CS70: Computability

Yining Liu

July 9, 2018

This document is largely adapted from CS70 computability notes. My knowledge of computability is also limited to the level of CS70. This document consists of what I learned about this topic after I took CS70. In that sense, it might not be comprehensive, and some of the language is probably not 100% accurate (as always, any correction or suggestion is welcomed!). I hope it will serve as a nice introduction to this topic.

1 The Halting Problem

Let's take another look at the famous **Halting Problem**: given a program P and an input x, does $P(x)$ halts? In other words, I want to see whether it is possible to write a program **TestHalt (P, x)** that does the following:

Algorithm 1 TestHalt

```
1: function TestHalt(P, x)
2:     if P(x) halts then
3:         return True
4:     else
5:         return False
```

Our goal is showing TestHalt is not computable, i.e. it does not exist. In order to show it doesn't exist, I'm actually going to assume TestHalt exists, and *use* it to define another program called **Turing**.

Algorithm 2 Turing

```
1: function Turing(P)
2:     if TestHalt(P, P) is True then
3:         loop forever
4:     else
5:         halt
```

Now, I'm going to take a small digression here and take a look at the set of all computer programs. Each computer program can be thought of a bit string

with finite length. Thus, the set of all computer programs are countable. So, I can enumerate all computer programs and put them into a table like this:

	P_0	P_1	P_2	$\cdots$
P_0				
P_1				
P_2				
$\vdots$				

Another observation is, since all computer programs are bit strings, so it's totally okay if I pass in one P_i as the argument into another P_j. I'm actually going to fill out my table like this: for each of the (i, j)-entry in my table, I'm calling $P_i(P_j)$ and see whether it halts - if it halts, I'm putting a H into the (i, j)-entry; if it doesn't halt, I'm putting a L into the (i, j)-entry.

For example, first I'm going to try $P_0(P_0)$ and check whether it halts. Let's say it does, so I'm putting a H in $(0, 0)$. If I keep doing this, I'm going to fill up my table and get something like this:

	P_0	P_1	P_2	$\cdots$
P_0	H	L	L	$\cdots$
P_1	L	H	H	$\cdots$
P_2	H	H	H	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

If **Turing** exists, then it has to appear somewhere in the table. Let's take a look at whether its box has filled in with a L or a H.

	P_0	P_1	P_2	$\cdots$	Turing	$\cdots$
P_0	H	L	L	$\cdots$		$\cdots$
P_1	L	H	H	$\cdots$		$\cdots$
P_2	H	H	H	$\cdots$		$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$		$\cdots$
Turing					?????	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Case 1: Fill in H This means Turing(Turing) **halts**. If we go back and check how we define Turing, this means the first *if* condition returns False. This is telling us TestHalt(Turing, Turing) returns False. If we go back and check how we define TestHalt, TestHalt returns False only if Turing(Turing) **does not halt**. Contradiction, so we cannot fill in H.

Case 2: Fill in L Using the similar argument (*Exercise: how do you arrive with the contradiction here?*), we cannot fill in L either.

Thus, Turing does not exist on the list, which means it is not computable.

2 Computability

Let's take another look at how we define Turing. *Where exactly does the problem arise?* It's prefectly fine to check whether a program returns True; it's also totally okay to write something that loops forever or halts. The problem is because **TestHalt** does not exist, i.e. uncomputable.

2.1 Reduction

What's the relationship between *Turing* and *TestHalt*? We used TestHalt to implement Turing, so we can say Turing **reduces** to TestHalt. Which of them is harder? I would argue TestHalt is harder - if you give me a function that does what TestHalt does, it would be very easy for me to implement Turing.

In general, if a program A reduces to another program B (i.e. $A \rightarrow B$), then that means you can use B to solve A. In that sense, B is "harder" than A.

2.2 Proof for uncomputability

Now, we can see how to prove a program P is uncomputable. We know TestHalt(P, x) doesn't exist. Intuitively, if P is "harder" than TestHalt, then P cannot be computable neither. More formally, if you want to prove P is uncomputable, you want to implement TestHalt using P. [1] This is the same as saying, if you want to prove P is uncomputable, reduce TestHalt [2] to P .

[1] This is essentially the following argument: Assume P exists. Then, TestHalt also exists (using your implementaion). Contradiction. Thus P does not exist.

[2] Do not reduce Turing to P. You can forget about Turing now. It has served its purpose.