# 1 Self-Reference

In this note we will explore the deep connection between proofs and computation. At the heart of this connection is the notion of self-reference, and it has far-reaching consequences for the limits of computation (the Halting Problem) and the foundations of logic in mathematics (Gödel's incompleteness theorem). Many of the results we will talk about are based on the idea of *self-reference*; thus, we spend the first part of this note exploring this idea.

## 1.1 The Liar's Paradox

Recall that propositions are statements that are either true or false. We saw in an earlier lecture that some statements are not well defined or too imprecise to be called propositions. But here is a statement that is problematic for more subtle reasons:

"All Cretans are liars."

So said a Cretan in antiquity, thus giving rise to the so-called liar's paradox which has amused and confounded people over the centuries. Why? Because if the statement above is true, then the Cretan was lying, which implies the statement is false. But actually the above statement isn't really a paradox; it simply yields a contradiction if we assume it is true, but if it is false then there is no problem.

Consider another manifestation of self-reference, due to the great logician Bertrand Russell. In a village with just one barber, everyone keeps themself clean-shaven. Some shave themselves, while others go to the barber. The barber proclaims:

"I shave all and only those who do not shave themselves."

It seems reasonable then to ask the question: Does the barber shave themself? Thinking more carefully about the question though, we see that, assuming that the barber's statement is true, we are presented with another self-referential paradox: a logically impossible scenario. If the barber does not shave themself, then according to the above statement, the barber shaves themself. But then if the barber does shave themself, the statement says exactly the opposite!

Of course, we can get around this paradox by simply saying that the barber's statement must be false. But there is no such way to get around the following (paradoxical) statement:

"This statement is false."

Is the statement above true? If the statement is true, then what it asserts must be true; namely that it is false. But if it is false, then it must be true. So it really is a paradox, and we see that it arises because of the self-referential nature of the statement.

## 1.2 Russell's Paradox

While the above paradoxes may be interesting to puzzle through, they hardly seem to have any practical or mathematical significance. However, mathematics is not immune from the power of self-reference, and the

next example we see shook the mathematical world when it was first noted in the early 20th century, again by Bertrand Russell.

Given a set $s$, we can always ask questions of the form "is $x$ an element of $s$" for any $x$. In particular, we can ask "is $s$ an element of itself", as sets are themselves allowed to contain sets. Suppose we define $S$ to be the set of all sets which do not contain themselves; that is $S = \{s \mid s \notin s\}$. We now ask: is $S$ an element of itself?

If the answer is no, ie if we have that $S \notin S$, then when defining $S$ we must have included it in itself, a contradiction. On the flip side, if the answer is no, meaning that $S \in S$, then we never would have included $S$ in itself, as it doesn't match the definition. Hence, we reach a contradiction no matter what the answer is — we have a paradox!

When this paradox first came to light, many mathematicians were concerned that it would undermine the consistency and usefulness of set theory, and by extension, most of modern mathematics. As you may have noticed, however, modern mathematics still exists.[1] Indeed, mathematicians were able to rethink the axioms of set theory in order to prevent the set $S$ from being defined, and hence to avoid this paradox. However, this came at a cost: while sets can still contain other sets, we cannot define the set of *all* sets, nor many other sets we would have found useful.

# 2 Computability

Self-reference is not only a problem for pure-mathematical realms like set-theory. As our main focus for this note, we explore the question of whether computers can do everything — and using self-reference, we show that the answer is actually no!

## 2.1 An Important Aside

Before diving in to our main results, we need to take a moment to make an important observation about computer programs. Many of you are used to writing code on a computer by this point — but what is that code actually? From your computer's point of view, it's just a (potentially very long) string of zeros and ones! Thus, when we think of computer programs, we can equally well think of bit strings, as we can use such strings to represent any program we might like to write.

This comes into play for our purposes because it means that it makes perfect sense for us to use programs as the inputs for other programs. After all, we would have no problems accepting a bit string as the input to a function, so we should be fine accepting a bit string and then interpreting it as a program. In particular, this will allow us to pass a program's source code *as an input to itself*, allowing for self-reference.

