Introduction to Formal Methods

Lecture 10

Verifying Programs with Arrays & Dynamic Allocation

Hossein Hojjat & Fatemeh Ghassemi

October 23, 2018

Weakest Precondition Rules: Summary

c	$\mathsf{wp}(c, Q)$
$x := e$	$Q[x \mapsto e]$
$\mathsf{assume}(b)$	$b \rightarrow Q$
$\mathsf{assert}(b)$	$\mathsf{wp}(b \wedge Q)$
$\mathsf{havoc}(x)$	$\forall y. Q[x \mapsto y]$
$c_1; c_2$	$\mathsf{wp}(c_1, \mathsf{wp}(c_2, Q))$
$\texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2$	$b \rightarrow \mathsf{wp}(c_1, Q) \wedge \neg b \rightarrow \mathsf{wp}(c_2, Q)$
$\texttt{while}\ b\ \texttt{do}\ c$	$I \wedge \forall \vec{y}. \Big((I \wedge b \rightarrow \mathsf{wp}(c, I)) \wedge (I \wedge \neg b \rightarrow Q) \Big)[\vec{x} \mapsto \vec{y}]$ ($\vec{x}$ are variables modified in c and I is the loop invariant)

Problem with Arrays

```
a[k]=1;
a[j]=2;
x=a[k]+a[j];
{x=3}
```

```
a[k]=1;
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

Problem with Arrays

- Now what? Can we use the standard rule for assignment?

$$\mathsf{wp}(x := e, C) = C[x \mapsto e]$$

```
a[k]=1;
a[j]=2;
x=a[k]+a[j];
{x=3}
```

⇒

```
a[k]=1;
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

⇒

Problem with Arrays

- Now what? Can we use the standard rule for assignment?

$$\mathsf{wp}(x := e, C) = C[x \mapsto e]$$

```
a[k]=1;
a[j]=2;
x=a[k]+a[j];
{x=3}
```

→

```
a[k]=1;
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

→

```
{1+2=3} = {true}
a[k]=1;
{a[k]+2=3}
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

Problem with Arrays

- Now what? Can we use the standard rule for assignment?

$$\text{wp}(x := e, C) = C[x \mapsto e]$$

```
a[k]=1;
a[j]=2;
x=a[k]+a[j];
{x=3}
```

⇒

```
a[k]=1;
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

⇒

```
{1+2=3} = {true}
a[k]=1;
{a[k]+2=3}
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

What if $k = j$?

Problem with Arrays

- Now what? Can we use the standard rule for assignment?

$$\text{wp}(x := e, C) = C[x \mapsto e]$$

```
a[k]=1;
a[j]=2;
x=a[k]+a[j];
{x=3}
```

⇒

```
a[k]=1;
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

⇒

```
{1+2=3} = {true}
a[k]=1;
{a[k]+2=3}
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

What if $k = j$?

Problem with Arrays

- Naïve array assignment axiom does not work

$$\{Q[A[e_1] \mapsto e_2]\} \;\; A[e_1] := e_2 \;\; \{Q\}$$

- Changes to $A[i]$ may also change $A[j]$, $A[k]$, ...
  - (since i might equal j, k, ...)
- **Solution:** enrich the assertion language with expressions $A\{e_1 \mapsto e_2\}$
- Meaning: the array equal to A except that index e_1 maps to value e_2

$$A\{e_1 \mapsto e_2\}[i] = \begin{cases} A[i] & \text{if } i \neq e_1 \\ e_2 & \text{if } i = e_1 \end{cases}$$

Assignment Rule with Theory of Arrays

$$\frac{}{\vdash \{Q[A \mapsto A\{i \mapsto e\}]\}\ A[i] := e\ \{Q\}}$$

```
a[k]=1;
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

→

```
{k≠j}
{a{k↦1}{j↦2}[k]+a{k↦1}{j↦2}[j]=3}
a[k]=1;
{a{j↦2}[k]+a{j↦2}[j]=3}
a[j]=2;
{a[k]+a[j]=3}
x=a[k]+a[j];
{x=3}
```

Exercise

Prove the array sum is correct

