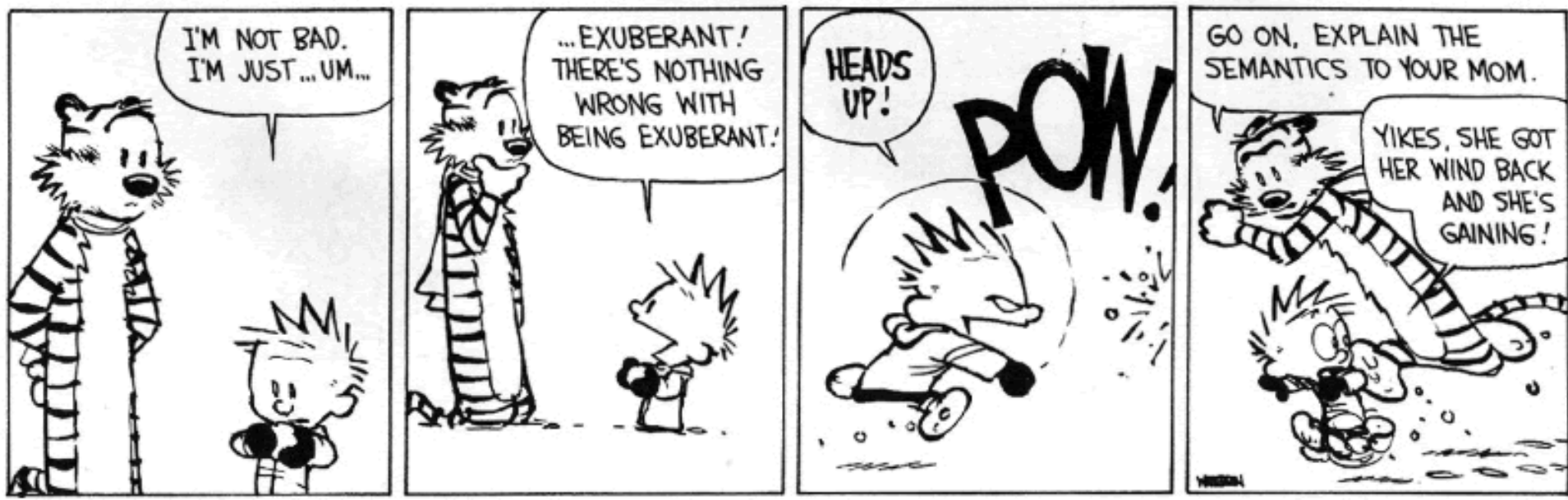


Symbolic Execution



Observations

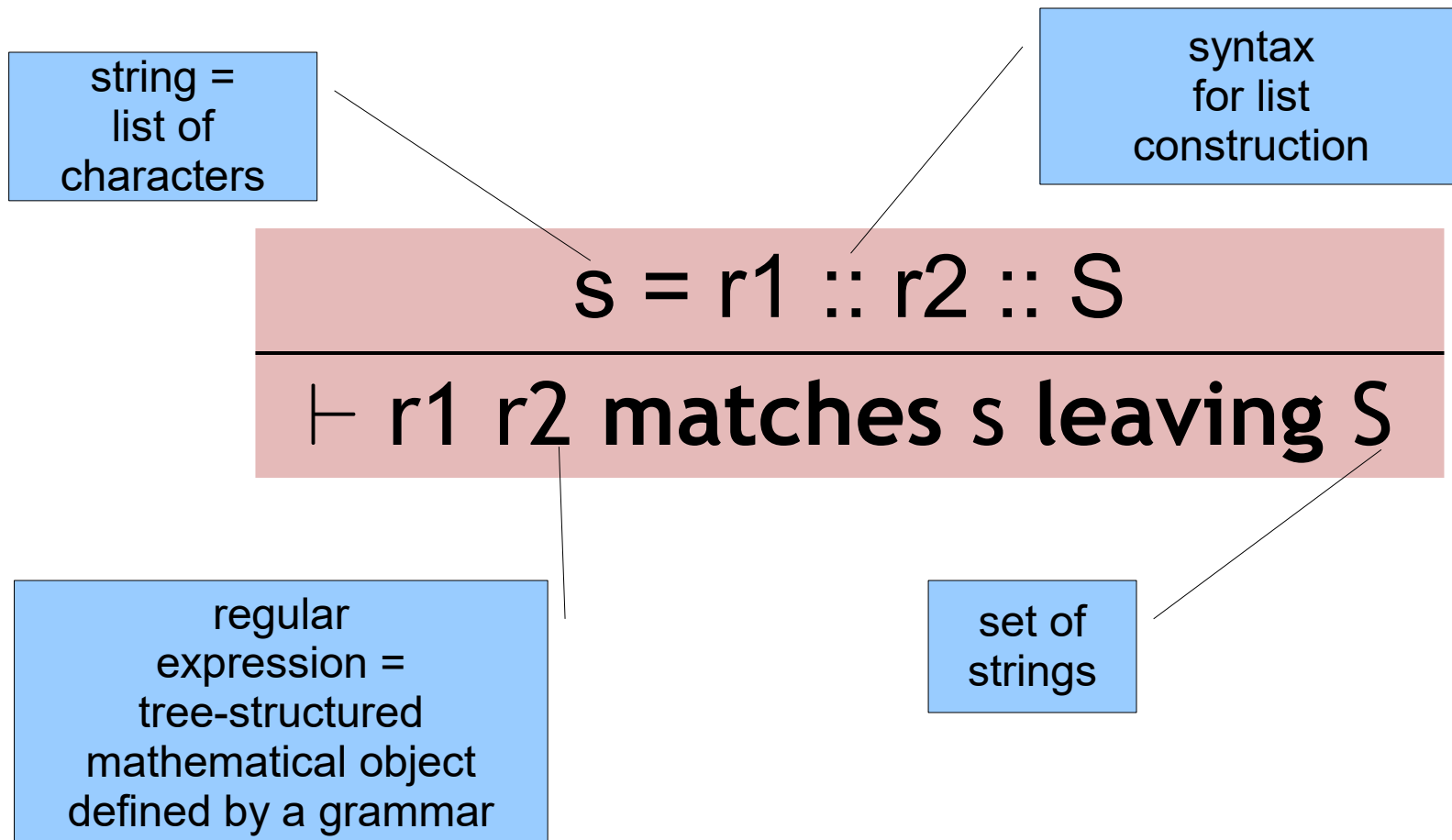
- A key part of doing research is noticing when something is incongruous. This is related to spotting patterns.

Observations

- A key part of doing research is noticing when something is incongruous. This is related to spotting patterns. Hint for HW 3:
- $\text{suffix} \quad === \text{state}$
- $r1 \ r2 \quad === c1 ; c2$
- $r1^* \quad === \text{while ? do } r1$
- $r1 \mid r2 \quad === \text{if ? then } r1 \text{ else } r2$

What's Wrong Here?

- Let's look closely at this “opsem rule”



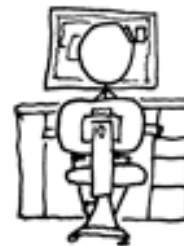
One-Slide Summary

- **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.

