Floyd-Hoare Logic

Ranjit Jhala
UC San Diego

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	for $n \in$ Int
	| x	for $x \in$ Loc
	| $e_1 + e_2$	for $e_1, e_2 \in$ Aexp
	| $e_1 - e_2$	for $e_1, e_2 \in$ Aexp
	| $e_1 * e_2$	for $e_1, e_2 \in$ Aexp

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 $e_1, e_2 \in$ Aexp
	| $e_1 < e_2$	for $e_1, e_2 \in$ Aexp
	| $!\, b$	for $b \in$ Bexp
	| $b_1 \,\|\|\, b_2$	for $e_1, e_2 \in$ Bexp
	| $b_1 \,\&\, b_2$	for $e_1, e_2 \in$ Bexp

Abstract Syntax: Commands (Comm)

`c ::= skip`	
| `x := e`	for $x \in$ L & $e \in$ Aexp
| $c_1 ; c_2$	for $c_1, c_2 \in$ Comm
| `if b then` c_1 `else` c_2	for $b \in$ Bexp & $c_1, c_2 \in$ Comm
| `while b do c`	for $c \in$ Comm & $b \in$ 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 σ is a function from Loc to Int
  - Value of variables at a given moment
  - Set of all states is Σ = Loc -> Int

Operational Semantics of IMP

Evaluation judgment for expressions:

- Ternary relation on expression, a state, and a value:
- We write: $\langle \mathbf{e}, \sigma \rangle \Downarrow n$

  "Expression **e** in state σ 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**, σ ?
- 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: $\langle \mathbf{c}, \sigma \rangle \Downarrow \sigma'$

  "Executing cmd **c** from state σ takes system into state σ' "
- Evaluation of a command has effect
  - but no direct value
  - So, "result" of a command is a new state σ'

Note: evaluation of a command may not terminate

Q: Can we view judgment as a function of 2 args **e**, σ ?
- Only if there is a unique successor state …

Evaluation Rules (for Aexp)

$$\frac{}{\langle \mathbf{n}, \sigma \rangle \Downarrow n} \qquad \frac{}{\langle \mathbf{x}, \sigma \rangle \Downarrow \sigma(\mathbf{x})}$$

$$\frac{\langle \mathbf{e}_1, \sigma \rangle \Downarrow n_1 \quad \langle \mathbf{e}_2, \sigma \rangle \Downarrow n_2}{\langle \mathbf{e}_1 \ \mathbf{+} \ \mathbf{e}_2, \sigma \rangle \Downarrow n_1 + n_2} \qquad \frac{\langle \mathbf{e}_1, \sigma \rangle \Downarrow n_1 \quad \langle \mathbf{e}_2, \sigma \rangle \Downarrow n_2}{\langle \mathbf{e}_1 \ \mathbf{-} \ \mathbf{e}_2, \sigma \rangle \Downarrow n_1 - n_2}$$

$$\frac{\langle \mathbf{e}_1, \sigma \rangle \Downarrow n_1 \quad \langle \mathbf{e}_2, \sigma \rangle \Downarrow n_2}{\langle \mathbf{e}_1 \ \mathbf{*} \ \mathbf{e}_2, \sigma \rangle \Downarrow n_1 * n_2}$$

Evaluation Rules (for Bexp)

$$\frac{}{\langle \mathtt{true}, \sigma\rangle \Downarrow \mathit{true}} \qquad \frac{}{\langle \mathtt{false}, \sigma\rangle \Downarrow \mathit{false}}$$

$$\frac{\langle \mathbf{e}_1, \sigma\rangle \Downarrow n_1 \quad \langle \mathbf{e}_2, \sigma\rangle \Downarrow n_2 \quad p \text{ is } n_1 = n_2}{\langle \mathbf{e}_1 = \mathbf{e}_2, \sigma\rangle \Downarrow p} \qquad \frac{\langle \mathbf{e}_1, \sigma\rangle \Downarrow n_1 \quad \langle \mathbf{e}_2, \sigma\rangle \Downarrow n_2 \quad p \text{ is } n_1 < n_2}{\langle \mathbf{e}_1 < \mathbf{e}_2, \sigma\rangle \Downarrow p}$$

$$\frac{\langle \mathbf{b}_1, \sigma\rangle \Downarrow p_1 \quad \langle \mathbf{b}_2, \sigma\rangle \Downarrow p_2}{\langle \mathbf{b}_1 \text{ Æ } \mathbf{b}_2, \sigma\rangle \Downarrow p_1 \text{Æ } p_2} \qquad \frac{\langle \mathbf{b}_1, \sigma\rangle \Downarrow p_1 \quad \langle \mathbf{b}_2, \sigma\rangle \Downarrow p_2}{\langle \mathbf{b}_1 \text{ Ç } \mathbf{b}_2, \sigma\rangle \Downarrow p_1 \text{Ç } p_2}$$

$$\frac{\langle \mathbf{b}, \sigma\rangle \Downarrow p}{\langle \text{:}\mathbf{b}, \sigma\rangle \Downarrow \text{:}\ p}$$

