If you think about it, you will see that the specifics of Tot do not play into the proof of ???. We designed \( h(x, y) \) to act like the constant function \( j(y) = 0 \) exactly when \( x \) is in \( K \); but we could just as well have made it act like any other partial computable function under those circumstances. This observation lets us state a more general theorem, which says, roughly, that no nontrivial property of computable functions is decidable.

Keep in mind that \( \varphi_0, \varphi_1, \varphi_2, \ldots \) is our standard enumeration of the partial computable functions.

**Theorem thy.1 (Rice’s Theorem).** Let \( C \) be any set of partial computable functions, and let \( A = \{ n : \varphi_n \in C \} \). If \( A \) is computable, then either \( C \) is \( \emptyset \) or \( C \) is the set of all the partial computable functions.

An *index set* is a set \( A \) with the property that if \( n \) and \( m \) are indices which “compute” the same function, then either both \( n \) and \( m \) are in \( A \), or neither is. It is not hard to see that the set \( A \) in the theorem has this property. Conversely, if \( A \) is an index set and \( C \) is the set of functions computed by these indices, then \( A = \{ n : \varphi_n \in C \} \).

With this terminology, Rice’s theorem is equivalent to saying that no nontrivial index set is decidable. To understand what the theorem says, it is helpful to emphasize the distinction between *programs* (say, in your favorite programming language) and the functions they compute. There are certainly questions about programs (indices), which are syntactic objects, that are computable: does this program have more than 150 symbols? Does it have more than 22 lines? Does it have a “while” statement? Does the string “hello world” every appear in the argument to a “print” statement? Rice’s theorem says that no nontrivial question about the program’s *behavior* is computable. This includes questions like these: does the program halt on input 0? Does it ever halt? Does it ever output an even number?

**Proof of Rice’s theorem.** Suppose \( A \) is neither \( \emptyset \) nor the set of all the partial computable functions, and let \( A \) be the set of indices of functions in \( C \). We will show that if \( A \) were computable, we could solve the halting problem; so \( A \) is not computable.

Without loss of generality, we can assume that the function \( f \) which is nowhere defined is not in \( C \) (otherwise, switch \( C \) and its complement in the argument below). Let \( g \) be any function in \( C \). The idea is that if we could decide \( A \), we could tell the difference between indices computing \( f \), and indices computing \( g \); and then we could use that capability to solve the halting problem.

Here’s how. Using the universal computation predicate, we can define a function

\[
    h(x, y) = \begin{cases} 
        \text{undefined} & \text{if } \varphi_x(x) \uparrow \\
        g(y) & \text{otherwise}.
    \end{cases}
\]

**Explanation**

With this terminology, Rice’s theorem is equivalent to saying that no nontrivial index set is decidable. To understand what the theorem says, it is helpful to emphasize the distinction between *programs* (say, in your favorite programming language) and the functions they compute. There are certainly questions about programs (indices), which are syntactic objects, that are computable: does this program have more than 150 symbols? Does it have more than 22 lines? Does it have a “while” statement? Does the string “hello world” every appear in the argument to a “print” statement? Rice’s theorem says that no nontrivial question about the program’s *behavior* is computable. This includes questions like these: does the program halt on input 0? Does it ever halt? Does it ever output an even number?

**Proof of Rice’s theorem.** Suppose \( A \) is neither \( \emptyset \) nor the set of all the partial computable functions, and let \( A \) be the set of indices of functions in \( C \). We will show that if \( A \) were computable, we could solve the halting problem; so \( A \) is not computable.

Without loss of generality, we can assume that the function \( f \) which is nowhere defined is not in \( C \) (otherwise, switch \( C \) and its complement in the argument below). Let \( g \) be any function in \( C \). The idea is that if we could decide \( A \), we could tell the difference between indices computing \( f \), and indices computing \( g \); and then we could use that capability to solve the halting problem.

Here’s how. Using the universal computation predicate, we can define a function

\[
    h(x, y) = \begin{cases} 
        \text{undefined} & \text{if } \varphi_x(x) \uparrow \\
        g(y) & \text{otherwise}.
    \end{cases}
\]
To compute $h$, first we try to compute $\varphi_x(x)$; if that computation halts, we go on to compute $g(y)$; and if that computation halts, we return the output. More formally, we can write

$$h(x, y) \simeq P^2_0(g(y), \text{Un}(x, x)).$$

where $P^2_0(z_0, z_1) = z_0$ is the 2-place projection function returning the 0-th argument, which is computable.

Then $h$ is a composition of partial computable functions, and the right side is defined and equal to $g(y)$ just when $\text{Un}(x, x)$ and $g(y)$ are both defined.

Notice that for a fixed $x$, if $\varphi_x(x)$ is undefined, then $h(x, y)$ is undefined for every $y$; and if $\varphi_x(x)$ is defined, then $h(x, y) \simeq g(y)$. So, for any fixed value of $x$, either $h(x, y)$ acts just like $f$ or it acts just like $g$, and deciding whether or not $\varphi_x(x)$ is defined amounts to deciding which of these two cases holds. But this amounts to deciding whether or not $h_x(y) \simeq h(x, y)$ is in $C$ or not, and if $A$ were computable, we could do just that.

More formally, since $h$ is partial computable, it is equal to the function $\varphi_k$ for some index $k$. By the s-m-n theorem there is a primitive recursive function $s$ such that for each $x$, $\varphi_{s(k,x)}(y) = h_x(y)$. Now we have that for each $x$, if $\varphi_x(x) \downarrow$, then $\varphi_{s(k,x)}$ is the same function as $g$, and so $s(k, x)$ is in $A$. On the other hand, if $\varphi_x(x) \uparrow$, then $\varphi_{s(k,x)}$ is the same function as $f$, and so $s(k, x)$ is not in $A$. In other words we have that for every $x$, $x \in K$ if and only if $s(k, x) \in A$. If $A$ were computable, $K$ would be also, which is a contradiction. So $A$ is not computable.

Rice’s theorem is very powerful. The following immediate corollary shows some sample applications.

**Corollary thy.2.** The following sets are undecidable.

1. $\{x : 17 \text{ is in the range of } \varphi_x\}$
2. $\{x : \varphi_x \text{ is constant}\}$
3. $\{x : \varphi_x \text{ is total}\}$
4. $\{x : \text{whenever } y < y', \varphi_x(y) \downarrow, \text{ and if } \varphi_x(y') \downarrow, \text{ then } \varphi_x(y) < \varphi_x(y')\}$

**Proof.** These are all nontrivial index sets.

**Photo Credits**

**Bibliography**

2