Axiomatic Semantics

1. Language for making assertions about programs
2. Rules for establishing, i.e. proving the assertions

Typical kinds of assertions:
- This program terminates.
- During execution if var $z$ has value 0, then $x$ equals $y$
- All array accesses are within array bounds

Some typical languages of assertions:
- First-order logic
- Other logics (e.g., temporal logic)

TODAY’S PLAN

1. Define a small language
2. Define a logic for verifying assertions

IMP: An Imperative Language

syntax and operational semantics
### IMP Syntactic Entities

- **Int**: integer literals, $n$
- **Bool**: booleans, \{true, false\}
- **Loc**: locations, $x, y, z, ...$
- **Aexp**: arithmetic expressions, $e$
- **Bexp**: boolean expressions, $b$
- **Comm**: commands, $c$

#### Abstract Syntax: Arith Expressions (Aexp)

$$e ::= n \quad \text{for } n \in \text{Int}$$

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>for $x \in \text{Loc}$</td>
</tr>
<tr>
<td>$e_1 + e_2$</td>
<td>for $e_1, e_2 \in \text{Aexp}$</td>
</tr>
<tr>
<td>$e_1 - e_2$</td>
<td>for $e_1, e_2 \in \text{Aexp}$</td>
</tr>
<tr>
<td>$e_1 \times e_2$</td>
<td>for $e_1, e_2 \in \text{Aexp}$</td>
</tr>
</tbody>
</table>

**Note:**
- Variables are not declared
- All variables have integer type
- There are no side-effects

#### Abstract Syntax: Bool Expressions (Bexp)

- **true**
  - true
  - false
- **e_1 = e_2** for $e_1, e_2 \in \text{Aexp}$
- **e_1 < e_2** for $e_1, e_2 \in \text{Aexp}$
- **!b** for $b \in \text{Bexp}$
- **b_1 || b_2** for $e_1, e_2 \in \text{Bexp}$
- **b_1 & b_2** for $e_1, e_2 \in \text{Bexp}$

#### Abstract Syntax: Commands (Comm)

- **c ::= skip**
- **x:= e** for $x \in \text{L}$ & $e \in \text{Aexp}$
- **c_1 ; c_2** for $c_1, c_2 \in \text{Comm}$
- **if b then c_1 else c_2** for $b \in \text{Bexp}$ & $c_1, c_2 \in \text{Comm}$
- **while b do c** for $c \in \text{Comm}$ & $b \in \text{Bexp}$

**Note:**
- Typing rules embedded in syntax definition
  - Other checks may not be context-free
  - need to be specified separately (e.g., variables are declared)
- Commands contain all the side-effects in the language
Semantics of IMP: States

- Meaning of IMP expressions depends on the values of variables

- A state $\sigma$ is a function from Loc to Int
  - Value of variables at a given moment
  - Set of all states is $\Sigma = \text{Loc} \rightarrow \text{Int}$

Operational Semantics of IMP

Evaluation judgment for expressions:

- Ternary relation on expression, a state, and a value:
  - We write: $<e, \sigma> \downarrow n$
  - "Expression $e$ in state $\sigma$ evaluates to $n$"

Q: Why no state on the right?
  - Evaluation of expressions has no side-effects:
    - i.e., state unchanged by evaluating an expression

Q: Can we view judgment as a function of 2 args $e, \sigma$?
  - Only if there is a unique derivation ...

Operational Semantics of IMP

Evaluation judgement for commands

- Ternary relation on expression, state, and a new state
  - We write: $<c, \sigma> \downarrow \sigma'$
  - "Executing cmd $c$ from state $\sigma$ takes system into state $\sigma'$"

- Evaluation of a command has effect
  - but no direct value
  - So, "result" of a command is a new state $\sigma'$

Note: evaluation of a command may not terminate

Q: Can we view judgment as a function of 2 args $e, \sigma$?
  - Only if there is a unique successor state ...

