Generating Verification Conditions

- Most interesting VC generation rule is for loops:
  \[ VC(\text{while } C \text{ do } [I] \ S, Q) = ? \]
- To ensure \( Q \) is satisfied after loop, what condition must hold?
- Assuming \( I \) holds initially, need to check \( I \) is loop invariant
- i.e., need to prove \( (I \land C)S(I) \)
- How can we prove this?

Verification Condition for Loops

- To summarize, to show \( I \) is preserved in loop, need:
  \[ I \land C \Rightarrow awp(S, I) \land VC(S, I) \]
- To show \( I \) is strong enough to establish \( Q \), need:
  \[ I \land \neg C \Rightarrow Q \]
- Putting this together, verification condition for a while loop
  \( S' = \text{while } C \text{ do } [I] \ S \) is:
  \[ VC(S', Q) = (I \land C \Rightarrow awp(S, I) \land VC(S, I)) \land (I \land \neg C \Rightarrow Q) \]
We also need rules to generate VC’s for other statements because there might be loops nested in them.

- VC(x := E, Q) = true
- VC(s1; s2, Q) = VC(s2, Q) \land VC(s1, awp(s2, Q))
- VC(if C then s1 else s2, Q) = VC(s1, Q) \land VC(s2, Q)

Thus, even if program obeys specification, might not be able to prove it b/c loop invariants we use are not strong enough.

Theorem: \(\{P\}S\{Q\}\) is valid if following formula is valid:

\[ VC(S, Q) \land P \rightarrow awp(S, Q) \quad (\ast) \]

Thus, if we can prove of validity of \((\ast)\), we have shown that program obeys specification.

Consider the following code:

\[ i := 1; \text{sum} := 0; \]
\[ \text{while } i \leq n \text{ do } \{ \]
\[ \quad j := 1; \]
\[ \quad \text{while } j \leq i \text{ do } \{ \]
\[ \quad \quad \text{sum} := \text{sum} + j; \quad j := j + 1 \]
\[ \quad \} \]
\[ \} \]

Show the VC’s generated for this program for post-condition \(\text{sum} \geq 0\) – can it be verified?

What is the post-condition we need to show for inner loop? \(\text{sum} \geq 0\)

Suppose annotated invariant for inner loop was \(\text{sum} \geq 0\) instead of \(\text{sum} \geq 0 \land j \geq 0\)

Could the program be verified then? no, because loop invariant not strong enough

While VC generation handles many tedious aspects of the proof, user must still come up with loop invariants (more on this in next few lectures).
IMP with functions and pointers

- The IMP language considered so far does not have many features of realistic PLs
- Let’s enrich IMP with two features, namely functions and pointers
- How to verify programs in this enriched language

Proof rules for Assert and Assume

- Proof rule for assertions:
  \[ P \Rightarrow E \]
  \[ \vdash \{ P \} \text{assert}(E) \{ P \land E \} \]
- Proof rule for assumption:
  \[ \vdash \{ P \} \text{assume}(E) \{ P \land E \} \]

IMP+: IMP with functions

- IMP+ programs defined according to following grammar:

  Program \( P \) := \( F^+ \)
  Function \( F \) := \{ function f(x_1, \ldots, x_n) \{ S; \text{return } e; \} \}
  Statement \( S \) := \( y := f(v_1, \ldots, v_n) \mid \ldots \)

Handling procedure calls

- How do we generate VCs if we encounter procedure calls?
  \[ y = f(x_1, \ldots, x_n) \]
- Just like we asked programmer to provide loop invariants, also ask them for method pre- and post-conditions
- Pre-condition specifies what is expected of \( f \)’s arguments
- Post-condition describes \( f \)’s return value (and side effects)
Pre- and post- Example

▶ Consider a method get that takes an array arr of size n and index i and returns the i'th element
▶ Pre-condition: $0 \leq i < n$
▶ Post-condition: $ret = a[i]$
▶ These pre- and post-conditions are referred to as the method contract

Generating VCs for method calls

▶ Contracts allow us to verify the program in a modular way – generate VCs one function at a time!
▶ There are two questions we need to answer:
  1. How do we verify that a method satisfies its contracts?
  2. How do we handle method calls when generating VCs?

