Chapter 7
Number Theory and Correctness of Programs: Incompleteness

Natural numbers, endowed with addition and multiplication, is one of the most ubiquitous structures in mathematics and our society. The reason it's so useful is that it can model so many things in our world, both the physical/natural world and the human created world, especially those that involve *discrete* objects. If I was a single lone jellyfish in deep sea where everything was murky and there was nothing discrete to discern, I may have less use for natural numbers. But we live in a world that is full of discrete objects and others that can be discretized by approximating them. For example, people are discrete objects; it's useful to know how many children one has, or how many people can vote in a country. Goats and cows are discrete, and important in early notions of wealth and trade. Time is not discrete, but we can discretize time into intervals, like seconds, and hence use numbers to count time. Planetary positions are not discrete, but can be discretized to arcs of degree, and hence modeled as natural numbers. With discretization, a significant aspect of the physical world can be modeled as numbers, and most *observations* of physical phenomena can be modeled using numbers.

The theory of numbers is hence ubiquitous and studied extremely well in mathematics. We learned how to represent them succinctly and how to do operations on them (using n-ary representations, which need a symbol for 0, and algorithms for computing operations on them). Much of elementary school is devoted to learning these simple algorithms.

Programs are also very much discrete in nature. They deal with inputs that are sequences, which can be seen as numbers, they do operations, which can be seen as similar to operations involving numbers (arithmetic/Boolean circuits), and the program itself, is a sequence of symbols that can be seen as numbers.

Consider an imperative program (choose your favorite programming language) with *assertions*. Assertions are basically properties of states that you assert in code, and are a form of *specification* asserting that the property must be true in all states where the assertion is reached. A program with assertions is said to be *correct* if in every execution of the program, whenever an assertion is reached, the asserted property holds.

The problem of *program verification* is to determine whether a given program with assertions is correct. The statement that a program P with assertions is correct, is really a theorem in mathematics. And a *proof* of such a theorem, no matter what the notion of proofs are, is a mechanically checkable sequence of statements, where it should be clear that the proof asserts in the end that the program is correct. There are several proof systems that prove programs correct— *Hoare logic* is a popular one. But you can imagine other formal arguments/proofs of why a program satisfies a particular assertion.

Incompleteness results in logic argue that there are no formal systems that can prove all theorems in certain models or classes of models. In other words, for certain models or classes of models, *not all theorems have proofs, in any proof system*.

In my view, there are at least *three* incompleteness results that are astonishing: (a) theorems expressed in FOL over natural numbers with addition and multiplication (Gödel's incompleteness theorem), (b) theorems expressed in FOL about the class of all finite structures (or even all finite graphs), and (c) theorems about correctness of programs.

In this book, we show that none of the above problems is r.e., which is another way of saying there is no formal proof system that contains proofs for these theorems. The incompleteness of all three problems are non-intuitive— it seems intuitive that all FO theorems about numbers ought to have proofs, that all FO properties of finite graphs ought to be provable, and all correct programs should have some proof of correctness. It's incredible that they don't. We intrinsically believe that all theorems ought to be provable, but incompleteness argues that this is not possible, at least not in any formal proof system. It shows that formal systems are intrinsically weak!

Incompleteness results put mathematics in a strange place than we intuitively imagined. There may be true theorems that may not even be provable using the formal rules of proof we accept. It opens up the possibility that theorems in number theory (including open problems) may not even have proofs. And similarly theorems about finite graphs. And similarly, the correctness of certain programs.

We proved the incompleteness of FO theorems of finite structures in Chapter 4 (Trakhtenbrot's theorem). We will show the other two incompleteness results in this chapter.

The incompleteness results all have a similar proof outline based on some form of *diagonalization*, similar to the one found by Cantor, and similar to the one used by Turing to show the undecidability of the halting problem. Since, as computer scientists, we already know of such results, in particular that the halting problem for Turing machines (or programs) is undecidable and the *non-halting* problem of Turing machines (or programs) is not even recursively enumerable, we will use these to prove our results.

7.1 Program Verification

Consider a TM that we want to check for non-halting. The TM can be realized by a program P (any programming language with infinite memory will do, as it can simulate the moves of a Turing machine; for example, a program with access to unbounded linked lists, or a program with access to an unbounded secondary storage device, or even a program that has unbounded integers). So the problem reduces to checking whether P does not halt.

