Lecture 5: The Halting Problem

Michael Beeson

Historical situation in 1930

- The diagonal method appears to offer a way to extend just about any definition of "computable."
- It appeared in the 1920s that it might even not be possible to define "computable" in a precise way.
- But n the 1930s it was realized that the way around this difficulty was to define *partial computable* functions: that is, to consider functions whose domain is only a subset of the natural numbers, not the entire set of natural numbers.

Notation $f(x) \cong g(x)$

When we write $f(x) \cong g(x)$, we mean that if either of $f(x)$ or $g(x)$ is defined, then the other is also, and the values are equal.

By contrast, $f(x) = g(x)$ means both are defined, and they are equal.

Unbounded search

The easiest way to introduce partial computable functions is just to add "unbounded search" to the primitive recursive functions. This is traditionally done by introducing the "μ operator" or "least-number" operator. Assuming $P(x,y)$ is total, i.e. defined for all x, y, we define

$$\mu x P(x,y) = \begin{cases} \text{the least } x \text{ such that } P(x,y) & \text{if there is one} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Here y can be a list of variables, and x does not have to be the first variable of P as shown.

μ-recursive functions

More generally, we do not need $P(x, y)$ to be total. If $f(x, y)$ is a partial function, then we define

$$\mu x(f(x,y)=0) \cong \begin{cases} \text{the least } x \text{ such that } f(x,y)=0 & \\ \qquad\qquad \text{and } \forall z < x(f(z,y) \text{ is defined} & \\ \text{undefined} & \text{if there is no such } x \end{cases}$$

We will later see that in fact this gets us no more functions than when we require P to be total, and moreover, only *one search* is enough. But in the definition, we allow the more general form:

Definition

f is *μ-recursive* if either f is primitive recursive, or $f(x, y) \cong \mu x(f(x, y) = 0)$ where f is partial recursive.

Partial recursive functions

We would call these functions “partial recursive” except that this phrase is defined in Kleene to mean an equation defined by some recursion equations. We are not studying Kleene’s notion of recursion equations in this course, and the functions so defined turn out to be the same as the μ-recursive functions. The nomenclature “μ-recursive” is not standard, so we will often, by force of habit, use the standard phrase “partial recursive” to refer to the μ-recursive functions.

Using partial functions blocks diagonalization

We can assign indices to the μ-recursive functions as we did for the primitive recursive functions, using $\langle 6, n, \ldots \rangle$ for the index of a function defined using the least-number operator. Then we can diagonalize, but we no longer get a contradiction, as we shall now show. Having enumerated the μ-recursive functions as $\phi_1, \phi_2, \ldots$ we consider f defined by $f(n) = \phi_n(n) + 1$. Then f must be ϕ_m for some m, and we consider $f(m)$, which if defined is equal to $\phi_m(m) + 1 = f(m) + 1$. But all we get is $f(m) \cong f(m) + 1$. This is not a contradiction; it only proves that $f(m)$ is undefined, i.e. m is not in the domain of f.

So is μ-recursive a good definition of “computable”?

Getting around diagonalization was a big step forwards. One might conjecture that every computable function is μ-recursive. But people were quite hesitant to do that, since they had already had a lot of experience in finding new, complicated but still computable, functions that went beyond this or that explicit definition of computability. In the exercises, you will see that certain generalizations do not actually give more functions.

The Zeitgeist of 1930

In the 1930s, computers did not exist yet, and the idea of "computer program" did not exist, and the idea of "algorithm", if it existed at all, was not commonplace. There are several other ideas that are commonplace in 2014, that were not well understood in 1930:

- Programs can be regarded as strings (words or finite lists of characters)
- Strings are data; therefore programs can also be data
- Characters in an alphabet can be represented by numbers (their ascii codes)
- Numbers can be represented as strings of binary digits
- Therefore any string can also be represented as string of binary digits
- Therefore any string can be represented as a (possibly large) number

Coding plus diagonalization

- Those ideas about “coding” things like proofs, machines, and computations as strings and ultimately as integers, which now seem obvious, constitute a sizable portion of the technical work in the original papers of Turing, Church, and Gödel.
- They added the method of “diagonalization”, which had earlier been introduced by Cantor to prove the uncountability of the reals, and used by Russell to derive his famous paradox.
- From coding and diagonalization, the main results of the subject follow.

The halting problem

- The halting problem asks for an algorithm that takes as input a computer program p and an integer x, and outputs YES or NO, according to whether program p run on input x eventually halts (instead of entering an infinite loop, say).
- Another version of the halting problem is about programs p that compute without input, and asks whether there is an algorithm to decide if such a program halts or not.
- Yet another version asks about whether a program p halts on itself, i.e. whether p halts at input p.

