Lecture 18: Important Undecidable Problems
(and what to do about them)

Undecidability + Rice + Church-Turing

Language and Problems: any problem can be restated as a language recognition problem

Undecidability: undecidable languages that cannot be decided by any Turing Machine

Rice’s Theorem: all nontrivial properties about the language of a TM are undecidable

Church-Turing Thesis: any mechanical computation can be done by some TM

Conclusion: any nontrivial property about general mechanical computations cannot be decided!

Program Halting Problem

• Input: a program $P$ in some programming language
• Output: true if $P$ terminates; false if $P$ runs forever.

Examples

halts(“2+2”)  True
halts("def f(n):
    if n==0: return 1
    else: return n * f(n-1)
    f(10)")  True
halts("def f(n):
    if n==0: return 1
    else: return n * f(n-1)
    f(10.5)")  False

Tougher Example

halts(""
def isPerfectNumber(n): # n is perfect if factors sum to n
divs = findDivisors(n)
    return n == sum(divs)
i = 3
    while not isPerfectNumber (i): i = i + 2
    print i")

Unknown  Note: it is unknown where an odd perfect number exists. (Numbers up to $10^{100}$ have been tried without finding one yet.)
If you had \texttt{halts}, you could prove or disprove nearly every open mathematical problem!
- Does an odd perfect number exist?
- Reimann hypothesis: The real part of any non-trivial zero of the Riemann zeta function is $\frac{1}{2}$.
- Goldbach conjecture: Every number $> 2$ is the sum of three primes (including 1).
- Poincaré conjecture: Every simply connected closed three-manifold is homeomorphic to the three-sphere.
- ...

This suggests it is unlikely \texttt{halts} exists...but doesn’t prove it (yet).

More Convincing Non-Existence Proof

```python
def paradox():
    if \texttt{halts(“paradox()”)}:
        while True: print “You lose”
    else:
        return “You lose”
```

*If \texttt{halts(“paradox()”)} is \texttt{True}: \texttt{paradox()} loops forever*
*If \texttt{halts(“paradox()”)} is \texttt{False}: \texttt{paradox()} halts*

Neither option makes sense, so \texttt{halts} must not exist!

Recall from Lecture 16...

Assume there exists some TM $H$ that decides $A_{TM}$.
Define $D(<M>) = \text{Construct a TM that:}$
Outputs the \textbf{opposite} of the result of simulating $H$ on input $<M, <M>>$

```
if \texttt{D accepts <D>}
\quad \texttt{H(D, <D>)} \text{ accepts and} \texttt{D(<D>)} \text{ rejects}
if \texttt{D rejects <D>}
\quad \texttt{H(D, <D>)} \text{ rejects and} \texttt{D(<D>)} \text{ accepts}
```

Whatever $D$ does, it must do the opposite, so there is a contraction!

Alternate Proof: Reduction

- A Python procedure that solves \texttt{halts} must not exist, since if it did we could:
  - Write a TM simulator in Python:
    ```python
def simulateTM(M, w):
    # simulates M on input w
    ...
    ```
  - Determine if a TM $M$ halts on $w$ using \texttt{halts}:
    ```python
    halts(“simulateTM(M,w)”)  # simulates M on input w
    ```
- But, we know $HALT_{TM}$ is undecidable. Hence, \texttt{halts} for Python must not exist.

Universal Programming Language

- Definition: a programming language that can describe every algorithm.
- Equivalently: a programming language that can simulate every Turing Machine.
- Equivalently: a programming language in which you can implement a Universal Turing Machine.

Does this work for Java?
Lecture 18: Important Undecidable Problems

Which of these are Universal Programming Languages?

- BASIC
- C++
- COBOL
- C#
- PostScript
- Python
- Fortran
- x86
- Java
- SQL
- HTML
- PLAN
- Scheme
- Ruby

Proofs

- BASIC, C, C++, C#, Fortran, Java, JavaScript, PostScript, Python, Ruby, Scheme, etc.:
  - Proof: implement a TM simulator in the PL
- HTML is not universal:
  - Proof: show some algorithm that cannot be implemented in HTML
  - Easy choice: an infinite loop
- PLAN (Packet Language for Active Networks):
  - Designed to be non-universal: resource-constrained language

Why is it impossible for a programming language to be both universal and resource-constrained?

Resource-constrained means it is possible to determine an upper bound on the resources any program in the language can consume.

All universal programming language are equivalent in power: they can all simulate a TM, which can carry out any mechanical algorithm.

Why so many programming languages?

Proliferation of Universal PLs

- “Aesthetics”
  - Some people like :=, others prefer =.
  - Some people think whitespace shouldn’t matter (e.g., Java), others think programs should be formatted like they mean (e.g., Python)
  - Some people like goto, others like throw.
- Expressiveness vs. “Truthiness”
  - How much you can say with a little code vs. how likely it is your code means what you think it does
Lecture 18: Important Undecidable Problems

Are any Important Problems Undecidable?

Virus Detection

- **Input:** a program \( P \)
- **Output:** **True** if executing \( P \) would cause a file on the host computer to be "infected"; **False** otherwise.

Rough Proof:
```
def halts(P):
    return isVirus("removePossibleInfections(P)
                   infectFile()")
```

Therefore: Anti-Virus programs cannot exist!

"The Art of Computer Virus Research and Defense"
Peter Szor, Symantec

Example: Morris Internet Worm (1988)

- \( P = \) fingerd
  - Program used to query user status (running on most Unix servers)
- \( \text{isVulnerable}(P) \)?
  - Yes, for \( w = \) "nop 400 pushl $68732f pushl $6e69622f movl r10 pushl $0 pushl $0 pushl $r10 pushl $3 movl
    sp, ap chnk $3b"
    - Worm infected several thousand computers (~10% of Internet in 1988)

Vulnerability Detection

- **Input:** a program \( P \)
- **Output:** **True** if there is some input \( w \), such that running \( P \) on \( w \) leads to a security compromise; **False** otherwise.

Rough Proof:
```
def haltsOnInput(P,w):
    # we know this doesn't exist either
    return isVulnerable("P(w)
                        compromiseSecurity()")
```

Lecture 18: Important Undecidable Problems

“Solving” Undecidable Problems

- Undecidable means there is no program that
  1. Always gives the correct answer, and
  2. Always terminates
- Must give up one of these:
  - Giving up #2 is not acceptable in most cases
  - Must give up #1: none of the anti-virus or vulnerability detection products are always correct
- Or change the problem
  - e.g., detect file infections during an execution, make resilient execution environments, etc.

Actual isVirus Programs

- Sometimes give the wrong answer:
  - “False positive”: say P is a virus when it isn’t
  - “False negative”: say P is safe when it is
- Database of known viruses: if P matches one of these, it is a virus
- Clever virus authors can make viruses that change each time they propagate
  - It is undecidable to determine if a given program is the same as a known virus
  - Emulate program for a limited number of steps; if it doesn’t do anything bad, assume it is safe

Recap

- If you can simulate a Turing Machine with programming language PL, it is a universal programming language
- There is no algorithm for deciding halts for $P \in PL$: if there was, we could decide $A_{TM}$.
- There is no way to determine in general if $P \in PL$ is a virus, or a program containing vulnerabilities, or any interesting property...
- We can build algorithms that get it right some of the time (and this can be valuable).

PS5 is due Tuesday