$ is uncomputable. Moreover, this holds even for the easier function $HASSOL : \{0,1\}^* \to \{0,1\}$ that given such a polynomial

P outputs 1 if there are $z_0, \ldots, z_{99} \in \mathbb{N}$ s.t. $P(z_0, \ldots, z_{99}) = 0$ and outputs 0 otherwise.

The difficulty in finding a way to distinguish between "code" such as NAND++ programs, and "static content" such as polynomials is just another manifestation of the phenomenon that *code* is the same as *data*. While a fool-proof solution for distinguishing between the two is inherently impossible, finding heuristics that do a reasonable job keeps many firewall and anti-virus manufacturers very busy (and finding ways to bypass these tools keeps many hackers busy as well).

[2] As usual, we assume some standard way to express numbers and text as binary strings. The constant 100 is of course arbitrary; in fact the number of variables can be reduced to nine, at the expense of the polynomial having a constant (but very large) degree. It is also possible to restrict attention to polynomials of degree four and at most 58 variables. See Jones's paper for more about this issue.

12.1.1 "Baby" MRDP Theorem: hardness of quantified Diophantine equations

Computing the function $HASSOL$ is equivalent to determining the truth of a logical statement of the following form:[3]

$$\exists_{z_0,\ldots,z_{99}\in\mathbb{N}} \text{ s.t.} P(z_0, \ldots, z_{99}) = 0 \,. \tag{12.1}$$

[3] Recall that $\exists$ denotes the *existential quantifier*; that is, a statement of the form $\exists_x \varphi(x)$ is true if there is *some* assignment for x that makes the Boolean function $\varphi(x)$ true. The dual quantifier is the *universal quantifier*, denoted by $\forall$, where a statement $\forall_x \varphi(x)$ is true if *every* assignement for x makes the Boolean function $\varphi(x)$ true. Logical statements where all variables are *bound* to some quantifier (and hence have no parameters) can be either true or false, but determining which is the case is sometimes highly nontrivial. If you could use a review of quantifiers, section 3.6 of the text by Lehman, Leighton and Meyer is an excellent source for this material.

Theorem 12.1 states that there is no NAND++ program that can determine the truth of every statements of the form Eq. (12.1). The proof is highly involved and we will not see it here. Rather we will prove the following weaker result that there is no NAND++ program that can determine the truth of more general statements that mix together the existential ($\exists$) and universal ($\forall$) quantifiers. The reason this result is weaker than Theorem 12.1 is because deciding the truth of more general statements (that involve both quantifier) is a potentially harder problem than only existential statements, and so it is potentially easier to prove that this problem is uncomputable. (If you find the last sentence confusing, it is worthwhile to reread it until you are sure you follow its logic; we are so used to trying to find solution for problems that it can be quite confusing to follow the arguments showing that problems are *uncomputable*.)

Definition 12.2 — Quantified integer statements. A *quantified integer statement* is a well-formed statement with no unbound variables involving integers, variables, the operators $>, <, \times, +, -, =$, the logical operations $\neg$ (NOT), $\wedge$ (AND), and $\vee$ (OR), as well as quantifiers of the form $\exists_{x\in\mathbb{N}}$ and $\forall_{y\in\mathbb{N}}$ where x, y are variable names.

Definition 12.2 is interesting in its own right and not just as a "toy version" of Theorem 12.1. We often care deeply about determining

the truth of quantified integer statements. For example, the statement that Fermat's Last Theorem is true for $n = 3$ can be phrased as the quantified integer statement

$$\neg \exists_{a \in \mathbb{N}} \exists_{b \in \mathbb{N}} \exists_{c \in \mathbb{N}} (a > 0) \wedge (b > 0) \wedge (c > 0) \wedge (a \times a \times a + b \times b \times b = c \times c \times c) \ . \tag{12.2}$$

The twin prime conjecture, that states that there is an infinite number of numbers p such that both p and $p + 2$ are primes can be phrased as the quantified integer statement

$$\forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \wedge PRIME(p) \wedge PRIME(p + 2) \tag{12.3}$$

where we replace an instance of $PRIME(q)$ with the statement $(q > 1) \wedge \forall_{a \in \mathbb{N}} \forall_{b \in \mathbb{N}} (a = 1) \vee (a = q) \vee \neg (a \times b = q)$.

