Chapter 13:

Sneak Peek at Mathematical Logic II: Gödel's Incompleteness Theorem

The journey towards the formalization of all of Mathematics has two legs: understanding the logic by which we formulate proofs, and understanding the basic axioms from which said proofs can derive all of the truths of Mathematics. In this book we completely figured out the first leg: we have a small number of simple logical axiomatic schemas and inference rules that allow us to syntactically prove anything that follows semantically from any set of axioms!

But what about the other leg, that of choosing the "correct" set of axioms for the whole of Mathematics? As we have briefly mentioned in Chapter 10, the "usual" set of axioms accepted by mathematicians is called **ZFC**: Zermelo–Fraenkel Set Theory with the axiom of Choice. All of the "usual" mathematics that you have learned is ultimately proved from these axioms: analysis, algebra, combinatorics, everything in this book, the theory of computation, and so on.

But, do these axioms suffice for proving *all of Mathematics*? What does that even mean? If not, maybe we need to add a single extra axiom or axiom schema? A few extra axioms or axiom schemas? The attempt to find a set of axioms that would suffice for settling any mathematical question was a central challenge during the beginning of the 20^{th} century. Let us make this question more concrete.

1 Complete and Incomplete Theories

Consider the group axioms that we have seen in chapter 10:

Group Axioms:

- Zero Axiom: $0 + x = x$
- Negation Axiom: $-x + x = 0$
- Associativity Axiom: $(x + y) + z = x + (y + z)$

predicates/some_proofs.py

```
#: The three group axioms
GROUP_AXIOMS = frozenset({'plus(0,x)=x', 'plus(minus(x),x)=0',
                          'plus(plus(x,y),z)=plus(x,plus(y,z))'})
```

Beyond the basic symbols of Predicate Logic, the "language" used in these axioms contains the binary function symbol $+$ (or alternatively in our programs, the binary function name 'plus'), the unary function symbol $-$ (or alternatively in our programs, the unary function

name 'minus'), and the constant name '0'. Now we can ask a question: do these axioms suffice for answering *any question* in this language? I.e., suppose that we have some sentence, e.g. '∀x∀y[x+y=y+x]'. Is it necessarily true that one can either prove or disprove it from our given axioms?

Definition (Complete Set of Axioms)**.** A set of axioms A is **complete** if for any predicate-logic sentence in its language (i.e., that only uses the function names, relation names, and constant names that appear somewhere in A) we have either $A \vdash \phi$ or $A \vdash$ '$\sim\phi$'.

Note that we have used the term "complete" here in a totally different sense than in the "Completeness" Theorem that was the main focus of this book. The Completeness Theorem (whether for Predicate Logic or for Propositional Logic) talks about completeness of a *proof system*, which asks whether a proof system can syntactically prove everything that is semantically correct. Completeness of *a set of axioms* in the sense just defined above talks about whether an axiomatic system can answer (positively or negatively) every mathematical question in its language. It is worthwhile to note that using the Completeness Theorem for Predicate Logic, we can equivalently replace the condition of "either $A \vdash \phi$ or $A \vdash$ '$\sim\phi$' " in the above definition of the completeness of an axiomatic system by "either $A \models \phi$ or $A \models$ '$\sim\phi$'. "

Back to the group axioms, it is not difficult to see that neither '∀x∀y[x+y=y+x]' nor its negation is provable from the group axioms. Why? because some groups are commutative (e.g., the real numbers) and some groups are not commutative (e.g., permutation groups[1]). I.e., we have two models, one in which '∀x∀y[x+y=y+x]' holds and the other in which its negation holds. So since the answer is not fully determined semantically by the group axioms, then by the Soundness Theorem for Predicate Logic it follows that these axioms can neither syntactically prove it nor its negation.