## 2.2 The Halting Problem

One natural question that arises in software engineering is "does this code work?" This question can be a bit amorphous as "work" is subjective, so we will focus on a more basic question: does this code eventually finish, or will it keep on running forever. One could see, for example, this being a useful tool to include in a compiler. If the compiler notices that your code is going to loop forever, it likely means that there is a bug somewhere in it, so the compiler can bring this to your attention and potentially save you from many hours of painful debugging.

[1] Citation needed.

Formally, in order to solve the "Halting Problem", we would need to write a program `TestHalt` that behaves as follows:

$$\texttt{TestHalt(P,x)} = \begin{cases} \text{"yes"}, & \text{if program } P \text{ halts on input } x \\ \text{"no"}, & \text{if program } P \text{ loops on input } x \end{cases} \tag{1}$$

Unfortunately, as Alan Turing proved in 1936, solving this problem is impossible.

**Theorem 11.1.** *The Halting Problem is uncomputable. That is, it is impossible to write a program* `TestHalt` *that behaves as specified by (1) on all possible inputs.*

*Proof.* Assume for the sake of contradiction that the Halting Problem is in fact computable. That is, assume that we have some program `TestHalt` which outputs true if $P(x)$ halts and false otherwise. We now construct a program on which `TestHalt` must fail:

```
Turing(P):
    if TestHalt(P, P) = True:  loop forever
    else:  halt
```

Solely under the assumption that `TestHalt` exists, it is straightforward to implement `Turing` using `TestHalt` as a subroutine.

Now consider what happens if we call `Turing` *with its own source code as input*. That is, what does `Turing(Turing)` do? This depends on what `TestHalt(Turing, Turing)` returns. If `TestHalt(Turing, Turing)` returns true, `Turing(Turing)` will loop forever, meaning the correct answer for `TestHalt(Turing, Turing)` was actually false. But if `TestHalt(Turing, Turing)` returns false, `Turing(Turing)` will halt immediately, meaning the correct answer for `TestHalt(Turing, Turing)` is actually true!

What this tells us is that, no matter what `TestHalt(Turing, Turing)` returns, it will be wrong![2] This contradicts our assumption that `TestHalt` behaves as desired on *all* possible inputs, so we must in fact have that no such function `TestHalt` can ever exist. □

In fact, there are many more questions we would like to answer about programs but cannot. For example, we cannot know if a program ever outputs anything or if it ever executes a specific line. We also cannot check if two programs produce the same output. And we cannot check to see if a given program is a virus. These issues are explored in greater detail in the advanced course CS172 (Computability and Complexity).

## 2.3 Diagonalization and the Halting Problem [OPTIONAL]

As an aside, we note that the above proof can also be phrased as a proof by diagonalization, the same technique that we used in the previous lecture to show that the real numbers are uncountable. Why? Since the set of all computer programs is countable (they are, after all, just finite-length strings over some alphabet, and the set of all finite-length strings is countable), we can enumerate all programs as in figure 1 (where $P_i$ represents the $i^{\text{th}}$ program).

The $(i,j)^{\text{th}}$ entry in the table above is H if program $P_i$ halts on input $P_j$, and L (for "Loops") if it does not halt. Now if the program `Turing` exists it must occur somewhere on our list of programs, say as $P_n$. But this cannot be, since if the $n^{\text{th}}$ entry in the diagonal is H, meaning that $P_n$ halts on $P_n$, then by its definition `Turing` loops on $P_n$; and if the entry is L, then by definition `Turing` halts on $P_n$. Thus the behavior of

[2] In order to be fully formal here, we should also consider the cases where `TestHalt(Turing, Turing)` loops forever or errors out. In any such case, we consider `TestHalt` to be wrong, as it certainly does not return the correct answer.

