All Turing-Computable* Functions are Recursive

Computability and Logic

Basic Idea

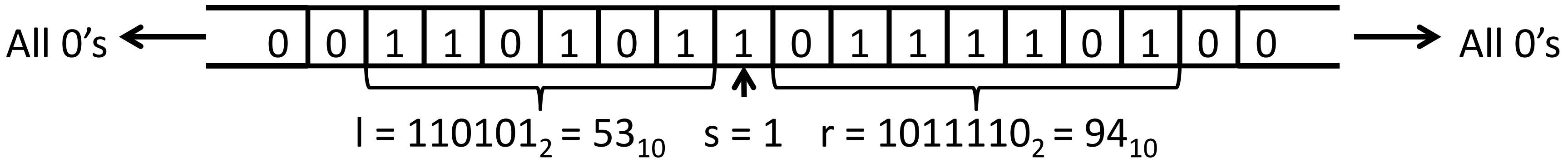
- We're going to code (i.e. assign natural numbers to) the machine and machine-configuration (tape contents, head location, internal state) at every step.
- We'll show how the configuration at each step is a recursive function of the configuration of the previous step and the nature of the machine
- Thus, the halting configuration is a recursive function of the starting configuration and machine.
- Add to this:
  - The starting configuration is a recursive function of the function input value.
  - The function output value is a recursive function of the halting configuration.
- And you get: the function output value is a recursive function of the function input value … if the function is Turing-computable.

Set-Up

- Consider a Turing-machine M that computes* function f(x):
  - M starts with [x] (x+1 consecutive 1's on otherwise blank (all 0) tape with head at leftmost 1)
  - M halts with [f(x)] if f(x) is defined
  - M does not halt, or halts in non-standard output configuration, if f(x) is not defined
  - M has states q_0, … q_k
  - M only uses 1's and 0's
  - M has a 0-transition and a 1-transition defined for every state q_1, … q_k
  - No 0-transition or 1-transition is defined for state q_0
  - M starts in state q_1
  - M halts in state q_0
- The proof can easily be modified to deal with functions with more than 1 argument, machines that use non-binary alphabet, and machines that don't follow the above convention regarding its internal states

Coding the Tape Configuration (Tape Contents and Head Location)

- Since we're only using 0's and 1's as our symbols, we can treat the part of the tape to the left and right of the head location as binary numbers (why?).
- Thus the tape configuration can be coded using 3 numbers (see diagram below):
  - l: the left number
  - r: the right number (note the flip!)
  - s: the symbol/number (0 or 1) that the head is looking at
- If using j different symbols (which can be coded as 0, …, j-1), treat the symbol string as the j-base representation of a number.



Coding Machine Configuration

- The machine configuration at each step of the computation is the tape configuration plus internal state q_i machine is in.
- So this can be coded by 4 numbers: l,r,s,i
- Code this by a single configuration number c that is a recursive function of l,r,s,i:
  - $c = conf(l,r,s,i) = 2^l*3^r*5^s*7^i$
- We can recover l,r,s,I from c using recursive functions:
  - l = left(c) = lo(c,2)
  - r = right(c) = lo(c,3)
  - s = sym(c) = lo(c,5)
  - i = state(c) = lo(c,7)

Coding the Machine

- Machine M will be coded by a sequence of numbers a_{10}, q_{10}, a_{11}, q_{11}, a_{20}, q_{20}, a_{21}, q_{21}, …, a_{k0}, q_{k0}, a_{k1}, q_{k1}
- Where:
  – a_{is} = action to take when in state i and looking at symbol s:
    - a = 0 for write 0
    - a = 1 for write 1
    - a = 2 for move left
    - a = 3 for move right

  – q_{is} = new state to go to when in state i and looking at symbol s

Getting Instructions

- Let m be the single code number for the sequence coding machine M: $m = 2^{a10} * 3^{q10} * \ldots$
- Then, the (codes of the) action to take and the new state to go to for machine M when in state q_i and looking at symbol s are recursive functions (the '–' function is the modified difference function) of m, i, and s:
  - action(m,i,s) = ent(m,4*(i-1) + 2*s) (remember: the exponent of 2 is considered the '0th' entry (since it usually encodes the length of a sequence, rather than the first entry of the sequence))
  - action(m,0,s) = 0 (whatever, will never be used)
  - newstate(m,i,s) = ent(m,4*(i-1) + 2*s + 1) if i > 0
  - newstate(m,0,s) = 0 (whatever, will never be used)

Performing Actions

- We can define the new left number l as a (recursive) function (through a definition by cases) of the old tape configuration (as indicated by l,r, and s), and the action a to take:
  - newleft(l,r,s,a) = l if a = 0 (write 0)
  - newleft(l,r,s,a) = l if a = 1 (write 1)
  - newleft(l,r,s,a) = quo(l,2) if a = 2 (move left)
  - newleft(l,r,s,a) = 2*l + s if a = 3 (move right)
- Likewise for newright (and now you understand why we flipped the right binary string!)
- And for the new symbol:
  - newsym(l,r,s,a) = 0 if a = 0 (=1 for a = 1)
  - newsym(l,r,s,a) = rem(l,2) if a = 2 (=rem(r,2) if a = 3)

