Floyd-Hoare Logic

Ranjit Jhala
UC San Diego

Axiomatic Semantics

Axiomatic Semantics

1. Language for making assertions about programs

Axiomatic Semantics

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

Axiomatic Semantics

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

Axiomatic Semantics

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

Typical kinds of assertions:

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.

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`

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

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

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:

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

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

IMP Syntactic Entities

- Int integer literals `n`

IMP Syntactic Entities

- Int integer literals `n`

IMP Syntactic Entities

• Int	integer literals	**`n`**
• Bool	booleans	**`{true,false}`**

IMP Syntactic Entities

• Int	integer literals	**`n`**
• Bool	booleans	**`{true,false}`**
• Loc	locations	**`x,y,z,…`**

IMP Syntactic Entities

- Int integer literals **n**
- Bool booleans **{true,false}**
- Loc locations **x,y,z,…**

IMP Syntactic Entities

• Int	integer literals	**n**
• Bool	booleans	**{true,false}**
• Loc	locations	**x,y,z,…**
• Aexp	arithmetic expressions	**e**

IMP Syntactic Entities

Int	integer literals	**`n`**
Bool	booleans	**`{true,false}`**
Loc	locations	**`x,y,z,…`**
Aexp	arithmetic expressions	**`e`**

IMP Syntactic Entities

Int	integer literals	**`n`**
Bool	booleans	**`{true,false}`**
Loc	locations	**`x,y,z,…`**
Aexp	arithmetic expressions	**`e`**
Bexp	boolean expressions	**`b`**

IMP Syntactic Entities

• Int	integer literals	**`n`**
• Bool	booleans	**`{true,false}`**
• Loc	locations	**`x,y,z,…`**
• Aexp	arithmetic expressions	**`e`**
• Bexp	boolean expressions	**`b`**

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)

$\mathbf{e} ::=$	$\mathbf{n}$	for $\mathbf{n} \in \mathrm{Int}$
	$\mid \mathbf{x}$	for $\mathbf{x} \in \mathrm{Loc}$
	$\mid \mathbf{e}_1 + \mathbf{e}_2$	for $\mathbf{e}_1, \mathbf{e}_2 \in \mathrm{Aexp}$
	$\mid \mathbf{e}_1 - \mathbf{e}_2$	for $\mathbf{e}_1, \mathbf{e}_2 \in \mathrm{Aexp}$
	$\mid \mathbf{e}_1 * \mathbf{e}_2$	for $\mathbf{e}_1, \mathbf{e}_2 \in \mathrm{Aexp}$

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

Abstract Syntax: Bool Expressions (Bexp)

$$
\begin{array}{lll}
\texttt{true} ::= & \texttt{true} & \\
 & \mid\ \texttt{false} & \\
 & \mid\ \texttt{e}_1 = \texttt{e}_2 & \text{for } \texttt{e}_1, \texttt{e}_2 \in \text{Aexp} \\
 & \mid\ \texttt{e}_1 < \texttt{e}_2 & \text{for } \texttt{e}_1, \texttt{e}_2 \in \text{Aexp} \\
 & \mid\ !\,\texttt{b} & \text{for } \texttt{b} \in \text{Bexp} \\
 & \mid\ \texttt{b}_1 \mid\mid \texttt{b}_2 & \text{for } \texttt{e}_1,\ \texttt{e}_2 \in \text{Bexp} \\
 & \mid\ \texttt{b}_1\ \&\ \texttt{b}_2 & \text{for } \texttt{e}_1,\ \texttt{e}_2 \in \text{Bexp}
\end{array}
$$

Abstract Syntax: Commands (Comm)

Abstract Syntax: Commands (Comm)

`c ::= skip`	
`| x:= e`	for $\mathbf{x} \in$ L & $\mathbf{e} \in$ Aexp
`|` $c_1;c_2$	for $c_1, c_2 \in$ Comm
`| if b then` c_1 `else` c_2	for $\mathbf{b} \in$ Bexp & $c_1, c_2 \in$ Comm

Abstract Syntax: Commands (Comm)

c ::= **skip**

| $x := e$ for $x \in L$ & $e \in \text{Aexp}$

| $c_1 ; c_2$ 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 $c \in \text{Comm}$ & $b \in \text{Bexp}$

Abstract Syntax: Commands (Comm)

$\mathbf{c}$ ::= **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

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:

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)

Abstract Syntax: Commands (Comm)

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

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

Operational Semantics of IMP

Evaluation judgment for expressions:

- Ternary relation on expression, a state, and a value:

Operational Semantics of IMP

Evaluation judgment for expressions:

- Ternary relation on expression, a state, and a value:
- We write: $<\mathtt{e}, \sigma> \Downarrow n$

  "Expression $\mathtt{e}$ in state σ evaluates to n"

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 $\mathbf{e}$ in state σ evaluates to n"

Q: Why no state on the right ?

- Evaluation of expressions has no side-effects:

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 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

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

Operational Semantics of IMP

Evaluation judgment for expressions:

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

  "Expression $\mathtt{e}$ 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 $\mathtt{e}$, σ ?

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 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

Operational Semantics of IMP

Evaluation judgement for commands

- Ternary relation on expression, state, and a new state

Operational Semantics of IMP

Evaluation judgement for commands

- Ternary relation on expression, state, and a new state
- We write: $<\mathbf{c}, \sigma> \Downarrow \sigma'$

  "Executing cmd $\mathbf{c}$ from state σ takes system into state σ'"

Operational Semantics of IMP

Evaluation judgement for commands

- Ternary relation on expression, state, and a new state
- We write: $<\texttt{c}, \sigma> \Downarrow \sigma'$

  "Executing cmd $\texttt{c}$ 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'

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 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'

Operational Semantics of IMP

Evaluation judgement for commands

- Ternary relation on expression, state, and a new state
- We write: $<\mathtt{c}, \sigma> \Downarrow \sigma'$

  "Executing cmd $\mathtt{c}$ 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

Operational Semantics of IMP

Evaluation judgement for commands

- Ternary relation on expression, state, and a new state
- We write: $<\mathtt{c}, \sigma> \Downarrow \sigma'$

  "Executing cmd $\mathtt{c}$ 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

Operational Semantics of IMP

Evaluation judgement for commands

- Ternary relation on expression, state, and a new state
- We write: $<\mathtt{c}, \sigma> \Downarrow \sigma'$

  "Executing cmd $\mathtt{c}$ 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 $\mathtt{e}$, σ ?

- Only if there is a unique successor state …

Evaluation Rules (for Aexp)

Evaluation Rules (for Aexp)

$$\frac{}{\langle \mathbf{n}, \sigma \rangle \Downarrow n}$$

Evaluation Rules (for Aexp)

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

Evaluation Rules (for Aexp)

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

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

Evaluation Rules (for Aexp)

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

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

$$\frac{\langle \mathtt{e}_1, \sigma\rangle \Downarrow n_1 \qquad \langle \mathtt{e}_2, \sigma\rangle \Downarrow n_2}{\langle \mathtt{e}_1 \;\mathtt{*}\; \mathtt{e}_2, \sigma\rangle \Downarrow n_1 * n_2}$$

Evaluation Rules (for Bexp)

Evaluation Rules (for Bexp)

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

Evaluation Rules (for Bexp)

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

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

Evaluation Rules (for Bexp)

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

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

Evaluation Rules (for Bexp)

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

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

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

Evaluation Rules (for Bexp)

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

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

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

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

Evaluation Rules (for Comm)

Evaluation Rules (for Comm)

$$\frac{}{\texttt{<skip}\,,\,\sigma\texttt{>} \Downarrow \sigma}$$

Evaluation Rules (for Comm)

$$\frac{}{\texttt{<skip}, \sigma\texttt{>} \Downarrow \sigma}$$

$$\frac{\texttt{<}c_1, \sigma\texttt{>} \Downarrow \sigma' \quad \texttt{<}c_2, \sigma'\texttt{>} \Downarrow \sigma''}{\texttt{<}c_1; c_2, \sigma\texttt{>} \Downarrow \sigma''}$$

Evaluation Rules (for Comm)

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

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

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

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

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

Evaluation Rules (for Comm)

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

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

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

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

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

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

Evaluation Rules (for Comm)

Evaluation Rules (for Comm)

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

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

Axiomatic Semantics

1. Language for making assertions about programs

Axiomatic Semantics

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

Axiomatic Semantics

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

Axiomatic Semantics

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

Typical kinds of assertions:

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.

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`

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

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

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:

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

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

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'

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

Hoare Triples

- Partial correctness assertion: $\{A\}\ \mathtt{c}\ \{B\}$

  If A holds in state σ and exists σ' s.t. $<\mathtt{c}, \sigma> \Downarrow \sigma'$

  then B holds in σ'

- Total correctness assertion: $[A]\ \mathtt{c}\ [B]$

  If A holds in state σ

  then there exists σ' s.t. $<\mathtt{c}, \sigma> \Downarrow \sigma'$ *and B holds in σ'*

- [A] is called precondition, [B] is called postcondition

Hoare Triples

- Partial correctness assertion: {A} `c` {B}

  If A holds in state σ and exists σ' s.t. $<$`c`$, \sigma > \Downarrow \sigma'$

  then B holds in σ'

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

  If A holds in state σ

  then there exists σ' s.t. $<$`c`$, \sigma > \Downarrow \sigma'$ *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

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

- 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

Semantics of Assertions

Formal definition of partial correctness assertion:

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

iff

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

Semantics of Assertions

- Total correctness assertion:

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

Semantics of Assertions

- Total correctness assertion:

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

  iff

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

Semantics of Assertions

- Total correctness assertion:

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

  iff

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

  and

Semantics of Assertions

- Total correctness assertion:

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

  iff

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

  and

  forall σ in Σ.

