Lecture Notes on Contracts

15-122: Principles of Imperative Computation
Frank Pfenning*

Lecture 2
January 13, 2011

1 Introduction

For an overview of 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.

*Edits by André Platzer

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 x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) r = x * r;
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

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 can spot the problem.

3 Forming a Conjecture

The first step it to execute the program on some input values to see its results. The code is in a file called mystery2.c0 so we invoke the `coin` interpreter to let us experiment with code.

```
% coin mystery2.c0
Coin 0.2.9 "Penny" (r10, Fri Jan 6 22:08:54 EST 2012)
Type ‘#help’ for help or ‘#quit’ to exit.
-->
```

At this point we can type in statements and they will be executed. One form of statement is an expression, in which case `coin` will show its value. For example:

```
--> 3+8;
11 (int)
-->
```

We can also use the function in the files that we loaded when we started `coin`. In this case, the mystery function is called `f`, so we can evaluate it on some arguments.

```
--> f(2,3);
8 (int)
--> f(2,4);
16 (int)
--> f(1,7);
1 (int)
--> f(3,2);
9 (int)
-->
```

Can you form a conjecture from these values?

From these and similar examples, you might form the conjecture is that $f(x, y) = x^y$, this is, x to the power y. One can confirm that with a few more values, such as

```
--> f(-2,3);
-8 (int)
--> f(2,8);
256 (int)
--> f(2,10);
1024 (int)
-->
```

It seems to work out! Our next task is to see why this function actually computes the power function. Understanding this is necessary so we can try to find the error and correct it.

4 Finding a Loop Invariant

Now we start to look inside the function and see how it computes.

```
int f (int x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) r = x * r;
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

We notice the conditional

```
if (y % 2 == 1) r = x * r;
```

The condition tests if y modulo 2 is 1. For nonnegative y, this is true if y is odd. We observe that in the loop body, y must indeed be positive (y will not turn negative by $y = y/2$) so this is a correct test for whether y is odd.

Each time around the loop we divide y by 2, using integer division (which rounds towards 0). It is exact division if y is even. If y starts as a power of 2, it will remain even throughout the iteration. In this case r will remain 1 throughout the execution of the function. Let's tabulate how the loop works for $x = 2$ and $y = 8$. But at which point in the program do we tabulate the values? It turns out generally the best place for a loop is *just before the exit condition is tested*. By *iteration 0* we mean when we enter the loop the first time and test the condition, *iteration 1* is after the loop body has been traversed once and we are looking again at the exit condition, etc.

iteration	x	y	r
0	2	8	1
1	4	4	1
2	16	2	1
3	256	1	1

After 3 iterations, $x = 256$ and $y = 1$, so the loop condition $y > 1$ becomes false and we exit the loop. We return $r * x = 256$.

To understand *why* this loop works we need to find a so-called *loop invariant*: a quantity that does not change throughout the loop. In this example, when y is a power of 2 then r is a loop invariant. Can you see a loop invariant involving just x and y?

Going back to our earlier conjecture, we are trying to show that this function computes x^y. Interestingly, after every iteration of the loop, this quantity is exactly the same! Before the first iteration it is $2^8 = 256$. After the first iteration it is $4^4 = 256$. After the second iteration it is $16^2 = 256$. After the third iteration is it is $256^1 = 256$. Let's note it down in the table.

iteration	x	y	r	x^y
0	2	8	1	256
1	4	4	1	256
2	16	2	1	256
3	256	1	1	256

Still concentrating on this special case where y is a power of 2, let's see if we can use the invariant to show that the function is correct.

5 Proving the Loop Invariant

To show that the quantity x^y is a loop invariant, we have to prove that if we execute the loop body once, x^y before will be equal to x^y after. We cannot write this as $x^y = x^y$, because that is of course always true, speaking mathematically. Mathematics does not understand the idea of assigning a new value to a variable. The general convention we follow is to add a prime ($'$) to the name of a variable to denote its value after an iteration.

So assume we have x and y, and y is a power of 2. After one iteration we have $x' = x * x$ and $y' = y/2$. To show that x^y is loop invariant, we have to show that $x^y = x'^{y'}$. So let's calculate:

$$\begin{array}{rcll} x'^{y'} & = & (x * x)^{y/2} & \text{By definition of } x' \text{ and } y' \\ & = & (x^2)^{y/2} & \text{Since } a * a = a^2 \\ & = & x^{2*(y/2)} & \text{Since } (a^b)^c = a^{b*c} \\ & = & x^y & \text{Since } 2 * (a/2) = a \text{ when } a \text{ is even} \end{array}$$

Moreover, if y is a power of 2, then $y' = y/2$ is also a power of 2 (subtracting 1 from the exponent).

We have confirmed that x^y is loop invariant if y is a power of 2. Does this help us to ascertain that the function is *correct* when y is a power of two?

6 Loop Invariant Implies Postcondition

The postcondition of a function is usually a statement about the result it returns. Here, the postcondition is that $f(x, y) = x^y$. Let's recall the function:

```
int f (int x, int y) {
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) r = x * r;
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

If y is a power of 2, then the quantity x^y never changes in the loop (as we have just shown). Also, in that case r never changes, remaining equal to 1. When we exit the loop, $y = 1$ because y starts out as some (positive) power of 2 and is divided by 2 every time around loop. So then

$$r * x = 1 * x = x = x^1 = x^y$$

so we return the correct result, x^y!

By using two loop invariant expressions (r and x^y) we were able to show that the function returns the correct answer if it does return answer. Does the loop always terminate?

7 Termination

In this case it is easy to see that the loop always terminates, because if we start with $y = 2^n$ we go around the loop exactly n times before $y = 2^{n-n} = 1$ and we exit the loop. We used here that $(2^k)/2 = 2^{k-1}$ for $k \geq 1$.

Our next challenge then will be to extend this result to arbitrary y. Before we do this, now that we have some positive results, let's try to see if we find some counterexample since the function is supposed to have a bug somewhere!

Please try to find a counterexample to the conjecture that $f(x, y) = x^y$ before you move on, taking the above information into account.

8 A Counterexample

We don't have to look at powers of 2 — we already know the function works correctly there. Some of the earlier examples were not powers of two, and the function still worked:

```
--> f(2,3);
8 (int)
--> f(-2,3);
-8 (int)
--> f(2,1);
2 (int)
-->
```

What about 0, or negative exponents?

```
--> f(2,0);
2 (int)
--> f(2,-1);
2 (int)
-->
```

Looks like we have found at least two problems. $2^0 = 1$, so the answer 2 is definitely incorrect. $2^{-1} = 1/2$ so one might argue it should return 0. Or one might argue in the absence of fractions (we are working with integers), a negative exponent does not make sense.

9 Preconditions for Functions

Let's go back to a *mathematical* definition of the power function x^y on integers x and y. We define:

$$\begin{aligned} x^0 &= 1 \\ x^{y+1} &= x * x^y \quad \text{for } y \geq 0 \end{aligned}$$

In this form it remains undefined for negative exponents. In programming, this is captured as a *precondition*: we require that the second argument to f not be negative. Preconditions are written as `//@requires` and come before the body of the function.

```
int f (int x, int y)
//@requires y >= 0;
{
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) r = x * r;
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

This is the first part of what we call the *function contract*. It expresses what the function requires of any client that calls it, namely that the second argument is positive. It is an error to call it with a negative argument; no promises are made about what the function might return otherwise. It might even abort the computation due to a contract violation.

But a contract usually has two sides. What does f promise? We know it promises to compute the exponential function, so this should be formally expressed.

10 Function Contracts

The C0 language does not have a built-in power function. So we need to write it explicitly ourselves. But wait! Isn't that what f is supposed to do? The idea in this an many other examples to capture a specification in the simplest possible form, even if it may not be computationally efficient, and then promise in the postcondition to satisfy this simple specification. Here, we can transcribe the mathematical definition into a recursive function.

```
int pow (int x, int y)
//@requires y >= 0;
{ if (y == 0)
    return 1;
  else
    return x * pow(x, y-1);
}
```

In the rest of the lecture we often silently go back and forth between x^y and $pow(x, y)$. Now we incorporate `pow` into a formal postcondition for the function. Postconditions have the form `//@ensures e;`, where e is a boolean expression. The are also written before the function body, by convention after the preconditions. Postconditions can use a special variable `\result` to refer to the value returned by the function.