Construct a program P' that is basically the program P modified so that if P halts, we add an assertion `assert false;` at the point where it halts. (An assertion of *false* doesn't hold in any program state— if you are uncomfortable with it, replace it with $x := 1; assert x = 0;$.)

Now it is clear that the program P' satisfies its assertion if and only if P does *not* halt. Consequently, program verification is not recursively enumerable.

Theorem 7.1 *Program verification is not a recursively enumerable problem.*

Let us now use the above result to prove Gödel's incompleteness theorem stated in terms of the theory of numbers not being recursively enumerable.

7.2 Incompleteness of the theory of natural numbers with additional and multiplication

We want to show that the theory of natural numbers with addition and multiplication is not recursively enumerable, i.e., $Th(\mathbb{N}, 0, 1, +, \times, =, \leq)$ is not recursively enumerable.

The intuition behind this result is that the theorem stating that a program is correct (or that Turing machines halts or does not halt) *is a first-order expressible theorem in number theory*! Consequently, there is no formal proof system such that all theorems in FO arithmetic have proofs in the system.

We want to reduce the problem of Turing machine non-halting (or halting[1]) to the validity problem of sentences over the theory of natural numbers. The following proof is adapted from Dexter Kozen's book "Automata and Computability".

First, we can express several interesting properties using first-order logic using addition and multiplication:

- q is the quotient and y is the remainder when x is divided by y:

$$IntDiv(x, y, q, r) : \quad x = qy + r \wedge r < y$$

- y divides x:

$$Div(y, x) : \quad \exists q.\ IntDiv(x, y, q, 0)$$

[1] Note that reducing the halting problem to validity also works to show non r.e.-ness since the theory is negation-complete; a negation conmplete theory is either decidable or not r.e.

- x is prime:

$$Prime(x): \quad x \geq 2 \wedge \forall y.\ (Div(y,x) \Rightarrow (y = 1 \vee y = x))$$

- y is a power of a particular fixed prime p, i.e., $y = p^k$ for some $k \in \mathbb{N}$:

$$Power_p(y): \quad \forall z.((Div(z,y) \wedge Prime(z)) \Rightarrow z = p)$$

We will now show a reduction from the non-halting problem of a Turing machine (on an empty tape) to validity of arithmetic sentences.

Given a Turing machine M with tape alphabet Γ and states Q, let us fix the alphabet $\Pi = \Gamma \cup (Q \times \Gamma)$. Let us choose a prime p larger than Π, and let us look upon sequences over Π as p-ary representations of numbers. A sequence $a_n \ldots, a_0$, where each $a_i \in [0, p-1]$ maps to the number $\Sigma_{i \in [1,n]}\ a_i p^i$.

The *computation* of M on the empty tape can be seen as a sequence of configurations $\sigma_0, \sigma_1 \ldots,$, where each σ is a configuration represented a word in Π^*. When M halts, configurations are bounded by some maximum length (depending on how much space M takes on the tape. Let H denote the subset of Π that has the halting state: $H = \Gamma \times HQ$, where $HQ \subseteq Q$ are the halting states. Hence M halts iff there is a sequence of configurations that represents valid moves of M that has the halting configuration, i.e., where some element of H occurs.

We now encode the halting of M as the existence of a number whose p-ary representation encodes a valid halting computation of M.

A finite computation sequence $\sigma_0, \sigma_1, \ldots, \sigma_n$ will be encoded as large enough blocks so that each σ_i fits into a block. If C is a large enough length to encode each configuration, we will use $c = p^C$ to capture this number. (In general, most numbers k related to the Turind machine that we need will be captured using p^k instead of k.) This number c will eventually be quantified in the formula we reduce to.

In order to say that a configuration sequence is correct, it is sufficient to demand that successive configurations are correct. In order to demand $\sigma, sigma'$, two successive configurations (encoded with sequence of the same length C) are correct, it is sufficient to check every *three-element* subsequence of σ with the corresponding three-element subsequence in σ' (since Turing machines make only local changes on the tape). The three element sequences either does not encode a state (i.e., is over Γ only), in which they must be the same, or the three element sequence in σ encodes a state in the middle, in which case the corresponding three-element sequence in σ' depicts the correct evolution according to the transitions of the Turing machine.

