Theory of Formal Languages (Primitive Recursive Functions)

Lecture 8: March 15, 2021

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

8.1 Primitive Recursive Functions

In the following we explain what primitive recursive functions are, and how they are constructed. We shall consider two ways of building new functions from old, namely generalized composition (i.e., substitution) and primitive recursive.The idea of inductive definition and primitive recursion goes back to Dedekind (1888) , who was first to give a charatcerization of the well-order structure on natural numbers. He also introduced the the class of primitive recursive functions, which were later defined by Skolem (in 1923) and Gödel (in 1931) [rogers1987]. The Primitive Recursive (PR) functions are an example of broad and interesting class of functions which can be obtained by formal characterization given in their definition in the following.

The primitive recursive functions are among the number-theoretic functions which are functions from the natural numbers (non negative integers) $0, 1, 2, ...$ to the natural numbers (i.e., $\mathbb{N} \to \mathbb{N}$). These functions take n arguments for some natural number n and are called n-ary, represented as $f : \mathbb{N}_1 \times \mathbb{N}_2 \times ... \times \mathbb{N}_n \to \mathbb{N}$. These functions demonstrate the computations in the most fundamental form.

The primitive recursive (PR) functions are defined using primitive recursion, and composition as central operations, they are proper subsets of recursive functions. The recursive functions are also called *computable functions*. In *computability theory*, the *PR* functions are a class of functions, which form an important building block on the way to fully formalize the *theory of computability*. These functions are also useful in *proof theory* to generate proofs of algorithms.

Most of the functions we study in number theory are primitive recursive. For example, addition, subtraction, multiplication, exponentiation, and nth factorial are all primitive recursive. These operations are basis for almost all arithmetic. Thus, it is difficult to think of a functions, which is not primitive recursive.

One method of characterizing a class of functions is to take, as members of the class, all functions obtainable by certain kind of *recursive definition*. A recursive definition for a function is, roughly speaking, a definition wherein values of the function for given arguments are directly related to values of the same function for "simpler" arguments or to values of "simpler" functions. The "simpler" here means, for example, constant functions, as the simplest of all. The recursive definitions can often be made to serve as algorithms.

The recursive definitions are familiar in mathematics. For instance, the function f defined by

$$\begin{aligned} f(0) &= 1, \\ f(1) &= 1, \\ f(x+2) &= f(x+1) + f(x), \end{aligned} \tag{8.1}$$

gives the Fibonacci sequence: $1, 1, 2, 3, 5, 8, \ldots$, where argument $x + 2$ is computed in the form of two simpler arguments $x+1$ and x; when x is 1 or 0, the value of function is already given.

8.1.1 Basic functions

The class of primitive recursive functions is defined in terms of *basic functions.*

Definition 8.1 Basic functions. *The following functions are called basic functions:*

- Zero Function. $Z(x) = 0$ *(The constant function* $\mathbb{N} \to \mathbb{N}$ *with value* 0*)*
- Successor Function. $S(x) = x + 1$
- Projection functions. $P_k^n(x_1, x_2, ..., x_k, ..., x_n) = x_k$ *(where* $0 \leq k \leq n$*)* $\square$

As a special case of a projection function, we may take $n = k = 1$, then we get the function $P_1^1(x) = x$, called *identity function.* Usually we denote this identity function by $I(x) = x$.

The identity functions are typically used to return individual arguments of the function being constructed.

Some examples the projections are:

$$\begin{aligned} u_1^1(x) &= x, \\ u_1^2(x, y) &= x, \\ u_2^2(x, y) &= y, \\ u_1^3(x, y, z) &= x. \end{aligned}$$

Because writing them every time becomes cumbersome, so for the sake of brevity, we can refer to arguments directly, by writing definitions like $f(x, y) = x + y$, instead of $f(x, y) = u_1^2(x, y) + u_2^2(x, y)$. However, we know that writing arguments directly, means an implicit application of one of the identity functions. Similarly, we should strictly form all constants by composition of the successor and zero functions, which again is cumbersome process. Hence we prefer to write 3 in place of $S(S(S(0)))$, where S is successor function.

A projection function is used to select one of the arguments. As an example, $f(w_1, w_2) = g \circ (P_2^2(w_1, w_2), P_1^2(w_1, w_2))$ such that $f(w_1, w_2) = g(w_2, w_1)$. The latter is an example of permutation of a function arguments.

The projection functions can be used to avoid the apparent rigidity in terms of the *arity* of the *PR* functions. Using the composition, with various projection functions, it is possible to pass a subset of the arguments of one function to another function. For example, consider that there are two functions g and h which are PR and each has *arity-2*. Then,

$$f(a,b,c) = g(h(c,a), h(a,b)),$$

is primitive recursive. A formal definition of above using projections function can be given to compute f using g and h.

$$f(a,b,c) = g(h(P_3^3(a,b,c), P_1^3(a,b,c)), h(P_1^3(a,b,c), P_2^3(a,b,c))).$$

8.1.2 Primitive Recursion

Merely the facility to compute the functions does not make the primitive recursion some thing special, but the its constructions that greatly expand the ability of primitive recursive functions. In the following we give a simple version to understand the main ideas. After this we will go for general version of PR.

