CSCI 334:
Principles of Programming Languages

Lecture 10: Proof by Reduction

Instructor: Dan Barowy

Williams

Topics

Proof techniques

Halting problem

Reductions of HP to other problems

Your to-dos

1. Lab 5, **due Wednesday 3/12** (partner lab)
2. Start studying for the midterm

Announcements

- CS Colloquium this **Friday, Mar 14 @ 2:35pm in Wege Auditorium (TCL 123)**

Effective Evaluations of Large Language Models for Complex Problems

Prof. Niranjan Balasubramanian (Stony Brook)

Evaluation efforts are central to understanding and improving the capabilities of Large Language Models on complex problems. In the first part of this talk, I will present a set of ideas to create effective complex reasoning tasks for a range of tasks such as question answering, theory-of-mind analyses, procedure understanding, and biomedical natural language inference. In the second half, I will present our recent work on developing a rigorous benchmark that can test automation for solving complex tasks involving commonly used apps.

Decidability Problems

Computable: the algorithm returns an answer in a **finite** amount of time.

Total: the output of the algorithm is defined for every possible input.

A problem is **decidable** if it is both **computable** and **total**.

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

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

… helps us to understand the difficulty of many other problems.

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)

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`

Reductio ad absurdum

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

Literally: “reduction to absurdity”.

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

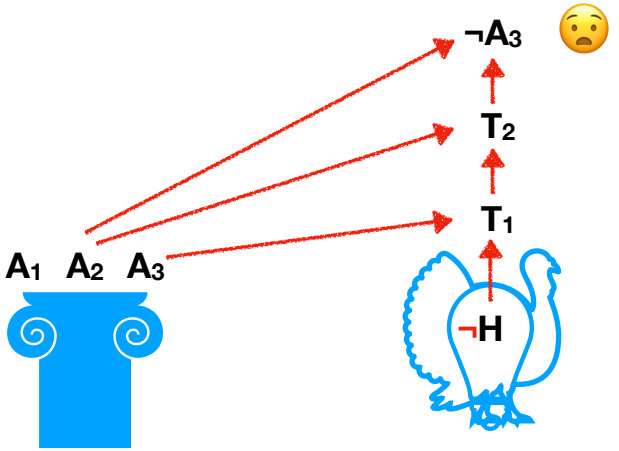
Then, following strict rules of logic, **derive new facts**.

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

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

Reductio ad Absurdum

One of the axioms is "T ∧ ¬T is false.”



Reductio ad Absurdum

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**. **We don't want them to 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` does not always halt!

`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 `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.

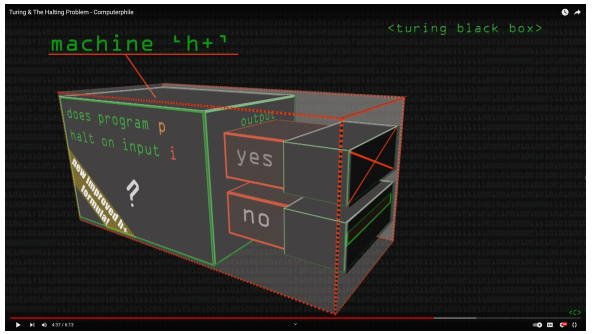
This literally makes no sense. **Contradiction!**

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

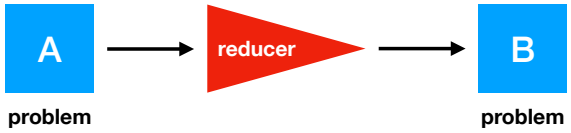
We can use the Halting Problem to show that other problems cannot be solved **by reduction** to the Halting Problem.

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**!
- … and **many more** properties!

Reductions

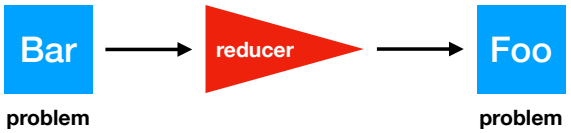
A **reduction** is an **algorithm** that transforms an instance of one problem into an instance of another. Reductions are often **employed to prove something** about a problem given a similar problem.



