Computability theory

Andrew Marks

February 25, 2024

These notes are a work in progress and full of errors: corrections are much appreciated. Thanks to Tyler Arant, Spencer Unger, and all of the Math 220B students in 2021, 2022, and 2023 for many corrections and helpful conversations about the material in these notes.

The main prerequisite for the notes is knowledge of basic model theory: structures, isomorphisms and embeddings of them, compactness, and completeness. Starred sections are optional. They discuss interesting mathematics related to the core concepts of the course, and often require knowledge of undergraduate algebra, analysis, and set theory.

Contents

1 Introduction: Hilbert's program

1.1 A brief history of rigor in mathematics

Throughout much of history, mathematics was a practical tool seen as largely inseparable from its applications. Even up through the 18th century, there was no sharp distinction drawn between mathematics and physics, and many famous mathematicians such as Isaac Newton were equally regarded as physicists. Little attention was paid to the formalization of mathematics, and axiomatic foundations were not widely sought. Euclid's axiomatic presentation of geometry was perhaps the lone exception. Euclidean geometry was viewed by many as the epitome of logical precision and rigor, though foundational questions remained such as the role of the parallel postulate.

This view began to change in the 19th and 20th centuries. Mathematics was increasingly seen as a subject in its own right, whose interest did not depend only on its utility. Moreover, numerous controversies moved the philosophy and foundations of mathematics into the spotlight. For example, Lobachevsky's discovery of models for non-Euclidean geometry in 1870s led to disagreements over their acceptability.

An early example of this was the formalization of analysis. Newton and Leibniz's calculus was inarguably revolutionary and of immense practical use. However, putting calculus on precise mathematical footing took hundreds of years and involved the efforts of many mathematicians. Cauchy, Riemann, and Weierstrass were among those who laid the modern foundations of analysis.[1]

Another foundational controversy was Cantor's set theory. Cantor's work in the late 19th century let to fierce debate over its validity. A notable opponent of Cantor's work was Kronecker, who famously opined that "Die ganzen Zahlen hat der liebe Gott gemacht, alles andere ist Menschenwerk" ["God made natural numbers; all else is the work of man"] [We]. Russell discovered his famous paradox in 1901, which made urgent the problem of putting set theory on firm foundations. By the 1920's Zermelo and Fraenkel had proposed their axioms for set theory based on the iterative conception of the set, which seemed to be a satisfactory resolution of Russell's paradox.

1.2 Hilbert's program

David Hilbert had long been involved with the formalization of mathematics. His work in this area began with his axiomatization of geometry, culminating in his influential 1899 tome *Grundlagen der Geometrie*. Since foundational questions had long been of particular importance to Hilbert, it was not surprising that his famous list of 23 problems for the 20th century included the problem of proving that the axioms of arithmetic are consistent.

Over the following couple decades, Hilbert's views on the foundations of mathematics became particularly influential. In the 1920s, Hilbert proposed what is now known as "Hilbert's program": finding a finite axiomatization of all of mathematics, and then proving that these axioms are consistent and complete. This would finally immunize mathematics from inconsistencies like Russell's paradox, and from constant arguments over foundations. Furthermore, Hilbert proposed that this consistency proof should be carried out in arithmetic, whose validity could not be doubted.

Hilbert's program was shown to be impossible in the 1930's by Gödel, Turing, and others. Their theorems sent shockwaves through the mathematical community and changed our views of mathematical truth. Their theorems are also the traditional starting point for a course on computability theory.

Key to proving Gödel and Turing's theorems was making a precise definition of a computable function. For centuries, mathematicians had an informal understanding of what an algorithm was: a finitely describable deterministic procedure which can be executed in a finite amount of time. This idea goes back at least to the ancient Greeks, with examples such as the Euclidean algorithm.

[1] In the 1960s, Abraham Robinson at UCLA finally made Leibniz's original notion of infinitesimals rigorous with his invention of nonstandard analysis.

However, there was no formal definition of all possible algorithms; mathematicians had a subjective view: "I know an algorithm when I see it".

Church, Gödel, Herbrand, Turing, and others in the 1930s formally defined computability. Once computable functions had been precisely defined, it now became possible to prove that there are functions that are *not* computable. For example, one of Gödel's theorems is that the function that maps each sentence in the language of arithmetic to whether it is true or false in the structure $(\mathbb{N}; 0, 1, +, \cdot, <)$ is not computable. Hence, the algorithm that searches for a proof of a sentence or its negation from the axioms of PA and returns true or false once it finds such a proof cannot compute this function. Since any sentence that is provable is true, this algorithm must not be total: there must therefore be a sentence that is independent from PA. This is essentially Gödel's first incompleteness theorem. A modern proof of Gödel's theorem is to show that the halting problem can be reduced to the problem of computing truth for sentences in the language of arithmetic. Hence, truth for sentences in arithmetic is incomputable as a corollary of Turing's theorem that the halting problem is incomputable.

It is worth pausing for a moment before we begin down this road. It may be too easy to forget now the plausibility and appeal of Hilbert's program. By the 1920's essentially all mathematics had been axiomatized, and so Hilbert's program was "half complete". Hilbert had also put forth a plausible outline as to how consistency could be proven.

Before we start this journey of incompleteness and undecidability, we will sketch a proof of a theorem that one could view as perhaps the last great triumph of Hilbert's program: the completeness and decidability of "elementary geometry".

1.3 The Tarski-Seidenberg theorem*

Theorem 1.1 (Tarski-Seidenberg)**.** *The structure* $(\mathbb{R}; 0, 1, +, \cdot, <)$ *has quantifier elimination, and a decidable theory: there is an algorithm which computes what sentences are true and false in this structure.*

Essentially all of elementary geometry can be formalized as first order statements in this structure, and it includes a great deal of deep and interesting mathematics. For example, an interesting family of problems that can be stated in this language is the kissing spheres problem. For each n, fixing a central unit sphere in $\mathbb{R}^n$, how many other unit spheres can one arrange that each touch the central sphere, but do not intersect each other? This problem in dimension three was the source of a famous disagreement between Isaac Newton and David Gregory, and remained unsolved for a few hundred years (the answer is 12 as Newton conjectured and not 13 as Gregory suggested). Several sketched solutions were given in the nineteenth century. However, it wasn't until 1953 that the first detailed correct proof was given by Schütte and van der Waerden. The four dimensional generalization of the kissing spheres problem was settled by Musin in 2003: it there can be at most 24 kissing spheres. The five dimensional version remains open, though the answer is known to be between 40 and 44. See the paper [PZ] for a survey of progress on this problem.

Recall the definition of elimination of quantifiers.

Definition 1.2. Suppose S is a structure with language $\mathcal{L}$. Then S has elimination of quantifiers if for every quantifier free formula $\varphi(x, \mathbf{y})$ in $\mathcal{L}$ having free variables x and $\mathbf{y}$, there is a formula φ' such that $S \vDash \exists x \varphi(x, \mathbf{y}) \leftrightarrow \varphi'(\mathbf{y})$. We say that S has computable elimination of quantifiers if the signature of S is computable, and there is an algorithm that takes as input a quantifier free formula $\varphi(x, \mathbf{y})$ and outputs a formula $\varphi'(\mathbf{y})$ such that $S \vDash \exists x \varphi(x, \mathbf{y}) \leftrightarrow \varphi'(\mathbf{y})$.

Note that we can similarly define quantifier elimination for theories. A theory T with language $\mathcal{L}$ has quantifier elimination if for every quantifier free formula $\varphi(x, \mathbf{y})$ in $\mathcal{L}$ having free variables x and $\mathbf{y}$, there is a formula φ' such that $T \vDash \exists x \varphi(x, \mathbf{y}) \leftrightarrow \varphi'(\mathbf{y})$.

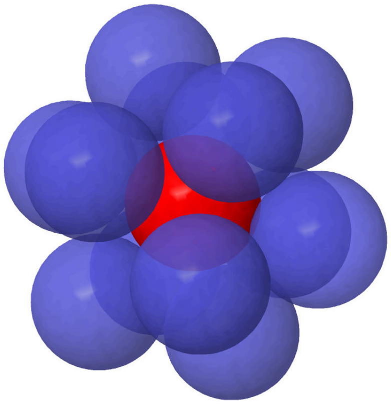


Figure 1: Twelve unit spheres kissing a central (red) one. Based on Sage code of Robert Bradshaw: `http://en.wikipedia.org/wiki/File:Kissing-3d.png`.

Exercise 1.3. Show that if a structure S has computable quantifier elimination, and the set of quantifier-free sentences that are true in S is computable, then the theory of S is computable. [Hint: observe that a formula can be computably put in prenex normal form, and then iteratively eliminate each of the quantifiers.]

You already know many examples of elimination of quantifiers in the structure $(\mathbb{R}; 0, 1, +, \cdot, <)$. For example, since a quadratic has a root if and only if its determinant is nonnegative,

$$\exists x(a^2x + bx + c = 0)$$

is equivalent to the quantifier-free formula

$$b^2 - 4ac \geq 0.$$

To prove that the structure $(\mathbb{R}; 0, 1, +, \cdot, <)$ has computable elimination of quantifiers, we will build on an earlier algorithm due to Sturm which can be used to decide whether a polynomial with rational coefficients has a root. One of the main tools used in Strum's algorithm is polynomial division, and we use the notation $\operatorname{rem}(p_0(x), p_1(x))$ to indicate the remainder when $p_0(x)$ is divided by $p_1(x)$, so that $p_0(x) = p_1(x)q(x) + \operatorname{rem}(p_0(x), p_1(x))$, for some $q(x)$, where $\operatorname{rem}(p_0(x), p_1(x))$ has degree strictly less than the degree of $p_1(x)$.

Theorem 1.4 (Sturm). *Given a real polynomial $p(x)$ and its derivative $p'(x)$, consider the sequence of polynomials given by repeatedly doing polynomial division, and taking remainders,*

$$\begin{aligned}
p_0(x) &= p(x) \\
p_1(x) &= p'(x) \\
p_2(x) &= -\operatorname{rem}(p_0(x), p_1(x)) \\
p_3(x) &= -\operatorname{rem}(p_1(x), p_2(x)) \\
&\vdots \\
p_n(x) &= -\operatorname{rem}(p_{n-2}(x), p_{n-1}(x))
\end{aligned}$$

so $p_n(x)$ is nonzero, but $p_n(x)$ divides into $p_{n-1}(x)$ with a remainder of 0. Let $s(x)$ be the number of times the sign of the number changes in the sequence $p_0(x), p_1(x), \ldots p_n(x)$. If $x < x'$, the number of roots of $p(x)$ between x and x' is $s(x') - s(x)$.

Before we prove this theorem, we give an example.

Example 1.5. If $p(x) = x^3 - 3x^2 + x - 1$, then the sequence of polynomials from Sturm's theorem is[2]:

$$\begin{aligned} p_0(x) &= x^3 - 3x^2 + x - 1 \\ p_1(x) &= 3x^2 - 6x + 1 \\ p_2(x) &= 4/3x + 2/3 \\ p_3(x) &= -19/4 \end{aligned}$$

Now taking the limit as $x \to -\infty$, we see $p_0(x)$ is negative, $p_1(x)$ is positive, $p_2(x)$ is negative, and $p_3(x)$ is negative. So $s(-\infty) = 2$ since the sign changes twice in this sequence. As $x \to \infty$, we see that $p_0(x)$ is positive, $p_1(x)$ is positive, $p_2(x)$ is positive, and $p_3(x)$ is negative, so $s(\infty) = 1$ since the sign in this sequence changes once. So Sturm's theorem says that there is $2 - 1 = 1$ root between $-\infty$ and ∞.

We're ready to prove Sturm's theorem.

Proof of Theorem 1.4. First, we prove the theorem in the case when $p_n(x)$ is a constant (which is not zero). This implies that $p(x) = p_0(x)$ and $p'(x) = p_1(x)$ do not have any common polynomial factor; a common factor of $p_0(x)$ and $p_1(x)$ must also be a common factor of $p_2(x)$, since $p_0(x) = p_1(x)q(x) - p_2(x)$ for some $q(x)$ and inductively, a common factor of $p(x)$ and $p'(x)$ must be a common factor of $p_i(x)$ for all i between 0 and n. (We are essentially doing the Euclidean algorithm for finding the greatest common divisor of $p(x)$ and $p'(x)$ in the definition of our sequence of the $p_i(x)$). Note that this means that the multiplicity of every root of $p(x)$ is 1; if a root of $p(x)$ has multiplicity greater than 1, its multiplicity in $p'(x)$ is one less.

We will show that as x increases, whenever $p_0(x)$ has a root, the number of sign changes in the sign sequence from the p_i drops, and whenever any other $p_i(x)$ has a root, the number of sign changes in the sequence stays the same. This is enough to prove the theorem.

First, if $p_0(x) = p(x)$ has a root at x, then $p_1(x) = p'(x)$ must be the opposite from $p_0(x - \epsilon)$ for some small ϵ. This is because if $p(x - \epsilon) > 0$, then p must be decreasing to have a root, so $p'(x) < 0$. Similarly, if $p(x - \epsilon) < 0$, then p must be increasing and so $p'(x) > 0$. Thus, the signs of $p_0(x)$ and $p_1(x)$ must either change from $+-$ to $--$, or from $-+$ to $++$.

Now suppose $p_{i+1}(x)$ has a root. By definition, $p_i(x) = p_{i+1}(x)q(x) - p_{i+2}(x)$ for some quotient polynomial $q(x)$. Since $p_i(x)$ and $p_{i+1}(x)$ have no common factor, they do not share any roots so if $p_{i+1}(x) = 0$, then $p_i(x) \neq 0$. Further, $p_i(x)$ and $p_{i+2}(x)$ have opposite signs when $p_i(x) = 0$. Hence, whenever $p_{i+1}(x)$ has a root and changes sign, then the total number of sign changes in our sequence says the same; the three signs of $p_i(x), p_{i+1}(x), p_{i+2}(x)$ either flip from $++-$ to $+--$ or vice versa, or $-++$ to $--+$ or vice versa.

So we have shown that whenever $p_0(x)$ has a root, the number of sign changes in the sign sequence of the p_is decreases by 1, and whenever $p_{i+1}(x)$ has a root, the total number of sign changes stays the same.

To do the general case now, if $p(x)$ and $p'(x)$ have a common factor $f(x)$, then the theorem follows by dividing the sequence $p_0(x), p_1(x), \ldots, p_n(x)$ by $f(x)$, and then applying the above argument. □

The proof of the Tarski-Seidenberg finishes by generalizing Sturm's algorithm so that it works symbolically (when the coefficients of the polynomial are variables) and so that it can determine if a collection of polynomials satisfies some combination of inequalities (instead of just whether a single polynomial equals zero). Some boolean combinations of inequalities can be combined using tricks of algebra. For example, $p(\mathbf{x}) = 0 \wedge q(\mathbf{x}) = 0 \leftrightarrow p(\mathbf{x})^2 + q(\mathbf{x})^2 = 0$, and $p(\mathbf{x}) = 0 \vee q(\mathbf{x}) = 0 \leftrightarrow p(\mathbf{x})q(\mathbf{x}) = 0$. However, other combinations need a generalization of Sturm's theorem. Here is an exercise which handles the case of eliminating the following quantifier: $\exists x(p(x) = 0 \wedge q(x) > 0)$.

[2] since for example, $x^3 - 3x^2 + x - 1 = (x/3 - 1/3)(3x^2 - 6x + 1) + (-4/3x - 2/3)$

Exercise 1.6. [Sturm's algorithm for inequalities] Suppose $p(x)$ and $q(x)$ are polynomials with integer coefficients, and consider the following sequence of polynomials:

$$\begin{aligned}
p_0(x) &= p(x)\\
p_1(x) &= p'(x)q(x)\\
p_2(x) &= -\operatorname{rem}(p_0(x), p_1(x))\\
p_3(x) &= -\operatorname{rem}(p_1(x), p_2(x))\\
&\vdots\\
p_n(x) &= -\operatorname{rem}(p_{n-2}(x), p_{n-1}(x))
\end{aligned}$$

so $p_n(x)$ is nonzero, but $p_n(x)$ divides into $p_{n-1}(x)$ with a remainder of 0. Suppose a and b are not roots of $p(x)$. Show that the number of sign changes in this sequence of polynomials $p_0(x), \dots, p_n(x)$ between $x = a$ and $x = b$ is equal to the number of roots $p(x)$ in (a, b) such that $q(x) > 0$ minus the number of roots of $p(x)$ in (a, b) such that $q(x) < 0$.

For a full proof of the Tarski-Seidenberg theorem, see Chapter 1 of Coste's book "An Introduction to Semialgebraic Geometry" [C00].

An interesting avenue of investigation is how much the Tarski-Seidenberg theorem can be generalized. Does the theorem remain true when we add more functions to our language so that we can discuss more complicated phenomena? For example, Tarski asked in 1940 whether one can prove the same theorem when exponentiation is added to our language:

Open Problem 1.7 ([Tar67]). *Is there an algorithm for computing what sentences are true in the structure* $(\mathbb{R}; 0, 1, +, \cdot, \exp, <)$*?*

Not only is this question an open problem, but we don't even know if there is an algorithm for deciding the truth of sentences such as $e^{-e^2} - 60e^{-15} = e^{-3e^1+2e^{-1}-e^{-9}}$ involving no variables or quantifiers! Are there any surprising identities involving exponentiation and the integers beyond obvious ones that follow from the fact that $e^x e^y = e^{x+y}$? This is a difficult open problem in transcendental number theory. However, there is a widely believed conjecture due to Schanuel which implies that indeed, the only such true identities are the obvious ones, and that there is an algorithm for deciding quantifier-free sentences. In fact, if Schanuel's conjecture is true, then Macintyre and Wilkie have shown Problem 1.7 has a positive answer [MW96]. See [Mar96] for a survey of this result.

What about if we change what number system we use to something other than the real numbers? This is also an interesting avenue of investigation, and may results are known. For example, if we work over the complex numbers instead, then Tarski showed in 1948 that the analogous theorem is true: there is algorithm to decide the truth of sentences in the complex field.

1.4 The efficiency of Tarski-Seidenberg*

How good is the Tarski-Seidenberg algorithm from a practical perspective? When we have a computer execute it, can it quickly solve interesting problems, such as the kissing spheres problem? The answer is that the algorithm takes far too much time to run except for sentences that have very few quantifiers. Each time a quantifier is eliminated, our sentences becomes exponentially larger, and so the formulas involved become massive.

Significant progress has been made on finding faster algorithms, using techniques such as cylindrical algebraic decomposition. There is an algorithm which decides sentences with n symbols in $O(2^{2^n})$ time, and an algorithm for deciding existential formulas (ones beginning with a single block of existential quantifiers, and containing no other quantifiers) in $O(2^n)$ time. This first result is known to essentially be optimal.

Alas, even these improved algorithms are still far too slow when run on practical problems. For example, modern implementations of quantifier elimination are able to solve the kissing spheres problem in 2 dimensions (with a little ingenuity to make the problem slightly easier, such as fixing the position of the first two kissing spheres). However, the kissing spheres problem in higher dimensions is completely out of reach even on modern supercomputers. Still, these algorithms are an important part of almost all computer algebra systems and receive a great deal of use for people working on practical mathematics. There are lots of interesting formulas which are rather short.

It is worth noting that in general, algorithms for quantifier elimination in any interesting structure (one with a nontrivial definable set) will always be inefficient. This is because any such quantifier elimination procedure can also solve the **quantified Boolean formula problem**: determining whether a sentence is true in the boolean algebra $(\{\top, \bot\}, \wedge, \vee, \neg)$. Simply replace $\top$ and $\bot$ in a given structure with being an element of this definable set and its complement (e.g. being equal to 0 or being not equal to 0 in the structure $(\mathbb{R}; 0, 1, +, \cdot, <)$). The quantified Boolean formula problem is known to be a PSPACE complete problem in computational complexity theory [AB, Theorem 4.13], and it is thus conjectured to require exponential time to solve.

1.5 Hilbert's 17th problem*

There is a beautiful application of the Tarski-Seidenberg theorem to Hilbert's 17th problem. Hilbert 17th problem asks whether every rational function f on $\mathbb{R}^n$ which is everywhere non-negative can be written as a sum of squares of finitely many rational functions $g_0, \ldots, g_k$, so $f = \sum_{i \leq k} g_i^2$.

As recounted in [Sch12] this problem had its roots in the thesis defense of Minkowski in 1885 where Hilbert was one of the official opponents. Minkowski in his defense conjectured that there were everywhere non-negative polynomials which cannot be expressed as a sum of squares of polynomials. Hilbert didn't believe this at the time, but Minkowski eventually convinced him that it was true. Hilbert proved the conjecture a few years later in 1888 [Hil88]

Here is an easy example of such a polynomial which is due to Motzkin from 1967 [Mot67]. The function

$$f(x,y) = x^4y^2 + x^2y^4 + 1 - 3x^2y^2 = \frac{x^2y^2(x^2+y^2+1)(x^2+y^2-2)^2 + (x^2-y^2)^2}{(x^2+y^2)^2}$$

is always positive (since it is a sum of squares of rational functions). However, if this function is a sum of squares of real polynomials $f = \sum_i g_i^2$, since $f(x,0) = f(0,y) = 1$, all the polynomials $g_i(x,0)$ and $g_i(0,y)$ must be constants. Hence, each g_i must be of the form $g_i = a_i + b_ixy + c_ix^2y + d_ixy^2$. Now the coefficient of x^2y^2 in the sum $\sum_i g_i^2$ is equal to $-3 = \sum_i b_i^2$. This is a contradiction.

Hilbert eventually came to believe Hilbert's 17th problem for rational functions had a positive solution. Indeed, he proved the special case that an everywhere nonnegative polynomial in two variables can be expressed as a sum of squares of rational functions [Hil93]. Eventually, Hilbert's 17th problem was completely solved by Artin in 1927. The connections between this problem in logic were first realized and pursued by Abraham Robinson in the mid 1950s.

Recall that an **ordered field** is a field F with an additional total ordering $\leq$ such that for all $a, b, c \in F$.

1. $b \leq c$ implies $a + b \leq a + c$.

2. If $0 \leq a$ and $0 \leq b$, then $0 \leq a \cdot b$. An ordered

field is called a **real closed field** if $a \geq 0$ implies there exists b such that $b^2 = a$, and every polynomial of odd degree with coefficients in F has at least one root in F.

Now we are ready to state an important consequence of the Tarski-Seidenberg theorem:

Theorem 1.8 (Tarski's transfer principle)**.** *Suppose F is an ordered field which contains $\mathbb{R}$, we have a quantifier-free formula $\varphi(x_1, \ldots, x_n)$ in the language $\{0, 1, +, \cdot, \leq\}$ and there exists $a_1, \ldots, a_n \in F$*

such making $\varphi(a_1, \dots, a_n)$ true in F. Then there exists $b_1, \dots, b_n \in \mathbb{R}$ such that $\varphi(b_1, \dots, b_n)$ is true in $\mathbb{R}$.

Proof sketch. A careful examination of the proof of the Tarski-Seidenberg theorem shows that same proof for the structure $(\mathbb{R}; 0, 1, +, \cdot, <)$ works for the theory of real closed fields. Hence all real closed fields have the same theory. One then proves the theorem by taking the real closure $\overline{F}$ of F. Since $F \vDash \exists x_1, \dots, x_n \varphi(x_1, \dots, x_n)$ we have that $\overline{F} \vDash \exists x_1, \dots, x_n \varphi(x_1, \dots, x_n)$ but then since $\overline{F}$ and $\mathbb{R}$ have the same theory, we must have that $\mathbb{R} \vDash \exists x_1, \dots, x_n \varphi(x_1, \dots, x_n)$ □

We'll use this theorem to solve Hilbert's 17th problem.

Let $F = \mathbb{R}(X_1, \dots, X_k)$ be the field of rational functions over $\mathbb{R}$ in the variables $X_1, \dots, X_k$. Let $I = \{\sum g_i^2 : g_i \in \mathbb{F} \text{ are nonzero}\}$. Then I has the property that

1. If $g \in F$ is nonzero, then $g^2 \in I$,
2. If $g, h \in I$, then $g + h$ and $gh \in I$.
3. $0 \notin I$.

In any field, a set with these properties is called an **order ideal**.

Lemma 1.9. *If I is an order ideal in a field F and $f \neq 0$ and $f \notin I$, then there exists an order ideal $J \supseteq I$ such that $-f \in J$.*

Proof. Let J be the set of nonzero polynomials in $(-f)$ with $g_i \in I$ as coefficients. Clearly J satisfies properties 1 and 2. Now if property 3 fails, then there is some polynomial p such that $p(-f) = 0$. Separate even and odd exponents to get $p(-f) = q(f^2) - fr(f^2)$ for some q, r. Note that if either q or r is a nonzero polynomial, then the corresponding $q(f^2)$ or $r(f^2)$ is nonzero, since it must then be in I. Hence, if $r(f^2) = 0$, then $q(f^2)$ must also be zero, since $p(-f) = 0$, but then this means p is the zero polynomial. Thus, $r(f^2)$ is nonzero and hence we may divide to get

$$f = \frac{q(f^2)}{r(f^2)} = q(f^2) \cdot r(f^2) \cdot \frac{1}{r(f^2)^2} \in I$$

which is a contradiction. □

Now we are ready to solve Hilbert's 17th problem:

Theorem 1.10 (Artin)**.** *Suppose $f(x_1, \dots, x_n)$ is a rational function which is positive everywhere it is defined. Then there exist rational functions $g_0, \dots, g_k$ such that $f = \sum_{i \leq k} g_i^2$.*

Proof. Work in the field $F = \mathbb{R}(X_1, \dots, X_k)$ of rational functions in the variables $X_1, \dots X_k$. So examples of elements of F are things like 3, $X_1 + X_2$, $1/(X_3^2) + 7X_5$, etc. Let $I \subseteq F$ be the order ideal of elements of F of the form $\sum_{i \leq k} g_i^2$ where every $g_i \neq 0$.

Now suppose $f \in F$ is such that $f \notin I$. By the lemma above, we can find $J \supseteq I$ containing $-f$. Iteratively use the lemma to extend J to a maximal order ideal $K \subseteq F$ using Zorn's lemma. Then K satisfies properties 1-3 and also satisfies that $\forall g \neq 0$ in F, either $g \in K$ or $-g \in K$. For $g, h \in K$, let $g < h$ iff $h = g \in K$. Then we have that F with this order is an ordered field which contains $\mathbb{R}$.

Now we use the transfer principle. Since $f < 0$ and f is a rational function in $\mathbb{R}(X_1, \dots, X_k)$ the following sentence is true over $\mathbb{F}$:

$$\exists x_1, \dots, x_n f(x_1, \dots, x_k) < 0$$

To see this is true just plug in $x_i = X_k$, and then $f(X_1, \dots, X_k) = f$.

Note that $f(x_1, \dots, x_k) < 0$ can be written as a quantifier-free formula in the language $\{0, 1, +, \cdot, \leq \}$. Hence, by Tarski's transfer principle, the formula

$$\exists x_1, \dots, x_n f(x_1, \dots, x_k) < 0$$

is also true over $\mathbb{R}$. Hence, we have proved that if $f(X_1, \dots, X_k)$ is not a sum of squares of rational functions, there are real numbers $b_1, \dots, b_k$ such that $f(b_1, \dots, b_k) < 0$. $\square$

2 Defining computability

The goal of this section is to define precisely what it means for a function $f \colon \mathbb{N} \to \mathbb{N}$ to be computable by an algorithm. The precise definitions in this section aren't used heavily in the rest of these notes; they only are used when we need the precise definition of computable functions for undecidability proofs. Instead, the definitions in this section are more of historical and philosophical interest. In subsequent sections, whenever we specify a computable function, we will typically just informally describe the algorithm that computes it.

In Section 2.4 we will discuss a few basic properties of all computable functions at that we *will* use a great deal: the existence of a universal machine, the S-m-n theorem, and the padding lemma. These are the some of the basic ingredients we will use to construct incomputable functions.

2.1 Partial recursive and primitive recursive functions

Historically, several mathematicians in the early 1930s (Church, Gödel, Herbrand, and others) gave what we now recognize to be the correct definition of computability. However, they were unable to give a convincing argument that their definition included all possible algorithms. For example, Herbrand-Gödel defined the class of **partial recursive functions** to be the smallest class of partial functions from $\mathbb{N}^k \to \mathbb{N}$ that

1. Contain the constant functions $x \mapsto n$ for each fixed $n \in \mathbb{N}$.
2. Contain the successor function $x \mapsto x + 1$.
3. Contain the projection functions $(x_1, \dots, x_k) \mapsto x_i$
4. Are closed under composition: if $h \colon \mathbb{N}^k \to \mathbb{N}$ is partial recursive and $g_1, \dots, g_k$ are partial recursive, then so is $h(g_1, \dots, g_k)$.
5. Are closed under primitive recursion: if $g \colon \mathbb{N}^k \to \mathbb{N}$ and $h \colon \mathbb{N}^{k+2} \to \mathbb{N}$ are partial recursive, then so is the function $f \colon \mathbb{N}^{k+1} \to \mathbb{N}$ defined by
$$f(y, x_1, \dots, x_k) = \begin{cases} g(x_1, \dots, x_k) & \text{if } y = 0 \\ h(y-1, f(y-1, x_1, \dots, x_k), x_1, \dots, x_k) & \text{if } y > 0 \end{cases}.$$
6. Are closed under the minimization operator: if $f \colon \mathbb{N}^{k+1} \to \mathbb{N}$ is partial recursive, the so is the function $g \colon \mathbb{N}^k \to \mathbb{N}$ where $g(x_1, \dots, x_k)$ is equal to the least y such that $f(y, x_1, \dots, x_k) = 0$ if such a y exists and for all $y' < y$ we have that $f(y', x_1, \dots, x_k)$ is defined. Otherwise, $g(x_1, \dots, x_k)$ is undefined.

We may then define the **recursive functions** to be all partial recursive functions $f \colon \mathbb{N}^k \to \mathbb{N}$ that are total. It should be clear that there is an algorithm to compute any recursive function since there are simple algorithms for computing constant, successor, and projection functions, composing them, and performing primitive recursion, and minimization.

Note here that the 6th item above is very important to the above definition. The smallest collection of functions closed under items 1-5 are called the **primitive recursive functions**. They were first defined by Gödel in his 1931 paper [G31] on incompleteness. However, there is an obvious algorithm for computing a function that is not a primitive recursive function: by diagonalizing against all primitive recursive functions (see Exercise 2.1). As we will discuss in Section 3.2, the partiality introduced in item 6 is necessary to prevent this sort of diagonalization.

Exercise 2.1. Show that there is an algorithm for computing a function that is not primitive recursive. [Hint: describe an algorithm for diagonalizing against all primitive recursive functions by defining a function g where $g(n) = f_n(n) + 1$ where f_n is the nth primitive recursive function according some listing of all such functions.]

Note also that sufficiently fast growing functions are not primitive recursive. Recall Knuth's up-arrow notation. The operation $\uparrow^1$ is exponentiation: $x \uparrow^1 y = x^y$, and the operation $x \uparrow^{n+1} y$ is the operation $\uparrow^n$ repeated y times. That is,

$$x \uparrow^{n+1} y = \underbrace{x \uparrow^n (x \uparrow^n (\cdots \uparrow^n x))}_{y \text{ times}}$$

So since repeated multiplication is exponentiation, $x \uparrow^1 y = x^y$. Similarly, $x \uparrow^2 y = \underbrace{x^{x^{x^{\cdots x}}}}_{y \text{ times}}$ is repeated exponentiation, which is sometimes called tetration. Define $x \uparrow^n 0 = 1$ for all n and all x. The following exercise shows that all primitive recursive functions grow slower than some $\uparrow^n$ function.

Exercise 2.2.

1. Show that if $f \colon \mathbb{N}^k \to \mathbb{N}$ is primitive recursive, then there exists some n so that $f(x_1, \ldots, x_k) \leq 2 \uparrow^n (\max_i(x_i) + 3)$.

2. Show that the function $f(x) = 2 \uparrow^x x$ is not primitive recursive.

So primitive recursion does not give the correct definition of computability. But what about the class of all recursive functions? Some mathematicians tried to argue that every possible function computable by an algorithm (in the informal sense) must be a recursive function, but no-one was particularly convinced. Their arguments basically boiled down to being unable to think of any counterexamples.

2.2 Turing machines

In 1936, Alan Turing suggested a new definition of computability via what are called now Turing machines. Turing's model leads to the same definition as the Herbrand-Gödel recursive functions (see Exercise 2.8). However, what was important about Turing's model is that it included a convincing philosophical argument that it encompassed all possible algorithms.

Turing's philosophical argument analyzes an idealized human agent executing an algorithm. We give this person an unlimited supply of paper, pencils, and erasers, and then observe everything they do during their execution of the algorithm until they finish. If you prefer, you can think of a machine executing the algorithm with access to an unlimited supply of memory. Turing made a few mild assumptions on this process:

1. Each piece of paper can only have finitely many possible things written on it and the pieces of paper are arranged in a fixed order, say, where the person can flip through the pages from left to right or right to left.

2. The agent executing the algorithm many only have finitely many possible "states" or "thoughts" they can think during the algorithm, no matter how long it runs, or what the input is. Another way of saying this is that there must be finitely many steps describing the algorithm. Note however, that there is unlimited space for storing information on the paper they have.

3. What the agent does at each stage is deterministic, and hence depends completely on what "state" they are in, and what is on the page in front of them.

For example, if we are adding two numbers, we cannot hold the entirety of arbitrarily long numbers in our head. Instead we should use the provided paper/memory for this. The steps of the

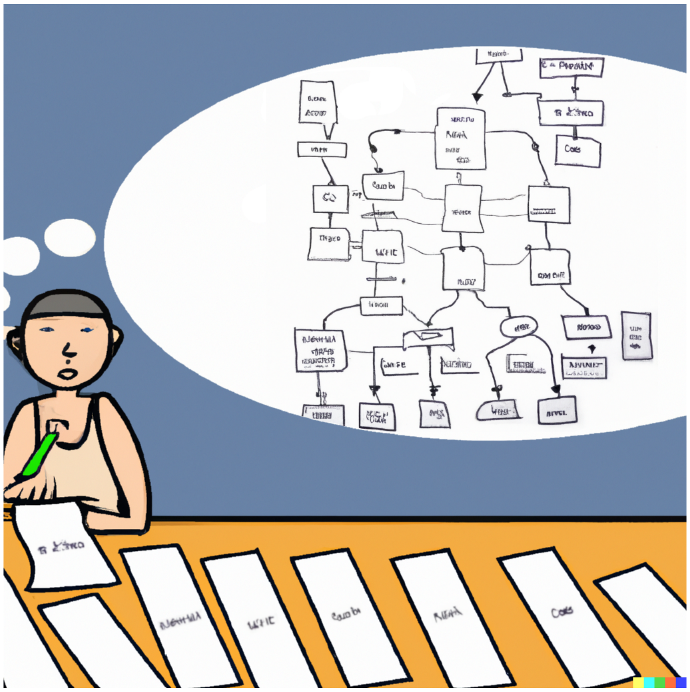


Figure 2: Turing's idealized agent executing an algorithm. This image was created with the assistance of DALL·E 2

algorithm for addition should be things like adding two digits, or carrying a one, or writing down part of the answer, etc.[3]

Turing broke down the execution of such an idealized description of an algorithm into finitely many steps, at which the agent does one or more of the following:

1. Erase what is on the current page and write something else.
2. Flip forward a page, or back a page.
3. Change to a new state
4. Decide they are finished performing the algorithm.

We now make a mathematical definition describing all processes having the features we've just described. Machines which execute these processes are called **Turing machines**. Instead of thinking of infinitely many pieces of paper, a Turing machine is described as having an infinite **tape** of infinitely many **cells** (what we were calling pages before) that are arranged left-to-right in order type $\mathbb{Z}$.

Definition 2.3 (Turing machine)**.** A Turing machine consists of:

[3] We remark that the determinism of assumption (3) is part of the usual definition of what an algorithm is. More generally, mathematicians and computer scientists often work with randomized algorithms, where we can flip a fair coin and use the result as part of the algorithm. However, probabilistic algorithms are much more important when we want to make efficient algorithms (e.g. in computational complexity theory) than in computability theory where we can compute how a probabilistic algorithm would perform on every sequence of possible coins flips, and the compute the probability it will behave any given way.

1. A finite set S called the **alphabet** of the Turing machine whose elements we call **symbols**. These symbols are the possible things that can be written on each tape cell of the Turing machine. The alphabet includes a distinguished blank symbol that we denote $\sqcup$.

2. A finite set Q of **states**, one of which is distinguished as the state at which the computation begins (the **starting state**), and one of which is distinguished as the state at which the computations finishes (the **halting state**).

3. A partial function $t\colon S \times Q \to S \times Q \times \{\mathrm{L}, \mathrm{R}\}$ called the **transition map** which specifies that if we are in the state q, the current cell contains the $a \in S$, and $t(a, q)) = (a', q', X)$, then we should change the current cell to be the symbol a', change to state q', and then move the tape of the Turing machine one cell in the direction X (i.e left or right) depending on whether X is L or R.

A **tape configuration** of a Turing machine is a function $T : \mathbb{Z} \to S$ A **state** of a Turing machine is a triple (T, q, n) where T is a tape configuration, $q \in Q$ is the state the machine is in, and $n \in \mathbb{Z}$ is the location of the Turing machine head.

Definition 2.4 (Execution of a Turing machine)**.** Given a Turing machine $M = (S, Q, q_{\text{start}}, q_{\text{halt}}, t)$, and a starting tape configuration $T_0\colon \mathbb{Z} \to S$, the corresponding **run** of the Turing machine is the sequence of states $(T_0, q_0, n_0), (T_1, q_1, n_0), \ldots$ defined as follows. The first state is $(T_0, q_0, 0)$ were q_0 is the distinguished starting state, and $n_0 = 0$. We define the rest of the sequence inductively. Given any (T_i, q_i, n_i), if q_i is the halting state, then the sequence giving the run of the Turing machine stops at this point (and is hence finite).Suppose, that q_i is not the halting state. Then if $t(T_i(n_i), q_i)$ is defined and equal to (a, q, X), then $q_{i+1} = q$, the head moves according to $n_{i+1} = n_i + 1$ if $X = R$ and $n_{i+1} = n_i - 1$ if $X = L$, and finally $T_{i+1}(n) = \begin{cases} a & \text{if } n = n_i \\ T_i(n) & \text{otherwise.} \end{cases}$

We give an example of a simple Turing machine:

Example 2.5. Consider the following Turing machine with alphabet $\Sigma = \{0, 1\}$ and three states $S = \{q_0, q_1, q_2\}$. q_0 is the distinguished starting state, and q_2 is the halting state. Finally, the transition map is given by the following table:

$$\begin{aligned}
\tau(0, q_0) &= (0, q_0, R) \\
\tau(1, q_0) &= (1, q_0, R) \\
\tau(B, q_0) &= (1, q_1, L) \\
\tau(0, q_1) &= (1, q_2, L) \\
\tau(1, q_1) &= (0, q_1, L) \\
\tau(B, q_1) &= (1, q_2, R)
\end{aligned}$$

The Turing machine we have just described is intended to do the following: if we write a number n in binary on the tape starting at the position 0, then this Turing machine will output $n + 1$ in binary. Roughly, what happens is that the machine will stay in the state q_0 and move right until it finds the least significant digit of the number, and then stays in the state q_1 repeatedly carrying a 1 and moving left until it halts at state q_2.

We illustrate an example run of this Turing machine in Figure 2.2. That is, what the tape looks like and where the head is at each stage of the computation.

Recall that if S is a set, then S^n is all **strings of length** n in S, and $S^{<\infty} = \cup_{n \geq 0} S^n$ is the set of all **strings** over S. Note that $S^{<\infty}$ includes the empty string.

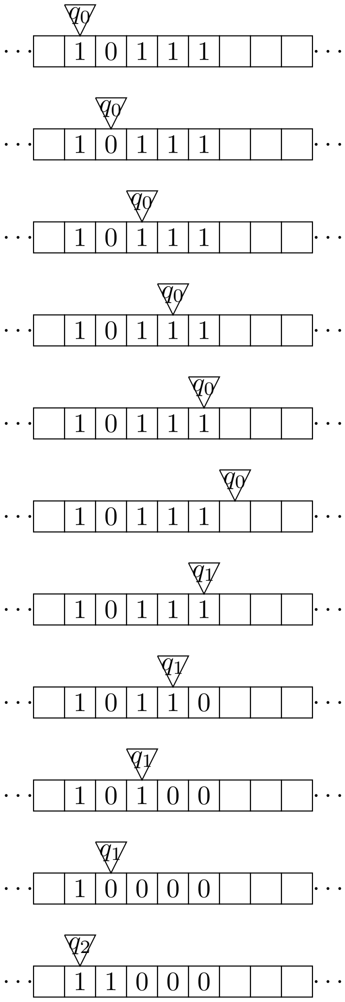


Figure 3: A run of the Turing machine from Example 2.5

Suppose S is a Turing machine alphabet, and $\Sigma = S \setminus \{\sqcup\}$ is all the non-blank symbols of the Turing machine. Say the tape configuration $T\colon \mathbb{Z} \to S$ **represents** a string $s \in S^n$ if

$$T(i) = \begin{cases} s(i) & \text{if } 0 \leq i < n \\ \sqcup & \text{otherwise} \end{cases}$$

so the tape configuration is blank except starting at 0 where the symbols of the string are written in the first n cells.

We can now define what it means for a function $f\colon \Sigma^{<\infty} \to \Sigma^{<\infty}$ to be computable.

Definition 2.6. Let Σ be a finite alphabet that does not include the blank symbol $\sqcup$. Say that a partial function $f\colon \Sigma^{<\infty} \to \Sigma^{<\infty}$ is partial computable iff there is a Turing machine M with alphabet whose non-blank symbols are Σ so that if we run the Turing machine M beginning with a tape configuration representing $s \in \Sigma^{<\infty}$, then if $f(s)$ is defined, the machine halts after a finite number of steps in a tape configuration representing $f(s)$, and otherwise if $f(s)$ is undefined, then the Turing machine does not halt.

Exercise 2.7. Show that the partial computable functions from $\Sigma^{<\infty} \to \Sigma^{<\infty}$ are closed under composition.

We can define partial computable functions from $\mathbb{N}^k \to \mathbb{N}$ by representing natural numbers in an alphabet in some reasonable way. For example, say that a function $f\colon \mathbb{N} \to \mathbb{N}$ is computable if the function $1^n \mapsto 1^{f(n)}$ is computable, where 1^n is the string with n 1s in a row in the alphabet $\Sigma = \{1\}$.

