Theory of Formal Languages (Implementing PR on TMS)

Lecture 9: March 20, 2021

Prof. K.R. Chowdhary *: Professor of CS*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

9.1 Primitive Recursion

9.1.1 Generalized Composition

Next we look for combining these functions into a new function. The first such method is composition (also called *substitution*). The composition of two functions $f, g : \mathbb{N} \rightarrow \mathbb{N}$ of one argument (also called variable) is well known; the composition is defined as follows:

$$(f \circ g)(x) = f(g(x)). \tag{9.1}$$

However, we usually drop the symbol '$\circ$', and simply write fg. Since composition of functions is associative, we also omit the parenthesis in the expression with iterated composition, e.g., we write $fgh(x)$ for $f(g(h(x)))$.

Many a times, we wish to deal with functions of more than one argument, this is called *generalized composition*.

Definition 9.1 Generalized composition. *Let $f : \mathbb{N}^n \rightarrow \mathbb{N}$ be a function of n variables ($n > 0$), and let $g_1, ..., g_n : \mathbb{N}^m \rightarrow \mathbb{N}$ be n functions of m variables. Then the function $h : \mathbb{N}^m \rightarrow \mathbb{N}$, is defined by,*

$$h(x_1, ..., x_m) = f(g_1(x_1, ..., x_m), ..., g_n(x_1, ..., x_m)) \tag{9.2}$$

is generalized composite of f and g_1, ..., g_n. □

Usually, we write $h = f \circ \langle g_1, ..., g_n \rangle$ for the composite just defined, but some times we also write with notation $h = comp(f, g_1, ..., g_n)$. Note that each of the functions $g_1, ..., g_n$, and the function h, must have the same number of arguments for this to work.

Any constant function can be obtained from basic functions using the composition. This can be also proved using induction method as follows:

The constant function $(x_1, ..., x_k) \mapsto 0$ is equal to composition $Z \circ P_1^k$. If $P_1^k(x_1, ..., x_k) \mapsto n$ is given, then composing this with the successor function gives the function $(x_1, ..., x_k) \mapsto n+1$.

Example 9.2 *Composing the functions.*

There are several ways to show that a function is built using basic functions. Consider the function $f(x, y) = y + 3$ could be decomposed as $f = SSSP_2^2$. The equally valid could be:

$$f = S \circ P_2^2 \circ \langle S \circ P_2^2, SS \circ P_2^2 \rangle.$$

The above can be verified by calculating as follows:

$$\begin{aligned} S \circ (P_2^2 \circ \langle S \circ P_2^2, SS \circ P_2^2 \rangle)(x, y) =& SP_2^2(SP_2^2(x, y), SSP_2^2(x, y)) \\ =& SP_2^2(S(y), SS(y)) \\ =& S(y + 2) \\ =& y + 3 \end{aligned}$$

Note that the expressions $SSSP_2^2$ and $S \circ P_2^2 \circ \langle S \circ P_2^2, SS \circ P_2^2 \rangle$ embody the algorithm for computing the function f. □

Definition 9.3 Primitive recursion: Parameterized version. *Let $k \geq 0$, and let $g : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ and $f : \mathbb{N}^k \rightarrow \mathbb{N}$ be given functions. Define a new function $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ as follows:*

$$\begin{aligned} h(x, 0) &= f(x), \\ h(x, n + 1) &= g(x, n, h(x, n)) \end{aligned} \tag{9.3}$$

The h is said to be defined from g, f by primitive recursion. We shall write $h = P_r[g, h]$. In generalized form x may be atken as $x_1, ..., x_m$. □

First observe that, the simple version of PR function is indeed an instance of this version, when $k = 0$. In addition, in the definition 9.3 above we call the variables $x_1, ..., x_n$ as parameters, and the last variable is *recursion variable*, i.e., $h(x, n)$ in equation 9.3. Apart from this, the pair of equations is of the above form is called *recursion equation* for h.

Before we go for the explanation for above, we proceed for another definition.

