CSCI3390-Lecture 11: Other computational models—small and weird programming languages

October 22, 2018

Here we look at some alternative models of computation. All of these have the same computational power as Turing machines—that is, they can compute anything that *can* be computed (subject to some restrictions about how inputs and outputs are encoded). This implies that the halting problem for each of these models is undecidable, and that there is a 'universal program' for each model.

1 A tiny subset of Python

Here we present a tiny subset of the programming language Python that is still computationally universal—in other words, it can do everything a Turing machine can do. 'Functions' definable in this programming language are actually *partial* functions

$$f : \mathbf{N} \times \cdots \times \mathbf{N} \to \mathbf{N}.$$

After deep reflection, I have decided to name this programming language Rubber Boa. [1]

Our programming language will be used to define functions, so we will include Python's syntax for introducing function definitions:

[1] I wanted to call this language 'Python with both hands tied behind its back', but quickly realized that the metaphor was zoologically inaccurate. I consulted Wikipedia and found out that the smallest constrictor snake in the world is the Rubber Boa, cute little guys that don't get more than two feet long and are so docile that they are used to help people overcome a fear of snakes..

```
def f(x,y,z):
```

Of course, you can have any number of arguments, and the argument names are arbitrary.

We also include Python's syntax for returning a function value:

```
return v
```

where `v` is a variable.

We also allow statements of the following type (where `u` and `v` are variables):

```
v=0
u=v
u+=1
v=f(u,w,x,y)
```

The third of these statements increases the value of `u` by 1. The fourth is meant as just an example of what is intended: f is assumed to be be a previously-defined function. We do not allow the function being defined to be used in the function definition—that is, *we do not allow recursion*.

We have to have *some* way of altering the flow of control, so we allow the following looping constructs from Python:

```
for u in range(v):
   SEQUENCE OF STATEMENTS

while u!=0:
   SEQUENCE OF STATEMENTS
```

The first of these repeatedly executes the sequence of statements in the body as `u` takes on the values $0, 1, .., v-1$. The second repeatedly executes the statements in the body as long as the value of `u` is nonzero.[2] Figure 3 shows several functions written in this language. These demonstrate how to add, multiply, test for zero, and decrement.

Programs in this language compute partial functions

$$f : \underbrace{\mathbf{N} \times \cdots \times \mathbf{N}}_{k \text{ times}} \to \underbrace{\mathbf{N} \times \cdots \times \mathbf{N}}_{\ell \text{ times}},$$

[2]What, you might be wondering (unless you are a Python expert) happens if I change the value of `v` *inside* the `for` statement? The answer is that the body of the loop is still executed $k-1$ times, where k is the value of `v` when execution *started*.

```
def add(x,y):
    u=x
    for j in range(y):
        u+=1
    return u

def mult(x,y):
    u=0
    for j in range(x):
        u=add(u,y)
    return u


#return 1 if x is zero, 0 otherwise
def iszero(x):
    u=0
    u+=1
    for j in range(x):
        u=0
    return u


#proper decrement. pred(0) is 0

def pred(x):
    z=0
    for j in range(x):
        z=j
    return z
```

Figure 1: How to add, multiply, test for zero, and decrement in Rubber Boa.

and the reason these are partial is that the `while` statement might never terminate.

We will shortly outline a proof that these programs, as simple as they are, can do anything a Turing machine can do, and thus can do anything that any algorithm can do. Let's give a more precise statement of this. A Turing machine $\mathcal{M}$ computes a partial function

$$f_{\mathcal{M}} : (\Gamma - \{\Box\})^* \to \Gamma^*,$$

where Γ is the tape alphabet of the machine. That is, we assume that there are no blank symbols within the input string v, and take the value of $f_{\mathcal{M}}(v)$ to be the string that is on the tape when, and if, $\mathcal{M}$ halts. We can treat the strings on the tape as encodings of integers, and under this encoding, $f_{\mathcal{M}}$ can be viewed as a partial function from $\mathbf{N}$ into itself. We will be more precise about the details of a particular encoding when we give the proof. With this in mind, our result is:

Theorem 1 *Any partial function computed by a Turing machine can be computed by a Rubber Boa program.*

