Announcements

- Homework 4 is due tonight
- Wed forum: papers on automated testing using symbolic execution

Questions?

Review of Soundness and Completeness of Axiomatic Semantics

One-Slide Summary

- A system of axiomatic semantics is sound if everything we can prove is also true.
  \[ \text{if } \vdash (A) \implies (B) \text{ then } \models (A) \implies (B) \]
- We prove this by nested induction on the structure of the operational semantics derivation and the axiomatic semantics proof.
- A system of axiomatic semantics is complete if we can prove all true things.
  \[ \text{if } \models (A) \implies (B) \text{ then } \vdash (A) \implies (B) \]
- Our system is relatively complete (i.e., just as complete as the underlying logic). We use weakest preconditions to reason about completeness. Verification conditions are preconditions that are easy to compute.

Where Do We Stand?

- We have a language for asserting properties of programs
- We know when such an assertion is true
- We also have a symbolic method for deriving assertions

\[ \vdash (A) \implies (B) \text{ (symbolic derivation, theorem proving)} \]
\[ \models (A) \implies (B) \text{ (completeness)} \]
\[ \models (A) \implies (B) \text{ (soundness)} \]
Completeness of Axiomatic Semantics

- If \( \models (A) c (B) \) can we always derive \( \vdash (A) c (B) \)?
- If so, axiomatic semantics is complete
- If not then there are valid properties of programs that we cannot verify with Hoare rules :-(
  - Good news: for our language the Hoare triples are complete
  - Bad news: only if the underlying logic is complete
    (whenever \( \models A \) we also have \( \vdash A \))
    - this is called relative completeness

Proof Idea

- Dijkstra's idea: To verify that \( (A) c (B) \)
  a) Find out all predicates \( A' \) such that \( \models (A') c (B) \)
     - call this set \( \text{Pre}(c, B) \) (Pre = "pre-condition")
  b) Verify for one \( A' \in \text{Pre}(c, B) \) that \( A \Rightarrow A' \)
- Assertions can be ordered:

  \[
  \text{false} \quad \Rightarrow \quad \text{true}
  \]
  \[
  \begin{array}{c|c|c}
  \text{strong} & \text{pre-condition: wp(c, B)} & \text{weak} \\
  \hline
  A & \Rightarrow & \text{wp(c, B)}
  \end{array}
  \]
- Thus: compute \( \text{wp}(c, B) \) and prove \( A \Rightarrow \text{wp}(c, B) \)

Weakest Preconditions

- Define \( \text{wp}(c, B) \) inductively on \( c \), following the Hoare rules:
  \[
  \begin{array}{l}
  \text{wp}(c_1; c_2, B) = \{ A \} c_1 \{ C \} \text{ wp}(c_2, B) \\
  \text{wp}(x := e, B) = \{ [e/x]B \} x := E \{ B \} \\
  \text{wp}(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = \{ A \} c_1 \{ B \} \lor \{ A \} c_2 \{ B \} \\
  \{ E \Rightarrow A_1 \land \neg E \Rightarrow A_2 \} \text{ if } E \text{ then } c_1 \text{ else } c_2 \{ B \} \\
  \text{wp}(E \Rightarrow \text{if } E \text{ then } c_1 \text{ else } c_2, B) = \{ E \Rightarrow \text{wp}(c_1, B) \land \neg E \Rightarrow \text{wp}(c_2, B) \}
  \end{array}
  \]

Proof Idea

- Completeness of axiomatic semantics:
  If \( \models (A) c (B) \) then \( \vdash (A) c (B) \)
- Assuming that we can compute \( \text{wp}(c, B) \) with the following properties:
  - \( \text{wp} \) is a precondition (according to the Hoare rules)
    \( \vdash \{ \text{wp}(c, B) \} c \{ B \} \)
  - \( \text{wp} \) is the weakest precondition
    \[ A \Rightarrow \text{wp}(c, B) \]
  - We also need that whenever \( \models A \) then \( \vdash A \) !

