Lecture 7: Primitive Recursion is Turing Computable

Michael Beeson

Closure under composition

Let f and g be Turing computable. Let $h(x) = f(g(x))$. Then h is Turing computable. Similarly if $h(\mathbf{x}) = f(g_1(\mathbf{x}), \ldots, g_n(\mathbf{x}))$.

Proof. In Exercise 7.2, you will show that it suffices to show that h is computable by a 2-tape Turing machine; we call these tapes the main tape and the auxiliary tape. Our new machine M has instructions that do on the auxiliary tape what the machine for g does, while not changing anything on the main tape. Its states all have numbers increased from those in the machine for g by some large number $2N$, more than the number N of states in the machine for f. M also will have states that do what the machine for f does, but on the main tape, ignoring the auxiliary tape. These states all have their numbers increased by N; so they still do not overlap the states we already put into M.

Closure under composition, continued

Now we add some new states, which cause the following behavior: when M starts, it copies $\mathbf{x}$ from the main tape to the auxiliary tape, then transfers control to the start state of g. When (and if) the machine for g terminates, instead of stopping, M will copy the contents of the auxiliary tape (up to the first blank) to the main tape, erasing the auxiliary tape as it goes, and then go back to the left end of the main tape and transfer control to the machine for f. When (and if) the machine for f terminates, M will halt. That completes the proof for simple composition. Generalized composition is treated the same way, but using a machine with n auxiliary tapes, on which we store the values of $g_1(\mathbf{x}), \ldots, g_n(\mathbf{x})$.

Closure under primitive recursion

To be proved: Let g and h be Turing computable, and suppose f is defined by

$$f(\mathbf{x}, 0) = g(\mathbf{x})$$

$$f(\mathbf{x}, n+1) = h(\mathbf{x}, f(\mathbf{x}, n))$$

Then f is Turing computable (in unary notation or in binary notation).

- We suppose we have Turing machines for g and h.
- We rename the states so there are no state names in common between these machines.
- We will construct a machine for f.

Constructing a machine M for f

- M should compute $f(\mathbf{x}, n)$ when it is started at any square on the tape, where $\mathbf{x}$ followed by n is written (using blank to separate the arguments), without ever altering anything to the left of the initial square.
- Thus parts of another computation that have been left there will not be disturbed.
- We will need to assume that there is a symbol, say $\#$, in the alphabet that is not used in the machines for h and g.
- In Exercise 7.1, you will show how to reduce the size of the alphabet if required.
- We are using the tape to store the "call stack" for recursion, with $\#$ to delimit the entries.

An example

Suppose $g(x) = 4$ and $h(x,4) = 3$ and $h(x,3) = 5$. Then

$$\begin{aligned} f(x,2) &= h(x, f(x,1)) \\ &= h(x, h(x, f(x,0))) \\ &= h(x, h(x, g(x))) \\ &= h(x, h(x,4)) \\ &= h(x,3) \\ &= 5 \end{aligned}$$

Example continued

We want to see the following tapes occur in the computation by M. ˆ indicates head position. x stands for specific input.

```
x 11              state 0
^

x#x 1          state 0 again
  ^

x#x#x       enter start state of machine for g
    ^

x#x#1111    after computing g(x) = 4
    ^

x#x 1111    enter start state of machine for h
  ^         erase # to pop stack
x#111          after computing h(x,4) = 3
  ^

x 111          enter start state of machine for h
^              stack popped again
11111             after computing h(x,3) = 5
^
```

The plan for M

The machine M works as follows:

- it moves right across the inputs $\mathbf{x} = x_1, \ldots x_m$.
- After M crosses the m-th blank it leaves a marker $\#$ in the position of that m-th blank. ("Push")
- Then it copies the entire string $\mathbf{x}$ to the right of $\#$, in the process moving the digits of n over to the right to make room (so in essence inserting $\mathbf{x}$ between $\#$ and n).
- Then it subtracts 1 from n (which is very easy in unary notation; computation in binary representation is discussed below) and moves back to the $\#$ marker, and then one square to the right.

Now we've reduced n by one, and next we'll either recurse or compute the base case.

After reducing n by one

Now machine M (reading the square one to the right of $\#$) tests whether it sees a blank or not. If it does not see a blank, then n has not been reduced to zero, so it enters its own start state (to recurse).

