Axiomatic Semantics

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)

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. $\langle$`c`$, \sigma \rangle \Downarrow \sigma'$

  then B holds in σ'

- Total correctness assertion: $[A]$ `c` $[B]$

  If A holds in state σ

  then there exists σ' s.t. $\langle$`c`$, \sigma \rangle \Downarrow \sigma'$ 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

$$
\begin{aligned}
A ::=\ & \text{true} \mid \text{false} \\
& \mid e_1 = e_2 \mid e_1 \leq e_2 \\
& \mid \neg A \mid A_1 \,\&\&\, A_2 \mid A_1 \,||\, A_2 \mid A_1 => A_2 \\
& \mid \backslash\text{exists } x.A \mid \backslash\text{forall } x.A
\end{aligned}
$$

- IMP boolean expressions are assertions

Semantics of Assertions

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

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

Semantics of Assertions

Formal definition of partial correctness assertion:

$\models \{ A \}\ c\ \{ B \}$

iff

forall σ in Σ. $\sigma \models A$

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

Semantics of Assertions

- Total correctness assertion:

  $\models [\,A\,]\ c\ [\,B\,]$

  iff

  $\models \{\,A\,\}\ c\ \{\,B\,\}$

  and

  forall σ in Σ.

  $\sigma \models A$ implies [exists σ' in Σ. $<c,\sigma> \Downarrow \sigma$]

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 $\vdash$ {A} `c` {B}

Derivation Rules for Hoare Triples

- Write $\vdash \{A\}\ \mathtt{c}\ \{B\}$ 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 |- {A} c {B}

Rules for each language construct

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

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

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

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:

  $\forall x.x = x$ 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. $[e'/x]$ 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 … then conseq. rule

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

Example: Conditional

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

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

- 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:

$$
\dfrac{\dfrac{x<=6 \;\&\; x<=5 \;=>\; x+1<=6 \qquad \dfrac{}{|\!-\ \{x+1<=6\}\ \mathbf{x:=x+1}\ \{x<=6\}}}{|\!-\ \{x<=6 \;\&\; x<=5\}\ \mathbf{x:=x+1}\ \{x<=6\}}}{|\!-\ \{x<=6\}\ \mathbf{while\ x<=5\ do\ x:=x+1}\ \{x<=6 \;\&\; x>5\}}
$$

- Finish off with consequence rule:

$$
\dfrac{x<=0 \;=>\; x<=6 \qquad |\!-\ \{x<=6\}\ \mathbf{W}\ \{x<=6 \;\&\; x>5\} \qquad x<=6 \;\&\; x>5 \;=>x=6}{|\!-\ \{x<=0\}\ \mathbf{W}\ \{x = 6\}}
$$

Soundness of Axiomatic Semantics

Formal Statement of Soundness:

If $\vdash \{A\}\ c\ \{B\}$ then $\models \{A\}\ c\ \{B\}$

Equivalently

If $H :: \vdash \{A\}\ c\ \{B\}$ then

forall σ if $\sigma \models A$ and $D :: \langle c, \sigma \rangle \Downarrow \sigma'$ then $\sigma' \models B$

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

$\vdash$ { y >10 } `x := y` {x > 0}

$\vdash$ { y >100 } `x := y` {x > 0}

$\vdash$ { 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** $\vdash$ {A} `c` {B} ?

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

$\mathrm{wp}(\mathtt{c_1;c_2}, B)$

$\quad = \mathrm{wp}(\mathtt{c_1}, \mathrm{wp}(\mathtt{c_2}, B))$

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

$\mathrm{wp}(\mathtt{x:=e}\ , B)$

$\quad = [e/x]B$

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

$\mathrm{wp}(\mathtt{if\ e\ then\ c_1\ else\ c_2}, B)$

$\quad = e \Rightarrow \mathrm{wp}(\mathtt{c_1}, B)\ \&\&\ !e \Rightarrow \mathrm{wp}(\mathtt{c_2}, B)$

