Program Analysis

INF4140

18.10.12

Lecture 7

Program Logic (PL)

- PL lets us *express* and *prove* properties about programs
- *Formulas* are on the form

$$\{P\}\ S\ \{Q\}$$

  - S: program statement(s)
  - P and Q: assertions over program states
  - P: Precondition
  - Q: Postcondition

If we can use PL to prove some property of a program, then this property will hold for all executions of the program

PL rules from last week

Sequential composition

$$\frac{\{P\}\ S_1;\ \{R\} \quad \{R\}\ S_2;\ \{Q\}}{\{P\}\ S_1; S_2;\ \{Q\}}$$

Conditional

$$\frac{\{P \land B\}\ S;\ \{Q\} \quad (P \land \neg B) \Rightarrow Q}{\{P\}\ \text{if}\ (B)\ S;\ \{Q\}}$$

- Blue: proof obligations
- **for loop**: exercise 2.22!

Consequence

$$\frac{P' \Rightarrow P \quad \{P\}\ S; \{Q\} \quad Q \Rightarrow Q'}{\{P'\}\ S\ \{Q'\}}$$

while loop

$$\frac{\{I \land B\}\ S;\ \{I\}}{\{I\}\ \text{while}\ (B)\ S;\ \{I \land \neg B\}}$$

the **while** rule needs a *loop invariant*!

While rule

- Cannot control the execution in the same manner as for if statements
  - Cannot tell from the code how many times the loop body will be executed
    $\{y \geq 0\}$ `while (y > 0) y = y - 1;`
  - Cannot speak about the state after the first, second, third iteration
- Solution: Find some assertion *I* that is maintained by the loop body
  - Loop invariant: express properties that are preserved by the loop
- Often hard to find suitable loop invariants
  - This course is *not* an exercise in finding complicated invariants

While rule

$$\frac{\{I \wedge B\}\ S;\ \{I\}}{\{I\}\ \text{while}\ (B)\ S;\ \{I \wedge \neg B\}}$$

Can use this rule to reason about the more general case:

$$\{P\}\ \texttt{while}\,(\texttt{B})\,\texttt{S}\ \{Q\}$$

where

- P need not be the loop invariant
- Q need not match $(I \wedge \neg B)$ syntactically

Combine While rule with Consequence rule to prove:

- **Entry:** $P \Rightarrow I$
- **Loop:** $\{I \wedge B\}\ \texttt{S}\ \{I\}$
- **Exit:** $I \wedge \neg B \Rightarrow Q$

While rule: example

$$\{0 \leq n\}\ k = 0;\ \{k \leq n\}\ \texttt{while}\ (\texttt{k} < \texttt{n})\ \texttt{k} = \texttt{k} + 1;\ \{k == n\}$$

Composition rule splits a proof in two: assignment and loop.
Let $k \leq n$ be the loop invariant

- **Entry:** $k \leq n$ follows from itself
- **Loop:**

$$\frac{k < n \Rightarrow k + 1 \leq n}{\{k \leq n \land k < n\}\ \texttt{k} = \texttt{k} + 1\ \{k \leq n\}}$$

- **Exit:** $(k \leq n \land \neg(k < n)) \Rightarrow k == n$

await statement

$$\frac{\{P \wedge B\}\ \texttt{S;}\ \{Q\}}{\{P\} < \texttt{await (B);S;} > \{Q\}}$$

Remember that we are reasoning about safety properties

- Termination is assumed
- Nothing bad will happen
- The rule does not speak about waiting or progress

Concurrent execution

Assume two statements S_1 og S_2 such that:

$$\{P_1\} < \mathtt{S}_1; > \{Q_1\}$$
$$\{P_2\} < \mathtt{S}_2; > \{Q_2\}$$

First attempt for a `co..oc` rule in PL:

$$\frac{\{P_1\} < \mathtt{S}_1; > \{Q_1\} \qquad \{P_2\} < \mathtt{S}_2; > \{Q_2\}}{\{P_1 \wedge P_2\}\ \mathtt{co}\ < \mathtt{S}_1; > \| < \mathtt{S}_2; >\ \mathtt{oc}\ \{Q_1 \wedge Q_2\}}$$

