Discrete Math and Reasoning about Software Correctness

Murali Sitaraman (murali@clemson.edu)
http://www.cs.clemson.edu/group/resolve/

This research is funded in part by NSF grants CCF-0811748, CCF-1161916, DUE-1022191, and DUE-1022941.

Lecture 1
Intro to Program Correctness

Overview
- Software correctness proofs
- Connections between software correctness and discrete math foundations

Discrete Math
- Typical discrete math proof set up
  - $P \Rightarrow Q$
  - Same as
    - Assume $P$
    - Confirm $Q$

Software Correctness
- Code is correct if it meets its specifications.
- Specification of an operation involves pre- and post-conditions, which are logical assertions
- Basic set up
  - Assume pre-conditions
  - Code
  - Confirm post-conditions

Software Correctness
- Pre-conditions are givens or facts that you can assume
- Post-conditions are guarantees or obligations that you must confirm (prove)
- Basic set up
  - Assume pre-conditions
  - Code
  - Confirm post-conditions
Software Correctness

- Assuming the pre-conditions, if you execute the code that follows, would you be able to confirm the logical assertions at the end of the code?
- For now: Assume that nothing goes wrong during the execution of the code.

Example #1: Is Code Correct?

- Example #1 (b is a Boolean variable)
  
  Assume 
b;
  
  \[ b := \text{not} \ b; \]  
  \[ // := \text{is assignment} \]
  
  Confirm 
b;

- C code corresponding to the above 
  
  Assume 
b;
  
  \[ b = \text{!b}; \]  
  \[ // = \text{is assignment} \]
  
  Confirm 
b;

- We will use := for assignment, etc.

Example #2: Is Code Correct?

- Example #1 (b is a Boolean variable)
  
  Assume 
b;
  
  \[ b := \text{not} \ b; \]
  
  Confirm 
\[ \text{not} \ b; \]

- Same as:
  
  Assume 
b = \text{true};
  
  \[ b := \text{not} \ b; \]
  
  Confirm 
b = \text{false};

Correctness: A Method

- Example #1 (I is an Integer variable)
  
  Assume \text{true};
  
  \[ I := 0; \]
  
  Confirm \ I = 17;

- Replace the value of I in the Confirm assertion with the “effect” of the code and remove the code
- Above code becomes
  
  Assume \text{true};
  
  Confirm \ 0 = 17;

Correctness: Method Continued

- Above code becomes
  
  Assume \text{true}; Confirm \ 0 = 17;

- Same as
  
  \text{true} \Rightarrow 0 = 17;

- Same as
  
  \text{true} \Rightarrow \text{false};

- Same as
  
  \text{not true} \lor \text{false};

- Same as
  
  \text{false};
Correctness: A Method

- Example #1 (I is an Integer variable)
  
  
- Assume true;
  
  
- I := 0;
  
  
- Confirm I = 17;
  
  
- After many transformations, above code becomes false;
  
  
- Code is incorrect

Correctness: A General Method

- Example (I is an Integer variable)
  
  
- Assume ... ;
  
  
- ... -- some code
  
  
- Confirm I = 17;
  
  
- Replace the value of I and reduce to:
  
  
- Assume ... ;
  
  
- ... -- some code
  
  
- Confirm J = 17;
  
  
- Repeat the process till there are no more statements

Correctness: Method

- Code is reduced to a logical assertion (starting with the last statement in the method used here)

- Code is correct if and only if the logical assertion is provable

- We will use various techniques and results from discrete math to complete the proofs

A Slightly Larger Example

- Software is correct if it meets its specifications.

- Is the following code segment correct? Assume all variables are Integers.

