LSS 2018: Computability and Incompleteness

1. Models of Computation

Michael Norrish

Michael.Norrish@data61.csiro.au

Course Outline

Five lectures:

1. Models of Computation. Turing Machines, Recursive Functions.
2. Computability Results. Halting Problem and more.
3. Logic and Computability. Undecidability of FOL.
4. Gödel's First Incompleteness Theorem.
   Representability, Indefinability of Truth.
5. Gödel's Second Incompleteness Theorem.
   Some (non-)implications.

Textbook: *Logic and Computability*, Boolos and Jeffrey.

Outline

1 Introduction

2 Turing Machines

3 Recursive Functions

4 Other Models
  Lambda Calculus

Computation—What's It Good For?

What can we do mechanically?

Is there a way to automatically solve interesting mathematical problems?

Is there a way to automatically solve interesting logical problems?

Alternatively: what logical/mathematical problems can we solve automatically?

- Which ones can't we solve?

Wanted: Formal Definition of "Computer"

Desiderata:

- ▶ Simple. Simple models are easier to reason about.
- ▶ Plausible. We have to believe that the model plausibly captures what it is to "compute".
- ▶ Abstract. Not tied to unnecessarily concrete details.

Outline

Turing Machines

Turing's definition of what it is to be a computer

Alan Turing
(1912–1954)

In the 1930s, a "computer" was a person who performed (arithmetic) calculations.

Turing's model drew inspiration from how people calculate with pen-and-paper:

- ▶ they have an infinite (!) working space;
- ▶ they perceive and operate on small bounded parts of that space.

Turing Machines, Informally

A Turing machine is a

- finite state machine (the "control")
- operating on an infinite tape of "cells"

Turing machine tape-cells

- are either blank or filled with one of a finite set of symbols

A Turing machine state

- sees as input the "current" tape-cell,
- may
    - alter the cell, or
    - move to an adjacent tape-cell
- it also specifies the next state (based on the input)

Making Turing Machines Compute Things

Put the machine down on a tape that has been primed with an encoding of the input.

If the machine stops and is pointing to a recognisable result on the tape, that's the answer.

Thus, the machine can fail in two ways:
- ▶ stops with bogus tape
- ▶ never stops

Turing machines compute partial functions.

Turing Machines and Numbers

We can work exclusively with one symbol, the "blob" (or 1).
    (Remember we also have blanks (or 0) on the tape).

The number n is represented on the tape as n blobs.

TMs can compute (partial) functions $\mathbb{N} \to \mathbb{N}$.

$\mathrm{TM}(n) = m$ when:

1. Put n blobs on tape.
2. Run TM, starting it pointing to left-most blob.
3. TM stops at leftmost position of m blobs.

(Handles 0 as input and output.)

Already, Uncomputability Rears Its Ugly Head

Turing machines are enumerable (finite controls!)

Functions $\mathbb{N} \to \mathbb{N}$ are uncountable.

Therefore, there must be functions that no Turing Machine can compute.

Surely it's not reasonable to allow infinite programs...

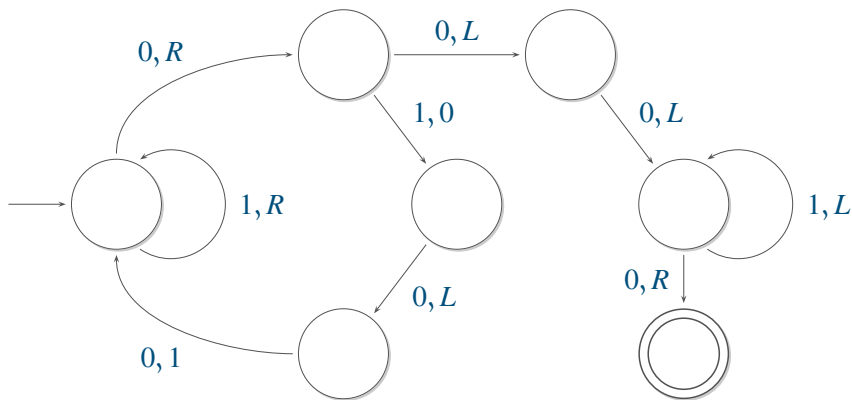# So What *Can* Turing Machines Compute?

