The Parameter Theorem

Let us say that you have a partially computable function f that takes \( m + n \) inputs:

\[ f(x_1, \ldots, x_m, u_1, \ldots, u_n) \]

This function is computed by a program with number \( y \). Now it turns out that you always want to use the same last \( n \) inputs \((u_1, \ldots, u_n)\) whenever you call \( f \). So you would like to use a function \( g \) taking only the first \( m \) inputs and using fixed values for the last \( n \) ones:

\[ g(x_1, \ldots, x_m) \]

This function is supposed to give the same outputs as \( f \) for any input \((x_1, \ldots, x_m)\) if \((u_1, \ldots, u_n)\) never changes.

Now how can I obtain this function \( g \)? It is very easy! Since there is a program that computes \( f(x_1, \ldots, x_m, u_1, \ldots, u_n) \), I can reuse the code for \( f \) and just prepend the following instructions to it:

\[
\begin{align*}
X_{m+1} & \leftarrow u_1 \\
X_{m+2} & \leftarrow u_2 \\
\vdots & \\
X_{m+n} & \leftarrow u_n
\end{align*}
\]

Obviously, this new program computes \( g(x_1, \ldots, x_m) \) with fixed values for \((u_1, \ldots, u_n)\).

The Parameter Theorem

We said that \( f \) is computed by program number \( y \). But what is the number \( z \) of the program that computes \( g \)? Clearly, the code for \( g \) depends on the fixed values we want to assign to \((u_1, \ldots, u_n)\), and it also depends on the original program with the number \( y \) and the number \( n \) of inputs that \( g \) will still take, and the number \( m \) of inputs that will be kept constant.

Computing \( z \) thus depends on all of these variables and is achieved by the function \( S \):

\[ S_m^n(u_1, \ldots, u_n, y) \]

The parameter theorem states that this function always exists and is primitive recursive.

The Recursion Theorem

Let us say that we have a partially computable function \( g \) that takes \( (m + 1) \) inputs. Then the recursion theorem tells us that we can always find at least one number \( e \), such that the program with number \( e \) takes the \( m \) last inputs of \( g \) and then computes the same output that \( g \) would produce for the same inputs and the first input being \( e \):

\[ \Phi_e(m)(x_1, \ldots, x_m) = g(e, x_1, \ldots, x_m). \]

Such programs generate copies of themselves. Thus the parameter theorem is the theoretical foundation of the existence of computer viruses.

Rice’s Theorem

Let \( \Gamma \) be some collection of partially computable functions of one variable. We associate with \( \Gamma \) the following index set \( R_{\Gamma} \):

\[ R_{\Gamma} = \{ t \in \mathbb{N} \mid \Phi_t \in \Gamma \}. \]

\( R_{\Gamma} \) will be a recursive set if there is an algorithm which accepts as input the number \( t \) of a program and returns the value TRUE or FALSE depending on whether or not the function computed by this program belongs to \( \Gamma \).
Rice’s Theorem

Some examples are:
1. \( \mathcal{I} \) is the set of computable functions;
2. \( \mathcal{I} \) is the set of primitive recursive functions;
3. \( \mathcal{I} \) is the set of partially computable functions which are defined for all but a finite number of values of \( x \).

**Theorem 7.1 (Rice’s Theorem):** Let \( \mathcal{I} \) be a collection of partially computable functions of one variable. Let there be partially computable functions \( f(x) \) and \( g(x) \) such that \( f(x) \) belongs to \( \mathcal{I} \) but \( g(x) \) does not. Then \( R_\mathcal{I} \) is not recursive.

Rice’s Theorem

Rice’s Theorem tells us that there is no way to algorithmically determine non-trivial properties of the function computed by another program. Trivial properties are those that apply to all partially computable functions or none of them. The theorem uses functions \( f(x) \) and \( g(x) \) such that \( f(x) \) belongs to a collection \( \mathcal{I} \) but \( g(x) \) does not - for the sole purpose of excluding such trivial cases.

