Lecture 3: Turing, computability, halting problem

David Lester

2017

Outline

1. Introduction
2. Computability
3. The Church-Turing Thesis
4. Computable functions for $\alpha \rightarrow \beta$
5. The Halting Problem: An informal Argument

Introduction

At the end of this lecture you will:

- Be able to show that a function is computable;

Introduction

At the end of this lecture you will:

- Be able to show that a function is computable;
- Understand the use of diagonalization;

Introduction

At the end of this lecture you will:

- Be able to show that a function is computable;
- Understand the use of diagonalization;
- Understand how to use the Church-Turing Thesis;

Introduction

At the end of this lecture you will:

- Be able to show that a function is computable;
- Understand the use of diagonalization;
- Understand how to use the Church-Turing Thesis;
- Be able to show that a predicate is decidable; and

Introduction

At the end of this lecture you will:

- Be able to show that a function is computable;
- Understand the use of diagonalization;
- Understand how to use the Church-Turing Thesis;
- Be able to show that a predicate is decidable; and
- Give an informal argument that it is not possible to test whether a program halts algorithmically.

Computable Functions $\mathbb{N} \to \mathbb{N}$

- A computable function is one for which we can write a program to implement the function.

Computable Functions $\mathbb{N} \to \mathbb{N}$

- A computable function is one for which we can write a program to implement the function.
- We will write these programs in our simple programming language `while`.

Computable Functions $\mathbb{N} \to \mathbb{N}$

- A computable function is one for which we can write a program to implement the function.
- We will write these programs in our simple programming language `while`.
- By convention we will take the argument to a unary function by setting the variable `x` in the initial state s.

Computable Functions $\mathbb{N} \to \mathbb{N}$

- A computable function is one for which we can write a program to implement the function.
- We will write these programs in our simple programming language `while`.
- By convention we will take the argument to a unary function by setting the variable `x` in the initial state s.
- Likewise, we will read out the answer from the variable `x` in the final state s', if the program terminates.

Definition

A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if, and only if,

- There exists a `while` program S; and

Definition

A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if, and only if,

- There exists a `while` program S; and
- for each n there is a starting state s with $s(\mathtt{x}) = n$; and

Definition

A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if, and only if,

- There exists a `while` program S; and
- for each n there is a starting state s with $s(\mathtt{x}) = n$; and
- the program S takes m steps to execute to a final state s' with s as starting state; *i.e.*

$$< S,\ s > \Rightarrow^m s';$$

Definition

A function $f : \mathbb{N} \to \mathbb{N}$ is *computable* if, and only if,

- There exists a `while` program S; and
- for each n there is a starting state s with $s(\mathtt{x}) = n$; and
- the program S takes m steps to execute to a final state s' with s as starting state; *i.e.*

$$< S,\ s > \Rightarrow^m s';$$

- then

$$s'(\mathtt{x}) = f(n).$$

while-computable functions

- When we need to make distinctions we may call this while-computability.

`while`-computable functions

- When we need to make distinctions we may call this `while`-computability.
- There are also Turing-computable, λ-computable, *etc.* functions.

`while`-computable functions

- When we need to make distinctions we may call this `while`-computability.
- There are also Turing-computable, λ-computable, *etc.* functions.
- As we will see later, these all define the same set of functions to be computable.

Lemma

There are uncountably many functions from $\mathbb{N} \to \mathbb{N}$*.*

Lemma

There are uncountably many functions from $\mathbb{N} \to \mathbb{N}$.

- We will prove this using *Diagonalization*.

Lemma

There are uncountably many functions from $\mathbb{N} \to \mathbb{N}$.

- We will prove this using *Diagonalization*.
- We will prove this in some detail.

Proof (I)

- There must be infinitely many functions, because there are infinitely many constant functions:

$$f_0(n) = 0,\ \ f_1(n) = 1,\ \ \ldots\ \ f_k(n) = k,\ \ \ldots$$

Proof (I)

- There must be infinitely many functions, because there are infinitely many constant functions:

$$f_0(n) = 0, \ f_1(n) = 1, \ \dots \ f_k(n) = k, \ \dots$$

- Each of these functions is computable because the program

$$x := k$$

implements the constant function $f_k(n) = k$.

Proof (I)

- There must be infinitely many functions, because there are infinitely many constant functions:

$$f_0(n) = 0, \; f_1(n) = 1, \; \ldots \; f_k(n) = k, \; \ldots$$

- Each of these functions is computable because the program

$$x := k$$

implements the constant function $f_k(n) = k$.
- Suppose that there are only *countably* infinitely many functions of type $\mathbb{N} \to \mathbb{N}$.