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

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

$\mathrm{sp}(\mathbf{c_1;c_2}, A) =$

$\mathrm{sp}(\mathbf{c_2}, \mathrm{sp}(\mathbf{c_1}, A))$

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

$\mathrm{sp}(\mathbf{x:=e}\ , A) =$

\exists x_0. $[x_0/x]A$ && $x=[x_0/x]e$

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

$\mathrm{sp}(\mathbf{if\ e\ then\ c_1\ else\ c_2}, A) =$

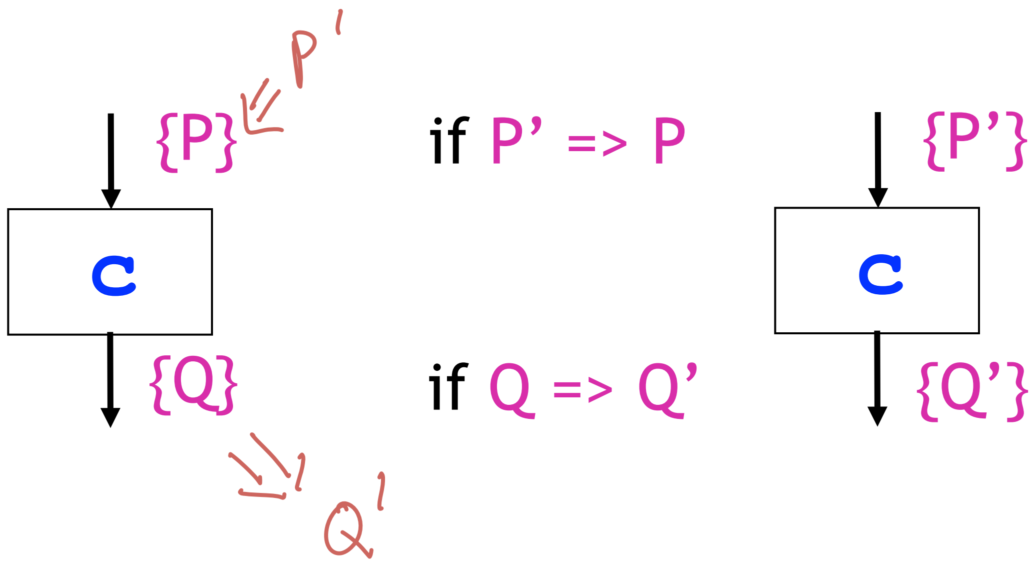
$\mathrm{sp}(\mathbf{c_1}, A\ \&\ e)\ ||\ \mathrm{sp}(\mathbf{c_2}, A\ \&\ !e)$

