HOARE LOGIC

VERIFICATION CONDITION GENERATION

- We have already seen that the weakest pre-condition operator can be used to prove Hoare Triples:
  - $\{P\}c\{Q\}$ iff $P \Rightarrow wp(Q, c)$
- Finding exact *wp* for loops is hard. We will instead use the loop invariant as an approximate *wp*.
  - $awp(Q, \text{while(F)@I do c}) = \text{I}$
  - Does this always hold?
- Also need to show that following side-conditions hold:
  - $\{\text{I} \wedge \text{F}\}\text{c}\{\text{I}\}$
  - $\text{I} \wedge \neg\text{F} \Rightarrow Q$

RELATION BETWEEN AWP AND WP

- Let us formally define *awp*:
  - $\forall \sigma \in awp(Q, c) \,.\, \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma', skip) \rightarrow \sigma' \in Q$
  - Homework: Prove that this holds for $awp(Q,$ while(F)@I do c) = I, when the side-conditions hold.
- We defined $wp(Q, c) \triangleq \{\sigma \mid \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma', \text{skip}) \rightarrow \sigma' \in Q\}$
  - $awp(Q, c) \subseteq wp(Q, c)$

RELATION BETWEEN AWP AND WP

- Let us formally define *awp*:
  - $\forall \sigma \in awp(Q, c) \, . \, \forall \sigma' \, . \, (\sigma, c) \hookrightarrow^* (\sigma', skip) \rightarrow \sigma' \in Q$
  - Homework: Prove that this holds for $awp(Q,$ while(F)@I do c) = I, when the side-conditions hold.
- We defined $wp(Q, c) \triangleq \{\sigma \mid \forall \sigma' \, . \, (\sigma, c) \hookrightarrow^* (\sigma', \text{skip}) \rightarrow \sigma' \in Q\}$
  - $awp(Q, c) \subseteq wp(Q, c)$
- $awp(i \geq 0,$ while(i < n)@(i >= 0) do i := i+1;) = ???

RELATION BETWEEN AWP AND WP

- Let us formally define *awp*:
  - $\forall \sigma \in awp(Q, c) \,.\, \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma', skip) \rightarrow \sigma' \in Q$
  - Homework: Prove that this holds for $awp(Q,$ while(F)@I do c) = I, when the side-conditions hold.
- We defined $wp(Q, c) \triangleq \{\sigma \mid \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma', \text{skip}) \rightarrow \sigma' \in Q\}$
  - $awp(Q, c) \subseteq wp(Q, c)$
- $awp(i \geq 0,$ while(i < n)@(i >= 0) do i := i+1;) = $i \geq 0$

RELATION BETWEEN AWP AND WP

- Let us formally define *awp*:
  - $\forall \sigma \in awp(Q, c) \,.\, \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma', skip) \rightarrow \sigma' \in Q$
  - Homework: Prove that this holds for $awp(Q,$ while(F)@I do c) = I, when the side-conditions hold.
- We defined $wp(Q, c) \triangleq \{\sigma \mid \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma',$ skip$) \rightarrow \sigma' \in Q\}$
  - $awp(Q, c) \subseteq wp(Q, c)$
- $awp(i \geq 0,$ while(i < n)@(i >= 0) do i := i+1;) = $i \geq 0$
  - $wp(i \geq 0,$ while(i < n)@(i >= 0) do i := i+1;) = ???

RELATION BETWEEN AWP AND WP

- Let us formally define *awp*:
  - $\forall \sigma \in awp(Q, c) \,.\, \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma', skip) \rightarrow \sigma' \in Q$
  - Homework: Prove that this holds for $awp(Q,$ while(F)@I do c) = I, when the side-conditions hold.
- We defined $wp(Q, c) \triangleq \{\sigma \mid \forall \sigma' \,.\, (\sigma, c) \hookrightarrow^* (\sigma', \text{skip}) \rightarrow \sigma' \in Q\}$
  - $awp(Q, c) \subseteq wp(Q, c)$
- $awp(i \geq 0,$ while(i < n)@(i >= 0) do i := i+1;) = $i \geq 0$
  - $wp(i \geq 0,$ while(i < n)@(i >= 0) do i := i+1;) =$n \geq 0 \vee i \geq 0$

