Bonus Lecture Post-Mortem

- Well, the food worked.
- It was a true test of endurance.
- Apparently more than one person wants to hear “the rest of the story”, so we’ll probably organize Bonus Lecture 2
  - Later.
  - Much later.
  - When the thought of spending more than 75 minutes here doesn’t induce nausea.

Old Questions Answered

- Denotational Semantics class question:
  - “What’s up with the continuity requirement?”
- A function $F : S^n \rightarrow S^n$ is **continuous** if for every chain $W \subset S^n$
  - $F(W)$ has a LUB $\sqcup F(W)$
  - and $F(\sqcup W) = \sqcup F(W)$
- See the Ed Lee paper retconned into the lectures page.

“The Real Deal”

Axiomatic Semantics

Soundness of Axiomatic Semantics

- Formal statement of **soundness**:
  - If $\vdash \{ A \} c \{ B \}$ then $\models \{ A \} c \{ B \}$
  - or, equivalently
    - For all $\sigma$, if $\sigma \models A$
      - and $Op : : <c, \sigma> \Downarrow \sigma'$
      - and $Pr : : \vdash \{ A \} c \{ B \}$
    - then $\sigma' \models B$
- "$Op" = "Opsem Derivation"
- "$Pr" = "Axiomatic Proof"

Simultaneous Induction

- Consider two structures $Op$ and $Pr$
  - Assume that $x < y$ iff $x$ is a substructure of $y$
- Define the ordering $(o, p) < (o', p')$ iff
  - $o < o'$ or $o = o'$ and $p < p'$
  - Called lexicographic (dictionary) ordering
- This $<$ is a well founded order and leads to simultaneous induction
- If $o < o'$ then $p$ can actually be larger than $p'$
- It can even be unrelated to $p'$

Soundness of the While Rule

(Indiana Proof and the Slide of Doom)

- Case: last rule used in $Pr : \vdash \{ A \} c \{ B \}$ was the while rule:

  $\vdash \{ A \} while b do c \{ A \land \neg b \}$

- Two possible rules for the root of $Op$ (by inversion)
  - We’ll only do the complicated case:
    - $Op_1 : : \langle b, \sigma \rangle \Downarrow \text{true}$
    - $Op_2 : : \langle c, \sigma \rangle \Downarrow \sigma'$
    - $Op_3 : : \langle \text{while } b \text{ do } c, \sigma \rangle \Downarrow \sigma''$

Assume that $\sigma \models A$

To show that $\sigma' \models A \land \neg b$

- By soundness of booleans and $Op_1$, we get $\sigma \models b$
  - Hence $\sigma \equiv A \land b$
- By IH on $Pr_1$ and $Op_1$, we get $\sigma' \equiv A$
- By IH on $Pr$ and $Op_3$, we get $\sigma'' \equiv A \land \neg b$, q.e.d.
  - This is the tricky bit!
Soundness of the While Rule

• Note that in the last use of IH the derivation Pr did not decrease
• See Winskel, Chapter 6.5, for a soundness proof with denotational semantics

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

General Plan

• OK, so:
  \( \models \{ x < 5 \wedge z = 2 \} y := x + 2 \{ y < 7 \} \)
• Can we prove it?
  \( \vdash \{ x+2 < 7 \} y := x + 2 \{ y < 7 \} \)
• Well, we could easily prove:
  \( \vdash x < 5 \wedge z = 2 \Rightarrow x+2 < 7 \)
• Shouldn’t those two proofs be enough?

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-conditions”)
  b) Verify for one \( A' \in \text{Pre}(c, B) \) that \( A \Rightarrow A' \)
• Assertions can be ordered:
  false \( \Rightarrow \) true
  weak \( \Rightarrow \) strong
  weakest precondition: \( \text{WP}(c, B) \)
• Thus: compute \( \text{WP}(c, B) \) and prove \( A \Rightarrow \text{WP}(c, B) \)

Proof Idea (Cont.)