  - Assume I = #I and J = #J;
  
  
  - Temp := I;
  
  
  - I := J;
  
  
  - J := Temp;
  
  
  - Confirm I = #J and J = #I;

Proof in 2 Parts

- Part 1:

  - Assume I = #I and J = #J;
  
  
  - Temp := I;
  
  
  - I := J;
  
  
  - J := Temp;
  
  
  - Confirm I = #J;

- Part 2:

  - Assume I = #I and J = #J;
  
  
  - Temp := I;
  
  
  - I := J;
  
  
  - J := Temp;
  
  
  - Confirm J = #I;

Proof of Part 1

- Assume I = #I and J = #J;

  - Temp := I;

  - I := J;

  - J := Temp;

  - Confirm I = #J;

- Replace J in the Confirm assertion with the effect of the highlighted statement
Simplify

Assume \( I = #I \) and \( J = #J \);

\[
\begin{align*}
& \ldots \\
& J := \text{Temp} \\
& \text{Confirm} \ I = #J;
\end{align*}
\]

becomes (there is no \( J \) to be replaced in the confirm assertion!)

Assume \( I = #I \) and \( J = #J \);

\[
\begin{align*}
& \text{Temp := I;} \\
& I := J; \\
& \text{Confirm} \ I = #J;
\end{align*}
\]

Simplify again

Assume \( I = #I \) and \( J = #J \);

\[
\begin{align*}
& \text{Temp := I;} \\
& I := J; \\
& \text{Confirm} \ I = #J;
\end{align*}
\]

Correctness Established

\[
\begin{align*}
& \text{Assume} \ I = #I \text{ and } J = #J; \\
& \text{Temp := I;} \\
& I := J; \\
& \text{Confirm} \ J = #J;
\end{align*}
\]

true, because \( P \) and \( Q \Rightarrow Q \);

Simplify one more time

Assume \( I = #I \) and \( J = #J \);

\[
\begin{align*}
& \text{Temp := I;} \\
& \text{Confirm} \ J = #J;
\end{align*}
\]

Correctness Established

\[
\begin{align*}
& \text{Assume} \ I = #I \text{ and } J = #J; \\
& \text{Temp := I;} \\
& I := J; \\
& \text{Confirm} \ J = #J;
\end{align*}
\]

Proof of Part 2

Assume \( I = #I \) and \( J = #J \);

\[
\begin{align*}
& \text{Temp := I;} \\
& I := J; \\
& J := \text{Temp;} \\
& \text{Confirm} \ J = #I;
\end{align*}
\]

\[
\begin{align*}
& \text{Assume} \ I = #I \text{ and } J = #J; \\
& \ldots \\
& J := \text{Temp;} \\
& \text{Confirm} \ J = #I;
\end{align*}
\]

\[
\begin{align*}
& \text{Assume} \ I = #I \text{ and } J = #J; \\
& \text{Temp := I;} \\
& I := J; \\
& \text{Confirm} \ \text{Temp} = #I;
\end{align*}
\]

\[
\begin{align*}
& \text{Assume} \ I = #I \text{ and } J = #J; \\
& \ldots \\
& J := \text{Temp;} \\
& \text{Confirm} \ J = #I;
\end{align*}
\]

\[
\begin{align*}
& \text{Assume} \ I = #I \text{ and } J = #J; \\
& \text{Temp := I;} \\
& I := J; \\
& \text{Confirm} \ \text{Temp} = #I;
\end{align*}
\]
Simplify again

Assume $I = \#I$ and $J = \#J$;
Temp := I;
I := J;
Confirm Temp = \#I;

Becomes (no $I$ in confirm assertion)
Assume $I = \#I$ and $J = \#J$;
Temp := I;
Confirm Temp = \#I;

Correctness Established

- Simplify one more time.
  Assume $I = \#I$ and $J = \#J$;
  Confirm $I = \#I$;

- The above is the same as:
  $(I = \#I$ and $J = \#J) \Rightarrow$
  $(I = \#I);$  

- true, because $P$ and $Q \Rightarrow P$;

Correctness Summary

- What did we just do?
- Mathematically prove that a piece of code is correct using integer theory and logic.
- The process is mechanical. You can do both parts in the proof simultaneously; we did it separately to make it easier.
- This is unlike testing where we can only show presence of errors, but not their absence.