The claim (mentioned in Hilbert's quote above) that are infinitely many primes of the form $p = 2^n + 1$ can be phrased as follows:

$$\begin{aligned} \forall_{n \in \mathbb{N}} \exists_{p \in \mathbb{N}} (p > n) \wedge PRIME(p) \wedge \\ (\forall_{k \in \mathbb{N}} (k \neq 2 \ \wedge \ PRIME(k)) \Rightarrow \neg DIVIDES(k, p - 1)) \end{aligned} \tag{12.4}$$

where $DIVIDES(a, b)$ is the statement $\exists_{c \in \mathbb{N}} b \times c = a$. In English, this corresponds to the claim that for every n there is some $p > n$ such that all of $p - 1$'s prime factors are equal to 2.

Syntactic sugar: To make our statements more readable, we often use syntactic sugar and so write $x \neq y$ as shorthand for $\neg(x = y)$, and so on. In Eq. (12.4) we also used the "implication operator" $a \Rightarrow b$ as "syntactic sugar" or shorthand for $\neg a \vee b$. Similarly, we will sometimes use the "if and only if operator" $a \Leftrightarrow$ as shorthand for $(a \Rightarrow b) \wedge (b \Rightarrow b)$.

Much of number theory is concerned with determining the truth of quantified integer statements. Since our experience has been that, given enough time (which could sometimes be several centuries) humanity has managed to do so for the statements that it cared enough about, one could (as Hilbert did) hope that eventually we will discover a *general procedure* to determine the truth of such statements. The following theorem shows that this is not the case:

Theorem 12.3 — Uncomputability of quantified integer statements. Let $QIS : \{0,1\}^* \to \{0,1\}$ be the function that given a (string represen-

tation of) a quantified integer statement outputs 1 if it is true and 0 if it is false. [4] Then *QIS* is uncomputable.

[4] Since a quantified integer statement is simply a sequence of symbols, we can easily represent it as a string. We will assume that *every* string represents some quantified integer statement, by mapping strings that do not correspond to such a statement to an arbitrary statement such as $\exists_{x\in\mathbb{N}} x = 1$.

Note that Theorem 12.3 is an immediate corollary of Theorem 12.1. Indeed, if you can compute *QIS* then you can compute *HASSOL* and hence if you *can't* compute *HASSOL* then you can't compute *QIS* either. But Theorem 12.3 is easier (though not trivial) to prove, and we will provide the proof in the following section.

12.2 *Proving the unsolvability of quantified integer statements.*

In this section we will prove Theorem 12.3. The proof will, as usual, go by reduction from the Halting problem, but we will do so in two steps:

1. We will first use a reduction from the Halting problem to show that a deciding *quantified mixed statements* is uncomputable. Unquantified mixed statements involve both strings and integers.
2. We will then reduce the problem of quantified mixed statements to quantifier integer statements.

12.2.1 *Quantified mixed statements and computation traces*

As mentioned above, before proving Theorem 12.3, we will give an easier result showing the uncomputability of deciding the truth of an even more general class of statements- one that involves not just integer-valued variables but also string-valued ones.

Definition 12.4 — Quantified mixed statements. A *quantified mixed statement* is a well-formed statement with no unbound variables involving integers, variables, the operators $>, <, \times, +, -, =$, the logical operations $\neg$ (NOT), $\wedge$ (AND), and $\vee$ (OR), as well as quantifiers of the form $\exists_{x\in\mathbb{N}}$, $\exists_{a\in\{0,1\}^*}$, $\forall_{y\in\mathbb{N}}$, $\forall_{b\in\{0,1\}^*}$ where x, y, a, b are variable names. These also include the operator $|a|$ which returns the length of a string valued variable a, as well as the operator a_i where a is a string-valued variable and i is an integer valued expression which is true if i is smaller than the length of a and the i^{th} coordinate of a is 1, and is false otherwise.

For example, the true statement that for every string a there is a string b that corresponds to a in reverse order can be phrased as the

following quantified mixed statement

$$\forall_{a\in\{0,1\}^*}\exists_{b\in\{0,1\}^*}(|a| = |b|) \wedge (\forall_{i\in\mathbb{N}} i < |a| \Rightarrow (a_i \Leftrightarrow b_{|a|-i}) \; . \tag{12.5}$$

Quantified mixed statements are more general than quantified integer statements, and so the following theorem is potentially easier to prove than Theorem 12.3:

Theorem 12.5 — Uncomputability of quantified mixed statements. Let $QMS : \{0,1\}^* \rightarrow \{0,1\}$ be the function that given a (string representation of) a quantified mixed statement outputs 1 if it is true and 0 if it is false. Then QMS is uncomputable.

