

CS 5000: Lecture 33

Vladimir Kulyukin

Department of Computer Science

Utah State University

Outline

- The Halting Problem
- Church's Thesis
- Universal Programs

Reasoning about Program Properties

- We can use Gödel numbers to reason about properties of programs.
- We can ask, for example, if a program halts on a given input.

Review: The Value of Y

$$\Psi_P^{(m)}(r_1, r_2, \dots, r_m) \Leftrightarrow$$

the value of Y at the terminal
snapshot.

The Halting Problem

HALT(x,y)

Let P be a program such that $\#(P) = y$.

$$HALT(x, y) = \begin{cases} 1 & \text{if } \Psi_P^{(1)}(x) \downarrow \\ 0 & \text{if } \Psi_P^{(1)}(x) \uparrow \end{cases}$$

HALT(x,y)

HALT(x, y) \Leftrightarrow program number y halts
on input number x .

Theorem 2.1 (Ch. 4)

HALT(x, y) is not a computable predicate.

Proof 2.1

Suppose $HALT(x, y)$ is computable. Consider the following program P :

[A] IF $HALT(x, x)$ GOTO A

Proof 2.1

What is $\Psi_P^{(1)}(x)$?

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if } HALT(x, x) = 1 \\ \downarrow & \text{if } HALT(x, x) = 0. \end{cases}$$

Or :

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if } HALT(x, x) \\ \downarrow & \text{if } \neg HALT(x, x). \end{cases}$$

Proof 2.1

Since P is a program, we can compute $\#(P)$.

Let $\#(P) = y_0$. Then

1. $HALT(x, y_0) = 1$ if $HALT(x, x) = 0$.

2. $HALT(x, y_0) = 0$ if $HALT(x, x) = 1$.

Or :

$(\forall x)HALT(x, y_0) \Leftrightarrow \neg HALT(x, x)$.

Proof 2.1

$$(\forall x)HALT(x, y_0) \Leftrightarrow \neg HALT(x, x).$$

Let $x = y_0$.

$$HALT(y_0, y_0) \Leftrightarrow \neg HALT(y_0, y_0).$$

Theorem 2.1 (Ch. 4)

- Theorem 2.1 gives us an example of a function that is not computable.
- A key lesson we can draw from Theorem 2.1 is this:

The fact that a function can be defined mathematically as total does not mean that the function can be computed.

Theorem 2.1: An Important Implication

There is no algorithm that, given a program P in L and an input x , can determine whether or not P will eventually halt on x .

Church's Thesis

Church's Thesis: Prelims

- The notion of *algorithm* can be defined only informally.
- There are many formal characterizations of algorithms (Note the rapid proliferation of different programming languages!).
- The claim that each of the standard formal characterizations of algorithm is satisfactory ***cannot be proved***.
- The claim must be accepted on ***empirical*** grounds.

Church's Thesis: Concrete Form

Any algorithm for computing on numbers
can be carried out by a program in L (C++, Java, Python, etc.)

Church's Thesis: General Form

Any algorithm for computing on numbers
can be carried out by a standard formalization
of the informal notion of algorithm.

Why is it a Thesis?

- Why is ***Church's Thesis*** a *thesis*?
- There is no general definition of algorithm separate from a standard formalization of algorithm (a particular programming language).
- The claim that a standard programming language provides a satisfactory characterization of computation remains empirical and cannot be proved.
- Church's Thesis cannot be proved as a theorem.

Is There Hope?

...many mathematicians have accepted the claim that the standard characterizations give a satisfactory formalization, or “rational reconstruction,” of the (necessarily vague) informal notions.

Hartley Rogers, Jr. “Theory of Recursive Functions and Effective Computability”

Is There Hope?

- Computability Proof Techniques
 - Functions with informal algorithms are partially computable.
- Automated Programming
 - Automated generation of programs from informal sets of instructions and/or specifications.

Universal Programs

Universality

- Coding programs by numbers gives us a mathematical theory of compilation.
- Compiled programs must be executed or interpreted.
- Universality gives us a mathematical theory of program execution and interpretation, i.e., a mathematical theory of operating systems.

Universal Programs: Step 1

- Suppose we have a program P that takes n arguments.
- Suppose that $\#(P) = y$.
- We would like to construct another program that can take $\#(P)$ and n argument values x_1, \dots, x_n and execute it on those values.

Universal Programs: Step 2

- But why stop with just one program?
- We can attempt to construct a program that can execute/interpret ***any*** program of ***n*** arguments.
- Thus, we can have programs that can execute any program of 1 argument, any program of 2 arguments, any program of 3 arguments, etc.

Universal Programs: Step 2

- For each $n > 0$, such a program is denoted by U_n and is called *universal*.
- Why is it *universal*? Because it can run *any* program of n arguments.
- We have a sequence of universal programs: $U_1, U_2, U_3, U_4, \dots$

Recommended Reading

- Sections 4.2, 4.3.