Lecture 8: Computation by Turing Machines

Michael Beeson

Coding sequences

By definition, Turing machines are finite sequences of instructions, and instructions are also certain finite sequences. If we want to code these things as integers, ultimately sequences of integers have to be coded as integers. Traditionally powers of primes were used. In Lecture 4, another method, based on ascii codes, was given. What we need is just this:

There are primitive recursive functions $length(x)$ and $x[i]$ (primitive recursive in both variables x and i) such that for every sequence $a_0, a_1, \ldots, a_{n-1}$, there exists an x such that

- $length(x) = n$ and
- for each j with $0 \leq j < n$ we have $x[j] = a_j$.

We will not refer in today's lecture to any specific implementation of sequence coding.

But instead of thinking of these methods as "coding" sequences, we can think of them as the **definition** of sequences. Thus when we write (a, b, c) or $\langle a, b, c \rangle$, we don't mean a function or a set, but an integer.

Turing machines are integers

- States are officially integers (perhaps representing the ascii of string state names)
- Symbols are 8-bit integers
- R, L, and C are shorthand for 0,1,2 in that order. So they are integers too.
- So, instructions are also integers.
- Turing machines are sequences of instructions.
- So, Turing machines are integers.
- Not just **coded by** integers. They ARE OFFICIALLY integers.
- Of course, we still work with string representations, e.g. to use the simulator.

Instantaneous tape descriptions

- A **tape square** is a 16-bit integers; the second 8 bits specify a symbol and the first 8 bits specify a **head-present flag** (which tells whether the machine's head is on that square at that moment). (Yes, 7 bits are wasted.)
- You can think of the head-present flag as the color "red."
- An ITD (**instantaneous tape description**) is a finite sequence consisting of one state, followed by a sequence of tape squares. In practice, exactly one of those tape squares has a nonzero head-present flag, but that isn't part of the definition.
- An ITD can be *padded* to any desired length by appending more tape squares containing zeroes (blank symbols and zero head-present flag).
- Two ITDs are *equivalent* if they both can be obtained by padding a common initial subsequence.

Comparison to Kleene's textbook

- An ITD is similar to a "machine-tape state" in Kleene's textbook.
- Kleene doesn't have the head-present flag.
- The head-present flag permits us to keep track of the head's location while keeping the tape descriptions lined up with the previous and next step of the computation.
- Think of the computation as arranged in a rectangle, with one step per row, given by its ITD.
- Kleene inserts the current state left of the scanned square, throwing everything off by one, so the rows of the computation don't always line up. The head-present flag fixes that problem.
- Kleene uses powers of primes to code the three items: tape left of the head, current state, tape right of the head. Of course this coding works too.

Computations, intuitive version

- A **computation** is a sequence of ITDs corresponding to successive steps by a given Turing machine M.
- We write them in rectangular form, one row for each step of computation.
- A column at the left keeps track of the current state.
- We still need to make precise the relation that must hold between two rows of a computation for it to correspond to the instruction set of that Turing machine.
- The first column has to be wide enough to hold a state of M. That doesn't matter, officially, since we use sequences of integers, not just sequences of bytes; it only matters for your mental picture of a computation.

Notation

- If x is an ITD for machine M, then x is (thought of as) a sequence of integers.
- Officially $x[0]$ is the current state. For readability we write $x[0]$ as $x.state$.
- Then $x[1], \ldots$ describes the tape. That sequence is $x.tape$.
- Each member $x.tape[i]$ is a 16-bit integer. The first 8 bits are the head-present flag and the second 8 bits are the symbol. For readability we write these as $x.tape[i].flag$ and $x.tape[i].symbol$.
- Alternately we could have made $x.tape[i]$ a sequence of length 2, but we did not choose to do that.

An ITD is an integer

- An ITD is officially a sequence of integers.
- But a sequence of integers is, by definition, an integer.
- Thus an ITD is an integer.

The **Next** relation

This is the relation between two adjacent rows of a computation.