Let V be the set of all 6-tuples $(a_1, a_2, a_3, b_1, b_2, b_3)$ that denote valid pairs of three-tuples. V includes all:

- Every $(a_1, a_2, a_3, b_1, b_2, b_3)$ such that $a_1, a_2, a_3, b_1, b_2, b_3 \in \Gamma$
- For every transition $\delta(q,a) = (b, q', R)$, the triples $(a_1, (q,a), a_2, a_1, b, (q', a_2))$, $((q,a), a_1, a_2, b, (q', a_1), a_2)$, and $(a_1, a_2, (q,a), a_1, a_2, b)$ are in V, for every $a_1, a_2 \in \Gamma$.
- For every transition $\delta(q,a) = (b, q', L)$, the triples $(a_1, (q,a), a_2, (q', a_1), b, a_2)$, $(a_1, (q,a), a_3, (q', a_1), b, a_3)$,

are in V, for every $a_1, a_3 \in \Gamma$.

Note that check inconsistencies of sequences only in tuples where the "middle" symbol in the first configuration, i.e., a_2, encodes a state.

We will write a formula that ensures that in a number encoding sequences of configurations, for every two consecutive configuration σ and σ', every three-element subsequence in σ and the corresponding three-element subsequence in σ', the 6 elements are related by V. This will ensure that the entire sequence of configurations is valid.

The crucial power of arithmetic with addition and multiplication is that we can encode sequences as numbers, and also *decode* sequences into their components. Here is an important formula, which says that the character in position Y of a sequence encoded by the number v is a (where $a \in [0, p-1]$. As we said before, we encode the position Y using the number $y = p^Y$. So the following really says that the position encoded in y of the sequence encoded by v is b (assuming y is a power of p):

$$Digit(v, y, a) = \exists u.\ \exists r.\ (v = r + ay + upy \wedge r < y \wedge a < p)$$

Intuitively, let's say v's p-ary representation can be split into $\rho_1 \cdot a \cdot \rho_2$, where $|\rho_2| = Y$. Then clearly $v = r + ap^Y + up^{Y+1}$, for some $r < y$ (where r encodes the number corresponding to ρ_2 and u encodes the number corresponding to ρ_1). Replacing p^Y by y gives $v = r + ay + upy$, which is what the above formula demands.

The intuition for the following formulae follow along similar lines, and we let the reader work this out for themselves.

We can demand that the 3-digit sequence of v at positions encoded by y are b_1,b_2, and b_3, using the formula:

$$3Digit(v, y, b_1, b_2, b_3) : \exists u.\ \exists r.\ (v = r + b_1 y + b_2 py + b_3 ppy + uppy$$

$$\wedge\ r < y\ \wedge\ b_1 < p\ \wedge\ b_2 < p\ \wedge\ b_3 < p)$$

Now we can demand that the three digits of v at the position encoded by y match correctly the three digits of v at the position encoded by z:

$$Match(v, y, z) : \bigvee_{(a_1,a_2,a_3,b_1,b_2,b_3) \in V} (3Digit(v, y, a_1, a_2, a_3) \wedge 3Digit(v, z, b_1, b_2, b_3))$$

We can now write a formula that says that the p-ary string that represents v encodes a valid sequence of configurations of the TM evolution. For technical reasons, we will parameterize this with c and d— c is the number that encodes the size of configurations (i.e., p raised to the power of the length of configurations) and d will encode a bound on the entire length of the sequence v. The formula checks whether all pairs of three-digit sequences precisely c apart (or rather $log_p(c)$ apart) in v match according to the Turing machine's moves, up to d:

$$ValidMoves(v, c, d) : \forall y.(Power_p(y) \wedge yppc < d) \Rightarrow Match(v, y, yc)$$