Evaluation Rules (for Aexp)

\[
\begin{align*}
<n, \sigma> & \downarrow n \\
<e_1 + e_2, \sigma> & \downarrow n_1 + n_2 \\
<e_1 - e_2, \sigma> & \downarrow n_1 - n_2 \\
<e_1 * e_2, \sigma> & \downarrow n_1 * n_2
\end{align*}
\]
Evaluation Rules (for Bexp)

- \( <true, σ> \Downarrow true \)
- \( <false, σ> \Downarrow false \)
- \( <e_1, σ> \Downarrow n_1 \)
- \( <e_2, σ> \Downarrow n_2 \)
- \( p \) is \( n_1 = n_2 \)
- \( <e_1 = e_2, σ> \Downarrow p \)
- \( <b_1 \land b_2, σ> \Downarrow p_1 \land p_2 \)
- \( <b_1 \lor b_2, σ> \Downarrow p_1 \lor p_2 \)
- \( <b_1 \implies b_2, σ> \Downarrow p_1 \implies p_2 \)
- \( <b, σ> \Downarrow p \)
- \( <b, σ> \Downarrow false \)
- \( <c_1, σ> \Downarrow σ' \)
- \( <c_2, σ'> \Downarrow σ'' \)
- \( <c_1; c_2, σ> \Downarrow σ'' \)

Evaluation Rules (for Comm)

- \( <skip, σ> \Downarrow σ \)
- \( <c_1, σ> \Downarrow σ' \)
- \( <c_2, σ'> \Downarrow σ'' \)
- \( <c_1; c_2, σ> \Downarrow σ'' \)

Axiomatic Semantics

1. Language for making assertions about programs
2. Rules for establishing, i.e. proving the assertions

Typical kinds of assertions:
- This program terminates.
- During execution if var \( z \) has value 0, then \( x \) equals \( y \)
- All array accesses are within array bounds

Some typical languages of assertions:
- First-order logic
- Other logics (e.g., temporal logic)
Axiomatic Semantics

History: Program Verification

- Turing 1949: Checking a large routine
- Floyd 1967: Assigning meaning to programs
- Hoare 1971: An 'axiomatic basis for computer programming'

Program Verifiers (70's - 80's)
- PREfix: Symbolic Execution for bug-hunting (WinXP)
- Software Validation tools

Foundation for Software Verification
- Deductive Verifiers: ESCJava, Spec#, Verifast, Y0, ...
- Model Checkers: SLAM, BLAST,...
- Test Generators: DART, CUTE, EXE,...

Hoare Triples

- Partial correctness assertion: \( \{A\} c \{B\} \)
  If \( A \) holds in state \( \sigma \) and exists \( \sigma' \) s.t. \( <c, \sigma> \Downarrow \sigma' \)
  then \( B \) holds in \( \sigma' \)

- Total correctness assertion: \([A] c \{B\}\)
  If \( A \) holds in state \( \sigma \)
  then there exists \( \sigma' \) s.t. \( <c, \sigma> \Downarrow \sigma' \) and \( B \) holds in \( \sigma' \)

- \([A]\) is called precondition, \([B]\) is called postcondition
- Example: \( \{y=x\} z := x; z := z+1 \{y < z\} \)

The Assertion Language

- Arith Exprs + First-order Predicate logic

\[ A ::= \text{true} \mid \text{false} \]
\[ \mid e_1 = e_2 \mid e_1 \land e_2 \]
\[ \mid \neg A \mid A_1 \land A_2 \mid A_1 \lor A_2 \mid A_1 \Rightarrow A_2 \]
\[ \mid \exists x.A \mid \forall x.A \]

- IMP boolean expressions are assertions
Semantics of Assertions

- **Judgment** $\sigma \models A$ means assertion holds in given state

  $\sigma \models true$ always

  $\sigma \models e_1 = e_2$ iff $\langle e_1, \sigma \rangle \downarrow n_1$, $\langle e_2, \sigma \rangle \downarrow n_2$ and $n_1 = n_2$

