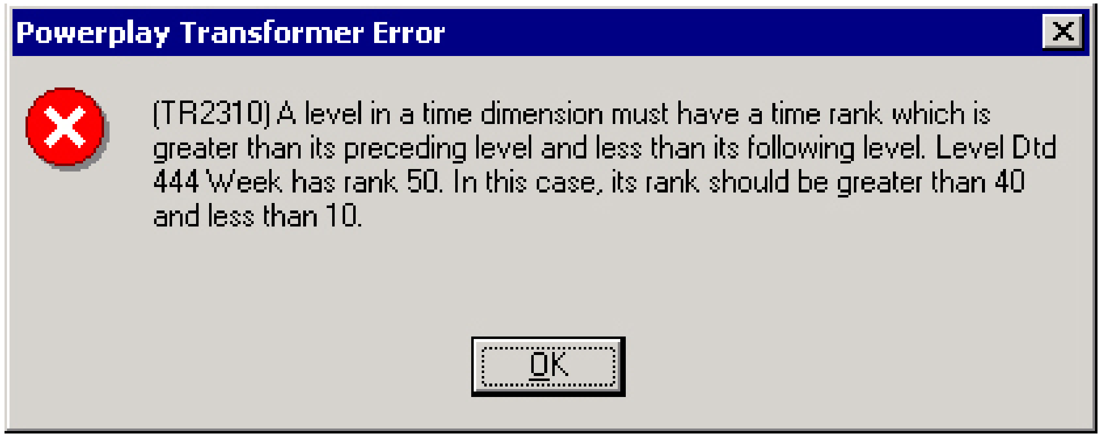
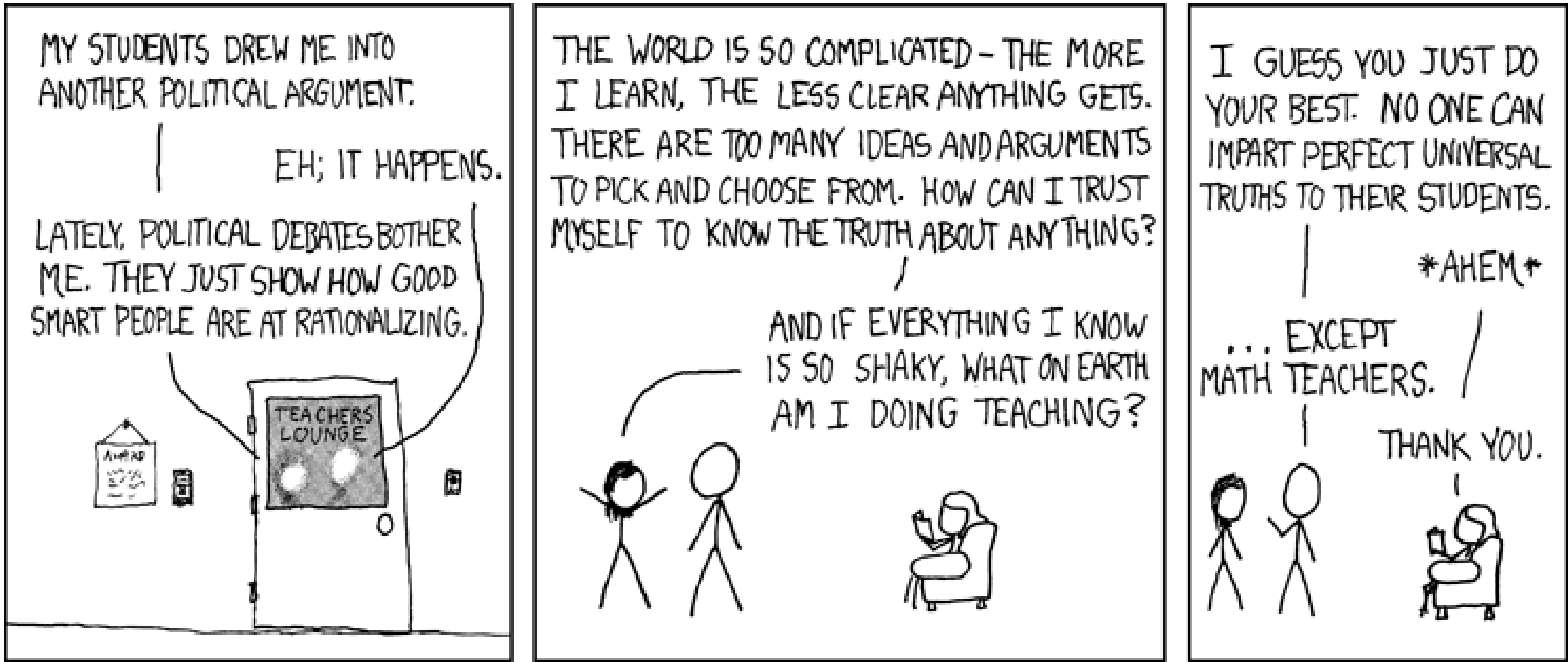

Powerplay Transformer Error

[TR2310] A level in a time dimension must have a time rank which is greater than its preceding level and less than its following level. Level Dtd 444 Week has rank 50. In this case, its rank should be greater than 40 and less than 10.

