LSS 2018: Computability and Incompleteness

2. Computability Results

Michael Norrish
Michael.Norrish@data61.csiro.au

Outline

Last Time…

Discussed two important computational models:

Turing Machines: Finite state machines with access to an infinite tape ("scratch space")

Recursive Functions: An inductive characterisation of functions (from $\mathbb{N}^n \to \mathbb{N}$) that are plausibly computable

Important Vocabulary

Decidable A set S is decidable if a computable "thing" correctly categorises putative element e.
If $e \in S$, then answer is "yes".
If $e \notin S$, then answer is "no".

Such sets are also known as recursive.

Enumerable A set is enumerable if it is the range of a computable function.

Usual phrase is actually recursively enumerable, or r.e.

Semi-decidable A set S is semi-decidable if a computable "thing" correctly says "yes" of an e, iff $e \in S$.
Implicitly, if $e \notin S$, thing does not terminate.

Semi-decidable is equivalent to r.e.

Some Related Logic-based Vocabulary

Assume we have a machine M that says “yes” of certain formulas ϕ.

We hope M is identifying theorems of a logical system.

M is:

Sound	If, when M says “yes”, ϕ really is a theorem.
Complete	If ϕ is a theorem, then M will say “yes” of it.

A sound and complete procedure (*e.g.*, an inference system) is a semi-decision procedure.

Outline

Turing Machines Implement the Recursive Functions

Earlier claimed this was "obvious".

Certainly, zero (z), successor (s) and projection ($p_{i,j}$) should be clear.

Composition, primitive recursion and "least" are trickier:

- They need to stage calls to sub-functions (perhaps many, many times)
- Need to guarantee that sub-machines are "well-behaved"

Well-Behaved Turing Machines

Argue inductively that it is possible to construct implementations of computable functions that

- Return results cleanly (no extraneous junk off to the right)
- Do not alter tape to the left of machine's starting position

Neither condition is generally true of Turing Machines.

But we will make them true of the TMs constructed to implement computable functions.

Computable Function Turing Machines (Sketches)

$\mathsf{Cn}[f, g_1, \ldots, g_n]$	Set up n copies of arguments. Run g_n's machine on rightmost. Shift answer left (past all $g_{i<n}$). Move to arguments copy $n-1$; apply g_{n-1}. Etc. Call f on the n results.
$\mathsf{Pr}[f, g]$	Behaviour on recursive argument n is to make n copies of calls to g extending right on tape. After call m is made, result is in place to take part in call $m+1$.
$\mathsf{Mn}[f]$	Record n to the left, have copy of it and args set up to the right. If call with n fails, can easily repeat with $n+1$.

Recursive Functions Implement Turing Machines

The problem is that recursive functions manipulate numbers, and Turing Machines are not numbers.

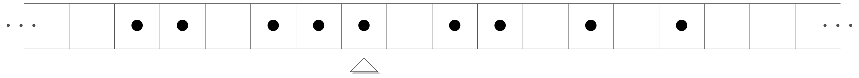
How to Encode a Turing Machine as a Number?

More accurately:

How to Encode a TM-state as a Number,
and the Transition Behaviour as a Function?

Encoding Tapes as Numbers

Assume that we are only using blobs and blanks.

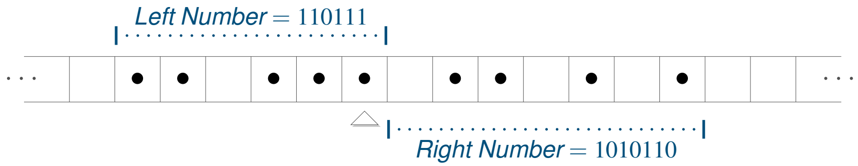


Tape portions to the left and right of the r/w head can be seen as a binary number, with least significant bit nearest to head.

- Tapes must be all blank beyond certain points to left and right.

Encoding Tapes as Numbers

Assume that we are only using blobs and blanks.



Tape portions to the left and right of the r/w head can be seen as a binary number, with least significant bit nearest to head.

- Tapes must be all blank beyond certain points to left and right.

Above is encoded as $\langle 103, 166 \rangle$.

Encoding the Turing Machine Control

The control can be seen as two functions

- The state-to-state transition: $\mathit{State} \times \mathit{Input} \longrightarrow \mathit{State}$; and
- The state-to-action function: $\mathit{State} \times \mathit{Input} \longrightarrow \mathit{Action}$

All the types above are finite, so the functions are finite,
and can be defined (primitive recursively) by cases.
(Use q for state-to-state, and a for action functions.)