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. $\langle c,\sigma\rangle \Downarrow \sigma$]

Deriving Assertions

Deriving Assertions

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

Deriving Assertions

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

Deriving Assertions

- Formal $\models$ {A} `c` {B} hard to use
- Defined in terms of the op-semantics

Deriving Assertions

- Formal $\models$ {A} `c` {B} hard to use
- Defined in terms of the op-semantics
- Next, symbolic technique (logic)

Deriving Assertions

- Formal $\models$ {A} `c` {B} hard to use
- Defined in terms of the op-semantics
- Next, symbolic technique (logic)

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

Derivation Rules for Hoare Triples

- Write $\vdash \{A\}$ `c` $\{B\}$ when we can derive the triple using derivation rules

Derivation Rules for Hoare Triples

- Write $\vdash \{A\}\ c\ \{B\}$ when we can derive the triple using derivation rules

Derivation Rules for Hoare Triples

- Write $\vdash$ {A} `c` {B} when we can derive the triple using derivation rules

- **One rule** per command

Derivation Rules for Hoare Triples

- Write |– {A} `c` {B} when we can derive the triple using derivation rules

- **One rule** per command

Derivation Rules for Hoare Triples

- Write |− {A} `c` {B} when we can derive the triple using derivation rules

- **One rule** per command

- Plus, the rule of consequence:

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:

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

Deriv. Rules for Hoare Logic |− {A} c {B}

Rules for each language construct

Deriv. Rules for Hoare Logic |- {A} c {B}

Rules for each language construct

$$\frac{}{\vdash \{A\}\ \texttt{skip}\ \{A\}}$$

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

Rules for each language construct

$$\frac{}{\vdash \{A\}\ \texttt{skip}\ \{A\}}$$

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

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

Rules for each language construct

$$\frac{}{\vdash \{A\}\ \texttt{skip}\ \{A\}}$$

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

$$\frac{\vdash \{A \mathbin{\&\&} b\}\ \texttt{c}_1\ \{B\} \qquad \vdash \{A \mathbin{\&\&} !b\}\ \texttt{c}_2\ \{B\}}{\vdash \{A\}\ \texttt{if b then c}_1\ \texttt{else c}_2\ \{B\}}$$

Deriv. Rules for Hoare Logic |- {A} c {B}

Rules for each language construct

$$\frac{}{\vdash \{A\}\ \texttt{skip}\ \{A\}}$$

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

$$\frac{\vdash \{A\ \&\&\ b\}\ \texttt{c}_1\ \{B\} \qquad \vdash \{A\&\&\ !b\}\ \texttt{c}_2\ \{B\}}{\vdash \{A\}\ \texttt{if b then c}_1\ \texttt{else c}_2\ \{B\}}$$

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

Deriv. Rules for Hoare Logic |- {A} c {B}

Rules for each language construct

$$\frac{}{\vdash \{A\}\ \texttt{skip}\ \{A\}}$$

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

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

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

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

Deriv. Rules for Hoare Logic |- {A} c {B}

Rules for each language construct

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

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

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

And the rule of consequence…

Free and Bound Variables

Free and Bound Variables

Key idea in logic/PL: scoping & substitution

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)

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 $\forall y. y = y$

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

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 $\forall z.\ \forall x. z = x$

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

Substitution

- [e'/x] e is substituting e' for x in e
  - Also written as e[e'/x]

Substitution

- [e’/x] e is substituting e’ for x in e
  - Also written as e[e’/x]
  - Note: only substitute the free occurrences

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'

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: $\forall z.\ e' = z$

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

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 ?

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

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

Example: Assignment

Assume `x` does not appear in `e`

Prove $\vdash$ {true} `x:=e` { x = e }

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

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

Example: Assignment

Assume `x` does not appear in `e`

Prove $\vdash$ {true} `x:=e` { x = e }

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

Use assignment rule … then conseq. rule

Example: Assignment

Assume $\mathbf{x}$ does not appear in $\mathbf{e}$

Prove |- {true} **x:=e** { x = e }

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

Use assignment rule … then conseq. rule

$$\frac{\mathbf{x} \text{ does not appear in } \mathbf{e}}{\quad}$$

Example: Assignment

Assume $\mathbf{x}$ does not appear in $\mathbf{e}$

Prove |- {true} $\mathbf{x:=e}$ { 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}

Example: Conditional

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

$$\frac{\vdash \{true \;\&\; y<=0\}\ \mathtt{x:=1}\ \{x>0\} \qquad \vdash \{true \;\&\; y>0\}\ \mathtt{x:=y}\ \{x>0\}}{\vdash \{true\}\ \mathtt{if\ y<=0\ then\ x:=1\ else\ x:=y}\ \{x > 0\}}$$

Example: Conditional

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

$$\frac{\vdash \{true \;\&\; y<=0\}\ \texttt{x:=1}\ \{x>0\} \qquad \vdash \{true \;\&\; y>0\}\ \texttt{x:=y}\ \{x>0\}}{\vdash \{true\}\ \texttt{if y<=0 then x:=1 else x:=y}\ \{x > 0\}}$$

- Rule for if-then-else

Example: Conditional

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

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

- Rule for if-then-else

Example: Conditional

Prove: $\{\mathrm{true}\}\ \texttt{if y<=0 then x:=1 else x:=y}\ \{x>0\}$

$$
\dfrac{
\dfrac{\mathrm{true}\ \&\ y<=0\ =>1>0 \quad |\!-\ \{1>0\}\ \texttt{x:=1}\ \{x>0\}}
{|\!-\ \{\mathrm{true}\ \&\ y<=0\}\ \texttt{x:=1}\ \{x>0\}}
\qquad
|\!-\ \{\mathrm{true}\ \&\ y>0\}\ \texttt{x:=y}\ \{x>0\}
}
{|\!-\ \{\mathrm{true}\}\ \texttt{if y<=0 then x:=1 else x:=y}\ \{x > 0\}}
$$

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

Example: Conditional

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

$$
\dfrac{\dfrac{true \,\&\, y<=0 \;=>1>0 \quad |\!-\; \{1>0\}\ \mathtt{x:=1}\ \{x>0\}}{|\!-\; \{true \,\&\, y<=0\}\ \mathtt{x:=1}\ \{x>0\}} \qquad \dfrac{true \,\&\, y>0 \;=> y>0 \quad |\!-\; \{y>0\}\ \mathtt{x:=y}\{x>0\}}{|\!-\; \{true \,\&\, y>0\}\ \mathtt{x:=y}\ \{x>0\}}}{|\!-\; \{true\}\ \mathtt{if\ y<=0\ then\ x:=1\ else\ x:=y}\ \{x > 0\}}
$$

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

Example: Loop

Example: Loop

- **Prove** |- {x<=0} `while x<=5 do x:=x+1` {x=6}

Example: Loop

- **Prove** |- {x<=0} `while x<=5 do x:=x+1` {x=6}

|- {x<=6} `while x<=5 do x:=x+1` {x<=6 & x>5}

Example: Loop

- **Prove** |- {x<=0} `while x<=5 do x:=x+1` {x=6}

$$\frac{\vdash \{x \le 6 \;\&\; x \le 5\}\ \texttt{x:=x+1}\ \{x \le 6\}}{\vdash \{x \le 6\}\ \texttt{while x<=5 do x:=x+1}\ \{x \le 6 \;\&\; x > 5\}}$$

Example: Loop

- **Prove** |- {x<=0} `while x<=5 do x:=x+1` {x=6}

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

Example: Loop

- **Prove** |- {x<=0} `while x<=5 do x:=x+1` {x=6}

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

Example: Loop

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

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

Example: Loop

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

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

Example: Loop

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

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

Example: Loop

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

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

Example: Loop

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

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

Example: Loop

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

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

- Finish off with consequence rule:

Example: Loop

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

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

- Finish off with consequence rule:

$$\vdash \{x<=6\}\ \mathtt{W}\ \{x<=6 \ \& \ x>5\}$$

Example: Loop

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

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

- Finish off with consequence rule:

x<=0 => x<=6 |- {x<=6} **`W`** {x<=6 & x>5}

Example: Loop

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

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

- Finish off with consequence rule:

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

Example: Loop

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

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

- Finish off with consequence rule:

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

Soundness of Axiomatic Semantics

Formal Statement of Soundness:

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

Algorithmic Verification

Hoare rules mostly syntax directed, but:

Algorithmic Verification

Hoare rules mostly syntax directed, but:

Algorithmic Verification

Hoare rules mostly syntax directed, but:

1. When to apply the rule of consequence ?

Algorithmic Verification

Hoare rules mostly syntax directed, but:

1. When to apply the rule of consequence ?
2. What invariant to use for while ?

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)?

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:

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

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

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

Technique: Weakest Preconditions

After what preconditions does postcond. `x>0` hold?

Technique: Weakest Preconditions

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

After what preconditions does postcond. $x>0$ hold?

Technique: Weakest Preconditions

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

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

After what preconditions does postcond. $x>0$ hold?

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?

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)

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} ?

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

Weakest Preconditions

Define wp(c, B) using Hoare rules

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

wp(c_1;c_2, B)

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

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

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

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

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

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

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

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

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

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

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

$\text{wp}(\mathbf{x := e}\ , B)$

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

$\mathrm{wp}(\mathtt{c}_1 \mathtt{;} \mathtt{c}_2, B)$

$= \mathrm{wp}(\mathtt{c}_1, \mathrm{wp}(\mathtt{c}_2, B))$

$$\frac{\vdash \{A\}\ \mathtt{c}_1\ \{B\} \quad \vdash \{B\}\ \mathtt{c}_2}{\vdash \{A\}\ \mathtt{c}_1\mathtt{;}\ \ \mathtt{c}_2\ \{C\}}$$

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

$= [e/x]B$

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

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

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

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

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

$= [e/x]B$

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

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

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

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

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

$= [e/x]B$

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

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

Weakest Preconditions

Define wp(c, B) using Hoare rules

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

$= wp(\mathbf{c_1}, wp(\mathbf{c_2}, B))$

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

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

$= [e/x]B$

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

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

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

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

Weakest Preconditions for Loops

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

Technique: Strongest Postconditions

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

Technique: Strongest Postconditions

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

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

Technique: Strongest Postconditions

$\vdash \{ y > 100 \}$ `x := y` $\{x > 10\}$

$\vdash \{ y > 100 \}$ `x := y` $\{x > 20\}$

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

Technique: Strongest Postconditions

$\vdash \{\, y > 100 \,\}$ `x := y` $\{x > 10\}$

$\vdash \{\, y > 100 \,\}$ `x := y` $\{x > 20\}$

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

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

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

Technique: Strongest Postconditions

$\vdash \{ y > 100 \}$ `x := y` $\{x > 10\}$

$\vdash \{ y > 100 \}$ `x := y` $\{x > 20\}$

$\vdash \{ y > 100 \}$ `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} ?