Evaluation Rules (for Comm)

$$\frac{}{\langle \mathtt{skip}\ , \sigma\rangle \Downarrow \sigma}$$

$$\frac{\langle \mathtt{c}_1, \sigma\rangle \Downarrow \sigma' \quad \langle \mathtt{c}_2, \sigma'\rangle \Downarrow \sigma''}{\langle \mathtt{c}_1 ; \mathtt{c}_2, \sigma\rangle \Downarrow \sigma''}$$

Define $\sigma[\mathtt{x} := n]$ as:

$\sigma[\mathtt{x} := n](\mathtt{x}) = n$

$\sigma[\mathtt{x} := n](\mathtt{y}) = \sigma(\mathtt{y})$

$$\frac{\langle \mathtt{e}, \sigma\rangle \Downarrow n}{\langle \mathtt{x}\ := \ \mathtt{e}, \sigma\rangle \Downarrow \sigma[\mathtt{x} := n]}$$

Evaluation Rules (for Comm)

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathit{true} \quad \langle \mathtt{c}_1, \sigma\rangle \Downarrow \sigma'}{\langle \mathtt{if\ b\ then\ c}_1 \ \mathtt{else\ c}_2, \sigma\rangle \Downarrow \sigma'}$$

$$\frac{\langle \mathtt{b}, \sigma\rangle \Downarrow \mathit{false} \quad \langle \mathtt{c}_2, \sigma\rangle \Downarrow \sigma'}{\langle \mathtt{if\ b\ then\ c}_1 \ \mathtt{else\ c}_2, \sigma\rangle \Downarrow \sigma'}$$

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 σ and exists σ' s.t. <**c**, σ >$\Downarrow$σ'
  then B holds in σ'

- Total correctness assertion: [A] **c** [B]
  If A holds in state σ
  then there exists σ' s.t. <**c**, σ >$\Downarrow$σ' *and B holds in σ'*

- [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 ::= true | false
  | $e_1 = e_2$ | e_1 ¸ e_2
  | ¬ A | A_1 && A_2 | A_1 || A_2 | A_1 => A_2
  | \exists x.A | \forall x.A

- IMP boolean expressions are assertions

Semantics of Assertions

- Judgment σ |= A means assertion holds in given state

σ |= true	always
σ |= $e_1 = e_2$	iff $\langle e_1, \sigma\rangle \Downarrow n_1$, $\langle e_2,\sigma\rangle \Downarrow n_2$ and $n_1 = n_2$
σ |= $e_1 <= e_2$	iff $\langle e_1, \sigma\rangle \Downarrow n_1$, $\langle e_2,\sigma\rangle \Downarrow n_2$ and $n_1 <= n_2$
σ |= A_1 && A_2	iff σ |= A_1 and σ |= A_2
σ |= A_1 || A_2	iff σ |= A_1 or σ |= A_2
σ |= A_1 => A_2	iff σ |= A_1 implies σ |= A_2
σ |= \exists x.A	iff for *some* n in Z. σ[x := n] |= A
σ |= \forall x. A	iff for *all* n in Z. σ[x := n] |= A

Semantics of Assertions

Formal definition of partial correctness assertion:

|= { A } c { B }

iff

forall σ in Σ. σ |= A

implies [forall σ' in Σ. $\langle c,\sigma\rangle \Downarrow \sigma'$ implies σ' |= B]

Semantics of Assertions

- Total correctness assertion:

|= [A] c [B]

iff

|= { A } c { B }

and

forall σ in Σ.

