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

Lecture 5: Computability (26 June - 10 July)

Lecturer: Ternovska, Eugenia *Scribe: Lily Li*

5.1 Recursive Functions

Definition 5.1 *Minimization is another operations we need for recursion (not primitive recursion).*

$$f(\bar{x}) = \mu y[g(\bar{x}, y) = 0] \iff f(\bar{x}) \text{ is the least } b \text{ s.t. } g(\bar{x}, b) = 0 \text{ and } g(\bar{x}, i) \neq \infty \text{ for } i < b$$

$f(\bar{x}) = \infty$ if no such b exists.

Note: *here 0 represents true, while 1 represents false.*

If g is computable then so is f since we can calculate $g(\bar{x}, i)$ for increasingly large i starting from 0.

Definition 5.2 *f is recursive if and only f can be obtained from the initial functions from finitely many applications of primitive recursion, composition, and minimization.*

Observe that P.R. function is recursive, and all recursive function are computable (since we can apply induction on the number of primitive recursion, composition and minimizations used).

We want to show the converse (below), but first we need some background.

Theorem 5.3 *All computable functions (those computable by a RM) are recursive.*

In fact, any function computable in exponential time is P.R. More generally if f can be computed in time $T(x) = O(A_m(x))$ for some fixed m, where A_m is the Ackermann's function, then f is P.R.

Let us see some examples of how we define function using our basic tools of primitive recursion, complementation, and minimization (mostly just p.r.).

Example 5.4 *First we define the predecessor function ρ, taking care that the range is within $\mathbb{N}$.*

$$\rho(x) = \begin{cases} x - 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0 \end{cases}$$

We can define ρ by P.R. as $\rho(0) = 0 = g$ and $\rho(x+1) = x = h(x, \rho(x))$ where $g = Z$ and $h = I_{2,1}$.

Next we define the limited subtraction function Δ:

$$f_\Delta(x, y) = x \Delta y = \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{otherwise} \end{cases}$$

Define $f_\Delta(x, 0) = x = g(x)$ where $g(x) = I_{1,1}(x)$. Further define $f_\Delta(x, y+1) = h(x, y, x\Delta y) = \rho(x\Delta y)$ where ρ is the predecessor function from before and $h = \rho(I_{3,3})$.

Finally we define the maximum function using our limited subtraction in $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$. *Define*

$$\max(x, y) = (x \Delta y) + y$$

Observe that if $x > y$ *then* $\max(x, y) = x$ *and if* $x \leq y$ *then* $\max(x, y) = y$ *as one would expect. Since both the subtraction and addition functions are P.R., their composition is P.R. as well.*

5.2 Relations and 0-1 Functions

For $R \subseteq \mathbb{N}^n$, n-ary relation. Define the characteristic function of R as:

$$R(\bar{x}) = \begin{cases} 0 & \text{if } \bar{x} \in R \\ 1 & \text{if } \bar{x} \notin R \end{cases}$$

Again note 0 is true and 1 is false for this part of the course. We consider R computable (resp. P.R.) if and only if the characteristic function is computable (resp. P.R.).

Next we will try to obtain logic using primitive recursive functions. But before we do that lets define a useful P.R. function sign, $\bar{sg} = 1 \Delta x$ i.e.

$$\bar{sg}(x) = \begin{cases} 0 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \end{cases}$$

Proposition 5.5 *If R and S are P.R. (resp. computable) then so are*

1. $\neg R$: $(\neg R)(\bar{x}) = \bar{sg}(R(x))$.
2. $R \vee S$: $(R \vee S)(\bar{x}) = R(\bar{x}) \cdot S(\bar{x})$.
3. $R \wedge S$: $(R \wedge S)(\bar{x}) = \neg(\neg R \vee \neg S)(\bar{x})$.

MOAR operations which preserves recursive functions and relations (as well as preserving computable relations and computable functions).

Example 5.6 ***Relations:***

$$\begin{aligned} (x < y) &= \bar{sg}\,(y \Delta x) = \begin{cases} 0 & \text{if } x < y \\ 1 & \text{if } x \geq y \end{cases} \\ (x = y) &= \neg(x < y) \wedge \neg(y < x) \\ (x \leq y) &= (x < y) \vee (x = y) \end{aligned}$$

Example 5.7 ***Bounded Sum***

$$g(\bar{x}, y) = \sum_{z<y} f(\bar{x}, z) = f(\bar{x}, 0) + \cdots + f(\bar{x}, y - 1)$$

g can be defined from f by P.R. as: $g(\bar{x}, 0) = 0$ *and* $g(\bar{x}, y + 1) = g(\bar{x}, y) + f(\bar{x}, y)$.