Technique: Strongest Postconditions

$\vdash \{ y > 100 \}$ `x := y` $\{x > 10\}$

$\vdash \{ y > 100 \}$ `x := y` $\{x > 20\}$

$\vdash \{ y > 100 \}$ `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)

Technique: Strongest Postconditions

$\vdash$ { y > 100 } `x := y` {x > 10}

$\vdash$ { y > 100 } `x := y` {x > 20}

$\vdash$ { 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

Strongest Postconditions

Define sp(c, B) following Hoare rules

Strongest Postconditions

Define sp(c, B) following Hoare rules

Strongest Postconditions

Define sp(c, B) following Hoare rules

sp(`c`$_1$`;c`$_2$, A) =

Strongest Postconditions

Define sp(c, B) following Hoare rules

$sp(c_1; c_2, A) =$

$sp(c_2, sp(c_1, A))$

Strongest Postconditions

Define sp(c, B) following Hoare rules

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

$sp(\mathtt{c_2}, sp(\mathtt{c_1}, A))$

Strongest Postconditions

Define sp(c, B) following Hoare rules

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

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

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

Strongest Postconditions

Define sp(c, B) following Hoare rules

sp(c_1;c_2, A) =

sp(c_2, sp(c_1,A))

sp(x:=e , A) =

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

Strongest Postconditions

Define sp(c, B) following Hoare rules

sp(`c`$_1$`;c`$_2$, A) =

sp(`c`$_2$, sp(`c`$_1$,A))

sp(`x:=e` , A) =

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

Strongest Postconditions

Define sp(c, B) following Hoare rules

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

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

sp(**x:=e** , A) =

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

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

Strongest Postconditions

Define sp(c, B) following Hoare rules

sp(`c1;c2`, A) =

sp(`c2`, sp(`c1`,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. $[x_0/x]A$ && $x=[x_0/x]e$

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

sp(`if e then c1 else c2`, A) =

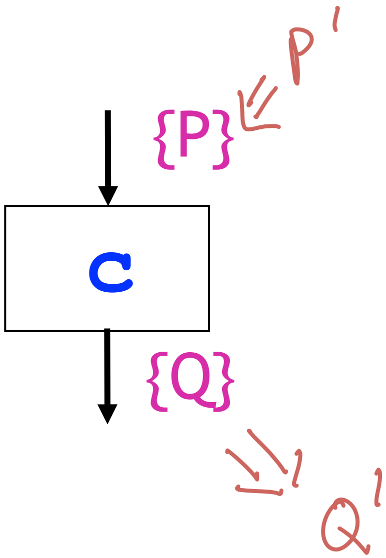
sp(`c1`, A & e) || sp(`c2`, A & !e)

$$\frac{\vdash \{A\&b\}\ c_1\ \{B\} \quad \vdash \{A\ \&\ !b\}\ c_2\ \{B\}}{\vdash \{A\}\ \texttt{if b then c}_1\ \texttt{else c}_2\ \{B\}}$$