  $\sigma \models e_1 \leq e_2$ iff $\langle e_1, \sigma \rangle \downarrow n_1$, $\langle e_2, \sigma \rangle \downarrow n_2$ and $n_1 \leq n_2$

  $\sigma \models A_1 \land A_2$ iff $\sigma \models A_1$ and $\sigma \models A_2$

  $\sigma \models A_1 \lor A_2$ iff $\sigma \models A_1$ or $\sigma \models A_2$

  $\sigma \models A_1 \rightarrow A_2$ iff $\sigma \models A_1$ implies $\sigma \models A_2$

  $\sigma \models \exists x. A$ iff for some $n$ in $\mathbb{Z}$. $\sigma[x := n] \models A$

  $\sigma \models \forall x. A$ iff for all $n$ in $\mathbb{Z}$. $\sigma[x := n] \models A$

Semantics of Assertions

- **Total correctness assertion:**
  
  $\models [A] c [B]$ iff $\models \{A\} c \{B\}$ and 
  
  forall $\sigma$ in $\Sigma$. $\sigma \models A$ implies [exists $\sigma'$ in $\Sigma$. $\langle c, \sigma \rangle \downarrow \sigma'$]

Deriving Assertions

- **Formal $\models \{A\} c \{B\}$ hard to use**

- Defined in terms of the op-semantics

- Next, symbolic technique (logic)

- for deriving valid triples $\models \{A\} c \{B\}$
Derivation Rules for Hoare Triples

• Write \( \vdash \{A\} \ c \ \{B\} \) when we can derive the triple using derivation rules

• One rule per command

• Plus, the rule of consequence:

\[
A' \Rightarrow A \quad \vdash \{A\} \ c \ \{B\} \quad B \Rightarrow B' \\
\vdash \{A'\} \ c \ \{B'\}
\]

Free and Bound Variables

Key idea in logic/PL: scoping & substitution

• Assertions are equivalent up to renaming of bound variables (a.k.a. alpha-renaming)

• Examples:

  \( \forall x. x = x \) is the same as \( \forall y. y = y \)
  - Rename bound \( x \) with \( y \)

  \( \forall x. \forall y. x = y \) is the same as \( \forall z. \forall x. z = x \)
  - Rename bound \( x \) with \( z \) and \( y \) with \( x \)

Substitution

• \([e'/x]e\) is substituting \( e' \) for \( x \) in \( e \)
  - Also written as \( e[e'/x] \)
  - Note: only substitute the free occurrences

• Alpha-rename bound variables to avoid conflicts
  - To subst. \([e'/x]e\) in \( \forall y. x = y \) rename \( y \) if it occurs in \( e' \)
  - Result of alpha-renaming: \( \forall z. e' = z \)

• We say that substitution avoids variable capture \([x/z]\) \( \forall x. z = x \) is ?
  - \( \forall x. x = x \) Wrong
  - \( \forall y. x = y \) Correct
Example: Assignment

Assume \( x \) does not appear in \( e \)

Prove \(|-\{true\} \, x:=e \, \{ x = e \} \ |

Note \([e/x](x = e) = e = [e/x]e = e = e\)

Use assignment rule \( \ldots \) then conseq. rule

\[
\begin{align*}
\text{x does not appear in e} \\
\text{true => e = e} \\
\text{|- \{e = e\} \, x:=e \, \{ x = e \}}
\end{align*}
\]

Example: Conditional

Prove: \( \{true\} \, if \, y<=0 \, then \, x:=1 \, else \, x:=y \, \{x>0\} \)

\[
\begin{align*}
\text{true & y<=0 => 1>0} & \implies |- \{1>0\} \, x:=1 \, \{x>0\} \\
\text{true & y>0 => y>0} & \implies |- \{y>0\} \, x:=y \, \{x>0\}
\end{align*}
\]