VC GENERATION - I

- We define $VC(Q, c)$ to collect the side-conditions needed for verifying that Q holds after execution of c.
- For while(F)@I do c, there are two side-conditions:
  - $\{I \wedge F\}c\{I\}$
  - $I \wedge \neg F \Rightarrow Q$
- $\{I \wedge F\}c\{I\}$ is valid if $I \wedge F \Rightarrow awp(I, c)$.
  - c may contain loops, so we also need to consider $VC(I, c)$.
- Hence,
  $VC(Q, \text{while(F)@I do c}) \triangleq (I \wedge \neg F \Rightarrow Q) \wedge (I \wedge F \Rightarrow awp(I, c)) \wedge VC(I, c)$

VC GENERATION - II

- $VC(Q, \text{x:=e}) \triangleq true$
  - Also defined as *true* for all simple program commands (assert, assume, havoc).
- $VC(Q, c_1; c_2) \triangleq ???$

VC GENERATION - II

- $VC(Q, \text{x:=e}) \triangleq true$
  - Also defined as *true* for all simple program commands (assert, assume, havoc).
- $VC(Q, c_1; c_2) \triangleq VC(Q, c_2) \wedge VC(awp(Q, c_2), c_1)$

VC GENERATION - II

- $VC(Q, \text{x:=e}) \triangleq true$
  - Also defined as *true* for all simple program commands (assert, assume, havoc).
- $VC(Q, c_1; c_2) \triangleq VC(Q, c_2) \wedge VC(awp(Q, c_2), c_1)$
- $VC(Q, \text{if(F) then c}_1 \text{ else c}_2) \triangleq ???$

VC GENERATION - II

- $VC(Q, \text{x:=e}) \triangleq true$
  - Also defined as *true* for all simple program commands (assert, assume, havoc).
- $VC(Q, c_1; c_2) \triangleq VC(Q, c_2) \land VC(awp(Q, c_2), c_1)$
- $VC(Q, \text{if(F) then c}_1 \text{ else c}_2) \triangleq VC(Q, \text{c}_1) \land VC(Q, \text{c}_2)$

VC GENERATION - III

- $awp(Q, c) \triangleq wp(Q, c)$ except for while loops, for which $awp(Q,$ while(F)@I do c$) =$ I.
- Putting it all together, $\{P\}c\{Q\}$ is valid if the following FOL formula is valid:
  - $(P \rightarrow awp(Q, c)) \wedge VC(Q, c)$

RELATION BETWEEN AWP AND HOARE TRIPLES

- What is the relation between $awp(Q, c)$ and validity of the Hoare Triple $\{P\}c\{Q\}$?
  - Is it possible that $P \rightarrow awp(Q, c)$ is valid and $\{P\}c\{Q\}$ is not valid?
  - Is it possible that $\{P\}c\{Q\}$ is valid and $\neg(P \rightarrow awp(Q, c))$ is satisfiable?
  - How about $\neg(P \rightarrow wp(Q, c))$?

VC GENERATION

SOUNDNESS AND COMPLETENESS

- Is the VC generation procedure sound?
  - Yes. Prove this!
- Is the VC generation procedure complete?
  - No. It is not even relatively complete.
  - The annotated loop invariant may not be strong enough.
- Can the VC generation procedure be fully automated?
  - Yes. Whole point of the exercise!