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

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

{P}

C_1

{Q}

C_2

{R}

{x >= y}

`x:= x-1`

{x >= y-1}

`y:= y-1`

{x >= y}

{x ¸ y}

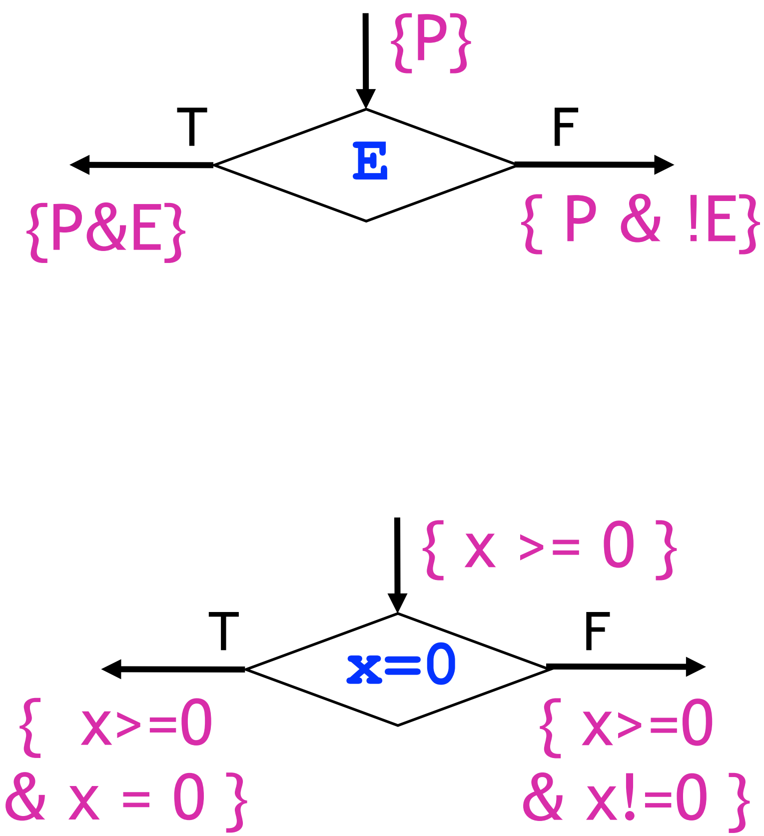
{\exists x_0.
x_0 >=y
& x=x_0-1 }

{\exists y_0 x_0.
x_0 >= y_0
& x = x_0-1
& y = y_0-1}

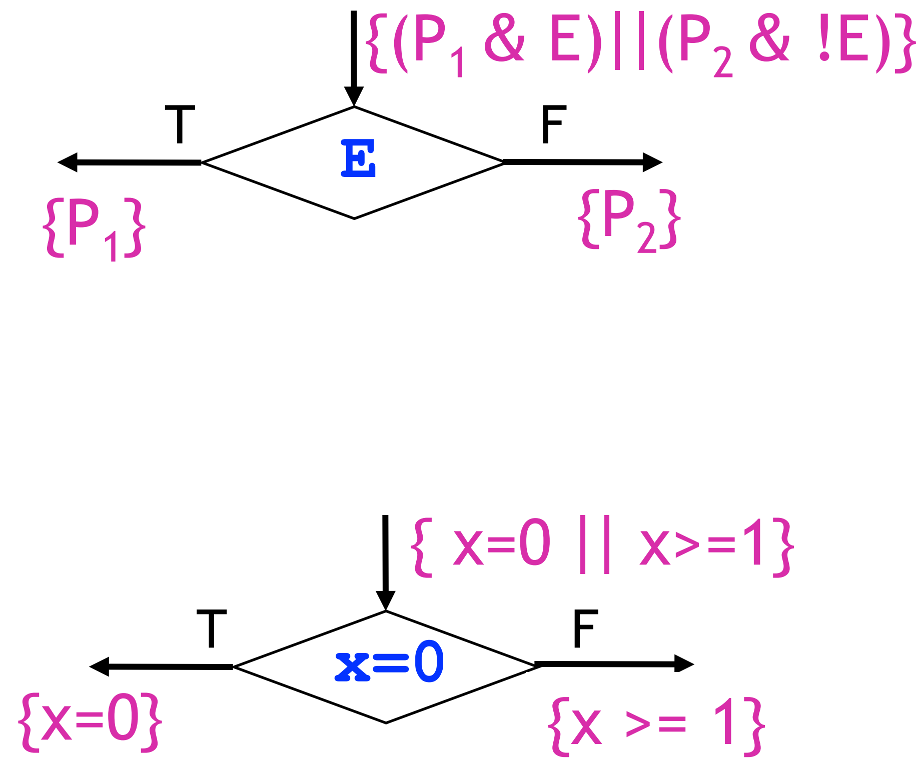
Backwards using weakest preconditions

Forwards using strongest postconditions