Demonstration

- Demo site: [www.cs.clemson.edu/group/resolve/](http://www.cs.clemson.edu/group/resolve/)
- Click on Components Tab at the top
- Click on Facilities in the Finder
- Click on Int_Swap_Example_Fac
- Comment out operations Exchange2 and Main and their procedure using (* ...
- Click on Verify

Demonstration

- Specifications have the form
  - Operation ...
    - requires pre-conditions
    - ensures post-conditions
- Given a piece of code to implement the above specification, we get:
  - Assume pre-conditions
  - Code
  - Confirm post-conditions
- This is the starting point
Demonstration

- Specification
  - **Operation** Exchange (**updates** I, J: Integer);
  - **ensures** I = #I and J = #I;
- Values of parameters I and J are updated or modified by the operation
- Operation has no requires clause (pre-condition)
- In the ensures clause (post-condition), #I and #J denote input values

- Implementation code (procedure)

```plaintext
Temp := I;
I := J;
J := Temp;
```

- **Putting it together get:**

  - **Assume true**; -- no pre-conditions
  - **Code**
  - **Confirm** I = #J and J = #I;

Demonstration Summary

- Correctness proofs can be automated
- It is mostly tedious work
- Need an integrated specification-programming language like RESOLVE
- There are many specifications languages, just like there are many programming languages
  - ANNA, B, Dafny, Eiffel, JML,...VDM, Z
- All of them use discrete math notations!

Arithmetic Example

- Example (Ans, I, and J are Integers)

  - **Assume true**;
  - Ans := J - I;
  - **Confirm** Ans := I - J;
- Is the code correct?
- Why or why not?

Example: Is Code Correct?

- Example (Ans, I, and J are Integers)

  - **Assume true**;
  - Ans := J - I;
  - **Confirm** Ans = I - J;
- Is the code correct?
- Substituting J - I for Ans, we have
  - **Assume true**;
  - **Confirm** J – I = I – J;
- After more transformations:
  - `false` //universal statement
Another Arithmetic Example

- Example (Ans, I, and J are Integers)
  `Assume true;`  
  `Ans := J + I;`  
  `Confirm Ans := I + J;`
- Is the code correct?
- Why or why not?

Example: Is Code Correct?

- Example (Ans, I, and J are Integers)
  `Assume true;`  
  `Ans := J + I;`  
  `Confirm Ans = I + J;`
- Is the code correct?
- Substituting J + I for Ans, we have
  `Assume true;`  
  `Confirm J + I = I + J;`
- After more transformations:
  `true // + is commutative!`

Take Home Exercise #1: Prove

- Assume all variables are Integers.
  `Assume I = #I and J = #J;`  
  `I := I + J;`  
  `J := I - J;`  
  `I := I - J;`
  `Confirm I = #J and J = #I;`
- Do it in two steps.
  - Confirm I = #J
  - Confirm J = #I

From Before: Is Code Correct?

- Example (Ans, I, and J are Integers)
  `Assume true;`  
  `Ans := J + I;`  
  `Confirm Ans = I + J;`
- Is the code correct?
- Substituting J + I for Ans, we have
  `Assume true;`  
  `Confirm J + I = I + J;`
- After more transformations:
  `true // + is commutative!`

Example Demonstration

- Try the arithmetic example at the demo site
  - Type this in; comment out other ops; click verify and continue with slides!
  `Operation Ans(eval I, J: Integer): Integer;`  
  `ensures Ans = I + J;`  
  `Procedure Ans := J + I;`  
  `-- same as return(J + I);`
  `end;`
- Are we able to prove it automatically?
- Why or why not?
Computational vs. Math Integers

- Cannot verify because assignment could cause an overflow or underflow!
- We really need to confirm two things!

  \[ \text{Assume true;} \]
  \[ \text{Confirm min}_\text{int} \leq (I + J) \text{ and} \]
  \[ (I + J) \leq \text{max}_\text{int}; \]
  \[ \text{-- pre-condition for adding} \]
  \[ \text{Ans} := I + J; \]
  \[ \text{Confirm Ans} = I + J; \]
- System knows! It puts in the confirm assertion before assignment.