This definition may seem arbitrary. Why are we representing the number n with n 1s written in a row, instead of represeting numbers in binary, or using Roman numerals? Of course, we are forced to represent numbers in some way for our Turing machines; our definition of Turing machine doesn't allow an infinite alphabet, so we can't just input a natural number on a single cell. Fortunately, while each of these choices is indeed rather arbitrary, they all give equivalent definitions. This is because there are Turing machines that convert between all of these different ways of representing numbers, and so if we can compute some function in any of these different representations then we can compute it in all of the others, by effectively "composing" the Turing machine computing the function along with two other Turing machines converting between how we represent the input and output.

This robustness is a fact that we will often use. Whenever we deal with natural numbers, polynomials, finite trees, equations in number theory, tilesets, strings, etc., the way that we define computable subsets or functions of these objects will not depend on the way we represent them to the computer, provided we do this in a reasonable way; all such definitions will be equivalent. from now on we won't bother to explicitly define ways of representing these types of objects to a computer. We will take it as given that we have implicitly picked some reasonable way of representing such objects for our Turing machines, and that this choice doesn't affect the definition of which such functions or sets are computable.

2.3 The Church-Turing thesis

The assertion that the our analysis above is a good one, and we've really captured the essence of what an algorithm is called the Church-Turing thesis, or Turing's thesis. Having a name for the idea that a definition is the correct formalization of an intuitive notion is quite unusual in mathematics (for instance, there isn't a "Cauchy's thesis" asserting the ϵ-δ-definition of continuity is the correct definition of the intuitive idea of "continuous"). We simply say this is the correct mathematical definition of continuity. Similarly, for the purposes of these notes, we'll simply take this as correct mathematical definition of computability (a fact which is not seriously disputed).

It is a tedious but important exercise that Turing's definition is the same as the earlier definition of recursive functions (and also equivalent to Church's λ-calculus, Minsky's register machines, the programs that you can write in python, etc.).

Exercise 2.8. Prove that every partial computable function from $\mathbb{N}^k \to \mathbb{N}$ is computable by a Turing machine [Hint: show that the functions computable by a Turing machine have all the closure properties 1-6 in the definition of partial recursive functions.]

Along similar lines, it is worth noting another way in which our definition of a computable function is robust: if we change the definition of what a Turing machine is in any inessential way (restricting the alphabet, adding more tapes, changing the tapes to be one-sided instead of two-sided, or higher-dimensional) this will not affect our definition of what a computable function is. The way we prove such theorems is by showing that when we do any a change to the definition, both kinds of Turing machine can "simulate" each other. For example, we can argue as follows that a Turing machine with alphabet $\{0, 1, \sqcup\}$ can "simulate" a Turing machine with alphabet $\{0, 1, 2, 3, \sqcup\}$ (and a similar trick will work for an alphabet of any size). We can have each two adjacent cells in our machine with alphabet $\{0, 1, \sqcup\}$ represent a single cell in a machine with alphabet $\{0, 1, 2, 3, \sqcup\}$ via some correspondence like $0 \mapsto 00$, $1 \mapsto 01$, $2 \mapsto 10$, $3 \mapsto 11$, and $\sqcup \mapsto \sqcup\sqcup$. We can then replace each single state in the machine with alphabet $\{0, 1, 2, 3, \sqcup\}$ with a handful of states in the machine with alphabet $\{0, 1, \sqcup\}$ which scan each two adjacent cells to determine what they represent, then overwrite each of them according to the transition function, then move to the next pair of cells.

Exercise 2.9. Consider the following variation on the definition of a Turing machine: instead of a transition function $t\colon S \times Q \to S \times Q \times \{L, R\}$ there is a transition function $t\colon S \times Q \to S \times Q \times \mathbb{Z}$ where if $t(a, q) = (a', q', n)$, then the symbol in the current cell is changed to a', the state is changed to q', and the head of the Turing machine moves by n cells. Explain why this new type of machine can compute exactly the same partial functions as the usual sort of Turing machine.

2.4 Basic properties of computable functions

An important consequence of our definition of a Turing machine is that we can explicitly list out all possible Turing machines. Indeed, we can do this in a computable way so that its possible to analyze and understand Turing machines using other Turing machines.

To be very clear, we could do something like the following: since a Turing machine is specified by just its alphabet, its states, and its transition function, and all these objects are finite, we can assume that the alphabet and states are just sets of the form $\{0, \ldots, n\}$ and then order a list of Turing machines by the size of their alphabet and numbers of states, where for each such pair, we list all possible transition functions in lexicographic order. As usual, the exact way we do this shouldn't matter provided it is computable. All we want to make explicit is that there is some way of making this list which is understandable by a Turing machine. We will fix some such method for the rest of these notes:

Definition 2.10. Fix a computable listing of all the Turing machines. Throughout these notes we will use the notation φ_n to denote the nth partial computable function computed by this nth Turing machine.

We have said above that we will use the notation φ_n to indicate the nth partial computable function, but we have not said between what type of sets. Are we talking about functions from $\mathbb{N} \to \mathbb{N}$, or functions from $\{0, 1\}^{<\infty} \to \{0, 1\}^{<\infty}$, or from $\mathbb{N}^2 \to \mathbb{N}$, or something else? This is a standard abuse of notation were we can regard the nth machine as operating on any of these kinds of inputs. So if we write $\varphi_n(i, j)$, we mean the partial function from $\mathbb{N}^2 \to \mathbb{N}$ computed by the nth Turing machine. If instead we write $\varphi_n(i)$ we mean to consider the nth Turing machine as computing a partial function from $\mathbb{N} \to \mathbb{N}$.

A vitally important feature of Turing's definition is the following theorem:

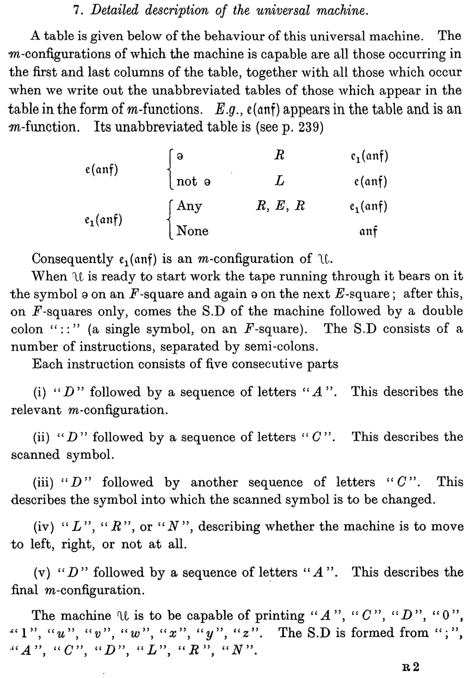
Theorem 2.11. *There exists a Turing machine (called a* ***universal Turing machine****) u which defines a function of two variables so that $\varphi_u(m, n) = \varphi_m(n)$ for all m, n. That is, $\varphi_u(m, n)$ halts and outputs $\varphi_m(n)$ if and only if $\varphi_m(n)$ halts.*

We will not prove this theorem. The proof simply consists of writing a Turing machine program which simulates any other Turing machine. Precisely, the universal machine takes as input a description of another Turing machine and an initial state of its tape, and then step by step updates our representation of its tape according to the given Turing machine program. Finally, if we have entered a halting state, then we stop and output the value that is left on the tape.

If you are particularly eager to look at the details of this proof, then you can find a proof of this theorem in many elementary computability theory books (or take Math 114C at UCLA).

Certainly, one should believe that taking a description of a Turing machine and then simulating how it runs step by step is an example of an algorithm. Thus, if one believes that our analysis of a Turing machine being able to implement any algorithm, you ought to believe Theorem 2.11.

We now give two more important properties of the set of computable functions that we will often use.



7. *Detailed description of the universal machine.*

A table is given below of the behaviour of this universal machine. The m-configurations of which the machine is capable are all those occurring in the first and last columns of the table, together with all those which occur when we write out the unabbreviated tables of those which appear in the table in the form of m-functions. *E.g.*, $\mathfrak{e}(\mathfrak{anf})$ appears in the table and is an m-function. Its unabbreviated table is (see p. 239)

$\mathfrak{e}(\mathfrak{anf})$	ə	R	$\mathfrak{e}_1(\mathfrak{anf})$
	not ə	L	$\mathfrak{e}(\mathfrak{anf})$
$\mathfrak{e}_1(\mathfrak{anf})$	Any	R, E, R	$\mathfrak{e}_1(\mathfrak{anf})$
	None		$\mathfrak{anf}$

Consequently $\mathfrak{e}_1(\mathfrak{anf})$ is an m-configuration of $\mathfrak{U}$.

When $\mathfrak{U}$ is ready to start work the tape running through it bears on it the symbol ə on an F-square and again ə on the next E-square; after this, on F-squares only, comes the S.D of the machine followed by a double colon "::" (a single symbol, on an F-square). The S.D consists of a number of instructions, separated by semi-colons.

Each instruction consists of five consecutive parts

(i) "D" followed by a sequence of letters "A". This describes the relevant m-configuration.

(ii) "D" followed by a sequence of letters "C". This describes the scanned symbol.

(iii) "D" followed by another sequence of letters "C". This describes the symbol into which the scanned symbol is to be changed.

(iv) "L", "R", or "N", describing whether the machine is to move to left, right, or not at all.

(v) "D" followed by a sequence of letters "A". This describes the final m-configuration.

The machine $\mathfrak{U}$ is to be capable of printing "A", "C", "D", "0", "1", "u", "v", "w", "x", "y", "z". The S.D is formed from ";", "A", "C", "D", "L", "R", "N".

R 2

Figure 4: The start of the description of a universal Turing machine from Turing's original paper

Lemma 2.12 (The padding lemma)**.** *Given any n, there are infinitely many Turing machine programs computing the nth Turing machine program φ_n. Moreover, these programs can be found computably. That is, there is a computable injective function $f\colon \mathbb{N}^2 \to \mathbb{N}$ so that for every n and every i, $\varphi_n = \varphi_{f(n,i)}$.*

Proof sketch. Given any Turing machine program, we can add extra unused states to make a larger program which computes the same partial function. □

Note that by two partial functions being equal, we mean that their domains are equal and they take the same value on all the elements of their domain.

We record one last more technical theorem about computable functions.

Theorem 2.13 (The S-m-n theorem)**.** *There is a injective computable function $s\colon \mathbb{N}^2 \to \mathbb{N}$ so that for all x, y, z*

$$\varphi_{s(x,y)}(z) = \varphi_x(y, z).$$

Proof sketch. The program defining $s(x, y)$ outputs a computer program which does the following. First, it writes y on the tape of the Turing machine before where z is written, then it executes the program x. It is easy to see that such a function is injective (or, we could define $s(x, y)$ by induction and use the padding lemma at each step to ensure that s is injective. □

As we will see going forward, having a precise mathematical definition of what a Turing machine is useful mathematically for proving that Turing machines *can't* do certain things. However, when we want to show that something *is* computable, it's almost never a good idea to write a giant Turing machine program to do it. It is almost always much more convincing and clear to simply describe informally the way that you can go about calculating the thing in question. For example, the following examples of computable functions:

Example 2.14. There is a computer program that takes as input a one-variable integer polynomial, and outputs whether it has an integer root. Given such a polynomial $a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0$ it first computes $b = \max(|a_n|, |a_{n-1}|, \ldots, |a_0|)$. Then since any root of the polynomial must contained in $[-nb^n, nb^n]$, the program tries each of these integers in turn to determine whether they are a root of the polynomial.

Here is another example:

Example 2.15. There is a computer program takes as input an integer representing a Turing machine that computes a bijection $f : \mathbb{N} \to \mathbb{N}$ and computes its inverse. To find $f^{-1}(n)$, the program calculates $f(0), f(1), \ldots$ until it finds some m such that $f(m) = n$. Then it outputs m.

The above level of detail is exactly what you should use when you're writing proofs about computability theory, and it is about the level of detail you will find throughout the computer science and computability theory literature.

2.5 What sorts of questions can we analyze using tools from computability?

We briefly discuss the types of questions we can analyze using computability theory. First, there is an algorithm implementing any function with a finite set of inputs and outputs. Hence, finite classes of problems become trivial from the perspective of computability. So for example, there is an algorithm which correctly outputs the answer to the twin prime conjecture. It is one of the following two algorithms:

$$\text{Algorithm 1: Output true}$$

$$\text{Algorithm 2: Output false}$$

(But it is an open problem which algorithm is the correct one). A more suitable related question to analyze from the perspective of computability would be the following: is there an algorithm which takes a natural number n as input and outputs whether there are infinitely many primes p such that $p + n$ is also prime? This question has recently been proved to have a positive answer; Yitang Zhang has shown that for any even number $n \geq 7 \cdot 10^7$, there are infinitely many primes with gap n[4]. There is therefore some algorithm which outputs yes for any even $n \geq 7 \cdot 10^7$, and for the finitely many values $n < 7 \cdot 10^7$, the algorithm it outputs the correct answers (though we do not know what these finitely many correct answers are).

Asking whether an algorithm exists for doing some task is sometimes a very different question from asking what the algorithm is, as we've demonstrated above. We will sometimes be able to prove that there is an algorithm to do something without being able to specify exactly what the algorithm is, or even narrow it down to less than countably many possibilities.

Exercise 2.16. Is there an algorithm which takes a positive integer n as input, and outputs "yes" if there is a sequence of n consecutive 7s in the decimal expansion of π and "no" if there is not?

[4] Subsequent work of the Polymath Project and James Maynard have since reduced this number to 246

3 Incomputability

3.1 The halting problem

Having defined what computability means, we now turn to incomputability. We begin with a very simple observation:

Theorem 3.1. *There is a function $f : \mathbb{N} \to \mathbb{N}$ that is not computable.*

Proof. There are countably many Turing machine programs, and uncountably many functions $\mathbb{N} \to \mathbb{N}$. □

Next, we'd like to give an explicit example of an incomputable function. First we give a definition.

Definition 3.2. We say that a computation $\varphi_n(m)$ halts, and write $\varphi_n(m)\downarrow$ if the computation of $\varphi_n(m)$ eventually terminates in a halting state. We say that $\varphi_n(m)$ does not halt and write $\varphi_n(m)\uparrow$ otherwise.

Our explicit example of an incomputable function comes from copying Cantor's diagonal argument:

Proposition 3.3. *The function*

$$f(n) = \begin{cases} \varphi_n(n) + 1 & \textit{if } \varphi_n(n)\downarrow \\ 0 & \textit{otherwise} \end{cases}$$

is not computable.

Proof. If f was computable, then we would have $f = \varphi_m$ for some m. Now $\varphi_m = f$ is total, so $f(m) = \varphi_m(m) + 1$ by definition of f. But this is a contradiction since $f(m) = \varphi_m(m)$ by our assumption $f = \varphi_m$. □

Our next goal is to find a more natural example of incomputability than Proposition 3.3. First we define what it means for a subset of $\mathbb{N}$ to be computable.

Definition 3.4. A set $A \subseteq \mathbb{N}^k$ is computable if its characteristic function

$$1_A(x_1, \ldots, x_k) = \begin{cases} 1 & \text{if } (x_1, \ldots, x_k) \in A \\ 0 & \text{if } (x_1, \ldots, x_k) \notin A \end{cases}$$

is computable.

For example, if $A \subseteq \mathbb{N}$ is computable, then $\mathbb{N} \setminus A$ is computable.

Our more natural example of incomputability is the **halting problem**.

Theorem 3.5 (Undecidability of the halting problem)**.** *The set $K = \{n \colon \varphi_n(n)\downarrow\}$ is not computable.*

Proof. If K was computable, we could then also compute the function f from Proposition 3.3 as follows: On input n, first compute whether $n \in K$. If $n \notin K$, then output 0. If $n \in K$, then simulate $\varphi_n(n)$ using a universal Turing machine (we know it will eventually halt) and then output $\varphi_n(n) + 1$. □

Many interesting mathematical statements can be transformed into questions about whether computer programs eventually halt. For example, there is a computer which examines each even integer $2, 4, 6, 8 \ldots$ in turn and halts if this integer can not be expressed as a sum of two primes. This computer program eventually halts if and only if the Goldbach conjecture is false. Thus, it shouldn't be that surprising that there is no easy way of figuring out whether a computer program eventually stops running; if there was, there would be an easy way of determining the truth of a large number of very difficult mathematical problems.

Note that the fact that we are running φ_n on input n is not so important to incomputability of K. For example:

Exercise 3.6. Show that $\{n \colon \varphi_n(0)\!\downarrow\}$ is incomputable.

3.2 The need for studying partial computable functions

We are mostly only interested in studying total computable functions in computability theory. Yet, our theory includes partial functions. One reason for this is that there is no set of functions $(f_n)_{n \in \mathbb{N}}$ (which we would like to be all total computable functions from $\mathbb{N} \to \mathbb{N}$) with the following properties:

1. The functions f_n are all total.

2. The functions f_n are closed under composition and include the function $x \mapsto x + 1$.

3. There function $x \mapsto f_x(x)$ is in the collection (i.e. there is a computable "universal machine").

This is because if we had all three properties, the function $x \mapsto f_x(x) + 1$ would be in this collection (by composing the function from (3) with the addition function using (2)). So $x \mapsto f_x(x)$ would be equal to f_n for some n, but then $f_n(n)$ is defined (since every f_n is total by (1)) and $f_n(n) = f_n(n) + 1$. Contradiction!

To make a reasonable theory of computation we have to include property (2). We also should have property (3): if executing the nth algorithm on the nth input is not computable, then we haven't made a notion of algorithm that we can actually compute. So we are forced to drop property (1).

Questions about what computations halt, and what programs are total, etc. will be quite important in computability theory.

3.3 Rice's theorem

The halting problem poses a very serious limitation on our ability to computably understand the behavior of Turing machines. However, the situation is actually much worse. Using the incomputability of the halting problem, we can show that there is not a single nontrivial property of partial computable functions which we can distinguish in a computable manner.

Definition 3.7. Say that a set $A \subseteq \mathbb{N}$ is an **index set** if for all n, m, if φ_n and φ_m compute the same partial function (i.e. $\varphi_n = \varphi_m$), then $n \in A \leftrightarrow m \in A$.

So an index set A consists of all programs that give partial computable functions that have some property. For example, all programs that compute total functions, all functions φ_n so that $\varphi_n(m)\!\downarrow$ for some n, all n such that $\varphi_n(m) = m + 1$, etc.

It turns out that there are no nontrivial computable index sets:

Theorem 3.8 (Rice's theorem)**.** *Suppose $A \subseteq \mathbb{N}$ is a computable index set. Then either $A = \mathbb{N}$ or $A = \emptyset$.*

Proof. By contradiction assume that A is a computable index set such that $A \neq \mathbb{N}$ and $A \neq \emptyset$. Let φ_{n_0} a partial computable function which is undefined on every input. Since A is computable iff its complement is computable, by exchanging A for its complement, we may as well assume $n_0 \in A$. Since $A \neq \mathbb{N}$, there is some n_1 such that $n_1 \notin C$.

Now we obtain a contradiction by showing that the halting problem is computable. Consider the partial computable function $f(n, m)$ deifned as follows to compute $f(n, m)$ we first compute

$\varphi_n(n)$, and if this halts, then we compute $\varphi_{n_1}(m)$ and output the answer. Let x be the program computing f so $\varphi_x(n,m) = f(n,m)$, and by the S-m-n theorem there is a computable s so that $\varphi_{s(x,n)}(m) = f(n,m)$. Now by the definition of f, if $\varphi_n(n)$ does not halt, then $\varphi_{s(x,n)}(m) = f(n,m)$ is undefined for all m, so $\varphi_{s(x,n)} = \varphi_{n_0}$, so $s(x,n) \in A$ since $n_0 \in A$. If $\varphi_n(n)$ does halt, then $\varphi_{s(x,n)} = f(n,m) = \varphi_{n_1}(m)$ for all m, so $\varphi_{s(x,n)} = \varphi_{n_1}$ and so $s(x,n) \notin A$ since $n_1 \notin C$.

Thus, $\varphi_n(n)$ halts iff $s(x,n) \notin A$. Since we are assuming A is computable and s is computable, we conclude that the halting problem K is computable. Contradiction! □

We are explicitly explaining where we are using the S-m-n theorem above. However, this type of argument – computably producing a computer program from finitely many inputs which has some desired behavior – is constantly used in the computability, and most books don't bother pointing out that by doing this they are implicitly using the S-m-n theorem. So you'll often see definitions like the following: Define a computable function $g\colon \mathbb{N} \to \mathbb{N}$ so that

$$\varphi_{g(n)}(m) = \begin{cases} \varphi_{n_1}(m) & \text{if } \varphi_n(n)\downarrow \\ \text{undefined} & \text{if } \varphi_n(n)\uparrow. \end{cases}$$

Then $\varphi_{g(n)} = \varphi_{n_1}$ if $\varphi_n(n)\downarrow$ and $\varphi_{g(n)} = \varphi_{n_0}$ if $\varphi_n(n)\uparrow$.

4 Computably enumerable sets

The central focus of computability theory is on understanding, analyzing, and classifying the *incomputability* of subsets of $\mathbb{N}$. In this section, we discuss an important class of subsets of $\mathbb{N}$ which includes sets that are just beyond being computable.

Definition 4.1. A set is **computably enumerable** or c.e. if it is the domain of a partial computable function. We let W_e denote the domain of the eth partial computable function φ_e; this is the eth c.e. set.

4.1 Characterizing c.e. sets

We have already seen an example of a c.e. set: the halting problem K. Consider the partial computable function $f\colon \mathbb{N} \to \mathbb{N}$ defined by letting $f(n)$ compute $\varphi_n(n)$, and if this computation halts, then output n. Then the domain of f is K.

K is connected to closely connected to the class of all c.e. sets. We will show eventually that in a precise sense, the halting problem is the "most complicated" c.e. set (see Theorem 5.5).

Proposition 4.2. *The following are equivalent for a set $A \subseteq \mathbb{N}$.*

1. *A is computably enumerable.*
2. *There is a computable relation R on $\mathbb{N}^{k+1}$ so that*

$$x \in A \leftrightarrow \exists y_1, \dots, y_k R(x, y_1, \dots, y_k).$$

Proof. First, suppose $f\colon \mathbb{N} \to \mathbb{N}$ is a partial computable function. Let $R(x, y)$ be defined by $R(x, y)$ is true iff the computation of $f(x)$ halts in y steps.

Conversely, suppose R is a computable relation. Then let $f\colon \mathbb{N} \to \mathbb{N}$ be defined by letting $f(x)$ be defined by searching through all tuples $(y_1, \dots, y_k)$ in lexicographic order and then outputting x once we find some tuple $(y_1, \dots, y_k)$ such that $R(x, y_1, \dots, y_k)$ is true. □

Using the above proposition, we have a more natural way of seeing that K is a c.e. set:

$$n \in K \leftrightarrow \exists s \text{``}\varphi_n(n) \text{ halts in } s \text{ steps''}.$$

Since the relation "$\varphi_n(n)$ halts in s steps" is a computable relation on pairs of n, s, K is c.e. by the above proposition.

If R and S are both computable relations, then $R \wedge S$ and $R \vee S$ are also computable relations. From this observation, it is easy to show the following:

Exercise 4.3. Show that if $A, B \subseteq \mathbb{N}$ are c.e. then $A \cup B$ and $A \cap B$ are c.e.

Our next characterization of c.e. sets will be via sequences of finite sets. We start by briefly describing how finite sets can be computably represented. Fix some computable bijection $\xi\colon \mathbb{N} \to [\mathbb{N}]^{<\infty}$ between $\mathbb{N}$ and all finite subsets of $\mathbb{N}$. For example, let ξ enumerate all finite sets by beginning with the empty set, then all sets with maximum element 0, all sets with maximum element 1, and so on in lexicographic order. Using ξ, we can regard each number n as representing a finite set via this bijection. By a **computable sequence of finite sets** $(A_s)_{s\in\mathbb{N}}$ we mean a computable function $f\colon \mathbb{N} \to \mathbb{N}$ so that $A_s = \xi(f(s))$. That is, there is a program computing the sth element A_s of the sequence for every s.

Definition 4.4. A **computable enumeration** of a set $A \subseteq \mathbb{N}$ is a computable sequence of finite sets $(A_s)_{s\in\mathbb{N}}$ so that

- $A_0 = \emptyset$.
- $(A_s)_{s\in\mathbb{N}}$ is increasing ($A_s \subseteq A_{s+1}$ for all s).
- For all s, $|A_{s+1} \setminus A_s| \leq 1$, so at most one new number is added to each A_s. If $x \in A_{s+1} \setminus A_s$, we say that x is **enumerated at stage** s.
- $\bigcup A_s = A$.

You should think of a computable enumeration of a set as a program that runs for infinitely many steps. At stage s, the program computes the set A_s which may contain at most one new number. Over time the program "enumerates" the entire set A. Once x is enumerated, we know that $x \in A$. However, we can never be sure at any finite stages s whether $x \notin A$; x could always be enumerated at some later stage .

Proposition 4.5. *The following are equivalent for a set $A \subseteq \mathbb{N}$.*

1. *A is computably enumerable.*
2. *There is a computable sequence of finite sets $(A_s)_{s\in\mathbb{N}}$ so that $A = \bigcup A_s$.*
3. *There is a computable enumeration of A.*

Proof. (1) → (2): let $A_s = \{n \leq s \colon f(n) \text{ halts in } \leq s \text{ steps}\}$.

(2) → (3): We make a computable enumeration $(B_s)_{s\in\mathbb{N}}$ of A. Let $B_0 = \emptyset$. Suppose we have already defined B_s so that $B_s = \bigcup_{i\leq n} A_i$. Let $k = |A_{n+1} \setminus B_s|$. If $k = 0$, then let $B_{s+1} = B_s$. Otherwise, let $A_{n+1} \setminus A_n = \{m_1, \ldots, m_k\}$, and define $B_{s+i} = B_s \cup \{m_1, \ldots, m_i\}$.

(3) → (1) Given a computable enumeration $(A_s)_{s\in\mathbb{N}}$, of a set $A \subseteq \mathbb{N}$, we can define a partial computable function $f \colon \mathbb{N} \to \mathbb{N}$ where $f(x) = s$ if x is enumerated at the sth stage of the enumeration procedure (i.e. $x \in A_{s+1} \setminus A_s$). To compute $f(x)$, we compute $A_0, A_1, \ldots$ in turn until we find some s so that $A_{s+1} \setminus A_s = \{x\}$. Then we halt outputting s. The domain of this partial computable function f is our computably enumerable set. □

In computability theory, we often verify that sets are c.e. using Proposition 4.2. The definition of most c.e. sets naturally have this logical form. When working with c.e. sets, we typically think about them intuitively using enumerations and Proposition 4.5. This is the form we will use when proving some basic properties of c.e. sets in the next couple sections.

We give one last characterization of c.e. sets as an exercise:

Exercise 4.6. A is computably enumerable iff A is the range of a partial computable function iff $A = \emptyset$ or A is the range of a total computable function.

4.2 Computable vs c.e. sets

In this section, we'll discuss several connections between computable and c.e. sets. We begin with the following observation:

Proposition 4.7. *Every computable set is c.e.*

Proof. If $A \subseteq \mathbb{N}$ is computable, then $A_s = A \upharpoonright s$ is a computable enumeration of A. □

Above, we are using the standard abuse of notation identifying each natural numbers s with the set of all smaller natural numbers $\{0, 1, \ldots, s-1\}$.

We know that the converse of the above proposition is false: the halting problem K is a c.e. set that is not computable.

We have the following relationship between c.e. sets and computability.

Theorem 4.8. *A is computable iff both A and $\mathbb{N} \setminus A$ are c.e.*

Proof. $\Rightarrow$: A is computable iff $\mathbb{N} \setminus A$ is computable. Hence $\mathcal{A}$ and $\mathbb{N} \setminus A$ are both c.e. by Proposition 4.7.

$\Leftarrow$: Fix computable enumerations of both A and $\mathbb{N} \setminus A$. We can compute whether $x \in A$ using the following procedure: run the enumeration of both A and $\mathbb{N} \setminus A$. Eventually x must either be enumerated into A or be enumerated into $\mathbb{N} \setminus A$. Once this happens we can output whether $x \in A$ or $x \notin A$. □

We give another connection between c.e. sets and computability.

Theorem 4.9. *A total function $f\colon \mathbb{N} \to \mathbb{N}$ is computable iff its graph*

$$\mathit{graph}(f) = \{(x, y)\colon f(x) = y\}$$

is a c.e. subset of $\mathbb{N}^2$.

Proof. If f is computable, then it is clear that its graph is computable (and hence c.e.): to determine if $(x, y) \in \operatorname{graph}(f)$, compute $f(x)$, and then output that $(x, y) \in \operatorname{graph}(f)$ iff $f(x) = y$.

If the graph of f is computably enumerable, then to compute $f(x)$, run the enumeration of $\operatorname{graph}(f)$ until some element (z, y) is enumerated such that $z = x$. Then $f(x) = y$. □

Exercise 4.10 (Uniformization of c.e. relations)**.** Suppose $A \subseteq \mathbb{N}^2$ is c.e. Then show that there is a partial computable function $f\colon \mathbb{N} \to \mathbb{N}$ such that $f(x)$ is defined iff $\exists y (x, y) \in A$, and for every $x \in \operatorname{dom}(f)$, $(x, f(x)) \in A$.

To help us understand when a c.e. set is computable, we having the following propositions which relate the computability of c.e. sets to properties of their enumerations:

Proposition 4.11. *Suppose A is computably enumerable. Then $A \subseteq \mathbb{N}$ is computable iff there is a computable enumeration $(A_s)_{s \in \mathbb{N}}$ of A that is in increasing order. That is, if $x \in A_{s+1} \setminus A_s$ and $x' \in A_{s'+1} \setminus A_{s'}$, where $s < s'$, then $x < x'$. So if x' is enumerated at a later stage than x, then $x' > x$.*

Proof. $\Rightarrow$: If A is computable, then $A_s = A \upharpoonright s$ is such an enumeration in increasing order.

$\Leftarrow$: Suppose A is c.e. and has an enumeration in increasing order. If A is finite, then clearly A is computable. Otherwise, if A is infinite, then to compute whether $x \in A$, search for a stage s so that $\max(A_s) \geq x$. Eventually we must find such a stage since A is infinite. Then $x \in A \leftrightarrow x \in A_s$. □

Roughly, the contrapositive of the above proposition says the following: If a c.e. set is incomputable, then it must be that the order of any enumeration is very complicated and impossible to make in increasing order.

A different corollary of the Proposition 4.11 is the following:

Exercise 4.12. Suppose A is an infinite c.e. set. Show that there is an infinite computable subset $B \subseteq A$. [Hint: fix an enumeration $(A_s)_{s \in \mathbb{N}}$ of A. Then let $B = \{x\colon (\exists s)[x \in (A_{s+1} \setminus A_s) \wedge x = \max(A_{s+1})\}$ be the elements enumerated into A that are greater than any previous elements that have been enumerated.]

Another way of understanding when a c.e. set is incomputable is via moduli. If (A_s) is a computable enumeration, then a **modulus** $m\colon \mathbb{N} \to \mathbb{N}$ is a function so that for all x, $A \upharpoonright x = A_{m(x)} \upharpoonright x$. That is, m tells us a large enough stage of the enumeration so that all the elements less than x that will ever be enumerated into A will be enumerated by stage $m(x)$.

Exercise 4.13. A c.e. set A is computable iff it has a computable enumeration that has a computable modulus.

Thus, the incomputability of a c.e. set arises from it being impossible, given some x, to compute a large enough stage s to be sure that all elements less than x that will ever be enumerated will have been enumerated by stage s. Any modulus must grow incomputably fast.

4.3 Computably inseparable c.e. sets

Way say that two disjoint sets $A, B \subseteq \mathbb{N}$ are **computably inseparable** if there is no computable set $C \subseteq \mathbb{N}$ such that $A \subseteq C$, and $C \cap B = \emptyset$.

Theorem 4.14. *$A = \{x \colon \varphi_x(x) = 0\}$ and $B = \{x \colon \varphi_x(x) = 1\}$ are computably inseparable c.e. sets.*

Proof. A and B are c.e. by Proposition 4.2.

Suppose C was a computable separating set. Then there is some e such that φ_e is a total computable function computing the characteristic function of C. But then $e \in C \to \varphi_e(e) = 1 \to e \in B \to e \notin C$. Similarly, $e \notin C \to \varphi_e(e) = 0 \to e \in A \to e \in C$. Contradiction! □

Computably inseparable c.e. sets are sometimes useful for proving incomputability results. For example, we will use them to prove Tennenbaum's theorem that there is no computable nonstandard model of PA. They also give one way of solving the following exercise:

Exercise 4.15. Show that there is a partial computable function $f \colon \mathbb{N} \to \mathbb{N}$ with no total computable extension $g \supseteq f$. That is, there is no total computable function $g \colon \mathbb{N} \to \mathbb{N}$ so that if $x \in \mathrm{dom}(f)$, then $g(x) = f(x)$.

In contrast to the computable inseparability of c.e. sets, we can computably separate co-c.e. sets.

Definition 4.16. A set $A \subseteq \mathbb{N}$ is co-c.e. iff $\mathbb{N} \setminus A$ is c.e.

Exercise 4.17 (Computable separation of co-c.e. sets)**.** Suppose $A, B \subseteq \mathbb{N}$ are disjoint co-c.e. Then show there is a computable set C such that $A \subseteq C$ and $C \cap B = \emptyset$.

5 Reducibilities and Myhill's theorem

5.1 Many-one and 1-1 reducibility

Reducibilities give us a way of comparing the relative incomputability of subset of $\mathbb{N}$.

Definition 5.1. Suppose $A, B \subseteq \mathbb{N}$. Say that A is many-one reducible to B and write $A \leq_m B$ if there is a computable function $f \colon \mathbb{N} \to \mathbb{N}$ such that $x \in A \leftrightarrow f(x) \in B$. In this case we call f a many-one reduction from A to B. Similarly say $A \leq_1 B$ and say A is one-one reducible to B if there is an injective computable function $f \colon \mathbb{N} \to \mathbb{N}$ such that $x \in A \leftrightarrow f(x) \in B$.

Note that the reducibilities $\leq_m$ and $\leq_1$ are reflexive (via the identity function) and transitive (by composing reductions). Hence, the relation where $A \equiv_m B$ iff $A \leq_m B$ and $B \leq_m A$ is an equivalence relation called many-one equivalence. We define $\equiv_1$ similarly.

Intuitively, $A \leq_m B$ means that A is "at least as easy" to compute as B, in the sense that we can computably transform any question about membership in A into a question about membership in B.

As an example, let $\mathrm{TOT} = \{x \colon \varphi_x \text{ is total}\}$. Both K and TOT are incomputable (note that TOT is a nontrivial index set). We claim that $K \leq_m \mathrm{TOT}$. To see this, let $f \colon \mathbb{N} \to \mathbb{N}$ be the computable function where $f(x)$ is the program that on any input, first computes $\varphi_x(x)$, and then halts outputting 1 if $\varphi_x(x)$ ever halts. So

$$\varphi_{f(x)}(n) = \begin{cases} 1 & \text{if } \varphi_x(x){\downarrow} \\ \text{undefined} & \text{otherwise} \end{cases}.$$

Then $\varphi_x(x){\downarrow}$ iff $\varphi_{f(x)}$ is total, and so f is a many-one reduction from K to TOT.

Exercise 5.2. Show that $\mathrm{TOT} \not\leq_m K$, so in the sense of many-one reducibility, TOT is strictly more complicated than K. [Hint: Show that $\overline{K} \leq_m \mathrm{TOT}$. Then conclude that if $\mathrm{TOT} \leq_m K$, then $\overline{K} \leq_m K$, and so $\overline{K}$ would be c.e., Contradiction!]

Here is an example of a proposition making precise the idea that if $A \leq_m B$, then A is at least as easy to compute as B:

Proposition 5.3. *If B is computable and $A \leq_m B$, then A is computable.*

Proof. Assume B is computable. To compute whether $x \in A$, compute $f(x)$, and then compute if $f(x) \in B$. □

The contrapositive of the above proposition gives us a way to show that a set is incomputable.

Corollary 5.4. *If A is incomputable and $A \leq_m B$, then B is incomputable.*

We have already used the above corollary. For example, in using this new terminology, Rice's theorem states that if C is an index set such that $C \neq \emptyset$ and $C \neq \mathbb{N}$, then either $K \leq_m C$ or $K \leq_m \overline{C}$.

In the sense of many-one reducibility, K is the most complicated c.e. set. Moreover, a set is c.e. iff it is many-one reducible to K.

Theorem 5.5. *$A \subseteq \mathbb{N}$ is c.e. iff $A \leq_m K$.*

Proof. $\Rightarrow$: Given a c.e. set A with a computable enumeration $(A_s)_{s \in \mathbb{N}}$, define a computable many-one reduction from A to K as follows. Given x, let $f(x)$ be the program that iteratively computes the enumeration $A_0, A_1, \ldots$ until it find s such that $x \in A_s$, then the program halts. So for every n, $\varphi_{f(x)}(n)$ halts iff $x \in A$. So $x \in A \leftrightarrow f(x) \in K$.

$\Leftarrow$: Suppose f is a many-one reduction from A to K. Then $n \in A \leftrightarrow f(n) \in K \leftrightarrow (\exists s)$"$\varphi_f(n)(f(n)){\downarrow}$ in s steps". Hence A is Σ_1 and so it is c.e. □

Note that using the padding lemma, it is easy to improve the above theorem to show that if $A \subseteq \mathbb{N}$, then A is c.e. iff $A \leq_1 K$.

Exercise 5.6. Let $\varphi_x \downarrow$ denote that the Turing machine x halts on the empty input, and let $\langle \cdot, \cdot \rangle \colon \mathbb{N}^2 \to \mathbb{N}$ denote some computable bijection from $\mathbb{N}^2$ to $\mathbb{N}$. Let $K' = \{x \colon \varphi_x(0) \downarrow\}$, and $K'' = \{\langle x, y \rangle \colon \varphi_x(y) \downarrow\}$. Show that $K \equiv_1 K' \equiv_1 K''$.

So all these versions of the halting problem are equivalent.

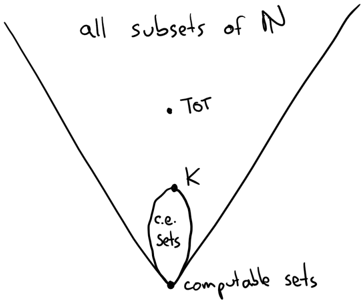


Figure 5: A picture of all subset of $\mathbb{N}$ organized by many-one reducibility

5.2 Myhill's isomorphism theorem

Next, we give a characterization of $\equiv_1$ due to Myhill. Say that $A, B \subseteq \mathbb{N}$ are **computably isomorphic** iff there is a computable bijection $h \colon \mathbb{N} \to \mathbb{N}$ such that for all x, $x \in A \leftrightarrow f(x) \in B$.

Theorem 5.7. *$A \equiv_1 B$ iff A and B are computably isomorphic.*

Recall that Cantor-Shröder-Bernstein theorem: if X and Y are arbitrary sets such that there is an injection from X to Y and an injection from Y to X, then there is a computable bijection between X and Y. This theorem above is essentially a computable version of this theorem, and is proved using a similar argument.

Proof of Theorem 5.7. Let $f \colon \mathbb{N} \to \mathbb{N}$ be the computable injection such that $x \in A \leftrightarrow f(x) \in B$. and let $g \colon \mathbb{N} \to \mathbb{N}$ be the computable injection so the $x \in B \leftrightarrow g(x) \in A$.

We define $h \colon \mathbb{N} \to \mathbb{N}$ in stages (essentially, we will give an enumeration procedure for the graph of h, and appeal to Theorem 4.9). Inductively we will assume that for all x such that $h(x)$ has been defined, $x \in A \leftrightarrow h(x) \in B$, and h is a partial injection. We will ensure that h is total by making sure $\text{dom}(h) = \mathbb{N}$ and $\text{ran}(h) = \mathbb{N}$.

At step $2s$, let x_0 be the least element of $\mathbb{N}$ not already in $\text{dom}(h)$. We define $h(x_0)$ using the following inductive procedure. Given x_i, let $y_i = f(x_i)$, and if $y_i \in \text{ran}(h)$, let $x_{i+1} = h^{-1}(y_i)$. Note that the y_i and x_i are all distinct since f and h^{-1} are partial injections, and $x_i \in A \leftrightarrow y_i \in B \leftrightarrow x_{i+1} \in A$ by our induction hypothesis. Since h is a finite partial function, we must eventually find some n so that $y_n \notin \text{ran}(h)$. Define $h(x_0) = y_n$ for this n. We have $x_0 \in A \leftrightarrow y_n \in B$ by the above, and h will be a partial injection since y_n was not yet in the range of h by definition.

At step $2s + 1$, let y_0 be the least element of $\mathbb{N}$ not already in $\text{ran}(h)$, and define $h^{-1}(y_0)$ by exchanging the roles of A, B, f, g, x_i, y_i and h, h^{-1} in the process above.

At the end of this process h will be total, since there cannot be a least element missing from its domain or range. We have that for every $x \in \mathbb{N}$, $x \in A \leftrightarrow h(x) \in B$ by our construction. Finally, h is computable, since its graph is c.e.; we can enumerate each step of our construction (so apply Theorem 4.9). □

In computability theory, we largely only care about properties of sets of natural numbers that are invariant under computable isomorphism. Similarly, in geometry we only care about properties of shapes that are invariant under isometries.

6 The recursion theorem, acceptable numberings

6.1 The recursion theorem

The recursion theorem is a fixed-point theorem for computable functions acting on programs:

Theorem 6.1 (Kleene's recursion theorem)**.** *Let $f : \mathbb{N} \to \mathbb{N}$ be a total computable function. Then there is some $e \in \mathbb{N}$ such that $\varphi_e = \varphi_{f(e)}$ (i.e. the two indices e and $f(e)$ for Turing machines define the same partial computable function).*

Proof. Let $d\colon \mathbb{N} \to \mathbb{N}$ be a total computable function defined so that for every n, $d(n)$ is the index for the program so $\varphi_{d(n)}(k)$ first computes $\varphi_n(n)$, and if this halts and outputs $m = \varphi_n(n)$, then the machine then computes $\varphi_m(k)$.[5] So

$$\varphi_{d(n)}(k) = \begin{cases} \varphi_{\varphi_n(n)}(k) & \text{if } \varphi_n(n) \text{ halts} \\ \text{undefined} & \text{otherwise} \end{cases}.$$

Now let v be an index for a Turing machine computing $f \circ d$, which is a total computable function, so $\varphi_v = f \circ d$. Then if we let $e = d(v)$, then

$$\varphi_e = \varphi_{d(v)} = \varphi_{\varphi_v(v)} = \varphi_{f(d(v))} = \varphi_{f(e)}$$

□

We illustrate some simple applications of the recursion theorem.