Example (Problem with this rule)

$$\frac{\begin{array}{c}\{x == 0\} < \mathtt{x} = \mathtt{x} + 1; > \{x == 1\} \\ \{x == 0\} < \mathtt{x} = \mathtt{x} + 2; > \{x == 2\}\end{array}}{\{x == 0\}\ \mathtt{co}\ < \mathtt{x} = \mathtt{x} + 1; > \| < \mathtt{x} = \mathtt{x} + 2; >\ \mathtt{oc}\ \{x == 1 \wedge x == 2\}}$$

but this conclusion is not *true*: the postcondition should be $x == 3$!

Interference problem

$$S_1 : \{x == 0\} < \mathtt{x = x + 1;} > \{x == 1\}$$
$$S_2 : \{x == 0\} < \mathtt{x = x + 2;} > \{x == 2\}$$

- The execution of S_2 interferes with the pre- and postconditions for S_1
  - The assertion $x == 0$ need not hold when S_1 starts execution
- The execution of S_1 interferes with the pre- and postconditions for S_2
  - The assertion $x == 0$ need not hold when S_2 starts execution

Solution: weaken the assertions to account for the other process:

$$S_1 : \{x == 0 \vee x == 2\} < \mathtt{x = x + 1;} > \{x == 1 \vee x == 3\}$$
$$S_2 : \{x == 0 \vee x == 1\} < \mathtt{x = x + 2;} > \{x == 2 \vee x == 3\}$$

Interference problem

Now we can try to apply the rule:

$$\frac{\{x == 0 \lor x == 2\} < \mathtt{x = x + 1;} > \{x == 1 \lor x == 3\} \quad \{x == 0 \lor x == 1\} < \mathtt{x = x + 2;} > \{x == 2 \lor x == 3\}}{\{PRE\}\ \mathtt{co}\ < \mathtt{x = x + 1;} > \| < \mathtt{x = x + 2;} >\ \mathtt{oc}\ \{POST\}}$$

where:

$$\begin{aligned} PRE &: (x == 0 \lor x == 2) \land (x == 0 \lor x == 1) \\ POST &: (x == 1 \lor x == 3) \land (x == 2 \lor x == 3) \end{aligned}$$

which gives:

$$\{x == 0\}\ \mathtt{co}\ < \mathtt{x = x + 1;} > \| < \mathtt{x = x + 2;} >\ \mathtt{oc}\ \{x == 3\}$$

Concurrent execution

Assume $\{P_i\}\ \texttt{S}_\texttt{i}\ \{Q_i\}$ for all $S_1, \ldots, S_n$

$$\frac{\{P_i\}\ \texttt{S}_\texttt{i};\ \{Q_i\} \qquad \text{are } \textit{interference free}}{\{P_1 \wedge \ldots \wedge P_n\}\ \texttt{co}\ \texttt{S}_1; || \ldots || \texttt{S}_\texttt{n};\ \texttt{oc}\ \{Q_1 \wedge \ldots \wedge Q_n\}}$$

- *Critical conditions* are assertions outside critical sections (P_i, Q_i)
- *Interference freedom:* The value of a critical condition is not changed by execution of other processes

Interference freedom

Interference freedom

$$\{C \wedge pre(S)\}\ \mathtt{S}\ \{C\}$$

C: critical condition
S: statement in some other process with precondition $pre(S)$

The critical condition "survives" execution of the other process

$$\frac{\{P_1\}\ \mathtt{S}_1;\ \{Q_1\} \qquad \{P_2\}\ \mathtt{S}_2;\ \{Q_2\}}{\{P_1 \wedge P_2\}\ \mathtt{co}\ \mathtt{S}_1; \|\ \mathtt{S}_2;\ \mathtt{oc}\ \{Q_1 \wedge Q_2\}}$$

Four interference requirements:

$$\begin{array}{ll} \{P_2 \wedge P_1\}\ S_1\ \{P_2\} & \{P_1 \wedge P_2\}\ S_2\ \{P_1\} \\ \{Q_2 \wedge P_1\}\ S_1\ \{Q_2\} & \{Q_1 \wedge P_2\}\ S_2\ \{Q_1\} \end{array}$$

