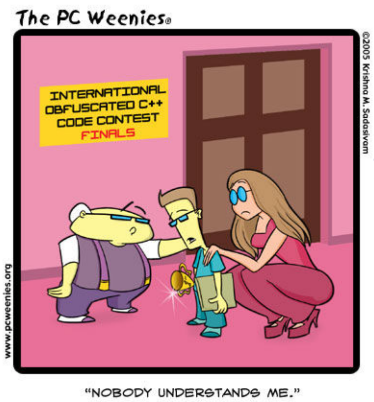
Bonus Lecture Post-Mortem

- 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 $F : S^m \to S^n$ is continuous if for every chain $W \subset S^m$
  - $F(W)$ 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 σ, 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

  $(o, p) \prec (o', p')$ iff

  $o < o'$ or $o = o'$ and $p < p'$
  - Called lexicographic (dictionary) ordering
- This $\prec$ 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:

$$\frac{Pr_1 :: \vdash \{A \wedge b\}\ c\ \{A\}}{\vdash \{A\}\ \text{while } b \text{ do } c\ \{A \wedge \neg b\}}$$

- Two possible rules for the root of Op (by inversion)
  - We'll only do the complicated case:

$$\frac{Op_1 :: <b, \sigma> \Downarrow true \qquad Op_2 :: <c,\sigma> \Downarrow \sigma' \qquad Op_3 :: <\text{while } b \text{ do } c, \sigma'> \Downarrow \sigma''}{<\text{while } b \text{ do } c, \sigma> \Downarrow \sigma''}$$

Assume that $\sigma \models A$
To show that $\sigma'' \models A \wedge \neg b$

- By soundness of booleans and Op_1 we get $\sigma \models b$
  - Hence $\sigma \models A \wedge b$
- By IH on Pr_1 and Op_2 we get $\sigma' \models A$
- By IH on Pr and Op_3 we get $\sigma'' \models A \wedge \neg b$, 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

#7

Completeness of Axiomatic Semantics

- If ⊨ {A} c {B} can we always derive ⊢ {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 ⊨ A we also have ⊢ A)
  - this is called relative completeness

#8

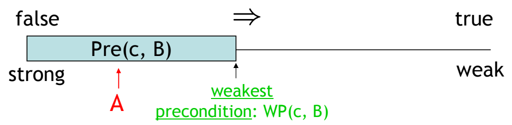
General Plan

- OK, so:
  ⊨ { x < 5 ∧ z = 2 } y := x + 2 { y < 7 }
- Can we prove it?
  ?⊢? { x < 5 ∧ z = 2 } y := x + 2 { y < 7 }
- Well, we *could* easily prove:
  ⊢ { x+2 < 7 } y := x + 2 { y < 7 }
- And we know …
  ⊢ x < 5 ∧ z = 2 ⇒ x+2 < 7
- Shouldn't those two proofs be enough?

#9

Proof Idea

- Dijkstra's idea: To verify that { A } c { B }
  a) Find out all predicates A' such that ⊨ { A' } c { B }
    - call this set Pre(c, B) (Pre = "pre-conditions")
  b) Verify for one A' ∈ Pre(c, B) that A ⇒ A'
- Assertions can be ordered:



- Thus: compute WP(c, B) and prove A ⇒ WP(c, B)

#10

Proof Idea (Cont.)

- Completeness of axiomatic semantics:
  If ⊨ { A } c { B } then ⊢ { A } c { B }
- Assuming that we can compute wp(c, B) with the following properties:
  1. wp is a precondition (according to the Hoare rules)
     ⊢ { wp(c, B) } c { B }
  2. wp is *(truly)* the weakest precondition
     If ⊨ { A } c { B } then ⊨ A ⇒ wp(c, B)

$$\frac{\vdash A \Rightarrow wp(c, B) \qquad \vdash \{wp(c, B)\}\ c\ \{B\}}{\vdash \{A\}\ c\ \{B\}}$$

- We also need that whenever ⊨ A then ⊢ A !

#11

Weakest Preconditions

- Define wp(c, B) inductively on c, following the Hoare rules:
- $wp(c_1;\ c_2,\ B) = wp(c_1,\ wp(c_2,\ B))$

$$\frac{\{A\}\ c_1\ \{C\} \qquad \{C\}\ c_2\ \{B\}}{\{ A \}\ c_1;\ c_2\ \{B\}}$$

- wp(x := e, B) = [e/x]B

$$\frac{}{\{ [e/x]B \}\ x := E\ \{B\}}$$

$$\frac{\{A_1\}\ c_1\ \{B\} \qquad \{A_2\}\ c_2\ \{B\}}{\{ E \Rightarrow A_1 \wedge \neg E \Rightarrow A_2\}\ \text{if } E \text{ then } c_1 \text{ else } c_2\ \{B\}}$$

