Chapter 7
Advanced Topics

We now address some more advanced topics: the idea of self-producibility of machines and its applications in proving decidability related results, and constructing computer worms ☹.

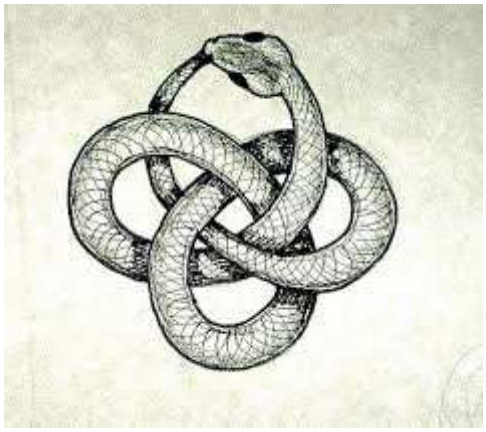
This shows further the capability of a computation process: a seemingly challenging problem is actually doable. ☺

Throughout this course, we applied various logic reasoning ideas, e.g., $A \rightarrow B$ iff $\neg B \rightarrow \neg A$. (Again? ☹)

To show the incapability of a computation process, we discuss the undecidability of certain logic theory: Although everything that can be proved must be "true" ☺; *it is not the case that everything "true" can be proved* ☹.

Self consumption vs self production

The essential logic of diagnolization is *self consumption:* What happens when we send the description of a machine to itself as an input?



It is a logic fallacy, which leads to the logic contradiction that we need to show the unsolvability of the A_{TM} problem (Cf. Pages 29 through 34 of the *Decidability* notes).

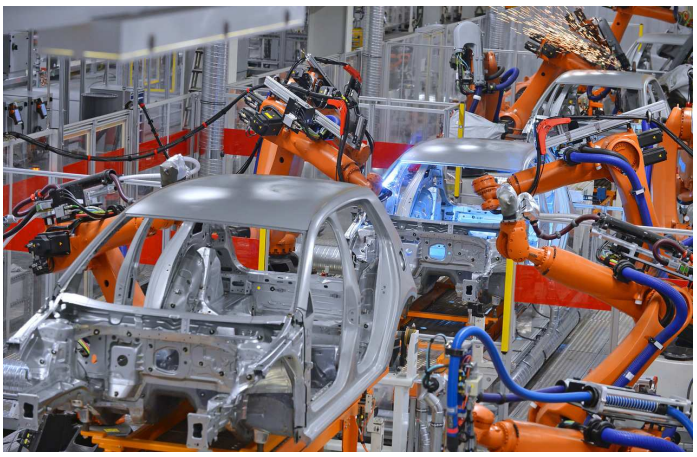
Question: How about its opposite? *Can a "machine" self produce?*

Answer: Yes. We cannot eat ourselves, but we can give birth to our children.

Self reproducibility

It seems true that when a machine A produces another machine B, A is more complicated than B. For example, a chef is more complicated than the dishes, since she knows the recipes, the design of all such dishes.

In a factory where cars are made with robots, the factory should be more complicated than cars, since it has the designs of all the cars.



It seems that we have to conclude that machine cannot self produce since it cannot be more complicated than *itself*. ☹ Don't we?

Perhaps not...

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

Based on the previous reasoning, living things cannot self produce, which certainly contradicts the facts: We now know how to print out our own organisms chart (1990-2003). *In other words, we can print out our own descriptions.* (Cf. Course page)

The way out of the box is that *a producer needs not be more complex than the produced.*

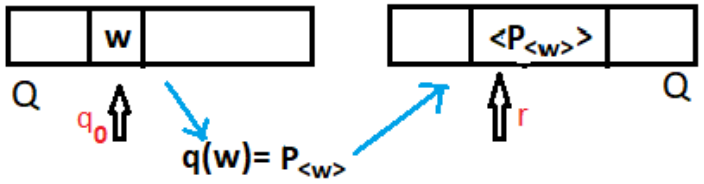
Technically speaking, the *recursion theorem* tells us *how* to achieve this self-production.

We will see how a machine can produce its own description, and moreover, how to use its own description to do something good ☺, or evil ☹.

The first step

We can construct the following TM, Q, that computes a function $q : \Sigma^* \mapsto \Sigma^*$ that, for any string $w, q(w)$, the corresponding output of q, is *the description of a TM*, P_w, that prints out w then halts.