• 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:
  1. \( \text{wp} \) is a precondition (according to the Hoare rules)
     \( \vdash \{ \text{wp}(c, B) \} c \{ B \} \)
  2. \( \text{wp} \) is (truly) the weakest precondition
     If \( \models [A] c \{ B \} \) then \( \vdash A \Rightarrow \text{wp}(c, B) \)
     \( \vdash A \Rightarrow \text{wp}(c, B) \) \( \vdash [\text{wp}(c, B)] c \{ B \} \)
     \( \vdash [A] c \{ B \} \)
• We also need that whenever \( \models A \) then \( \vdash A \) !

Weakest Preconditions

• Define \( \text{wp}(c, B) \) inductively on \( c \), following the Hoare rules:
  \( \text{wp}(c_1; c_2, B) = \text{wp}(c_1, \text{wp}(c_2, B)) \)
  \( \{ A \} c_1; c_2 [B] \)
  \( \{ \text{wp}(c_1) \} c_2 [B] \)
• \( \text{wp}(x := e, B) = [e/x]B \)
  \( [\text{wp}(c_1, B)] x := E [B] \)
  \( \{ A \} c_1 [B] \)
  \( \{ A \} c_2 [B] \)
  \( \{ E \Rightarrow A_1 \wedge \neg E \Rightarrow A_2 \} \text{if } E \text{ then } c_1 \text{ else } c_2 [B] \)
• \( \text{wp}(\text{if } E \text{ then } c_1 \text{ else } c_2, B) = E \Rightarrow \text{wp}(c_1, B) \wedge \neg E \Rightarrow \text{wp}(c_2, B) \)
Weakest Preconditions for Loops

- 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 \(W = \wp(w, B)\)
- We have that
  \[W = b \Rightarrow \wp(c, W) \land \neg b \Rightarrow B\]
- But this is a recursive equation!
  - We know how to solve these using domain theory
  - But we need a domain for assertions

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?
  \[A \preceq A' \iff \models A' \Rightarrow A\]
- Is this partial order complete?
  - Take a chain \(A_1 \preceq A_2 \preceq \ldots\)
  - Let \(\wedge A_i\) be the infinite conjunction of \(A_i\)
    \[\sigma \models \wedge A_i\] if for all \(i\) we have that \(\sigma \models A_i\)
  - I assert that \(\wedge A_i\) is the least upper bound
- Can \(\wedge A_i\) be expressed in our language of assertions?
  - In many cases: yes (see Winskel), we'll assume yes for now

Weakest Precondition for WHILE

- Use the fixed-point theorem
  \[F(A) = b \Rightarrow \wp(c, A) \land \neg b \Rightarrow B\]
  - (Where did this come from? Two slides back!)
  - I assert that \(F\) is both monotonic and continuous
- The least-fixed point (= the weakest fixed point) is
  \[\wp(w, B) = \wedge F^i(\text{true})\]
- Notice that unlike for denotational semantics of IMP we are not working on a flat domain!

Weakest Preconditions (Cont.)

- Define a family of \(\wp\)'s
  \[\wp_k(\text{while } e \text{ do } c, B) = \text{ weakest precondition on which the loop terminates in } B \text{ if it terminates in } k \text{ or fewer iterations}\]
  \[\wp_0 = \neg E \Rightarrow B\]
  \[\wp_1 = E \Rightarrow \wp(c, \wp_0) \land \neg E \Rightarrow B\]
  \[
  \vdots
  \]
  \[\wp(\text{while } e \text{ do } c, B) = \wedge_{k \geq 0} \wp_k = \text{lub} \{\wp_k | k \geq 0\}\]
- See Necula document 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:
  \[
  \begin{array}{c|c|c}
  \text{false} & \Rightarrow & \text{true} \\
  \hline
  \text{strong} & \text{weakest} & \text{weak} \\
  \hline
  \text{verification condition: } \text{VC}(c, B) \\
  \end{array}
  \]
- Construct a verification condition: \(\text{VC}(c, B)\)
  - Our loops will be annotated with loop invariants!
  - VC is guaranteed to be stronger than WP
  - But still weaker than \(A: A \Rightarrow \text{VC}(c, B) \Rightarrow \wp(c, B)\)

Groundwork

- 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 \(\text{VC}(\text{annotated_command}, \text{post_condition})\) is called \(\text{VCGen}\)
Verification Condition Generation

