1 Floyd-Hoare Logic Overview

- Floyd invented the inductive assertions method.
- Hoare extended the method, formulating it into a logic of programs.
- Historically, one of the first Hoare logic provers has been the “Stanford Pascal Verifier”
- Gordon’s book has an extensive discussion of this method and provides a simple Hoare-logic prover.
- The PiVC prover from Stanford is a modern version of the same idea
- What has changed: mainly the power of the first-order decision procedures.
- Why study Floyd/Hoare logic now:
  - Defines the semantics of imperative computations
  - Many “tricky” loops (e.g. SRT division to be used in a microprocessor) must be understood “deeply.” That means: find a suitable loop invariant.
  - We will be using pieces of this knowledge for finding out inputs that force a program through specified paths.

2 GCD Loop Invariant

Inputs: $x \in \mathbb{N}$ and $y \in \mathbb{N}$ for $x, y$ in Nat

```
| o L               |
| L true           |
| (x>y) \rightarrow gcd(x,y) = x |
| Y                |
| o A              |
| B                |
| (\leq y) \rightarrow x := x-y \rightarrow L |
| true            |
| o C              |
| \emptyset y := y-x \rightarrow L |
| D                |
```
If we set the loop invariant $gcd(x,y) = gcd(X,Y)$, we can justify it.
Example of a VC:
$gcd(x,y) = gcd(X,Y) \land X > Y \Rightarrow gcd(x,y) = gcd(X - Y,Y)$
Just need to prove that
$gcd(X,Y) = gcd(X - Y,Y)$.

3 Path Justification Example

Find inputs that force the GCD program below to follow the path L,C,D,A,B,Y:
Solution: Walk back the path and compute the formula to be satisfied:
- Must be at the output; hence true at Y
- Back at L, we need $x = y$
- Back at B, we need $x - y = y$ or $x = 2y$
- Back at A, we need $x = 2y \land x > y$ which is $x = 2y$
- Back at D, we need $x = 2(y - x)$ or $3x = 2y$
- This flows back to the input!

4 Some Loop Invariant Examples

These are examples from Dershowitz’s great book “Evolution of Programs” (some of them are from his paper with the same title).

Please note: finding the loop invariant is a guess. Once the LI is found, we check it by using the ideas shown in class. To summarize:
- Let the input block be A
- Let the loop block be B
- Let the output block be C
- Let the loop invariant be LI
- Let pre and post denote the pre/post conditions

Then we will have these Verification Conditions (VCs):
- LI implies (exit-condition implies post)
- LI implies wp(LI,B)
- pre implies wp(LI,A)
4.1 Finding the minimum

Write a loop invariant:

\{ n in Nat \}

\[ z := A[0]; \]
\[ i := 0; \]

while (i != n) do
\[ i := i + 1; \]
\[ if A[i] < z then z := A[i] endif; \]
endwhile;

\{ z <= A[0:n] \}

Notice that i is sweeping up. Thus at any point, \( A[0:i] \) must have been scanned, and \( z \) reflects min up to this point. Thus \( z \leq A[0:i] \).

4.2 Finding the quotient within an \( \epsilon \) bound

\{ 0 < c < d, and e > 0, and all these are real numbers \}

\[ q := 0; \]
\[ y := 1; \]

while (y >= e) do
\[ y := y/2; \]
\[ if (d.(q+y) < c) then q := q+y endif; \]
endwhile;

\{ (c/d - q) < e \}

Notice that \( e \) is quite arbitrary. Thus the answer ought to be somehow reflected into the loop invariant. Also, we are doing a “binary search” in that \( y \) is going through all the reciprocals of powers of 2. Thus inside the loop, the quotient is a “wimpy” approximation of the real, with \( y \) guaranteeing to upper bound any \( e \). Thus, \( (q + y) \geq c/d \).

4.3 Finding the Tournament Min

A tournament is represented in an array. Assume a balanced binary tree representing the tournament. This means we are taking the min over the leaves.

\{ n >= 0 \}

\[ y := n; \]

while (y !=0 ) do
\[ endif; \]
\[ y := y-1; \]
endwhile;
\( A[0] = \min A[n:2n] \)

Each time, we are taking the mins of elements at \( 2y \) and \( 2y-1 \) and putting it into \( y-1 \). This will maintain the invariant
\[
\min A[y : 2y] = \min A[n : 2n]
\]

4.4 Gordon’s book example 28, page 37

The loop invariant at the program point called IIc is of interest. We say “the loop invariant” in the sense that it is the desired one. There are many LIs possible.

How about \( S-R = y.(x-X) \)?