Symbolic Execution

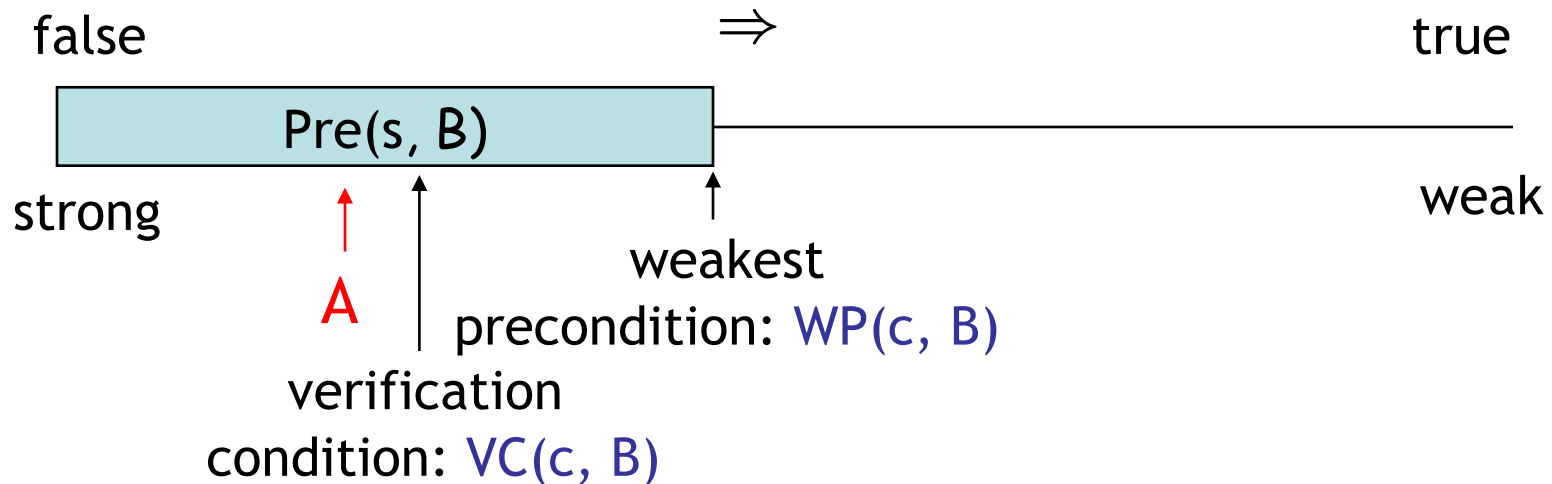


Copyright 2006 Hans Bjordahl



Not Quite Weakest Preconditions

- Recall what we are trying to do:



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

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 $\text{VC}(\text{annotated_command}, \text{post_condition})$ is called VCGen

Verification Condition Generation

- Mostly follows the definition of the wp function:

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

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

$$\begin{aligned} \text{VC}(\text{if } b \text{ then } c_1 \text{ else } c_2, B) = \\ b \Rightarrow \text{VC}(c_1, B) \wedge \neg b \Rightarrow \text{VC}(c_2, B) \end{aligned}$$

