Verification Conditions for Loops, Functions, and Pointers

Işıl Dillig

Weakest Preconditions for Loops

- Last lecture: To prove $\{P\}S\{Q\}$, compute $wp(S, Q)$ and check if it is implied by P
- Unfortunately, we can't compute weakest preconditions for loops exactly.
- Idea: approximate it using $awp(S, Q)$
- $awp(S, Q)$ may be stronger than $wp(S, Q)$ but not weaker
- To verify $\{P\}S\{Q\}$, show $P \Rightarrow awp(S, Q)$
- Hope is that $awp(S, Q)$ is weak enough to be implied by P although it may not be the weakest

Approximate Weakest Preconditions

- For all statements except for while loops, computation of $awp(S, Q)$ same as $wp(S, Q)$
- To compute, $awp(S, Q)$ for loops, we will rely on loop invariants provided by oracle (human or static analysis)
- Assume all loops are annotated with invariants `while` C `do` $[I]$ S
- Now, we'll just define $awp(\texttt{while } C \texttt{ do } [I]\ S, Q) \equiv I$

Verification with Approximate Weakest Preconditions

- If $P \Rightarrow awp(S, Q)$, does this mean $\{P\}S\{Q\}$ is valid?
- 1.

  2.
- For each statement S, generate verification condition $VC(S, Q)$ that encodes additional conditions to prove

Generating Verification Conditions

- Most interesting VC generation rule is for loops:

$$VC(\texttt{while } C \texttt{ do } [I]\ S, Q) = ?$$

- 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 $S' = \texttt{while } C \texttt{ do } \{I\}\ S$ is:

$$VC(S', Q) = (I \land C \Rightarrow awp(S, I) \land VC(S, I)) \land (I \land \neg C \Rightarrow Q)$$

Verification Condition for Other Statements

- We also need rules to generate VC's for other statements because there might be loops nested in them
- $VC(x := E, Q) = true$
- $VC(s_1; s_2, Q) = VC(s_2, Q) \wedge VC(s_1, awp(s_2, Q))$
- $VC(\texttt{if } C \texttt{ then } s_1 \texttt{ else } s_2, Q) = VC(s_1, Q) \wedge VC(s_2, Q)$

Verification of Hoare Triple

- Thus, to show validity of $\{P\}S\{Q\}$, need to do following:
  1. Compute $awp(S, Q)$
  2. Compute $VC(S, Q)$
- Theorem: $\{P\}S\{Q\}$ is valid if following formula is valid:

$$VC(S, Q) \ \wedge \ P \rightarrow awp(S, Q) \quad (*)$$

- Thus, if we can prove of validity of $(*)$, we have shown that program obeys specification

Discussion

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

$$VC(S, Q) \ \wedge \ P \rightarrow awp(S, Q) \quad (*)$$

- Question: If $\{P\}S\{Q\}$ is valid, is $(*)$ valid?
- 1.

  2.
- Thus, even if program obeys specification, might not be able to prove it b/c loop invariants we use are not strong enough

Example

- Consider the following code:

```
i := 1; sum := 0;
while i ≤ n do [sum ≥ 0] {
    j := 1;
    while j ≤ i do [sum ≥ 0 ∧ j ≥ 0]
       sum := sum + j; j := j + 1
    i := i + 1
}
```

- Show the VC's generated for this program for post-condition $sum \geq 0$ – can it be verified?
- What is the post-condition we need to show for inner loop? $sum \geq 0$

Example, cont.

- Generate VC's for inner loop:

$$\begin{array}{ll} (1) & (sum \geq 0 \wedge j \geq 0 \wedge j > i) \Rightarrow sum \geq 0 \\ (2) & (j \leq i \wedge sum \geq 0 \wedge j \geq 0) \Rightarrow (sum + j \geq 0 \wedge j + 1 \geq 0)) \end{array}$$

- Now, generate VC's for outer loop:

$$\begin{array}{ll} (3) & (i \leq n \wedge sum \geq 0) \Rightarrow (sum \geq 0 \wedge 1 \geq 0) \\ (4) & (i > n \wedge sum \geq 0) \Rightarrow sum \geq 0 \end{array}$$

- Finally, compute awp for outer loop: $(5)\ 0 \geq 0$
- Feed the formula $(1) \wedge (2) \wedge (3) \wedge (4) \wedge (5)$ to SMT solver
- It's valid; hence program is verified!

Example: Variant

- Suppose annotated invariant for inner loop was $sum \geq 0$ instead of $sum \geq 0 \wedge j \geq 0$
- 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 realistics PLs
- Let's enrich IMP with two features, namely functions and pointers
- How to verify programs in this enriched language

IMP with assertions and assumptions

- Before considering functions, we will first add assertions and assumptions to IMP
- The statement $\texttt{assert}(E)$ fails if E evaluates to *false*
- The statement $\texttt{assume}(E)$ tells us that E is *true*

Proof rules for Assert and Assume

- Proof rule for assertions:

$$\frac{P \Rightarrow E}{\vdash \{P\}\ assert(E)\ \{P \wedge E\}}$$

- Proof rule for assumption:

$$\vdash \{P\}\ assume(E)\ \{P \wedge E\}$$

Weakest Precondition for Assert and Assume

- What is wp($\texttt{assert}$(P), Q)?
- What is wp($\texttt{assume}$(P), Q)?
- Given a statement S, how can we generate a statement S' such that $\{P\}S\{Q\}$ is a valid Hoare triple iff $\{true\}S'\{true\}$ is a valid Hoare triple?
- Prove this property!

IMP+: IMP with functions

- IMP+ programs defined according to following grammar:

$$\begin{array}{lcl}
\text{Program } P & := & F^+ \\
\text{Function } F & := & \texttt{function f}(x_1, \ldots, x_n)\ \{S; \texttt{return } e;\} \\
\text{Statement } S & := & y := f(e_1, \ldots, e_n) \mid \ldots
\end{array}$$

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:

```
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:

$$\{\texttt{Pre}\}\ \texttt{Body};\ \texttt{ret} := e\ \{\texttt{Post}\}$$

Verifying Calls

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

$$\{P\}\ y := f(e_1, \ldots, e_n)\ \{Q\}$$

- 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:

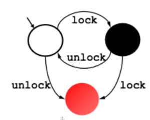
```
assert(Pre[e1/x1, ... en/xn]);
assume(Post[tmp/res,e1/x1, ... en/xn]);
y := tmp;
```

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



"An attempt to re-acquire an acquired lock or release a released lock will cause a deadlock."

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

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 $\texttt{havoc}(\vec{x})$ assigns every variable $\texttt{x} \in \vec{x}$ to an unknown value
- What is $wp(S, \phi)$ 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 v1, v2, ...
```

- 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 $y := foo(x_1, \ldots, x_n)$ 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!

$$\begin{array}{lcl} \text{Program } P & := & F^+ \\ \text{Function } F & := & \texttt{function f}(x_1, \ldots, x_n)\ \{S; \texttt{return } e;\} \\ \text{Statement } S & := & y := *x \mid *x = e \mid \ \ldots \end{array}$$

- 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:

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

- Does this 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:

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

- Proof rule for stores:

$$\vdash \{Q[M\langle x \lhd e\rangle]/M\}\ *x := e\ \{Q\}$$

Revisiting Example

- Let's consider the previous example again

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

- 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!