- The relation in question does not depend on the whole row, but only on the squares near the scanned square.
- It is enough to consider a small rectangle containing six squares, the scanned square, the ones to its left and right, and the three in the next row below those three.
- The predicate $Next$ takes a Turing machine e, two states $s[0]$ and $s[1]$, and six tape squares, which we label $t[i, j]$ for $i = 0, 1, 2$ and $j = 0, 1$.
- It will be defined so that it is true if these six tape squares could describe a portion of a computation by machine e.

Definition of **Next**

Arguments of **Next**:

- Turing machine e
- states $s[0]$ and $s[1]$
- six tape squares $t[i, j]$ for $i = 0, 1, 2$ and $j = 0, 1$.

$Next$ checks that the second triple of tape squares can be obtained from the first triple using one of the instructions in machine e. Specifically, there is a first instruction in e that specifies, "when reading $t[0][1].symbol$ in state $s[0]$, write $t[1][1].symbol$ and move R, L, or C", $t[i, j].flag$ is set (nonzero) in exactly two of the six squares, namely $t[0, 1]$ and one of $t[1, j]$, where $j = 0$ if the instruction said move left, $j = 1$ if it said not to move, and $j = 2$ if it said move right.

Partial computations, precisely defined

A **partial computation** by machine e is a sequence (of length m, say), of ITDs $x[i]$ all of the same length N (with zero-based indexing, so the first one is $x[0]$ and the last one is $x[N-1]$), such that

- $x[0].state = 0$ (the start state)
- $x[0].tape[j].flag = 1$ for exactly one $j < N$. (The head starts on square j.)
- for each row $i < m-1$, and $0 < j < N$, if $x[i].tape[j].flag = 1$ then $j < N-1$ and $Next$ is true when evaluated at e, $x[i].state$, $x[i+1].state$, and the six tape squares $x[i].tape[j-1]$, $x[i.tape[j]$, $x[i].tape[j+1]$, $x[i+1].tape[j-1]$, $x[i+1].tape[j]$, and $x[i+1].tape[j+1]$ (except if $j = 0$ see next slide).
- and $x[i+1].state$ is the new state dictated by e and $x[i].tape[j].symbol$ and $x[i].state$.
- and if $x[i].tape[j].flag = 0$ then $x[i].tape[j].symbol = x[i+1].tape[j].symbol$.

One-way tape or two-way tape?

- We are formalizing a two-way infinite tape, but using a list of squares indexed starting at zero.
- That's why the head starts in square j, not necessarily square 0.
- We make the definition of partial computation provide that if $x[i].tape[0].flag = 1$, then either $x[i+1].tape[0].flag = 1$ or $x[i+1].tape[1].flag = 1$.
- That is, the head did not move left.
- In effect that forces the machine to halt if it tries to move left of square 0.
- For any terminating computation, we can choose the initial square j so that only positive-indexed squares are used.
- This is not quite the same as a two-way infinite tape in the case of divergent computations, but since it agrees on halting computations, it computes the same functions.
- To formalize a one-way infinite tape, require the initial square to be $j = 0$ or $j = 1$; either one works but with subtle differences.

The $\mathbf{T}$-predicate and U-function

- A **computation** is a partial computation such that final row represents the machine in a halting configuration, that is, no instruction of the machine is applicable.
- If the only reason for halting was that the next instruction would have caused the head to move left of square zero, that doesn't count as a computation.
- $\mathbf{T}(e, x, k)$ means e is a Turing machine, and k is a computation by machine e at input x.
- If $\mathbf{T}(e, x, k)$, then $U(k)$ is defined to be the (integer coding the) string remaining on the tape at the end of the computation starting at the final head location; this is obtained by stripping out the bit reserved for the head marker.
- Using the head location to mark the start of the output means that if we pad a computation with blanks on the left, the output doesn't change.

Kleene Normal-form Theorem

Every Turing computable function has the form

$$f(\mathbf{x}) \cong U(\mu k \mathbf{T}(e, \mathbf{x}, k))$$

where e is a Turing machine that computes f.

Proof. $f(\mathbf{x})$ is defined if and only if there is some computation by machine e at input x. If there is such a computation that computation (regarded as a number) is a k such that $T(e, \mathbf{x}, k)$, and the result $U(k)$ of this computation is the value $y = f(x)$. That completes the proof.