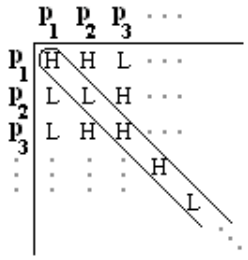


Figure 1: The Halting Problem as Diagonalization

`Turing` is different from that of $P_n$, and hence `Turing` does not appear on our list. Since the list contains all possible programs, we must conclude that the program `Turing` does not exist. And since `Turing` is constructed by a simple modification of `TestHalt`, we can conclude that `TestHalt` does not exist either. Hence the Halting Problem cannot be solved.

## 2.4 The "Easy" Halting Problem

As noted above, the key idea in establishing the uncomputability of the Halting Problem is self-reference: Given a program $P$, we ran into trouble when deciding whether $P(P)$ halts. But in practice, how often do we want to execute a program with its own description as input? Is it possible that if we disallow this kind of self-reference, we can solve the Halting Problem?

Taking this idea to its logical extreme, what happens if we simply don't allow our program to take in any input at all? Formally, we say that we can solve the "Easy" Halting Problem if we can write a program `EasyTestHalt` such that

$$\texttt{EasyTestHalt(P)} = \begin{cases} \text{"yes"}, & \text{if program } P \text{ halts given no input} \\ \text{"no"}, & \text{if program } P \text{ loops given no input} \end{cases} \tag{2}$$

Not allowing our program to take an input seems like it should make the problem easier, hence the name "Easy Halting Problem". However, it turns out that even this seemingly simpler problem is still uncomputable. This is formalized in the following theorem.

**Theorem 11.2.** *The Easy Halting Problem is undecidable. That is, there is no program `EasyTestHalt` that behaves as specified in (2) on all possible inputs.*

*Proof.* Suppose for the sake of contradiction that there was a program `EasyTestHalt` that worked on every possible input. I claim that we can implement `TestHalt` as follows:

```
TestHalt(P, x):
    def P'():
        return P(x)
    return EasyTestHalt(P')
```

Indeed, notice that despite the fact that `P'` takes no input, we defined it such that it will halt if and only if `P(x)` does. Thus, since we have assumed that `EasyTestHalt` always returns the correct answer, whatever answer it returns on input `P'` will be the correct answer for `TestHalt` on inputs `P` and `x`.

Thus, we have now given a way to implement `TestHalt` such that it works on every possible input. But Theorem 11.1 tells us that this is impossible! Thus, we have reached a contradiction, and so can conclude that the Easy Halting Problem must in fact be uncomputable. □

What this proof is effectively saying is what the quotation marks around "Easy" may have tipped you off to: solving the Easy Halting Problem is no easier than solving the original. Indeed, our proof showed that any algorithm to solve the Easy Halting Problem can also be used to solve the regular Halting Problem, as we can just have the input to our function "hard-coded" into it. This notion of using one problem to solve another, and hence showing that the former can be no easier than the latter, is known as a *reduction*.

## 2.5 Reductions

We say that a problem A reduces to a problem B if we can use problem B in order to solve problem A. That is, we can write a program $\texttt{P}_\texttt{A}$ to solve A so long as we have a program $\texttt{P}_\texttt{B}$ solving B that we can use as a subroutine.[3] For the purposes of the reduction, we treat $\texttt{P}_\texttt{B}$ as a "black box" — we know that it works, but we don't know (or need to know) how precisely it works. Being agnostic to the inner workings of $\texttt{P}_\texttt{B}$ in this way means that our reduction works *regardless of how* $\texttt{P}_\texttt{B}$ *is implemented*.

In the case of the previous section, this means that `EasyTestHalt` cannot exist no matter how clever we are in our implementation. Had we "opened up the black box" and made assumptions about how `EasyTestHalt` worked, we would only have proved that `EasyTestHalt` could not be implemented in that particular way, while leaving open the possibility that there is some other way of doing it which does not fit our assumptions.

