CSE 311 Foundations of Computing I

Lecture 27

Computability: Turing machines, Undecidability of the Halting Problem

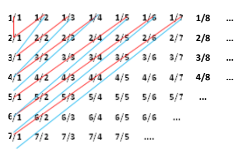
Autumn 2011

Announcements

- Reading
  - 7th edition: p. 201 and 13.5
  - 6th edition: p 177 and 12.5
  - 5th edition: p. ? and 11.5

Last lecture highlights

- Cardinality
- A set S is *countable* iff we can write it as $S=\{s_1, s_2, s_3, ...\}$ indexed by $\mathbb{N}$
- Set of rationals is countable
  - "dovetailing"



- Σ^* is countable
  - $\{0,1\}^* = \{0,1,00,01,10,11,000,001,010,011,100,101,...\}$
- Set of all (Java) programs is countable

Last lecture highlights

- The set of real numbers is not countable
  - "diagonalization"

		1	2	3	4	5	6	7	8	9	...
r_1 D= 0.	0.	5^{1}	0	0	0	0	0	0	0	...	...
r_2	0.	3	3^{5}	3	3	3	3	3	3	...	...
r_3	0.	1	4	2^{5}	8	5	7	1	4	...	...
r_4	0.	1	4	1	5^{1}	9	2	6	5	...	...
r_5	0.	1	2	1	2	2^{5}	1	2	2	...	...
r_6	0.	2	5	0	0	0	0^{5}	0	0	...	...
r_7	0.	7	1	8	2	8	1	8^{5}	2	...	...
r_8	0.	6	1	8	0	3	3	9	4^{5}	...	...
...	....	...	....	....	...	...	...	...	...	......	

  - Why doesn't this show that the rationals aren't countable?

Last lecture highlights

- There exist functions that cannot be computed by any program
  - The set of all functions $f : \mathbb{N}\rightarrow\{0,1,...,9\}$ is not countable
  - The set of all (Java/C/C++) programs is countable
  - So there are simply more functions than programs

Do we care?

- Are any of these functions, ones that we would actually want to compute?
  - The argument does not even give any example of something that can't be done, it just says that such an example exists
- We haven't used much of anything about what computers (programs or people) can do
  - Once we figure that out, we'll be able to show that some of these functions are really important

Turing Machines

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

Components of Turing's Intuitive Model of Computers

- 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

What is a Turing Machine?

What is a Turing Machine?

- 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 state of mind 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

What is a Turing Machine?

Turing Machine ≡ Ideal Java/C Program

- Ideal C/C++/Java programs
  - Just like the C/C++/Java you're used to programming with, except you never run out of memory
    - constructor methods always succeed
    - **malloc** 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 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"
- You already are used to this idea:
  - We'll write **<P>** for the code of program **P**
  - i.e. **<P>** is the program text as a sequence of ASCII symbols and **P** is what actually executes

Turing's Idea: A Universal Turing Machine

- A Turing machine interpreter **U**
  - On input **<P>** and its input **x**, **U** outputs the same thing as **P** does on input **x**
  - At each step it decodes which operation **P** would have performed and simulates it.
- One Turing machine is enough
  - Basis for modern stored-program computer
    - Von Neumann studied Turing's UTM design

Halting Problem

- Given: the code of a program **P** and an input **x** for **P**, i.e. given **(<P>,x)**
- Output: **1** if **P** halts on input **x**
  0 if **P** does not halt on input **x**

Theorem (Turing): There is no program that solves the halting problem
"The halting problem is undecidable"

Undecidability of the Halting Problem

- Suppose that there is a program **H** that computes the answer to the Halting Problem
- We'll build a table with
  - all the possible programs down one side
  - all the possible inputs along the other side
- Then we'll use the supposed program **H** to build a new program that can't possibly be in the table!

input **x**

program code **<P>**

	λ	0	1	00	01	10	11	000	001	010	011	....
λ	0	1	1	0	1	1	1	0	0	0	1	....
0	1	1	0	1	0	1	1	0	1	1	1	....
1	1	0	1	0	0	0	0	0	0	0	1	....
00	0	1	1	0	1	0	1	1	0	1	0	....
01	0	1	1	1	1	1	1	0	0	0	1	....
10	1	1	0	0	0	1	1	0	1	1	1	....
11	1	0	1	1	0	0	0	0	0	0	1	....
000	0	1	1	1	1	0	1	1	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

(<P>,x) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

A note on the table

- To make it easier to draw we've allowed every string to be the code of some program
  - We can just interpret each string that isn't well-formed to always halt whenever we try to run it.
- Alternatively, we could have only included rows and columns that are codes for programs
  - These are the only ones that matter anyway