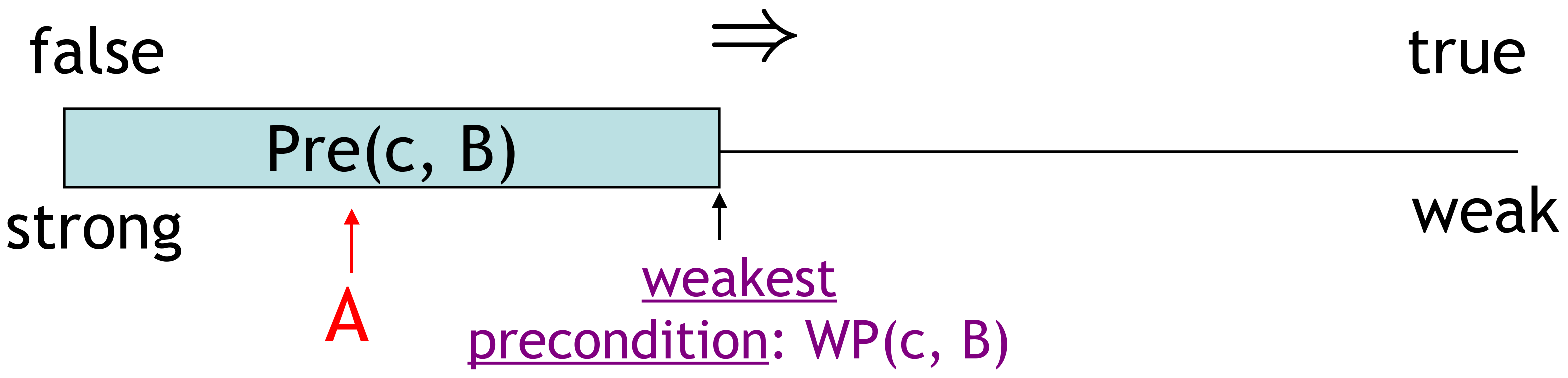
OK

Axiomatic Semantics: Preconditions


MY STUDENTS DREW ME INTO ANOTHER POLITICAL ARGUMENT.
EH; IT HAPPENS.
LATELY, POLITICAL DEBATES BOTHER ME. THEY JUST SHOW HOW GOOD SMART PEOPLE ARE AT RATIONALIZING.
TEACHERS LOUNGE
THE WORLD IS SO COMPLICATED – THE MORE I LEARN, THE LESS CLEAR ANYTHING GETS. THERE ARE TOO MANY IDEAS AND ARGUMENTS TO PICK AND CHOOSE FROM. HOW CAN I TRUST MYSELF TO KNOW THE TRUTH ABOUT ANYTHING?
AND IF EVERYTHING I KNOW IS SO SHAKY, WHAT ON EARTH AM I DOING TEACHING?
I GUESS YOU JUST DO YOUR BEST. NO ONE CAN IMPART PERFECT UNIVERSAL TRUTHS TO THEIR STUDENTS.
AHEM
... EXCEPT MATH TEACHERS.
THANK YOU.

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 Pre(c, B) (Pre = "<u>pre-conditions</u>")
  - b) Verify for one A' $\in$ Pre(c, B) that A $\Rightarrow$ A'
- Assertions can be ordered:



- Thus: compute WP(c, B) and <u>prove</u> A $\Rightarrow$ 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 wp(c, B) with the following properties:
  - wp is a precondition (according to the Hoare rules)

    $\vdash \{ wp(c, B) \}\ c\ \{ B \}$
  - wp is *(truly)* the weakest precondition

    If $\models \{ A \}\ c\ \{ B \}$ then $\models A \Rightarrow wp(c, B)$

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

- We also need that whenever $\models A$ then $\vdash A$!

Weakest Preconditions

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

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

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

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

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

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

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

$$W = b \Rightarrow wp(c, W) \quad \wedge \quad \neg b \Rightarrow B$$

- 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 ?

  $A \sqsubseteq A'$ 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

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

$$wp(w, B) = \wedge F^i(true)$$

- 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(while e do c, B)** = weakest precondition on which the loop terminates in B <u>if</u> it terminates in k or fewer iterations

