Announcements
- Homework 7 due Thu
- HW6 Time Spent:
  - 5.5 hrs mean, 4.2 hrs stddev, 4 hrs median

Review of Verification Conditions

- Axiomatic semantics: The meaning of a program is *putting properties on programs*
- Hoare triples
  - weakest precondition:
    - $\{A\} \implies \{B\}$
  - verification condition:
    - $A \implies \{B\}$

Not Quite Weakest Preconditions

- Recall what we are trying to do:
  - false $\Rightarrow$ true
  - weakest precondition: $WP(c, B)$
  - verification condition: $VC(c, B)$
- Inv holds on entry
- Inv is preserved in an arbitrary iteration
- $B$ holds when the loop terminates in an arbitrary iteration

VCGen for while

- $VC(while b \implies c \implies \{b \Rightarrow VC(c, Inv) \land \neg \neg \Rightarrow B)\} = Inv \land (x_1 \implies x_n \implies Inv = ((b \Rightarrow VC(c, Inv)) \land \neg \neg \Rightarrow B)))$
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 unstructured languages (e.g., assembly language)
  - Uses symbolic execution, a technique that has broad applications in program analysis
    - e.g., the PREfix tool (Intrinsa, Microsoft) works this way

End of Review
Questions?

Plan for Applying VCGen

- Symbolic Execution and 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 “Semantic Checksum”

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

Symbolic Execution Symbolic State

- We set up a symbolic execution state:
  \[ \Sigma : \text{Var} \rightarrow \text{SymbolicExpressions} \]
- \( \Sigma(x) = \text{the symbolic value of } x \text{ in state } \Sigma \)
- \( \Sigma[x:=e] = \text{a new state where } x \text{'s value is } e \)
- We use states as substitutions:
  \( \Sigma(e) \leftarrow e \text{ where } x \text{ replaced by } \Sigma(x) \text{ for any } x \)
- So far, much like operational semantics

Symbolic Execution Invariant State

- The symbolic executor keeps track of the encountered invariants
- A new part of symex state: Inv \( \subseteq \{1\ldots n\} \)
- If \( k \in \text{Inv} \) then \( I_k \) is an invariant instr. 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
Symbolic Execution Rules

• Define a VC function as an interpreter:
  \[ VC : \text{Address} \times \text{SymbolicState} \times \text{InvariantState} \rightarrow \text{Assertion} \]

\[
\begin{align*}
VC(k, \Sigma, \text{Inv}) &= \Sigma(e) \quad \text{if } I_k = \text{return} \\
VC(k, \Sigma, \text{Inv}) &= \Sigma(\text{Pre}(f)) \land \\
& \quad \land \forall a_1, a_2, \ldots, a_m. \Sigma'(e) \\
& \quad \text{if } I_k = f() \\
VC(k, \Sigma, \text{Inv}) &= \Sigma(\text{Post}(f)) \\
& \quad \land \forall a_1, a_2, \ldots, a_m. \Sigma'(e) \\
& \quad \text{if } I_k = \text{return} \\
& \quad \text{and } a_1, a_2, \ldots, a_m \text{ are fresh parameters} \\
& \quad \text{and } \Sigma' = \Sigma(y_1 := a_1, \ldots, y_m := a_m) \\
& \quad \text{if } I_k = \text{return} \\
& \quad \text{and } a_1, a_2, \ldots, a_m \text{ are fresh parameters} \\
& \quad \text{and } \Sigma' = \Sigma(y_1 := a_1, \ldots, y_m := a_m) \\
& \quad \text{if } I_k = \text{return} \\
\end{align*}
\]

Symbolic Execution Invariants

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 \( y_1, y_2, \ldots, y_n \) be the variables that could be modified on a path from the invariant back to itself
   - Let \( a_1, a_2, \ldots, a_n \) be fresh new symbolic parameters
   - \( VC(k, \Sigma, \text{Inv}) = \Sigma(e) \land \forall a_1, a_2, \ldots, a_n. \Sigma'(e) \rightarrow VC(k+1, \Sigma', \text{Inv}) \)
     with \( \Sigma' = \Sigma(y_1 := a_1, \ldots, y_n := a_n) \)

2. We see the invariant for the second time
   - \( I_k = \text{inv E} \)
   - \( k \in \text{Inv} \)
   - \( VC(k, \Sigma, \text{Inv}) = \Sigma(e) \) (like a function return)

Symbolic Execution Top-Level

• Let \( x_1, \ldots, x_n \) be the variables and \( a_1, \ldots, a_n \) fresh params
• Let \( \Sigma_0 \) 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:
  \[ \forall a_1, a_2, \ldots, a_n. \Sigma_0(\text{Pre}(f)) \Rightarrow VC(f, a_1, a_2, \ldots, a_n) \]
  - If you start the program by invoking \( f \) in a state that satisfies \( \text{Pre}(f) \), then the program will execute such that
    - At all "inv e" the \( e \) holds, and
    - If the function returns then \( \text{Post}(f) \) 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

