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 pre- and post-conditions for functions, loop invariants, and other assertions about programs. 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 demystifying the program and proving it correct.

In term of our learning goals, this lecture addresses:

**Computational Thinking:** Specification versus implementation; correctness of programs

**Algorithms and Data Structures:** See Section 17

**Programming:** Contracts

If you have not seen this example, we invite you to read these lecture notes 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.

```c
int f (int x, int y)
{
    while (x != y) {
        if (x > y) { x = x - y; }
        else { y = y - x; }
    }
    return 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 any
problems.
3 Forming a Conjecture

The first step is to execute the program on some input values and examine the results. The code is in a file called `mystery3.c0` so we invoke the `coin` interpreter to let us experiment with the function.

```%
coin mystery3.c0
Coin 0.3.0 'Nickel'(r103, Mon Aug 27 15:30:29 EDT 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 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);
1 (int) --> f(2,4);
2 (int) --> f(9,12);
3 (int) --> f(20,20);
20 (int) -->```

Can you form a conjecture from these values?
From these and similar examples, you might form the conjecture is that $f(x, y)$ return the greatest common divisor (GCD) of $x$ and $y$. One can try confirm that with a few more values, such as

```plaintext
--> f(18, 12);
  6 (int)
--> f(2000, 5000);
  1000 (int)
--> 
```

It seems to work out! We could now try to find the error by inspecting the code, or try to see why this function actually computes the GCD. See if you can find inputs on which $f$ does not return the GCD.
4 Finding Counterexamples

Let’s look at the code again.

```c
int f (int x, int y)
{
    while (x != y) {
        if (x > y) { x = x - y; }
        else { y = y - x; }
    }
    return x;
}
```

We notice the following: when \( x = y \), the loop is never executed, so we always return \( x \). This is correct as long as \( x \) is positive, but it is not correct for negative numbers. For example, \( \text{GCD}(−3, −3) = 3 \), but \( f(−3, −3) = −3 \). Assuming \( f \) is really supposed to return the GCD, we have found a bug!

We can also notice that if \( x > 0 \) and \( y = 0 \), then in the loop we set \( x \) to \( x − y = x − 0 = x \), so both \( x \) and \( y \) remain unchanged. So we loop forever. We have found a second bug! Indeed:

```
--> f(2,0);
^C
KeyboardInterrupt
Last position: mystery3.c0:10.16-10.25
    f from <stdio>:1.1-1.7
```

We have to type Ctrl-c (holding down the control key and c at the same time) to interrupt the program which has gone into an infinite loop.
5 Imposing a Precondition

In mathematics it is sometimes convenient to define the greatest common divisor only on the positive integers 1, 2, ... . Perhaps the function \( f \) was intended to be only used on positive integers. The counterexamples so far use zero and negative values, so we conjecture that the function is correct, but only on positive inputs.

In C0, preconditions for functions are expressed using an annotation @requires which comes before the function body. We write:

\[
\text{int } f \ (\text{int } x, \text{int } y) \\
//@requires x > 0 \&\& y > 0; \\
\{ \\
\ \ \ \ \text{while } (x \neq y) \{ \\
\ \ \ \ \ \ \ \text{if } (x > y) \{ x = x - y; \} \\
\ \ \ \ \ \ \ \text{else } \{ y = y - x; \} \\
\ \ \ \ \} \\
\ \ \ \ \text{return } x; \\
\}
\]

This is the first example of a contract: The caller of this function is required to make sure that the arguments are positive. They also must be integers, and that is already expressed by writing \( f(\text{int } x, \text{int } y) \). A nice properties of \textit{types} such as \textit{int} is that the compiler (\textit{cc0}, in this case) or the interpreter (\textit{coin}, in this case) checks that the types are respected and issues an error message if they are not. Contracts, on the other hand, are \textit{not} verified before the program is executed.

Each of the tools (compiler \textit{cc0}, interpreter \textit{coin}, or debugger \textit{code}) takes a flag \(-d\) (where “\(d\)” stands for \textit{dynamic checking}). If this flag is specified, all contracts will be checked while the program is executing. This may slow down the program significantly, but it will immediately report contract violations, helping you in writing and debugging your code. Keep in mind that contracts are only tested when the program is executed, so it is very important to have good test cases.

