

Chapter 7

Advanced Topics

We now address some more advanced topics: the idea of self-productibility of machines and its applications in proving decidability and constructing computer worms. This shows further the capability of a computation process. To show the incapability of a computation process, we also discuss the undecidability of certain logical theory.

Finally, we will talk more about the complexity theory, and its application in compiler construction, to further characterize the capability of a computation process.

Self Reproducibility

It seems to be true that when a machine A produces another machine B , A is more complicated than B .

For example, in a factory that produces cars with robots, the factory is more complicated than a car, since it has the design of that car.

If we accept this reasoning, then we conclude that machine cannot self produce since it can't be more complicated than itself.

An issue

We also generally agree that living things are machines because, based on modern biological theory, organisms behave in a mechanical way.

Thus, based on the previous reasoning, living things cannot self produce, which certainly contradicts the facts.

The way out of the box is that the producer need not be more complex than the produced. We will see how a machine can produce its own description, and moreover, how to use its own description to do something useful or evil.

The recursion theorem tells us *how* to achieve this self-production.

A Lemma

We firstly show that there is a computable function $q : \Sigma^* \mapsto \Sigma^*$, that for any string w , $q(w)$ is the description of a TM P_w that prints out w then halts.

Below is such a machine computing the function q :

0. Given the input w
1. Construct the following machine P_w :
“On any input:
 1. Erase input
 2. Write w on the tape
 3. Halt.”
2. Output $\langle P_w \rangle$.

A machine *SELF*

We now make a TM *SELF* that ignores its input and prints out a copy of its own description. This machine *SELF* is made up of two parts, *A* and *B*. Thus, we will let *SELF* print out $\langle AB \rangle = \langle A \rangle \langle B \rangle$.

When *SELF* runs, *A* runs first, and when it is done, it passes control to *B*. More specifically, *A* is to print out a description of *B*, while *B* is to print out a description of *A*, put together, we have the description of the whole thing.

A is just $P_{\langle B \rangle}$ as described in the lemma, which prints out $\langle B \rangle$. The problem is, before we can print out a description of *B*, we have to know *it* first, which requires us to construct *B*.

It is obvious that *B* cannot be $P_{\langle A \rangle}$. Otherwise, we will have a circular definition.

Once B is done, so is $SELF$.

B is to print out a description of A . Since we define A to be $P_{\langle B \rangle}$, if B knows $\langle B \rangle$, the only thing B needs to do is simply apply the function q , as discussed in the lemma, gets $q(\langle B \rangle)$, which by the lemma, is the description of $P_{\langle B \rangle} = \langle A \rangle$, then print it out.

The question now is how to let B know $\langle B \rangle$? It is tricky but also trivial, since when $A = P_{\langle B \rangle}$ completes, it simply leaves $\langle B \rangle$ on the tape for B to read.

Thus, once A is done, B picks up A 's output, apply q to obtain $\langle A \rangle$ and add it to the front of the tape to construct $\langle A \rangle \langle B \rangle = \langle AB \rangle$.

The machine *SELF*

More specifically, we have that $SELF = A, B$; where $A = P_{\langle B \rangle}$, and B is described as follows:

0. Given the input $\langle M \rangle$, part of a description of a TM
1. Compute $q(\langle M \rangle)$
2. Combine $q(\langle M \rangle)$ and $\langle M \rangle$ to make a description of a TM
3. Print out this description

Thus, when we run *SELF*, A runs first, and prints $\langle B \rangle$ on the tape. When B starts, it finds $\langle B \rangle$ on the tape, gets $q(\langle B \rangle) (= \langle A \rangle)$, then combines $\langle A \rangle$ with $\langle B \rangle$ to obtain $\langle AB \rangle = SELF$.

Self reference (I)

We can code the *SELF* machine in any programming language to obtain a program to print out a copy of itself. We can even do it in English.

The following instruction asks us to produce an exact copy of itself.

Print out this sentence.

Notice that the word 'this' in the above sentence plays a self-referential role, which is really not needed in this case.

An alternative could be the following:

Print out two copies of the following, the second one in quotes: "Print out two copies of the following, the second one in quotes:"

This latter construction is the same as we used in constructing the machine *SELF*. Here, part *B* produces the following

Print out two copies of the following,
the second one in quotes:

and part *A* produces the same, but in quotes, i.e., a coding of the above instruction.