Proposition 6.2. *There is an e such that φ_e is the constant function e.*

Proof. Let $f : \mathbb{N} \to \mathbb{N}$ be the computable function so $\varphi_{f(e)}(n) = e$ for all e.[6] So by the recursion theorem, there exists an e such that $\varphi_e = \varphi_{f(e)}$. □

Proposition 6.2 is essentially a program that outputs its own source code. Such programs (which are sometimes called "Quines") date back to ideas of von Neumann in the 1940s. A simple example of such a program uses the following basic idea:

> Print the following twice, the second time in quotes:
> "Print the following twice, the second time in quotes:"

Exercise 6.3. Show there is an e so that $W_e = \{e\}$.

Another nice application of the recursion theorem is to minimal programs.

Definition 6.4. An index n for a Turing machine is called **minimal** if for all $m < n$ we have $\varphi_n \neq \varphi_m$. That is, n is the least index implementing the partial computable function φ_n.

Proposition 6.5. *There is no infinite computable set of minimal indices. In particular, the set of all minimal indices is not computable.*

[5] There is a computable function $u(n, m)$ so $u(n, m) = \varphi_n(m)$ (by the existence of a universal Turing machine Theorem 2.11). Let g be the computable function $g(n, k) = u(u(n, n), k) = \varphi_{\varphi_n(n)}(k)$. Note $g(n, k)$ is undefined if $\varphi_n(n)$ does not halt. By the S-m-n theorem, since $g(n, k) = \varphi_x(n, k)$ for some x, there is a computable function s so that $\varphi_{s(x,n)}(k) = g(n, k) = \varphi_{\varphi_n(n)}(k)$. Let d be the computable function where $d(n) = s(x, n)$.

[6] There is a computable function g so that $g(e, n) = e$ for all e, n. So $g(e, n) = \varphi_x(e, n)$ for some x. By the S-m-n theorem, $\varphi_{s(x,e)}(n) = \varphi_x(e, n) = g(e, n) = e$. Let f be the computable function $f(e) = s(x, e)$.

Proof. For a contradiction, suppose M is a infinite computable set of minimal indices. Let $f : \mathbb{N} \to \mathbb{N}$ a the total computable function mapping e to an index $f(e)$ for a Turing machine which on input k, first computes the least element n of M such that $n > e$, and then computes $\varphi_n(k)$. So for every $\varphi_f(e) = \varphi_n$ for some $n > e$ where $n \in M$. (Note that f is total; it is computing a description of a Turing machine, and not actually running the Turing machine itself).

By the recursion theorem there is an e such that $\varphi_e = \varphi_{f(e)} = \varphi_n$ for some $n > e$ where $n \in M$. But this implies that n is not a minimal index. Contradiction! □

Exercise 6.6. Show there is no infinite c.e. set of minimal indices.

Generally, the way we apply the recursion theorem is that it allows us to use the index e for a program φ_e as part of the program itself. Formally, we can justify this by letting f map e to an index of a program $f(e)$ that implements the behavior we want (and whose description can include e), and then by applying the recursion theorem.

We finish this section by reproving that the halting problem is incomputable using the recursion theorem. We'll give a formal and informal version of this proof to illustrate this type of formal vs informal argument.

Proposition 6.7. $K' = \{n \colon \varphi_n(0)\downarrow\}$ *is not computable.*

Proof. Informal proof: Assume for a contradiction that K' is computable. By the recursion theorem, consider the program φ_e so that $\varphi_e(0)$ first computes whether $e \in K'$ (using the computability of K') and then halts if $e \notin K'$, and never halts if $e \in K'$. Then $\varphi_e(0)\downarrow$ iff $e \notin K'$ iff $\varphi_e(0)\uparrow$. Contradiction!

Formal proof: Assume for a contradiction K' is computable. Let $f \colon \mathbb{N} \to \mathbb{N}$ be a total computable function where $\varphi_{f(e)}(0)$ first computes whether $e \in K'$ (using the computability of K') and then halts if $e \notin K'$, and never halts if $e \in K'$. (The existence of such a total computable f follows from the S-m-n theorem). By the recursion theorem, let e be such that $\varphi_e = \varphi_{f(e)}$. Then $\varphi_e(0)\downarrow$ iff $\varphi_{f(e)}(0)\downarrow$ iff $e \notin K'$ iff $\varphi_e(0)\uparrow$. Contradiction! □

There is a version of the recursion theorem that includes parameters. It proof uses the uniformity in the proof of the recursion theorem: we use the same computable procedure for finding a fixed point for any computable function f.

Exercise 6.8. If $f \colon \mathbb{N}^2 \to \mathbb{N}$ is a computable function, then there is a computable injection $n \colon \mathbb{N} \to \mathbb{N}$ such that for all y,

$$\varphi_{n(y)} = \varphi_{f(n(y),y)}$$

Exercise 6.9. Prove that K is not an index set.

6.2 Acceptable numberings*

In Definition 2.10, we fixed some computable enumeration of all Turing machines. However, there are many possible ways we could have done this: enumerating all Turing machines is a different order, using a different notion of computation like general recursive functions, python programs, etc.. In this section, we discuss **acceptable numbers**: orderings of all partial computable functions which would also give us a reasonable computability theory.

Definition 6.10. An **acceptable numbering** is a sequence $(\psi_n)_{n\in\mathbb{N}}$ of partial computable functions so that:

1. The sequence includes every partial computable function: $\bigcup_n \{\varphi_n\} = \bigcup_n \{\psi_n\}$.
2. There are computable functions $f \colon \mathbb{N} \to \mathbb{N}$ and $g \colon \mathbb{N} \to \mathbb{N}$ such that for every x, $\psi_x = \varphi_{f(x)}$, and $\varphi_x = \psi_{g(x)}$.

We call $(\varphi_n)_{n \in \mathbb{N}}$ the standard numbering of all partial computable functions.

One way to think of condition (2) above is that there must be a "compiler" that converts programs of the type ψ_x into standard programs φ_y, and also a "compiler" that converts standard programs φ_x into programs of type ψ_y.

We will show using the recursion theorem that any acceptable numbering actually has the much stronger property of being computably isomorphic to the standard numbering:

Theorem 6.11. *Suppose $(\psi_n)_{n \in \mathbb{N}}$ is an acceptable numbering of partial computable functions. Then there is a computable bijection $h \colon \mathbb{N} \to \mathbb{N}$ such that for all x, $\varphi_x = \psi_{h(x)}$.*

It follows from this theorem that if we picked any other acceptable numbering of the partial computable functions, then all the all the various sets we have defined: the halting problem, the set of total functions, etc. would be computably isomorphic to the standard versions. Hence, identical from the perspective of computability theory.

Proof. If f and g are injective, then the theorem follows easily using the same idea as in the proof of Myhill's isomorphism theorem Theorem 5.7. So it suffices to show that we can find injective f and g satisfying condition (2) in the definition of an acceptable numbering.

For the function f, we can inductively use the padding lemma to inductively define a computable $f' \colon \mathbb{N} \to \mathbb{N}$ so that for all n, $\varphi_{f'(n)} = \varphi_{f(n)}$, and $f'(n) \neq f'(i)$ for any $i < n$. However, to do the same for the function g, we need to prove there is a version of the padding lemma for our acceptable numbering $(\psi_n)_{n \in \mathbb{N}}$.

Precisely, we need to define a computable function $h \colon \mathbb{N}^2 \to \mathbb{N}$ so that for all x, $\varphi_{h(e,x)} = \varphi_e$, and for all $x \neq x'$ $g(h(e,x)) \neq g(h(e,x'))$. We define $h(e,k+1)$ recursively, where $h(e,0) = e$. Let $B_k = \{g(h(e,0)), \ldots, g(h(e,k))\}$. By the recursion theorem, consider the program n where:

$$\varphi_n(z) = \begin{cases} \varphi_e(z) & \text{if } g(n) \notin B_k \\ \text{undefined} & \text{otherwise} \end{cases}$$

Case 1: If $g(n) \notin B_k$, then $\varphi_n = \varphi_e$, and $g(n) \neq g(h(e,0)), \ldots, g(h(e,k))$. So we define $h(e,k+1) = n$.

Case 2: if $g(n) \in B_k$, then φ_n is the program which is undefined on all inputs by the second clause of its definition, and so since $g(n) \in B_k$, $g(n) = g(h(e,i))$ for some $i \leq k$, and so $\varphi_n = \psi_{g(n)} = \psi_{g(h(e,i))} = \varphi_e$. So ψ_e, and all the other partial computable functions in B_k are programs which are undefined on all inputs.

Now m be the program where

$$\varphi_m(z) = \begin{cases} 1 & \text{if } m \in B_k \\ \text{undefined} & \text{if } m \notin B_k \end{cases}.$$

If $m \in B_k$, then $\varphi_m(z) = 1$ for all z by definition, but $m \in B_k$ implies $\varphi_m(z)$ is undefined for all z by the above paragraph. Hence, we must have $m \notin B_k$, and hence $\varphi_m(z)$ is undefined for all z. Let $h(e,k+1) = m$. □

Often the recursion theorem is taken to mean that there is a formal trick that lets us "pretend" that we know the index n of a program φ_n while defining it. However, using Theorem 6.11 we can show that we can literally use a programming language where one of the possible instructions in the language is to return the index of the program that is running. Consider such a programming language. It is easy to show that there are computable functions $f, g \colon \mathbb{N} \to \mathbb{N}$ converting back and form between this language, and the language where we do not include this "self-reference" instruction (simply replace the request for the index of the currently running program with the number giving the index). Hence, this different programming language gives an acceptable numbering of programs.

Exercise 6.12. Suppose $(\psi_n)_{n\in\mathbb{N}}$ is an acceptable numbering of partial computable functions. Show there is an $e \in \mathbb{N}$ so that $\psi_e = \varphi_e$.

Exercise 6.13. Show that there is a computable strictly increasing sequence $e_0 < e_1 < \dots$ such that for every n, $W_{e_n} = \{e_{n+1}\}$ (in the standard numbering of r.e. sets). [Hint: use the padding lemma, and the observation from the previous paragraph. For an alternate proof, see [Mi08]].

Exercise 6.14 (A computable listing of all partial computable functions that isn't an acceptable numbering)**.** Let $f_0, f_1\colon \mathbb{N} \to \mathbb{N}$ be computable functions so that $x \mapsto (f_0(x), f_1(x))$ is a computable bijection. Define ψ_n as follows.

$$\psi_n(m) = \begin{cases} \varphi_{f_1(n)}(m) & \text{if } m > 0 \\ \text{undefined} & \text{if } m = 0 \text{ and } f_0(n) = 0 \\ f_0(n) - 1 & \text{if } m = 0 \text{ and } f_0(n) > 0 \end{cases}$$

1. Show that $\bigcup_n \{\psi_n\} = \bigcup_n \{\varphi_n\}$.
2. Show that the function $(n, m) \mapsto \psi_n(m)$ is partial computable.
3. Prove that $(\psi_n)_{n\in\mathbb{N}}$ is *not* an acceptable numbering.

7 Undecidability and tiling problems

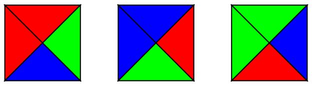
Definition 7.1. A yes/no problem is said to be **undecidable** if there is no computer program that correctly computes answers to each instance of the problem.

Famous examples of undecidable problems include deciding whether a sentence of number theory true, deciding whether a diophantine equation have a solution, and the word problem for finitely presented groups.

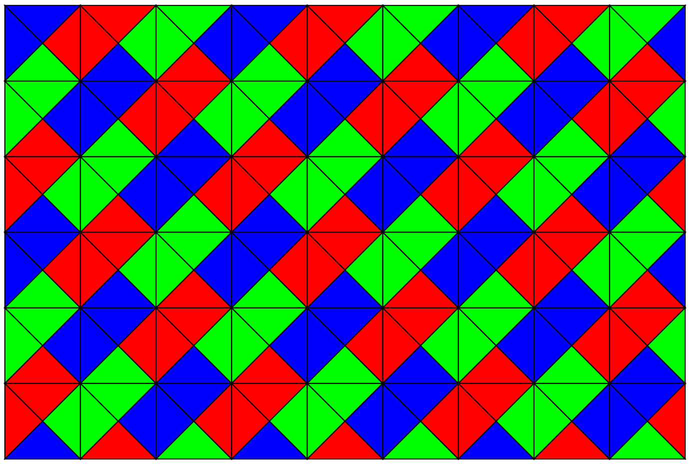
Undecidability is found throughout mathematics and draws a diving line through most subjects: those classes of problems which can be completely solved, and those we will never be able to completely understanding. An excellent introduction to undecidable problems in mathematics is Poonen's paper [P].

7.1 Wang tiles

A Wang tile is a square tile whose left, right, top, and bottom edges have each been given by some label/color. For example, here are three Wang tiles, where we use the colors red, blue and green for the labels:



A **tiling of the plane using a finite set T of Wang tiles** is a function $f\colon \mathbb{Z}^2 \to T$ so that for all $(n,m) \in \mathbb{Z}^2$, the top label of the tile $f(n,m)$ matches the bottom label of the tile $f(n,m+1)$, and the right label of the tile $f(n,m)$ matches the left label of the tile $f(n+1,m)$. Intuitively, we are arranging copies of the tiles from T in an infinite grid where the edges of adjacent tiles match each other. In such a tiling we may repeat any tile as many times as we like, however, each tile may only be translated horizontally and vertically (and not reflected or rotated). Here is a picture of a tiling using the above three tiles:



A famous theorem of Berger says that determining whether a finite set of Wang tiles can tile the infinite plane is incomputable.

Theorem 7.2 (Berger, 1966)**.** *There is no computer program which takes as input a finite set T of Wang tiles and outputs whether there is a tiling of the plane using the tiles T.*

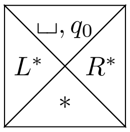
We will prove a simpler special case of Berger's theorem: it is undecidable whether there is a tiling of the plane with a finite set T of Wang tiles that includes a particular Wang tile $t_0 \in T$.

Theorem 7.3 (Undecidability of the completion problem for Wang tiles)**.** *The problem of whether a finite set T of Wang tiles and has a tiling of the infinite plane that uses a given $t_0 \in T$ is undecidable.*

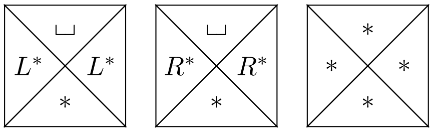
Proof. Fix a Turing machine M with alphabet S states Q, starting state q_0, and transition function $t \colon S \times Q \to S \times Q \times \{L, R\}$. We will define a set of tiles $T(M)$ and a tile $t_0 \in T(M)$ so that $(T(M), t_0)$ is computable from M. This tileset will have the property that M halts iff there is no tiling of the plane using the tiles $T(M)$ and containing t_0. The labels in our tileset will elements of $S \cup \{L, R\} \cup \{*, L^*, R^*\}$ where $\{*\}$ is a special symbol we introduce to help represent tiles below, to the left, and to the right of the start of the computation.

The finite set $T(M)$ of tiles consists of:

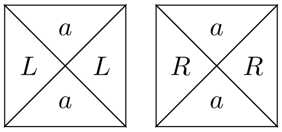
1. The tile t_0:



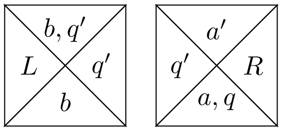
2. The three tiles:



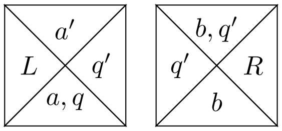
3. For every $a \in S$



4. For every transition of the form $t(a, q) = (a', q', L)$, and every $b \in S$, the tiles



5. For every transition of the form $t(a, q) = (a', q', R)$, and every $b \in S$, a tile:



Let $f \colon \mathbb{Z}^2 \to T(M)$ be a tiling using the tile t_0. By shifting the origin of a tiling $f \colon \mathbb{Z}^2 \to T(M)$ we may assume that $f(0, 0) = t_0$. Since the only tile whose left label is L^* is the first tile from (2), inductively $f(-n, 0)$ must be this tile for every $n > 0$. Similarly, $f(n, 0)$ must be the second tile in (2) for every $n > 0$. Since the bottom label of every tile $f(n, 0)$ for $n \in \mathbb{Z}$ must be $*$, inductively, $f(n, m)$ must be the third tile from (2) for every $m < 0$.

Now inductively, we show that for each $i \geq 0$, there is exactly one n such that the top label of $f(n,i)$ is of the form $S \times Q$. We have already shown this for $i = 0$. For $i > 0$, note there must be at least one such tile because of the rules (4) and (5): if a tile has a label in $S \times Q$, then the tile above and right or above and left of it must have a label in $S \times Q$. Furthermore, because of (3) combined with (4) and (5), any tile to the left of a tile with top label in $S \times Q$ must have the label on the right side equal to L, and any tile to the right of a tile with top label in $S \times Q$ must have its label on the left side equal to R. So there is exactly one such title.

Now for each i, let $T_i(n) = a$ if the top label of $f(n,i)$ is a or (a,q) for some q. Let n_i be the unique n such that the top label of $f(n,i)$ is in $S \times Q$, and let q_i be such that the top label of $f(n_i,i) = (T_i(n), q_i)$. Now we show inductively that (T_i, q_i, n_i) is the state of the Turing machine M at step i when run on the empty input, and hence M never halts. This is by (3), (4), (5). For all m, $T_{i+1}(m) = T_i(m)$ unless $m = n_i$ in which case the symbol $T_{i+1}(m)$ changes in accordance to the Turing machine rules (because of rules (4) and (5)) and the next state and value of n_{i+1} change according to $t(T_i(n_i), q_i)$ similarly.

So we have shown that if there is a tiling using the tileset T and incorporating t_0, then M must not halt when run on the empty input. The proof that if M halts, then there is no tiling is similar. Any such tiling must similarly have the ith row exactly corresponding to the run of the Turing machine, but once the state of the Turing machine becomes the halting state q_halt, since $t(a, q_halt)$ is undefined, there is no tile we can place above this tile. So if the machine halts, then there is no such tiling. □

A compactness argument can be used to show that Wang tiling problems are equivalent to checking that we can tile arbitrarily large squares.

Exercise 7.4.

1. A finite set of Wang tiles can tile the infinite plane iff for every $n, > 0$, this set of Wang tiles can tile an $n \times n$ square.

2. Suppose T is a finite set of Wang tiles, and $t_0 \in T$. Then there is a tiling of the infinite plane which includes the tile t_0 if and only if for every $n > 0$ there is a tiling of the region $[-n, n] \times [-n, n]$ with the tile t_0 at the origin.

By part (2) of above exercise, the set of pairs $\mathsf{TILE}_0 = \{(T, t_0)\colon$ there is a tiling of the the plane using the tiles T include the tile $t_0\}$ is a c.e. set, so $\mathsf{TILE}_0 \leq_m K$. Our proof of Theorem 7.3 above show that $K \leq_m \mathsf{TILE}_0$. So these types of tiling problem are equivalent to the halting problem in a strong sense.

Exercise 7.5. A 1-dimensional Wang tile is a rectangle with a label on just its left and right side. Show that the completion problem for finite sets of 1-dimensional Wang tiles is decidable. [Hint: use the pigeonhole principle to show that if a 1-dimensional tileset can tile an infinite line, it can do so periodically].

8 Consequences of undecidability

Once we know that a problem of mathematics is undecidable, we can sometimes learn something about the problem by considering potential algorithms that attempt to solve it.

8.1 Aperiodic Wang tilings

In Exercise 7.5 you showed that 1-dimensional tiling problems are decidable. Key to the solution is using the pigeonhole principle to show that if a 1-dimensional tileset can tile the infinite plane, it can do so periodically.

Hao Wang conjectured in 1961 that the same phenomenon holds true in 2 dimensions. Say that 2-dimensional Wang tiling is periodic if there are $M, N > 0$, so that for every $x, y \in \mathbb{Z}$ the tile at location (x, y) is equal to the tile at location $(x + iN, y + jM)$ for every $i, j \in \mathbb{Z}$. Hence, the tiling is the same $m \times n$ rectangle repeated over and over again.

Conjecture 8.1 (Wang's conjecture)**.** *If a finite set of Wang tiles can tile the infinite plane, then it has a periodic tiling.*

Wang's conjecture would have implied that the problem of determining whether a finite set of Wang tiles can tile the infinite plane is decidable.

Proposition 8.2. *If Wang's conjecture is true, then there is an algorithm for deciding whether a finite tileset can tile the infinite plane.*

Proof. The algorithm goes as follows. For each m, n, check first whether there is an $n \times m$ rectangle which can periodically tile the plane (so the colors on the left match the colors on the right, and the colors on the top match the colors on the bottom). If so, output that there is a tiling of the plane. Then check if there is no tiling at all of an $m \times n$ rectangle. If there is none, then output there is no tiling of the plane.

This algorithm will always halt assuming Wang's conjecture, since either there is a tiling of the plane (and hence a periodic tiling by Wang's conjecture which we will eventually find), or there is no tiling of the plane (and hence by Exercise 7.4

Exercise 8.3. Consider the following variation on the definition of a Turing machine: instead of a transition function $t \colon S \times Q \to S \times Q \times \{L, R\}$ there is a transition function $t \colon S \times Q \to S \times Q \times \mathbb{Z}$ where if $t(a, q) = (a', q', n)$, then the symbol in the current cell is changed to a', the state is changed to q', and the head of the Turing machine moves by n cells. Explain why this new type of machine can compute exactly the same partial functions as the usual sort of Turing machine.

there is no tiling of some $n \times m$ rectangle, which we will eventually find. □

In 1966, Berger refuted Wang's conjecture in a strong way:

Theorem 8.4 ([B66] (see [DRS] for a modern proof))**.** *There is no algorithm for checking in a finite amount of time whether a given finite tileset can tile the infinite plane.*

Corollary 8.5. *there is a finite set of Wang tiles that can tile the plane, but only aperiodically.*

Proof. If Wang's conjecture is true, it would give an algorithm for deciding what tilesets can tile the infinite plane by Proposition 8.2. Since Berger proved there is no such algorithm, Wang's conjecture is false. □

By tracing through the undecidability result, and implementing a tiling that simulates Turing machine that performs Wang's algorithm, Berger found an explicit example of such a tiling set. It used more than a hundred Wang tiles. In recent years, the smallest possible such tileset (which tiles the plane but only aperiodically) has been found by Jeandel and Rao [JR]. It uses 11 tiles and 4 colors (and no set of Wang tiles with either fewer than 11 tiles or fewer than 4 colors is aperiodic):

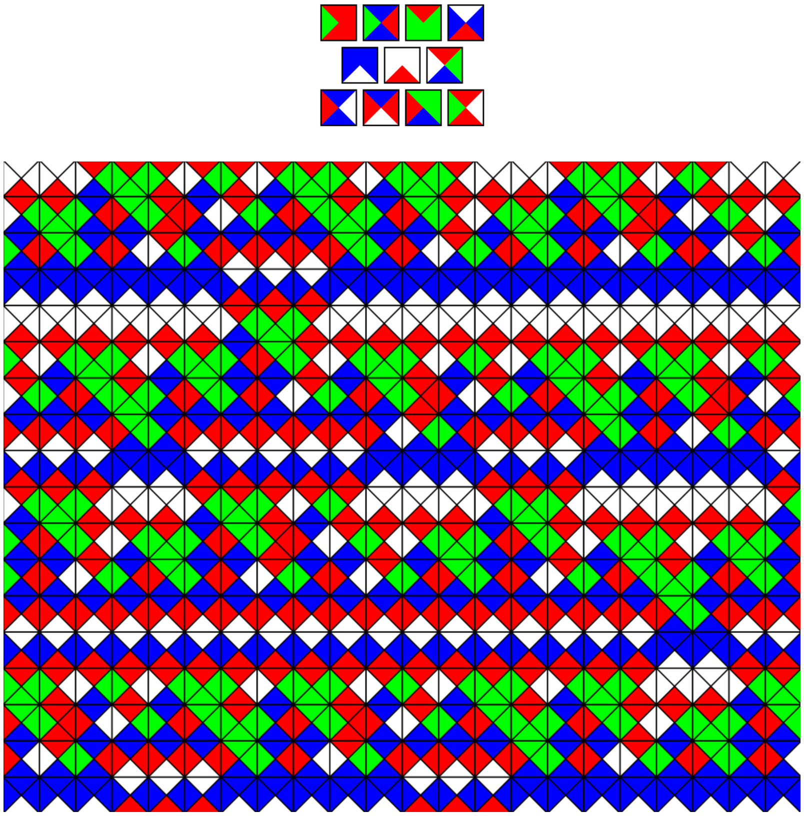


Figure 6: The smallest set of tiles that can tile the plane, but only aperiodically. Taken from the paper [JR]

8.2 Undecidability implies incompleteness

Suppose we have some undecidable problem. One algorithm for attempting to solve the problem is to fix some computable set of axioms (e.g. PA or ZFC) and notion of proof (e.g. proofs in a Hilbert-style proof system) for, and search through all proofs until we find a proof that the answer to the problem is yes, or the answer is no. Since this type of algorithm cannot always work to decide the problem (since we're assuming the problem is undecidable), either the algorithm outputs the wrong answer on some input (hence we can prove a false statement from the axioms), or the algorithm never halts on some input (hence, one of our problems can never be proved to have a yes or no answer).

Gödel's first incompleteness theorem essentially boils down to proving that the problem of deciding whether a sentence of arithmetic is true or false is undecidable, and then appealing to the above observation. Hence, for any computable system of axioms that is true about the natural numbers $\mathbb{N}$ there must be a sentence that is independent from these axioms.

The more general Gödel-Rosser incompleteness theorem removes the assumption that our axioms are true (this is useful for studying nonstandard models, for instance), and just that it is consistent, and extends some basic axiom system (e.g. PA or Robinson's Q).

We will prove these incompleteness theorems carefully in following sections.

8.3 Non-residually finite groups via the undecidability of the word problem*

In this section, we give another example of a an interesting mathematical consequence of an undecidability result. (This example is not historically how the existence of non-residually finite groups was proved, but similar proof techniques have been used to great effect for other types of approximation properties of algebraic structures.)

We will prove that there is a finitely presented non-residually finite group by exploiting the incomputability of the word problem for groups. Recall that a group G is **residually finite** if for all nonidentity $g \in G$, there is a homomorphism $\varphi\colon G \to F$ to a finite group F such that $\varphi(g) \neq 1_F$. Residual finiteness is a kind of finite approximation property for groups. By taking products of homomorphisms to finite groups, one can easily see that a group G is residually finite iff for every finite sequence $g_1, \ldots, g_n \in G$ with $g_i \neq 1_G$ there is a homomorphism $\varphi\colon G \to F$ to a finite group F so that $\varphi(g_i) \neq 1_F$ for all $i \leq n$. So roughly speaking, a group is residually finite iff any finite part of its Cayley table can be replicated inside of a finite group.

Exercise 8.6. Show that for every n, the group $\mathbb{Z}^n$ is residually finite.

Exercise 8.7. For each n, show that the free group on n generators $\mathbb{F}_n$ is residually finite. [Hint: given a word g of length m in $\mathbb{F}_n$, construct a homomorphism from $\mathbb{F}_n$ to the symmetric group S_m on a set of size $m + 1$.

Lemma 8.8 (McKinsey, Dyson, Mostowski)**.** *The word problem for a finitely presented residually finite group is computable.*

Proof. The following algorithm decides if a word w in a finitely presented residually finite group $\langle S|R\rangle$ is equal to the identity. Simultaneously search for both:

- A sequence of relations showing that $w = 1$
- A homomorphism $\varphi\colon \langle S|R\rangle \to F$ to a finite group F where $\varphi(w) \neq 1_F$.

Note that even though the group $\langle S|R\rangle$ may be infinite, a homomorphism from $\langle S|R\rangle$ to a group F is determined by its image on the generators S, and to check that φ is a homomorphism, we just need to check that the images under φ of the relations in R are satisfied in F. Hence, it's easy to enumerate all homomorphisms φ from $\langle S|R\rangle$ to finite groups F.

Eventually one of these two searches must terminate showing that either $w = 1$ or $w \neq 1$. □

It is a famous theorem of Boone and Novikov that the word problem for finitely presented groups is incomputable. We will prove this in the next section. Hence, we have:

Corollary 8.9. *There is a finitely presented non-residually finite group.*

Proof. If every finitely presented group was residually finite, then the word problem for all finitely presented groups would be computable by Lemma 8.8. □

A more typical proof of this corollary would be an example like the following:

Exercise 8.10. Show that the Baumslag-Solitar group $BS(2,3) = \langle a, b | a^{-1}b^2a = b^3\rangle$ is not residually finite.

8.4 Unsolvability of the word problem*

In this section, we prove the undecidability of the word problem for finitely presented groups. The following exercise follows easily from the normal form theorem for HNN extensions, and encapsulates exactly the facts about them which we will need to carry out the proof.

See Chapter IV in [LS] for a detailed introduction to HNN extensions. We encapsulate the basic facts about HNN extensions that we will need in the following exercise:

Exercise 8.11. Let G be a group, $A, B \leq G$ be subgroups of G that are isomorphic via $\phi : A \to B$, and let the HNN extension of G with respect to A, B, and ϕ be $G^* = \langle G, t; t^{-1}at = \phi(a)\rangle_{a \in A}$. Show that:

1. If A is generated by $a_0, a_1, \ldots a_n$, then G^* is equivalently presented as $\langle G, t; t^{-1}a_0t = \phi(a_0), \ldots, t^{-1}a_nt = \phi(a_n)\rangle$.
2. For all $g \in G$, if there exists a $g' \in G$ such that $t^{-1}gt = g'$, then $g \in A$.
3. If H is a subgroup of G such that $\phi(H \cap A) = H \cap B$, then if H^* is the subgroup of G^* generated by H and t, then $H^* \cap G = H$.

It is an important aspect of undecidability proofs that it is often easier to prove a particular undecidability using a particular model of computation. For example, the undecidability of Tilings problems is easiest to prove using the Turing machine model. Whereas the undecidability of solving diophantine equations is easiest to prove using register machines (see [JM91]). Our proof of the undecidability of the word problem will use a variation of the notion of a Turing machine called a modular machine. This will make our undecidability proof much cleaner.

Definition 8.12 (Definition of a modular machine)**.** A **modular machine program** is some $M > 1$, and a finite set called the **transitions** of the machine of the form (a, b, c, R) or (a, b, c, L) where $a, b, c \geq 0$, $a, b < M$ and $c < M^2$. The set of transitions must have the property that there is at most one transition beginning with any particular pair of a and b. A **configuration of a modular machine** is a pair $(\alpha, \beta) \in \mathbb{N}^2$.

Definition 8.13 (The execution of modular machine)**.** Fix a modular machine program. To each configuration (α, β) of the modular machine, we definite an infinite sequence of configurations that . Suppose $\alpha = uM + a$ and $\beta = vM + b$ for $0 \leq a, b, u, v < M$. Then if there is a transition of the form (a, b, c, R), then the next configuration after (α, β) is defined to be $(uM^2 + c, v)$. If there is a transition of the form (a, b, c, L), then the next configuration after (α, β) is defined to be $(u, vM^2 + c)$. Finally, if there is no transition beginning with $(a, b, \ldots)$, then the next configuration is defined to be (α, β).

By using the base M representation of α to represent the left half of a Turing machine tape and base M representation of β to represent the right half of a Turing machine tape, and by storing the current state of the Turing machine in either $a = \text{rem}(\alpha, M)$ or $b = \text{rem}(\beta, M)$, it is straightforward to show the following:

Exercise 8.14. Show that there is a modular machine such that the set of (α, β) that eventually reach $(0, 0)$ is incomputable.

The undecidability of the word problem is originally due to Boone and Novikov, however, the proof of the theorem we will present is due to Aanderaa and Cohen from 1980.

Theorem 8.15 (Boone-Novikov)**.** *There is a finitely presented group whose word problem is incomputable.*

Proof. By Exercise 8.14 Fix a modular machine with modulus M and transitions $\{(a_i, b_i, c_i, R) : i \in I\} \cup \{a_j, b_j, c_j, L) : j \in J\}$ such that there is no transition beginning $(0, 0, \dots)$ and the set of configurations (α, β) that eventually reach $(0, 0)$ is incomputable. We will construct a finitely presented group such that from the word problem for this group, we can determine which configurations (α, β) in this machine eventually reach $(0, 0)$.

We begin with the group $G = \langle x, y, p; xy = yx\rangle$ in which x and y generate a copy of $\mathbb{Z}^2$. For every $(\alpha, \beta) \in \mathbb{Z}^2$, we let $p(\alpha, \beta) = x^\alpha y^\beta p x^{-\alpha} y^{-\beta}$. Intuitively, we will think of $p(\alpha, \beta)$ as representing the configuration (α, β) in our modular machine, when $\alpha, \beta \in \mathbb{N}$. Now of course, the group G does contain any information about the modular machine program or what computations it does. The next step in our proof addresses this issue; we will move to an iterated HNN extension G^* of G in which we add stable letters for each transition in the modular machine program.

Now given any $K, N > 0$ and (a, b), the subgroup $G^{K,L}_{a,b}$ of G generated by x^K, y^L, and $p(a, b)$ is isomorphic to G, via the unique isomorphism ϕ which maps $\phi(x^K) = x$, $\phi(y^L) = y$, and $\phi(p(a, b)) = p$ (the reader should explicitly check this if it is not obvious). Therefore, these subgroups $G^{K,L}_{a,b}$ are all isomorphic to each other.

Our motivation for considering these subgroups and isomorphisms between them is that we can use them to represent transitions from our modular machine program. Indeed, for each $i \in I$ and corresponding transition (a_i, b_i, c_i, R), consider the unique isomorphism $\phi_i : G^{M,M}_{a_i,b_i} \to G^{M^2,1}_{c_i,0}$ for which $\phi_i(x^M) = x^{M^2}$, $\phi_i(y^M) = y$, and $\phi_i(p(a_i, b_i)) = p(c_i, 0)$. Note that this isomorphism correctly implements the transition (a_i, b_i, c_i, R) in our modular machine:

$$\begin{aligned}\phi_i(p(uM + a_i, vM + b_i)) &= \phi_i(x^{-uM} y^{-vM} p(a_i, b_i) x^{uM} y^{vM})\\ &= \phi_i(x^{-uM})\phi_i(y^{-vM})\phi_i(p(a_i, b_i))\phi_i(x^{uM})\phi_i(y^{vM})\\ &= x^{-uM^2} y^{-v} p(c_i, 0) x^{uM^2} y^{v} = p(uM^2 + c_i, v)\end{aligned}$$

Similarly, for each $j \in J$, the unique isomorphism $\psi_j : G^{M,M}_{a_j,b_j} \to G^{1,M^2}_{0,c_j}$ for which $\psi_j(x^M) = x$, $\psi_j(y^M) = y^{M^2}$, and $\psi_j(p(a_i, b_i)) = p(0, c_i)$ carries out the transition (a_j, b_j, c_j, L).

Now we do an iterated HNN extension of G using each pair of subgroups and the associated isomorphism discussed in the previous paragraph. We will use the stable letters r_i for the isomorphisms ϕ_i and l_j for the isomorphisms ψ_j. Then by Exercise 8.11.1, we can equivalently present the resulting group as:

$$\begin{aligned}\langle G, r_i, l_j; r_i^{-1} x^M r_i = x^{M^2}, r_i^{-1} y^M r_i = y, r_i^{-1} p(a_i, b_i) r_i = p(c_i, 0)&\\ l_j^{-1} x^M l_j = x, l_j^{-1} y^M l_j = y^{M^2}, l_j^{-1} p(a_j, b_j) l_j = p(0, c_j)\rangle_{i\in I, j\in J}&\end{aligned}$$

Now this group G^* has a good way of representing configurations of our machine, as well as its transitions. However this group still has a computable word problem; informally, the reason is because from the word problem of G^*, we can only figure out answers to questions such as: "does the following finite sequence of transitions change the configuration (α, β) to the configuration (α', β')"? We will need to take one more HNN extension to get a group with an incomputable word problem. Our goal is now to analyze the set of configurations that eventually halt, and how they appear in this group, this will be the key to finishing.

Let $C_{\text{halt}} = \{p(\alpha, \beta) : (\alpha, \beta) \in \mathbb{N}^2 \wedge\ (\alpha, \beta)$ eventually reaches $(0, 0)\}$; the set of configurations that eventually reach $(0, 0)$. Now let H be the subgroup of G generated by C_{halt}. Note that since $\langle p(\alpha, \beta) : (\alpha, \beta) \in \mathbb{Z}^2\rangle$ generates a free group (the reader should check this), we have that $p(\alpha, \beta) \in H$ if and only if $p(\alpha, \beta) \in C_{\text{halt}}$.

Now let $H^* = \langle H, r_i, l_j\rangle_{i\in I, j\in J}$. We claim that this is equal to the group $\langle p, r_i, l_i\rangle_{i\in I, j\in J}$. Since $p(0, 0) = p$, it is clear that H^* contains this group. To show that $\langle p, r_i, l_i\rangle_{i\in I, j\in J}$ contains H^* it is enough to show that it contains all the generators $p(\alpha, \beta$ of H. We can do this by induction on the number of steps it takes for the configuration (α, β) to reach $(0, 0)$. The base case is clear since $p(0, 0) = p$. Now for our induction step, if $(\alpha, \beta) = (uM + a_i, vM + b_i)$ for some $i \in I$ so that the next transition applied to (α, β) is (a_i, b_i, c_i, R) then since $r_i^{-1} p(\alpha, \beta) r_i = \phi_i(p(uM + a_i, vM + b_i)) = p(c_i, 0)$ by our discussion of ϕ_i above, we are finished. A similar argument works in the case our next transition is of the form (a_j, b_j, c_j, L). Hence, by Exercise 8.11.(3) we have that $p(\alpha, \beta)$ is in the group $\langle p, r_i, l_j \rangle_{i \in I, j \in J}$ if and only if (α, β) eventually reaches $(0, 0)$ in our machine.

We're now ready to finish our proof; there's a beautiful way of taking one more HNN extension so that the word problem of the resulting group can be used to determine what words are elements of the subgroup $\langle p, r_i, l_j \rangle_{i \in I, j \in J}$ of G^*. Let $\theta : \langle p, r_i, l_j \rangle_{i \in I, j \in J} \to \langle p, r_i, l_j \rangle_{i \in I, j \in J}$ be the identity isomorphism, and let G^{**} be the HNN extension of G^* with respect to these subgroups and this isomorphism, so that:

$$G^{**} = \langle G, t; t^{-1} p t = p, t^{-1}, t^{-1} r_i t = r_i, t^{-1} l_j t = l_j \rangle_{i \in I, j \in J}$$

Now by Exercise 8.11.(2) we have that a word w of G^* is an element of $\langle p, r_i, l_j \rangle_{i \in I, j \in J}$ if and only if $t^{-1} w t = w$ is true in G^{**}. Thus we are done; (α, β) eventually reaches $(0, 0)$ if and only if $t^{-1} p(\alpha, \beta) t = p(\alpha, \beta)$ is true in G^{**}. □

Above, we have been concerned with finitely presented groups. Say that a group is **computably presented** if it has a computable set of generators, and a computable set of relations. For example, every finitely presented group is computably presented. It is easy to show that there is a computably presented group with an undecidable word problem.

Exercise 8.16. Show (without using the Boone-Novikov theorem) that there is a computably presented group with an undecidable word problem. [Hint: let S be an infinite set of generators $g_0, g_1, \ldots$. Describe a computable set of relations R so that the nth program φ_n does not correctly compute whether $g_n = 1$.]

By using the same basic tools as the Boone-Novikov theorem above, one can show that there is a finitely presented group so that every computably presented group embeds into it in a computable way. This is Higman's embedding theorem.

Theorem 8.17 (Higman [?H61]). *There is a finitely presented group G so that for every computable group H, there is a computable embedding of H into G.*

Higman's embedding theorem gives a different proof of the undecidability of the word problem for finitely presented groups; simply use Exercise 8.16.

9 Undecidability in arithmetic

9.1 Undecidability of truth in $(\mathbb{N}; 0, 1, +, \cdot, <)$

In this section we'll prove that the set of sentences that are true in the structure $(\mathbb{N}; 0, 1, +, \cdot, <)$ is undecidable. The key to this proof is a way of representing sequences using natural numbers.

Definition 9.1 (Gödel's β function)**.** Define the function $\beta \colon \mathbb{N}^3 \to \mathbb{N}$ by $\beta(x_1, x_2, x_3) = \mathrm{rem}(x_1, 1 + (x_3 + 1) \cdot x_2)$. That is, β is the remainder when x_1 is divided by $1 + (x_3 + 1) \cdot x_2)$.

Note that $\beta(x_1, x_2, x_3) = r$ iff $(\exists q \leq x_1)[q \cdot (1 + (x_3 + 1) \cdot x_2) + r = x_1]$. Hence, since β is a definable function in $(\mathbb{N}; 0, 1, +, \cdot, <)$ we can rewrite any formula containing β with an equivalent formula in this structure that just uses the symbols $0, 1, +, \cdot, <$. Our next lemma says that we can use the β function to represent sequences.

Lemma 9.2 (Gödel's β lemma)**.** *For any sequence of natural numbers $a_0, a_1, \ldots a_n$, there are natural numbers b and c so that for every $i \leq n$, $\beta(b, c, i) = a_i$.*

Proof. By the Chinese remainder theorem, if $m_0, \ldots, m_n$ are relatively prime and $a_0, \ldots, a_n$ are integers, there is some x such that $x \equiv a_i \bmod m_i$ for every $i \leq n$. Choose k such that $k > a_i$ for every i, and $k > n$. Let b be such an x solving this system of equations for the moduli $m_i = 1 + (1 + i)k!$, and $c = k!$. Note that the m_i are all relatively prime for $i \leq n$ since if $p \mid 1 + (1 + i)k!$ and $p \mid 1 + (1 + j)k!$, then p divides the difference $p \mid (i - j)k!$, and since $|i - j| < k$, this implies $p \leq k$. But this contradicts $p \mid 1 + (1 + i)k!$. □

Exercise 9.3. Prove the Chinese remainder theorem: if $m_0, \ldots, m_n \in \mathbb{N}$ are relatively prime then for every integer sequence $a_0, \ldots, a_n$ there is some integer x such that $x \equiv a_i \bmod m_i$. [Hint: first show there are $b_0, \ldots, b_n$ such that $b_i \equiv 1 \bmod m_i$, and $b_i \equiv 0 \bmod m_j$ for $j \neq i$. Then let $x = \sum_{i \leq n} a_i b_i$.]

Now we prove the undecidability of the problem of deciding what sentences are true in the structure $(\mathbb{N}; 0, 1, +, \cdot, <)$. We could directly reduce the problem of whether the nth Turing machine halts to a sentence of arithmetic, but it is a little cleaner to instead use the model of computation given by partial recursive functions.

Lemma 9.4. *For every partial recursive function $f \colon \mathbb{N}^k \to \mathbb{N}$ there is a formula $\varphi_f(x_1, \ldots, x_k, y)$ that represents it in the sense that $\mathbb{N} \vDash \varphi_f(x_1, \ldots, x_k, y)$ if and only if $f(x_1, \ldots, x_k) = y$.*

Proof. We define this map by recursion.