Turing's famous result

The halting problem is unsolvable: there is no such algorithm. Here we give a sketch of the proof, as written up by the number theorist Bjorn Poonen in his paper, *Undecidability in Number Theory*.

Sketch of proof. Fix an encoding of programs as nonnegative integers; identify programs with their integer codes. Suppose that there were an algorithm for deciding when program p halts on input x. Using this we could build a new program H such that for any x, H halts on input x if and only if program x does not halt on input x. Taking $x = H$, we find a contradiction: H halts on input H if and only if H does not halt on input H.

More precision needed!

To turn that sketch into a proof we must

- define “program” precisely
- Assign integer codes to programs
- Show that there is a program that computes the result $App(e, x)$ of applying program (with index) e to input x.

How to achieve the required precision

Today, these things can be done in dozens of different ways, and many of these different ways are of independent interest. Historically, the unsolvability of the halting problem was proved long before the invention of the many programming languages in use today. Indeed, the bulk of Turing's work, as well as Church's, was to produce a "model of computation", i.e. an abstract mathematical notion of computability. Turing's model was based on an analysis of mechanical computation, and is called "Turing machines." Church's model is based on an analysis of ways of defining functions, and is called "λ-calculus". We will consider both of these models of computation, but in the twenty-first century, it makes more sense to start with the well-known modern programming languages.

Programs as strings

Pick your favorite programming language $\mathcal{L}$. Then programs in $\mathcal{L}$ are strings over the ASCII alphabet (codes 32-127, plus a newline code, say 10. The precise definition of "program" is given in the manual for $\mathcal{L}$.

Integer codes for programs are a special case of coding any string as an integer. Integer codes for strings are obtained by regarding any string as a number base 2, with eight bits per character. Thus the string `cab` is represented by the number obtained by concatenating the codes of the three characters, which are 99, 97, and 98, respectively. For technical reasons it may be helpful to follow the null-terminator convention of adding 8 zero bytes to mark the end of a string. In binary, 97 is `01100001`, so the numerical code of `cab` is `01100001011000110110001000000000`.

Interpreters

Finally, we need to define $App(e, x)$. The program for App is known as an "interpreter" for $\mathcal{L}$. It takes a program e (in string form) and an input x (also in string form) and emulates the execution of e at input x.

Therefore, to make Poonen's proof sketch precise, we just need to pick a language $\mathcal{L}$, and write an interpreter for $\mathcal{L}$ in $\mathcal{L}$. It is well known that interpreters can be written for various languages, but each one involves some level of technical detail. In the next slide, we consider the options.

An interpreter for $\mathcal{L}$ written in $\mathcal{L}$

One the most popular languages in 2014 is Java. Is there a Java interpreter written in Java? The "Java virtual machine" (JVM) presents itself as an obvious candidate. But actually, it works not on Java programs, but on a compiled version of Java programs known as "bytecode." Of course, the JVM, after being written in Java, is itself compiled into bytecode, and bytecode programs can also be considered as strings, so the JVM can be considered as an interpreter for Java bytecode written in Java bytecode. But to use this to complete our proof of the unsolvability of the halting problem, we would need to prove the correctness of the JVM, i.e., we would have to prove that the JVM's emulation of any bytecode program e at input x gives the same output as program e at input x. Because the Java language is complicated, that would be a difficult proof indeed. One would probably only be satisfied with a computer-checked correctness proof. I do not know whether any JVM has been "formally verified" in this sense.

How about C?

Googling for “C interpreter”, one quickly finds an article in Dr. Dobb’s Journal with a complete listing for a C interpreter. This is beautiful and humanly-readable code. I recommend it to any student who is also a C programmer. It is somewhat surprising that it is easier to find a C interpreter than a Java interpreter.

But the students of this subject are not all C programmers, so we will say no more about this.

LISP (List Processing Language)

The most elegant solution is LISP, a computer language widely used in the early decades of artificial intelligence research. Time permitting, we may later examine the language of "pure LISP" because of its connection to Church's lambda-calculus and to Chaitin's work on algorithmic information theory. But for those with an acquaintance with LISP, let me point out that LISP is the only major computer language with a built-in interpreter: the EVAL function of LISP is exactly the App that we need to define. And every course in LISP includes a chapter on writing EVAL in LISP.

However, in the interest of following the textbook closely, and of not getting bogged down trying to write (and debug) actual running programs, we won't go into LISP now.

A universal μ-recursive function

One way of getting a precise proof of the unsolvability of the halting problem is to use the μ-recursive functions. In an earlier slide, we assigned an integer index to each μ-recursive function. Let ϕ_n be the partial recursive function with index n.