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**.

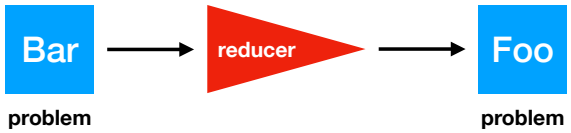


The above is a **contradiction**, meaning that **Foo is not possible**.

Reductions

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

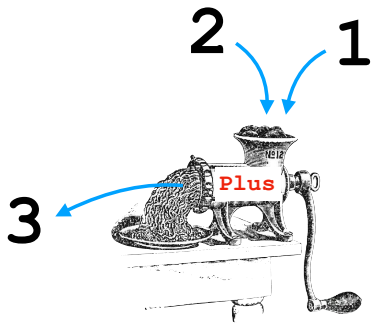
The reducer **should not solve the problem**. A reducer just converts problems from one form to another.



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

(this is a wonderful and mind-expanding idea and I'm glad I pushed myself to take a course on computability theory)

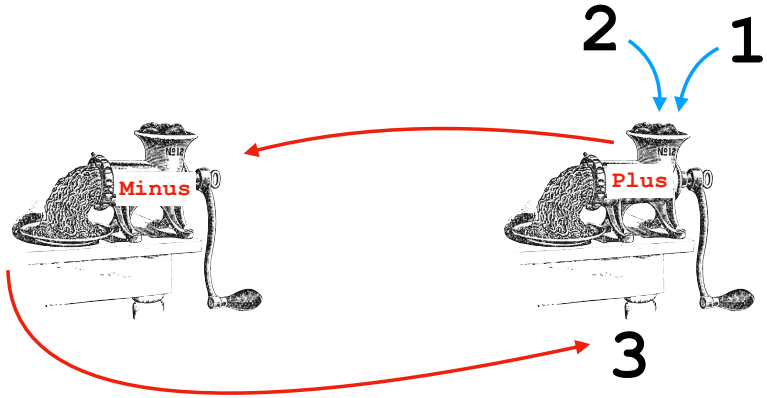
Reductions



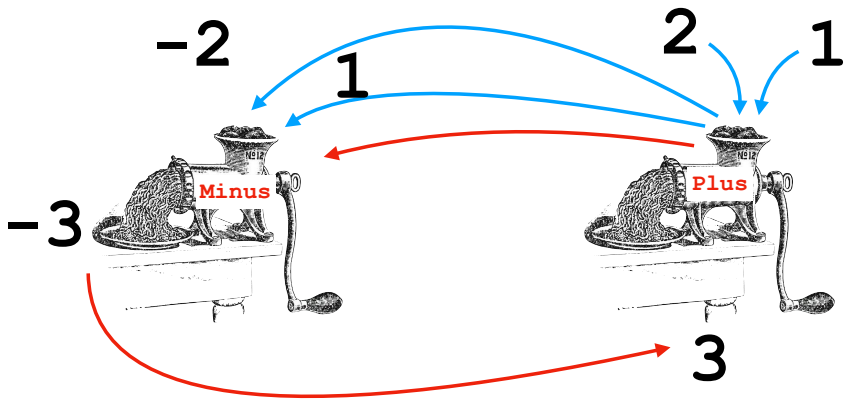
The humble algorithm.

(sorry, vegetarians)

Reductions

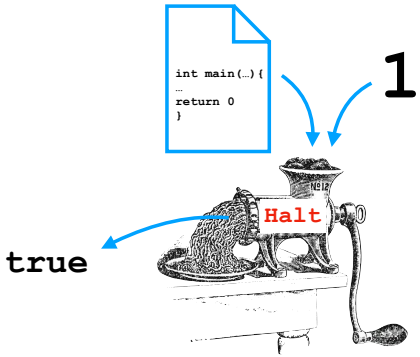


Reductions



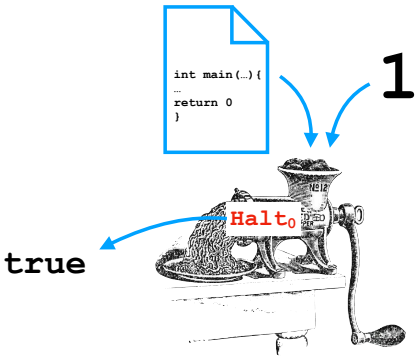

```
let reducer(x: int)(y: int) = -(-x-y)
```

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 $\texttt{Halt}_0$ computable?