12.2.2 *"Unraveling" NAND++ programs and quantified mixed integer statements*

We will first prove Theorem 12.5 and then use it to prove Theorem 12.3. The proof is again by reduction to $HALT$ (see Fig. 12.1). That is, we do so by giving a program that transforms any NAND++ program P and input x into a quantified mixed statement $\varphi_{P,x}$ such that $\varphi_{P,x}$ is true if and only if P halts on input x. This will complete the proof, since it will imply that if QMS is computable then so is the $HALT$ problem, which we have already shown is uncomputable.

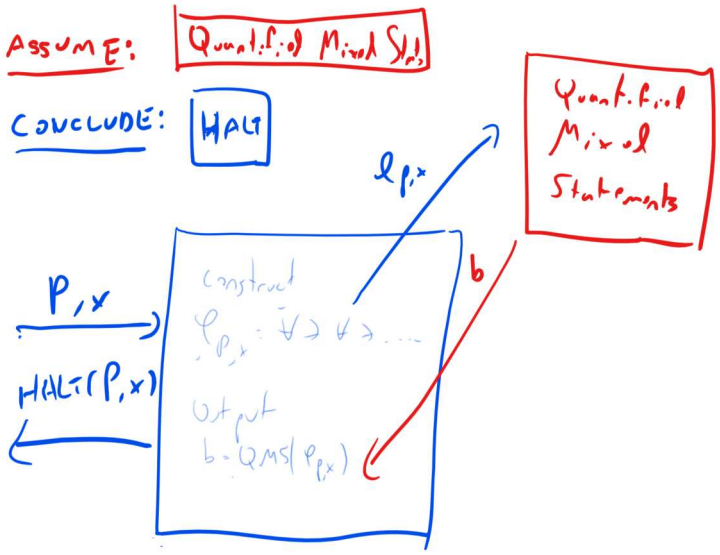

Figure 12.1: We prove that QMS is uncomputable by giving a reduction that maps every pair (P, x) into a quantified mixed statements $\varphi_{P,x}$ that is true if and only if P halts on x.

The idea behind the construction of the statement $\varphi_{P,x}$ is the following. The statement will be true if and only if there exists a string $\Delta \in \{0,1\}^*$ which corresponds to a summary of an *execution trace* that proves that P halts on input x. At a high level, the crucial insight is that unlike when we actually run the computation, to verify the correctness of a execution trace we only need to verify *local consistency* between pairs of lines.

Informally, an execution trace of a program P on an input x is a string that represents a "log" of all the lines executed and variables assigned in the course of the execution. For example, if we execute on nandpl.org the parity program

```
tmp_1 := seen_i NAND seen_i
tmp_2 := x_i NAND tmp_1
val := tmp_2 NAND tmp_2
ns := s NAND s
y_0 := ns NAND ns
u := val NAND s
v := s NAND u
w := val NAND u
s := v NAND w
seen_i := zero NAND zero
stop := validx_i NAND validx_i
loop := stop NAND stop
```

