Computability, the Halting Problem, and Program Analysis

CS251 Programming Languages
Fall 2016, Lyn Turbak

Department of Computer Science
Wellesley College

Motivation

Which of these following Python programs has inputs for which it loops forever?

```
def f(x):
  return x+1
```

```
def g(x):
  while True:
    pass
  return x
```

```
def g2(x):
  return g2(x)
```

```
def k(x):
  while x != 1:
    if (x % 2) == 0:
      x = x/2
    else:
      x = 3*x + 1
  return 1
```

```
def h(x):
  while x > 0:
    x = x+1
  return x
```

```
def h2(x):
  if x <= 0:
     return x
  else:
     return h(x+1)
```


Big Idea of this Lecture

It is generally **impossible** to answer any interesting question about program analysis!

This is a consequence of **Rice's Theorem** (see CS235).

For example, will this program ever:

- halt on certain inputs
- encounter an array index out of bounds error?
- throw a NullPointerException?
- access a given object again?
- send sensitive information over the network?
- divide by 0?
- run out of memory, starting with a given amount available?
- try to treat an integer as an array?

Key Concepts from CS235

This lecture summarizes key concepts from *CS235 Formal Languages and Automata* that are important to understand for PL design:

- Countable and uncountable sets
- Computable functions
- Uncomputable functions/undecidable languages
  - The halting problem
  - Reduction
  - Uncomputability and PL design
- The Church/Turing hypothesis
- Turing-completeness

Computability

- A function *f* is computable if there is a program that takes some finite number of steps before halting and producing output *f(x)*.
- Computable: $f(x) = x + 1$, for natural numbers
  - addition algorithm
- Uncomputable (a.k.a. undecidable) functions exist!
  - We'll first prove this by a "counting argument": there are way more functions than there are programs to compute them!
  - Then we'll show a concrete example: the halting problem.

Some Simple Sets

Bool = the booleans = {true, false}

Nat = the natural numbers = {0, 1, 2, 3 …}

Pos = the positive integers = {1, 2, 3, 4, …}

Int = all integers = { …, -3, -2, -1, 0, 1, 2, 3, …}

Rat = all rational numbers (fractions, w/o duplicates)
= {…, -3/2, -2/3, -1/3, -2/1, -1/1, 0/1,
1/1, 1/2, 2/1, 1/3, 2/3, 3/2, …}

Real = all real numbers = {0, 17, -2.5, 1.736, -5.3333…, 3.141…, ….}

Irrat = all irrational numbers (cannot be expressed as fractions
= {sqrt(2) =1.414.., pi = 3.14159…, e = 2.718…, …}

Nat and Pos Have the "Same Size" (≅)

Nat ≅ Pos by the pictured bijection

Nat	0	1	2	3	…
	↕	↕	↕	↕	
Pos	1	2	3	4	…

Nat and Int Have the Same Size!

Nat ≅ Int by the pictured bijection

Nat	0	1	2	3	4	5	6	…
f	↓	↓	↓	↓	↓	↓	↓	
Int	0	1	-1	2	-2	3	-3	…

Int …	-3	-2	-1	0	1	2	3	…
f^{-1}	↓	↓	↓	↓	↓	↓	↓	
Nat …	6	4	2	0	1	3	5	…

This is an example of **proof by construction**.

Countable and Uncountable Sets

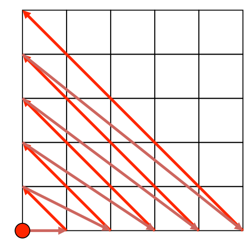
A set S is

- **finite** iff S ≅ {1, 2, …, n} for some n. E.g. Bool (n = 2)
- **infinite** iff S is not finite. E.g. Nat, Int, Rat, Real
- **countably infinite** iff S ≅ Nat. E.g. Pos, Int
- **countable** iff S is finite or countably infinite.
  I.e., there is a procedure for enumerating all the elements of S. E.g., Bool, Pos, Int
- **uncountable** iff S is not countable

Now we'll see that (1) Rat is countable and
(2) Real and Irrat are uncountable

Rat is Countable

Key idea: can enumerate Nat x Nat as follows:



Mopping up:

- Need to eliminate duplicates, e.g., 1/2= 2/4
- Need to handle negative rational (as in showing Int countable).

Real is Uncountable: Diagonalization

Key idea: use a special form of **proof by contradiction** known as **diagonalization**.

Assume that [0,1) ⊆ Real is countable and derive a contradiction.

If [0,1) is countable, there must be a bijection f ∈ Nat → [0,1) that enumerates all real numbers between 0 (inclusive) and 1 (exclusive). I.e., if r ∈ [0,1), then there is an n ∈ Nat s.t. f(n) = r.