1.1 Primitive recursion and min-recursion

The Rubber Boa programming language actually predates the Turing machine as a general model of computation. Of course, there being no computers back then, much less computer programming languages, they were described somewhat differently. Let's construct a family $\mathcal{F}$ of partial functions

$$f : \mathbf{N}^k \to \mathbf{N},$$

where we use $\mathbf{N}^k$ to mean the cartesian product of k copies of $\mathbf{N}$. The rules are, that $\mathcal{F}$ contains the zero function and the successor function, defined by:

$$\text{zero}(x) = 0, \text{succ}(x) = x + 1$$

for all $x \in \mathbf{N}$. $\mathcal{F}$ contains the projection functions

$$\pi_i^k(x_1, \ldots, x_k) = x_i$$

for all $n \geq 1$ and all $1 \leq i \leq n$, and is closed under composition. Closure under composition means, for example, that if

$$f : \mathbf{N} \times \mathbf{N} \times \mathbf{N} \to \mathbf{N}$$

is in $\mathcal{F}$, and the functions

$$g_1, g_2, g_3 : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$$

are in $\mathcal{F}$, then $\mathcal{F}$ contains the function

$$h : \mathbf{N} \times \mathbf{N} \to \mathbf{N}$$

defined by

$$h(x, y) = f(g_1(x, y), g_2(x, y), g_3(x, y)).$$

$\mathcal{F}$ is also closed under an operation called *primitive recursion*. This means that if $\mathcal{F}$ contains functions

$$f : \mathbf{N}^k \to \mathbf{N},$$

and

$$g : \mathbf{N}^{k+2} \to \mathbf{N},$$

then it contains the function

$$h : \mathbf{N}^{k+1} \to \mathbf{N}$$

defined by

$$h(x_1, \ldots, x_k, 0) = f(x_1, \ldots, x_k),$$

and

$$h(x_1, \ldots, x_k, y + 1) = g(x_1, \ldots, x_k, y, h(x_1, \ldots, x_k, y)).$$

If this looks odd, think of it in Rubber Boa:

```
def h(x,y,z):
    u=f(x,y)
    for v in range(z):
        u=g(x,y,v,u)
    return u
```

The functions in $\mathcal{F}$ constructed by repeated application of these rules are called *primitive recursive functions*. They are exactly the functions that you can define by programs that do not use the `while` statement. You may have noticed that all the examples we gave above used only `for` loops. All functions defined this way are *total*—that is, they are defined on every choice of arguments.

You obtain the partial *recursive* functions by closing up under one more operation, called min-recursion or μ-recursion, which adds back the power of `while`: If

$$f : \mathbf{N}^{k+1} \to \mathbf{N}$$

is in $\mathcal{F}$, then so is the partial function

$$g : \mathbf{N}^k \to \mathbf{N}$$

defined by

$$g(x_1, \ldots, x_k) = \min\{y : f(x_1, \ldots, x_k, y) = 0\}.$$

This corresponds to

```
def g(x,y,z):
    u=f(x,y,z,0)
    v=0
    while u!=0:
        v+=1
        u=f(x,y,z,v)
    return v
```

1.2 Simulating Turing machines

We will now sketch the proof that any Turing machine behavior can be captured by a program in this language.

First, we will suppose that our Turing machine has a *one way infinite tape.* As you showed in an earlier homework assignment, this will be sufficient, because any Turing machine with a two-way tape can be simulated by the one-way version.

Secondly, we will fix a way of encoding integers as strings over the tape alphabet, and conversely. Let

$$\Gamma = \{\gamma_0, \gamma_1 \ldots \gamma_{k-1}\},$$

where $k = |\Gamma|$ and γ_0 is the blank symbol. Given an integer n, we define $v_n \in \Gamma^*$ to be its base k representation *in the reverse of the usual order,* so that the least significant digit is on the left, and where the digit i is represented by γ_i. For example, if the tape alphabet is

$$\gamma_0 = \square, \gamma_1 = a, \gamma_2 = b, \gamma_3 = X,$$

then the string

$$v = abbb$$

represents the integer n whose base 4 representation is 2221. That is, $n = 89$. If the Turing machine, when started on this string, halts with

$$\Box XXX$$

on the tape, then $f_{\mathcal{M}}(89) = 252$, which is the integer whose base 4 encoding is 3330.

It should be stressed that *any* system of encoding strings by integers and vice-versa, as long as the encoding can be accomplished by an algorithm, will do here. This particular choice of encoding makes the argument particularly easy.

We can thus encode a configuration of the Turing machine as a triple of integers (c, q, p), where c encodes the tape as above, q encodes the state, and p encodes the position of the read-write head. To encode a state in Q, we simply number the states $0, 1, \ldots, |Q| - 1$ so that 0 is the start state and 1 the halt state. The other state numbers can be chosen arbitrarily. We encode the position p by the integers $1, k, k^2, \ldots$, where in general, k^j represents the j^{th} cell on the tape (with the leftmost cell treated as cell 0). Now consider the three functions

$$c' = \text{next-content}(c, q, p)$$

$$q' = \text{next-state}(c, q, p)$$

$$p' = \text{next-position}(c, q, p)$$

that give the updated contents, state, and position, after one step of the Turing machine computation. We claim that each of these functions is primitive recursive—that is, that each of them can be computed by a function in Rubber Boa that does not use `while`.