- Rule for if-then-else
- Rule for assignment + consequence

Example: Loop

- Prove \(|-\{x<=0\} \, \text{while} \, x<=5 \, \text{do} \, x:=x+1 \, \{x=6\} \)
- Use the rule for while with invariant \( x <= 6 \):

\[
\begin{align*}
\text{x<=6 \& x<=5 => x+1<=6} & \implies |- \{x+1<=6\} \, x:=x+1 \, \{x<=6\} \\
\text{|- \{x+1<=6\} \, x:=x+1 \, \{x<=6\}} & \implies |- \{x<=6 \& x<=5\} \, x:=x+1 \, \{x<=6\} \\
\text{|- \{x<=6\} \, \text{while} \, x<=5 \, \text{do} \, x:=x+1 \, \{x<=6 \& x>5\}} & \implies |- \{x<=6 \& x>=5\} \, x:=x+1 \, \{x<=6 \& x>5\}
\end{align*}
\]

- Finish off with consequence rule:

\[
\begin{align*}
\text{x<=0 => x=6} & \implies |- \{x=6\} \\
\text{|- \{x=6\} \, w \, \{x<=6 \& x>5\} \, x<=6 \& x>5 => x=6} & \implies |- \{x=6\} \, w \, \{x = 6\}
\end{align*}
\]

Soundness of Axiomatic Semantics

Formal Statement of Soundness:

If \(|-\{A\} \, c \, \{B\}\) then \(|=\{A\} \, c \, \{B\}\)

Equivalently

If \(H:: \ |\{-\{A\} \, c \, \{B\}\} \) then \(|=\{A\} \, c \, \{B\}\)

for all \(\sigma\) if \(\sigma \|= A\) and \(D::<c,\sigma> \Downarrow \sigma'\) then \(\sigma' \|= B\)

Proof:
Simultaneous induction on structure of \(D\) and \(H\)
Algorithmic Verification

Hoare rules mostly syntax directed, but:

1. When to apply the rule of consequence?
2. What invariant to use for while?
3. How to prove implications (conseq. rule)?

Hint:
(3) involves ... SMT
(2) invariants are the hardest problem
(1) lets see how to deal with ...

Making Floyd-Hoare Algorithmic:
Predicate Transformers

Technique: Weakest Preconditions

WP(c,B): weakest predicate s.t. \{WP(c,B)\} c \{B\}

- For any A we have \{A\} c \{B\} iff A => WP(c, B)

How to verify |- {A} c \{B\}?
1. Compute: WP(c,B)
2. Prove: A => WP(c,B)

Weakest Preconditions

Define \(wp(c, B)\) using Hoare rules

\[
\begin{align*}
wp(c_1;c_2, B) &= wp(c_1, wp(c_2, B)) \\
wp(x:=e , B) &= [e/x]B \\
wp(\text{if } e \text{ then } c_1 \text{ else } c_2, B) &= e=>wp(c_1, B) \&\& !e=>wp(c_2, B) \\
\end{align*}
\]

\[
\begin{align*}
|- \{A\} c_1 \{B\} &\quad |- \{B\} c_2 \\
|- \{A\} c_1; c_2 \{C\} \\
|- \{A\} c_1 \{B\} &\quad |- \{A \& !b\} c_2 \{B\} \\
|- \{A\&b\} c_1 \{B\} \\
|- \{A \& \!b\} c_2 \{B\} \\
\end{align*}
\]
Weakest Preconditions for Loops

Start from the equivalence

\[
\text{while } b \text{ do } c = \begin{cases} \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip} \end{cases}
\]

Let \( W = \wp(\text{while } b \text{ do } c, B) \)

It must be that: \( W = [b \to \wp(c, W) \& \neg b \Rightarrow B] \)

But this is a recursive equation! How to compute?!

- We’ll return to finding loop WPs later …

Technique: Strongest Postconditions

What postcond. is guaranteed after prec. \( y > 100 \)?

\( \text{SP}(c, A): \text{strongest predicate s.t. } \{A\} \ c \ \{\text{SP}(c, A)\} \)

- For any \( B \) we have \( \{A\} \ c \ \{B\} \iff \text{SP}(c, A) \Rightarrow B \)

How to verify \( \{A\} \ c \ \{B\} \)?