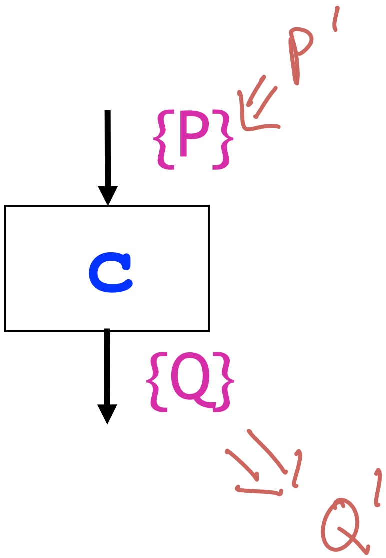
Axiomatic Semantics on Flow Graphs

Floyd’s Original Formulation

Axiomatic Semantics over Flow Graphs

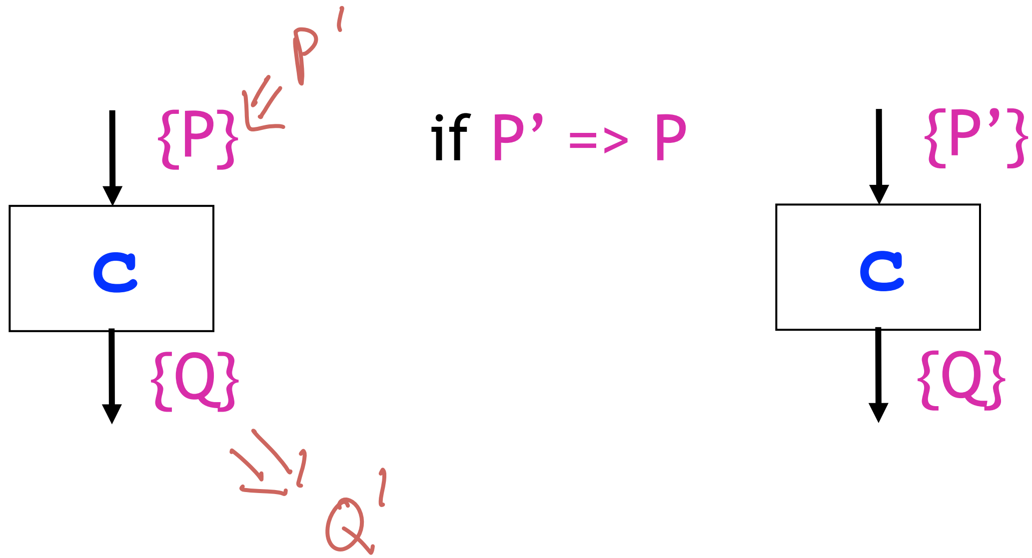


Axiomatic Semantics over Flow Graphs

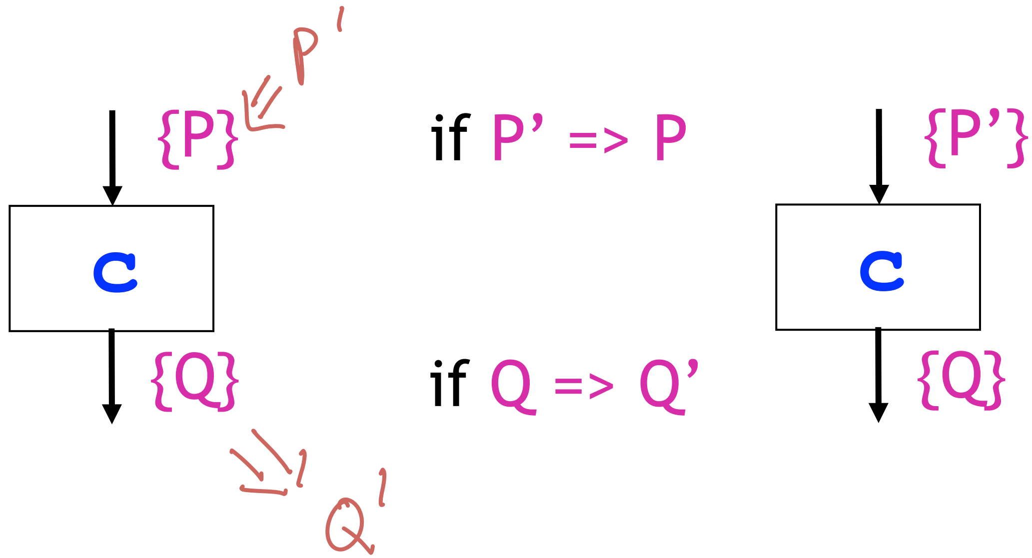


if P’ => P

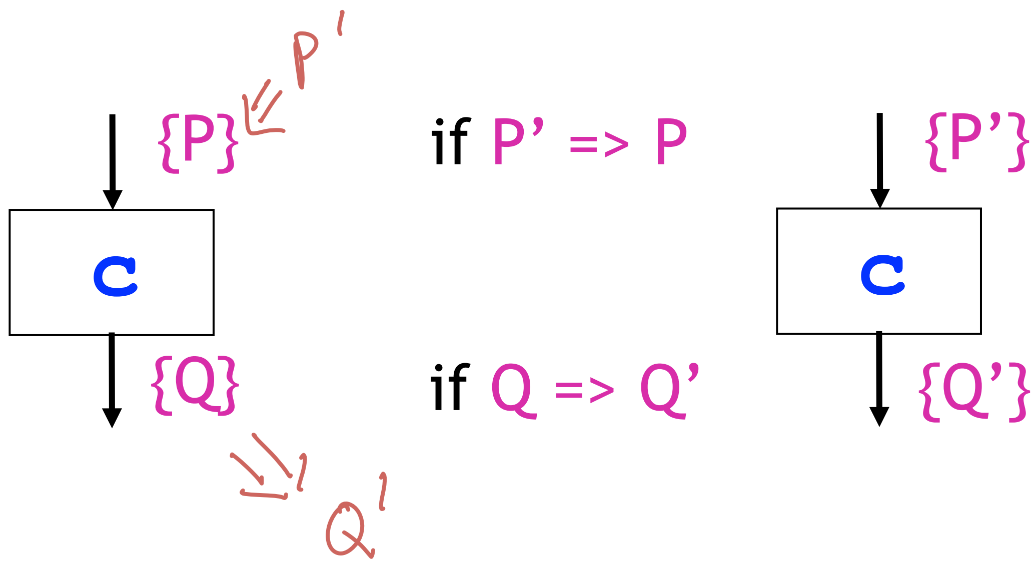
Axiomatic Semantics over Flow Graphs



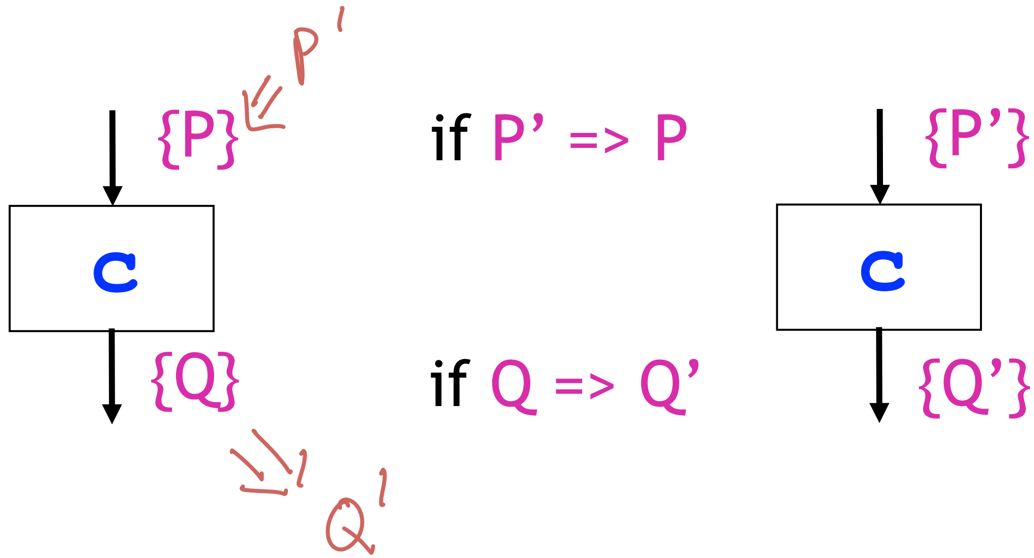
Axiomatic Semantics over Flow Graphs



Axiomatic Semantics over Flow Graphs



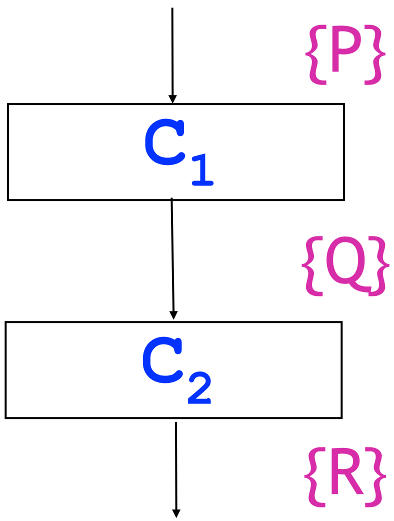
Axiomatic Semantics over Flow Graphs



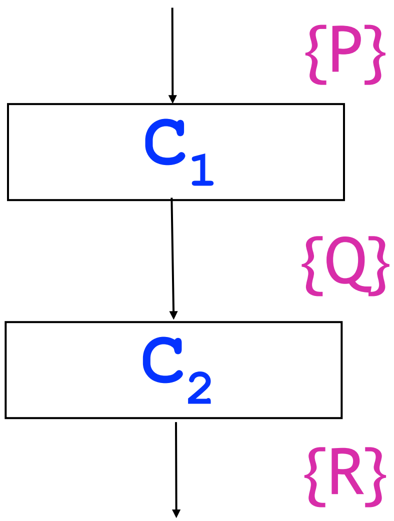
Relaxing Specifications via Consequence

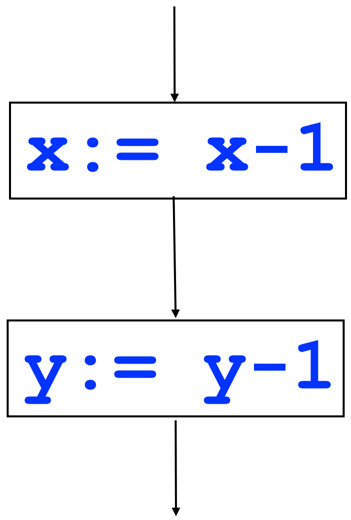
Will revisit later as **subtyping**

Sequential Composition

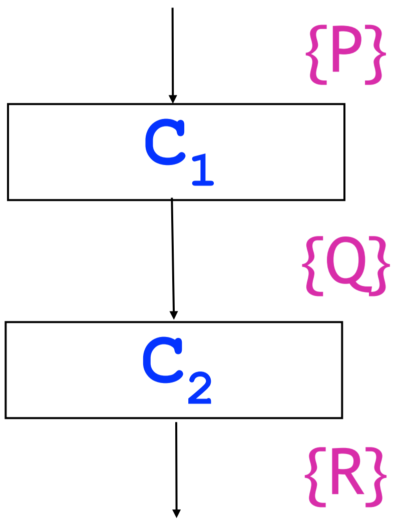


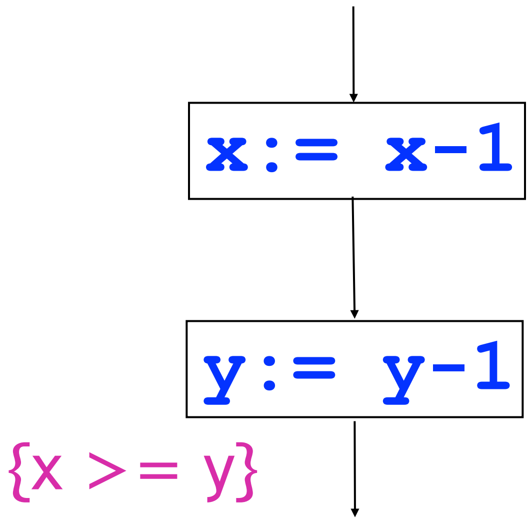
Sequential Composition



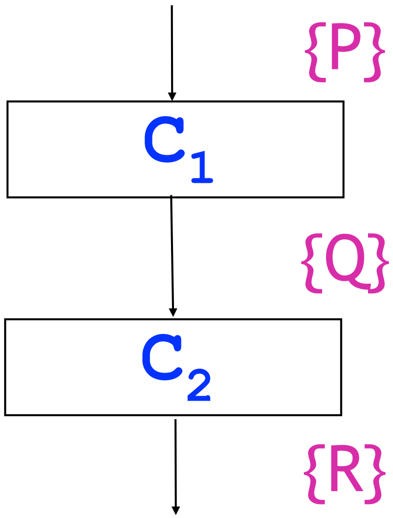


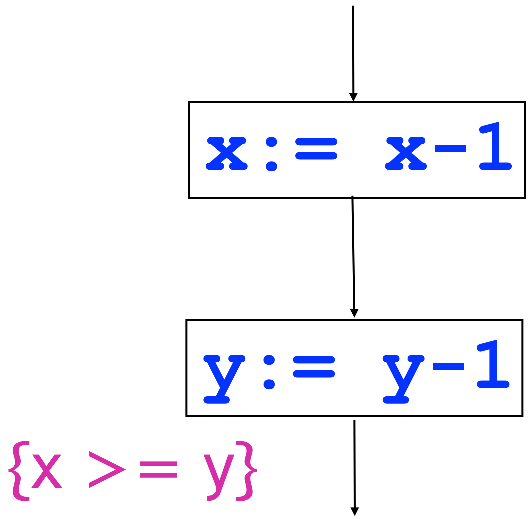
Sequential Composition