To see this, note that each of the three functions has the form:

```
def next(c,q,p):
    if q==r and current_symbol==s:
        do_something
    elif q==r' and current_symbol==s':
       do_something_else
    elif....
    .
    .
    return result
```

You will show in the homework how to implement statements like `if (x==y)...elif (x'==y'..` in Rubber Boa. How do we find out the symbol currently being scanned? It is given by the standard Python expression

```
(c//p)% k
```

In the code posted on the website, you will find implementations of the `div` and `mod` functions, which enable you to perform this computation.

What is the 'something' that gets done? In the case of the next state function, it is just assigning a new constant value to the variable `q`. This is trivial to render in the language. For example, `x=3` is just

```
x=0
x+=1
x+=1
x+=1
```

In the case of the next position function, the 'something' is just move right or move left, which is $p = k * p$ or $p = p//k$. Observe that the use of a one-way infinite tape helps us here: We can check first if $p = 1$, so that a move left will give us back 1.

In the case of the next content function, we have to replace the symbol at position `p` by a new symbol $d \in \{0, \ldots, k-1\}$. This is accomplished by

```
left=c%p
right=(c//p)//k
newright=k*right+d
newcontent=p*newright+left
```

Once more, all of these operations can be carried out in the programming language.

So our computation of $f_{\mathcal{M}}$ will be carried out by initializing c to the value written on the tape at the outset, q to the initial state 0, and p to 1. We then repeatedly update the configuration until the state is 1. Thus

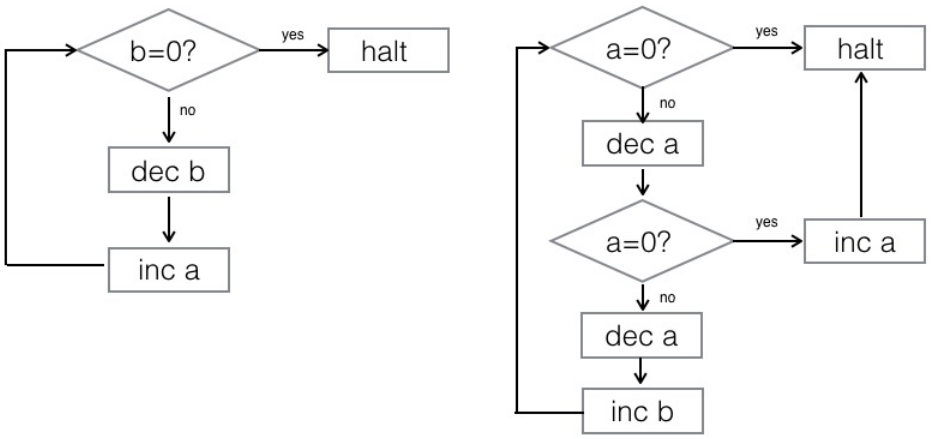

Figure 2: *A flowchart view of two counter machines, one for addition, the other computing quotient and remainder mod 2.*

```
def f(c):
        q=0
        p=1
        while q != 1:
            c=next-content(c,p,q)
            q=next-state(c,p,q)
            p=next-position(c,p,q)
        return c
```

Our basic building blocks for programs only allow `while q != 0`, but it is easy enough to implement `while q != 1` with a little tweak (see the homework problems).

2 Counter machines

Look at the flowchart diagrams in Figure 1.

The left diagram operates on a pair (a, b) of nonnegative integers. The instruction dec a means 'decrease a by 1', or, as the CS types prefer to say 'decrement a', and inc a means 'increase a by 1' (increment). Which instruction to execute next is determined by the arrows. As you can see, there is a special branching

```
loop:
goto finish if b=0
dec b
inc a
goto loop
finish:
halt
```

Figure 3: *A program view of the addition counter machine.*

instruction that tests to see if a value is 0, branching to different next instructions depending on the result of the test. By the way, if you decrement a value that is already 0, the result is taken to be 0, not minus 1. These diagrams live in a world in which there are no negative numbers.

Think of the names a and b as *variables*, or *counters* that hold values. If you start with x as the value of a and y as the value of b, then the value of a when the halt instruction is reached will be $x + y$. Thus this flowchart gives an algorithm for adding two natural numbers. You can think of this as a machine whose 'state' is the next instruction to execute: At each step, the machine either decrements or increments one of the counters, or tests whether a counter is zero, and changes to a new state. Alternatively, you can think of the flowchart as the *program* shown in Figure 2. This programming language has statements to increment and decrement variables, to branch conditionally to another statement if the value of a variable is zero, and to jump unconditionally to another statement. Labels (in this program `loop` and `finish`) are used as targets of the branch insturctions.