Verifying Contract

▶ Consider the following function declaration:

```plaintext
function f(x1, ..., xn)
  requires(Pre)
  ensures(Post)
  Body;
  return e;
```

▶ Assuming that Post refers to variable ret, we can verify this contract by checking the validity of this Hoare triple:

```
{Pre} Body; ret := e {Post}
```

Verifying Calls

▶ Since method bodies now contain calls, we need to be able to verify Hoare triples involving calls:

```
{P} y := f(e1, ..., en) {Q}
```

▶ To verify this triple, we need to prove that f's precondition Pre is satisfied
▶ But we can also assume that f's post-condition Post holds after the call – why?
▶ Thus, we can model the function call as:

```plaintext
assert(Pre[e1/x1, ..., en/xn]);
assume(Post[tmp/res, e1/x1, ..., en/xn]);
y := tmp;
```
where tmp is a fresh temporary variable.

Modular Verification: Recap

▶ When verifying a callee:
  ▶ We assume the precondition
  ▶ We assert the postcondition
▶ When verifying caller:
  ▶ We assert callee's precondition
  ▶ We assume callee's postcondition
▶ This is crucial for modular verification – decomposes verification task into individual functions

Exercise: Locking Protocol

▶ Suppose we represent locks as integers – 0 means locked; 1 means unlocked
▶ What are the contracts for methods lock and unlock?

“An attempt to re-acquire an acquired lock or release a released lock will cause a deadlock.”
Exercise: Locking Protocol, cont.

- Show the verification conditions for the following caller of lock and unlock:

  ```
  assume(b=0 || b=1);
  l := b;
  if(b != 0) l := lock(l);
  else l := unlock(l);
  if(b = 0) l := lock(l);
  else l := unlock();
  ```

One more complication: Global variables

- So far, we assumed function call does not have side effects
- But suppose that f can modify global variable glob
- Is the previous rule still correct?

Havoc

- To deal with this difficulty, we introduce a new statement called havoc
- The statement `havoc(⃗x)` assigns every variable \( x \in ⃗x \) to an unknown value
- What is wp(\(S, φ\)) where \( S \) is a havoc statement?

Function calls with Side Effects

- To deal with side effects, we assume method contracts also contain info about side effects
- New method contract:

  ```
  Requires P
  Ensures Q
  Modifies \( v_1, v_2, ... \)
  ```

  In addition to checking \( \{P\} Body \{Q\} \), also need to check that the function only modifies variables mentioned in modifies clause

Function calls with Side Effects, cont.

- Given such a method contract, we can model call site \( y := foo(x_1, ..., x_n) \) as follows:

  ```
  assert(Pre[e1/x1, ... en/xn]);
  havoc(v1, ..., vn);
  assume(Post[\{tmp/res, e1/x1, ... en/xn\}]);
  y := tmp;
  ```

IMP with Pointers

- Let’s also add pointers to IMP!

  ```
  Program P := F^+
  Function F := function f(x_1, ..., x_n) { S; return e; } 
  Statement S := y := *x | *x = e | ...
  ```

- Does the old assignment rule still work with pointers?
Counterexample

To see why the old assignment rule does not work, consider the following code snippet:

```plaintext
x := y; *y := 3;
*x := 2; z := *y;
assert(z = 3)
```

Does this assertion hold?

What is the weakest precondition?

Verification with Pointers

As shown by previous example, we cannot deal with references using the standard assignment rule

Key problem: Due to pointer aliasing, \( *x := e \) can affect values of expressions beyond \*x

Solution: Treat memory as a gigantic array \( M \) that maps addresses to values

Need to use theory of arrays & also need new rules for loads and stores

Proof Rules for Loads and Stores

Proof rule for loads:

\[ \vdash \{ Q[M[y/x]] \} \quad x := *y \{ Q \} \]

Proof rule for stores:

\[ \vdash \{ Q[M(x ◁ e)]/M] \} \quad *x := e \{ Q \} \]

Revisiting Example

Let’s consider the previous example again

```plaintext
x := y; *y := 3;
*x := 2; z := *y;
```

What is the weakest precondition for this code snippet?

Deductive Verifiers in Practice

Deductive verification tools are based on these principles we discussed

Examples: Boogie, Dafny, Smack, ESC/Java, Why3, …

They automate VC generation, but require human to provide loop invariants and method pre- and post-conditions (tedious!)

Fortunately, many techniques that can be used to automatically synthesize these annotations!