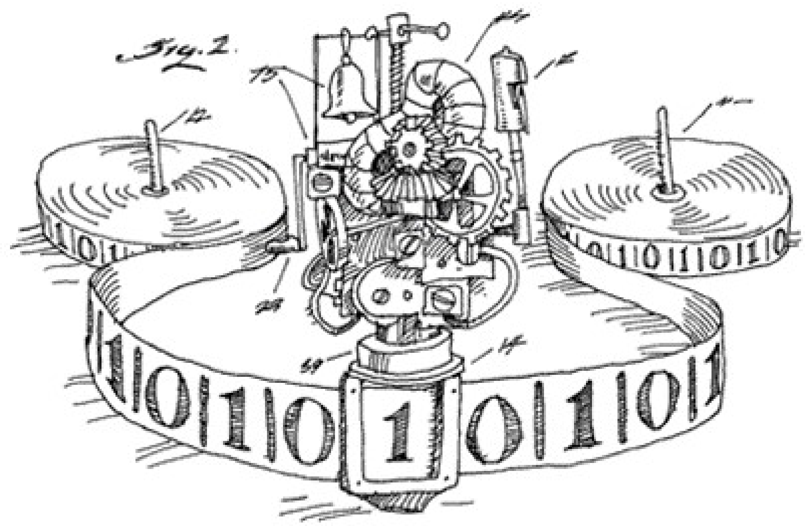
CSE 311: Foundations of Computing

Lecture 28: Undecidability, Reductions, and Turing Machines



Final exam

- **Monday** at either **2:30-4:20 p.m.** or **4:30-6:20 p.m.**
  - **Sieg Hall 134**
  - You need to fill out **Catalyst Survey** to say which you are taking by **midnight Sunday** night.
  - Bring your **UW ID** and have it out and ready during the exam
- **Comprehensive** coverage. If you had a homework question on it, it is fair game. See link on webpage.
  - Includes pre-midterm topics, e.g. formal proofs. Will contain the same sheets at end.
- **Review session: Sunday 3:30-5:00 p.m. EEB 105**
  - **Bring your questions !!**

Review: Countability vs Uncountability

- To prove a set A countable you must show
  - There exists a listing $x_1, x_2, x_3, \ldots$ such that every element of A is in the list.

$f: \mathbb{N} \to S$

- To prove a set B uncountable you must show
  - For *every* listing $x_1, x_2, x_3, \ldots$ there exists some element in B that is not in the list.

  - The diagonalization proof shows how to describe a missing element d in B based on the listing $x_1, x_2, x_3, \ldots$.
    Important: the proof produces a d no matter what the listing is.

Last time: Undecidability of the Halting Problem

`CODE(P)` means "the code of the program **P**"

The Halting Problem

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**

Theorem [Turing]: **There is no program that solves the Halting Problem**

Proof: By contradiction.

Assume that a program **H** solving the Halting program does exist. Then program **D** must exist

Does D(CODE(D)) halt?

```
public static void D(x) {
    if (H(x,x) == true) {
        while (true); /* don't halt */
    }
    else {
        return;        /*   halt   */
    }
}
```

H solves the halting problem implies that

  H(CODE(D),x) is **true** iff D(x) halts, H(CODE(D[illegible]ot

Suppose that D(CODE(D)) **halts**.   x = CODE(D)

  Then, by definition of H it mus[illegible]

    H(CODE(D), CO[illegible]

  Which by the defi[illegible] D(CODE(D)) **doesn't halt**

$A \rightarrow F$

Suppose th[illegible] **doesn't halt**.

  Th[illegible] of H it must be that

    [illegible]ODE(D), CODE(D)) is **false**

  Whi[illegible] by the definition of D means D(CODE(D)) **halts**

$\neg A \rightarrow F$

Contradiction!

The ONLY assumption was the program H exists so that assumption must have been false.

SCOOPING THE LOOP SNOOPER
A proof that the Halting Problem is undecidable

by Geoffrey K. Pullum (U. Edinburgh)

No general procedure for bug checks succeeds.
Now, I won't just assert that, I'll show where it leads:
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called *P*
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and *P* gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs...

SCOOPING THE LOOP SNOOPER

…

Here's the trick that I'll use – and it's simple to do.
I'll define a procedure, which I will call *Q*,
that will use *P*'s predictions of halting success
to stir up a terrible logical mess.

…

And this program called *Q* wouldn't stay on the shelf;
I would ask it to forecast its run on *itself*.
When it reads its own source code, just what will it do?
What's the looping behavior of *Q* run on *Q*?