Before moving on, we will give one more example of a reduction. In this case, we will do the reverse of what we did in the proof of Theorem 11.2: instead of reducing the Halting Problem to the Easy Halting Problem, we will reduce the Easy Halting Problem to the regular one. To do this, we suppose that we have access to a working version of `TestHalt` and write `EasyTestHalt` as follows:

```
EasyTestHalt(P):
    def P'(x):
        P()
    return TestHalt(P', 42)
```

We notice here that `P'` simply ignores its input and runs `P`. Thus, `P'(42)` will halt precisely when `P()` does, meaning that the (correct) return value from `TestHalt(P', 42)` is precisely the value we wished to return for `EasyTestHalt(P)`.

What this tells us is that, at the very least, the Easy Halting Problem is no more difficult than the regular Halting Problem. Combining this with our results from Section 2.4, we see that the Easy and regular Halting Problems are in fact *the same difficulty*, ie, a solution to either problem could be used to construct a solution to the other without much difficulty.

# 3 Beyond Computability

In the previous sections, we have seen examples of problems for which it is impossible to write an algorithm which always finishes in finite time and gives the correct answer. However, if we relax these constraints, we can sometimes get somewhere — and uncover hidden structures of uncomputable problems.

[3] This is the most general form of reduction, known as a Turing reduction, as it allows us to use $\texttt{P}_\texttt{B}$ as many times as we want and use its response however we choose. There are many other types of reductions which add restrictions on how and how often we can use $\texttt{P}_\texttt{B}$, but we will not be discussing them today.

## 3.1 Recursive Enumerability

One idea that may have come up when you first saw the definition of the Halting Problem would be the following "algorithm": on inputs `P` and `x`, we simply simulate `P(x)`. If it halts, we can immediately output true; otherwise, we output false.

However, this doesn't actually form an algorithm as specified by (1) — how do we know when to declare that `P(x)` is never going to halt? We could, for example, wait for a very long time, say 9000 years, before giving up and declaring that it must be looping forever. But then what happens if `P(x)` does actually halt, but takes 9001 years to do so? We'll have given a wrong answer!

The only way to make this algorithm avoid giving a wrong answer, then, is to have it never give up on an program; that is, we will "wait forever" for `P(x)` to halt. However, this means that if `P(x)` never halts, `TestHalt(P, x)` won't either. This is not allowed in our original definition in (1), but we can relax our definition in order to make it fine. This relaxation leads to the notion of *recursive enumerability*:

**Definition 11.1.** *We say a problem A is "recursively enumerable" or "recognizable" if there exists a "recognizer" $R_A$ such that*
*(1) For any x for which the correct answer is "yes", $R_A(x)$ outputs "yes" in finite time.*
*(2) For any x for which the correct answer is "no", $R_A(x)$ either outputs "no" in finite time or loops forever.*

Applying this definition, we see that our previous "algorithm" of simulating `P(x)` and returning true if/when it halts is in fact a recognizer for the Halting Problem. Hence, while the Halting Problem is not computable, it is recognizable.

## 3.2 Hilbert's Entscheidungsproblem

In 1928, the famous mathematicians David Hilbert and Wilhelm Ackermann proposed a series of challenges for their peers to ponder. Here, we will take a closer look at the third of these challenges, known as the Entscheidungsproblem (German for "decision problem"), which roughly asked for an algorithm that could determine the truth or falsehood of any statement in a sufficiently formal logical system.

In 1936, at the same time that he proved the halting problem is undecidable, Alan Turing also gave a negative answer to Hilbert's Entscheidungsproblem — no algorithm solving it can possibly exist. In short, Turing showed that a sufficiently expressive logical system can encode statements of the form "`P(x)` halts". This means that an algorithm for Hilbert's Entscheidungsproblem could be used to solve the Halting Problem by simply asking it if the statement "`P(x)` halts" is true or false.[4]

While this shows that the Entscheidungsproblem is not computable, it turns out that it is still recursively enumerable! To prove this, we give the following recognizer for it. We start by trying all possible proofs that involve only one step. If one of them proves our statement, we output "yes". Otherwise, we move on to checking all possible proofs that take two steps. If any of them works, we output "yes"; otherwise, we move on to proofs with three steps. If a statement has a finite length proof, this recognizer will eventually find it and output "yes"; if there is no proof, the recognizer will loop forever.[5]

We note here that our recognizer makes two key assumptions. The first is that a proof can be algorithmically checked for correctness, while the second is that at any point in the proof, there are only a finite number of

[4] Note that this is another example of a reduction!