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

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

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

VCGen for WHILE

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

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

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

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

The Loop Invariant Need Not Be “Trusted”

```
$ ./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: extensions to Symbolic Execution

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

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 $\forall x. x \leq 0 \Rightarrow I(x)$
- 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

Forward VCGen

- Traditionally the VC is computed backwards
 - That's how we've been doing it in class
 - Backwards 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
 - Test input generation, document generation, specification mining, security analyses, ...
 - This technique is perennially popular

Symbolic Execution is Popular

POPL & PLDI 2024

- **Symbolic execution** is a powerful technique for bug finding by generating test inputs to systematically explore all feasible paths within a given threshold. However, its practical usage is often limited by the path explosion problem.
 - *Compatible Branch Coverage Driven Symbolic Execution for Efficient Bug Finding*
- We define QSE, a **symbolic execution** framework for quantum programs by integrating symbolic variables into quantum states and the outcomes of quantum measurements. The **soundness** of QSE is established through a theorem that ensures the correctness of symbolic execution within **operational semantics**.
 - *Symbolic Execution for Quantum Error Correction Programs*
- For every collected constraint that contains an expectation operator, the algorithm **symbolically evaluates** the expected values to obtain ...
 - *Equivalence and Similarity Refutation for Probabilistic Programs*

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 $x_1 := e_1$, $\Sigma_1 = \{ x_1 \rightarrow e_1 \}$
 - After $x_2 := e_2$, $\Sigma_2 = \{ x_1 \rightarrow e_1, x_2 \rightarrow e_2[e_1/x_1] \}$
 - 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:
 $I ::= x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid L: \mid \text{return} \mid \text{inv } e$
- The “**inv e**” instruction is an annotation
