CSCI 334:
Principles of Programming Languages

Lecture 12: Reduction Proofs

Instructor: Dan Barowy
Williams

1

Topics

Refresher: The Halting Problem

Reductions

Garbage Collection

2

Your to-dos

1. Lab 6, **due Monday 4/8** (partner lab)
2. Be sure to read *Proof by Reduction* **before Thursday**.

3

Announcements

- **WCMA visit**, Thursday, April 11.
- Colloquium: **Algorithmic Approaches to Subset Sum (and Other Hard Problems)** at 2:35pm in Wege Auditorium.

The Subset Sum problem is the most fundamental NP-complete problem concerned with adding numbers together. However, progress on exact algorithms for this problem has been slow: Since Horowitz and Sahni's 1974 invention of the "Meet-in-the-Middle" approach, our best algorithms have relied on simple enumeration and dynamic programming strategies. The lack of an algorithm for Subset Sum that leverages our modern understanding of addition points to important gaps in our knowledge about the behavior of the integers.

4

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`.

5

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

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

6

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**!

7

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).

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.

8

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**.

9

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.

10

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

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

Is $\texttt{Halt}_0$ is computable?

11

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

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?

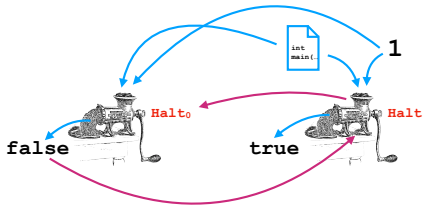
Assume that $\texttt{Halt}_0$ is computable.
(e.g., it's in your standard library)

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

12

Solution. In a way, a reduction is just a wrapper function. Notice that the *direction* of the reduction is a little counterintuitive. The way we've constructed it, we are letting the user determine whether a function halts on a given input by calling Halt0. Not the other way around.

Reductions

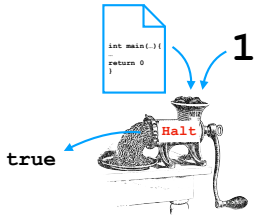


Reduction: **Construct** `Halt` using $\texttt{Halt}_0$.

13

Graphically, it works something like this.

Reductions

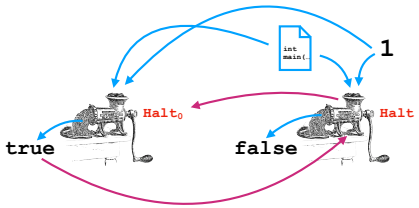


We know that `Halt` is not computable.

14

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 $\texttt{Halt}_0$?

$\texttt{Halt}_0$ is **not computable**.

15

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.

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

16

Let's try to understand what makes the proof tick. The first idea is that of a "logical implication." A logical implication is a statement of the form "if A then B" which we conventionally write as A ⇒ B. Logical implications themselves are statements that can be true or false. But if the implication is valid, it lets us form chains of reasoning. For example, a true logical implication is "if it is sunny then 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?

17

Implications can be used in two ways though. For example, if we look outside, we can see that it is CLOUDY. In other words it is true that "not (it is not cloudy)". Therefore, we can conclude that "not (it is sunny)".

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**.

18

More generally, if you know that A is true, then an implication tells you that B is true.
Conversely, if you know that B is false, then an implication tells you 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…

19

So our reduction proof works because our implication is “if Halt0 is computable then Halt is computable.” That’s a true statement demonstrated by the fact that we were able to write a Halt function that calls Halt0. The key thing to know about logical implication is that it just ties the truth of two facts together. By itself, it says nothing about whether either of them is actually true or false.

That said, we ALREADY KNOW “not (Halt is computable)”. Therefore “not

---

Reduction Activity

20

Try one on your own.

Garbage collection

A **garbage collection algorithm** is an algorithm that determines whether the storage, occupied by a value used in a program, **can be reclaimed for future use**. Garbage collection algorithms are often tightly integrated into a programming language **runtime**.

