Functions, Oracles, and Reductions

Agenda for Today

Today, we’ll be talking about functions as they relate to computability

- Computable functions
- Reductions
    - Oracle Reductions (also known as Turing Reductions)
    - Computable Many-one Reductions

Review U_{TM}, A_{TM}, and $A_{TM} \notin R$

- U_{TM} is a *universal Turing machine*. It takes the description of another Turing machine *M* and an input *x* to that Turing machine and simulates *M* on *x*.
- A_{TM} is the language of U_{TM}. Remember, the language of a TM is the set of strings that TM accepts, so

$$A_{TM}=\{\langle M, x\rangle | M \ is \ a \ TM \text{ which accepts input } x\}$$

Theorem: $A_{TM} \notin R$

Proof: Suppose $A_{TM} \in R$. Then there is some Turing machine *A* which decides A_{TM}. Consider the following TM *D*, which must exist if *A* exists:

On input $\langle M \rangle$, *D* uses *A* as a subroutine to decide if $\langle M, \langle M \rangle \rangle \in A_{TM}$, that is, if M accepts the input which is M's description.

If $\left\langle M, \langle M \rangle \right\rangle \in A_{TM}$, that is, if the subroutine *A* accepts, then *D* rejects.

Otherwise, when the subroutine *A* rejects, *D* accepts.

Observe that since *A* halts on all inputs, *D* halts on all inputs – it only takes one more step after running *A*. In particular, *D* halts on input $\langle D \rangle$.

Since *D* halts on $\langle D \rangle$, it either accepts or rejects.

Suppose *D* accepts $\langle D \rangle$. Then $\left\langle D, \langle D \rangle \right\rangle \in A_{TM}$. However, by our design of *D*, this implies that *D* rejects. This is a contradiction.

Suppose *D* instead rejects $\langle D \rangle$. Then $\left\langle D, \langle D \rangle \right\rangle \notin A_{TM}$. However, by our design of *D*, this implies that *D* accepts. This is again a contradiction.

Since in each case our assumption that *A* exists has lead to a contradiction, we must conclude that our assumption is false and that $A_{TM} \notin R$. □

What is a computable function?

- There are many equivalent definitions!
- Call the *output* of a TM *M* on input x whatever is written on *M*'s tape when it halts. We will denote this as *M(x)*.
- We say $f: \Sigma^* \to \Sigma^*$ is computed by a TM *M* if whenever given the input *x*, *M* halts with *f(x)* written on its tape.
- Basically, *f* is computable if there is an algorithm which, given the input to *f*, finds the corresponding output.

Examples of Computable Functions

- $f(\langle n \rangle) = \langle 2^n \rangle$
- $g(\langle n \rangle) = \langle \lceil \log(n) \rceil \rangle$
- $h(\langle M, k \rangle) = x$ where x is the content of TM *M*'s tape after computing for *k* steps.
- $i(\langle D, x \rangle) = 1$ if D is a DFA and x in L(D) or 0 otherwise.

For shorthand, we will say a function $f: \mathbb{N} \rightarrow \mathbb{N}$ is computable if $g: \Sigma^* \rightarrow \Sigma^*$ is computable and $g(\langle n \rangle) = \langle f(n) \rangle$. For example, the function n^2 is computable.

Relationship between functions and languages

- For each language *L*, we say *f* is an *indicator function* for *L* if

$$f(x) = 1 \text{ if } x \in L \text{ and } f(x) = 0 \text{ otherwise}$$

Note this gives us some examples of functions which are not computable!

Busy Beaver Function

- The Busy Beaver function $BB: \mathbb{N} \to \mathbb{N}$ is defined as the largest number a TM with *n* states can output on input ε (remember the TM has to halt).
- Exercise: Why isn't this function Computable?
- Note from when we tried to prove this at the end of class – if you change the definition to be:

> The Busy Beaver function $BB: \mathbb{N} \to \mathbb{N}$ is defined as the largest number a TM with *n* states can output on input $\langle n \rangle$ (remember the TM has to halt).

Then the proof we showed would work very elegantly! As an exercise, you should show that we can still use the same proof technique modify it a bit to account for this.

Oracles

- An *oracle Turing machine* is like a Turing machine which has a special subroutine called an *oracle* which decides some language for the TM. Oracles don't have to be Turing machines – they can decide any language!
- Think about oracle Turing machines as algorithms which get a special subroutine which tells you things you couldn't normally ever compute with a Turing machine.
- For example, with an oracle for A_{TM}, I can decide A_{TM}.

A trivial oracle machine:

Oracle machine M^O with oracle for language *O* does the following:

```
On input x,
        ask O if $x \in O$
              if yes, accept
              otherwise, reject
```

A slightly less trivial Oracle machine

Let HALT_{TM}=$\{\langle M, x\rangle | M\ is\ a\ TM$ which halts on input $x\}$

We will construct an oracle machine $M^{\text{HALT}_{\text{TM}}}$ with an oracle for HALT_{TM} which decides A_{TM}.

Let $M^{\text{HALT}_{\text{TM}}}$ on input $\langle M, x\rangle$ do the following:

```
    Ask if $\langle M, x\rangle \in \text{HALT}_{\text{TM}}$.
        if yes, simulate $M$ on $x$.
            If $M$ accepts, accept.
            Otherwise, reject
        otherwise, reject.
```

Why does this decide A_{TM}?

What did we just show?