Example 5.8 ***Bounded Product***

$$h(\bar{x}, y) = \prod_{z<y} f(\bar{x}, z) = f(\bar{x}, 0) \cdots f(\bar{x}, y-1).$$

Similarly, h can be defined from f by P.R. as $h(\bar{x}, 0) = 1 = g$ and $h(\bar{x}, y+1) = h(\bar{x}, y) \cdot f(\bar{x}, y)$.

Example 5.9 ***Bounded Quantification***

$$\begin{aligned} S(\bar{x}, y) &= \exists z < y R(\bar{x}, z) \\ &= \prod_{z<y} R(\bar{x}, z) = R(\bar{x}, 0) \cdots R(\bar{x}, y-1) \\ &= h(\bar{x}, y) \end{aligned}$$

Similarly, the universal quantifier is P.R. since

$$\begin{aligned} T(\bar{x}, y) &= \forall z < y R(\bar{x}, z) \\ &= \neg \exists z < y \neg R(\bar{x}, z) \end{aligned}$$

Thus the computability of R implies the computability of S and T.

Example 5.10 ***Number Theoretical Functions***

$$\begin{aligned} \mathsf{Prime}(x) &= \begin{cases} 0 \text{ if } x \text{ is prime } 1 \text{ otherwise} \end{cases} \\ &= (1 < x) \wedge (\forall z < x : \neg(z|x) \vee z = 1) \end{aligned}$$

where

$$x|y = \exists z \leq y : x \cdot z = y$$

Definition 5.11 *We will define a conditional by cases:*

$$\begin{aligned} f(\bar{x}) &= \begin{cases} g(\bar{x}) \text{ if } R(\bar{x}) \\ h(\bar{x}) \text{ otherwise} \end{cases} \\ &= Cond(R(\bar{x}), g(\bar{x}), h(\bar{x})) \\ &= \bar{sg}(x) \cdot y + \bar{sg}(\bar{sg}(x)) \cdot z \end{aligned}$$

Where

$$Cond(x, y, z) = \begin{cases} y & \text{if } x = 0 \\ z & \text{if } x > 0 \end{cases}$$

As before, if g, h, R are P.R. (resp. computable) then so is f.

Example 5.12 ***Bounded Minimization***

$$f(\bar{x}, y) = \min z < y R(\bar{x}, z) = \begin{cases} \text{ least } z \text{ s.t. } z < y \text{ and } f & \text{if such } z \text{ exists} \\ y & \text{otherwise} \end{cases}$$

Notice that (1) f is always total since R is a total $0-1$ valued function, and (2) f is P.R. (resp. computable) whenever R is P.R. (resp. computable) since

$$f(\bar{x}, y) = \sum_{z<y} (\exists v \leq z : R(\bar{x}, v))$$

The idea is to use bounded summation as bounded minimization.

5.3 Simulating RM with Recursive Function

We want to consider all computable function (those computable by RM) and show that these can be expressed recursively. Consider

$$T([P], \bar{x}, y) \iff y \text{ is a halting computation of program } P \text{ on input } \bar{x}$$

where $[P]$ is the encoding of program P. If T is P.R. then the function $\min_y : T([P], \bar{x}, y)$ is recursive and so the function associated with P is recursive. In the following we construct T and show that it is P.R.

5.4 Gödel Numbering

We wish to encode the follow: commands of RMs, programs (sequences of commands), states, computations (sequences of states) as natural numbers. To do this, we must first define the i^{th} prime function $f(i) = p_i$ and note that $f(i)$ is P.R. Let

$$\begin{aligned} p_0 &= f(0) & &= 2 \\ p_{x+1} &= f(x+1) & &= \min y < p_x^{p_x} \end{aligned}$$

This means that $p_x < y$ and $\mathsf{Prime}(y)$ so $f(x+1)$ is the first prime after p_x.