σ |= A implies [exists σ' in Σ. $\langle c,\sigma\rangle \Downarrow \sigma$]

Deriving Assertions

- Formal |= {A} **c** {B} hard to use
- Defined in terms of the op-semantics
- Next, symbolic technique (logic)
- for deriving valid triples |– {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:

$$\frac{A' \Rightarrow A \qquad \vdash \{A\}\ \mathtt{c}\ \{B\} \qquad B \Rightarrow B'}{\vdash \{A'\}\ \mathtt{c}\ \{B'\}}$$

Deriv. Rules for Hoare Logic $\vdash$ {A} `c` {B}

Rules for each language construct

$$\frac{}{\vdash \{A\}\ \mathtt{skip}\ \{A\}} \qquad \frac{\vdash \{A\}\ \mathtt{c_1}\ \{B\} \quad \vdash \{B\}\ \mathtt{c_2}\ \{C\}}{\vdash \{A\}\ \mathtt{c_1;c_2}\ \{C\}}$$

$$\frac{\vdash \{A \ \&\&\ b\}\ \mathtt{c_1}\ \{B\} \quad \vdash \{A \&\& \ !b\}\ \mathtt{c_2}\ \{B\}}{\vdash \{A\}\ \mathtt{if\ b\ then\ c_1\ else\ c_2}\ \{B\}}$$

$$\frac{\vdash \{A \ \&\&\ b\}\ \mathtt{c}\ \{A\}}{\vdash \{A\}\ \mathtt{while\ b\ do\ c}\ \{A \ \&\&\ !b\}} \qquad \frac{}{\vdash \{[e/x]A\}\ \mathtt{x:=e}\ \{A\}}$$

And the rule of consequence…

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:

  ∀x.x = x is the same as ∀y.y = y
  - Rename bound x with y

  ∀x. ∀y.x = y is the same as ∀z. ∀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] in ∀y.x = y rename y if it occurs in e'
  - Result of alpha-renaming: ∀z. e' = z
- We say that substitution avoids variable capture

  [x/z] ∀x.z = x is ?
  - ∀x.x = x   Wrong
  - ∀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 … then conseq. rule

$$\dfrac{\text{true} \Rightarrow e = e \qquad \dfrac{\texttt{x} \text{ does not appear in } \texttt{e}}{\vdash \{e = e\}\ \texttt{x:=e}\ \{x = e\}}}{\vdash \{\text{true}\}\ \texttt{x:=e}\ \{x = e\}}$$

Example: Conditional

Prove: {true} `if y<=0 then x:=1 else x:=y` {x>0}

$$\dfrac{\dfrac{\text{true} \,\&\, y{<=}0 \Rightarrow 1{>}0 \quad \vdash \{1{>}0\}\ \texttt{x:=1}\ \{x{>}0\}}{\vdash \{\text{true} \,\&\, y{<=}0\}\ \texttt{x:=1}\ \{x{>}0\}} \qquad \dfrac{\text{true} \,\&\, y{>}0 \Rightarrow y{>}0 \quad \vdash \{y{>}0\}\ \texttt{x:=y}\{x{>}0\}}{\vdash \{\text{true} \,\&\, y{>}0\}\ \texttt{x:=y}\ \{x{>}0\}}}{\vdash \{\text{true}\}\ \texttt{if y<=0 then x:=1 else x:=y}\ \{x > 0\}}$$

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

Example: Loop

- **Prove** |- {x<=0} `while x<=5 do x:=x+1` {x=6}
- **Use** the rule for while with invariant x <= 6:

$$\dfrac{\dfrac{x{<=}6 \,\&\, x{<=}5 \Rightarrow x{+}1{<=}6 \qquad \dfrac{}{\vdash \{x{+}1{<=}6\}\ \texttt{x:=x+1}\ \{x{<=}6\}}}{\vdash \{x{<=}6 \,\&\, x{<=}5\}\ \texttt{x:=x+1}\ \{x{<=}6\}}}{\vdash \{x{<=}6\}\ \texttt{while x<=5 do x:=x+1}\ \{x{<=}6 \,\&\, x{>}5\}}$$

- Finish off with consequence rule:

$$\dfrac{x{<=}0 \Rightarrow x{<=}6 \qquad \vdash \{x{<=}6\}\ \texttt{W}\ \{x{<=}6 \,\&\, x{>}5\} \qquad x{<=}6 \,\&\, x{>}5 \Rightarrow x{=}6}{\vdash \{x{<=}0\}\ \texttt{W}\ \{x = 6\}}$$

Soundness of Axiomatic Semantics

Formal Statement of Soundness:

If |- {A} `c` {B} then |= {A} `c` {B}

Equivalently

If H:: |- {A} `c` {B} then

forall σ if σ |= A and D::<`c`,σ> ⇓ σ' then σ' |= 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

|- { y >10 } **x := y** {x > 0}

|- { y >100 } **x := y** {x > 0}

|- { x=2 & y=5 } **x := y** {x > 0}

After what preconditions does postcond. x>0 hold?

