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:

  \[ R_{r^*}^{s} = \{ s \} \cup \bigcup_{y} \iff \exists x \in R_{r}^{s} \land y \in R_{r^*}^{x} \]

• Why doesn’t this work?

Forward VC Gen Intuition

• Consider the sequence of assignments

  \( x_1 := \text{e}_1; x_2 := \text{e}_2 \)

• The VC(c, B) = \( [\text{e}_1/x_1][\text{e}_2/x_2]B \)

• We can compute the substitution in a forward way using symbolic execution (aka symbolic evaluation)
  - Keep a symbolic state that maps variables to expressions
  - After \( x_1 := \text{e}_1, \Sigma_1 = \{ x_1 \rightarrow \text{e}_1 \} \)
  - After \( x_2 := \text{e}_2, \Sigma_2 = \{ x_1 \rightarrow \text{e}_1, x_2 \rightarrow \text{e}_2/x_1 \} \)
  - Note that we have applied \( \Sigma_1 \) as a substitution to right-hand side of assignment \( x_2 := \text{e}_2 \)

Simple Assembly Language

• Consider the language of instructions:

  \[
  \begin{align*}
  I & ::= x := e \mid f() \mid \text{if } e \text{ goto } L \mid \text{goto } L \mid \text{return } \mid \text{inv } e \\
  L: & = \text{return } \mid \text{inv } e \\
  \end{align*}
  \]

• The “\text{inv } e” instruction is an annotation
  - Says 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:

  \[ \Sigma : \text{Var} \rightarrow \text{SymbolicExpressions} \]

  \[ \Sigma(x) = \text{the symbolic value of } x \text{ in state } \Sigma \]

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

• 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:
  \[ VC : \text{Address} \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion} \]

  - IF \( I_k = \text{return} \)
    \[ \Sigma \]
  - IF \( I_k = \text{if } e \text{ goto } L \)
    \[ e \Rightarrow \text{VC}(L, \Sigma, \text{Inv}) \]
    \[ \neg e \Rightarrow \text{VC}(k+1, \Sigma, \text{Inv}) \]

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

  - IF \( I_k = f() \)
    \[ \Sigma(\text{Pre } f) \land \forall a_1, \ldots, a_n. \Sigma'(\text{Post } f) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv} \cup \{ k \}) \]

Symex Invariants (2a)

Two cases when seeing an invariant instruction:

1. We see the invariant for the first time
   - \( I_k = \text{inv } e \)
   - \( k \notin \text{Inv} \)
   - Let \( \{ y_1, \ldots, y_m \} \) be the variables that could be modified on a path from the invariant back to itself
   - Let \( a_1, \ldots, a_n \) be fresh new symbolic parameters

   \[ \text{VC}(k, \Sigma, \text{Inv}) = \Sigma(e) \land \forall a_1, \ldots, a_n. \Sigma'(e) \Rightarrow \text{VC}(k+1, \Sigma', \text{Inv} \cup \{ k \}) \]

   (like a function call)

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)

Symex Invariants (2b)

- 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

Where Are We?

- **Axiomatic Semantics**: the meaning of a program is what is true after it executes
- **Hoare Triples**: \{ A \} \text{c} \{ B \}
- **Weakest Precondition**: \{ WP(c,B) \} \text{c} \{ B \}
- **Verification Condition**: \( A \Rightarrow \text{VC}(c,B) \Rightarrow \text{WP}(c,b) \)
  - 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”**

Symex Summary

- Let \( x_1, \ldots, x_n \) be all the variables and \( a_1, \ldots, a_n \) fresh parameters
- Let \( \Sigma_0 \) be the state \[ x_1 := a_1, \ldots, x_n := a_n \]
- Let \( \emptyset \) be the empty \text{Inv} set

  \[ \forall a_1, \ldots, a_n. \Sigma_0(\text{Pre}_i) \Rightarrow \text{VC}(f_{entry}, \Sigma_0, \emptyset) \]

  If you start the program by invoking any \( f \) in a state that satisfies \( \text{Pre}_i \), then the program will execute such that
  - At all “\text{inv } e” the \( e \) holds, and
  - If the function returns then \( \text{Post} \), holds

  Can be proved w.r.t. a real interpreter (operational semantics)
  - Or via a proof technique called co-induction (or, \text{assume-guarantee})
Forward VCGen Example

• Consider the program

  Precondition: \( x \leq 0 \)

  Loop: \( \text{inv} \ x \leq 6 \)
     if \( x > 5 \) goto End
     \( x := x + 1 \)
     goto Loop

  End: return

Postcondition: \( x = 6 \)

Forward VCGen Example (2)

\( \forall x. \)

\( x \leq 0 \Rightarrow \\
  x \leq 6 \land \\
  \forall x'. \\
  (x' > 5 \Rightarrow x' = 6) \land \\
  x' \leq 5 \Rightarrow x' + 1 \leq 6 ) \)

• 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

  \[ 
  x < 0 \land x \leq 3 \Rightarrow P(x+3) \land \\
  x < 0 \land x > 3 \Rightarrow P(x) \land \\
  x \geq 0 \land x < 3 \Rightarrow P(x) 
  \]

  There is one conjunct for each path
  \( \Rightarrow \) exponential number of paths!

  - Conjuncts for infeasible paths have un-satisfiable guards!
  - Try with \( P(x) = x \geq 3 \)

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?

