Theory of Formal Languages (PR Functions are computable, PR Predicates)

Lecture 11: March. 30, 2021

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

11.1 PR functions are Computable

We want to show that every PR function is mechanically computable. Given the general strategy we described in previous section, it is enough to show it in three statements:

1. The basic functions (Z, S, P) are computable.
2. If f is defined by composition from computable functions g and h, then f is also computable.
3. If f is defined by primitive recursion from the computable functions g and h, then f is also computable.

The statement '1.' above is trivial: the initial functions S (successor), Z (zero), and P_i^k (projection) are effectively computable by a simple algorithm; and statement '2.' – the composition of two computable functions g and h is computable (we just feed the output from whatever algorithmic routine evaluates g as input, into the routine that evaluates h).

To illustrate statement '3.' above, return to factorial function, defined as:

$$0! = 1$$
$$(Sy)! = y! \times Sy$$

The first clause gives the value of the function for the argument 0;,then we repeatedly use the second clause which is recursion, to calculate the function's value for $S0$, then for $SS0$, $SSS0$, etc. The definition encapsulates an algorithm for calculating the function's value for any number, and corresponds exactly to a certain simple kind of computer routine.

Compute PR functions using loop

Compare the PR definition of the factorial with the following schematic program:

1. $fact = 1$; {initialize $fact$ by 0!}

2. $for\ y = 0\ to\ n - 1$
3. $\quad fact = (fact \times Sy)$
4. $loop$

In the line 1, *fact* is a memory register that we initialize with 0!, in 2nd line program enters a loop. The crucial thing about executing a 'for'loop is that the total number of iterations to be run through is fixed in advance: we number the loops from 0, and in executing the loop, we increment the counter y by one each in cycle. So in each step, the program replaces the value in the register with Sy times the previous value (we assume that computer already knows how to find the successor of y and do that multiplication). When the program exits the loop after a total of n iterations, the value in the register *fact* is $n!$ [petsmth10].

More generally, for any one-place function f defined by recursion in terms of g and the computable function h, the same program structure always does the trick for calculating $f(n)$, where h is multiplication function. Now compare,

$$\begin{aligned} f(0) &= g \\ f(Sy) &= h(y, f(y)) \end{aligned} \tag{11.1}$$

with the corresponding program:

1. $func = g$
2. $for\ y = 0\ to\ n - 1$
3. $\quad func = h(y, func)$
4. $loop$

Note that, so long as h is (already) computable, the value of $f(n)$ will be computable using 'for' loop that terminates with the required value in the register $func$. Similarly, of course, for many-place functions are computed. For example, the value of the two-place function defined by,

$$\begin{aligned} f(x, 0) &= g(x) \\ f(x, Sy) &= h(x, y, f(x, y)) \end{aligned} \tag{11.2}$$

is calculated by the algorithmic program

1. $func = g(m)$
2. $for\ y = 0\ to\ n - 1$
3. $\quad func = h(m, y, func)$
4. $loop$

which gives the value for $f(m, n)$ so long as g and h are computable.

Now, our mini-program for the factorial calls the multiplication function which can itself be computed by a similar *for loop.* that invokes addition, and addition can in turn be computed by another *for loop* that invokes the successor.

There is a program for the factorial, containing nested *for loops*, which ultimately calls the primitive operation of incrementing the contents of a register by 1 (or other operations like setting a register to zero corresponding to the zero function, or copying the contents of a register corresponding to an identity projection function).

The above point obviously generalizes, giving us: primitive recursive functions are effectively computable by a series of (possibly nested) *for'loops.*

The converse is also true. Take a *for* loop which computes the value of a function f for given arguments – a loop that calls on two prior routines, one computes a function g (used to set the value of f with some key argument set to zero), and other computes a function h (which is used on each loop to fix the next value of f as that argument is incremented). This plainly corresponds to a definition by recursion of f in terms of g and h. It generalizes as follows: if a function can be computed by a program using just *for loops* as its main programming structure – with the program's built-in functions all being PR – then the newly defined function will also be primitive recursive. This gives us a way of convincing ourselves that a new function is PR, i.e., sketch out a routine for computing it and check that it can all be done with a succession of (possibly nested) *for* loops which only invoke already known PR functions: then the new function will be primitive recursive.