- $wp(\text{if } E \text{ then } c_1 \text{ else } c_2,\ B) = E \Rightarrow wp(c_1,\ B) \wedge \neg E \Rightarrow wp(c_2,\ B)$

#12

Weakest Preconditions for Loops

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

#13

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 ?
  $A \sqsubseteq A'$ iff $\models A' \Rightarrow A$
- Is this partial order complete?
  - Take a chain $A_1 \sqsubseteq A_2 \sqsubseteq \ldots$
  - Let $\wedge A_i$ be the infinite conjunction of A_i
    $\sigma \models \wedge A_i$ iff 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

#14

Weakest Precondition for WHILE

- Use the fixed-point theorem
  $F(A) = b \Rightarrow wp(c, A) \wedge \neg b \Rightarrow B$
  - (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!

#15

Weakest Preconditions (Cont.)

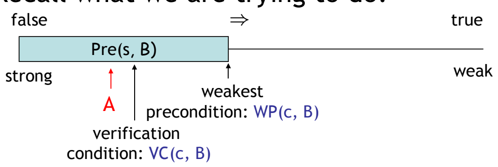
- Define a family of wp's
  - wp_k(while e do c, B) = weakest precondition on which the loop terminates in B if it terminates in k or fewer iterations

  $wp_0 = \neg E \Rightarrow B$
  $wp_1 = E \Rightarrow wp(c, wp_0) \wedge \neg E \Rightarrow B$
  …
- wp(while e do c, B) = $\wedge_{k \geq 0} wp_k = \text{lub } \{wp_k \mid k \geq 0\}$
- 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 ?

#16

Not Quite Weakest Preconditions

- Recall what we are trying to do:



- Construct a verification condition: VC(c, B)
  - Our loops will be annotated with loop invariants!
  - VC is guaranteed to be stronger than WP
  - But still weaker than A: $A \Rightarrow VC(c, B) \Rightarrow WP(c, B)$

#17

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:
  while_{Inv} b do c
  - Invariant formula Inv must hold every time before b is evaluated
- A process for computing VC(annotated_command, post_condition) is called VCGen

#18

Verification Condition Generation

- Mostly follows the definition of the wp function:

$VC(skip, B) = B$

$VC(c_1; c_2, B) = VC(c_1, VC(c_2, B))$

$VC(\text{if } b \text{ then } c_1 \text{ else } c_2, B) =$

$\quad b \Rightarrow VC(c_1, B) \wedge \neg b \Rightarrow VC(c_2, B)$

$VC(x := e, B) = [e/x]\ B$

$VC(\text{let } x = e \text{ in } c, B) = [e/x]\ VC(c, B)$

$VC(\text{while}_{Inv}\ b \text{ do } c, B) = ?$

VCGen for WHILE

$VC(\text{while}_{Inv}\ e \text{ do } c, B) =$

$$\underbrace{Inv}_{\text{Inv holds on entry}} \wedge (\forall x_1 \ldots x_n.\ \underbrace{Inv \Rightarrow (e \Rightarrow VC(c, Inv))}_{\text{Inv is preserved in an arbitrary iteration}} \wedge \underbrace{\neg e \Rightarrow B}_{\text{B holds when the loop terminates in an arbitrary iteration}})$$

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

$P(0) \wedge \forall n \in \mathbb{N}.\ P(n) \Rightarrow P(n+1)$

Example VCGen Problem

- Let's compute the VC of this program with respect to post-condition $x \neq 0$

```
x = 0;
y = 2;
while_{x+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):

$\text{while}_{x+y=2}\ y > 0 \text{ do}$
$\quad y := y - 1;$
$\quad x := x + 1$

- $VCGen(w, x \neq 0) = x+y=2 \wedge$
  $\forall x,y.\ x+y=2 \Rightarrow (y>0 \Rightarrow VC(c, x+y=2) \wedge y \leq 0 \Rightarrow x \neq 0)$
- $VCGen(y:=y-1\ ;\ x:=x+1,\ x+y=2) =$
  $(x+1) + (y-1) = 2$
- w Result: $x+y=2 \wedge$
  $\forall x,y.\ x+y=2 \Rightarrow (y>0 \Rightarrow (x+1)+(y-1)=2 \wedge y \leq 0 \Rightarrow x \neq 0)$

Example of VC (2)

- $VC(w, x \neq 0) = x+y=2 \wedge$
  $\forall x,y.\ x+y=2 \Rightarrow$
  $(y>0 \Rightarrow (x+1)+(y-1)=2 \wedge y \leq 0 \Rightarrow x \neq 0)$
- $VC(x := 0;\ y := 2\ ;\ w,\ x \neq 0) = 0+2=2 \wedge$
  $\forall x,y.\ x+y=2 \Rightarrow$
  $(y>0 \Rightarrow (x+1)+(y-1)=2 \wedge y \leq 0 \Rightarrow x \neq 0)$
- So now we ask an automated theorem prover to prove it.

Thoreau, Thoreau, Thoreau

