10 Uncomputable Problems

10.1 Hilbert's Tenth Problem

There is a another problem about multivariate polynomials that is concerned with integer solutions.

Definition 10.1. The *integer-zero problem* takes a multivariate polynomial p with integer coefficients as input and asks whether there exist integer values (members or $\mathcal{Z}$) for the variables that occur in p that make $p = 0$.

In 1900, mathematician David Hilbert posed a list major challenges in mathematics. The tenth problem in the list was to find an algorithm to solve the integer-zero problem or to show that no such algorithm exists. It was not until 1970 that Hilbert's Tenth Problem was solved, in the negative, when Russian mathemetician Yuri Matiyasevich showed that the integer-zero problem is uncomputable.

A proof that Hilbert's Tenth Problem is not computable is far out of reach for us. Matiyasevich relied on work by Martin Davis, Hilary Putnam and Julia Robinson spanning 21 years, and they relied on prior work. But we will be able to prove that some other problems are uncomputable.

10.2 Infinite Loops

Recall that we only say that program p computes function f or language L if p stops on every input. But there are programs that do not stop on every input (that, by definition, do not compute any function or language).

Let's write $p(x)$ to indicate the value that program p returns when it is given input (or parameter) x. Because a program might not always stop, $p(x)$ might not have a value. It is useful to create a special value, $\perp$ (called "bottom"), and say that $p(x) = \perp$ when p runs forever on input x.

You can't know by running p on input x whether it loops forever; it might just take a very, very long time to stop. But, from a mathematical standpoint, p either stops or it doesn't, so either $p(x) = \perp$ or $p(x) \neq \perp$.

When a "value" might be $\perp$, we use relation $\cong$ instead of $=$, where $x \cong y$ is read as "x is equivalent to y."

Definition 10.2. If x and y are strings then $x \cong y \leftrightarrow x = y$. Also, $\perp \cong \perp$. But $x \ncong \perp$ and $\perp \ncong x$ for any string x.

Definition 10.3. $p(x)\downarrow$ (p halts on input x) is equivalent to $p(x) \ncong \perp$. $p(x)\uparrow$ (p does not halt on input x) is equivalent to $p(x) \cong \perp$.

10.3 Interpreters

Your familiarity with computers tells you that, except for resource limitations, any computer can run programs written in any programming language. For example, you can run a Python program on a computer by loading a Python interpreter onto it.

Interpreters are important tools of computability theory. An interpreter allows you to take a program (a string) and run it inside some other program. Running a program via an interpreter must produce the same results as running it directly.

We have allowed programs to be written in any sufficiently strong programming language. Whatever that programming language L is, a self-interpreter is an interpreter for L written in L.

Definition 10.4. A *self-interpreter* (or, more briefly, an interpreter) is a program I having the property that, for every program p and string x, $I(p, x) \cong p(x)$.

Theorem 10.5. There exists a program I that is an interpreter.

Because we know that an interpreter exists, it is acceptable to write $p(x)$ within the body of a program, where p is a string that is either a parameter of the program or that is computed by the program. Running p is just a matter to running a fixed program, the interpreter, that can be built into your own program.

10.4 Problems about Programs

Some decision problems ask questions about programs. An easy one is: Does program p contain a variable called z? But consider the following decision problem, analogous to the acceptance problem for FSMs.

Definition 10.6. The *acceptance problem for programs* is the following decision problem.

Input. Program p and string x.

Question. Is $p(x) \cong 1$?

An obvious approach to solving the acceptance problem is to run p on input x and see whether the result is 1. But what if p loops forever? Clearly, that approach does not work.

We have seen that the failure of an obvious approach does not allow us to conclude that no algorithm exists. Concluding that the acceptance problem is not computable needs a rock-solid proof. We will give such a proof in a later section.

10.5 An Uncomputable Decision Problem

Now we identify a decision problem that we can prove is uncomputable.

Definition 10.7. The *Halting Problem* is language

$$HLT = \{(p, x) \mid p(x)\downarrow\}.$$

That is, it is the following decision problem.

Input. Program p and string x.

Question. Does p ever stop when it is run on input x?

Theorem 10.8. The Halting Problem is not computable.

Proof.

1. The proof is by contradiction. Start by assuming that the Halting Problem is computable.

Know:	The Halting Problem is computable.
Goal:	**F**.

2. Now we know something that uses term *computable*, and that suggests using a definition. By the definition of a computable decision problem, saying that HLT is computable is equivalent to saying that there exists a program r that stops on all inputs and where, for all y,

$$r(y) \cong 1 \iff y \in \text{HLT},$$

$$r(y) \cong 0 \iff y \notin \text{HLT},$$

But HLT is a set or ordered pairs. It only makes sense to ask if $y \in$ HLT if y is an ordered pair. So lets say that $y = (p, x)$.

Know:	There exists a program r that halts on all inputs so that, for all p and x, $r(p, x) \cong 1 \leftrightarrow (p, x) \in \text{HLT}$ and $r(p, x) \cong 0 \leftrightarrow (p, x) \notin \text{HLT}$.
Goal:	**F**.