 - Says that boolean expression **e** is true 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:

$\Sigma : \text{Var} \rightarrow \text{SymbolicExpressions}$

$\Sigma(x)$ = 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 ...
- (A *symbolic* value can be an expression like $x+2$, not just a constant like 7.)

Symex Invariants

- The symbolic executor tracks invariants reached so far
- 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

Symex Rules

- Define a VC function as an interpreter:

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

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

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



Recall: $\text{Inv} =$
“invariants
visited so far”

Symex Invariants (2a)

Two cases when seeing an invariant instruction:

1. We see the invariant for the first time

- $I_k = \text{inv } e$
- $k \notin \text{Inv}$ (= “not in the set of invariants we’ve seen”)
- Let $\{y_1, \dots, y_m\}$ = the variables that could be modified on a path from the invariant back to itself
- Let a_1, \dots, a_m be fresh new symbolic parameters

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

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

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

(like a function call)

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 (many?) tools use a more simplistic approach
 - They do not require invariants
 - Instead, iterate through the loop a fixed # of times
 - e.g., PREFIX, versions of Extended Static Checking
 - Sacrifice completeness for usability

Symex Summary

- Let x_1, \dots, x_n be all the variables and a_1, \dots, a_n fresh parameters
- Let Σ_0 be the state $[x_1 := a_1, \dots, x_n := a_n]$
- Let \emptyset be the empty *Inv* set
- For all functions f in your program, prove:
$$\forall a_1 \dots a_n. \Sigma_0(\text{Pre}_f) \Rightarrow \text{VC}(f_{\text{entry}}, \Sigma_0, \emptyset)$$
- 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 (op sem)
- Or via a proof technique called co-induction (or, assume-guarantee)

Forward VCGen Example

- Consider the program

Precondition: $x \leq 0$

Loop: *inv* $x \leq 6$

if $x > 5$ goto End

$x := x + 1$

goto Loop

End: return *Postconditon: $x = 6$*

Forward VCGen Example (2)

$\forall x.$

$$x \leq 0 \Rightarrow$$

$$x \leq 6 \wedge$$

$\forall x'.$

$$(x' \leq 6 \Rightarrow$$

$$x' > 5 \Rightarrow x' = 6$$

\wedge

$$x' \leq 5 \Rightarrow x' + 1 \leq 6)$$