Next we need to observe that prime decomposition is also P.R. Notation: z_x is the exponent of p_x in the prime decomposition of z where

$$z = p_0^{z_0} \cdot p_1^{z_1} \cdots p_m^{z_m}$$

Thus we can write z_x as

$$\min y < z : \neg(p_x^{y+1} | z)$$

Finally we need length $l(z)$ to be P.R. Length is defined as follows:

$$l(z) = m + 1 \qquad \text{where } z = p_0^{z_0} \cdots p_m^{z_m}.$$

Alternatively, we can encode $l(z)$ as

$$\begin{aligned} l(z) &= \min y < z : \prod_{x<y} P_x^{z_x} = z \\ &= 1 + \text{ subscript of largets prime which divides } z \\ &= \max y < z : (p_y | z) + 1 \end{aligned}$$

5.4.1 Numbering Programs

Using the three P.R functions above, we can number programs as follows. If $P = \langle c_0, ..., c_{n-1} \rangle$ then $\#(P) = p_0^{\#(c_0)} \cdots p_m^{\#(c_{n-1})}$ where

$$\begin{aligned} \#(z_i) &= 2^i \\ \#(s_i) &= 3^i \\ \#(J_{i,j,k}) &= 5^i 7^j 11^k \end{aligned}$$

Thus distinct programs get distinct numbers and the encoding $\#P$ of P is P.R.

5.4.2 Encoding Relations

$$\begin{aligned}
\mathsf{Succ}(x,i) &\iff x = \#(s_i) \\
\mathsf{Zero}(x,i) &\iff x = \#(z_i) \\
\mathsf{Jump}(x,i,j,k) &\iff x = \#(J_{i,j,k}) \\
\mathsf{Command}(x) &\iff \exists i < x : (\mathsf{Succ}(x,i) \lor \mathsf{Zero}(x,i)) \lor (\exists i < x, \exists j < x, \exists k < x : \mathsf{Jump}(x,i,j,k))
\end{aligned}$$

Let $\mathsf{Prog}(z) \iff z = \#(P)$ for some program P. $\mathsf{Prog}(z)$ is P.R since

$$\begin{aligned}
\mathsf{Prog}(z) &\iff z \text{ encodes a sequence of commands} \\
&\iff \forall j < l(z) : \mathsf{Command}(z_j)
\end{aligned}$$

5.4.3 States

Consider RM program P, which uses registers $R_1, ..., R_m$. A state for P is $s = \langle k, R_1, ..., R_m\rangle$ where k is the index of the next command to execute and $R_1, ..., R_m$ is the value in each register. We encode a state by:

$$u = code(s) = p_0^{k_0} p_1^{R_1} \cdots p_m^{R_m}$$

5.4.4 Computations

Next we consider computations. Define

$$\hat{z} = \begin{cases} z \text{ if } Prog(z) \text{ encodes a program} \\ 1 \text{ otherwise} \end{cases}$$

And for our use define

$$\{z\} = \begin{cases} \text{program } P \text{ s.t. } \#(P) = \hat{z} & \text{if } \mathsf{Prog}(z) \\ \Lambda(\text{empty program}) & \text{if } \neg\mathsf{Prog}(z) \end{cases}$$

$\mathsf{Nex}(u, z)$ = the code of the state u' which results when program $\{z\}$ executes one step from the state encoded by u. If u is a halting state $u' = u$. To show that Nex is P.R. we would iterate through the cases.

if $(u_0, ..., u_t)$ is a sequence of codes of states in a computation of P, we encode the entire computation by

$$y = p_0^{u_0} \cdot p_1^{u_1} \cdots p_t^{u_t}$$

Define $\mathsf{halt}(u, z)$ = "u encodes a halting state of $\{z\}$". Observe that $\mathsf{halt}(u, z)$ is P.R. since $\mathsf{halt}(u, z) = (u)_o \geq l(\hat{z})$. Recall

- $u = code(s) = p_0^k \cdot p_1^{R_1} \cdots p_m^{R_m}$ and $(u)_0$ = exponent of p_0 in the prime decomposition of u, so $(u)_0 = k$.
- If $z = p_0^{z_0} \cdots p_m^{z_m}$, is the encoding of a program ten $l(z) = m + 1$ (there are $m + 1$ primes).

5.5 Kleene T Predicate

Recall $T(P, \bar{x}, y)$ from above. Using Gödel encoding we can rewrite T as $T_n(z, x_1, ..., x_n, y) = y$ encodes the computation of $\{z\}$ on input $\bar{x} = (x_1, ..., x_n)$ where T_n is a $n+2$-ary relation.

Theorem 5.13 *For each $n \geq 1$, T_n is a P.R. relation.*

Proof: $T_n(z, \bar{x}, y)$ holds if and only if y encodes a computation $(u_0, ..., u_t)$, with initial state

$$u_0 = p_0^0 p_1^{x_1} \cdots p_n^{x_n}$$

and for all $i < t$,

$$u_{i+1} = \mathsf{Nex}(u_i, z)$$

with the last state as a halting state and all previous states, non-halting.

$$\begin{aligned} T_n(z, \vec{x}, y) = [(y)_0 = p_1^{x_1} p_2^{x_2} \cdots p_n^{x_n} \wedge \forall y < t[(y)_{i+1} = \mathsf{Nex}((y)_i, z)] \\ \wedge \mathsf{halt}((y)_t, z) \\ \wedge \forall i < t \neg \mathsf{halt}((y)_i, z) \end{aligned}$$

where the first line signifies the initial state, the second that the last state is halting and third that no intermediate state is halting. ■

The output function $U(y)$ is the contents of R_0 in the final state of the computation encoded by y so $U(y) = ((y)_{l(y)-1})_1$ and is P.R. Note that $\{z\}_n$ is the n-ary function computed by program $\{z\}$.