1. Compute: \( \text{SP}(c, A) \)
2. Prove: \( \text{SP}(c, A) \Rightarrow B \)

Strongest Postconditions

Define \( \text{sp}(c, B) \) following Hoare rules

\[
\begin{align*}
\text{sp}(c_1; c_2, A) &= \\
\text{sp}(c_2, \text{sp}(c_1, A)) \\
\text{sp}(x := e, A) &= \exists x_0. [x_0/x]A \& \& x = [x_0/x]e \\
\text{sp}(\text{if } e \text{ then } c_1 \text{ else } c_2, A) &= \begin{cases} \text{sp}(c_1, A \& e) \& \& \text{sp}(c_2, A \& \neg e) \end{cases}
\end{align*}
\]

Axiomatic Semantics on Flow Graphs

Floyd’s Original Formulation
Axiomatic Semantics over Flow Graphs

Relaxing Specifications via Consequence
Will revisit later as subtyping

Sequential Composition

Backwards using weakest preconditions
Forwards using strongest postconditions

Conditionals

Joins
Conditional+Join: Forward

\[
\begin{align*}
\{ x \neq 0 || a = 0 \} \\
\{ x \neq 0 \} &~~ T \\
\{ x \neq 0 \} &~~ F \\
\{ x \neq 0 \} &~~ \text{T} \\
\{ a = 2x \} &~~ \text{F} \\
\{ a = 2x \} &~~ \text{F} \\
\{ a = 2x \} &~~ \text{T} \\
\{ x \neq 0 || a = 0 \} &~~ \text{F}
\end{align*}
\]

- Check the implications (simplifications)

Forward or Backward?

- Forward reasoning
  - Know the precondition
  - Want to know what postcond the code guarantees

- Backward reasoning
  - Know what we want to code to establish
  - Want to know under what preconditions this happens

Another Example: Double Locking

"An attempt to re-acquire an acquired lock or release a released lock will cause a deadlock."

Calls to lock and unlock must alternate.
Locking Rules

Boolean variable `locked` states if lock is held or not

• `{ !locked & P[true/locked] } lock { P }
  lock behaves as `assert(!locked); locked := true`

• `{ locked & P[false/locked] } unlock { P }
  unlock behaves as `assert(locked); locked := false`

Locking Example

```
{ !locked & x = 0 }
lock ...
{ locked & x = 0 }

{ locked & x = 0 }
unlock ...
{ !locked & x = 0 }
```

Review

```
\[ \begin{align*}
\text{\{ Q \} } \quad \text{\{ P \} } \\
\quad \text{\{ P \} } \quad \text{\{ P \} } \\
\quad \text{\{ P \} } \\
\quad \text{T} \quad \text{E} \quad \text{F} \\
\quad \text{\{ P \} } \\
\quad \text{\{ P \} } \quad \text{\{ P \} } \\
\quad \text{\{ P \} } \\
\end{align*} \]
```

```
\text{Implication is always in the direction of the control flow}
```

What about real languages?

• Loops
• Function calls
• Pointers
Reasoning about loops: Rules

| - {A & b} c {A}
| - {A} while b do c {A & !b}

Rewrite A with I: Loop Invariant

| - {I & b} c {I}
| - {I} while b do c {I & !b}

P => I

| - {I} while b do c {I & !b}  I & !b => Q

Rule of Consequence

Reasoning about loops: Flow Graphs

- Loops can be handled using conditionals and joins
- Consider the while b do S statement

if P => I (loop invariant holds initially)
and I & !b => Q (loop establishes the postcondition)
and { I & b } S { I } (loop invariant is preserved)

Loop Example

Verify:
{x=8 & y=16} while(x>0){x--; y-=2;} {y = 0}

Find an appropriate invariant I
- Holds initially  x = 8 & y = 16
- Holds at end    y == 0

Loop Example (II)

Guess invariant y = 2*x

Check :
- Initial:     x = 8 & y = 16     => y = 2*x
- Preservation: y = 2*x & x>0     => y-2 = 2*(x-1)
- Final:       y = 2*x & x<=0    => y = 0  Invalid
Loop Example (III)

