Floyd-Hoare Logic & Verification Conditions

Ranjit Jhala, UC San Diego

April 16, 2013
A Small Imperative Language

data Var

data Exp

data Pred
A Small Imperative Language

data Com = Asgn Var Expr
    | Seq Com Com
    | If Exp Com Com
    | While Pred Exp Com
    | Skip
Verification Condition Generation

Use the State monad to log individual loop invariant requirements

type VC = State [Pred]  -- validity queries for SMT solver
Top Level Verification Function

The top level verifier, takes:

- **Input**: precondition \( p \), command \( c \) and postcondition \( q \)
- **Output**: True iff \( \{p\} \ c \ \{q\} \) is a valid Hoare-Triple

\[
\text{verify} :: \text{Pred} \rightarrow \text{Com} \rightarrow \text{Pred} \rightarrow \text{Bool}
\]

\[
\text{verify} \ p \ c \ q \ = \ \text{all} \ \text{smtValid} \ \text{queries}
\quad \text{where}
\quad (q', \ \text{conds}) = \ \text{runState} \ \text{(vcgen} \ q \ c) \ \text{[]} \\
\quad \text{queries} \quad = \ p \ \text{"implies"} \ q' : \ \text{conds}
\]
Verification Condition Generator

vcgen :: Pred \to Com \to VC Pred

vcgen (Skip) q
  = return q

vcgen (Asgn x e) q
  = return $ q \text{ `subst`} (x, e)

vcgen (Seq s1 s2) q
  = vcgen s1 =<< vcgen s2 q

vcgen (If b c1 c2) q
  = do q1 \leftarrow vcgen c1 q
     q2 \leftarrow vcgen c2 q
     return $ (b \text{ `implies`} q1) \text{ `And`} (\text{Not} b \text{ `implies`} q2)

vcgen (While i b c) q
  = do q' \leftarrow vcgen c i
     sideCondition $ (i \text{ `And`} b) \text{ `implies`} q'
     sideCondition $ (i \text{ `And`} \text{Not} b) \text{ `implies`} q
     return $ i
vcgen Helper Logs All Side Conditions

sideCond :: Pred -> VC ()
sideCond p = modify $ \conds -> p : conds
Next: Some Examples

Now, let's *use* the above verifier to check some programs
Example 1

Consider the program c defined:

```c
while (x > 0) {
    x = x - 1;
    y = y - 2;
}
```

Lets prove that

```
{x==8 && y==16} c {y == 0}
```
Example 1

Add the pre- and post-condition with assume and assert

```plaintext
assume(x == 8 && y == 16);
while (x > 0) {
    x = x - 1;
    y = y - 2;
}
assert(y == 0);
```

What do we need next?
Example 1: Adding A Loop Invariant

Let's use a placeholder $I$ for the invariant

```c
assume(x == 8 && y == 16);
while (x > 0) {
    invariant(I);
    x = x - 1;
    y = y - 2;
}
assert(y == 0);
```

**Question:** What should $I$ be?

1. **Weak** enough to hold *initially*
2. **Inductive** to prove *preservation*
3. **Strong** enough to prove *goal*
Example 1: Adding A Loop Invariant

Lets try the candidate invariant $y == 2 \times x$

```plaintext
assume(x == 8 && y == 16);
while (x > 0) {
    invariant(y == 2 * x);
    x = x - 1;
    y = y - 2;
}
assert(y == 0);
```

1. Holds initially?
   - SMT-Valid $(x == 8 && y == 16) \Rightarrow (y == 2 \times x)$?
   - [Yes]
Example 1: Adding A Loop Invariant

Let's try the candidate invariant \( y == 2 \times x \)

```plaintext
assume(x == 8 && y == 16);
while (x > 0) {
    invariant(y == 2 \times x);
    x = x - 1;
    y = y - 2;
}
assert(y == 0);
```

2. Preserved?

- SMT-Valid \((y = 2 \times x && x > 0) \Rightarrow (y-2 == 2 \times (x - 1))\)?
- [Yes]
Example 1: Adding A Loop Invariant

