Lecture 1
Contracts

15-122: Principles of Imperative Computation (Fall 2021)
Frank Pfenning, Iliano Cervesato

In these notes we review *contracts*, which we use to collectively denote function contracts, loop invariants, and other assertions about a program. Contracts will play a central role in this class, since they represent the key to connect algorithmic ideas to imperative programs. We do this through an example, developing annotations for a given program that express the contracts, thereby making the program understandable (and allowing us to find a bug).

Additional Resources

- Review slides (`https://cs.cmu.edu/~15122/slides/review/01-contracts.pdf`)
- OLI modules (`https://cs.cmu.edu/~15122/misc/oli-01.shtml`)
- Code for this lecture (`https://cs.cmu.edu/~15122/code/01-contracts.tgz`)

In term of our learning goals, this lecture addresses:

Computational Thinking: Developing contracts (preconditions, postconditions, assertions, and loop invariants) that establish the safety and correctness of imperative programs.

Using point-to reasoning to develop proofs of the safety and correctness of code with contracts.

Developing informal termination arguments for programs with loops and recursion.

Identifying the difference between *specification* and *implementation*.

Algorithms and Data Structures: Integer algorithms (fast power).

Programming: Identifying, describing, and effectively using **`while`**-loops and contracts (in C0).

We invite you to read this chapter section by section to see how much of the story you can figure out on your own before moving on to the next section.

1 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.

```
1  int f(int x, int y) {
2    int r = 1;
3    while (y > 1) {
4      if (y % 2 == 1) {
5        r = x * r;
6      }
7      x = x * x;
8      y = y / 2;
9    }
10   return r * x;
11 }
```

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.

2 Forming a Conjecture

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

```
% coin mystery.c0
C0 interpreter (coin) 0.3.3 'Nickel'
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 functions 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 that $f(x, y) = x^y$, that 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.

3 Finding a Loop Invariant

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

```
1  int f(int x, int y) {
2    int r = 1;
3    while (y > 1) {
4      if (y % 2 == 1) {
5        r = x * r;
6      }
7      x = x * x;
8      y = y / 2;
9    }
10   return r * x;
11 }
```

We notice the conditional

```
4      if (y % 2 == 1) {
5        r = x * r;
6      }
```

The condition tests if y modulo 2 is 1. For positive y, this is true if y is odd. We also observe that in the loop body, y must indeed be positive 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 find 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, 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.

4 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 executing the body will be equal to x^y after executing the body. 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 a loop invariant, we have to show that $x^y = x'^{y'}$. So let's calculate:

$$
\begin{array}{rcll}
x'^{y'} & = & (x * x)^{y/2} & \text{by lines 7 and 8 (definition of } x' \text{ and } y') \\
& = & (x^2)^{y/2} & \text{by math (since } a * a = a^2) \\
& = & x^{2*(y/2)} & \text{by math (since } (a^b)^c = a^{b*c}) \\
& = & x^y & \text{by math (since } 2 * (a/2) = a \text{ when } a \text{ is even})
\end{array}
$$

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 a 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?

5 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:

```
1  int f(int x, int y) {
2    int r = 1;
3    while (y > 1) {
4      if (y % 2 == 1) {
5        r = x * r;
6      }
7      x = x * x;
8      y = y / 2;
9    }
10   return r * x;
11 }
```

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 an answer. Does the loop always terminate?

6 Termination

In this case it is easy to see that the loop always terminates. To show that a loop always terminates, we need to define some expression whose value *always gets strictly smaller* during any arbitrary iteration of the loop, that can never become smaller than 0, and that causes the loop guard to be false when equal to zero. This means that the loop can only run a finite number of times.

The expression $y/2$ is always less than y when $y > 0$, so on any arbitrary iteration of the loop, y gets strictly smaller, and it can never become negative. Moreover, the loop guard is `false` when `y==0`. Therefore, we know the loop has to terminate.

> *By the same token, we could identify any lower bound, not just zero, that causes the loop guard to be false and an expression whose value strictly decreases and never passes that lower bound,* or *we could identify an upper bound that causes the loop guard to be false and an expression whose value strictly increases but never passes that upper bound!*

7 A Counterexample

We don't have to look at y being a power 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)
-->
```

It 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. In any case, $f(2, -1)$ should certainly not return 2.