```
{n≥0}
j = 0;
s = 0;
while (j<n) do{
  s = s + a[j];
  j = j + 1;
}
```

$\{ s = \sum_{0 \leq i < n} a[i] \}$

$$\frac{A \to I \qquad \vdash \{b \wedge I\}\ c\ \{I\} \qquad I \wedge \neg b \to B}{\vdash\ \{A\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{B\}}$$

Exercise

Prove the array sum is correct

```
{n≥0}
j = 0;
s = 0;
while (j<n) do{
  s = s + a[j];
  j = j + 1;
}
{ s = ∑_{0≤i<n} a[i] }
```

$$\frac{A \to I \qquad \vdash \{b \land I\}\ c\ \{I\} \qquad I \land \neg b \to B}{\vdash\ \{A\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{B\}}$$

Choose invariant $(s = \sum_{0 \le i < j} a[i]) \land 0 \le j \le n$

Step 1. Prove invariant is maintained throughout the loop

$$\{j < n \land (s = \sum_{0 \le i < j} a[i]) \land 0 \le j \le n\}$$

```
s = s + a[j]; j = j + 1
```

$$\{(s = \sum_{0 \le i < j} a[i]) \land 0 \le j \le n\}$$

Exercise

Prove the array sum is correct

```
{n≥0}
j = 0;
s = 0;
while (j<n) do{
  s = s + a[j];
  j = j + 1;
}
{ s = ∑_{0≤i<n} a[i] }
```

$$\frac{A \to I \qquad \vdash \{b \wedge I\}\ c\ \{I\} \qquad I \wedge \neg b \to B}{\vdash\ \{A\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{B\}}$$

Choose invariant $(s = \sum_{0 \leq i < j} a[i]) \wedge 0 \leq j \leq n$

Step 2. Prove invariant is initially *true*

$$\{n \geq 0\}$$

$$\texttt{j = 0; s = 0}$$

$$\{(s = \sum_{0 \leq i < j} a[i]) \wedge 0 \leq j \leq n\}$$

Exercise

Prove the array sum is correct

```
{n≥0}
j = 0;
s = 0;
while (j<n) do{
  s = s + a[j];
  j = j + 1;
}
{ s = ∑_{0≤i<n} a[i] }
```

$$\frac{A \to I \qquad \vdash \{b \wedge I\}\ c\ \{I\} \qquad I \wedge \neg b \to B}{\vdash\ \{A\}\ \texttt{while}\ b\ \texttt{do}\ c\ \{B\}}$$

Choose invariant $(s = \sum_{0 \le i < j} a[i]) \wedge 0 \le j \le n$

Step 3. Prove invariant and exit condition implies postcondition

$$\left((s = \sum_{0 \le i < j} a[i]) \wedge 0 \le j \le n \wedge j \ge n\right) \to$$

$$s = \sum_{0 \le i < n} a[i]$$

Proof Obligations

Step 1. Prove invariant is maintained throughout the loop

$$\{(s + a[j] = \sum_{0 \le i < j+1} a[i]) \wedge 0 \le j + 1 \le n\}$$ (by assignment rule)

```
s = s + a[j]
```

$$\{(s = \sum_{0 \le i < j+1} a[i]) \wedge 0 \le j + 1 \le n\}$$ (by assignment rule)

```
j = j + 1
```

$$\{(s = \sum_{0 \le i < j} a[i]) \wedge 0 \le j \le n\}$$

Need to show:

$$(0 \le j \le n \wedge (s = \textstyle\sum_{0 \le i < j} a[i]) \wedge j < n) \rightarrow$$
$$(0 \le j + 1 \le n \wedge (s + a[j] = \textstyle\sum_{0 \le i < j+1} a[i]))$$

Proof Obligations

Step 2. Prove invariant is initially *true*

$$\{(0 = \sum_{0 \le i < 0} a[i]) \land 0 \le 0 \le n\}$$ (by assignment rule)

```
j = 0
```

$$\{(0 = \sum_{0 \le i < j} a[i]) \land 0 \le j \le n\}$$ (by assignment rule)