Lets try the candidate invariant $y == 2 \times x$

```c
assume(x == 8 && y == 16);
while (x > 0) {
    invariant(y == 2 * x);
    x = x - 1;
    y = y - 2;
}
assert(y == 0);
```

3. Strong Enough To Prove Goal?

- SMT-Valid $(y = 2 \times x \&\& \neg x > 0) \Rightarrow (y == 0)$?
- [No]

Uh oh. Close, but no cigar...
Example 1: Adding A Loop Invariant (Take 2)

Let's try \((y == 2 * x) \&\& (x >= 0)\)

\textbf{assume}(x == 8 \&\& y == 16);
while (x > 0) {
    \textbf{invariant}(y == 2 * x \&\& x >= 0);
    x = x - 1;
    y = y - 2;
}
\textbf{assert}(y == 0);

SMT Valid Check

1. **Initial** \((x == 8 \&\& y == 16) \Rightarrow (y == 2 * x)\)
   ▶ Yes

2. **Preserve** \((y = 2 * x \&\& x > 0) \Rightarrow (y-2 == 2 * (x - 1))\)
   ▶ Yes

3. **Goal** \((y = 2 * x \&\& x >=0 \&\& !x > 0) \Rightarrow (y == 0)\)
   ▶ Yes
Example 2

```plaintext
assume(n > 0);
var k = 0;
var r = 0;
var s = 1;
while (k != n) {
    invariant(I);
    r = r + s;
    s = s + 2;
    k = k + 1;
}
assert(r == n * n);

Whoa! What's a reasonable invariant I?
```
Example 2

Let's try the obvious thing ... \( r == k \times k \)

```javascript
assume(n > 0);
var k = 0;
var r = 0;
var s = 1;
while (k != n) {
    invariant(r == k * k);
    r = r + s;
    s = s + 2;
    k = k + 1;
}
assert(r == n * n);
```

- **Initial** \((k == 0 && r == 0) \Rightarrow (r == k * k)\) YES
- **Goal** \((r == k * k && k == n) \Rightarrow (r == n * n)\) YES
- **Preserve** \((r== k*k && k != n) \Rightarrow (r + s == (k+1)*(k+1))\) NO!
Finding an invariant that is preserved can be tricky... 
... typically need to strengthen to get preservation 
... that is, to add extra conjuncts
Example 2: Take 2

Strengthen I with facts about s

\[ \begin{align*}
\text{assume}(n > 0); \\
\text{var } k = 0; \\
\text{var } r = 0; \\
\text{var } s = 1; \\
\text{while } (k != n) \{ \\
\quad \text{invariant}(r == k^2 && s == 2k + 1); \\
\quad r = r + s; \\
\quad s = s + 2; \\
\quad k = k + 1; \\
\} \\
\text{assert}(r == n \times n); \\
\end{align*} \]

1. Initial

\[\begin{align*}
\rightarrow (k == 0 && r == 0 && s==1) \Rightarrow (r == k^2 && s == 2k + 1) \\
\rightarrow \text{YES}
\end{align*}\]
Example 2: Take 2

Strengthen I with facts about s

```javascript
assume(n > 0);
var k = 0;
var r = 0;
var s = 1;
while (k != n) {
    invariant(r == k*k && s == 2*k + 1);
    r = r + s;
    s = s + 2;
    k = k + 1;
}
assert(r == n * n);
```

2. Goal

- (r == k*k && s == 2*k + 1 && k == n) ⇒ (r == n*n)
- YES
Example 2: Take 2

Strengthen I with facts about $s$

```plaintext
assume(n > 0);
var k = 0;
var r = 0;
var s = 1;
while (k != n) {
    invariant(r == k*k && s == 2*k + 1);
    r = r + s;
    s = s + 2;
    k = k + 1;
}
assert(r == n * n);
```