If this is so, we can construct a table of f whose rows are f(n) and whose columns show the digits after the decimal point for each number.

f(0)	1	4	1	5	
f(1)	7	3	8	2	…
f(2)	5	4	9	6	
f(3)	8	2	7	3	

⋮

Real Diagonalization Continued

f(0)	1	4	1	5	
f(1)	7	3	8	2	…
f(2)	5	4	9	6	
f(3)	8	2	7	3	

⋮

Focus on the diagonal table entries, and construct a number whose decimal representation differs from every position in the diagonal*. E.g., .2786 …

Any such number is **not** a row in the table and so is not in the image of f. Thus, the assumption that f is a bijection is wrong! **X proof by contradiction**

Indeed, it's **way** wrong. The number of counterexamples we can construct is a **way bigger infinity** (an **uncountable** infinity) than the row s in the table.

Diagonalization is the heart of the halting theorem proof we'll see soon.

* For technical reasons, should not use 0 or 9 in the constructed number.

Irrat is Uncountable

Real = Rat U Irrat.

We know Rat is countable.

Assume Irrat is countable. Then Real would be countable.

But we know Real is uncountable. Thus, the assumption that Irrat is countable is wrong. **X proof by contradiction.**

Conclusion: Irrat is uncountable.

Alphabets, Strings, and Languages

An **alphabet** is a set of symbols.
E.g.: $\Sigma_1 = \{0,1\}$; $\Sigma_2 = \{-,0,+\}$ $\Sigma_3 = \{a,b, \ldots, y, z\}$; Σ_4 = {☺, ⇛, $\boxed{a}$, $\boxed{aa}$}

A **string over Σ** is a sequence of symbols from Σ.
The empty string is often written ε.
Σ^* denotes all strings over Σ. E.g.:

- $\Sigma_1{}^*$ contains ε, 0, 1, 00, 01, 10, 11, 000, …
- $\Sigma_2{}^*$ contains ε, -, 0, +, --, -0, -+, 0-, 00, 0+, +-, +0, ++, ---, …
- $\Sigma_3{}^*$ contains ε, a, b, …, aa, ab, …, bar, baz, foo, wellesley, …
- $\Sigma_4{}^*$ contains ε, ☺, ⇛, $\boxed{a}$, $\boxed{aa}$, …, $\boxed{a}$⇛ ☺, …, $\boxed{a}\boxed{aa}$, $\boxed{aa}\boxed{a}$, …

A **language over Σ** is any subset of Σ^*.
I.e., it's a set of strings over Σ. E.g.:

- L_1 over Σ_1 is all sequences of 1s and all sequences of 10s.
- L_2 over Σ_2 is all strings with equal numbers of -, 0, and +.
- L_3 over Σ_3 is all lowercase words in the OED.
- L_4 over Σ_4 is {☺, ☺ ⇛ ☺, $\boxed{a}\boxed{aa}$}.

Programs in any PL are countable!

- For any finite alphabet Σ, the language Σ^* of all strings over Σ is countable.
  - Why? We can enumerate all the strings in order by length and eventually get to any given string.
- Any language over a finite alphabet Σ is countable, because subsets of countable sets are countable.
- For any programming language L (e.g., Python, Java, etc.), the valid programs in L are countable!

Predicates on Nat

A predicate on Nat is any function that takes a natural number as an input and returns T (true) or F (false) as an output.

Mathematically, we can represent such functions as input/output pairs. For example:

- leqTwo = { (0, T), (1, T), (2, T), (3, F), (4, F), (5, F), (6, F), (7, F), …}
- isEven = { (0, T), (1, F), (2, T), (3, F), (4, T), (5, F), (6, T), (7, F), …}
- isPrime = { (0, F), (1, F), (2, T), (3, T), (4, F), (5, T), (6, F), (7, T), …}
- isNat = { (0, T), (1, T), (2, T), (3, T), (4, T), (5, T), (6, T), (7, T), …}

Define NatPred = the set of all predicates on Nat

= {leqThree, isEven, isPrime, isNat, ….}

Important! Mathematical functions like elements of NatPred are not programs! You **must** understand this, or else all the following slides won't make sense.

NatPred is Uncountable!

Assume there's a bijection f : Nat → NatPred. E.g.

f(0) = leqTwo
f(1) = isEven
f(2) = isPrime
f(3) = isNat
…

Now make a diagonalization argument:

	0	1	2	3	
f(0)	T	T	T	F	
f(1)	T	F	T	F	…
f(2)	F	F	T	T	
f(3)	T	T	T	T	
⋮					

The Nat predicate { (0,F), (1,T), (2,F), (3,F), … } that negates every element on the diagonal is not f(i) for any i in Nat.

By diagonalization, NatPred is uncountable!

Uncomputable Functions: Summary So Far

NatPred is uncountable.

Programs in any PL are countable. So they can't possibly express all the predicates in NatPred.