```
int f (int x, int y)
//@requires y >= 0;
//@ensures \result == pow(x,y);
{
  int r = 1;
  while (y > 1) {
    if (y % 2 == 1) r = x * r;
    x = x * x;
    y = y / 2;
  }
  return r * x;
}
```

Note that as far as the function f is concerned, if we are considering calling it we do not need to look at its body at all. Just looking at the pre- and post-conditions (the `@requires` and `@ensures` clauses), tells us everything we need to know as long as the f adheres to its contract and we do to ours (that is, pass only positive y).

11 Dynamically Checking Contracts

During the program development phase, we can instruct the C0 compiler or interpreter to check adherence to contracts. This is done with the `-d` flag on the command line, which stands for *dynamic checking*. Let's see how the implementation now reacts to correct and incorrect inputs, assuming we have added `pow` as well as pre- and postconditions as shown above.

```
% coin solution2a.c0 -d
Coin 0.2.9 "Penny" (r10, Fri Jan 6 22:08:54 EST 2012)
Type ‘#help’ for help or ‘#quit’ to exit.
solution2a.c0:22.5-22.6:error:
  cannot assign to variable ‘x’ used in //@ensures annotation
Unable to load files, exiting...
%
```

The error is that we are changing the value of x in the body of the loop, while the postcondition refers to x. If it were allowed, it would violate the principle that we need to look at the contract only when calling the function, because assignments to x change the meaning of the postcondition. We want `\result == pow(x,y)` for the *original* x and y we passed as arguments to f and not the values x and y might hold at the end of the function.

We therefore change the function body, creating auxiliary variables b (for base) and e (for exponent) to replace x and y which we leave unchanged.

```
int f (int x, int y)
//@requires y >= 0;
//@ensures \result == pow(x,y);
{
  int r = 1;
  int b = x; /* base */
  int e = y; /* exponent */
  while (e > 1) {
    if (e % 2 == 1) r = b * r;
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

Now invoking the interpreter with `-d` works correctly when we return the right answer, but raises an exception if we give it arguments where we know the function to be incorrect, or arguments that violate the precondition to the function.

```
% coin solution2b.c0 -d
Coin 0.2.9 "Penny" (r10, Fri Jan 6 22:08:54 EST 2012)
Type ‘#help’ for help or ‘#quit’ to exit.
--> f(3,2);
9 (int)
--> f(3,-1);
solution2b.c0:16.4-16.19: @requires annotation failed
Last position: solution2b.c0:15.1-27.15
               f from <stdio>:1.1-1.7
--> f(2,0);
solution2b.c0:17.4-17.30: @ensures annotation failed
               f from <stdio>:1.1-1.6
-->
```

The fact that `@requires` annotation fails in the second example call means that our call is to blame, not f. The fact that the `@ensures` annotation fails in the third example call means the function f does not satisfy its contract and is therefore to blame.

12 Generalizing the Loop Invariant

Before fixing the bug with an exponent of 0, let's figure out why the function apparently works when the exponent is odd. Our loop invariant so far only works when y is a power of 2. It uses the basic law that $b^{2*c} = (b^2)^c = (b * b)^c$ in the case where the exponent $e = 2 * c$ is even.

What about the case where the exponent is odd? Then we are trying to compute b^{2*c+1}. With analogous reasoning to above we obtain $b^{2*c+1} = b * b^{2*c} = b * (b * b)^c$. This means there is an additional factor of b in the answer. We see that we exactly multiply r by b in the case that e is odd!

```
int f (int x, int y)
//@requires y >= 0;
//@ensures \result == pow(x,y);
{
  int r = 1;
  int b = x; /* base */
  int e = y; /* exponent */
  while (e > 1) {
    if (e % 2 == 1) r = b * r;
    b = b * b;
    e = e / 2;
  }
  return r * b;
}
```

What quantity remains invariant now, throughout the loop? Try to form a conjecture for a more general loop invariant before reading on.

Let's make a table again, this time to trace a call when the exponent is not a power of two, say, while computing 2^7 by calling $f(2, 7)$.

iteration	b	e	r	b^e
0	2	7	1	128
1	4	3	2	64
2	16	1	8	16

As we can see, b^e is not invariant, but $r * b^e = 128$ is! The extra factor from the equation on the previous page is absorbed into r.

We now express this proposed invariant formally in C0. This requires the `@loop_invariant` annotation. It must come immediately before the loop body, but it is checked just before the loop exit condition. We would like to say that the integer expression `r * pow(b,e)` is invariant, but this is not possible directly.

