Overview

- The IMP language considered so far does not have many features of realistic PLs
- Our goal today: 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 `assert(E)` fails if `E` evaluates to `false`
- The statement `assume(E)` tells us that `E` is `true`

Weakest Precondition for Assert and Assume

- What is `wp(assert(P), Q)` ? `P ∧ Q`
- What is `wp(assume(P), Q)` ? `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?
  ```
  assume(P); S; assert(Q)
  ```
- Prove this property!

IMP+: IMP with functions

- IMP+ programs defined according to following grammar:
  ```
  Program P := F+
  Function F := function f(x1, ..., xn) {S; return e;}
  Statement S := y := f(e1, ..., en) | ...
  ```

Proof rules for Assert and Assume

- Proof rule for assertions:
  ```
  P ⇒ E
  ⊢ {P} assert(E) {P ∧ E}
  ```
- Proof rule for assumption:
  ```
  ⊢ {P} assume(E) {P ∧ E}
  ```

1
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 \( \text{get} \) that takes an array \( \text{arr} \) of size \( n \) and index \( i \) and returns the \( i \)'th element
  - Pre-condition: \( 0 \leq i < n \)
  - Post-condition: \( \text{ret} = a[i] \)
- 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:
  \[
  \text{function } f(x_1, \ldots, x_n) \\
  \text{requires(Pre)} \text{ ensures(Post)} \\
  \text{Body} ; \\
  \text{return } e ;
  \]
- Assuming that Post refers to variable \( \text{ret} \), we can verify this contract by checking the validity of this Hoare triple:
  \[
  \{ \text{Pre} \} \text{Body} ; \text{ret} := e \{ \text{Post} \}
  \]

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 \( \text{Pre} \) is satisfied
- But we can also assume that \( f \)'s post-condition \( \text{Post} \) holds after the call – why?
- Thus, we can model the function call as:
  \[
  \text{assert(Pre}[e_1/x_1, \ldots \ e_n/x_n]) ; \\
  \text{assume(Post}[\text{tmp/res}, e_1/x_1, \ldots \ e_n/x_n]) ; \\
  y := \text{tmp} ; \\
  \text{where } \text{tmp} \text{ is a fresh temporary variable.}
  \]

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?

One more complication: Global variables

So far, we assumed function call does not have side effects.

But suppose that function f can modify global variable glob

Is the previous rule still correct?

Counterexample:
\[
glob := 0;
f();
assert(glob = 0);
\]

Havoc

To deal with this difficulty, we introduce a new statement called havoc.

The statement \( \text{havoc}(\vec{x}) \) assigns every variable \( x \in \vec{x} \) 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 \( v_1, v_2, \ldots \)

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

\[
y := \text{foo}(x_1, \ldots, x_n)
\]

as follows:

\[
\text{assert(Pre[el/x_1, \ldots en/x_n]);}
\text{havoc(v_1, \ldots, v_n);}
\text{assume(Post[tmpr/res, el/x_1, \ldots en/x_n]);}
\text{y := tmp;}
\]
IMP with Pointers

- Let’s also add pointers to IMP!

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

- 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:

\[
x := y; \quad \ast y := 3;
\ast x := 2; \quad z := \ast y; \quad \text{assert}(z = 3)
\]

- Does this this assertion hold? No!

- What is the weakest precondition? true

Verification with Pointers

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

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

- 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:

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

- Proof rule for stores:

\[
\vdash \{Q[M(x \triangleleft e)/x]\} \ast x := e \{Q\}
\]

Revisiting Example

- Let’s consider the previous example again

\[
x := y; \quad \ast y := 3;
\ast x := 2; \quad z := \ast y; \quad \text{assert}(z = 3)
\]

- What is the weakest precondition for this code snippet?

Another Example

- Consider the following code snippet

\[
\begin{align*}
&\ast x = 5; \\
&t1 = \ast x; \\
&t2 = \ast y; \\
&\text{assert}(t1+t2=10)
\end{align*}
\]

- What is the weakest precondition for this code snippet?

\[
M(x \triangleleft 5)[x] + M(x \triangleleft 5)[y] = 10
\]

- Can we verify this code if the precondition says \( x = y \)?
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!

Discussion: Size of WPs

- Given a program \( P \), what is the size of its WP? Exponential
- Root cause of the problem is if statement!
  \[
  wp (if (P) then S_1 else S_2, Q) = (P \rightarrow wp (S_1, Q)) \land (\neg P \rightarrow wp (S_2, Q))
  \]
- We are essentially duplicating \( Q \) every time we encounter an if statement \( \Rightarrow \) size of the wp doubles!
- This can lead to scalability problems.

Controlling WP size

- There are techniques that generate much more compact WPs
- Requires converting the program to SSA (static single assignment) form
- In SSA form, every variable is defined exactly once – ensured by introducing multiple “versions” for each variable
- Given program in SSA form, we can generate WPs that are only quadratic in the size of the program
- To learn more, see the paper “Efficient weakest preconditions” by Leino