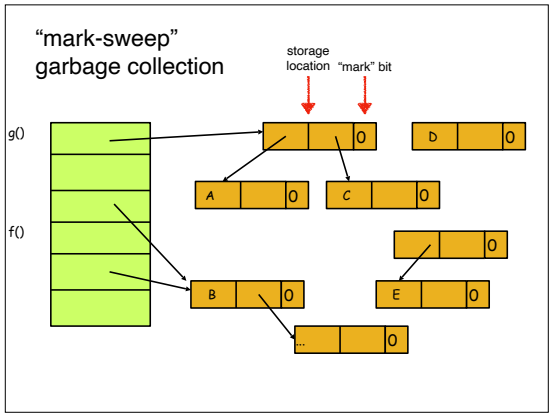
21

One of the most famous uncomputable problems is garbage collection. But it's only uncomputable when stated in a very strong form, known as "precise garbage collection." In practice, because precise garbage collection it not computable, we solve a weaker form of garbage collection. We solve it imprecisely. Exactly why imprecise garbage collection is OK is the subject of this week's lab.

John McCarthy

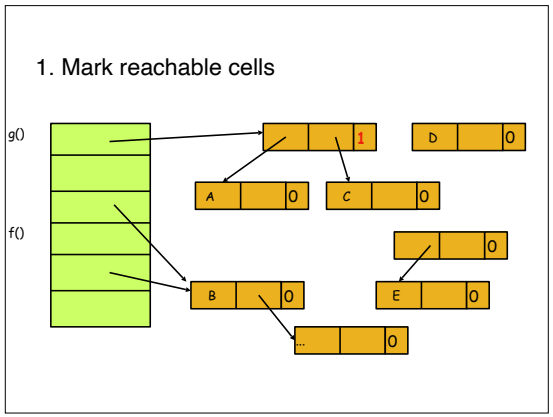
22

Garbage collection (GC) was invented by John McCarthy when he invented LISP. The reason GC was needed was because he did not want his programmers to have to worry about memory management and related bugs while they were trying to solve other trough questions about how artificial intelligence should work. He invented two algorithms, "reference counting GC" and "mark-sweep GC." Each has a tradeoff. Reference counting is very fast. Mark-sweep is comparatively slow. However, reference counting cannot collect object graphs containing cycles (e.g., it cannot correctly collect something like a doubly-linked list). Which GC algorithm your language uses depends on the goals of the language designer. You've probably mostly encountered mark-sweep (or one of its variants) because that's what Python, Java, etc use.



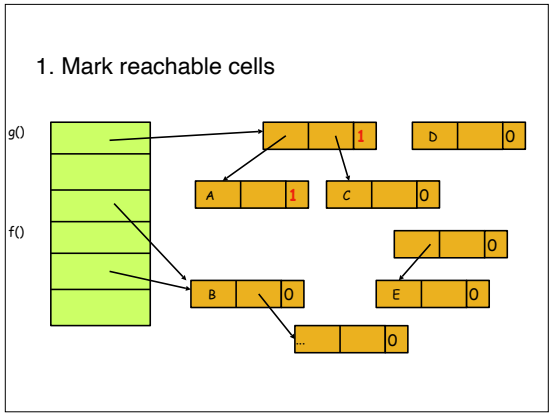
23

Here I have a simple memory layout with a call stack (to the left) and a heap (to the right). Recall that the call stack keeps track of active functions, and it stores each function's local variables. If a variable refers to an object, the object itself is actually stored on the heap, and the variable on the stack simply stores a pointer to the object. Since objects themselves can also store references to other objects, what you have is an object graph that looks a bit like this. In reality, all of these data structures are stored in one big array, because that's what memory is. But you also know that data structures are all about abstraction, and so here's the memory abstraction that a programming language provides.