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

wp($\mathbf{c}_1$;$\mathbf{c}_2$, B)
 = wp($\mathbf{c}_1$, wp($\mathbf{c}_2$, B))

$$\frac{\vdash \{A\}\ \mathbf{c}_1\ \{B\} \qquad \vdash \{B\}\ \mathbf{c}_2}{\vdash \{A\}\ \mathbf{c}_1;\ \mathbf{c}_2\ \{C\}}$$

wp(**x:=e** , B)
 = [e/x]B

$$\frac{}{\vdash \{[e/x]A\}\ \mathbf{x{:=}e}\ \{A\}}$$

wp(**if e then** $\mathbf{c}_1$ **else** $\mathbf{c}_2$, B)
 = e=>wp($\mathbf{c}_1$, B) && !e=>wp($\mathbf{c}_2$, B)

$$\frac{\vdash \{A\&b\}\ c_1\ \{B\} \qquad \vdash \{A\ \&\ !b\}\ c_2\ \{B\}}{\vdash \{A\}\ \mathtt{if}\ b\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2\ \{B\}}$$

Weakest Preconditions for Loops

Start from the equivalence

```
while b do c =
  if b then (c; while b do c) else skip
```

Let W = wp(`while b do c`, B)

It must be that: W = [b => wp(`c`, W) & !b=>B]

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

- We'll return to finding loop WPs later …

Technique: Strongest Postconditions

|- { y > 100 } `x := y` {x > 10}

|- { y > 100 } `x := y` {x > 20}

|- { y > 100 } `x := y` {x > 100}

What postcond. is guaranteed after prec. y>100 ?

SP(`c`,A): **strongest predicate** s.t. {A} `c` {SP(`c`,A)}

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

How to verify {A} `c` {B} ?

1. **Compute:** SP(`c`,A)
2. **Prove:** SP(`c`,A) => B

Strongest Postconditions

Define sp(c, B) following Hoare rules

sp(c_1;c_2, A) =
  sp(c_2, sp(c_1,A))

$$\frac{\vdash \{A\}\ c_1\ \{B\} \quad \vdash \{B\}\ c_2}{\vdash \{A\}\ c_1;\ c_2\ \{C\}}$$

sp(`x:=e` , A) =
  \exists x_0. $[x_0/x]$A && x=$[x_0/x]$e

$$\frac{}{\vdash \{[e/x]A\}\ x{:=}e\ \{A\}}$$

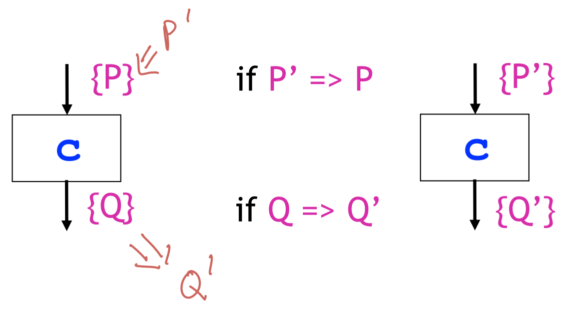
sp(`if e then` c_1 `else` c_2, A) =
  sp(c_1, A & e) || sp(c_2, A & !e)

$$\frac{\vdash \{A\&b\}\ c_1\ \{B\} \quad \vdash \{A\ \&\ !b\}\ c_2\ \{B\}}{\vdash \{A\}\ \texttt{if b then}\ c_1\ \texttt{else}\ c_2\ \{B\}}$$

