15-354: CDM **K. Sutner**

Assignment 2 Due: **Sep. 15, 2023**.

1. Loopy Loops (40)

Background

Consider a small programming language LOOP with the following syntax:

constants	$0 \in \mathbb{N}$
variables	x, y, z, ... ranging over $\mathbb{N}$
operations	increment $x{+}{+}$
assignments	$x = 0$ and $x = y$
sequential composition	$P; Q$
control	`do` $x : P$ `od`

The semantics are obvious, except for the loop construct: `do` $x : P$ `od` is intended to mean: "Let n be the value of x before the loop is entered; then execute P exactly n times." Thus, the loop terminates after n rounds even if P changes the value of x. For example, the following LOOP program computes addition:

```
//  add : x, y --> z
        z = x;
        do y :
              z++;
        od
```

Here x and y are input variables, and the result is in z. We assume that all non-input variables are initialized to 0. So, we have a notion of a LOOP-computable function.

Task

A. Show how to implement multiplication and the predecessor function as LOOP programs.

B. What function does the following loop program compute?

```
// mystery : x --> x
   do x:
      do x: x++ od
   od
```

C. Show that every primitive recursive function is LOOP-computable.

D. Show that every LOOP-computable function is primitive recursive.

Solution: Loop Programs

Part A: Multiplication, Predecessor

Here is multiplication as a LOOP-program (with nested loops).

```
//  mult : x, y --> z
    do x :
```

```
    do y :
       z++;
    od
od
```

We have indicated the input variables x and y, and the output variable z in line 1 of the program.

And predecessor:

```
// pred : x --> z
    do x :
       z = v;
       v++;
    od
```

So z is lagging behind v by 1 after line 4 is executed, and upon termination v has value x.

Part B: Mystery

The function is $x \mapsto x \cdot 2^x$: the inner loop simply doubles the value of x.

Part C: PR is Loop

We should start with a careful definition of the semantics of a loop program. Since the basic idea is clear, we only will refer to the input variables of a program and the output variable; other details are ignored here.

It is easy to see that the basic primitive recursive functions can all be computed by a LOOP program, so we only have to show that we can deal with composition and primitive recursion.

Composition

Consider $f = g \circ (h_1, h_2, \ldots, h_m)$ where each h_i is computed by some loop program H_i and g is computed by loop program G. We may safely assume that there is no clash between the variables in these programs (otherwise rename). Let's suppose that the input variables for H_i are x_{ij} and the output variable is y_i. Also, the input variables for G are z_i and the output variable is z. Let $x_1, x_2, \ldots, x_n$ be fresh variables. Then the loop program

$$\begin{aligned} x_{11} = x_1; x_{12} = x_2; \ldots; x_{1n} = x_n; &\quad H_1; \\ x_{21} = x_1; x_{22} = x_2; \ldots; x_{2n} = x_n; &\quad H_2; \\ \ldots & \\ x_{m1} = x_1; x_{m2} = x_2; \ldots; x_{mn} = x_n; &\quad H_m; \\ z_1 = y_1; z_2 = y_2; \ldots; z_m = y_m; &\quad G \end{aligned}$$

computes f with input variables $x_1, x_2, \ldots, x_n$ and output variable z. So this is really just a question of handing over appropriate values; composition is built into loop programs.

Primitive Recursion

Consider $f = \mathsf{Prec}[g, h]$. For simplicity assume that f is binary so that $f(0, y) = g(y)$ and $f(x^+, y) = h(x, f(x, y), y)$.

Assume by induction that loop programs G and H compute g and h, respectively. Assume further that G has input y, output z and H has input xx, z and y, and output u. Then the program

```
G;  xx = 0;
do x:
   H;
   z = u;
   x++;
od
```

computes f with input x, y and output z.

Part D: Loop is PR

Let P be any LOOP program, and suppose that the variables in P are $x_1, \ldots, x_m$. We denote by x_i' the value of x_i after execution of P. To avoid complications, let us assume that all variables have been assigned specific values before execution of P. We will also be a little sloppy in distinguishing between a variable and its value. We have to show that the update functions

$$f_i(x_1, \ldots, x_m) = x_i'.$$

which describe the value of any specific variable as a function of all the variables, after one execution of P, are all primitive recursive.

Proof is by induction on P. For loop-free P the claim is trivial, and is not hard to see that composition of programs preserves this property. So suppose P has the form

```
// P
    do t:  Q;  od
```

We may safely assume that t does not occur in Q, otherwise rename variables. But then we can ignore variable t altogether since it's value does not change. In one step the values change according to

$$\begin{array}{cccc} x_1 & x_2 & \ldots & x_m \\ f_1(\boldsymbol{x}) & f_2(\boldsymbol{x}) & \ldots & f_m(\boldsymbol{x}) \end{array}$$

Recall that we can code all the inputs into a single value using sequence numbers. We can accordingly modify the f_i so that they take this sequence number as input; call the new p.r. functions g_i. Now define

$$h(x) = \langle g_1(x), g_2(x), \ldots, g_m(x) \rangle$$

Clearly h is p.r. By primitive recursion define

$$\begin{aligned} H(0, \boldsymbol{x}) &= \langle \boldsymbol{x} \rangle \\ H(t^+, \boldsymbol{x}) &= h(H(t, \boldsymbol{x})) \end{aligned}$$

Then H is p.r. and we can obtain a p.r. update functions for P by decoding (extracting the ith component).