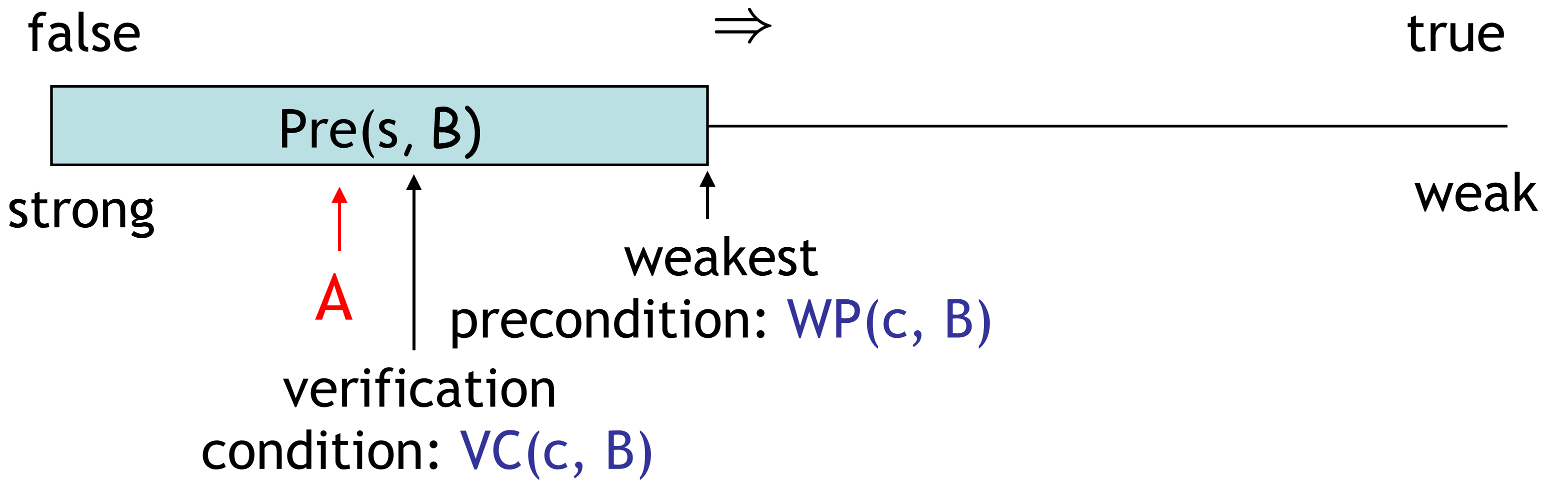
  $wp_0 = \neg E \Rightarrow B$

  $wp_1 = E \Rightarrow wp(c, wp_0) \wedge \neg E \Rightarrow B$

  …
- wp(while e do c, B) = $\bigwedge_{k \geq 0} wp_k = \text{lub } \{wp_k \mid k \geq 0\}$
- 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?

Not Quite Weakest Preconditions

- Recall what we are trying to do:


false
$\Rightarrow$
true
Pre(s, B)
strong
weak
A
weakest precondition: WP(c, B)
verification condition: VC(c, B)

- 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: $A \Rightarrow VC(c, B) \Rightarrow WP(c, B)$

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 Inv must hold every time before b is evaluated
- A process for computing VC(annotated_command, post_condition) is called VCGen

Verification Condition Generation

- Mostly follows the definition of the wp function:

$\mathrm{VC}(\text{skip}, B) = B$

$\mathrm{VC}(c_1; c_2, B) = \mathrm{VC}(c_1, \mathrm{VC}(c_2, B))$

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

$b \Rightarrow \mathrm{VC}(c_1, B) \wedge \neg b \Rightarrow \mathrm{VC}(c_2, B)$

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

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

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

VCGen for WHILE

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

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

- 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
```

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

Example of VC

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