The constant function $f(x) = n$ is represented by the formula $y = 0$ if $n = 0$, and otherwise by the formula $y = \underbrace{1 + 1 + \ldots + 1}_{n \text{ times}}$.

The successor function is represented by the formula $y = x + 1$.

The projection function $(x_1, \ldots, x_k) \mapsto x_i$ is represented by the formula $y = x_i$.

If $h \colon \mathbb{N}^k \to \mathbb{N}$ and $g_1, \ldots, g_k \colon \mathbb{N}^m \to \mathbb{N}$ are partial recursive functions, then their composition $h(g_1(x_1, \ldots, x_m), \ldots, g_k(x_1, \ldots, x_m))$ is maps to the formula

$$(\exists y_1, \ldots, y_k) \left[\left(\bigwedge_{i \in \{1, \ldots, k\}} \varphi_{g_i}(x_1, \ldots, x_m, y_i) \right) \wedge \varphi_h(y_1, \ldots, y_k, y) \right].$$

Next suppose f is defined by recursion where $g \colon \mathbb{N}^k \to \mathbb{N}$ and $h \colon \mathbb{N}^{k+2} \to \mathbb{N}$ are partial recursive, and $f \colon \mathbb{N}^{k+1} \to \mathbb{N}$ is defined by

$$f(z, x_1, \ldots, x_k) = \begin{cases} g(x_1, \ldots, x_k) & \text{if } z = 0 \\ h(z - 1, f(z - 1, x_1, \ldots, x_k), x_1, \ldots, x_k) & \text{if } z > 0. \end{cases}$$

Now note that $f(z, x_1, \dots, x_k) = y$ if and only if there is a sequence $a_0, \dots, a_z$ so that $a_0 = g(x_1, \dots, x_k)$, and for every $i < z$ $h(i, a_i, x_1, \dots, x_k) = a_{i+1}$, and $y = a_z$. So using Gödel's β lemma to represent this sequence $a_0, \dots, a_z$, we can represent f by the formula

$$\begin{aligned}(\exists b, c)\big[&(\exists w < b)[\beta(b, c, 0) = w \wedge \varphi_g(x_1, \dots, x_k, w)]\\ &\wedge (\forall i < z)(\exists w < b)(\exists v < b)[\beta(b, c, i) = w \wedge \beta(b, c, i+1) = v \wedge \varphi_h(i, w, x_1, \dots, x_k, v)]\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\wedge \beta(b, c, z) = y\big]\end{aligned}$$

From Gödel's β lemma we have that this formula holds iff there is such a sequence having the properties above.

Finally, for the minimization operator, suppose $f \colon \mathbb{N}^{k+1} \to \mathbb{N}$ is partial recursive. Consider the partial recursive function $g \colon \mathbb{N}^k \to \mathbb{N}$ where $g(x_1, \dots, x_k)$ which is equal to the least y such that $f(y, x_1, \dots, x_k) = 0$ if such a y exists and for all $y' < y$ $f(y', x_1, \dots, x_k)$ is defined. So this formula holds iff there is a sequence $a_0, \dots, a_y$ such that $a_i = f(i, x_1, \dots, x_k)$ and $a_y = 0$ and $a_{y'} \neq 0$ for all $y' < y$. So using the β lemma, we can define $\varphi_g(x_1, \dots, x_k)$ to be

$$(\exists b, c)[(\forall i \leq y)(\exists z < b)(\varphi_f(i, x_1, \dots, x_k, z) \wedge \beta(b, c, i) = z) \wedge \beta(b, c, y) = 0 \wedge (\forall y' < y)\neg\beta(b, c, y') = 0]$$

□

Corollary 9.5. *The problem of determining what first-order sentences are true in the structure* $(\mathbb{N}; 0, 1, +, \cdot, <)$ *is undecidable.*

Proof. Let $f \colon \mathbb{N} \to \mathbb{N}$ be the partial computable function where $f(n) = \varphi_n(n)$ if $\varphi_n(n)$ is defined, and $f(n)$ is undefined otherwise. Then $\varphi_n(n)\downarrow \leftrightarrow \mathbb{N} \vDash (\exists y)\varphi_f(\overline{n}, y)$. So the function mapping the number n to the formula $(\exists y)\varphi_f(\overline{n}, y)$ is a many-one reduction from the halting problem to the set of true sentences in the structure $(\mathbb{N}; 0, 1, +, \cdot, <)$. Hence $\{\theta \colon \mathbb{N} \vDash \theta\}$ is incomputable. □

From this undecidability of truth in arithmetic, we get a weak version of Gödel's first incompleteness theorem as a corollary.

Corollary 9.6. *Suppose T is a computable first order theory that is true in* $(\mathbb{N}; 0, 1, +, \cdot, <)$. *Then there is some sentence φ that is independent of T (so $T \nvdash \varphi$ and $T \nvdash \neg\varphi$).*

Proof. As discussed in Section 8.2, consider the algorithm that attempts to decide whether sentences φ in $(\mathbb{N}; 0, 1, +, \cdot, <)$ are true or false by searching for a proof that $T \vdash \varphi$ or $T \vdash \neg\varphi$ (here we are using the fact that T is computable to make such an algorithm). By Corollary 9.5, this algorithm cannot correctly halt giving the right answer for each sentence.

The algorithm never outputs an incorrect answer since T is true in $(\mathbb{N}; 0, 1, +, \cdot, <)$ by assumption, and by soundness of first order logic. Thus, the algorithm must fail to halt on some input, and hence there is some sentence φ independent from T. □

10 Peano arithmetic

Our next goal is to prove a stronger form of incompleteness: any consistent computable extension T of Peano Arithmetic is incomplete (even if T is not true in $(\mathbb{N}; 0, 1, +, \cdot, <)$). To do this we first we need to replace much of our work in the previous section about what is true in the structure $(\mathbb{N}; 0, 1, +, \cdot, <)$ with the more technical concept of what is provable in PA.

10.1 Standard and nonstandard models of PA, and initial segments

The theory of Peano arithmetic contains basic algebraic facts about the natural numbers along with axioms for induction. These axioms are in the **language of arithmetic** $\mathcal{L}_A$ which is the first order language whose signature has constants $0, 1$ the relation $<$, and binary functions $+, \cdot$. Below, we will use $x \leq y$ as an abbreviation of the formula $x < y \vee x = y$.

Definition 10.1. Peano Arithmetic or PA is the following theory in the language of arithmetic. First, the theory contains the axioms of the positive part of a discrete ordered semiring. We call these following 16 axioms PA^-:

1. $(\forall x, y, z)((x + y) + z = x + (y + z))$.
2. $(\forall x, y)(x + y = y + x)$.
3. $(\forall x, y, z)((x \cdot y) \cdot z = x \cdot (y \cdot z))$.
4. $(\forall x, y)((x \cdot y) = (y \cdot x))$.
5. $(\forall x, y, z)(x \cdot (y + z) = x \cdot y + x \cdot z)$.
6. $(\forall x)(x + 0 = x)$.
7. $(\forall x)(x \cdot 0 = 0)$.
8. $(\forall x)(x \cdot 1 = x)$.
9. $(\forall x, y, z)(x < y \wedge y < z \rightarrow x < z)$.
10. $(\forall x)(\neg x < x)$.
11. $(\forall x, y, z)(x < y \vee y < x \vee x = y)$.
12. $(\forall x, y, z)(x < y \rightarrow x + z < y + z)$.
13. $(\forall x, y, z)(0 < z \wedge x < y \rightarrow x \cdot z < y \cdot z)$.
14. $(\forall x, y)(x < y \rightarrow (\exists z) x + z = y)$.
15. $0 < 1 \wedge (\forall x)[x > 0 \rightarrow x \geq 1)]$.
16. $(\forall x)(x \geq 0)$.

Second, the theory contains an induction axiom for each formula φ:

$$\varphi(0) \wedge \forall k(\varphi(k) \rightarrow \varphi(k+1))] \rightarrow \forall n \varphi(n) \qquad (\mathrm{Ind}_\varphi)$$

So $\mathsf{PA} = \mathsf{PA}^- + \cup_\varphi \{\mathrm{Ind}_\varphi\}$.

A good reference for Peano Arithmetic is Kaye's book [K91].

Note that induction is an axiom schema which contains countably many axioms; one for each formula. So PA contains infinitely many axioms. We will prove later that there is no finite set of equivalent axioms.

The **standard model** of PA is $(\mathbb{N}; 0, 1, +, \cdot, <)$. However, there are many nonstandard models of PA (i.e. models that are not isomorphic to the standard model). Each element of the standard model has a term in the language of arithmetic that represents it. For each $n \in \mathbb{N}$, we will use the notation $\overline{n}$ to mean the term in the language of arithmetic where $\overline{0}$ denotes the constant symbol 0, and if $n > 0$, then

$$\overline{n} \text{ denotes } \underbrace{1 + 1 + \ldots + 1}_{n \text{ times}}$$

One way to construct nonstandard models of PA is via a compactness argument. Consider the language of arithmetic but where we add a new constant c, and the theory $\mathrm{Th}(\mathbb{N})$ with an additional axiom that $c > \overline{n}$ for each $n \in \mathbb{N}$. Each finite subset of these axioms are consistent, since any finite subset is true in the standard model if we interpret c to be a sufficiently large natural number. However, a model M of the entire theory must have $M \vDash c > \overline{n}$ for each n, and hence M cannot be isomorphic to the standard model. Note that such a nonstandard model can be a model of $\mathrm{Th}(\mathbb{N})$.

A more elaborate version of such a compactness argument shows the following:

Exercise 10.2. There are $2^{\aleph_0}$ non-isomorphic countable models of $\mathrm{Th}(\mathbb{N})$. [Hint: Let p_n denote the nth prime number. Add a new constant symbol c to $\mathcal{L}_A$, and for each subset $A \subseteq \mathbb{N}$, consider the theory $\mathrm{Th}(\mathbb{N}) \cup \{\overline{p_n} \mid c\}_{n\in A} \cup \{\overline{p_n} \nmid c\}_{n\notin A}$.]

What does a nonstandard model of PA look like? To answer this question, we will begin with the following lemma, which will implies that the natural numbers for an initial segment of every model of PA^-.

Lemma 10.3. *For every $k, m, n \in \mathbb{N}$,*

- $n + m = k$ *implies* $\mathsf{PA}^- \vdash \overline{n} + \overline{m} = \overline{k}$.
- $n \cdot m = k$ *implies* $\mathsf{PA}^- \vdash \overline{n} \cdot \overline{m} = \overline{k}$.
- $n < m$ *implies* $\mathsf{PA}^- \vdash \overline{n} < \overline{m}$

Proof. If $n = 0$, $\mathsf{PA}^- \vdash 0 + \overline{m} = \overline{m}$ by axiom 6. Now inductively, suppose $n + m = k$ and $\mathsf{PA}^- \vdash \overline{n} + \overline{m} = \overline{k}$. Then $(n + 1) + m = (k + 1)$, so $\mathsf{PA}^- \vdash \overline{n+1} + \overline{m} = \overline{n} + \overline{m} + 1 = \overline{k} + 1 = \overline{k+1}$ by axiom 1. Hence, the first bullet point is true for each m by induction on n.

Next, if $n = 0$, $\mathsf{PA}^- \vdash \overline{0} \cdot \overline{m} = \overline{0}$ for each m, by Axiom 7, and if $\mathsf{PA}^- \vdash \overline{n} \cdot \overline{m} = \overline{k}$, then $\mathsf{PA}^- \vdash \overline{n+1} \cdot \overline{m} = \overline{n} \cdot \overline{m} + \overline{m} = \overline{nm+m}$ by axiom 5, and the first bullet point. So the second bullet point is also true for each m by induction on n.

Lastly, $\mathsf{PA}^- \vdash 0 < 1$ by axiom 15, and $\mathsf{PA}^- \vdash 0 < \overline{m} \to \mathsf{PA}^- \vdash 0 < \overline{m+1}$ by axiom 9 since $PA^- \vdash \overline{m} < \overline{m+1}$ by axioms 12.

Finally, given any $n < m$, let $k = m - n$. Then $\mathsf{PA}^- \vdash \overline{0} < \overline{k}$ by the above paragraph, and so $\mathsf{PA}^- \vdash \overline{n} < \overline{k+n}$ by axiom 12, and so $\mathsf{PA}^- \vdash \overline{n} < \overline{m}$. □

Recall that if $(X, <_X)$ is a linear order, an **initial segment** of X is a subset that is closed downwards. That is $Y \subseteq X$ is an initial segment of X if for every $a \in Y$ and $b \in X$, if $b < a$, then $b \in Y$.

Corollary 10.4. *If M is any model of PA^-, then $(\mathbb{N}; 0, 1, +, \cdot, <)$ embeds into M as an initial segment.*

Proof. Consider the map $n \mapsto \overline{n}^M$; the mapping from $n \in \mathbb{N}$ to the interpretation of $\overline{n}$ in M. Note that this map preserves the interpretations of the functions $+$ and $\cdot$ by Lemma 10.3, and it preserves the truth of $<$ by the same lemma and also axiom 9.

We claim that the range of the map $n \mapsto \overline{n}$ is an initial segment of M. Suppose $b \in M$ is such that $b \leq \overline{n}$ for some n. Any subset of $\mathbb{N}$ has a least element by induction in $\mathbb{N}$, so we may assume n is least with this property. Now we claim $b = \overline{n}$. If $n = 0$, then $b \leq 0$ and $b \geq 0$ by axiom 16, so $b = 0$ (if $x \geq y$ and $y \geq x$, then $x = y$ by axioms 9 and 10). If $n \neq 0$, then since n is least such that $b \leq \overline{n}$, we must have $\overline{n-1} < b$. Hence $\overline{n-1} + z = b$ for some z by axiom 14. We must have $z > 0$ by axiom 10, and so $z \geq 1$ by axiom 15. But then $b = \overline{n-1} + z \geq \overline{n-1} + 1 = \overline{n}$ by axiom 12. Hence, putting this together with our assumption that $b \leq \overline{n}$, we have $b = \overline{n}$. □

Order-theoretically, it is easy to understand nonstandard models of PA.

Exercise 10.5. Suppose that $(X; 0, 1, +, \cdot, \leq)$ is a countable nonstandard model of PA. Define the equivalence relation $\sim$ on X by $a \sim b$ if there is some n such that $\overline{n} + a = b$ or $\overline{n} + b = a$. Show that $(X/\sim, \leq)$ is a dense linear order having a least element, but no maximum element. Hence, the ordertype of $(X, \leq)$ is $\mathbb{N} + \mathbb{Q} \cdot \mathbb{Z}$.

However, note that we will later prove Tennenbaum's theorem that there is no computable nonstandard model of PA.

We end this section with an interesting model of PA^- that is quite far from being a model of PA.

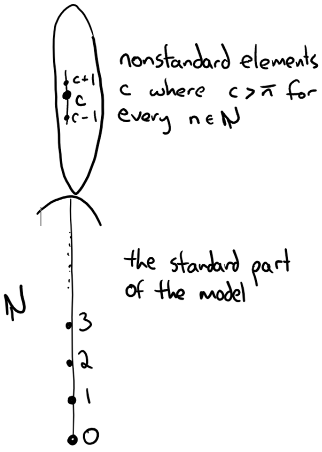

nonstandard elements c where c > $\bar{n}$ for every n ∈ N
c+1
c
c-1
the standard part of the model
N
3
2
1
0

Figure 7: A nonstandard model of PA

Exercise 10.6. Consider the set of all integer polynomials $\mathbb{Z}[X]$ with the linear ordering where $p < q$ iff for sufficiently large $x \in \mathbb{Z}$, $p(x) < q(x)$. Consider the structure $\mathbb{Z}[X]^+$ in $\mathcal{L}_A$ with universe $\{p \in \mathbb{Z}[X] \colon p \geq 0\}$, and the usual operations $+, \cdot$ on polynomials, and the ordering $<$ defined above. Show that $\mathbb{Z}[X]^+ \vDash \mathsf{PA}^-$, but $\mathbb{Z}[X] \nvDash \mathsf{PA}$. [Hint: $\mathsf{PA} \vDash \forall x \exists y (2y = x \vee 2y + 1 = x)$].

11 Gödel-Rosser incompleteness

11.1 Provability of Δ_0 and Σ_1 formulas

We use the abbreviations $(\exists x < y)\varphi(x,y)$ for $(\exists x)[x < y \wedge \varphi(x,y)]$, and $(\forall x < y)\varphi(x,y)$ for $(\forall x)[x < y \rightarrow \varphi(x,y)]$. We call $(\exists x < y)$ and $(\forall x < y)$ **bounded quantifiers** (as opposed to the unbounded quantifiers $(\exists x)$ and $(\forall y)$). To avoid trivialities, in a bounded quantifier we require that y must be a variable not equal to x. Formulas containing only bounded quantifiers (and no unbounded quantifiers) are called Δ_0 formulas, and they have an important absoluteness properties which we will discuss below. Note that $(\exists x < y)$ and $(\forall x < y)$ are dual to each other in the sense that $\neg(\exists x < y)\varphi(x,y)$ is equivalent to $(\forall x < y)\neg\varphi(x,y)$. Similarly, the unbounded quantifiers $(\exists x)$ and $(\forall y)$ are dual to each other.

Definition 11.1. A formula φ in the language of arithmetic is Δ_0 if every quantifier in φ is bounded. A formula φ is Σ_1 if it is of the form

$$\exists x_1,\dots,x_n\psi(x_1,\dots,x_n,y_1,\dots,y_m)$$

where ψ is a Δ_0 formula. A formula φ is Π_1 if it is of the form

$$\forall x_1,\dots,x_n\psi(x_1,\dots,x_n,y_1,\dots,y_m)$$

where ψ is a Δ_0 formula.

For example, $y > 1 \wedge (\forall x_1, x_2)[x_1 \cdot x_2 = y \rightarrow x_1 = 1 \vee x_2 = 1]$ is a Δ_0 formula expressing that y is a prime number.

Clearly if $\mathsf{PA}^- \vdash \varphi$, then $\mathbb{N} \vDash \varphi$, since the axioms of PA^- are true in $\mathbb{N}$. The converse of this statement is false – we've already seen the example of the sentence $\forall x\exists y(x = 2y \vee x = 2y+1)$ which is not provable in PA^-, but is true in $\mathbb{N}$. However, the next two lemmas are special cases where this is true: types of sentences φ with the property that if φ is true in $\mathbb{N}$, then φ is provable from PA^-.

Lemma 11.2. *Suppose $\varphi(x_1,\dots,x_k)$ is a Δ_0 formula. Then for all $n_1,\dots,n_k \in \mathbb{N}$, we have that $\mathbb{N} \vDash \varphi(n_1,\dots,n_k) \leftrightarrow \mathsf{PA}^- \vdash \varphi(\overline{n_1},\dots,\overline{n_k})$.*

Proof. Essentially, this is true since every model of PA^- has $\mathbb{N}$ as an initial segment, and bounded quantifiers in Δ_0 formulas can only quantify over numbers in this initial segment.

By the completeness theorem, we just need to show that if M is a model of PA^-, then

$$\mathbb{N} \vDash \varphi(n_1,\dots,n_k) \leftrightarrow M \vDash \varphi(\overline{n_1},\dots,\overline{n_k}). \qquad (*)$$

(*) is clearly true when φ is quantifier free since $n \mapsto \overline{n}$ is an embedding of $\mathbb{N}$ into M. The class of formulas satisfying (*) is clearly closed under $\wedge$, and $\neg$. Finally, the class of formulas satisfying (*) is closed under bounded quantification since if φ satisfies (*), then $\mathbb{N} \vDash \exists x < m\varphi(n_1,\dots,n_k,x)$ iff there exists $n_{k+1} < m$ such that $\mathbb{N} \vDash \varphi(n_1,\dots,n_k,n_{k+1})$ iff there exists $n_{k+1} < m$ such that $M \vDash \varphi(\overline{n_1},\dots,\overline{n_k},\overline{n_{k+1}})$ iff $\exists x <^M \overline{m} M \vDash \varphi(\overline{n_1},\dots,\overline{n_k},x)$ iff $M \vDash \exists x < \overline{m}\varphi(\overline{n_1},\dots,\overline{n_k})$. The second-to-last equivalence here is true since the embedding $n \mapsto \overline{n}$ is into an initial segment of m, and so $\{x \in M : x <^M \overline{m}\} = \{\overline{n} : n < m\}$. □

Corollary 11.3. *Suppose φ is a Σ_1 sentence. If $\mathbb{N} \vDash \varphi$, then $\mathsf{PA}^- \vdash \varphi$.*

Proof. Suppose $\mathbb{N} \vDash \exists x_1,\dots,x_n\varphi(x_1,\dots,x_n)$ where φ is a Δ_0 formula. Take $n_1,\dots,n_k \in \mathbb{N}$ witnessing this statement, so $\mathbb{N} \vDash \varphi(n_1,\dots,n_k)$. Then $\mathsf{PA}^- \vdash \varphi(\overline{n_1},\dots,\overline{n_k})$ by Lemma 11.2, and so $\mathsf{PA}^- \vdash \exists x_1,\dots,x_k\varphi(x_1,\dots,x_k)$. □

Next, we would like to apply Corollary 11.3 to the formulas φ_f that we used in the proof of Lemma 9.4. Unfortunately, these formulas are not quite Σ_1 formulas, but they are equivalent to them via the following quantifier manipulations:

Lemma 11.4.

- *If $\varphi(\vec{x})$ and $\psi(\vec{y})$ are Σ_1 formulas, then $\varphi(\vec{x}) \wedge \varphi(\vec{y})$ is equivalent to a Σ_1 formula.*
- *If $\varphi(\vec{x})$ and $\psi(\vec{y})$ are Σ_1 formulas, then $\varphi(\vec{x}) \vee \varphi(\vec{y})$ is equivalent to a Σ_1 formula.*
- *Is $\varphi(\vec{x})$ is a Π_1 formula, then $\neg\varphi(\vec{x})$ is equivalent to a Σ_1 formula.*

Proof. If θ and ξ are Δ_0, then:
$\exists\vec{z}\theta(\vec{x},\vec{z}) \wedge \exists\vec{w}\xi(\vec{y},\vec{w}) \leftrightarrow (\exists\vec{z},\vec{w})[\theta(\vec{x},\vec{z}) \wedge \xi(\vec{y},\vec{w})]$.
$\exists\vec{z}\theta(\vec{x},\vec{z}) \vee \exists\vec{w}\xi(\vec{y},\vec{w}) \leftrightarrow (\exists\vec{z},\vec{w})[\theta(\vec{x},\vec{z} \vee \xi(\vec{y},\vec{w})]$.
$\neg\forall\vec{z}\theta(\vec{x},\vec{z}) \leftrightarrow \exists\vec{z}(\neg\theta(\vec{x},\vec{z})$. □

We also note than any Σ_1 formula is equivalent to a formula with a single unbounded existential quantifier, assuming PA^-:

Lemma 11.5. *If $\varphi(\vec{x})$ is a Σ_1 formula, then there is a Σ_1 formula $\varphi'(\vec{x})$ with a single unbounded existential quantifier such that $\mathsf{PA}^- \vdash \varphi(\vec{x}) \leftrightarrow \varphi'(\vec{x})$.*

Proof. Since PA^- proves that $<$ is a linear order with no maximal element, we can quantify to find an upper bound for $y_1,\ldots,y_n$, and then replace the quantifiers over $y_1,\ldots,y_n$ with bounded quantifiers $(\exists y_1 < y)(\exists y_2 < y)\ldots$ So $\mathsf{PA}^- \vdash \exists y_1,\ldots,y_n\theta(\vec{x},\vec{y}) \leftrightarrow (\exists y)(\exists y_1 < y)(\exists y_2 < y)\ldots(\exists y_n < y)\theta(\vec{x},\vec{y})$. □

The equivalences in Lemma 11.4 just use the rules of first order logic and are true in any structure. The next lemma is that Σ_1 formulas are closed under bounded quantification. However, we only state it just in the model $\mathbb{N}$. (The lemma is true more generally in PA, and uses some induction in the proof).

Lemma 11.6. *If $\varphi(\vec{x},y)$ is a Σ_1 formula, then there is a Σ_1 formula $\psi(\vec{x},z)$ so that $\mathbb{N} \vDash (\forall y < z)\varphi(\vec{x},y) \leftrightarrow \psi(\vec{x},z)$.*

Proof. By Lemma 11.5, we may assume that φ has a single existential quantifier.

Let $\varphi(\vec{x},y)$ be the formula $\exists w\theta(\vec{x},y,w)$ where θ is Δ_0. Then $(\forall y < z)(\exists w)\theta(\vec{x},y,w)$ is true iff there exists $w_0,w_1,\ldots,w_{z-1}$ such that for all $y < z$, $\theta(\vec{x},y,w_y)$. Using Gödel's β lemma to encode this sequence, we therefore have

$$\mathbb{N} \vDash (\forall y < z)(\exists w)\theta(\vec{x},y,w) \leftrightarrow (\exists b,c)(\forall y < z)\theta(\vec{x},y,\beta(b,c,y)).$$

Note here that β is Δ_0 definable. □

11.2 The complexity of the set of provable sentences

Combining all the above manipulations of Σ_1 formulas and Corollary 11.3, we have the following:

Theorem 11.7. *For every partial computable function f, there is a Σ_1 formula $\psi_f(x_1,\ldots,x_k,y)$ such that $f(x_1,\ldots,x_k) = y$ iff $\mathbb{N} \vDash \psi_f(x_1,\ldots,x_k,y)$. Furthermore, if $f(x_1,\ldots,x_k = y)$, then $\mathsf{PA}^- \vdash \psi_f(\overline{x_1},\ldots,\overline{x_k},\overline{y})$, and $\mathsf{PA}^- \vdash \forall y'\psi_f(\overline{x_1},\ldots,\overline{x_k},y') \to y' = \overline{y}$.*

The basic idea to prove this theorem is that the formula φ_f from Lemma 9.4 is equivalent to a Σ_1 formula by our quantifier manipulations, and Corollary 11.3 implies that this Σ_1 formula is therefore provable in PA^-. The slight difficult in the proof will be dealing with the second condition $\mathsf{PA}^- \vdash \forall y'\psi_f(\overline{x_1},\ldots,\overline{x_k},y') \to y' = \overline{y}$. We handle this by including in the definition of $\psi_f(x_1,\ldots,x_k,y)$ that no there is no smaller witness that $\varphi_f(x_1,\ldots,x_k,y')$ is true for any $y' \neq y$.

Proof. Consider the formula φ_f from Lemma 9.4, where $f(x_1, \dots, x_k, y) \leftrightarrow \mathbb{N} \vDash \varphi_f(x_1, \dots, x_k, y)$. By applying Lemmas 11.6, 11.5, and 11.4 to the proof of Lemma 9.4, it is clear that $\varphi_f(x_1, \dots, x_k, y)$ is equivalent in $\mathbb{N}$ to a Σ_1 formula $(\exists z)\theta(x_1, \dots, x_k, y, z)$, where θ is Δ_0, and we may also assume $y < z$ using the idea of Lemma 11.5. Define ψ_f to be the formula

$$(\exists z)[y < z \wedge \theta(x_1, \dots, x_k, y, z) \wedge (\forall z' < z)(\forall y' < z)(y \neq y' \to \neg\theta(x_1, \dots, x_k, y', z'))]$$

Suppose $f(x_1, \dots, x_k) = y$. Then $\mathbb{N} \vDash \varphi_f(x_1, \dots, x_k, y)$, so the equivalent Σ_1 statement is true: $\mathbb{N} \vDash \exists z\theta(x_1, \dots, x_k, y, z)$. So by Corollary 11.3, $\mathsf{PA}^- \vdash \exists z\theta(\overline{x_1}, \dots, \overline{x_k}, \overline{y}, z)$.

Furthermore, since $\mathbb{N} \vDash \varphi_f(x_1, \dots, x_k, y)$ iff $f(x_1, \dots, x_k) = y$, we must have that for every $y' \in \mathbb{N}$ with $y' \neq y$, $\mathbb{N} \vDash \neg\exists z\theta(x_1, \dots, x_k, y', z)$. Hence, for any model M of PA^- (which has $\mathbb{N}$ as an initial segment), $M \vDash (\exists z)[y < z \wedge \theta(x_1, \dots, x_k, y, z) \wedge (\forall z' < z)(\forall y' < z)\neg\theta(x_1, \dots, x_k, y', z')]$. This is true since we can take z to be the standard natural number witnessing $\mathbb{N} \vdash \exists z\theta(x_1, \dots, x_k y, z)$, and hence any $y', z' < z$ are also standard numbers so $M \vDash \neg\theta(\overline{x_1}, \dots, \overline{x_k}, \overline{y'}, \overline{z'})$ because $\mathbb{N} \vDash \neg\theta(\overline{x_1}, \dots, \overline{x_k}, \overline{y'}, \overline{z'})$.

Now we claim $\mathbb{N} \vDash \psi_f(x_1, \dots, x_k, y)$ iff $f(x_1, \dots, x_k) = y$. We have already proved the reverse direction, and the forward direction is true since $\mathbb{N} \vDash \psi_f(x_1, \dots, x_k, y)$ implies $\mathbb{N} \vDash \exists z\theta(x_1, \dots, x_k, y, z)$ which is true iff $\mathbb{N} \vDash \varphi_f(x_1, \dots, x_k)$ which is true iff $f(x_1, \dots, x_k) = y$ by Lemma 9.4.

So now assume that $M \vDash \psi_f(\overline{x_1}, \dots, \overline{x_k}, y^*)$ for some $y^* \neq y$. The witness z^* for this existential statement must be a nonstandard number by previous paragraph, since Δ_0 formulas with standard number parameters are true in M iff they are true in $\mathbb{N}$. But then if z^* is nonstandard, then $y < z^*$ and the witness z that $\exists z\theta(x_1, \dots, x_k, y, z)$ is true is also standard and hence less then z^*. Hence, $M \vDash \neg\varphi_f(\overline{x_1}, \dots, \overline{x_k}, y^*)$. Thus, $M \vDash (\forall y')[\psi_(\overline{x_1}, \dots, \overline{x_k}, y') \to y = y']$, and so $\mathsf{PA}^- \vdash (\forall y')[\psi_(\overline{x_1}, \dots, \overline{x_k}, y') \to y = y']$ by the completeness theorem. □

In what follows, we represent formulas in the language or arithmetic by natural numbers in some computable way (analogously to how we have represented finite sets via natural numbers). Typically, we use the notation $\ulcorner\varphi\urcorner$ to denote the number coding the formula φ. This is called the **Gödel number** of φ after Gödel who first gave an explicit way to do this. All we need to assume here is that operations of conjunction , negation, and existential quantification, substitution, etc. are all computable.

From Theorem 11.7, we can show that the set of sentences provable from PA^- are a complete c.e. set.

Theorem 11.8. *$\{\ulcorner\varphi\urcorner\colon \mathsf{PA}^- \vdash \varphi\}$ is a c.e. set and $K \leq_m \{\ulcorner\varphi\urcorner\colon \mathsf{PA}^- \vdash \varphi\}$.*

Proof. We can enumerate all provable sentences by enumerating all proofs from PA^-.

Let $f(n)$ be the partial computable function which computes $\varphi_n(n)$, and outputs 0 if this computation halts. Then $n \in K \leftrightarrow \varphi_n(n)\downarrow \leftrightarrow f(n) = 0 \leftrightarrow \mathbb{N} \vDash \psi_f(n, 0) \leftrightarrow \mathsf{PA}^- \vdash \psi_f(\overline{n}, 0)$. The last three equivalence are true since $f(n) = 0$ implies $\mathsf{PA}^- \vdash \psi_f(n, 0)$ by Theorem 11.7 which implies $\mathbb{N} \vDash \psi_f(n, 0)$ (since $\mathbb{N} \vDash \mathsf{PA}^-$) which implies $f(n) = 0$ by Theorem 11.7. So that map $n \mapsto \ulcorner\psi_f(\ulcorner n\urcorner, 0)\urcorner$ is our desired many-one reduction. □

Similarly, the sentences that are provable from PA^- and whose negations are provable from PA^- are computably inseparable, similarly to how we prove $\{n\colon \varphi_n(n)0\}$ and $\{n\colon \varphi_n(n) = 1\}$ are computably inseparable.

Theorem 11.9. *The two sets $\{\ulcorner\varphi\urcorner\colon \mathsf{PA}^- \vdash \varphi\}$ and $\{\ulcorner\varphi\urcorner\colon \mathsf{PA}^- \vdash \neg\varphi\}$ are computably inseparable c.e. sets.*

Proof. Let $f(n)$ be the partial computable function which computes $\varphi_n(n)$, and outputs 0 if this computation halts and outputs 0, and outputs 1 if this function halts and outputs any other value.

Now suppose C was a computable separating set for $\{\ulcorner\varphi\urcorner\colon \mathsf{PA}^- \vdash \varphi\}$ and $\{\ulcorner\varphi\urcorner\colon \mathsf{PA}^- \vdash \neg\varphi\}$. Consider $D = \{n\colon \ulcorner\psi_f(\overline{n}, 0)\urcorner \in C\}$. D is computable since C is. Now $\varphi_n(n)\downarrow = 0$ implies $\mathsf{PA}^- \vdash$

$\psi_f(\overline{n}, 0)$, and so $n \in D$. Similarly $\varphi_n(n)\downarrow = 1$ implies $f(n) = 1$ and so $\mathsf{PA}^- \vdash \psi_f(n, 1)$ and also $\mathsf{PA}^- \vdash \neg\psi_f(\overline{n}, 0)$ by Theorem 11.7. Hence $\ulcorner \psi_f(\overline{n}, 0) \urcorner$ is in $\{\ulcorner \varphi \urcorner : \mathsf{PA}^- \vdash \neg\varphi\}$ and so $n \notin D$.

But this is a contradiction to Theorem 4.14. □

Corollary 11.10 (Gödel-Rosser incompleteness)**.** *Let T be any computable consistent extension of PA^-. Then T is incomplete.*

Proof. If T is complete, then $\{\ulcorner \varphi \urcorner : T \vdash \varphi\}$ is computable; simply search through every proof from T until we find a proof of φ or $\neg\varphi$; we are guaranteed to find proof or the other eventually. But then $\{\ulcorner \varphi \urcorner : T \vdash \varphi\}$ would be a computable separating set for $\{\ulcorner \varphi \urcorner : \mathsf{PA}^- \vdash \varphi\}$ and $\{\ulcorner \varphi \urcorner : \mathsf{PA}^- \vdash \neg\varphi\}$, contradicting Theorem 11.9. □

11.3 Representability in PA^-

In the following section, we'll give a different proof of incompleteness which uses the following notion of representability in PA^-. It also gives yet another natural way of defining computability.

Definition 11.11. A function $f\colon \mathbb{N}^k \to \mathbb{N}$ is **representable in** PA^- if there is a formula ψ_f so that if $f(x_1, \ldots, x_k) = y$, then $\mathsf{PA}^- \vdash \psi_f(\overline{x_1}, \ldots, \overline{x_k}, \overline{y}) \wedge (\forall y')[\varphi_f(\overline{x_1}, \ldots, \overline{x_k}, y') \to y' = \overline{y}]$.

Theorem 11.12. *A function is computable iff it is representable in* PA^-*.*

Proof. $(\Rightarrow)$: by Theorem 11.7.

$(\Leftarrow)$: if f is representable in PA^-, then there is an algorithm that computes f: search through all y and all proofs from PA^- until we find a y such that $\mathsf{PA}^- \vdash \psi_f(\overline{x_1}, \ldots, \overline{x_k}, \overline{y})$. □

The theory here PA^- does not really matter much to this theorem. We could replace PA^- with a stronger theory like PA or ZFC and we could still prove the same theorem: a function is computable iff it is representable in ZFC. All that matters here is that PA^- is strong enough to prove that $\mathbb{N}$ is an initial segment of any model of PA^-.

Exercise 11.13. Say that a relation R on $\mathbb{N}^k$ is representable in PA^- if there is a formula φ_R such that for all $x_1, \ldots, x_k \in \mathbb{N}^k$, $R(x_1, \ldots, x_k) \to \mathsf{PA}^- \vDash \varphi_R(\overline{x_1}, \ldots, \overline{x_k})$, and $\neg R(x_1, \ldots, x_k) \to \mathsf{PA}^- \vDash \neg\varphi_R(\overline{x_1}, \ldots, \overline{x_k})$. Prove that a relation R is representable in PA^- iff R is computable.

11.4 The search for natural examples of statements independent from PA

Once Gödels incompleteness theorem was proved, mathematicians began searching for "natural" theorems of mathematics that are independent of PA, but don't have a "logical" flavor of self-referential statements or consistency statements. A beautiful example of such a mathematical statement in Ramsey theory that is independent of PA is due to Paris and Harrington. See [PH77].

12 Gödel's original proof of incompleteness

In this section, we'll give a different proof of the incompleteness theorem: Gödel's original proof. This proof will give nice and explicit examples of independent sentences. These independent sentences will be important for our proof of the second incompleteness theorem.

12.1 The first incompleteness theorem

In 1931, there wasn't yet a satisfactory notion of computability. Instead, Gödel instead used primitive recursive functions, and proved a version of Theorem 11.7 for them. Gödel also did not have the unsolvability of the halting problem to reduce to the problem of what sentences can be proved in PA^- to obtain a contradiction. Instead, he used a similar idea as our second proof of the undecidability of the halting problem via the recursion theorem in Proposition 6.7, but in the context of sentences of arithmetic instead of computer programs. This required him to prove a fixed-point theorem analogous to the recursion theorem, but for formulas in arithmetic.

In the following, we will abuse notation by writing natural numbers such as n in formulas of $\mathcal{L}_A$ instead of using our notation $\overline{n}$ for the term representing them. (So for example, the conclusion of Lemma 12.1 should be $\mathsf{PA}^- \vdash \theta \leftrightarrow \eta(\overline{\ulcorner\theta\urcorner})$). We do this to make the theorem and proofs easier to read.

Lemma 12.1 (The fixed point lemma)**.** *Let $\eta(x)$ be a formula in $\mathcal{L}_A$ with one free variable. Then there is a sentence θ such that $\mathsf{PA}^- \vdash \theta \leftrightarrow \eta(\ulcorner\theta\urcorner)$.*

Proof. The proof is essentially the same as the proof of the recursion theorem.

Fix a computable function $d\colon \mathbb{N} \to \mathbb{N}$ where we define $d(z)$ as follows that if $z = \ulcorner\varphi(x)\urcorner$ is the Gödel number of a formula ψ with one free variable, then $d(z) = \ulcorner\varphi(\ulcorner\varphi(x)\urcorner)\urcorner$.

By Theorem 11.7, d is represented by some formula ψ_d, so for each formula $\varphi(x)$,

$$\mathsf{PA}^- \vdash (\forall y)[\psi_d(\ulcorner\varphi(x)\urcorner, y) \leftrightarrow y = \ulcorner\varphi(\ulcorner\varphi(x)\urcorner\urcorner)] \qquad (*)$$

Given the formula $\eta(x)$, let $\varphi(x)$ be the formula $(\exists y)[\psi_d(x, y) \wedge \eta(y)],$. By definition, $\varphi(\ulcorner\varphi(x)\urcorner)$ is just the formula:

$$(\exists y)[\psi_d(\ulcorner\varphi(x)\urcorner, y) \wedge \eta(y)]$$

Since (*) is true, PA^- proves $y = \ulcorner\varphi(\ulcorner\varphi(x)\urcorner)\urcorner$ is the unique witness to the first half of this formula, so

$$\mathsf{PA}^- \vdash \varphi(\ulcorner\varphi(x)\urcorner) \leftrightarrow \eta(\ulcorner\varphi(\ulcorner\varphi(x)\urcorner)\urcorner)$$

Let θ be the sentence $\varphi(\ulcorner\varphi(x)\urcorner)$. So

$$\mathsf{PA}^- \vdash \theta \leftrightarrow \eta(\ulcorner\theta\urcorner)$$

□

Definition 12.2 (The provability predicate)**.** Suppose T is a computable set of axioms. Let $f_T\colon \mathbb{N} to \mathbb{N}$ be the partial computable where $f_T(x)$ is defined iff $x = \ulcorner\varphi\urcorner$ is the Gödel number of some sentence φ, and there is a proof of φ from T. By Theorem 11.7, let ψ_{f_T} be a formula representing the function f_T. Finally, define the formula $\mathrm{Prov}_T(x)$ to be $(\exists y)\psi_{f_T}(x, y)$ which expresses that there exists some proof of the formula with Gödel number x.

Lemma 12.3. *Suppose T is a computable set of axioms extending PA^-. Then*

$$\mathsf{PA}^- \vdash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \leftrightarrow \mathbb{N} \vDash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \leftrightarrow T \vdash \theta$$

Proof. $\mathrm{Prov}_T(x)$ is a Σ_1 formula, and hence by Corollary 11.3 $\mathsf{PA}^- \vdash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \leftrightarrow \mathbb{N} \vDash \mathrm{Prov}_T(\ulcorner\theta\urcorner)$. By the definition of Prov_T and the partial function f_T, $T \vdash \theta$ iff $\mathbb{N} \vDash \mathrm{Prov}_T(\ulcorner\theta\urcorner)$. □

Gödel's original proof of the first incompleteness theorem was via analyzing the following "Gödel sentence".

Definition 12.4 (The "Gödel sentence")**.** Let T be a computable theory extending PA^-. Then by the fixed point lemma (letting $\eta(x)$ be the formula $\neg \operatorname{Prov}_T(x)$), there is a sentence θ so that $\mathsf{PA}^- \vdash \theta \leftrightarrow \neg \operatorname{Prov}_T(\ulcorner\theta\urcorner)$. We call θ the Gödel sentence of T.

We'll show that if T is a computable consistent theory such that $\mathbb{N} \vDash T$ and θ is the Gödel sentence of T, then θ is independent of T. We'll break this proof into two separate lemmas, since proves $T \nvdash \theta$ requires fewer assumptions than showing $T \nvdash \neg\theta$.

Lemma 12.5. *Suppose T is a computable theory extending PA^- and let θ be the Gödel sentence of T. If $T \vdash \theta$, then T is inconsistent.*

Proof. Suppose $T \vdash \theta$. Then $\mathsf{PA}^- \vdash \operatorname{Prov}_T(\ulcorner\theta\urcorner)$ by Lemma 12.3, and hence $T \vdash \operatorname{Prov}_T(\ulcorner\theta\urcorner)$ since T extends PA^-. Now since $\mathsf{PA}^- \vdash \theta \leftrightarrow \neg \operatorname{Prov}_T(\ulcorner\theta\urcorner)$, since we are assuming $T \vdash \theta$, we have $T \vdash \neg \operatorname{Prov}_T(\ulcorner\theta\urcorner)$. Hence T proves both $\operatorname{Prov}_T(\ulcorner\theta\urcorner)$ and its negation, and so T is inconsistent. □

To show that $T \nvdash \neg\theta$, we need to assume more about T than just T is computable and extends PA^-.