Conditionals

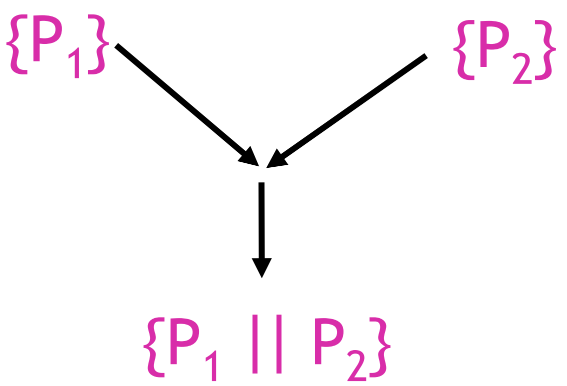


Forwards

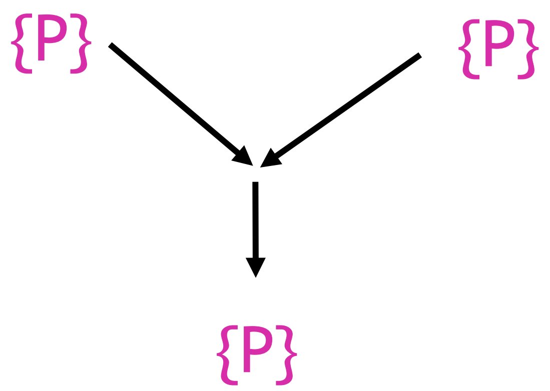


Backwards

Joins

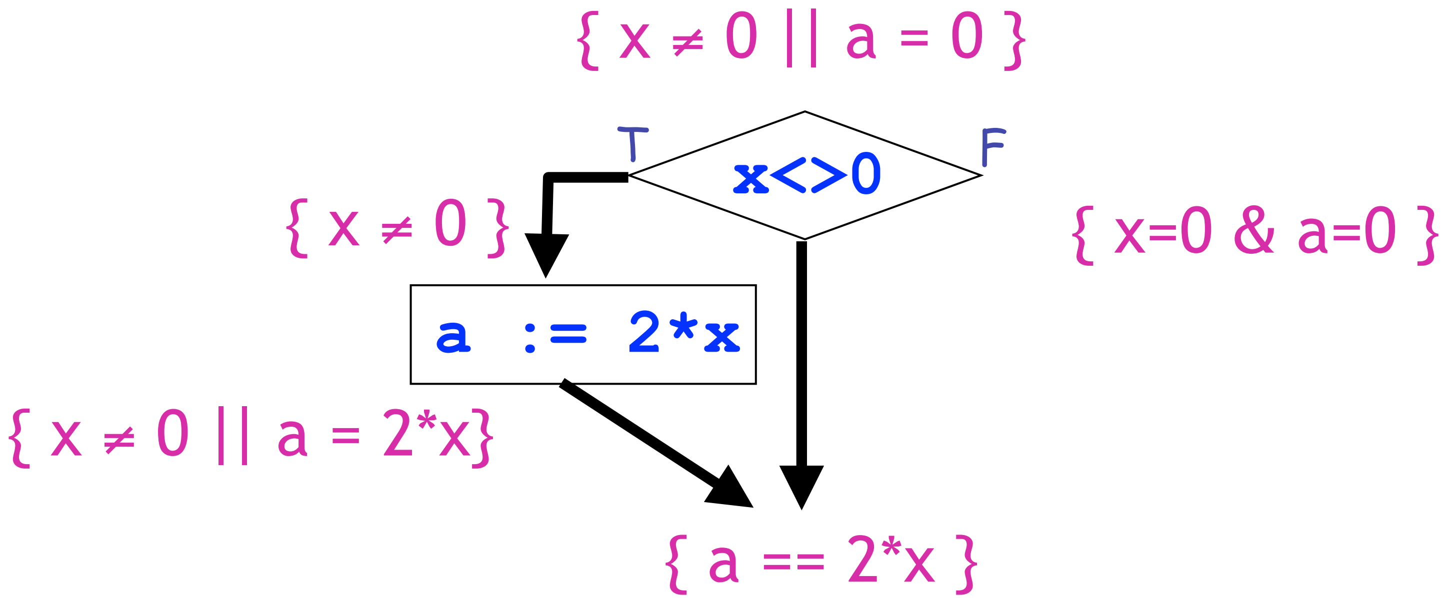


Forwards



Backwards

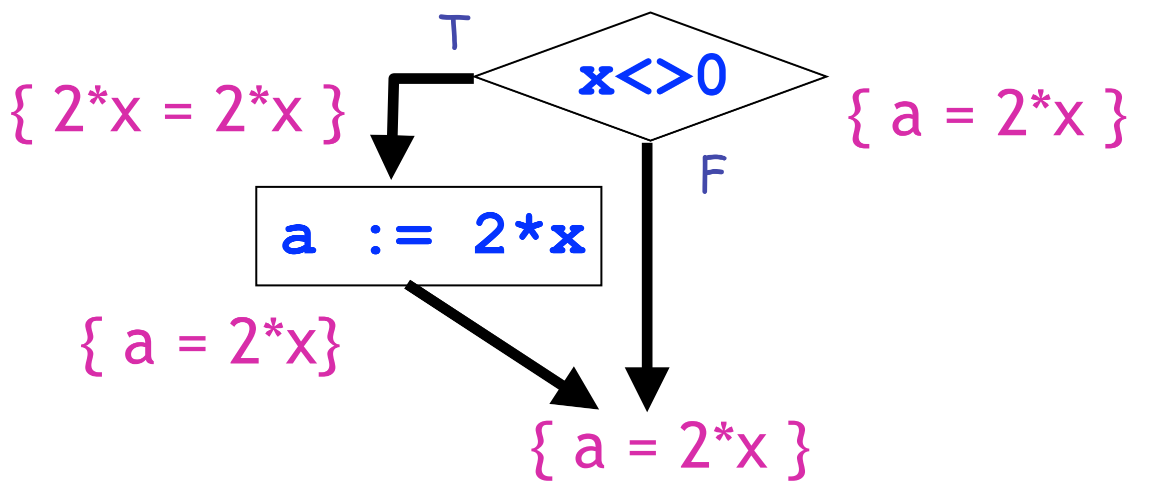
Conditional+Join: Forward



- Check the implications (simplifications)

Conditionals+Joins: Backward

{ (x ≠ 0 & true) || (x = 0 & a = 2*x) }

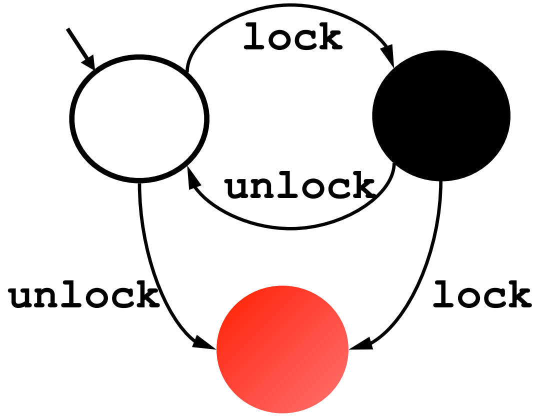


Forward or Backward ?

