CMPT 409: Theoretical Computer Science **Summer 2017**

Lecture 4: Computability (12 - 16 June)

Lecturer: Ternovska, Eugenia *Scribe: Lily Li*

4.1 Functions

Let ∞ denote *undefined.*

Definition 4.1 *A **partial function** is a funtion $f : (\mathbb{N} \cup \{\infty\})^n \to \mathbb{N}\{\infty\}$ such that $f(c_1, ..., c_n) = \infty$ if some $c_i = \infty$.*

*The **domain** of function f is $Dom(f) = \{\bar{x} \in \mathbb{N}^n : f(\bar{x}) \neq \infty\}$ where $\bar{x} = (x_1, ..., x_n)$. If $Dom(f) = \mathbb{N}^n$ then f is **total**.*

4.2 Register Machines

A register machine RM is a finite set of registers, $R_1, ..., R_m$, each storing some natural number. Program $P = \langle c_0, ..., c_{h-1}\rangle$ where each c_i is a command. The list of allowed commands are:

1. z_i: zero register R_i.
2. S_i: increment register R_i.
3. $J_{i,j,k}$: jump to command c_k if $R_i = R_j$.

The **state** of a RM is $\langle k, R_1, ..., R_m\rangle$ where k is the index of the next command to execute and R_i the content of register i.

The map $\mathsf{Next}_P(s)$ returns the state resulting from the execution of one step of the program P in state s. State s_i is **halting** if $k = h$. Then $\mathsf{Next}_P(s) = s$. For current state $s = \langle k, R_1, ..., R_m\rangle$, $\mathsf{Next}_P(s)$ is defined as:

1. If $k < h$ and $c_k = s_i$ then the new state is $\langle k + 1, R_1, ..., R_i + 1, ..., R_m\rangle$.
2. If $k < h$ and $c_k = z_i$ then the new state is $\langle k + 1, R_1, ..., R_{i-1}, 0, ..., R_m\rangle$.
3. If $k < h$ and $c_k = J_{i,j,k'}$ then the new state is $\langle k', R_1, ..., R_m\rangle$ if $R_i = R_j$ and $\langle k + 1, R_1, ..., R_m\rangle$ if $R_i \neq R_j$ and $k < h$.

The computation of P is a sequence (finite or infinite) of states $s_0, s_1, ...$ such that $s_{i+1} = \mathsf{Next}_P(s_i)$ when $i \geq h$ — ending with the halting state. A program computes a function $f(a_1, ..., a_n)$ if $s_0 = \langle 0, a_1, ..., a_n, 0, ..., 0\rangle$ and when started from s_0, P halts with $R_1 = f(a_1, ..., a_n)$. If P fails to halt, then $f(a_1, ..., a_n) = \infty$.

Conversely, for each P and $n \geq 0$, there is an n-ary function f_P computable by P. We say that such an f is *computable* if some RM computes it.

Example 4.2 *Consider how we would implement the function $f(x) = x - 1$ using a RM. First observe that we are dealing with natural numbers we need to assume that $x - 1 \geq 0$. We begin with $s_0 = \langle 0, x, 0 \rangle$. At a high level, the program should set R_3 (currently zero) to 1, then simultaneously increment R_1 and R_3 until the latter is equal to x. Formally the computation can be preformed using the following sequence — with s_i the halting state:*

$$c_0 = s_3, c_1 = J_{2,3,5}, c_2 = s_1, c_3 = s_3, c_4 = J_{1,1,1}$$

It is a good exercise to consider how you would implement $f(x, y) = x \cdot y$ as a RM.

Definition 4.3 *A function f defined from g and h by* ***primitive recursion*** *if and only if*

$$f(\bar{x}, 0) = g(\bar{x})$$
$$f(\bar{x}, y + 1) = h(\bar{x}, y, f(\bar{x}, y))$$

where $\bar{x} = x_1, ..., x_n$. $n = 0$ is allowed. Here you want to think of g as the base case and h as the recursive step.

Example 4.4 *Lets consider how to define addition using primitive recursion: we define $f_+(x, y)$ as follows: let $f_+(x, 0) = g(x) = x$ and $f_+(x, y + 1) = h(x, y, f_+(x, y)) = f(x, y) + 1$. Notice that g is essentially the identity and h is the successor function on the third argument.*

Next we use $f_+(x, y)$ to define $f_.(x, y)$. The base case is $f_.(x, 0) = g(x) = 0$ (see below). Let $f_.(x, y + 1) = h(x, y, f_.(x, y)) = x + f_.(x, y)$. In particular $h(x, y, z) = f_+(I_{3,1}(x, y, z), I_{3,3}(x, y, z)$ where $z = f_.(x, y)$.

Finally we define exponentiation $f_\uparrow(x, y)$ using $f_.(x, y)$. For our base case $f_\uparrow(x, 0) = g(x) = s(z_1(x))$ where $z_1(0)$ is the constant function which evaluates to zero (see below). Next $f_\uparrow(x, y + 1) = g(x, y, f_\uparrow(x, y)) = x \cdot f_\uparrow(x, y)$ in particular $g(x, y, z) = f_.(I_{3,1}(x, y, z), I_{3,3}(x, y, z)$.

You should think of primitive recursion as those functions which are bounded by construct. This is not true of functions in general, so RM is a strict subset of all TM.

Definition 4.5 *f is defined from g and $h_1, ..., h_m$ by* ***composition*** *if an only if*

$$f(\bar{x}) = g\left(h_1(\bar{x}), ..., h_m(\bar{x})\right)$$

where $f, h_1, ..., h_m$ are h-ary and g is m-ary.