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 description of a TM.
3. Stop.



The relationship between a machine and its description is the same as that between a baked turkey and the recipe that we followed. ☺

A machine *SELF*

We now make a TM *SELF* that ignores its input and prints out a copy of its own description $\langle SELF \rangle$.

SELF is made up of two parts, A and B. When *SELF* runs, A runs first to *print out,* $\langle B \rangle$, a description of B; then passes control to B. B prints out $\langle A \rangle$, a description of A. Thus, *SELF* prints out $\langle A \rangle \langle B \rangle = \langle AB \rangle = \langle SELF \rangle$.

A is just $P_{\langle B \rangle}$, making use of Q, as discussed on Page 5, which prints out $\langle B \rangle$. The problem is, *before* we can print out a description of B, we have to know *it*, which requires us to construct B *first*. Before we can eat a turkey, it has to be put on the table already.

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

"A hill is a smaller mountain, and a mountain is a bigger hill.", What is a hill? ☺

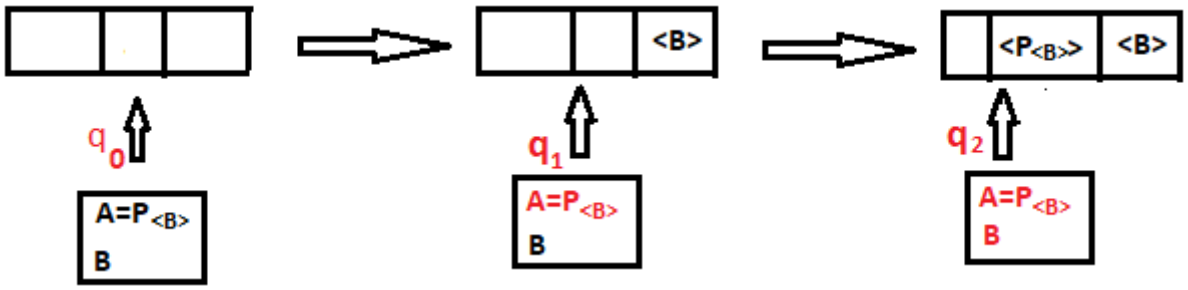
Once B is done, so is *SELF*.

A is to print out $\langle B \rangle$ and B is to print out $\langle A \rangle$, a description of A, in front of $\langle B \rangle$.

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 to apply the computable function q, making use of the machine Q, to its own description, gets $q(\langle B \rangle)$, which, by the lemma, is a description of a TM, $q(\langle B \rangle) = \langle P_{\langle B \rangle} \rangle = \langle A \rangle$, then prints *it* out.

Questions: How to let B know about its own description, i.e., $\langle B \rangle$?

Answer: It is tricky but also trivial: When $A{=}P_{\langle B \rangle}$ completes, it simply leaves $\langle B \rangle$ on the tape for B to read.



The machine *SELF*

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. Use Q to compute $q(\langle M \rangle)$, a description of a machine that prints out $\langle M \rangle$, i.e., $\langle P_{\langle M \rangle} \rangle$
2. Print out $\langle P_{\langle M \rangle} \rangle$
3. Stop

Recap: How does *SELF* work?

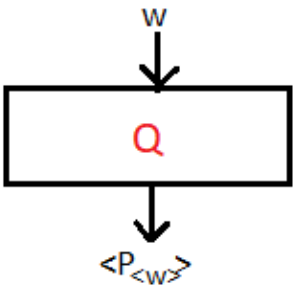
When *SELF* runs, A runs first, and prints $\langle B \rangle$, i.e., a description of a TM corresponding to the above algorithm, on the tape.

B then runs, with $\langle B \rangle$ as its input. It applies q, the computable function, (Cf. Page 5) to $\langle B \rangle$ to get $q(\langle B \rangle)(= \langle P_{\langle B \rangle} \rangle)$, which is exactly $\langle A \rangle$.