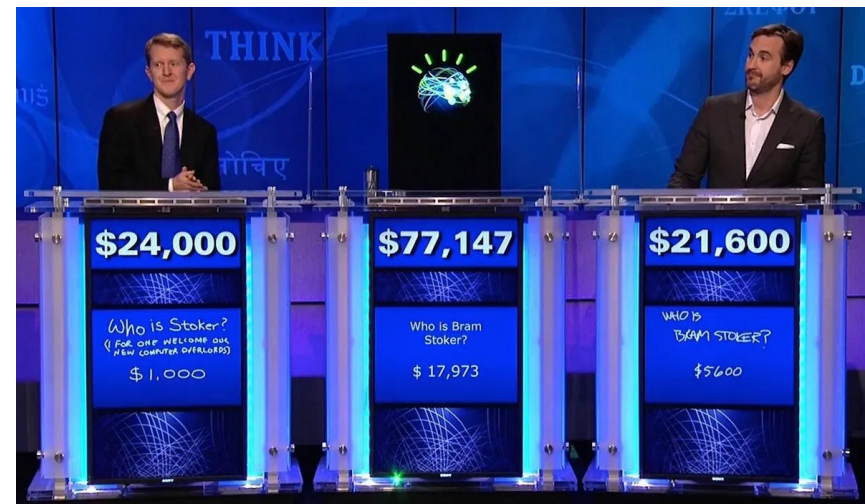
- VC contains both proof obligations and assumptions about the control flow

VCs Can Be Large

- Consider the sequence of conditionals
(if $x < 0$ then $x := -x$); (if $x \leq 3$ then $x += 3$)
 - With the postcondition $P(x)$
- The VC is
$$\begin{array}{lll} x < 0 \wedge -x \leq 3 & \Rightarrow P(-x + 3) & \wedge \\ x < 0 \wedge -x > 3 & \Rightarrow P(-x) & \wedge \\ x \geq 0 \wedge x \leq 3 & \Rightarrow P(x + 3) & \wedge \\ x \geq 0 \wedge x > 3 & \Rightarrow P(x) & \end{array}$$
- There is one conjunct for each path
 \Rightarrow **exponential** number of paths!
 - Conjuncts for **infeasible paths have un-satisfiable guards!**
- Try with $P(x) = x \geq 3$

Game Shows

- Name the former computer programmer who set the record as the highest-earning American game show contestant. He won *Jeopardy!* 74 consecutive times and is currently its host.
- “Who is ...?”



Video Game Music

- This Japanese composer (植松 伸夫) made music for most of the main entries in the Final Fantasy series (including the recent Final Fantasy VII Remake). He also composed for Chrono Trigger, Nier: Automata, and created the theme for Super Smash Bros. Brawl.



English Prose

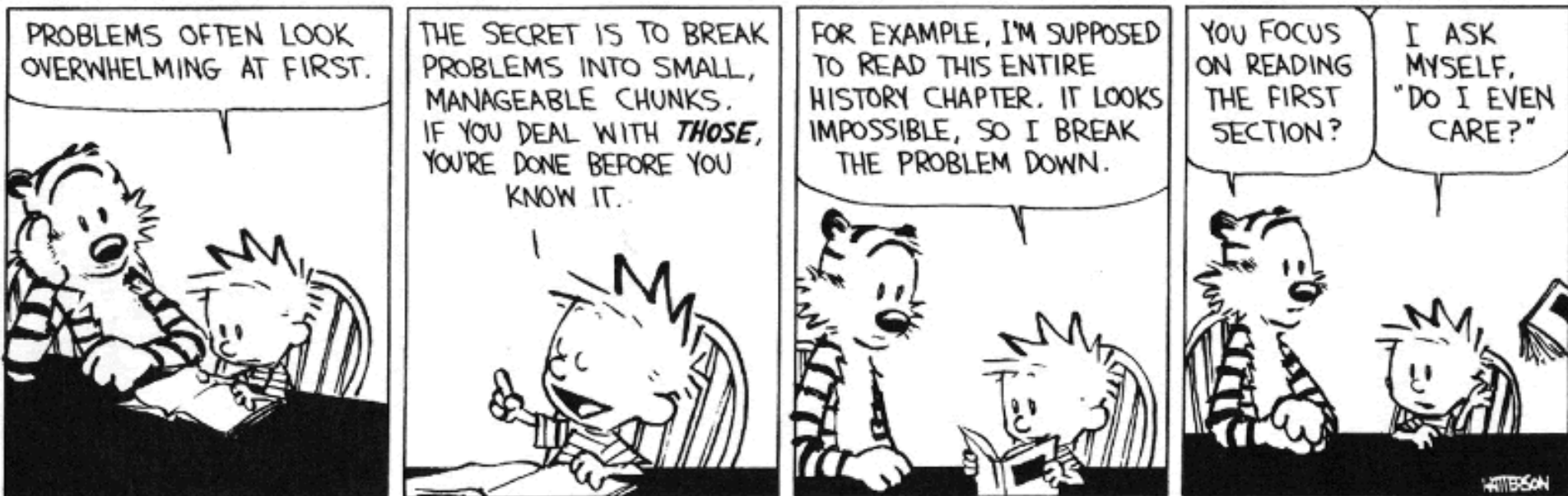
341. Van and Hitomi walked an inaudible distance from those guy's Van was hanging out with.
253. However, when he got into his chamber and sat down with a blank canvas propped up on its easel, his vision vanished as if it were nothing but a floating dust moat.
352. "Good evening my league." He picked her up by the wrist. "I think that you and I have some talking to do, actually I have a preposition"

Computer Science

- This American Turing award winner is known for the “law” that “Adding humans to a late software project makes it later.” The Turing Award citation notes landmark contributions to operating systems, software engineering and computer architecture. Notable works include *No Silver Bullet: Essence and Accidents of Software Engineering* and *The Mythical ____*.

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?



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)

$$\text{VC}(\text{after } c \text{ establish } \text{Inv}, P) =_{\text{def}} \text{VC}(c, \text{Inv}) \wedge \forall x_i. \text{Inv} \Rightarrow P$$

- where x_i are the ModifiedVars(c)
- Use when c contains many paths
 - after if $x < 0$ then $x := -x$ establish $x \geq 0$;
