Theory of Formal Languages (Partially computable, and Ackermann's Functions)

Lecture 12: April 03, 2021

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

12.1 Partially Computable Functions

A *partially computable* function may be thought of as one for which we possess an algorithm that enables us to compute its value for elements of its domain, but will require us to compute forever in attempting to obtain a functional value for an element not in its domain, without ever assuring that no value is forthcoming. That is, the algorithm may spend infinite amount of time in vain to search for an answer. For example, $f(x) = e^x$ is effectively computable function, while $f(x) = tan(x)$ is partially computable, as $tan(\pi/2) = \infty$.

Definition 12.1 Partial recursive functions. *For* $\Sigma = \{a_1, a_2, ..., a_N\}$*, the class of* partial recursive functions *is the smallest class of functions (over* Σ^**) which contains the base functions and are closed under composition, primitive recursion, and minimization. The class of recursive functions is the subset of the class of partial recursive functions consisting of functions defined for every input.*□

A *partial recursive function* is one that can be computed by a *Turing Machine*. A *total recursive* function is partial recursive function, which is defined for every input. Every primitive function is total recursive, but not all total recursive functions are primitive recursive. An *Ackermann's function* $A(m, n)$ is well known example of a total recursive function that is not primitive recursive. The Fig. 12.1 shows the relation between these functions.

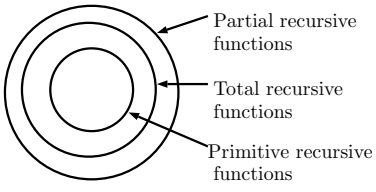

Figure 12.1: Functions hierarchy

12.2 Recursive Functions

With our discussions so far about PR functions we are convinced that every day mathematical functions are PR. In the following we shall discuss two examples of functions which are not primitive, but still effectively computation. These functions will motivate for introducing some important theoretical ideas that will be used at many places in our discussions later on.

The first example is the function called Ackermann function, after its inventor. It is defined using a scheme that does not fit in the scheme of primitive recursion; the scheme is called *double-recursion*, such functions grow very fast – faster than PR functions, and do not use the basic functions like Z, S, P and composition, as used in the PR functions. However, they follow a perfect algorithm for computing the values for given arguments.

12.2.1 PR functions are Countable

We have seen that for every PR function there exists an expression in terms of basic functions, composition, and primitive recursion. This has an important consequence: the PR functions are countable. Taking that advantage, we shall construct certain enumeration of PR functions, like, f_0, f_1, ..., such that every PR function occurs some where in this sequence, may be more than once!

Why the PR functions are countable? Since there are countable many basic functions (Z, S, and for each $i < n$, a projection function). Now we look at the following sequence of sets:

$$\begin{aligned} PR_0 &= \text{class of basic functions} \\ PR_1 &= PR_0 \cup \{h \mid h = comp[f, g_1, ..., g_n]\} \text{ for some } f, g_i \in PR_0 \\ PR_2 &= PR_1 \cup \{h \mid h = P_r[f, g]\} \text{ for some } f, g \in PR_1 \\ .... &= ... \end{aligned}$$

Now, each of the classes of PR_i is countable. Since,

$$PR = \bigcup_{i \in \mathbb{N}} PR_i \tag{12.1}$$

the class of PR is a countable union of countable sets, hence countable.

Now the idea is to assign a natural number to each definition of a PR function in the following way:

- the code of Z function is [0]
- the code or successor function is [1]
- the code for projection function is P_i^n is $[2, n, i]$

This gives us the code for all the basic functions.The idea is that the first number gives the information that we are dealing with the basic function (level 0). Note that, if we are

given a natural number x, then we can effectively determine whether x is a code of a basic function and if so, of which.

- Let codes $p, q_1, ..., q_k$ of PR fictions $f, g_1, ..., g_k$, respectively be given. Let $g_1, ..., g_k$ all have arity n, and that f has arity k. Then the composite $comp[f, g_1, ..., g_k]$ gets code $[3, p, q_1, ..., q_k]$.