On the other hand, suppose that we *add* to the group axioms another axiom that states that there are exactly two elements: '(∃y[~y=0]&∀x∀y∀z[((x=y|y=z)|x=z)])', and get the axiomatic system for groups with two elements. It turns out that now we can answer not only the question of commutativity but also *every* question. How do we know this? Again we can go through the semantics and observe (using just a little bit of thought) that there is only a single possible model of a group with two elements (that is, in a course on Algebraic Structures we would say that up to trivial degrees of freedom such as renaming the elements, there is only one group with two elements), and thus by the Completeness Theorem for Predicate Logic everything that is true in that group is provable from the axioms for a group with two elements. Thus, since for any statement, either it or its negation is true in that group, one of these is provable from the four axioms for a group with two elements (i.e., the three group axioms and the above additional two-element axiom), and so this set of axioms is complete—the theory of groups with two elements is complete (but also quite boring).

So some sets of axioms are complete and others are not complete. While we are perfectly happy with certain theories not being complete—indeed, field theory would be quite boring if each statement held either in all fields or in no field—we would very much like other theories to be complete. For example, since "in our minds" it is clear what the natural numbers are, we would want to axiomatize the natural numbers in a way that completely determines every truth about them—to find a set of axioms for the theory of the natural numbers so that given any sentence about the natural numbers, either this sentence or its negation follows from these axioms. One of the foundational challenges faced by early

[1]For more details, you are highly encouraged to take a course on Algebraic Structures.

20^{th}-century mathematicians and logicians was indeed to find a complete set of axioms for the language of natural numbers that has the constant name '0', the unary successor function name 's', and the binary function symbols + and · (multiplication), and more generally, to find a complete set of axioms for "all of Mathematics," say for the language of Set Theory that only has the binary relation symbol '$\in$'. More concretely: Is Peano Arithmetic complete? Is ZFC itself complete?

Gödel's **Incompleteness Theorem**, to which this chapter is dedicated, gives a resounding *negative* answer: not only is Peano Arithmetic *not* complete for the language of natural numbers and ZFC is *not* complete for the language of Set Theory, but these deficiencies *cannot* be fixed: for any additional axioms (or axiom schemas) that we add to any of these axiomatic systems, provided that it is possible to tell whether a formula is a valid axiom, the enlarged set of axioms remains incomplete! In fact, any sound axiomatic system that is sufficiently powerful, is necessarily incomplete. Thus, the quest to find the "ultimate" set of axioms for Mathematics is inherently doomed to failure.

So for the rest of this chapter let us fix some sound **"ultimate" axiomatic system (UAS)** that is at least as powerful as Peano Arithmetic in the sense that it can prove the axioms of Peano Arithmetic.

```
#: An ultimate axiomatic system; can you find such a set of schemas?
UAS = frozenset({···})
```

We will prove that this "ultimate" axiomatic system still cannot be complete.

2 Gödel Numbering

Gödel's main ingenious "trick" was to let the axiomatic system "talk about itself." But since Peano Arithmetic talks about natural numbers, Gödel's first order of business was to encode formulas, proofs, etc., as natural numbers. In this chapter we will want our formulas to talk about **programs**, which are technically not really that far from logical formulas, so we will encode these instead. Here the program can be given in any general purpose ("Turing complete") language, e.g., a Python program given as a string.[2] Thus we can formalize this as the following programming "tasks":[3]

```
def encode_program(program: Program) -> int:
    """Encodes the given program as a natural number.

    Parameters:
        program: program to encode.

    Returns:
        A natural number encoding the given program.
    """
```

[2]One language for programs, which may seem less intuitive for programmers at first sight, but which is designed to capture any program in a theoretically simple way that allows for easy analysis, is to describe each program as a **Turing machine**, named after the English mathematician, computer scientist, cryptanalyst, philosopher, and theoretical biologist Alan Turing, who is widely considered to be the father of theoretical computer science and artificial intelligence and who played a pivotal role in cracking Nazi cyphers during World War II, but who was persecuted during his lifetime due to his homosexuality, eventually leading to his tragic early death.