If it does see a blank, that means that the subtraction has reduced n to zero (the empty string). Then M does the following:

- enters a state designed to remember that it saw a blank
- moves left across the input $\mathbf{x}$ (passing $m-1$ blanks) and continuing left until it encounters a blank or a $\#$
- then moves right one square and enters the start state of the machine for g.
- If the machine for g terminates, then M moves left and erases the $\#$ (popping the stack).
- Then if what is to the left is not blank, it means that the computation of g finally completed a recursive call to f. So to the left we have $\mathbf{x}$ of the calling environment.

After completing a recursive call

- Then M moves left, passing $m-1$ blanks, until it encounters either another blank (the m-th one) or another $\#$.
- It moves one square to the right.
- Then it has $\mathbf{x}$ followed by the already-computed value of $f(\mathbf{x}, n)$ to the right.
- It enters the start state of h.
- When the machine for h terminates, M does the same as when the machine for g terminates.

Recognizing we're done

- After g or h terminates, if there is a blank to the left, that was the toplevel call.
- So we're done.
- That completes the description of M.

Correctness proof for M

Now we prove by induction on n that if M is started with a tape containing $\mathbf{x}$ followed by n, with blanks separating the arguments and a blank to the left of $\mathbf{x}$, with the scanned square the leftmost square of $\mathbf{x}$ (the "initial square") then M will terminate with $f(x, n)$ on the tape starting with the scanned square, and without having altered any square to the left of the initial square.

- The proof is straightforward, following the construction of M.
- That completes the proof, for unary representation.

Computing in binary

In order to compute f in binary representation, there is only one change required: that is in the part where we need to "decrement n", where n at that point lies to the right of $\#$ and nothing is to the right of n. So it will suffice to invoke at that point a Turing machine that computes the predecessor of n (in binary representation) without altering the tape to the left of the initial scanned square. For the existence of such a machine, we cannot appeal to the fact that predecessor is primitive recursive, since we need that machine in the proof of this theorem. Instead, it must be directly coded. You have been asked to do that in one of the exercises.

A programming project

Someone could write some nice computer programs as follows:

- Input: a set of primitive recursive function indices. These wouldn't be integers but strings.
- Output: TM code for the functions indexed by the inputs, that could be run in a simulator.
- Second program: to compute TM indices from human-friendly definition syntax.
- These are really not that difficult. I'm surprised they don't seem to exist.

Theorem: Every μ-recursive function is Turing computable

- We already proved closure under primitive recursion and composition.
- It only remains to prove that the Turing computable functions are closed under the μ-operator.

Suppose

$$f(\mathbf{x}) = \mu k P(\mathbf{x}, k)$$

and we have a Turing machine M that computes the representing function of $P(\mathbf{x}, k)$, without altering anything to the left of the original input. We show that f is computable (in unary or binary) by the following two-tape machine, which has an auxiliary tape and a main tape. We pad the instructions of M (which is a one-tape machine) to do nothing on the auxiliary tape, and we rename the start state of M to avoid confusion with the start state of our new machine; these instructions are part of our new machine.

Progamming a Turing machine to search

- Starting in state `loop` with $\mathbf{x}$ on the input tape, where $\mathbf{x} = x_1, \ldots, x_n$, and k on the auxiliary tape, we move into the start state of M.
- Then we allow machine M to run, to try to compute $P(\mathbf{x}, k)$.
- If M terminates, having computed that $P(\mathbf{x}, k)$ is true, we are almost finished: we just copy k to the main tape, starting at the original scanned square, and end with an extra blank (or if you like, erase everything on the main tape). Then go to the terminating state.
- If M terminates, having computed that $P(\mathbf{x}, k)$ is false, we increment k on the auxiliary tape, erase the main tape and go to state `loop`. (We have thus completed one loop in the search, and are ready to begin the loop again with the next value of k.)
- The incrementing of k on the auxiliary tape can be done either in unary or binary (or decimal for that matter) and our machine will compute f in that same representation.

Summary

- The starting state of our new machine is `loop`.
- If the machine is started with blank main tape and blank auxiliary tape, then it clearly computes $f(\mathbf{x})$ in the sense that the output of $f(\mathbf{x})$ appears on the main tape at termination.
- You may use the technique of Exercise 7.2 to create a one-tape machine equivalent to this 2-tape machine.
- That completes the proof.
- This theorem shows at one blow that LOTS of functions are Turing computable.

Questions

- Is Ackermann's function Turing computable?
- Is Ackermann's function μ-recursive?