- Given the codes p, q of PR functions f, g where $f : \mathbb{N}^k \to \mathbb{N}$, $g : \mathbb{N}^{k+2} \to \mathbb{N}$, $P_r[g, f]$ gets the code $[4, p, q]$.

At this point we should convince ourselves that: this representation is well defined, and given a number c, one can effectively test whether that number is a code of some thing. We may define the following sequence of unary function, $f_0, f_1, ...$, such that,

$$f_n = \begin{cases} \text{function with code } n, & \text{if } n \text{ codes a unary function} \\ \text{function } f(x) = 0, & \text{otherwise.} \end{cases} \tag{12.2}$$

Every unary PR function is of the form f_i for some i; conversely, if we are given a number n then we can effectively unravel it as to obtain a derivation of the function f_n. Thus, we have obtained the effective enumeration of the unary PR functions.

12.2.2 Diagonalization

Recall that in the PR functions, we have applied the schemes of generalized composition and primitive recursion, to the total functions. Now we are considering the non-total functions. Hence the name *partial recursive functions*.

Definition 12.2 Recursive functions/Partial recursive function. *A function is called* recursive function *or* partial recursive function *or* general recursive function *if it can be obtained from the basic functions using finitely many applications of composition, primitive recursion, and unbounded minimization.*□

Standard computer languages of course have programming structures which implement just this kind of unbounded search. Because just like "for loops," they allow "do until" loops (or equivalently, "do while" loops). In other words, they allow some process to be iterated until a given condition is satisfied – where no prior limit is put on the the number of iterations to be executed.

If we think of what are the unbounded searches, which leads to computations, then it looks very plausible that not everything computable will be primitive recursive. True, that is as yet only a plausibility consideration. Our remarks so far leave open the possibility that computations can always somehow be turned into procedures using "for loops" with a bounded limit on the number of steps. But in fact we can now show that is not always the case.

Theorem 12.3 *There are effectively computable numerical functions that are not primitive recursive.*

Proof: The set of PR functions is effectively enumerable. That is to say, there is an effective way of numbering of functions f_0, f_1, f_2, ..., such that each of the f_i is PR, and each PR function appears somewhere on the list [petsmth10].

This holds because, by definition, every PR function has a 'recipe' in which it is defined by recursion or composition from other functions which are defined by recursion or composition from other functions which are defined ... ultimately in terms of primitive starter functions: Z, S, P. So we choose some standard formal specification language for representing these recipes. Then we can effectively generate "in alphabetical order" all possible strings of symbols from this language; and as we go along, we select the strings that obey the rules for being a recipe for a PR function. That generates a list of recipes which effectively enumerates the PR functions, repetitions allowed, as shown in Table 12.1.

Table 12.1: All possible strings of symbols PR functions language

	0	1	2	3	...
f_0	$\underline{f_0(0)}$	$f_0(1)$	$f_0(2)$	$f_0(3)$	...
f_1	$f_1(0)$	$\underline{f_1(1)}$	$f_1(2)$	$f_1(3)$	...
f_2	$f_2(0)$	$f_2(1)$	$\underline{f_2(2)}$	$f_2(3)$	...
f_3	$f_3(0)$	$f_3(1)$	$f_3(2)$	$\underline{f_3(3)}$	...
...	...	...	...	...	$\searrow$

Now consider this table. Down the table we list off the PR functions f_0, f_1, f_2, ... an individual row then gives the values of f_n for each argument. Let us define the corresponding diagonal function δ, by putting $\delta(n) = f_n(n) + 1$, so that,

$$\begin{aligned}\delta(0) &= f_0(0) + 1\\ \delta(1) &= f_1(1) + 1\\ &\ldots\\ \delta(n) &= f_n(n) + 1.\end{aligned}$$

To compute $\delta(n)$, we just run our effective enumeration of the recipes for PR functions until we get to the recipe for f_n. We follow the instructions in that recipe to evaluate that function for the argument n. We then add one. Each step is entirely mechanical. So our diagonal function is effectively computable, using a step-by-step algorithmic procedure.