Guess invariant \( y = 2^x \land x \geq 0 \)

Check
- Initial: \( x = 8 \land y = 16 \) \( \Rightarrow y = 2^x \land x \geq 0 \)
- Preserv: \( y = 2^x \land x \geq 0 \land x > 0 \) \( \Rightarrow y = 2(y-2) = 2^x = x = x - 1 \geq 0 \)
- Final: \( y = 2^x \land x \geq 0 \land x \leq 0 \) \( \Rightarrow y = 0 \)

Note: Invariant depends on your proof goal!

Verification Example

```c
int square(int n) {
  int k=0, r=0, s=1;
  while(k != n) {
    r = r + s;
    s = s + 2;
    k = k + 1;
  }
  return r;
}
```

Need: \( \{ r = k^2 \land s = 2k+1 \} \Rightarrow \{ r = k^2 \land s = 2k+1 \} \)

i.e. \( \{ r = k^2 \land s = 2k+1 \} \Rightarrow WP(c, \{ r = k^2 \land s = 2k+1 \}) \)

Invalid
What about real languages?

- Loops
- Function calls
- Pointers

Functions are big instructions

Suppose we have verified `bsearch`

```c
int bsearch(int a[], int p) {
    { sorted(a) }
    ...}
    { r=-1 || (r>=0 & r < a.length & a[r]=p)}
    return r;
}
```

- Function `spec = precondition + postcondition`
- Also called a `contract`

Function Calls

- Consider a call to function `y := f(e)`
  - return variable `r`
  - precondition `Pre`, postcondition `Post`

- Rule for function call:

  |- P => Pre[e/x]  |- {Pre} f {Post}  |- Post[e/x,y/r] => Q
  |- {P} y := f(e){Q}

Function Calls

- Consider a call to function `y := f(e)`
  - return variable `r`
  - precondition `Pre`, postcondition `Post`

- Rule for function call:

  ```
  y := f(e)
  { P }  if P => Pre[E/x]
  { Q }  and Post[E/x,y/r] => Q
  ```
Consider the call:

```c
int bsearch(int a[], int p) {
    { sorted(a) }
    ...  
    { r=-1 || (r>=0 && r<a.length && a[r]=p)}
    return r;
}
```

```
Consider the call
{sorted(arr) }
y:=bsearch(arr,5)
{y=-1 || arr[y]=5}
if(y!=-1){
   ... }
```

- sorted[array] => Pre[a := arr]
- Post[y/r, arr/a, 5/p] => (y=-1 || arr[y]=5)

### Assignment and Aliasing

Does assignment rule work with aliasing?

If *x* and *y* are aliased then:

```
{x=y} *x:=5 {*}x + *y=10
```

**Hoare Rules: Assignment and References**

- When is the following Hoare triple valid?
  ```
  { A } *x := 5 {*}x + *y = 10 
  ```

- A should be " *y = 5 or x = y "

- but Hoare rule for assignment gives:
  ```
  [5/*x](x + *y = 10)
  = 5 + *y = 10
  = *y = 5
  (uh oh! we lost one case! What happened?)
  ```
Hoare Rules: Assignment and References

Modeling writes with memory expressions

- Treat memory as a whole with memory variables \((M)\)
- \(\text{upd}(M, E_1, E_2)\): update \(M\) at address \(E_1\) with value \(E_2\)
- \(\text{sel}(M, E_1)\): read \(M\) at address \(E_1\)

Reason about memory expressions with McCarthy’s rule

\[
\text{sel}(\text{upd}(M, E_1, E_2), E_3) = \begin{cases} 
E_2 & \text{if } E_1 = E_3 \\
\text{sel}(M, E_3) & \text{if } E_1 \neq E_3
\end{cases}
\]

Assignment (update) changes the value of memory

\[
\{B[\text{upd}(M, E_1, E_2)/M]\} \ *E_1 := E_2 \ [B]
\]