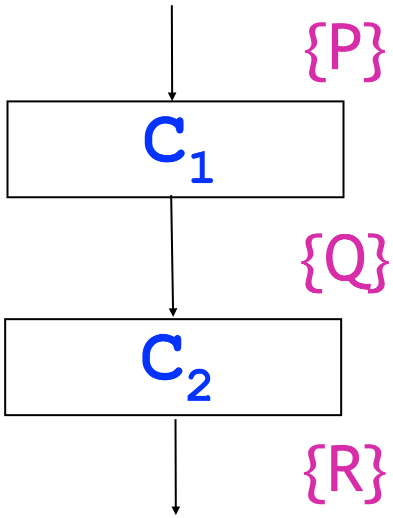
Sequential Composition

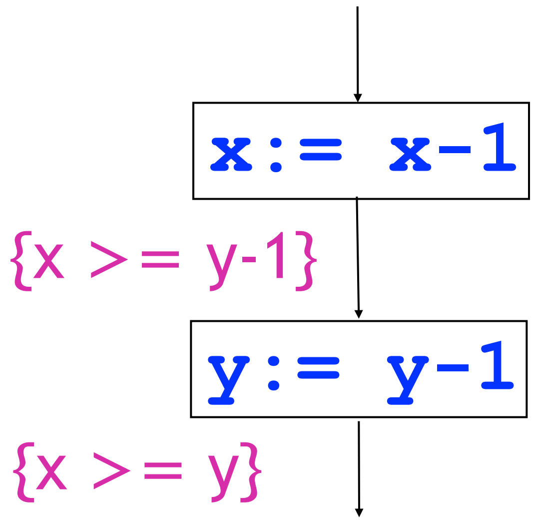




Backwards using weakest preconditions

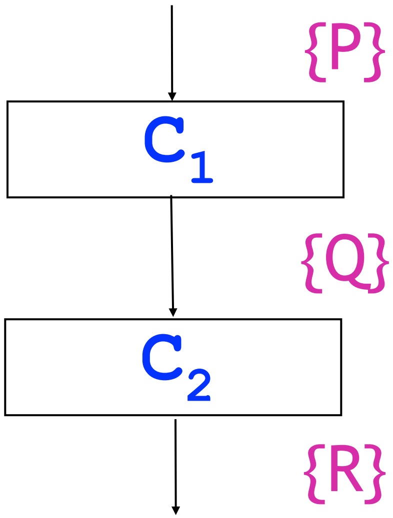
Sequential Composition

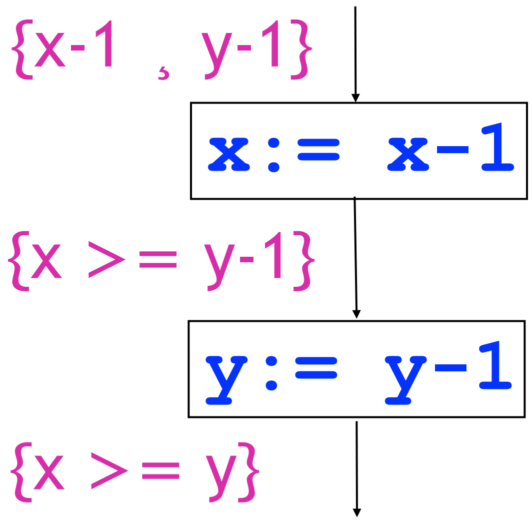




Backwards using weakest preconditions

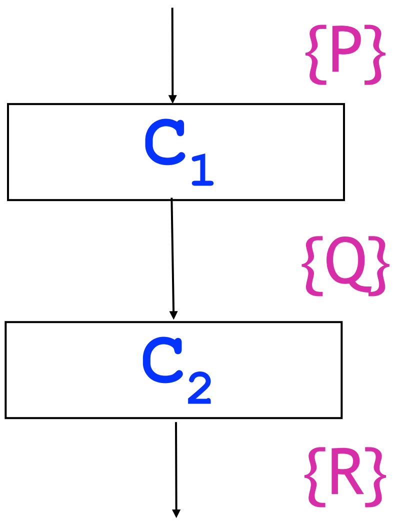
Sequential Composition

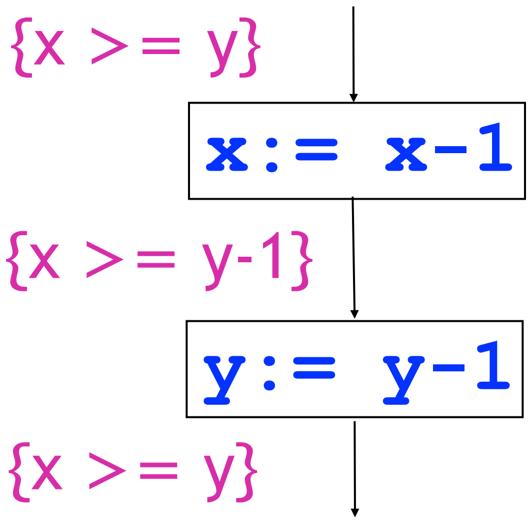




Backwards using weakest preconditions

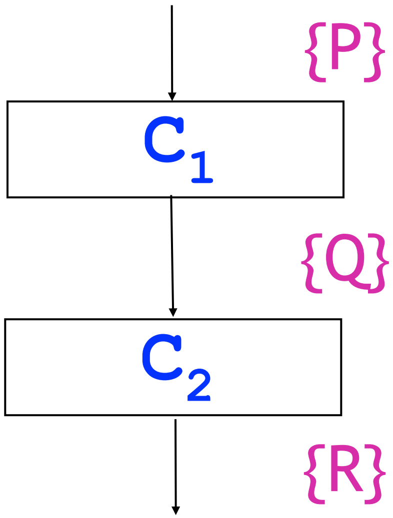
Sequential Composition

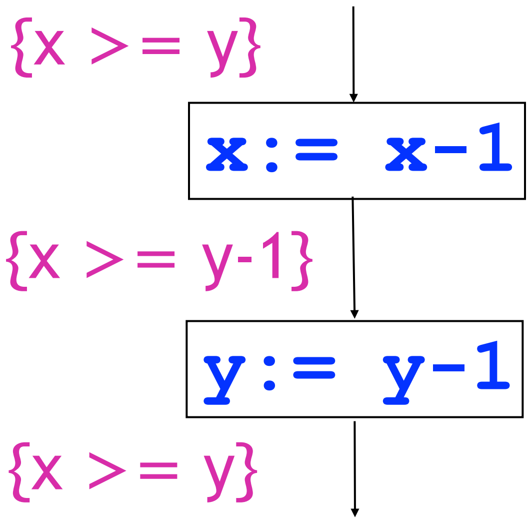




Backwards using weakest preconditions

Sequential Composition





Backwards using weakest preconditions

Forwards using strongest postconditions

Sequential Composition

{P}

C_1

{Q}

C_2

{R}

{x >= y} {x ¸ y}

```
x:= x-1
```

{x >= y-1}

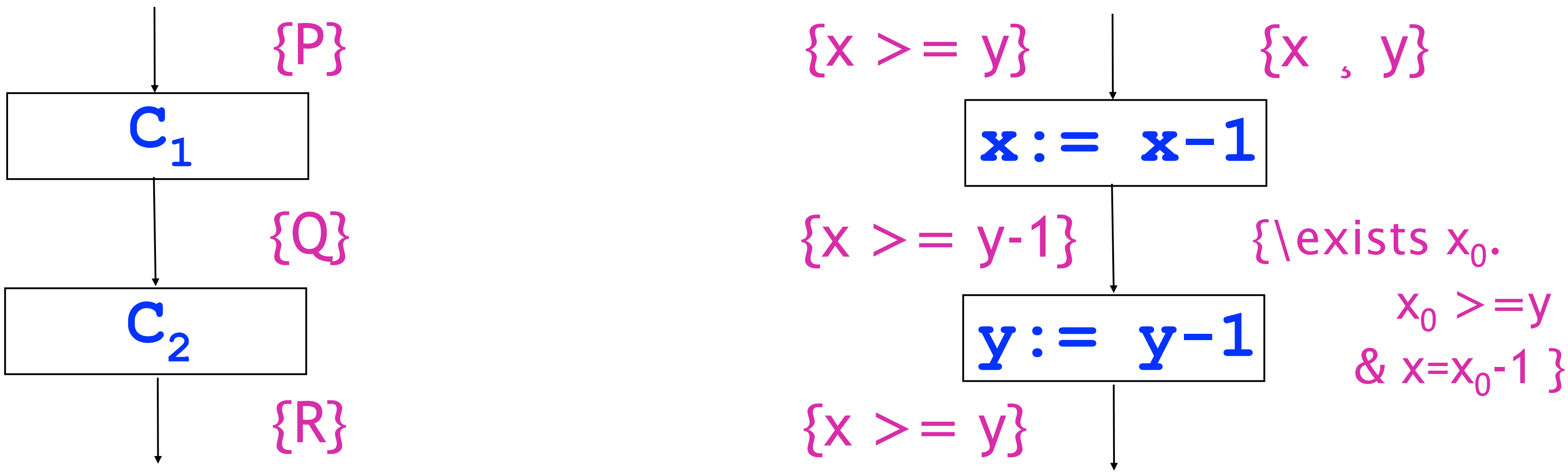
```
y:= y-1
```

{x >= y}

Backwards using weakest preconditions

Forwards using strongest postconditions

Sequential Composition