As with Reals, the uncountable infinity of NatPred is **way bigger** than the countable infinity of ProgramsInPython. From the probability perspective, 0% of predicates in NatPred can be written in Python! (We can clearly write lots of them, but that number is infinitesimally small compared to what we *want* to write!)

Depressing conclusion: we can't even express the vast majority of predicates in NatPred in Python, Java, etc., so clearly we can't express the vast majority of other mathematical functions!

Do we care in practice?

Could it be that we don't care about the mathematical functions that we can't express with programs? Maybe they don't matter …

Amazingly (and sadly) we can describe particular mathematical functions related to PLs that we care **a lot** about that are uncomputable.

The most famous example is the **halting problem**. It has to do with analyzing programs that might not halt (e.g., they loop forever on some inputs).

Programs that loop vs. take a long time

How do we distinguish programs that run a long time from ones that loop?

E.g. 3x+1 problem (saw this back on slide 2-2).

$$f(x) = \begin{cases} 3x + 1, & \text{if x is odd} \\ x/2, & \text{if x is even} \end{cases}$$

Problem: for all n, is there some i such that $f^i(n) = 1$? I.e., is it the case that iterating f at a starting point never loops?

No one knows! This is an open problem!

You might *think* you can tell when a Python program will loop, but this example shows that you're wrong!

Halting Problem

HALT(P,x): Does program P halt when run on input x?
(For simplicity, assume P and x are strings, and P is a program in your favorite PL.)

I.e., on input x, does P terminate after a finite number of steps and return a result

HALT is a mathematical function that is provably uncomputable.

Why do we care?

- Canonical undecidable problem.
- **BIG** implications for what we can and cannot decide about programs.

Hand-wavy intuition

- Run P on x for 100 steps. Did it halt?
- Run P on x for 1000 steps. Did it halt?
- ...
- P on x could always run at least one step longer than we check ...

But, perhaps we can be cleverer. Back on slide 2-2, we didn't have to actually run the programs in order to determine whether some halted on some inputs.

Proof: Halting Problem is Uncomputable

Proof by contradiction using diagonalization.

- Suppose HaltImpl(P,x) is an implementation of *HALT* in your favorite PL.
  - halts on all inputs and returns true if running program P on input x will halt and false if it will not.
- Define Sly(P) in your favorite PL as the following program:
  - Run HaltImpl(P,P). This will always halt and return a result.
  - If the result is true, loop forever, otherwise halt.
- So...
  - Sly(P) will run forever if P(P) would halt and
  - Sly(P) will halt if P(P) would run forever.
  - (Not actually running P(P), just asking what it *would* do *if run*.)
- Run Sly(Sly).
  - It first runs HaltImpl(Sly,Sly), which halts and returns a result.
  - If the result is true, it now loops forever, otherwise it halts.
- So...
  - If Sly(Sly) halts, HaltImpl(Sly,Sly) told us that Sly(Sly) would run forever.
  - If Sly(Sly) runs forever, HaltImpl(Sly,Sly) told us that Sly(Sly) would halt.
- Contradiction! No implementation HaltImpl of the *HALT* function can exist!

Let's be more concrete

Suppose someone has written `HaltPy` in Python.

```
def HaltPy (P_string, x_string):
  # Program arg P_string is Python function as multiline string
  … # Code that always returns True or False.
```

You write:

```
def SlyPy (P_string): # Program arg is multiline string
  if HaltPy(P_string, P_string):
    while True: # loop forever
      pass
  else:
    return 'halted'
```

```
fString = \
'''def f(x):
  return x+1''' # triple quotes for multiline Python strings
```

```
SlyPy(fString)  # What does this do?
SlyPy(gString)  # What does this do?
SlyPy(SlyPyString)  # What does this do?
```

How is this resolved in practice?

In practice, any Python function HaltPy has to either:

- Loop infinitely on certain inputs (i.e., it never returns True or False on these inputs)
- Be allowed to return an answer that means "I don't know"

Rice's Theorem: In PL, Uncomputable = Interesting

As a consequence of what is known as **Rice's theorem** (see CS235), most interesting questions about programs are uncomputable = undecidable. For example:

Will this program ever:

- halt on certain inputs
- encounter an array index out of bounds error?
- throw a NullPointerException?
- access a given object again?
- send sensitive information over the network?
- divide by 0?
- run out of memory, starting with a given amount available?
- try to treat an integer as an array?

Proving Undecidability

There are two approaches for showing that a problem is uncomputable = undecidable.

1. Use **diagonalization** argument like that for HALT. This is cumbersome.
2. Transform an existing undecidable language to L via a technique called **reduction**. Much easier in practice:
   - To prove a problem *P* is undecidable, *reduce* a known undecidable problem *Q* to it:
     - Assume *DecideP* decides the problem *P*.
     - Show how to translate an instance of *Q* to an instance of *P*, so *DecideP* decides *Q*. *(translation must halt)*
     - Contradiction.
   - *Q* is typically the halting problem.