- Mostly follows the definition of the wp function:
  VC(skip, B) = B
  VC(c₁; c₂, B) = VC(c₁, VC(c₂, B))
  VC(if b then c₁ else c₂, B) = b ⇒ VC(c₁, B) ∧ ¬b ⇒ VC(c₂, B)
  VC(x := e, B) = [e/x] B
  VC(let x = e in c, B) = [e/x] VC(c, B)
  VC(while Inv b do c, B) = Inv ∧ (∀x₁…xₙ. Inv ⇒ (e ⇒ VC(c, Inv) ∧ ¬e ⇒ B))

- Inv is the loop invariant (provided externally)
- x₁, ..., xₙ are all the variables modified in c
- The ∀ is similar to the ∀ in mathematical induction:
P(0) ∧ ∀n ∈ N. P(n) ⇒ P(n+1)

Example VCGen Problem

- Let’s compute the VC of this program with respect to post-condition x ≠ 0
  x = 0;
  y = 2;
  whileₓ,y=2 y > 0 do
  y := y - 1;
  x := x + 1

Example of VC

- By the sequencing rule, first we do the while loop (call it w):
  whileₓ,y=2 y > 0 do
  y := y - 1;
  x := x + 1
  VCGen(w, x ≠ 0) = x+y=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
  VCGen(y:=y-1 ; x:=x+1, x+y=2) = (x+1) + (y-1) = 2
  w Result: x+y=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
  So now we ask an automated theorem prover to prove it.

Example of VC (2)

- VC(w, x ≠ 0) = x+y=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
- VC(x := 0; y := 2 ; w, x ≠ 0) = 0+2=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
- So now we ask an automated theorem prover to prove it.

VCGen for WHILE

- Inv holds on entry
- Inv is preserved in an arbitrary iteration
- B holds when the loop terminates in an arbitrary iteration

Example of VC

- By the sequencing rule, first we do the while loop (call it w):
  whileₓ,y=2 y > 0 do
  y := y - 1;
  x := x + 1
  VCGen(w, x ≠ 0) = x+y=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
  VCGen(y:=y-1 ; x:=x+1, x+y=2) = (x+1) + (y-1) = 2
  w Result: x+y=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
  So now we ask an automated theorem prover to prove it.

Example of VC (2)

- VC(w, x ≠ 0) = x+y=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
- VC(x := 0; y := 2 ; w, x ≠ 0) = 0+2=2 ∧ (∀x,y. x+y=2 ⇒ (y>0 ⇒ (x+1)+(y-2)=2 ∧ y<0 ⇒ x ≠ 0))
- So now we ask an automated theorem prover to prove it.

Thoreau, Thoreau, Thoreau

$ ./Simplify
$ ./Simplify
$ ./Simplify

- Huzzah!
- 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 too weak, like “true”.
  - VC = true ∧ ∀x,y. true ⇒ (y > 0 ⇒ true ∧ y ≤ 0 ⇒ x ≠ 0)
- Let’s use a loop invariant that is false, like “x ≠ 0”.
  - VC = 0 ≠ 0 ∧ ∀x,y. x ≠ 0 ⇒ (y > 0 ⇒ x + 1 ≠ 0 ∧ y ≤ 0 ⇒ x ≠ 0)

Emerson, Emerson, Emerson

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

Counterexample: context:
  (AND
    (EQ x 0)
    (<= y 0)
  )
1: 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 invariant!
- Next: properties and extensions of VCs

VC and Invariants

- Consider the Hoare triple:
  \[\{ x ≤ 0 \} while_{\geq x} x ≤ 5 \rightarrow x := x + 1 \{ x = 6 \}\]
- The VC for this is:
  \[x ≤ 0 \Rightarrow I(x) \land \forall x. I(x) \Rightarrow (x > 5 \Rightarrow x = 6 \land x ≤ 5 \Rightarrow I(x+1))\]
- Requirements on the invariant:
  - Holds on entry \[\forall x. x ≤ 0 ⇒ I(x)\]
  - Preserved by the body \[\forall x. I(x) \land x ≤ 5 ⇒ I(x+1)\]
  - Useful \[\forall x. I(x) \land x > 5 ⇒ x = 6\]
- Check that \(I(x) = x ≤ 6\) 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 un-structured languages (e.g., assembly language)
  - Uses symbolic execution, a technique that has broad applications in program analysis
    - e.g., the Prefix tool (Intrinsa, Microsoft) does this

