CSCI 334:
Principles of Programming Languages

Lecture 9: Computability

Instructor: Dan Barowy

Williams

Topics

Desugaring code

Function graphs

Decidability

Your to-dos

1. Lab 4, **due Sunday 10/8** (partner lab)
2. Read *Proof by Reduction* **for Thur, 10/12**

Announcements

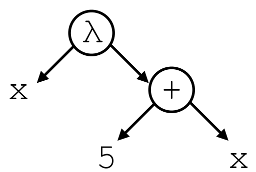
- **Midterm exam**, in class, Thursday, Oct 19.
- **Field trip to WCMA**, Thursday, Nov 2.
- Colloquium: **Leveraging ML Predictions for Beyond-Worst-Case Algorithm Design**, 2:35pm in Wege Auditorium.

Traditionally, we measure the performance of algorithms in the worst-case model. That is, the algorithms are designed to perform well against an adversarial input sequence. While the worst-case paradigm provides extremely strong guarantees, it can often be too pessimistic compared to the empirical performance on typical datasets. This talk is about a growing line of work that incorporates machine learned predictions to break through worst-case running time barriers.

Translating real code to lambda expressions

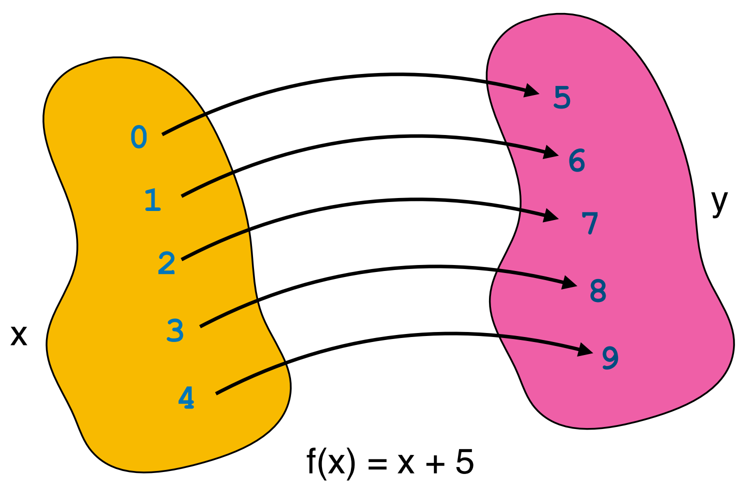
```
def f(x):
    return 5 + x
```

f = λx.(+ 5 x)



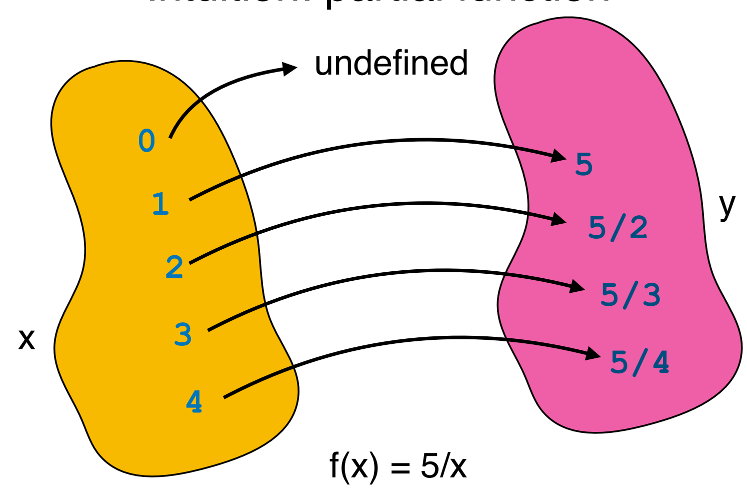
Computability

Intuition: total function



For every element in x, there is a corresponding element in y. x maps to at most one element in y.

Intuition: partial function



x still maps to at most one element in y, however, there is not a y for every x.

The **graph** of a function

$f(x) = x + 5$

$\{\langle x, x+5\rangle \mid x \in \mathbb{Z}\}$

$\{\langle x, x+5\rangle \mid x \text{ is an integer}\}$

The graph is **not a picture**!

The **graph** of a function

$f(x) = 5/x$

$\{\langle x, 5/x\rangle \mid x \in \mathbb{Z} \wedge x \neq 0\}$

The graph is **not a picture**!

Undefinedness

$x/0$

Activity

Decidability Problems

A **decidability problem** is a question with a **yes** or **no** answer about a **particular input**.

"Is x prime?"

In CS, we care about whether there is an **algorithm** for solving decidability problems.

If there is **no algorithm**, then the problem is **undecidable**.

The Halting Problem

Decide whether program **P** halts on input **x**.

Given program P and input x,

`Halt(P,x)` = { returns `true` if `P(x)` halts; returns `false` otherwise

How might this work?

