Lecture Notes on Contracts

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 1
August 24, 2010

1 Introduction

For an overview the course goals and the mechanics and schedule of the course, please see course Overview. In these notes we review *contracts*, which we use to collectively denote function contracts, loop invariants, and other assertions about the program. Contracts will play a central role in this class, since they represent the key to connect algorithmic ideas to imperative programs. We follow the example from lecture, developing annotations to a given program that express the contracts, thereby making the program understandable (and allowing us to find the bug).

If you have not seen this example, we invite you to read this section by section to see how much of the story you can figure out on your own before moving on to the next section.

2 A Mysterious Program

You are a new employee in a company, and a colleague comes to you with the following program, written by your predecessor who was summarily fired for being a poor programmer. Your colleague claims he has tracked a bug in a larger project to this function. It is your job to find and correct this bug.

```
int f (int n) {
  int i = 0; int k = 0;
  while (k <= n) {
    k += (i<<1) + 1;
    i++;
  }
  return i-1;
}
```

Before you read on, you might examine this program for a while to try to determine what it does, or is supposed to do, and see if you spot the problem.

3 A Shift in Perspective

The code contains a few compound operators that we can expand to make it more readable. `k += e;` expands into `k = k + e;`. `i++;` expands into `i = i + 1;`.

What about `i<<1`? Remembering from your introductory programming class, or looking it up, you recall that this expression returns the result of shifting the bit representation of `i` to the left by `1` bit. This means that the contribution of each bit to the value of `i` is doubled.[1] Hence this operation corresponds to doubling the value of `i`, modulo the precision of the representation of `int`s (typically 32 or 64 bits).

With these insights we can rewrite the program into the following, which is closer to a mathematical expression of the computation.

```
int f (int n) {
  int i = 0; int k = 0;
  while (k <= n) {
    k = k + 2*i + 1;
    i = i + 1;
  }
  return i-1;
}
```

[1] see more in Lecture 3: Ints

4 Making a Conjecture

The next step is to symbolically execute the program on some input values to see its results. We notice that the *computation* of the loop is always the same, regardless of input, but that the number of iterations is controled by the input n. We compose a small table with values of i and k in the loop on each iteration. It is generally convenient in this kind of enterprise to record the values of variables *just before the loop exit condition is tested* because we can then directly see whether the test will succeed or fail.

Iteration	i	k
0	0	0
1	1	1
2	2	4
3	3	9
4	4	16

From this we conjecture several propositions:

1. i just records the number of iterations through the loop.
2. $k = i^2$

The loop exits if $k > n$ and we return $i - 1$, which was the value of i on the previous iteration. For example, for $n = 10$ we return after iteration 4 and return $4 - 1 = 3$. Nothing changes if n increases until it hits 16, in which case we iterate once more (since $k = n$) and return $5 - 1 = 4$. This leads us to the conjecture that the function is supposed to return the integer square root. For a perfect square that is the root, for other numbers it rounds the answer down.

5 Finding a Loop Invariant

Next, we would like to try to verify that, indeed, the function returns the integer square root. For this purpose we want to find the loop invariant. A *loop invariant* is a boolean expression (that is, evaluating to `true` or `false`) which is true the first time a loop is entered, and remains true every time around the loop. A loop invariant is checked just before the loop exit condition is evaluated.

From the simulated execution we conjecture that `i * i == k` is a loop invariant. We use the syntax for annotations in C0, which is derived from similar syntax in JML and Spec# which are languages supporting contracts for Java and C#, respectively.

```
int f (int n) {
  int i = 0; int k = 0;
  while (k <= n)
    //@loop_invariant i * i == k;
    {
      k = k + 2*i + 1;
      i = i + 1;
    }
  return i-1;
}
```

6 Verifying a Loop Invariant

The next job is to verify the loop invariant, or find a counterexample in case it is wrong. Since we have been told to expect a bug in the program, we have to account for the possibility that the function is indeed incorrect. The best way of finding a counterexample is often to *try* to verify the loop invariant and see where the reasoning breaks down.

To verify a loop invariant we have to prove two things:

1. The loop invariant is true when we enter the loop the first time.
2. Assuming the loop invariant is true at the start of one iteration, it will still be true at the end of that iteration.