Remark 12.6. *Say that a theory T in $\mathcal{L}_A$ is Σ_1-sound if for every Σ_1 sentence θ, if $T \vdash \theta$, then $\mathbb{N} \vDash \theta$.*

Of course, one way for a theory T to be Σ_1 sound is for it to simply be true of $\mathbb{N}$ (e.g. PA is Σ_1 sound). But in general Σ_1 soundness of T is a weaker assumption than assuming $\mathbb{N} \vDash T$.

Lemma 12.7. *Suppose T is a computable theory extending PA^- and let θ be the Gödel sentence of T. If T is Σ_1 sound and $T \vdash \neg\theta$, then T is inconsistent.*

Proof. If $T \vdash \neg\theta$, then $\mathbb{N} \vDash \neg\theta$ since T is Σ_1 sound. Now $T \vdash \operatorname{Prov}_T(\ulcorner\theta\urcorner)$ since $\mathsf{PA}^- \vdash \theta \leftrightarrow \neg \operatorname{Prov}_T(\ulcorner\theta\urcorner)$ by definition of the Gödel sentence. Hence $\mathbb{N} \vDash \operatorname{Prov}_T(\ulcorner\theta\urcorner)$ by Lemma 12.3 and so $T \vdash \theta$. Thus T is inconsistent since it proves θ and $\neg\theta$. □

As a corollary we have Gödel's first incompleteness theorem.

Corollary 12.8. *Suppose T is a computable consistent Σ_1 sound theory extending PA^-. Then if θ is the Gödel sentence of T, then θ is independent of T (i.e. $T \nvdash \theta$ and $T \nvdash \neg\theta$.*

There's a more computational way of thinking of the above proof of Gödel's incompleteness theorem. Consider a program φ_e which (using the recursion theorem) searches for a proof from PA that it does not halt. If it finds such a proof, then the program halts. This cannot happen since if a program halts, then PA proves that it halts (using Theorem 11.7 to represent this statement), and so PA would prove a contradiction. So the program does not halt, but PA cannot prove this. So this statement "φ_e halts" is independent of PA. This program φ_e is analogous to the Gödel sentence θ.

Building on the proof of Theorem **??**, Rosser proved incompleteness for all consistent computable theories extending PA^- using what is called Rosser's trick:

Theorem 12.9. *Let T be any computable consistent theory extending PA^-. Then T is incomplete.*

Proof. Let $\tilde{P}_T(x, y)$ be the relation which holds iff $x = \ulcorner\varphi\urcorner$ is the Gödel number of some sentence φ, and y codes a proof of $\neg\varphi$ from T. Let $\psi_{\tilde{P}_T}$ be a formula representing the relation $\tilde{P}_T$. Finally, define the modified provability predicate $\operatorname{Prov}'_T(x)$ by $(\exists y)[\psi_{P_T}(x, y) \wedge (\forall y' < y)\neg\psi_{\tilde{P}_T}(x, y')]$. Informally, this expresses that there exists some proof of the formula with Gödel number x, and no shorter proof of its negation. The use of this modified provability predicate is known as "Rosser's trick". Note that $\operatorname{Prov}'_T(x)$ implies $\operatorname{Prov}_T(x)$.

By the fixed point Lemma **??**, there is some formula θ such that $\mathsf{PA}^- \vdash \theta \leftrightarrow \neg\operatorname{Prov}'_T(\ulcorner\theta\urcorner)$.

Case 1: $T \vdash \theta$. In this case, $T \vdash \neg\operatorname{Prov}'_T(\ulcorner\theta\urcorner)$. We claim that $\mathbb{N} \vDash \neg\operatorname{Prov}'_T(\ulcorner\theta\urcorner)$. If this were not the case, then $\mathbb{N} \vDash \operatorname{Prov}'_T(\ulcorner\theta\urcorner)$, so $\mathsf{PA}^- \vdash \operatorname{Prov}'_T(\ulcorner\theta\urcorner)$ since the witness y to this statement is standard, and for every $y' < y$ $PA^- \vdash \neg\psi_{\tilde{P}_T}(\ulcorner\theta\urcorner, y')$ by our assumption that T is consistent. So $\mathbb{N} \vDash \neg\operatorname{Prov}'_T(\ulcorner\theta\urcorner)$ is true. But this means there is truly no proof of θ from T, which contradicts our assumption that $T \vdash \theta$.

Case 2: $T \vdash \neg\theta$. So $T \vdash \operatorname{Prov}'_T(\ulcorner\theta\urcorner)$. Let $M \vDash T$, so

$$M \vDash (\exists y)[\psi_{P_T}(\ulcorner\theta\urcorner, y) \wedge (\forall y' < y)\neg\psi_{\tilde{P}_T}(\ulcorner\theta\urcorner, y')]$$

Consider the witness y to this statement. Now y cannot be nonstandard. If it was, our assumption that $T \vdash \neg\theta$ means there is some standard natural number y' coding a proof that $T \vdash \neg\theta$, and so $\mathsf{PA}^- \vdash \psi_{\tilde{P}_T}(\ulcorner\theta\urcorner, y')$, but this standard y' is less than y, contradiction. So y must be standard, but that means that there is a real proof in $\mathbb{N}$ that $T \vdash \theta$. So $T \vdash \theta$ and $T \vdash \neg\theta$ contradicting our assumption that T is consistent.

So neither case 1 nor case 2 can hold, so θ must be independent of T. □

12.2 The truth of the Gödel sentence

Suppose that T is a computable consistent theory extending PA^-. Let θ be the Gödel sentence for T. We've shown that the Gödel sentence θ is independent of T, but is it really true or false in the model $\mathbb{N}$? This question is easy to answer; we must have $\mathbb{N} \vDash \theta$.

Suppose for a contradiction that $\mathbb{N} \vDash \neg\theta$. Then $\mathbb{N} \vDash \operatorname{Prov}_T(\ulcorner\theta\urcorner)$ by the fact that θ is a Gödel sentence. Hence, $T \vdash \theta$. However, we have already proved this is impossible in Lemma 12.5 since this implies that T is inconsistent.

In the above proofs, θ is a Π_1 sentence. We have the following more general lemma:

Proposition 12.10. *If a Π_1 sentence φ is independent of PA^-, then $\mathbb{N} \vDash \varphi$.*

Proof. If $\mathbb{N} \vDash \neg\varphi$, then $\mathsf{PA}^- \vdash \neg\varphi$ by Corollary 11.3 since $\neg\varphi$ is Σ_1. □

13 The second incompleteness theorem

13.1 Gödel's second incompleteness theorem

Let $\mathrm{Con}(T)$ denote the sentence $\neg\,\mathrm{Prov}_T(\ulcorner 0 = 1 \urcorner)$ of $\mathcal{L}_A$ that expresses that there is no proof of a contradiction from T, i.e. T is consistent. Here we have taken $0 = 1$ as an easily written contradiction, but any other false statment would do. Recall that if θ is the Gödel sentence, then $\mathbb{N} \models \neg\theta$ iff $\mathbb{N} \models \mathrm{Prov}_T(\ulcorner \theta \urcorner)$ iff $T \vdash \theta$ by definition of θ and Lemma 12.3. So a different way of stating Lemma 12.5 (which says $T \vdash \theta$ implies T is inconsistent) is that $\mathbb{N} \models \neg\theta$ implies T is inconsistent. That is, "if $\neg\theta$ is true, then $\neg\,\mathrm{Con}(T)$".

Now the proof of Lemma 12.5 can be formalized in PA, and so a careful analysis of the proof shows that if T is a consistent computable extension of PA, then $T \vdash \neg\theta \to \neg\,\mathrm{Con}(T)$, or equivalently $T \vdash \mathrm{Con}(T) \to \theta$. However, we have already shown that $T \nvdash \theta$. Hence $T \nvdash \mathrm{Con}(T)$, since if $T \vdash \mathrm{Con}(T)$, then $T \vdash \theta$ which is a contradiction. This is Gödel's second incompleteness theorem: no computable consistent theory extending PA can prove its own consistency. Formalizing the proof of Lemma 12.5 in PA requires some careful analysis of what PA can prove about the provability predicates Prov_T. The requisite properties used to prove the second incompleteness theorem are as follows:

Definition 13.1. Suppose T is a computable theory in $\mathcal{L}_A$. Then we say that T satisfies the **Hilbert-Bernays provability conditions**[7] if:

1. If $T \vdash \varphi$, then $T \vdash \mathrm{Prov}_T(\ulcorner \varphi \urcorner)$.
2. $T \vdash (\mathrm{Prov}_T(\ulcorner \varphi \to \psi \urcorner) \to (\mathrm{Prov}_T(\ulcorner \varphi \urcorner) \to \mathrm{Prov}_T(\ulcorner \psi \urcorner))$.
3. $T \vdash \mathrm{Prov}_T(\ulcorner \varphi \urcorner) \to \mathrm{Prov}_T(\ulcorner \mathrm{Prov}_T(\ulcorner \varphi \urcorner) \urcorner)$.

We now have the following lemma:

Lemma 13.2. *If T is any computable extension of PA, then Prov_T satisfies the Hilbert-Bernays provability conditions.* □

We do not prove this lemma, but we give some idea of the proof. First, item (1) is trivial. If $T \vdash \varphi$, then the Σ_1 formula $\mathrm{Prov}_T(\ulcorner \varphi \urcorner)$ is true in $\mathbb{N}$ and hence it is provable by Lemma 12.3.

Item (2) requires showing that we can combine a proof that $\varphi \to \psi$ and a proof that φ is true to give a proof of ψ.

Item (3) is the messiest. It follows from the following more general lemma that we state without proof (note that $\mathrm{Prov}_T(\ulcorner \varphi \urcorner)$ is a Σ_1 sentence):

Lemma 13.3. *Suppose T is a computable consistent extension of PA. Then if φ is a Σ_1 formula, then $\mathsf{PA} \vdash \varphi \to \mathrm{Prov}_T(\varphi)$.* □

The basic idea of the proof of Lemma 13.3 is that a number x witnessing some Σ_1 formula $\exists x \psi(x)$ is true can be used be written down and then the Δ_0 property $\psi(x)$ can be finitely verified, and then this consistutes a proof the the formula $\exists x \psi(x)$ is true. Essentially, this amounts to formalizing the proofs of Lemma 11.2 and Corollary 11.3 inside PA.

The proofs of these Lemmas 13.2 and 13.3 use induction and they are not true in just PA^-. We need induction to prove basic facts like the Chinese remainder theorem which is used to code

[7] A more compact notation for $\mathrm{Prov}_T(\ulcorner \varphi \urcorner)$ that is sometimes used is $\Box\varphi$. Using this notation, these three conditions are:

1. If $T \vdash \varphi$, then $T \vdash \Box\varphi$.
2. $T \vdash [\Box(\varphi \to \psi) \wedge \Box\varphi] \to \Box\psi$.
3. $T \vdash \Box\varphi \to \Box\Box\varphi$.

sequences in Gödel's β lemma, and to prove basic facts about how representations of sequences can be concatenated. We need also induction to prove item (3) using structural induction on formulas.

We note here that this property of Σ_1 sentences: that if they are true, then they are provable, is not true for all sentences φ.

Exercise 13.4. Show that there is a sentence φ so that $\mathsf{PA} \nvdash \varphi \to \mathrm{Prov}_{\mathsf{PA}}(\varphi)$.

Gödel merely sketched a proof of the second incompleteness theorem in his 1931 paper, and promised the tedious details of a full proof later. He never gave a detailed proof; everyone was convinced by his sketch. A fully rigorous proof of the second incompleteness theorem was given by Hilbert and Bernays in 1938. Some details were further refined by Löb in 1955 [L55]. The precise conditions on the provability predicate in Definition 13.1 are due to Löb.

Theorem 13.5 (Gödel's second incompleteness theorem)**.** *Let T be any computable consistent theory extending* PA*. Then $T \nvdash \mathrm{Con}(T)$.*

Proof. Let θ be the Gödel sentence so $T \vdash (\theta \leftrightarrow \neg\,\mathrm{Prov}_T(\ulcorner\theta\urcorner))$. Below in our explanations we refer to the Hilbert-Bernays provability conditions.

$T \vdash \theta \to \neg\,\mathrm{Prov}_T(\ulcorner\theta\urcorner)$	since θ is the Gödel sentence
$T \vdash \mathrm{Prov}_T(\ulcorner\theta \to \neg\,\mathrm{Prov}_T(\ulcorner\theta\urcorner)\urcorner)$	by (3)
$T \vdash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \to \mathrm{Prov}_T(\ulcorner\neg\,\mathrm{Prov}_T(\ulcorner\theta\urcorner)\urcorner))$	by (2)
$T \vdash \neg\,\mathrm{Prov}_T(\ulcorner\theta\urcorner) \to (\mathrm{Prov}_T(\ulcorner\theta\urcorner) \to 0 = 1)$	by basic logic
$T \vdash \mathrm{Prov}_T(\ulcorner\neg\,\mathrm{Prov}_T(\ulcorner\theta\urcorner)\urcorner) \to \mathrm{Prov}_T(\ulcorner\mathrm{Prov}_T(\ulcorner\theta\urcorner) \to 0 = 1\urcorner))$	by (1) and (2)
$T \vdash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \to \mathrm{Prov}_T(\ulcorner\mathrm{Prov}_T(\ulcorner\theta\urcorner) \to 0 = 1\urcorner))$	combining the above lines
$T \vdash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \to \big(\mathrm{Prov}_T(\ulcorner\mathrm{Prov}_T(\ulcorner\theta\urcorner)\urcorner) \to \mathrm{Prov}_T(\ulcorner 0 = 1\urcorner))\big)$	by (2)
$T \vdash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \to \mathrm{Prov}_T(\ulcorner\mathrm{Prov}_T(\ulcorner\theta\urcorner)\urcorner)$	by (3)
$T \vdash \mathrm{Prov}_T(\ulcorner\theta\urcorner) \to \mathrm{Prov}_T(\ulcorner 0 = 1\urcorner)$	combining the above
$T \vdash \neg\,\mathrm{Prov}_T(\ulcorner 0 = 1\urcorner) \to \neg\,\mathrm{Prov}_T(\ulcorner\theta\urcorner)$	contraposition
$T \vdash \mathrm{Con}(T) \to \theta$	definition of $\mathrm{Con}(T)$ and since θ is a Gödel sentence

Now to finish, we have that if $T \nvdash \theta$ by Lemma 12.5. Hence, $T \nvdash \mathrm{Con}(T)$. □

If T is a consistent computable extension of PA that is Σ_1-sound (e.g. if $N \vDash T$), then we cannot have $T \vdash \neg\,\mathrm{Con}(T)$. This is because $\neg\,\mathrm{Con}(T)$ is Σ_1, and so by Σ_1-soundness we would have $\mathbb{N} \vDash \neg\,\mathrm{Con}(T)$, and hence T would be inconsistent, contrary to our assumption. So if T is Σ_1-consistent, then $\mathrm{Con}(T)$ is independent of T. This is true, for example, for PA itself: $\mathrm{Con}(\mathsf{PA})$ is independent of PA.

Note, however, that there are computable consistent extensions of T of PA so that $T \vdash \neg\,\mathrm{Con}(T)$. For example, let T be the theory $\mathsf{PA} + \neg\,\mathrm{Con}(\mathsf{PA})$. This theory is consistent, since $\mathrm{Con}(\mathsf{PA})$ is independent from PA. However, for this T, $T \vdash \neg\,\mathrm{Con}(\mathsf{PA})$, which implies $T \vdash \neg\,\mathrm{Con}(T)$, since any proof from the axioms of PA is also a proof from the larger set of axioms T.

13.2 Löb's theorem

If we can prove that there exists a proof of φ, does that imply that φ is true? Löb's theorem shows that whenever we have a sentence φ and we can prove that the existence of a proof of φ implies φ is true, then φ is actually provable.

Theorem 13.6. *Suppose T is a computable consistent theory extending* PA*. If $T \vdash \mathrm{Prov}_T(\ulcorner\varphi\urcorner) \to \varphi$, then $T \vdash \varphi$.*

Proof. Assume $T \vdash \text{Prov}_T(\ulcorner\varphi\urcorner) \to \varphi$. By Gödel's fixed point lemma applied to the formula $\text{Prov}_T(y) \to \varphi$, there is a sentence θ such that $T \vdash \theta \leftrightarrow (\text{Prov}_T(\ulcorner\theta\urcorner) \to \varphi)$. So

$$\begin{array}{ll}
T \vdash \theta \to (\text{Prov}_T(\ulcorner\theta\urcorner) \to \varphi) & \text{by defn of } \theta \\
T \vdash \text{Prov}_T(\ulcorner\theta \to (\text{Prov}_T(\ulcorner\theta\urcorner) \to \varphi)\urcorner)) & \text{by (1)} \\
T \vdash \text{Prov}_T(\ulcorner\theta\urcorner) \to \text{Prov}_T(\ulcorner(\text{Prov}_T(\ulcorner\theta\urcorner) \to \varphi)\urcorner)) & \text{by (2)} \\
T \vdash \text{Prov}_T(\ulcorner\theta\urcorner) \to (\text{Prov}_T(\ulcorner\text{Prov}_T(\ulcorner\theta\urcorner)\urcorner) \to \text{Prov}_T(\ulcorner\varphi\urcorner)) & \text{by (2)} \\
T \vdash \text{Prov}_T(\ulcorner\theta\urcorner) \to \text{Prov}_T(\ulcorner\text{Prov}_T(\ulcorner\theta\urcorner)\urcorner) & \text{by (3)} \\
T \vdash \text{Prov}_T(\ulcorner\theta\urcorner) \to \text{Prov}_T(\ulcorner\varphi\urcorner) & \text{combining the above two lines} \\
T \vdash \text{Prov}_T(\ulcorner\varphi\urcorner) \to \varphi & \text{by assumption} \\
T \vdash \text{Prov}_T(\ulcorner\theta\urcorner) \to \varphi & \text{combining the above two lines} \\
T \vdash \theta & \text{since } T \vdash \theta \leftrightarrow (\text{Prov}_T(\ulcorner\theta\urcorner) \to \varphi) \\
T \vdash \text{Prov}_T(\ulcorner\theta\urcorner) & \text{by (1)} \\
T \vdash \varphi & \text{by combining the above}
\end{array}$$

□

Löb's theorem can be used to easily deduce Gödel's second incompleteness theorem. By Löb's theorem,

$$(T \vdash \text{Prov}_T(\ulcorner 0 = 1\urcorner) \to 0 = 1) \to T \vdash 0 = 1.$$

Taking the contrapositive, $T \nvdash 0 = 1$ implies $T \nvdash 0 \neq 1 \to \neg\,\text{Prov}_T(\ulcorner 0 = 1\urcorner)$. That is, if T is consistent, then $T \nvdash \text{Con}(T)$.

13.3 An analogy

One way to understand Löb's theorem is via the following logical paradox which uses a self-referential sentence. This is similar to how Gödel's theorem can be thought of as a formalized version of the Liar paradox: "this sentence is false".

Let's prove the moon is made of cheese.

1. Let X be the sentence "If X is true, the moon is made of cheese."
2. Suppose X is true.
3. Then the statement of X is true: "If X is true, then the moon is made of cheese."
4. Hence, since we are assuming X is true, it follows that the moon is made of cheese.
5. We have just proved that if X is true, then the moon is made of cheese.
6. But this is X; we have just proved X is true.
7. By repeating our above argument we conclude the moon is made of cheese.

14 Tennenbaum's theorem

14.1 Standard systems of models of PA

Definition 14.1. Suppose $M \vDash \mathsf{PA}$. Then the **standard system** of M is the set of $X \subseteq \mathbb{N}$ so that there exists a formula in $\mathcal{L}_A$ and parameters $\vec{a} \in M$ so that $n \in X \leftrightarrow M \vDash \varphi(n, \vec{a})$.

For example, for the model $\mathbb{N}$, the standard system of $\mathbb{N}$ is called the **arithmetical sets**; the sets which can be defined in the structure $\mathbb{N}$.

There is an alternate way of characterizing the standard system of a nonstandard model. To do this, we first need to prove the overspill principle.

Suppose $M \vDash \mathsf{PA}$. A **cut** I in M is an nonempty initial segment of the model so that if $x \in I$, then $x + 1 \in I$ (i.e. I is closed under the successor operation). We will call a cut I a **proper cut** if $I \neq M$. If $x \in M$, we write $x > I$ if $M \vDash x > y$ for every $y \in I$.

Lemma 14.2 (The overspill principle)**.** *Suppose $M \vDash \mathsf{PA}$ and I is a proper cut in M. Then if $M \vDash \varphi(x)$ for every $x \in I$, then there is some $y > I$ such that $M \vDash \varphi(y)$.*

Proof. Suppose $M \vDash \varphi(x)$ for all $x \in I$, however if $y > I$, then $M \vDash \neg\varphi(y)$. Then $M \vDash \varphi(0) \wedge (\forall x)(\varphi(x) \to \varphi(x + 1))$. However since any $y > I$ has $M \vDash \neg\varphi(y)$, $M \vDash \neg(\forall y)\varphi(y)$. Hence M does not satisfy the induction axiom Ind_φ for the formula φ. □

Now every natural number can be regarded as coding a finite set. One simple way of doing this is the following. Let $p(n)$ be the nth prime number. Then we can regard a number a as coding the set $\{n \colon p(n) \mid a\}$.

Note that since p is a computable function, there is a formula ψ_p representing it, so $\mathbb{N} \vDash \psi_p(n, y)$ iff $p(n) = y$ for each n. We will regard this formula ψ_p as defining the function p in any model of PA.

Even if M is a nonstandard model of PA, we can think of some nonstandard $b \in M$ as similarly coding a set $\{n \colon M \vDash p(\overline{n}) \mid b\}$. We will show that the standard system of a nonstandard model of PA is equal to all the subset of $\mathbb{N}$ that are coded in this way by elements of M.

Lemma 14.3. *Let M be a nonstandard model of PA. Then the standard system of M is equal to the set of $X \subseteq \mathbb{N}$ such that there is some $x \in M$ such that $n \in X \leftrightarrow M \vDash p(\overline{n}) \mid x$. More formally, $n \in X \leftrightarrow M \vDash (\exists y)[\psi_p(\overline{n}, y) \wedge (\exists z) y \cdot z = x]$.*

Proof. Clearly for each $x \in M$, the set of n such that $M \vDash (\exists y)[\psi_p(\overline{n}, y) \wedge (\exists z) y \cdot z = x]$ is in the standard system of M.

Conversely, suppose $\varphi(n, \vec{y})$ is any formula of $\mathcal{L}_A$, and $\vec{a}$ is a tuple of element of M. Consider the formula with free variable x:

$$(\exists b)(\forall n \leq x)[\varphi(n, \vec{a}) \leftrightarrow p(n) \mid b] \tag{*}$$

I claim that the formula holds in M for every natural number x. Fix $x \in \mathbb{N}$. Then if $A = \{n \leq x \colon M \vDash \varphi(\overline{n}, \vec{a})\}$, then since A is coded as a finite set by some natural number, there is a b such that

- $\mathbb{N} \vDash p(n) \mid b$ and thus $M \vDash p(\overline{n}) \mid \overline{b}$ for every $n \leq x$ such that $n \in A$.
- $\mathbb{N} \vDash p(n) \nmid b$ and thus $M \vDash p(\overline{n}) \nmid \overline{b}$ for every $n \leq x$ such that $n \notin A$.

Thus, by overspill in M for the formula defined in (*) and for the cut of standard natural numbers, there is some $b \in M$ so that $M \vDash p(\overline{n}) \mid b$ iff $M \vDash \varphi(n, \vec{a})$. □

14.2 Computable structures

We say that a signature L is computable if there is a program which computes all the constant symbols, function symbols, relation symbols, and their arities of this signature. For example, this implies that the set of first order formulas using the variable $\{x_n \colon n \in \mathbb{N}\}$ is computable. So for example, we can discuss whether a particular set of formulas in this signature is computable.

If L is a computable signature, we say that an L-structure S is **computable** if the universe of S is computable, and

1. There is a program which given any constant symbol c returns the value of c^S.
2. There is a program which takes as input a relation symbol R and a tuple $\vec{a} \in S$, and returns whether $R^S(\vec{a})$ provided $\vec{a}$ has the same length as the arity of S.
3. There is a program which takes as input a function symbol f and a tuple $\vec{a} \in S$, and returns the value $f^S(\vec{a})$ provided $\vec{a}$ has the same length as the arity of R.

Note that any computable set is countable. Hence, the universe of any computable structure is a countable set. So for example $(\mathbb{R}; 0, 1, +, \cdot, <)$ is not a computable structure.

Exercise 14.4. An L-structure is computable iff its atomic diagram (the set of true atomic sentences) is computable.

So for example, the signature of $\mathcal{L}_A$ is computable, and then the standard model $(\mathbb{N}; 0, 1, +, \cdot, <)$ is a computable structure since the universe $\mathbb{N}$ is computable, and function $+, \cdot$ and the relation $<$ are computable.

Similarly, the structure $\mathbb{Z}[X]^+$ discussed in Exercise 10.6 is a computable structures whose universe is all integer polynomials that are eventually nonnegative.

The following exercise consists of verifying the standard Henkin construction in model theory is computable.

Exercise 14.5. Suppose L is a computable signature, and T is a complete computable first order theory in L. Then there is a computable L-structure S so that $S \vDash T$.

14.3 Tennenbaum's theorem

We are ready to show that there is no computable nonstandard model of PA.

Lemma 14.6. *Suppose M is a model of PA. Then there is an incomputable set in the standard system of M.*

Proof. Let $f \colon \mathbb{N} \to \mathbb{N}$ be the partial computable function where $f(n)$ runs the program $\varphi_n(n)$, and then outputs the value of $\varphi_n(n)$. Let ψ_f be the formula from Theorem **??** representing f. Then consider the set

$$C = \{n \colon M \vDash \psi_f(\overline{n}, 0)$$

This set C contains $\{n \colon \varphi_n(n)\downarrow = 0\}$, since any Σ_1 formula true in $\mathbb{N}$ is true in any model of PA^-. The set C is disjoint from $\{n \colon \varphi_n(n)\downarrow = 1\}$, since if $\varphi_n(n)\downarrow = 1$, then $\mathbb{N} \vDash \psi_f(\overline{n}, 1)$, so $M \vDash \psi_f(\overline{n}, 1)$, and finally $\mathsf{PA}^- \vdash \psi_f(\overline{n}, 1) \to (\forall y')(\psi_f(\overline{n}, y') \to y' = 1)$.

Hence, C separates the two sets $A = \{n \colon \varphi_n(n)\downarrow = 1\}$ and $B = \{n \colon \varphi_n(n)\downarrow = 1\}$ and hence C is incomputable by Theorem 4.14. □

Now we use the fact that if M is nonstandard, then any set in its standard system is coded by some element. From this we can deduce that there is no computable nonstandard model of PA.

Theorem 14.7 (Tennenbaum)**.** *There is no computable nonstandard model of PA.*

Proof. Suppose M was a computable nonstandard model of PA. By Lemma **??** there is some incomputable set C in the standard system of M, and by Lemma 14.3 there is some $b \in M$ so that $n \in C \leftrightarrow M \vDash p(\overline{n}) \mid b$.

We claim that we can compute whether $p(\overline{n}) \mid b$. Fix 1^M. By computing

$$\underbrace{1^M +^M 1^M + \ldots +^M 1^M}_{p(n) \text{ times}}$$

we can compute $p(\overline{n})^M$. Now $\mathsf{PA} \vdash (\forall x, p)(\exists q, r)(x = p \cdot q + r \wedge r < p)$. Hence, given x and $p(\overline{n})^M$, we can search for $q \in M$ and $r \in M$ with $r < p(\overline{n})$ such that $x = p(\overline{n})^M q + r$. We must eventually find such q and r. Then we have $p(n) \mid x$ iff $r = 0$. □

We note here that Tennenbaum's theorem implies a version of the first incompleteness theorem. Suppose T is a computable consistent theory extending PA. Add a new constant symbol c to our language and consider the theory $T \cup \{c > \overline{n}\}_{n \in \mathbb{N}}$. This theory is consistent by the compactness. It is not difficult to adapt the construction of Exercise 14.5 to show that there must be a computable model of this theory. This would be a nonstandard model of PA contradicting Tennenbaum's theorem.

Exercise 14.8.

- There is no nonstandard model of PA whose universe is a computable set and where the addition operation $+$ is computable.
- There is no nonstandard model of PA whose universe is a computable set and where the multiplication operation $\cdot$ is computable.

Exercise 14.9. Show there is no computable model of ZFC.

15 Relative computability

15.1 Computability relative to an Oracle

Given some incomputable $A \subseteq \mathbb{N}$, what partial computable functions can be computed if we are allowed to access the information in A as part of our algorithm? This kind of **relative computability** was first introduced by Turing. Informally, a partial computable function $f : \mathbb{N} \to \mathbb{N}$ is **computable relative to** A if there is an algorithm for computing $f(n)$ where the algorithm may ask (and will receive answers to) as many question of the form "is $m \in A$?" as it would like during its execution.

Formally, to define the functions which are partial computable relative to A, we can use the notion of an **oracle Turing machine**. An oracle Turing machine has an extra read-only tape on which the bits of A (i.e. where the nth bit is 0 if $n \notin A$, and 1 if $n \in A$) are written, as well as its usual working tape where the input is written, and where the output will be written. The two tapes may each move independently of each other, depending on the current state, and the bit written on each cell of each of the tape.

So for example, both the characteristic function of A and the characteristic function of $\mathbb{N} \setminus A$ are partial computable relative to A. If f is partial computable, then f is partial computable relative to A for any $A \subseteq \mathbb{N}$ (just compute f as normal and never use the ability to ask questions about A). If $A \subseteq \mathbb{N}$ is computable, then every partial computable function relative to A is partial computable. (Just replace the algorithm that asks question of the form "is $m \in A$" with the algorithm which computes the answer to this question at this step using the computability of A).

A key fact about oracle computability that we will often use is that if some computation $\varphi_n^A(m)$ halts relative to some oracle A. Since the computation only takes finitely many steps, only finitely many bits of the oracle can be queried. So there must be some k so that $\varphi_n^{A \cap \{0,\ldots,k\}}(m)$ halts and gives the same answer. This fact is sometimes called the **use principle**.

Just as with ordinary computability, relative computability has many equivalent definitions.

Exercise 15.1. The partial recursive functions relative to $A \subseteq \mathbb{N}$ are equal to the smallest collection of functions that

1. Contain the characteristic function χ_A.

2. Contain the constant, successor, and projection functions.

3. Are closed under composition, primitive recursion, and minimization.

Show that the partial recursive functions relative to $A \subseteq \mathbb{N}$ are equal to the partial computable functions relative to A.

Just as we did in Section 2, we fix notation for the partial computable functions relative to A.

Definition 15.2. Fix a computable listing of all oracle Turing machine programs. If $A \subseteq \mathbb{N}$, let φ_n^A denote the partial function given by the nth oracle Turing machine oracle using A as the oracle.

For relative computability we still have versions of the padding lemma, the S-m-n theorem, existence of a universal Turing machine, and the recursion theorem for oracle Turing machines. Their proofs are essentially identical to the non-oracle versions:

Exercise 15.3.

- (The S-m-n theorem) There is a injective computable function $s : \mathbb{N}^2 \to \mathbb{N}$ so that for all x, y, z and all $A \subseteq \mathbb{N}$,
$$\varphi^A_{s(x,y)}(z) = \varphi^A_x(y, z).$$

- (The padding lemma) There is a computable injective function $f : N^2 \to N$ so that for every n and every i, $\varphi_n^A = \varphi^A_{f(n,i)}$.

- (The recursion theorem) Let $f : \mathbb{N} \to \mathbb{N}$ be a computable function. Then there is some $e \in \mathbb{N}$ such that $\varphi_e^A = \varphi_{f(e)}^A$ (i.e. the two indices e and $f(e)$ for Turing machines define the same partial computable function).

In general, it is a very common (but not universal) phenomenon that any theorem about partial computable functions **relativizes** to give an analogous theorem for all partial computable functions relative to any oracle, just as in Exercise 15.3 above. When a proof relativizes, typically the exact same argument works where we just replace each φ_n with φ_n^A.

15.2 Turing reducibility and the Turing jump

Relative computability gives us another way to compare the incomputability of subsets of $\mathbb{N}$.

Definition 15.4 (Turing reducibility)**.** If $A, B \subseteq \mathbb{N}$, we say that A is **Turing reducible** to B and write $A \leq_T B$ if the characteristic function of A is computable relative to the oracle B.

Turing reducibility is a coarser reducibility than many-one equivalence.

Proposition 15.5. *If $A \leq_m B$, then $A \leq_T B$.*

Proof. Suppose $f\colon \mathbb{N} \to \mathbb{N}$ is a computable function so that for all $x \in \mathbb{N}$, $x \in A \leftrightarrow f(x) \in B$. Then to compute $\chi_A(x)$ using B as an oracle, first compute $f(x)$, and then output 1 if $f(x) \in B$, and 0 if $f(x) \notin B$. □

Note that the converse is false. For example, if K is the halting problem, then we have shown $\overline{K} \not\leq_m K$, but clearly $\overline{K} \leq_T K$. Indeed, for every $A \subseteq \mathbb{N}$ we have $\overline{A} \leq_T A$. So Turing reducibility is a coarser reducibility.

We have the following basic properties of Turing reducibility:

Exercise 15.6. Turing reducibility $\leq_T$ is a reflexive and transitive.

By the above exercise, the symmetrization of $\leq_T$ is an equivalence relation.

Definition 15.7. Turing equivalence, denoted $\equiv_T$, is the equivalence relation where $A \equiv_T B$ if $A \leq_T B$ and $B \leq_T A$.

The equivalence classes of $\equiv_T$ are called the **Turing degrees**, $\mathcal{D}_T$, and $\leq_T$ forms a partial order on $\mathcal{D}_T$. (Note that $\leq_T$ is not a partial order on subsets of $\mathbb{N}$ since it is not antisymmetric).

What is the structure of the partial order of Turing degrees? Is it linear? Is it dense? How long can chains and antichains be? What countable partial orders embed into it? What is its automorphism group? These types of questions became a central focus of computability theory in the second half of the twentieth century.

Lets begin listing some basic facts about this structure. First, the Turing degree of $\emptyset$ is the smallest Turing degree, and it contains exactly the computable sets.

Exercise 15.8. For all $A \subseteq \mathbb{N}$, $A \equiv_T \emptyset$ if and only if A is computable.

Next, observe that there is no largest Turing degree.

Definition 15.9. If $X \subseteq \mathbb{N}$, the **Turing jump of** X is defined to be $X' = \{n\colon \varphi_n^X(n)\downarrow\}$. That is, the set of n so that $\varphi_n^X(n)$ halts.

Theorem 15.10. *For all $X \subseteq \mathbb{N}$, $X <_T X'$.*

Proof. First we show that from X' we can compute X. That is, $X' \geq_T X$. In fact we have the stronger statement that $X \leq_m X'$. For each n, construct a program $\varphi^X_{f(n)}(m)$ which halts (on any input m) if $n \in X$, and does not halt if $n \notin X$. Then the computable function f witnesses that $X \leq_m X'$ since $n \in X$ iff $\varphi^X_{f(n)}(f(n))\downarrow$ iff $f(n) \in X'$.

The fact that X' is not computable relative to X is a relativized version of the proof that the halting problem K is not computable. Below we copy this old proof word-for-word, replacing φ_n with φ^X_n and occasionally adding the words "relative to X".

To see that $X \not\geq_T X'$, first note that the function

$$f(n) = \begin{cases} \varphi^X_n(n) + 1 & \text{if } \varphi^X_n \text{ halts on input } n \\ 0 & \text{otherwise} \end{cases}$$

is not computable relative to X since If f was computable, then we would have $f = \varphi_m$ for some m. Now φ^X_m must be total, so $f(m) = \varphi^X_m(m) + 1$ by definition of m. But this is a contradiction since $f(m) = \varphi^X_m(m)$ by our assumption $f = \varphi^X_m$.

Now if X' was computable relative to X, we could then also compute the function f from Proposition 3.3 as follows: On input n, first compute whether $n \in X'$. If $n \notin X'$, then output 0. If $n \in X'$, then simulate $\varphi^X_n(n)$ using a universal Turing machine and output $\varphi^X_n(n)+1$ (we know this computation will eventually halt). □

The Turing jump of $\emptyset$ is just the usual halting problem

Exercise 15.11. $\emptyset' \equiv_1 K$.

Exercise 15.12. Show that $A \leq_T B$ iff $A' \leq_m B'$.

The Turing degrees are an **upper semilattice** meaning that any two elements have a least upper bound.

Definition 15.13. If $A, B \subseteq \mathbb{N}$, let $A \oplus B = \{2n\colon n \in A\} \cup \{2n+1\colon n \in B\}$, so $A \oplus B$ codes A using its even bits and B using its odd bits.

Note that clearly $A \leq_T A \oplus B$. We also have that $A \oplus B$ is the least upper bound of A and B.

Proposition 15.14. *For all $A, B \subseteq \mathbb{N}$, if $C \geq_T A$ and $C \geq_T B$, then $C \geq_T A \oplus B$. Hence, $A \oplus B$ is the least upper bound of A and B.*

Proof. If $C \geq_T A$ and $C \geq_T B$, then to compute $A \oplus B$ relative to C, combine these two programs and use the program computing A from C to compute the even bits of $A \oplus B$, and the program computing B from C to compute the odd bits of $A \oplus B$. □

We can similarly define recursive joins of finitely many elements subset of $\mathbb{N}$:

$$A_0 \oplus A_1 \oplus A_2 \oplus \ldots \oplus A_n = (((A_0 \oplus A_1) \oplus A_2) \oplus \ldots \oplus A_n).$$

and it is easy to show that $A_0 \oplus \ldots \oplus A_n$ is the least upper bound of $A_0, \ldots, A_n$.

We can also define a computable join of countably many subsets of $\mathbb{N}$. Let $\langle \cdot, \cdot \rangle\colon \mathbb{N}^2 \to \mathbb{N}$ denote a computable bijection between $\mathbb{N}^2$ and $\mathbb{N}$, and if $A_i \subseteq \mathbb{N}$ for each i, then let

$$\bigoplus_i A_i = \{\langle n, i \rangle\colon n \in A_i\}$$

Clearly $\bigoplus_i A_i \geq_T A_i$ for all the A_i.

Compared to joins of finitely many sets, countable joins are very poorly behaved. For example, countable joins are not a well defined operation on the Turing degrees, and do not give a least upper bound of the sequence $(A_i)_{i \in \mathbb{N}}$.

Exercise 15.15.

1. Show finite joins are well-defined on the Turing degrees. If $A_0 \equiv_T B_0$ and $A_1 \equiv_T B_1$, then $A_0 \oplus A_1 \equiv_T B_0 \oplus B_1$.

2. Show there are sequences $(A_i)_{i\in\mathbb{N}}$ and $(B_i)_{i\in\mathbb{N}}$ so that $A_i \equiv_T B_i$ for every i, but $\bigoplus_i A_i \not\equiv_T \bigoplus_i B_i$.

3. Show there is a sequence $(A_i)_{i\in\mathbb{N}}$ so that $\bigoplus_i A_i$ is not a least upper bound of the collection $(A_i)_{i\in\mathbb{N}}$.

Indeed, we will eventually show that there are countable sequences $(A_i)_{i\in\mathbb{N}}$ with *no* least upper bounds in the Turing degrees Note, however, $\bigoplus_i A_i$ is the least *uniform* upper bound of $(A_i)_{i\in\mathbb{N}}$ in the following sense:

Exercise 15.16. Suppose $(A_i)_{i\in\mathbb{N}}$ is a sequence of subsets of $\mathbb{N}$. Suppose also that $C \geq_T A_i$ **uniformly** in the sense that there is a single program relative to C which on input i, n computes whether $n \in A_i$. Then $C \geq_T \bigoplus_i A_i$.

One consequence of taking countable joins is that we see there cannot be any sequence $(A_i)_{i\in\mathbb{N}}$ that is cofinal in the Turing degrees in the sense that for all $C \subseteq \mathbb{N}$, there is some i so that $A_i \geq_T C$. To see this, take any sequence $(A_i)_{i\in\mathbb{N}}$ and let $C = (\bigoplus_i A_i)'$. Then $C >_T A_i$ for every i.

16 The arithmetical hierarchy

16.1 The arithmetical hierarchy

The arithmetical hierarchy measures the complexity of definable subsets of $\mathbb{N}$ by how complicated they are to define. We measure the complexity of a definition by how many alternations of quantifiers it has. We introduce the arithmetical hierarchy in this section, and prove some relationships between it, Turing reducibility, and the Turing jump.

Definition 16.1 (The arithmetical hierarchy)**.** Suppose $A \subseteq \mathbb{N}^k$ and $X \subseteq \mathbb{N}^m$. Then A is Σ_n^X if there is a relation $R(x,y)$ on $\mathbb{N}^{k+m}$ which is computable relative to X so that

$$x \in A \leftrightarrow \exists y_1 \forall y_2 \exists y_3 \ldots Q y_n R(x, y_1, \ldots, y_n).$$

where Q stands for the quantifier $\exists$ if n is even and $\forall$ if n is odd. A is Π_n^X if there is a relation $S(x, y_1, \ldots, y_n)$ on $\mathbb{N}^{k+n}$ which is computable relative to X so that

$$x \in A \leftrightarrow \forall y_1 \exists y_2 \forall y_3 \ldots Q y_n S(x, y_1, \ldots, y_n).$$

$A \subseteq \mathbb{N}^k$ is Δ_n^X if it is both Σ_n^X and Π_n^X. In this case where $X = \emptyset$, we just write Σ_n, Π_n, or Δ_n.

For example, the set of programs defining defining finite c.e. sets is Σ_2: FIN $= \{e \colon W_e \text{ is finite}\} = \{e \colon \exists k \forall s \geq k \text{“}W_{e,k} = W_{e,s}\text{”}\}$. The set of programs defining total functions is Π_2: TOT $= \{e \colon \varphi_e \text{ is total}\} = \{e \colon \forall n \exists s \text{“}\varphi_e(n) \text{ halts in } s \text{ steps”}\}$. In both cases, the quoted relations on e, n, s are computable.

Note that taking complements converts Σ_n to Π_n sets and vice versa.

Proposition 16.2. *$A \subseteq \mathbb{N}^k$ is Σ_n if and only if its complement is Π_n.*

Proof. If

$$x \in A \leftrightarrow \exists y_1 \forall y_2 \exists y_3 \ldots Q y_n R(x y_1, \ldots, y_n),$$

then the negation of this formula flips all the $\exists$ quantifiers to $\forall$ and vice versa. So

$$x \notin A \leftrightarrow \forall y_1 \exists y_2 \forall y_3 \ldots Q y_n \neg R(x_1, \ldots, x_k, y_1, \ldots, y_n).$$

Finally note that a relation $R(x, y_1, \ldots, y_n)$ is computable iff its negation $\neg R(x, y_1, \ldots, y_n)$ is computable. □

Note that if we work instead in the language of arithmetic, and replace computable relations in the above definition with Δ_0 formula in arithmetic, we would still obtain the same definable sets. For example:

Exercise 16.3. Show that a set $A \subseteq \mathbb{N}$ is Σ_1 iff it is defined by a Σ_1 formula in $\mathcal{L}_A$ in the structure $(\mathbb{N}; 0, 1, +, \cdot, <)$.