3. Preserve

$$ (r == k * k && s == 2 * k + 1 && k != n) $$

$$ => $$

$$ (r + s == (k+1) * (k+1) && s+2 == 2 * (k+1) + 1) $$
Adding Features To IMP

- Functions
- Pointers
IMP + Functions

data Fun = F String [Var] Com

data Com = ...
    | Call Var Fun [Expr]
    | Return Expr

data Pgm = [Fun]
IMP + Functions

A function is a big sequence of Com which does not modify formals

```c
function f(x1,...,xn){
    requires(pre);
    ensures(post);
    body;
    return e;
}
```

Precondition

- Predicate over the **formal** parameters x1,...,xn
- That records **assumption** about inputs

Postcondition

- Predicate over the **formals** and **return value** $\text{result}$
- That records **assertion** about outputs
Modular Verification With Contracts

- Together, pre- and post- conditions called contracts
- We can generate VC (hence, verify) one-function-at-a-time
- Using just contracts for all called functions

Questions

1. How to verify each function with callee contracts?
2. How to verify Call commands?
Verifying A Single Function

To verify a single function

```plaintext
function f(x1,...,xn){
  requires(pre);
  ensures(post);
  body;
  return e;
}
```

we need to just verify the Hoare-triple

```plaintext
{pre} body ; $result := r {post}
```

Exercise How will you handle return sprinkled within body?
Verifying A Single Call Command

To establish a Hoare-triple for a single call command

\{P\}

\begin{align*}
y & := f(e) \\
{Q}\end{align*}

1. We must \textbf{guarantee} that \texttt{pre} (of \(f\)) holds \textit{before} the call
2. We can \textbf{assume} that \texttt{post} (of \(f'\)) holds \textit{after} the call

Hence, the above triple reduces to verifying that

\{P\}

\begin{align*}
& \text{assert} \ (\texttt{pre}[e_1/x_1, \ldots, e_n/x_n]) \ ; \\
& \text{assume} \ (\texttt{post}[e_1/x_1, \ldots, e_n/x_n, \ tmp/\$result]) \ ; \\
& y := \ tmp; \\
{Q}\end{align*}

where \(tmp\) is a fresh temporary variable.
Caller-Callee Contract Duality

Note that at the **callsite** for a function, we

- **assert** the pre-condition
- **assume** the post-condition

while when checking the **callee** we

- **assume** the pre-condition
- **assert** the post-condition

This is key for **modular verification**

- Breaks verification up into pieces matching function abstraction
Example

Consider a function

```java
function binarySearch(a, v){
    requires(sorted(a));
    ensures($result == -1
       || 0 <= $result < a.length && a[$result] == v
    );
    ...
}
```

where we want to verify

```java
assume(sorted(arr));
y = binarySearch(arr, 12);
if (y != -1){
    assert (arr[y] == 12)
    ...
}
```
Example: Precondition VC

Consider a function

```java
function binarySearch(a, v){
    requires(sorted(a));
    ensures($result == -1
             || 0 <= $result < a.length && a[$result] == v);

    ...
}
```

Replace call with `assert` and `assume`

```java
//pre[arr/a, 12/v]
assert(sorted(arr));

//post[arr/a, 12/v, y/$result]
assume(y===-1
       || 0<=y<a.length && a[y] == 12);
```
Example: A Locking Protocol

Figure: Calls to lock and unlock Must Alternate

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

The lock and unlock functions

```cpp
function lock(l){
    assert(l == 0);  //UNLOCKED
    return 1;        //LOCKED
}

function unlock(l){
    assert(l == 1);  //UNLOCKED
    return 0;        //LOCKED
}
```

State of lock encoded in value

What are the contracts? Pretty easy…
Example: A Locking Protocol

The lock and unlock functions with contracts

```plaintext
function lock(l){
    requires(l == 0);
    ensures($result == 1);
    assert(l == 0); //UNLOCKED
    return 1;       //LOCKED
}

function unlock(l){
    requires(l == 1);
    ensures($result == 0);
    assert(l == 1); //UNLOCKED
    return 0;       //LOCKED
}
```
Example: Lock Verification

To verify this program

```plaintext
assume(l == 0);
if (n % 2 == 0) {
    l = lock(l);
}
...
if (n % 2 == 0) {
    l = unlock(l);
}

we just verify

assume(l == 0);
if (n % 2 == 0) {
    assert(l == 0);
    assume(tmpa == 1);
    l = tmpa;
}
```
Adding Features To IMP