```
    while_{x+y=2} y > 0 do
        y := y - 1;
        x := x + 1
```

Preserve loop invariant

Ensure post on exit

- VCGen(w, $x \neq 0$) = $x+y=2 \wedge$

  $\forall x,y.\ x+y=2 \Rightarrow (y>0 \Rightarrow \mathrm{VC}(c,\ x+y=2) \ \wedge\ y \leq 0 \Rightarrow x \neq 0)$

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

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

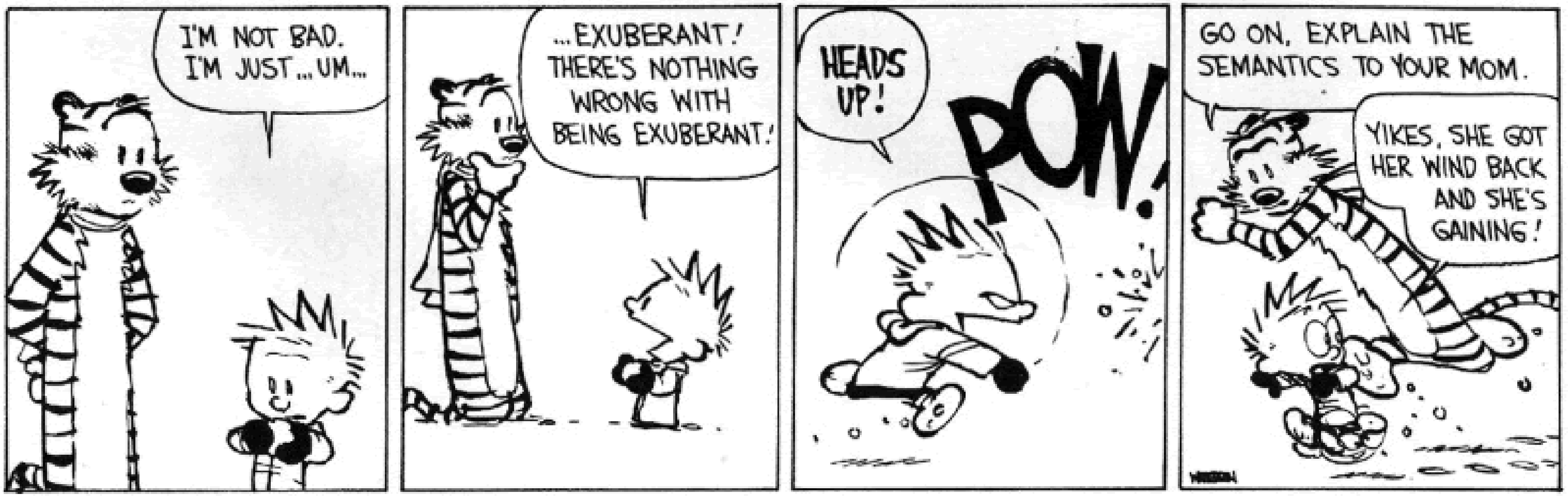
$$\models \{\ VC(c,B)\ \}\ c\ \{\ B\ \}$$

- Or equivalently that

$$\models VC(c, B) \Rightarrow wp(c, B)$$

- Proof is by induction on the structure of c
  - Try it!
- Soundness holds for any choice of invariant!
- Next: properties and extensions of VCs

Axiomatic Semantics III --- The Verification Crusade


I'M NOT BAD. I'M JUST... UM...
...EXUBERANT! THERE'S NOTHING WRONG WITH BEING EXUBERANT!
HEADS UP!
POW!
GO ON, EXPLAIN THE SEMANTICS TO YOUR MOM.
YIKES, SHE GOT HER WIND BACK AND SHE'S GAINING!

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
  - Forward VC (Symbolic Exec) works for assembly
  - Here we are today …

Today's Cunning Plan

- Symbolic Execution & 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 "Dictator For Life"

VC and Invariants

- Consider the Hoare triple:

  $\{x \leq 0\}\ \text{while}_{I(x)}\ x \leq 5\ \text{do}\ x := x + 1\ \{x = 6\}$

- 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 $\quad \forall x.\ x \leq 0 \Rightarrow I(x)$
  - Preserved by the body $\quad \forall x.\ I(x) \wedge x \leq 5 \Rightarrow I(x+1)$
  - Useful $\quad \forall x.\ I(x) \wedge x > 5 \Rightarrow x = 6$
- 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 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
  - 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 $x_2 := e_2$

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

Symex States

- We set up a symbolic execution state:

Σ : Var $\rightarrow$ 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 …

Symex Invariants

- The symbolic executor tracks invariants passed
- A new part of symex state: $\mathrm{Inv} \subseteq \{1\ldots n\}$
- If $k \in \mathrm{Inv}$ 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

Symex Rules

- Define a VC function as an interpreter:

$\text{VC} : \text{Address} \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion}$

$\text{VC}(k, \Sigma, \text{Inv}) =$

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

Symex Invariants (2a)

Two cases when seeing an invariant instruction:

2. 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 $\{y_1, \ldots, y_m\}$ = the variables that could be modified on a path from the invariant back to itself
   - Let $a_1, \ldots, a_m$ be fresh new symbolic parameters

VC(k, Σ, Inv) =

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

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

(like a function call)

Symex Invariants (2b)

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

   $VC(k, \Sigma, Inv) = \Sigma(e)$

   (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

Symex Summary

- Let $x_1, \ldots, x_n$ be all the variables and $a_1, \ldots, a_n$ fresh parameters
- Let Σ_0 be the state $[x_1 := a_1, \ldots, x_n := a_n]$
- Let $\emptyset$ be the empty Inv set

• For all functions f in your program, prove:

$$\forall a_1 \ldots a_n.\ \Sigma_0(\mathrm{Pre}_f) \Rightarrow \mathrm{VC}(f_{entry}, \Sigma_0, \emptyset)$$

• If you start the program by invoking any f in a state that satisfies Pre_f, then the program will execute such that
  - At all "inv e" the e holds, and
  - If the function returns then Post_f holds

• Can be proved w.r.t. a real interpreter (operational semantics)

• Or via a proof technique called co-induction (or, assume-guarantee)

Forward VCGen Example

- Consider the program

Precondition: $x \leq 0$