Reduction *or* The Blue Elephant Gun

Q: How do you shoot a blue elephant?

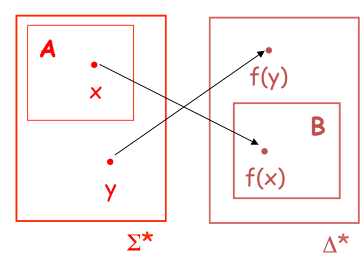
A: With a blue elephant gun, of course!

Q: How do you shoot a white elephant?

A: Hold its trunk until it turns blue, and then shoot it with a blue elephant gun!

A (many-to-one) **reduction of A to B** is a function f: $\Sigma^* \rightarrow \Delta^*$ such that x in A iff f(x) in B.

f must be computable by a terminating program.



How To Use Reduction

In **proofs by construction**:

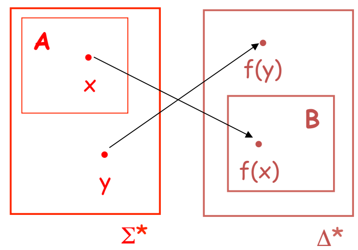
Given a B that is known to be solvable, use it to solve A.

E.g. A = sorting the lines of a file
B = sorting the elts of an array.

In **proofs by contradiction**:

Given an A that is known to be unsolvable, show that if B existed, it could be used to solve A. So B must be unsolvable too!

E.g. A = HALT
B = The problem you're trying to show is unsolvable.



Example: HALT_SOME(Q) is Undecidable

- *HALT_SOME(Q)*:
  - does an input exist on which program Q halts?
- Suppose that *HALT_SOME(Q)* is decidable
- Solve *HALT(P,x)* with *HALT_SOME(Q)*:
  - Build a new program R that ignores its input and runs P(x).
  - *HALT_ANY*(R) returns true if and only if P halts on x.
    - R(...) always does same thing, so if one halts, all do.
- Contradiction!

More Concretely (in Python)

Suppose someone has written `HaltSomePy` in Python.

```
def HaltSomePy (Q_string):
   … # Code that returns True if Q is a string for a
     # one-argument Python function that halts on some input
     # and otherwise returns False
```

You write:

```
def HaltPy (P_string, x_string):
  R_string = \
  'def R(ignore):\n' \
  + P_string + '\n' \
  + x_string + '\n' \
  + 'return P(x)' # Assume defs named P and x
                  # and indentation OK
  return HaltSomePy(R_string)
```


More Concretely (in Python) 2

```
def fString =
'''  def P(y):
    return y+1'''

def numString =
'''  x = 17'''
```

Then in the call

```
HaltPy (fString, numString)
```

R_string is the string

```
'''def R(ignore):
  def P(y):
    return y+1
  x = 17
  return P(x)'''
```

In practice: must be conservative

Programs that take programs as inputs typically can't answer "yes" or "no", Instead, they must answer "yes", "no", or "I give up; not sure."

For example:

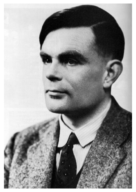
- type systems
- garbage collection
- program analysis

Alternatively, can restrict expressiveness of system so that a "yes" or "no" answer is always possible. E.g., Java type system.

Early Theory of Computation

- In the 1920s – 1940s, before the advent of modern computing machines, mathematicians were wrestling with the notion of effective computation: formalisms for expressing algorithms.
- Many formalisms evolved:
  - Turing Machines (Turing); **CS235!**
  - Lambda-calculus (Church, Kleene); **CS251!**
  - combinatory logic (Schönfinkel, Curry);
  - Post systems (Post);
  - m-recursive functions (Gödel, Herbrand).
- All of these formalisms were proven to be equivalent to each other!

The Church-Turing Thesis and Turing-Completeness



- **Church-Turing Thesis**: Computability is the common spirit embodied by this collection of formalisms.
- This thesis is a claim that is widely believed about the intuitive notions of algorithm and effective computation. It is not a theorem that can be proved.
- Because of their similarity to later computer hardware, Turing machines have become the gold standard for effectively computable.
- We'll see in CS251 that the lambda-calculus formalism is the foundation of modern programming languages.
- A consequence: programming languages all have the "same" computational "power" in term of what they can express. All such languages are said to be **Turing-complete**.

Expressiveness and Power

- About:
  - ease
  - elegance
  - clarity
  - modularity
  - abstraction
  - ...
- Not about: computability
- Different problems, different languages
  - Facebook or web browser in assembly language?

A Humorous Take on Computability

"In the long run, we are all dead."
– John Maynard Keynes

http://xkcd.com/1266/

Next time

- First case study: Lisp, Racket, and functional programming
- Clean slate approaching language.