Clarifications:

`P(x)` is the output of program `P` run on input `x`.

The type of `x` does not matter; assume `string`.

The Halting Problem

Decide whether program **P** halts on input **x**.

Given program P and input x,

`Halt(P,x)` = { returns `true` if `P(x)` halts; returns `false` otherwise

How might this work?

Fact: it is provably impossible to write `Halt`

Notes on the proof

We use two key ideas:

- Function **evaluation by substitution**
- **Reductio ad absurdum** (proof form)

Notes on the proof

The *form* of the proof is ***reductio ad absurdum***.

Literally: “reduction to absurdity”.

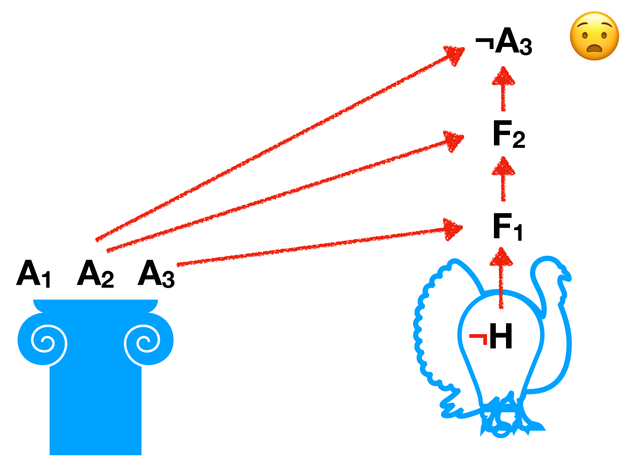
Start with **axioms** and **presuppose the outcome** we want to show.

Then, following strict rules of logic, **derive new facts**.

Finally, derive a fact that **contradicts** another fact.

Conclusion: the **presupposition must be false**.

Reductio ad Absurdum



Function Evaluation by Substitution

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

`addone(1)`	`λx.(+ x 1)1`
`[1/x]x + 1`	`[1/x](+ x 1)`
`1 + 1`	`(+ 1 1)`
`2`	`2`

The Halting Problem

Notes on the proof:

The proof relies on the kind of **substitution** that we’ve been using to “compute” functions in the lambda calculus.

Remember: **we are looking to produce a contradiction**.

The proof is hard to “understand” because the facts it derives **don’t actually make sense**. Don’t read too deeply.

The Halting Problem: Proof

Suppose:

`Halt(P,x)` = { returns `true` if `P(x)` halts; returns `false` otherwise } — `Halt` always halts!

Construct:

`DNH(P)` = { if `Halt(P,P)` is `true`, `while(1){}`; returns `false` otherwise } — `DNH` does not always halt!

The Halting Problem: Proof

Observations so far:

`DNH(P)` will run forever if `Halt(P,P)` is `true`.
`DNH(P)` will halt if `Halt(P,P)` is `false`.

Rewrite:

`DNH(P)` = { if `Halt(P,P)` is `true`, `while(1){}`; returns `false` otherwise }

The Halting Problem: Proof

Observations so far:

`DNH(P)` will run forever if `Halt(P,P)` is `true`.
`DNH(P)` will halt if `Halt(P,P)` is `false`.

Rewrite:

`DNH(P)` = { if `P(P)` halts, run forever; returns `false` otherwise }

The Halting Problem: Proof

Observations so far:

`DNH(P)` will run forever if `Halt(P,P)` is `true`.
`DNH(P)` will halt if `Halt(P,P)` is `false`.

Rewrite:

`DNH(P)` = { if `P(P)` halts, run forever; halt }

The Halting Problem: Proof

Observations so far:

`DNH(P)` will run forever if `P(P)` halts.
`DNH(P)` will halt if `P(P)` runs forever.

Rewrite:

`DNH(P)` = { if `P(P)` halts, run forever
halt

The Halting Problem

Isn't `DNH` itself a program?

What happens if we call `DNH(DNH)`?

`P = DNH`

`DNH( P )` will run forever if `P ( P )` halts.
`DNH( P )` will halt if `P ( P )` runs forever.

The Halting Problem

Isn't `DNH` itself a program?

What happens if we call `DNH(DNH)`?

`P = DNH`

`DNH(DNH)` will run forever if `DNH(DNH)` halts.
`DNH(DNH)` will halt if `DNH(DNH)` runs forever.

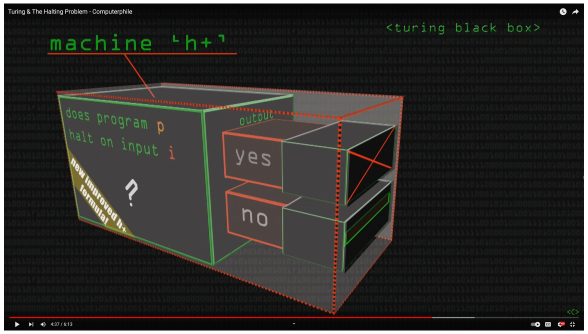
This literally makes no sense. **Contradiction!**

What was our one assumption? `Halt` **exists**.