Numeric calculations: addition, subtraction, multiplication, pairing. . .
(example to follow)

Numeric tests: "is zero?", "is even?", "is prime?". . .

Operations on other data types: encoding, decoding . . .

Arrow label x, A means

"if you see an x, do action A (and follow arrow to next state)"

Programming Turing Machines

Turing Machines are the ultimate in low-level computing devices.

Programming them can be appealing as a puzzle.

But it's hard to develop much modularity.

Somewhat easier if you have a simulator to play with.

Cue demo. . .

Programming Turing Machines

Turing Machines are the ultimate in low-level computing devices.

Programming them can be appealing as a puzzle.

But it's hard to develop much modularity.

Somewhat easier if you have a simulator to play with.

Cue demo...

Credit to http://ironphoenix.org/tril/tm/
and Suzanne Skinner

Deciding Sets with Turing Machines

Turing Machines can implement "tests" on arguments.

$$\left(\begin{array}{l} \text{Put } n \text{ blobs on tape, machine whirs away, leaves} \\ \text{either one blob ("yes") or no blobs behind ("no").} \end{array} \right)$$

This can be seen as the action of deciding set membership.

A set is decidable if a machine exists that always correctly says "yes" or "no" of possible elements.

E.g., the set of prime numbers is decidable.

Unary Functions as Enumerations

A function f of type $\mathbb{N} \to \mathbb{N}$ can be seen as an enumeration.

The enumerated set is the range of the function.

$$\left(\begin{array}{l} \text{Can insist (or not) that enumeration gives values to} \\ \text{successive values: if } f(n) \text{ is defined, then so too must} \\ \text{be } f(m) \text{ for all } m < n. \\ \text{Can insist (or not), that there are no repeats (that is, } f \\ \text{is injective).} \end{array}\right)$$

Every decidable set can be enumerated:

Given input n, run decision machine on n.
If it says "yes", return n; otherwise go into infinite loop.

Enumerations Give Semi-Decision Procedures

A semi-decision procedure for S is a machine that will correctly say "yes" of an input n iff $n \in S$.

If we have an enumeration machine E, and want to test n, run enumeration on successive values ($0, 1 \ldots$) until we find an i such that $E(i) = n$.

(Fast and Loose Alert: I am assuming E is a "strong" enumerator that has no gaps in its domain.)

Semi-Decision Procedures Give Enumerations

This should be familiar:

Given input n, run decision machine on n.
If it says "yes", return n; otherwise go into infinite loop.

The difference is that "otherwise" now includes the failure of the machine to terminate.

(Fast and Loose Alert: I am producing an enumeration that probably does have gaps in its domain.)

Gappy Enumerations Can Be Made Non-Gappy

Say $E(n)$ is undefined.

Want to be able to scan ahead for an $m > n$ where $E(m)$ is OK.
(If there is none such, it's OK for us to loop.)

But How?

Can't just run $E(n + 1)$ and wait for its answer.

Gappy Enumerations Can Be Made Non-Gappy

Say $E(n)$ is undefined.

Want to be able to scan ahead for an $m > n$ where $E(m)$ is OK.
(If there is none such, it's OK for us to loop.)

But How?

Can't just run $E(n+1)$ and wait for its answer.

Need to be able to run a set of machines in parallel for fixed number of steps.

This is called dovetailing.

The Universal Machines

There exists a universal machine U that when given as input (i, n) can simulate the behaviour of machine i on input n.

> That is, if $\varphi_i(n)$ terminates with output x, so too does $U(i, n)$.
> If $\varphi_i(n)$ loops, so too does $U(i, n)$.

