CSCI 334:
Principles of Programming Languages

Lecture 11: Undecidability

Instructor: Dan Barowy

Williams

1

Topics

The Halting Problem

Reductions

2

Your to-dos

1. Lab 5, **due Monday 03/11** (partner lab)
2. Start studying for the midterm

3

Announcements

- **Midterm exam**, in class, Thursday, March 14.
- Colloquium: **Thinking About Graduate School?** 2:35pm in Wege Auditorium.

CS faculty will discuss your burning questions about graduate school including; deadlines, personal statements, finding an advisor, research, application process, and choosing the right school.

4

Quiz

5

Decidability Problems

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

"Is x prime?"

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

Generally, we want algorithms that work for **all inputs in a domain**.

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

6

Now that we have some terminology to discuss functions, let's turn to decidability.

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`.

7

Perhaps the most famous problem in decidability: the halting problem. Can you write a program that can tell you whether another program halts on a given input? We want this function to work for all possible programs.

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`

8

This turns out not to be possible. Recall the dream of Leibniz, Hilbert, etc: that maybe we could make a machine that could answer any logical question. There's nothing obviously illogical about this question. So to discover that it is fundamentally impossible was seriously disappointing to many.

Notes on the proof

We use two key ideas:

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

9

Let's walk through the proof. We need to know about two proof techniques.

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`

10

You've seen evaluation by substitution before. Not only it is the way functions really work (see Python function on the left), it's how our foundational model about computation works too. Functions are essentially substitution machines.

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**.

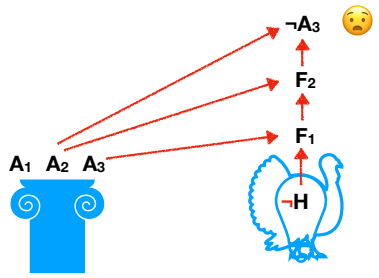
Finally, derive a fact that **contradicts** another fact.

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

11

Reductio ad absurdum is an important proof technique and the proof about the Halting Problem relies on it.

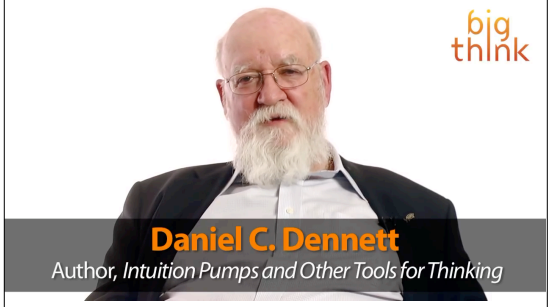
Reductio ad Absurdum



12

The basic idea is to start by assuming the inviolable truths (the "axioms"; the As on the pillar) and also the claim (the "hypothesis," H) you want to disprove. Then, using the rules of logic, we combine our axioms and our hypothesis to derive new facts (the Fs). If we've done this work correctly, and H truly is false, then we will derive a contradiction. One basic axiom is a statement like "x is true and x is false" cannot be true. So if we derive a contradiction like that, then our hypothesis must be false.

Reductio ad Absurdum



https://www.youtube.com/watch?v=sVUMAqMmy7o

13

Here's another person saying the same thing, using Galileo's famous reductio ad absurdum proof that heavy objects cannot fall faster than light objects.

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.

14

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

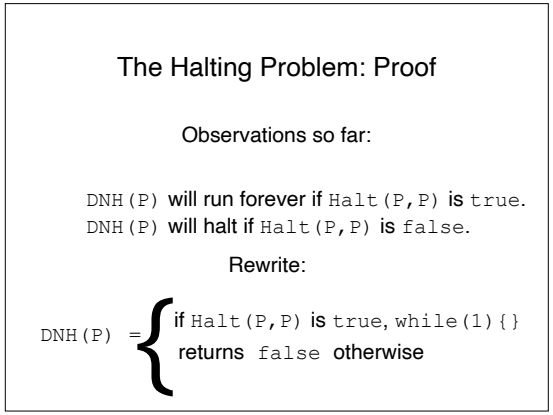
15

We define Halt simply. Observe that Halt is a total function. Since we want to prove something about Halt, using reductio ad absurdum, we assume that we have a Halt function. Imagine that it is in your programming language's standard library. E.g., we can write 'import java.util.Process.*' and there we have it.