[5] Technically, this recognizes a slightly different problem than what we were originally interested in: it recognizes whether or not there is a *proof* of a statement, rather than whether or not the statement is true. However, in addition to his more famous incompleteness theorem, Gödel also proved a completeness theorem which says that every "true" statement in first-order logic can be proved. What exactly "true" means in this context is beyond our scope.

"next steps" one might take. These assumptions are generally not true of the human-readable proofs we are used to, but are true in the formal proof systems generally considered for these sorts of problems.

## 3.3 Complete Problems

One thing you may have noticed is that the recognizers for the Halting Problem and the Entscheidungsproblem have significant parallels in their design. Indeed, one reason why this is the case is that they are both *complete* for the class or recognizable problems; that is, any recursively enumerable problem can be reduced to the Halting Problem and to the Entscheidungsproblem.

**Theorem 11.3.** *Let A be recursively enumerable. Then A can be reduced to the Halting Problem.*

*Proof.* Since A is recursively enumerable, we know that we have some recognizer for it $\texttt{R}_\texttt{A}$. Now consider the following solver for A, which uses both that recognizer and a "black box" solver for `TestHalt`:

```
Solver(x):
    if TestHalt(R_A, x) = false:  return false
    else:  return R_A(x)
```

On input x, there are three things $\texttt{R}_\texttt{A}$ might do: it could return "yes", it could return "no", or it could loop forever. In the former two cases, we know from Definition 11.1 that the answer $\texttt{R}_\texttt{A}$ returns is the correct answer for x, so we are correct to return in. For the final case, Definition 11.1 tells us that $\texttt{R}_\texttt{A}$ will only loop infinitely if the correct answer is "no", meaning we again in this case will return correctly. □

Notice here that we critically used `TestHalt` in order to circumvent the problems we would have had if $\texttt{R}_\texttt{A}$ loops forever. The possibility of looping on a "no" input is what makes $\texttt{R}_\texttt{A}$ a recognizer rather than a solver, so being able to spot when that would happen and act accordingly gives us an algorithm for our problem!

---

*Exercise.* Prove Theorem 11.3 with Hibert's Entscheidungsproblem instead of the Halting Problem.

---

## 3.4 Co-Recursively Enumerable

Having worked with recursive enumerability, we might now be wondering what was so special about the "no" case. That is, what happens if instead of allowing the algorithm to loop if the answer to its input is false, we allow it to loop if the answer to its input is true? With this change, we get the following definition:

**Definition 11.2.** *We say that a problem A is "co-recursively enumerable" or "co-recognizable" if there exists a "co-recognizer"* $\texttt{CR}_\texttt{A}$ *such that*
*(1) For any x for which the correct answer is "yes",* $\texttt{CR}_\texttt{A}\texttt{(x)}$ *outputs "yes" in finite time or loops forever.*
*(2) For any x for which the correct answer is "no",* $\texttt{CR}_\texttt{A}\texttt{(x)}$ *outputs "no" in finite time.*

Here, the "co-" stands for "complement". This is because every co-recognizable problem is the complement of a recognizable one; that is, it is a recognizable problem with all "yes" and "no" answers reversed. For example, the "Looping Problem", which asks if a program loops forever, is co-recursively enumerable, as it is the complement of the Halting Problem.

---

*Sanity check!* Give a co-recognizer for the Looping Problem.

---

We might now well ask how Definitions 11.1 and 11.2 compare. In particular, is it possible for a problem to be both recognizable and co-recognizable? As it turns out, the answer is yes — but only if the problem is actually computable!

**Theorem 11.4.** *A problem A is both recursively enumerable and co-recursively enumerable if and only if it is computable.*

*Proof.*
**If**: Suppose that A is computable. This means that we have a solver for A which always runs in finite time and always outputs the correct answer. Notice that a solver is in particular both a recognizer and a co-recognizer, so A fits both definitions 11.1 and 11.2.