Reserve an additional state number to represent halted.
In the state-to-state function move to this state when no other behaviour is called for.

Functions by Finite Cases are Primitive Recursive

They're just a chain of nested ifs…

- "Nested" $=$ "use Cn"

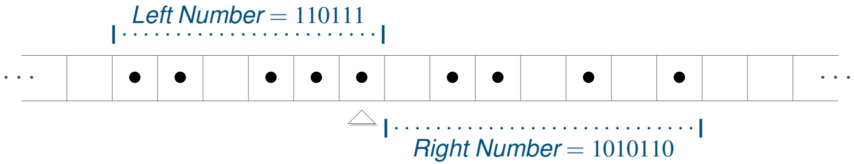
How do you do if?

- Predicates return 1 for true, and 0 for false.
- $(\text{if } P(x) \text{ then } f(x) \text{ else } g(x)) = (P(x) \times f(x) + (1 - P(x)) \times g(x))$

Can do $m = n$ with $m - n = 0$.

- $x = 0$ is easy.
- Subtraction is recursed predecessor.
- Predecessor is easy.

Describe Actions' Effect on Tape



- Moving Left: divides Left Number by two; multiplies Right Number by two (and adds one if Left Number was odd).
  Moving right analogous.
- Writing a Blank: Subtracts one from Left Number if it was odd.
- Writing a Blob: Adds one to Left Number if it was even.

Basic Strategy for One Step of Turing Machine

- State is a triple of $\langle\text{Left Number}, \text{Right Number}, \text{Current State}\rangle$.
- $\mathit{Input} = \text{Left Number} \bmod 2$
- Action is $a(\text{Current State}, \mathit{Input})$.
- New Current State is $q(\text{Current State}, \mathit{Input})$
- New Left Number, Right Number given by function from previous slide (applied to Action).

Computable Tuples

Hold on!—Computable functions are $\mathbb{N}^n \to \mathbb{N}$.

But our “one step” function returns three new numbers (left, right, state).

Computable Tuples

Hold on!—Computable functions are $\mathbb{N}^n \to \mathbb{N}$.

But our "one step" function returns three new numbers (left, right, state).

Luckily, there are computable bijections between $\mathbb{N}^2$ and $\mathbb{N}$.
(They're even primitive recursive.)

So we can encode three numbers into one.

Definition of "one step" function becomes littered with applications of E, D_1, D_2 and D_3.

- properties include things like $D_2(E(x, y, z)) = y$
- Exercise: implement E, D_i.

Putting It All Together

Imagine Turing Machine computes binary function (on (x, y)).

Define (primitive recursion)

$$\begin{aligned} t(0, x, y) &= \langle \text{initial state for } (x, y) \rangle \\ t(n+1, x, y) &= \text{one-step}(t(n, x, y)) \end{aligned}$$

Ensure that the special halt state is 0.

Then $h = \mathsf{Mn}[\mathsf{Cn}[D_3, t]]$ computes the number of steps required to get the machine to halt.

And $D_2(t(h(x, y), x, y))$ is the TM's result.

Outline

Setting the Scene

We already know that:

- We can turn TMs into numbers
- TMs can simulate TMs given their number.

  Either

  - Run to completion
  - Run for fixed number of steps

Various forms of intensional analysis are possible.

Alternatively: we can look inside a machine and determine properties of its construction.

An Important Extensional Property

$\cdots$ Will my program terminate? $\cdots$

Byron Cook of Microsoft Research has a very cool program called Terminator that performs termination analysis of device driver code.

- Computers that loop inside device drivers typically need to be rebooted. . .

We could implement a sound and complete algorithm for solving part of this problem.

Sound and Complete Termination Method

“Theorems” are of the form:

Machine M with input n terminates.

Method: run Universal Machine on that input.

Soundness and completeness follows from properties of UM.

This is a semi-decision procedure.

This is *not* how Terminator works.

The Halting Problem

Let φ_i be the i-th Turing Machine.

The set $\{ i \mid \varphi_i(i) \text{ halts} \}$ is not recursive.

- Corollary: the general problem (machine i on input n) is not recursive either.
- Nor indeed, is the problem of determining if machine i halts on all inputs.

Proof is due to Turing.

- By contradiction and a diagonalisation argument

Proof of the Halting Problem

Imagine M decides the halting problem set ($\{\, i \mid \varphi_i(i) \text{ halts}\}$).

Let M's index in the enumeration of all machines be i.

