Fall 2015
Lecture 29: Reductions and Turing machines

When it came to eating strips of candy buttons, there were two main strategies. Some kids carefully removed each bead, checking closely for paper residue before eating.

Others tore the candy off haphazardly, swallowing large scraps of paper as they ate.

Then there were the lonely few of us who moved back and forth on the strip, eating rows of beads here and there, pretending we were Turing machines.
the Halting problem is undecidable

Given:  - CODE(P) for any program P
         - input x

Output: true if P halts on input x
        false if P does not halt on input x
        (or CODE(P) is not a valid program)

It isn’t possible to write a program that solves the Halting Problem.
Does $D(\text{CODE}(D))$ halt?

$H$ solves the halting problem implies that $H(\text{CODE}(D), x)$ is \textbf{true} iff $D(x)$ halts, $H(\text{CODE}(D), x)$ is \textbf{false} iff not

Suppose $D(\text{CODE}(D))$ halts.
Then, we must be in the \textbf{second} case of the if.
So, $H(\text{CODE}(D), \text{CODE}(D))$ is \textbf{false}
Which means $D(\text{CODE}(D))$ doesn't halt

Suppose $D(\text{CODE}(D))$ doesn't halt.
Then, we must be in the \textbf{first} case of the if.
So, $H(\text{CODE}(D), \text{CODE}(D))$ is \textbf{true}.
Which means $D(\text{CODE}(D))$ halts.

Contradiction!
proving that a language is undecidable

Consider a language $L \subseteq \Sigma^*$

We say that $L$ is **undecidable** if there is no Java program that takes $x \in \Sigma^*$ as input and outputs

- **true** if $x \in L$
- **false** otherwise

HALTING = \{ $(C, x) : C = \text{CODE}(P)$ and $P$ halts on input $x$ \}

**Theorem:** HALTING is **undecidable**.

**Strategy:** To show that some other language $L$ is undecidable, we could show that if we could decide $L$, we could also decide HALTING. Since HALTING is undecidable, it must be that $L$ is also undecidable!
Define the language:

\[ \text{HaltsNoInput} = \{ P : P \text{ is a program that halts when run with no input} \} \]

**Goal**: Show that if we could decide \( \text{HaltsNoInput} \), we could also decide \( \text{HALTING} \).
a useful tool: hardcoding

HaltsNoInput

= \{ P : P is a program that halts when run with no input \}
equivalence of programs is undecidable

EQUIV = \{ (P, Q) : programs P and Q have same behavior on every input \}
division by zero

\[ \text{DivByZero} = \{ (Q, x) : Q \text{ attempts to divide by 0 when run on input } x \} \]
computers and algorithms

- Does Java (or any programming language) cover all possible computation? Every possible algorithm?

- There was a time when computers were people who did calculations on sheets of paper to solve computational problems.

- Computers as we know them arose from trying to understand everything these people could do.
1930’s:

How can we formalize what algorithms are possible?

• **Turing machines** (Turing, Post)
  – basis of modern computers

• **Lambda Calculus** (Church)
  – basis for functional programming

• **μ-recursive functions** (Kleene)
  – alternative functional programming basis
Church-Turing Thesis:
Any reasonable model of computation that includes all possible algorithms is equivalent in power to a Turing machine

Evidence
- Intuitive justification
- Huge numbers of equivalent models to TM’s based on radically different ideas
Turing machines

• **Finite Control**
  – Brain/CPU that has only a finite # of possible “states of mind”

• **Recording medium**
  – An unlimited supply of blank “scratch paper” on which to write & read symbols, each chosen from a finite set of possibilities
  – Input also supplied on the scratch paper

• **Focus of attention**
  – Finite control can only focus on a small portion of the recording medium at once
  – Focus of attention can only shift a small amount at a time
• **Recording medium**
  – An infinite read/write “tape” marked off into cells
  – Each cell can store one symbol or be “blank”
  – Tape is initially all blank except a few cells of the tape containing the input string
  – Read/write head can scan one cell of the tape - starts on input

• **In each step, a Turing machine**
  – Reads the currently scanned symbol
  – Based on current state and scanned symbol
    Overwrites symbol in scanned cell
    Moves read/write head left or right one cell
    Changes to a new state

• Each Turing Machine is specified by its **finite set of rules**
Turing machines

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_1$</td>
<td>$(1, L, s_3)$</td>
<td>$(1, L, s_4)$</td>
</tr>
<tr>
<td>$s_2$</td>
<td>$(0, R, s_1)$</td>
<td>$(1, R, s_1)$</td>
</tr>
</tbody>
</table>

- $s_3$
- $s_4$

Input: 

```
_ _ 1 1 0 1 1 _ _
```
Ideal Java/C programs:
- Just like the Java/C you’re used to programming with, except you never run out of memory
  Constructor methods always succeed `malloc` in C never fails

Equivalent to Turing machines except a lot easier to program:
- Turing machine definition is useful for breaking computation down into simplest steps
- We only care about high level so we use programs
Turing’s big idea: machines as data

Original Turing machine definition:
- A different “machine” $M$ for each task
- Each machine $M$ is defined by a finite set of possible operations on finite set of symbols
  $M$ has a finite description as a sequence of symbols, its “code” denoted $<M>$

You already are used to this idea with the notion of the program code or text but this was a new idea in Turing’s time.
Turing’s big idea: a universal TM

• A Turing machine interpreter $U$
  – On input $<M>$ and its input $x$, $U$ outputs the same thing as $M$ does on input $x$
  – At each step it decodes which operation $M$ would have performed and simulates it.

• One Turing machine is enough
  – Basis for modern stored-program computer
    Von Neumann studied Turing’s UTM design

```
input x → M → M(x)  output

x → U → M(x)
< M > → U → M(x)
```
Rice’s theorem (“can’t tell a book by its cover”)

Not every problem on programs is undecidable!
Which of these is decidable?

- Input CODE($P$) and $x$
  Output: true if $P$ prints “ERROR” on input $x$
  after less than 100 steps
  false otherwise

- Input CODE($P$) and $x$
  Output: true if $P$ prints “ERROR” on input $x$
  after more than 100 steps
  false otherwise

Compilers Suck Theorem (informal):
Any “non-trivial” property the input-output behavior of Java programs is undecidable.
• Can’t rely on the idea of improved compilers and programming languages to eliminate major programming errors
  – truly safe languages can’t possibly do general computation
• Document your code
  – there is no way you can expect someone else to figure out what your program does with just your code; since in general it is provably impossible to do this!
foundations I, complete (almost)

What’s next?


The **final exam** is Monday, Dec 14, 2015

**Notes:** One page of notes allowed, front and back.

**Review session:**
• Sunday at 2pm in EEB 105

And then...