```
Loop: inv x ≤ 6
        if x > 5 goto End
        x := x + 1
        goto Loop
End:  return
```

Postconditon: $x = 6$

Forward VCGen Example (2)

$$
\begin{array}{l}
\forall x. \\
\quad x \leq 0 \Rightarrow \\
\quad\quad x \leq 6 \wedge \\
\quad\quad\quad \forall x'. \\
\quad\quad\quad\quad (x' \leq 6 \Rightarrow \\
\quad\quad\quad\quad\quad x' > 5 \Rightarrow x' = 6 \\
\quad\quad\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad x' \leq 5 \Rightarrow x' + 1 \leq 6\ )
\end{array}
$$

- VC contains both proof obligations and assumptions about the control flow

VCs Can Be Large

- Consider the sequence of conditionals

  (if x < 0 then x := - x); (if x ≤ 3 then x += 3)

  - With the postcondition $P(x)$
- The VC is

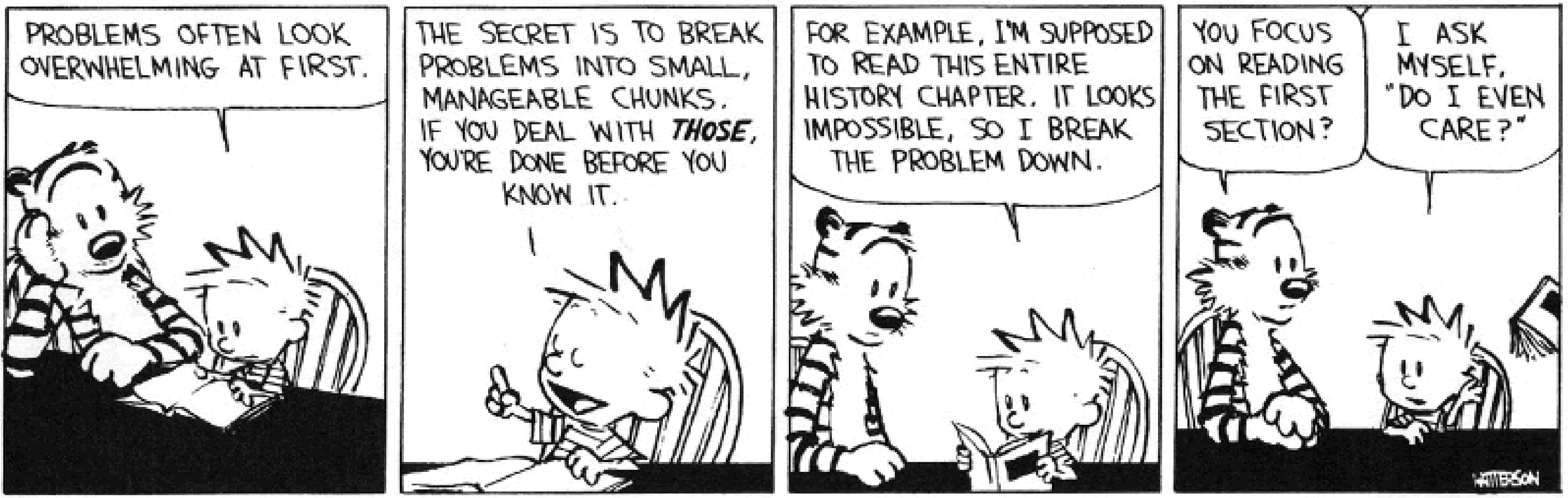
$$\begin{array}{lll} x < 0 \wedge -x \le 3 & \Rightarrow P(-x + 3) & \wedge \\ x < 0 \wedge -x > 3 & \Rightarrow P(-x) & \wedge \\ x \ge 0 \wedge x \le 3 & \Rightarrow P(x + 3) & \wedge \\ x \ge 0 \wedge x > 3 & \Rightarrow P(x) & \end{array}$$

- There is one conjunct for each path

  ⇒ exponential number of paths!
  - Conjuncts for infeasible paths have un-satisfiable guards!
- Try with $P(x) = x \ge 3$

VCs Can Be Exponential

- VCs are exponential in the size of the source because they attempt relative completeness:
  - Perhaps the correctness of the program must be argued independently for each path
- Unlikely that the programmer wrote a program by considering an exponential number of cases
  - But possible. Any examples? Any solutions?


PROBLEMS OFTEN LOOK OVERWHELMING AT FIRST.
THE SECRET IS TO BREAK PROBLEMS INTO SMALL, MANAGEABLE CHUNKS. IF YOU DEAL WITH THOSE, YOU'RE DONE BEFORE YOU KNOW IT.
FOR EXAMPLE, I'M SUPPOSED TO READ THIS ENTIRE HISTORY CHAPTER. IT LOOKS IMPOSSIBLE, SO I BREAK THE PROBLEM DOWN.
YOU FOCUS ON READING THE FIRST SECTION?
I ASK MYSELF, "DO I EVEN CARE?"
WATTERSON

VCs Can Be Exponential

- VCs are exponential in the size of the source because they attempt relative completeness:
  - Perhaps the correctness of the program must be argued independently for each path
- Standard Solutions:
  - Allow invariants even in straight-line code
  - And thus do not consider all paths independently!

Invariants in Straight-Line Code

- Purpose: modularize the verification task
- Add the command "after c establish Inv"
  - Same semantics as c (Inv is only for VC purposes)

$$\text{VC(after c establish Inv, P)} =_{\text{def}}$$

$$\text{VC}(c,\text{Inv}) \wedge \forall x_i.\ \text{Inv} \Rightarrow P$$

  - where x_i are the ModifiedVars(c)
- Use when c contains many paths

