

Formal Reasoning and Verification Basics

School of Computing
Clemson University
Clemson, SC 29634 USA

CPSC 2150: Monday, Mar. 13, 2023

Slide Sources:

- 1 Previous/Current CPSC 2150 Instructors
- 2 The lecture slides have also benefitted from instructors at Ohio State: Paolo Bucci, Wayne Heym, Paul Sivilotti, and Bruce Weide.

Last Class

- 1 Recall
 - ▶ Blackbox
 - ▶ Whitebox
- 2 Blackbox Testing Examples
- 3 Path Testing
 - ▶ In-Class Exercises
- 4 Comparisons
- 5 Testing and Code Review Process

Homework Assignment

- 1 Programming assignment 3 is posted on your lab Canvas page.

Lab This Week

Lab this week is attendance required! If you don't attend, you will not receive any credit for this week's lab.

- ▶ Talk to your course instructor if you have any concerns

Quality Assurance Techniques

- ▶ Testing
- ▶ Tracing (or Inspection)
- ▶ Formal Reasoning (or Formal Verification)

- ▶ **Objective**

- ▶ Find inputs where the code is defective.
 - ▶ i.e., fails to behave as specified in its contract
- ▶ Successful tests show failures!

- ▶ **Approach**

- ▶ Execute the code on selected inputs and check if it fails.
- ▶ **Key question:** What inputs have the best likelihood of revealing defects?

Central Limitation of Testing

“Program testing can be used to show the presence of bugs, but never to show their absence!”

— Edsger W. Dijkstra (1972)

- ▶ This is where formal reasoning and verification comes in. . .

- ▶ **Objective**

- ▶ Prove using mathematical means that code functions correctly
 - ▶ i.e., behaves as specified in its contract.
- ▶ Establish code correctness on **all** inputs that satisfy preconditions!

- ▶ **Approach**

- ▶ Analyze, but not execute code
- ▶ Show logical errors, when proof fails

Recall

Tracing and Debugging

- ▶ **Objective**

- ▶ Locate errors using test inputs

- ▶ **Approach**

- ▶ Analyze the code on sample test inputs to understand why code fails to behave as specified in its contract

- ▶ **Tracing and Debugging**

- ▶ **Tracing:** Analyze, but do not execute code
 - ▶ **Debugging:** Execute code on selected inputs and follow the execution

Tracing Table

What does this code do?

<i>Code</i>	<i>State</i>
<pre>x = y + 17; y = x - 16; x = x - 18;</pre>	

Tracing Table

Putting It Together

Code	State
	$x = 10$ $y = 100$
$x = y + 17; \quad // \text{ S1}$ $y = x - 16; \quad // \text{ S2}$ $x = x - 18; \quad // \text{ S3}$	
	$x = 99$ $y = 101$

Tracing Table

What does this code do?

- ▶ First figure out specifically what it did for the example input values
- ▶ Then generalize. . .
 - ▶ **Warning:** This generalization may not be sound, in general, because you have traced it on only some inputs and that may be misleading!

Formal Reasoning

- ▶ Instead of tracing through the code with specific input values and generalizing
 - ▶ Error Prone
 - ▶ If we have an input value of 2, and a final value of 4 did it add 2, multiply by 2 or square the number?
 - ▶ Not a true mathematical proof
- ▶ Trace through the code with generalized or abstract values
 - ▶ Keep our variables in terms of the original value
 - ▶ $\#x$
 - ▶ Allows us to have a general result
 - ▶ $\#x + 2$
 - ▶ $\#x * 2$
 - ▶ $\#x^2$

Tracing Table

Reasoning Using Abstract Values

Code	State
	$x = \#x$ $y = \#y$
$x = y + 17;$ $y = x - 16;$ $x = x - 18;$	
	$x = ?$ $y = ?$

Tracing Table

Reasoning Using Abstract Values - Step 1

<i>Code</i>	<i>State</i>
	$x = \#x$ $y = \#y$
$x = y + 17;$	
	$x = ?$ $y = ?$

Tracing Table

Reasoning Using Abstract Values - Step 1

Code	State
	$x = \#x$ $y = \#y$
$x = y + 17;$	
	$x = \#y + 17$ $y = \#y$

Tracing Table

Reasoning Using Abstract Values - Step 2

Code	State
	$x = \#y + 17$ $y = \#y$
$y = x - 16;$	
	$x = ?$ $y = ?$

Tracing Table

Reasoning Using Abstract Values - Step 2

Code	State
	$x = \#y + 17$ $y = \#y$
$y = x - 16;$	
	$x = \#y + 17$ $y = (\#y + 17) - 16$

Tracing Table

Reasoning Using Abstract Values - Step 2

Code	State
	$x = \#y + 17$ $y = \#y$
$y = x - 16;$	
	$x = \#y + 17$ $y = \#y + 1$

