Tiane Zhu

CSCI3390 Topics in Computer Science
Instructor: Professor Howard Straubing

Explicit Construction of a Self-Reproducing Machine

Introduction:

This report aims at demonstrating the construction of a self-reproducing turing machine, and the procedure of building its specification. I used 2 versions of Turing machine simulator: a version by Professor Howard Straubing from Boston College, written in python, and a version of the simulator translated into $C++$ by me. The number of steps of Turing Machine execution is in $\mathcal{O}(n^2)$ while the coefficient is hard to estimate as we will see later we are copying things from left end to right end and from right end the left end over and over again.Just for a reference, I built about 5 versions of the specification files: machine version four when feeding its encoded specification file into itself runs for 5 hours in the python version of the software.

The construction in this paper is being outlined Michael Sipser's book: Introduction to the Theory of Computation. In the appendix, there is a small C program from Ken Thompson's famous article, "Reflections on trusting trust",and a small $Java$ Program that employs the same idea. Both of them would be used to illustrate the recursion Theorem in a higher level language.

The problem

We will show that, regardless of the input string, there exists a Turing Machine $\mathcal{M}$ that can write itself on a tape; and we will explicitly construct such a Turing Machine, by providing specification files that can be simulated using our provided Turing Machine simulator. And we will briefly extend the recursion theorem to illustrate the possibility of the machine doing more than just self-reproducing.

The Construction

The Construction Theory

Since our construction employs the idea from Michael Sipser's book, "Introduction to the Theory of Computation", we begin the discussion by listing out the details that has been provided in the book.

(1) Let w be a string defined on Σ^*, $\mathcal{P}_w$ be a Turing machine that on any input string erases the input and writes w on the tape.

(2) Let $\mathcal{Q}$ be a TM that, on input string w, outputs the specification of a TM, $\mathcal{P}_w$, denoted as $< \mathcal{P}_w >$. Also denote the function computed by $\mathcal{Q}$ as

$$q : \Sigma^* \rightarrow \Sigma^*.$$

(3) Then actual construction begins:
   (a) Let $\mathcal{A} = \mathcal{P}_{<\mathcal{B}>}$.
   (b) Let $\mathcal{B}$ be a machine that on input string $< \mathcal{T} >$, where $\mathcal{T}$ is a portion of a TM,
      (i) computes $q(< \mathcal{T} >)$, writes $< \mathcal{P}_{<\mathcal{T}>} >$ on tape.
      (ii) concatenates $< \mathcal{P}_{<\mathcal{T}>} >$ with $< \mathcal{T} >$; specifically, concatenation also entails changing the initial state of $\mathcal{T}$ to the state before the halt state of $\mathcal{P}_{<\mathcal{T}>}$.
      (iii) outputs $< \mathcal{P}_{<\mathcal{T}>} >< \mathcal{T} >$. (State adjusted), we name this output TM, $\mathcal{SELF}$.

The encoding $< \mathcal{T} >$ that we feed into $\mathcal{B}$ is the following:

$$< A >=< \mathcal{P}_{<\mathcal{B}>} >$$

So by running $\mathcal{B}$ on this input, it prints out a machine, say $\mathcal{B}'$ that can print out $< \mathcal{P}_{<\mathcal{B}>} >$, and does what $\mathcal{P}_{<\mathcal{B}>}$ does.

$$< \mathcal{B}' >=< \mathcal{P}_{<\mathcal{P}_{<\mathcal{B}>}>} >< \mathcal{P}_{<\mathcal{B}>} >$$

By running $\mathcal{B}'$, we can get

$$< \mathcal{SELF} >=< \mathcal{P}_{<\mathcal{B}>} >< \mathcal{B} > .$$

Then, if we run self, we first print out $< \mathcal{B} >$ and does what $\mathcal{B}$ does on $< \mathcal{B} >$, outputs $< \mathcal{P}_{<\mathcal{B}>} >$ then concatenate it with $< \mathcal{B} >$ with initial state of $\mathcal{B}$ adjusted. Then we are left with

$$< \mathcal{SELF} >$$

on the tape.

To illustrate, in terms of the appended programs in higher level languages, the char array in the C program and the string array in the java program are two versions of the encoding $< \mathcal{T} >$ that's being fed into our proposed machine, i.e., the $< \mathcal{P}_{<\mathcal{B}>} >$ part. The first call on printf function performs the work of $< \mathcal{P}_{<\mathcal{T}>} >$ part while the second call and the loop performs the work of the $< \mathcal{T} >$ part. In the java version, the idea is exactly the same, perform the work of $< \mathcal{P}_{<\mathcal{T}>} >$ part in the execution of the first for loop, then perform the work of $< \mathcal{T} >$ part in the execution of the third for loop.

The Encoding Scheme

The encoding of this problem is extremely important. The encoding determines the number of steps that the Turing Machine execution is going to take. To make the simulation practical due to the time constraint, we need to carefully design it.

