CSCI 334: Principles of Programming Languages

Lecture 12: Computability

Instructor: Dan Barowy

Williams

Topics

Halting problem

Reductions

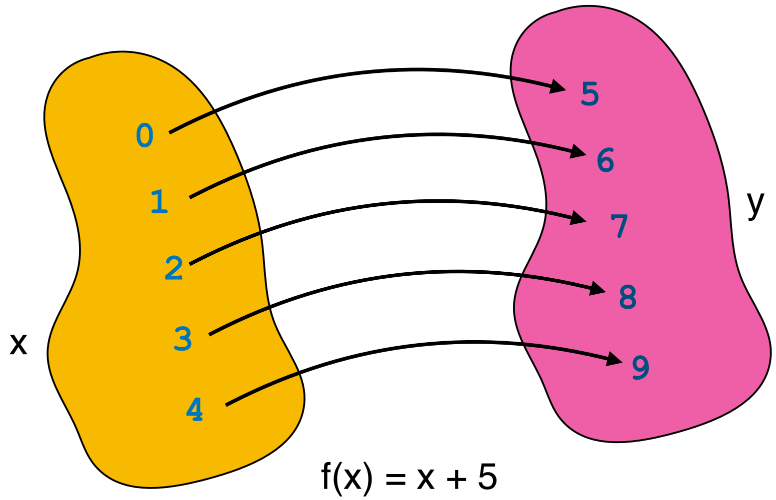
Your to-dos

1. Lab 5, **due Sunday 3/13** (partner lab)
   (last one before midterm!)
2. No reading response next week!

Announcements

- **Midterm exam**, in class, Thursday, March 17.

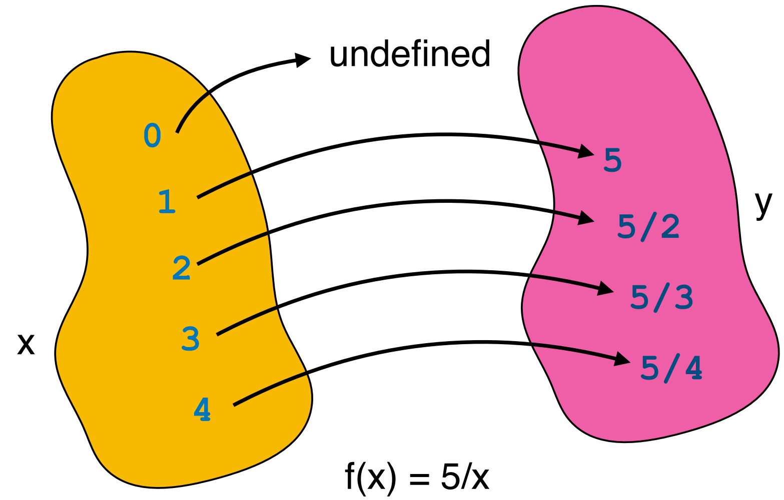
Intuition: total function



f(x) = x + 5

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

Intuition: partial function



f(x) = 5/x

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

The Halting Problem

Decide whether program P halts on input x.

Given program P and input x,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

How might this work?

<u>Clarifications:</u>

`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,

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

How might this work?

Fact: it is provably impossible to write `Halt`

Notes on the proof

We utilize 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”.

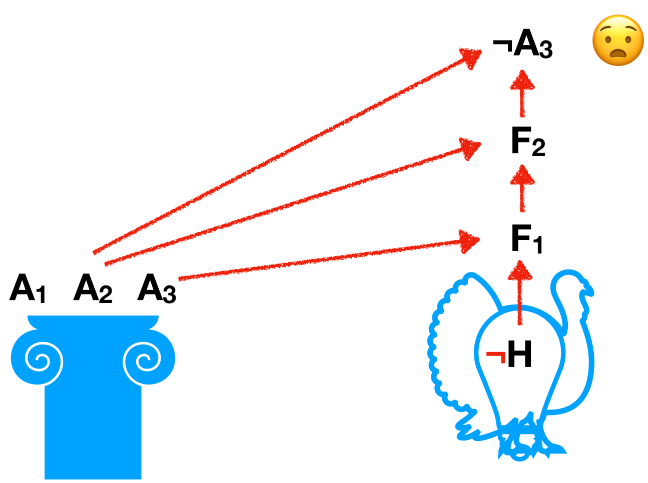
Start with **axioms** and **presuppose the outcome** we want to show.