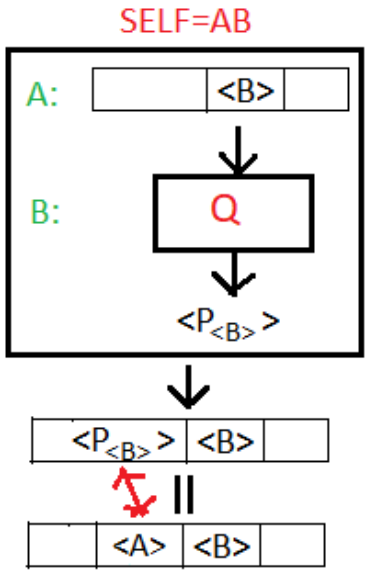
Thus, the output of *SELF* is $\langle A \rangle$ followed by $\langle B \rangle$, i.e., $\langle AB \rangle = \langle A \rangle \langle B \rangle = \langle \mathit{SELF} \rangle$.

I want to see ...

Below is what the printer, `Q`, sends out, a description of a machine that prints out any input w. Check out Page 5 about *how* it does *it*.



This is how the `SELF` prints out its own description. Check out Page 8 as *how* it does *it*.



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. It is even available in *Java*, but this word of "this" 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:"

How does this work out?

This latter construction is similar to what we used in constructing the machine *SELF,* although in an opposite order, printing out Part 1 then Part 2; while with *SELF,* the second part is printed first.

When running a TM M, someone provides a coding, i.e., a description, of M: "Print out two copies of the following, the second one in quotes: 'Print out two copies of the following, the second one in quotes:" ', and leave this on the tape for M to pick up.

When M starts to work, it processes the copy that it gets from the tape.

Notice here M functions like a program, or a TM; obtaining the coding, or a description, of a program in quotes.

Self reference (II)

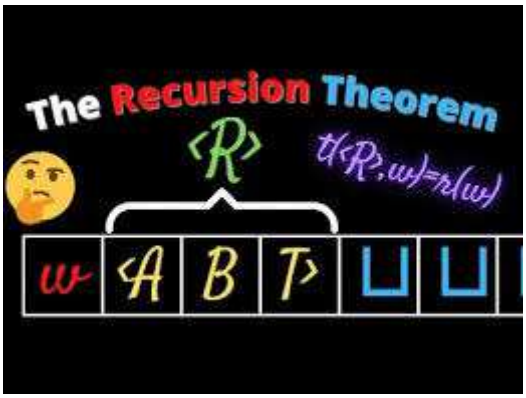
With the *SELF* machine, once finding out its own description, the machine, B, only prints it out. We can certainly move another step forward: do something *with its own description.*

This is what the *recursion theorem* does: it adds the ability of self-referential of "*this*" into any programming language. With this facility, any program can refer to itself, to be exact, its own description.

This ability could be useful as we will see, but not always in a positive sense. ☹

Recursion theorem

Let T be a TM that computes a function $t : \Sigma^* \times \Sigma^* \mapsto \Sigma^*$, with two inputs. Then we can build a TM R that computes a function $r : \Sigma^* \mapsto \Sigma^*$, with just one input, where for every w, $r(w) = t(\langle R \rangle, w)$, where $\langle R \rangle$ describes itself.



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

Question: How does this work?

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* $\langle A \rangle$, an encryption of the process in which A ($= \langle P_{\langle B \rangle} \rangle$)) prints out B's description. End of story.

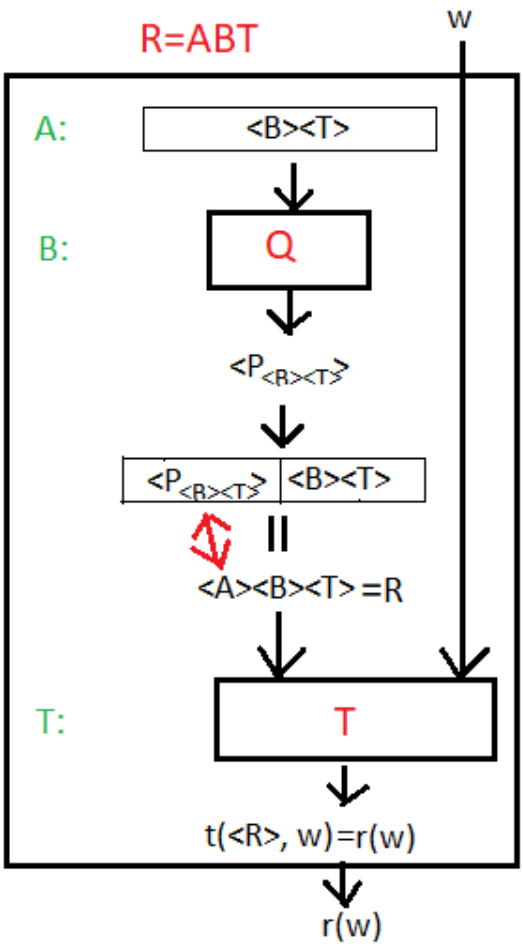
But, once a machine identifies itself, it can also do some more complex computation than just printing out itself.