Loop invariants in C0 are *boolean* expressions which must be either true or false. We can achieve this by stating that `r * pow(b,e) == pow(x,y)`. Observe that x and y do not change in the loop, so this guarantees that `r * pow(b,e)` never changes either. But it says a little more, stating what the invariant quantity is in terms of the original function parameters.

```
int f (int x, int y)
//@requires y >= 0;
//@ensures \result == pow(x,y);
{
  int r = 1;
  int b = x; /* base */
  int e = y; /* exponent */
  while (e > 1)
    //@loop_invariant r * pow(b,e) == pow(x,y);
    {
      if (e % 2 == 1) r = b * r;
      b = b * b;
      e = e / 2;
    }
  return r * b;
}
```

13 Fixing the Function

The bug we have discovered so far was for $y = 0$. In that case, $e = 0$ so we never go through the loop and return $r * b$ instead of 1. Why is that case different than our reasoning in the cases $y > 0$? If we exit the loop with a value $e = 1$, then the loop invariant implies the function postcondition. To see this, note that we return $r * b$ and $r * b = r * b^1 = r * b^e = x^y$, where the last equation is the loop invariant. When y (and therefore e) is 0, however, this reasoning does not apply because we also exit the loop because $e \leq 1$, yet that is because $e = 0$ and not 1. And the returned value $r * b$, then, is generally different from $x^y = r * b^e = r * b^0 = r$.

Think about how you might fix the function and its annotations before reading on.

We can fix it by carrying on with the while loop until $e = 0$. On the last iteration e is 1, which is odd, so we set $r' = b * r$. This means we now should return r' (the new r) after the one additional iteration of the loop, and not $r * b$.

```
int f (int x, int y)
//@requires y >= 0;
//@ensures \result == pow(x,y);
{
  int r = 1;
  int b = x; /* base */
  int e = y; /* exponent */
  while (e > 0)
    //@loop_invariant r * pow(b,e) == pow(x,y);
    {
      if (e % 2 == 1) r = b * r;
      b = b * b;
      e = e / 2;
    }
  return r;
}
```

Now when the exponent $y = 0$ we skip the loop body and return $r = 1$, which is the right answer for x^0! Indeed:

```
% coin solution2d.c0 -d
Coin 0.2.9 "Penny" (r10, Fri Jan 6 22:08:54 EST 2012)
Type ‘#help’ for help or ‘#quit’ to exit.
--> f(2,0);
1 (int)
-->
```

14 Strengthening the Loop Invariant Again

We would now like to show that the improved function is correct. That requires two steps: one is that the loop invariant implies the postcondition; another is that the proposed loop invariant is indeed a loop invariant. The loop invariant, $r * b^e = x^y$ implies that the result $r = x^y$ if we know that $e = 0$ (since $b^0 = 1$).

But how do we know that $e = 0$ when we exit the loop? Actually, we don't: the loop invariant is too weak to prove that. The negation of the exit condition only tells us that $e \leq 0$. However, if we add another loop invariant, namely that $e \geq 0$, then we know $e = 0$ when the loop is exited and the postcondition follows. For clarity, we also add a (redundant) assertion to this effect after the loop and before the return statement.

```
int f (int x, int y)
//@requires y >= 0;
//@ensures \result == pow(x,y);
{
  int r = 1;
  int b = x; /* base */
  int e = y; /* exponent */
  while (e > 0)
    //@loop_invariant e >= 0;
    //@loop_invariant r * pow(b,e) == pow(x,y);
    {
      if (e % 2 == 1) r = b * r;
      b = b * b;
      e = e / 2;
    }
  //@assert e == 0;
  return r;
}
```

The `@assert` annotation can be used to verify an expression that should be true. If it is not, our reasoning must have been faulty somewhere else. `@assert` is a useful debugging tool and sometimes helps the reader understand better what the code author intended.

15 Verifying the Loop Invariants

It seems like we have beaten this example to death: we have added pre- and post-conditions, stated loop invariants, fixed the original bug and shown that the loop invariants imply the postcondition. But we have not yet verified that the loop invariant actually holds! Ouch! Let's do it.

We begin with the invariant $e \geq 0$. We have to demonstrate two properties.

Init: The loop invariant holds initially, otherwise we cannot even rely on it when the loop starts. When we enter the loop, $e = y$ by the assignment before the loop and we know that $y \geq 0$ holds by the precondition of the function. Done.