Axiomatic Semantics on Flow Graphs

Floyd's Original Formulation

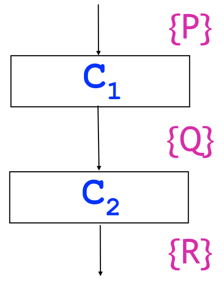
Axiomatic Semantics over Flow Graphs

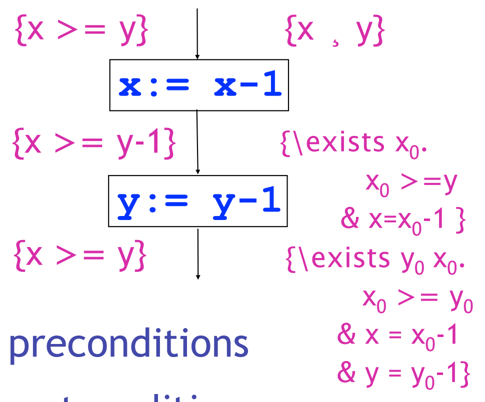


Relaxing Specifications via Consequence

Will revisit later as **subtyping**

Sequential Composition



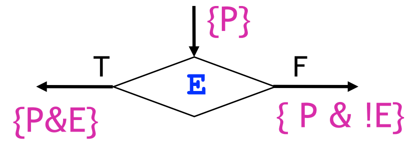


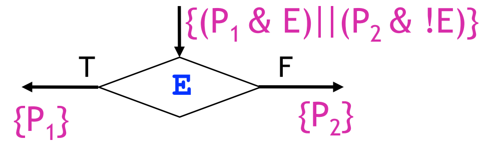
{x >= y}
x:= x-1
{x >= y-1}
y:= y-1
{x >= y}
{x ¸ y}
{\exists x0. x0 >=y & x=x0-1 }
{\exists y0 x0. x0 >= y0 & x = x0-1 & y = y0-1}

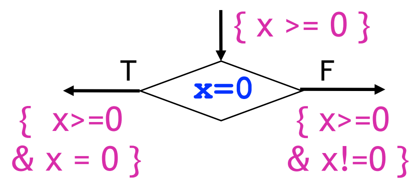
Backwards using weakest preconditions

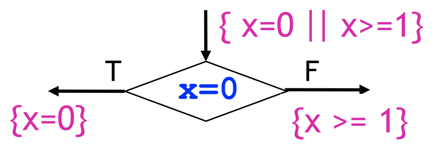
Forwards using strongest postconditions

Conditionals





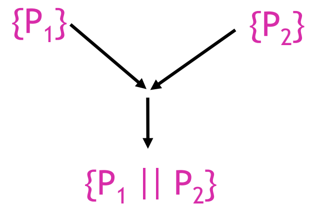


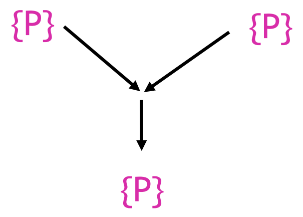


Forwards

Backwards

Joins

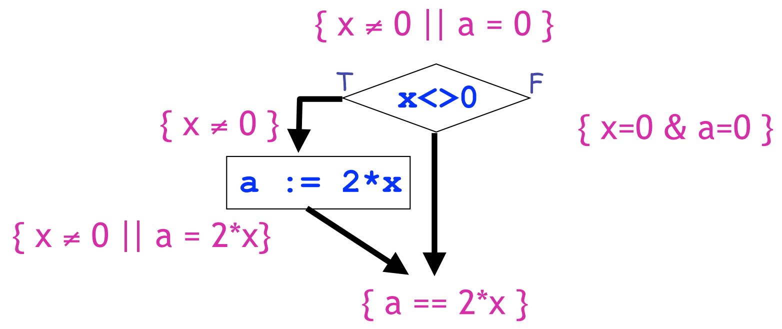




Forwards

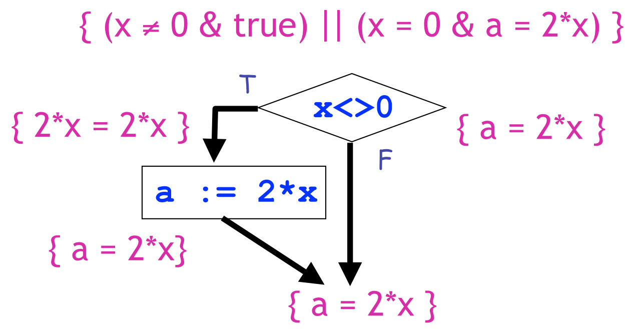
Backwards

Conditional+Join: Forward



- Check the implications (simplifications)

Conditionals+Joins: Backward

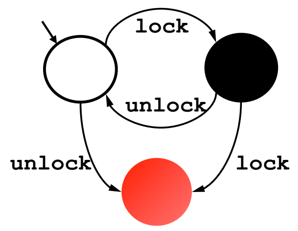


Forward or Backward ?