8 Imposing a Precondition

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

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

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 be non-negative. It is an error to call it with a negative argument; no promises are made about what the function might return in such case. It might even abort the computation due to a contract violation. When calling a function, its preconditions must be satisfied for the arguments it is called with. If this is the case, the call is *safe*. We always need to have a reason to believe that function calls are safe.

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.

9 Promising a Postcondition

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 and many other examples is 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. They 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.

```
1  int f(int x, int y)
2  //@requires y >= 0;
3  //@ensures \result == POW(x,y);
4  {
5    int r = 1;
6    while (y > 1) {
7      if (y % 2 == 1) {
8        r = x * r;
9      }
10     x = x * x;
11     y = y / 2;
12   }
13   return r * x;
14 }
```

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 we adhere to our contract and pass f a non-negative y, then f will adhere to its contract and return x^y.

The postconditions of a function are facts that ought to be true when the function returns. Like here, we often use postconditions to describe what the function is expected to do. A function is *correct* if its postconditions are always satisfied when called with inputs that satisfy its preconditions.

10 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 mystery2b.c0 -d
mystery2b.c0:20.5-20.6:error:cannot assign to variable 'x'
  used in @ensures annotation
    x = x * x;
    ~
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 only at the contract 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.

```
1  int f(int x, int y)
2  //@requires y >= 0;
3  //@ensures \result == POW(x,y);
4  {
5    int r = 1;
6    int b = x;           /* base     */
7    int e = y;           /* exponent */
8    while (e > 1) {
9      if (e % 2 == 1) {
10       r = b * r;
11     }
12     b = b * b;
13     e = e / 2;
14   }
15   return r * b;
```

```
16 }
```

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 mystery2c.c0 -d
C0 interpreter (coin) 0.3.3 'Nickel'
Type '#help' for help or '#quit' to exit.
--> f(3,2);
9 (int)
--> f(3,-1);
mystery2c.c0:11.4-11.20: @requires annotation failed

Last position: mystery2c.c0:11.4-11.20
               f from <stdio>:1.1-1.8
--> f(2,0);
mystery2c.c0:12.4-12.32: @ensures annotation failed
Last position: mystery2c.c0:12.4-12.32
               f from <stdio>:1.1-1.7
-->
```

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.

11 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!

```
1  int f(int x, int y)
2  //@requires y >= 0;
3  //@ensures \result == POW(x,y);
4  {
5    int r = 1;
6    int b = x;           /* base     */
7    int e = y;           /* exponent */
8    while (e > 1) {
9      if (e % 2 == 1) {
10       r = b * r;
11     }
12     b = b * b;
13     e = e / 2;
14   }
15   return r * b;
16 }
```

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

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 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.

```
1  int f(int x, int y)
2  //@requires y >= 0;
3  //@ensures \result == POW(x,y);
4  {
5    int r = 1;
6    int b = x;           /* base     */
7    int e = y;           /* exponent */
8    while (e > 1)
9    //@loop_invariant r * POW(b,e) == POW(x,y);
10   {
11     if (e % 2 == 1) {
12       r = b * r;
13     }
14     b = b * b;
15     e = e / 2;
16   }
17   return r * b;
18 }
```

12 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. If we exit the loop and $e = 1$, 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 exit the loop and $e = 0$, not 1: $x^0 = 1$ but $r * b = x$ since $r = 1$ and $b = x$.

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 $r * b$.