Diagonal construction

- Consider a row corresponding to some program code **<P>**
  - the infinite sequence of **0**'s and **1**'s in that row of the table is like a **fingerprint** of **P**
- Suppose a program **H** for the halting problem exists
  - Then it could be used to figure out the value of any entry in the table
  - We'll use it to create a new program **D** that has a **different fingerprint** from every row in the table
  - **But that's impossible since there is a row for every program ! Contradiction**

input **x**

program code <P>

	λ	0	1	00	01	10	11	000	001	010	011	....
λ	0	1	1	0	1	1	1	0	0	0	1	....
0	1	1	0	1	0	1	1	0	1	1	1	....
1	1	0	1	0	0	0	0	0	0	0	1	....
00	0	1	1	0	1	0	1	1	0	1	0	....
01	0	1	1	1	1	1	1	0	0	0	1	....
10	1	1	0	0	0	1	1	0	1	1	1	....
11	1	0	1	1	0	0	0	0	0	0	1	....
000	0	1	1	1	1	0	1	1	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

(<**P**>,**x**) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

input **x**

program code <P>

	λ	0	1	00	01	10	11	000	001	010	011	....
λ	**0**	1	1	0	1	1	1	0	0	0	1	....
0	1	**1**	0	1	0	1	1	0	1	1	1	....
1	1	0	**1**	0	0	0	0	0	0	0	1	....
00	0	1	1	**0**	1	0	1	1	0	1	0	....
01	0	1	1	1	**1**	1	1	0	0	0	1	....
10	1	1	0	0	0	**1**	1	0	1	1	1	....
11	1	0	1	1	0	0	**0**	0	0	0	1	....
000	0	1	1	1	1	0	1	**1**	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

(<**P**>,**x**) entry is **1** if program **P** halts on input **x** and **0** if it runs forever

input **x** **Flipped Diagonal**

program code <P>

	λ	0	1	00	01	10	11	000	001	010	011	....
λ	**1**	1	1	0	1	1	1	0	0	0	1	....
0	1	**0**	0	1	0	1	1	0	1	1	1	....
1	1	0	**0**	0	0	0	0	0	0	0	1	....
00	0	1	1	**1**	1	0	1	1	0	1	0	....
01	0	1	1	1	**0**	1	1	0	0	0	1	....
10	1	1	0	0	0	**0**	1	0	1	1	1	....
11	1	0	1	1	0	0	**1**	0	0	0	1	....
000	0	1	1	1	1	0	1	**0**	0	1	0	....
001	.	.	.	.	.	.	.	.	.	.	.	
.	.	.	.	.	.	.	.	.	.	.	.	
.												

Want to create a new program whose halting properties are given by the **flipped** diagonal

Code for D assuming subroutine H that solves the Halting Problem

- Function **D(x)**:
  - if **H**(**x**,**x**)=**1** then
    - **while** (true); /* loop forever */
  - else
    - **no-op**; /* do nothing and halt */
  - endif
- **D**'s fingerprint is different from every row of the table
  - **D** can't be a program so **H** cannot exist!

That's it!

- We proved that there is no computer program that can solve the Halting Problem.
- This tells us that there is no compiler that can check our programs and guarantee to find any infinite loops they might have
  - The full story is even worse

Using undecidability of the halting problem

- We have one problem that we know is impossible to solve
  - Halting problem
- Showing this took serious effort
- We'd like to use this fact to derive that other problems are impossible to solve
  - don't want to go back to square one to do it

Another undecidable problem

The "always halts" problem

- **Given:** <**Q**>, the code of a program **Q**
- **Output:** **1** if **Q** halts on every input
  0 if not.

Claim: the "always halts" problem is undecidable

Proof idea:

- Show we could solve the Halting Problem **if** we had a solution for the "always halts" problem.
- No program solving for Halting Problem exists $\Rightarrow$ no program solving the "always halts" problem exists

What we would like

- To solve the Halting Problem need to handle inputs of the form (<**P**>,**x**)
- Our program will create a new program code <**Q**> so that
  - If **P** halts on input **x**
    - then **Q** **always** halts
  - If **P** runs forever on input **x**
    - then **Q** runs forever on at least one input
- In fact, the <**Q**> we create will act the same on all inputs

Creating <Q> from (<P>,x)

- Given (<**P**>,**x**) modify code of **P** to:
  - Replace all input statements of **P** that read input **x**, by assignment statements that 'hard-code' **x** in **P**
- This creates a new program text <**Q**>
- It would be easy to write a program **T** that changes (<**P**>,**x**) to <**Q**>

The transformation

```
int main(){
  …
  scanf("%d",&u);
  …
  scanf("%d",&v);
  …
  }
123 712
```

(<**P**>,**x**)

```
int main(){
  …
  u = 123;
  …
  v = 712;
  …
  }
```

<**Q**>

Program to solve Halting Problem if "always halts" were decidable

- Suppose "always halts" were solvable by program **A**
- On input (<**P**>,**x**)
  - execute the program **T** to transform (<**P**>,**x**) into <**Q**> as on last slide
  - call **A** with <**Q**> (the output of **T**) as its input and use **A**'s output as the answer.
- This would do the job of **H** which we know can't exist so **A** can't exist