- 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

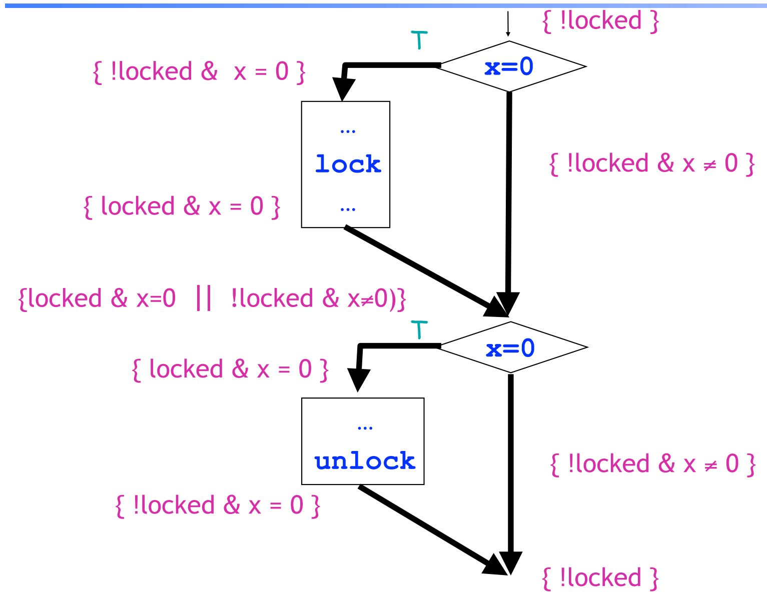
- {!locked & P[true/locked] } **`lock`** { P }

  `lock` behaves as **`assert(!locked);locked:=true`**

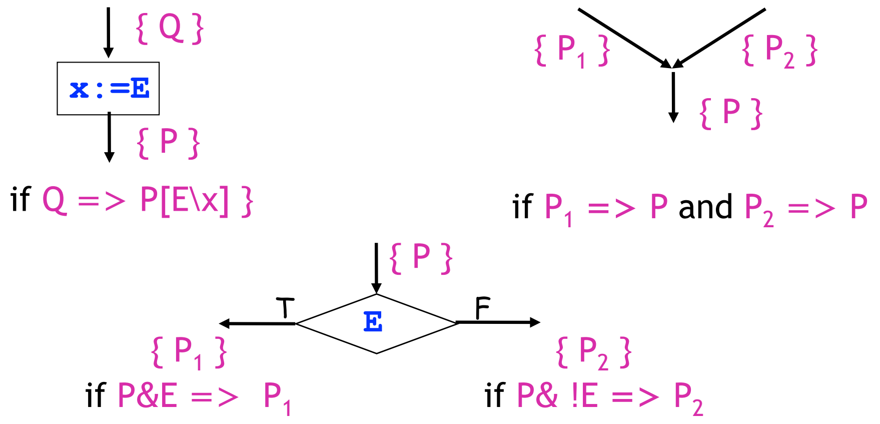
- { locked & P[false/locked] } **`unlock`** { P }

  `unlock` behaves as **`assert(locked);locked:=false`**

Locking Example



Review



Implication is always in the direction of the control flow

What about real languages ?

- Loops
- Function calls
- Pointers

Reasoning about loops: Rules

$$\frac{\vdash \{A \,\&\, b\}\ c\ \{A\}}{\vdash \{A\}\ \texttt{while b do c}\ \{A \,\&\, !b\}}$$

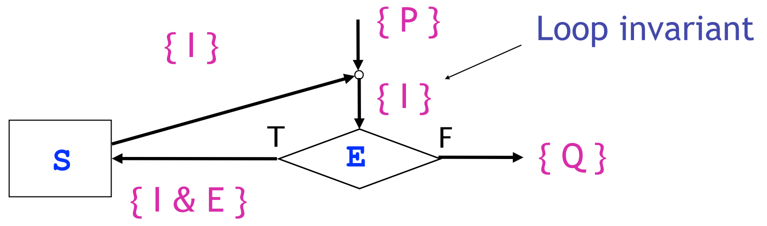
Rewrite A with I : Loop Invariant

$$\frac{P \Rightarrow I \qquad \dfrac{\vdash \{I \,\&\, b\}\ c\ \{I\}}{\vdash \{I\}\ \texttt{while b do c}\ \{I \,\&\, !b\}} \qquad I \,\&\, !b \Rightarrow Q}{\vdash \{P\}\ \texttt{while b do c}\ \{Q\}}$$

Rule of Consequence

Reasoning about loops: Flow Graphs

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

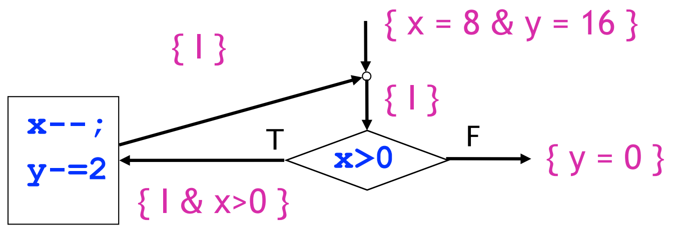


if	P => I	(loop invariant holds initially)
and	I & !b => Q	(loop establishes the postcondition)
and	{ I & b } S { I }	(loop invariant is preserved)