This explains that \( X \) amount of work has been done (\( X \) stair-cases counted). The number of stair-cases remaining to be counted is \( (x-X) \). But we are partially into one stair-case. Thus if you take \( y.(x-X) \) and add \( R \) to it (the partial count), we will have the current \( S \) value.

Work out the details as per lecture-outline.txt in Week3.

5 Locking Protocols

We have been alternating between sequential and parallel programs. The notion of invariants is of course useful in both cases. The transition systems that describe concurrent systems are of course non-deterministic.

I’m following the Herlihy/Shavit notes here.

5.1 Counter class, Fig 2.1

This is obviously not thread safe.

A thread-safe increment method is shown in Figure 2.3 where a lock is acquired before incrementing the value.

This shows one use of the lock class – to provide mutual exclusion. Locks are one mechanism to provide mutex.

The author introduces the notion of events and event precedences. Then intervals, and occurrence numbers for events/intervals and also precedences.

Now the properties of any mutex protocol are:

- Mutex: no overlap in the critical sections
  - For threads \( A \) and \( B \) and integers \( j \) and \( k \),
    \[
    CS^k_A \rightarrow CS^j_B \text{ or } CS^j_B \rightarrow CS^k_A.
    \]
- No deadlock (communal progress). If some thread wants in, some thread gets in. If some thread calls \( \text{acq} \) but never gets the lock, other threads must have completed an infinite num of cs crossings.
- No lockout (individual progress) – implies no deadlock. EVERY call to acquire eventually returns.

We now study different ways of realizing \texttt{acquire} and \texttt{release}
5.2 Fig 2.4, Fig 2.5

Fig 2.4: Declare two flags, one per process. Set own 'interested' flag. While other is interested, wait. This can deadlock if they interleave acq and release. It won’t deadlock if acq/rel calls are scheduled w/o interruption. Mutex is of course guaranteed. The only execution histories that exist are those where the CS intervals don’t overlap. I.e. acq is followed by own release.

Fig 2.5: One victim integer. Store PID in victim (sacrifice). While I’m victim, spin. Here, there is no execution history with a single thread calling acquire followed by itself calling release. All histories are where acq of one is followed by acq of the other.

We can argue that the Fig 2.4 protocol satisfies Mutex. Suppose we have both threads entering. Consider the last such entry. We then have

\[
\begin{align*}
\text{w}_a(f[a]=t) &\rightarrow \text{r}_a(f[b]=f) \rightarrow \text{CS}_a \\
\text{w}_b(f[b]=t) &\rightarrow \text{r}_b(f[a]=f) \rightarrow \text{CS}_b
\end{align*}
\]

We also have

\[
\begin{align*}
\text{r}_b(f[b]=f) &\rightarrow \text{w}_b(f[b]=t) \\
\text{CS}_b &\rightarrow \text{w}_a(f[b]=t) \rightarrow \text{r}_a(f[b]=f) \rightarrow \text{CS}_a
\end{align*}
\]

Assuming a sequential consistent memory, this is not possible. This is possible in x86 (and TSO) for many location types.

5.3 Peterson’s Protocol

Combines the above two versions. Keep 'interested' flags and a victim also. The condition checked is “wait while other interested and I am victim.” That is, the worst of the two blocking conditions are combined.

5.3.1 Mutex satisfied

Suppose not.

If both threads successfully pass the acquire, consider the last such calls of acquire.

We must have

\[
\begin{align*}
\text{w}_a(f[a]=t) &\rightarrow \text{w}_a(v=a) \rightarrow \text{r}_a(f[b]) \rightarrow \text{CS}_a (2.8) \\
\text{w}_b(f[b]=t) &\rightarrow \text{w}_b(v=b) \rightarrow \text{r}_b(f[a]) \rightarrow \text{CS}_a (2.9)
\end{align*}
\]

WLG assume A was the last to write victim. So we have

\[
\begin{align*}
\text{w}_b(v=b) &\rightarrow \text{w}_a(v=a)
\end{align*}
\]

This means that

\[
\text{r}_a(v) \text{ is really } \text{r}_a(v)==A \text{ in (2.8).}
\]

Since A entered the CS, it must be the case that

\[
\text{r}_a(f[b]) \text{ must be } \text{r}_a(f[b])==false.
\]

Then we can obtain from 2.9 etc

\[
\begin{align*}
\text{w}_b(f[b]=t) &\rightarrow \text{w}_b(v=b) \rightarrow \text{w}_a(v=a) \rightarrow \text{r}_a(f[b])==false.
\end{align*}
\]

5.3.2 No lockout

See the book chapter.