We have the following characterization of Σ_1 sets:

Proposition 16.4. *A set $A \subseteq \mathbb{N}$ is c.e. relative to X iff it is Σ_1^X iff $A \leq_m X'$.*

Proof. By relativizing the proof of Proposition 4.2 and Proposition 5.5. □

Our next goal it to prove a generalization of this theorem to all levels of the arithmetical hierarchy. First, we will need to basic closure properties of Σ_n and Π_n sets.

Lemma 16.5 (Closure properties of Σ^X_n)**.** *Suppose $X \subseteq \mathbb{N}$ is some oracle.*

1. *Suppose $A, B \subseteq \mathbb{N}^k$ are Σ^X_n. Then $A \cup B$ and $A \cap B$ are Σ^X_n.*
2. *If $A \subseteq \mathbb{N}^k$ is Σ^X_n, then $\{(x_1, \ldots, x_{k-1}) \colon \exists x_k (x_1, \ldots, x_{k-1}) \in A\}$ is Σ^X_n.*
3. *If $A \subseteq \mathbb{N}^k$ is Σ^X_n, then $\{(x_1, \ldots, x_k) \colon (\forall y < x_k)(x_1, \ldots, x_{k-1}, y) \in A\}$ is Σ^X_n.*
4. *If $X \leq_T Y$ and $A \subseteq \mathbb{N}$ is Σ^X_n, then A is Σ^Y_n.*
5. *If A is Σ^X_1, and X is Σ^Y_n for some $Y \subseteq \mathbb{N}$, then A is Σ^Y_{n+1}.*

Proof. (1) If R, S are relations computable from X, both $R \wedge S$ and $R \vee S$ are computable from X.

(2) The idea here is that we can contract two existential quantifiers $\exists x \exists y$ into a single existential quantifier $\exists z$ by using a computable bijection between $\mathbb{N}^2$ and $\mathbb{N}$. Fix computable functions $\pi, \rho \colon \mathbb{N}^2 \to \mathbb{N}$ so that $n \mapsto (\pi(n), \rho(n))$ is a bijection from $\mathbb{N}^2 \to \mathbb{N}$. Then $\exists x_1 \exists y_1 \forall y_2 \ldots Q y_n R(x_1, \ldots, x_k, y_1, \ldots, y_n)$ is equivalent to $\exists z \forall y_2 \ldots Q y_n R(x_1, \ldots, \pi(z), \rho(z), \ldots, y_n)$. (Note here that $R(x_1, \ldots, \pi(z), \rho(z), \ldots, y_n)$ is a computable relation)

(3) Is proved using the same idea as Lemma 11.6 and induction. We can replace the quantifier $\forall y < x_z \exists z R(y, z, \ldots)$ with the statement $\exists w \forall x < y$ "w codes a sequence $z_0, \ldots, z_{y-1}$ and $R(x, z_x, \ldots)$ holds".

(4) is since $\leq_T$ is transitive. If R is a relation computable from X and $X \leq_T Y$, then R is computable from Y.

(5) To prove (2), suppose $x \in A \leftrightarrow (\exists y) R(x, y)$ where R is computable relative to Y. Fix a computable listing $F_0, F_1, \ldots,$ of all finite subset of $\mathbb{N}$. Let e be the program used to compute R, so $R(x, y)$ iff $\varphi^Y_e(x, y) = 1$. Then

$$x \in A \leftrightarrow (\exists y)(\exists n)(\exists k)(\exists s)\text{``}\varphi^{F_n}_e(x, y) = 1 \text{ halts in } s \text{ steps and only queries oracle bits } < k\text{ ''}$$
$$\wedge (\forall m < k) m \in F_n \leftrightarrow m \in X$$

The reason this is a valid definition of A is the use principle: if the computation $\varphi^X_e(x, y)$ halts, the it only uses finitely many bits of X, so there is some finite set $F_n = X \cap \{0, \ldots, k-1\}$ so that $\varphi^{F_n}_e(x, y)$ halts and gives this same answer and only queries bits of the oracle smaller than k.

Note that since the set of (n, m) such that $m \in F_n \leftrightarrow m \in X$ is Σ^Y_{n+1} (since both Σ^Y_n and Π^Y_n sets are Σ^Y_{n+1}), and so Hence, the above shows that A is a Σ^Y_{n+1} definition, using closure properties (1), (2), and (3) above. □

Our next goal is to relate the arithmetical hierarchy to Turing reducibility and the Turing jump, which we define as follows:

Definition 16.6 (The iterated Turing jump)**.** If $X \subseteq \mathbb{N}$, define $X^{(0)} = X$, and $X^{(n+1)} = (X^{(n)})'$. So $X^{(n)}$ is the nth Turing jump of X.

Theorem 16.7 (Post's hierarchy theorem)**.** *For every $n \geq 1$, and every X, A is Σ^X_n iff A is c.e. relative to $X^{(n-1)}$ iff $A \leq_m X^{(n)}$.*

Proof. We have proven the base case that A is Σ^X_1 iff A is c.e. relative to X iff $A \leq_m X'$ in Proposition 16.4.

Also, by relativizing Proposition 16.4 to the oracle $X^{(n-1)}$, we have that A is c.e. relative to $X^{(n-1)}$ iff $A \leq_m X^{(n)}$.

Now inductively assume the theorem is true for n, suppose A is Σ^X_{n+1}. Then $x \in A \leftrightarrow \exists y_1 \forall y_2 \ldots Q y_{n+1} R(x, y_1, \ldots, y_n)$ for some relation R computable relative to X. So $x \in A \leftrightarrow \exists y_1 S(x, y_1)$ where $S(x, y_1)$ holds iff $\forall y_2 \ldots Q y_n R(x, y_1, \ldots, y_{n+1})$. Note A is Σ^S_1. Now S is Π^X_n and so by our induction hypothesis the complement of S is $\leq_m X^{(n)}$. Hence $S \leq_T X^{(n)}$. So by Lemma 16.5.(4), A is $\Sigma^{X^{(n)}}_1$, and so A is c.e. relative to $X^{(n)}$ by Lemma 16.4.

Now to finish, assume A is c.e. relative to $X^{(n)}$ (i.e $\Sigma^{X^{(n)}}_1$. Then by our induction hypothesis, $X^{(n)}$ is Σ^X_n, since $X^{(n)}$ is many-one reducible to itself. So A is Σ^X_{n+1} by Lemma 16.5.(5) □

Corollary 16.8. *Suppose $X \subseteq \mathbb{N}$ and $A \subseteq \mathbb{N}$. Then A is Δ_n^X if and only if $A \leq_T X^{(n-1)}$.*

Proof. A is Δ_n^X if and only if A is Σ_n^X and Π_n^X iff A is c.e. relative to $X^{(n-1)}$ and A is co-c.e. relative to $X^{(n-1)}$ iff A is computable relative to $X^{(n-1)}$. (Here we are using the relativized version of Theorem 4.8. A is computable relative to Y if and only if A is c.e. relative to Y and co-c.e. relative to Y. □

Corollary 16.9 (Properness of the arithmetical hierarchy). *For every n, there is some $A \subseteq \mathbb{N}$ so that A is Σ_n and A is not Π_n.*

Proof. $\emptyset^{(n)}$ cannot be both Σ_n and Π_n, since then it would be Δ_n, and hence $\emptyset^{(n)} \leq_T \emptyset^{(n-1)}$ by Corollary 16.8. But this contradicts the properness of the Turing jump: Theorem 15.10 □

Note that in the above corollary, even though a relativized version of the above result is true (i.e. for every X, there is some A so that A is Σ_n^X and not Π_n^X), we have just stated the unrelativized version. This is usually done in all theorems of computability theory. Even though the result is true when relativized to an arbitrary oracle, we just state the unrelativized version for simplicity of notation.

Note that the complement of such an A likewise gives an example of a Π_n set that is not Σ_n.

More generally the following class of sets can be used to show properness of the arithmetical hierarchy:

Definition 16.10. Say that $A \subseteq \mathbb{N}$ is Σ_n complete if A is Σ_n and for all Σ_n sets $B \subseteq \mathbb{N}$, $B \leq_m A$.

So by Post's theorem, $\emptyset^{(n)}$ is Σ_n complete.

Proposition 16.11. *If A is Σ_n complete, then $X^{(n)} \leq_m A$. Hence A cannot be Δ_n since then $A \leq_T X^{(n-1)}$ which would implies $X^{(n)} \leq X^{(n-1)}$.*

16.2 The limit lemma

In this section, we give a characterization of what sets are computable relative to the halting problem (or equivalently, what sets are Δ_2^0). Suppose $(A_s)_{s\in\mathbb{N}}$ is a sequence of subsets of $\mathbb{N}$, and $A \subseteq \mathbb{N}$. Say that $A = \lim_s A_s$ exists if for all x, $x \in A$ implies $(\forall r > s)[x \in A_r]$ and $x \notin A$ implies $(\exists s)(\forall r > s)[x \notin A_r]$. That is, the characteristic function χ_A of A is the limit of the characteristic functions χ_{A_s}. We say that $\lim_s A_s$ exists if there is some A such that $A = \lim_s A_s$.

Definition 16.12. A set $A \subseteq \mathbb{N}$ is **limit computable** iff there is a computable sequence $(A_s)_{s\in\mathbb{N}}$ of finite subsets of $\mathbb{N}$ so that $A = \lim_s A_s$.

So for example, every c.e. set A is limit computable: take any computable enumeration $(A_s)_{s\in\mathbb{N}}$ of A. However, not limit computability is more general than being computably enumerable: there is no requirement in Definition 16.12 that the sets A_s are increasing. For example, given any c.e. set W_e and its standard enumeration $W_{e,s}$, let $A_s = \{0,\dots,s\} \setminus W_{e,s}$. Then it is easy to check that $\lim A_s = \mathbb{N} \setminus W_e$. Hence, any co-c.e. set is computable.

We characterize the limit computable sets:

Theorem 16.13 (The limit lemma). *Suppose $A \subseteq \mathbb{N}$. Then $A \leq_T \emptyset'$ (or equivalently A is Δ_2) if and only if A is limit computable.*

Proof. First suppose that A is limit computable and $A = \lim_s A_s$. We give an algorithm to compute A relative to $\emptyset'$ as an oracle. For each x and s, construct a program $\varphi_{f(x,s)}$ that halts iff there is some $r > s$ so that $x \in A_s x \notin A_r$ or $x \notin A_s$ and $x \in A_r$. Now for each x and s, using the halting problem as an oracle, we can ask if $\varphi_{f(x,s)}$ halts. Now for each x, since $\lim_s A_s$ exists by asking this question for larger and larger s, there must be some s so that for all larger r, $x \in A_s \leftrightarrow x \in A_r$. So we will find some s so that $\varphi_{f(x,s)}$ never halts. Then $x \in A$ iff $x \in A_s$ for this s.

Conversely, suppose that $A \leq_T \emptyset'$ via the program $\varphi_e^{\emptyset'}$. Let $(K_s)_{s \in \mathbb{N}}$ be a computable enumeration of $\emptyset'$. Let A_s be the set of x such that $\varphi_e^{K_s}(x)$ halts in $\leq s$ steps and outputs 1 Now given any x, we know that $\varphi_e^{\emptyset'}$ halts in some number of steps. Let r be larger than the number of steps this computation takes, and let r also be large enough so that all elements of $\emptyset'$ that are every queried in this computation have been enumerated before stage r. Then clearly $x \in A_r$ iff $x \in A$. □

Exercise 16.14.

1. Show that $f \colon \mathbb{N} \to \mathbb{N}$ is computable relative to $\emptyset'$ iff there is a function $f \colon \mathbb{N}^2 \to \mathbb{N}$ so that for every x, $f(x) = \lim_y g(x, y)$.

2. Show that $f \colon \mathbb{N} \to \mathbb{N}$ is computable relative to $\emptyset^{(n+1)}$ iff there is a function $g \colon \mathbb{N}^{n+1} \to \mathbb{N}$ so that for every x,
$$f(x) = \lim_{y_1} \lim_{y_2} \ldots \lim_{y_n} g(x, y_1, \ldots, y_n).$$

17 Completeness in the arithmetical hierarchy

17.1 Complete Σ_n and Π_n sets up to computable isomorphism

Up to computable isomorphism there is exactly one complete Σ_n set (and exactly one complete Π_n set) for each n. It is the set $\emptyset^n$ (and $\mathbb{N} \setminus \emptyset^{(n)}$).

This follows from the following special property of the Turing jump: being many-one equivalent to X' is equivalent to being 1-1 equivalent to X'.

Lemma 17.1. *Suppose $A \equiv_m X'$. Then $A \equiv_1 X'$.*

Proof. By relativizing the proof of Exercise 5.6, we have that $X' \equiv_1 \{n \colon \varphi_n^X(0)\downarrow\}$. So we may instead work with this version of the halting problem.

By the padding lemma, it is clear if $A \leq_m \{n \colon \varphi_n^X(0)\downarrow\}$, then $A \leq_1 \{n \colon \varphi_n^X(0)\downarrow\}$.

Suppose $\{n \colon \varphi_n^X(0)\downarrow\} \leq_m A$. We would like to show $\{n \colon \varphi_n^X(0)\downarrow\} \leq_1 A$. We will use an identical idea to our proof of the acceptable number theorem: Theorem 6.11.

Let $g \colon \mathbb{N} \to \mathbb{N}$ be the many-one reduction witnessing $\{n \colon \varphi_n^X(0)\downarrow\} \leq_m A$. We would like to find a function $h \colon \mathbb{N}^2 \to \mathbb{N}$ so that for all x, $\varphi_{h(e,x)}^X(0)\downarrow$ iff $\varphi_e^X(0)\downarrow$, and for all $x \neq x'$ we have $g(h(e,x)) \neq g(h(e,x'))$. Given such an h, we can define a 1-1 reduction g' from $\{n \colon \varphi_n^X(0)\downarrow\} \leq_m A$ to A by defining $g'(0) = g(0)$, and $g'(e{+}1)$ is $g(h(e{+}1,n))$ where n is least such that $g(h(e{+}1,n)) \neq g'(i)$ for any $i \leq e$.

We define $h(e,k+1)$ recursively, where $h(e,0) = e$. Let $B_k = \{g(h(e,0)), \ldots, g(h(e,k))\}$. By the recursion theorem, consider the program n where:

$$\varphi_n^X(z) = \begin{cases} \varphi_e^X(z) & \text{if } g(n) \notin B_k \\ \text{undefined} & \text{otherwise} \end{cases}$$

Case 1: If $g(n) \notin B_k$, then $\varphi_n^X = \varphi_e^X$, and $g(n) \neq g(h(e,0)), \ldots, g(h(e,k))$. So we define $h(e,k+1) = n$.

Case 2: if $g(n) \in B_k$, then since $g(n) = g(h(e,i))$ for some $i \leq k$, we have $\varphi_n^X(0)\downarrow \leftrightarrow g(n) \in A \leftrightarrow g(h(e,i)) \in A \leftrightarrow \varphi_{h(e,i)}^X(0)\downarrow \leftrightarrow \varphi_e^X(0)\downarrow$. By definition of n, we must also have that $\varphi_n^X(0)\uparrow$, hence $\varphi_e^X(0)\uparrow$.

Now let m be the program where

$$\varphi_m^X(z) = \begin{cases} 1 & \text{if } g(m) \in B_k \\ \text{undefined} & \text{if } g(m) \notin B_k \end{cases}.$$

If $g(m) \in B_k$, then $g(m) = g(h(e,i))$ for some i, So $\varphi_m^X(0)\downarrow \leftrightarrow g(m) \in A \leftrightarrow g(h(e,i)) \in A \leftrightarrow \varphi_{h(e,i)})^X(0)\downarrow \leftrightarrow \varphi_e^X(0)\downarrow$. But this is a contradiction since $\varphi_m^X(0)\downarrow$ and $\varphi_e^X(0)\uparrow$.

So we must have that $g(m) \notin B_k$ and so $\varphi_m^X(0)\uparrow$ by definition of m. Hence $\varphi_m^X(0)\downarrow iff \varphi_e^X(0)\downarrow$. So let $h(e,k+1) = m$. □

Note that we are using special properties of the Turing jump to prove the above theorem:

Exercise 17.2. Show there are incomputable sets $A, B \subseteq \mathbb{N}$ so that $A \equiv_m B$, but $A \not\equiv_1 B$.

Corollary 17.3. *If A is Σ_n complete, then $A \equiv_1 \emptyset^{(n)}$.*

Proof. Since A is Σ_n, $A \leq_m \emptyset^{(n)}$ by Post's hierarchy theorem. Since $\emptyset^{(n)}$ is Σ_n and A is Σ_n complete, we therefore have that $\emptyset^{(n)} \leq_m A$. So $A \equiv_m \emptyset^{(n)}$. Now letting $X = \emptyset^{(n-1)}$ and applying Lemma 17.1, we see $A \equiv_1 \emptyset^{(n)}$. □

17.2 Examples of complete Π_2/Σ_2 sets

Many naturally occurring sets are Σ_n/Π_n complete. We prove some theorems illustrating how these types of results are proven.

Proposition 17.4. $\mathrm{TOT} = \{e \colon \varphi_e$ *is total*$\}$ *is* Π_2 *complete.*

Proof. First, we need to establish that TOT is Π_2. $e \in \mathrm{TOT}$ iff $(\forall x)(\exists s)\varphi_e(x)$ halts in s steps. Since the relation on tuples (e, x, s) that "$\varphi_e(x)$ halts in s steps" is computable, TOT is Π_2.

Now suppose $A \subseteq \mathbb{N}$ is any Π_2 set. We must show that $A \leq_m \mathrm{TOT}$. Now $x \in A \leftrightarrow \forall y \exists z R(x, y, z)$ for some computable relation R. For each x, consider the program $\varphi_{f(x)}(y)$ where $\varphi_{f(x)}(y)$ searches through all possible z until it finds a z such that $R(x, y, z)$ is true. If it finds such a z, then it halts and outputs this z.

If $x \in A$, for every y there is some z such that $R(x, y, z)$ is true, and so $\varphi_{f(x)}$ will be total. $x \in A \to f(x) \in \mathrm{TOT}$.

If $x \notin A$, then there is some y so that for all z, $R(x, y, z)$ is false. Hence $\varphi_{f(x)}(y)$ will be undefined, and so $x \notin A \to f(x) \notin \mathrm{TOT}$. □

We give another example:

Proposition 17.5. $\mathrm{FIN} = \{e \colon W_e$ *is finite*$\}$ *is* Σ_2 *complete.*

Proof. First, we need to establish that FIN is Σ_2. W_e is finite iff $(\exists n)(\forall s)W_{e,s} \subseteq \{0, \ldots, n\}$. Since this last condition is computable, FIN is clearly Σ_2.

Now suppose $A \subseteq \mathbb{N}$ is any Σ_2 set. We must show that $A \leq_m \mathrm{FIN}$. Now $x \in A \leftrightarrow \exists y \forall z R(x, y, z)$ for some computable relation R. For each x, consider the program $f(x)$ which considers each pair $(y, z) \in \mathbb{N}^2$ infinitely many times and enumerates y if $R(x, y, z)$ is false, and every $y' < y$ has already been enumerated. Note this construction is similar to the proof of Proposition 17.4. The one new feature is that we considering $y' < y$ when making the decision about whether to enumerate y.

If $x \in A$, then there must be some y so that for all z, $R(x, y, z)$ is true. Hence, y will never be enumerated and thus every number larger than y will also never be enumerated. So $W_{f(x)}$ will be finite. So $x \in A \to f(x) \notin \mathrm{FIN}$

If $x \notin A$, then for every y there is some z such that $R(x, y, z)$ is false. Hence, by induction we can see that we will eventually enumerate every y, and hence $W_{f(x)} = \mathbb{N}$. So $x \notin A \to f(x) \notin \mathrm{FIN}$. □

Exercise 17.6. Show that the $\{(d, e) \colon W_d \subseteq W_e\}$ is Σ_2 complete.

17.3 Complete Σ_3 sets

Recall that a set $A \subseteq \mathbb{N}$ is **cofinite** if its complement is finite.

Theorem 17.7. $\mathrm{COF} = \{e \colon W_e$ *is cofinite*$\}$ *is* Σ_3 *complete.*

Proof. $e \in \mathrm{COF}$ if $(\exists n)(\forall x)(\exists s)(x > n \to x \in W_{e,s})$. Since the relation $x > n \to x \in W_{e,s}$ is computable, COF is therefore Σ_3.

Now suppose $A \subseteq \mathbb{N}$ is any Σ_3 set. We must show that $A \leq_m \mathrm{COF}$. Now $x \in A \leftrightarrow \exists y S(x, y)$ for some Π_2 relation S. Since FIN is Σ_2 complete, its complement $\{e \colon W_e$ is infinite$\}$ is if Π_2 complete. Hence, there is some computable function $g \colon \mathbb{N}^2 \to \mathbb{N}$ so that $S(x, y) \leftrightarrow W_{g(x,y)}$ is infinite, and thus $x \in A \leftrightarrow (\exists y)[W_{g(x,y)}$ is infinite$]$.

For each x, we will give a program $f(x)$ so that the c.e. set $W_{f(x)}$ is cofinite iff $(\exists y)[W_{g(x,y)}$ is infinite$]$. The program $f(x)$ works as follows. We watch the enumeration of the sets $W_{g(x,y)}$ for all y. (For example, at step $n = \langle y, t \rangle$ of the program, run the tth step of the enumeration of $W_{g(x,y)}$). Let $b_0^s < b_1^s < b_2^s \ldots$ be the elements that we have *not* yet been enumerated into $W_{f(x)}$ by stage s. If we see an element enumerated into $W_{g(x,y)}$, we enumerate b_y^s.

We claim $x \in A \leftrightarrow e \in \mathrm{COF}$.

Case 1: For all y, $W_{g(x,y)}$ is finite. Then if s is sufficiently large so that no new elements of $W_{g(x,y')}$ are ever enumerated after stage s for any $y' \leq y$, then we will never enumerate any of the numbers $b_0^s, \dots, b_y^s$. Hence, for each y, $b_y = \lim b_y^s$ is never enumerated, and so the set $W_{f(x)}$ has an infinite complement.

Case 2: There exists y such that $W_{g(x,y)}$ is infinite. Note that $b_y^s \leq b_y^{s+1}$, and if we enumerate b_y^s at stage s, then $b_y^s < b_y^{s+1}$. Hence, if we enumerate the number b_y^s at infinitely many stage s, then $\lim_s b_y^s = \infty$, and so the complement of W_e contains less than or equal to y many elements. □

Exercise 17.8. Show that $\{e \colon W_e \text{ is computable}\}$ is Σ_3 complete.

18 The structure of the Turing degrees

18.1 The Kleene-Post theorem

By identifying subsets of $\mathbb{N}$ with their characteristic functions, we can think of any $A \subseteq \mathbb{N}$ as an infinite binary string. We use the notation $2^{\mathbb{N}}$ for the set of all infinite binary strings, and $2^{<\mathbb{N}}$ for the set of all finite binary strings. (The reason for the base 2 in this notation is the usual identification logicians make where we identify 2 with the set $\{0, 1\}$). If $s \in 2^{<\mathbb{N}}$ we let $|s|$ denote the length of s. We let $s(n)$ denote the nth bit of s, which may be 0, 1, or undefined if n is greater than or equal to the length of s. If $s, t \in 2^{<\mathbb{N}}$ are finite binary strings, we say that s is an initial segment of t and write $s \subseteq t$ if for all $n < |s|$, $s(n) = t(n)$. If s, t are finite binary strings, we let $s^\frown t$ denote the finite binary string which is s concatenated with t, so that $(s^\frown t)(n) = s(n)$ if $n < |s|$, and $(s^\frown t)(n) = t(n - |s|)$ otherwise.

Suppose $(s_n)_{n \in \mathbb{N}}$ is an increasing sequence of finite binary strings so $s_0 \subseteq s_1 \subseteq s_2 \ldots$, and $\lim_n |s_n| = \infty$. Then the union $A = \bigcup_n s_n$ of this sequence is an infinite binary string $A \in 2^{\mathbb{N}}$. In computability theory, we can construct $A \in 2^{\mathbb{N}}$ with interesting computability-theoretic properties is in this way by building an increasing sequence of finite initial segments $(s_n)_{n \in \mathbb{N}}$ whose limit is our desired element of $2^{\mathbb{N}}$.

The reason we have switched from using subset of $\mathbb{N}$ to binary strings is that using finite subsets of $\mathbb{N}$ to approximate infinite subset of $\mathbb{N}$ is more clumsy, since it is not as natural to include information that some number is missing from a finite set S if this number is greater than $\max(S)$. However, with finite binary strings we can have zeroes in this sequence which encode this negative information.

We now discuss some computability-theoretic aspects of computing with finite and infinite binary sequences. If $s \in 2^{<\mathbb{N}}$ is a finite binary sequence then we define the computation $\varphi_e^s(x)$ using the oracle s identically to how we define $\varphi_e^A(x)$ except that if the program $\varphi_e^s(x)$ asks for the nth bit of the oracle for some $n \geq |s|$ (so $s(n)$ is undefined), then the computation $\varphi_e^s(x)$ is undefined. Note that if $\varphi_e^s(x)$ is defined, then for all extensions $t \supseteq s$ of s, we must also have that $\varphi_e^t(x)$ is defined and $\varphi_e^s(x) = \varphi_e^t(x)$.

Suppose $A \in 2^{<\mathbb{N}}$, and $\varphi_e^A(x)$ halts. Then since $\varphi_e^A(x)$ runs in finite time, it must only access finitely many bits of the oracle A, and hence there is some initial segment $s \subseteq A$ so that $\varphi_e^s(x)$ is defined and $\varphi_e^s(x) = \varphi_e^A(x)$.

We can put a topology on $2^{\mathbb{N}}$ where the basic open neighborhoods are the sets $N_s = \{A \in 2^{\mathbb{N}} : s \subseteq A\}$ of infinite binary strings extending a given finite binary string $s \in 2^{\mathbb{N}}$. For each e, define a partial function $\Phi_e \colon 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ by defining $A \in \mathrm{dom}(\Phi_e)$ if $\varphi_e^A(n)$ halts for every n, and the nth bit of $\Phi_e(A)$ is $\Phi_e(A)(n) = 0$ if $\varphi_e^A(n) = 0$ and $\Phi_e(A)(n) = 1$ otherwise (so essentially we are forcing the output to always be 0 or 1). So informally, Φ_e maps each $A \in 2^{\mathbb{N}}$ to the infinite binary sequence computed by the eth program using A as an oracle.

Topologically, our discussion above says that the function Φ_e is a partial continuous function. To prove this, note that the set of A so that $\varphi_e^A(n)$ halts and is equal to a particular value is an open set. It is equal to the union of the N_s where $\varphi_e^s(n)$ halts and equals this value.

Theorem 18.1 (Kleene-Post)**.** *There are $A, B \in 2^{\mathbb{N}}$ such that $A \not\geq_T B$ and $B \not\geq_T A$.*

Proof. We construct A and B by finite initial segments. We will definite increasing binary sequence $(s_n)_{n \in \mathbb{N}}$ and $(t_n)_{n \in \mathbb{N}}$ and at the end of the construction let $A = \bigcup_n s_n$ and $B = \bigcup_n t_n$. Begin by defining $s_0 = t_0 = \emptyset$ to be the empty sequences. We will ensure in our construction that the length of these sequences increases at each step ($|s_{n+1}| > |s_n|$ and $|t_{n+1}| > |t_n|$) so that the unions are infinite binary sequences.

At stage $2n + 1$, suppose we have already defined s_{2n} and t_{2n}. We will define $s_{2n+1} \supseteq s_{2n}$ and $t_{2n+1} \supseteq t_{2n}$ to ensure that $\Phi_n(A) \neq B$. Let $m = |t_n|$ be so that $t_n(m)$ has not been defined.

Case 1: suppose that there is some extension $s^* \supseteq s_{2n}$ such that $\varphi_n^{s^*}(m)$ halts. Then define $s_{2n+1} = s^*$ and if $\varphi_n^{s^*}(m) = 0$, define $t_{2n+1} = t_{2n}{}^\frown 1$. Otherwise, define $t_{2n+1} = t_{2n}{}^\frown 1$. At the end of the construction, since $A \supseteq s_{2n+1}$ and $B \supseteq t_{2n+1}$, we therefore have that $\varphi_n^A(m) \neq B$, and hence $\Phi_n(A) \neq B$.

Case 2: if there is no extension $s^* \supseteq s_{2n}$ such that $\varphi_n^{s^*}(m)$ halts, the define $s_{2n+1} = s_{2n}{}^\frown 0$ and $t_{2n+1} = t_{2n}{}^\frown 0$. Then at the end of the construction, since $A \supseteq s_{2n}$, we must have that $\varphi_n^A(m)$ does not halt. If it did halt, then there would be some initial segment $s^* \subseteq A$ such that $\varphi_n^{s^*}(m)$ halts, contradicting that we are in case 2. Thus, $\Phi_n(A)$ is undefined and so $\Phi_n(A) \neq B$.

At stage $2n+2$, we perform the same process, switching the roles of A and B and s_n and t_n. □

By paying closer to exactly how complicated each step in this construction is to compute, we have the following refinement of the theorem.

Theorem 18.2. *There are $A, B \leq_T \emptyset'$ such that $A \not\geq_T B$ and $B \not\geq_T A$.*

Proof. This follows by noting that each step in the construction of Theorem 18.1 can be computed relative to $\emptyset'$ as an oracle, so from $\emptyset'$, we can compute the sequences $(s_n)_{n\in\mathbb{N}}$ and $(t_n)_{n\in\mathbb{N}}$.

Given any s_{2n} and m, we can write a program which searches through all s^* extending s_{2n} and possible finite running times and halts if it ever finds any s^* so that $\varphi_n^{s^*}(m)$ halts. By asking $\emptyset'$ if this program ever halts we can determine whether we are in Case 1 or Case 2. If we are in Case 1, then we can computably perform this search again until we actually find such a s^*, and then let $s_{2n+1} = s^*$, and define t_{2n+1}. Otherwise, the extension made in Case 2 is clearly computable. □

Note that in the proof of the above theorem, we really aren't using anything about computability other than the fact that computability is partial continuous.

19 Baire category, forcing, and the Turing degrees

19.1 Forcing Σ_1 sentences

Say that a set $D \subseteq 2^{<\mathbb{N}}$ of strings is **dense** if for every $s \in 2^{<\mathbb{N}}$ there exists some $s^* \supseteq s$ so that $s^* \in D$. Likewise, say that a set $D \subseteq 2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$ of pairs of strings is **dense** if for all $(s,t) \in 2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$ there exists $(s^*, t^*) \in D$ so that $s^* \supseteq s$ and $t^* \supseteq t$.

Example 19.1. The set of strings of even length $\{scolon|s| \text{ is even}\}$ is dense. The set of strings containing a 1 is dense. If $A \in 2^{\mathbb{N}}$, $\{s \colon s \not\subseteq A\}$ is dense.

Suppose we are going to build an infinite binary string by finite initial segments. In this middle of the construction we will have committed to some finite initial segment s of this infinite binary string, but we will have not yet defined any of the remaining bits. Still, it is useful to have a "name" for the infinite binary string we are building, even though we have not finished with the construction yet. We typically let the symbol G be this name. (Formally, G is part of what is called the **forcing language**.)

Definition 19.2. Suppose $s \in 2^{<\mathbb{N}}$ is a finite string. Say that s **forces** $\varphi_e^G(m)\downarrow$ and write $s \Vdash \varphi_e^G(m)\downarrow$ if $\varphi_e^s(m)\downarrow$. Similarly, say that s **forces** $\varphi_e^G(m) = k$ and write $s \Vdash \varphi_e^G(m) = k$ if $\varphi_e^s(m) = k$. Say s **forces** $\varphi_e^G(m)\uparrow$ and write $s \Vdash \varphi_e^G(m)\uparrow$ if for all s^* extending s, it is not the case that $s \Vdash \varphi_e^G(m)\downarrow$.

Exercise 19.3. The relations $s \Vdash \varphi_e^G(m)\downarrow$ and $s \Vdash \varphi_e^G(m)\uparrow$ are both computable relative to $\emptyset'$. That is, the set of tuples (s, e, m) so that these relations hold are computable relative to $\emptyset'$.

In what follows, we will re-examine the proof of Theorem 18.1, and break up the argument into some smaller sublemmas that will be useful on their own.

Lemma 19.4. *Fix a program e. Let D_e be the set of strings $(s,t) \in 2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$ such that there exists some m so that either:*

- *$s \Vdash \varphi_e^G(m)\uparrow$, or*
- *$s \Vdash \varphi_e^G(m) = k$ and $t(m) \neq k$.*

Then D_e is dense.

Proof. Given any $(s_0, t_0) \in 2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$, we will show there is some $(s,t) \in D_e$ so that $s \supseteq s_0$ and $t \supseteq t_0$. Let $m = |t_0|$.

Case 1: there is some extension $s \supseteq s_0$ such that $s \Vdash \varphi_e^G(m)\downarrow$. Then choose $t \supseteq t_0$ so that $t(m) \neq \varphi_e^s(m)$, and note that $(s,t) \in D_e$.

Case 2: there is no extension $s \supseteq s_0$ such that $s \Vdash \varphi_e^G(m)\downarrow$. Then $s_0 \Vdash \varphi_e^G(m)\uparrow$ by definition, and so $(s_0, t_0) \in D_e$. □

If $A \in 2^{\mathbb{N}}$ and $D \subseteq 2^{<\mathbb{N}}$, say that A **meets** D if there is some $s \in D$ such that $A \supseteq s$. Similarly, if $D' \subseteq 2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$, and $A, B \in 2^{\mathbb{N}}$, say that (A, B) meets D' if there is some $(s,t) \in D'$ such that $A \supseteq s$ and $B \supseteq t$.

The importance of the dense set D_e from Lemma 19.4 is the following.

Lemma 19.5. *If $(A, B) \in 2^{\mathbb{N}} \times 2^{\mathbb{N}}$ meets the set D_e from Lemma 19.4, then $\Phi_e(A) \neq B$.*

Proof. Fix $(s,t) \in D_e$ so that $A \supseteq s$ and $B \supseteq t$. Then there is some m so that either

- $s \Vdash \varphi_e^G(m)\uparrow$. In this case $\varphi_e^A(m)\uparrow$, since otherwise if $\varphi_e^A(m)\downarrow$, there would be some finite initial segment $s^* \subseteq A$ so that $\varphi_e^A(m)\downarrow$. Here we make take s^* sufficiently long so that $|s^*| \geq |s|$. But then $s^* \supseteq s$, contradicting that definition of $s \Vdash \varphi_e^G(m)\uparrow$.

- $s \Vdash \varphi_e^G(m) = k$ and $t(m) \neq k$. In this case, we also have $\varphi_e^A(m) = k$ since $A \supseteq s$, and $B(m) \neq k$ since $B \supseteq t$. So $\Phi_e(A) \neq B$.

□

Now we prove that we can meet countably many dense sets:

Lemma 19.6. *Suppose $D_0, D_1, \dots$ are countably many dense sets in $2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$. Then there are $(A, B) \in 2^{\mathbb{N}} \times 2^{\mathbb{N}}$ such that (A, B) meets D_n for every $n \in \mathbb{N}$.*

Proof. We define increasing sequences $(s_n)_{n\in\mathbb{N}}$ and $(t_n)_{n\in\mathbb{N}}$. Let $s_0 = t_0 = \emptyset$. Given (s_n, t_n) define (s_{n+1}, t_{n+1}) to be such that (s_{n+1}, t_{n+1}) extends (s_n, t_n), $(s_{n+1}, t_{n+1}) \in D_n$, and $|s_n| \geq n$ and $t_{n+1}| \geq n$. Such an (s_{n+1}, t_{n+1}) exists by the density of D_n. To finish, define $A = \bigcup_n s_n$ and $B = \bigcup_n t_n$. So (A, B) meets D_n since $A \supseteq s_{n+1}$ and $B \supseteq t_{n+1}$. □

If we think topologically, the above lemma is really a special case of the Baire category theorem. The topology on the space $2^{\mathbb{N}}$ that we have already defined (with basic open sets of the form N_s) can be generated by the metric d where $d(A, B) = 1/\min\{n \colon A(n) \neq B(n)\}$.

Exercise 19.7 (Baire category theorem)**.** Suppose X is a complete metric space, and $(U_n)_{n\in\mathbb{N}}$ is a countable sequence of open dense subsets of X. Then $\bigcap U_n$ is nonempty.

We could put together the above lemmas to re-prove the Kleene-Post theorem. Instead, we will prove a slightly stronger theorem. Since there are countably many pairs of natural numbers, we can construct countably many $A_n \in 2^{\mathbb{N}}$ that pairwise meet a countable collection of dense sets.

Lemma 19.8. *Suppose $(D_n)_{n\in\mathbb{N}}$ are countably many dense sets in $2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$. Then there are countably many $(A_n)_{n\in\mathbb{N}}$ in $2^{\mathbb{N}}$ so that (A_i, A_j) meets D_n for every i, j, n with $i \neq j$.*

Proof. For each i, we will define an increasing sequence $(s_{i,n})_{n\in\mathbb{N}}$ of finite strings such that $\bigcup_n s_{i,n} = A_i$. Let $s_{i,0} = \emptyset$ for every i.

Choose a bijection $\rho \colon \mathbb{N} \to \mathbb{N}^3$. At step k, if $\rho(k) = (i, j, n)$ define $s_{i,k+1}$ and $s_{j,k+1}$ extending $s_{i,k}$ and $s_{j,k}$ to be so that $(s_{i,k+1}, s_{j,k+1}) \in D_n$, and $|s_{i,k+1}| \geq k$ and $|s_{j,k+1}| \geq k$. This is possible by the density of D_n. For all i' where $i' \neq i$ and $i' \neq j$, let $s_{i',k+1} = s_{i,k}$. □

Corollary 19.9. *There are countably many $(A_n)_{n\in\mathbb{N}}$ where $A_n \in 2^{\mathbb{N}}$ such that $A_i \not\geq_T A_j$ for any $i \neq j$.*

Proof. Apply Lemma 19.8 to the dense sets in Lemma 19.4 and apply Lemma 19.5. □

19.2 Turing incomparability with a given $A \in 2^{\mathbb{N}}$

Suppose $A \in 2^{\mathbb{N}}$ is incomputable. Must there always be some B such that $B \not\geq_T A$ and $A \not\geq_T B$? The answer to this question is yes, and this follows from the following lemma.

Lemma 19.10. *Suppose $A \in 2^{\mathbb{N}}$ is incomputable. Let D_e^A be the set of $s \in 2^{\mathbb{N}}$ such that there exists an m such that*

- $s \Vdash \varphi_e^G(m)\uparrow$*, or*
- $s \Vdash \varphi_e^G(m) \neq A(m)$*.*

Then D_e^A is dense.

Proof. Suppose $s_0 \in 2^{<\mathbb{N}}$. We need to show there is some $s \supseteq s_0$ so that $s \in D_e^A$. Now if there is some m so that $s_0 \Vdash \varphi_e^G(m)\uparrow$, then we are done. So assume that for every m, $s_0 \nVdash \varphi_e^G(m)\uparrow$ (which by definition means that for some $s \supseteq s_0$, $s \Vdash \varphi_e^G(m)\downarrow$).

We claim there must be some m and $s \supseteq s_0$ so that $s \Vdash \varphi_e^G(m) \neq A(m)$. If this were not the case, then we could compute A using the following algorithm. To compute $A(m)$, search through all $s \supseteq s_0$ and all possible running times until we find some s such that $s \Vdash \varphi_e^G(m)\downarrow$. We must eventually find such an s by our assumption that $s_0 \nVdash \varphi_e^G(m)\uparrow$. Now for this s, we must have $\varphi_e^s(m) = A(m)$ by our assumption that there is no s such that $s \Vdash \varphi_e^G(m) \neq A(m)$. □

Corollary 19.11. *If $A \in 2^{\mathbb{N}}$ is incomputable, then there is some $B \in 2^{\mathbb{N}}$ such that $A \not\geq_T B$ and $B \not\geq_T A$.*

Proof. For each set $C \in 2^{\mathbb{N}}$ that is computable relative to A, the set $D_C = \{s \colon \exists m s(m) \neq C(m)\}$ is dense, and if B meets this dense set D_C, the $B \neq C$. Construct B to meet the countably many dense sets D_e^A in the above lemma, and also the dense sets D_C. Then $B \not\geq_T A$, and we also have that B is not equal to any C that is computable relative to A, so $A \not\geq_T B$. □

A similar proof shows that if we have a countable set $A_0, A_1 \ldots$ of incomputable element of $2^{\mathbb{N}}$, there is some B that is Turing incomparable with all of them, by meeting the countably many dense sets $D_e^{A_i}$ for every i and e. By taking larger and large Turing incomparable sets in this way, we can construct antichains in the Turing degrees of size $\aleph_1$.

19.3 Trees and antichains

How big can antichains in the Turing degrees be? We will show there are antichains with cardinality $2^{\aleph_0}$.

Definition 19.12. A **tree in** $2^{<\mathbb{N}}$ is a nonempty set $T \subseteq 2^{<\mathbb{N}}$ closed downwards under $\subseteq$ so that if $s \in T$ and $t \subseteq s$, then $t \in T$. If T is tree, then the set of paths through T, noted $[T]$ is the set of $A \in 2^{\mathbb{N}}$ so that for all $s \subseteq A$ we have $s \in T$. An element $s \in T$ is a **leaf** if there is no proper extension $t \supsetneq s$ so that $t \in T$. A tree T is **perfect** if for all $t \in T$, there are $t_0, t_1 \in T$ so that $t_0 \supseteq t$ and $t_1 \supseteq t$ so $t_0, t_1 \in T$ and t_0 and t_1 are incompatible (i.e. $t_0 \not\subseteq t_1$ and $t_1 \not\subseteq t_0$).

If T is a tree we say that $t \in T$ is a **splitting node** if $t^\frown 0 \in T$ and $t^\frown 1 \in T$. So a tree T is perfect if for every $t \in T$, there is some $t^* \supseteq t$ that is a splitting node. If $s, t \in 2^{\mathbb{N}}$ we say that s, t are compatible, and write $s \parallel t$ if either $s \subseteq t$ or $t \subseteq s$. Using this language, note that if $t^* \supseteq t$ is the splitting above t of minimal length, then for all $s \supseteq t$, we must have $s \parallel t^*$.