Preservation: Assume the invariant holds just before the exit condition is checked. We have to show that it is true again when we reach the exit condition after one iteration of the loop. That is, we consider one iteration of the loop. We have already shown that the loop invariant was established successfully initially. We can assume the loop invariant holds before the loop exit condition is checked in the iteration that we consider, because the loop invariant has already been established in the previous iteration of the loop (otherwise the previous iteration would already have been incorrect and, e.g., dynamic checking of the contracts would have reported a failure). But we need to show that the loop invariant holds again at the end of the loop, otherwise the argument for the next loop iteration could not assume the loop invariant to hold. Overall we need to do the following.

Assumption: $e \geq 0$.

To show: $e' \geq 0$ where $e' = e/2$, with integer division. This clearly holds, because division by 2 does not give a negative result e' from positive input e.

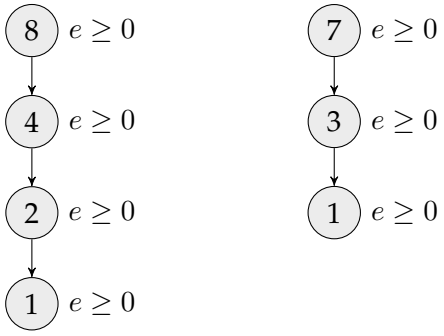
Before we look at the next loop invariant $r * pow(b, e) = pow(x, y)$ to consider, let us first make sure we understand why the above proof did establish that $e \geq 0$ really is a loop invariant. Think back what our value tables

looked like, e.g.:

iteration	b	e	r	b^e
0	2	8	1	256
1	4	4	1	256
2	16	2	1	256
3	256	1	1	256

iteration	b	e	r	b^e
0	2	7	1	128
1	4	3	2	64
2	16	1	8	16

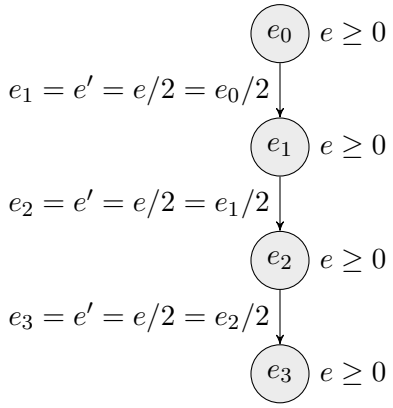
For those particular input numbers, we can see that $e \geq 0$. Graphically, we can visualize the values of e changing in those two respective tables from top to bottom as follows:



But we can hardly write down all tables for all possible concrete input numbers and check each of them by hand. Instead, we want to have one argument capturing all input values x, y with $y \geq 0$ (according to the precondition) at once. We do not know what particular values the variable e will have in each iteration anymore. They may all be different values. We denote the value that variable e has right before the loop exit condition in iteration i by e_i. So we just say that e_0 is the initial value, that variable e has right before the loop. And e_1 is the value that e has at the exit condition after the first iteration, e_2 is the value that e has after the second iteration and so on. We could proceed similarly with b and say that b_i is the value that variable b has before the loop exit condition in iteration number i. This gives us a table with placeholders $e_0, e_1, e_2, \ldots$ instead of concrete numbers

iteration	b	e	r	b^e
0	b_0	e_0	r_0	$b_0^{e_0}$
1	b_1	e_1	r_1	$b_0^{e_0}$
2	b_2	e_2	r_2	$b_0^{e_0}$
3	b_3	e_3	r_3	$b_0^{e_0}$
$\ldots$				

In our visual representation of what happens, this would look as follows:



The loop body shows how the values e_i and e_{i+1} of e are related. The loop body does $e' = e/2$, where e' is the value of e after the loop body executes, which implies that $e_{i+1} = e_i/2$, because this is the operation that is performed on the data. Initially, that is for e_0, we need to verify that the loop invariant $e \geq 0$ holds, which is exactly what the step "Init" in our proof above shows. The first iteration of the loop can, thus, rest assured that the loop invariant $e \geq 0$ holds before the loop exit condition, because that corresponds to $e_0 \geq 0$. But since we want the second iteration to also be able to assume the loop invariant to hold before the loop exit, we need to check that the loop invariant actually holds after the first loop iteration completes. We do not know what specific number e_1 is, only that it satisfies the loop invariant $e_1 \geq 0$. So we need to prove $e_2 \geq 0$ when assuming that $e_1 \geq 0$ and using our knowledge what the loop changes, i.e., $e_{i+1} = e_i/2$. This is what the "Preservation" step in the above proof shows. So we know that $e_2 \geq 0$, i.e., the loop invariant holds before the loop exit condition for the third iteration but need to convince ourselves that the loop invariant holds after the third iteration. We still do not know what specific number e_2, but only that it satisfies $e_2 \geq 0$. So we assume all we know ($e_2 \geq 0$) and verify that the invariant holds after the third iteration, i.e., $e_3 \geq 0$. This is what the "Preservation" step above also showed already. In other words, we notice that the same argument actually applies no matter which iteration i of the loop we look at. So the "Preservation" step can safely drop the subscripts. We can simply assume the loop invariant $e \geq 0$ holds before the loop (i.e., before the loop exit condition) and show that $e' \geq 0$ holds after the loop (i.e., right before the loop exit condition is checked again). This is why loop invariant verification can use the actual program variables, which represent symbolic values, instead of having to worry about subscripting variables.