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:

$$\texttt{Halt(P,x)} = \begin{cases} \text{returns } \texttt{true} \text{ if } \texttt{P(x)} \text{ halts} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

`Halt` always halts!

Construct:

`DNH` does not always halt!

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{Halt(P,P)} \text{ is } \texttt{true}, \texttt{while(1)\{\}} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{Halt(P,P)} \text{ is } \texttt{true}, \texttt{while(1)\{\}} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{P(P)} \text{ halts, run forever} \\ \text{returns } \texttt{false} \text{ otherwise} \end{cases}$$

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{P(P)} \text{ halts, run forever} \\ \text{halt} \end{cases}$$

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:

$$\texttt{DNH(P)} = \begin{cases} \text{if } \texttt{P(P)} \text{ halts, run forever} \\ \text{halt} \end{cases}$$

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**.

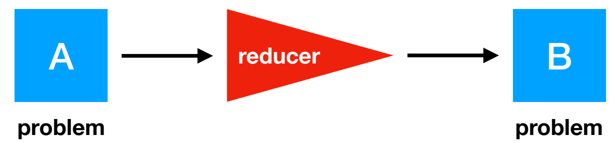
Therefore, the `Halt` function **cannot exist**.

The Halting Problem

… helps us to understand many other problems.

Reductions

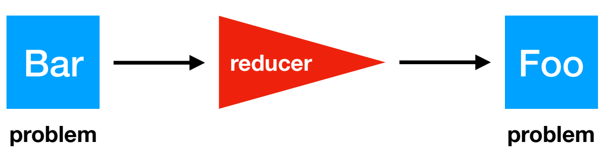
A **reduction** is an **algorithm** that transforms an instance of one problem into an instance of another. Reductions are often **employed to prove something** about a problem given a similar problem.



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**.

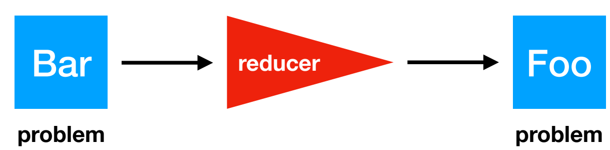


The above is a **contradiction**, meaning that **Foo is not possible**.

Reductions

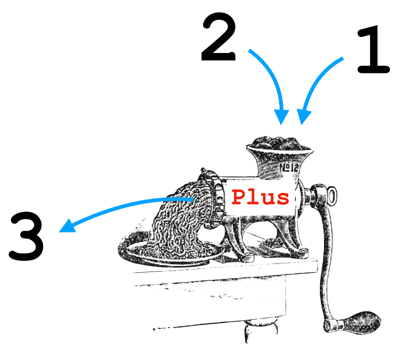
An important part of a reduction is that the reducer be an **ordinary algorithm**.

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.

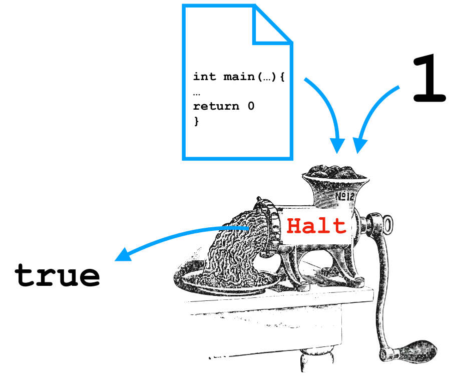
Reductions



The humble algorithm.

