Computable functions

Much of what we say henceforth could be about any reasonable ‘domain’: graphs, integers, rationals, trees. We will use $\mathbb{N}$ as the canonical domain, and rely on the (usually obvious) fact that any reasonable domain can be encoded into $\mathbb{N}$.

A (total) function $f : \mathbb{N} \rightarrow \mathbb{N}$ is **computable** if there is an RM/TM which computes $f$ (e.g. by taking $x$ in $R_0$ and leaving $f(x)$ in $R_0$) and always terminates.

Note that predicates (e.g. ‘does the machine halt?’) are functions, viewing 0 as ‘no’ and 1 as ‘yes’.

You will also see the term ‘recursive function’. This is confusing, because it doesn’t mean ‘recursive’ in the sense you know, though there is a strong connection. This terminology is slowly going out of use. We may come to it . . .
Proving (un)decidability

Is the halting problem the only natural (?) undecidable problem?

How can we show that a ‘real’ problem is decidable or undecidable?

To show a problem decidable: write a program to solve it, prove the program terminates. Alternatively, translate it to a problem you already know how to decide.

Why doesn’t this contradict the halting theorem?

To show a problem undecidable: prove that if you could solve it, you could also solve the halting problem.
Decision problems and oracles

A decision problem is a set $D$ (domain) and a subset $Q$ (query) of $D$. The problem is ‘is $d \in Q$?’, or ‘is $Q(d)$ 0 or 1?’. The problem is computable or decidable iff the predicate $Q$ is computable. For example:

- the domain $\mathbb{N}$ and the subset $Primes$;
- the domain $RM$ of register machine (encodings), and the subset $H$ that halt.

Given a decision problem $(D, Q)$, an oracle for $Q$ is a ‘magic’ RM instruction $\text{ORACLE}_Q(i)$ which assumes that $R_i$ contains (an encoding of) $d \in D$, and sets $R_i$ to contain $Q(d)$.

If $Q$ is itself decidable, $\text{ORACLE}_Q(i)$ can be replaced by a call to an RM which computes $Q$ – thus a decidable oracle adds nothing.
Reductions

Reductions are a key technique. Simple concept: turn one problem into another. But there are subtleties.

A Turing transducer is an RM that takes a instance \( d \) of a problem \((D, Q)\) in \( R_0 \) and halts with an instance \( d' = f(d) \) of \((D', Q')\) in \( R_0 \). (Thus \( f \) is a computable function \( D \to D' \).)

A mapping reduction or many–one reduction from \( Q \) to \( Q' \) is a Turing transducer \( f \) as above such that \( d \in Q \) iff \( f(d) \in Q' \).

Equivalently, an m-reduction is a Turing transducer which after putting \( f(d) \) in \( R_0 \) then calls \( ORACLE_Q(0) \) and halts.

**NOTE** that there is no post-processing of the oracle’s answer!

Is this intuitively reasonable?

If you allow the transducer to call the oracle freely, you get a Turing reduction. These are also useful; but they’re more powerful, so don’t make such fine distinctions of computing power.
Showing undecidability

Reductions are the key (really the only) tool for showing undecidability, thus:

- Given \((D, Q)\), construct a reduction \(\text{Red}(H, Q)\) **from** \((RM/TM, H)\) **to** \((D, Q)\).

- Suppose \(Q\) is decidable. Then given \(M \in RM\), feed it to \(\text{Red}(H, Q)\) to decide \(M \in H\). But if \(Q\) is decidable, we can replace the oracle, and \(\text{Red}(H, Q)\) is just an ordinary RM. Hence \(H\) is decidable – contradiction. So \(Q\) must be undecidable.

Constructing \(\text{Red}\) is usually either very easy, or rather long and difficult!

For a relatively simple example of a long and complicated reduction, see https://hal.archives-ouvertes.fr/hal-00204625v2, a simplified proof by Nicolas Ollinger of the famous ‘tiling problem’ due to Hao Wang.
The Uniform Halting Problem

A simple example:

The **Uniform Halting Problem** (**UH**) asks, given a machine \( M \), does \( M \) halt on *all* inputs?

‘Clearly’ at least as hard as \( H \), so must be undecidable. Proof:

- We need an m-reduction **from \( H \) to \( UH \)**:
  - given machine \( M \), input \( R \), build a machine \( M' \) which *ignores* its input and overwrites it with \( R \), then behaves as \( M \).
  - Then \( M' \) halts on any input iff \( M \) halts on \( R \).

*Check that this really is a transducer.*
The Looping Problem

Let $L$ be the subset of RMs (or TMs) that go into an infinite loop. Show that $L$ is undecidable.

Since $L$ is just the complement of $H$, this seems equally easy: just flip the answer. But m-reductions can’t post-process the answer.

Can you devise an m-reduction from $H$ to $L$?

No! You can’t.
When ‘yes’ is easier than ‘no’

The halting problem is not symmetrical:

- if $M \in H$, then we **can** determine this: run $M$, and when it halts, say ‘yes’;
- if $M \notin H$, then we **can’t** determine this: run $M$, but it never stops, and we can never say ‘no’.

Such problems are called **semi-decidable**.