Now that we have understood the proof method, let us return to the verification and look at the loop invariant $r * pow(b, e) = pow(x, y)$.

Init: The invariant holds initially, because when entering the loop we have $r = 1$, $b = x$ and $e = y$.

Preservation: We show that the invariant is perserved on every iteration. For this, we distinguish two cases: e is even and e is odd.

Assumption: $r * pow(b, e) = pow(x, y)$.

To show: $r' * pow(b', e') = pow(x, y)$, where r', b', and e' are the values of r, b, and e after one iteration.

Case: e is even. Then $r' = r$, $b' = b * b$ and $e' = e/2$ and we reason:

$$\begin{array}{rcll} r' * pow(b', e') & = & r * pow(b * b, e/2) & \\ & = & r * pow(b, 2 * (e/2)) & \text{Since } (a^2)^c = a^{2*c} \\ & = & r * pow(b, e) & \text{Since } e \text{ is even} \\ & = & pow(x, y) & \text{By assumption} \end{array}$$

Case: e is odd. Then $r' = b * r$, $b' = b * b$ and $e' = (e - 1)/2$ (because e is odd, integer division rounds towards 0, and $e \geq 0$) and we reason:

$$\begin{array}{rcll} r' * pow(b', e') & = & (b * r) * pow(b * b, (e - 1)/2) & \\ & = & (b * r) * pow(b, 2 * (e - 1)/2) & \text{Since } (a^2)^c = a^{2*c} \\ & = & (b * r) * pow(b, e - 1) & \text{Since } e - 1 \text{ is even} \\ & = & r * pow(b, e) & \text{Since } a * (a^c) = a^{c+1} \\ & = & pow(x, y) & \text{By assumption} \end{array}$$

This shows that both loop invariants hold on every iteration.

16 Termination

The previous argument for termination still holds. By loop invariant, we know that $e \geq 0$. When we enter the body of the loop, the condition must be true so $e > 0$. Now we just use that $e/2 < e$ for $e > 0$, so the value of e is strictly decreasing and positive, which means it must eventually become 0, upon which we exit the loop and return from the function after one additional step. The reason is that we cannot do integer division by two infinitely often from a nonnegative number without reaching 0 eventually.

17 A Surprise

Now, let's try our function on some larger numbers, computing some powers of 2.

```
% coin -d solution2e.c0
Coin 0.2.9 "Penny" (r10, Fri Jan 6 22:08:54 EST 2012)
Type ‘#help’ for help or ‘#quit’ to exit.
--> f(2,30);
1073741824 (int)
--> f(2,31);
-2147483648 (int)
--> f(2,32);
0 (int)
-->
```

2^{30} looks plausible, but how could 2^{31} be negative or 2^{32} be zero? We claimed we just proved it correct!

The reason is that the values of type `int` in C0 or C and many other languages actually do not represent arbitrarily large integers, but have a fixed-size representation. In mathematical terms, this means we that we are dealing with *modular arithmetic*. The fact that $2^{32} = 0$ provides a clue that integers in C0 have 32 bits, and arithmetic operations implement arithmetic modulo 2^{32}.

In this light, the results above are actually correct. We examine modular arithmetic in detail in the next lecture.

Exercises

Exercise 1 *After Lecture 3 on modular arithmetic go back to the correctness proof in this lecture and determine if all of the reasoning is valid. Explain which steps are questionable and why they are correct or not. Is the mystery function correct if all operations (including the* `pow` *function) are interpreted in modular arithmetic and two's complement representation of fixed precision integers?*

Exercise 2 *Rewrite first* `pow` *and then* `f` *so that it signals an error in case of an overflow rather than silently working in modular arithmetic.*