(Provisional) Lecture 36: Debugging and Divide and Conquer

10:00 AM, Dec 1, 2017

Contents

Objectives

By the end of this lecture, you will know:

- how to better debug your code
- how to apply the concept of divide and conquer to the infinite domain

1 Debugging

Some of you may have noticed that the code we wrote in the last class to find diagonals doesn't work.

However, the ideas from the class do work. The point of CS17 has been to make you good at translating the ideas to actual programs.

So what do you do when your code doesn't work?

You could either try to debug your code, or you could throw away (or comment out) your code, and write it again. Often, debugging your code takes a lot more time and effort than simply rewriting your code.

If you do choose to debug, here are some helpful tips.

- Never walk past a mistake: You should try to fix even the smallest mistake. Even if you find some very trivial mistake that you think couldn't possibly be causing the problem, you should still fix it because even if it isn't causing the problem, fixing it makes it easier to find the real problem.
- Simplify your code: For example, when you've got a big procedure that's producing type errors, try replacing various match cases with `failwith ``dummy error''`. This way, you can try and figure out which match cases were causing the type errors. If you replaced a match

case with this `failwith` and the procedure stops producing type errors, then you know that the error was coming from this match case.

Or, if you have a procedure that involves returning a list, temporarily replace the return value with []. This way, you can figure out if the bug was in the computed return value, or somewhere else in the program.

- If a particular expression has a bug, you can use `#trace`. You can also take that expression and evaluate it on its own through terminal by using `let` to bind the unbound values to something reasonable. If the evaluation works, then you know the problem is outside your expression.

Some other big ideas of CS17 include:

- Scale: Scale is all about what hapens when your problem gets larger and larger. For example, analysis deals with scale, because it is trying to see how algorithms will perform when the problems get very large.

  Modules are also about scale, because it is a way of simplifying and abstracting out code when the codebase gets very large. Modules can help deal with name-collisions and complexity of interactions in large codebases by restricting the visibility of different names and procedures.

- Divide and Conquer: This concept is the idea of dividing a problem into two or more smaller programs, and then solving those. One example of divide and conquer is the concept of dividing a list into its `first` and `rest`, and using recursion to solve problems on lists.

  Another example is splitting a list in the middle and using recursion on the two smaller lists, which results in a logarithmic time complexity instead of a linear time complexity.

- Remember Stuff: Another big idea in CS17 is the concept of remembering stuff in your code through using accumulators or `let` expressions. For example, we used an accumulator in `fast-reverse` in order to remember a partial result, which sped up our porgram from $O(n \mapsto n^2)$ to $O(n \mapsto n)$.

  Additionally, by using `let` expressions, we can avoid making the same recursive call twice, which speeds up our program from $O(n \mapsto 2^n)$ to something much smaller.

2 Divide and Conquer

We're going to apply the divide and conquer concept to a new domain - the infinite domain.

We're going to write a procedure to find a root of a function over a particular interval.

The inputs of this procedure are:

- an interval $a \leq x \leq b$ of the real number line, denoted $[a, b]$
- a continuous function $f : \mathbb{R} \mapsto \mathbb{R}$, represented by a `float -> float` procedure, with $f(a) \geq 0$ and $f(b) \leq 0$.

The output is:

- A number $c \in [a, b]$ with $f(c) = 0$

In other words, given this continuous function f that is positive at a lower bound and negative at an upper bound, we want to find where this function f crosses the x-axis between the lower and upper bounds.

The Intermediate Value Theorem in calculus says that if f is a continuous function on an interval $[a, b]$ with $f(a) \geq 0$ and $f(b) \leq 0$, then there's a point $c \in [a, b]$ with $f(c) = 0$.

Therefore, we know that there is an answer c to this problem.

Note: floats are not the same thing as real numbers. This is because floats are represented by 64 bits, so there are only 2^{64} possible floats. In contrast, there are infinitely many real numbers.

How can we find c? There might actually be multiple values of c with $f(c) = 0$. So then, how can we find some number c such that $f(c) = 0$? We actually cannot do so, because floats cannot represent all the possible real numbers. The best we can do is find a very small interval in which c must lie.

Our revised inputs and outputs are:

Inputs:

- an interval $a \leq x \leq b$ of the real number line, denoted $[a, b]$
- a continuous function $f : \mathbb{R} \mapsto \mathbb{R}$, represented by a `float -> float` procedure, with $f(a) \geq 0$ and $f(b) \leq 0$.
- A tolerance h

Output:

- An interval $[d - h, d + h]$, where $d \in [a, b]$, such that for some $c \in [d - h, d + h]$, we have $f(c) = 0$.

An alternative might be to find a number d such that $|f(d)| < h$. However, this might find values where f comes very close to touching the x-axis, but doesn't actually cross it, so we're going to go with the first option.

The solution is to first take the interval $[a, b]$, and check whether $b - a < 2h$. If so, then return this interval.

Otherwise, find the midpoint, q.

If $f(q) = 0$, return the interval $[q, q]$.

Otherwise, instead of solving the problem on the large interval $[a, b]$, we can either solve it on $[a, q]$, or $[q, b]$.

Which interval do we solve it on? We should solve it on the interval $[m, n]$ such that $f(m)$ and $f(n)$ have opposite signs (one is positive, and one is negative). This is because based on the intermediate value theorem, this interval is guaranteed to have a point c within it such that $f(c) = 0$.

Then, we repeat this process on the smaller interval by again dividing it in half, and so on, until the length of the interval becomes less than $2h$.

So if $f(q) \geq 0$, use q to replace a in a recursive call. If $f(q) \leq 0$, use q to replace b in a recursive call.

The code for this procedure looks like

```
let rec find_root (f: float->float) (a: float) (b: float) (h: float) : (
    float * float):
  if(b -. a < 2. *. h) then (a, b)
  else let q = (a +. b) /. 2 in
    let v = f(q) in
      if(v = 0.0) then (q, q)
      else
       if (v < 0.0) then
       (find_root f a q h)
       else
       (find_root f q b h);;
```

3 Summary

Ideas

- Debugging code often involves simplifying code or checking to make sure individual parts of the code work
- CS17 covered big ideas such as scale and divide-and-conquer

Skills

- We applied the concept of divide and conquer to the infinite domain

---

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS17document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs017/feedback.