Version 1:

I started this project with unary encoding as below:

TapeSymbol	Encoding
separator	0
1	111111
B	1111
0	11111
-3	1
R	11
L	111
States	**Encoding**
0	1111111
1	11111111
2	111111111
⋮	⋮

This encoding makes an important subroutine: addition, trivial; basically, we only need to append 1's in operand one to the end of 1's of operand two. However, in terms of states, and the length of the encoding of states; if we have $1,000$ states, the length of one line of specification is going to be at least $1,000 \times 2$; and since there are at least $1,000$ such lines, going from the left end to the right end of the tape one time will take more than $1,000,000$ steps. (Here we took the average of the longest specification line and the shortest line as the estimate of an overall line size. $(2000 + 0)/2 = 1,000$)

However, as we will see later the machine entails a lot of copying from the left end of the tape to the right end of the tape; besides, $1,000$ is slightly less than the number of states we will be using, this encoding will be practically impossible to use.

Version 2:

I end up using this second version of binary encoding, as below:

TapeSymbol	Encoding
separator	B
1	111
B	10
0	110
-3	11
R	100
L	101
States	Encoding
0	1000
1	1001
2	1010
⋮	⋮

For an integer i, let $|i|$ denote the number of digits i have in decimal form. Binary encoding for i takes an average length of $\log_2(10)|i| \approx 3.3|i|$. The encodings under this encoding scheme will effectively reduce the length of the encoding of the entire machine and thus, number of steps that takes for the machine to copy something from the left end to the right end. It turns out that copying is a key feature in $\mathcal{SELF}$.

On the other hand, this encoding scheme does not contain any encoding starting with 0 or of length 1. Both features greatly help the simplicity of the TM, as we will see later.

The Construction Flow

We need to construct the following phases of the TM, $\mathcal{SELF}$.

(1) phase 0: copy input string: The reason why this phase is necessary because TMs cannot possess information of the tape position, so since we have 2 parts to construct $\mathcal{P}_{<\mathcal{B}>}$ and $<\mathcal{B}>$, we need 2 sets of input strings.

1 So for an input string 10010B1010, we start from the left most symbol.

2 In this step, we copy 1 or 0 to the left end and right end of our string. Both ends are found by three consecutive blank symbols, BBB.

10010B1010 $\rightarrow$ 1BB10010B1010BB1 $\rightarrow$ 1BBB0010B1010BB1.

3 keep repeating step 2 until we copied 2 blank symbols, BB. So on this input string, after this phase we are left with

0101B01001BBBBBBBBBBBBBB10010B1010.

Note that the one of these string are from right to left, so we need to be careful in the later phase. We name the left part of the copied string (0101B01001 in this case) as $copied_{left}$, and the right part (10010B1010 in this case) as $copied_{right}$.

This part is provided by lines below "## phase 0" and above "## phase 1" in the attached file $v6.phase012$.

(2) phase 1: Specification about this phase is also contained in file $v6.phase012$; we name this phase as a TM, $\mathcal{Q}$, which computes function q on an input string.

$<\mathcal{Q}>$ should contain the first few lines of specification deleting any input string. (First prints out these lines) We call this part of the encoding of our final machine, the Deleting Encodings. The Deleting Encodings should be separated from the right part of the copied string by 2 blank symbols, BB.

(3) phase 2: next, $< \mathcal{Q} >$ should read the right part of the copied string. For every tape symbol starting from the left of this right part, copy the new state from the previous line of configuration, write B10B for encoded tape symbol B, then copy the state of this line as the new state then increment the state of this line by 1. Then write the tape symbol (1 or 0) then append $B100B$ indicating encoded R.

Note that I made a mistake in this part: I took the encoded string, say 1010 for state 2 as one tape symbol for $\mathcal{Q}$. This should not happen, as each of the 1s and 0s should be treated as a tape symbol. And I believe this is the only reason that $v6.tm$ is not doing what it is supposed to do.

(4) phase 3: This part entails copying the state with the current highest number value of the encoding to the left of $< \mathcal{Q} >$, separated by BB. Finding this new state is tricky, but by the nature of phase 2, the state with the highest value is automatically the last new state. Name this new state q_{concat}.

If our encoding is as below, where $< \ldots >$ denote all undisplayed encodings of $\mathcal{Q}$.

$$copied_{left}BBB \ldots BBBBBBBBBBBBB < \ldots > B1001010B100B1001011B10B100$$

Then after this phase we should get

$$copied_{left}BBB \ldots BBBB1001011BB < \ldots > B1001010B100B1001011B10B100$$

(5) phase 4: We copy q_{concat} to the right end of $< \mathcal{Q} >$ we have for now, getting

$$copied_{left}BBB \ldots BBBB1001011BB < \ldots > B1001010B100B1001011B10B100B1001011$$

then go to $copied_{left}$, delete the first state. And take copy the first symbol over to the right end of our machine encoding.