Tracing Table

Reasoning Using Abstract Values - Step 3

Code	State
	$x = \#y + 17$ $y = \#y + 1$
$x = x - 18;$	
	$x = ?$ $y = ?$

Tracing Table

Reasoning Using Abstract Values - Step 3

Code	State
	$x = \#y + 17$ $y = \#y + 1$
$x = x - 18;$	
	$x = (\#y + 17) - 18$ $y = \#y + 1$

Tracing Table

Reasoning Using Abstract Values - Step 3

Code	State
	$x = \#y + 17$ $y = \#y + 1$
$x = x - 18;$	
	$x = \#y - 1$ $y = \#y + 1$

Tracing Table

Reasoning Using Abstract Values - Putting It Together

Code	State
	$x = \#x$ $y = \#y$
$x = y + 17; \quad // \text{ S1}$ $y = x - 16; \quad // \text{ S2}$ $x = x - 18; \quad // \text{ S3}$	
	$x = \#y - 1$ $y = \#y + 1$

Formal Reasoning

- ▶ We have our final values for x and y in terms of the original input values of x and y
- ▶ Want to know the final values for any specific input?
 - ▶ Just plug in those input values for $\#x$ and $\#y$ in the equation
- ▶ We now know our generalization is correct

Complications

- ▶ `if` statements, loops and recursions can make our proofs incredibly complicated
- ▶ Our expression of what the code does not need to be an statement of equality
 - ▶ Any Boolean comparison will work
- ▶ We still may be able to determine the outcome of a loop by determining what causes the loop to continue

Consider...

Consider all possible paths in the code listed below:

```
int x = getUserInput();  
if (x < 0) {  
    x = 0;  
}  
else if (x > 10) {  
    x = 10;  
}
```

// What can you determine about x?

Consider...

```
int x = getUserInput();
int y = getUserInput();

// remember x and y as #x and #y

if (y < 0) {
    y = y * -1;
}

for(int i = 0; i < y; i++) {
    x += x;
}

// Express x and y in terms of #x and #y
```

Methods

- ▶ What if while we are reasoning through our code, we call a method?
- ▶ Don't have to trace through the whole method
 - ▶ Are you meeting the precondition?
 - ▶ If not the code is wrong
 - ▶ Jump straight to the post condition
- ▶ This is why formal pre and postconditions are better!

Reasoning about Correctness

- ▶ Proving our code is without faults
- ▶ Show that the code does what its contract says it must do
- ▶ Show that the code does not violate any contracts on which it depends

Reasoning with Methods

- ▶ We use the contracts to reason about our `public` methods
- ▶ To prove the correctness of the method:
 - ▶ Assume the precondition is *true*
 - ▶ This eliminates junk data from our proofs
 - ▶ Assume invariants are *true*
 - ▶ Trace through with generalized values
 - ▶ Show that the postcondition and invariants are *true*
- ▶ Once that proof is completed, we know that our code is correct

Reasoning with Methods

- ▶ Since we use general values for our input, we can now show that this method will work on all inputs that meet the precondition!
- ▶ These proofs can be very complicated and tedious to write
- ▶ The proofs depend on the contracts
 - ▶ Precondition has to handle inputs that would break our proofs
 - ▶ Divide by zero errors
 - ▶ Negative values
 - ▶ etc.
- ▶ We can now easily reason with any code that uses these methods by skipping straight to the postcondition
 - ▶ As long as we meet the precondition

Contracts in Verification

- ▶ The role of contracts in formal reasoning and verification
- ▶ Preconditions
 - ▶ The starting point of our proof
 - ▶ What we know about the initial values
- ▶ Postconditions
 - ▶ What we are trying to prove to be *true*
- ▶ Invariants
 - ▶ Assume it is *true* at the beginning of the **public** method
 - ▶ Must prove it is still *true* at the end of the **public** method

Verification vs Testing

- ▶ The goal of testing is to find failures
 - ▶ Lets us know that a fault exists
 - ▶ Can't use testing to show that no faults exist
- ▶ The goal of verification is to prove that no faults exist
- ▶ If formal reasoning/verification shows that our code works on all inputs, while testing does not, why don't we just verify everything and skip testing
 - ▶ Verification is very difficult and time-consuming
 - ▶ Requires a ton of formal mathematical logic
 - ▶ Proofs must be very rigorous
 - ▶ Not everything can be verified (right now)

How to Use Verification and Reasoning

- ▶ Formal reasoning can help you understand how your code works
 - ▶ A better version of tracing
- ▶ You can verify simpler methods to build confidence in your code
 - ▶ Makes it easier to reason through the complicated code that calls the simpler methods

Homework Assignment

- 1 Programming assignment 3 is posted on your lab Canvas page.

Lab This Week

Lab this week is attendance required! If you don't attend, you will not receive any credit for this week's lab.

- ▶ Talk to your course instructor if you have any concerns

Summary

- 1 Recall
 - ▶ Testing
 - ▶ Central Limitation of Testing
 - ▶ Tracing and Debugging
- 2 Formal Reasoning
 - ▶ Contracts
 - ▶ Complications
- 3 Tracing Table Examples
 - ▶ In-Class Exercises