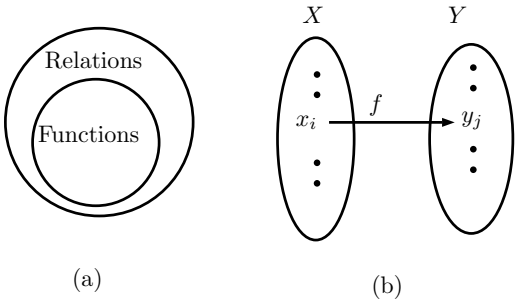
Theory of Formal Languages (Turing Computable Functions)

Lecture 7: Jan. 12, 2019

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

7.1 Introduction

In modern mathematics, the prevailing notion of a function is that of "functions as graphs": 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 Figure 7.1).



.

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.

However, 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 $g(x) = sin(e^x)$ from calculus. As before, two functions are *extensionally* equal if they have the same input-output behavior.

But now we can also speak of another notion of equality: two functions are *intensionally* equal if they are given by (essentially) the same formula.

When we think of functions as given by formulas, it is not always necessary to know the domain and codomain (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.

In most of cases, the "functions as graphs" paradigm is the most elegant and appropriate way of dealing with functions. Graphs define a more general class of functions, because it includes functions that are not necessarily given by a rule.

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.

7.2 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, a definite statement - True or 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 predicate. 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,

$$\{x_1, x_2, ..., x_n \mid P(x_1, x_2, ..., x_n)\}. \tag{7.1}$$

The above extension means set of all the n-tuples for which $P(a_1, a_2, ..., a_n)$ is true. Hence,

$$(a_1, ..., a_n) \in \{x_1, ..., x_n \mid P(x_1, ..., x_n\} \tag{7.2}$$

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

7.3 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 "1" by S_1. If n is a positive integer, we write S_i^n for the expression $|S_iS_i...S_i| = n$. For completeness, we write S_i^0 to be the null expression. Then we may conclude the following definition:

With each number n we associate the tape expression $n = 1^{n+1}$. Hence, as per the definition above $3 = 1111$, 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 $(\overline{n_1, n_2, ..., n_k})$, where

$$(\overline{n_1, n_2, ..., n_k}) = \overline{n_1}B\overline{n_2}B...B\overline{n_k}. \tag{7.3}$$

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, $\langle 11BS_4q_3 \rangle = 2$, and $\langle PQ \rangle = \langle P \rangle + \langle Q \rangle$.

Let T be a Turing machine. Then for each 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 $\alpha_1 = q_1(m_1, m_2, ..., m_n)$, and we distinguish between the two cases:

1. There exists a computation of T, $\alpha_1, ..., \alpha_p$, 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 $\Psi_T(n)$ for $\Psi_T^{(1)}(x)$.

Definition 7.1 *An n-ary function*$f(x_1, ..., x_n)$ *is* partially computable *if there exists a Turing machine* T *such that*

$$f(x_1, ..., x_n) = \Psi_T(x_1, ..., x_n)$$

In this way T *computes* f*. If, in addition,* $f(x_1, ..., x_n)$ *is* total *function, then it is called* computable*.*

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 which enables us to compute its value for elements of its domain. But which will require us to compute for ever in attempting to obtain a functional value for an element not in its domain, without ever assuring us 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 $tan(\pi/2) = \infty$.

We shall now comment briefly on the *adequacy* of our identification of *effective calculability* with computability. Historically, proposals were made by a number of different persons 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 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.

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 quadruples

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

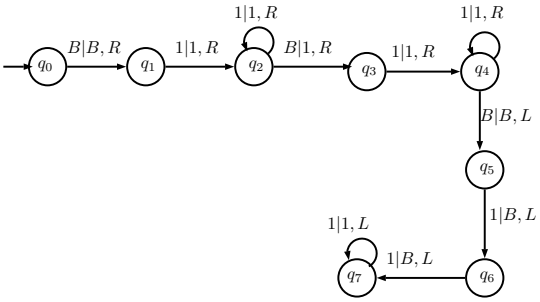


Figure 7.2: Function to add two integers

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.