```
1  int f(int x, int y)
2  //@requires y >= 0;
3  //@ensures \result == POW(x,y);
4  {
5    int r = 1;
6    int b = x;           /* base     */
7    int e = y;           /* exponent */
8    while (e > 0)
9      //@loop_invariant r * POW(b,e) == POW(x,y);
10     {
11       if (e % 2 == 1) {
12         r = b * r;
13       }
14       b = b * b;
15       e = e / 2;
16     }
17   return r;
18 }
```

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 mystery2e.c0 -d
C0 interpreter (coin) 0.3.3 'Nickel'
Type '#help' for help or '#quit' to exit.
--> f(2,0);
1 (int)
-->
```

13 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.

```
1  int f(int x, int y)
2  //@requires y >= 0;
3  //@ensures \result == POW(x,y);
4  {
5    int r = 1;
6    int b = x;           /* base     */
7    int e = y;           /* exponent */
8    while (e > 0)
9      //@loop_invariant e >= 0;
10     //@loop_invariant r * POW(b,e) == POW(x,y);
11     {
12       if (e % 2 == 1) {
13         r = b * r;
14       }
15       b = b * b;
16       e = e / 2;
17     }
18   //@assert e == 0;
19   return r;
20 }
```

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.

14 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 `//@loop_invariant e >= 0;`, which we write in the mathematical form $e \geq 0$ for convenience. We have to demonstrate two properties.

Init: The invariant holds initially. When we enter the loop, $e = y$ and $y \geq 0$ by the precondition of the function. Done.

More formally,

$e = y$	by line 7
$y \geq 0$	by line 2 (precondition)
$e \geq 0$	by math (logic on the previous two lines)

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.

Assumption: $e \geq 0$.

To show: $e' \geq 0$ where $e' = e/2$, with integer division.

Here's the simple proof:

$e \geq 0$	by assumption
$e' = e/2$	by line 16
$e' \geq 0$	by math (definition of integer division)

Next, we look at the invariant `//@loop_invariant r * POW(b,e) == POW(x,y);`, again written in its mathematical form as $r * b^e = x^y$ for clarity.

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

The mathematical proof is as follows:

$r = 1$	by line 5
$b = x$	by line 6
$e = y$	by line 7
$r * b^e = x^y$	by math on the above three lines

Preservation: We show that the invariant is preserved on every iteration.

Assumption: $r * b^e = x^y$.

To show: $r' * b'^{e'} = x^y$, where r', b', and e' are the values of r, b, and e after one iteration.

To prove this, we distinguish two cases: e is even and e is odd.

Case e is even, so that $e = 2 * n$ for some positive n. Then, the proof proceeds as follows:

r'	$=$	r	since r does not change when e is even
b'	$=$	$b * b$	by line 15
e'	$=$	$e/2$	by line 7
	$=$	$2 * n/2$	because e is even
	$=$	n	by math
$r' * b'^{e'}$	$=$	$r * (b * b)^n$	by math from the above
	$=$	$r * b^{2*n}$	by math (since $(a^2)^c = a^{2*c}$)
	$=$	$r * b^e$	by math (since we set $e = 2 * n$)
	$=$	x^y	by assumption

Case e is odd, thus $e = 2 * n + 1$ for some non-negative n. Then, the proof proceeds as follows:

r'	$=$	$b * r$	by line 13
b'	$=$	$b * b$	by line 15
e'	$=$	$e/2$	by line 16
	$=$	$(2 * n + 1)/2$	because e is odd
	$=$	n	by math (definition of integer division)
$r' * b'^{e'}$	$=$	$(b * r) * (b * b)^n$	by math from the above
	$=$	$(b * r) * b^{2*n}$	by math (since $(a^2)^c = a^{2*c}$)
	$=$	$r * b^{2*n+1}$	by math (since $a * (a^c) = a^{c+1}$)
	$=$	$r * b^e$	by math (since we set $e = 2 * n + 1$)
	$=$	x^y	by assumption

This shows that both loop invariants hold on every iteration.