[3]Note that in Python, integers are not restricted in size.

```
    # Task 13.1
```

```
def is_validly_encoded_program(encoded_program: int) -> bool:
    """Checks whether the given natural number validly encodes a program.

    Parameters:
        encoded_program: natural number to check.

    Returns:
        ``True`` if the given natural number validly encodes a program, ``False``
        otherwise.
    """
    # Task 13.2
```

We could naturally encode programs in, say, ASCII (or unicode), but Gödel was careful to choose a simple encoding, called **Gödel numbering**, that is "compatible" with Peano Arithmetic. Specifically, Gödel's encoding allows the implementation of the following task for Peano Arithmetic:

```
def sentence_saying_program_halts_on_input(encoded_program: int, input: int) \
        -> Formula:
    """Computes a predicate-logic sentence in the language of Peano arithmetic
    that states that the given encoded program halts when run on the given input.

    Parameters:
        encoded_program: natural number encoding a program.
        input: input to the given encoded program.

    Returns:
        A predicate-logic sentence in the language of Peano arithmetic that
        holds in any model for (the axioms of) Peano arithmetic if the program
        encoded by `encoded_program` halts (i.e., eventually terminates rather
        than gets stuck in an infinite loop) when run on the given input, and
        does not hold in any such model otherwise.
    """
    assert is_validly_encoded_program(encoded_program)
    # Task 13.3
```

To implement `sentence_saying_program_halts_on_input()`, we desire to be able to express, using a logical formula in the language of Peano Arithmetic, the behavior of a computer program on a certain input. In general we would like to be able to express any question about what the program is doing at any given point in time (e.g., is it still running?, what is the content of its memory?, etc.), but for simplicity we will focus on the simple question of whether it **halts** or keeps running forever (e.g., gets stuck in an infinite loop.)

Technically, there is some quite nontrivial work in allowing logical formulas to capture the behavior of programs, and that is beyond the scope of this chapter. (Presumably, that is what a second book on Mathematical Logic that is written in the same vein as this book would do.) To get some insight, though, let us give a high-level overview of what the "sentence saying x halts on y" would look like. The basic idea is simple: it would claim the existence of a natural number T and a sequence $\bar{s} = s_1, \ldots, s_T$ (encoded as a natural number) that describes the state of the program (i.e., content of all memory) at all time steps, and then say that the sequence of steps encodes a correct computation. I.e., the sentence saying that the program encoded by x halts when run on input y

may be expressed as '$\exists\bar{s}[(Start_x(s_1,y)\&(\forall i[Follows_x(s_{i+1},s_i)]\&HaltingState_x(s_T)))]$', where '$Start_x(s_1,y)$' is a (sub)formula stating that the initial state s_1 corresponds to the the initial state of the program x when run with input y, '$Follows_x(s_{i+1},s_i)$' is a formula stating that state s_{i+1} is the result of a single computation step of the program x applied to state s_i, and '$HaltingState_x(s_T)$' is a formula stating that state s_T is a halting state of the program x. Each of these three statements requires significant work for us to be able to express it via a predicate-logic formula. For example, the logic of "reading" from the code of a program the information saying what one step of it does to the state of the whole memory is a complex task. Even the basic building blocks of encoding a whole sequence $\bar{s} = s_1, \ldots, s_T$ as a single number (since Peano Arithmetic only has numbers) is a nontrivial task.

Despite the many details that are needed for the implementation of the above task (and which could easily fill an entire separate "Mathematical Logic II through Programming" follow-up book), the overall picture should be clear: Programming (including, say, the Python programming language) is part of Mathematics, and thus it should not surprise us that a language that should handle "usual" mathematics can also logically express whatever we want about the behavior of a computer program.

To conclude this section, we make the crucial observation that since we assumed that our "ultimate" axiomatic system UAS is at least as powerful as Peano Arithmetic, any model of UAS is a model of Peano Arithmetic, and therefore the guarantee of `sentence_saying_program_halts_on_input()` with respect to Peano Arithmetic in fact *also holds with respect to the language and axioms of UAS*: the returned sentence is in the language of UAS, holds in any model of UAS if the program encoded by `encoded_program` halts when run on `input`, and does not hold in any such model if that program does not halt when run on `input`.