Avoiding interference: Weakening assertions

$$\begin{array}{ll} S_1: & \{x == 0\} < \mathtt{x = x + 1;} > \{x == 1\} \\ S_2: & \{x == 0\} < \mathtt{x = x + 2;} > \{x == 2\} \end{array}$$

Here we have interference, for instance the precondition of S_1 is not maintained by execution of S_2:

$$\{(x == 0) \land (x == 0)\}\ \mathtt{x = x + 2;}\ \{x == 0\}$$

is not true

However, after weakening:

$$\begin{array}{ll} S_1: & \{x == 0 \lor x == 2\} < \mathtt{x = x + 1;} > \{x == 1 \lor x == 3\} \\ S_2: & \{x == 0 \lor x == 1\} < \mathtt{x = x + 2;} > \{x == 2 \lor x == 3\} \end{array}$$

$$\{(x == 0 \lor x == 2) \land (x == 0 \lor x == 1)\}\ \mathtt{x = x + 2}\ \{x == 0 \lor x == 2\}$$

(Correspondingly for the other three critical conditions)

Avoiding interference: Disjoint variables

- *V set:* global variables referred (i.e. read or written) to by a process
- *W set:* global variables written to by a process
- *Reference set:* global variables in a critical condition of one process

No interference if:

- W set of S_1 is disjoint from reference set of S_2
- W set of S_2 is disjoint from reference set of S_1

However, variables in a critical condition of one process will often be among the written variables of another

Avoiding interference: Global invariants

Global invariants are:

- Some condition that only refers to global (shared) variables
- Holds initially
- Preserved by all assignments

We avoid interference if critical conditions are on the form $\{I \land L\}$ where:

- I is a global invariant
- L only refers to local variables of the considered process

Avoiding interference: Synchronization

- Hide critical conditions
- MUTEX to critical sections

$$\mathsf{co} \ldots; S; \ldots \,||\, \ldots; S_1; \{C\} S_2; \ldots \mathsf{oc}$$

S might interfere with C
Hide the critical condition by a critical region:

$$\mathsf{co} \ldots; S; \ldots \,||\, \ldots; < S_1; \{C\} S_2; > \ldots \mathsf{oc}$$

Example: Producer/ consumer synchronization

Let `Producer` be a process that delivers data to a `Consumer` process

$PC : c \leq p \leq c + 1 \wedge (p == c + 1) \Rightarrow (buf == a[p - 1])$

Let PC be a *global invariant* of the program:

```
int buf, p = 0, c = 0;
```

```
process Producer {
  int a[n];
  while (p < n) {
    < await (p == c) ; >
    buf = a[p]
    p = p+1;
  }
}
```

```
process Consumer {
  int b[n];
  while (c < n) {
    < await (p > c) ; >
    b[c] = buf
    c = c+1;
  }
}
```

Example: Producer

Loop invariant of Producer:

$I_P : PC \land p <= n$

```
process Producer {
  int a[n];
  {$I_P$}                      //  entering loop
  while (p < n) {              $\{I_P \land p < n\}$
    < await (p == c); >        $\{I_P \land p < n \land p == c\}$
                               $\{I_P\}_{p \leftarrow p+1, buf \leftarrow a[p]}$
    buf = a[p];                $\{I_P\}_{p \leftarrow p+1}$
    p = p + 1;                 $\{I_P\}$
  } $\{I_P \land \neg(p < n)\}$            //  exit loop
    $\Leftrightarrow \{PC \land p == n\}$
}
```

$\{I_P \land p < n \land p == c\} \Rightarrow \{I_P\}_{p \leftarrow p+1, buf \leftarrow a[p]}$

Example: Consumer

Loop invariant of Consumer:

$I_C : PC \land c <= n \land b[0 : c-1] == a[0 : c-1]$

```
process Consumer {
  int b[n];
  {$I_C$}                       // entering loop
  while (c < n) {               {$I_C \land c < n$}
    < await (p > c) ; >         {$I_C \land c < n \land p > c$}
                                {$I_C$}$_{c \leftarrow c+1, b[c] \leftarrow buf}$
    b[c] = buf;                 {$I_C$}$_{c \leftarrow c+1}$
    c = c + 1;                  {$I_C$}
  } {$I_C \land \neg(c < n)$}        // exit loop
    $\Leftrightarrow$ {$PC \land c == n \land b[0 : c-1] == a[0 : c-1]$}
}
```

$\{I_C \land c < n \land p > c\} \Rightarrow \{I_C\}_{c \leftarrow c+1, b[c] \leftarrow buf}$

Example: Producer/Consumer

The final state of the program satisfies:

$$PC \land p == n \land c == n \land b[0 : c - 1] == a[0 : c - 1]$$

which ensures that all elements in `a` are received and occur in the same order in `b`

Interference freedom is ensured by the global invariant and `await` statements

If we combine the two assertions after the `await` statements, we get:

$$I_P \land p < n \land p == c \land I_C \land c < n \land p > c$$

which gives *false*!
At any time, only one process can be after the await statement!

Monitor Invariant

```
monitor name {
   monitor variable          # shared global variable
   initialization            # for the monitor's procedures
   procedures
}
```

- A monitor invariant (*I*) is used to describe the monitor's inner state
- Express relationship between monitor variables
- Maintained by execution of procedures:
  - Must hold after initialization
  - Must hold when a procedure terminates
  - Must hold when we suspend execution due to a call to `wait`
  - Can assume that the invariant holds *after* `wait` and when a procedure starts
- Should be as *strong* as possible!

Axioms for Signal and Continue (1)

Assume that the monitor invariant I and predicate P *does not* mention `cv`. Then we can set up the following axioms:

$\{I\}$ `wait(cv)` $\{I\}$	
$\{P\}$ `signal(cv)` $\{P\}$	for arbitrary P
$\{P\}$ `signal_all(cv)` $\{P\}$	for arbitrary P

Monitor solution to reader/writer problem

Verification of the invariant over `request_read`

$$I : (nr == 0 \vee nw == 0) \wedge nw \leq 1$$

```
procedure request_read() {
      $\{I\}$
      while (nw > 0) {     $\{I \wedge nw > 0\}$
           $\{I\}$ wait(oktoread); $\{I\}$
      }
      $\{I \wedge nw == 0\}$
      nr = nr + 1;
      $\{I\}$
}
```

$(I \wedge nw > 0) \Rightarrow I$

$(I \wedge nw == 0) \Rightarrow I_{nr \leftarrow (nr+1)}$

Axioms for Signal and Continue (2)

Assume that the invariant can mention the number of processes in the queue to a condition variable.

- Let $\#cv$ be the number of processes waiting in the queue to cv.
- The test $empty(cv)$ is then identical to $\#cv == 0$

wait(cv) is *modelled* as an extension of the queue followed by processor release:

$$wait(cv) : \{?\}\ \#cv = \#cv + 1; \{I\}\ sleep\{I\}$$

by assignment axiom:

$$wait(cv) : \{I_{\#cv \leftarrow (\#cv+1)}\}\ \#cv = \#cv + 1; \{I\}\ sleep\{I\}$$

Axioms for Signal and Continue (3)

signal(*cv*) can be modelled as a reduction of the queue, if the queue is not empty:

$$signal(cv) : \{?\}\ if\ (\#cv \mathrel{!=} 0)\ \#cv = \#cv - 1\ \{P\}$$

$$\begin{aligned} signal(cv):\quad & \{((\#cv == 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow (\#cv-1)})\} \\ & if\ (\#cv \mathrel{!=} 0)\ \#cv = \#cv - 1 \\ & \{P\} \end{aligned}$$

- *signal_all*(*cv*): $\{P_{\#cv \leftarrow 0}\}\ \ \#cv = 0\ \{P\}$

Axioms for Signal and Continue (4)