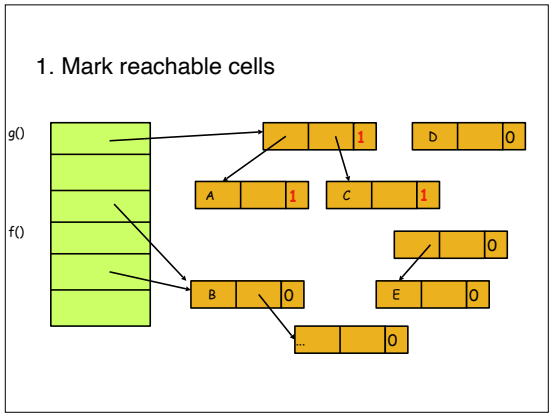


24

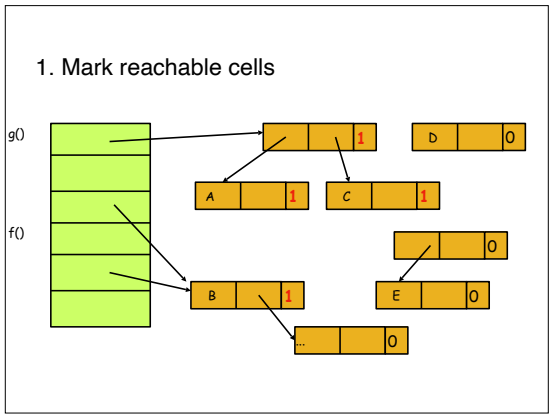
When the mark-sweep algorithm is called, it starts by tracing references on the stack into the object graph in the heap. For every reachable object, it flips the "mark bit" from 0 to 1.



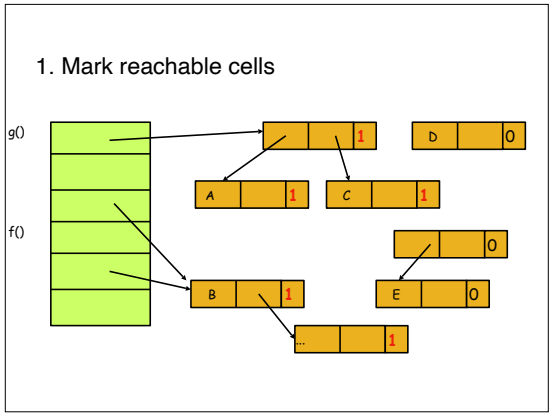
25


1. Mark reachable cells
g()
f()
1
D
0
A
1
C
1
0
B
0
E
0
...
0

26

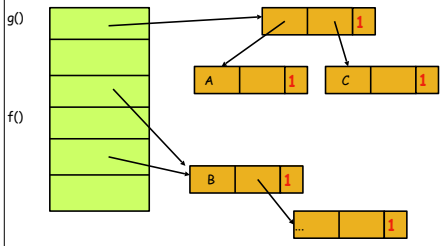

1. Mark reachable cells
g()
f()
1
D
0
A
1
C
1
0
B
1
E
0
...
0

27


1. Mark reachable cells
g()
f()
1
D
0
A
1
C
1
0
B
1
E
0
...
1

28

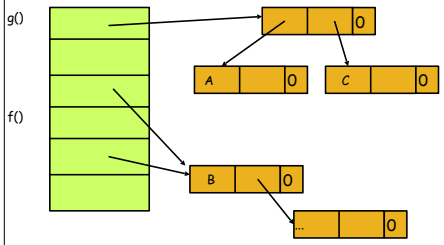
2. Free (“sweep”) unreachable cells



29

At the conclusion of the “marking” phase, all of the objects with 0’s in their mark bits are reclaimed for reuse (or “swept”).

3. Clear tags



30

Finally, all of the 1s are reset back to 0s so that the algorithm can start over.

(students sometimes ask: couldn’t you just skip the last step and remember that 1 now means “unmarked”? and the answer is, definitely, yes, that’s a nice optimization)

Recap & Next Class

Today:

- The Halting Problem
- Reductions
- Garbage Collection

Next class:

- Type inference

31