3 Undecidability of the Halting Problem

So, say that we sorted out the technical details of how to express the notion that "the program encoded by x halts when run on input y" as a predicate-logic sentence. (Once again, this is nontrivial to pull off technically, and the bulk of a second course on Mathematical Logic is usually dedicated to this.) Since it is natural to have programs (such as compilers) that handle programs, it is tempting to implement a *program* (rather than a logical formula) to answer the question of whether "the program encoded by x halts when run on input y," i.e., to implement the following function:

```
def does_halt(encoded_program: int, input: int) -> bool:
    """Checks if the given encoded program halts when run on the given input.

    Parameters:
        encoded_program: natural number encoding a program.
        input: input to the given encoded program.

    Returns:
        ``True`` if the program encoded by `encoded_program` halts (i.e.,
        eventually terminates rather than gets stuck in an infinite loop) when
        run on the given input, ``False`` otherwise.
    """
    assert is_validly_encoded_program(encoded_program)
    # Can you implement this function?
```

Turing's famous result is that the **Halting Problem** is **undecidable**, that is, it is

impossible to correctly implement the function `does_halt()`! Historically, Turing's **Undecidability Theorem** came after Gödel's Incompleteness Theorem, but we will still use it (after proving it without using Gödel's Incompleteness Theorem, of course) as an intermediate step in the proof of the the Incompleteness Theorem:

Theorem (Undecidability of the Halting Problem)**.** *There does not exist any way to implement* `does_halt()` *correctly.*

Proof. We assume by way contradiction that `does_halt()` is implemented correctly and provide a contradiction. For this consider the following function:

```
def diagonalize(encoded_program: int) -> None:
    """Halts if the given encoded program does not halt when run on its given
    encoding as input, and enters an infinite loop otherwise.

    Parameters:
        encoded_program: natural number encoding a program.
    """
    assert is_validly_encoded_program(encoded_program)

    if does_halt(encoded_program, encoded_program):
        while True: pass # Infinite loop.
    else:
        return
```

Looking at the code of `diagonalize()`, we see that if it gets (an encoding of) a program x that halts when run on x itself, then `diagonalize` enters an infinite loop, while if x does *not* halt when run on x, then `diagonalize` halts immediately. All this, assuming of course, that the function `does_halt()` was indeed implemented correctly (an assumption that, recall, we are really trying to disprove by way of contradiction.)

Let us look at the encoding of this function `diagonalize()` (an encoding that contains also all the sub-functions used in its implementation, including in particular the assumed `does_halt()` function), and let us call it $n_{diagonalize}$. Question: what happens when we run `diagonalize(`$n_{diagonalize}$`)`? Well, looking at the code of `diagonalize()`, there are two possibilities, either the function call `does_halt(`$n_{diagonalize}$`,` $n_{diagonalize}$`)` returns `True` or it returns `False` (notice that by assumption `does_halt()` was implemented correctly so it never gets stuck in an infinite loop and it always halts, returning either `False` or `True`).

Case I: `does_halt(`$n_{diagonalize}$`,` $n_{diagonalize}$`)` returns `True`

By the specification of `does_halt()` this means that the program encoded by $n_{diagonalize}$ when run on input $n_{diagonalize}$ halts. I.e., when we run `diagonalize(`$n_{diagonalize}$`)`, it halts. But now look at the implementation of `diagonalize()`: since `does_halt(`$n_{diagonalize}$`,`$n_{diagonalize}$`)` returns `True`, we have that `diagonalize(`$n_{diagonalize}$`)` immediately enters an infinite loop. I.e., the program encoded by $n_{diagonalize}$ when run on input $n_{diagonalize}$ does *not* halt. A contradiction.