Proof (I)

- There must be infinitely many functions, because there are infinitely many constant functions:

$$f_0(n) = 0,\ f_1(n) = 1,\ \ldots\ f_k(n) = k,\ \ldots$$

- Each of these functions is computable because the program

$$x := k$$

implements the constant function $f_k(n) = k$.

- Suppose that there are only *countably* infinitely many functions of type $\mathbb{N} \to \mathbb{N}$.
- A countably infinite set A has a bijection ϕ with $\mathbb{N}$.

Proof (I)

- There must be infinitely many functions, because there are infinitely many constant functions:

$$f_0(n) = 0,\ f_1(n) = 1,\ \dots\ f_k(n) = k,\ \dots$$

- Each of these functions is computable because the program

$$x := k$$

  implements the constant function $f_k(n) = k$.
- Suppose that there are only *countably* infinitely many functions of type $\mathbb{N} \to \mathbb{N}$.
- A countably infinite set A has a bijection ϕ with $\mathbb{N}$.
- Thus we can lay out the set of functions in a sequence

$$f_0,\ f_1,\ \dots,\ f_k,\ \dots$$

Proof (II)

- We now *construct* a function which is *not* already in our list.

Proof (II)

- We now *construct* a function which is *not* already in our list.
- Two functions are the same (written $f = g$) if, and only if

$$\forall\ (n \in \mathbb{N}).\ f(n) = g(n)$$

This is known as *extensionality*.

Proof (II)

- We now *construct* a function which is *not* already in our list.
- Two functions are the same (written $f = g$) if, and only if

$$\forall\ (n \in \mathbb{N}).\ f(n) = g(n)$$

  This is known as *extensionality*.
- The new function is defined as follows:

$$f(n) = f_n(n) + 1$$

Proof (II)

- We now *construct* a function which is *not* already in our list.
- Two functions are the same (written $f = g$) if, and only if

$$\forall\ (n \in \mathbb{N}).\ f(n) = g(n)$$

  This is known as *extensionality*.
- The new function is defined as follows:

$$f(n) = f_n(n) + 1$$

- Because f is different from f_n for argument n, we know that

$$f \neq f_n$$

Proof (III)

To recap, we have shown:

Proof (III)

To recap, we have shown:

- There are infinitely many functions $\mathbb{N} \to \mathbb{N}$; and

Proof (III)

To recap, we have shown:

- There are infinitely many functions $\mathbb{N} \to \mathbb{N}$; and
- If we assume that we *can* enumerate all of the functions in $\mathbb{N} \to \mathbb{N}$, it turns out that we *cannot*, because there is a missing function (f).

Proof (III)

To recap, we have shown:

- There are infinitely many functions $\mathbb{N} \rightarrow \mathbb{N}$; and
- If we assume that we *can* enumerate all of the functions in $\mathbb{N} \rightarrow \mathbb{N}$, it turns out that we *cannot*, because there is a missing function (f).
- Therefore we have shown that there are *uncountably* many functions $\mathbb{N} \rightarrow \mathbb{N}$.

Countable *vs* Uncountable

- We have shown that there are countably many *computable* functions of type $\mathbb{N} \to \mathbb{N}$.

Countable *vs* Uncountable

- We have shown that there are countably many *computable* functions of type $\mathbb{N} \to \mathbb{N}$.
- But there are uncountably many functions of type $\mathbb{N} \to \mathbb{N}$.

Countable *vs* Uncountable

- We have shown that there are countably many *computable* functions of type $\mathbb{N} \to \mathbb{N}$.
- But there are uncountably many functions of type $\mathbb{N} \to \mathbb{N}$.
- This means that there *must* be functions which are *not* computable.

Non-computability

Corollary

There are non-computable functions of type $\mathbb{N} \to \mathbb{N}$.

The Church-Turing Thesis

- During the 1930s many different ways were found to define what we would now call the concept of computation.

The Church-Turing Thesis

- During the 1930s many different ways were found to define what we would now call the concept of computation.
- The driver for this inventiveness was a desire to fill in some missing details in Gödel's Incompleteness Theorem.

Definitions of Computation

Amongst the better known are:

Schönfinkel's Combinators 1924

Church's λ-Calculus 1936

Gödel-Kleene μ-recursive functions 1936

Turing's Turing Machines 1936

Post Production System 1943

Markov Computable Functions 1954

Shepherdson and Sturgiss' URM 1963

The Church-Turing Thesis

Thesis (Church-Turing)

Any sensible definition of computation will define the same functions to be computable as any other definition.

The Church-Turing Thesis

Important We can paraphrase the Church-Turing Hypothesis as: "A function is computable whenever we can write a program to implement it."