When run, *A* provides a coding of *B*, i.e., “Print out two copies of the following, the second one in quotes:” to *B* as the input, then *B* restores the message to the original instruction, i.e.,

Print out two copies of the following,
the second one in quotes:

and combines the two together to obtain the final program

Print out two copies of the following, the second one in quotes: “Print out two copies of the following, the second one in quotes:”

Notice here B produces a program, while A produces it in quotes, more like a coding of a program. They are actually the outputs of B and A , respectively.

Self reference (II)

With the *SELF* machine, once finding out its own description, the machine, *B*, only prints it out. The recursion theorem moves another step forward, the machine can actually do something with its own description.

In other words, the recursion theorem adds the ability of self-referential *this* into any programming language. With this facility, any program can refer to itself, to be exact, its own description, which is useful as we well see.

Recursion theorem

Let T be a TM that computes a function $t : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$. There is a TM R that computes a function $r : \Sigma^* \mapsto \Sigma^*$, where for every w ,

$$r(w) = t(\langle R \rangle, w).$$

This theorem says that for any machine T which, given any two inputs, can produce one output, we can construct a machine R , that will operate exactly as T does, but automatically throw itself as the first input to such a machine.

Proof: It is a simple extension to *SELF*. With *SELF*, it consists of two parts, *A* and *B*, *A* prints out $\langle B \rangle$, and *B* simply *print out* an encryption of the process in which *A* prints out *B*'s description.

But, once a machine finds out itself, it can also do some more complex computation than merely printing it out.

In this case, $R=ABT$, where $A=P_{\langle BT \rangle}$, when *A* completes, it leaves $\langle BT \rangle$ in the tape. *B* checks the tape, applies q to $\langle BT \rangle$ to obtain $q(\langle BT \rangle) = \langle A \rangle$, combines *A*, *B* and *T* and leave it in the tape, then *passes the control to T*.

When *T* starts, it has two inputs, one is the description of $\langle ABT \rangle = \langle R \rangle$, and the other is simply w . Upon its completion, it certainly produces $t(\langle R \rangle, w)$.

The essence of worm

A *worm* is a program that is designed to spread itself to other machines, while a *virus* adds itself to other programs. A very important feature of a worm is that once spread to another machine, it stays infectious in the sense that it still has the ability to be spread again.

What essentially happens is that when it is spread, this program is self copied, or self produced.

Recursion theorem and worm

Let T be an algorithm, consisting of three steps:

1. Do whatever it wants with its input;
2. Find out IP addresses from, e.g., the address book of MS Outlooker;
3. Send its first input, as a message, to each and every of these addresses.

Then, by the Recursion theorem, and the Church-Turing Thesis, there exists a program, R , with only one input. R will do exactly what T does, but will always fill in its encryption as the first input of T , and thus this same program R will be transmitted to whatever IP addresses it finds out, and the same stuff happens over *there*.

This is the basis of all the worms.

A_{TM} is not decidable

Another application is that now we can state, in designing algorithm, that a machine can obtain its own description, via the recursion theorem. For example, we once saw that both A_{DFA} and A_{CFG} are decidable, but A_{TM} is not decidable. We now have another proof for the last result.

Just assume H decides A_{TM} , namely, given any machine M and w , H is able to decide if M accepts w .

We now construct the following machine B :
on any given w , B does the following:

1. Obtain its own description $\langle B \rangle$
2. Run H on input $\langle \langle B \rangle, w \rangle$
3. Do the opposite of what H says, *accept* if H rejects and *reject* if H accepts

Since H gives the exactly opposite answer as B about whether B accepts w , H can't decide $\langle B, w \rangle$. Hence, in general, H does not decide A_{TM} .

Undecidability of logical theories

Mathematical logic deals with such issues as statement, theorem, truth, proof, and tries to answer such questions as can an algorithm decide which statements are true, are all true statements provable, etc..

It is clear that the truth of some statements are not decidable. For example,

This statement is not true.

Assume that a decider states that it is true, then by this very statement, it is not true; on the other hand, if the decider states that it is not true, then, by this very statement, it is true.

What is a statement?

To rigorously discuss such matters, we need to have a precise definition of a statement. This definition is given in the first part of §6.2. Essentially, we start with some basic *relations*, then get to the general statements by applying logical operations such as \vee , \wedge , and \neg , as well as quantifiers such as \forall and \exists .