…

Full poem at:

http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html

The Halting Problem isn't the only hard problem

- Can use the fact that the Halting Problem is undecidable to show that other problems are undecidable

General method:

Prove that if there were a program deciding B then there would be a way to build a program deciding the Halting Problem.

"B decidable $\rightarrow$ Halting Problem decidable"

Contrapositive:

"Halting Problem undecidable $\rightarrow$ B undecidable"

Therefore B is undecidable

Contrapositive

Last time: A CSE 141 assignment

Students should write a Java program that:

- Prints "Hello" to the console
- Eventually exits

GradeIt, PracticeIt, etc. need to grade the students.

How do we write that grading program?

WE CAN'T: THIS IS IMPOSSIBLE!

A related undecidable problem

- **HelloWorldTesting Problem:**
  - **Input: CODE(Q)** and **x**
  - **Output:**

    True if **Q** outputs "HELLO WORLD" on input **x**

    False if **Q** does not output "HELLO WORLD" on input **x**

- **Theorem: The HelloWorldTesting Problem is undecidable.**
- **Proof idea: Show that if there is a program T to decide HelloWorldTesting then there is a program H to decide the Halting Problem for code(P) and x.**

A related undecidable problem

- **Suppose there is a program T that solves the HelloWorldTesting problem. Define program H that takes input CODE(P) and x and does the following:**
  - **Creates CODE(Q) from CODE(P) by**

    redones System.out

    (1) removing all output statements from CODE(P), and

    (2) adding a System.out.println("HELLO WORLD") **immediately before any spot where P could halt**

    Then runs T on input CODE(Q) and x.

- If **P halts on input x** then **Q** prints HELLO WORLD and halts and so **H** outputs **true** (because **T** outputs true on input **CODE(Q)**)
- If **P doesn't halt on input x** then **Q** won't print anything since we removed any other print statement from CODE(**Q**) so **H** outputs **false**

We know that such an H cannot exist. Therefore T cannot exist.

The HaltsNoInput Problem

- **Input: CODE(R)** for program **R**
- **Output: True** if **R** halts without reading input

  False otherwise.

Theorem: HaltsNoInput is undecidable

General idea "hard-coding the input":

- Show how to use **CODE(P)** and **x** to build **CODE(R)** so **P halts on input x $\Leftrightarrow$ R halts without reading input**

The HaltsNoInput Problem

"Hard-coding the input":

- Show how to use **CODE(P)** and **x** to build **CODE(R)** so
  P halts on input x ⇔ R halts without reading input

  R: P(x)

- Replace input statement in **CODE(P)** that reads input **x** into variable **var,** by a hard-coded assignment statement:

  var = x

  to produce **CODE(R)**.

- So if we have a program **N** to decide **HaltsNoInput** then we can use it as a subroutine as follows to decide the Halting Problem, which we know is impossible:
  - On input CODE(**P**) and **x**, produce CODE(**R**). Then run **N** on input CODE(**Q**) and output the answer that **N** gives.

- The impossibility of writing the CSE 141 grading program follows by combining the ideas from the undecidability of HaltsNoInput and HelloWorld.

More Reductions

- Can use undecidability of these problems to show that other problems are undecidable.

- For instance:

$\mathrm{EQUIV}(P, Q)$: **True** if $P(x)$ and $Q(x)$ have the same behavior for every input x

False otherwise

$Q(x)$ prints "Hello World"

$P(x)$ input to HelloWorldTesting.

Rice's theorem

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

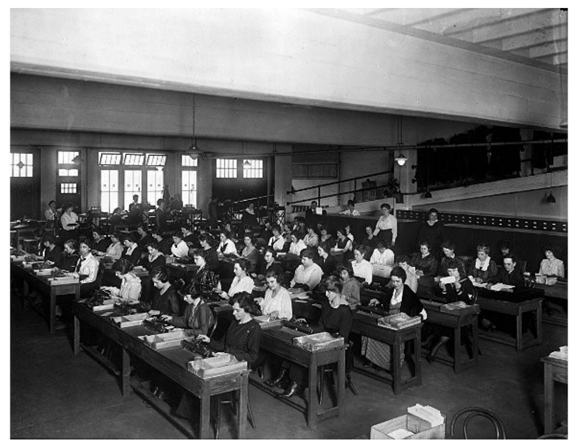
- **Input `CODE(P)` and x**
  Output: true if P prints "ERROR" on input x after more than 100 steps
  false otherwise

Rice's Theorem (a.k.a. Compilers Suck Theorem - informal):

Any "non-trivial" property of the input-output behavior of Java programs is undecidable.