EXAMPLE

$\{true\}$

```
i := 1;
sum := 0;
while(i <= n) do
   j := 1;
   while(j <= i) do
      sum := sum + j; j := j + 1;
   i := i + 1;
```

$\{\texttt{sum} \geq 0\}$

EXAMPLE

$\{true\}$

```
i := 1;
sum := 0;
while(i <= n)@(sum ≥ 0) do
    j := 1;
    while(j <= i)@(sum ≥ 0 ∧ j ≥ 0) do
        sum := sum + j; j := j + 1;
    i := i + 1;
```

$\{\texttt{sum} \geq 0\}$

- $VC(sum \geq 0,\text{outer loop})$:
  - $sum \geq 0 \wedge i > n \rightarrow sum \geq 0$
  - $sum \geq 0 \wedge i \leq n \rightarrow sum \geq 0 \wedge 1 \geq 0$
  - $VC(sum \geq 0,\text{inner loop})$

EXAMPLE

$\{true\}$

```
i := 1;
sum := 0;
while(i <= n)@(sum ≥ 0) do
    j := 1;
    while(j <= i)@(sum ≥ 0 ∧ j ≥ 0) do
        sum := sum + j; j := j + 1;
    i := i + 1;
{sum ≥ 0}
```

- $VC(sum \geq 0$,inner loop):
  - $sum \geq 0 \wedge j \geq 0 \wedge j > i \rightarrow sum \geq 0$
  - $sum \geq 0 \wedge j \geq 0 \wedge j \leq i \rightarrow sum + j \geq 0 \wedge j + 1 \geq 0$

EXAMPLE

$\{true\}$

```
i := 1;
sum := 0;
while(i <= n)@(sum ≥ 0) do
    j := 1;
    while(j <= i)@(sum ≥ 0 ∧ j ≥ 0) do
        sum := sum + j; j := j + 1;
    i := i + 1;
```

$\{sum \geq 0\}$

- Final Formula:
    - $true \rightarrow 0 \geq 0 \wedge VC(sum \geq 0,$outer loop$)$

ADDING FUNCTIONS TO IMP

p = F*

F = function $f(x_1, \ldots, x_n)$ {c}

c = x := exp | x := havoc

= | assume(F) | assert(F)

= | skip | c; c | if(F) then c else c | while(F) do c

= | x := $f(\mathrm{exp}_1, \ldots, \mathrm{exp}_n)$ | return exp

MODULAR VERIFICATION

- Each function is annotated with a pre-condition and a post-condition.
- Pre-condition specifies what is expected of the function's arguments
  - Formula in FOL whose free variables are the formal parameters of the function.
- Post-condition describes the function's return value
  - Formula in FOL whose free variables are the formal parameters and a special variable called ret.
- Together, pre-condition and post-condition specify a *contract.*
  - If the function is called with values which obey the pre-condition, then the output of the function will obey the post-condition.

VERIFYING FUNCTION CONTRACT

```
function f(x1,…,xn)
    requires(Pre)
    ensures(Post)
    {Body;}
```

- The function contract can be verified by proving the validity of the Hoare Triple $\{Pre\}\ Body\ \{Post\}$

VERIFYING FUNCTION CALLS

- The function body may have calls to other functions (or even itself)
  - $\{P\}x := f(e_1, \ldots, e_n)\{Q\}$
- If we can guarantee that the function's pre-condition holds before the call, then we can assume that the function's post-condition will hold after the call.
- We model the function call as follows:

VERIFYING FUNCTION CALLS

- The function body may have calls to other functions (or even itself)
  - $\{P\}x := f(e_1, \ldots, e_n)\{Q\}$
- If we can guarantee that the function's pre-condition holds before the call, then we can assume that the function's post-condition will hold after the call.
- We model the function call as follows:

```
assert(Pre[e1/x1,…,en/xn]);
assume(Post[tmp/ret,e1/x1,…,en/xn]);
y := tmp;
```

VERIFYING FUNCTION CALLS

- The function body may have calls to other functions (or even itself)
  - $\{P\}x := f(e_1, \ldots, e_n)\{Q\}$
- If we can guarantee that the function's pre-condition holds before the call, then we can assume that the function's post-condition will hold after the call.
- We model the function call as follows:

```
assert(Pre[e1/x1,…,en/xn]);
assume(Post[tmp/ret,e1/x1,…,en/xn]);
y := tmp;
```

- Why do we have to use tmp?
- What is the generated VC?