Weakest Preconditions for while

- We start from the unwinding equivalence
  \[
  \text{while } b \text{ do } c = \text{if } b \text{ then } c \text{; while } b \text{ do } c \text{ else skip}
  \]
- Let \( w = \text{while } b \text{ do } c \) and \( W = \text{wp}(w, B) \)
- We have that
  \[ W = b \Rightarrow \text{wp}(c, W) \land \neg b \Rightarrow B \]
- But this is a recursive equation!
  - We know how to solve these using domain theory
  - But we need a domain for assertions

End of Review
A Partial-Order for Assertions

- Which assertion contains the least information?
  - “true” does not say anything about the state
- What is an appropriate information ordering?
  \[ A \sqsubseteq A' \quad \text{iff} \quad \sigma \models A' \implies A \]
- Is this partial order complete?
  - Take a chain \[ A_1 \sqsubseteq A_2 \sqsubseteq \cdots \]
  - Let \( \wedge A_i \) be the infinite conjunction of \( A_i \)
    \[ \sigma \models \wedge A_i \quad \text{iff} \quad \forall i \sigma \models A_i \]
- Can \( \wedge A_i \) be expressed in our language of assertions?
  - Often yes (see Winskel), we’ll assume yes for now

Weakest Precondition for while

- Use the fixed-point theorem
  \[ F(A) = b \implies wp(c, A) \land \neg b \implies B \]
- (Where did this come from? Three slides back!)
  - I assert that \( F \) is both monotonic and continuous
- The least-fixed point (= the weakest fixed point) is
  \[ wp(w, B) = \wedge F(\text{true}) \]

Weakest Preconditions

- Define a family of wp’s
  \[ wp(\text{while } e \text{ do } c, B) \] is weakest precondition on which the loop terminates in \( B \) if it terminates in \( k \) or fewer iterations
    \[ wp_0 = E \implies B \]
    \[ wp_k = E \implies wp(c, wp_{k-1}) \land \neg E \implies B \]
- \( \text{wp(while } e \text{ do } c, B) = \wedge_{k \geq 0} \text{wp}_k = \text{lub } \{ \text{wp}_k \mid k \geq 0 \} \]
- See notes on the web page for the proof of completeness with weakest preconditions
- Weakest preconditions are
  - Impossible to compute (in general)
  - Can we find something easier to compute yet sufficient?

Not Quite Weakest Preconditions

- Recall what we are trying to do:
  \[
  \begin{array}{c|c|c}
  \text{false} & \text{false} & \text{false} \\
  \text{true} & \text{false} & \text{true} \\
  \text{strong} & \text{false} & \text{true} \\
  \text{weak} & \text{false} & \text{true} \\
  \text{A} & \text{false} & \text{true} \\
  \text{precondition: WP(c, B)} & \text{false} & \text{true} \\
  \text{verification condition: VC(c, B)} & \text{false} & \text{true} \\
  \end{array}
  \]
- We shall construct a verification condition \( VC(c, B) \)
  - The loops are annotated with loop invariants!
  - \( VC \) is guaranteed stronger than WP
  - But hopefully still weaker than \( A \vdash A \implies VC(c, B) \implies WP(c, B) \)

Verification Condition Generation

- Factor out the hard work
  - Loop invariants
  - Function specifications (pre- and post-conditions)
- Assume programs are annotated with such specs
  - Good software engineering practice anyway
  - Requiring annotations = Kiss of Death?
- New form of while that includes a loop invariant:
  \[ \text{while}_\text{Inv} b \text{ do } c \]
  - Invariant formula \( \text{Inv} \) must hold every time before \( b \) is evaluated
  - A process for computing \( VC(\text{annotated_command, post_condition}) \) is called \( \text{VCGen} \)
Verification Condition Generation

- Mostly follows the definition of the wp function:

\[
\begin{align*}
\text{VC}(\text{skip}, B) & = B \\
\text{VC}(c_1; c_2, B) & = \text{VC}(c_1, \text{VC}(c_2, B)) \\
\text{VC}(\text{if } b \text{ then } c_1 \text{ else } c_2, B) & = b \Rightarrow \text{VC}(c_1, B) \land \neg b \Rightarrow \text{VC}(c_2, B) \\
\text{VC}(x := e, B) & = [e/x] B \\
\text{VC}(\text{let } x = e \text{ in } c, B) & = [e/x] \text{ VC}(c, B) \\
\text{VC}(\text{while } \text{Inv} b \text{ do } c, B) & = ?
\end{align*}
\]

- Inv is the loop invariant (provided externally)
- x_1, …, x_n are all the variables modified in c
- The \(\forall\) is similar to the \(\forall\) in mathematical induction:

\[
P(0) \land \forall n \in \mathbb{N}. P(n) \Rightarrow P(n+1)
\]

Example VCGen Problem

- Let’s compute the VC of this program with respect to post-condition \(x \neq 0\)

\[
\begin{align*}
x & = 0 \\
y & = 2 \\
\text{while } x+y=2 \land y > 0 \text{ do } \\
y & := y - 1 \\
x & := x + 1
\end{align*}
\]

First, what do we expect? What pre-condition do we need to ensure \(x \neq 0\) after this?