The problem $L$ is the opposite: we can determine ‘no’, but not ‘yes’. It is called **co-semi-decidable**.

There are two ways of exploiting this asymmetry – we’ll see them quickly now, and in more detail in the second section of the course.
Enumeration

It’s convenient to talk of RMs that ‘output an infinite list’ – as in
\[ i = 0; \text{ while ( true ) } \{ \text{ print i++; } \} \] – which can be formalized in several ways. (Exercise: think of a few.)

Suppose \((\mathbb{N}, Q)\) is decidable. Then we can write a machine which outputs the set \(Q\):

- for each \( n \in \{0, 1, 2, \ldots \} \), compute \( Q(n) \) and output \( n \) iff \( Q(n) \);
- thus every \( n \in Q \) will eventually be output.

This is an enumeration of \( Q \), and we say that \( Q \) is computably enumerable (c.e. for short).

\( H \) is not decidable – but can we still enumerate it?
Enumeration by interleaving

Observe that the set of valid register machine encodings is decidable: given \( n \), we can check whether \( n \) is \( \langle M \rangle \) for a syntactically valid machine \( M \).

Therefore we can enumerate \( \langle M_0 \rangle, \langle M_1 \rangle, \ldots \)

Now consider the following pseudo-code:

```plaintext
machineList := \langle \rangle # sequence storing a list of machine (code)s
for (i:=0; true; i++):
  add \( \langle M_i \rangle \) to machineList
  foreach \( \langle M \rangle \) in machineList:
    run \( M \) for one step and update in machineList
    if \( M \) has halted:
      output \( \langle M \rangle \)
      delete \( \langle M \rangle \) from machineList
```

This program outputs \( H \): every halting machine is eventually output.

\( H \) is computably enumerable.
Semi-decidability and enumerability

The interleaving technique lets us see that

- Any semi-decidable problem is also computably enumerable

**Exercise:** Show that the converse is true:

- Any c.e. problem is semi-decidable.

Now we’ve shown that semi-decidability is the same as c.e.-ness. Note that if \((D, Q)\) is semi-decidable, then \((D, D \setminus Q)\) is co-semi-decidable.
Suppose that \((D, Q)\) is both semi-decidable and co-semi-decidable.

- So \(Q\) and \(D \setminus Q\) are both semi-decidable.
- So run a semi-deciding machine for \(Q\) in parallel with a semi-decider for \(D \setminus Q\), using interleaving.
- Whatever instance \(d\) we look at, one of the two will halt – so stop then.

So \((D, Q)\) is decidable.

Now we know that \(L\) cannot be semi-decidable, as it’s the complement of semi-decidable \(H\).

**Exercise:** When I say the complement of \(L\) is \(H\), what is the domain \(D\)? Do I need to take care?
Uniform Halting again

We showed, easily, that $UH$ was undecidable. Is it semi-decidable?

We used reductions from $H$ to show un-decidability. Do reductions from $L$ show un-semi-decidability? Yes!

Suppose $f$ reduces $Y$ to $X$, and suppose that $X$ is semi-decidable by machine $M_X$. Then since $Y(y) = X(f(y))$, if $Y(y) = 1$ we can translate $y$ to $f(y)$, run $M_X$, and get 1. If $Y(y) = 0$, then $M_X$ applied to $f(y)$ either gives 0, or doesn’t halt.

So is $X$ is semi-decidable, so is $Y$; or if $Y$ is not semi-decidable, then $X$ can’t be.
Reducing $L$ to $UH$

We want a transducer $f(M, R)$ such that $f(M)$ halts on all inputs iff $M$ loops on $R$. Not obvious . . . we could ignore the inputs to $f(M)$, but how do we make it stop?

By a clever trick: $f(M)$ will measure how long $M$ runs for, with a timeout as its input.

- Let $M' = f(M, R)$ be a machine which takes a number $n$ as input, and then simulates $M$ on $R$ for (at most) $n$ steps.
- If $M$ halts before $n$ steps, then $M'$ goes into a loop;
- while if $M$ hasn’t halted, $M'$ just halts.
- Hence if $M$ loops, $M'$ halts on all inputs; and if $M$ halts, then $M'$ loops on all sufficiently large inputs.
So $UH$ is not . . .

The reduction from $L$ shows that $UH$ is not semi-decidable.
Is $UH$ co-semi-decidable (like $L$)?
No! The (trivial) reduction from $H$ to $UH$ also shows that $UH$ is not co-semi-decidable.

$H$ has from ‘$\exists n. M$ halts after $n$’; $L$ has form $\forall n. M$ doesn’t halt after $n$; $UH$ has form ‘$\forall R. \exists n. M$ halts after $n$ on $R$’. Really the diagram should be . . .
Harder and harder

Suppose we consider $RM^H$, the class of register machines with an oracle for $H$.

We can easily adapt our universal machine to produce a universal $^H$ machine for this class.

So we can apply diagonalization, and show that:

- There is no $RM^H$ machine that computes the halting problem for $RM^H$

We can keep doing this, and generate harder and harder problems – the arithmetical hierarchy. Sadly, we don’t have time to explore this fascinating topic.