- 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

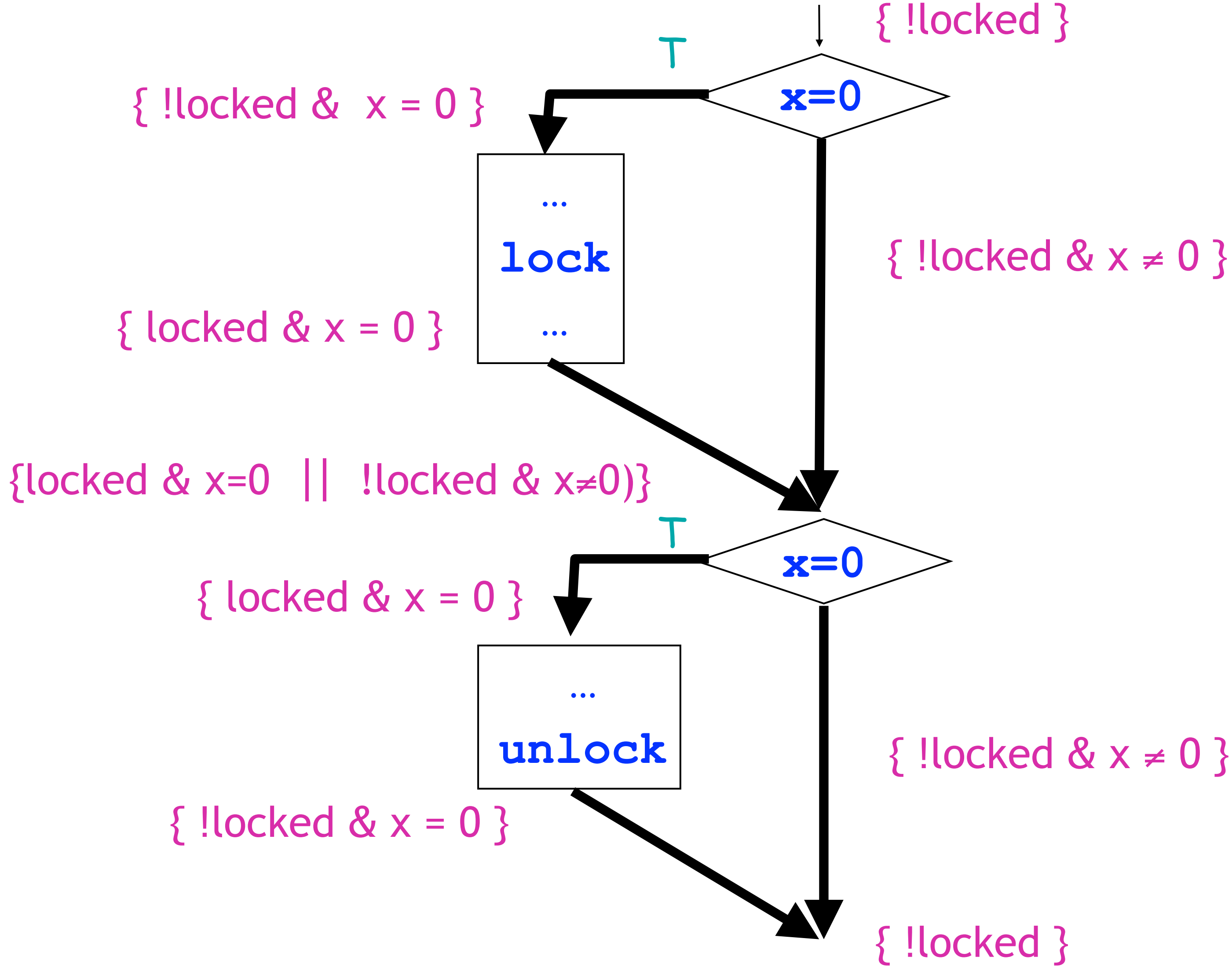
- {!locked & P[true/locked] } `lock` { P }

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

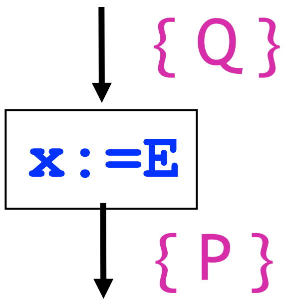
- { locked & P[false/locked] } `unlock` { P }

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

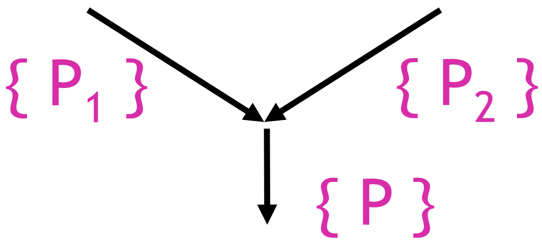
Locking Example



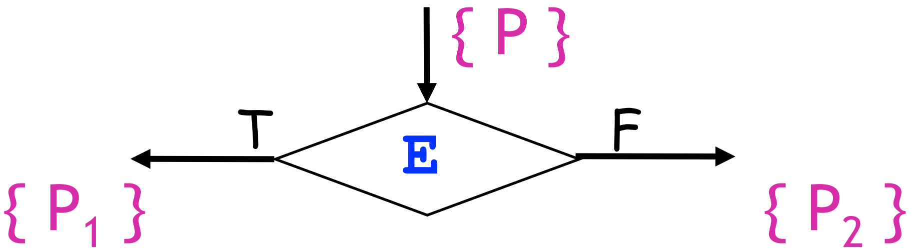
Review



if $Q \Rightarrow P[E\backslash x]$ }



if $P_1 \Rightarrow P$ and $P_2 \Rightarrow P$



if $P \& E \Rightarrow P_1$

if $P \& \, !E \Rightarrow P_2$

Implication is always in the direction of the control flow

What about real languages ?

