The Halting Problem

Let us define the predicate $\text{HALT}(x, y)$. For a given number $y$, let $\varphi$ be the program such that $\#(\varphi) = y$. Then $\text{HALT}(x, y)$ is true if $\psi_{\varphi}(1)(x)$ is defined and false if $\psi_{\varphi}(1)(x)$ is undefined.

In other words:

$\text{HALT}(x, y) \iff \text{program number } y \text{ eventually halts on input } x.$

Here comes a surprise:

**Theorem 2.1:** $\text{HALT}(x, y)$ is not a computable predicate.

**Proof (by contradiction):**

Assume that $\text{HALT}(x, y)$ were computable. Then we could write the following program $\varphi$:

$[A] \text{ IF } \text{HALT}(X, X) \text{ GOTO A}$

This program $\varphi$ would compute the following function:

$\psi_{\varphi}(1)(x) = \uparrow$ if $\text{HALT}(x, x)$

$= 0$ if $\sim \text{HALT}(x, x)$

Now let $\#(\varphi) = y_0$. Then by the definition of HALT we get:

$\text{HALT}(x, y_0) \iff \sim \text{HALT}(x, x)$

For input $x = y_0$ we then have:

$\text{HALT}(y_0, y_0) \iff \sim \text{HALT}(y_0, x)$

Contradiction!

So finally we have an example of a predicate that is not computable by any program in the language $L$.

We would even like to conclude the following:

There is no algorithm that, given a program of $L$ and an input to that program, can determine whether or not the given program will eventually halt on the given input.

This is called the unsolvability of the halting problem.

The Halting Problem

If there were such an algorithm, we could use it to determine the truth value of $\text{HALT}(x, y)$ for given $x$ and $y$:

- We would first obtain the program $\varphi$ so that $\#(\varphi) = y$.
- Then we would check whether $\varphi$ eventually halts on input $x$.

However, we have reason to believe that any algorithm for computing on numbers can be carried out by a program of $L$. (Church's Thesis).

So this would contradict the fact that $\text{HALT}(x, y)$ is not computable.

The Halting Problem

This program is based on Goldbach's conjecture, which assumes that every even number $\geq 4$ is the sum of two prime numbers.

For example, $4 = 2 + 2, 6 = 3 + 3, 48 = 41 + 7$.

It would be easy to write a program in $L$ that searches for a counterexample to this conjecture.

This program would check the following predicate for increasing values $n$:

$\neg \exists x \exists y (\prime(x) \land \prime(y) \land x + y = n)$

Nobody knows whether this program will ever halt.
Universality
For each $n > 0$, let us define:
\[ \Phi(n)(x_1, \ldots, x_n, y) = \psi_P(n)(x_1, \ldots, x_n), \quad \text{where } \#(P) = y. \]

Theorem 3.1 (Universality Theorem):
For each $n > 0$, the function $\Phi(n)(x_1, \ldots, x_n, y)$ is partially computable.

This is one of the most important theorems in computability theory.
We will prove it by providing instructions for writing a program $U_n$ that computes $\Phi(n)$ for each $n > 0$.

Universality
In other words, for each $n > 0$ we want to have:
\[ \psi_{U_n}(n+1)(x_1, \ldots, x_n, x_{n+1}) = \Phi(n)(x_1, \ldots, x_n, x_{n+1}). \]
These programs $U_n$ are called universal.
For example, $U_1$ can be used to compute any partially computable function of one variable.
If there is a program $P$ that computes $f(x)$, and $\#(P) = y$, then $f(x) = \Phi(1)(x, y) = \psi_{U_1}(2)(x, y)$.

It is useful to think of the programs $U_n$ in terms of interpreters of programs in $L$.

Universality
The universal programs must
• decode the number of the program given to them,
• keep track of the current snapshot during program execution, and
• generate the next snapshot based on the current one and the current instruction.

When we write such programs, we will freely use macros referring to functions that we already know to be primitive recursive.
We will also freely use label and variable names beyond those specified for the language $L$.

Universality
In describing the state of a computation, we assume all variables to have the value 0 if not assigned a different value.
Then we can code the state of the computation by the Gödel number $[a_1, \ldots, a_m]$, where $a_i$ is the value of the $i$-th variable in our ordered list.
Obviously, $m$ is chosen so that all $a_i$ for $i > m$ are 0.
For example, the state $Y = 1, X = 2, Z^2 = 1$ is coded by the following number:
$[1, 2, 0, 0, 1] = 2 \cdot 3^2 \cdot 11 = 198$.

Universality
In order to store the current snapshot, we need to keep track of two numbers:
• $K$ is the number of the instruction to be executed next, and
• $S$ is the current state coded as a Gödel number (see previous slide).

Now we are ready to write the program $U_n$ for computing $Y = \Phi(n)(X_1, \ldots, X_n, X_{n+1})$.
We will explain $U_n$ piece by piece and finally put the pieces together.
Universality

Then we append the following instruction:

\[ [C] \quad \text{IF } K = \text{Lt}(Z) + 1 \lor K = 0 \text{ GOTO F} \]

So if the computation has ended, GOTO F, where the proper value will be output.

Otherwise, the current instruction is decoded and executed:

\[ U \leftarrow \tau((Z)_k) \]
\[ P \leftarrow p_{uj+1} \]

Remember that \((Z)_k = (a, (b, c))\) is the number of the \(K\)-th instruction.

So \(U = (b, c)\) is the code of the statement to be executed.

The variable mentioned in this statement is the \((c + 1)\)-th, i.e., the \((r(U) + 1)\)-th.