If you do not specify the \(-d\) flag, then the contracts are treated just like a comment. Your program will be much faster, but it may be more difficult to discover errors since the source of the problem (like a contract violation) and its manifestation (like an infinite loop) may be hard to connect.
6 Promising a Postcondition

A contract is between two parties. In the case of programming, a contract is between a caller and a callee. The caller promises to respect the precondition of the function (expressed in a @requires annotation), while the callee promises to respect the postcondition of the function. A postcondition is expressed in a @ensures annotation.

In this example, we would like to show that the function returns the GCD of \( x \) and \( y \). So:

```plaintext
int f (int x, int y)
//@requires x > 0 && y > 0;
//@ensures \result == GCD(x,y);
{
    while (x != y) {
        if (x > y) { x = x - y; }
        else { y = y - x; }
    }
    return x;
}
```

The @ensures annotation uses a special keyword of the language, \( \text{\texttt{\result}} \), which stands for the result returned by the function. It can only be used in the postcondition, because only then do we know the return value.

There are two problems with this postcondition: first, we do not have a function \texttt{GCD}! And if we did, why would we implement it again? The answer is that \texttt{GCD} might be a really inefficient specification of the function and we want to give a more efficient implementation. This is actually very common case. We postpone writing the specification of the greatest common divisor for now, and simply assume it has been written. You can find it in Section 16. \texttt{GCD} also requires that its arguments are positive, which fortunately we know.

The second problem is that \( x \) and \( y \) are modified inside the function. But we would like the contract to apply to the original arguments to the function: as a caller, we have no control over (and should not care) what happens inside the function.

In order to solve the second problem we introduce some local variables (that is, for use only inside the function) called \( a \) and \( b \) that take the role of \( x \) and \( y \). Local variables have to be declared with their type. We also initialize them immediately to \( x \) and \( y \), respectively.
int gcd (int x, int y)
//@requires x > 0 && y > 0;
//@ensures \result == GCD(x,y);
{ int a = x;
    int b = y;
    while (a != b) {
        if (a > b) { a = a - b; }
        else { b = b - a; }
    }
    return a;
}

We also renamed the function to gcd to clarify our suspicion and follow reasonable naming conventions. Note that gcd and GCD are two different functions: the latter one is only used as a specification function in contracts, and we assume the usual mathematical properties for it, while gcd is the implementation we define right now.
7 Contracts So Far

If we look just at the precondition and postconditions of the function (always listed first, before the body)

```plaintext
int gcd (int x, int y)
//@requires x > 0 && y > 0;
//@ensures \result == GCD(x,y);
```

we can already use it in our own programs. As long as we make sure to call it only with positive arguments, the function promises to return the greatest common divisor as a result. This allows us to call this function without looking at its implementation at all! The importance of this point cannot be overemphasized. In order to reason about a big program with many functions and function calls, using contracts will allow us make the reasoning local: as long as each function adheres to its contract, and we reason correctly, the overall program will work correctly.

We should also examine when contracts are actually checked during the program development phase, assuming we use the -d flag, of course.

The precondition of a function will be checked after the function is called, but just before the body is executed. If something goes wrong, the calling location is blamed.

The postcondition of a function will be checked just after the body is executed and the return value has been calculated, but just before the return statement is executed. If something goes wrong, the return statement is blamed (and therefore implicitly the function containing it).
8 Finding a Loop Invariant

We have formed a reasonable conjecture (for which we know no counterexamples) and expressed it in the function contract. The next task is to see if we can prove that the function satisfies the contract. This means that we assume the precondition and have to prove the postcondition.

In this example, most of the computation takes place in a loop. Why does it do the right thing? Generally, this is very difficult to discover. Let’s work backwards, which is a good problem-solving heuristic in this case.

```plaintext
int gcd (int x, int y)
//@requires x > 0 && y > 0;
//@ensures \result == GCD(x,y);
{ int a = x;
  int b = y;
  while (a != b) {
    if (a > b) { a = a - b; }
    else { b = b - a; }
  }
  return a;
}
```

We return a. Why might this be the GCD of x and y? We notice that if the loop exits, a must be equal to b (otherwise we would have continued in the loop). We also remember (perhaps from high school), that the GCD of a positive number and itself must be that number. In other words, GCD(a, a) = a for a > 0. So what we are returning at the end of the function is actually the GCD of a and b.

How does this help? a and b start out as x and y. Now comes the really clever insight behind this program:

> The GCD of a and b remains the same no matter how many times we iterate through the loop.