Memory Aliasing

- Consider again: \{A\} \ *x := 5 \ \{*x + *y = 10 \}

\[
A = \left[\text{upd}(M, x, 5)/M\right] (*x + *y = 10)
\]

\[
= \left[\text{upd}(M, x, 5)/M\right] \left(\text{sel}(M,x) + \text{sel}(M,y) = 10\right)
\]

\[
= \left[\text{sel}(\text{upd}(M, x, 5), x) + \text{sel}(\text{upd}(M, x, 5), y) = 10\right]
\]

\[
= \left[5 + \text{sel}(\text{upd}(M, x, 5), y) = 10\right]
\]

\[
= \left[\text{sel}(\text{upd}(M, x, 5), y) = 5\right]
\]

\[
= (x = y & 5 = 5) \mid \mid (x \neq y & \text{sel}(M, y) = 5)
\]

\[
= x=y \mid \mid *y = 5
\]

Program Verification Tools

- Semi-automated
  - You write some invariants and specifications
  - Tool tries to fill in the other invariants
  - And to prove all implications
  - Explains when implication is invalid: counterexample for your specification

- ESC/Java is one of the best tools
- ... Spec#, Verifast, VCC

Algorithmic Program Verification

... or how does ESC/Java work?

Q: How to algorithmically prove \{P\} \(c\) \{Q\}?

If no loops:
1. Compute: \(WP(c, Q)\)
2. Prove: \(P \Rightarrow WP(c, Q)\)

Verification Condition
Proved By SMT Solver
VC Generation for Loops

Suppose all loops annotated with Invariant

\[
\text{while } I \text{ } b \text{ } do \text{ } c
\]

Compute VC:

\[
\text{SMTValid(VC)} \implies \{P\} \text{ } c \{Q\}
\]

Q: Why not iff?

1. Loop invariants may be bogus...
2. SMT solver may not handle logic...

VCGen

We will write a function

\[
v\text{cgen} :: \text{Pred} \rightarrow \text{Com} \rightarrow (\text{Pred}, \{\text{Pred}\})
\]

Suppose \((Q',L') = V\text{CG}(c,(Q,L;))\)

Then VC for \(
\{P\} \text{ } c \{Q\}
\) is:

\(P \implies Q' \&\& \{f \text{ in } L'\} f\)

- \(L'\) : the set of conditions that must be true
  - From loops (init, preservation, final)
- \(Q'\) : “precondition” modulo invariants...

VCGen

\[
v\text{cgen} :: \text{Pred} \rightarrow \text{Com} \rightarrow \text{VC Pred}
\]

\[
v\text{cgen} \text{ (Skip) } q \n = \text{return } q
\]

\[
v\text{cgen} \text{ (Asgn } x \text{ e) } q \n = \text{return } \$(q \text{ `subst` (x, e))}
\]

\[
v\text{cgen} \text{ (If } b \text{ c1 c2) } q \n = \text{do } q1 \leftarrow v\text{cgen } q \text{ c1}
\]

\[
q2 \leftarrow v\text{cgen } q \text{ c2}
\]

\[
\text{return } \$(b \text{ `And` q1) `Or` (Not b `And` q2)}
\]

\[
v\text{cgen} \text{ (While } i \text{ b c) } q \n = \text{do } q' \leftarrow v\text{cgen } i \text{ c}
\]

\[
\text{valid } \$(i \text{ `And` Not b) `implies` q'}
\]

\[
\text{valid } \$(i \text{ `And` b) `implies` q}
\]

\[
\text{return } i
\]

VCGen

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

\[
\text{verify } p \text{ c q} = \text{all smtValid queries}
\]

\[
\text{where}
\]

\[
(q', \text{ conds}) = \text{runState } (v\text{cgen } q \text{ c}) []
\]

\[
\text{queries} = p \text{ `implies` q' : conds}
\]
**ESC/Java**

Semi-automated “Deductive Verification”

- You write the invariants

- **ESC/Java:**
  - VCGen
  - Simplify: SMT used to prove VC

- Explains when implication is invalid: counterexample for your specification