Definition 8.2 Primitive Recursion: Simple version. *Let $g : \mathbb{N}^2 \to \mathbb{N}$ be a function of two variables, and let $c \in \mathbb{N}$ be a number. Define a new function $h : \mathbb{N} \to \mathbb{N}$ as follows:*

$$\begin{aligned} h(0) =& c, \\ h(x+1) =& g(x, h(x)), \end{aligned} \tag{8.2}$$

then h is said to be defined from g, c by primitive recursion. *We will use a notation $h = P_r[g, c]$.*$\square$

Note that we defined h in its own terms. The idea is that, in order to calculate $h(x)$ for given value of x, we need to calculate $h(x-1)$ (its previous value), and for then extending this calculation to $h(x-(x-1)) = h(1)$, and finally to $h(x-(x)) = h(0)$. Note that $h(1) = g(0, h(0))$, so having calculated $h(0)$ we are able to compute $h(1)$, then $h(2)$, ..., $h(x)$. The $c = h(0)$ is given as constant. Thus, the above definition of $h(x)$ is an algorithm for computing $h(x)$.

The following example explains the primitive recursions.

Example 8.3 *Explain the computation of the function defined by $h(x) = 2x$.*

Note that this function satisfies the PR equations:

$$\begin{aligned} h(0) =& 0, \\ h(x+1) =& h(x) + 2. \end{aligned}$$

In order to get the above equation in the form of equation 8.2, we construct another equation $g(m, n) = n + 2$, so that $g(x, h(x)) = h(x) + 2$. The function g can in turn be obtained from the basic functions using composition, i.e., $g = SSP_2^2$, and thus h can be obtained via composition and primitive recursion, using the basic functions[1].

Note that the expression of h has embedded in it an algorithm for computing its values: for example, to compute $h(3)$, we use $h(3) = h(2+1) = g(2, h(2))$. This reduces the computation to $h(2) = h(1 + 1) = h(1) + 2$. Thus, we need to compute $h(1) = h(1 + 0) = h(0) + 2$, and finally we substitute $h(0) = 0$ in this equation. Now we work backward to get the answer:

$$h(3) = h(2) + 2 = h(1) + 2 + 2 = h(0) + 2 + 2 + 2 = 0 + 2 + 2 + 2 = 6.$$

Consider the function,

$$f(x) = \begin{cases} 1, & \text{if } x = 0, \\ x - 1, & \text{otherwise.} \end{cases} \tag{8.3}$$

We may formulate this function as follows:

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

The above equations means, taking $g(x, y) = x$ gives $f(x + 1) = x = g(x, f(x))$, the latter matches with the PR functions equation 8.2

In the following we make use of *projection* function.

$$f(x) = \begin{cases} 1, & \text{if } x = 0, \\ x * f(x - 1), & \text{otherwise.} \end{cases} \tag{8.4}$$

We may formulate this as,

$$\begin{aligned} f(0) &= 1, \\ f(x + 1) &= g((x + 1), f(x)), \end{aligned} \tag{8.5}$$

where $f(x+1) = g((x+1), f(x))$ matches with equation 8.2. The g is multiplication function that computes $(x + 1) * f(x)$.

Example 8.4 *Use the definition of equation 8.4 to compute functions $f(4)$ using primitive recursion.*

[1] In many texts, the *basic functions* are also called *base functions*

$$\begin{aligned} f(4) =& g(4, f(3)) \\ =& g(4, g(3, f(2))) \\ =& g(4, g(3, g(2, f(1)))) \\ =& g(4, g(3, g(2, g(1, f(0))))) \\ =& g(4, g(3, g(2, g(1, 1)))) \\ =& g(4, g(3, g(2, 1))) \\ =& g(4, g(3, 2)) \\ =& g(4, 6) \\ =& 24 \end{aligned}$$

Note that the function defined in equation 8.4 is a function to compute the factorial of a positive integer, and its primitive recursion function's definition is given in by equation 8.5. □

Consider that, it is required to compute the sum of integers 1 to n, i.e.,

$$sumint(x) = \sum_{n=0}^{x} n, \tag{8.6}$$

which adds all integers from 0, up to x.

The sum of all integers up to and including x may be specified recursively with two functions: $g() = 0$ and $g(x, y) = x + y + 1$, which allows us to calculate that

$sumint(0) = g() = 0,$
$sumint(1) = g(0, sumint(0)) = g(0, 0) = 1,$
$sumint(2) = g(1, sumint(1)) = g(1, 1) = 3,$
$sumint(3) = g(2, sumint(2)) = g(2, 3) = 6,$
$sumint(4) = g(3, sumint(3)) = g(3, 6) = 10,$

and so on. Note that, the function *sumint* is "add" function, and $g(x, y) = S(x + y)$. Thus, to find out the value of $sumint(4)$, we move backward, i.e., find the $sumint(3)$, and use this and 3 are arguments of function g, the latter is $S \circ add$ functions, i.e, find the addition, and applies successor function S. □