Theorem 5.14 ***(Kleene Normal Form Theorem)*** *There is a P.R. function U and $\forall n \geq 1$ a P.R. predicate T_n s.t.*

$$\{z\}_n(x_1, ..., x_n) = U(\mu y T_n(z, \bar{x}, y))$$

is the content of R_0 in the final state of computation of program $\{z\}$ on input $x_1, ..., x_n$. Note: the least y is only y satisfying the condition.

Corollary 5.15 *Every computable function is recursive and can be obtained using at most one application of μ.*

5.6 Universal Function Φ_n

Define $\Phi_n(z, \bar{x}) = \{z\}_n(\bar{x})$. Φ_n is the universal function as it encodes every computable function of n variables as z varies. Φ_n is recursive (thus computable), for $n = 1, 2, ...$ We call a program computing Φ_n an **interpreter**. Observe that if we define $\phi_i(x) = \Phi_1(i, x)$ then we can enumerate all (parital) unary computable functions. Here it is important to include non-total functions.

Theorem 5.16 *There is no computable universal function for the set of all total computable unary functions.*

Proof: By a diagonalization argument. Suppose for a contradiction that there exists a satisfying computable universal function. Enumerate all unary computable functions $\phi_0, \phi_1, ...$

Let $F(z,x) = \phi_z(x)$. We show that F is not computable in the usual way. Consider the diagonal function $D(x) = F(x,x) + 1$. If F were computable, then so would D. Thus $D = \phi_e$ for some e. However $f_e(e) = D(e) = F(e,e) + 1 = f_e(e) + 1$ which is a contradiction, thus D and by extension F is not computable. ∎

Consider why this same proof *cannot* be used as a proof that no computable universal function for the set of all computable unary functions (note here the functions need not be total). Consider the argument again: we enumerate all computable unary functions. Again we construct another function $D(x)$ which has the value $F(x,x) + 1$. Since we assumed that

Theorem 5.17 ***(Church's Thesis)*** *Every "algorithmically computable" function is computable (on a RM, TM, λ-calculus). If we give an informal algorithm to compute a function then we claim that it is computable.*

Note: this theorem cannot be proved because many believe that algorithmically computable is an informal notion. Attempt to challenge this notion by attempting a definition.

5.7 Recursive Sets

Theorem 5.18 *The halting set is not recursive (since recursive and computable functions are the same, this is equivalent to saying that the halting set is not computable).*

Proof: Let $\{x\}$ be the unary function computable by program $\{x\}$. Define

$$K = \{x : \{x\}_1(x) \neq \infty\} = \{x : \text{ program coded by } x \text{ halts on input } x\ \}$$

As a function

$$K(x) = \begin{cases} 0 (true) \end{cases}$$

The structure of the proof is as follows: first we will construct and show that some function D is not computable. Then by reduction we will show that if K is computable then so is D.

With a suggestive name like D, you will can probably guess that we are going to use the diagonalization trick again.

Next suppose that K, the halting set, is computable. The process is quite straight forward. Construct a program as follows: on input x, execute x on K. If $\{x\}$ halts on input x, then the program enters an infinite loop. Otherwise the program returns 0. Observe that this program exactly computes D. Note: if you want to be more specific in defining the program using a register machine, you should zero R_1, increment R_1 to x then run the register machine for program $\{x\}$ with modified outputs. ∎

Corollary 5.19 *Let $f(x) = \mu y T(x,x,y)$. Then f is a (partial) computable function which has no extension to a total computable function i.e. there is no total computable function $g(x)$ such that $g(x) = f(x)$ for each x such that $f(x) \neq \infty$.*

Proof: Suppose there was a totally computable extension of f. Then this same extension will show that K is recursive, a contradiction. ∎

Definition 5.20 *Set $A \subset \mathbb{N}^n$ (and also a relation) is* recursive *(also computable, also decidable) if $A(\bar{x})$, its characteristic function, is computable as a total 0-1 valued function.*

This implies that A is recursive if and only if there exists a program P that, given any n-tuple $\bar{x}$ as input, halts in finite time and outputs 0 if $\bar{x}$ in the set A and 1 otherwise.