We also define another function DNH. Observe that DNH is partial, because it Does Not Halt when Halt(P,P) is true. Also observe that we did

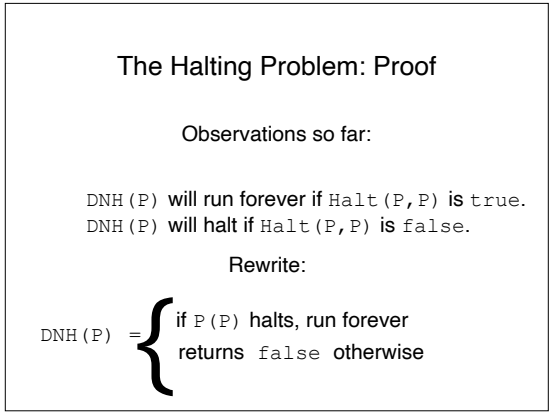
not do anything that violates the rules of logic (or of programming languages). We can easily write DNH in our language of choice (e.g., Java) provided that Halt is in our standard library.

You might be wondering why the heck we would want to call Halt(P,P), but hang on for a minute. Trust me that there's a good reason.



16

So here I just restate in plain English what our DNH definition tells us.



17

In fact, let's simplify our DNH definition a little more.

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

18

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

19

But hey, we can update our definition too. Let's make it as simple as possible.

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.

20

So here's a simple, but probably not obvious observation about what we have so far. If Halt made it possible to write DNH, and DNH can take in ANY PROGRAM, and DNH is itself a program, couldn't we call DNH with itself?

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**.

21

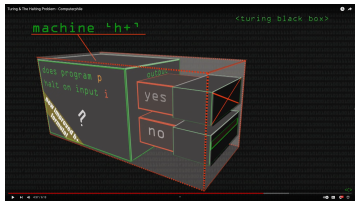
So if P is DNH, what does that mean? Here we use our "evaluation by substitution" trick. We just replace P with DNH.

But, OH NO. What have we done?!!! This makes no sense at all!

Of course, that was our aim all along. Since the only iffy assumption we made was that Halt exists in our standard library, then this means that it CANNOT EXIST!

Need more explanation?

Watch this!



https://youtu.be/macM_MtS_w4

22

If you are new to this style of reasoning, it's a little hard to wrap your mind around it. I encourage you to watch this excellent video to hear the proof explained in a slightly different (but essentially the same) way.

Many of you probably plan to have careers in science or mathematics. This style of reasoning is very common in technical fields, so you will be well served by learning to become comfortable arguing this way. It might not help you win any arguments with your partner, but it will certainly help

---

The Halting Problem

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

23

Aside from the fact that undecidability and the halting problem in particular are interesting, a big reason why we care about these ideas is that decision problems pop up all the time in real life. For example the question "are we done using this variable?" pops up when we want a programming language to automatically manage memory resources for us, a problem we call "garbage collection." Another example is "if I install this program, will it harm my computer?" which pops up in computer virus detection. Can we solve those problems?

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**!

24

The shocking thing is… NO! We actually cannot solve garbage collection or virus detection in their full generality. And if you can recognize that fact early, you can either avoid a lot of pain and false promises, or you can take proactive steps to change the problem a little to make some solutions feasible. Programming languages DO automatically manage memory, and the tradeoff is to do it imprecisely. Virus scanners DO detect viruses, and the tradeoff is generality (i.e., by constraining the set of programs that we analyze).

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 **any arbitrary program**.

25

So it's important to remember that the Halting Problem, and many other questions about decidability, are the strongest possible form of those questions. The Halting Problem considers whether it is possible to write an algorithm that can tell you whether ANY PROGRAM halts. We know that we cannot do that. But if you restrict your domain; if you are willing to sacrifice generality, of course there are some solutions. E.g., I can easily detect whether this specific program halts. And if I know my lambda calculus, I might even be able to detect when somebody tries to slip the same function with different names past me. But it will never work for ANY program.

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.

26

So how DO we go about determining whether one kind of problem tells us about another kind of problem? The answer is that we write a proof called a "reduction." This is a different kind of reduction than the reductions we use in the lambda calculus (the name is an unfortunate coincidence). A reduction simply shows that a reducer exists that translates problems of type A into problems of type B. If we have a solver for problems of type B, great, we can now solve problems of type A too.

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**.

27

When we want to show that something is impossible, we use reductions in a counterintuitive way. For example, suppose we want to show that Foo is not possible. We construct a reduction such that the thing we KNOW something about already (e.g., we know that Bar) is the problem that gets reduced. Why? Because if it turns out that we CAN reduce a Bar to a Foo, and we assume that we have a Foo, then it means that we can also have a Bar. But we KNOW that we can't have a Bar. Therefore we cannot have a Foo.

I go into this argument in more detail in the "Proof by Reduction" chapter of the course packet. Have a look. I kept it short and, hopefully, it will help you understand this in more detail.