Computable functions for $\alpha \to \beta$

- We now generalize the definition of computability to functions of other types.

Computable functions for $\alpha \to \beta$

- We now generalize the definition of computability to functions of other types.
- To do this, we use coding techniques to code other types into $\mathbb{N}$.

Computable functions for $\alpha \to \beta$

- We now generalize the definition of computability to functions of other types.
- To do this, we use coding techniques to code other types into $\mathbb{N}$.
- The most important coding technique is $\phi_X : (\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, defined as:

$$\phi_X(n, m) = 2^n(2m + 1) - 1$$

Computable functions of type $(\mathbb{N} \times \mathbb{N}) \to \mathbb{N}$

Definition

We say that a function $f : (\mathbb{N}, \mathbb{N}) \to \mathbb{N}$ is *computable*, if, and only if, the function $g : \mathbb{N} \to \mathbb{N}$ is computable using the previous Definition, where

$$f(x, y) = g(\phi_X(x, y))$$

Computable functions of type $\mathbb{N} \to (\mathbb{N} \times \mathbb{N})$

Definition

We say that a function $f : \mathbb{N} \to (\mathbb{N}, \mathbb{N})$ is *computable*, if, and only if, the function $g : \mathbb{N} \to \mathbb{N}$ is computable using the previous Definition, where

$$f(x) = \phi_X^{-1}(g(x))$$

Generalized Computability on $\mathbb{N}^m \to \mathbb{N}^n$

Definition

A function $f : \mathbb{N}^m \to \mathbb{N}^n$, with $n, m \geq 1$, is *computable* if, and only if, there is a function $g : \mathbb{N} \to \mathbb{N}$ which is computable in the sense of the Definition for $\mathbb{N} \to \mathbb{N}$, such that

$$g(\phi_x(x_1,\ \phi_x(x_2,\ \ldots \phi_x(x_{n-1},\ x_n)))) = \\ (\phi_x(y_1,\ \phi_x(y_2,\ \ldots\ \phi_x(y_{m-1},\ y_m)\ \ldots\ ))$$

where,

$$(y_1,\ y_2,\ \ldots y_{m-1},\ y_m) = f(x_1,\ x_2, \ldots,\ x_{n-1},\ x_n))$$

Decidable Predicates

Definition

The predicate P is *decidable* if, and only if, there is a computable function $f : \mathbb{N} \to \mathbb{N}$ such that:

$$f(x) = \begin{cases} 1 & \text{if } P(x) \text{ holds} \\ 0 & \text{if } P(x) \text{ doesn't hold} \end{cases}$$

Notes

- A function that is not decidable is *undecidable*.

Notes

- A function that is not decidable is *undecidable*.
- The associated function f is the characteristic function for the predicate P.

Notes

- A function that is not decidable is *undecidable*.
- The associated function f is the characteristic function for the predicate P.
- The `while` program implementing f is called the *decision procedure* for P.

Lemma

If P and Q are decidable predicates, then all of the following are also decidable:

- $\neg P$;
- $P \wedge Q$;
- $P \vee Q$; *and*
- $P \Rightarrow Q$.

Partially Decidable Predicates

- Notice that decidable predicates are total, *i.e.* every input value gives a value of true or false.

Partially Decidable Predicates

Definition

Partially Decidable Predicates

Partially Decidable Predicates

Definition

Partially Decidable Predicates

- The partial function P is *partially decidable* if, and only if, there exists a computable partial function $f : \mathbb{N} \hookrightarrow \mathbb{N}$ with

$$f(x) = \begin{cases} 1 & \text{if } P(x) \text{ holds} \\ \text{undefined} & \text{if } P(x) \text{ doesn't hold} \end{cases}$$

Partially Decidable Predicates

Definition

Partially Decidable Predicates

- The partial function P is *partially decidable* if, and only if, there exists a computable partial function $f : \mathbb{N} \hookrightarrow \mathbb{N}$ with

$$f(x) = \begin{cases} 1 & \text{if } P(x) \text{ holds} \\ \text{undefined} & \text{if } P(x) \text{ doesn't hold} \end{cases}$$

- The partial function f is called the *partial characteristic function* of P, and

Partially Decidable Predicates

Definition

Partially Decidable Predicates

- The partial function P is *partially decidable* if, and only if, there exists a computable partial function $f : \mathbb{N} \hookrightarrow \mathbb{N}$ with

$$f(x) = \begin{cases} 1 & \text{if } P(x) \text{ holds} \\ \text{undefined} & \text{if } P(x) \text{ doesn't hold} \end{cases}$$

- The partial function f is called the *partial characteristic function* of P, and
- the associated progam in `while` is a *partial decision procedure* for P.