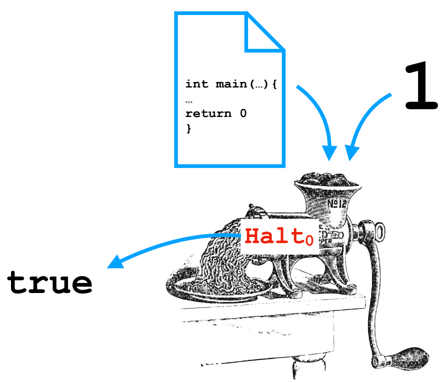
(sorry, vegetarians)

Reductions



We **know** that `Halt` is not computable.

Reductions



A function `f(i)` **halts not** if and only if `f` **does not halt** on input `i`.

Is `Halt`$_0$ computable?

Reductions

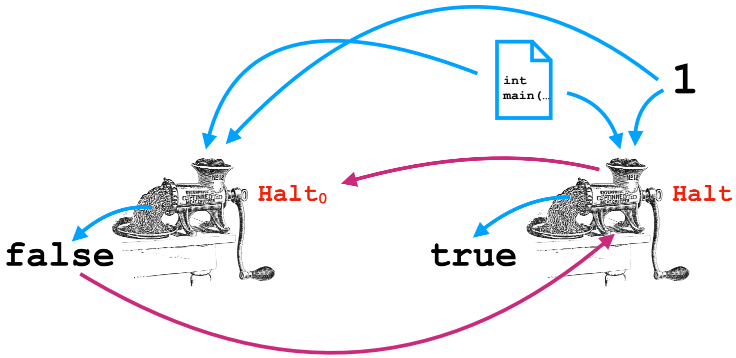
A function `f(i)` **halts not** if and only if `f` **does not halt** on input `i`.

If `Halt`$_0$ is computable, couldn't we do this?

Assume that `Halt`$_0$ is computable.
(e.g., it's in your standard library)

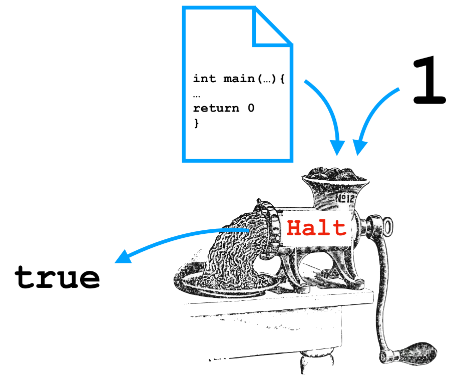
```
def halt(f, i):
  return not halt0(f, i);
```

Reductions



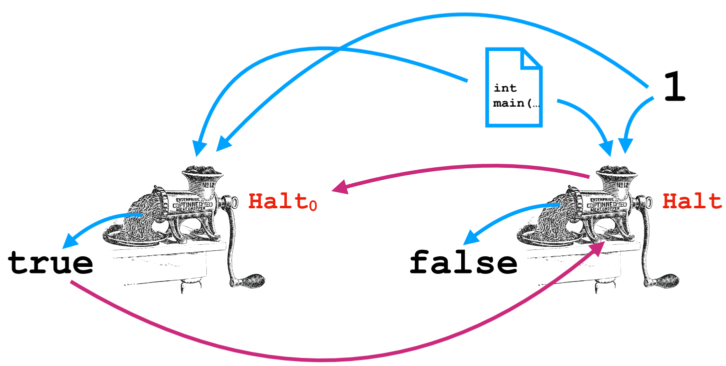
Reduction: **Construct Halt** using **Halt$_0$**.

Reductions



We know that **Halt** is not computable.

Reductions



If we can build this new machine,
what does that mean for **Halt$_0$**?

Halt$_0$ is **not computable**.

Reductions

We can use the Halting Problem to show that other problems cannot be solved **by reduction** to the Halting Problem.

We cannot tell, in general…

… if a program will **run forever**.

… if a program will **eventually produce an error**.

… if a program **is done using a variable**.

… if a program **is a virus**!

Generality

```
def myprog(x):
  return 0
```

```
def Halt(f,i):
  if(f = "def myprog(x):\n\treturn 0"):
    return true
  else
    return false
```

The Halting Problem is about **an arbitrary program**.

Recap & Next Class

Today:

The Halting Problem

Reductions

Next class:

Midterm Q&A