μ-recursiveness and Turing computability

We showed already that every μ-recursive function is Turing computable. We would like to prove the converse.

The Kleene normal-form theorem is the key. If we can show that

- the T-predicate is primitive recursive
- the U-function is primitive recursive

then every Turing computable function is μ-recursive:

$$f(\mathbf{x}) \cong U(\mu k \mathbf{T}(e, \mathbf{x}, k))$$

Definability of $\mathbf{T}$

- Recall from Lecture 4 that every predicate defined by a bounded arithmetical formula is primitive recursive; more generally the primitive recursive predicates are closed under logical operations and bounded quantification.
- So we will show that $\mathbf{T}$ is definable by using logical operations and bounded quantification, applied to primitive recursive predicates.
- Then $\mathbf{T}$ will be shown to be primitive recursive.
- This is a bit technical, but it's the easiest way to actually prove everything.

A technical note about sequences

- It is tempting to claim that $\mathbf{T}$ is definable by a bounded arithmetic predicate, but we can't do that without more work.
- The issue here is the definability of "sequence numbers", in particular the predicate $x[i] = j$, which we have left unspecified except to say it is primitive recursive.
- Without specifying a particular encoding of sequences, we can't say for sure that it is defined by a bounded formula.
- That doesn't actually matter, since what we need is merely the closure of the primitive recursive predicates under logical operations and bounded quantification.

Some detailed definitions

- $Instr(q)$, "q is a TM instruction", is defined by
$$length(q) = 5 \wedge x[1] < 128 \wedge x[3] < 128 \wedge x[5] < 3$$
- e is a Turing machine:
$$\forall i < length(e)\, Instr(e[i])$$
- $Square(p)$ is defined by
$$(p[0] = 0 \vee p[0] = 1) \wedge p[1] < 128$$
- $ITD(r)$ is defined by
$$\forall j < length(r)\, (j > 0 \supset (Square(r[j])))$$
- The definition of $Next$ involved a bounded quantifier over e to look for an instruction in e of a certain form; so $Next$ is definable by a bounded arithmetical formula.
- $Terminal(e, r)$ says that for all $j < length(r)$, with $j > 0$, either $r[j].flag = 0$ or no instruction of e begins with state $r[0]$ and old symbol $r[j].symbol$

Bounded arithmetical definition of $\mathbf{T}(e, x, k)$

$$
\begin{aligned}
&\forall i < length(k)\, (ITD(k[i]) \wedge length(k[i]) = length(k[0]) \\
&\wedge \forall i < length(k) - 1\, \forall j < length(k[i]) - 1 \\
&\qquad (j > 0 \wedge k[i][j].flag = 1 \supset \\
&\qquad Next(e, k[i, j-1], k[i, j], k[i, j+1], \\
&\qquad\qquad k[i+1, j-1], k[i+1, j], k[1+1, j+1]) \\
&\wedge \forall j < length(k[0])(j > 0 \supset k[0][j].flag = 0) \\
&\wedge k[0][0].flag = 1 \\
&Terminal(e, k[length(k) - 1])
\end{aligned}
$$

Result-extracting function U is primitive recursive

$U(k)$ has to

- extract the last row $q = k[length(k) - 1]$ from the sequence k.
- find the head location in that row, i.e. the least $j < N$ such that $k[length(k) - 1, j].flag = 1$.
- discard $q[0], lots, q[j-1]$ and form the string of $q[j].symbol$, $q[j+2].symbol, \ldots$.
- return the integer whose binary representation is given as an ascii string by that sequence.
- These steps can be defined by course-of-values recursion
- Hence U is primitive recursive.

Recapping

- We proved that $\mathbf{T}(e, x, n)$ is defined by a bounded arithmetic formula.
- Therefore, it is primitive recursive.
- U is also primitive recursive.
- Therefore,
$$\phi_e(\mathbf{x}) \cong U(\mu k \mathbf{T}(e, \mathbf{x}, k))$$
is μ-recursive.
- But that is the function computed by the Turing machine e.
- Therefore, Turing computable implies μ-recursive.
- But we already proved μ-recursive implies Turing computable.
- Hence a function is Turing computable if and only if it is μ-recursive.