We can state the sequence representing v starts with the initial configuration. Let *init* be the number encoding the symbol $(q_0, \#)$, the Turing machine reading the blank symbol. Let *blank* denote the number encoding the blank symbol #. Note that the start configuration is then $(q_0, \#) \cdot \# \cdot \# \ldots \cdot \#$. The following formula forces this as the first configuration of v:

$$Start(v, c) : Digit(v, 1, init) \wedge \forall y.\ \big(Power_p(y) \wedge y > 1 \wedge y < c \Rightarrow Digit(v, y, blank)\big)$$

We can also state that the halting configuration occurs in v before d by the formula:

$$Halt(v, d) : \exists y.\left(Power_p(y) \wedge y < d \wedge \bigvee_{b \in H} Digit(v, y, b)\right)$$

We can now ready to write a formula that says that v is a valid sequence of configurations that halts. In order to do this, we first express that the number d represents an upper bound on the length of v (we then interpret v using the p-ary representation of v, with 0's padded to the left, if necessary):

$$Length(v, d) : Power_p(d) \wedge v < d$$

Note that v along with d (where $Length(v, d)$ holds) represents the precise p-ary sequence we wish to express properties about. We can now write that this sequence represents a valid halting computation:

$$ValidHaltComp(v) : \exists d.\ \exists c.\ (Length(v, d) \wedge Power_p(c) \wedge\ c < d$$

$$\wedge\ Start(v, c) \wedge ValidMoves(v, c, d) \wedge Halt(v, d))$$

We can now finally write a sentence that says that the Turing machine M does *not* halt:

$$\neg \exists v.\ ValidHaltComp(v)$$

The above formula is valid over the standard model of natural numbers with addition and multiplication iff the Turing machine does not halt. The relation $<$ can be expressed with the other relations using the following equivalence:

$$x < y \Leftrightarrow \exists z. \neg(z = 0) \wedge x + z = y$$

We hence have:

Theorem 7.2 *The first-order theory of natural numbers with addition and multiplication,* $Th((\mathbb{N}, 0, 1, +, \times, =))$ *is not recursively enumerable.*

7.3 Further Remarks

Gödel's proof and strengthenings

The crux of the above proof is that sequences over an alphabet, related in simple syntactic ways (like the moves of a Turing machine) can be encoded in arithmetic. Gödel was the first to discover this, and he used it in what's called Gödel numbering in order to encode *proofs* into numbers; proofs are also sequences whose validity is syntactic. This was done before a solid notion of computation (such as Turing machines or lambda calculus) existed. In fact, Gödel showed that one can encode a self-referential formula, where for any reasonable proof system, one can state in arithmetic a statement that says: "There is no proof of this statement." A proof system is damned it it proves this statement, and damned if it does not. If it proves it, then it's proved a wrong statement! And if it doesn't prove it, it's a correct statement it cannot prove! Gödel's proof combines encoding proofs/sequences as numbers and a diagonalization argument. In our proofs, we proved undecidability and hence non-r.e.-ness of Turing machine non-halting using diagonalization, and a separate proof of encoding existence of sequences into statements about numbers.

Note that the above proof extends beyond first-order arithmetic. Any logic that is more powerful than first-order arithmetic does not have a complete proof system. In fact, it turns out that the above theorem can be strengthened to show that even *quantifier-free arithmetic with addition and multiplication* (i.e., implicitly universally quantified) is undecidable and not recursively enumerable. In fact, the even simpler problem of solving Diophantine equations (given a set of polynomial equations, deciding whether there is solution using integer values) is undecidable, and checking whether there is no solution to them is not r.e.. This is a celebrated problem, called Hilbert's Tenth Problem, which was open for a long time and settled in a famous theorem by Yuri Matiyasevich in 1970.

Axiomatizations

Due to the *completeness theorem*, we know that any model whose theory is axiomatizable is *decidable*. Since the theory of arithmetic with addition and multiplication is undecidable, it follows that there can be no recursive axiomatization of it.

The Peano axioms formulated in first-order logic was an attempt to axiomatize arithmetic. It has an *infinite* set of axioms, including an axiom schema for induction, which essentially says that any first-order property about numbers (formulated as a formula with a single free variable) can be proved by induction. However, as we know from the results in this section *and* the completeness theorem, this axiom system, if sound, must be incomplete. In fact, there are natural *concrete* theorems that can be stated in FOL that are not provable in Peano arithmetic (see the results of Paris and Harrington where a version of Ramsey's theorem is shown to be unprovable in

Peano arithmetic). The *Principia Mathematica* is another formal system for which the incompleteness theorem applies, showing that there can be no recursive of it that is consistent and complete.