Let us go through the steps.

1. When we enter the loop, we have $i = 0$ and $k = 0$, so $i \times i = 0 = k$. The loop invariant holds.
2. Now we assume that $i \times i = k$ and we go around the loop once (that is, we execute one full iteration of the loop). We obtain a new k, let's call it k'.

   $$k' = k + 2i + 1$$

   We also obtain a new i, let's call it i'.

   $$i' = i + 1$$

   Now we have to show that the loop invariant still holds for i' and k' (that is, $i' \times i' = k'$), assuming that it holds for i and k.

$i' \times i'$	$=$	$(i+1) \times (i+1)$	by definition if i'
	$=$	$i \times i + 2i + 1$	by distributivity of multiplication
	$=$	$k + 2i + 1$	by assumption
	$=$	k'	by definition of k'

So, indeed, the loop invariant still holds! We were correct: on every iteration, $i \times i = k$.

7 A Postcondition for the Function

What do we know when the loop exits? Every time around the loop, just before the exit test, we have $i \times i = k$. When we exit the loop we also know that the loop test fails, that is, $k > n$ (otherwise, we would continue with the loop). We can therefore say definitively that

$$i \times i > n$$

when we exit the loop. We return $i - 1$ from the function. If we add 1 to the return value on argument n its square must be bigger than n

$$(f(n) + 1) \times (f(n) + 1) > n$$

In our language of contracts, the return value is denoted by `\result`. We express a postcondition using the syntax `@ensures e;` where e is a boolean expression.

```
int f (int n)
//@ensures (\result+1) * (\result+1) > n;
{
  int i = 0; int k = 0;
  while (k <= n)
    //@loop_invariant i * i == k;
    {
      k = k + 2*i + 1;
      i = i + 1;
    }
  return i-1;
}
```

Of course, this postcondition is not what we really want of the function. For example, if the input n is a perfect square, we want $f(n) \times f(n) = n$, which is not guaranteed by the current invariant.

8 A Strengthened Postcondition

What we really want is that

$$f(n) \times f(n) \leq n \quad \text{and} \quad (f(n)+1) \times (f(n)+1) > n$$

So let's state that, formally, reversing the second inequality for ease of reading.

```
int f (int n)
//@ensures \result * \result <= n && n < (\result+1) * (\result+1);
{
  int i = 0; int k = 0;
  while (k <= n)
    //@loop_invariant i * i == k;
    {
      k = k + 2*i + 1;
      i = i + 1;
    }
  return i-1;
}
```

Now the loop invariant is too weak to prove the postcondition. What do we know after the loop? We know that $i \times i = k$ (the loop invariant) and we know $k > n$ (because we exited the loop). Since we return $i - 1$ we can only deduce `(\result+1) * (\result+1) > n` as before, but not `\result * \result <= n`.

9 A Strengthened Loop Invariant

Working backwards, what we would like to state as a second loop invariant to obtain this postcondition is that $(i-1) \times (i-1) \leq n$. However, this does not work upon entry into the loop because $-1 \times -1 = 1$ which may not be less or equal to n if $n = 0$. So we state the either $i = 0$, or $i > 0$ and the condition above holds.

```
int f (int n)
//@ensures \result * \result <= n && n < (\result+1) * (\result+1);
{
  int i = 0; int k = 0;
  while (k <= n)
    //@loop_invariant i * i == k;
    //@loop_invariant i == 0 || (i > 0 && (i-1)*(i-1) <= n);
    {
      k = k + 2*i + 1;
      i = i + 1;
    }
  return i-1;
}
```

Of course, we have to verify that the loop invariant actually holds!

1. On loop entry, we have $i = 0$.
2. Now assume we test the condition of the loop and it is true, so we have to execute loop body. What do we know at this point?
   (a) We know the first loop invariant: $i \times i = k$.
   (b) We know the second loop invariant, that is, $i = 0$ or $i > 0$ and $(i-1) \times (i-1) \leq n$.
   (c) We know that the loop condition was true (because we didn't exit the loop), so $k \leq n$.