Together this gives:

$$\{I_{\#cv \leftarrow (\#cv+1)}\}\ \texttt{wait(cv)}\ \{I\}$$
$$\{((\#cv == 0) \Rightarrow P) \wedge ((\#cv \neq 0) \Rightarrow P_{\#cv \leftarrow (\#cv-1)})\}\ \texttt{signal(cv)}\ \{P\}$$
$$\{P_{\#cv \leftarrow 0}\}\ \texttt{signal\_all(cv)}\ \{P\}$$

If we know that $\#cv \neq 0$ whenever we signal, then the axiom for `signal(cv)` be simplified to:

$$\{P_{\#cv \leftarrow (\#cv-1)}\}\ \texttt{signal(cv)}\ \{P\}$$

Note! *$\#cv$ is not allowed in statements!*

- Only used for reasoning

Example: FIFO semaphore verification (1)

```
monitor FIFO_semaphore {
   int s = 0;                    # value of semaphore
   cond pos;                     # signalled only when #pos>0

    procedure Psem() {               procedure Vsem() {
       if (s==0)                        if empty(pos)
           wait(pos);                      s=s+1;
       else                             else
          s = s-1;                         signal(pos);
    }                                }
}
```

Consider the following monitor invariant:

$$s \geq 0 \wedge (s > 0 \Rightarrow \#pos == 0)$$

No process is waiting if the semaphore value is positive

Example: FIFO semaphore verification (2)

$$I: \quad s \geq 0 \land (s > 0 \Rightarrow \#pos == 0)$$

```
procedure Psem() {
{$I$}
   if (s==0) {$I \land s == 0$}
      {$I_{\#pos \leftarrow (\#pos+1)}$} wait(pos); {$I$}
   else {$I \land s \neq 0$}
      {$I_{s \leftarrow (s-1)}$} s = s-1; {$I$}
{$I$}
}
```

Example: FIFO semaphore verification (3)

$$I: \quad s \geq 0 \land (s > 0 \Rightarrow \#pos == 0)$$

This gives two proof obligations:
If branch:

$$\begin{aligned}
(I \land s == 0) &\Rightarrow I_{\#pos \leftarrow (\#pos+1)} \\
s == 0 &\Rightarrow s \geq 0 \land (s > 0 \Rightarrow \#pos + 1 == 0) \\
s == 0 &\Rightarrow s \geq 0
\end{aligned}$$

Else branch:

$$\begin{aligned}
(I \land s \neq 0) &\Rightarrow I_{s \leftarrow (s-1)} \\
(s > 0 \land \#pos == 0) &\Rightarrow s - 1 \geq 0 \land (s - 1 \geq 0 \Rightarrow \#pos == 0) \\
(s > 0 \land \#pos == 0) &\Rightarrow s > 0 \land \#pos == 0
\end{aligned}$$

Example: FIFO semaphore verification (4)

$$I: \quad s \geq 0 \land (s > 0 \Rightarrow \#pos == 0)$$

```
procedure Vsem() {
{$I$}
   if empty(pos) {$I \land \#pos == 0$}
       {$I_{s \leftarrow (s+1)}$}s=s+1; {$I$}
   else {$I \land \#pos \neq 0$}
      {$I_{\#pos \leftarrow (\#pos-1)}$} signal(pos); {$I$}
{$I$}
}
```

Example: FIFO semaphore verification (5)

$$I: \quad s \geq 0 \land (s > 0 \Rightarrow \#pos == 0)$$

As above, this gives two proof obligations:
If branch:

$$\begin{aligned}
(I \land \#pos == 0) &\Rightarrow I_{s \leftarrow (s+1)} \\
(s \geq 0 \land \#pos == 0) &\Rightarrow s+1 \geq 0 \land (s+1 > 0 \Rightarrow \#pos == 0) \\
(s \geq 0 \land \#pos == 0) &\Rightarrow s+1 \geq 0 \land \#pos == 0
\end{aligned}$$

Else branch:

$$\begin{aligned}
(I \land \#pos \neq 0) &\Rightarrow I_{\#pos \leftarrow (\#pos-1)} \\
(s == 0 \land \#pos \neq 0) &\Rightarrow s \geq 0 \land (s > 0 \Rightarrow \#pos - 1 == 0) \\
s == 0 &\Rightarrow s \geq 0
\end{aligned}$$