Loop Example

Verify:

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

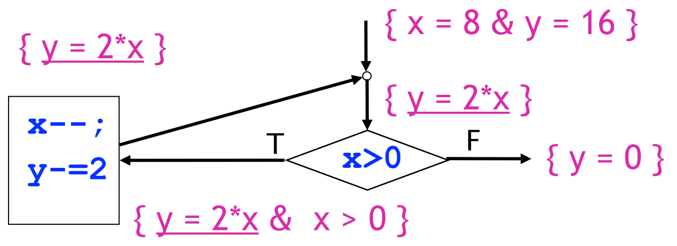


Find an appropriate invariant I

- Holds initially x = 8 & y = 16
- Holds at end y == 0

Loop Example (II)

Guess invariant y = 2*x

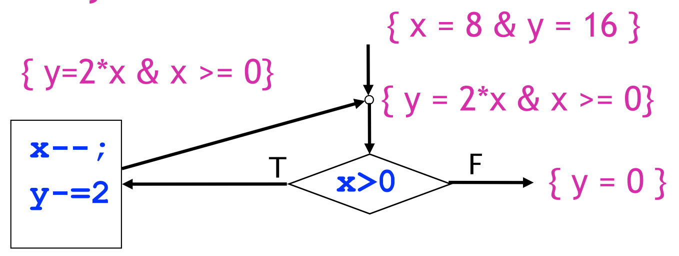


Check :

- 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 & x >= 0



Check

- Initial : x = 8 & y = 16 => y = 2*x & x >= 0
- Preserv: y = 2*x & x >= 0 & x>0 =>y-2 = 2*(x -1) & x-1 >=0
- Final: y = 2*x & x >= 0 & x <= 0 => y = 0

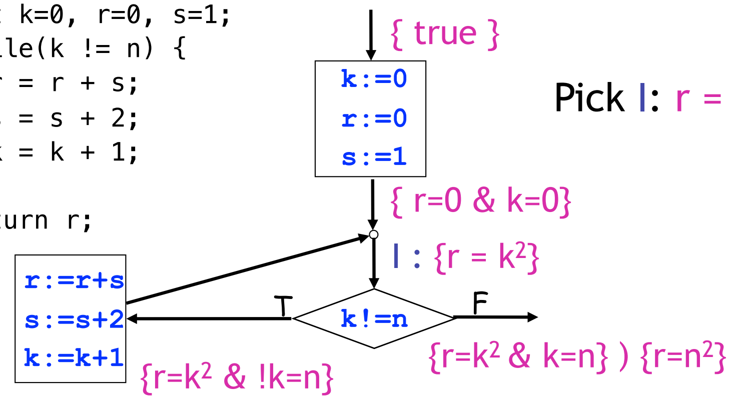
Loops Discussion

- Simple forward/backward propagation fails
- Require loop invariants
  - Hardest part of program verification
  - Guess the invariants (existing programs)
  - Write the invariants (new programs)

Note: Invariant depends on your proof goal!

Verification Example

```
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;
}
```



Pick I: $r = k^2$

Need: $\{r=k^2$ & $!k=n\}$ **c** $\{r=k^2\}$

i.e. $\{r=k^2$ & $!k=n\}$ => WP(**c**,$\{r=k^2\}$)