Theorem

$App(e, x) := \phi_e(x)$ *is a μ-recursive function.*

Proof. This is by no means trivial. I do not know a direct proof; we will derive it later indirectly. The definition of $App(e, x)$ is by a complicated recursion on e, so the way to try to prove it is to show that the class of partial recursive functions is closed under arbitrary recursions. That's what led Kleene and Gödel to consider systems of recursion equations as a definition of computation. But we are going to follow Turing's approach instead. Eventually, using Turing machines, we *will* show that the μ-recursive functions are closed under arbitrary recursions!

A “universal function” is just an interpreter

In connection with models of computation based on “machines”, it is traditional to use the terminology “universal machine” (introduced by Turing) instead of “interpreter”. The idea is that a universal machine (for a certain class of machines) is capable of emulating any machine in that class of machines. The classical case is Turing machines, which we take up in the next lecture.

Models of Computation

Historically, the unsolvability of the halting problem was proved long before the invention of Java and the other programming languages in use today. Indeed, the bulk of Turing's work, as well as Church's, was to produce a "model of computation", i.e. an abstract mathematical notion of computability sufficient to carry out the diagonal argument above.

We consider the diagonal argument for the unsolvability of the halting problem, and try to extract what is essential, without saying exactly what a "program" is. Whatever programs are, we assume they can be represented as strings, and they can "request inputs", and they can "terminate". Since programs and data can both can be considered to be strings, it makes sense to have a partial operation App that applies program e to input x, where e and x are both strings.

What is a model of computation?

With so many examples at hand, it's good to write down what the essence of the matter is. (That is called the "axiomatic method.")

We write $\mathbf{x}$ for $x_1, \ldots, x_n$. We assume that there is, for each n, an application function $App(e, \mathbf{x})$, that applies program e to input $\mathbf{x}$.

- The essential point is that App should itself be computable.
- We require the existence of an "interpreter" or "universal function" e such that

$$App(e, x, y) = App(x, y)$$

- A "model of computation" is determined by some set X, together with partial operations App (one for each n) from X^{n+1} to X, obeying the law above.

If you want to think more concretely, take X to be the set of strings over some alphabet Σ, or take X to be the set $\mathbb{N}$ of natural numbers.

Technically, we also need some simpler axioms, but our focus now is just on the main idea.

Modern examples of models of computation

- $App(x, y)$ is the output of the JVM (Java Virtual Machine) emulating program x at input y, if any. (It is undefined if x is not a valid Java program, since then input y is never requested; and it is undefined if $x(y)$ does not terminate.)
- $App(x, y)$ is the output of a C interpreter emulating program x at input y, if any.
- $App(x, y)$ is the output of a LISP interpreter evaluating the S-expression (xy).

Models from the 1930s

The early history of computation theory consisted in the construction of several different models of these axioms. The axioms were only formulated later. In 1930, if they *had* been formulated, it would have been by no means obvious that there exists anything satisfying these axioms. The models that were constructed were

- Church's λ-calculus
- Turing machines
- Gödel's equation calculus

Each of these models has a rich historical thread leading into aspects of modern computer science as well as logic. The important conclusion that has been drawn is that all these models of computation are equivalent, in that the same functions are computable.

Computations

For some of the results about computability, we need more than just a universal machine (or interpreter). We need to know that computation proceed in "steps" or "stages." This is axiomatized by the "T-predicate" and the "U-function." The T-predicate $T(e, x, k)$ means that k encodes a finite number of steps of computation by program e at input x. In that case $U(k)$ is the result of the computation. Formally, we require

$$App(e, x) = y \leftrightarrow \exists k\,(T(e, x, k) \wedge U(k) = y)$$

When we develop specific models of computation, we will also demonstrate that they satisfy this axiom. Kleene introduced the notation T and U for a specific model of computation, but we retain that terminology here in this axiomatic setting.

The Church-Turing thesis

A function f is said to be "computed by" a program p if, whenever p is given input x, the program produces output $f(x)$. It is more or less certain, as we shall eventually show convincingly, that any function that can be computed by any computer program, written in any computer language, for any computer, no matter how powerful, can also be computed by a Turing machine, or equivalently, is computable in the λ-calculus. Sometimes this is called the "Church-Turing thesis". In this form there are today no dissenters to the Church-Turing thesis.

Related philosophical arguments

Sometimes the phrase “Church-Turing thesis’ refers instead to the philosophical claim that any function computable by a human being is computable by a Turing machine. In view of the first thesis, this is equivalent to the claim that any function computable by a human is computable by some computer program. There are some dissenters to this version: these dissenters point to the possible non-mechanical parts of human brain, be those quantum-mechanical features, spiritual features, or “mathematical intuition.” For example, one of these dissenters is the famous physicist Penrose.

Approaching the homework

The skill to be acquired is to recognize intuitively when a function (usually the characteristic function of a set or relation) is computable, and when it is at least not obviously computable. We will try some examples in class, and more are taken up in the homework.