Then we copy q_{concat} to the end again, now we go to the left part of the string copied in phase 0, ADD the new state to q_{concat} we just copied. This addition is extremely complicated, but it is specified inside of $add.tm$.

Lastly, we copy the new symbol and direction over from copied string on the left.

Note that in phase 4 and phase 5 when we are copying from $copied_{left}$, we delete the copied part at the same time so we can progress to the next line of the input encoding. This is not the case when we are copying q_{concat}.

Also Note that phase 4 only deals with 1 line of the input encoding, this phase aims to connect the states of $< \mathcal{T} >$ to the end of the calculated $< \mathcal{P}_{<\mathcal{T}>} >$. For connecting, since there could be multiple usage of state 0, multiple usage of state 0 makes the state connection hard; however, we could specially design the input encoding to have state 0 only on the first line so concatenation is simple.

This is also due to the benefit of our encoding scheme with encodings starting with only symbol 1; this is saying that in state 0 we are going to only read 1, then we can progress to another state. We do not have to deal with the situation of in state 0 reading tape symbol 0.

(6) phase 5: for all lines of encoding string in $copied_{left}$, we add the state to q_{concat} as the state for our output encoding. Then copy the symbol over to the right end; next, add the new state of the input encoding to q_{concat} as the new state for our output encoding; finally copy the new symbol and direction over.

(7) We halt after phase 5.

Appendix

The following C program is Cited from Ken Thompson's famous Article "Reflections on Trusting Trust". I made some minor changes to make it able to actually compile with GCC compiler and run.

```
< − − − − −C program− − − − − >
#include<stdio.h> #include<string.h>
char s[ ]={
        '\t',
        '0',
        '\n',
        '}',
        ';',
        ⋮
        0
};
main(){
        int i;
        printf("#include<stdio.h>\n#include<string.h>\nchar \ts[]={\n");
        for(i=0;s[ i ];i++)
                printf("\t%d,\n",s[ i ]);
        printf("%s",s);
}
```

This Java program is inspired by Gary Thompson's webpage "The Quine Page".

< − − − − −Java program− − − − − >

```
1 public class Self
2   public static void main(String[] args)
3    char q = 34;
4    String[] l = {
5     "public class Self{",
6     "   public static void main(String[] args){",
7     "    char q = 34;",
8     "    String[ ] l = {",
9     "    ",
10    "    };",
11    "    for(int i=0; i<4; i++)",
12    "     System.out.println(l[ i ]);",
13    "    for(int i=0; i<l.length; i++)",
14    "     System.out.println(l[ 4 ] + q + l[ i ] + q + ',');",
15    "    for(int i=5; i<l.length; i++)",
16    "     System.out.println(l[ i ]);",
17    "    }",
18   "  }"
19  };
20  for(int i=0; i<4; i++)
21   System.out.println(l[ i ]);
22  for(int i=0; i<l.length; i++)
23   System.out.println(l[ 4 ] + q + l[ i ] + q + ',');
24  for(int i=5; i<l.length; i++)
25   System.out.println(l[ i ]);
26  }
27 }
```

The Scripts

I used some scripts to generate me the desired state numbers for a Turing Machine. Here is a list

(1) binary_to, binary_from scripts: source code: binary_to.cpp, binary_from.cpp;

Both of these source codes contains a DEBUG macro that allows us to see which of the states in decimal match to which of the states in binary.

To run binary_to binary tm output file, we need to create a file called *output.scheme* (there is a sample .scheme file attached) and rename the binary tm output file as *output.enc*. The decoded file will be called *output.dec*(Only *output.dec* will be changed):

./binary_from output

We run the binary_from from on *output.dec* with *output.scheme*, we will get back *output.enc* file. (Only *output.enc* will be changed.):

./binary_to output mult

Note that if we specify the third argument as "mult" the *output.enc* will be in multiple line. Also note that we shall not use \t in any of the files, since that symbol is not considered as a separator, while in the sample *output.scheme*, the separator symbol is a space.

(2) Also, since we are building the tm by parts we need to be able to connect the state: source code connect_states.cpp:

./connect output.dec 138

Upon executing this command, a new file called output.connected138 will be created and every state number in output.connected138 is 138 more than its original state number in output.dec

(3) Finally, the TM simulators in C. And a small program that computes the function q for us in C. Note that the input file $v6.tm$ the first line contains the tape symbols, the second line is a single for the tm simulator to notice; the third line is the start state, the fourth line is another . Then the actual specification starts.

tm is compiled from tm_slow_powerful_v2.cpp, while tms is compiled from tm_sim.cpp which is much faster:

./tms v6

Work Cited

Sipser, Michael. "6.1 Self-reference." Introduction to the Theory of Computation. Australia: Course Technology Cengage Learning, 2013. 246+. Print.

Thompson, Ken. "Reflections on Trusting Trust." (1984): Rpt. in Communications of the ACM. Vol. 27. 762. Print.

Thompson, Gary P., II. "The Quine Page." http://www.nyx.net/ gthompso/quine.htm, Web. 09 May 2016.