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 paper to solve computational problems



- Computers as we known them arose from trying to understand everything these people could do.

Before Java

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, LISP
- **μ-recursive functions** (Kleene)
  - alternative functional programming basis

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 models based on radically different ideas turned out to be equivalent to TMs

Turing machines

- **Finite Control**
  - **Brain/CPU** that has only a **finite # of possible "states of mind"**
- **Recording medium** (memory)
  - **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**

Turing machines

- 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
  1. **Reads** the **currently scanned cell**
  2. **Based on current state and scanned symbol**
     i. **Overwrites symbol in scanned cell**
     ii. **Moves read/write head left or right one cell**
     iii. **Changes to a new state**

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

Turing machines

	_	0	1
s_1	(1, L, s_3)	(1, L, s_4)	(0, R, s_2)
s_2	(0, R, s_1)	(1, R, s_1)	(0, R, s_1)
s_3			
s_4			

_	_	1	1	0	1	1	_	_

UW CSE's Steam-Powered Turing Machine

Original in Sieg Hall stairwell

Turing machines

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 part 1: 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

– So… **M** has a finite description as a sequence of symbols, its “code”, which we denote **<M>**

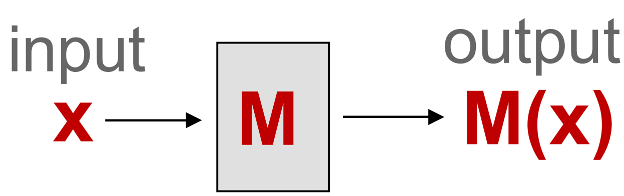
You already are used to this idea with the notion of the program code or text but this was a new idea in Turing’s time.

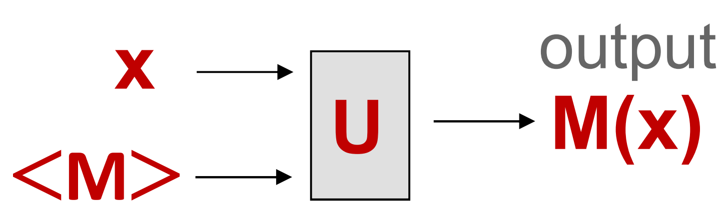
Turing's big idea part 2: A Universal TM

- **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





Takeaway from undecidability

- You 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!

We’ve come a long way!

- Propositional Logic.
- Boolean logic and circuits.
- Boolean algebra.
- Predicates, quantifiers and predicate logic.
- Inference rules and formal proofs for propositional and predicate logic.
- English proofs.
- Set theory.
- Modular arithmetic.
- Prime numbers.
- GCD, Euclid's algorithm, modular inverse, and exponentiation.

We’ve come a long way!

- Induction and Strong Induction.
- Recursively defined functions and sets.
- Structural induction.
- Regular expressions.
- Context-free grammars and languages.
- Relations and composition.
- Transitive-reflexive closure.
- Graph representation of relations and their closures.

We’ve come a long way!

- DFAs, NFAs and language recognition.
- Product construction for DFAs.
- Finite state machines with outputs at states.
- Minimization algorithm for finite state machines
- Conversion of regular expressions to NFAs.
- Subset construction to convert NFAs to DFAs.
- Equivalence of DFAs, NFAs, Regular Expressions
- Finite automata for pattern matching.
- Method to prove languages not accepted by DFAs.
- Cardinality, countability and diagonalization
- Undecidability: Halting problem and evaluating properties of programs.

What's next? ...after the final exam...

- **Foundations II** (312)
  - Fundamentals of counting, discrete probability, applications of randomness to computing, statistical algorithms and analysis
  - Ideas critical for machine learning, algorithms
- **Data Abstractions** (332)
  - Data structures, a few key algorithms, parallelism
  - Brings programming and theory together
  - Makes heavy use of induction and recursive defns

Course Evaluation Online

- Fill this out by Sunday night!
  - Your ability to fill it out will disappear at 11:59 p.m. on Sunday.
  - It will be worth your while to fill it out!

Final exam

- **Monday** at either **2:30-4:20 p.m.** or **4:30-6:20 p.m.**
  - **Sieg Hall 134**
  - You need to fill out **Catalyst Survey** to say which you are taking by **midnight Sunday** night.
  - Bring your **UW ID** and have it out and ready during the exam
- **Comprehensive** coverage. If you had a homework question on it, it is fair game. See link on webpage.
  - Includes pre-midterm topics, e.g. formal proofs. Will contain the same sheets at end.
- **Review session: Sunday 3:30-5:00 p.m. EEB 105**
  - **Bring your questions !!**