- Functions
- Pointers
IMP + Pointers

Let us add references to IMP

data Com = Deref Var Var -- x := *y
  | DerefAsgn Var Expr -- *x := e

We find that our assignment rule does not work with aliasing
Assignments and Aliasing

As $*x$ and $*y$ are aliased, the following is valid

$$\{ x == y \} \quad *x = 5 \quad \{ *x + *y == 10 \}$$
Assignments and Aliasing

In general, for what $P$ is the following valid?

\[
\{P\} \quad \ast x = 5 \quad \{\ast x + \ast y == 10\}
\]

Intuitively, $P$ is something like

\[
\ast y == 5 \quad |\quad x = y
\]

- In the first case, the two sum upto 10.
- In the second case, the aliasing kicks in.
Assignments and Aliasing

In general, for what $P$ is the following valid?

$$\{P\} \quad *x = 5 \quad \{*x + *y == 10\}$$

But the Hoare-rule gives us

$$(*x + *y == 10)[5 / *x]$$

$$== (5 + *y == 10)$$

$$== (*y == 5)$$
Assignments and Aliasing

Uh oh! We lost one case! What happened?!
The substitution \([e/x]\) only works when

- \(x\) is the **only** representation for the value in the predicate

Here, there were **two possible** representations

- \(*x\)
- \(*y\)

and we say *possible* because it depends on the aliasing.

- This is why **aliasing is tricky**
Verification With References

Key idea

Beef up our logic to handle memory as a *monolithic entity*

1. Extend **logic theory** (and SMT solver)
2. Extend **Hoare-Rule**

**Note:** Classical solution due to McCarthy

- It has its issues but thats another story...
A Logic For Modelling References

1. Memory **Variables** $M :: \text{Mem}$
2. **Select** Operator for reading memory $sel$
3. **Update** Operator for writing memory $upd$
4. **Axioms** for reasoning about $sel$ and $upd$

forall $M, A1, A2, V$.

$A1 = A2 \Rightarrow sel(upd(M, A1, V), A2) = V$

forall $M, A1, A2, V$.

$A1 \neq A2 \Rightarrow sel(upd(M, A1, V), A2) = sel(M, A2)$
Updated Hoare-Rule for References

New rule for **deref-read**

\[
\{ B \ [sel(M,y)/x]\} \ x := \ *y \ \{ B \}
\]

New rule for **deref-write**

\[
\{ B \ [upd(M,x,e)/M]\} \ \*x := e \ \{ B \}
\]
Assignments and Aliasing Revisited

In general, for what $P$ is the following valid?

\[
\{P\} \quad *x = 5 \quad \{*x + *y == 10\}
\]

Or rather,

\[
\{P\} \quad *x = 5 \quad \{\text{sel}(M, x) + \text{sel}(M, y) == 10\}
\]

Now, with the new **deref-write** rule $P$ becomes

\[
\text{TODO FIX THIS ~~~~}{.javascript} \quad A = [\text{upd}(M, x, 5)/M] \\
(x + y == 10) = [\text{upd}(M, x, 5)/M] \ (\text{sel}(M, x) + \text{sel}(M, y) == 10) = \\
\text{sel}(\text{upd}(M, x, 5), x) + \text{sel}(\text{upd}(M, x, 5), y) = 10 = 5 + \\
\text{sel}(\text{upd}(M, x, 5), y) = 10 = \text{sel}(\text{upd}(M, x, 5), y) = 5 = (x = y \& \\
5 = 5) \ || \ (x != y \& \text{sel}(M, y) = 5) = x = y || *y = 5 ~~~~
\]

Which is exactly what we wanted!
Deductive Verifiers

We have just scratched the surface

Many *industrial strength* verifiers for real languages

- Why3
- ESC-Java

And these, which you can play with online

- VCC
- Spec# 1 Spec# 2
- Verifast

All very impressive: Try them out and see!

Main hassle: writing invariants, pre and post...