Numerical Representation of Strings

For example, if we have the string \( w = 372 \), then \( k = 2, i_2 = 3, i_1 = 7, i_0 = 2 \).

To find the number associated with this string, we use exactly the following formula:

\[
x = i_k n^k + i_{k-1} n^{k-1} + \ldots + i_1 n^1 + i_0 n^0
\]

\[
x = 3 \cdot 10^2 + 7 \cdot 10^1 + 2 = 372.
\]

If \( w = 372 \) is an octal representation of an integer, then we would have \( n = 8 \) and therefore:

\[
x = 3 \cdot 8^2 + 7 \cdot 8^1 + 2 = 192 + 56 + 2 = 250
\]

Numerical Representation of Strings

Now let us develop such a method for strings on an alphabet \( A \).

Remember that the set of all strings on an alphabet \( A \), including the empty string, is called \( A^* \).

We write \( A = \{s_1, \ldots, s_n\} \) and define that the sequence \( s_{i_k} \ldots s_{i_1} \) corresponds to this order of symbols.

Then any string \( w \) on \( A \) can be written as \( w = s_{i_k} s_{i_{k-1}} \ldots s_{i_1} s_{i_0} \), where \( 1 \leq i_m \leq n \) and \( k = |w| - 1 \). The empty string is indicated by \( w = 0 \).

Numerical Representation of Strings

Then we use exactly the same formula as before to associate \( w \) with an integer \( x \):

\[
x = \sum_{0}^{k} i_m n^m
\]

With \( w = 0 \) we associate the number \( x = 0 \).

For example, consider the alphabet \( A = \{a, b, c\} \) and the string \( w = caba \).

Then \( x = 3 \cdot 3^3 + 1 \cdot 3^2 + 2 \cdot 3^1 + 1 = 81 + 9 + 6 + 1 = 97 \).

Now why is this representation unique?

We can prove this by showing how to retrieve the subscripts \( i_0, i_1, \ldots, i_k \) from \( x \) for any \( x > 0 \).

Numerical Representation of Strings

First, we define two primitive recursive functions

\[
R^r(x, y) = \begin{cases} R(x, y) & \text{if } y \mid x \\ y & \text{otherwise} \end{cases}
\]

\[
Q^r(x, y) = \begin{cases} \lfloor x / y \rfloor & \text{if } y \mid x \\ \lfloor x / y \rfloor - 1 & \text{otherwise} \end{cases}
\]

where \( R(x, y) \) and \( \lfloor x / y \rfloor \) are defined as in Section 3.7.

Basically, \( R^r \) and \( Q^r \) are the "usual" remainder and quotient functions, except that remainders are now in the range between 1 and \( y \) instead of 0 and \( y - 1 \).
Numerical Representation of Strings

So whenever y divides x, we do not have a remainder of 0 but a remainder of y, and accordingly the quotient is one number below the “actual” quotient.

Therefore, like with the usual quotient and remainder, it is still true that:
\[ x/y = Q(x, y) + R(x, y)/y, \]
only that now we have \( 1 \leq R(x, y) \leq y. \)

We will use the functions \( Q^* \) and \( R^* \) to show how to obtain the subscripts \( i_0, i_1, \ldots, i_k \) from any integer \( x > 0. \)

**Example:** Find binary representation of number 13:

Then \( u_0 = x = 13; \ n = 2 \)

\[
\begin{align*}
\text{Then we have:} \\
u_0 &= i_k \cdot 2^k + i_{k-1} \cdot 2^{k-1} + \ldots + i_1 \cdot 2 + i_0 \\
u_1 &= i_k \cdot 2^{k-1} + i_{k-1} \cdot 2^{k-2} + \ldots + i_1 \\
\vdots \\
u_k &= i_k \\
\text{The remainders } R^* \text{ are exactly the values of the } i_m: \\
i_m &= R^*(u_m, n), \ m = 0, \ldots, k.
\end{align*}
\]

Thus \( k = 3 \) and we have
\[
x = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0
\]