```
after if x < 0 then x := - x  establish x ≥ 0;
if x ≤ 3 then x += 3 { P(x) }
```

- VC is now:

$$(x < 0 \Rightarrow -x \geq 0) \wedge \ (x \geq 0 \Rightarrow x \geq 0) \wedge$$
$$\forall x.\ x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3) \wedge \ x > 3 \Rightarrow P(x))$$

Dropping Paths

- In absence of annotations, we can drop some paths
- VC(if E then c_1 else c_2, P) = choose one of
  - $E \Rightarrow VC(c_1, P) \wedge \neg E \Rightarrow VC(c_2, P)$ (drop no paths)
  - $E \Rightarrow VC(c_1, P)$ (drops "else" path!)

    $\neg E \Rightarrow VC(c_2, P)$ (drops "then" path!)
- **We sacrifice soundness!** (we are now unsound)
  - No more guarantees
  - Possibly still a good debugging aid
- Remarks:
  - A recent trend is to sacrifice soundness to increase usability (e.g., Metal, ESP, even ESC)
  - The PREfix tool considers only 50 non-cyclic paths through a function (almost at random)

VCGen for Exceptions

- We extend the source language with exceptions without arguments (cf. HW2):
  - throw — throws an exception
  - try c_1 catch c_2 — executes c_2 if c_1 throws
- Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?

VCGen for Exceptions

- We extend the source language with exceptions without arguments (cf. HW2):
  - throw   throws an exception
  - try c_1 catch c_2 executes c_2 if c_1 throws
- Problem:
  - We have non-local transfer of control
  - What is VC(throw, P) ?
- Standard Solution: use 2 postconditions
  - One for normal termination
  - One for exceptional termination

VCGen for Exceptions (2)

- VC(c, P, Q) is a precondition that makes c either not terminate, or terminate normally with P or throw an exception with Q
- Rules

  $VC(skip, P, Q) = P$

  $VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q)$

  $VC(throw, P, Q) = Q$

  $VC(try\ c_1\ catch\ c_2, P, Q) = VC(c_1, P, VC(c_2, P, Q))$

  $VC(try\ c_1\ finally\ c_2, P, Q) = ?$

VCGen Finally

- Given these:

  $VC(c_1; c_2, P, Q) = VC(c_1, VC(c_2, P, Q), Q)$

  $VC(\text{try } c_1 \text{ catch } c_2, P, Q) = VC(c_1, P, VC(c_2, P, Q))$

- Finally is somewhat like “if”:

  $VC(\text{try } c_1 \text{ finally } c_2, P, Q) =$

  $\quad VC(c_1, VC(c_2, P, Q), \text{true}) \qquad \land$

  $\quad VC(c_1, \text{true}, VC(c_2, Q, Q))$

- Which reduces to:

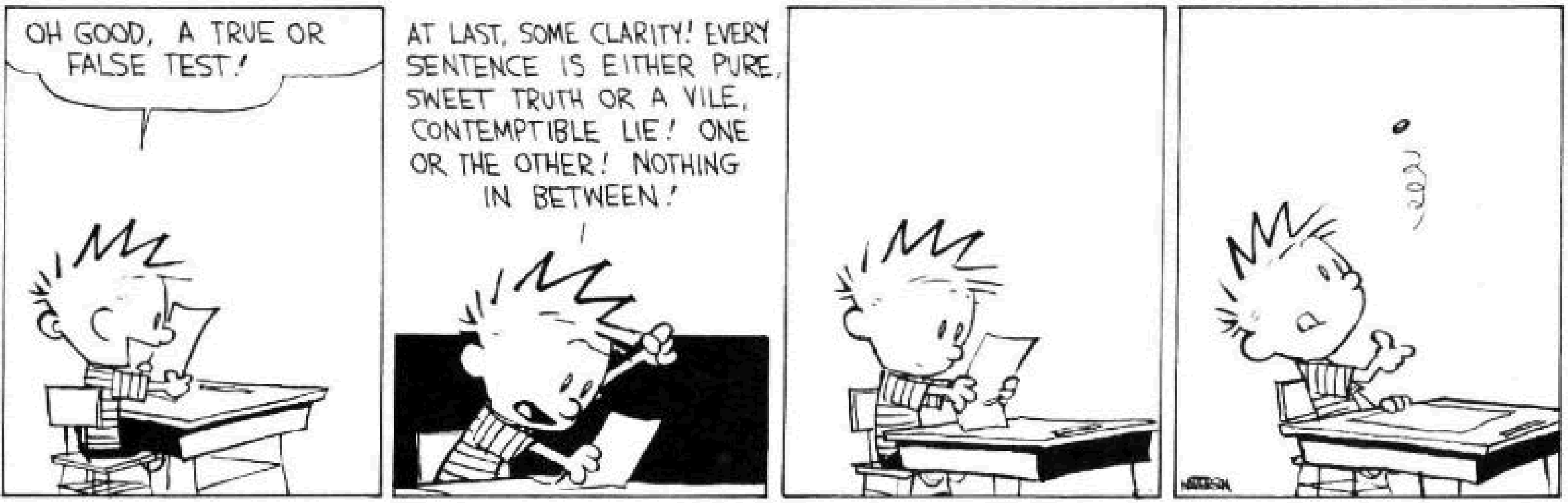
$$VC(c_1, VC(c_2, P, Q), VC(c_2, Q, Q))$$

Hoare Rules and the Heap

- When is the following Hoare triple valid?

  { A } *x := 5 { *x + *y = 10 }

- A *should be* "*y = 5 or x = y"
- The Hoare rule for assignment would give us:
  - [5/*x](*x + *y = 10) = 5 + *y = 10 =
  - *y = 5 (we lost one case)
- Why didn't this work?


OH GOOD, A TRUE OR FALSE TEST!
AT LAST, SOME CLARITY! EVERY SENTENCE IS EITHER PURE, SWEET TRUTH OR A VILE, CONTEMPTIBLE LIE! ONE OR THE OTHER! NOTHING IN BETWEEN!

Handling The Heap

- We do not yet have a way to talk about memory (the heap, pointers) in assertions
- Model the state of memory as a symbolic mapping from addresses to values:
  - If A denotes an address and M is a memory state then:
  - sel(M,A) denotes the contents of the memory cell
  - upd(M,A,V) denotes a new memory state obtained from M by writing V at address A

More on Memory

- We allow variables to range over memory states
  - We can quantify over all possible memory states
- Use the special pseudo-variable μ(mu) in assertions to refer to the current memory
- Example:

$$\forall i.\ i \geq 0 \wedge i < 5 \Rightarrow sel(\mu, A + i) > 0$$

says that entries $0..4$ in array A are positive

Hoare Rules: Side-Effects

- To model writes we use memory expressions
  - A memory write changes the value of memory

$$\frac{}{\{ B[upd(\mu, A, E)/\mu] \}\ {*A} := E\ \{B\}}$$

- Important technique: treat memory as a whole
- And reason later about memory expressions with inference rules such as (McCarthy Axioms, ~'67):

$$sel(upd(M, A_1, V), A_2) = \begin{cases} V & \text{if } A_1 = A_2 \\ sel(M, A_2) & \text{if } A_1 \neq A_2 \end{cases}$$

Memory Aliasing

- Consider again: { A } *x := 5 { *x + *y = 10 }
- We obtain:

  A = [upd(μ, x, 5)/μ] (*x + *y = 10)

  = [upd(μ, x, 5)/μ] (sel(μ, x) + sel(μ, y) = 10)

(1) = sel(upd(μ, x, 5), x) + sel(upd(μ, x, 5), y) = 10

  = 5 + sel(upd(μ, x, 5), y) = 10

  = if x = y then 5 + 5 = 10 else 5 + sel(μ, y) = 10

(2) = x = y or *y = 5

- Up to (1) is theorem generation
- From (1) to (2) is theorem proving

Alternative Handling for Memory

- Reasoning about aliasing can be expensive
  - It is NP-hard (and/or undecideable)
- Sometimes completeness is sacrificed with the following (approximate) rule:

$$\mathrm{sel}(\mathrm{upd}(M, A_1, V), A_2) = \begin{cases} V & \text{if } A_1 = \text{(obviously) } A_2 \\ \mathrm{sel}(M, A_2) & \text{if } A_1 \neq \text{(obviously) } A_2 \\ P & \text{otherwise (p is a fresh new parameter)} \end{cases}$$

- The meaning of "obviously" varies:
  - The addresses of two distinct globals are ≠
  - The address of a global and one of a local are ≠
- PREfix and GCC use such schemes

VCGen Overarching Example

- Consider the program
  - Precondition: *$B : bool \wedge A : array(bool, L)$*