Definition 4.6 *f is* ***primitive recursive*** *if and only if f can be obtained from the initial functions: $Z : 0, S : s(x) = x+1, I_{n,i}(x_1, ..., x_n) = x_i$ where $i \leq i \leq n$ (projection) by finitely many applications of primitive recursion and composition.*

Example 4.7 *Consider using the above definition of primitive recursion to define the constant function $z_1(x) = 0$. Let $z_1(0) = Z = g$ (Z as in the definition). $z_1(y + 1) = z_1(y) = h(y, z_1(y))$ where $h = I_{1,1}$.*

Generally for the constant function $k_{n,i}$,

$$k_{n,i}(x_1, ..., x_n) = i$$

We can also define $k_{n,i}$ using recursion! In particular our base case is $z_1(x)$. $k_{n,i} = s\left(\cdots s\left(z_1\left(I_{i,1}\right)\right)\cdots\right)$ where the successor function s (again defined above) is applied i times.

Proposition 4.8 *All primitive recursive functions are* total. *Recall a function is* total *if $f(\bar{x}) \neq \infty$ for every $\bar{x}$ in the domain.*

Proof: By structural induction. Note that the three initial functions are total and primitive recursion and composition preserve totality. ■

Theorem 4.9 *All primitive recursive functions are computable (on a RM).*

Proof: By induction on the number of compositions and primitive recursions of f. In the base case we must compute the initial functions using register machines. This is simple. Recall the three primitive recursions of f are: zero, successor of x and projection onto variable x_i. These correspond to zeroing, successor function on cell i and a repeated successor function followed by a jump instruction to the end.

First we show that composition is computable. Assume functions $g, h_1, ..., h_m$ are computable by programs $P_g, P_1, ..., P_m$ respectively by the induction hypothesis. We will show that $f(\bar{x}) = g(h_1(\bar{x}), ..., h_k(\bar{x}))$.

Suppose the initial input $(x_1, ..., x_n)$ at registers $R_1, ..., R_n$. We will copy $x_1, ..., x_n$ into registers $R_{1+k}, ..., R_{n+k}$ where k is greater than any register used by $P_1, ..., P_m$. Execute each P_i after modifying the jump instructions to point to something sensible. Put the result h_i of P_i in cell R_{n+k+i}. Upon starting a new P_j copy $x_1, ...x_n$ back into registers $R_1, ..., R_n$. When all P_i have been executed, copy $h_1, ..., h_n$ into registers $R_1, ..., R_n$ and zero all other registers. Run P_g on input $h_1, ..., h_n$.

To show that primitive recursion is computable on an RM, remark that it is defined using a base and an inductive function. These can be simulated on an RM in the usual fashion. ■

From the previous theorem we see that primitive recursive functions are a subset of all functions computable by a register machine. In particular, this inclusion is strict since there exists function which are not total but are computable by RM (recall all primitive recursive functions *must* be total). Here are two other ways to show that the subset is strict.

Theorem 4.10 *There exists total and computable functions that are not primitive recursive.*

Proof: Use Cantor's diagonalization argument. First we specify a natural number to each primitive recursive function. This is possible since each P.R. function can be uniquely defined by its application of primitive recursion and composition. Turn these description strings into numbers in the standard fashion.

Let function f_e be specified by $e \in \mathbb{N}$. Since we are only interested in unary functions, define the sequence $\mathcal{G} = g_1, g_2, ...$ where $g_i = f_i$ if f_i is unary and g_i be the constant zero function otherwise. $\mathcal{G}$ is an "effective" enumeration of all unary P.R. functions — that is, the universal function $U(x, y) = g_x(y)$ is computable.

We show that U is not P.R. Suppose for a contradiction that U is P.R. Then the diagonal function $D(x) = g_x(x) + 1$ is also primitive recursive and there exists an e such that $D = g_e$. This is a contradiction since $D \neq g_e$ for all $e \in \mathbb{N}$. ■

Theorem 4.11 *Ackermann's function is total but not primitive recursive. (It is also clearly computable and therefore RM computable by Church Thesis.)*

Proof: Define Ackermann's function as follows:

$$A_0(x) = \begin{cases} x+1 & \text{If } x = 0 \text{ or } x = 1 \\ x+2 & \text{Otherwise} \end{cases}$$

And for each $i > 0$, A_i is defined recursively as:

$$A_{n+1}(0) = 1$$
$$A_{n+1}(0) = A_n(A_{n+1}(x))$$

In general we can define $A(n, x) = A_n(x)$. Further we will define $F(x) = A(x, x)$ and we will show the following lemma:

Lemma 4.12 *$F(x)$ is not primitive recursive.*

Proof: (This is only a sketch of the proof. There is one induction step missing.) For any P.R. function $h(\vec{x})$ is *dominated* $A(n, x)$. That is there exists some $B \in \mathbb{N}$ such that if $x_1, ..., x_n > B$ then $h(x_1, ..., x_n) < A(n, \max(x_1, ..., x_n))$ — this can be proven induction. This proves to be an issue since if $F(x)$ is a primitive recursive function then there must be constants n, B such that $A(n, x) > A(x, x) = F(x)$ given $x > B$ which is impossible (since $A(x, y)$ is a monotone increasing function in both x and y). ■

Since $F(x)$, a restricted form, is not primitive recursive, neither is $A(n, x)$ in general. ■