Since a and b start out as x and y, the GCD of a and b (that never changes) is the same as the one for x and y.

In order to make this form of reasoning precise, we would like to express, in the program, what remains invariant in the loop. Like a pre- or post-condition, this should be a boolean expression which we can test. So we claim that the GCD of a and b is always the GCD of x and y.
int gcd (int x, int y)
//@requires x > 0 && y > 0;
//@ensures \result == GCD(x,y);
{ int a = x;
   int b = y;
   while (a != b)
     //@loop_invariant GCD(a,b) == GCD(x,y);
     {
       if (a > b) { a = a - b; }
       else { b = b - a; }
     }
   return a;
}
9 Heed Your Preconditions!

In the new loop invariant we call our specification function GCD, which requires its arguments to be positive. We know \( x \) and \( y \) are positive, by the function’s precondition, but we don’t know that \( a \) and \( b \) will always be positive. Turns out we are okay, because we can prove that they will always be positive: they are at the beginning of the loop (since \( x \) and \( y \) are), and they remain positive since because inside the loop we always subtract a strictly smaller number from a larger one. To express this more rigorously, we add a second loop invariant.

```c
int gcd (int x, int y)
//@requires x > 0 && y > 0;
//@ensures \result == GCD(x,y);
{ int a = x;
  int b = y;
  while (a != b)
    //@loop_invariant a > 0 && b > 0;
    //@loop_invariant GCD(a,b) == GCD(x,y);
    {
      if (a > b) { a = a - b; }
      else { b = b - a; }
    }
  return a;
}
```


10 Checking Loop Invariants

When are loop invariants checked (if we use the \(-d\) flag)?

*Loop invariants are checked just before the loop condition is evaluated.*

Why there? The main reason is that we want to exploit the loop invariant to show the postcondition of the function. If we check it just before we check whether we need to exit the loop, then we can assume the loop invariant holds just after the loop.
11 Proving Loop Invariants

The heart of the matter is now to prove that the claimed loop invariant is indeed a loop invariant. That means it must be true on every iteration just before the exit test for the loop.

We prove this in two easy pieces.

First (Init) we show that the loop invariant holds when we enter the loop the first time.

Second (Preservation) we show that the loop invariant is preserved. This means we assume it holds just before an iteration of the loop and show it holds again just after an iteration of the loop. These two points are actually the same, because after executing the body of the loop we go back to its beginning.

So we set up the proof for the two loop invariants. In order to justify our reasoning, it is convenient to assign line numbers to the code so we can refer to the lines in our proof. We have slightly reformatted the code to put a little less code on some of the lines. We also introduce a blank line just after the loop so we don't have to renumber when we later introduce another annotation.

```c
1 int gcd (int x, int y)
2 //@requires x > 0 && y > 0;
3 //@ensures \result == GCD(x,y);
4 { int a = x;
5 int b = y;
6 while (a != b)
7 //@loop_invariant a > 0 && b > 0;
8 //@loop_invariant GCD(a,b) == GCD(x,y);
9 { 
10 if (a > b) {
11 a = a - b;
12 } else {
13 b = b - a;
14 }
15 }
16 
17 return a;
18 }
```
Init. For the first loop invariant (line 7):
\[
\begin{align*}
a &= x & \text{by line (4) (assignment)} \\
> 0 & \text{by line (2) (precondition)}
\end{align*}
\]
\[
\begin{align*}
b &= y & \text{by line (5) (assignment)} \\
> 0 & \text{by line (2) (precondition)}
\end{align*}
\]
For the second loop invariant, line (8):
\[
\begin{align*}
\text{GCD}(a, b) &= \text{GCD}(x, b) & \text{by line (4)} \\
&= \text{GCD}(x, y) & \text{by line (5)}
\end{align*}
\]

Preservation. We assume both loop invariants:
\[
a > 0, b > 0, \text{GCD}(a, b) = \text{GCD}(x, y)
\]
We have to show that
\[
a' > 0, b' > 0, \text{GCD}(a', b') = \text{GCD}(x, y)
\]
where \(a'\) and \(b'\) represent the values of the program variables \(a\) and \(b\) after one iteration of the loop. Strictly speaking, we should also write \(\text{GCD}(x', y')\), but \(x\) and \(y\) do not change in the loop.

We distinguish three cases: \(a = b\), \(a > b\), and \(b < a\).