**Only If**: Suppose that A is both recognizable and co-recognizable. This means that we have a recognizer $\texttt{R}_\texttt{A}$ and a co-recognizer $\texttt{CR}_\texttt{A}$. We can create a solver for A by running $\texttt{R}_\texttt{A}$ and $\texttt{CR}_\texttt{A}$ "in parallel" and outputting the answer of whichever of them terminates first. Note that regardless of what the correct answer is for a given input x, at least one of $\texttt{R}_\texttt{A}\texttt{(x)}$ or $\texttt{CR}_\texttt{A}\texttt{(x)}$ must halt in finite time — and whenever either of them halt, they output the correct answer. □

One small technicality we need to deal with in the above proof is what exactly it means to run the recognizer and co-recognizer "in parallel". On a real-world computer, we are able to multi-thread, so this is not a problem, but in many formal models of computation, this is not available to us. However, we note that what we can instead do is interleave the steps of our simulations of the two programs. That is, we first run one step of $\texttt{R}_\texttt{A}$, then one step of $\texttt{CR}_\texttt{A}$, then the second step of $\texttt{R}_\texttt{A}$ and of $\texttt{CR}_\texttt{A}$, and so forth. As long as at least one of these two functions eventually halts (which we know is true), this interleaved simulation will also finish in at most twice that amount of time.

Now that we have Theorem 11.4, we can state a few facts which it seems difficult to prove in any other way. In particular, since we know that the "Looping Problem" is co-recongizable but not computable, we know that it is not recognizable. Similarly, we can say that neither the Halting Problem nor Hilbert's Entscheidungsproblem are co-recognizable.

## 3.5 Postscript

In this note, we have barely scratched the surface of the beautiful and complex structure that is undecidable problems. Indeed, there are problems so "hard" as to be neither recursively enumerable nor co-recursively enumerable, as well as uncomputable problems which are strictly easier than the Halting Problem, meaning we can reduce those problems to the Halting Problem, but can't go the other way around. There are even uncomputable problems which are incomparable to one another — you can't do a reduction in either direction! If you'd like to learn more about this, you can may consider taking Math 136, Incompleteness and Undecidability.

# 4 Godel's Incompleteness Theorem [OPTIONAL]

In 1900, the great mathematician David Hilbert posed the following two questions about the foundation of logic in mathematics:

1. Is arithmetic consistent?
2. Is arithmetic complete?

To understand the questions above, we recall that mathematics is a formal system based on a list of axioms (for example, Peano's axioms of the natural numbers, axiom of choice, etc.) together with rules of inference. The axioms provide the initial list of true statements in our system, and we can apply the rules of inference to prove other true statements, which we can again use to prove other statements, and so on.

The first question above asks whether it is possible to prove both a proposition $P$ and its negation $\neg P$. If this is the case, then we say that arithmetic is *inconsistent*; otherwise, we say arithmetic is *consistent*. If arithmetic is inconsistent, meaning there are false statements that can be proved, then the entire arithmetic system will collapse because from a false statement we can deduce anything, so every statement in our system will be vacuously true.

The second question above asks whether every true statement in arithmetic can be proved. If this is the case, then we say that arithmetic is *complete*. We note that given a statement, which is either true or false, it can be very difficult to prove which one it is. As a real-world example, consider the following statement, which is known as Fermat's Last Theorem:

$$(\forall n \geq 3)\ \neg(\exists x, y, z \in \mathbb{Z}^+)(x^n + y^n = z^n).$$

This theorem was first stated by Pierre de Fermat in 1637,[6] but it has eluded proofs for centuries until it was finally proved by Andrew Wiles in 1994.