11.2 Predicates

Consider the expression $x + y = 7$. As it stands, this is clearly not a *statement.* Because we are not in a position to say it true or false. However, if we replace x and y by a numbers, it becomes a statement, and True/False is obtained. Thus, $3 + 4 = 7$ is True, but $3 + 5 = 7$ is False.

A statement like above, which behaves as *True* or *False* is called *predicate.* We usually employ uppercase letters of Roman alphabets, like $\{P, Q, R, S\}$ to designate predicates. The lowercase letters that appear in a predicate are called its *arguments.* Thus, a predicate P may have $x_1, x_2, ..., x_n$ arguments. If these are replaced by $(a_1, a_2, ..., a_n)$, the predicate becomes $P(a_1, a_2, ..., a_n)$.

Let $P(x_1, x_2, ..., x_n)$ be an n-ary predicate. Then we write the *extension* of P as,

$$\{x_1, x_2, ..., x_n \mid P(x_1, x_2, ..., x_n)\}. \tag{11.3}$$

The above extension means set of all the n-tuples $(x_1, ..., x_n)$ for which $P(x_1, x_2, ..., x_n)$ is true. Hence,

$$(a_1, ..., a_n) \in \{x_1, ..., x_n \mid P(x_1, ..., x_n\} \tag{11.4}$$

if and only if $P(a_1, ..., a_n)$ is True. Thus, if we assume $P = \{x, y \mid x + y = 5\}$ then we have $(2, 3) \in P, (4, 1) \in P, (1, 4) \in P, (6, 2) \notin P, (5, 0) \in P$.

Any two predicates are said to be equal if they have the same extension. We write,

$$P(x_1, ..., x_n) \leftrightarrow Q(x_1, ..., x_n)$$

to indicate that P and Q are equivalent. Thus,

$$P(x_1, ..., x_n) \leftrightarrow Q(x_1, ..., x_n)$$

if and only if,

$$\{x_1, ..., x_n \mid P(x_1, ..., x_n)\} = \{x_1, ..., x_n \mid Q(x_1, ..., x_n)\}.$$

For example, $x + y = 5 \leftrightarrow x + y + 1 = 6$, are the predicates which are equal.

Note that predicates are in no way very different from functions. A function $y = f(x_1, ..., x_n)$ will provide the result y, depending on the values of $x_1, ..., x_n$, where x_i, and y may belong to any domain, not necessarily be in same domain, e.g., y may be real and x_i's may be natural numbers. The expression for predicate $P(x_1, ..., x_n)$ will take only the values of True and False. Hence, if $(a_1, ..., a_n) \in \{x_1, ..., x_n \mid P(x_1, ..., x_n)\}$, then $P(a_1, ..., a_n)$ is True, else False.

Definition 11.1 *An n-ry predicate P over $\mathbb{N}^n$ is any subset of $\mathbb{N}^n$. We write a tuple $(x_1, ..., x_n)$ satisfies P as $(x_1, ..., x_n) \in P$ or as $P(x_1, ..., x_n)$. The characteristic function of a predicate P is the function $\chi_P : \mathbb{N}^n \to \{0, 1\}$ defined by,*

$$\chi_P(x_1, ..., x_n) = \begin{cases} 1, & \textit{iff } P(x_1, ..., x_n) \\ 0, & \textit{not } P(x_1, ..., x_n). \end{cases} \tag{11.5}$$

A predicate P is primitive recursive iff its characteristic function χ_P is primitive recursive.□

It can be easily shown that if P and Q are PR predicates over $\mathbb{N}^n$, then $P \vee Q$, $P \wedge Q$, $\neg P$, and $\neg Q$ are also PR.