Returning to Program Verification: The Method using Invariants

Consider the problem of program verification again. How do people actually prove programs correct (partially correct, i.e., satisfy their assertions) in practice? The predominant method is the *invariant* method, which is basically a proof by induction. People postulate essentially a set of configurations $Inv(\overline{x})$ (called an invariant), captured as a formula in logic over a set of variables $\overline{x}$, and prove the following properties about it:

- The initial states are contained in *Inv*:

$$\forall \overline{x}.Init(\overline{x}) \Rightarrow Inv(\overline{x})$$

- If $Post(\overline{x}, \overline{x}')$ represents how the program can change configurations in a single step, then the invariant is closed under *Post*:

$$\forall \overline{x}, \overline{x}' : (Inv(\overline{x}) \wedge Post(\overline{x}, \overline{x}')) \Rightarrow Inv(\overline{x}')$$

- The invariant set and the set of *unsafe* states where the assertion is violated, do not intersect:

$$\forall \overline{x}(Inv(\overline{x}) \Rightarrow \neg Unsafe(\overline{x}))$$

Once we postulate such an invariant set, program verification boils down to proving validity of the above assertions! Clearly, if there is such an invariant set that contains all the initial states and closed under any move done by the program, then the set contains all the reachable states, and since it doesn't intersect the unsafe sets, it satisfies the assertion.

Verification methodologies such as that of Floyd and Hoare are essentially stylized proof techniques that follow the above method (expressing invariants at only loop headers or method boundaries). In fact, these stylistic methods break down (become too weak) for complex programs (such as concurrent programs or programs that pass programs/program-pointers as parameters); however, the global invariant method above is still robust and always is viable.

Now, one could ask whether such an invariant always exists. It clearly does whenever the program is correct— choose the invariant to be the set of *all* reachable states of the program: it satisfies all the above requirements.

So then where exactly lies the problem in not being able to prove a program correct? It lies in two aspects: (a) invariants may exists but may not be *expressible* in logic, and (b) invariants may be expressible in logic, but there may be no *proofs* (in a fixed formal system) that the above formulas are valid, i.e., the logic used to express the above conditions may be incomplete.

It turns out that either can happen. If we choose a weak enough logic, say a decidable logic, then we would be able to decide validity of the above formulas... but the invariant may not be expressible in logic. However, it turns out that for any reasonable programming language, the *invariant* (or the precise set of reachable states) is *always* expressible in a powerful enough logic, such as arithmetic with addition and multiplication! However, then, the logic becomes incomplete, and there may be no proofs for proving the above properties. And hence program verification, in general, remains incomplete either way. In automated program verification, one either chooses a weaker logic and builds automated decision procedures to check the properties above (this is good for shallow properties), or chooses a very expressive logic, and builds incomplete automation for logical reasoning to find proofs that establish the above properties!

Several open mathematical problems can be reduced to program verification. Consider Goldbach's conjecture, which asserts that all even integers larger than two is a sum of two primes. This problem can be reduced to program verification. Build a program that enumerates all even integers in increasing order, check if they are a sum of two primes (which can be done since both primes obviously need to be less than the considered number), and if some number isn't expressible as a sum of two primes, halt (or assert false). Then this program does not halt (or is correct) iff Goldbach's conjecture is true.

It turns out that there are similar reductions reducing the Riemann hypothesis to program verification/non-halting [**?**], even with a focus on building relatively small TMs.

Do Theorems have Proofs?

The incompleteness theorems raise metamathematical concerns. Is it possible that theorems may not have proofs? In any consistent formal system that is at least as expressive as arithmetic with addition and multiplication, the incompleteness theorem argues that there must be a statement (even expressible in FOL) such that either it or its negation is not provable. The methods mathematicians use is after all some formal system, which if consistent, will have unprovable theorems.

For example, one could take any of the various unsolved problems in mathematics or theoretical computer science, and ask whether the problem is *independent* of the formal systems we are assuming.

Now one could ask whether this theorem potentially has no proof.