Example of VCGen

- \(\text{VC}(w, x \neq 0) = x+y=2 \land \forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow (x+1)+(y-1)=2 \land y \leq 0 \Rightarrow x \neq 0)\)
- \(\text{VC}(x := 0; y := 2; w, x \neq 0) = 0+2=2 \land \forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow (x+1)+(y-1)=2 \land y \leq 0 \Rightarrow x \neq 0)\)

So now we ask an automated theorem prover to prove it.

VCGen for while

\[
\begin{align*}
\text{VC}(\text{while } \text{Inv} e \text{ do } c, B) & = \text{Inv} \land (\forall x_1, \ldots, x_n. \text{Inv} \Rightarrow (e \Rightarrow \text{VC}(c, \text{Inv}) \land \neg e \Rightarrow B)) \\
\text{Inv} & \text{ holds on entry} \\
\text{Inv} & \text{ is preserved in an arbitrary iteration} \\
B & \text{ holds when the loop terminates in an arbitrary iteration}
\end{align*}
\]

Example of VCGen

- By the sequencing rule, first we do the while loop (call it \(w\)):

\[
\begin{align*}
\text{while } x+y=2 \land y > 0 \text{ do } \\
y & := y - 1 \\
x & := x + 1
\end{align*}
\]

- \(\text{VC}(w, x \neq 0) = x+y=2 \land \forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow \text{VC}(c, x+y=2) \land y \leq 0 \Rightarrow x \neq 0)\)
- \(\text{VC}(y := y - 1; x := x + 1, x+y=2) = (x+1)+(y-1)=2 \land y \leq 0 \Rightarrow x \neq 0)\)

- \(\text{w Result: } x+y=2 \land \forall x,y. x+y=2 \Rightarrow (y>0 \Rightarrow (x+1)+(y-1)=2 \land y \leq 0 \Rightarrow x \neq 0)\)

Prove it!

\[
\begin{align*}
& \text{\$ ./Simplify} \\
& > (\text{AND (EQ (+ 0 2) 2) (FORALL (x y) (IMPLIES (EQ (+ x y) 2) (AND (IMPLIES (EQ (+ x y) 2) (EQ (+ (+ x 1) (- y 1)) 2)) (IMPLIES (EQ (+ (+ x 1) (- y 1)) 2)))))}) \\
& : \text{Valid.}
\end{align*}
\]

- Great!
- Simplify is a non-trivial five megabytes
Can We Mess Up VCGen?

- The invariant is from the user (= the adversary, the untrusted code base)
- Let’s use a loop invariant that is false, like \( x \neq 0 \).
  \[ VC = 0 \neq 0 \land \forall x,y. x \neq 0 \Rightarrow (y > 0 \Rightarrow y \leq 0 \Rightarrow y \neq 0) \]
- Let’s use a loop invariant that is too weak, like "true".
  \[ VC = true \land \forall x,y. true \Rightarrow (y > 0 \Rightarrow true \land y \leq 0 \Rightarrow x \neq 0) \]

Prove it!

$ ./Simplify 
> (AND TRUE 
  (FORALL ( x y ) (IMPLIED TRUE 
    (AND (IMPLIED (y > 0) TRUE) 
      (IMPLIED (<= y 0) (NEQ x 0)))))

Counterexample: context:
(AND 
 (EQ x 0) 
 (<= y 0) 
)
l: Invalid.

- OK, so we won’t be fooled.

Soundness of VCGen

- Simple form
  \[ \vdash \{ VC(c, B) \} c \{ B \} \]
- Or equivalently that
  \[ \vdash VC(c, B) \Rightarrow wp(c, B) \]
- Proof is by induction on the structure of \( c \)
  - Try it!
- Soundness holds for any choice of the invariant!
- Next: properties and extensions of VCs

Applying Verification Condition Generation

One-Slide Summary

- Verification conditions make axiomatic semantics practical. We can compute verification conditions forward for use on unstructured code (= assembly language). This is sometimes called symbolic execution.
- We can add extra invariants or drop paths (dropping is unsound) to help verification condition generation scale.
- We can model exceptions, memory operations and data structures using verification condition generation.