Proposition 5.21 *If $R(\bar{x}, y)$ is recursive, $f(\bar{x})$ is total and computable, then $R(\bar{x}, f(\bar{x})$ is recursive. This is necessary due to the closure of total computable functions under composition.*

5.8 Rice's Theorem

Definition 5.22 *$A \subseteq \mathbb{N}$ is a* function index set *if*

$$\forall e \in A \forall e' : \{e\} = \{e'\} \implies e' \in A$$

i.e. A contains a code for a program that computes a unary function ϕ then A must contain codes for all programs that computes ϕ.

Theorem 5.23 *(Rice) If A is a function index set and A is non-trivial ($A \neq \emptyset$ and $A \neq \mathbb{N}$), then A is not computable.*

Another useful theorem for reductions is:

Theorem 5.24 *(SMN - special case) Let $g(x, y)$ be computable, then $\exists$ total computable function $f : \mathbb{N} \to \mathbb{N}$ s.t. $\{f(x)\}(y) = g(x, y)$ for all x, y.*

Proof: Essentially this theorem says that given a function g and a fixed $x = x_0$ we can create a unary function $f_{x_0}(y)$ which behaves the same way as $g(x_0, y)$. Think about it. Yeah, isn't this really straight forward? Let us exhibit such a program. Put input y in register R_2, and x_0 in the register R_1. Run the program for g with adjusted jump instructions (now you have a couple more steps at the start to do the copying). ■

Lets see how we can use the S-m-n theorem to do a reduction:

Example 5.25 *Let $H = \{x : \{x\}_1(0) \neq \infty\}$. Show that $K \leq_m H$ where K is the diagonal halting problem. We construct the binary function $g(x, y) = \{x\}(x) = \Phi(x, x)$ where Φ is the universal function. By the S-m-n theorem there exists a function f such that*

$$\{f(x)\}_1(y) = g(x, y) = \{x\}_1(x), \quad \text{for all} x, y$$

thus $\{f(x)\}_1(0) = \{x\}_1(x)$ and $\{f(x)\}_1(0) \neq \infty$ if and only if $\{x\}_1(x) \neq \infty$.

5.9 Recursively Enumerable Sets

Definition 5.26 *If $A \subseteq \mathbb{N}^n$ then A is* recursively enumerable *(aka. computably enumerable, semi-decidable if there exists a recursive relation $R \subseteq \mathbb{N}^{n+1}$ such that*

$$\vec{x} \in A \iff \exists y R(\vec{x}, y), \quad \text{for all } \vec{x} \in \mathbb{N}^n$$

Intuitively, we understand this as all sets which can be enumerated by some program.

Note: every recursive set is recursively enumerable, but the converse is not true — consider the diagonal halting set.

Lemma 5.27 *If $A \subset \mathbb{N}$ then A is R.E. if and only $A = \emptyset$ or $A = ran(f)$, where ran is the range function, where $f : \mathbb{N} \to \mathbb{N}$ is a total computable function.*

Proof: In the forward direction we suppose that A is R.E. and $A \neq \emptyset$. That is there exists a relation R such that $a \in A \implies \exists y : R(x, y)$. We will first construct a binary function $F(x, y)$ from the given relation then transform F into a unary function.

In the converse direction, suppose that $A = ran(f)$ for some total computable function f. We need to come up with a recursive relation R. Consider $R(x, y) := (x = f(y))$. Since equality and f are both recursive functions, R is recursive. ■

Now we have many equivalent characterizations of R.E. sets.

1. A is RE
2. $A = \text{dom}(f)$ for some recursive function f.
3. $A = \{x : \exists y R(x, y)\}$ for some primitive recursive relation R.
4. $A = \emptyset$ or $A = \text{ran}(f)$ for some primitive recursive unary function f.
5. $A = \text{ran}(f)$ for some recursive unary function f (f not necessarily total).

Definition 5.28 *Set A is R.E.-complete if and only if A is R.E. and for any other R.E. set B, $B \leq_m A$.*

Remark: the notion of R.E.-completeness precedes that of NP-completeness. In-fact you can say that Cook was inspired by this notion to define and prove NP-completeness.

Theorem 5.29 *Suppose f is a partial n-ary function then f is computable if and only if graph(f) is R.E.*

Proof:

■

Theorem 5.30 *A is recursive if and only if both A and A^c are R.E.*

Proof: In the forward direction, since A is recursive, A is R.E. Next we observe that the complement of a recursive function ■