- We will outline an informal argument that we cannot write a program to detect when another program will terminate with a given input.

The Halting Problem: An informal Argument

- We will outline an informal argument that we cannot write a program to detect when another program will terminate with a given input.
- To generate a contradiction, we will *assume* that the 'halt-tester' program is S_{halt}, and that this takes the program p and the program's input n as inputs (as a pair in variable x) and outputs either 0 or 1 in variable x, representing false and true respectively.

The Halting Problem: An informal Argument

- We will outline an informal argument that we cannot write a program to detect when another program will terminate with a given input.
- To generate a contradiction, we will *assume* that the 'halt-tester' program is S_{halt}, and that this takes the program p and the program's input n as inputs (as a pair in variable x) and outputs either 0 or 1 in variable x, representing false and true respectively.
- In other words we have assumed that the predicate $\text{halts}(p,\ n)$ is decidable, and has decision procedure S_{halt}.

Program `self`

- The next program to define is S_{self}, this takes a program p as input and returns true $(x = 1)$ if the program halts when its input is itself, and false $(x = 0)$ otherwise.

Program `self`

- The next program to define is S_{self}, this takes a program p as input and returns true $(x = 1)$ if the program halts when its input is itself, and false $(x = 0)$ otherwise.
- We can define the decision procedure S_{self} as:

$$
\begin{array}{l}
z := x;\ y := 1;\ \texttt{while}\ 1 \leq z\ \texttt{do}\ (y := y \times 2;\ z := z - 1); \\
x := (2 \times x + 1) \times y - 1; \\
S_{\text{halt}}
\end{array}
$$

Program `weird`

We now come to the clever bit.

- We define the following weird partial function:

$$\text{weird}(p) = \begin{cases} \text{undefined} & \text{if self}(p) \\ \text{true} & \text{otherwise} \end{cases} \tag{1}$$

Program `weird`

We now come to the clever bit.

- We define the following weird partial function:

$$\text{weird}(p) = \begin{cases} \text{undefined} & \text{if self}(p) \\ \text{true} & \text{otherwise} \end{cases} \tag{1}$$

- The partial function weird is computable, because we can write its program S_{weird} as:

$$S_{\text{self}};\ \texttt{if}\ x = 1\ \texttt{then}\ (\texttt{while true do skip})\ \texttt{else}\ x := 1$$

We now come to a paradox, *i.e.* something that is both logically true and logically false. What happens when we supply the partial function wierd with itself as input?

- Using Equation 1, we see that

$$\text{weird}(\text{weird}) = \begin{cases} \text{undefined} & \text{if self}(\text{weird}) \\ \text{true} & \text{if } \neg\text{self}(\text{weird}) \end{cases} \tag{2}$$

We now come to a paradox, *i.e.* something that is both logically true and logically false. What happens when we supply the partial function wierd with itself as input?

- Using Equation 1, we see that

$$\mathrm{weird}(\mathrm{weird}) = \begin{cases} \mathrm{undefined} & \text{if } \mathrm{self}(\mathrm{weird}) \\ \mathrm{true} & \text{if } \neg\mathrm{self}(\mathrm{weird}) \end{cases} \tag{2}$$

- But

$$\mathrm{self}(\mathrm{weird}) = \mathrm{halt}(\mathrm{weird},\ \mathrm{weird})$$

Case Analysis

There are now two cases for halt(weird, weird):

true In this case we take the first branch of Equation 2, which goes into an infinite loop, *i.e.* it fails to terminate. But this is the effect of running the program weird using its own representation as input, and the halt-tester tells us this terminates. It is therefore a contradiction.

Thus no matter whether the result is true or false, we have generated a contradiction. We have therefore shown that it is impossible to write a halt-tester program in our while language.

Case Analysis

There are now two cases for halt(weird, weird):

true
: In this case we take the first branch of Equation 2, which goes into an infinite loop, *i.e.* it fails to terminate. But this is the effect of running the program weird using its own representation as input, and the halt-tester tells us this terminates. It is therefore a contradiction.

false
: In this case we take the second branch of Equation 2, which returns true, and thus running the program weird with itself as its input terminates. However, this contradicts the result given by the halt-tester, which is false. It is therefore also a contradiction.

Thus no matter whether the result is true or false, we have generated a contradiction. We have therefore shown that it is impossible to write a halt-tester program in our while language.

Conclusion

This does not quite show that it is impossible to write a halt-tester, because maybe the problem lies in the expressiveness of the programming language, and perhaps using a different programming language with extra features will permit us to write the halt-tester.

The notes will show that this is not the case, by making the proof more formal.