i.e. $\{r=k^2$ & $!k=n\}$ => $\{r+s=(k+1)^2\}$

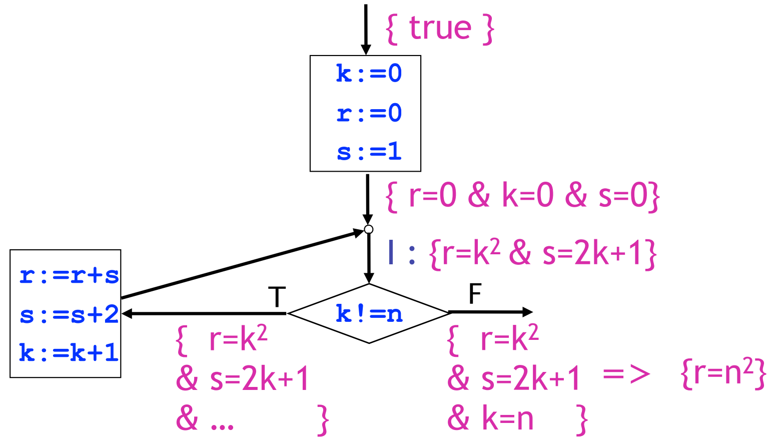
Invalid

Verification Example

Need: $\{r=k^2$ & $s=2k+1$ & …$\}$ **c** $\{r=k^2$ & $s=2k+1\}$

i.e. $\{r=k^2$ & $s=2k+1$ …$\}$ => WP(**c**, $\{r=k^2$ & $s=2k+1\}$)

i.e. $\{r=k^2$ & $s=2k+1$ …$\}$ => $\{r+s=(k+1)^2$ Æ $(s+2) = 2(k+1)+1\}$ Valid



What about real languages ?

- Loops
- Function calls
- Pointers

Functions are big instructions

Suppose we have verified **bsearch**

```
int bsearch(int a[], int p) {
    { sorted(a) }          ← Precondition "Requires"
    …
    { r=-1 || (r>=0 & r < a.length & a[r]=p)}   ← Postcondition "Ensures"
    return r;
}
```

- Function spec = precondition + postconditon
- Also called a contract

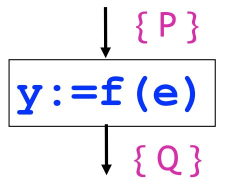
Function Calls

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

$$\frac{\vdash P \Rightarrow Pre[e/x] \quad \vdash \{Pre\}\ \mathtt{f}\ \{Post\} \quad \vdash Post[e/x,y/r] \Rightarrow Q}{\vdash \{P\}\ \mathtt{y:=f(e)}\ \{Q\}}$$

Function Calls

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



if P => Pre[E/x]

and Post[E/x,y/r] => Q

Function Call: Example

```
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){
    {y!=-1 & (y=-1 || arr[y]=5}
    {arr[y]=5}
```

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

What about real languages ?

- Loops
- Function calls
- Pointers

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)
- upd(M,E_1,E_2) : update M at address E_1 with value E_2
- sel(M,E_1) : read M at address E_1

Reason about memory expressions with McCarthy's rule

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

Assignment (update) changes the value of memory

$$\frac{}{\{B[\mathrm{upd}(M, E_1, E_2)/M]\}\ \texttt{*}E_1 \texttt{:=} E_2\ \{B\}}$$

Memory Aliasing

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

A = [upd(M, x, 5)/M] (*x+*y=10)
= [upd(M, x, 5)/M] (sel(M,x) + sel(M,y) = 10)
= sel(upd(M, x, 5), x) + sel(upd(M, x, 5), y) = 10
= 5 + sel(upd(M, x, 5), y) = 10
= sel(upd(M, x, 5), y) = 5
= (x = y & 5 = 5) || (x != y & sel(M, y) = 5)
= x=y || *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 => WP(`c`,Q)

Verification Condition

Proved By SMT Solver

VC Generation for Loops

Suppose all loops annotated with Invariant

while$_I$ b do c

Compute VC:

SMTValid(VC) **implies** |- {P} **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

vcgen :: Pred -> Com -> (Pred, [Pred])

Suppose (Q’,L’) = VCG(c,(Q,L;))

Then VC for {P} c {Q} is: $P \Rightarrow Q' \;\&\&_{\{f \text{ in } L'\}} f$

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

VCGen

```
----------------------------------------------------
verify       :: Pred -> Com -> Pred -> Bool
----------------------------------------------------

-- | The top level verifier, takes:
--   in : pre `p`, command `c` and post `q`
--   out: True iff {p} c {q} is a valid Hoare-Triple

verify            :: Pred -> Com -> Pred -> Bool
verify p c q      = all smtValid queries
  where
    (q', conds) = runState (vcgen q c) []
    queries     = p `implies` q' : conds
```

VCGen

```
vcgen :: Pred -> Com -> VC Pred

vcgen (Skip) q
  = return q

vcgen (Asgn x e) q
  = return $ q `subst` (x, e)

vcgen (If b c1 c2) q
  = do q1     <- vcgen q c1
       q2     <- vcgen q c2
       return $ (b `And` q1) `Or` (Not b `And` q2)

vcgen (While i b c) q
  = do q'     <- vcgen i c
       valid  $ (i `And` Not b) `implies` q'
       valid  $ (i `And` b)     `implies` q
       return $ i
```

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