on the input `01`, the trace will be the following text (truncated here, since it is not the most riveting of reading material):

```
Executing commmand "tmp_1 := seen_i NAND seen_i", seen_0 has
     value 0, seen_0 has value 0, tmp_1 assigned value 1
Executing commmand "tmp_2 := x_i NAND tmp_1", x_0 has value
    0, tmp_1 has value 1, tmp_2 assigned value 1
Executing commmand "val_0 := tmp_2 NAND tmp_2", tmp_2 has
    value 1, tmp_2 has value 1, val_0 assigned value 0
Executing commmand "ns_0 := s_0 NAND s_0", s_0 has value 0,
    s_0 has value 0, ns_0 assigned value 1
Executing commmand "y_0 := ns_0 NAND ns_0", ns_0 has value
    1, ns_0 has value 1, y_0 assigned value 0
Executing commmand "u_0 := val_0 NAND s_0", val_0 has value
    0, s_0 has value 0, u_0 assigned value 1
Executing commmand "v_0 := s_0 NAND u_0", s_0 has value 0,
    u_0 has value 1, v_0 assigned value 1
Executing commmand "w_0 := val_0 NAND u_0", val_0 has value
    0, u_0 has value 1, w_0 assigned value 1
```

```
Executing commmand "s_0 := v_0 NAND w_0", v_0 has value 1,
    w_0 has value 1, s_0 assigned value 0
Executing commmand "seen_i := zero_0 NAND zero_0", zero_0
    has value 0, zero_0 has value 0, seen_0 assigned value 1
Executing commmand "stop_0 := validx_i NAND validx_i",
    validx_0 has value 1, validx_0 has value 1, stop_0
    assigned value 0
Executing commmand "loop := stop_0 NAND stop_0", stop_0 has
    value 0, stop_0 has value 0, loop assigned value 1
Entering new iteration
Executing commmand "tmp_1 := seen_i NAND seen_i", seen_1 has
     value 0, seen_1 has value 0, tmp_1 assigned value 1
...
...
...
Executing commmand "seen_i := zero_0 NAND zero_0", zero_0
    has value 0, zero_0 has value 0, seen_2 assigned value 1
Executing commmand "stop_0 := validx_i NAND validx_i",
    validx_2 has value 0, validx_2 has value 0, stop_0
    assigned value 1
Executing commmand "loop := stop_0 NAND stop_0", stop_0 has
    value 1, stop_0 has value 1, loop assigned value 0
Result: 1 (1)
```

The line by line execution trace is quite long and tedious, but note that it is very easy to locally check, without the need to redo the computation ourselves: we just need to see that each line computes the NAND correctly, and that the value that it claims for the variables on the righthand side of the assignment is the same value that was written to them in the previous line that accessed them.

More formally, we will use the notion of a *modification log* or "Deltas" of a NAND++ program, as presented in Definition 7.6.[5] Recall that given a NAND++ program P and an input $x \in \{0,1\}^n$, if P has s lines and takes T iterations of its loop to halt on x, then the *modification log* of P on x is the string $\Delta \in \{0,1\}^{sT+n}$ such that for every $\ell \in [n]$, $\Delta_\ell = x_\ell$ and for every $\ell \in \{n, n+1, \ldots, sT+n-1\}$, Δ_ℓ corresponds to the value that is assigned to a variable during step number $(\ell - n)$ of the execution. Note that for every $\ell \in \{n, n+1, \ldots, sT+n+-1\}$, Δ_ℓ is the NAND of Δ_j and Δ_k where j and k are the last lines in which the two variables referred to in the corresponding line are assigned a value.

[5] We could also have proven Godel's theorem using the sequence of all configurations, but the "deltas" have a simpler format.

The idea of the reduction is that given a NAND++ program P and an input x, we can come up with a mixed quantifier statement

$\Psi_{P,x}(\Delta)$ such that for every $\Delta \in \{0,1\}^*$, $\Psi_{P,x}(\Delta)$ is true if and only if Δ is a consistent modification log of P on input x that ends in a halting state (with the `loop` variable set to 0). The full details are rather tedious, but the high level point is that we can express the fact that Δ is consistent as the following conditions:

- For every $i \in [n]$, $\Delta_i = x_i$. (Note that this is easily a mixed quantifier statement.)
- For every $\ell \in \{n, \ldots, Ts + n - 1\}$, $\Delta_\ell = 1 - \Delta_j \Delta_k$ where j and k are the locations in the log corresponding to the last steps before step $\ell - n$ in which the variables on the righthand side of the assignment were written to. (This is a mixed quantifier statement as long as we can compute j and k using some arithmetic function of i, or some integer quantifier statement using i as a parameter.)
- If t_0 is the last step in which the variable `loop` is written to, then $\Delta_{t_0+n} = 0$. (Again this is a mixed quantifier statement if we can compute t_0 from the length of Δ.)