Proposition 19.13. *Suppose $\{t_s\}_{s \in 2^{<\mathbb{N}}}$ of strings so that:*

- *If $s \subseteq s^*$, then $t_s \subseteq t_{s^*}$.*
- *For all s, $t_{s^\frown 0}$ and $t_{s^\frown 1}$ are incompatible.*

Then $T = \{t \colon t \subseteq t_s$ for some $s \in 2^{<\mathbb{N}}\}$ is a perfect tree.

Proof. T is clearly closed downwards, and if $t \subseteq t_s$, then $t_{s^\frown 0}$ and $t_{s^\frown 1}$ are two incompatible extensions of t inside T. □

In fact, every tree is of this form.

Lemma 19.14. *If $T \subseteq 2^{<\mathbb{N}}$ is **perfect**, then there is some set $\{t_s\}_{s \in 2^{<\mathbb{N}}}$ of strings so that:*

- *If $s \subseteq s^*$, then $t_s \subseteq t_{s^*}$.*
- *For all s, $t_{s^\frown 0}$ and $t_{s^\frown 1}$ are incompatible.*

and $T = \{t \colon t \subseteq t_s$ *for some* $s \in 2^{<\mathbb{N}}\}$.

Proof. Let $t_\emptyset$ be the minimal splitting in T above $\emptyset$. We will inductively ensure that t_s is a splitting node for each s. Given $t_s \in T$, since t_s is a splitting node, $t_s{}^\frown 0$ and $t_s{}^\frown 1$ are both elements of T. So let $t_{s^\frown 0}$ be the least splitting node above $t_s{}^\frown 0$ and $t_{s^\frown 1}$ be the least splitting node above $t_s{}^\frown 1$. Clearly $t_s \in T$ for every s, and so the tree $\{t \colon t \subseteq t_s$ for some $s \in 2^{<\mathbb{N}}\}$ is contained in T. Conversely, for any $t' \in T$, it is easy to prove that $t' \in \{t \colon t \subseteq t_s$ for some $s \in 2^{<\mathbb{N}}\}$ by induction on the number of splitting nodes in t'. □

Lemma 19.15. *If* T *is a perfect tree, then there is a continuous bijection* $f \colon 2^{\mathbb{N}} \to [T]$*, and hence the cardinality of* $[T]$ *is equal to the cardinality of* $2^{\mathbb{N}}$ *which is equal to* $2^{\aleph_0}$*.*

Proof. Let $\{t_s\}_{s \in 2^{<\mathbb{N}}}$ be a set of strings as in Lemma 19.14. Then define $f(A) = \bigcup_{s \subseteq A} t_s$.

The map f is an injection since if $A \neq B$, then if n is least such that $A \upharpoonright n \neq B \upharpoonright n$, then since $A \upharpoonright n$ and $B \upharpoonright n$ are equal to $s^\frown 0$ and $s^\frown 1$ for some s. Hence, $f(A)$ and $f(B)$ extend incompatible strings $t_{s^\frown 0}$ and $t_{s^\frown 1}$.

Now f is onto, since if $Y \in [T]$, then we can find the $X \in 2^{\mathbb{N}}$ such that $f(X) = Y$ as follows. We will construct a sequence $(s_n)_{n \in \mathbb{N}}$ such that $Y = \bigcup_n t_{s_n}$. Let $s_0 = \emptyset$, and given s_n, let t^* be the least splitting node in T above t_{s_n}, so since Y extends t_{s_n}, Y extends t^*. If $Y \supseteq t^{*\frown} 0$, let $s_{n+1} = s_n{}^\frown 0$. Otherwise, if $Y \supseteq t^{*\frown} 1$, let $s_{n+1} = s_n{}^\frown 1$. So then $Y \supseteq t_{s_{n+1}}$. It is easy to check that f is continuous. □

Exercise 19.16. Suppose $C \subseteq 2^{\mathbb{N}}$. Then $C = [T]$ for some perfect tree T iff there is a continuous injection $f \colon 2^{\mathbb{N}} \to 2^{\mathbb{N}}$ so that $c = \operatorname{ran}(f)$.

Lemma 19.17. *If* $(D_n)_{n \in \mathbb{N}}$ *are countably many dense sets in* $2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$*, then there is a perfect tree* $T \subseteq 2^{<\mathbb{N}}$ *so that for any two* $A, B \in [T]$ *such that* $A \neq B$*, we have that* (A, B) *meets* D_n *for every* n*.*

Proof. First, we claim that for any collection $s_0, \ldots, s_k$ of strings and any n, we can find $s_0^*, \ldots, s_k^*$ extending them (i.e. $s_i^* \supseteq s_i$ for all $i \leq k$) so that for any $i, j \leq k$ we have (s_i^*, s_j^*) meets D_m for every $m \leq n$. This is because there are finitely many $i, j \leq k$ and for each such pair, we can extend the strings s_i, s_j to meet each D_m for $m \leq n$.

We will construct strings $(t_s)_{s \in 2^{<\mathbb{N}}}$ where $t_{s^*} \supseteq t_s$ if $s^* \supseteq s$, and let our tree T be $\{t \colon (\exists s) t \subseteq t_s\}$. Let $t_\emptyset = \emptyset$.

Now suppose we have defined t_s for all s of length n. Let $r_{s^\frown 0} = t_s{}^\frown 0$ and $r_{s^\frown 1} = t_s{}^\frown 1$ for every $s \in 2^{<\mathbb{N}}$. Now take the collection $\{r_s \colon |s| = n+1\}$ and extend them by the above claim to a collection $\{t_s \colon |s| = n+1\}$ so $t_s \supseteq r_s$ and so given any $s \neq s'$ we have the (t_s, t_s') meets D_m for every $m \leq n$.

Now if $A, B \in [T]$ as defined above, and $A \neq B$, then we must have $A \supseteq t_s$ and $B \supseteq t_{s'}$ for some incompatible s, s' where $|s| \geq n$ and $|s'| \geq n$. Hence, (A, B) meets D_n. □

Corollary 19.18. *There is an antichain in* $(\mathcal{D}_T, \leq_T)$ *of cardinality* $2^{\aleph_0}$

Proof. Applying Lemma 19.17 to the dense sets D_n from Lemma 19.4, we obtain a perfect tree T so that for any $A, B \in [T]$ with $A \neq B$, we have $A \not\geq_T B$ and $B \not\geq_T A$. □

20 1-generics

20.1 Forcing=truth for 1-generics

There is an important asymmetry in the definition of the forcing relation. Suppose that $A \in 2^{\mathbb{N}}$. If $\varphi_e^A(n)\downarrow$, then there is some $s \subseteq A$ so that $s \Vdash \varphi_e^G(n)\downarrow$. However, if $\varphi_e^A(n)\uparrow$, then it is not necessarily the case that there is some $s \subseteq A$ so that $s \Vdash \varphi_e^G(n)\uparrow$. For example, consider the case where A is the infinite binary sequence of all 0s, and let φ_e be the program that halts if there is a some n so that the nth bit of the oracle is a 1.

However, for certain types of sets, we will have this property that anything true about A must be forced.

Definition 20.1. Suppose $S \subseteq 2^{<\mathbb{N}}$. The **densification** of S is the set $S' = S \cup \{s \colon (\forall s^* \supseteq s)[s^* \notin S]\}$. That is, S' is S together with the strings having no extension above S.

It is clear that the densification of any set is a dense set. For example, the densification of the empty set $\emptyset$ is $2^{<\mathbb{N}}$

Definition 20.2. Say that $A \in 2^{\mathbb{N}}$ is **1-generic** if for every Σ_1 set $S \subseteq 2^{<\mathbb{N}}$, either A meets S, or there is some $s \subseteq A$ so that for all $s^* \supseteq s$, we have $s^* \notin S$. That is, A meets the densification of S.

Note that the set of 1-generics is not empty, since we can meet the countably many dense sets in its definition.

Lemma 20.3. *Suppose $A \subseteq 2^{\mathbb{N}}$. The following are equivalent.*

1. *A is 1-generic.*
2. *For every e, n, there is some $s \subseteq A$ such that $s \Vdash \varphi_e^G(n)\downarrow$ or $s \Vdash \varphi_e^G(n)\uparrow$.*

Proof. (1) ⇒ (2): Suppose A is 1-generic. Then for each e, n $\{s \colon s \Vdash \varphi_e^G(n)\downarrow\}$ is a Σ_1 set of strings; we can enumerate this set by running $\varphi_e^s(n)$ for all s and all possible finite numbers of steps, and enumerating the s for which this computation halts. Hence, either A meets this set, or by definition of 1-genericity, there is some $s \subseteq A$ so that no $s^* \supseteq s$ has $s^* \Vdash \varphi_e^G(n)\downarrow$. But by definition this means that $s \Vdash \varphi_e^G(n)\uparrow$.

(2) ⇒ (1): Suppose $S \subseteq 2^{<\mathbb{N}}$ is a Σ_1 set of strings. Then we can construct an oracle program which watches the enumeration of S, and halts if there is some initial segment of its oracle which is in this set. Thus, there must either be some $s \subseteq A$ so that $s \in S$, or some $s \subseteq A$ so that no $s^* \supseteq s$ has $s^* \in S$. □

Corollary 20.4 (Forcing=Truth for 1-generics)**.** *If A is 1-generic iff for all e, n*

- *$\varphi_e^A(n)\downarrow$ iff $(\exists s \subseteq A) s \Vdash \varphi_e^G(n)\downarrow$.*
- *$\varphi_e^A(n)\uparrow$ iff $(\exists s \subseteq A) s \Vdash \varphi_e^G(n)\uparrow$.*

Lemma 20.5. *Suppose A is 1-generic. Then $A' \leq_T \emptyset' \oplus A$.*

Proof. $e \in A'$ if and or equivalently $\varphi_e^A(e)\downarrow$. To determine if $\varphi_e^A(e)\downarrow$ using A as an oracle, we take each finite initial segment $s \subseteq A$, and then compute using $\emptyset'$ as an oracle whether $s \Vdash \varphi_e^G(e)\downarrow$, or $s \Vdash \varphi_e^G(e)\downarrow$. □

The property that $A' \leq_T \emptyset' \oplus A$ is sometimes called being GL_1. Note that all A do not have this property. For example, if $A \geq_T \emptyset'$, then A cannot be GL_1

Exercise 20.6. If $A, B \in 2^{\mathbb{N}}$, then $A \oplus B$ is 1-generic iff A is 1-generic and B is 1-generic relative to A.

20.2 Friedberg jump inversion

Theorem 20.7. *Suppose $B \in 2^{\mathbb{N}}$ is such that $B \geq_T \emptyset'$. Then $B \equiv_T A'$ for some $A \in 2^{\mathbb{N}}$.*

Proof. We will find A such that

$$A' \equiv_T A \oplus \emptyset' \equiv_T B$$

Fix a program which runs using the oracle $\emptyset'$, which takes $s \in 2^{\mathbb{N}}$ and a c.e. set R of strings, and then outputs some $s^* \supseteq s$, so that s^* is in the densification of S. (i.e. the program outputs s if there is no extension $s^* \supseteq s$ so that $s^* \in S$, and otherwise outputs some outputs some $s^* \supseteq s$ such that $s^* \in S$.

We will construct A by finite initial segments, so $A = \bigcup_n s_n$. Let R_n be the nth c.e. set of finite binary strings. Let $s_0 = \emptyset$. Given s_n, let $s^* \supseteq s_n$ be the string found by the program using the oracle $\emptyset'$ so that s^* is in the densification of R_n, then let $s_{n+1} = s^{*\frown}0$ if $B(n) = 0$, and $s_{n+1} = s^{*\frown}1$ if $B(n) = 1$.

Clearly, this set A can be computed by B, since $B \geq_T \emptyset'$. So $B \geq_T A \oplus \emptyset$. Conversely, $A \oplus \emptyset$ can compute B, since using $\emptyset$, we can find this s^* extending s_n, and from the next bit we can recover the value $B(n)$.

Now A will be 1-generic, so $A \geq_T A \oplus \emptyset'$. Finally, $A' \geq_T \emptyset'$ and $A \geq_T A$ are trivial, so $A' \geq_T A \oplus \emptyset'$. □

21 Forcing in arithmetic

21.1 The forcing relation and n-generics

In what follows, we will always use lowercase letters n, m, x, y to indicate natural numbers, and uppercase letters A, B, X, Y to indicate elements of $2^{\mathbb{N}}$.

Suppose $R(x_1, \dots, x_n, Y_1, \dots, Y_n)$ is a relation where $x_1, \dots, x_n$ are variables taking values in $\mathbb{N}$ and $Y_1, \dots, Y_N$ take values in $2^{\mathbb{N}}$. We say that R is **computable** if its truth value can be computed by an oracle Turing machine with oracle $Y_1 \oplus \dots \oplus Y_n$ on input $(x_1, \dots, x_n)$. For example, the relation $R(x, Y)$ which is true if $x \in Y$ is a computable relation.

We define the arithmetical hierarchy on formulas built from these relations as usual. A formula φ is Σ^0_n if it of the form

$$\exists x_1 \forall x_2 \dots Q x_n R(x_1, \dots, x_n, y_1, X_1, \dots, X_n)$$

where R is computable. A formula is Π^0_n if it is of the form

$$\forall x_1 \exists x_2 \dots Q x_n R(x_1, \dots, x_n, y_1, X_1, \dots, X_n)$$

where R is computable.

We similarly say a set is Σ^0_n/Π^0_n iff it is defined by a Σ^0_n/Π^0_n relation.

Definition 21.1. Define the forcing relation on arithmetical formulas $\psi(X)$ with one free variable as follows. For computable relations R, $s \Vdash R(G)$ if the computation of R terminates returning true using the bits of s for the oracle. Then inductively,

- $s \Vdash \exists x \psi(x, G)$ iff there exists some n such that $s \Vdash \psi(n, G)$.
- $s \Vdash \neg\psi(G)$ if for all $s^* \supseteq s$, it is not the case that $s \Vdash \psi(G)$.

Note here that we regard $\forall x$ as being an abbreviation for $\neg\exists x\neg$. If we wanted, we could also define the forcing relation for conjunctions: $s \Vdash \psi(G) \wedge \theta(G)$ if $s \Vdash \psi(G)$ and $s \Vdash \theta(G)$, however we will not use this in our proofs, since all the formulas we will consider will be in prenex normal form followed by a computable relation.

In our definition of the forcing relation above, it is not true that if ψ and θ are logically equivalent, then $s \Vdash \psi(G)$ iff $s \Vdash \theta(G)$. For example, this is true even if θ is the formula $\neg\neg\psi$. (In the set theory literature, what we have defined above is usually called the strong forcing relation, which differs slightly from what is usually called the forcing relation).

Lemma 21.2. *For each Σ^0_n formula ψ, $\{s \colon s \Vdash \psi(G)\}$ is a Σ^0_n set. For each Π^0_n formula ψ, $\{s \colon s \Vdash \psi(G)\}$ is a Π^0_n set.*

Proof. By induction. If R is computable, $\{s \colon s \Vdash \exists x R(G, x)\}$ is c.e., since we can enumerate all s for which there is some n such that $R(n, s)$ halts and is true.

If ψ is Π^0_1, For each Π^0_n

For each Σ^0_n formula $\exists x R(x, G)$, we have that $s \Vdash \exists x S(x, G)$ iff $(\exists n)[s \Vdash S(n, G)]$ and so $\{s \colon s \Vdash \exists x R(x, G)\}$ is Σ^0_n assuming that $\{(s, n) \colon s \Vdash S(n, G)\}$ is uniformly Σ^0_n.

For each Π^0_n formula $\forall x S(x, G)$ where S is Σ^0_{n-1}, we have

$$\begin{aligned} s \Vdash \forall x R(x, G) &\leftrightarrow s \Vdash \neg\exists x \neg R(x, G) \\ &\leftrightarrow (\forall s^* \supseteq s) \neg s \Vdash \exists x \neg R(x, G) \\ &\leftrightarrow (\forall s^* \supseteq s) \neg (\exists n) s^* \Vdash \neg R(n, G) \\ &\leftrightarrow (\forall s^* \supseteq s) \neg (\exists n) (\forall s^{**} \supseteq s^*) \neg s^{**} \Vdash R(n, G) \\ &\leftrightarrow (\forall s^* \supseteq s) (\forall n) (\exists s^{**} \supseteq s^*) s^{**} \Vdash R(n, G) \end{aligned}$$

which is $\boldsymbol{\Pi}^0_n$ since the relation $s^{**} \Vdash R(n, G)$ is $\boldsymbol{\Sigma}^0_{n-1}$ by our induction hypothesis. □

Definition 21.3. Say that $A \in 2^{\mathbb{N}}$ is n-generic iff for every Σ^0_n set $S \subseteq 2^{<\mathbb{N}}$, either there is some $s \in S$ such that $s \subseteq A$, or there is some $s \subseteq A$ such that for all $s^* \supseteq s$, $s^* \notin S$.

Lemma 21.4 (Forcing = truth)**.** *Suppose A is n-generic. Then for every Σ^0_n or Π^0_n formula $\psi(X)$, $\psi(A)$ is true iff there is some $s \subseteq A$ such that $s \Vdash \psi(G)$.*

Proof. By induction. We have already shown that this is true $n = 1$. Suppose that A is $k+1$-generic and $\psi(X) = \exists x R(x, X)$ is a Σ^0_{k+1} formula. Then $\psi(A)$ is true iff there is some n such that $R(n, A)$ is true which is true iff there is some n and some $s \subseteq A$ such that $s \Vdash R(n, G)$ (since A is k-generic) iff there is some $s \subseteq A$ such that $s \Vdash \exists x R(x, G)$ (by definition).

Suppose now that $\psi(X) = \forall x R(x, X)$ is a Π^0_{k+1} formula. Now $\neg R(x, X)$ is a Π^0_k relation. Hence, $\{s \colon s \Vdash \exists x \neg R(x, G)\}$ is Σ^0_{k+1}. Thus, either

1. there exists $s \subseteq A$ such that $s \Vdash \exists x \neg R(x, G)$, or

2. there exists $s \subseteq A$ so that no $s^* \supseteq s$ has $s^* \Vdash \exists x \neg R(x, G)$.

(1) and (2) are mutually exclusive, and (1) is true iff $\psi(A)$ is false by the above. So $\psi(A)$ is true iff (2) is true iff $s \Vdash \neg \exists x \neg R(x, G)$. □

Exercise 21.5. Suppose $A \subseteq 2^{\mathbb{N}}$. The following are equivalent.

1. A is k-generic.

2. For every Σ_n sentence $\varphi(G)$ with one free variable X, there is some $s \subseteq A$ such that $s \Vdash \varphi(G)$ or $s \Vdash \neg\varphi(G)$.

Exercise 21.6. For each n, there are $A, B \in 2^{\mathbb{N}}$ such that $A^{(n)} \not\geq_T B$ and $B^{(n)} \not\geq_T A$.

Exercise 21.7. If A is n-generic, then $A^{(n)} \equiv_T A \oplus \emptyset^{(n)}$.

Exercise 21.8. For each n, if $B \geq_T \emptyset^n$ then there is some A such that $A^{(n)} \equiv_T B$.

22 A minimal Turing degree

Our goal in this section is to prove Spector's theorem that there is a minimal Turing degree.

We will prove this theorem using a forcing construction, however, we will not approximate A using finite initial segments. Instead, our approximations to A will be computable perfect trees. (In set theory, forcing using perfect trees is called Sacks forcing and forcing with finite binary sequences is called Cohen forcing).

Definition 22.1. Suppose $e \in \mathbb{N}$ and $T \subseteq 2^{<\omega}$ is a computable perfect tree. Let $\{t_s\}_{s \in 2^{<\mathbb{N}}}$ be the set of splitting nodes in T as in Lemma 19.14. Say that T is **e-splitting** if for all $s \in 2^{\mathbb{N}}$, there exists some k so that both $\varphi_e^{t_{s^\frown 0}}(k)$ and $\varphi_e^{t_{s^\frown 1}}(k)$ halt and $\varphi_e^{t_{s^\frown 0}}(k) \neq \varphi_e^{t_{s^\frown 1}}(k)$.

For example, if φ_e is a program which just outputs its oracle, so $\varphi_e^s(n) = s(n)$ and so $\Phi_e(A) = A$ for all $A \in 2^{\mathbb{N}}$, then every computable perfect tree T is e-splitting. For all s, if t_s has length k, then $\varphi_e^{t_{s^\frown 0}}(k) \neq \varphi_e^{t_{s^\frown 1}}(k)$. If $\varphi_{e'}$ is a program which ignores its oracle and always just computes a fixed computable function, then no computable perfect tree is e'-splitting.

The key property of an e-splitting tree is the following lemma:

Lemma 22.2. *Suppose T is a computable perfect e-splitting tree, $A \in [T]$, and $A \geq_T B$ via e, so $\Phi_e(A) = B$. Then $B \geq_T A$.*

Proof. Suppose $A \in [T]$ and $\Phi_e(A) = B$. We will give an algorithm for computing A from B. We will recursively define a sequence $(s_n)_{n\in\mathbb{N}}$ where $s_{n+1} \supseteq s_n$ for all n so that $A = \bigcup_n t_{s_n}$. Let $s_0 = \emptyset$. Suppose we have already determined t_{s_n}, and we would like to compute $t_{s_{n+1}}$. Since T is computable, we can compute $t_{s_n{}^\frown 0}$ and $t_{s_n{}^\frown 1}$. We need to compute which of these two strings A extends.

Since T is e-splitting, we know there must be some k such that $\varphi_e^{t_{s_n{}^\frown 0}}(k) \neq \varphi_e^{t_{s_n{}^\frown 1}}(k)$. By searching through all possible k and running times, we can eventually find such a k. Now since $\Phi_e(A) = B$, we can check the kth bit of B to determine whether A extends $t_{s_n{}^\frown 0}$ or A extends $t_{s_n{}^\frown 1}$. □

Now we define an opposite type of notion:

Definition 22.3. Say that a computable perfect tree T is **e-trivial** if there is no two branches $s, t \in T$ and k such that $\varphi_e^s(k)$ and $\varphi_e^t(k)$ both halt and $\varphi_e^s(k) \neq \varphi_e^t(k)$.

We have a corresponding lemma for e-trivial trees.

Lemma 22.4. *Suppose T is a computable perfect e-trivial tree, $A \in [T]$, and $A \geq_T B$ via e, so $\Phi_e(A) = B$. Then B is computable.*

Proof. We claim that we can compute B as follows. To compute $B(k)$, search for some $t \in T$ such that $\varphi_e^t(k)$ halts. We can do this search computably since T is computable, and it must eventually terminate since some initial segment of $s \subseteq A$ makes $\varphi_e^s(k)$ halt, and we have $s \in T$. Finally, the answer we get from $\varphi_e^t(k)$ must be $B(k)$. This is because $\varphi_e^t(k) = \varphi_e^s(k)$ since otherwise we would contradict T being e-trivial. □

Finally, we have the following lemma:

Lemma 22.5. *For all computable perfect trees T and all e, there is a subtree $T' \subseteq T$ such that either T' is e-splitting, or T' is e-trivial.*

Proof. Case 1: Suppose that for every $t \in T$, there exists k and $t', t'' \supseteq t$ such that $\varphi_e^{t'}(k)\downarrow \neq \varphi_e^{t''}(k)\downarrow$. Then we claim that we can find an e-splitting subtree $T' \subseteq T$. We will define the computable $T' \subseteq T$ by defining a sequence $(t_s)_{s\in 2^{<\mathbb{N}}}$ so that $s \subseteq s^*$ implies $t_s \subseteq t_{s^*}$ and for every s, $t_{s^\frown 0}$ and $t_{s^\frown 1}$ are incomparable as usual (so $T' = \{t\colon (\exists s) t \subseteq t_s\}$). Let $t_\emptyset = \emptyset$. Given t_s, define $t_{s^\frown 0}$ and $t_{s^\frown 1}$ as follows: search for k and two extensions $t', t'' \supseteq t$ such that $\varphi_e^{t'}(k)\downarrow \neq \varphi_e^{t''}(k)\downarrow$. This search must eventually terminate by our assumption of Case 1. Then let $t_{s^\frown 0} = t'$ and $t_{s^\frown 1} = t''$.

Case 2: There is some $t \in T$ such that for all k and $t', t'' \supseteq t$, if $\varphi_e^{t'}(k)$ and $\varphi_e^{t''}(k)$ halt, then $\varphi_e^{t'}(k) = \varphi_e^{t''}(k)$. Then let $T' = \{s \in T \colon s \subseteq t \vee s \supseteq t\}$. Clearly T' is computable since T is. □

Definition 22.6. If T is a perfect tree, say t is the **trunk** of T is the splitting node in T of minimal length. That is

Theorem 22.7 (Spector, 1956)**.** *There is an incomputable $A \in 2^{\mathbb{N}}$ so that if $A \geq_T B$, then either $B \equiv_T \emptyset$ or $B \equiv_T A$.*

Proof. We can construct a decreasing sequence $(T_n)_{n\in\mathbb{N}}$ of computable perfect trees where $T_n \supseteq T_{n+1}$ for every n so that:

1. For all e, there is some n such that T_n is e-splitting or e-trivial
2. The lengths of the trunks of the T_n goes to infinity
3. For all computable $B \in 2^{\mathbb{N}}$ there is some n such that $B \neq [T_n]$.

The reason we can construct such a sequence $(T_n)_{n\in\mathbb{N}}$ is that each of these properties is "dense". (1) can be achieved using Lemma 22.5. We can achieve (2) since for every computable perfect tree T, choose any $t \in T$ such that $|t| \geq n$, then $T' = \{s \in T \colon s \subseteq t \vee s \supseteq t\}$ is also a computable perfect tree, and has a trunk of length $\geq n$. We can achieve (3) since for every computable perfect tree T, and every computable $B \in 2^{\mathbb{N}}$ we can find some $t \in T$ that is incompatible with B. Then $T' = \{s \in T \colon s \subseteq t \vee s \supseteq t\}$ does not contain B.

Given such a sequence $(T_n)_{n\in\mathbb{N}}$, let $t_n \in 2^{<\mathbb{N}}$ be the trunk of T_n for each n. Note that $t_n \subseteq t_{n+1}$ for every n. Now if $m \geq n$, then $t_m \in T_m \subseteq T_n$ so $t_m \in T_n$ for every $m \geq n$, and so $A = \bigcup_n t_n$ is in T_n for every n. Note that A is the unique path that is in all these trees; if $A' \neq A$, then A' is incompatible with t_n for some n, and hence $A' \neq T_n$.

We claim that A has a minimal Turing degree. Suppose $A \geq_T B$ via the eth program, so $\Phi_e(A) = B$. Then there is some n such that T_n is either e-splitting, or e-trivial. Since $A \in T_n$, we therefore have that either $B \geq_T A$ by Lemma 22.2, or B is computable by Lemma 22.4. □

Note that we cannot make a minimal Turing degree using Cohen forcing.

Exercise 22.8. If $A \in 2^{\mathbb{N}}$ is 1-generic, then A does not have minimal Turing degree. [Hint: show that if $B \in 2^{\mathbb{N}}$ is the even bits of A, so $B(n) = A(2n)$ for every n, then B is incomputable, but $B \not\geq_T A$].

23 Diagonal incomputability

23.1 DNC functions

Our original proof that that halting problem is incomputable made a function $f\colon \mathbb{N} \to \mathbb{N}$ with the property that for every n, $f(n) \neq \varphi_n(n)$, if $\varphi_n(n)$ is define. Functions with this property are called **diagonally noncomputable**. We investigate these types of functions in this section. Does all incomputability come from this type of diagonally incomputability?

Theorem 23.1. *There is an incomputable $A \in 2^{\mathbb{N}}$ such that A cannot compute any* DNC *function.*

Proof. This is an easy construction by finite initial segments. We claim that for each $e \in \mathbb{N}$, the set of $s^* \in 2^{\mathbb{N}}$ such that there exists an n such that $s^* \Vdash \varphi_e^G(n)\uparrow$ or $s^* \Vdash \varphi_e^G(n) = \varphi_n(n)$ is dense.

Suppose that $s \in 2^{\mathbb{N}}$. We may assume that there for every n there exists some $s^* \supseteq s$ such that $\varphi_e^{s^*}(n)\downarrow$, since otherwise we can extend s to some s^* that forces $s^* \Vdash \varphi_e^G(n)\uparrow$. Then we claim there is some n and $s^* \supseteq s$ such that $s^* \Vdash \varphi_e^G(n) = \varphi_n(n)$. If this was not the case, then there would be a computable DNC function f: to compute $f(n)$, search for some extension $s^* \supseteq s$ such that $\varphi_e^{s^*}(n)$ halts, and let $f(n)$ be the value of $\varphi_e^{s^*}(n)$ for the first such s^* that we find. This contradicts the fact that there is no computable DNC function! □

23.2 DNC$_k$ functions and trees

A function $f\colon \mathbb{N} \to \mathbb{N}$ is DNC$_k$, if $\operatorname{ran}(f) \subseteq \{0, \dots, k-1\}$.

As we will see, DNC$_k$ functions are closely connected with computable trees. In this section we'll prove a few more basic facts about computable trees, and establish some of these connections.

Lemma 23.2 (König's lemma)**.** *$T \subseteq 2^{\mathbb{N}}$ has an infinite path iff T is infinite.*

Proof. If $A \in T$, then the strings $A \upharpoonright n$ are in T for ever n.

Conversely, suppose T is an infinite tree. We will construct an increasing sequence $(s_n)_{n\in\mathbb{N}}$ of strings so that for every n, s_n has infinitely many extensions in T. Then $A = \bigcup_n s_n$ will be in $[T]$.

Let $s_0 = \emptyset$. Inductively, given s_n, since there are infinitely many extensions of s_n, by the pigeonhole principle, there are infinitely many extensions of $s_n{}^\frown 0$ or $s_n{}^\frown 1$. Let s_{n+1} be one of these two strings having infinitely many extensions. □

We have the following corollary of our proof of König's lemma:

Lemma 23.3. *If T is an infinite computable tree, $\emptyset'$ can compute a path in $[T]$.*

Proof. $\emptyset'$ can compute a sequence $(s_n)_{n\in\mathbb{N}}$ as in Lemma 23.2, since given any $s \in T$, there is a program that searches the tree about s, and halts if s only has finitely many extensions (if there is some length $n > |s|$ so that there are no extensions of s in T of length n). □

Lemma 23.4. *If T is a co-r.e. tree, there is a computable tree T' so that $[T] = [T']$.*

Proof. Define T' as follow. To compute whether $s \in T'$, for each $n \leq |s|$, run the enumeration of the complement of T for n steps and see if $s \upharpoonright n$ is enumerated into the complement of T, if so $s \notin T'$. Otherwise, $s \in T'$.

Now $T' \supseteq T$, so $[T'] \supseteq [T]$. We also have $[T'] \subseteq [T]$. Since if $A \notin [T]$, then there is some initial segment $t \subseteq A$ so that $t \notin T$. But then if s is enumerated into the complement of T in n steps, then the initial segment of length $\sup(|s|, n)$ is not in T'. □

Theorem 23.5. *There is a computable tree T so that $[T]$ is exactly the* DNC$_2$ *functions.*

Proof. Let $T = \{s \colon \forall n < |s| s(n) \neq \varphi_n(n)\}$. Then T is clearly a co-r.e. tree, and its paths are exactly the DNC_2 functions. There is a computable tree with the same set of infinite paths by Lemma 23.4. □

Exercise 23.6. Suppose T is a computable tree, and $[T]$ contains exactly one infinite path A. Then show that A is computable.

We now have the following alternative characterizations of DNC_2 functions.

Theorem 23.7. *Suppose $A \in 2^{\mathbb{N}}$. The following are equivalent:*

1. *$A \in 2^{\mathbb{N}}$ can compute a complete consistent extension of* PA*.*
2. *For every computable infinite tree $T \subseteq 2^{\mathbb{N}}$, A can compute a path in $[T]$.*
3. *A can compute a DNC_2 function.*

Proof. (1) ⇒ (2): Given a infinite computable tree T, we can compute an increasing sequence $(s_n)_{n\in\mathbb{N}}$ of elements of T as follows. Construct a program that searches both extensions of $s_n{}^\frown 0$ and $s^\frown 1$, and halts if there is some $m > n$ so that there are no extensions of $s_n{}^\frown 0$ of length m, but there is an extension of $s^\frown 1$ of length m. The question of whether this program halts can be transformed into a Σ_1 sentence in the language of arithmetic.

If the set of extensions of $s_n{}^\frown 0$ is finite, then PA proves that it halts. Taking the contrapositive, then if this sentence is false in the extension of PA, then $s_n{}^\frown 0$ must be infinite, and so let $s_{n+1} = s_n{}^\frown 0$. Otherwise, it the the sentence is true, then let $s_{n+1} = s_n{}^\frown 1$.

(2) ⇒ (3): by Theorem 23.5.

(3) ⇒ (1): Suppose A can compute a DNC_2 function f. We can compute a complete extension of PA as follows. Suppose we have already computed that the sentences $\psi_0, \psi_1, \dots, \psi_n$ are in the theory. Given another sentence ψ, we need to compute whether to add θ or $\neg\theta$ to the theory. Now construct a program e which searches for a contradiction from $\mathsf{PA} + \{\psi_0, \psi_1, \dots, \psi_n\} + \theta$, and if it finds one outputs 0, and also searches for a contradiction from $\mathsf{PA} + \{\psi_0, \psi_1, \dots, \psi_n\} + \neg\theta$, and if it finds one outputs 1. If $f(e) = 0$, then this program doesn't halt outputting 0, and so we can put θ into our theory. Otherwise if $f(e) = 1$, then this program doesn't halt outputting 1 and so we can put $\neg\theta$ into our theory. □

Theorem 23.8. *If $A \in 2^{\mathbb{N}}$ can compute a DNC_k function, then it can compute a DNC_2 function.*

Proof. It suffices to show that if A can compute a DNC_{k^2} function f, then A can compute a DNC_k function. Suppose A computes a DNC_{k^2} function f. Let $\langle \cdot, \cdot \rangle \colon \mathbb{N}^2 \to \mathbb{N}$ be a computable bijection which maps $\{0, \dots, k-1\} \times \{0, \dots, k-1\}$ to $\{0, \dots, k^2 - 1\}$. Now for each programs a and b, we can compute a program c so that $\varphi_c(c) = \langle \varphi_a(a), \varphi_b(b) \rangle$. So let $f_1(a, b)$ and $f_2(a, b)$ be functions so that $f(c) = \langle f_1(a, b), f_2(a, b) \rangle$. So $f_1, f_2 \colon \mathbb{N}^2 \to \mathbb{N}$ are both computable functions, and for every a and b, either

$$f_1(a, b) \neq \varphi_a(a) \text{ or } f_2(a, b) \neq \varphi_b(b).$$

Case 1: For every a, there exists a b with $f_2(a, b) = \varphi_b(b)$. Then we can compute a DNC_k function f as follows. Given any a, we can search for and find such a b. Then set $g(a) = f_1(a, b)$.

Case 2: There is an a so that for all b, $f_2(a, b) \neq \varphi_b(b)$. Then fix such an a, and set $g(b) = f_2(a, b)$ for all b. □

It turns out that we have to break up into cases like this, and we won't be able to computably tell which case we are in. Even though from a DNC_k function we can compute a DNC_2 function, there is not a single algorithm that takes a DNC_k function as input and from it computes a DNC_2 function. That is, there is no **uniform** way of computing a DNC_2 function from a DNC_k function.

Exercise 23.9. There is no program e so that for all DNC_3 functions f, is such that φ_e^f is a DNC_2 function.

24 The structure of the c.e. sets

Post in a famous 1943 address proposed the problem of studying the structure of the Turing degrees of the c.e. sets. At the time, only two such Turing degrees were known: that of $\emptyset$, and $\emptyset'$. The problem of whether there were any others became known as Post's problem. The problem was independently solved by Friedberg and Muchnik in the 1950s, using a technique that is now called a priority argument.

24.1 Post's problem and the Friedberg-Muchnik theorem

Theorem 24.1 (Friedberg-Muchnik)**.** *There are c.e. sets $A, B \subseteq \mathbb{N}$ such that $A \not\geq_T B$ and $B \not\geq_T A$.*

Proof. We describe ways of enumerating such c.e. sets A and B.

We define the requirements

$$R_e : \Phi_e(A) \neq B$$

and

$$Q_e : \Phi_e(B) \neq A.$$

If we satisfy all of these requirements for our c.e. sets A and B, then we will be finished. At times during our construction, there will be conflicts between actions that R_e and Q_e would like to take, so we order our requirements by **priority**. R_0 has highest priority, followed by $Q_0, R_1, Q_1, \ldots$.

We will have a **strategy** for satisfying each requirement. At each point in time a strategy may **restrain** finitely bits of A or B so that they may not be enumerated. Strategies may also request that a bit of A or B be enumerated. If a higher priority strategy requests a bit be enumerated that a lower priority strategy has restrained, then the lower priority strategy will be **injured**, and this bit will be enumerated, and the lower priority strategy will be restarted.

Our strategy for satisfying R_e is as follows. When we first start this strategy, we pick some **witness** $n \in B$ that has not yet been restrained, and then restrain it. At the sth step of the construction, this strategy computes $\varphi_e^{A_s}(n)$ for s steps. If we see this halt and output 0, then we say the strategy **requires attention**. If the strategy is allowed to act, then we enumerate n into B, and restrain all the bits of A used in this computation so that they may no longer be enumerated. (Hence, if this strategy is never injured from this point on, the computation $\varphi_e^{A_r}(n)$ remains the same at all stages $r > s$). We then restart all strategies of lower priority that were using bits of A that we have restrained as witnesses, or have restrained the bit of B that we enumerated.

Our strategy for satisfying Q_e is identical, with the role of A and B switched.

At step s, we run the strategies R_e, Q_e for $e \leq s$. We pick the highest priority strategy that requires attention, and allow it to act.

We claim that all the requirements only act finitely many times, and are eventually satisfied. We prove this by induction. Suppose that s is large enough so that all strategies of higher priority than R_e act before stage s. Then if the strategy for R_e ever sees $\varphi_e^{A_s}(n)$ halt at some stage and equal 0, then the strategy will restrain the bits of A used in this computation, and enumerate the nth bit of B, so $\Phi_e(A) \neq B$. From this point on the strategy will never be injured, since no higher priority strategies will ever injure it. Otherwise, n will never be enumerated, but we never see $\varphi_e^{A_s}(n)$ halt and equal 0. So $\Phi_e(A) \neq B$. In either case, the strategy for R_e will act at most once after stage s, and the requirement will be satisfied. □

Exercise 24.2. Show there is an incomputable c.e. set A so that $A' = \emptyset'$

24.2 The theory of the c.e. degrees*

After Friedberg and Muchnik proved Theorem 24.1, priority arguments opened the door to deeply understanding the structure of the c.e. Turing degrees. Soon, much more complicated priority arguments were found in which some of the strategies would be injured infinitely many times (but they way in which this happens means the corresponding requirements will still be satisfied at the end of the construction). An early theorem proved using these types of infinite injury priority arguments was the following theorem of Sacks:

Theorem 24.3 (Sack's density theorem, 1964)**.** *Suppose $A <_T B$ are c.e. sets. Then there is a c.e. set C so that $A <_T C <_T B$.*

After Sacks's density theorem, Shoenfield conjectured that the r.e. Turing degrees under $\leq_T$ were a dense upper semi-lattice analogously to how the rationals are a dense linear ordered. Shoenfield's conjecture was quickly proved to be false by the following result of Lachlan and Yates that there is a minimal pair in the c.e. degrees.

Theorem 24.4 (Lachlan-Yates, 1966)**.** *There are incomputable c.e. sets A and B such that if $C \leq_T A$ and $C \leq_T B$, then A is computable.*

However, not only is Shoenfield's conjecture is false. It is in some sense as false as possible. Shoenfield's conjecture would imply that the that set of formulas in the language $\{\leq_T\}$ that are true about the c.e. Turing degrees would be computable (just as the theory of dense linear orders has quantifier elimination and is computable). But in fact, the theory of the c.e. degrees is as complicated as possible. Let $\mathcal{R}$ denote the Turing degrees of all c.e. sets.

Theorem 24.5 (Harrington-Shelah 1982)**.** *The first order theory of $(\mathcal{R}, \leq_T)$ is incomputable.*

Indeed, not only is the theory incomputable, it is as incomputable as possible. Each statement about $(\mathcal{R}, \leq_T)$ can be computably transformed into a sentence about the structure $(\mathbb{N}; 0, 1, +, \cdot, <)$. This gives an obvious upper bound to how complicated the theory of the r.e. degrees can be. This upper bound is the best possible:

Theorem 24.6 (Harrington-Slaman and Slaman-Woodin)**.** $\mathrm{Th}(\mathcal{R}, \leq_T)$ *is computably isomorphic to* $\mathrm{Th}(\mathbb{N}; 0, 1, +, \cdot, <)$.

In more computably-theoretic terms, these theories are computably isomorphic to $\emptyset^\omega = \bigoplus_n \emptyset^{(n)}$.

The type of phenomenon seen in Theorem 24.6 has become a common theme in computability theory. The structures studied in computability theory often end up being as complicated as possible.

25 Complexity theory*

25.1 $\mathsf{TIME}(f)$, P, and the time hierarchy theorem*

Computational complexity theory studies how efficiently computable sets can be computed. The sets considered are typically subsets of $2^{<\mathbb{N}}$ instead of $\mathbb{N}$, since binary strings is the format in which most models of computer more naturally work. A subset of $2^{<\mathbb{N}}$ is called a **decision problem** or **language** in complexity theory (an unfortunately conflicting terminology with languages in model theory).

Definition 25.1. If $f\colon \mathbb{N} \to \mathbb{N}$ is a function, a language $A \subseteq 2^{<\mathbb{N}}$ is computable in $O(f)$-time, if there is some Turing machine program so that for all strings $s \in 2^{<\mathbb{N}}$, when this machine is run with s as input, it halts in $O(f(|s|))$ steps and then halts outputting whether $s \in A$. $\mathsf{TIME}(f)$ is set of all languages computable in $O(f)$ time.

For every $f\colon \mathbb{N} \to \mathbb{N}$, $\mathsf{TIME}(f)$ is a countable set of languages, and the union of $\mathsf{TIME}(f)$ over all f is equal to the set of all computable languages. (Which is also equal to the union of $\mathsf{TIME}(f)$ over all computable f).

A very important complexity class is P: the class of problems which can be solved in polynomial time.

Definition 25.2. $\mathsf{P} = \bigcup_{k\in\mathbb{N}} \mathsf{TIME}(n^k)$.

One reason that the class P is nice is that it has many desirable closure properties. For example, since the polynomials are closed under multiplication and composition, polynomial-time algorithms that use polynomial-time computable subroutines are also polynomial time computable. If a language A is polynomial time computable, then any language B computable in polynomial time using the oracle A is also in P.