Computational vs. Math Integers

- This is provable because now we get:

  \[ \text{Assume min}_\text{int} \leq (I + J) \text{ and} \]
  \[ (I + J) \leq \text{max}_\text{int}; \]
  \[ \text{Confirm Ans} \leq (I + J) \text{ and} \]
  \[ (I + J) \leq \text{max}_\text{int}; \]
  \[ \text{Ans} := J + I; \]
  \[ \text{Confirm Ans} = I + J; \]

Exercise: Part 1, Step 1

- **Assume** \( I = #I \text{ and } J = #J; \)
  \[ I := I + J; \]
  \[ J := I - J; \]
  \[ I := I - J; \]
  \[ \text{Confirm } I = #J; \]
- becomes

  \[ \text{Assume } I = #I \text{ and } J = #J; \]
  \[ I := I + J; \]
  \[ J := I - J; \]
  \[ \text{Confirm } I - J = #J; \]

Take Home Exercise #1: Prove

- You know how to do the rest!
- What all do we have to confirm (prove) to show correctness of this code?
- Here’s the exercise.

  \[ \text{Assume } I = #I \text{ and } J = #J; \]
  \[ I := I + J; \]
  \[ J := I - J; \]
  \[ I := I - J; \]
  \[ \text{Confirm } I = #J \text{ and } J = #I; \]

\[ \text{Operation } \text{Ans}(\text{eval } I, J: \text{Integer}): \text{Integer}; \]
\[ \text{requires } \text{min}_\text{int} \leq (I + J) \text{ and} \]
\[ (I + J) \leq \text{max}_\text{int}; \]
\[ \text{ensures } \text{Ans} = I + J; \]

\[ \text{Procedure } \]
\[ \text{Ans} := J + I; \]
\[ \text{end}; \]
Take Home Exercise #1: Prove

- What all do we have to confirm (prove) to show correctness of this code?
- 5 verification conditions (VCs)!

Assume $I = \#I$ and $J = \#J$;

Confirm ...

- $I := I \ + \ J$; -- system knows
- $J := I \ - \ J$; -- system knows
- $I := I \ - \ J$; -- system knows
- $I = \#J$ and $J = \#I$; -- 2 here

Take Home Exercise: Demo

- Good exercise, but too tedious to work this out by hand.
- Comment out procedures Exchange and Main using (* ... * )
- Click on the VC tab, not the verify tab!
- See 4 blue icons (labeled VC) pop to the left; there are 5 VCs (one VC icon corresponds to 2)
- Click on the VC button to the left of ensures and look at VC 0_4

Exercise, Example VC

- Verification Condition VC 0_4
  Ensures Clause of Exchange2: Int_Swap_Example_Fac.fa(20)

  - Goal: $((I + J) - (I + J) - J)) = J$
  - Given:
    - $(\text{min} \_\text{int} <= 0)$
    - $(0 < \text{max} \_\text{int})$
    - $(\text{Last} \_\text{Char} \_\text{Num} > 0)$
    - $(\text{min} \_\text{int} <= J) \ and \ (J <= \text{max} \_\text{int})$
    - $(\text{min} \_\text{int} <= I) \ and \ (I <= \text{max} \_\text{int})$

VC is the same as implication

- Verification Condition VC 0_4
  Ensures Clause of Exchange2: Int_Swap_Example_Fac.fa(20)

  $(\text{min} \_\text{int} <= 0)$ and
  $(0 < \text{max} \_\text{int})$ and
  $(\text{Last} \_\text{Char} \_\text{Num} > 0)$ and
  $(\text{min} \_\text{int} <= J) \ and \ (J <= \text{max} \_\text{int})$ and
  $(\text{min} \_\text{int} <= I) \ and \ (I <= \text{max} \_\text{int})$

  $((I + J) - (I + J) - J)) = J$

Exercise, Example VC

- Verification Condition VC 0_5
  Ensures Clause of Exchange2: Int_Swap_Example_Fac.fa(20)

  - Goal: $(I + J) - J = I$
  - Given:
    - $(\text{min} \_\text{int} <= 0)$
    - $(0 < \text{max} \_\text{int})$
    - $(\text{Last} \_\text{Char} \_\text{Num} > 0)$
    - $(\text{min} \_\text{int} <= J) \ and \ (J <= \text{max} \_\text{int})$
    - $(\text{min} \_\text{int} <= I) \ and \ (I <= \text{max} \_\text{int})$