By construction, however, the function δ cannot be primitive recursive. For suppose otherwise. Then δ must appear somewhere in the enumeration of PR functions, i.e., be the function f_d for some index number d. But now ask what the value of $\delta(d)$ is? By hypothesis, the function δ is none other than the function f_d , so $\delta(d) = f_d(d)$. But by the initial definition of the diagonal function, $\delta(d) = f_d(d) + 1$, which is a contradiction.

So we have 'diagonalized out' of the class of PR functions to get a new function δ which is effectively computable but not primitive recursive. □ ■

But hold on! Why is the diagonal function not a PR function? Where are the open-ended searches involved in computing it?' Well, consider evaluating $d(n)$ for increasing values of n. For each new argument, we will have to look along the sequence of strings of symbols in our computing language until we find the next one that gives us a well-constructed recipe for a PR function. That is not given to us a bounded search.

12.3 Ackermann Functions

Our second example of functions that are not PR bu which are still computable are Ackermann functions. These Ackermann's function are based on recursive schme which is wider than primitive recursion.

The PR function is based on the idea of defining a function in terms of its previous values. The actual recursion took place on one recursion, where as the Ackermann function is a double recursion.

Ackermann functions are examples of nonprimitive recursive functions which grow so fast with respect to their arguments, that it becomes almost impossible to imagine the order of the magnitudes involved. It is easy to give the equations for these functions, but writing the solutions is by no means straightforward. Of the many versions of Ackermann functions, the one given below is perhaps the simplest. The functions can be defined by the equations.

Definition 12.4 Ackermann. *Define a function* $A : \mathbb{N}^2 \to \mathbb{N}$ *by double recursion:*

$$\begin{aligned} A(0,n) &= n+1, \quad n \geq 0; \\ A(m,0) &= A(m-1,1), \quad m > 0; \\ A(m,n) &= A(m-1, A(m,n-1)), \quad m,n > 0. \end{aligned} \tag{12.3}$$

It is easy to write the first few functions explicitly, but only the first few.

Example 12.5 *Evaluate:* $A(1,2)$.

$$\begin{aligned} A(1,2) &= A(0, A(1,1)) \\ &= A(0, A(0, A(1,0))) \\ &= A(0, A(0, A(0,1))) \\ &= A(0, A(0,2)) \\ &= A(0,3) \\ &= 4 \end{aligned}$$

Similarly, the $A(2,2) = 7$. □

The value of Ackermann's function exhibit a remarkable rate of growth. By fixing the first variable, Ackermann's function generates the one-variable functions,

$$\begin{aligned} A(1,n) &= n+2 \\ A(2,n) &= 2n+3 \\ A(3,n) &= 2^{n+3} - 3 \\ A(4,n) &= \underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}\ -3 \quad \text{the power of 2 is repeated for n+3 times} \end{aligned}$$

Table 12.2: Growth of Ackerman's function

$A(m,n)$	$n=0$	$n=1$	$n=2$	$n=3$
$m=0$	1	2	3	4
$m=1$	2	3	4	5
$m=2$	3	5	7	9
$m=3$	5	13	29	61
$m=4$	13	65533	$2^{65533}-3$	$A(3,2^{65533}-3)$
	$=2^{2^2}-3$	$=2^{2^{2^2}}-3$	$=2^{2^{2^{2^2}}}-3$	$=2^{2^{2^{2^{2^2}}}}-3$
$m=5$	65533	$A(4,65533)$	$A(4,A(5,1)$	$A(4,A(5,2)$

For example, $A(4,0)=16-3$, $A(4,1)=2^{16}-3$, and $A(4,2)$ is $2^{2^{16}}-3$ The first variable of Ackermann's function determines the rate of growth of the function vales.

The Table 12.2 shows more values of Ackerman's function.

Following are the properties of Ackerman's function:

- It is a well defined total function.
- Computable but not primitive recursive.
- Grows faster than any primitive recursive function.
- It is μ-recursive.

Following are the applications of Ackerman's functions:

- In Computational complexity of some algorithms
- In theory of recursive functions
- As a benchmark of a compiler's ability to optimize recursion
- In specifying huge dimensions in certain theories such as Ramsey Theory