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)

  \[ 
  \text{VC} \text{c} \text{Inv} \equiv \\
  \text{VC}(c, \text{Inv}) \land \forall x, \text{Inv} \Rightarrow P 
  \]

  where \( x \) 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:

  \[ 
  (x < 0 \Rightarrow x \geq 0) \land \\
  (x \geq 0 \Rightarrow x \geq 0) \land \\
  \forall x, x \geq 0 \Rightarrow (x \leq 3 \Rightarrow P(x+3)) \land x > 3 \Rightarrow P(x) 
  \]
Dropping Paths

- In absence of annotations, we can drop some paths
  - $\text{VC}(\text{if } E \text{ then } c_1 \text{ else } c_2, P) = \text{choose one of}$
  - $E \Rightarrow \text{VC}(c_1, P)$ (drop no paths)
  - $\neg E \Rightarrow \text{VC}(c_2, P)$ (drops “else” 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):
  - $\text{throw}$ throws an exception
  - $\text{try } c_1 \text{ catch } c_2$ executes $c_2$ if $c_1$ throws

Problem:
- We have non-local transfer of control
- What is $\text{VC}(\text{throw}, P)$?

VCGen for Exceptions (2)

- $\text{VC}(c, P, Q)$ is a precondition that makes $c$
  - either not terminate, or terminate normally
    - with $P$ or throw an exception with $Q$
  - Problem:
  - We have non-local transfer of control
  - What is $\text{VC}(\text{throw}, P)$?

Rules
- $\text{VC}(\text{skip}, P, Q) = P$
- $\text{VC}(c_1; c_2, P, Q) = \text{VC}(c_1, \text{VC}(c_2, P, Q), Q)$
- $\text{VC}(\text{throw}, P, Q) = Q$
- $\text{VC}(\text{try } c_1 \text{ catch } c_2, P, Q) = \text{VC}(c_1, P, \text{VC}(c_2, P, Q))$
- $\text{VC}(\text{try } c_1 \text{ finally } c_2, P, Q) = ?$

VCGen Finally

- Given these:
  - $\text{VC}(c_1; c_2, P, Q) = \text{VC}(c_1, \text{VC}(c_2, P, Q), Q)$
  - $\text{VC}(\text{try } c_1 \text{ catch } c_2, P, Q) = \text{VC}(c_1, P, \text{VC}(c_2, P, Q))$
- Finally is somewhat like “if”:
  - $\text{VC}(\text{try } c_1 \text{ finally } c_2, P, Q) = \text{VC}(c_1, \text{VC}(c_2, P, Q), \text{true}) \land \text{VC}(c_1, \text{true}, \text{VC}(c_2, Q, Q))$
- Which reduces to:
  - $\text{VC}(c_1, \text{VC}(c_2, P, Q), \text{VC}(c_2, Q, Q))$

Hoare Rules and the Heap

- When is the following Hoare triple valid?
  - $\{ A \} \ast x := 5 \{ \ast x + \ast y = 10 \}$
- A should be “$y = 5$ or $x = y$”
- The Hoare rule for assignment would give us:
  - $[5/x]\{ \ast x + \ast y = 10 \} = 5 + \ast y = 10$
  - $\ast y = 5$ (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 $\mu$ (mu) in assertions to refer to the current memory
• Example:

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

  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:

  $$A = \begin{cases} \text{upd}(\mu, x, 5)/\mu \} \{ *x + *y = 10 \} \\ \text{sel}(\text{upd}(\mu, x, 5)/\mu) \{ \text{sel}(\mu, x) + \text{sel}(\mu, y) = 10 \} \\ \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + \text{sel}(\mu, y) = 10 \end{cases}$$

  (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 $\text{NP}$-hard (and/or undecidable)
• Sometimes completeness is sacrificed with the following (approximate) rule:

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

  • 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: $B : \text{bool} \land A : \text{array(boo, L)}$
  1: I := 0
  2: R := B
  3: inv I \geq 0 \land R : \text{bool}
    if I \geq L goto 9
    assert saferd(A + I)
  4: T := *(A + I)
  5: I := I + 1
  6: R := T
  7: goto 3
  8: return R
• Postcondition: $R : \text{bool}$
VCGen Overarching Example

\[ \forall A. \forall B. \forall L. \forall \mu \]
\[ B : \text{bool} \land A : \text{array(bool, L)} \Rightarrow 0 \geq 0 \land B : \text{bool} \land \\
\forall I. \forall R. \\
I \geq 0 \land R : \text{bool} \Rightarrow \\
I \geq L \Rightarrow R : \text{bool} \land \\
I < L \Rightarrow \text{saferd}(A + I) \land \\
sel(\mu, A + I) : \text{bool} \]

- 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
  - \( r.f1 = \text{sel}(r,f1) \) and \( r.f1 := E \) is \( r := \text{upd}(r,f1,E) \)
- Method 2: one “memory” for each field
  - The record address is the index
  - \( r.f1 = \text{sel}(f1,r) \) 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 + sizeof(T1)

VC as a “SemanticChecksum”

- 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
  - \( x := 4 \\
x := x == 5 \\
\text{assert } x : \text{bool} \\
x := \text{not } x \\
\text{assert } x \)
- High-level type checking is not appropriate here
- The VC is: \( 4 == 5 : \text{bool} \land \text{not } (4 == 5) \)
- 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 subexpr elimination, 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
  ```c
  int partition(int *a, int L, int H, int pivot) {
    int L = L, H = H;
    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?

Homework

- Homework 4 Due Thursday
- Read Cousot & Cousot article
- Read Abramski article
- Project Proposal Due In One Week