Some examples of interesting problems in P are deciding whether a number is prime (by the AKS theorem), linear programming (by a theorem of Khachiyan in 1979), and whether a graph has an Euler cycle.

Polynomial-time computability is regarded as containing the class of problem that can be efficiently solved in the real world. One reason for this is that there are many commonly used n^2-time and n^3-time algorithms that are often used in the real world, and making a good theory of efficient computations requires us to have closure under the types of multiplication and composition described above. Another reason that poly-time computability is natural is that different models of computer (the Turing machine model, Turing machines with several tapes or 2-dimensional tapes, random access machines, etc.) may take different amounts of time to compute some given computable language. However, all natural models of computer can simulate each other in polynomial time, so the model of computation that we use does not matter if we just care about whether a problem is poly-time computable. The idea that all natural models of computation that can be made in the real world can be simulated in polynomial time on a Turing machine is sometimes called the extended Church-Turing thesis. Compelling evidence for it is that a Turing machine can be modeled inside classical physics, and likewise classical physics can be simulated in polynomial time on a Turing machine.[8]

The time hierarchy theorem adapts the proof of the undecidability of the halting problem to show that as f grows, the class $\mathsf{TIME}(f(n))$ also grows. Here we need a technical assumption that f is time constructible. A function $f\colon \mathbb{N} \to \mathbb{N}$ is **time constructible** if $f(n) \geq n$ for all n, and there is a Turing machine that computes $f(n)$ when given a string of n ones in time $f(n)$.

Theorem 25.3. *Suppose $f\colon \mathbb{N} \to \mathbb{N}$ is time constructible. Then* $\mathsf{TIME}(f) \subsetneq \mathsf{TIME}(f(n)^2)$.

[8] Quantum computers, however, are believed to be able to not be able to be simulated on classical computers, though this is an open.

Proof. Consider the language $A = \{s\colon$ s codes a Turing machine program M and an integer k so that the program M halts in at most $kf(|s|)$ steps$\}$. Assuming that f is time constructible, and there is a universal Turing machine that runs in quadratic time, it is easy to see A is computable in $O(f(n)^2)$ time.

The language A is not computable in $O(f(n))$ time by the usual diagonalization argument. Suppose there was a program than ran in $O(f(n))$ time that computed A. Then using the recursion theorem (which can be implemented in n^2 time assuming there is a universal Turing machine that runs in n^2 time), make a Turing machine that runs this program to determine whether it will hall, and then do the opposite. □

So for example, if $\mathsf{EXPTIME} = \bigcup_k \mathsf{TIME}(2^{n^k})$, then $P \subsetneq \mathsf{EXPTIME}$ since $\mathsf{TIME}(p) \subseteq \mathsf{TIME}(2^n)$ for every polynomial p, since 2^n is time constructible and grows faster than any polynomial, and $\mathsf{TIME}(2^n) \subsetneq \mathsf{TIME}(2^{n^2})$.

25.2 NTIME(f), NP, and the Cook-Levin theorem*

Definition 25.4. A language $A \subseteq 2^{<\mathbb{N}}$ is in $\mathsf{NTIME}(f)$ if there is a f-time computable set $B \subseteq 2^{<\mathbb{N}} \times 2^{<\mathbb{N}}$ and a function $g \in O(f)$ so that

$$s \in A \leftrightarrow (\exists t)[|t| \leq g(|s|) \wedge (s,t) \in B]$$

The string t is often called a **witness** or **certificate** that $s \in A$.

Definition 25.5. $\mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}(n^k)$

Some example of problems in NP include deciding whether a graph is 3-colorable, or deciding whether a boolean formula has a satisfying assignment. We can easily check whether a purported coloring or satisfying assignment really is one in polynomial time, and the 3-colorable graphs/satisfiable formulas are those having such a coloring/satisfying assignment.

If we think of P as being analogous to all computable sets, then NP is analogous to all c.e. sets. Indeed, one can similarly make a version of the arithmetical hierarchy called the polynomial hierarchy, where computability is replaced with poly-time computability. For example, a language A is Σ^{P}_2 if there is a language B in P and a polynomial p such that

$$s \in A \leftrightarrow (\exists t_1, |t_1| \leq p(|s|))(\forall t_2, |t_2| \leq p(|s|))[(s,t_1,t_2) \in B]$$

Just as we compare the relative computability of problems in computability theory by Turing reducibility, we can compare the relative complexity of problems in complexity theory by polynomial-time computable many-one reducibility noted $\leq^{\mathsf{P}}_m$ (sometimes called Karp reductions), or polynomial-time Turing reducibility noted $\leq^{\mathsf{P}}_T$ (sometimes called Cook reductions). So for example, $A \leq^{\mathsf{P}}_T B$ if there is an algorithm which computes A in polynomial time using B as an oracle.

Just as we know of many complete c.e. sets, Cook and Levin showed in the early 1970s that many natural problems are NP complete under $\leq^P_m$. That is, any problem in NP is poly-time many-one reducible to them. For example, boolean formula satisfiability is NP complete. To see this, given any language A in NP, we show we can map each $s \in 2^{<\mathbb{N}}$ to a formula which is satisfiable iff $s \in A$. This formula will contain variables that describe each of the bits of a certificate t, variables giving the state of a Turing machine at each point in time, and variables for the symbol written on each cell of the Turing machine at time. The natural formula stating that the rules of the Turing machine are follows and halts accepting its input is poly-time computable. This formula is satisfiable iff $s \in A$. Graph 3-coloring can be shown to be NP complete by reducing boolean satisfiability to it.

Some of the very basic facts about computability theory and complexity theory mirror each other via the analogy described above between computable and polynomial-time computable. However, this analogy quickly stops being helpful. The question analogous to whether there is an incomputable c.e. set is open:

Open Problem 25.6. *Is* $\mathsf{P} = \mathsf{NP}$*?*

One might hope now to copy the same proof that the halting problem is incomputable. However, the language containing the pairs (M, p) where M is a Turing machine program that halts in time $p(|M|)$ is not naturally in NP; it is in $\mathsf{EXPTIME}$ (and is essentially how we separate P and $\mathsf{EXPTIME}$ using the time hierarchy theorem). More generally, the theory of NP has many provable difference from the theory of the Σ_1 sets. For example, it is easy to show there is *not* a universal language in NP under polynomial-time reducibility. This follows from a version of the TIME hierarchy theorem for NTIME.

Exercise 25.7. Suppose c, d are integers and $c > d$. Then show $\mathsf{NTIME}(n^c) \subsetneq \mathsf{NTIME}(n^d)$.

It is widely believed that $\mathsf{P} \neq \mathsf{NP}$. There are thousands of known NP complete problems that have been studied in many forms in many different fields of mathematics for thousands of years. In not a single case has there been any real better algorithm found for solving them in general than going through every possible certificate t, and checking whether it is valid. In contrast, most other problems in these fields which are not known to be NP complete like primality checking, linear programming, graph isomorphism, etc., it is very often the case that we have often found deep mathematical structures hidden in them that make solving them much easier, even if they are not known to be in P (e.g. it is still open whether graph isomorphism is in P).

The P vs NP question is one of the most important open problems in mathematics. Is there hidden structure in the thousands of NP complete problems that we find in math that we can use to understand and quickly solve all of them? Because the problem of whether we can efficiently understand and compute solutions to problems in mathematics is a vital part of almost every field, it touches on almost all of modern mathematics.

25.3 The relativization barrier*

There are theorems explaining why the P vs NP problems is quite hard, and cannot be solved using simple diagonalization tricks like those used to separate the computable and c.e. sets is the following theorem:

Theorem 25.8 (Gill, Baker, and Solovay, 1970s)**.** *There is an oracle* $A \subseteq 2^{<\mathbb{N}}$ *so that* $\mathsf{P}^A \neq \mathsf{NP}^A$*. There is also an oracle* $B \subseteq 2^{<\mathbb{N}}$ *so that* $\mathsf{P}^B = \mathsf{NP}^B$*.*

Proof sketch. A language that is always in NP^A is the set of strings s so that the oracle A contains a string of length $|s|$. The certificate t is just the string of the same length as s that is in the oracle A (which can be verified by one query to A). However, we can easily construct an oracle A so that this problem is not poly-time computable relative to A. Any polynomial-time algorithm can only check a polynomial-time number of strings of length $|s|$ to see if they are in A, before it makes its decision. We define A so that all the strings that it checks of this length are not in A. Once it halts, we can then put a string of this length into A that it has not checked to diagonalize if it says there is a string of this length, and otherwise leave out all strings in A of this length.

The construction of an oracle B so that $\mathsf{P}^B = \mathsf{NP}^B$ is similarly easy. We code the answers to problems in NP^B into strings of larger polynomial length so that they are polynomial-time computable relative to B. □

Thus, any proof technique that relativizes cannot solve the P vs NP question. This basically rules out all the ideas and techniques from computability theory. Indeed, modern complexity theory has very little to do with modern computability theory, though these two fields have the same origins. Computability theorists and complexity theorists don't publish in the same journals, attend the same conferences, or talk to each other very often. Modern combinatorics, algebraic geometric, number theory, functional analysis, etc. are more closely connected to complexity theory than logic or computability. These fields all contain problem whose computational complexity is of huge importance. Likewise, mathematical progress on them on them leads to breakthroughs in computational complexity theory. For example, the theory of expander graphs in combinatorics has become hugely important in theoretical computer science and is connected to derandomization, probabilistically checkable proofs, etc.

26 Computing with real numbers*

The study of computability on the real numbers goes back to Turing, who used real numbers instead of infinite binary sequences in his famous 1936 paper.

26.1 Representations of real numbers and functions on the reals*

A little bit of care needs to be taken in how we represent real numbers in an appropriate way for doing computability theory. For example, suppose we want a function $f\colon 2^{\mathbb{N}} \to \mathbb{R}$ which maps infinite binary sequences to real numbers that they represent. Any reasonable such function must be continuous: once we know a large finite amount of information about an infinite binary sequence representing a real number, we should be able to determine this real number within some small ϵ. However, this implies that f cannot be total and injective. The space $2^{\mathbb{N}}$ is compact, and any injective continuous function from a compact space to a Hausdorff space is a homeomorphism onto its image. However, $2^{\mathbb{N}}$ is zero dimensional it has a basis of clopen sets. Hence, $2^{\mathbb{N}}$ cannot be homeomorphic to $\mathbb{R}$, or any intervals (a, b) or $[a, b]$, since there is not a basis of clopen sets for these spaces; clopen sets in these spaces are trivial. Thus, we cannot have an injective way of representing real numbers.

We are used to such duplicate representations of real numbers: for example, in decimal, the number 1 has two representations: 1, and $0.999\overline{9}\ldots$. However, decimal representations (or more generally representations in any base) are a poor choice of representation for computability theory:

Proposition 26.1. *There is no computer program which takes as input two programs for computing decimal representation of reals numbers, and outputs a program computing a decimal representation of their sum.*

Proof. Suppose we had such a program $\varphi_n(a, b)$. Now we construct two programs for computing real numbers that this procedure does not correctly add. Our first program computes $0.50000\ldots$, and keeps appending 0s until we decide otherwise. Our second number begins $0.49999\ldots$ and keeps appending more 9s to this number until we decide otherwise. By the recursion theorem, we may watch the computation of the sum of these two numbers. If their sum begins 1.something, then we keep making our first number equal to 0.5, but we make our second number less than 1/2 by making all further digits zeroes: $0.49999\ldots0000\ldots$, so this program computes their sum incorrectly. If the program outputs that the sum is 0.something, then we make our first number greater than 0.5, and our second number equal to 1/2 by having the remaining digits be 9s forever: making the decimal representation $0.4999\overline{9}\ldots$, so this program also computes their sum incorrectly. If the program computing their sum never outputs the first decimal digit, then it also not a correct computation of their sum. □

Essentially the problem is in decimal, we are forced to declare whether any number is ≥ 1 or ≤ 1 (or more generally, $\geq \frac{m}{10^n}$ or $\leq \frac{n}{10^n}$ for any integers m and $n > 0$. Instead, we'll take the following approach:

Definition 26.2. A **representation of a real number** x is a sequence of rational numbers $(a_n)_{n\in\mathbb{N}}$ so the $x = \lim_n a_n$ and so $|x - a_n| \leq \frac{1}{n}$ for all n. A real number x is **computable** if it has a computable representation.

The bound $\frac{1}{n}$ is not really important here; we could replace it with any computable sequence (y_n) of positive real numbers so that $\lim_n y_n = 0$. There are other reasonable ways of representing real numbers. For instance, a representation could be a sequence $[b_n, c_n]$ of closed intervals whose lengths go to 0, which represent $x = \lim b_n = \lim c_n$. Or, we could use functions $\lambda\colon \mathbb{Q}^+ \to \mathbb{Q}$, which represent the real number x where $|x - \lambda(\epsilon)| < \epsilon$ for each rational $\epsilon > 0$. Each of these types of representations can be computably turned into each of the other types. They key point is that in an representation of a real number, we want to be able to obtain each rational ϵ, an approximation of the number x within epsilon.

Suppose a number x is the limit of a computable sequence of rational numbers $x = \lim_n a_n$ (but with no bound on how far a_n is from x). Then it is not necessarily the case that x is computable.

Definition 26.3. A real number x is **left-computably enumerable** if there is a computable increasing sequence of rational numbers $(b_n)_{n\in\mathbb{N}}$ so that $x = \lim_n b_n$. x is **right-computably enumerable** if there is a computable decreasing sequence of rational numbers $(c_n)_{n\in\mathbb{N}}$ so that $x = \lim_n c_n$.

Proposition 26.4. *There is a left-c.e. real number that is not computable.*

Proof. Let A be any c.e. set that is not computable. Then let x be the number $\sum_{n\in A} \frac{1}{2^n}$, whose nth binary digit encodes the nth bit of A. If $(A_s)_{s\in\mathbb{N}}$ is a computable enumeration of A, then the sequence of rational numbers $(a_s)_{s\in\mathbb{N}}$ where $a_s = \sum_{n\in A_s} \frac{1}{2^n}$ are computable, and $\lim_n a_n = x$, so x is left-c.e. However, if we were able to compute x up to an error of $\frac{1}{2^{n+1}}$, we would be able to compute the first n bits of A. (Note that A is not eventually 0 or eventually 1). Hence, x is not computable. □

Proposition 26.5. *If a real number x is left-c.e. and right-c.e., then x is computable.*

Proof. Let $(b_n)_{n\in\mathbb{N}}$ be a computable increasing sequence of rationals so that $\lim_n b_n = x$, and let $(c_n)_{n\in\mathbb{N}}$ be a computable decreasing sequence of rational numbers so that $\lim_n c_n = x$. Then $x \in [b_n, c_n]$ for each x. So To obtain some a_m so that $|x - a_m| \leq \frac{1}{m}$, compute b_n and c_n until we find some large enough n so that $|b_n - c_n| \leq \frac{1}{m}$. Then let $a_m = b_n$. □

Exercise 26.6. If x is computable, then it is left-c.e. and right-c.e.

We note that it is incomputable to determine whether a computable real number is ≥ 0. This is related to un-suitableness of the representation of number by its decimal expansion:

Proposition 26.7. *There is a computable function $f\colon \mathbb{N} \to \mathbb{N}$ which maps each n to a program computing a representation of a real number so that $\varphi_n(n)\downarrow$ if and only if the real number computed by $f(n)$ is ≥ 0.*

Proof. The Cauchy sequence computed by $f(n)$ is

$$a_k = \begin{cases} 0 & \text{if } \varphi_n(n) \text{ does not halt in } \leq k \\ -\frac{1}{s} & \text{if } \varphi_n(n) \text{ halts in } s \leq k \text{ steps} \end{cases}.$$

□

Now we have the following definition of a computable function on $\mathbb{R}$.

Definition 26.8. A function $f\colon \mathbb{R}^k \to \mathbb{R}$ is **computable** if there is a computable function φ_e which given representations of real numbers $x_1, \ldots, x_k$ as oracles, computes a representation of $f(x_1, \ldots, x_k)$.

Note that the function φ_e must correctly compute $f(x_1, \ldots, x_k)$ given *any* representations of $x_1, \ldots, x_k$.

For example, the step function $f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$ is *not* computable because of Proposition 26.7. We cannot compute whether a representation of a real number represents a number ≥ 0. Indeed, every computable function must be continuous.

Proposition 26.9. *If f is computable function from $\mathbb{R}^k \to \mathbb{R}$, then f is continuous.*

Proof. For notational convenience, we'll prove the theorem for $k = 1$. The proof for $k > 1$ is identical.

We need to show that for ever x, for every $\epsilon > 0$, there exists a $\delta > 0$ so that $|x_0 - x| < \delta$ implies $|f(x) - f(x_0)| < \epsilon$. Take any representation $(a_n)_{n\in\mathbb{N}}$ of x so that $|a_n - x| \leq \frac{1}{2n}$ for each n. For any $\epsilon > 0$, the function computing f must compute finitely many elements of a representation of $f(x)$ which specify its value within $\epsilon/2$ using only finitely many elements of the sequence $(a_n)_{n\in\mathbb{N}}$. Let k be the largest element of the sequence $(a_n)_{n\in\mathbb{N}}$ that the computation uses. Then any x_0 with $|x_0 - x| \leq \frac{1}{2k}$ has a representation which begins $a_0, \ldots, a_k$, hence such a representation of x_0 will be mapped to a representation of a real number within ϵ of $f(x)$. □

Theorem 26.10. *There is a computable function $f\colon \mathbb{N} \to \mathbb{N}$ which maps each n to a program computing a representation of a real number so that $\varphi_n(n)\downarrow$ if and only if the real number computed by $f(n)$ is ≥ 0.*

Exercise 26.11. Show that addition, multiplication, subtraction, and the exponential function, are all computable.

Theorem 26.12 (Kleene)**.** *If $f\colon \mathbb{R}^k \to \mathbb{R}$ is continuous, there is an oracle $A \in 2^N$, so that f is computable relative to A.*

Proof. The oracle A should record for each rational interval $[a, b]$, representations of the endpoints c, d of the closed interval $[c, d] = f([a, b])$. □

Kleene's theorem suggests a way of studying analysis and functions on the real numbers: stratify them by how complicated they are to compute, and try to prove theorems about them by leveraging these types of computable representations. We will see some examples of theorems proved by a fine grained analysis of this type in later sections.

26.2 The incomputability of the derivative*

Recall that the sup norm $\|f\|_\infty$ of a function f is $\|f\|_\infty = \sup_{x\in\mathbb{R}} |f(x)|$.

We begin with an easy lemma:

Lemma 26.13. *Suppose $(f_n)_{n\in\mathbb{N}}$ is a uniformly computable sequence of continuous functions, and $\|f_n\|_\infty \leq \frac{1}{2^n}$ for all n. Then $\sum_n f_n$ is computable.*

Proof. To compute $\sum_n f_n(x)$ within $\frac{1}{2^k}$ compute each of $f_0(x), \ldots, f_{k+1}(x)$ within $\frac{1}{(k+2)2^{k+1}}$. Then since $\left\|\sum_{n>k} f_n(x)\right\| \leq \frac{1}{2^{k+1}}$ this approximation of $f_0(x) + \ldots + f_{k+1}(x)$ is within $\frac{1}{2^k}$ of $f_0(x), \ldots, f_n(x)$. □

Note that in this lemma we could replace the series $\frac{1}{2^n}$ with any summable series whose partial sums are uniformly computable.

Theorem 26.14 (Myhill)**.** *There is a differentiable computable function whose derivative is continuous but not computable.*

Proof. Let $f_{n,s}\colon \mathbb{R} \to \mathbb{R}$ be the piecewise linear function:

$$f_{n,\epsilon}(1/n) = \begin{cases} 0 & \text{if } x \leq \frac{1}{n} - \epsilon \\ \frac{\frac{1}{n} - x}{\epsilon} + 1 & \text{if } \frac{1}{n} - \epsilon < x \leq \frac{1}{n} \\ \frac{x - \frac{1}{n}}{\epsilon} + 1 & \text{if } \frac{1}{n} < x \leq \frac{1}{n} + \epsilon \\ 0 & \text{if } x > \frac{1}{n} - \epsilon \end{cases}$$

which is a triangle with height 1 and width 2ϵ reaching its maximum at the point $\frac{1}{n}$, so if $F_{n,\epsilon}(x) = \int_0^x f_{n,\epsilon}(t)\,\mathrm{d}t$, then $\|F_{n,\epsilon}\| = \epsilon$.

Let A be an incomputable c.e. set, and

$$f = \sum_{\{(n,s):n\in A_s\setminus A_{s-1}\}} \frac{1}{n} f_{n,1/2^s}$$

Now f is a continuous function, and if we could compute f from some oracle B, we could also compute A from B by determining whether $f(1/n)$ is equal to 0 or $1/n$.

However, the antiderivative $F(x) = \int_0^x f(t)\,\mathrm{d}t$ of f is computable since the functions $F_{n,1/2^s}$ are uniformly computable, and their norms satisfy the hypotheses of Lemma 26.13. (Identifying this sequence indexed by $\mathbb{N}^2$ with a sequence indexed by $\mathbb{N}$ using a computable bijection between $\mathbb{N}^2$ and $\mathbb{N}$). □

27 Kolmogorov complexity*

27.1 Kolmogorov complexity and failure of subadditivity*

Kolmogorov complexity goes back to the work of Solomonoff, Kolmogorov, and Levin in the 1960s and 1970s. Suppose $M\colon 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ is a partial computable function (sometimes called a **machine** in the literature. If $M(s) = t$, we can think of string s as a M**-description** of the string t. We define the shortest length of an M-description of t as:

$$C_M(t) = \min\{|s| \colon M(s) = t\}.$$

In everyday life, we are used to shortening descriptions of strings in this way. Files on computers are long binary strings t, and often we would like to "compress" these files and store them in a shorter form. We are familiar with many algorithms of this sort like the Lempel-Ziv-Welch compression algorithm which is used in the ZIP and GIF file formats.

Proposition 27.1. *For every n, there is some string t of length n so that $C_M(t) \geq |t|$.*

Proof. By the pigeonhole principle. There are only $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ strings of length less than n, but there are 2^n strings of length n. □

We say that a partial computable function $U\colon 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ is **universal** if for every computable $M\colon 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$, there is a constant c so that for every t, $C_U(t) \leq C_M(t) + c$. For example, consider the machine U which on input $0^e{}^\frown 1^\frown s$ runs the eth machine on the input s (where 0^e is the string of e zeroes). Then for the eth machine M_e, $C_U(t) \leq C_{M_e}(t) + e + 1$ for every t, since if s is an M_e-description of t, then $0^e{}^\frown 1^\frown s$ is a U-description of t.

In the theory of Kolmogorov complexity, we are typically only interested in results up to an additive constant. If f, g are functions to $\mathbb{R}$, we write $f \leq^+ g$ to note that there is some c so that for all x, $f(x) \leq^+ g(x)$. So if U is universal, then for every machine M, $C_U \leq^+ C_M$, and we let $C(t)$ denote $C_U(t)$ for some universal machine U, whose value is well defined up to an additive constant (which suffices for most theorems studying infinite families of binary strings).

For example,

Proposition 27.2. *$C(t) \leq^+ |t|$ for all t.*

Proof. If $M\colon$ *strings* $\to 2^{<\mathbb{N}}$ is the identity function, then for all t, $C(t) \leq^+ C_M(t) = |t|$. □

Now Kolmogorov complexity has some unfortunate drawbacks, chief among them is that it fails to be subadditive. Let $\langle \cdot, \cdot \rangle\colon 2^{<\mathbb{N}} \times 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ be a computable bijection coding pairs of strings using single strings. We would like it to be true that $C(\langle s, t\rangle) \leq^+ C(s) + C(t)$: the shortest way to describe two strings s and t the sum of the shortest descriptions of each of them. Unfortunately this is badly false. We can't concatenate a description of s and a description of t because we will not be sure where the description of s ends, and the description of t starts.

Theorem 27.3. *There is a constant c so that for arbitrarily large n, there are strings s, t so that $|s^\frown t| = n$, and $C(\langle s, t\rangle) \geq C(s) + C(t) + \log_2 n + c$.*

Proof. Note that $C(\langle s, t\rangle) \geq C(s^\frown t) + c_1$ for some c_1, since given any universal machine U, there is a machine M which takes a U-description of a string $\langle s, t\rangle$ as input, and then outputs $s^\frown t$. We'll show the stronger theorem that there are strings s, t of arbitrarily length $|s^\frown t| = n$ such that

$$C(s^\frown t) \geq C(s) + C(t) + \log_2 n + c$$

The basic idea of the proof is to take a long string w that has no short description (by Proposition 27.1), and then find a way of breaking it up into two strings s and t so that $w = s^\frown t$ where the length of s can be used as extra information to give a shorter description of s than just $|s|$.

By Proposition for each d, we can find a string w of length $n = 2^{d+1} + d$ such that $C(w) \geq |w|$.

Let $L\colon \mathbb{N} \to 2^{<\mathbb{N}}$ take a number n to the nth binary string in lexicographic order, by length. Let M be a machine which on input r, output $L(|r|)^\frown r$. Let k be such that $L(k)$ is the first d bits of w. Note that $\sum_{i<d} 2^i = 2^d - 1 \leq k \leq 2^{d+1} - 1$. Let r be string of length k that is the dth through $d+k$th bits of w, and let s be the string of length $d+k$ that is the first $d+k$ bits of w. So $M(r) = s$. Then let t be the remaining bits of w, so $s^\frown t = w$.

Then there is a constant c_2 so that $C(s) \leq C_M(s) + c_2 \leq k + c_2 = |s| - d + c_2$, and $C(t) \leq |t| + c_3$ by Proposition 27.2. So $C(s^\frown t) \geq n$, and $C(s) + C(t) + \log_2(n) + c \leq n$. □

27.2 Prefix-free Kolmogorov complexity*

Prefix free Kolmogorov complexity fixes the failure of subadditivity that we saw with C in Theorem 27.3. In many ways prefix-free Kolmogorov complexity has nicer properties that make it better for studying the interplay between randomness, dimension, and complexity.

Definition 27.4. Say that a function $M\colon 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ is **prefix-free** if for all $s \subseteq s^*$, if $s \in \mathrm{dom}(M)$ and $s^* \in \mathrm{dom}(M)$, then $s = s^*$.

If M is a partial computable function that is prefix-free, then we use the notation K_M instead of C_M to denote the corresponding notion of Kolmogorov complexity.

$$K_M = \min\{|s|\colon M(s) = t\}.$$

Just as with C, for K there are **universal** prefix-free partial computable functions $U\colon 2^{<\mathbb{N}} \to 2^{<\mathbb{N}}$ so that for any prefix-free partial computable function M, we have that $K_U(t) \leq^+ K(t)$. For example, consider the machine U which on input $0^{e\frown}1^\frown s$ starts computing M_e on all possible strings as input. (Where 0^e is the string of length e of all zeroes). Then if we every see that $M_e(s)$ halts and outputs t, then the machine U also outputs t, provided there has not already seen a string s^* that $M_e(s^*)$ halts, where $s* \subseteq s$ or $s \subseteq s^*$. Clearly U will be prefix-free and if M_e is any partial computable function that is prefix free, then $K_U(t) \leq K_{M_e}(t) + e + 1$.

One use of Kolmogorov complexity is to characterizing randomness in computability theory.

Theorem 27.5. *Suppose $A \in 2^{\mathbb{N}}$ is an infinite binary string. Then the following are equivalent.*

- *For all n, $K(A \upharpoonright n) \geq^+ n$. (The initial segments of A have highest possible Kolmogorov complexity and are as difficult as possible to describe.*
- *For every uniformly c.e. set of strings S_n, where $\sum_{s\in S_n} \frac{1}{2^{|s|}} \leq \frac{1}{2^n}$, $A \notin \bigcap \bigcup_{s\in S_n} N_s$. (A is not an any nullset that can be described in an uniformly c.e. way).*
- *For every martingale $B\colon 2^{<\mathbb{N}} \to [0,\infty)$ so $B(s^\frown 0) + B(s^\frown 1) = 2B(s)$, that is uniformly left c.e., $\limsup_n B(A \upharpoonright n) < \infty$. (No c.e. betting strategy succeeds on A).*

Proof. See [?]. □

A infinite binary string A that has these equivalent properties is called Martin-Löf random.

28 Kolmogorov complexity and Hausdorff dimension*

28.1 Hausdorff dimension*

Hausdorff dimension is a fractal dimension notion for subsets of $\mathbb{R}^n$. In a s-dimension space $\mathbb{R}^s$, a ball of radius r has volume $c_s r^s$ where c_s is a constant[9]. Lebesgue measure in dimension s measures a subset of $\mathbb{R}^s$ by the smallest measure of an open cover of s. By carefully packing balls, it is straightforward to show that this is equal to the inf over $\sum_i c_s r_i^s$ where $(B_{r_i}(x_i))_{i\in\mathbb{N}}$ is an open cover of E by open balls. Note that we can make the sup of the radiuses $\sup_i r_i$ arbitrarily small here and still be arbitrarily close to this infimum.

Suppose we have a set $E \subseteq \mathbb{R}^n$ which is also a subset of an s-dimensional space. Then in $\mathbb{R}^n$ we can take the same covers of E by balls, and show that the inf of $\sum_i c_s r_i^s$ over all covers by open balls $(B_{r_i}(x_i))_{i\in\mathbb{N}}$ of E (where now $B_{r_i}(x_i)$ is the ball in n-dimensional space) correctly gives the s-dimensional Lebesgue measure of E.

Using this idea, we make a general definitions of this type for any real number $s > 0$. If $E \subseteq \mathbb{R}^n$, $s \geq 0$, and $\delta > 0$ then define:

$$H^s_\delta(E) = \inf\{\sum_{i\in\mathbb{N}} r_i^s \colon (B_{r_i}(x))_{i\in\mathbb{N}} \text{ is cover of } E \text{ by open balls with radiuses } r_i < \delta\}$$

We then define the s-dimensional Hausdorff outer measure $H^s(E)$ to be:

$$H^s(E) = \lim_{\delta\to 0^+} H^s_\delta(E)$$

This quantity is always defined since as $\delta \to 0^+$, $H^s_\delta(E)$ is nondecreasing. Finally, we define $\dim_H(E) = \inf\{s > 0\colon H^s(E) = 0\}$. We drop the constant c_s in the definition of $H^s_\delta(E)$ since we are interested in finding the smallest dimension s in which $H^s(E)$ is zero (and so the constant c_s does not change this).

A **Kakeya set** in $\mathbb{R}^n$ is a set containing a unit line segment in every direction.

Open Problem 28.1 (The Kakeya problem)**.** *Suppose $E \subseteq \mathbb{R}^n$ contains a line segment in every direction. Must $\dim_H(E) = n$?*

The Kakeya problem has connections to a surprising number of other fields of mathematics. It is related to geometric measure theory, harmonic analysis, and arithmetic combinatorics. Variants and generalizations in other contexts such as over finite fields also have applications to problems in computer science and randomness extraction.

In this section, we will give an argument of Lutz and Lutz that the $n = 2$ case of the Kakeya problem has a positive answer, using computability theory.

28.2 Effective dimension and the point-to-set principle*

We can measure the Kolmogorov complexity of rational numbers $q \in \mathbb{Q}$ via some coding of rational numbers via binary sequence. Using this notion, we can define the Kolmogorov complexity of real numbers at some given precision r. If $x \in \mathbb{R}^n$ and $r \in N$, then define

$$K_r(x) = \min\{K(q)\colon q \in Q^n \cap B_{2^{-r}}(x)\}$$

to be the Kolmogorov complexity of x at precision r. For example, if $x \in \mathbb{R}$, then $K_r(x)$ is roughly the shortest description of the first r bits of x in binary.

We now define the effective dimension of $x \in \mathbb{R}^n$ as follows.

[9] c_s is equal to $\frac{\pi^{s/2}}{\Gamma(s/2+1)}$ where Γ is Euler's gamma function

$$\dim(x) = \liminf_{r\to\infty} \frac{K_r(x)}{r}$$

So if $\dim(x) \leq \alpha$, for $x \in \mathbb{R}^n$ then for arbitrarily large k, we can find descriptions of the first k digits of each of the n coordinates of x where the description has length $\approx \alpha x$.

Exercise 28.2. Show that $0 \leq \dim(x) \leq n$ for every $x \in \mathbb{R}^n$.

Exercise 28.3. Show that if $x \in \mathbb{R}^n$ is computable then $\dim(x) = 0$.

Exercise 28.4. The set of $x \in \mathbb{R}^n$ such that $\dim(x) < n$ is a Lebesgue null set. [Hint: for every r, c, the set of x such that $K_r(x) \leq r - c$ has measure at most 2^{-c+1}. Now use the Borel-Cantelli lemma.

One can show (see [?]) that for every $0 \leq \alpha \leq n$ there are uncountably many $x \in \mathbb{R}^n$ such that $\dim(x) = \alpha$.

We can similarly define the dimension of x relative to an oracle $A \in 2^{\mathbb{N}}$ by relativizing the definition of Kolmogorov complexity to A. Using this notion, we have the following point-to-set principle for Hausdorff dimension:

Theorem 28.5 ([LL18])**.** *For every set $E \subseteq \mathbb{R}^n$*

$$\dim_H(E) = \min_{A\in 2^{\mathbb{N}}} \sup_{x\in E} \dim^A(x)$$

Proof sketch. To show $\dim_H(E) \geq \min_{A\in 2^{\mathbb{N}}} \sup_{x\in E} \dim^A(x)$ we can construct an oracle A from a countable sequences open covers used to witness the value of $\dim_H(E)$. Relative to this oracle, any $x \in E$ can be described by what elements of the cover it belongs to. This shows that $\dim_H(E) \geq \dim^A(x)$.

Now given any $A \in 2^N$ we can show that $\dim_H(E) \leq \sup_{x\in E} \dim^A(x)$. To do this, take the descriptions used relative to A to describe element of x as x varies of all of E. One can show this gives a sequence of open covers of E showing its Hausdorff dimension is at most $\sup_{x\in E} \dim^A(x)$.

For details, see [LL18, Theorem 1]. □

The utility of this theorem is that in order to prove a lower bound $\dim_H(E) \geq \alpha$, it suffices to show that for every $A \subseteq \mathbb{N}$ and every $\epsilon > 0$, there is a point $x \in E$ such that $\dim^A(x) \geq \alpha - \epsilon$.

28.3 The Kakeya problem in dimension 2*

If $t, r \in 2^{<\mathbb{N}}$ are strings, then the **conditional Kolmogorov complexity of t given r**, noted $K(t|r)$, is defined by

$$K(t|r) = \inf\{|s| \colon U(\langle s, r\rangle) = t\}$$

where U is a universal machine. Informally, it is the shortest description of t, given that we already have the information r given to us.

For example, there is a constant c so that for all $t \in 2^{<\mathbb{N}}$, $K(t|t) \leq c$. We can similarly define conditional complexity for real numbers, given some finite data. If $x \in \mathbb{R}^n$, and $q \in \mathbb{R}^m$ is rational, then

$$K_r(x|q) = \min\{K(p|q) \colon p \in \mathbb{Q}^n \cap B_{2^{-r}}(x)\}$$

and if $y \in \mathbb{R}^m$, then

$$K_{r,s}(x|y) = \min\{K_r(x|q) \colon q \in Q^m \cap B_{2^{-s}}(y)\}.$$

and we use the notation $K_r(x|y)$ to denote $K_{r,r}(x|y)$.

Now we have the following technical lemmas:

Lemma 28.6.

$$K_r(x,y) = K_r(x|y) + K_r(y) + o(r)$$

Proof sketch. $K(s,t) =^+ K(s) + K(t|s^*)$. See [?LV, Theorem 3.9.1]. This is called the chain rule for prefix-free Kolmogorov complexity. The lemma follows easily. □

Using this, it is not difficult to prove the following:

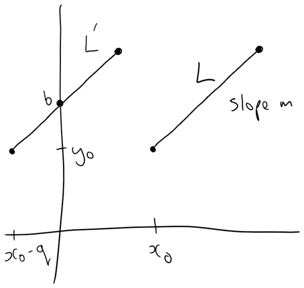
Lemma 28.7. *Suppose $m \in [0,1]$ and $b \in \mathbb{R}$. Then for a.e. $x \in [0,1]$ (wrt Lebesgue measure),*

$$\liminf_{r\to\infty} \frac{K_r(m,b,x) - K_r(b|m)}{r} \leq \dim(x, mx+b)$$

We can now use this machinery to give a solution to the Kakeya problem in dimension 2.

Theorem 28.8 (Davies [D71])**.** *Suppose $E \subseteq \mathbb{R}^2$ is a Kakeya set. Then E has Hausdorff dimension $\dim_H(E) = 2$.*

Proof. Let A be as in Theorem 28.5 so that $\dim_H(E) = \sup_{x\in E} \dim^A(x)$. Let $m \in [0,1]$ be such that $\dim^A(m) = 1$. (Such an m exists since almost every m (wrt Lebesgue measure) has this property). Let L be a unit line segment contained in E of slope m. Let (x_0, y_0) be the left endpoint of L. Now choose a rational number q so that the unit line segment L' of slope m starting at $x_0 - q$ has a y intercept b.



We need to find some point $(x,y) \in L$ such that $\dim^A(x,y) = 2$. We know that for a.e. $x \in [0,1/2]$ we have

$$\dim^{A,m,b}(x) = 1 \qquad (*)$$

(by Exercise 28.4) and

$$\liminf_{r\to\infty} \frac{K_r(m,b,x) - K_r(b|m)}{r} \leq \dim^A(x, mx+b) \qquad (**)$$

(by the relativized version of Lemma 28.7). So choose an x with both properties (*) and (**). Note that $(x, mx+b) \in L'$ and $(x+q, mx+b) \in L$, and $\dim^A(x+q, mx+b) = \dim^A(x, mx+b)$. So it suffices to show $\dim^A(x, mx+b) = 2$.

$$\begin{aligned}
\dim^A(x, mx+b) &\geq \liminf_{r\to\infty} \frac{K_r^A(m,b,x) - K_r^A(b|m)}{r} && \text{by } (**) \\
&= \liminf_{r\to\infty} \frac{K_r^A(m,b,x) - K_r^A(b,m) + K^A(r)(m)}{r} && \text{by Lemma 28.7} \\
&= \liminf_{r\to\infty} \frac{K_r^A(x|b,m) + K_r^A(m)}{r} && \text{by Lemma 28.7} \\
&\geq \liminf_{r\to\infty} \frac{K_r^{A,b,m}(x)}{r} + \frac{K_r^A(m)}{r} \\
&= \dim^{A,b,m}(x) + \dim^A(m) && = 2
\end{aligned}$$

by our choice of m and x. □

References

[AB] S. Arora and B. Barak, *Computational complexity: a modern approach*, Cambridge University Press (2009).

[B66] R. Berger, *The undecidability of the domino problem*, Mem. Amer. Math. Soc. **66** No. 72 (1966).

[C00] M. Coste, *An Introduction to Semialgebraic Geometry*, Dottorato di Ricerca in Matematica, Istituti Editoriali e Poligrafici Internazionali, Pisa (2000)

[D71] R. O. Davies, *Some remarks on the Kakeya problem*, Proc. Cambridge Phil. Soc., **69** (1971) 417–421.

[DRS] B. Durand, A. Romashchenko, A. Shen, *Fixed-point tile sets and their applications*, J. Comput. Syst. Sci. **78** No. 3 (2012), 731–764.

[G31] K. Gödel *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme, I*, Monatshefte für Mathematik und Physik, **38** no. 1 (1931) 173–198. `https://doi.org/10.1007/BF01700692`

[Hig61] G. Higman, *Subgroups of finitely presented groups.*, Proceedings of the Royal Society of London, series A, mathematical and physical sciences, (1961) vol. 262, 455–475.

[Hil88] D. Hilbert, *Über die Darstellung definiter Formen als Summe von Formenquadraten.* Math. Ann. **32** (1888), no. 3, 342-350.

[Hil93] D. Hilbert, *Über tenäre definite Formen.* Acta Math. **17** (1893), no. 1, 169–197.

[JR] E. Jeandel and M. Rao, *An aperiodic set of 11 Wang tiles*, Advances in Combinatorics, 2021:**1**. `https://doi.org/10.19086/aic.18614`

[JM91] J. P. Jones and Y. V. Matijasevic, *Proof of recursive unsolvability of Hilbert's tenth problem*, Amer. Math. Monthly **98** (1991), no. 8, 689–709. `https://doi.org/10.1080/00029890.1991.11995778`

[K91] R. Kaye, *Models of Peano Arithmetic*, Oxford Logic Guides, Oxford University Press, Oxford. 1991.

[LV] LV] M. Li and P. Vitányi, *An introduction to Kolmogorov complexity and its applications*, Springer, New York, (2008).

[L55] M.H. Löb, *Solution of a Problem of Leon Henkin*, J. Symb. Logic **20** (1955), 115-118.

[LL18] J.H. Lutz and N. Lutz, *Algorithmic information, plane Kakeya sets, and conditional dimension*, ACM Transactions on Computation Theory **10** (2018), article 7.

[LS] Lyndon and Schupp, *Combinatorial Group Theory*, Springer-Verlag, 1977.

[MW96] A. Macintyre, and A.J. Wilkie, On the decidability of the real exponential field, in Kreiseliana, P. Odifreddi ed. (1996) 451–478.

[Mar96] D. Marker, *Model theory and exponentiation*, Notices Amer. Math. Soc, **43**, (1996) 753–759.

[Mi08] A. Miller, *The Recursion Theorem and Infinite Sequences* arXiv: https://arxiv.org/abs/0801.2097.

[Mot67] T.S. Motzkin, *The arithmetic-geometric inequality* Inequalities (Proc. Sympos Wright-Patterson Air Force Base, Ohio, 1965) 205–224 Academic Press, New York.

[PH77] J. Paris and L. Harrington, *A Mathematical Incompleteness in Peano Arithmetic.* In Barwise, J. (ed.). Handbook of Mathematical Logic. Amsterdam, Netherlands: North-Holland. `https://doi.org/10.1016/S0049-237X(08)71130-3`

[P] B. Poonen, *Undecidable Problems: A sampler*, In J. Kennedy (Ed.), Interpreting Gd̈el: Critical Essays (2004), pp. 211-241. Cambridge: Cambridge University Press. `https://doi.org/10.1017/CBO9780511756306.015`.

[PZ] F. Pfender and G. Ziegler, *Kissing numbers, sphere packings, and some unexpected proofs*, Not. Amer. Math. Soc., **51** No 8 (2004) 873–883.

[Sch12] K. Schmüdgen, *Around Hilbert's 16th problem.* Doc. Math. 2012 433-438.

[Tar67] A. Tarski *The completeness of elementary algebra and geometry*, Centre National de la Rechershe Scientifique, Institut Blaise Pascal, Paris 1967, iv + 50pp.

[We] H. Weber, *Leopold Kronecker* Mathematische Annalen, **43** (1893) 1–25, doi:10.1007/BF01446613.