VERIFYING FUNCTION CALLS

- The function body may have calls to other functions (or even itself)
  - $\{P\}x := f(e_1, \ldots, e_n)\{Q\}$
- If we can guarantee that the function's pre-condition holds before the call, then we can assume that the function's post-condition will hold after the call.
- We model the function call as follows:

```
assert(Pre[e1/x1,…,en/xn]);
assume(Post[tmp/ret,e1/x1,…,en/xn]);
y := tmp;
```

- Why do we have to use tmp?
- What is the generated VC? $P \rightarrow (Pre \wedge (Post \rightarrow Q[tmp/y]))$

EXAMPLE

```
FindMax(a,l,u)
  requires(l >= 0 && l <= u && u < |a|)
  ensures(∀i. l<=i<=u → ret >= a[i])
  {
     if (l == u)
        return a[l];
     else
        m := FindMax(a,l+1,u);
        if (a[l] > m)
           return a[l];
        else
           return m;
  }
```

EXAMPLE

```
FindMax(a,l,u)
  requires(l >= 0 && l <= u && u < |a|)
  ensures(∀i. l<=i<=u → ret >= a[i])
  {
     if (l == u)
        return a[l];
     else
        assert(Pre[l+1/l]);
        assume(Post[tmp/ret,l+1/l]);
        m := tmp;
        if (a[l] > m)
           return a[l];
        else
           return m;
  }
```

EXAMPLE

$\{l \geq 0 \wedge l \leq u \wedge u < |a|\}$

```
if (l == u)
    ret:=a[l];
else
    assert(Pre[l+1/l]);
    assume(Post[tmp/ret,l+1/l]);
    m := tmp;
    if (a[l] > m)
        ret:=a[l];
    else
        ret:=m;
```

$\{\forall i . l \leq i \leq u \rightarrow ret \geq a[i]\}$

$$
\begin{aligned}
Pre \rightarrow\ & (l = u \rightarrow Post[a[l]/ret]) \wedge \\
& l \neq u \rightarrow Pre[(l+1)/l] \\
& \wedge Post[tmp/ret, (l+1)/l] \rightarrow \\
& (a[l] > tmp \rightarrow Post[a[l]/ret]) \wedge (a[l] \leq tmp \rightarrow Post[tmp/ret])
\end{aligned}
$$

EXAMPLE - BINARY SEARCH

```
BinarySearch(a,l,u,e)
  requires(l >= 0 && u < |a|)
  ensures(ret ↔ ∃i.l <= i <= u & a[i] == e)
  {
    if (l > u) then
      return false;
    else
    {
      m := (l+u)/2;
      if (a[m]==e) then
        return true;
      else
      {
        if (a[m] < e)
          return BinarySearch(a,m+1,u,e);
        else
          return BinarySearch(a,l,m-1,e);
      }
    }
  }
```

EXAMPLE - BINARY SEARCH

```
BinarySearch(a,l,u,e)
  requires(l >= 0 && u < |a| && sorted(a,l,u) )
  ensures(ret ↔ ∃i.l <= i <= u & a[i] == e)
  {
    if (l > u) then
      return false;
    else
    {
      m := (l+u)/2;
      if (a[m]==e) then
        return true;
      else
      {
        if (a[m] < e)
          return BinarySearch(a,m+1,u,e);
        else
          return BinarySearch(a,l,m-1,e);
      }
    }
  }
```

$$sorted(a, l, u) \Leftrightarrow \forall i, j \,.\, l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

EXAMPLE - BINARY SEARCH

```
BinarySearch(a,l,u,e)
  requires(l >= 0 && u < |a| && sorted(a,l,u) )
  ensures(ret ↔ ∃i.l <= i <= u & a[i] == e)
  {
    if (l > u) then
      return false;
    else
    {
      m := (l+u)/2;
      if (a[m]==e) then
        return true;
      else
      {
        if (a[m] < e)
          return BinarySearch(a,m+1,u,e);
        else
          return BinarySearch(a,l,m-1,e);
      }
    }
  }
```

$$sorted(a, l, u) \Leftrightarrow \forall i, j \,.\, l \leq i \leq j \leq u \rightarrow a[i] \leq a[j]$$