```
1: I := 0
   R := B
3: inv $I \geq 0 \wedge R : bool$
   if I ≥ L goto 9
   assert saferd(A + I)
   T := *(A + I)
   I := I + 1
   R := T
   goto 3
9: return R
```

  - Postcondition: *$R : bool$*

VCGen Overarching Example

$$
\begin{array}{l}
\forall A.\ \forall B.\ \forall L.\ \forall \mu \\
\quad B : \text{bool} \wedge A : \text{array}(\text{bool}, L) \Rightarrow \\
\quad\quad 0 \geq 0 \wedge B : \text{bool} \wedge \\
\quad\quad\quad \forall I.\ \forall R. \\
\quad\quad\quad\quad I \geq 0 \wedge R : \text{bool} \Rightarrow \\
\quad\quad\quad\quad\quad I \geq L \Rightarrow R : \text{bool} \\
\quad\quad\quad\quad\quad\quad \wedge \\
\quad\quad\quad\quad\quad I < L \Rightarrow \text{saferd}(A + I) \ \wedge \\
\quad\quad\quad\quad\quad\quad I + 1 \geq 0 \wedge \\
\quad\quad\quad\quad\quad\quad \text{sel}(\mu, A + I) : \text{bool}
\end{array}
$$

- VC contains both proof obligations and assumptions about the control flow

Mutable Records - Two Models

- Let r : RECORD { f1 : T1; f2 : T2 } END
- For us, records are reference types
- Method 1: one "memory" for each record
  - One index constant for each field
  - r.f1 is sel(r,f1) and r.f1 := E is r := upd(r,f1,E)
- Method 2: one "memory" for each field
  - The record address is the index
  - r.f1 is sel(f1,r) and r.f1 := E is f1 := upd(f1,r,E)
- Only works in strongly-typed languages like Java
  - Fails in C where &r.f2 = &r + sizeof(T1)

VC as a "Semantic Checksum"

- Weakest preconditions are an expression of the program's semantics:
  - Two equivalent programs have logically equivalent WPs
  - No matter how different their syntax is!

- VC are almost as powerful

VC as a "Semantic Checksum" (2)

- Consider the "assembly language" program to the right