15 Termination

The previous argument for termination still holds. By the 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, as an integer, means it must eventually become 0, upon which the loop guard becomes false causing to exit the loop and return from the function after one additional step.

Now that we have learned about priming a variable to refer to its value after the current iteration of the loop, we can make this termination argument more precise. We will do so in the context of our example, where the expression we claim is strictly decreasing is e and the lower bound is 0. What we want to show is that, during an arbitrary iteration of the loop

If $e \geq 0$, *then* $e' < e$ *and* $e' \geq 0$

The checks $e \geq 0$ and $e' \geq 0$ state that 0 is indeed a lower bound for our expression. Instead, $e' < e$ says that e is strictly decreasing.

Let's put ourselves in an arbitrary iteration of the loop. Since we enter the body, the loop guard holds and therefore $e > 0$. This satisfies the antecedent of this property: $e \geq 0$. We then need to show that both consequents have to be true:

$e' < e$: By line 15, we know that $e' = e/2$. For strictly positive integers (recall that $e > 0$ — line 8), mathematics tell us that $e/2 < e$.

$e' \geq 0$: In this case, we use the exact same argument to show that $e/2 > 0$.

16 A Surprise

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

```
% coin mystery2f.c0 -d
C0 interpreter (coin) 0.3.3 'Nickel'
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 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.

17 Summary: Contracts, and Why They are Important

We have introduced *contracts*, using the example of an algorithm for integer exponentiation.

Contracts are expressed in the form of annotations, started with `//@`. These annotations are checked when the program is executed if it is compiled or interpreted with the `-d` flag. Otherwise, they are ignored.

The forms of contracts, and how they are checked, are:

`@requires`: A precondition to a function. This is checked just before the function body executes.

`@ensures`: A postcondition for a function. This is checked just after the function body has been executed. We use `\result` to refer to the value returned by the function to impose a condition on it.

`@loop_invariant`: A loop invariant. This is checked every time just before the loop exit condition is tested.

`@assert`: An assertion. This is like a statement and is checked every time it is encountered during execution.

Contracts are important for two purposes.

Testing: Contracts represent a kind of generic test of a function. Rather than stating specific inputs (like `f(2,8)` and testing the answer `256`), contracts talk about expected properties for *arbitrary* values. On the other hand, contracts are only useful in this regard if we have a good set of test cases, because contracts that are not executed with values that cause them to fail cannot cause execution to abort.

Reasoning: Contracts express properties of programs so we can *prove* them. Ultimately, this can mathematically verify program correctness. Since correctness is *the* most important concern about programs, this is a crucial aspect of program development. Different forms of contracts have different roles, reviewed below.

The proof obligations for contracts are as follows:

`@requires`: At the call sites we have to prove that the precondition for the function is satisfied for the given arguments. We can then assume it when reasoning in the body of the function.

`@ensures`**:** At the return sites inside a function we have to prove that the postcondition is satisfied for the given return value. We can then assume it at the call site.

`@loop_invariant`**:** We have to show:

Initialization: The loop invariant is satisfied initially, when the loop is first encountered.

Preservation: Assuming the loop invariant is satisfied at the beginning of the loop (just before the exit test), we have to show it still holds when the beginning of the loop is reached again, after one iteration of the loop.

We are then allowed to assume that the loop invariant holds after the loop exits, together with the exit condition.

`@assert`**:** We have to show that an assertion is satisfied when it is reached during program execution. We can then assume it for subsequent statements.

Contracts are crucial for reasoning since (a) they express what needs to be proved in the first place (give the program's *specification*), and (b) they *localize* reasoning: from a big program to the conditions on the individual functions, from the inside of a big function to each loop invariant or assertion.

18 Reasoning about Code

Throughout this lecture, we have been reasoning about code. We did so to prove that our candidate loop invariants for f were valid, to argue that f terminated, and that its fixed version was correct. But how do we reason about code in general?

There are two common ways to reason about code:

- *Logical reasoning* is using what we know to be true to prove certain properties about code.
- *Operational reasoning* is drawing conclusions about how things change when certain lines of code are run.

Both are sound reasoning strategies, but operational reasoning can be tricky. In particular, it is easy to make mistakes when reasoning operationally about large pieces of code (and even small ones).

For this reason, every time we need to reason about code, we will use **point-to reasoning**, which relies on any form of logical reasoning as well as one very simple form of operational reasoning. Intuitively, to justify a reasoning step using point-to reasoning, we need to be able to point to a line of code that supports our argument (or rely on general math principles). With the only exception of termination, all the proofs in this course will use point-to reasoning exclusively.

To make things a bit more concrete, we will demonstrate various forms of logical and of operational reasoning on the following toy program. Here, we use reasoning to justify the many `//@assert` annotations in the program. The comment after each of them uses logical or operational reasoning for this purpose. Comments in green are uses of logical or operational reasoning that are valid forms of point-to reasoning. Comments in red are disallowed uses of operational reasoning.

Do not yet look at these justifications in detail. Instead, skip just past the program to learn about common forms of logical and of operational reasoning, and which among them are also valid forms of point-to reasoning. As you learn about them, to back to the code to see them in action.