To ensure that we can implement the above as mixed quantifier statements we observe the following:

1. Since the $INDEX$ function that maps the current step to the location of the value `i` was an explicit arithmetic function, we can come up with a quantified integer statement $INDEX(t,i)$ that will be true if and only if the value of `i` when the program executes step t equals i. (See Exercise 12.2.)
2. We can come up with quantified integer statements $PREV_1(t,t')$ and $PREV_2(t,t'')$ that will satisfy the following. If at step $t-n$ the operation invokved is `foo := bar NAND baz` then $PREV_1(t,t')$ is true if and only if t' is n plus the last step before t in which `bar` was written to and $PREV_2(t,t'')$ is true if and only if t'' is n plus the last step before t in which `baz` was written to. (If one ore both of the righthand side variables are input variables, the t' and/or t' will correspond to the index in $[n]$ of that variable.) Therefore, checking the validity of Δ_t will amount to checing that $\forall_{t' \in \mathbb{N}} \forall_{t'' \in \mathbb{N}} \neg PREV_1(t,t') \vee \neg PERV_2(t,t'') \vee (\Delta_t = 1 - \Delta_{t'} \Delta_{t''})$. Note that these statements will themselves use $INDEX$ because if `bar` and/or `baz` are indexed by `i` then part of the condition for $PREV_1(t,t')$ and $PREV_2(t,t'')$ will be to ensure that the `i` variable in these steps is the same as the variable in step $t-n$. Using this, plus the observation that the program has only a constant number of lines, we can create the statements $PREV_1$ and $PREV_2$. (See Exercise 12.3)

3. We can come up with a quantified integer statement $LOOP(t)$ that will be true if and only if the variable written to at step t in the execution is equal to `loop`.

Together these three steps enable the construction of the formula $\Psi_{P,x}(\Delta)$. (We omit the full details, which are messy but ultimately straightforward.) Given a construction of such a formula $\Psi_{P,x}(\Delta)$ we can see that $HALT(P,x) = 1$ if and only if the following quantified mixed statement $\varphi_{P,x}$ is true

$$\varphi_{P,x} = \exists_{\Delta \in \{0,1\}^*} \Psi_{P,x}(\Delta) \tag{12.6}$$

and hence we can write $HALT(P,x) = QMS(\varphi_{P,x})$. Since we can compute from P,x the statement $\varphi_{P,x}$, we see that if QMS is computable then so would have been $HALT$, yielding a proof by contradiction of Theorem 12.5.

12.2.3 *Reducing mixed statements to integer statements*

We now show how to prove Theorem 12.3 using Theorem 12.5. The idea is again a proof by reduction. We will show a transformation of every quantifier mixed statement φ into a quantified *integer* statement ξ that does not use string-valued variables such that φ is true if and only if ξ is true.

To remove string-valued variables from a statement, we encode them by integers. We will show that we can encode a string $x \in \{0,1\}^*$ by a pair of numbers $(X,n) \in \mathbb{N}$ s.t.

- $n = |x|$
- There is a quantified integer statement $COORD(X,i)$ that for every $i < n$, will be true if $x_i = 1$ and will be false otherwise.

This will mean that we can replace a quantifier such as $\forall_{x \in \{0,1\}^*}$ with $\forall_{X \in \mathbb{N}} \forall_{n \in \mathbb{N}}$ (and similarly replace existential quantifiers over strings). We can later replace all calls to $|x|$ by n and all calls to x_i by $COORD(X,i)$. Hence an encoding of the form above yields a proof of Theorem 12.3, since we can use it to map every mixed quantified statement φ to quantified integer statement ξ such that $QMS(\varphi) = QIS(\xi)$. Hence if QIS was computable then QMS would be computable as well, leading to a contradiction.

To achieve our encoding we use the following technical result :

Lemma 12.6 — Constructible prime sequence. There is a sequence of prime numbers $p_0 < p_1 < p_3 < \cdots$ such that there is a quantified integer statement $PCOORD(p,i)$ that is true if and only if $p = p_i$.

Using Lemma 12.6 we can encode a $x \in \{0,1\}^*$ by the numbers (X,n) where $X = \prod_{x_i=1} p_i$ and $n = |x|$. We can then define the statement $COORD(X,i)$ as

$$\forall_{p \in \mathbb{N}} PCOORD(p,i) \Rightarrow DIVIDES(p,X) \tag{12.7}$$

where $DIVIDES(a,b)$, as before, is defined as $\exists_{c \in \mathbb{N}} a \times c = b$. Note that indeed if X,n encodes the string $x \in \{0,1\}^*$, then for every $i < n$, $COORD(X,i) = x_i$, since p_i divides X if and only if $x_i = 1$.