3. When you know there exists something with a particular property, you ask someone else to give you such a thing. Let's do that, and call the program that was given to us r. The fact that r halts on all inputs is implicit in the two equivalences (1) and (2).

Known variables:	r (a program)
Know (1):	For all p and x, $r(p, x) \cong 1 \leftrightarrow (p, x) \in \text{HLT}$.
Know (2):	For all p and x, $r(p, x) \cong 0 \leftrightarrow (p, x) \notin \text{HLT}$.
Goal:	**F**.

4. By the definition of HLT,

$$(p, x) \in \text{HLT} \leftrightarrow p(x)\downarrow .$$

Known variables:	r (a program)
Know (1):	For all p and x, $r(p,x) \cong 1 \leftrightarrow p(x)\downarrow$.
Know (2):	For all p and x, $r(p,x) \cong 0 \leftrightarrow p(x)\uparrow$.
Goal:	**F**.

5. So far everything has been boilerplate for a proof by contradition. We have only used definitions. Now comes the inspiration. Every programmer knows how to write an infinite loop. We will allow ourselves to write "loop forever" in a program to indicate an infinite loop. Let's define program s as follows.

```
"{s(z):
   if r(z,z) = 1
      loop forever
   else
      return 1
}"
```

That program looks like it comes out of nowhere, but the discussion after this proof gives motivation for defining it. Program s is written to have two properties.

$$r(z,z) \cong 1 \rightarrow s(z)\uparrow .$$

$$r(z,z) \cong 0 \rightarrow s(z)\downarrow .$$

Both of those properties should be obvious from the definition of s.

Known variables:	r and s (two programs)
Know (1):	For all p and x, $r(p,x) \cong 1 \rightarrow p(x)\downarrow$.
Know (2):	For all p and x, $r(p,x) \cong 0 \rightarrow p(x)\uparrow$.
Know (3):	For all z, $r(z,z) \cong 1 \rightarrow s(z)\uparrow$.
Know (4):	For all z, $r(z,z) \cong 0 \rightarrow s(z)\downarrow$.
Goal:	**F**.

6. Since facts (1) and (2) hold for all p and x, they must hold for $p = s$ and $x = s$. Since facts (3) and (4) hold for all z, they must hold for $z = s$. (Here, we make use of the fact that program s is a string.) Making those substitutions yields the following knowledge.

Known variables:	r and s (two programs)
Know (1):	$r(s,s) \cong 1 \leftrightarrow s(s)\downarrow$.
Know (2):	$r(s,s) \cong 0 \leftrightarrow s(s)\uparrow$.
Know (3):	$r(s,s) \cong 1 \rightarrow s(s)\uparrow$.
Know (4):	$r(s,s) \cong 0 \rightarrow s(s)\downarrow$.
Goal:	**F**.

7. Using known facts (3) and then (2), we get

$$\begin{aligned} r(s,s) \cong 1 &\rightarrow s(s)\uparrow \\ &\rightarrow r(s,s) \cong 0 \end{aligned}$$

If $r(s,s) \cong 1$, that leads to a contradiction. ($r(s,s)$ cannot be both 1 and 0.) So it is not possible for $r(s,s) \cong 1$.

Using known facts (4) and then (1), we get

$$\begin{aligned} r((s,s)) \cong 0 &\rightarrow s(s)\downarrow \\ &\rightarrow r(s,s) \cong 1 \end{aligned}$$

If $r(s,s) \cong 0$, that also leads to a contradiction. So it is not possible for $r(s,s) \cong 0$.

But facts (1) and (2) tell us that $r(s,s)$ *must* be either 0 or 1. (After all, either $s(s)\uparrow$ or $s(s)\downarrow$.) So no matter what, we have reached a contradiction, and have proved **F**.

♢

What was the motivation for program s in step 5? Notice that, later in the proof, we are only concerned with what s does when its parameter z is s. But, when z is s, the definition of s look as follows.

"{$s(s)$:
  if $r(s, s) = 1$
    loop forever
  else
    return 1
}"

(That is not really allowed, since we cannot define a function with its parameter being itself, but let's allow it to understand where the definition of s comes from.)

Now remember that $r(p, x) \cong 1$ if and only if $p(x)\downarrow$ because r was chosen to be a program that solves the halting problem. In the if-statement, s asks r whether s halts on input s. If r says that s halts on input s, then s says, not I don't, and enters an infinite loop. If r says that s does not halt on input s, then s says, yes I do, and s halts and returns 1.

In fact, the proof is quite constructive in the sense that, for every program r that purports to solve the Halting Problem, the proof provides an input (s, s) that r answers incorrectly.

10.6 Diagonalization

The above proof that the Halting Problem is uncomputable uses pairs of strings of the form (s, s). If you think about points in the Cartesian plane, points of the form (x, x) are on the diagonal defined by equation $y = x$. Based on that analogy, the proof that the Halting Problem is uncomputable is called a *proof by diagonalization*.

prev **next**