Exercise, Unprovable VC

- Verification Condition VC 0_1
  Requires Clause of Sum in Procedure Exchange2: Int_Swap_Example_Fac.fa(22)

  - Goal: $(\text{min} \_\text{int} <= (I + J))$ and $(I + J) <= \text{max} \_\text{int})$
  - Given:
    - $(\text{min} \_\text{int} <= 0)$
    - $(0 < \text{max} \_\text{int})$
    - $(\text{Last} \_\text{Char} \_\text{Num} > 0)$
    - $(\text{min} \_\text{int} <= J) \ and \ (J <= \text{max} \_\text{int})$
    - $(\text{min} \_\text{int} <= I) \ and \ (I <= \text{max} \_\text{int})$
### Exercise, Remaining VCs
- Verification Condition VC 0.2
  - Requires Clause of Difference in Procedure Exchange2: Int_Swap_Example_Fac.fa(23)
- Verification Condition VC 0.3
  - Requires Clause of Difference in Procedure Exchange2: Int_Swap_Example_Fac.fa(24)
- Are the above provable?
- Don’t click on the verify button (the timer is set short so even provable things won’t verify!)

### Demonstration Summary
- Correctness proofs can be automated
- It is mostly tedious work
- System knows and uses all kinds of results from discrete math
- Proof search might time out, so not all correct code might be proved automatically, but no wrong code would be proved!

### Discussion and Demo
- Math integers vs. computational integers
- Assume and confirm assertions come from requires and ensures clauses of operations
- Mechanical "proof" rules exist for all programming constructs, including if statements, while statements, objects, pointers, etc.
- They all have discrete math foundations.
- Demo site: [http://www.cs.clemson.edu/group/resolve/](http://www.cs.clemson.edu/group/resolve/)

### More Complex Examples
- Remaining Lecture: How do we deal with programs dealing with objects and operation calls on objects, beyond simple types like Integers
- Future Lecture 3: How do we prove correctness of non-trivial code involving loops and recursion (Hint: induction)

### Discrete Math and Math Modeling
- Discrete structures, such as numbers, sets, etc., are used in mathematical modeling of software
- Example using another discrete structure: Mathematical strings
- Strings are useful to specify and reason about CS structures, such as stacks, queues, lists, etc.

### Discrete Structure: Math Strings
- Unlike sets, strings have order
  - Example: Str(Z) for String of integers
- Notations
  - Empty string (written empty_string or Λ)
  - Concatenation ( alpha o beta )
  - Length ( |alpha| )
  - String containing one entry ( <5> )
Stacks and Queues

- Stacks and Queues of entries can be mathematically viewed as “strings of entries”
- Their operations can be specified using string notations

Stack Operations

- Example operation specifications
  - **Operation** Push (updates \( S: \text{Stack} \)) \( \text{requires} |S| < \text{Max\_Depth}; \)
  - **ensures** \( S = <\#E> \circ \#S; \)

- **Operation** Pop (updates \( S: \text{Stack} \)) \( \text{requires} |S| > 0; \)
  - **ensures** ...

- **Operation** Depth (\( S: \text{Stack} \)): Integer \( \text{ensures} \) Depth = \( |S| \);

Demonstration

- [www.cs.clemson.edu/group/resolve/](http://www.cs.clemson.edu/group/resolve/)
- Click on Components Tab at the top
- Click on Concepts in the Finder
- Select Stack_Template from the list
- Select Enhancements
- Select Do_Nothing_Capability
- Select Realizations
- Select Do_Nothing_Realiz
- Click on Verify

Do_Nothing, Example VC

- **VC 0_2**
  - Requires Clause of Push in Procedure Do_Nothing: Do_Nothing_Realiz.rb(6)
  - **Goal:** \(|S'| + 1| \leq \text{Max\_Depth}; \)
  - **Given:**
    - \(|S| \leq \text{Max\_Depth} \)
    - \(|S| > 0 \)
    - \(S = (<\text{Next\_Entry}'> \circ S')\)
  - Here, \( S \) and \( S' \) are values of Stack \( S \) in two different states