Thus all that is left to conclude the proof of Theorem 12.3 is to prove Lemma 12.6, which we now proceed to do.

Proof. The sequence of prime numbers we consider is the following: We fix C to be a suficiently large constant ($C = 2^{2^{34}}$ will do and define p_i to be the smallest prime number that is in the interval $[(i+C)^3 + 1, (i+C+1)^3 - 1]$. It is known that there exists such a prime number for every $i \in \mathbb{N}$. Given this, the definition of $PCOORD(p,i)$ is simple:

$$(p > (i+C) \times (i+C) \times (i+C)) \wedge (p < (i+C+1) \times (i+C+1) \times (i+C+1)) \wedge \left(\forall_{p'} \neg PRIME(p') \vee (p' \leq i) \vee (p' \geq p)\right) , \tag{12.8}$$

We leave it to the reader to verify that $PCOORD(p,i)$ is true iff $p = p_i$. ■

12.3 *Hilbert's Program and Gödel's Incompleteness Theorem*

> *"And what are these ... vanishing increments? They are neither finite quantities, nor quantities infinitely small, nor yet nothing. May we not call them the ghosts of departed quantities?"*, George Berkeley, Bishop of Cloyne, 1734.

The 1700's and 1800's were a time of great discoveries in mathematics but also of several crises. The discovery of calculus by Newton and Leibnitz in the late 1600's ushered a golden age of problem solving. Many longstanding challenges succumbed to the new tools that were discovered, and mathematicians got ever better at doing some truly impressive calculations. However, the rigorous foundations behind these calculations left much to be desired. Mathematicians manipulated infinitsemal quantities and infinite series cavalierly, and while most of the time they ended up with the correct results, there were a few strange examples (such as trying to calculate the value of the infinite series $1 - 1 + 1 - 1 + 1 + \ldots$) which seemed to give

out different answers depending on the method of calculation. This led to a growing sense of unease in the foundations of the subject which was addressed in works of mathematicians such as Cauchy, Weierstrass, and Reimann, who eventually placed analysis on firmer foundations, giving rise to the ϵ's and δ's that students taking honors calculus grapple with to this day.

In the beginning of the 20th century, there was an effort to replicate this effort, in greater rigor, to all parts of mathematics. The hope was to show that all the true results of mathematics can be obtained by starting with a number of axioms, and deriving theorems from them using logical rules of inference. This effort was known as the *Hilbert program*, named after the very same David Hilbert we mentioned above. Alas, Theorem 12.1 yields a devastating blow to this program, as it implies that for *any* valid set of axioms and inference laws, there will be unsatisfiable Diophantine equations that cannot be proven unsatisfiable using these axioms and laws.

To study the existence of proofs, we need to make the notion of a *proof system* more precise. What axioms shall we use? What rules? Our idea will be to use an extremely general notion of proof. A *proof* will be simply a piece of text- a finite string- such that:

1. *(effectiveness)* Given a statement x and a proof w (both of which can be encoded as strings) we can verify that w is a valid proof for x. (For example, by going line by line and checking that each line does indeed follow from the preceding ones using one of the allowed inference rules.)

2. *(soundness)* If there is a valid proof w for x then x is true.

Those seem like rather minimal requirements that one would want from every proof system. Requirement 2 (soundndess) is the very definition of a proof system: you shouldn't be able to prove things that are not true. Requirement 1 is also essential. If it there is no set of rules (i.e., an algorithm) to check that a proof is valid then in what sense is it a proof system? We could replace it with the system where the "proof" for a statement x would simply be "trust me: it's true".

Let us give a formal definition for this notion, specializing for the case of Diophantine equations:

Definition 12.7 — Proof systems for diophantine equations. A proof system for Diophantine equations is defined by a NAND++ program V. A *valid proof* in the system corresponding to V of the unsatisfiability of a diophantine equation "$P(\cdot, \cdots, \cdot) = 0$" is some string

$w \in \{0,1\}^*$ such that $V(P,w) = 1$. The proof system corresponding to V is *sound* if there is no valid proof of a false statement. That is, for every diophantine equation "$P(\cdot,\cdots,\cdot) = 0$", if there exists $w \in \{0,1\}^*$ such that $V(P,w) = 1$ then for every $x_1,\ldots,x_t \in \mathbb{Z}$, $P(x_1,\ldots,x_t) \neq 0$.