 - if $x \leq 3$ then $x += 3$ { $P(x)$ }
- VC is now:

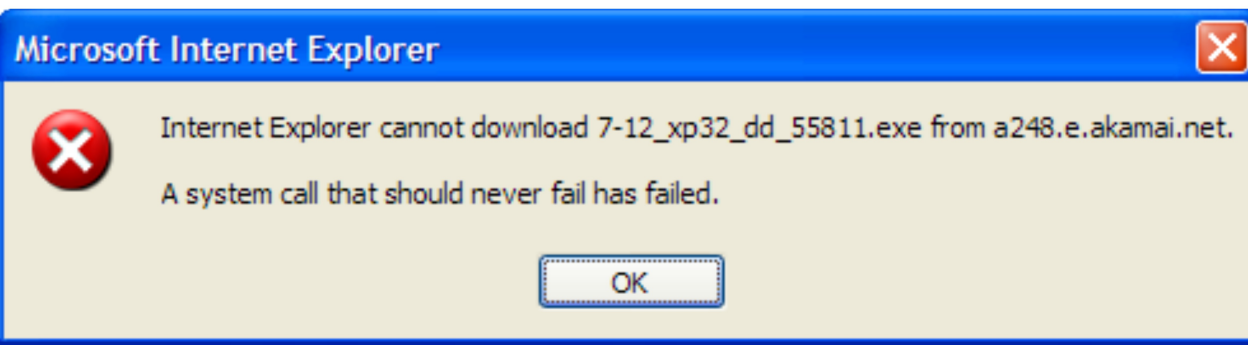
$$(x < 0 \Rightarrow -x \geq 0) \wedge (x \geq 0 \Rightarrow x \geq 0) \wedge \\ \forall x. x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3) \wedge x > 3 \Rightarrow P(x))$$

Dropping Paths

- In absence of annotations, we can drop some paths
- $VC(\text{if } E \text{ then } c_1 \text{ else } c_2, P) = \text{choose one of}$
 - $E \Rightarrow VC(c_1, P) \wedge \neg E \Rightarrow VC(c_2, P)$ (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:
 - An established trend is to sacrifice soundness to increase usability (e.g., Dawson Engler et al.)
 - 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 $VC(\text{throw}, P)$?



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 $VC(\text{throw}, P)$?
- 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

$$VC(\text{skip}, P, Q) = P$$

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

$$VC(\text{throw}, P, Q) = Q$$

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

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

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”:

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

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

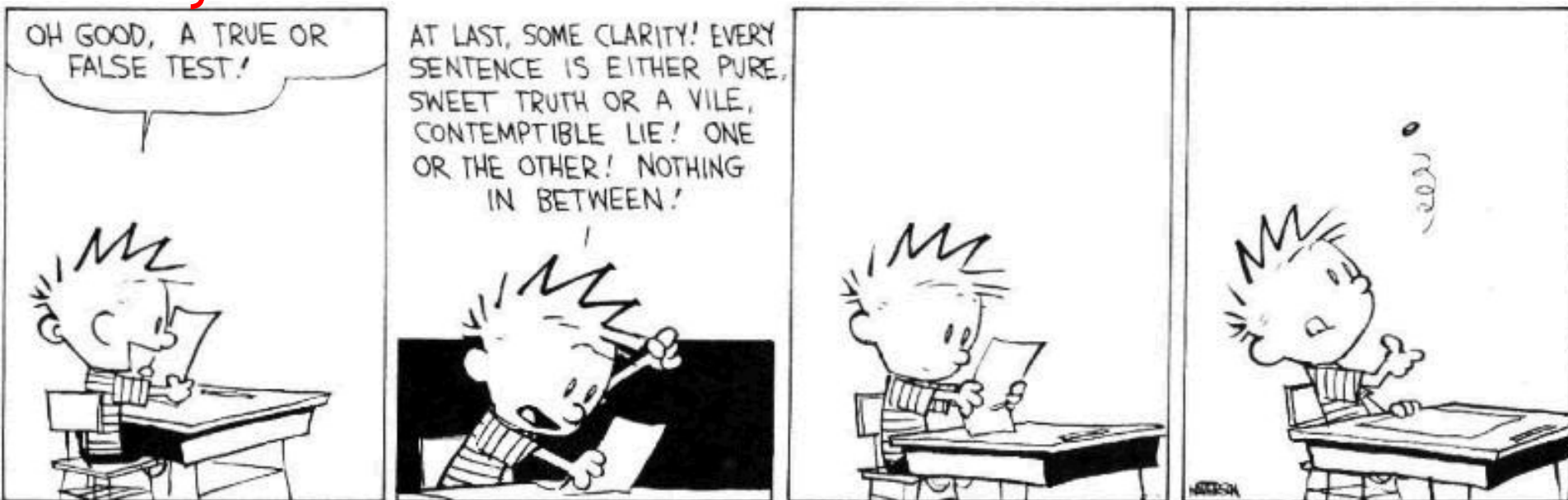
$$VC(c_1, \text{true}, VC(c_2, Q, Q))$$

- 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?



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:
 - $\text{sel}(M, A)$ denotes the contents of the memory cell
 - $\text{upd}(M, A, V)$ 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:

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

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

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

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

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

Memory Aliasing

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

$$\begin{aligned} A &= [\text{upd}(\mu, x, 5)/\mu] (*x + *y = 10) \\ &= [\text{upd}(\mu, x, 5)/\mu] (\text{sel}(\mu, x) + \text{sel}(\mu, y) = 10) \\ (1) \quad &= \text{sel}(\text{upd}(\mu, x, 5), x) + \text{sel}(\text{upd}(\mu, x, 5), y) = 10 \\ &= 5 + \text{sel}(\text{upd}(\mu, x, 5), y) = 10 \\ &= \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + \text{sel}(\mu, y) = 10 \\ (2) \quad &= x = y \text{ or } *y = 5 \end{aligned}$$