```
x := 4
x := x == 5
   assert x : bool
x := not x
   assert x
```

- High-level type checking is not appropriate here
- The VC is: 4 == 5 : bool $\wedge$ not (4 == 5)
- No confusion from reuse of x with different types

Invariance of VC Across Optimizations

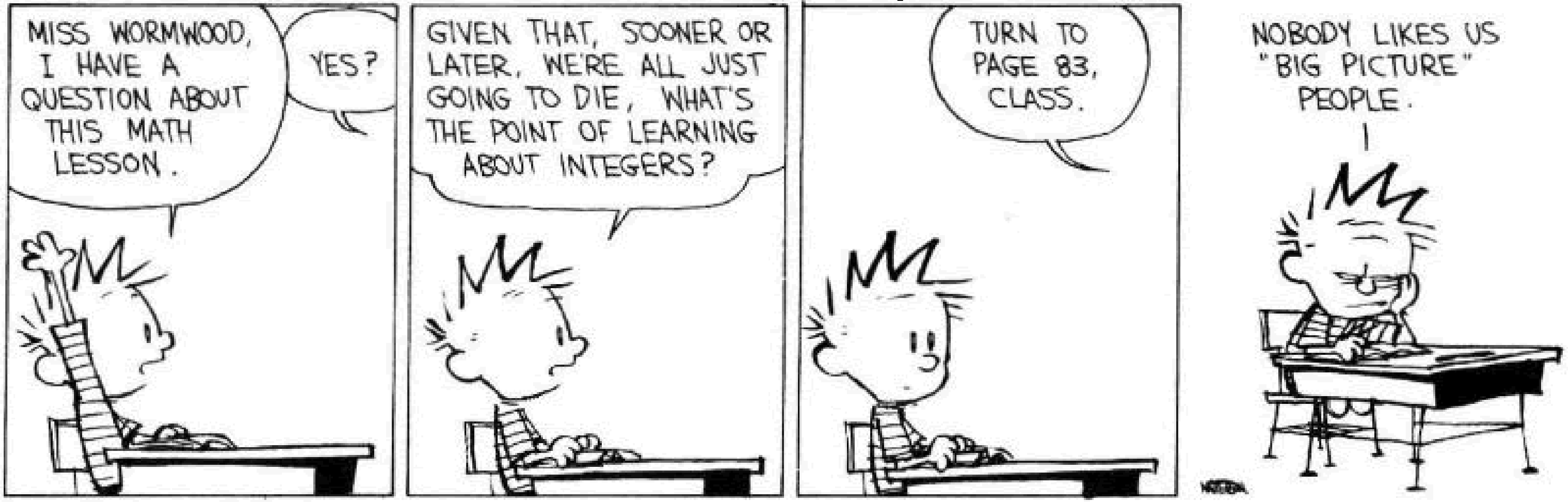
- VC is so good at abstracting syntactic details that it is *syntactically preserved* by many common optimizations
  - Register allocation, instruction scheduling
  - Common subexp elim, constant and copy propagation
  - Dead code elimination
- We have *identical* VCs whether or not an optimization has been performed
  - Preserves syntactic form, not just semantic meaning!
- This can be used to verify correctness of compiler optimizations (Translation Validation)

VC Characterize a Safe Interpreter

- Consider a fictitious "safe" interpreter
  - As it goes along it performs checks (e.g. "safe to read from this memory addr", "this is a null-terminated string", "I have not already acquired this lock")
  - Some of these would actually be hard to implement
- The VC describes all of the checks to be performed
  - Along with their context (assumptions from conditionals)
  - Invariants and pre/postconditions are used to obtain a finite expression (through induction)
- VC is valid $\Rightarrow$ interpreter *never fails*
  - We enforce same level of "correctness"
  - But better (static + more powerful checks)

VC Big Picture

- Verification conditions
  - Capture the semantics of code + specifications
  - Language independent
  - Can be computed backward/forward on structured/unstructured code
  - Make Axiomatic Semantics practical


MISS WORMWOOD, I HAVE A QUESTION ABOUT THIS MATH LESSON.
YES?
GIVEN THAT, SOONER OR LATER, WE'RE ALL JUST GOING TO DIE, WHAT'S THE POINT OF LEARNING ABOUT INTEGERS?
TURN TO PAGE 83, CLASS.
NOBODY LIKES US "BIG PICTURE" PEOPLE.

Invariants Are Not Easy

- Consider the following code from QuickSort

```
int partition(int *a, int L_0, int H_0, int pivot) {
    int L = L_0, H = H_0;
    while(L < H) {
        while(a[L] < pivot) L ++;
        while(a[H] > pivot) H --;
        if(L < H) { swap a[L] and a[H] }
    }
    return L
}
```

- Consider verifying only memory safety
- What is the loop invariant for the outer loop ?

Wei Hu Memorial Homework Award

- Many turned in HW3 code like this:

```
let rec matches re s = match re with
| Star(r) -> union (singleton s)
                    (matches (Concat(r,Star(r))) s)
```

- Which is a direct translation of:

$$R[\![r^*]\!]s = \{s\} \cup R[\![rr^*]\!]s$$

or, equivalently:

$$R[\![r^*]\!]s = \{s\} \cup \{\ y \mid \exists x \in R[\![r]\!]s \wedge y \in R[\![r^*]\!]x\ \}$$

- Why doesn't this work?