[Diagram showing a forward VCGen example with a precondition, loop, and postcondition]

• VC contains both proof obligations and assumptions about the control flow
Forward VCGen Example

Precondition: \( x \leq 0 \)

Loop: if \( x \leq 6 \)
- \( x \leftarrow -x \)
- goto Loop
End: \( x = 6 \)

For convenience, name fresh parameter “\( x' \)
(instead of “\( x \)"

\( \forall x. \quad x \leq 0 \Rightarrow x \leq 6 \land \sum_{i=0}^{6} G_i \leq 6 \)

- 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
    \[ x < 0 \land x \leq 3 \Rightarrow P(x = 3) \land x \geq 0 \land x \leq 3 \Rightarrow P(x + 3) \land x = 0 \land x \leq 3 \Rightarrow P(x) \]
  - There is one conjunct for each path
    \( \Rightarrow \) exponential number of paths!
  - Conjuncts for infeasible paths have unsatisfiable 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?

VCs Can Be Exponential. Solutions?

- 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?

  - 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)
  \[ VC(\text{after c establish Inv}, P) = \text{def} \]

Dropping Paths

• Without annotations, we can drop some paths

\[ VC(\text{if } E \text{ then } c_1 \text{ else } c_2, P) = \text{choose one of} \]
  - \[ E \Rightarrow VC(c_1, P) \land \neg E \Rightarrow VC(c_2, P) \] (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

VCGen for Exceptions

• Extend the 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 \( VC(\text{throw}, P) \) ?

• Solutions?

VCGen for Exceptions

• Extend the 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**

- **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 Try-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":**
  - \(VC(\text{try } c_1 \text{ finally } c_2, P, Q) =
    \begin{align*}
    VC(c_1, & \text{ VC(c}_2 \text{, P, Q), true }) \\
    & \lor VC(c_1, \text{ VC(c}_2 \text{, Q, Q})
    \end{align*}\)

- **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?**
  - \(\{ A \} \ast x := 5 \{ \ast x + \ast y = 10 \}\)
  - **A should be \(\ast y = 5 \text{ or } x = y\)**

- **The Hoare rule for assignment would give us:**
  - \(5/\ast x)(\ast x + \ast y = 10) = 5 + \ast y = 10 = \ast y = 5\) (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

• Allow variables to range over memory states
  - 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

\[
\{ E_1 : E_2 \} \; (B[\text{upd}(\mu, E_1, E_2/\mu)])
\]

More on Memory

• 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

Memory Aliasing

• Consider again: \( \{ A \} \; *x := 5 \; (\; *x + *y = 10 \) \)
• We obtain:
  \[ A = [\text{upd}(\mu, x, 5)/\mu] \;(*x + *y = 10) \]
  \[ = [\text{upd}(\mu, x, 5)/\mu] \; (\text{sel}(\mu, x) + \text{sel}(\mu, y) = 10) \]
  \[ (1) \; \text{sel}(\text{upd}(\mu, x, 5), x) + \text{sel}(\text{upd}(\mu, x, 5), y) = 10 \]
  \[ = 5 + \text{sel}(\text{upd}(\mu, x, 5), y) = 10 \]
  \[ = \text{if } x = y \text{ then } 5 + 5 = 10 \text{ else } 5 + \text{sel}(\mu, y) = 10 \]
  \[ (2) \; \text{if } x = y \text{ or } *y = 5 \]
• 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 undecidable)
• Sometimes completeness is sacrificed with the following (approximate) rule:
  \[ \text{sel}(\text{upd}(M, A_1, V), A_2) = \begin{cases} 
  V & \text{if } A_1 = A_2 \\
  \text{sel}(M, A_2) & \text{if } A_1 \neq A_2 \\
  p & \text{otherwise (p is a fresh new parameter)} 
\end{cases} \]
• The meaning of "obvious" 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: \( B : \text{bool} \land A : \text{array(bool, L)} \)
  1: \( I := 0 \)
  \( R := B \)
  3: \( \text{inv } I \geq 0 \land R : \text{bool} \)
    - if \( I \geq L \) goto 9
    - 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**

\( \forall A, B, L, \mu. \forall I. B : \text{bool} \land A : \text{array(bool, L)} \Rightarrow \)
\( 0 \geq 0 \land B : \text{bool} \land \)
\( \forall I. \forall R. I \geq 0 \land R : \text{bool} \Rightarrow \)
\( I \geq L \Rightarrow R : \text{bool} \land \)
\( I < L \Rightarrow \text{saferd}(A + I) \land \)
\( I + 1 \geq 0 \land \)
\( \text{sel}(\mu, A + I) : \text{bool} \)

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

**Mutable Records - Two Models**

- Let \( r : \text{RECORD \{ f1 : T1; f2 : T2 \} END} \)
- 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 + 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"**

- 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 diff. 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 ⇒ 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 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?

Questions?

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.