Backwards using weakest preconditions

Forwards using strongest postconditions

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}

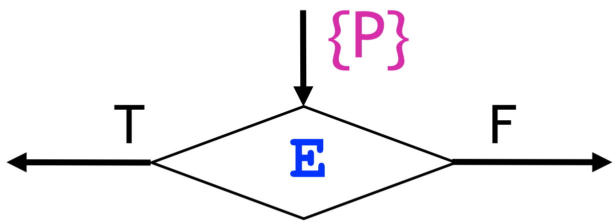
Backwards using weakest preconditions

Forwards using strongest postconditions

Conditionals

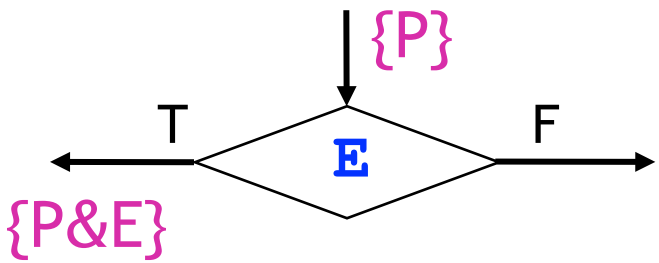
Forwards

Conditionals



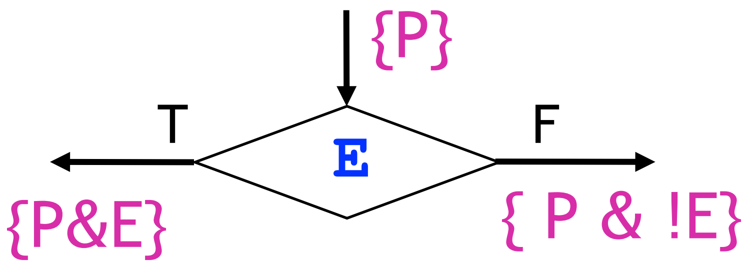
Forwards

Conditionals



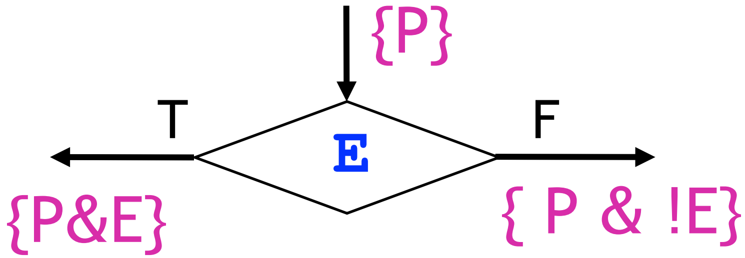
Forwards

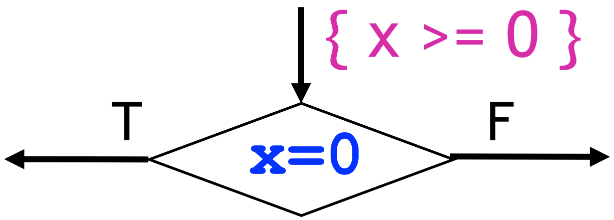
Conditionals



Forwards

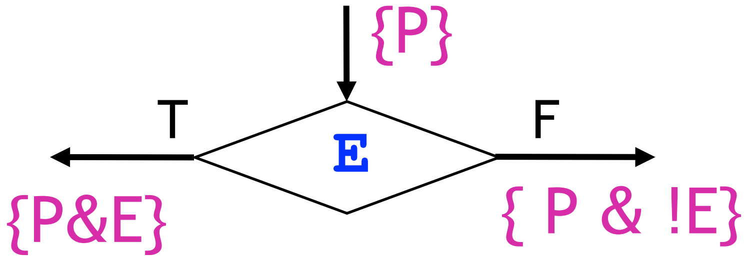
Conditionals

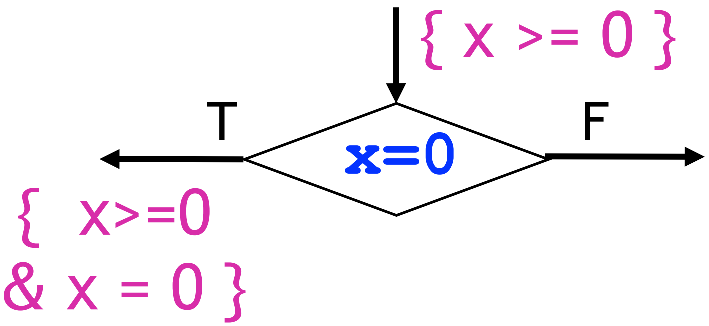




Forwards

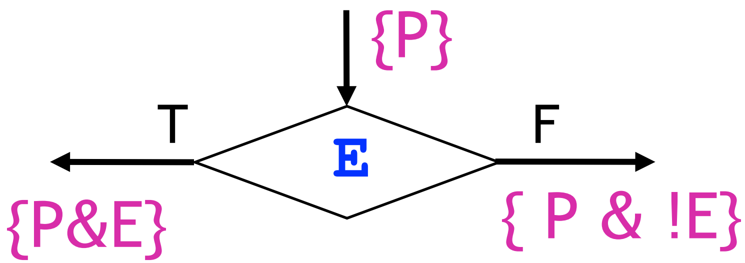
Conditionals

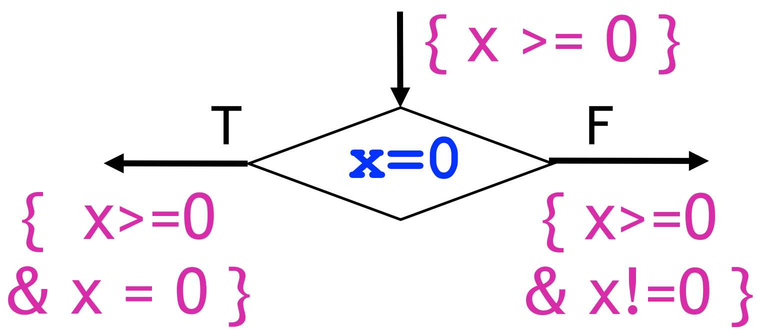




Forwards

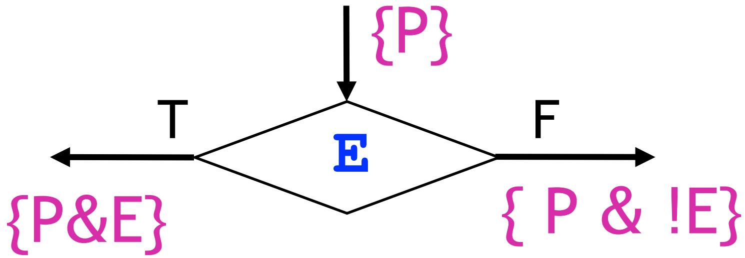
Conditionals

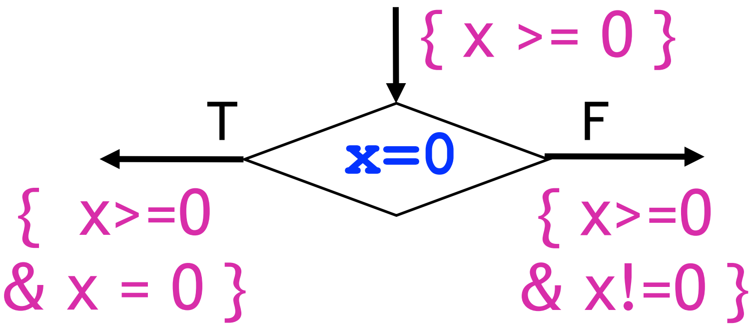




Forwards

Conditionals

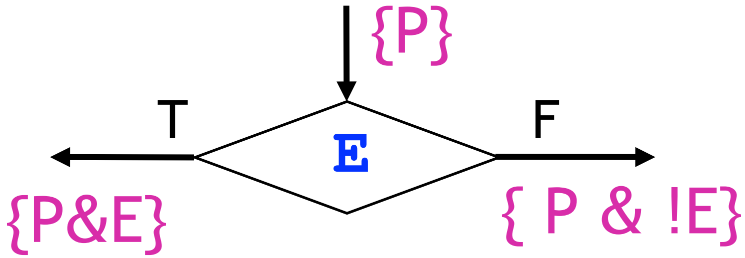


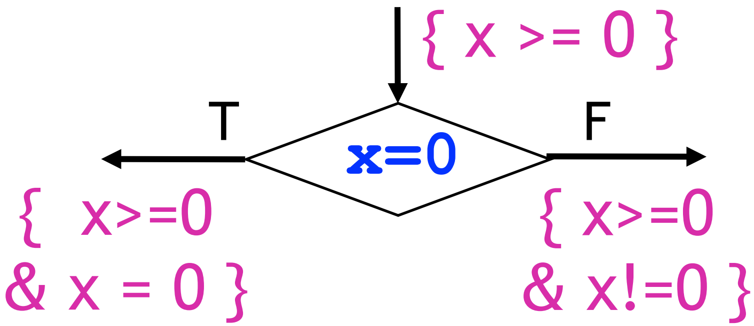


Forwards

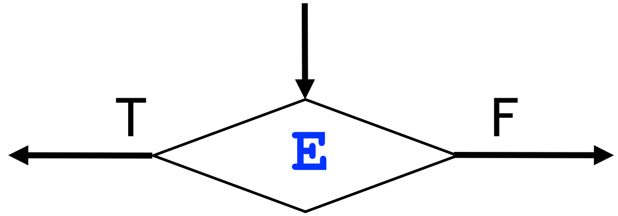