Case II: `does_halt(`$n_{diagonalize}$`,` $n_{diagonalize}$`)` returns `False`

By the specification of `does_halt()` this means that the program encoded by $n_{diagonalize}$ when run on input $n_{diagonalize}$ does *not* halt. I.e., when we run `diagonalize(`$n_{diagonalize}$`)`, it does *not* halt. But look again at the implementation of `diagonalize()`: since `does_halt(`$n_{diagonalize}$`,`$n_{diagonalize}$`)` returns `False`, we have that `diagonalize(`$n_{diagonalize}$`)` immediately halts. I.e., the program encoded by $n_{diagonalize}$ when run on input $n_{diagonalize}$ halts. A contradiction. □

4 The Incompleteness Theorem

So, we have seen that the Halting Problem is undecidable. But now comes the final blow: if our axiomatic system UAS is complete then we *can* implement the `does_halt()` function after all. To do so, in a nutshell, we will simply search either for a proof of the sentence saying that the given program halts, or for a proof of its negation. More technically, we will need to be able to iterate over all possible proofs. For this it would first be convenient to implement parsing for proofs, similar to the parsing that you implemented for formulas:

predicates/proofs.py

```
class Proof:
        ⋮
    @staticmethod
    def parse(string: str) -> Union[Proof, None]:
        """Parses the given valid string representation into a proof.

        Parameters:
            string: string to parse.

        Returns:
            A proof `p` such that `str(p)` returns the given string, or ``None``
            if the given string is not a valid representation of any proof.
        """
        # Task 13.4
```

Finally, it will also be convenient to have access to a generator (very similar to the generator `fresh_variable_name_generator` that you have used throughout this book) that simply generates all one-character Python strings, then all two-character Python strings, and so forth. (So for each Python string, if we invoke the generator enough times we will eventually get that string.) Writing such a generator is a quite an easy exercise for anyone familiar with writing Python generators (and specifically, since you used our implementation of `fresh_variable_name_generator`, it should be clear to you that there is no problem in implementing such a generator), so we will assume that we have it implemented as `string_generator` (possibly imported from, say, `logic_utils`).

Now, assuming that our "ultimate" axiomatic system UAS is complete, we can finally show how to implement `does_halt()` after all:

```
def does_halt(encoded_program: int, input: int) -> bool:
    """Checks if the given encoded program halts when run on the given input.

    Parameters:
        encoded_program: natural number encoding a program.
        input: input to the given encoded program.

    Returns:
        ``True`` if the program encoded by `encoded_program` halts (i.e.,
        eventually terminates rather than gets stuck in an infinite loop) when
        run on the given input, ``False`` otherwise.
    """
    assert is_validly_encoded_program(encoded_program)

    program_halts_sentence = \
        sentence_saying_program_halts_on_input(encoded_program, input)
    program_loops_sentence = Formula('~', program_halts_sentence)
```

```
    while True:
        proof_string = next(string_generator)
        proof = Proof.parse(proof_string)
        if proof is not None and proof.assumptions == UAS and proof.is_valid():
            if proof.conclusion == program_halts_sentence:
                return True
            if proof.conclusion == program_loops_sentence:
                return False
```

I.e., our implementation just looks at all possible proofs, trying to find a proof either that the given encoded program halts on the given input or that the given encoded program does not halt on the given input. Had UAS been complete, then it would have been assured that one of these would have been found. Since this would have contradicted the Undecidability Theorem, we have that UAS cannot be complete:

Theorem (The Incompleteness Theorem)**.** *Any such axiomatic system UAS is not complete.*

Proof. We will show that if UAS had been a complete axiomatic system then our implementation of `does_halt()` would have been correct, which would have contradicted the undecidability of the Halting Problem. So, assume by way of contradiction that UAS is complete.