Below are some examples:

$$\forall q \exists p \forall x \forall y [p > q \wedge (x, y > 1 \rightarrow xy \neq p)]$$

$$\forall a \forall b \forall c \forall n [a, b, c > 0 \wedge n > 2 \rightarrow a^n + b^n \neq c^n]$$

What is a true statement?

We can give *interpretation* to each and every relation we use in our statement, i.e., what does that relation mean. For example, for '=', we typically mean 'equality'.

Once we give interpretation of all the relations, we provide a *model* for that statement. Then a statement is either true or false according to that model.

For example, the above first statement according to the usual interpretations of all the involved relations, means that "there are infinite amount of prime numbers", which has been known to be true for over 2,300 years.

The second, known as Fermat's Last Theorem, stating that all positive a, b, c , and $n > 2$, $a^n + b^n \neq c^n$, has been known to be true only for the last few years.

Finally, $\forall x(x \neq x)$ is certainly false in any model that interpretes the relation '=' in the normal way.

Given a model, i.e., interpretations of all the relations, some of the statements consisting of those relations could be true, others could be false. We use $\text{Th}(\mathcal{M})$ to denote the collection of all the true statements according to a model \mathcal{M} .

Decidable theory

Number theory is a very basic and important branch of mathematics that deals with \mathcal{N} , the set of non-negative numbers.

By $\text{Th}(\mathcal{N}, +)$, we mean the logical theory of the model $(\mathcal{N}, +)$, where \mathcal{N} refers to all the non-negative numbers, and $+$ is interpreted as the normal addition relation. Thus, $\forall x \exists y (x + x = y)$ is a member of this theory, while $\exists y \forall x (x + x = y)$ is not.

By $\text{Th}(\mathcal{N}, +, \times)$, we mean the logical theory of the model $(\mathcal{N}, +, \times)$, where \mathcal{N} refers to all the non-negative numbers, $+$ and \times are interpreted as the normal addition and multiplication.

For example, the following statement

$$\forall x \forall y \forall z [x \times (y + z) = x \times y + x \times z]$$

is a member of this theory; while

$$\forall x \forall y \forall z [x + (y \times z) = (x + y) \times (x + z)]$$

is not.

Is a theory decidable?

Generally speaking, a logical theory is a set of statements. By a *decidable theory*, we mean a set of statements for which there is a TM that can tell, given a statement in this theory, it is either true or false.

Theorem 6.9 proves that $\text{Th}(\mathcal{N}, +)$ is decidable; while Church showed that $\text{Th}(\mathcal{N}, +, \times)$ is undecidable, i.e., there does not exist a TM, which can tell the truth of all the statements about natural numbers involving only '+' and '×'.

Church's result is fundamental since it shows that mathematics is not mechanistic.

Proof of a theorem

Loosely speaking, the proof of a statement ϕ , is a sequence of statements $S_0, S_1, \dots, S_n (= \phi)$.

Each of the S_i is either an basic *axiom*, such as $\forall x(x = x)$, or follows from the previous statements by applying one of the implication rules, such as $[A \wedge (A \rightarrow C)] \implies C$.

It is clear then that any proof of a statement can be mechanically verified. Thus, the collection of theorems is decidable.

Moreover, any statement that can be proved is true, since all the axioms are true and any statement obtained by applying any inference rule on true statements is also true.

Is every true statement provable?

Kurt Gödel's *Incompleteness Theorem* shows that in any reasonable system formalizing the notion of provability of number theory, some true statement is not provable.

Theorem: Some true statement in $\text{Th}(\mathcal{N}, +, \times)$ is not provable.

Proof: Just assume every true statement in the theory is provable. We construct a TM P , for any given statement ϕ , P systematically checks any string to see if it is a proof of ϕ . P accepts ϕ if some string is a proof of it.

Now the real proof starts. Given any ϕ , we now apply P to both ϕ and $\neg\phi$. Since one of the two must be true, thus, by our assumption, is provable. Assume ϕ is true, P will eventually come up with a proof for it. Otherwise, P will come up with a proof for $\neg\phi$.

This process leads to a TM to decide $\text{Th}(\mathcal{N}, +, \times)$, which contradicts a previous result.

Therefore, *this* theorem is proved.