Backwards

Conditionals



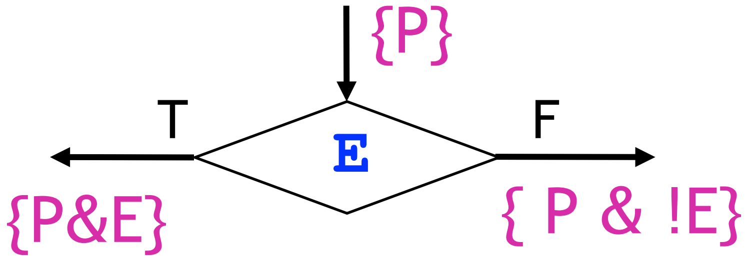


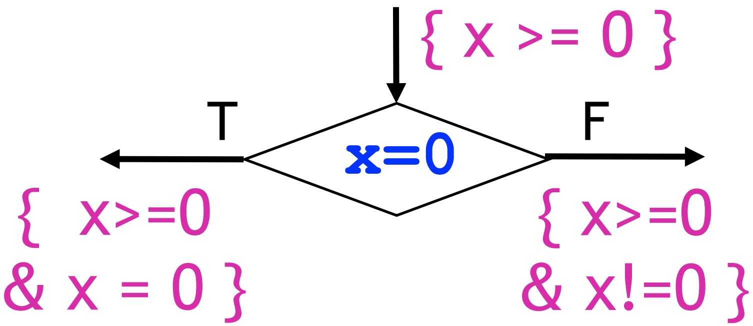
Forwards



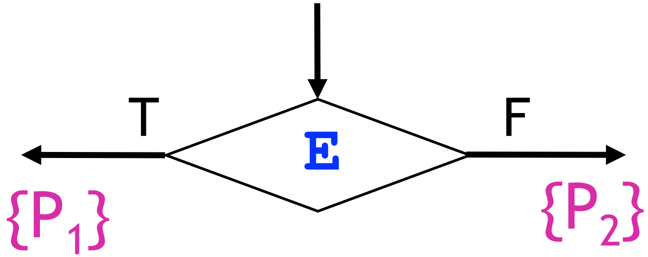
Backwards

Conditionals



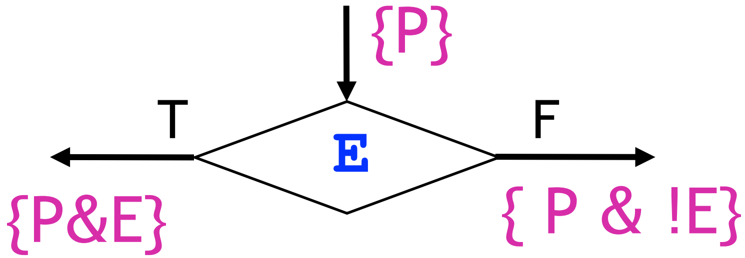


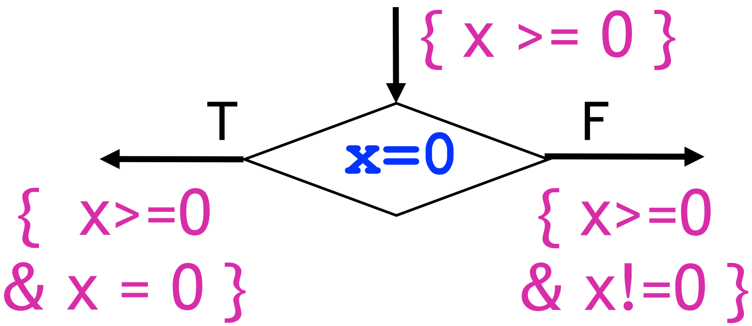
Forwards



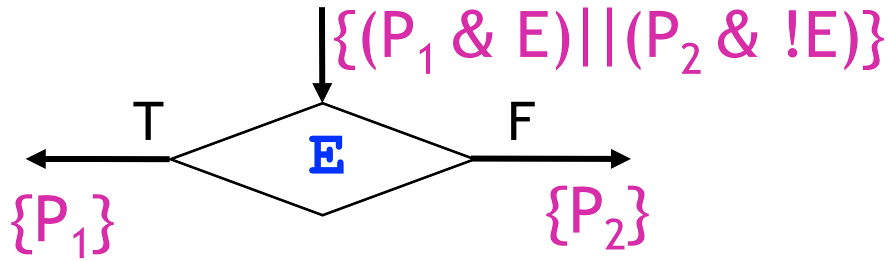
Backwards

Conditionals



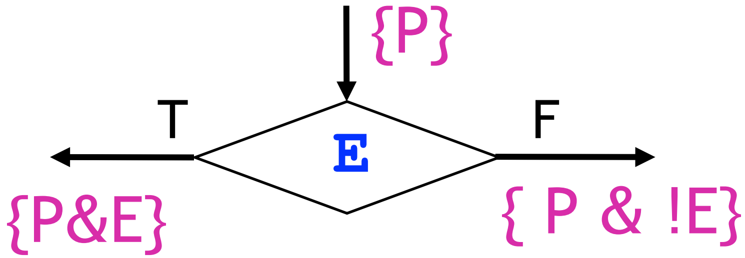


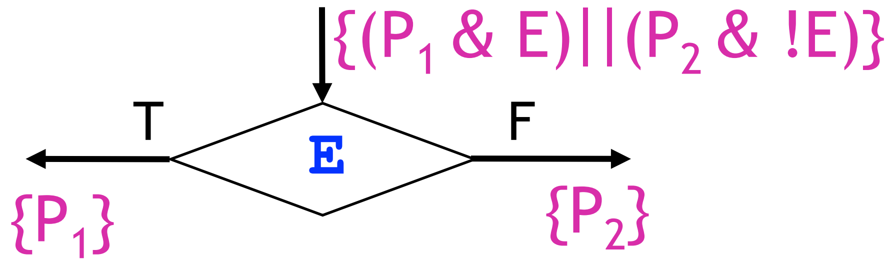
Forwards

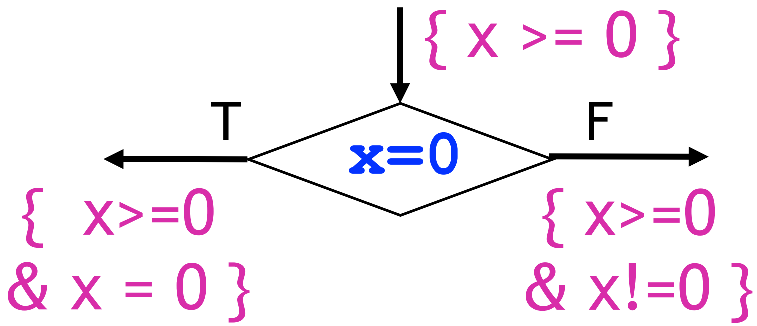


Backwards

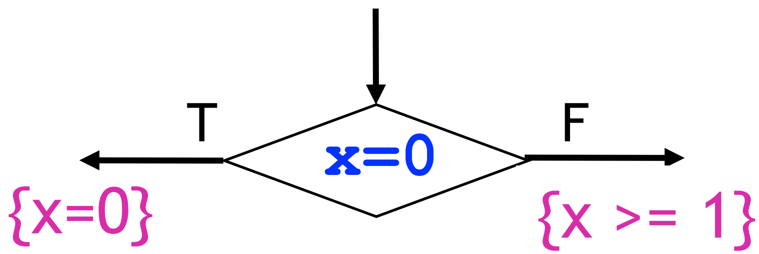
Conditionals





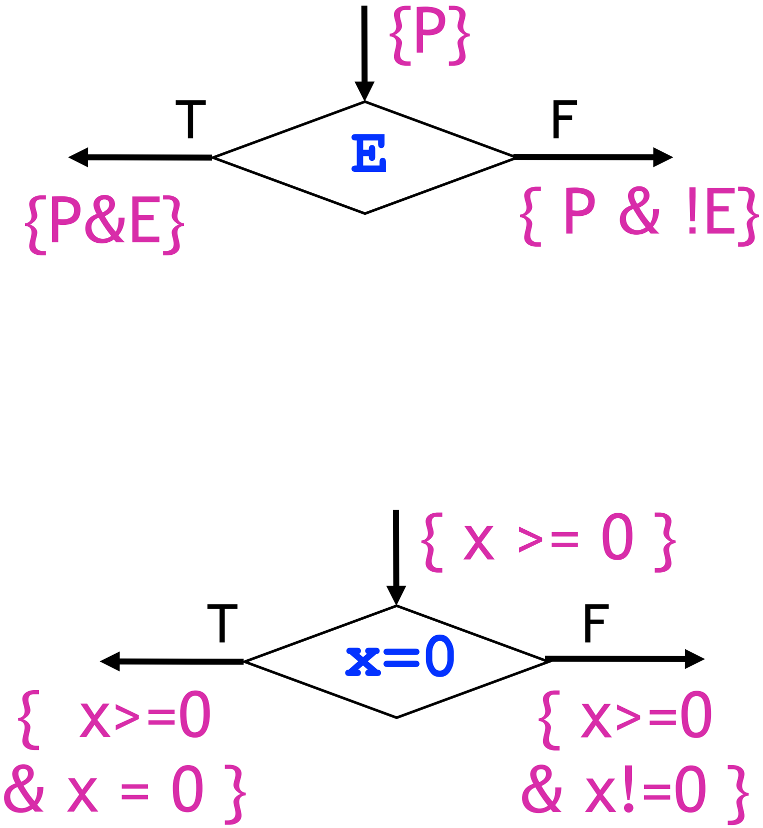


Forwards

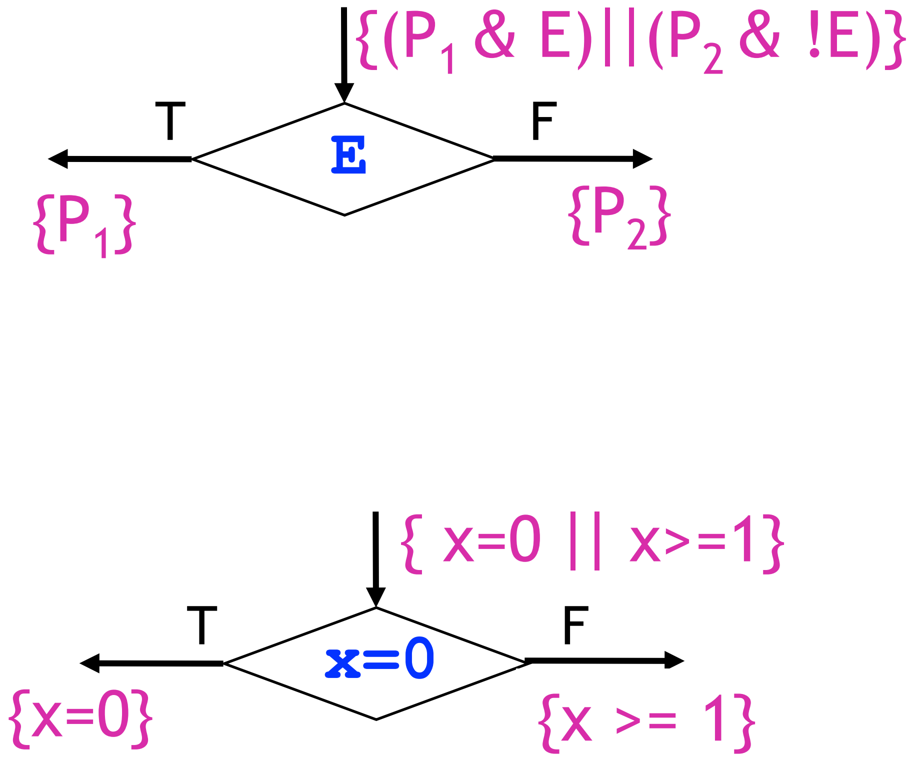


Backwards

Conditionals

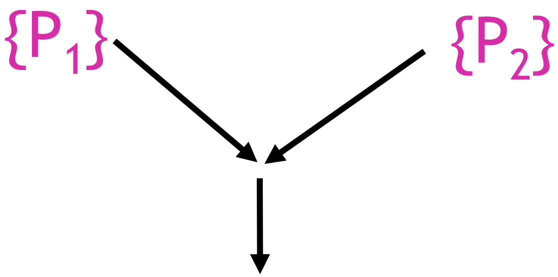


Forwards



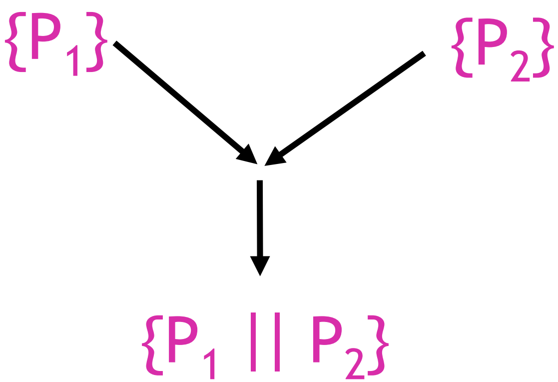
Backwards

Joins



Forwards

Joins



Forwards

Joins

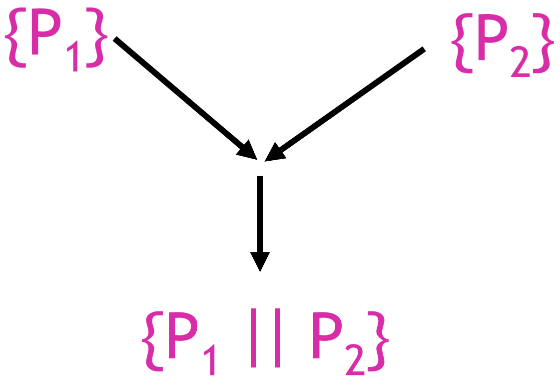
$\{P_1\}$ $\{P_2\}$

$\{P_1 \mid\mid P_2\}$

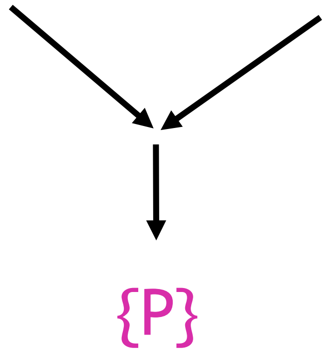