Definition 9.4 Class of primitive Recursive Functions. *The class $\mathcal{PR}$ is the least class of functions with following properties:*

1. *The basic functions Z, S, P are in $\mathcal{PR}$*
2. *If $f, g_1, ..., g_n$ are all in $\mathcal{PR}$, then so is $f\langle g_1, ..., g_n \rangle$, provided this is well-defined. The class $\mathcal{PR}$ is closed under generalized composition.*
3. *If $f : \mathbb{N}^k \rightarrow \mathbb{N}$, and $g : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ are in $\mathcal{PR}$, then so is $P_r[g, f]$. That is, the class $\mathcal{PR}$ is closed under primitive recursion.*

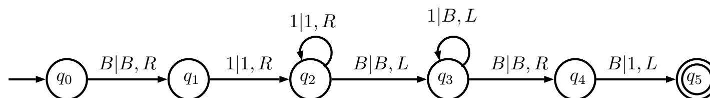

Figure 9.1: Turing machine for zero function

The functions $\mathcal{PR}$ are called *primitive recursive* functions. The above formal definition can be abbreviated as an informal definition, as follows:

Definition 9.5 *The $\mathcal{PR}$ is the least set which conatins the basic functions, and is closed under composition and primitive recursion.* □

The inductive characteristic of $\mathcal{PR}$ means that in order to show that a given function is primitive recursive, we need to show that it is either a basic function, or that it can be obtained via composition or primitive from the functions which we have already been shown to be primitive recursive.

In the following we consider number of examples of primitive recursive functions, and the implementation of some of them through Turing machine. Given a function, it requires to rewrite it to match the definition of PR function in definition 9.3 above, and then show that its constituents are themselves primitive recursive.

9.2 Implementation of PR functions on TM

Following are some examples of primitive recursive functions. Most number-theoretic functions, which can be defined using recursion on a single variable are primitive recursive.

9.2.1 Zero function

The zero function Z is a function,

$$Z(x) = 0, \tag{9.4}$$

that makes tape contents of Turing machine as zero. The machine that computes zero function, converts all 1s into Bs, except the first 1', i.e., $Z(1.1^n) = 1$.

Note that, in the Fig. 9.1, the Turing machine scans the entire input, and while returning back to original position it converts all 1s to 0s, and then first digit is made 1 to indicate that value is zero. This could also be done by leaving 1st 1 in left most position unchanged, and remaining all the 1s are converted into Bs in forward move of tape-head.

The two different mechanisms computing the same function indicates the difference between function and algorithm – the two machines are two algorithms to computes the same function Zero.

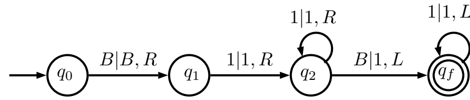

Figure 9.2: Turing machine (S) for successor function

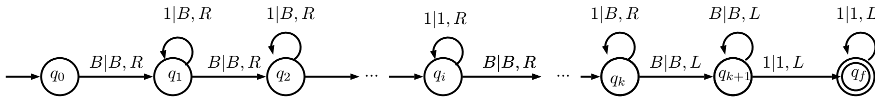

Figure 9.3: Turing machine for Projection function

9.2.2 Successor Function

The successor function S is the function,

$$S(x) = x + 1, \tag{9.5}$$

that converts the contents of Turing machine from 1^k to 1^{k+1}. Hence, the machine that computes the successor function, add a 1 immediately after the input to its right (see Fig. 9.2).

9.2.3 Projection Function

A k-variable projection function $P_i^{(k)}$ returns ith argument out of total k arguments, i.e.,

$$P_i^{(k)}(n_1, n_2, ..., n_i, ..., n_k) = n_i. \tag{9.6}$$

The TM in Fig. 9.3 shows the arguments $n_1, n_2, ..., n_k$ available on tape separated by 'B's initially. Once the TM is run, all arguments, except n_i becomes blank, and only n_i remains on the tape with tape-head pointing to the begin of this argument.

9.2.4 Predecessor Function

The predecessor function is defined as,

$$pred(n) = \begin{cases} 0, & \text{if } n = 0 \\ n - 1, & \text{otherwise.} \end{cases} \tag{9.7}$$

It is computing D machine (i.e., decrement), which removes the right most 1 for the input greater than zero, else it keeps input unchanged (see Fig. 9.4).

The *predecessor* function acts as the opposite of the successor function and is recursively defined by the following rules:

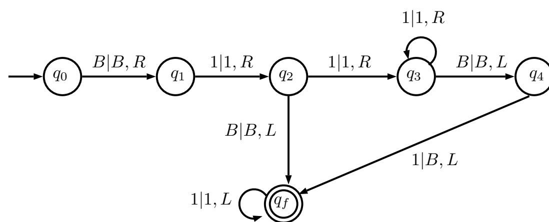

Figure 9.4: TM D for Predecessor function

$$\begin{aligned} pred(0) =& 0, \\ pred(y+1) =& y. \end{aligned} \tag{9.8}$$

These rules can be converted into a more formal definition by primitive recursion:

$$\begin{aligned} pred(0) =& 0, \\ pred(S(y)) =& P_2^2(pred(y), y). \end{aligned} \tag{9.9}$$

9.2.5 Addition Function

the addition $h(x, y) = x + y$. We show that this function can be obtained by primitive recursion from simpler functions. First consider the equations:

$$\begin{aligned} x + 0 =& x \\ x + (y+1) =& (x+y) + 1. \end{aligned} \tag{9.10}$$

This suggests that we take $f(x) = x$, and $g(x, y, z) = z + 1$. Then,

$$\begin{aligned} x + 0 =& f(x) \\ x + (y+1) =& g(x, y, x+y,) \end{aligned} \tag{9.11}$$

as required, and $h = P_r[g, f]$. Since f is a basic function and g can be obtained as composite of the successor and a projection, both are PR and thus h is PR.