A universal Turing machine

A **universal Turing machine** is a machine M that computes the function $App(e, \mathbf{x})$, defined to be the output of Turing machine e at input $\mathbf{x}$.

We can now easily prove that there exists a universal Turing machine.

Proof. By the normal form theorem, we have

$$App(e, \mathbf{x}) \cong U(\mu k \, (T(e, \mathbf{x}, k)))$$

As we just showed, T and U are primitive recursive. Hence the right side is μ-recursive. But then, it is Turing computable (with variable e), so there some Turing machine that computes it. That is a universal machine.

Turing machines as a model of computation

- Thus we have proved rigorously that Turing machines provide a "model of computation" in the sense that there is an $App(e, x)$ function.
- Moreover, the $\mathbf{T}$-predicate shows that "computations proceed in stages". While k officially is a computation by machine e at input x, it can also be thought of as counting the steps.
- "$App(e, x)$ is defined if and only if it is defined in some number of steps" is one way of looking at the normal form theorem.

Unsolvability of the halting problem

- The predicates "Turing machine e halts at input e" and "Turing machine e halts at input x" are not Turing computable.
- More formally, the predicate

$$\exists k\, \mathbf{T}(e, x, k)$$

  is not Turing computable.

Proof. We have already shown that this follows from the existence of App, but we repeat the proof for Turing machines on the next slide.

Proof of unsolvability of halting problem

If $\exists k\, \mathbf{T}(e, x, k)$ were Turing computable, then so would be

$$f(x) \cong \begin{cases} 0 & \text{if } \exists k\,(\mathbf{T}(e, x, k) \wedge U(k) = 1) \\ 1 & \text{otherwise} \end{cases}$$

Then f is total. Suppose f is computed by Turing machine e. Then since f is total, there is some k such that $\mathbf{T}(e, e, k)$. Then $f(e) = U(k)$. Hence $U(k)$ cannot be 1, as then $f(e) = 0 \neq U(k)$. Therefore the first case in the definition cannot hold, and $f(e)$ must be 1. But then there is a computation k showing that $f(e) = 1$; and for that k, the first case of the definition will hold after all, contradiction. That completes the proof.

Towards an explicit universal TM

I do not know of any actual explicitly-constructed universal Turing machine that one can run in a simulator. All textbooks that I have seen either ask the student to take the existence of a universal machine on faith or authority, or they prove its existence by a chain of "compiling down" from more abstract machines or programming languages to Turing machines. But those compilers are to my knowledge never written out as executable computer code, so one never actually arrives at a specific universal Turing machine. Not that we will do better here; but let us survey what would be involved.

Steps to construct a universal Turing machine

- Write a program to translate a bounded arithmetic formula to a primitive recursive definition of its representing program.
- Apply that program to get an explicit primitive recursive definition of the $\mathbf{T}$ predicate.
- Write a program to translate an explicit primitive recursive definition of a function into a Turing machine that computes that function.
- Apply that program to get a Turing machine that computes the $\mathbf{T}$ predicate.
- Use the TM for the $\mathbf{T}$ predicate to obtain a machine that computes $f(e,x) = \mu k\,(\mathbf{T}(e,x,k)))$.
- Obtain a Turing machine to compute the result-extracting function U. (Either by direct programming, or by finding a primitive recursive definition of U and converting that to a Turing machine.)
- Combine U with the previous step to obtain a machine that computes $App(e,x) \cong U(f(x))$. That machine is a universal Turing machine.

Next time

- That still doesn't show, for example, that Ackermann's function, or a Python interpreter, is Turing computable.
- Next time, we will prove the "recursion theorem", showing that the Turing computable functions are closed under **arbitrary** recursions, not just primitive recursions.
- That is the convincing argument that this is really the class of "computable functions", because interpreters for computer languages can be written using recursion.
- With primitive recursion, we have for-loops. With μ-recursion, we have (unbounded) while-loops. But can you implement arbitrary recursion with loops?