What about our second flowchart? Table 1 traces the values of the counters a and b if the initial value of a is 5. (We will assume that the counter b is initially set to 0.) The variable a is decremented twice in each pass through the main loop, and the variable b is increased once for each such pass. So b is keeping count of the number of times we can reduce a by 2 before falling to 0. If a falls to zero in the middle of the loop (after 1 rather than 2 decrements) then we set a back to 1. Thus, when we reach the halt instruction, a will hold the remainder of the original value modulo 2—that is, a will be 0 if the original value of a was even, and 1 if it was odd. The counter b will hold the integer part of the quotient of the original value upon division by 2. So this program computes the function

$$x \mapsto (x \bmod 2, \lfloor x/2 \rfloor).$$

a	b
5	0
4	0
3	0
3	1
2	1
1	1
1	2
0	2
1	2

Table 1: *Trace of the execution of the second counter machine on inputs $a = 5$ and $b = 0$. The result is $5 \bmod 2$ in a and $\lfloor 5/2 \rfloor$ in b. The analogous result will occur for any initial value of the counter a, so this machine computes the quotient and remainder of a upon division by 2.*

Every counter machine begins with an initial setting of several counters (in the case of our first example, two counters a and b, and in the case of the second, one counter a) and finishes with a possibly different setting of several of the counters. (In our first example, a was the only 'output' counter, but in our second, both a and b hold results we are interested in). Thus our counter machine computes a partial function

$$f : \underbrace{\mathbf{N} \times \cdots \times \mathbf{N}}_{k \text{ times}} \to \underbrace{\mathbf{N} \times \cdots \times \mathbf{N}}_{\ell \text{ times}}.$$

Why partial? Because like any programming environment, counter machines allow the possibility of infinite loops, so the machine may fail to halt on some inputs.

Theorem 2 *Any partial function computed by a Turing machine can be computed by a counter machine.*

To prove that this is true, it's enough to show that we can simulate counter machines in Rubber Boa. It's easy enough to see how to simulate `for` and `while` loops with counter machines, and that's about all you need to do to make the translation. All the usual consequences of universal computation follow from this: for example, there is no algorithm to tell whether a given counter machine halts on a given input.

3 FRACTRAN–the strangest programming language

FRACTRAN was invented by the mathematician John H. Conway. The idea seems to date from the 1960's, although Conway's paper on FRACTRAN programming did not appear until much later.

As Conway writes, the entire FRACTRAN syntax can be learned in a period of ten seconds. Actually two seconds. A FRACTRAN program is a sequence of fractions, such as

$$\left[\frac{2}{3}\right]$$

or

$$\left[\frac{11}{14}, \frac{5}{7}, \frac{3}{11}, \frac{7}{2}\right].$$

You'll need another five seconds or so to learn the semantics. A FRACTRAN program is given as input a positive integer N. At each step, we find the first fraction f in the sequence such that Nf is an integer, and replace N by Nf. We continue like this until no such fraction f can be found.

As an example, let's run the first program with input $N = 144$:

$$\begin{aligned} 144 &\longrightarrow_{\times\frac{2}{3}} 96 \\ &\longrightarrow_{\times\frac{2}{3}} 64. \end{aligned}$$

And the second with input $N = 32$:

$$\begin{aligned} 32 &\longrightarrow_{\times\frac{7}{2}} 112 \\ &\longrightarrow_{\times\frac{11}{14}} 88 \\ &\longrightarrow_{\times\frac{3}{11}} 24 \\ &\longrightarrow_{\times\frac{7}{2}} 84 \\ &\longrightarrow_{\times\frac{11}{14}} 66 \\ &\longrightarrow_{\times\frac{3}{11}} 18 \\ &\longrightarrow_{\times\frac{7}{2}} 63 \\ &\longrightarrow_{\times\frac{5}{7}} 45 \end{aligned}$$

So what? What is being 'computed' here? What I didn't tell you is how the inputs and outputs are encoded. The first program is supposed to take two inputs x, y and encode them as $2^x 3^y$. The output is supposed to be z, where 2^z is the

value on which the program halts. In our example, $144 = 2^4 \cdot 3^2$, and the final result is $64 = 2^6$. You can see that if I repeatedly multiply $2^x \cdot 3^y$ by $\frac{2}{3}$, I will wind up with 2^{x+y}. So our first FRACTRAN program adds two integers.

What about the second? The representation of inputs and outputs here is to encode a single input x as 2^x, and to decode the result, which will have the form $3^y \cdot 5^z$, as (y, z). In our example, $x = 5$, $y = 2$ and $z = 1$. In this case, y and z are respectively the quotient and remainder that x leaves upon division by 2, and in fact that is what the second program will give you for *every* initial value x.

Subject to these rules about the encoding of inputs and outputs, every Turing-computable function can be computed by a FRACTRAN program, so here is another universal model of computation.

How were these FRACTRAN programs arrived at? There is relatively simple trick for converting counter machine programs into FRACTRAN programs. (This is one of the project topics, and I will leave the details to any students who choose this topic.)