BM CHAPTER 5 CONTAINS THE COMPLETE EXAMPLE

IN THE BOOK…

- More Examples (Chapters 5,6)
  - Linear Search
  - Bubble Sort
  - Quick Sort
- A slightly different VC generation procedure
- Heuristics for crafting loop invariants

HANDLING GLOBAL VARIABLES

- If there are global variables shared across functions, then executing a function can cause side effects.
  - Is the previous approach still sound?
- We will use havoc assignments to model side-effects.
- Function contracts now specify global variables which may be modified.

```
function f(x1,…,xn)
    requires(Pre)
    ensures(Post)
    modifies(v1,…,vm)
    {Body;}
```

HANDLING GLOBAL VARIABLES

- How to check correctness of the function contract?
- $y := f(e_1, \ldots, e_n)$ is replaced by

```
assert(Pre[e1/x1,…,en/xn]);
v1:=havoc;… vm:=havoc;
assume(Post[tmp/ret,e1/x1,…,en/xn]);
y := tmp;
```

ADDING POINTERS TO IMP

- We add two more program statements:
  - x := *y
  - *x := e
- Consider the following code:
  - $\{true\}$x := y; *y := 3; *x := 2; z := *y;$\{z = 3\}$
  - Does it satisfy the specification? What is $wp(z = 3, c)$?
- We need new rules for assignment statements involving pointers.

HANDLING POINTERS

- We treat the memory as a giant array M, with the pointer variables behaving as indices into the array.
  - x := *y becomes x := M[y]
  - *x := e becomes M := M⟨x◁e⟩
- $\{???\} x := *y \{Q\}$

Adapted from Isil Dillig's Lectures

HANDLING POINTERS

- We treat the memory as a giant array M, with the pointer variables behaving as indices into the array.
  - x := *y becomes x := M[y]
  - *x := e becomes M := M⟨x◃e⟩
- $\{Q[M[y]/x]\}x := *y\{Q\}$

HANDLING POINTERS

- We treat the memory as a giant array M, with the pointer variables behaving as indices into the array.
  - x := *y becomes x := M[y]
  - *x := e becomes M := M⟨x◁e⟩
- $\{Q[M[y]/x]\} x := *y \{Q\}$
- $\{???\} *x := e \{Q\}$

HANDLING POINTERS

- We treat the memory as a giant array M, with the pointer variables behaving as indices into the array.
  - x := *y becomes x := M[y]
  - *x := e becomes M := M⟨x◃e⟩
- $\{Q[M[y]/x]\}x := *\,y\{Q\}$
- $\{Q[M\langle x \triangleleft e\rangle/M]\} *\, x := e\{Q\}$

HANDLING POINTERS

- We treat the memory as a giant array M, with the pointer variables behaving as indices into the array.
  - x := *y becomes x := M[y]
  - *x := e becomes M := M⟨x◁e⟩
- $\{Q[M[y]/x]\} x := *y \{Q\}$
- $\{Q[M\langle x \lhd e\rangle/M]\} *x := e \{Q\}$
- Consider the code again:
  - $\{true\}$x := y; *y := 3; *x := 2; z := *y;$\{z = 3\}$

HANDLING POINTERS

- We treat the memory as a giant array M, with the pointer variables behaving as indices into the array.
  - x := *y becomes x := M[y]
  - *x := e becomes M := M⟨x◁e⟩
- $\{Q[M[y]/x]\}x := *y\{Q\}$
- $\{Q[M\langle x \triangleleft e\rangle/M]\} *x := e\{Q\}$
- Consider the code again:
  - $\{true\}$x := y; *y := 3; *x := 2; z := *y;$\{z = 3\}$
  - VC: $true \rightarrow M\langle y \triangleleft 3\rangle\langle y \triangleleft 2\rangle[y] = 3$

Adapted from Isil Dillig's Lectures