```
1 int g(int w)
2 //@requires w >= 0;
3 //@ensures \result > 0;
4 {
5   return w + 3;
6 }
```

```
7
8  int f(int x)
9  //@requires 0 <= x && x < 42;
10 {
11   int y = 42;
12   //@assert x >= 0;         // by line 9 (precondition)
13   //@assert y == 42;        // by line 11 (assignment to y)
14   int z = 84;
15   int k = 168;
16   //@assert z == 2*y;       // by math (laws of modular arithmetic)
17                             //    on lines 11 (or 13) and 14
18   while (y > x)
19     //@loop_invariant y >= x;
20     //@loop_invariant z == 2*y;
21     {
22       //@assert y > x;      // by line 18 (loop guard)
23       //@assert z == 2*y;   // by line 20 (loop invariant 2)
24       y = y-1;
25       z = z-2;
26       k = k-4;
27     }
28
29   //@assert y <= x;          // by line 18 (negation of the loop guard)
30   //@assert y >= x;          // by line 19 (loop invariant 1)
31   //@assert y == x;          // by math (laws of two's complement)
32                              //    on lines 29 and 30
33   //@assert z == 2*y;        // by line 20 (loop invariant 2)
34   //@assert z - x == 2*y - x;   // by math (laws of modular arithmetic)
35                                 //    on line 33
36   //@assert k == 4*y;        // NOT POINT-TO REASONING (peeking inside
37                              //    a loop body)
38   z = g(x);
39   //@assert z >= 0;          // by line 3 (postcondition of g)
40   //@assert z >= 3;          // NOT POINT-TO REASONING (peeking inside
41                              //    a function body)
42   if (x > 0) {
43     //@assert x > 0;         // by line 42 (conditional)
44     //@assert x+1 > 0 || x == int_max();  // by math (laws of two's
45                                           //    complement) on line 42
46     y = x;
47     //@assert y == x;        // by line 46 (assignment to y)
48     //@assert y > 0;         // by math (laws of logic) on lines 42 and 46
49   }
50   else {                     // by line 50 (negation of conditional)
51     //@assert x <= 0;
```