Did we show that A_{TM} is computable?

We showed that, if $HALT_{TM}$ were computable, then A_{TM} would be computable! What can we conclude from this?

In general, if we can compute *L* with an oracle Turing machine with oracle *O*, we say $L \in R^{O}$. So we showed that $A_{TM} \in R^{HALT_{TM}}$.

This gives us a technique for proving things to be undecidable

In general, if I know that *A* is an undecidable language, and I show that an oracle TM with with an oracle for language *B* can decide *A*, then I can conclude *A* is undecidable. (Why?)

We call this an *oracle reduction* or *Turing reduction* from *A* to *B*. That is, we say *A* has an oracle reduction to *B* if $A \in R^B$.

Symbolically, we denote the relation "*A* has an oracle reduction to *B*" as

$$A \leq_T B.$$

What we just did in reduction terms

- So what we just showed was that $A_{TM} \leq_T HALT_{TM}$
- Let’s show that $HALT_{TM} \leq_T A_{TM}$
- How do we do that?

Theorem: $\text{HALT}_{\text{TM}} \leq_T \text{A}_{\text{TM}}$

Proof: We will construct an oracle TM $M^{\text{A}_{\text{TM}}}$ which decides HALT_{TM}.

On input $\langle M, x\rangle$, do the following:

```
    Change all of M's transitions to its reject state to transition to its
    accept state. Call this new machine M'.
    Ask if $\langle M', x\rangle \in \text{A}_{\text{TM}}$
        if yes, accept
        if no, reject
```

Observe that $M^{\text{A}_{\text{TM}}}$ decides HALT_{TM}. Suppose $\langle M, x\rangle \in \text{HALT}_{\text{TM}}$. Then, on input *x*, *M* either accepts or rejects. This implies that *M'* accepts *x* by our design of *M'*, and therefore that $\langle M', x\rangle \in \text{A}_{\text{TM}}$. Therefore, M^{ATM} accepts *x*.

Suppose $\langle M, x\rangle \notin \text{HALT}_{\text{TM}}$. Then *M* never reaches the accept or reject state on input *x*. By our construction of *M'*, *M'* only reaches the accept state when *M* would have reached its accept or reject state. So *M'* must not accept *x*, ergo $\langle M', x\rangle \notin \text{A}_{\text{TM}}$, and therefore $M^{\text{A}_{\text{TM}}}$ rejects *x*.

Since $M^{\text{A}_{\text{TM}}}$ accepts all $\langle M, x\rangle \in \text{HALT}_{\text{TM}}$ and rejects all $\langle M, x\rangle \notin \text{HALT}_{\text{TM}}$, $M^{\text{A}_{\text{TM}}}$ decides HALT_{TM} and therefore $\text{HALT}_{\text{TM}} \leq_T \text{A}_{\text{TM}}$. □

Exercise: Show A_{TM} oracle reduces to E_{TM}

$E_{TM} = \{\langle M\rangle | L(M) = \emptyset\}$

That is, E_{TM} is the set of (descriptions of) Turing machines which don't accept anything.

What does it mean to show $A_{TM} \leq_T E_{TM}$?

What's an idea we could use for our oracle Turing machine?

Another kind of reduction

Let $A \subseteq \Sigma^*$ and $B \subseteq \Sigma^*$ (let *A* and *B* be languages). We say a function $f: \Sigma^* \to \Sigma^*$ is a *many-one reduction* or a *map reduction* from *A* to *B* if

$$x \in A \text{ iff } f(x) \in B$$

That is, *f* assigns elements of *A* to elements of *B*, and assigns elements outside *A* to elements outside *B*.

IMPORTANT TO NOTICE: Different elements of *A* can go to the same element of *B*, and we don't have to hit every element of *B*!

Lastly, we say such a function is a *computable many-one reduction* or a *computable map reduction* if *f* is also computable. If there is such an *f*, we say $A \leq_m B$.

An example

- Let's show that A_{TM} has a computable many-one reduction to $HALT_{TM}$

What does *f* look like?

How do we show *f* is a computable many-one reduction?

$A_{TM} \leq_m HALT_{TM}$

We'll define f by an algorithm.

On input $\langle M, x\rangle$:

> Create a new machine *M'* by changing all of *M's* transitions to its reject state into transitions to a new loop state, in which *M'* will s simply loop.
>
> Output $\langle M', x\rangle$

f is clearly computable since we gave an algorithm for it, so we just need to show that it's a many-one reduction.

$A_{TM} \leq_m HALT_{TM}$

We'll define *f* by an algorithm.

On input $\langle M, x \rangle$:

> Create a new machine *M'* by changing all of *M's* transitions to its reject state into transitions to a new loop state, in which *M'* will s simply loop.
>
> Output $\langle M', x \rangle$

What does it mean for *f* to be a many-one reduction?

$A_{TM} \leq_m HALT_{TM}$

We'll define *f* by an algorithm.

On input $\langle M, x\rangle$:

> Create a new machine *M'* by changing all of *M's* transitions to its reject state into transitions to a new loop state, in which *M'* will s simply loop.
>
> Output $\langle M', x\rangle$

Show that if $\langle M, x\rangle \in A_{TM}$, then $f(\langle M, x\rangle) \in HALT_{TM}$ and

Show that if $\langle M, x\rangle \notin A_{TM}$, then $f(\langle M, x\rangle) \notin HALT_{TM}$