The addition function can be recursively defined with following rules.

$$\begin{aligned} add(x, 0) &= g(x) = x \\ add(x, y+1) &= h(x, y, add(x, y)) \\ &= SP_3^3(x, y, add(x, y)) \\ &= S(add(x, y)) \end{aligned}$$

Thus, we note that addition is achieved by applying the S function repeatedly.

The function *add* computes the sum of two natural numbers. The definition of $add(x, 0)$ indicates that the sum of any number and zero is number itself. The recursion step defines the sum of x and $y + 1$ in the form of sum of x and y, which is the result of the addition of previous value of the recursive variable, incremented by 1.

An application of above is:

$$\begin{aligned} add(3, 2) &= add(3 + 1) + 1 \\ &= S(add(3 + 1)) \\ &= S(S(add(3 + 0) + 1)) \\ &= S(S(S(add(3 + 0)))) \\ &= S(S(S(3))) \\ &= 5. \end{aligned}$$

In order to fit in strict primitive recursive definition, we define the *Add* function as follows.

$$\begin{aligned} add(x, 0) &= P_1^{(1)}(x) \\ add(x, S(y)) &= SP_3^{(3)}(x, y, add(y, x)) \end{aligned}$$

Here $P_3^{(3)}$ is the projection function that takes three arguments and returns third argument. The $P_1^{(1)}$ in the first rule is simply identity function. Its inclusion is required by the definition of the primitive recursive function operator above, and it plays the role of g in equation 9.3 (page no. 9-2). The composition of S and $P_3^{(3)}$, which is primitive recursive, plays the role of f.

9.2.6 Subtraction function

Because primitive recursive functions use natural numbers rather than integers, and the natural numbers are not closed under subtraction, a limited subtraction function is studied in this context. This limited subtraction function $sub(a, b)$ returns $b - a$ if it is non negative and returns 0 otherwise.

The limited subtraction function is definable from the predecessor function in a manner analogous to the way addition is defined from successor:

$$\begin{aligned} sub(0, x) =& P_1^{(1)}(x), \\ sub(S(y), x) =& pred(P_3^{(3)}(x, y, sub(y, x))). \end{aligned} \tag{9.12}$$

Here $sub(a, b)$ corresponds to $b - a$; for the sake of simplicity, the order of the arguments has been switched from the standard definition to fit the requirements of primitive recursion. This could be easily rectified using composition with suitable projections.

9.2.7 Multiplication Function

The multiplication is implemented using primitive recursive functions of addition, subtraction, and predecessor. The multiplication $h(x, y) = x.y$ can be obtained by considering first,

$$\begin{aligned} x.0 =& 0 \\ x.(y+1) =& (x.y) + x. \end{aligned} \tag{9.13}$$

Thus we take $f(x) = 0$, and $g(x, y, z) = z + x$, so that,

$$\begin{aligned} x.0 =& f(x) \\ x.(y+1) =& g(x, y, x.y). \end{aligned} \tag{9.14}$$

Now if f is a basic function, and g may be written as the composite $g(x, y, z) = P_3^3(x, y, z) + P_1^3(x, y, z)$. Since addition has been already shown to be PR, this shows that multiplication is also PR. Also, we have,

$$\begin{aligned} 1 * x =& x \\ x * (y+1) =& x * pred(y+1) + x. \end{aligned} \tag{9.15}$$

Alternatively, it can be also specified as,

$$\begin{aligned} mult(0, x) =& 0, \\ mult(y+1, x) =& add(mult(y, x), x). \end{aligned} \tag{9.16}$$

9.2.8 Exponentiation function

The exponentiation function can be implemented by multiplication.

$$\begin{aligned} x^0 &= 1, \\ x^{y+1} &= x^y * x \end{aligned} \tag{9.17}$$

Alternatively, we can have

$rexpt(0, x) = 1,$
$rexp(y+1, x) = mult(rexp(y, x), x),$
$rexp(y, x) = rexp \circ (P_2^2, P_1^2).$
$supexp(0, x) = 1,$
$supexp(y+1, x) = exp(x, supexp(y, x)).$