Reductions

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

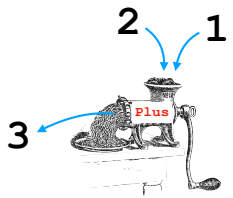
The reducer **should not solve the problem**. A reducer just converts problems from one form to another.

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

28

Remember that a reducer should be an ordinary algorithm. And it just changes the form of the problem. It does not attempt to solve it. Foo is going to be doing the solving.

Reductions



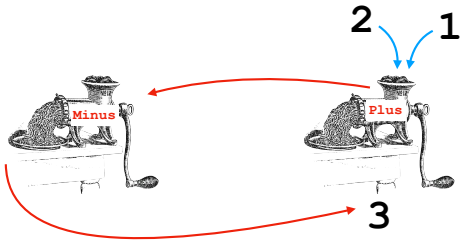
The humble algorithm.

(sorry, vegetarians)

29

Here's a super concrete example of a reduction. Suppose we want to build a Plus machine.

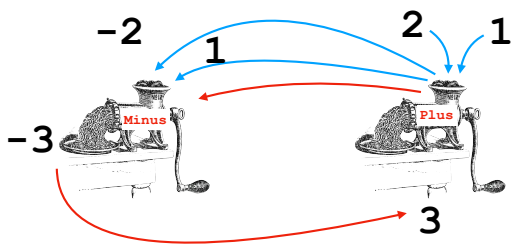
Reductions



30

But, because we didn't have much money, we bought a computer that can only subtract. Can we still add with that machine? In other words, can we reduce the problem of adding to the problem of subtracting?

Reductions

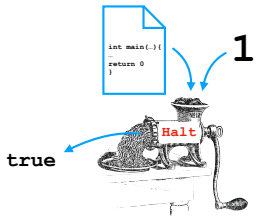



```
let reducer(x: int)(y: int) = -(-x-y)
```

31

Yes! In fact, here's a reduction that does that. Observe that our reducer is an ordinary function. No tricks.

Reductions

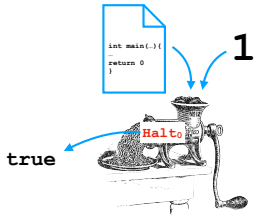


We **know** that `Halt` is not computable.

32

So, turning back to the Halting problem, we know that Halt is not computable. Put that in your back pocket so that you can whip it out at a moment's notice.

Reductions



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

Is $\texttt{Halt}_0$ computable?

33

Here's a (possibly silly) question. Is Halt-not computable? A function halts not if and only if it does not halt on input i.

(this turns out to be the inverse of the statement of the Halting problem)

The answer to this might be obvious to you, but how do we really prove it one way or the other?

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?

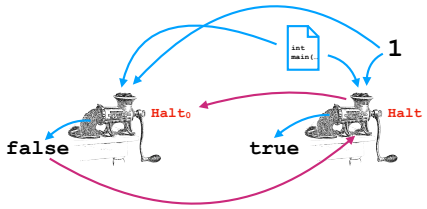
Assume that $\texttt{Halt}_0$ is computable.
(e.g., it's in your standard library)

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

34

Let's try a reduction. Remember, a reduction is just a function. Can we convert a problem of type Halt into a problem of type Halt-not? Again, recall, we make it work in this direction so that we can derive the right kind of contradiction.

Reductions

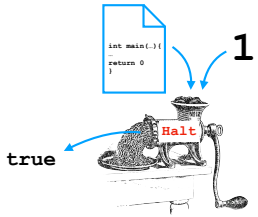


Reduction: **Construct** `Halt` using $\texttt{Halt}_0$.

35

Graphically, it works something like this.

Reductions

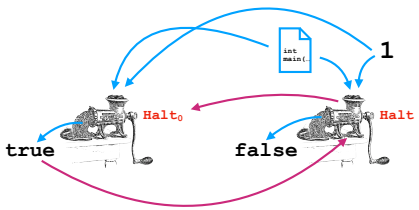


We know that `Halt` is not computable.

36

To really belabor the point. Remember that we can't do this.

Reductions



If we can build this new machine, what does that mean for `Halt₀`?

`Halt₀` is **not computable**.

37

But… our reduction, which was an ordinary program, showed that it was possible to construct a Halt problem. Our only assumption was that Halt-not existed. Therefore, it might be the case that Halt-not exists. Halt-not is undecidable.

We will do more of these proofs after spring break. For now, just appreciate how cool (or disappointing) it is that we know that we cannot solve certain kinds of problems. I personally think that the fact that many problems are undecidable is one of the constraints that makes computer science an interesting challenge. And we'll talk about how people address these problems—particularly garbage collection—after the break.

Recap & Next Class

Today:

- The Halting Problem
- Reductions

Next class:

- Midterm Review

38