Reductions

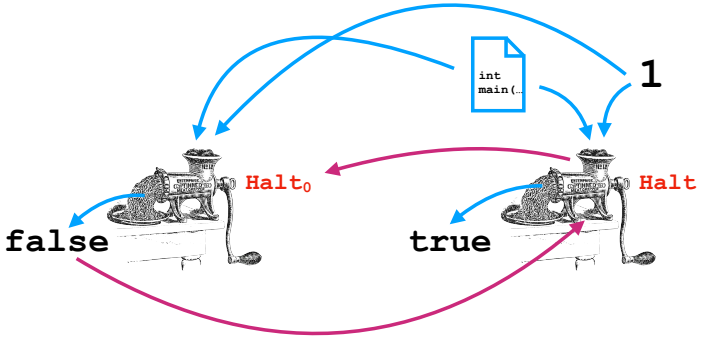
A function `f(i)` **halts not** if and only if `f` **does not halt** on input `i`.

If $\texttt{Halt}_0$ is computable, couldn't we do this?

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

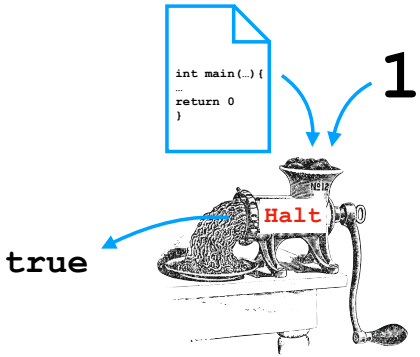
```
def halt(f, i):
  return not halt₀(f, i);
```

Reductions



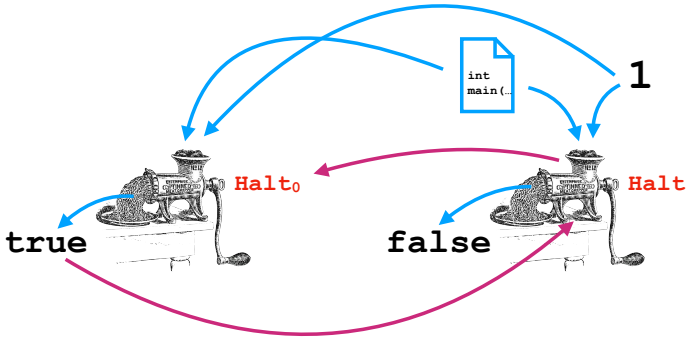
Reduction: **Construct** `Halt` using `Halt`$_0$.

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**.

Why does this proof work?

The proof relies on the logical implication,

A ⇒ **B**

In plain language, we read this as "if **A** is true, then **B** is true."

For example, one (true) logical implication is:

it is sunny ⇒ **it is not cloudy**

Why does this proof work?

But just look outside, and it is

Combined with "**it is sunny ⇒ it is not cloudy**",

what can we conclude?

Why does this proof work?

So logical implications can be used in two different ways.

A ⇒ B

If you know that **A** is **true**, then you also know that **B** is **true**.

If you know that **B** is **false**, then you also know that **A** is **false**.

Why does this proof work?

The proof relies on the logical implication,

`Halt`$_0$ is computable ⇒ `Halt` is computable

Which is clearly a **true** statement, since we can actually construct a function that computes `Halt` if we are given a function that computes `Halt`$_0$.

But we know that `Halt` is not computable.

So…

Reduction Activity

Recap & Next Class

Today:

- Proof techniques
- Halting problem
- Reductions

Next class:

- How to construct a programming language