There also exists a stepping function that takes a machine state and runs it for a specified number of steps.

(These facts are both extremely cool and extremely ... obvious.)

One proof is by construction...

Is There Anything These Machines Can't Do?

Church's Thesis:
  Turing Machines can compute anything that any reasonable model of computation can compute.

No proof, but true for all models devised so far.

Outline

And Now For Something Completely Different

Turing Machines are so concrete!

Recursive Functions give us a way to capture computable functions much more mathematically.

Basic Method:

- ▶ Construct a family of functions all of which are "obviously OK"
- ▶ Provide methods for making new functions from ones already "in the family".

First Family Group: the Primitive Recursive Functions

Functions are generally of type $(\mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}) \to \mathbb{N}$

Base cases:

- The zero function: $z(n) = 0$
- The successor function: $s(n) = n + 1$
- Projection functions: $p_{i,n}(x_1, \ldots, x_n) = x_i$

Building new from old:

- Composition: if each g_i takes n arguments, and f takes m, then $f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$ is primitive recursive.

  Write as $\text{Cn}[f, g_1, \ldots, g_m]$ (it takes n arguments)

Can Already Define Some Good Stuff

Constant functions:

$$Three = \mathsf{Cn}[s, \mathsf{Cn}[s, \mathsf{Cn}[s, z]]]$$

$$Three(x) = s(s(s(z(x))))$$
$$= 3$$

Adding a constant:

$$AddTwo = \mathsf{Cn}[s, s]$$

$$AddTwo(x) = s(s(x))$$

Can Already Define Some Good Stuff

Constant functions:

$$\begin{aligned} \textit{Three} &= \mathsf{Cn}[s, \mathsf{Cn}[s, \mathsf{Cn}[s, z]]] \\ \textit{Three}(x) &= s(s(s(z(x)))) \\ &= 3 \end{aligned}$$

Adding a constant:

$$\begin{aligned} \textit{AddTwo} &= \mathsf{Cn}[s, s] \\ \textit{AddTwo}(x) &= s(s(x)) \end{aligned}$$

None of these functions examine their arguments.

(Primitive) Recursion Makes It Even Better

If g (the "base case") takes n arguments,

and h (the "recursive case") takes $n + 2$ arguments,

then $\Pr[g, h]$ is the function f of $n + 1$ arguments such that

$$f(x_1, \ldots, x_n, 0) = g(x_1, \ldots, x_n)$$
$$f(x_1, \ldots, x_n, m + 1) = h(x_1, \ldots, x_n, m, f(x_1, \ldots, x_n, m))$$

Allow $n = 0$, in which case g can just be a number.

Simple Arithmetic is Primitive Recursive

Addition (*Plus*) is $\Pr[p_{1,1}, \mathrm{Cn}[s, p_{3,3}]]$:

$$\begin{aligned}
\Pr[p_{1,1}, \mathrm{Cn}[s, p_{3,3}]](x, 0) &= p_{1,1}(x) \\
&= x
\end{aligned}$$

$$\begin{aligned}
\Pr[p_{1,1}, \mathrm{Cn}[s, p_{3,3}]](x, y+1) &= \mathrm{Cn}[s, p_{3,3}](x, y, Plus(x, y)) \\
&= s(p_{3,3}(x, y, Plus(x, y))) \\
&= s(Plus(x, y))
\end{aligned}$$

Simple Arithmetic is Primitive Recursive

Addition (*Plus*) is $\text{Pr}[p_{1,1}, \text{Cn}[s, p_{3,3}]]$:

$$\begin{aligned}
\text{Pr}[p_{1,1}, \text{Cn}[s, p_{3,3}]](x, 0) &= p_{1,1}(x) \\
&= x \\
\text{Pr}[p_{1,1}, \text{Cn}[s, p_{3,3}]](x, y+1) &= \text{Cn}[s, p_{3,3}](x, y, Plus(x, y)) \\
&= s(p_{3,3}(x, y, Plus(x, y))) \\
&= s(Plus(x, y))
\end{aligned}$$