The formal definition is a bit of a mouthful, but what it states the natural notion of a logical proof for the unsatisfiability of an equation. By the Church-Turing Thesis, we can replace NAND++ with any other reasonable computational model for proof verification. Hilbert believed that for all of mathematics, and in particular for settling diophantine equations, it should be possible to find some set of axioms and rules of inference that would allow to derive all true statements. However, he was wrong:

Theorem 12.8 — Gödel's Incompleteness Theorem. For every sound proof system V, there exists a diophantine equation "$P(\cdot,\cdots,\cdot) = 0$" such that there is no $x_1,\ldots,x_t \in \mathbb{N}$ that satisfy it, but yet there is no proof in the system V for the statement that the equation is unsatisfiable.

Proof. Suppose otherwise, that there exists such a system. Then we can define the following algorithm S that computes the function $HASSOL : \{0,1\}^* \to \{0,1\}$ described in Theorem 12.1. The algorithm will work as follows:

- On input a Diophantine equation $P(x_1,\ldots,x_t) = 0$, for $k = 1,2,\ldots$ do the following:

1. Check for all $x_1,\ldots,x_t \in \{0,\ldots,k\}$ whether $x_1,\ldots,x_t$ satisfies the equation. If so then halt and output 1.
2. For all $n \in \{1,\ldots,k\}$ and all strings w of length at most k, check whether $V(P,w) = 1$. If so then halt and output 0.

Under the assumption that for *every* diophantine equation that is unsatisfiable, there is a proof that certifies it, this algorithm will always halt and output 0 or 1, and moreover, the answer will be correct. Hence we reach a contradiction to Theorem 12.1 ■

Note that if we considered proof systems for more general quantified integer statements, then the existence of a true but yet unprovable statement would follow from Theorem 12.3. Indeed, that was the content of Gödel's original incompleteness theorem which was proven in 1931 way before the MRDP Theorem (and initiated the line of research which resulted in the latter theorem). Another way

to state the result is that every proof system that is rich enough to express quantified integer statements is either inconsistent (can prove both a statement and its negation) or incomplete (cannot prove all true statements).

Examining the proof of Theorem 12.8 shows that it yields a more general statement (see Exercise 12.4): for every uncomputable function $F : \{0,1\}^* \to \{0,1\}$ and every sound proof system, there is some input x for which the proof system is not able to prove neither that $F(x) = 0$ nor that $F(x) \neq 0$ (see Exercise 12.4).

Also, the proof of Theorem 12.8 can be extended to yield Gödel's second incompleteness theorem which, informally speaking, says for that every proof system S rich enough to express quantified integer statements, the following holds:

- There is a quantified integer statement φ that is true if and only if S is consistent (i.e., if there is no statement x such that S can prove both x and $NOT(x)$).
- There is no proof in S for φ.

Thus once we pass a sufficient level of expressiveness, we cannot find a proof system that is strong enough to prove its own consistency. This in particular showed that Hilbert's second problem (which was about finding an axiomatic provably-consistent basis for arithmetic) was also unsolvable.

12.3.1 *The Gödel statement*

One can extract from the proof of Theorem 12.8 a procedure that for every proof system V (when thought of as a verification algorithm), yields a true statement x^* that cannot be proven in V. But Gödel's proof gave a very explicit description of such a statement x which is closely related to the "Liar's paradox". That is, Gödel's statement x^* was designed to be true if and only if $\forall_{w \in \{0,1\}^*} V(x,w) = 0$. In other words, it satisfied the following property

$$x^* \text{ is true} \Leftrightarrow x^* \text{ does not have a proof in } V \tag{12.9}$$

One can see that if x^* is true, then it does not have a proof, but it is false then (assuming the proof system is sound) then it cannot have a proof, and hence x^* must be both true and unprovable. One might wonder how is it possible to come up with an x^* that satisfies a condition such as Eq. (12.9) where the same string x^* appears on

both the righthand side and the lefthand side of the equation. The idea is that the proof of Theorem 12.8 yields a way to transform every statement x into a statement $F(x)$ that is true if and only if x does not have a proof in V. Thus x^* needs to be a *fixed point* of F: a sentence such that $x^* = F(x^*)$. It turns out that we can always find such a fixed point of F. We've already seen this phenomenon in the λ calculus, where the Y combinator maps every F into a fixed point YF of F. This is very related to the idea of programs that can print their own code. Indeed, Scott Aaronson likes to describe Gödel's statement as follows:

> The following sentence repeated twice, the second time in quotes, is not provable in the formal system V. "The following sentence repeated twice, the second time in quotes, is not provable in the formal system V."