```
s = 0
```

$$\{(s = \sum_{0 \le i < j} a[i]) \land 0 \le j \le n\}$$

Need to show:

$$(n \ge 0) \to (0 = \textstyle\sum_{0 \le i < 0} a[i]) \land 0 \le 0 \le n$$

Proof Obligations

Step 3. Prove invariant and exit condition implies postcondition

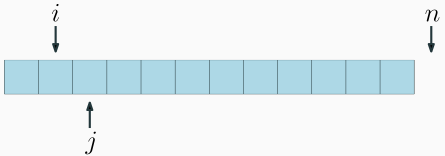
$$((s = \sum_{0 \leq i < j} a[i]) \land 0 \leq j \leq n \land j \geq n) \rightarrow$$
$$(s = \sum_{0 \leq i < n} a[i])$$

Exercise

Consider the following program:

```
{0≤i<n}
j = i+1 ;
while (j<n) {
    a[i] = max(a[i],a[j]);
    j = j+1;
}
```

$\{ \forall_{i\leq k<n} \; a_0[k] \leq a[i] \}$



Is the following a loop invariant?

$$\{\forall_{i\leq k<j} \; a_0[k] \leq a[i] \wedge 0 \leq j \leq n\}$$

(a_0 is the initial array)

Invariant Proof

Prove invariant is maintained throughout the loop

$\{\forall_{i \le k < j+1}\ a_0[k] \le \mathsf{max}(a[i], a[j]) \land 0 \le j+1 \le n\}$

$\{\forall_{i \le k < j+1}\ a_0[k] \le a\{i \mapsto \mathsf{max}(a[i], a[j])\}[i] \land 0 \le j+1 \le n\}$ (by array assignment)

```
a[i] = max(a[i],a[j])
```

$\{\forall_{i \le k < j+1}\ a_0[k] \le a[i] \land 0 \le j+1 \le n\}$ (by assignment)

```
j = j + 1
```

$\{\forall_{i \le k < j}\ a_0[k] \le a[i] \land 0 \le j \le n\}$

Need to show:

$$(\forall_{i \le k < j}\ a_0[k] \le a[i] \land j < n) \to$$
$$(\forall_{i \le k < j+1}\ a_0[k] \le \mathsf{max}(a[i], a[j]) \land 0 \le j+1 \le n)$$

We don't know that $a_0[j] \le \mathsf{max}(a[i], a[j])$!

Conjoin a new constraint $(\forall_{j \le k < n}\ a[k] = a_0[k]) \land i < j$

Array Bounds

- Check if an array index is within the bounds of the array

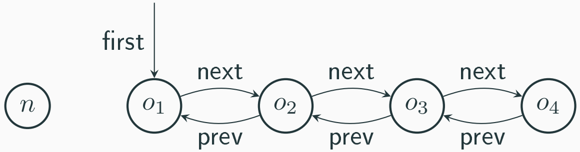
x := a[i]

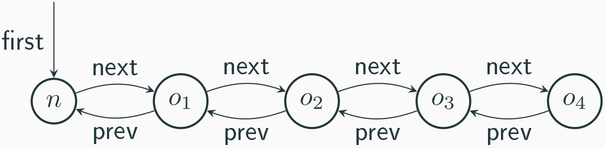
```
assert(0 ≤ i ∧ i <  size(a))
x  :=  a[i]
```

a[i] := x

```
assert(0 ≤ i ∧ i <  size(a))
a  :=  a[i↦x]
```

Linked List Example

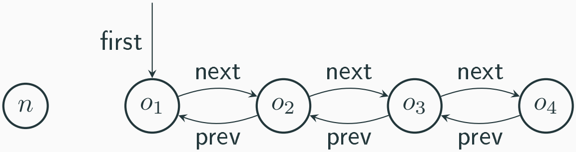





```
insert(first,n):
if (first == null)
  first = n;
else {
  n.next = first;
  first.prev = n;
  first = n;
}
```

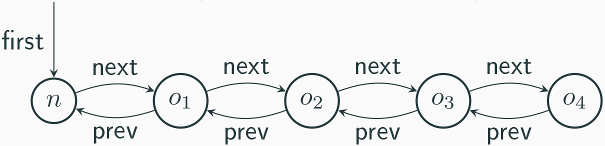
How to verify such code?

Linked List Example



$$\text{next} = \{(o_1, o_2), (o_2, o_3), (o_3, o_4)\}$$

$$\text{prev} = \{(o_2, o_1), (o_3, o_2), (o_4, o_3)\}$$



$$\text{next} = \{(o_1, o_2), (o_2, o_3), (o_3, o_4), (n, o_1)\}$$

$$\text{prev} = \{(o_2, o_1), (o_3, o_2), (o_4, o_3), (o_1, n)\}$$

