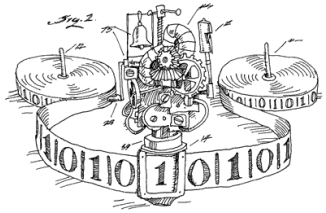
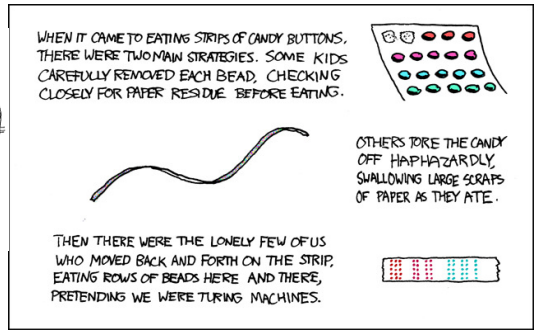
CSE 311: Foundations of Computing

Fall 2014

Lecture 29: Turing machines and more decidability



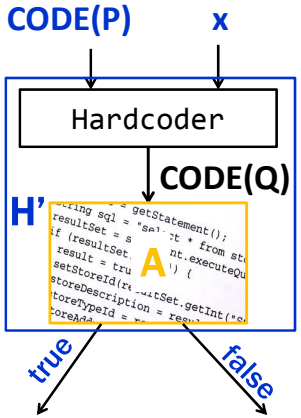


Proving that problem/set S is undecidable

- The main part is a programming task!
  - Figure out how you could use a subroutine that decided S to build a program to decide one of the problems you already know is undecidable.
- You also need to show that your program would really solve the known problem
  - The slides last class mostly focused on how to build that program but not on the correctness argument.
  - The need to show correctness should guide your programming solution

Showing there is no program solving **HaltsNoInput**

Suppose that hypothetical program A solves **HaltsNoInput** problem. *Combine with* `Hardcoder`:



H' outputs true on inputs
CODE(P) *and* x
iff A outputs true
on input CODE(Q) by diagram
iff Q() reads no input
and (always) halts by property of A
iff P(x) *halts* by definition of Hardcoder

If A existed then H' would solve the Halting Problem: Impossible

Showing **EQUIV** is Undecidable

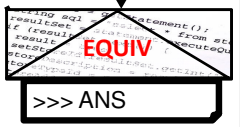
Consider the set:
EQUIV = {(CODE(P), CODE(R)): P, R are programs, P(x) = R(x) for all inputs x}

CODE(Q)

Question: Does Q() halt?

Step 1: Construct P:
`public static boolean P() {return true;}`
Step 2: Construct R:
Step a: Replace return type of Q with boolean
Step b: Replace all return values with true
Step c: Add "return true;" to the end of the program
Call this program R

(CODE(P),CODE(R))



Combined program outputs **true**
iff **P** and **R** are equivalent by diagram
iff **R** always returns **true** by defn of **P**
iff **Q** halts by construction of **R** from **Q**

Proving set S is undecidable

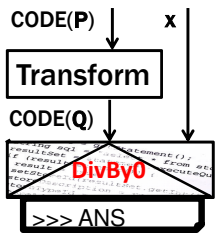
1. Assume that there is some (hypothetical) program that decides S
2. Choose some known undecidable set K
3. Show how to build a program to decide K that uses the program for S as a subroutine
   - Often you only need one call to S and can return the same answer
4. Prove that your algorithm outputs true on its input if and only if that input is in K, using the correctness of the hypothetical program for S
   - If you made one call and returned the same answer you just need to show that the input to the program is in K iff the value in the call to the subroutine is in S.
5. "Since K is undecidable, the program deciding S can't exist. Therefore S is undecidable."

Checking Division By Zero is undecidable

DivBy0={ (CODE(**Q**),**x**): **Q** executes a divide by 0 when run on input **x**}

Use **Halts**={(CODE(**P**),**x**): **P** halts on input **x**}

Let's try this with one call using same answer



Want to transform CODE(**P**) into CODE(**Q**) s.t.
- **Q** does a divide by 0 on input **x** iff **P** halts on input **x**

Checking Division By Zero is undecidable

Want **Q** does a divide by 0 on input **x** iff **P** halts on input **x**

Ideas for transformation from P to Q:

- Put a divide by 0 in place of each return statement of **P** and at the end of **P**
  - ensures that if **P** halts there will be a divide by 0 in **Q**
- Add a test before each original division in **P** to make sure that the divisor is not 0. If the divisor is 0 then don't do the division, print "error" and halt.
  - ensures that the only divide by 0 in **Q** occurs when **P** halts

Computers and algorithms: Programs and People

- 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

Recall: A brief history of reasoning

- 1900
  - Hilbert's famous speech outlines goal: mechanize all of mathematics
    23 problems
- 1930's
  - Gödel, Turing show that Hilbert's program is impossible.
    Gödel's Incompleteness Theorem
    Undecidability of the Halting Problem

Both use ideas from Cantor's proof about reals & rationals

before Java...more from our brief history of reasoning

- 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

All are equivalent!

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 computation

- **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

Some quotes from Turing's original paper

- Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable ... the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares.
- I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in place of single symbols. The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether 9999999999999999 and 999999999999999 are the same.
- The behavior of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite. The reasons for this are of the same character as those which restrict the number of symbols. If we admitted an infinity of states of mind, some of them will be "arbitrarily close" and will be confused. Again, the restriction is not one which seriously affects computation, since the use of more complicated states.
- [He then discusses simple operations that allow the computer to change one of the observed squares]
- .... the simple operations must include changes of distribution of observed squares. The new observed squares must be immediately recognisable by the computer. I think it is reasonable to suppose that they can only be squares whose distance from the closest of the immediately previously observed squares does not exceed a certain fixed amount. Let us say at each of the new observed squares is within L squares of an immediately previously observed square.

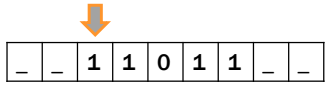
what is a Turing machine?

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

sample Turing machine

	_	0	1
s_1	$(1,s_3)$	$(1,s_2)$	$(0,s_2)$
s_2	(H,s_3)	(R,s_1)	(R,s_1)
s_3	(H,s_3)	(R,s_3)	(R,s_3)



what is a Turing machine?

turing machine ≡ ideal Java/C program

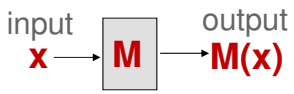
- Ideal Java/C programs
  - Just like the Java/C you're used to programming with, except you never run out of memory
    - Constructor methods always succeed
    - **malloc** in C never fails
- Equivalent to Turing machines except a lot easier to program !
  - Turing machine definition is useful for breaking computation down into simplest steps
  - We only care about high level so we use programs

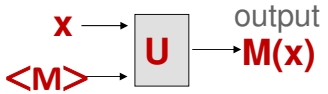
Turing's big idea: machines as data

- 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 idea: a Universal Turing Machine

- 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





General phenomenon: can't tell a book by its cover

and you can't tell what a program does just by its code…

Rice's Theorem: In general there is no way to tell anything about the input/output (I/O) behavior of a program **P** just given it code!

Note: The statement above is not precise, and we didn't prove it, so this isn't something you can use on homework or exams

Even harder problems

- With the halting problem, by using the Universal machine (a program interpreter) we can simulate **P** and input **x** and always get the **true** answers correct
  - we can't be sure about answering **false**
- For other problems we can always answer **false** correctly but maybe not the **true** answers
- There are natural problems where you can't even do that!
  - The **EQUIV** problem is an example of this kind of even harder problem

Quick lessons

- Don'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!