15-122: Principles of Imperative Computation

Recitation Week 1

Rob Simmons, Josh Zimmerman

Contracts

There are 4 types of annotations in C0, each requiring a boolean expression:

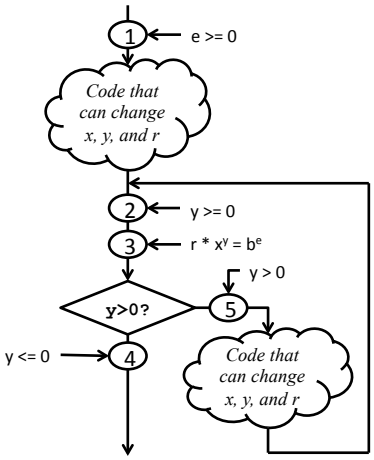
Annotation	Checked
`//@requires` *boolean_expr*;	before function execution
`//@ensures` *boolean_expr*;	before function returns
`//@loop_invariant` *boolean_expr*;	before the loop condition is checked
`//@assert` *boolean_expr*;	wherever you put it in the code

The special variable `\result` can be used only in `//@ensures` statements, and it will give you the return value of the function. (There are others that we'll get to later in the semester.)

Proving correctness of the mystery function

We use contracts to both test our code and to logically reason about code. With contracts, careful reasoning and good testing both help us to be confident that our code is correct.

Here's a different way of looking at the mystery function from lecture yesterday. Once we have loop invariants for the mystery function, we can view the whole thing as a control flow diagram:



The circle labeled **1** is a precondition of the function, and the circles labeled **2** and **3** are loop invariants. The circles labeled **4** and **5** just capture information we get from the result of the loop guard (or loop condition), but we might write **4** as an `//@assert` statement.

To prove this function correct, we need to reason about the two pieces of code (pieces that this diagram hides in the two cloud-bubbles) to ensure that our contracts never fail. When we reason about the upper code bubble, we assume that **1** is true before the code runs and show that **2** and **3** are true afterwards. When we reason about the lower code bubble, we assume **2**, **3**, and **5** are true before the code runs and show that **2** and **3** are true afterwards. To reason that the returned value r is equal to b^e, we combine the information from circles **2** and **4** to conclude that $y = 0$. Together with the information in circle **3**, this implies that $r = b^e$.

In addition, we have to reason about termination: every time the lower code bubble runs, the value y gets strictly smaller, because $y/2 < y$ for every $y > 0$, and the loop invariant ensures y never becomes negative.

Checkpoint 0

In general, what are four steps for proving the correctness of a function with one loop using loop invariants:

-
-
-
-

Preservation of loop invariants

Preserving loop invariants can be a bit confusing, because we have to assume that the loop invariant, like `y >= 0` or `r * POW(x,y) == POW(b,e)` is true before the loop invariant is checked by assuming that the loop invariant was true the last time the loop invariant was checked (we also assume that the loop guard subsequently evaluated to `true`).

```
1   while (y > 0)
2   //@loop_invariant y >= 0;
3   //@loop_invariant r * POW(x,y) == POW(b, e);
4   {
5       r = r * x;
6       y = y - 1;
7       x = x;
8   }
```

Checkpoint 1

When an arbitrary loop begins, we know _____ and

_____.

After an arbitrary iteration of the loop, we use primed values to represent the new values in terms of the old ones:

$x' =$ _____

$y' =$ _____

$r' =$ _____

We need to show that _____

This is true because _____

This terminates because _____

Because we haven't changed any loop invariants, the rest of the correctness proof for exponentiation is the same as it was in class. By keeping the loop invariant the same, we still have a proven-correct function, even though we tore out loop body and replaced it with a different (and less efficient) one!