Multiplication (*Mult*) is $\text{Pr}[z, \text{Cn}[Plus, p_{1,3}, p_{3,3}]]$:

$$\begin{aligned}
Mult(x, y+1) &= \text{Cn}[Plus, p_{1,3}, p_{3,3}](x, y, Mult(x, y)) \\
&= Plus(p_{1,3}(x, y, Mult(x, y)), p_{3,3}(x, y, Mult(x, y))) \\
&= Plus(x, Mult(x, y))
\end{aligned}$$

Some Properties of Primitive Recursive Functions

Enumerable: Each prim. rec. function is captured by a finite string.

Total: Only interesting case to consider is recursion; all such must be total by induction on the argument that is "recursed".

Primitive Recursion Does Not Capture Computability

Consider the famous Ackermann function:

$$A(0, m) = m + 1$$
$$A(n + 1, 0) = A(n, 1)$$
$$A(n + 1, m + 1) = A(n, A(n + 1, m))$$

- Must be computable
- Is total (well-founded induction on lexicographic ordering of arguments)
- Grows very quickly

In fact, for every prim. rec. function f, there is a J such that, for all possible arguments $x_1, \ldots, x_k$

$$f(x_1, \ldots, x_k) < A(J, \Sigma x_i)$$

What Are We Missing?

Ackermann's function cannot be captured by the primitive recursive functions.

What can we add so that it can be computed?

Introducing the Recursive Functions

Keep the formation rules for primitive recursive functions.

Add the following:

> *If f is a recursive function of $n + 1$ arguments,*
> *then $\text{Mn}[f]$ is a function of n arguments $(x_1, \ldots, x_n)$ that*
> *returns the least m such that $f(m, x_1, \ldots, x_n) = 0$.*

Suddenly we're no longer in the land of total functions!

Computationally, can view $\text{Mn}[f]$ as a potentially unbounded search.

Is This Really Enough?

It may not be obvious that adding Mn is sufficient.

- ▶ Just adding something that may not terminate is not a clear improvement.

On the other hand, it should be obvious that Mn is not implementable with primitive recursion.

Exercise: Show you can implement Ackermann's function by providing a recursive function to calculate it.

Recursive Functions Are Equivalent to Turing Machines

Each model can emulate the other.

Turing Machines can implement the recursive functions.

- ▶ I hope this is obvious

The recursive functions can implement Turing Machines.

- ▶ Perhaps not so obvious; in the next lecture!

Outline

Register Machines

Called abacus machines in Boolos and Jeffery.

Finite state machines with access to arbitrary number of "registers" (fixed per program).

Registers can contain arbitrarily big numbers.

Programs can add one, subtract one and branch.

Slightly more realistic "hardware".

The Lambda Calculus

The world's simplest programming language:

$$M ::= \ v \mid M_1 M_2 \mid (\lambda v. M)$$

Behaviour captured by one rule:

$$(\lambda v. M) N \rightarrow_\beta M[v := N]$$

Apply this rule (β-reduction) wherever you can within a term; rename bound variables to avoid capture.

The Lambda Calculus Does It All

Can represent numbers.

Can go into infinite loops:

$$(\lambda x.\, x\, x)(\lambda x.\, x\, x) \rightarrow_\beta (\lambda x.\, x\, x)(\lambda x.\, x\, x)$$

Can implement arbitrary recursion: the famous Y combinator.

Very expressive: much the easiest model to show capable of emulating the others.

Summary

- Turing Machines. The "hardware guy's model of computation".

  Also, important notions:
  - Decidability
  - Enumerability ($=$ semi-decidability)

- Recursive functions. The "mathematician's model of computation".
  - Starting with primitive recursive functions.

- The Lambda Calculus (briefly)