Theory of Formal Languages (Sequential Operation of Turing Machines)

Lecture 9: Jan. 19, 2019

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

9.1 Sequential Operation of Turing Machines

A Turing machine is designed to accomplish a single task, and it corresponds to an algorithm. Such Turing machines can be combined to operate in sequential order in sufficient numbers to perform complex operations. In such cases, the output produced by a Turing machine on the tape becomes input for next Turing machine. For example, two Turing machines Z and S can be operated sequentially, one after other, so that whatever is the current tape contents, it will be made zero by Z Turing machine, followed this it is incremented by 1 by the S Turing machine. In this situation, the final state of the first machine (Z) becomes the start state of S, as shown in Fig. 9.1.

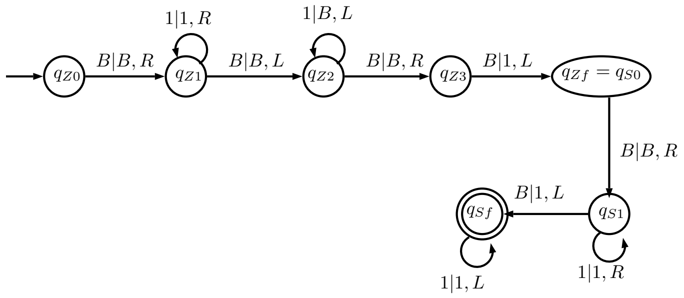

Figure 9.1: Sequential operation of TMs Z and S

The TMs Z and S can be used as components of more complex Turing machines. The component machines are called Macros – the concept used in assembly and C languages. Following are some of the commonly used macros.

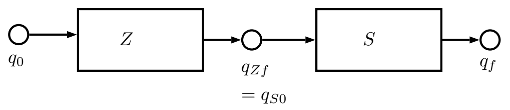

Figure 9.2: Symbolic representation for operation of TMs Z and S in sequential order

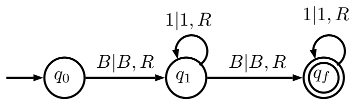

Figure 9.3: TM for MR_2

MR_i (Move Right) Macro This macro moves the tape head right through i consecutive natural numbers on the tape[1]. For example, MR_2 macro is defined by the Turing machines shown in Fig. 9.3.

A move macro does not effect the tape contents to the left of initial position of tape head. The initial and final configuration for MR_2 are as follows:

Initial configuration: $B\bar{n}_1Bq_0\bar{n}_2B\bar{n}_3B\bar{n}_4B$

Final configuration: $B\bar{n}_1B\bar{n}_2B\bar{n}_3Bq_f\bar{n}_4B$

For a macro, it is necessary that proper input is provided, e.g., for MR_2 it is necessary that at least two natural numbers exists after the tape head. A macro never effects the contents outside the bounds of that macro. Henceforth, we will indicate tape head position by underscore, (e.g., $\underline{B}$ or $\underline{n_i}$).

ML_i (Move left) Macro The ML_i macro is defined in the similar way as MR_i, except that it moves the tape head to left. The initial and final configurations of ML_2 macro are as follows.

Initial configuration: $B\bar{n}_1B\bar{n}_2B\bar{n}_3B\bar{n}_4\underline{B}$

Final configuration: $B\bar{n}_1B\bar{n}_2\underline{B}\bar{n}_3B\bar{n}_4B$

FR and FL (Find right and find left) Macros The find right macro moves the tape-head into a position to process the first natural number in right position. The Following is an example of find right macro.

Initial configuration: $\underline{B}B^i\bar{n}B \quad i \geq 0$

Final configuration: $B^i\underline{B}\bar{n}B$

[1]A natural natural number on tape is sequence of 1s separated from both sides by Bs.

Note that in the initial configuration, there are B^i blank symbols between head position and the right natural number, and after head moves right to the first natural number, it points to the blank position just before the natural number. Similarly, find left macro moves the tape-head to find the first left natural number from head position.

Initial configuration: $B\bar{n}B^i\underline{B} \quad i \geq 0$

Final configuration: $\underline{B}\bar{n}B^iB$

E_k **(Erase) Macro** The erase macro erases a sequence of k natural numbers, and halts with tape head in original position.

Initial configuration: $\underline{B}\bar{n}_1B\bar{n}_2B...B\bar{n}_kB \quad k \geq 1$

Final configuration: $\underline{B}B..................B$

$CPY_{k,i}$**(Copy through i numbers) Macro** The copy macro produces a copy of specified number of integers, in the area of tape that is assumed to be blank. The macro $CPY_{k,i}$ expects total $k+i$ integers on the tape, and when it is executed, it copies first k numbers of integers following to original $k+i$ integers, making total $2k+i$ integers when operation is performed. For example, $CPY_{3,2}$ copies first three numbers at the end, making total 8 numbers, and $CPY_{3,0}$ copies first three numbers at the end, making total six numbers.