Let `encoded_program` be an encoded program let and `input` be an input to the program encoded by `encoded_program`. Let ϕ be the formula called `program_halts_sentence` in the above alleged implementation of `does_halt()` when run on `encoded_program` and on `input`. I.e., let ϕ be the formula returned by `sentence_saying_program_halts_on_input(encoded_program, input)`. Due to our assumption that UAS is a complete axiomatic system (an assumption that, recall, we are really trying to disprove by way of contradiction), we have that either ϕ or '$\sim\phi$' has a valid proof from UAS. Recall that by definition, this proof is finite and so is its string representation. Call the string representation of that proof (i.e., the string returned by calling `str()` on that proof) s^*. Now, at some point (i.e., after finitely many steps) during the execution of `does_halt(encoded_program, input)`, the call to `next(string_generator)` will return s^*. If the proof represented by s^* is a valid proof from UAS of ϕ, then our implementation of `does_halt()` will return `True`. In this case, due to the Soundness Theorem for Predicate Logic,[4] the given program indeed halts when run on `input`. Thus the answer returned by our implementation of `does_halt()` is the correct one! Similarly, if the proof represented by s^* is a valid proof from UAS of '$\sim\phi$', then our implementation of `does_halt()` will return `False`. In this case, again due to the Soundness Theorem for Predicate Logic, the given program indeed does *not* halt when run on `input`. Thus the answer returned by our implementation of `does_halt()` is again the correct one!

Since we are assured that for any possible `encoded_program` and `input`, one of these two possibilities (a proof of ϕ or a proof of its negation) will be found after finitely many steps, our implementation of `does_halt()` always returns the correct answer, hence solving the Halting Problem. A contradiction. □

Re-examining the above chain of arguments, we note that indeed all that it requires is that the function `sentence_saying_program_halts_on_input()` can be implemented, or in other words, that the axioms of Peano Arithmetic are sufficiently strong to deal with programs. Implied here is that if such a sentence is provable from Peano Arithmetic then

[4] I.e., since UAS is sound, whatever is provable from UAS indeed follows semantically from UAS.

indeed it soundly describes the behavior of the given encoded program when run on the given input. As already noted above, there is no problem in implementing this function, but it is technically challenging, and the bulk of a second book on Mathematical Logic following this book would be dedicated to developing the techniques to do this.

Have we made any other assumptions regarding our hypothetical "ultimate" axiomatic system UAS? As you may have noticed, for convenience we have implicitly assumed (in the Python code, at least) that it consists of finitely many schemas. We nonetheless note that in fact nothing in our proof depends on such finiteness or on our specific definition of how schemas may be instantiated. Indeed, we could have replaced the test for whether `proof.assumptions == UAS` in our implementation of `does_halt()` with any arbitrary code that tests that all of the assumptions of `proof` are in UAS (or are instantiations of schemas in UAS, however we may wish to define schemas and instantiation).[5] This code could even be very inefficient, but all that would be required of it for our proof to still follow through would be that such code (that never gets stuck in an infinite loop) would exist, that is, that given a formula, it would be possible to unambiguously decide on whether or not it is a valid axiom of UAS.

Let us conclude by briefly discussing this decidability condition. We first note that this is a reasonable condition since an axiomatic system is useless if we cannot verify whether an assumption used in a proof is part of that system or not. Second, we note that this condition really is a necessary and meaningful condition of our analysis. Indeed, consider the following strategy for constructing UAS: we could have "closed" (as in Chapter 12) the set of axioms of Peano Arithmetic without losing consistency, to obtain a closed superset of them, and could have chosen this (infinite) closed set as our axiomatic set. As shown in Chapter 12, a consistent closed set essentially has a single model, so for any sentence, either the sentence or its negation is true in this model (and hence in any model of this closed set of axioms), so this closed set of axioms is complete! Nonetheless, our proof shows that the fatal flaw with this approach is that while such a closed set that contains the axioms of Peano Arithmetic does exist (we have proved this in Chapter 12), and while any such closed set indeed is a complete set of axioms, for any such closed set it is undecidable whether a formula belongs to it or not—that is, it is impossible to implement code that correctly answers the question "is the following axiom/schema in this set?". This of course means that it is hopeless to verify proofs that use any such set as their axiomatic system, and so such a closed (and hence complete) axiomatic system, while theoretically existing, is useless to us.

Indeed, the above condition, of decidability of the axiomatic system, really is rather weak, and the fact that our proof hinges on only this condition shows that the dream of finding a complete axiomatic system for Mathematics is indeed just that: a dream.

[5] Similarly, we could have changed the test for whether `proof.is_valid()` to allow for more inference rules in our proofs, etc.