- 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 undecidable)**
- Sometimes completeness is sacrificed with the following (approximate) rule:

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

- The meaning of “obviously” varies:
 - The addresses of two distinct globals are \neq
 - The address of a global and one of a local are \neq
- PREFIX and GCC use such schemes

VCGen Overarching Example

- Consider the program
 - Precondition: $B : \text{bool} \wedge A : \text{array}(\text{bool}, L)$
 - 1: $I := 0$
 - $R := B$
 - 3: $\text{inv } I \geq 0 \wedge R : \text{bool}$
 - if $I \geq L$ goto 9
 - $\text{assert } \text{saferd}(A + I)$
 - $T := *(A + I)$
 - $I := I + 1$
 - $R := T$
 - goto 3
 - 9: return R
 - Postcondition: $R : \text{bool}$

VCGen Overarching Example

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

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

Mutable Records - Two Models

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

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: $((4 == 5) : \text{bool}) \wedge (\text{not } (4 == 5))$
- No confusion from reuse of x with different types

Invariance of VC Across Optimizations

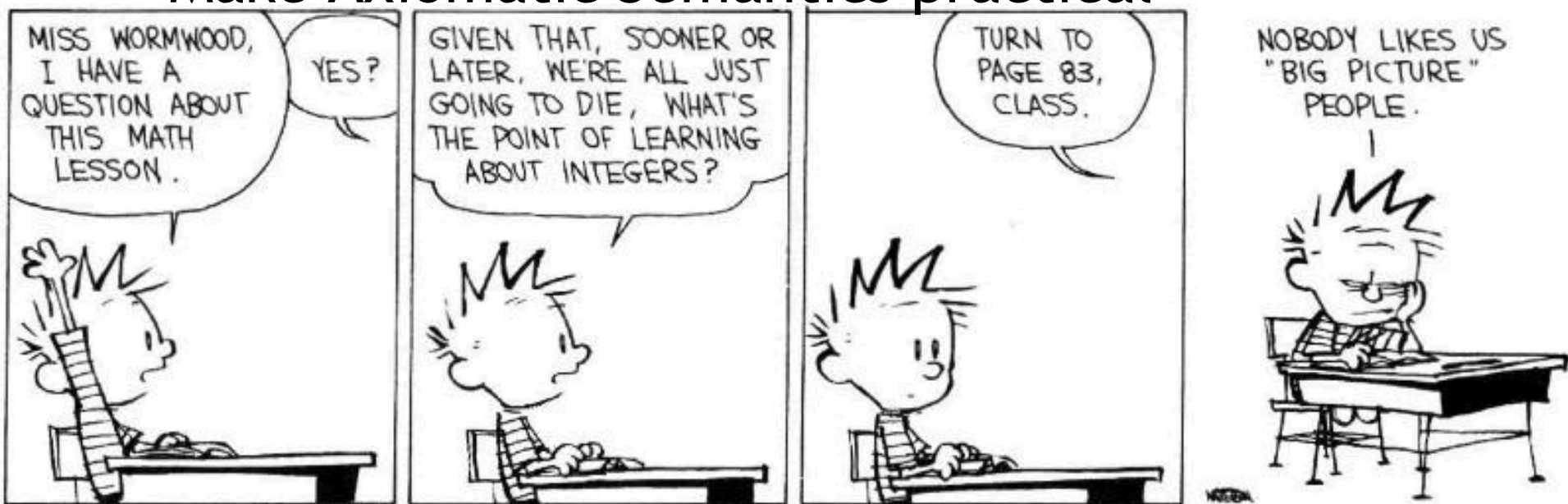
- VC is so good at abstracting syntactic details that it is *syntactically preserved* by many common optimizations
 - Register allocation, instruction scheduling
 - Common subexp elim, constant and copy propagation
 - Dead code elimination
- We have *identical* VCs whether or not an optimization has 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



Invariants Are Not Easy

- Consider the following code from QuickSort

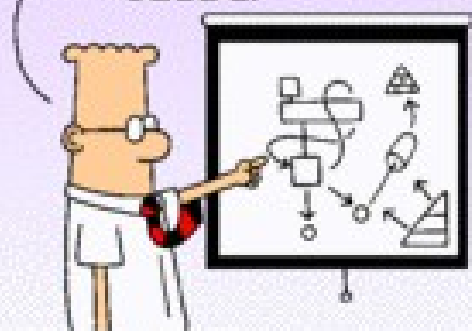
```
int partition(int *a, int L0, int H0, int pivot) {  
    int L = L0, H = H0;  
    while(L < H) {  
        while(a[L] < pivot) L++;  
        while(a[H] > pivot) H--;  
        if(L < H) { swap a[L] and a[H] }  
    }  
    return L  
}
```

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

Done!

- Questions?

YOU WON'T READ MY
TECHNICAL REPORT SO
I SUMMARIZED IT IN
THIS COMPLICATED
SLIDE.



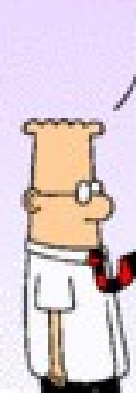
www.dilbert.com scottadams@aol.com

IF YOU STARE AT IT
LONG ENOUGH YOU WILL
EITHER EXPERIENCE THE
ILLUSION OF UNDER-
STANDING IT OR BE TOO
EMBARRASSED TO ADMIT
YOU DON'T.



4-4-08 © 2008 Scott Adams, Inc./Dist. by UPS, Inc.

DO YOU HAVE ANY
QUESTIONS TO BETRAY
YOUR IGNORANCE?



IS THE
TRIANGLE
THING MAD AT
THE TUBE?