Case: \(a = b\). Then we immediately exit the loop, so we do not need to show preservation.

Case: \(a > b\). First, we want to show that the new values \(a'\) and \(b'\) for the variables \(a\) and \(b\) after one iteration are still positive.
\[
\begin{align*}
a' &= a - b & \text{by lines (10) and (11)} \\
> 0 & \text{by assumption } a > b
\end{align*}
\]
\[
\begin{align*}
b' &= b & \text{since } b \text{ is not assigned to in this case} \\
> 0 & \text{by line (7) (assumed loop invariant)}
\end{align*}
\]

Next we want to show the GCD is preserved. This is a bit trickier. We fall back on an observation made earlier which we will prove as a lemma. A lemma is an auxiliary fact that we prove separately and use in a main proof.
\[
\begin{align*}
\text{GCD}(a', b') &= \text{GCD}(a - b, b) & \text{by lines (10, 11)} \\
&= \text{GCD}(a, b) & \text{by Lemma and line (7) } (a > 0) \\
&= \text{GCD}(x, y) & \text{by line (8) (assumed loop invariant)}
\end{align*}
\]

Case: \(b > a\). Symmetric to the previous case.
12 A Lemma about GCD

Lemmas about the underlying mathematical nature of our data will often come up. Typically, we will not prove them in detail, but it is useful to see the kind of argument one has to make in such situations.

To complete the proof of the GCD function above we have to check a lemma, namely

Lemma 1 \( \text{GCD}(a, b) = \text{GCD}(a - b, b) \) if \( a > b > 0 \).

This is in fact the essential why the \text{gcd} function is correct, so it is worth considering it in detail.

Proof: We show that pairs \((a, b)\) and \((a - b, b)\) have exactly the same common divisors. Therefore, they also have the same greatest common divisor.

First Direction: Assume \(d\) is a common divisor of \(a\) and \(b\). So \(a = p \times d\) and \(b = q \times d\) for some positive integers \(p\) and \(q\). Then \(d\) is also a divisor of \(a - b\), which we can see as follows:

\[
    a - b = p \times d - q \times d = (p - q) \times d
\]

Hence \(d\) is a common divisor of \(a - b\) and \(b\).

Second Direction: Assume \(d\) is a common divisor of \(a - b\) and \(b\). So \(a - b = p \times d\) and \(b = q \times d\) for some positive integers \(p\) and \(q\). The \(d\) is also a divisor of \(a\), which we can see as follows:

\[
    a = (a - b) + b = p \times d + q \times d = (p + q) \times d
\]

Hence \(d\) is a common divisor of \(a\) and \(b\).
13 Proving the Postcondition

By now we have proven both loop invariants and it is time to bridge the gap to the postcondition. Let’s reexamine the function:

```c
1 int gcd (int x, int y)
2 //@requires x > 0 && y > 0;
3 //@ensures \result == GCD(x,y);
4 { int a = x;
5   int b = y;
6   while (a != b)
7     //@loop_invariant a > 0 && b > 0;
8     //@loop_invariant GCD(a,b) == GCD(x,y);
9     {
10       if (a > b) {
11         a = a - b;
12       } else {
13         b = b - a;
14       }
15     }
16   }
17   return a;
18 }
```

After the loop exits, on line 16, we know:

- \(a > 0, b > 0\) by line (7) (first loop invariant)
- \(\text{GCD}(a, b) = \text{GCD}(x, y)\) by line (8) (second loop invariant)
- \(a = b\) by line (6) (loop exit condition)

Now we reason as follows:

\[
\begin{align*}
\text{\texttt{result}} & = a & \text{by line (17)} \\
& = \text{GCD}(a, a) & \text{by line (7) (} a > 0 \text{) and Lemma}_2 \\
& = \text{GCD}(a, b) & \text{by line (6) (} a = b \text{)} \\
& = \text{GCD}(x, y) & \text{by line (8)}
\end{align*}
\]

We do not prove the elementary fact, here called Lemma}_2, that \(\text{GCD}(a, a) = a\) for \(a > 0\).
14 Termination

So far we have proved partial correctness of the gcd function.

**Definition:** A function is partially correct if its preconditions imply its postconditions, assuming it terminates.

We also have:

**Definition:** A function terminates if it returns a value.

A function that is partially correct and terminates is said to be totally correct. To prove our implementation of GCD to be totally correct, it remains to show that it terminates.