Therefore, the `Halt` function **cannot exist**.

Need more explanation?

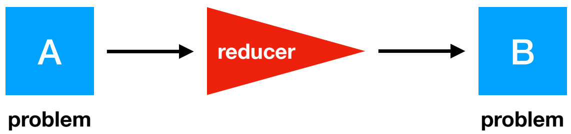
Watch this!



https://youtu.be/macM_MtS_w4

Reductions

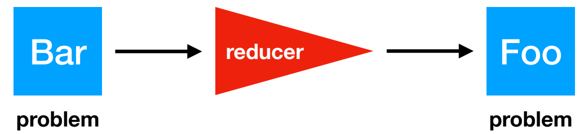
A **reduction** is an **algorithm** that transforms an instance of one problem into an instance of another. Reductions are often **employed to prove something** about a problem given a similar problem.



Reductions

Reductions are often used in a **counterintuitive** way.

For example, if we **want to know whether problem Foo is impossible**, we assume Foo is possible, and then use that fact to show that problem Bar (which **we already know** to be impossible) **appears to be possible**.

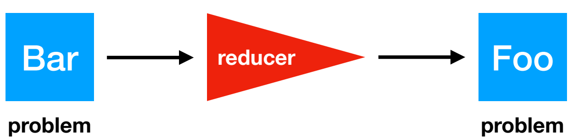


The above is a **contradiction**, meaning that **Foo is not possible**.

Reductions

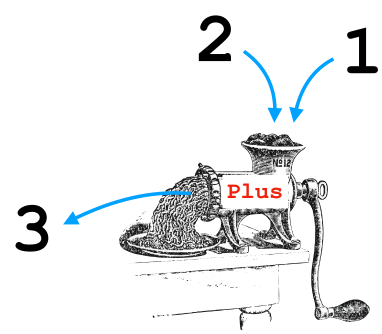
An important part of a reduction is that the reducer be an **ordinary algorithm**.

The reducer **should not solve the problem**. A reducer just converts problems from one form to another.



You will get **a lot** more exposure to reductions in CSCI 361.

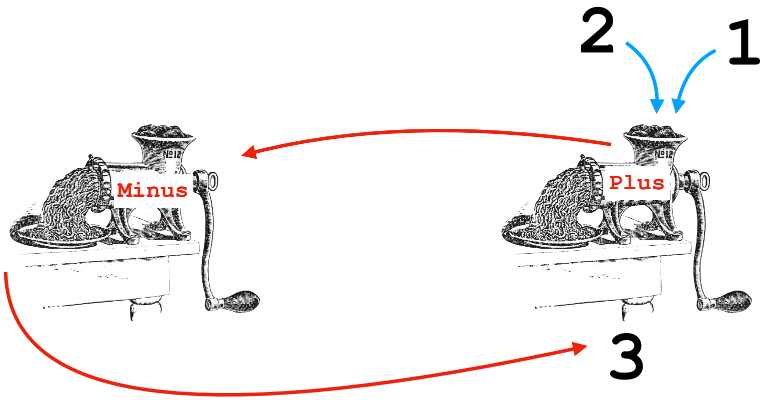
Reductions



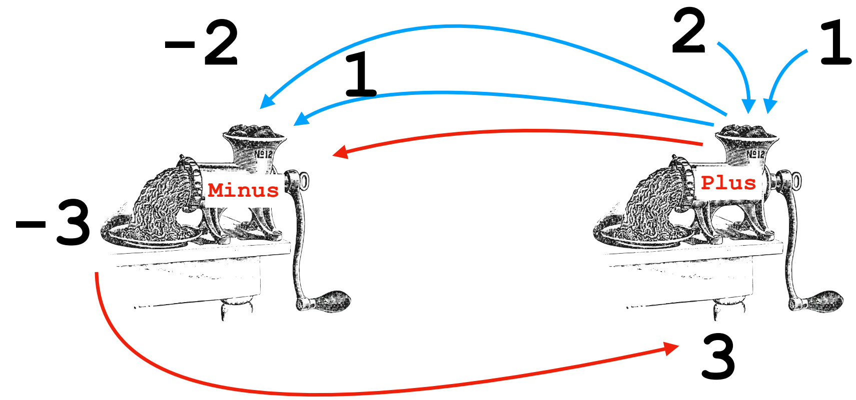
The humble algorithm.

(sorry, vegetarians)

Reductions



Reductions




```
let reducer(x: int)(y: int) = -(-x-y)
```

Recap & Next Class

Today:

- More lambda reductions
- Function graphs
- Decidability

Next class:

- Consequences of computability for PL design