Forward VC Gen Intuition

- Consider the sequence of assignments
  \[x_1 := e_1; x_2 := e_2\]
- The VC(c, B) = \([e_1/x_1][e_2/x_2]B\) = \([e_1/x_1], e_2[e_1/x_1]/x_2] B\)
- We can compute the substitution in a forward way using symbolic execution (aka symbolic evaluation)
  - Keep a symbolic state that maps variables to expressions
    - Initial: \(\Sigma_0 = \{\}\)
    - After \(x_i := e_i\), \(\Sigma_i = \{ x_i → e_i \}\)
    - After \(x_j := e_j, \Sigma_i = \{ x_j → e_j, x_i → e_i[e_i/x_i] \}\)
  - Note that we have applied \(\Sigma_i\) as a substitution to right-hand side of assignment \(x_j := e_j\)
Simple Assembly Language

- Consider the language of instructions:
  \[ I ::= x := e | f() | \text{if } e \text{ goto } L | \text{goto } L | L: | \text{return } | \text{inv } e \]
- The “\text{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 \( k \)

Symex States

- We set up a symbolic execution state:
  \[ \Sigma : \text{Var} \rightarrow \text{SymbolicExpressions} \]
  \[ \Sigma(x) = \text{the symbolic value of } \Sigma \text{ at } x \]
  \[ \Sigma[x:=e] = \text{a new state in which } x \text{'s value is } e \]
  - We use states as substitutions:
  \[ \Sigma(e) = \text{obtained from } e \text{ by replacing } x \text{ with } \Sigma(x) \]
- Much like the opsem so far ...

Symex Invariants

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

Symex Rules

- Define a VC function as an interpreter:
  \[ \text{VC} : \text{Address} \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion} \]
  \[ \text{VC}(L, \Sigma, \text{Inv}) \]
  \[ \text{if } I_k = \text{return} \]
  \[ \Sigma(\text{Post}_f) \land \forall y_1, ... y_m. \Sigma'(\text{Post}_f) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv'}) \]
  \[ \text{if } I_k = \text{inv } e \]
  \[ \Sigma(e) \land \forall a_1, ... a_m. \Sigma'(e) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv'}) \]
  \[ \text{if } I_k = \text{f()} \]
  \[ \Sigma(\text{Pre}_f) \land \forall \Sigma'(k+1) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv'}) \]
  \[ \text{if } I_k = \text{goto } L \]
  \[ \text{VC}(L, \Sigma, \text{Inv}) \]

Symex Invariants (2a)

Two cases when seeing an invariant instruction:
1. We see the invariant for the first time
   - \( I_k = \text{inv } e \)
   - \( k \in \text{Inv} \) (“not in the set of invariants we’ve seen”)
   - Let \( \{y_1, ..., y_m\} \) = the variables that could be modified on a path from the invariant back to itself
   - Let \( a_1, ..., a_m \) be fresh new symbolic parameters
   \[ \text{VC}(k, \Sigma, \text{Inv}) = \Sigma(e) \land \forall a_1, ..., a_m. \Sigma'(e) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv \cup \{k\}}) \]
   \[ \text{with } \Sigma' = \Sigma[y_1 := a_1, ..., y_m := a_m] \]

Symex Invariants (2b)

2. We see the invariant for the second time
   - \( I_k = \text{inv } e \)
   - \( k \in \text{Inv} \)
   \[ \text{VC}(k, \Sigma, \text{Inv}) = \Sigma(e) \]
   (like a function return)
- Some tools take a more simplistic approach
  - Do not require invariants
  - Iterate through the loop a fixed number of times
  - \( \text{PreFix} \), versions of ESC (DEC/Compaq/HP SRC)
  - Sacrifice completeness for usability
Homework

- Homework 3 Due Today
  - If you’re stuck on 3, note that r* is just like WHILE
- Homework 4 Out Today (Due Thur Feb 16)
- Read Winskel 7.4-7.6 (on VC’s)
- Read Dijkstra article