Define new Configuration

- Now we want to show that the (code of the) new configuration is a recursive function of the (code of the) current configuration and (code of the) machine.
  - action(m,c) = action(m,state(c),sym(c))
  - newstate(m,c) = newstate(m,state(c),sym(c))
  - newconf(m,c) = conf(
    ```
    newleft(left(c),right(c),sym(c),action(m,c)),
    newright(left(c),right(c),sym(c),action(m,c)),
    newsym(left(c),right(c),sym(c),action(m,c)),
    newstate(m,c)
    )  if state(c) > 0
    ```
  - newconf(m,c) = c   if state(c) = 0

Configuration at Any Time

- Now we can show that the configuration at any step during the computation is a recursive function of the machine and its starting configuration. Using Recursion:
- conf(m,x,t) = the (code of the) configuration that machine M (with code m) is in after t steps when started on input tape [x]:
  - conf(m,x,0) = conf(0,2^x-1,1,1) (verify this!)
  - conf(m,x,t+1) = newconf(m,conf(m,x,t))

Has M Halted in Standard Configuration?

- Define relation Done(m,x,t) iff machine M (coded by m) with input tape [x] has halted in some standard output configuration after t steps (or before that).
  - Done(m,x,t) iff

    $\mathrm{left}(\mathrm{conf}(m,x,t)) = 0 \wedge$

    $\mathrm{right}(\mathrm{conf}(m,x,t)) = 2^{\mathrm{lg}(\mathrm{right}(\mathrm{conf}(m,x,t)),2)+1} - 1 \wedge$

    $\mathrm{sym}(\mathrm{conf}(m,x,t)) = 1 \wedge$

    $\mathrm{state}(\mathrm{conf}(m,x,t)) = 0$

When is M done, if ever?

- Let's figure out when M is done (as defined on previous slide), if ever (i.e. after how many steps does M, when started on [x], halt in standard output configuration, if it ever does?)
  - halttime(m,x) = $Mn[c_{\neg Done}](m,x)$
  - Explanation:
    - Mn[f](x) returns smallest y for which f(x,y) = 0 if such a y exists, otherwise it is undefined.
    - $c_{\neg Done}(m,x,t) = 0$ iff M when started on [t] has halted in standard output configuration
    - So, $Mn[c_{\neg Done}](m,x)$ returns smallest t for which M has halted in standard output configuration when started on [x], if such a t exists, otherwise it is undefined.

Getting function value from Output Tape

- Finally, let’s get the output value f(x) of the function computed by M when given input [x]:
  - f(m,x) = lg(right(conf(m,x,halttime(m,x))),2)+1
- Note that if M does not halt in standard output configuration for some input [x], then halttime(m,x) is undefined, meaning that f(m,x) will be undefined as well … which is what it should be.

Summing Up

- Suppose f is Turing-computable*.
- Then there is some Turing-machine that computes* f, i.e. for any x:
  - M transforms [x] into [y] iff f(x) = y
  - M, when started in [x] does not halt in standard output configuration iff f(x) is undefined
- But we just showed:
  - M transforms [x] into [y] iff f(m,x) = y
  - M, when started in [x] does not halt in standard output configuration iff f(m,x) is undefined
- So, for any x: f(x) = f(m,x)
- Since f(m,x) is a recursive function, f(x) is recursive as well.
- So, all Turing-computable* functions are recursive.

So What?

- We have shown that:
  - All Abacus-computable* functions are Turing-computable*
  - All recursive functions are Abacus-computable*
  - All Turing-computable* functions are recursive
- Hence, these 3 sets of functions are exactly the same set of functions!
- Seeing that alternative proposed definitions of computability (or calculability) turn out to be equivalent to each other provides further evidence for the Church-Turing Thesis that every computable function is Turing-computable*

Moreover

- The proof that all Turing-computable* functions are recursive can easily be modified to include machines that use an alphabet of any (finite) size (how?).
- In the proof that all Abacus-computable* functions are Turing-computable* we showed that any Abacus machine can be simulated using a Turing-machine using a binary alphabet only.
- Thus, having more than 2 symbols does not increase the power of Turing-machines: anything that a Turing-machine can compute using any number of symbols can be computed using a Turing machine using only 2 symbols.

Even More

- In the proof that all Abacus-computable* functions are Turing-computable* we showed that any Abacus machine can be simulated using a Turing-machine that never goes more than 2 squares to the left of its starting position.
- Hence, anything that is Turing-computable* can be computed using a Turing-machine that has a one-sided infinite tape only: you just need to either represent the input on the tape preceded by two 0's, or start any computation by shifting the input two places to the right while never moving to the left of the starting point (HW question: design such a Turing-machine)

Oh, and one more Little Thing

- Since the function f(m,x) from our proof is recursive, it is Turing-computable.
- So, there is some Turing-machine that is able to take in (the codes of) any Turing-machine M and input x and output what that machine M outputs when provided with input x.
- This is of course a Universal Turing-machine!
- So, we have established the existence of a Universal Machine.