Since execution proceeds line by line, we only need to show that the loop terminates. In order to prove that a loop terminates we need to show a quantity that is bounded from below and strictly decreases on every iteration. This property is sometimes stated as saying that every strictly descending chain of natural numbers has a minimum.

Before you read on, can you find the termination argument?
The quantity that decreases is \( \max(a, b) \). This decreases because in each iteration we subtract a strictly positive number from the greater of the two numbers, therefore decreasing the maximum by at least one. The maximum of \( a \) and \( b \) is bounded from below by 1 (since \( a > 0 \) and \( b > 0 \) their maximum must be at least 1). Hence the loop (and also the function) must terminate.

In order to justify the strict decrease in more detail we consider 3 cases.

**Case:** \( a > b \). Then \( \max(a, b) = a \) and \( \max(a', b') = \max(a - b, b) < a \) since either \( \max(a - b, b) = b \) (and \( a > b \)) or \( \max(a - b, b) = a - b < b \) (since \( b > 0 \)).

**Case:** \( a = b \). Then we exit the loop.

**Case:** \( a < b \). Symmetric to the first case.
15 Assertions

So far we have seen pre- and post-conditions as well as loop invariants.

Another kind of “contract” is an assertion, written as an annotation `@assert`. In fact, it is a contract “with ourselves”: we are responsible for proving that whenever execution reaches the assertion it is true, but then we are allowed to assume it subsequently. An assertion allows us to localize reasoning even further, in essence introducing a kind of lemma into our program.

In the `gcd` program there are two places where an assertion may be helpful: in the else-branch of the conditional and after the loop. These are places where it is often advisable to clarify knowledge about the variables with an assertion.

```plaintext
1 int gcd (int x, int y)
2 //@requires x > 0 && y > 0;
3 //@ensures \result == GCD(x,y);
4 { int a = x;
5   int b = y;
6   while (a != b)
7     //@loop_invariant a > 0 && b > 0;
8     //@loop_invariant GCD(a,b) == GCD(x,y);
9     {
10       if (a > b) {
11         a = a - b;
12       } else { //@assert b > a;
13         b = b - a;
14       }
15     }
16     //@assert a == b && a > 0;
17   return a;
18 }
```

The proof obligations incurred by these assertions are easily discharged. We abbreviate it here by just stating that:

**Line 12:** $b \geq a$ is justified by lines (6) and (10).

**Line 16.1:** $a = b$ is justified by line (6)

**Line 16.2:** $a > 0$ is justified by line (7)
16 A Specification of GCD

A simple way to find the greatest common divisor of two positive numbers is to run through all common divisors and remember the greatest one. The following function uses this idea.

```c
int GCD(int x, int y)
//@requires x > 0 && y > 0;
{
    int cd = 1; /* initialize common divisor */
    int i = 2;
    while (i <= x && i <= y) {
        if (x % i == 0 && y % i == 0)
            cd = i;
        i = i+1;
    }
    return cd;
}
```

Since we start \( i \) at 2 and counting upwards, when we find a new common divisor it must be greater than the one we already have. So we can update \( cd \), our current understanding of what the greatest common divisor might be.
17 Summary: Contracts, and Why They are Important

We have introduced contracts, using the example of a version of Euclid’s algorithm for computing the greatest common divisor.

Contracts are expressed in 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 `\texttt{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 `gcd(9, 12)` and testing the answer 3), 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 cannot cause execution to abort.

**Reasoning**: Contracts express important 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:

- **Init**: 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.
Exercises

**Exercise 1** After reading 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 are interpreted in modular arithmetic and two’s complement representation of fixed range integers?

**Exercise 2** Rewrite first \texttt{gcd} so that it works for positive as well as negative numbers. Note that, for example, \texttt{GCD}(0, 0) is undefined (0 and 0 have arbitrarily large common divisors) and so this case should be excluded in the precondition.

**Exercise 3** After reading Lecture 3 on modular arithmetic, go back to your answer to the previous exercise. Is your function correct on all arguments, interpreted in two’s complement arithmetic? If not, make appropriate changes.

**Exercise 4** Usually, the GCD is implemented much more efficiently using the modulus operation, written in C0 as \texttt{a \% b} (\texttt{a} modulo \texttt{b}). Rewrite the \texttt{gcd} function to use the modulus operation to make it more efficient. How are the loop invariants affected? Write out the proof of correctness for this new, more efficient version of your function.