Let N be the machine that takes a single input n and calls $M(n)$.

- If $M(n)$ says "yes", then N goes into a loop.
- If $M(n)$ says "no", N stops with answer 0.

Let j be N's index. Consider $N(j)$'s behaviour.

- Noting that $N(j) = \varphi_j(j)$

Uncomputability Has Real Consequences!

Wouldn't it be great to be able to mechanically determine if a program will terminate?

But We Can't!!!

We already know that the computable functions cannot be all functions.

But now we also know that the uncomputable functions include some that are interesting, relevant, useful. . .

Outline

The Set K and Reductions To It

Let $f(x)\downarrow$ mean that $f(x)$ is defined/halts.

Let $K = \{\, i \mid \varphi_i(i)\downarrow \,\}$

Halting Problem proves that K is undecidable.

Standard Undecidability Proof Strategy: show that if my problem was decidable, then K would be too.

Extensional Properties

Take extensional property to mean

> *A property that depends only on the behaviour of the machine, not its internal construction.*

Alternatively, let us treat our machines as black boxes.

Note that “halting on one’s own index” is not strictly extensional—we have to look inside a machine to figure out its index.

Extensional Properties and Index Sets

Let P be a predicate on partial functions ($\mathbb{N} \to \mathbb{N}$).

Let indexes$(P) = \{ i \mid P(\varphi_i) \}$

If indexes(P) is not empty, it will be infinite:

> there are an infinite number of ways to implement one mathematical function.

Examples of P (predicate on function f):

- f has finite domain;
- f terminates on even numbers;
- for all x, if $f(x)\downarrow$, then $f(x) < 100$

Rice's Theorem

For all possible P, if indexes(P) is recursive, then it is either empty or the universal set.

Alternatively, there are only two decidable extensional properties, and they're both trivial.

Proof of Rice's Theorem

Assume indexes(P) is recursive, but that it is neither empty nor the universal set. A contradiction will follow.

First: is the index of the machine (Ω) that never halts in indexes(P)? If so, work with complement of indexes(P) instead.

(Complement is recursive, non-empty and non-universal too.)

Second: Let a be an element of our decidable set, and M be the machine that decides the set.
(We now know $M(a) = 1$ and $M(\Omega) = 0$ for example.)

Proof of Rice's Theorem Continued

Let f be the machine that when given a number i returns the index of h where

$h(x)$ first runs $\varphi_i(i)$, and then (if that terminates), runs $\varphi_a(x)$.

Note that $h(x)$ is "extensionally" the same as

- Ω if $\varphi_i(i)$ loops; and
- φ_a if $\varphi_i(i)$ terminates

$M(a)$ returns true, so $M(f(i))$ returns true if $\varphi_i(i)$ halts.

$M(\Omega)$ returns false, so $M(f(i))$ returns false if $\varphi_i(i)$ loops.

We've solved the Halting Problem!

Outline

The “Is a Function Total?” Problem

A total function is one that terminates on all inputs.

$$\mathsf{total}(f) = \forall n.\, f(n)\downarrow$$

Consider the set of indices of total (computable) functions

$$\{\, i \mid \mathsf{total}(\phi_i)\,\}$$

This set is not recursive (very directly by Halting Problem).

Total Functions Continued

There are uncountably many total functions, so
we can't enumerate all of them.

However, we can't even enumerate the computable total functions!

If we could, we'd have a computable machine M to do it.

And then $f = (\lambda n.\ \varphi_{M(n)}(n) + 1)$ would be computable and total too.

So, f would have an index i, and there would be a j such that $M(j) = i$

And then $f(j) = f(j) + 1$

Another “Not Even Enumerable” Set

If a set and its complement are enumerable, then they must both be recursive.

- Remembering that “enumerable” also means “is semi-decidable”.

We know K is recursively enumerable.
We know K is not recursive.

Therefore $\overline{K}$ is not recursively enumerable.

Degrees of Unsolvability

There are various formal descriptions of the ways in which completely unsolvable problems are more or less difficult than each other.

For example, the arithmetical hierarchy characterises sets of numbers by the logical formulas needed to describe them.

The more quantifier alternations, the worse.

- For example, "is a total function" is $\forall\exists$
- $\overline{K}$ is just $\forall$

Summary

One piece of good news:

- The computational models (recursive functions, Turing Machines) are equivalent

And the rest, all bad:

- The Halting Problem is undecidable
- Most of everything is undecidable (Rice's Theorem)
- Some problems are more undecidable than others