The Recursion Theorem

Sipser – pages 217- 224

Self replication

- Living things are machines
- Living things can self-reproduce
- Machines cannot self reproduce

- Are these things all true?

The robot factory

- Suppose there exits a robot factory that can make things
- For example we program the factory to make cars.
- But cars are simpler than robot factories, so this seems reasonable.
- Can we program the robot factory to make robot factories?

Can we write a program?

- self x = …

- Such that when we run self on any input we get the description of self as output?

- This problem is related to the robot factory dilema – Can machines (programs, Turing Machines) encode enough information to reproduce their own descriptions.
- The answer is yes!

Consider the Haskell Program

```
copy s = putStr s >> print s
self2 x = copy "self2 x = copy "
```

How do `putStr` and `print` differ

```
*Main> self2 5
self2 x = copy "self2 x = copy "
```

A program like this is called a Quine, this is one of the smallest Quines I know of, but it can be done in any language.

Copy

- `copy s = putStr s >> print s`

- Copy just makes two copies of its input.
- One where the string is not quoted, and the other where the string is quoted.

self2

- `self2 x = copy "self2 x = copy "`

- Self2 just applies “copy” to a string that forms the body of everything upto the string.

Technical fix

```
copy s = putStr s >> print s
self2 x = copy "self2 x = copy "
```

- Some will claim that self2 isn't really self reproducing because it does not reproduce the code for copy. For such skeptics

```
self x = (\s -> putStr s >> print s) "self x = (\\s -> putStr s >> print s) "
```

The recursion theorem

- The recursion theorem states that some Turing Machines can reproduce their own descriptions
- It is implied that we can turn any TM into an equivalent one that has this property.

From Haskell to TMs

- Our Haskell program had two components

```
self x = (\s -> putStr s >> print s) "self x = (\\s -> putStr s >> print s) "
```

A. A program that returned that combined the string it was given. Once quoted, once not

```
(\s -> putStr s >> print s)
```

B. A program that always returned a constant string .

```
"self x = (\\s -> putStr s >> print s) “
```

- To translate our results to TM's we will need similar components

The copy component

- Q (w)
  1. Construct the following Turing Machine P_w
     1. $P_w(x)$ = on any input x
        1. Erase the input
        2. Write w on the tape
        3. Halt
     2. Output <P_w>

The self reproducing TM

- The TM SELF comes in 2 parts A and B
  - We want SELF to print out <SELF> =<AB>
  - By the way TMs run we want A to run first, the to pass control to B. The output of A is on the tape when B starts
  - When A runs, it leaves a description of B on the tape. This is easy as we can use P_B by essentially defining A to be Q(B)

What does B look like?

- B (<M>)
  - Compute q(<M>)
  - Combine result with <M> (already on tape) to make a complete TM
  - Leave this complete TM on tape and Halt.

  - Compare with

```
self x = (\s -> putStr s >> print s) "self x = (\\s -> putStr s >> print s) "
```

Why?

- The recursion theorem says we can implement self referential programs in any sufficiently strong programming languages.
- Any program can refer to its own description!
- Not only can it obtain its own description, it can use this description to compute with!

Recursion Theorem

- Let T be a TM that computes a function from $\Sigma^* \times \Sigma^* \rightarrow \Sigma^*$.
- There exists another TM, R, where for every w
  - R(w) = T(<R>,w)
  - There is an equivalent Turing Machine that computes the same result given just w, but we can program T as if it had access to R's description!

Terminology

- When describing a TM, M, one may include the words “obtain own description <M>” in the informal description of how M operates.

- The machine might do things like
  - Print out M
  - Count the number of states in M
  - Simulate M

A very simple example

- SELF(x) =
  - Obtain via recursion theorem, its own description <SELF>
  - Print <SELF>

- The recursion theorem says given the TM T
  - T(<M>,w) = Print <W> and Halt
- We can obtain SELF above for free
- SELF(w) = T(<SELF>,w)

A_{TM} is undecidable

- Proof by contradiction
- Assume H decides A_{TM}
- We construct B as follows
  - B(w) =
    - Obtain own description <B>
    - Run H(<B>,w)
    - Do the opposite of what H says
      - That is B rejects if H(<B>,w) accepts
      - B accepts if H(<B>,w) rejects
- But B does the opposite of B it H is to be believed, so it must be the case that H cannot be deciding A_{TM}

Minimal Machines

- If M is TM, then we say the length of the description <M> is the number of (tape) symbols in the string describing M.

- We say that M is minimal, if there is no equivalent TM to M that has a shorter description
- Define the language
  - $MIN_{TM} = \{ \langle M \rangle \mid M \text{ is a minimal TM}\}$

MIN_{TM} is not Turing recognizable

- Assume some TM, E, enumerates the MIN_{TM}, and obtain a contradiction.
- Let C(w) =
  - Obtain (via recursion theorem) own description <C>
  - Run E until a machine D appears with a longer description than that of C
  - Simulate D on input w
- There must be a D (why?) D has length greater than C, but C behaves just like D. So D cannot be minimal, because C has smaller length.
- So E can't enumerate MIN_{TM} so our assumption must be false.

Fixed Points

- Let T: $\Sigma^* \rightarrow \Sigma^*$ be a computable function.
- Then there exists a TM F for which T(<F>) describes a TM equivalent to F
  - If the input to T isn't a proper TM encoding then T should return a TM that immediately rejects all strings

Proof

- Let F be the following TM
- F(W) =
  - Obtain (via recursion theorem) own description <F>
  - Compute T(<F>) to obtain description <G>
  - Simulate G on w

- Since F simulates G, they are clearly the same
- But F = T(<F>) = G