```
insert(first,n):
if (first == null)
  first = n;
else {
  n.next = first;
  first.prev = n;
  first = n;
}
```

Change of relations (partial functions):

$$\text{next}' = \text{next} \cup \{(n, o_1)\}$$

$$\text{prev}' = \text{prev} \cup \{(o_1, n)\}$$

using assignments:

$$\text{next} = \text{next}[n \mapsto \text{first}]$$

$$\text{prev} = \text{prev}[\text{first} \mapsto n]$$

Reading Fields

- Statement

```
y = x.next
```

- Computes the value of `y` simply as

```
y = next(x)
```

- We should not de-reference `null`

```
assert(x ≠ null);
y = next(x)
```

- We assume that the type system ensures that if `x` is not `null` then the value `next(x)` is defined
- Otherwise, we could add the corresponding check

```
assert(x ∈ dom(next));
y = next(x)
```

Writing Fields

- We represent each field using a global partial function
- Statement

```
x.next = y
```

- Changes heap according to this update:

$$\texttt{next}' = \texttt{next[x}\mapsto\texttt{y]}$$

- which is a notation that expands to:

$$\mathsf{next}' = \{(u,v) | (u = x \land v = y) \lor (u \neq x \land (u,v) \in \mathsf{next})\}$$

- We should not assign fields of `null` so we also add this check

```
assert(x≠null);
next' = next[x↦y]
```

Why we Need Functions?

- Say we have `x.f` and `y.f` in the program
- Why not replace them simply with fresh variables x_f and y_f?
- Does this assertion hold for two distinct values `p`, `q`?

```
var xf = ...
var yf = ...
xf = p
yf = q
assert(xf == p)
```

- Yes. The value of `xf` is still `p`

Why we Need Functions?

- Say we have `x.f` and `y.f` in the program
- Why not replace them simply with fresh variables x_f and y_f?
- Does this assertion hold for two distinct values p, q?

```
var xf = ...
var yf = ...
xf = p
yf = q
assert(xf == p)
```

- Yes. The value of `xf` is still p
- Does this assertion hold?

```
...
x.f = p
y.f = q
assert(x.f == p)
```

Why we Need Functions?

- Say we have `x.f` and `y.f` in the program
- Why not replace them simply with fresh variables x_f and y_f?
- Does this assertion hold for two distinct values p, q?

```
var xf = ...
var yf = ...
xf = p
yf = q
assert(xf == p)
```

- Yes. The value of `xf` is still `p`
- Does this assertion hold?

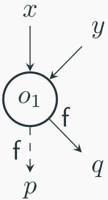
```
...
x.f = p
y.f = q
assert(x.f == p)
```

- Depends.

Aliasing

Does the assertion hold in this case:

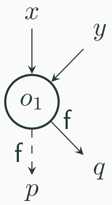
```
x = y
x.f = p
y.f = q
assert(x.f == p)
```



Aliasing

Does the assertion hold in this case:

```
x = y
x.f = p
y.f = q
assert(x.f == p)
```

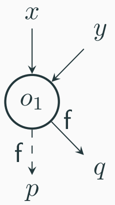


- No! `y` and `x` are aliased references, denote the same object
- Even though left hand sides `x.f` and `y.f` look different, they interfere

Aliasing

Does the assertion hold in this case:

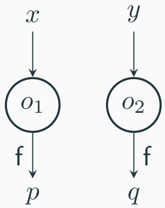
```
x = y
x.f = p
y.f = q
assert(x.f == p)
```



- No! y and x are aliased references, denote the same object
- Even though left hand sides x.f and y.f look different, they interfere

Does it hold in this case?

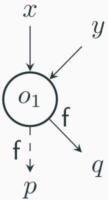
```
assume(x≠y)
x.f = p
y.f = q
assert(x.f == p)
```



Aliasing

Does the assertion hold in this case:

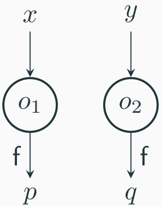
```
x = y
x.f = p
y.f = q
assert(x.f == p)
```



- No! y and x are aliased references, denote the same object
- Even though left hand sides x.f and y.f look different, they interfere

Does it hold in this case?

