Theory of Formal Languages (Turing Computable Functions)

Lecture 7: Mar. 09, 2021

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

7.1 Introduction

A classification of all the computable functions is given in terms of *sub-recursive* programming languages. These classes are those which also arise from the relation "primitive recursive." By distinguishing between honest and dishonest classes, the classification is related to the computational complexity of the functions classified, and the classification has a wide degree of measure invariance.

In modern mathematics, the prevailing notion of a function is that of "functions as graph": each function f has a fixed *domain* X and *codomain* Y, and a function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$ (see Fig. 7.1(b)). A function between one set to another is one-way relation, where as a *relation* can be two-way also, like for a relation R, another relation R^{-1} may also exist. Thus, set of functions is a subset of set of relations (see Fig. 7.1(a)).

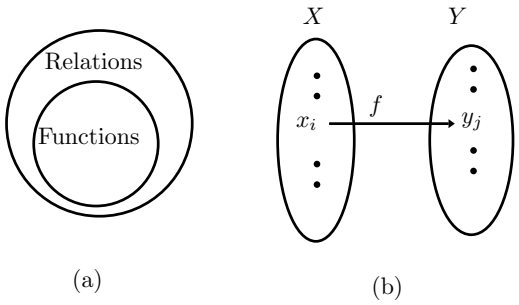

Figure 7.1: (a) Relations vs. Function, (b) Function as a mapping from one set to another

Two functions $f, g : X \to Y$ are considered equal if they yield the same output on each input, i.e., $f(x) = g(x)$ for all $x \in X$. This is called the *extensional* view of functions, because it specifies that the only thing observable about a function is how it maps inputs to outputs, in other words which domain values map to what co-domain values.

Before the 20th century, functions were rarely looked at in this way. An older notion of functions is that of "functions as rules" (formula). In this view, to give a function means to give a rule for how the function is to be calculated. Often, such a rule can be given by a formula, for instance, the familiar $f(x) = x^2$ or other function $g(x) = sin(e^x)$ from

calculus. As before, two functions are *extensionally* equal if they have the same input-output behavior.

Now, we can also speak of another notion of equality: two functions are *intensionally* equal if they are given by (essentially) the same formula or computation steps. When we think of functions as given by formulas, it is not always necessary to know the domain and co-domain (range) of a function. Consider for instance the function $f(x) = x$. This is, of course, the *identity function*. We may regard it as a function $f : X \to X$ for any Set X.

However, in most of cases, the "functions as graphs" paradigm is the most elegant and appropriate way of dealing with functions. The graphs define a more general class of functions, because it includes functions that are not necessarily given by a rule, or, the rule may be very complex.

In computer science, the "functions as rules" paradigm is often more appropriate. Think of a computer program as defining a function that maps input to output. Most computer programmers (and users) do not only care about the extensional behavior of a program (which inputs are mapped to which outputs?), but also about how the output is calculated? How much time does it take? How much memory and disk space is used in the process? How much communication bandwidth is used? These are *intensional* questions having to do with the particular way in which a function was defined.

The mathematical approach to defining what is meant by computable function consist of starting with a few elementary functions, and then build new functions by allowing certain constructions on functions. The end result is a class of functions on the natural numbers (with several arguments), and the claim is that this class captures exactly those functions which are effectively computable.

In this chapter we introduce a smaller class of functions, called *primitive recursive* (PR) functions. This class already contains quite a few functions that are used in everyday mathematics.

Next, we introduce two important ideas, namely arithmetization and diagonalization. These will allow us to construct examples of functions which are not primitive recursive. These examples indicate that the class of primitive recursive functions is till "not large enough": the examples are functions that are effectively computable, but not PR. Thus we allow one further construction, called *minimalization*, which enlarges our class of functions, and the result is called the class of *partial recursive* functions.

7.2 Computable and Partially Computable Functions

We know that Turing machines can be used to perform symbolic computations. In order to have Turing machines perform numeric computations, it is necessary that we introduce suitable symbolic representations for numbers. The simplest way of doing this is to choose one symbol as a basic. Suppose we choose S_1, and symbolize a number by an expression consisting entirely of occurrences of this symbol. We may write S_1 for "1" by . If n is a positive integer, we represent S_1^n for the expression $|S_1S_1...S_1| = n$. For completeness, we write S_1^0 to be the null expression. Then we may conclude the following convention. With each number n we associate the tape expression $n = 1^{n+1}$. Hence, $3 = 1111$, and in general

$n = 1111...1, (n+1)$ times. With each k-tuple $(n_1, n_2, ..., n_k)$, of integers, we associate the tape expression $(\overline{n_1, n_2, ..., n_k})$ such that,

$$(\overline{n_1, n_2, ..., n_k}) = \overline{n_1} B \overline{n_2} B ... B \overline{n_k}. \tag{7.1}$$

Thus, $(\overline{3, 2, 0}) = 1111B111B1$. This notation is convenient in connection with initial data or inputs.

Let M be any expression and $\langle M \rangle$ be the number of 1's in M. Then, $\langle 111BS_4q_3 \rangle = 2$, and $\langle PQ \rangle = \langle P \rangle + \langle Q \rangle$.