```
52      x = k;
53      // ???? assert x > 0;  // NOT POINT-TO REASONING (no info about k)
54    }
55
56    //@assert z >= 0;         // by line 3 (postcondition of g)
57    return z;
58  }
```

Let's start with **logical reasoning**. We said that logical reasoning is

using what we know to be true to prove certain properties about code.

The following table gives examples of logical reasoning relative to the above program. We group these situations into three classes: boolean conditions, contracts, and math. (The last two situations in the "math" group will be defined precisely in the next lecture.)

Every kind of logical reasoning is a valid form of point-to reasoning.

Logical Reasoning

Method	Situation	E.g.	Point-to?
Boolean conditions	Condition of an `if` statement in the "then" branch	L. 43	YES
	Negation of the condition of an `if` statement in the "else" branch	L. 51	YES
	Loop guard inside the body of a loop	L. 22	YES
	Negation of the loop guard after the loop	L. 29	YES
Contracts	Preconditions of the current function	L. 12	YES
	Loop invariants inside the body of a loop	L. 23	YES
	Loop invariants after a loop	L. 30, 33	YES
	Postconditions of a function after calling it	L. 39	YES
	Earlier (fully justified) assertion (if still valid)	L. 56	YES
Math	Laws of logic	L. 48	YES
	Laws of modular arithmetic	L. 16, 34	YES
	Laws of two's complement (carefully)	L. 31 44	YES

Operational reasoning is

> *drawing conclusions about how things change when certain lines of code are run.*

The simplest kind of change is assigning a value to a variable. Using the fact that this variable has this value (until we assign this variable to something else) is the only form of operational reasoning that counts as point-to reasoning.

Other forms of operational reasoning are too error-prone to use in this class, and we will will not accept them in a proof. For example, it is tempting to reason about what happens in a loop by talking about all its iterations. Operational reasoning about loops are justifications that use words like "always", "never", and "all": if we find ourselves using these words (or being able to rephrase our justification in a way that uses them), we are not doing point-to reasoning (we can't point to a line of code as the justification). In general, when outside of a loop, **we should pretend that the body of the loop is not there**: we can only refer to its loop guard and loop invariants.

Operational reasoning about loops is not all bad. In fact, it is a good way to come up with potential loop invariants: if you think about something that refers to all iterations of a loop, turn it into a loop invariant and use point-to reasoning to check that it is valid.

Another error-prone form of operational reasoning is referring to the body of a function we are calling (unless it is a specification function). When reasoning about the outcome of such a function call, **we should pretend that the body of the function is not there**: we can only use its contracts (specifically its postconditions). *Specification functions* are different as they are typically a transcription of behind-the-scene mathematical properties.

Operational Reasoning

Situation	**E.g.**	**Point-to?**
Value of variables right after assigning to them	L. 13, 46	**YES**
Things that refer to the body of an earlier loop (but are not mentioned in its loop invariants)	L. 36	**NO**
Things that happen in a function we are calling (but are not mentioned in its postconditions) — *unless it's a specification function*	L. 40	**NO**
A previously true statement after a variable in it has changed	L. 53	**NO**

The only situation where we use (non point-to) operational reasoning is when proving that a loop terminates. The argument we use has the form

> *During an arbitrary iteration of the loop, the expression _____ always gets strictly bigger/smaller and can never become bigger/smaller than _____ on which the loop guard is false.*

Notice the use of "always" and "never". (We can use point-to reasoning also for termination, but we typically don't do it in this course.)

19 Exercises

Exercise 1 (sample solution on page 37)**.** *Find an input for* `f` *that fails our first guess of a loop invariant in section 3:*

```
//@loop_invariant POW(b,e) == POW(x,y);
```

Exercise 2 (sample solution on page 37)**.** *Consider the following function:*

```
int foo(int x)
//@requires ___________;
//@ensures ____________;
{
  int p = 0;
  for (int i = 0; i < x; i++)
  //@loop_invariant ____________;
  //@loop_invariant ____________;
  {
     p += POW(2, i);
  }
  return p;
}
```

where `POW` *is the power function defined in this lecture.*

After running this function on a few inputs, form a conjecture as to what it does. Then, express your conjecture by filling in the precondition with any constraints on the input `x` *and the postcondition with a description of what it computes. Finally, fill in the loop invariants that enables you to prove that the safety of every statement in the loop body and that the postcondition holds whenever the input satisfies the precondition.*

Exercise 3 (sample solution on page 37)**.** *The* greatest common divisor *(GCD) of two positive integers a and b is the largest integer d such that $a \,\%\, d = 0$ and $b \,\%\, d = 0$. The following function computes the GCD of a and b by trying all possible values for d from the smallest among a and b down to 1.*