2. Register Machines and Sequence Numbers (30)

Background

Recall the coding function for sequences of natural numbers introduced in class:

$$\begin{aligned} \pi(x, y) &= 2^x(2y + 1) \\ \langle \mathsf{nil} \rangle &= 0 \\ \langle a_1, \ldots, a_n \rangle &= \pi(a_1, \langle a_2, \ldots, a_n \rangle) \end{aligned}$$

Task

A. Give a simple bound on $\langle a_1, \ldots, a_n \rangle$ in terms of n and $\max a_i$.

B. Construct a register machine program `digcnt` that, on input x, returns the number of binary digits of x (no leading zeros).

C. Construct a register machine program `append` that, on input $\langle a_1, \ldots, a_n \rangle$ and b, returns $\langle a_1, \ldots, a_n, b \rangle$.

D. Roughly, what is the running time of your programs?

Comment

Make sure to give a detailed explanation of how your programs work, plain RMP code translates into 0 credit. A flowgraph might be a good idea, too.

For the running time do not try to come up with a precise answer, just order of magnitude.

Solution: RMs and Sequence Numbers

Part A: Bound

A simple induction on n shows that $\langle a_1, \ldots, a_n \rangle$ has the form

$$1\underbrace{00\ldots0}_{a_n}1\underbrace{00\ldots0}_{a_{n-1}}\ldots1\underbrace{00\ldots0}_{a_2}1\underbrace{00\ldots0}_{a_1}$$

Hence we have the bound

$$\langle a_1, a_2, \ldots, a_n \rangle < 2^{n+\sum a_i} \leq 2^{n(a+1)}$$

where $a = \max a_i$.

Part B: Digcnt

This is really a modification of the digit sum machine, except that this time we need to increment our counter for all digits, not just 1s.

```
0:  dec X 1 3
1:  dec X 2 3
2:  inc Y 0
3:  inc D 4
4:  dec Y 5 6
5:  inc X 4
6:  halt
```

Part C: Append

Append is a bit harder than prepend (see lecture slides), we have to add a block $1\underbrace{0\ldots0}_{b}$ to the binary expansion of x.

To this end, suppose we already have a register $U = X$ and D holding the digit count of x. Then execute the following program to compute $Z = 2^{D+b+1}$

```
 0:  dec B 1 2
 1:  inc D 0
 2:  inc D 3
 3:  inc Z 4
 4:  dec D 5 10
 5:  dec Z 6 8
 6:  inc V 7
 7:  inc V 5
 8:  dec V 9 4
 9:  inc Z 8
10:  halt
```

For the append program, instead of halting, have the last machine add X to Z. Done.

Part D: Running Time

Each round takes about $5/2\,x+1$ steps, and the value of x is reduced to $x/2$. So the total damage is about $5\,x+O(\log x)$, linear in x.

For the whole append operation we get $O(x2^b)$ steps.

3. The Busy Beaver Function (RM) (30)

Background

The Busy Beaver function β is a famous example of a function that is just barely non-computable. For our purposes, let's define $\beta(n)$ as follows. Consider all register machines P with n instructions and no input (so all registers are initially 0). Executing such a machine will either produce a diverging computation or some output x_P in register R_0. Define $\beta(n)$ to be the maximum of all x_P as P ranges over n-instruction programs that converge.

It is intuitively clear that β is not computable: we have no way of eliminating the non-halting programs from the competition. Alas, it's not so easy to come up with a clean proof. One line of reasoning is somewhat similar to the argument that shows that the Ackermann function is not primitive recursive: one shows that β grows faster than any computable function.

Task

A. Show that, for any natural number m, there is a register machine without input that outputs m and uses only $O(\log m)$ instructions.

B. Assume $f : \mathbb{N} \to \mathbb{N}$ is a strictly increasing computable function. Show that for some sufficiently large x we must have $f(x) < \beta(x)$.

C. Conclude that β is not computable.

D. Prof. Dr. Blasius Wurzelbrunft sells a device called HaltingBlackBox™ that allegedly solves the Halting Problem for register machines. Explain how Wurzelbrunft's gizmo could be used to compute β.

Comment

The bound in part (A) is far from tight in special cases: some numbers m have much shorter programs: think about 2^{2^k}. But, in general $\log m$ is impossible to beat (Kolmogorov-Chaitin program-size complexity). Part (D) says that β is K-computable.

Solution: The Busy Beaver Function

Part A: Generating m

A simple loop can be used to replace the contents x of a register by $2x$ and likewise for $2x + 1$. By using $\log m$ many of these loops we can construct m from its binary expansion.

Part B: Domination

Suppose f is a computable, strictly increasing function. Then, for any number n, we can construct a program P_n as follows:

```
// P_n:
   m = f(2*n);
   return m;
```

So P_n returns $f(2n)$. Written out as register machine, P_n requires at most $\log n + c$ instructions for some constant c (essentially the submachine computing f) by part (A). But then for sufficiently large n (more precisely, we need $\log n + c \leq n$) we have

$$\beta(n) \geq f(2n) > f(n),$$

as required.

Part C: Non-Computability

Follows from part (B): If β were computable, $\beta'(n) = \max\big(\,\beta(i) \mid i < n\,\big) + n$ would also be computable and is strictly increasing by brute force.

Alternatively we could argue here that β itself must already be strictly increasing, so there is really no need for β'.

Part D: Wurzelbrunft

Given Wurzelbrunft's device we can compute $\beta(n)$ as follows. First generate a list of all programs of size n. While this list is wildly exponential in size, it is still finite and easy to generate in principle. Next use Wurzelbrunft's box to remove all those programs from the list that fail to terminate. Now run all the remaining programs to completion: since they all halt we can simply keep computing till all the machines have finished. Lastly, determine which machine has generated the largest output.

Offering money for Wurzelbrunft's device is a bad idea since it contradicts the unsolvability of the Halting problem. You might as well purchase a perpetual motion machine or believe in sustainable growth.