In the argument above we actually showed that x^* is *true*, under the assumption that V is sound. Since x^* is true and does not have a proof in V, this means that we cannot carry the above argument in the system V, which means that V cannot prove its own soundness (or even consistency: that there is no proof of both a statement and its negation). Using this idea, it's not hard to get Gödel's second incompleteness theorem, which says that every sufficiently rich V cannot prove its own consistency. That is, if we formalize the statement c^* that is true if and only if V is consistent (i.e., V cannot prove both a statement and the statement's negation), then c^* cannot be proven in V.

12.4 *Lecture summary*

- Uncomputable functions include also functions that seem to have nothing to do with NAND++ programs or other computational models such as determining the satisfiability of diophantine equations.

- This also implies that for any soudn proof system (and in particular every finite axiomatic system) S, there are interesting statements X (namely of the form "$F(x) = 0$" for an uncomputable function F) such that S is not able to prove either X or its negation.

12.5 Exercises

Exercise 12.1 — Expression for floor. Let $FSQRT(n,m) = \forall_{j \in \mathbb{N}}((j \times j) > n) \vee (j < m) \vee (j = m)$. Prove that $FSQRT(m,n)$ is true if and only if $m = \lfloor \sqrt{m} \rfloor$. ■

Exercise 12.2 — Expression for computing the index. Recall that in Exercise 7.1 asked you to prove that at iteration t of a NAND++ program the the variable `i` is equal to $t - r(r+1)$ if $t \leq (r+1)^2$ and equals $(r+2)(r+1)t$ otherwise, where $r = \lfloor \sqrt{t + 1/4} - 1/2 \rfloor$. Prove that there is a quantified integer statement $INDEX$ with parameters t, i such that $INDEX(t,i)$ is true if and i is the value of `i` after t iterations. ■

Exercise 12.3 — Expression for computing the previos line. Give the following quantified integer expressions:

1. $MOD(a,b,c)$ which is true if and only if $c = a \mod c$. Note if a program has s lines then the line executed at step t is equal to $t \mod s$.

2. Suppose that P is the three line NAND program listed below. Give a quantified integer statement $LAST(n,t,t')$ such that $LAST(t,t')$ is true if and only if $t' - n$ is the largest step smaller than $t - n$ in which the variable on the righthand side of the line executed at step $t - n$ is written to. If this variable is an input variable `x_i` then let $LAST(n,t,t')$ to be true if the current index location equals t' and $t' < n$. ■

```
y_0 := foo_i NAND foo_i
foo_i := x_i NAND x_i
loop := validx_i NAND validx_i
```

Exercise 12.4 — axiomatic proof systems. For every representation of logical statements as strings, we can define an axiomatic proof system to consist of a finite set of strings A and a finite set of rules $I_0, \ldots, I_{m-1}$ with $I_j : (\{0,1\}^*)^{k_j} \to \{0,1\}^*$ such that a proof $(s_1, \ldots, s_n)$ that s_n is true is valid if for every i, either $s_i \in A$ or is some $j \in [m]$ and are $i_1, \ldots, i_{k_j} < i$ such that $s_i = I_j(s_{i_1}, \ldots, i_{k_j})$. A system is *sound* if whenever there is no false s such that there is a proof that s is true Prove that for every uncomputable function $F : \{0,1\}^* \to \{0,1\}$ and every sound axiomatic proof system S (that is characterized by a finite number of axioms and inference rules), there is some input x for which the proof system S is not able to prove neither that $F(x) = 0$ nor that $F(x) \neq 0$. ■

[6] TODO: Maybe add an exercise to give a MIS that corresponds to any regular expression.

12.6 Bibliographical notes

12.7 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

12.8 Acknowledgements

Thanks to Alex Lombardi for pointing out an embarrassing mistake in the description of Fermat's Last Theorem. (I said that it was open for exponent 11 before Wiles' work.)