```
$ ./Simplify
> (AND (EQ (+ 0 2) 2)
  (FORALL ( x y ) (IMPLIES (EQ (+ x y) 2)
     (AND (IMPLIES (> y 0)
                      (EQ (+ (+ x 1)(- y 1)) 2))
          (IMPLIES (<= y 0) (NEQ x 0))))))
1: Valid.
```

- 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 $\wedge$ $\forall$x,y. true $\Rightarrow$ (y>0 $\Rightarrow$ true $\wedge$ y$\leq$0 $\Rightarrow$ x $\neq$ 0)
- Let's use a loop invariant that is false, like "x $\neq$ 0".
- VC = 0 $\neq$ 0 $\wedge$ $\forall$x,y. x $\neq$ 0 $\Rightarrow$ (y>0 $\Rightarrow$ x+1 $\neq$ 0 $\wedge$ y$\leq$0 $\Rightarrow$ x $\neq$ 0)

#25

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.

#26

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

#27

VC and Invariants

- Consider the Hoare triple:
  {x $\leq$ 0} $\text{while}_{I(x)}$ x $\leq$ 5 do x := x + 1 {x = 6}
- The VC for this is:
  x $\leq$ 0 $\Rightarrow$ I(x) $\wedge$ $\forall$x. (I(x) $\Rightarrow$ (x > 5 $\Rightarrow$ x = 6 $\wedge$ 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) $\wedge$ x $\leq$ 5 $\Rightarrow$ I(x+1)
  - Useful $\forall$x. I(x) $\wedge$ x > 5 $\Rightarrow$ x = 6
- Check that I(x) = x $\leq$ 6 satisfies all constraints

#28

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

#29

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
  - Initially, $\Sigma_0 = \{\ \}$
  - After $x_1 := e_1$, $\Sigma_1 = \{ x_1 \to e_1 \}$
  - After $x_2 := e_2$, $\Sigma_2 = \{x_1 \to e_1,\ x_2 \to e_2[e_1/x_1] \}$
  - Note that we have applied Σ_1 as a substitution to right-hand side of assignment $x_2 := e_2$

#30

Simple Assembly Language

- Consider the language of instructions:
  I ::= x := e | f() | if e goto L | goto L | L: | return | inv e
- The "inv e" instruction is an annotation
  - Says that boolean expression e holds at that point
- Each function f() comes with Pre_f and $Post_f$ annotations (pre- and post-conditions)
- New Notation (yay!): I_k is the instruction at address k

#31

Symex States

- We set up a symbolic execution state:

Σ : Var $\to$ SymbolicExpressions

$\Sigma(x)$ = the symbolic value of x in state Σ

$\Sigma[x:=e]$ = a new state in which x's value is e

- We use states as substitutions:

$\Sigma(e)$ - obtained from e by replacing x with $\Sigma(x)$

- Much like the opsem so far …

#32

Symex Invariants

- The symbolic executor tracks invariants passed
- A new part of symex state: Inv $\subseteq$ {1…n}
- If k $\in$ Inv then I_k is an invariant instruction 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

#33

Symex Rules

- Define a VC function as an interpreter:

VC : Address $\times$ SymbolicState $\times$ InvariantState $\to$ Assertion

VC(k, Σ, Inv) =

VC(L, Σ, Inv)	if I_k = goto L
e $\Rightarrow$ VC(L, Σ, Inv) $\wedge$ $\neg$ e $\Rightarrow$ VC(k+1, Σ, Inv)	if I_k = if e goto L
VC(k+1, $\Sigma[x:=\Sigma(e)]$, Inv)	if I_k = x := e
$\Sigma(Post_{current\text{-}function})$	if I_k = return
$\Sigma(Pre_f)$ $\wedge$ $\forall a_1..a_m.\Sigma'(Post_f) \Rightarrow$ VC(k+1, Σ', Inv) (where y_1, …, y_m are modified by f) and a_1, …, a_m are fresh parameters and $\Sigma' = \Sigma[y_1 := a_1, …, y_m := a_m]$	if I_k = f()

#34

Symex Invariants (2a)

Two cases when seeing an invariant instruction:

1. We see the invariant for the first time
   - I_k = inv e
   - k $\notin$ 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

VC(k, Σ, Inv) =

$\Sigma(e) \wedge \forall a_1…a_m.\ \Sigma'(e) \Rightarrow VC(k+1, \Sigma', Inv \cup \{k\})$

with $\Sigma' = \Sigma[y_1 := a_1, …, y_m := a_m]$

(like a function call)

#35

Symex Invariants (2b)

2. We see the invariant for the second time
   - I_k = inv E
   - k $\in$ Inv

   VC(k, Σ, 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
  - PREfix, versions of ESC (DEC/Compaq/HP SRC)
  - Sacrifice completeness for usability

#36

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

#37