```
assume(x≠y)
x.f = p
y.f = q
assert(x.f == p)
```



Yes!

Example: wp Computation

- Recall $\mathsf{wp}(x := e, Q) = Q[x \mapsto e]$ (substitution)
- Ignoring null checks, we have the following:

$$
\begin{aligned}
&\mathsf{wp}(x.f := p; y.f := q \ , \ x.f = p) = \\
&\mathsf{wp}(f = f[x \mapsto p]; f = f[y \mapsto q] \ , \ f(x) = p) = \\
&\mathsf{wp}(f = f[x \mapsto p] \ , \ (f[y \mapsto q])(x) = p) = \\
&((f[x \mapsto p])[y \mapsto q])(x) = p
\end{aligned}
$$

- If h is a function then

$$h[a \mapsto b](u) = v \Leftrightarrow (u = a \wedge v = b) \vee (u \neq a \wedge v = h(u))$$

- Thus

$$
\begin{aligned}
&((f[x \mapsto p])[y \mapsto q])(x) = p \\
\Leftrightarrow &(x = y \wedge p = q) \vee (x \neq y \wedge p = (f[x \mapsto p])(x)) \\
\Leftrightarrow &(x = y \wedge p = q) \vee (x \neq y \wedge p = p) \\
\Leftrightarrow &(x = y \wedge p = q) \vee x \neq y
\end{aligned}
$$

Characterizes precisely the weakest condition under which assertion holds

Exercise

```
class C {
  var f: C
}
```

- Translate into checks and function updates

```
x.f.f = z.f + y.f.f.f
```

Exercise

```
class C {
  var f: C
}
```

- Translate into checks and function updates

```
x.f.f = z.f + y.f.f.f
```

Solution.

$$\texttt{assume(z}\neq\texttt{null)}$$
$$\texttt{assume(y}\neq\texttt{null)}$$
$$\texttt{assume(f(y)}\neq\texttt{null)}$$
$$\texttt{assume(f(f(y))}\neq\texttt{null)}$$
$$\texttt{assume(f(x)}\neq\texttt{null)}$$
$$\texttt{f := f}\Big[\ \texttt{f(x)} \mapsto \big(\texttt{f(z) + f(f(f(y)))}\big)\ \Big]$$

Modeling Dynamic Allocation

- Can we prove this?

```
x = new  C();
y = new  C();
assert(x≠y);
```

Modeling Dynamic Allocation

- Can we prove this?

```
x = new  C();
y = new  C();
assert(x≠y);
```

- Can we introduce global variables and assumptions that correctly describe fresh objects?

Modeling Dynamic Allocation

- Can we prove this?

```
x = new  C();
y = new  C();
assert(x≠y);
```

- Can we introduce global variables and assumptions that correctly describe fresh objects?
- Global set `alloc` denotes objects allocated so far

```
x = new  C();
```

- denotes (for now):

```
havoc(x);
assume(x ∉ alloc);
alloc = alloc ∪ {x}
```

alloc Set

Original program

```
x = new C();
y = new C();
assert(x≠y);
```

Becomes

```
havoc(x);
assume(x ∉ alloc)
alloc = alloc ∪ {x};
havoc(y);
assume(y ∉ alloc);
alloc = alloc ∪ {y};
assert(x≠y);
```

Renaming variables we obtain:

```
havoc(x);
assume(x ∉ alloc)
alloc1 = alloc ∪ {x};
havoc(y);
assume(y ∉ alloc1);
alloc2 = alloc1 ∪ {y};
assert(x≠y);
```

Assertion holds because

$$(\mathtt{alloc}_1 = \mathtt{alloc} \cup \{\mathtt{x}\}) \wedge (\mathtt{y} \notin \mathtt{alloc}_1) \rightarrow \mathtt{x} \neq \mathtt{y}$$