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 $q(\langle BT \rangle) = \langle P_{BT} \rangle = \langle A \rangle$, combines $\langle A \rangle$, $\langle B \rangle$ and $\langle T \rangle$ and leave *it,* $R = ABT$, on the tape, then *passes the control to* T, before it stops.

When T starts, it has two inputs, one is the description of $\langle ABT \rangle = \langle R \rangle$, and the other is simply w. T proceeds to produce $t(\langle R \rangle, w)$. $\square$

Question: Isn't this cute? ☺ But, what does this mean? ☹

I want to see...



Notice that, at the end, we simply take $t(\langle R\rangle, w)$ as the value of $r(w)$.

This is not necessary: Once we get $t(\langle R\rangle, w)$, we can do anything about it, and send out $r(w)$, or not.

Application: The essence of a 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 it is spread to another machine, it stays infectious in the sense that it still has the ability to be spread again.

So, a worm stays contagious. ☹

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, with two inputs, that does the following:

1. Find out IP addresses stored in a machine identified by the second input, m, e.g., its IP address.

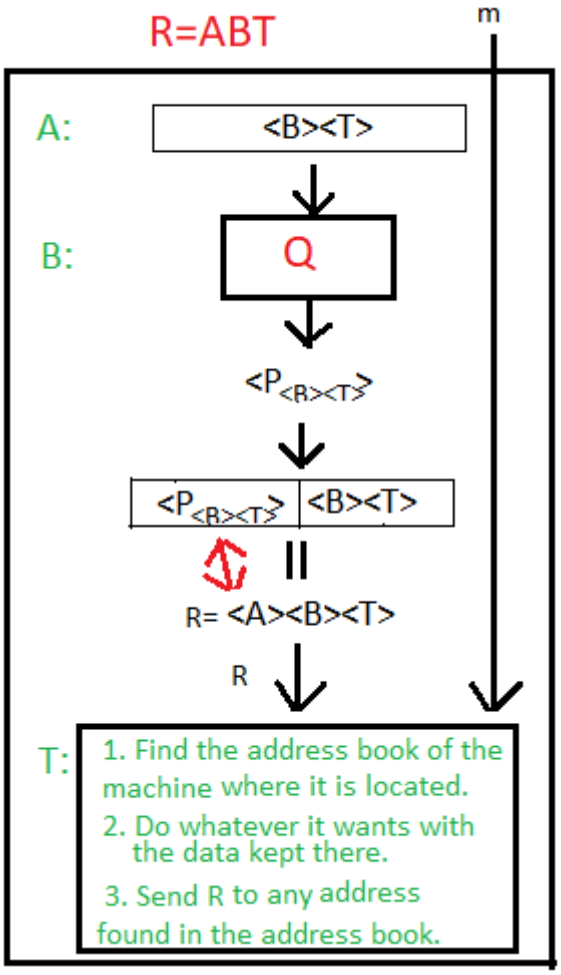
2. Do whatever it wants with m, e.g., erase its hard disk, or send over a ransom message.

3. Send its first input, as a message, to each and every one of these addresses found in Step 1.

By the Recursion theorem, and the Church-Turing Thesis, there exists a program, R, with only one input, e.g., the address a machine m.

R will do exactly what T does with its only input m, but will always fill in its encryption as the first input of T.

I want to see...

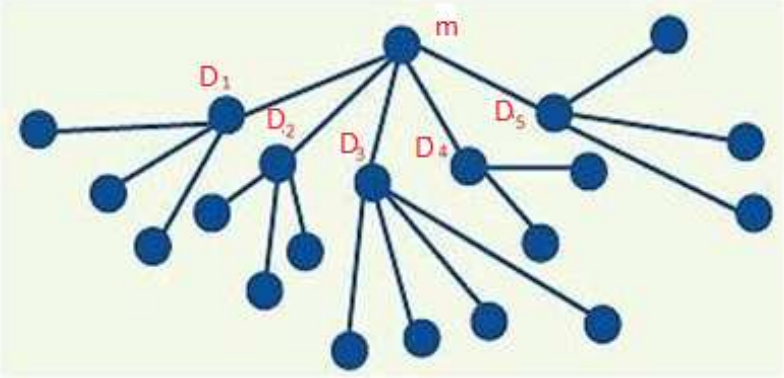


Notice that, in this case, `T` sends nothing out as its result, nor does `R`.