   Now compute $k' = k + 2i + 1$ and $i' = i + 1$. We have to verify that $i' = 0$ or $i' > 0$ and $(i'-1) \times (i'-1) \leq n$. First we see that $i' > 0$ since $i > 0$. Second, we see that $i' - 1 = i$, and $i \times i = k$ and $k \leq n$, so $(i'-1) \times (i'-1) \leq n$.

The first proof shows that the loop invariant is true when we enter the loop; the second shows that is remains true if we go through the loop. So it must hold on every iteration.

Unfortunately, the new loop invariant does not the postcondition. If $i = 0$, the result may be -1 which is not the intended answer.

10 A Precondition

Tracing through we can see that the loop invariant does not imply the post-condition, because if $i = 0$ and $n < 0$ we will not ever go around the loop and return -1. In fact, in this case no square root exists, so we couldn't possibly return a correct answer.

This suggests that the function is missing a precondition, namely $n \geq 0$. Preconditions are stated with an `@requires` clause.

```
int f (int n)
//@requires n >= 0;
//@ensures \result * \result <= n && n < (\result+1) * (\result+1);
{
  int i = 0; int k = 0;
  while (k <= n)
    //@loop_invariant i * i == k;
    //@loop_invariant i == 0 || (i > 0 && (i-1)*(i-1) <= n);
    {
      k = k + 2*i + 1;
      i = i + 1;
    }
  return i-1;
}
```

At this point we know that we will go around the loop at least once, so that the second part of the loop invariant will be true upon loop exit. Moreover, because $i \times i = k$ and $k > n$ when we exit the loop, it follows that $i \neq 0$ and the second disjunct of the second part of the loop invariant must be true upon exit, that is, $(i - 1) \times (i - 1) \leq n$.

All these facts together are now sufficient to establish the postcondition, if the function is called with an argument satisfying the precondition. This is what we mean by *partial correctness* of the function. *Total correctness* requires in addition that the function always terminates.

11 Termination

At this point we have established that *if* the function returns when called with a positive argument, *then* it will satisfy the postcondition which says that it implements the integer square root function.

Armed with this information we go back to our colleague to discuss our findings so far.

If there really is a bug, this leaves open a couple of possibilities. It could be, that the function is called with a negative argument. In order to test this hypothesis, we make the precondition explicit with an `assert` statement in the program, run it, and see if this condition arises. It turns out, in our scenario it does not; the function f is called with only with positive arguments.

Another possibility is that the function was intended to implement something else. For example, it may have been intended to round up instead of rounding down. But apparently the function works most of the time, so that seems unlikely.

What remains is the possibility that the function never terminates, and indeed, our colleague states that "*the programs goes off into Never Never Land*".

How is it possible for this function to run on forever?

12 Fixed Precision Arithmetic

To solve this final puzzle and find the bug we must know that in many programming languages integers have a fixed size, typically 32 or 64 bits. This means that an overflow condition can occur. A language may or may not detect overflow (most don't), and in fact the computation may "wrap around" and return a negative number. In the next lecture we will see exactly when and why this happens.

In our particular example, the square root of a positive number is less or equal to the number itself, so the answer itself cannot overflow. But we step i one past the correct result r (which is why we subtract 1 at the end), and we compute $(r+1) \times (r+1)$ when r is the answer to see if it exceeds n. But this number could be larger than the maximally representable integer and become negative. In that case we have found our square root, but we fail to detect this because k has become negative, so the loop condition $k \leq n$ remains true.

Summary: if k ever becomes negative due to overflow, we have exceeded the largest possible integer on $i \times i$. Since n was positive, the last value of i is indeed the integer square root and we should return it. We have to fix the postcondition and the loop exit test to reflect this insight.

```
int f (int n)
//@requires n >= 0;
//@ensures \result * \result <= n
//@ensures n < (\result+1) * (\result+1) || (\result+1)*(\result+1) < 0;
{
  int i = 0; int k = 0;
  while (0 <= k & k <= n)
    //@loop_invariant i * i == k;
    //@loop_invariant i == 0 || (i > 0 && (i-1)*(i-1) <= n);
    {
      k = k + 2*i + 1;
      i = i + 1;
    }
  return i-1;
}
```

This function terminates because k strictly increases each time around the loop and therefore must eventually either exceed n or become negative due to overflow.