Discussion

- How many VCs (verification conditions) are there?
- Why?
- Proofs of these VCs involve discrete structure results involving mathematical strings
  - Theorem: \( \forall x : \Sigma, \ |<x>| = 1; \)
  - Theorem: \( \forall \alpha, \beta : \text{Str}(\Sigma), \]
    \( |\alpha \circ \beta| = |\alpha| + |\beta|; \)

Take Home Exercise #2

- Do the string theory tutorial at this site
- Do the exercises at the end of the tutorial
Lecture 3

Induction and Correctness of Programs with Loops and Recursion

Example

```c
int sum(int j, int k)
// requires j >= 0
// ensures result = j + k
{
    if (j == 0) {
        return k;
    } else {
        j--; int r = sum(j, k);
        return r + 1;
    }
}
```

Reasoning Pattern

- Similar to an inductive proof
- Base case (e.g., j == 0)
  - Reason code works for the base case
- Recursive case (e.g., j != 0)
  - Assume that the recursive call
    j--; r = sum(j, k) works
  - Reason that the code works for the case of j
- Show assumption is legit, i.e., show termination

Recursion: Base case

```c
int sum(int j, int k)
// requires j >= 0
// ensures result = j + k
{
    if (j == 0) {
        return k;
        // Assume: (j = 0) ^ (result = k)
        // Confirm ensures: result = 0 + k
    } else { ...
    }
}
```

Recursion: Inductive Assumption

```c
int sum(int j, int k)
// requires j >= 0
// ensures result = j + k
{
    if (j == 0) { ...
    } else {
        j--; int r = sum(j, k);
        // Assume: r = (j - 1) + k
        return r + 1;
        // Assume: (r = (j - 1) + k) ^
        // (result = r + 1)
        // Confirm ensures: result = j + k
    }
}
```

Recursion: Inductive Proof Step

```c
int sum(int j, int k)
// requires j >= 0
// ensures result = j + k
{
    if (j == 0) { ...
    } else {
        j--; int r = sum(j, k);
        return r + 1;
        // Assume: (r = (j - 1) + k) ^
        // (result = r + 1)
        // Confirm ensures: result = j + k
    }
```
Reasoning: Recursive Case

- For the inductive proof to be legit, the inductive assumption must be legit.
- This requires showing that an argument passed to the recursive call is strictly smaller.
- This is the proof of termination.
- To prove termination automatically, programmers need to provide a progress metric (\( j \) decreases in the example).

Discrete Math and Specifications

- Discrete structures, such as numbers, sets, etc., are used in mathematical modeling of software.
- Example using another discrete structure: Mathematical strings.
- Strings are useful to specify and reason about CS structures, such as stacks, queues, lists, etc.

Queue Flip Demonstration

- Click on Components Tab at the top.
- Click on Concepts in the Finder.
- Select Queue_Template from the list.
- Select Enhancements.
- Select Flipping_Capability.
- Select Realizations.
- Select Recursive_Flipping_Realiz.
- Click on Verify.

Discussion

- Discuss select VCs.

Reasoning about iterative code

- Also appeals to induction.
- Loops need to include an “invariant” and a progress metric for termination.
- Invariant is established for the base case (i.e., before the loop is entered).
- Invariant is assumed at the beginning of an iteration and confirmed at the beginning of the next.
- Also needs a progress metric and a proof of termination.

Theorems from string theory

- Various theorems from string theory are necessary to prove software correctness.
- VCs in proving correctness of recursive realization.
- Ex. Theorem: string reversal theorem:
  \[ \forall \alpha, \beta : \text{Str}(\Sigma), \text{Reverse}(\alpha \circ \beta) = \text{Reverse}(\beta) \circ \text{Reverse}(\alpha); \]
Stack Flip Realization

- Click on Components Tab at the top
- Click on Concepts in the Finder
- Select Stack_Template from the list
- Select Enhancements
- Select Flipping_Capability
- Select Realizations
- Select Obvious_Flip_Realiz
- Click on Verify

Discussion

- Discuss select VCs

Summary

- Discrete math foundations are central for developing correct software.
- Modern programming languages are being enhanced with specification language counterparts to facilitate verification of software correctness
- Example spec/programming language combinations include RESOLVE