In 1928, Hilbert formally posed the questions above as the Entscheidungsproblem. Most people believed that the answer would be "yes," since ideally arithmetic should be both consistent and complete. However, in 1930 Kurt Gödel proved that the answer is in fact "no": Any formal system that is sufficiently rich to formalize arithmetic is either inconsistent (there are false statements that can be proved) or incomplete (there are true statements that cannot be proved). Gödel proved his result by exploiting the deep connection between proofs and arithmetic. Actually Gödel's theorem also embodies a deep connection between proofs and computation, which was illuminated after Turing formalized the definition of computation in 1936 via the notion of Turing machines and computability.

In the rest of this note, we will first sketch the essence of Gödel's proof, and then we will outline an easier proof of the theorem using what we know about the Halting Problem.

## 4.1 Sketch of Godel's Proof

Suppose we have a formal system $F$, which consists of a list of axioms and rules of inference, and assume $F$ is sufficiently expressive that we can use it to express all of arithmetic.

Now suppose we can write the following statement:

$$S(F) = \text{"This statement is not provable in } F\text{."}$$

Once we have this statement, there are two possibilities:

1. Case 1: $S(F)$ is provable. Then the statement $S(F)$ is true, but by inspecting the content of the statement itself, we see that this implies $S(F)$ should not be provable. Thus, $F$ is inconsistent in this case.

2. Case 2: $S(F)$ is not provable. By construction, this means the statement $S(F)$ is true. Thus, $F$ is incomplete in this case, since there is a true statement (namely, $S(F)$) that is not provable.

[6] Along with the famous note: "I have discovered a truly marvelous proof of this, which this margin is too narrow to contain."

To complete the proof, it now suffices to construct such a statement $S(F)$. This is the difficult part of Gödel's proof, which requires a clever encoding (a so-called "Gödel numbering") of symbols and propositions as natural numbers.

## 4.2 Proof via the Halting Problem

Let us now see how we can prove Gödel's result by reduction to the Halting Problem. Here we proceed by contradiction: Suppose arithmetic is both consistent and complete; we will use this assumption to solve the Halting Problem, which we have seen is impossible.

Recall that in the Halting Problem we want to decide whether a given program $P$ halts on a given input $x$. For fixed $P$ and $x$, let $S_{P,x}$ denote the proposition "$P$ halts on input $x$." The key observation is that this proposition can be phrased as a statement in arithmetic. The form of the statement $S_{P,x}$ will be

$$\exists z(z \text{ encodes a valid halting execution sequence of } P \text{ on input } x).$$

Although the details require some work, your programming intuition should hopefully convince you that such a statement can be written, in a fairly mechanical way, using only the language of standard arithmetic, with the usual operators, connectives and quantifiers: basically the statement just has to check, step by step, that the string $z$ (encoded as a very long integer in binary) lists out the sequence of states that a computer would go through when running program $P$ on input $x$.

Now let us assume, for contradiction, that arithmetic is both consistent and complete. This means that, for any $(P,x)$, the statement $S_{P,x}$ is either true or false, and that there must exist a proof in arithmetic of either $S_{P,x}$ or its negation, $\neg S_{P,x}$ (and not both). But now recall that a proof is simply a finite binary string. Therefore, there are only countably many possible proofs, so we can enumerate them one by one and search for a proof of $S_{P,x}$ or $\neg S_{P,x}$. The following program performs this task:

```
Search(P,x)
  for every proof q:
    if q is a proof of S_{P,x} then output "yes"
    if q is a proof of ¬S_{P,x} then output "no"
```

The program Search takes as input the program $P$, and proceeds to check every possible proof until it finds either one that proves $S_{P,x}$, or one that proves $\neg S_{P,x}$. By assumption, we know that one of these proofs always exists, so the program Search will terminate in finite time, and it will correctly solve the Halting Problem. On the other hand, since we have already established that the Halting Problem is uncomputable, such a program Search cannot exist. Therefore, our initial assumption must be wrong, so it is not true that arithmetic is both consistent and complete.

Note that in the argument above we rely on the fact that, given a proof, we can construct a program that mechanistically checks whether it is a valid proof of a given proposition. This is a manifestation of the intimate connection between proofs and computation.