Let T be a Turing machine. Then for each integer n, we associate with T an n-ary function $\Psi^{(n)}(x_1, x_2, ..., x_n)$ as follows: for each n-tuple $(m_1, m_2, ..., m_n)$, we set a configuration[1] $\alpha_1 = q_1(m_1, m_2, ..., m_n)$, we distinguish between the two cases:

1. There exists a computation $\alpha_1, ..., \alpha_p$, of Turing machine T, where,

$$\Psi_T^{(n)}(m_1, ..., m_n) = \langle \alpha_p \rangle = \langle Res_T(\alpha_1) \rangle,$$

2. There exists no computation $\alpha_1, ..., \alpha_P$, where

$$\Psi_T^{(n)}(m_1, ..., m_n),$$

is undefined. We will prefer to use $\Psi_T(x)$ for $\Psi_T^{(1)}(x)$.

It is concept of computable function that we propose to identify with the intuitive concept of *effectively calculable* 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 7.1 *Let $\Sigma = \{a_1, a_2, ..., a_N\}$. 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.*□

We shall now comment briefly on the *adequacy* of our identification of *effective calculability* with computability. Historically, proposals were made by a number of scientists at about the same time (1936), mostly, independently of one another, to identify the concept of effectively calculable function, with various precise concepts. In this connection, we may mention, Church's notion of *lambda-definability*, Gödel-Herbrand, and Kleene's notion of

[1]The α_1 is computation result when Turing machine T, originally is in state q_1 and tape comprises a n-tuple argument $(m_1, ..., m_n)$.

general recursiveness, and Turing's notion of Computability. Now, it has been well established that these different concepts turned out to be equivalent.

Next we may note that every computable function must surely be regarded as effectively calculable. For $f(m)$ be computable, let T be *Turing machine* which computes $f(m)$. Then, if we were given a number m_0, we may begin with the *instantaneous description*, α_1,

$$\alpha_1 = q_0\overline{m_0} = q_0\underbrace{11..1}_{m_0+1},$$

and successively obtain *instantaneous descriptions*, $\alpha_1, \alpha_2, ..., \alpha_p$ where

$$\alpha_1 \Rightarrow \alpha_2 \Rightarrow \alpha_3 \Rightarrow ... \Rightarrow \alpha_p,$$

and where α_p is terminal. Since $f(m)$ is computable, such a terminal α_p must be obtainable in a finite number of steps. Here we have, $f(m_0) \Rightarrow^* \langle\alpha_p\rangle$ and $\langle\alpha_p\rangle$ is simply the number of 1's in α_p. This procedure would ordinarily be regarded as "effective", is clear from the fact that for a given instantaneous description α of TM T, whether or not there exists an instantaneous description β such that $\alpha \Rightarrow \beta$. If the answer is Yes, it is to be determined which β satisfies this condition. It suffices to write α in the form Pq_iS_jQ and to locate the quadruple of TM T, if at all one exists, which begins with q_iS_i. Here P and Q are expressions, q_i is state, and S_j is symbol. The examples of quadruples may be:

$$q_iS_jS_kq_l$$
$$q_iS_jRq_l$$
$$q_iS_jLq_l$$

where L, R are left and right movements of Turing machine, q_i is initial state, S_j and S_k are original and final symbols. Where there is no left or right symbols, it indicates that machine does not take any movement, in that case it modifies the original symbol S_j by S_k while keeping the head stationary.

Following is an example of $\alpha_1 \Rightarrow \alpha_2 \Rightarrow ... \Rightarrow \alpha_p$ for a TM T.

Example 7.2 Addition. *Let $f(x, y) = X + Y$. We construct a Turing machine T which computes $f(x, y)$ that is,*

$$\Psi_T^{(2)}(x, y) = x + y.$$

We take T to consist of quintuples (5-tuples), $q_iS_iS_jdq_j$, where $d \in \{L, R, S\}$ (see Fig. 7.2). Note that, we used a Turing machines instead of Post Machine[2]

[2]For the sake of simplicity – in terms of number of transitions, we have made use of TM instead of Post machine. Ordinarily, though the post machine make use of quadruple instead of 5-tuple, the number of transitions will be too many. This is analogous to a situation in instruction set of modern computer, where an algorithm based on 3-operand instructions will far smaller than that of one operand instructions.

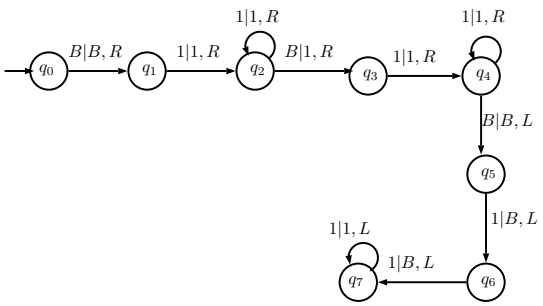

Figure 7.2: Function to add two integers

$$
\begin{aligned}
q_0B111B11B &\vdash Bq_1111B11B \vdash B1q_211B11B \vdash B11q_21B11B \\
&\vdash B111q_2B11B \vdash B1111q_311B \vdash B11111q_41B \vdash B111111q_4B \\
&\vdash B11111q_51B \vdash B1111q_61BB \vdash B111q_71BBB \vdash B11q_711BBB \\
&\vdash B1q_7111BBB \vdash Bq_71111BBB \vdash q_7B1111BBB
\end{aligned}
$$

Note that we have explicitly not indicated the L or R move, but it is apparent, as symbol next to q_i is where the tape-head points.