Question: How does `R` stay contagious?

It will go over...

The machine `R` starts at a machine with its IP address being m, and then acts with the machine `T`. It then sends `R` to all the machines with their addresses found in the address book kept in machine m, i.e., D_1, D_2, D_3, D_4, D_5, and each will do the same as R does.

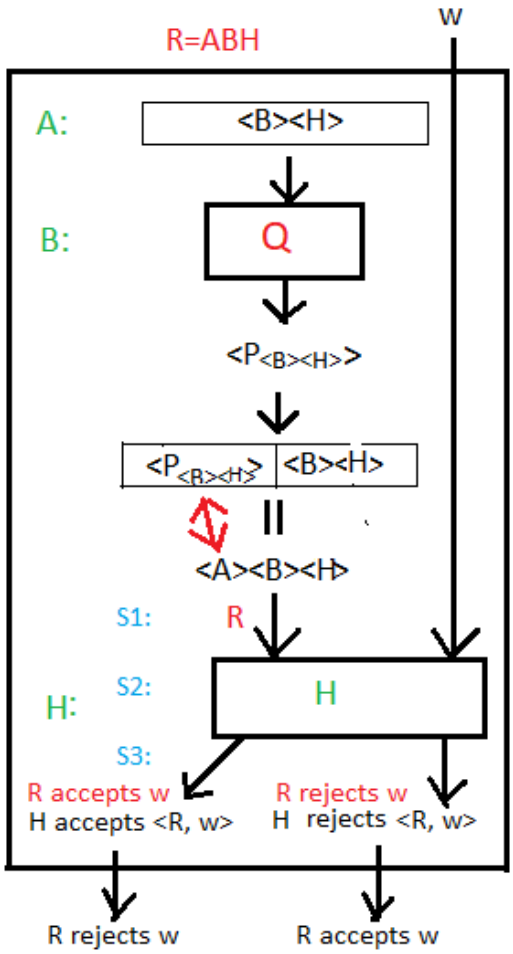


Thus this same program `R` will be transmitted throughout the entire network.

This is the basis of all the worms. ☹

A_{TM} is not decidable

We once followed the diagnolization technique to show that A_{TM} is not decidable.(Cf. Pages 29 through 34 in the *Decidability* notes). We now have another proof for the last result, using this self-referential capability.



What does *it* mean?

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

By the Recursion theorem, using H as the TM T, there is TM, R, $r(w) = H(\langle R \rangle, w)$. we now construct the following machine R. On any given w, R does the following:

1. Obtain its own description $\langle R \rangle$
2. Run H on input $\langle R, w \rangle$
3. R *accept* w if H rejects $\langle R, w \rangle$, i.e., R rejects w; and R *reject* w if H accepts $\langle R, w \rangle$, i.e., R accepts w. Notice that $r(w) = H(\langle R \rangle, w)$.

Notice also, in this case, R does the opposite of whatever H does, as suggested on Page 15.

Thus, R accepts w iff R does not accept w.

This is a clear contradiction. Hence, in general, no such an H that decides A_{TM}.

Undecidability of logical theories

Mathematical logic deals with such issues as statement, theorem, truth, proof, etc..

There are essentially two central notions: "Is something true?", and "Is something provable?" The former involves *model theory*: what does *that* mean by "truth"? The latter involves *proof theory*: How to prove a statement, based on axioms and deduction rules?

Regarding their relationship, one thing is clear: everything that can be proved must be "true".

Essentially, all the axioms are "true", e.g.,

$$\forall x\, x = x.$$

and rules bring one "true" statement to another, e.g., $\{A, A \to B\} \to B$.

This is the very basis of all the mathematical proof, starting from Aristotle's syllogism, deduction, proof by contradiction, induction, etc.. (Check out the stuff in the course page.)

Two undecidable problems

One problem is that, given a statement, can we decide if it is "true"?

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

> "This is not true."