- Loops
- Function calls
- Pointers

Reasoning about loops: Rules

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

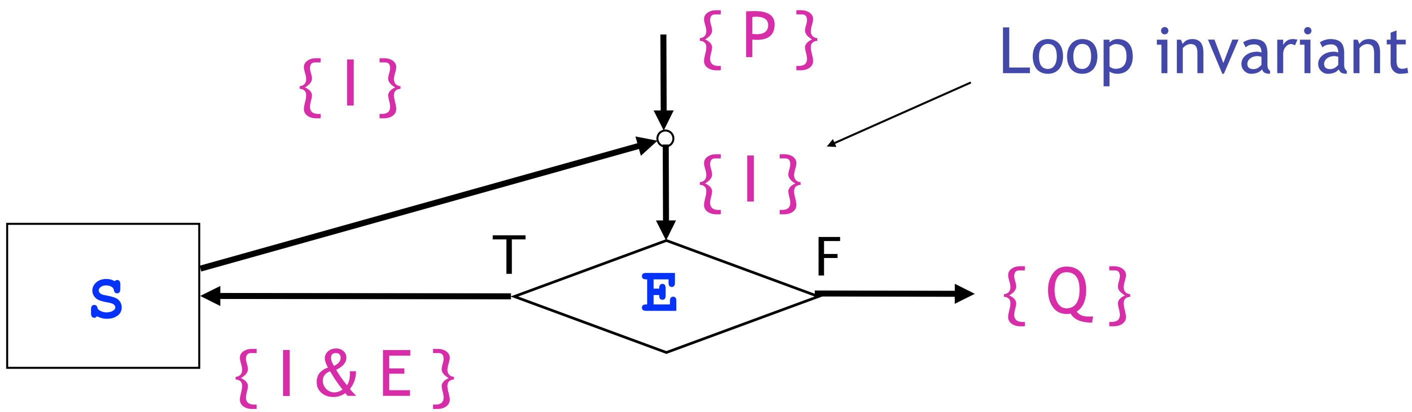
Rewrite A with I : Loop Invariant

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

Rule of Consequence

Reasoning about loops: Flow Graphs

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

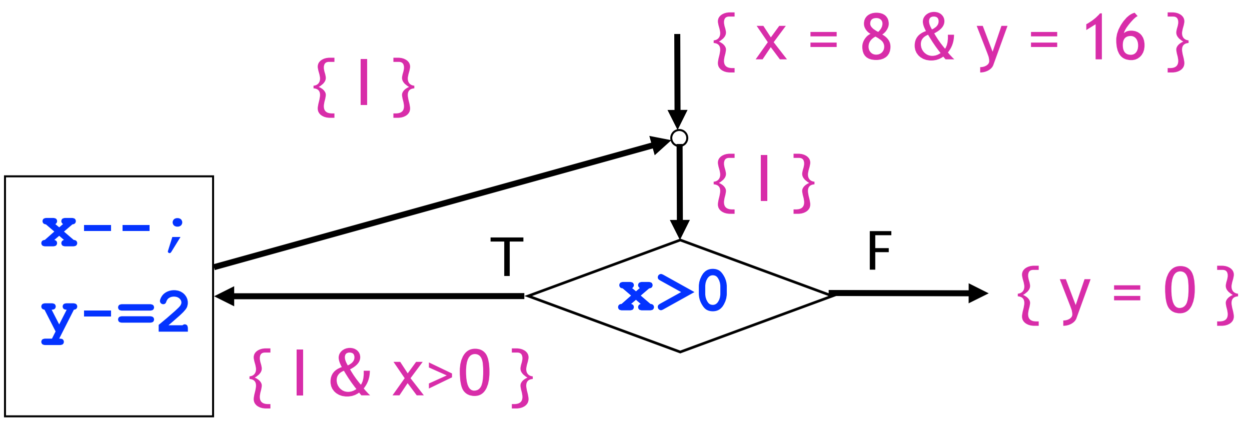


if	P => I	(loop invariant holds initially)
and	I & !b => Q	(loop establishes the postcondition)
and	{ I & b } **S** { I }	(loop invariant is preserved)

Loop Example

Verify:

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

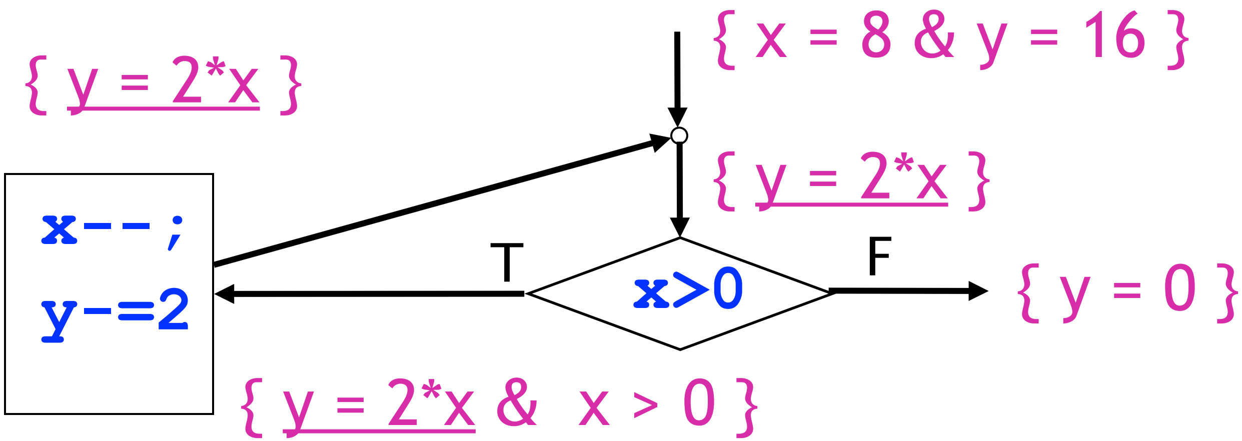


Find an appropriate invariant I

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

Loop Example (II)