Where Are We?

- Axiomatic semantics: The meaning of a program is what is true after it executes
  - Hoare triples: \( \{A\} c \{B\} \)
  - Weakest Precondition: \( \{wp(c,B)\} c \{B\} \)
  - Verification Condition: \( A \Rightarrow VC(c,B) \Rightarrow \wp(c,B) \)
    - Requires Loop Invariants
    - Backward VC works for structured programs

20
Plan for Applying VCGen

• Symbolic Execution and Forward VCGen
• Handling Exponential Blowup
  - Invariants
  - Dropping Paths
• VCGen For Exceptions (double trouble)
• VCGen For Memory (McCarthyism)
• VCGen For Structures (have a field day)
• VCGen For "Semantic Checksum"

VC and Invariants

• Consider the Hoare triple:
  \((x \leq 0) \text{ while } (x \leq 5) \Rightarrow x \equiv x + 1 (x = 6)\)
• The VC for this is:
  \(x \leq 0 \Rightarrow I(x) \land \forall x. (I(x) \Rightarrow (x > 5 \Rightarrow x = 6 \land x \leq 5 \Rightarrow I(x+1)))\)
• Requirements on the invariant:
  - Holds on entry
    \(\forall x. x \leq 0 \Rightarrow I(x)\)
  - Preserved by the body
    \(\forall x. I(x) \land x \leq 5 \Rightarrow I(x+1)\)
  - Useful
    \(\forall x. I(x) \land x > 5 \Rightarrow x = 6\)
• Check that \(I(x) = x \leq 6\) satisfies all constraints

Forward VCGen

• Traditionally the VC is computed backwards
  - That's how we've been doing it in class
  - It works well for structured code
• But it can also be computed forward
  - Works even for unstructured languages (e.g., assembly language)
  - Uses symbolic execution, a technique that has broad applications in program analysis
  - e.g., the PREfix tool (Intrinsa, Microsoft) works this way

Forward VC Gen Intuition

• Consider the sequence of assignments
  \(x_1 \leftarrow e_1; x_2 \leftarrow e_2\)
• The VC\((c, B) = [e_1/x_1][e_2/x_2]B\)
• We can compute the substitution in a forward way using symbolic execution (aka symbolic evaluation)
  - Keep a symbolic state that maps variables to expressions
  - Initially, \(\Sigma_0 = \{\}\)
  - After \(x_1 \leftarrow e_1\), \(\Sigma_1 = \{x_1 \rightarrow e_1\}\)
  - After \(x_2 \leftarrow e_2\), \(\Sigma_2 = \{x_1 \rightarrow e_1, x_2 \rightarrow [e_1/x_1]e_2\}\)
  - Note that we have applied \(\Sigma_1\) as a substitution to right-hand side of assignment \(x_2 \leftarrow e_2\)

Simple Assembly Language

• Consider the language of instructions:
  \(I ::= x ::= e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid L: \text{ return } \mid \text{inv } e\)
• The "inv e" instruction is an annotation
  - Says that boolean expression \(e\) holds at that point
• Each function \(f()\) comes with \(\text{Pre}_f\) and \(\text{Post}_f\) annotations (pre- and post-conditions)
• New Notation (yay!): \(I_k\) is the instruction at address \(k\)
Symbolic Execution Symbolic State

- We set up a symbolic execution state: 
  \( \Sigma : \text{Var} \rightarrow \text{SymbolicExpressions} \)
  \( \Sigma(x) \) = the symbolic value of \( x \) in state \( \Sigma \)
  \( \Sigma[x:=e] \) = a new state where \( x \)'s value is \( e \)
- We use states as substitutions:
  \( \Sigma(e) = e \) where \( x \) replaced by \( \Sigma(x) \)
- So far, much like operational semantics

Symbolic Execution Invariant State

- The symbolic executor keeps track of the encountered invariants
- A new part of symex state: \( \text{Inv} \subseteq \{1..n\} \)
- If \( k \in \text{Inv} \) then \( I_k \) is an invariant instr. that we have already executed
- Basic idea: execute an inv instruction only twice:
  - The first time it is encountered
  - Once more time around an arbitrary iteration