Initial: $\underline{B}\bar{n}_1B\bar{n}_2B...B\bar{n}_kB\bar{n}_{k+1}B...B\bar{n}_{k+i}BB...B \quad k \geq 1$

Final: $\underline{B}\bar{n}_1B\bar{n}_2B...B\bar{n}_kB\bar{n}_{k+1}B...B\bar{n}_{k+i}B\bar{n}_1B\bar{n}_2B...B\bar{n}_kB$

T **(Translate) macro** The translate macro changes the location of the first natural number so that it is to the right of tape head position. The computation is terminated with the tape head in the position it occupied in the beginning, with translated string (natural number) to its immediate right.

Initial Configuration: $\underline{B}B^i\bar{n}B \quad i \geq 0$

Final Configuration: $\underline{B}\bar{n}B^iB$

A (Add) Macro The Add macro adds the two consecutive naturals after the tape head, and replaces these by a single number equal to sum of these two. The initial and final configurations of the Turing machines are:

Initial Configuration: $\underline{B}\bar{m}B\bar{n}B$

Final Configuration: $\underline{B}\overline{m+n}B$

Example 9.1 *Construct a sequence of macros to exchange two integer numbers m and n available on the tape of TM.*

The initial and final configurations are as given below.

Initial configuration: $\underline{B}\bar{m}B\bar{n}B$

Final configuration: $\underline{B}\overline{m+n}B$

The Fig. 9.4 show the six macros sequentially performed to swap the two numbers m and n.

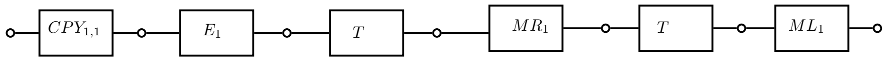

Figure 9.4: Sequential operations of Turing machines to swap two numbers

The change in configuration after each macro is performed, to swap two numbers, are shown below.

Initial configuration: $\underline{B}\bar{m}B\bar{n}B$

Configuration after $CPY_{1,1}$: $\underline{B}\bar{m}B\bar{n}B\bar{m}B$

Configuration after E_1: $\underline{B}B^mB\bar{n}B\bar{m}B$

Configuration after T: $\underline{B}\bar{n}B^mBBB\bar{m}B$

Configuration after MR_1: $B\bar{n}\underline{B}B^mB\bar{m}B$

Configuration after T: $B\bar{n}\underline{B}\bar{m}B^mBB$

Configuration after ML_1: $\underline{B}\bar{n}B\bar{m}B^m BBB$

Example 9.2 *Construct sequential TMs to multiply a natural number n by 2.*

Initial configuration: $\underline{B}\bar{n}B$

Configuration after $CPY_{1,0}$: $\underline{B}\bar{n}B\bar{n}B$

Configuration after (Add) A: $\underline{B}\overline{n+n}B$ (This is final configuration)

The Fig. 9.5 shows the construction of TM to add two numbers.

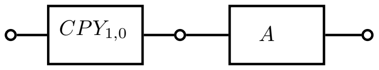

Figure 9.5: Sequential operations of Turing machines to multiply a number by 2

9.2 Computable but not PR functions

We have seen that any PR function is mechanically computable. But, in fact not all effectively computable numerical functions are primitive recursive. At first we make some plausibility considerations. We have just seen that the values of a given primitive recursive function can be computed by a program involving 'for' loops as its main programming structure. Each loop goes through a specified number of iterations. However, we do allow computations to involve open-ended searches, with no prior bound on the length of search. This permission is used when we prove that 'negation-complete theories are decidable'[2] – the process is allowed to 'enumerate the theorems and wait to see which of ϕ or $\neg\phi$ turns up' to count as a computational decision procedure [petsmth10].

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

[2]Negation-complete theorem:

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 9.2.

Table 9.1: 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	$\underline{f_1(0)}$	$\underline{f_1(1)}$	$f_1(2)$	$f_1(3)$	...
f_2	$f_2(0)$	$\overline{f_2(1)}$	$\underline{f_2(2)}$	$f_2(3)$	...
f_3	$f_3(0)$	$f_3(1)$	$\overline{f_3(2)}$	$\underline{f_3(3)}$	...
...	...	...	...	...	$\searrow$

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, by putting $\delta(n) = f_n(n) + 1$, so that,

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

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, $\delta(d) = f_d(d) + 1$. 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.

9.3 Ackerman Functions

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:

$$\begin{aligned} A(0,n) &= n+1, & n &\geq 0; \\ A(m,0) &= A(m-1,1), & m &> 0; \\ A(m,n) &= A(m-1,A(m,n-1)), & m,n &> 0. \end{aligned} \tag{9.1}$$

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

Example 9.4 *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,

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

The number of 2's in the exponential chain in $A(4,n)$ is y. 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 9.2 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.

Table 9.2: 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	$A(4,65533)$	$A(4,A(5,1)$	$A(4,A(5,2)$

- Grows faster than any primitive recursive function.
- It is μ-recursive.

Following are the applications of Ackerman's function:

- 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