Assume that a decider states that it is true, then by this very statement, it is false; on the other hand, if the decider states that it is false, then, by this very statement, it is true. Thus, this statement is neither true nor false. ☹

Question: Can we decide a more specific statement in a more specific context?

Question: Does every true statement have a proof?

What are we talking about?

To rigorously discuss such matters, we need to have a precise definition of a statement.

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:

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

The above states that there does not exist a maximum prime number. In other words, there are infinite number of prime numbers.

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

This one just states Fermat's Last Theorem, which was suggested by Pierre de Fermat in 1637, and finally proved in 1995 by Andrew Wiles.

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 to all the relations, we have provided 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 $n > 2$, $a^n + b^n \neq c^n$, although it is clear this one holds for $n = 2$ for all right triangles, i.e., the Pythagorean theorem.

Finally, $\forall x \ (x \neq x)$ is certainly false in any model that interprets 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 Th($\mathcal{M}$) to denote the collection of all the true statements according to a model $\mathcal{M}$.

Then, to decide the truth of a statement in terms of $\mathcal{M}$ is to decide whether it falls into Th($\mathcal{M}$).

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 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, as it intuitively means that any number, when doubled, is even.

On the other side, $\exists y \forall x (x + x = y)$ is not; otherwise, it would mean there is exactly one even number. ☺

Indeed, just let $x = y + 1$, we have $x + x = 2y + 2 \neq y$, which is a counter example to the above statement.

By $\mathrm{Th}(\mathcal{N}, +, \times)$, 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, since this is just the distributive property of multiplication over addition; while

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

is not, since addition is not distributive over multiplication ☹.

Although this property does hold for Boolean logic.

$$z \vee (x \wedge y) = (z \vee x) \wedge (z \vee y).$$

Question: Do you trust Dr. Shen? ☺

Let's find it out...

... with the following truth table.

x	y	z	$x \wedge y$	$(x \wedge y) \vee z$	$x \vee z$	$y \vee z$	$(x \vee z) \wedge (y \vee z)$
0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	1
0	1	0	0	0	0	1	0
0	1	1	0	1	1	1	1
1	0	0	0	0	1	0	0
1	0	1	0	1	1	1	1
1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1

It is thus clear that

$$(x \wedge y) \vee z = (x \vee z) \wedge (y \vee z).$$

As a result, because both '$\wedge$' and '$\vee$' are commutative operators,

$$z \vee (x \wedge y) = (z \vee x) \wedge (z \vee y).$$

Thus the statement that "Dr.Ṡhen always lies" is false... . ☺

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 6.12: $\mathrm{Th}(\mathcal{N}, +)$ is decidable. , Alonzo Church, the same one as in *Church-Turing Thesis,* showed that, as stated in Theorem 6.13 in the textbook, that $\mathrm{Th}(\mathcal{N}, +, \times)$ 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 '$\times$'.

Church's result is fundamental since it shows that mathematics is not mechanic, thus mathematicians are needed. ☺

Indeed, we are hiring two right now.

Proof of a theorem

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

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 $[A \wedge (A \rightarrow B)] \rightarrow B$.

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

It turns out that *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.

Question: What about the other side of the coin?

Is every true statement provable?

Kurt Gödel's *Incompleteness Theorem* (1931) 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 $\mathrm{Th}(\mathcal{N}, +, \times)$ is not provable.

Proof: Just assume the opposite: Every true statement in the theory is provable.

We now construct a TM, P, for any given statement ϕ, P systematically checks any string to see if it is a proof of ϕ, as defined on Page 31, and accepts ϕ if some string is a proof of ϕ.

Now the real proof for this result starts. Given any ϕ, we apply P to both ϕ and $\neg\phi$.

Since one of the two must be true, it is thus provable by our assumption.

Assume ϕ is true, P will eventually come up with a proof for it. Otherwise, P will come up with a proof for $\neg\phi$. When either case happens, we know either ϕ is true or false.

Thus, such a TM decides $\mathrm{Th}(\mathcal{N}, +, \times)$, which contradicts Church's result, as given in **Theorem 6.13**.

Therefore, *this* theorem is proved, i.e., it is not the case that every true statement is provable.

Well, with this last example of

$$A \rightarrow B \text{ iff } \neg B \rightarrow \neg A,$$

we have to wrap up this course *here* .... ☺