The Halting Problem Does Not Matter

Suppose that \( P \) is a program whose set of acceptable data \( \text{accept}(P) \) is recursive. Then there is another program \( P' \) with the same acceptable data, which cannot loop, and which behaves like \( P \) for all input data in \( \text{accept}(P) \).

A key measure of program quality is robustness. A program is said to be robust if it will behave sensibly, whatever its input. In particular, it should never under any circumstances go into an infinite loop. There are certain conceivable kinds of data which can only be distinguished and accepted by programs which might loop. Such a set of possible data is called recursively enumerable but not recursive. (This is the terminology of Ref. 1.) If anybody can construct a set of conceivable data which is not even recursively enumerable, then he has discovered a counterexample to Church’s thesis. In practice, all useful programs accept a set of data which is not only recursively enumerable, but also actually recursive.

**Proposition**

Suppose that \( P \) is a program whose set of acceptable data \( \text{accept}(P) \) is recursive. Then there is another program \( P' \) with the same acceptable data:

\[
\text{accept}(P') = \text{accept}(P)
\]

and which cannot loop:

\[
\text{loop}(P') \text{ is empty}
\]

and which behaves like \( P \) for all input data in \( \text{accept}(P) \):

if \( P \) and \( P' \) are both given the same word \( w \) from \( \text{accept}(P) \) as initial datum, then their outputs will be the same.

**Proof**

Throughout what follows, all programs will be regarded as Turing machines. The output of a program is the content of its tape on termination. The key assumption is that \( \text{accept}(P) \) is recursive. This means that there is a program \( Q \) and

\[
\text{accept}(Q) = \text{accept}(P)
\]

and \( Q \) cannot loop. The program \( P' \) is constructed from \( P \) and \( Q \).

Let \( P \) have some input datum \( w \). This datum is a sequence of symbols stored on the left hand end of the machine’s tape. First, \( P' \) copies this word one position to the right on the tape, and inserts a special character (say \( E \), not recognized by \( P \)) before it. Secondly, it inserts another special symbol (say \( F \), not recognized by \( P \) or \( Q \), after the word, and makes a new copy of \( w \) to the right of \( F \). At the end of this, \( P' \) inserts a third special symbol (say \( G \), not recognized by \( Q \)). If \( w \) is the sequence

\[
w_1, w_2, w_3, \ldots, w_n
\]

then by this stage the tape appears as

\[
Ew_1 w_2 \ldots w_n F w_1 w_2 \ldots w_n G
\]

Thirdly, the tape head is moved to the tape cell to the right of the one holding \( F \). Next, \( P' \) performs all the operations of \( Q \). Just one modification is required in \( Q \): when ever the tape head is situated over a cell containing \( G \), \( P' \) will

(a) over-write this cell with a blank character, and move the head to the right
(b) write \( G \), and move the head back left.

During this stage, the input will be rejected if and only if the original datum \( w \) would be rejected by \( Q \).

Where \( Q \) would halt, \( P' \) contains a subroutine which restores the tape to its initial state. This is possible because \( E \) and \( F \) still delimit a copy of \( w \), and \( G \) marks the extent of tape which has been over-written. At the end of the subroutine, which cannot fail or loop, \( P' \) enters the original program \( P \).

Q.E.D.

In practice, programmers worthy of the name always write their code on these lines. The halting problem does not concern them. It should be seen for what it is: a profound feature of mathematics, and a curiosity in the history of computing.

**Reference**


Received April 1983