Guess invariant $y = 2*x$

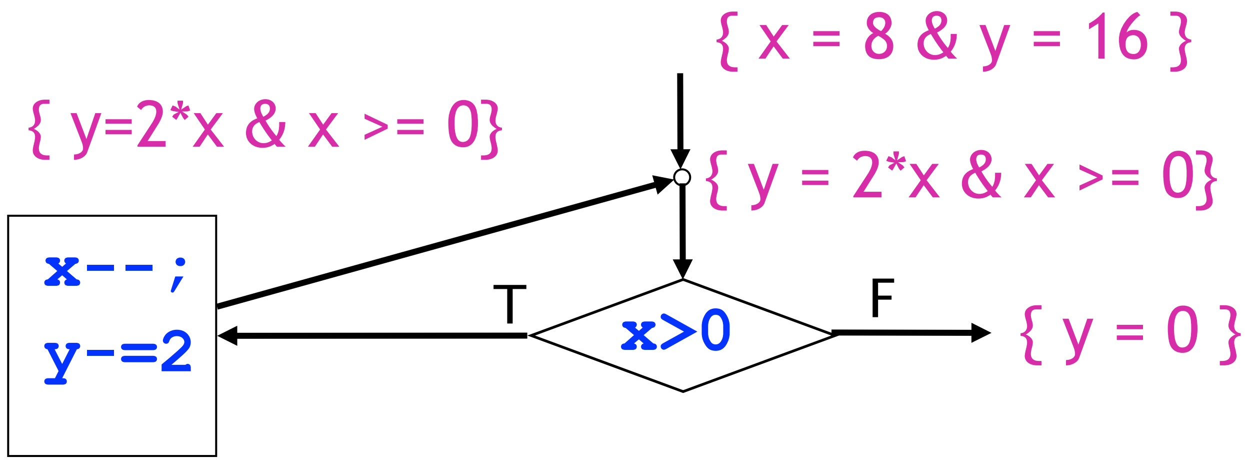


Check :

- Initial: $x = 8 \ \& \ y = 16 \implies y = 2*x$
- Preservation: $y = 2*x \ \& \ x>0 \implies y-2 = 2*(x-1)$
- Final: $y = 2*x \ \& \ x<=0 \implies y = 0$ 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

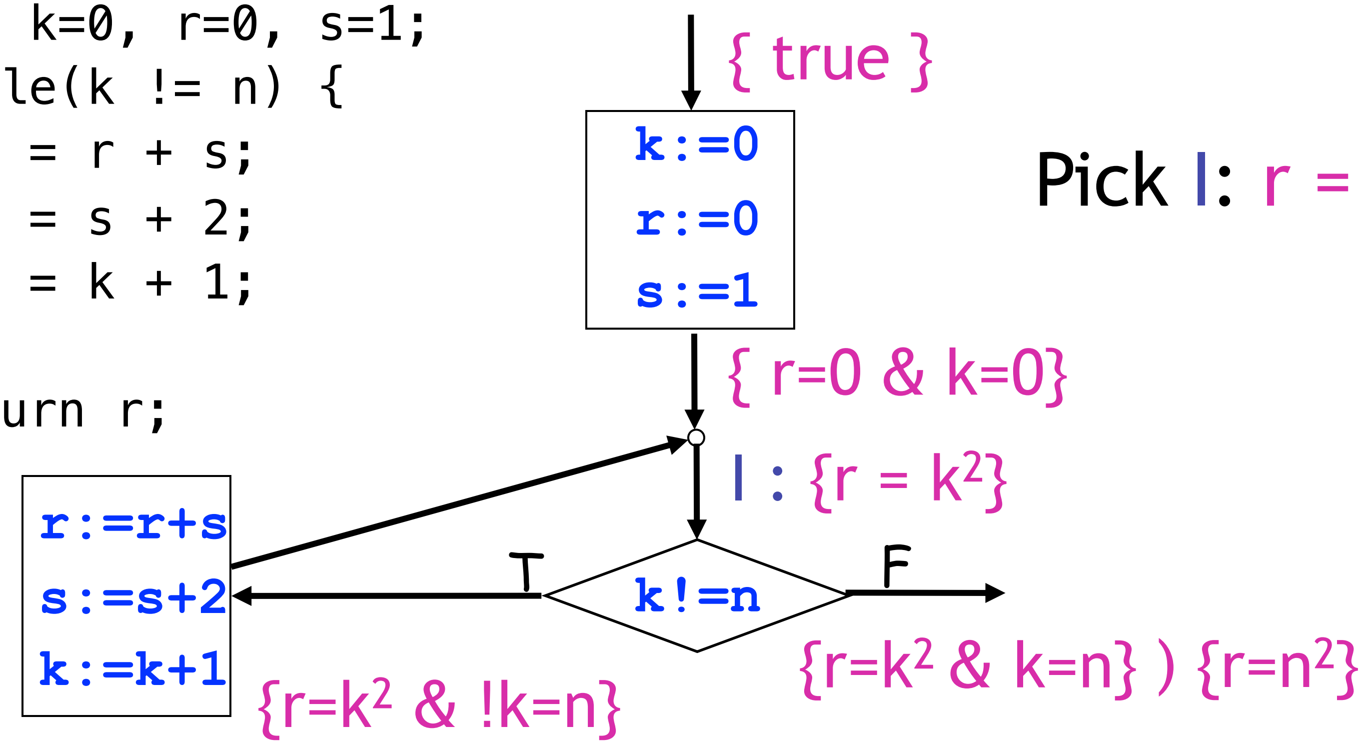
- Simple forward/backward propagation fails