```
1 int GCD(int a, int b)
2 //@requires a > 0 && b > 0;
3 //@ensures \result >= 1;
4 //@ensures a % \result == 0 && b % \result == 0;
5 {
6   int d = min(a, b);
7
```

```
8   while (d > 1)
9   //@loop_invariant d >= 1;
10  {
11    if (a % d == 0 && b % d == 0)
12      return d;
13    d = d - 1;
14  }
15  return d;
16 }
```

Using the methodology studied in this chapter and point-to reasoning, we will show that this code is correct. Recall that correctness means that the postconditions must be true for any input that satisfies the preconditions. Note that the postconditions say nothing about the returned value being the greatest *common divisor of the inputs, only that it is one of their divisors.*

We will proceed in a number of steps.

a [INIT] Show that the loop invariant on line 9 holds just before checking the loop guard for the very first time.

b [PRES] Show that it is preserved by an arbitrary iteration of the loop.

c [EXIT] Show that the loop invariant and the negation of the loop guard imply the postconditions.

d [TERM] Show that the loop terminates.

e But what if the function exits on line 12? Using point-to reasoning, show that the postconditions are satisfied also in this case.

Exercise 4 (sample solution on page 38)**.** Euclid's algorithm *computes the greatest common divisor of two numbers, a problem we already explored in Exercise 3. It is often more efficient, but not as obviously correct. In this exercise, we will use the methodology developed in his chapter to convince ourselves (and others) of its correctness.*

This code uses the function `GCD` *from Exercise 3 as a specification function.*

```
1 int euclid(int a, int b)
2 //@requires a > 0 && b > 0;
3 //@ensures \result == GCD(a, b);
4 {
5   int x = a;
6   int y = b;
```

```
7
8   while (x != y)
9   //@loop_invariant x > 0 && y > 0;
10  //@loop_invariant GCD(x, y) == GCD(a, b);
11  {
12    if (x > y)
13      x = x - y;
14    else
15      y = y - x;
16  }
17  //@assert x == y;
18  return x;
19 }
```

We will follow the usual steps to prove correctness, plus one, to ensure safety.

a *Show that the calls to* `GCD` *on line 10 are safe.*

b *[INIT] Show that the loop invariants on lines 9–10 hold just before checking the loop guard for the very first time.*

c *[PRES] Show that they are preserved by an arbitrary iteration of the loop.*

d *[EXIT] Show that the loop invariants and the negation of the loop guard imply the postconditions.*

e *[TERM] Show that the loop terminates.*

Sample Solutions

Solution of exercise 1 A call that causes this loop invariant to fail is `f(2,7)`.

Solution of exercise 2 This function computes $2^x - 1$ according to the formula

$$2^x - 1 = \sum_{i=0}^{x-1} 2^i$$

Here is the resulting code with all contracts filled in:

```
int foo(int x)
//@requires x >= 0;
//@ensures \result == POW(2,x) - 1;
{
  int p = 0;
  for (int i = 0; i < x; i++)
  //@loop_invariant 0 <= i && i <= x;
  //@loop_invariant p == POW(2,i) - 1;
  {
     p += POW(2, i);
  }
  return p;
}
```

Solution of exercise 3

a [INIT] Show that the loop invariant on line 9 holds just before checking the loop guard for the very first time.

We need to show that `d >= 1` initially.

A. `a > 0` and `b > 0`	by line 2
B. `a >= 1` and `b >= 1`	by math on A
C. `d == min(a,b)`	by line 6
D. `min(a,b) >= 1`	by math on B and C

b [PRES] Show that it is preserved by an arbitrary iteration of the loop.

We need to show that if `d >= 1` as we enter an arbitrary iteration of the loop, then `d' >= 1`.

A.	`d > 1`	by line 8
B.	`d' == d-1`	by line 13
C.	`d' >= 1`	by math on A and B

In this proof, we did not make use of the assumption that `d >= 1`. This is not typical as most proofs of preservation rely critically on their assumption.

c [EXIT] Show that the loop invariant and the negation of the loop guard imply the postconditions.

We need to show that `d >= 1` and `a % d == 0 && b % d == 0` hold on line 15.

A.	`d >= 1`	by line 9
B.	`d <= 1`	by line 8
C.	`d == 1`	by math on A and B
D.	$n \,\%\, 1 = 0$ for any $n > 0$	by math

This proof suggest simplifying line 15 into `return 1`, since this is the only possible value that `d` can assume.

d [TERM] Show that the loop terminates.

During an arbitrary iteration of the loop, the expression `d` strictly decreases and can never get smaller than 1.

e But what if the function exits on line 12? Using point-to reasoning, show that the postconditions are satisfied also in this case.

We need to show that `d >= 1` and `a % d == 0 && b % d == 0` hold on line 12.

A.	`d >= 1`	by line 9
B.	`a % d == 0 && b % d == 0`	by line 11

Solution of exercise 4

a) Show that the calls to `GCD` on line 10 are safe.

We need to show that the preconditions of both calls to `GCD` are satisfied, i.e., that `x > 0 && y > 0` for `GCD(x,y)` and `a > 0 && b > 0` for `GCD(a,b)`.

A.	`x > 0 && y > 0`	by line 9
B.	`a > 0 && b > 0`	by line 2

b) [INIT] Show that the loop invariants on lines 9–10 hold just before checking the loop guard for the very first time.

We need to show that `x > 0 && y > 0` and that `GCD(x,y) == GCD(a,b)`. initially.

A.	`x = a`	by line 5
B.	`y = b`	by line 6
C.	`a > 0 && b > 0`	by line 2
D.	`GCD(a,b) == GCD(a,b)`	by math on A–B

c) [PRES] Show that they are preserved by an arbitrary iteration of the loop.

Assuming that `x > 0 && y > 0` and that `GCD(x,y) == GCD(a,b)`, we need to show that `x' > 0 && y' > 0` and `GCD(x',y') == GCD(a,b)`.

We need to consider two cases depending on whether `x > y` or `x < y`.

Assume that `x > y`:

A.	`x > y`	by assumption
B.	`x - y > 0`	by math on A
C.	`x' == x - y`	by line 13
D.	`x' > 0`	by B–C
E.	`y' > 0`	by line 9 and `y` is unchanged
F.	$GCD(n, m) == GCD(n - m, m)$	by math
G.	`GCD(x-y,y) == GCD(a,b)`	by math on E and line 9

Assume that `x < y`: the proof is similar.

d) [EXIT] Show that the loop invariants and the negation of the loop guard imply the postconditions.

A.	`x == y`	by line 8
B.	`GCD(x,x) == GCD(a,b)`	by line 10
C.	$GCD(n, n) == n$ for any $n > 0$	by math
D.	`x == GCD(a,b)`	by math on C-D

Step (A) is also noted as an assertion on line 17.

e) [TERM] Show that the loop terminates.

During an arbitrary iteration of the loop, the expression `x + y` is strictly decreasing and can never get smaller than 2. The fact that `x + y` is strictly decreasing relies on the fact that `x` and `y` are positive by the second loop invariant (line 10): if `x > y` the expression `x + y` is updated to `x`, and otherwise it is updated to `y`, either of which is strictly smaller than `x + y`. The lower bound (2) is a consequence of the first loop invariant.