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

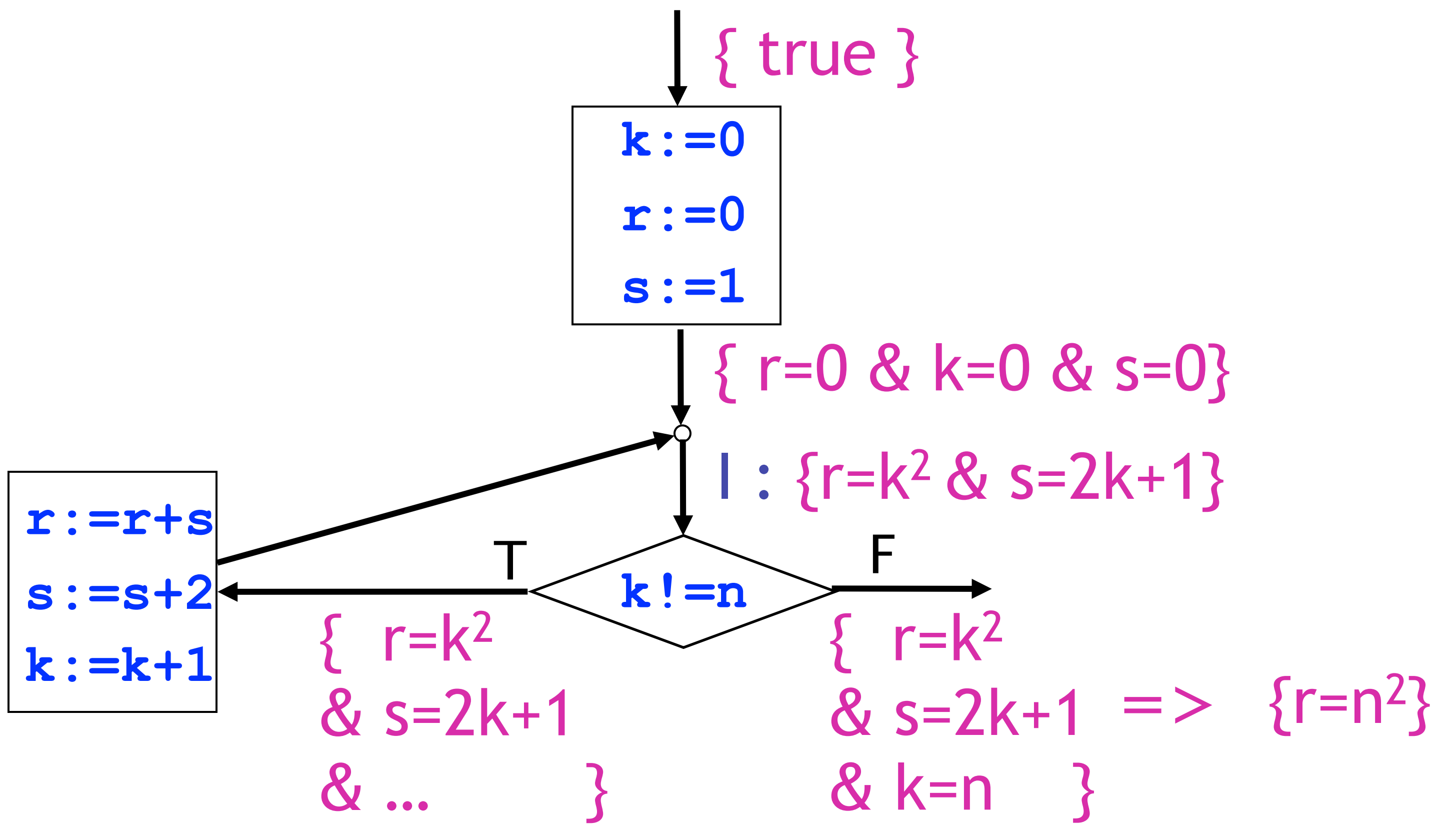
Invalid

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) }
    …
    { r=-1 || (r>=0 & r < a.length & a[r]=p)}
    return r;
}
```

Precondition
"Requires"

Postcondition
"Ensures"

- 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:

{ P } if P => Pre[E/x]

`y:=f(e)`

{ Q } 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

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

Assignment (update) changes the value of memory

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

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