Forwards

Backwards

Joins

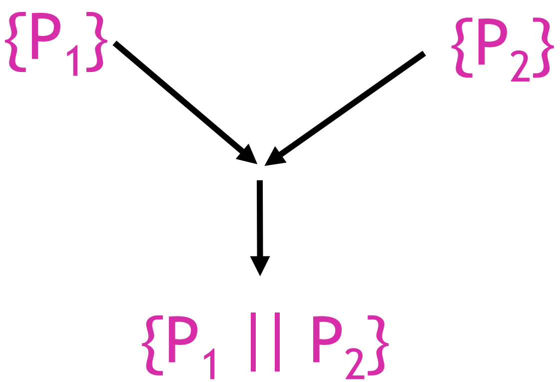


Forwards

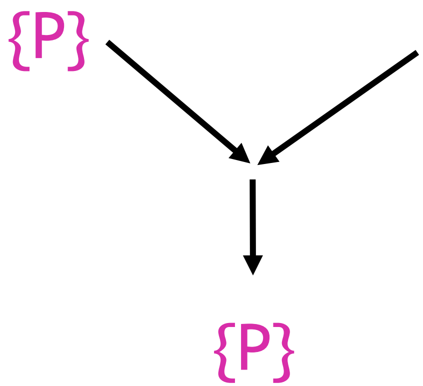


Backwards

Joins

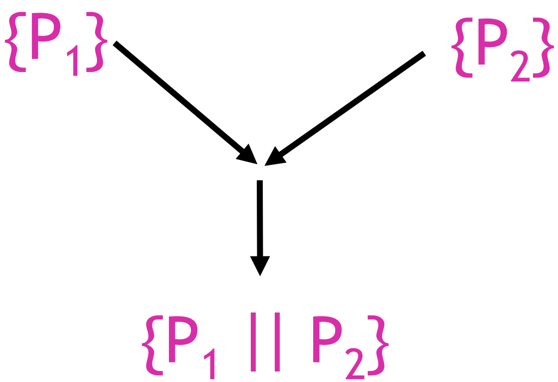


Forwards

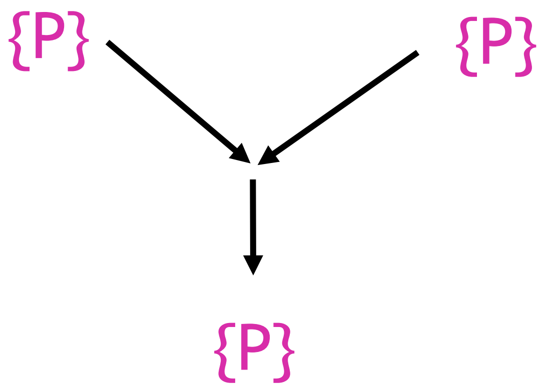


Backwards

Joins

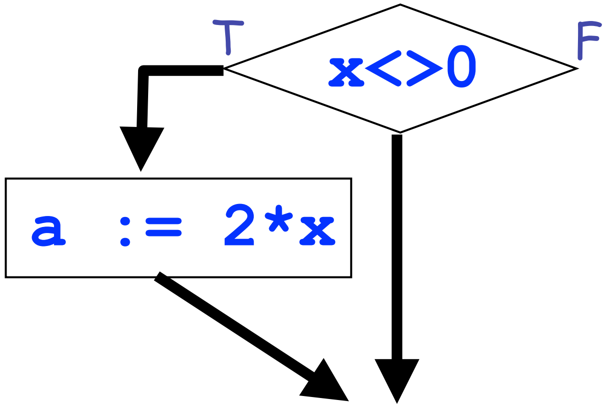


Forwards

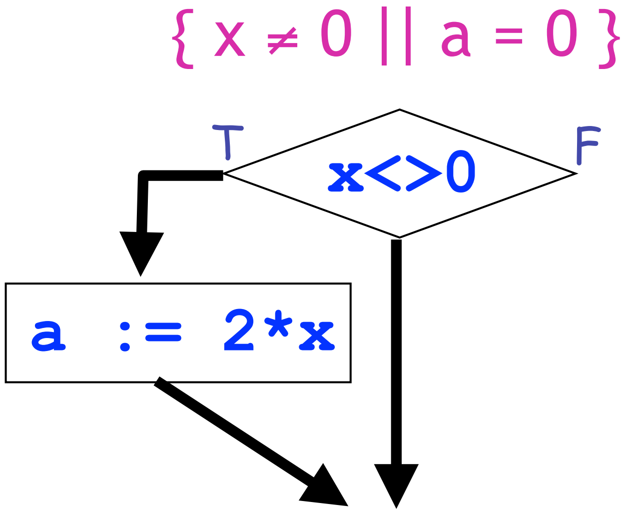


Backwards

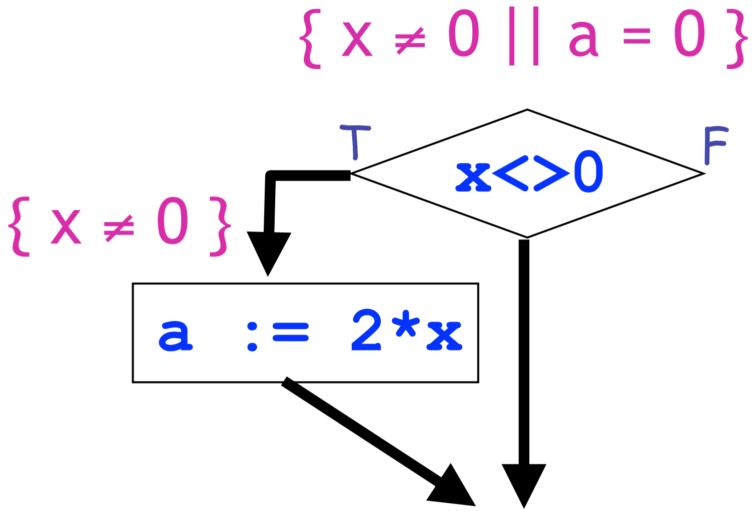
Conditional+Join: Forward



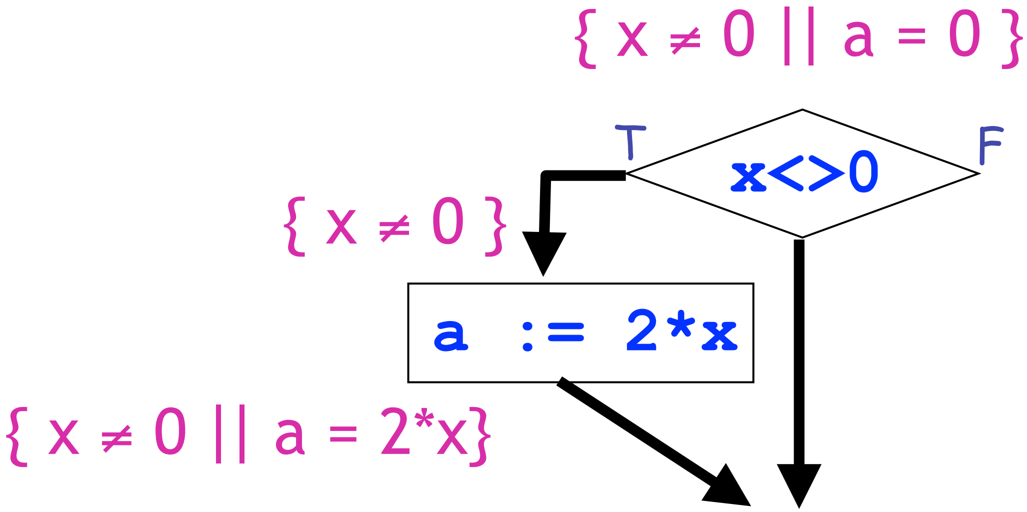
Conditional+Join: Forward



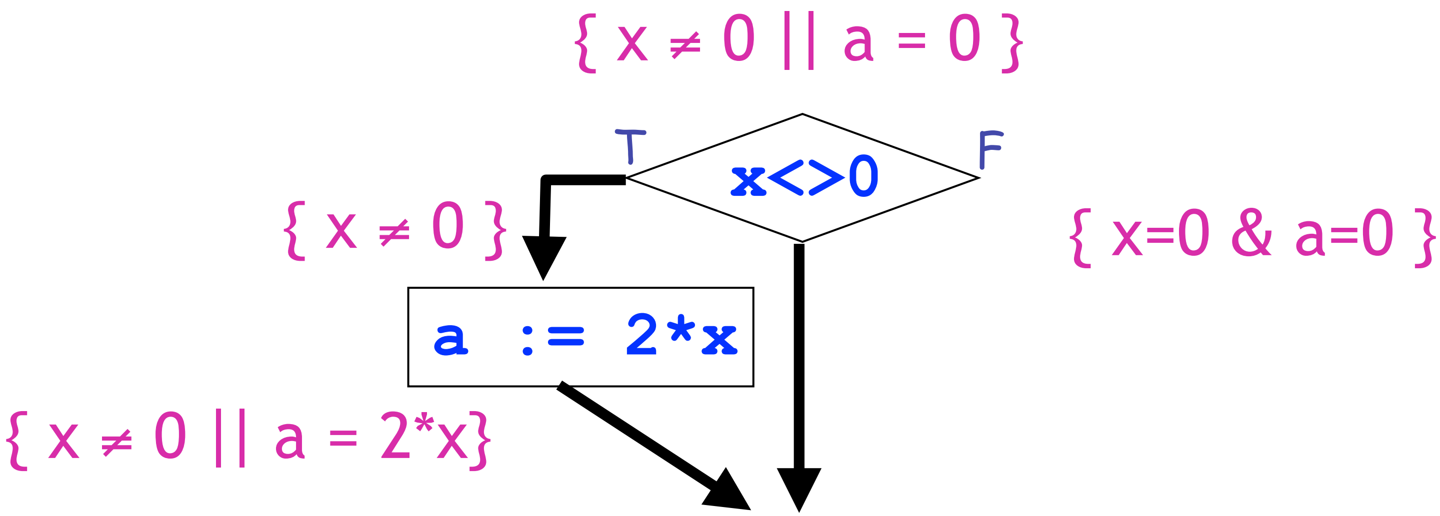
Conditional+Join: Forward



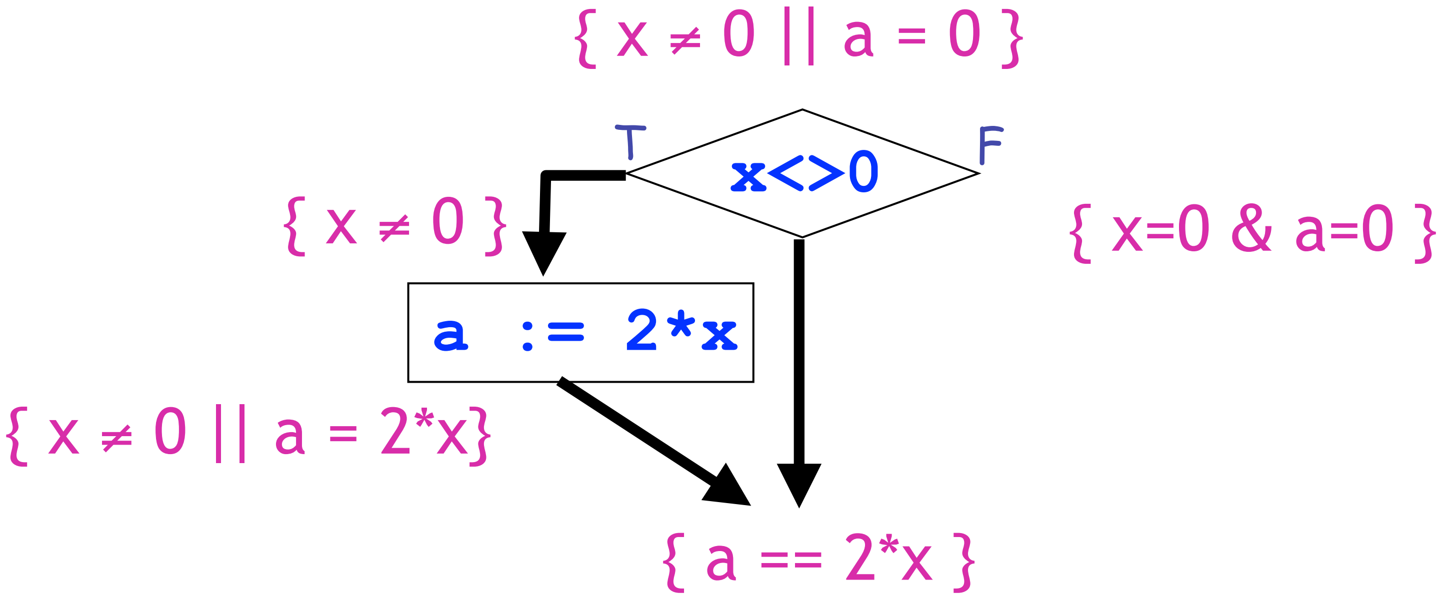
Conditional+Join: Forward



Conditional+Join: Forward

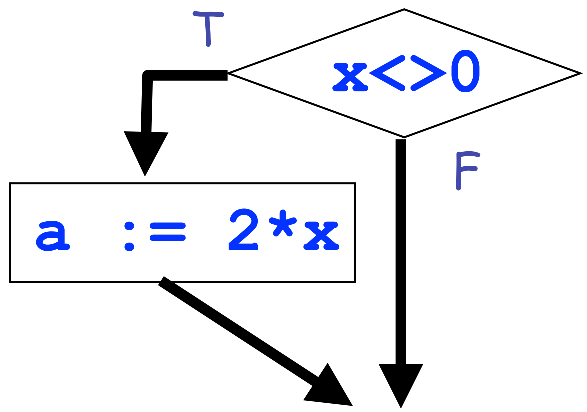


Conditional+Join: Forward

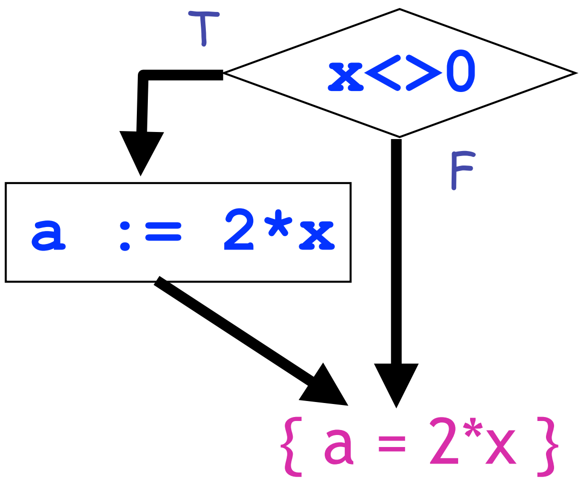


- Check the implications (simplifications)

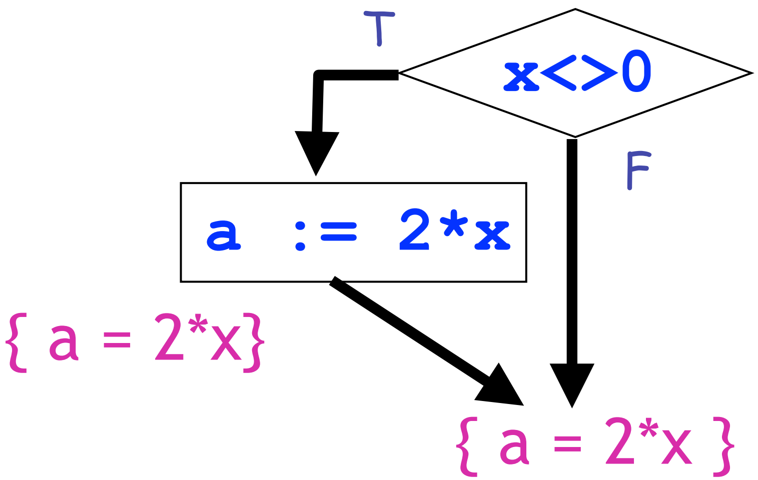
Conditionals+Joins: Backward



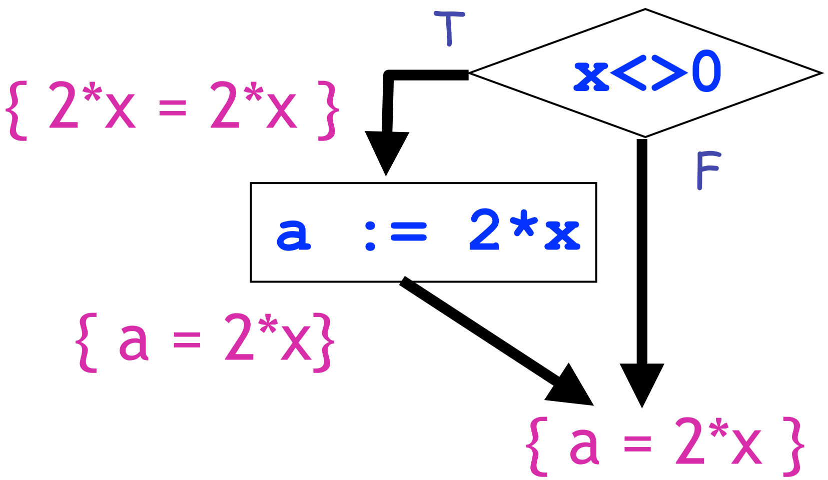
Conditionals+Joins: Backward



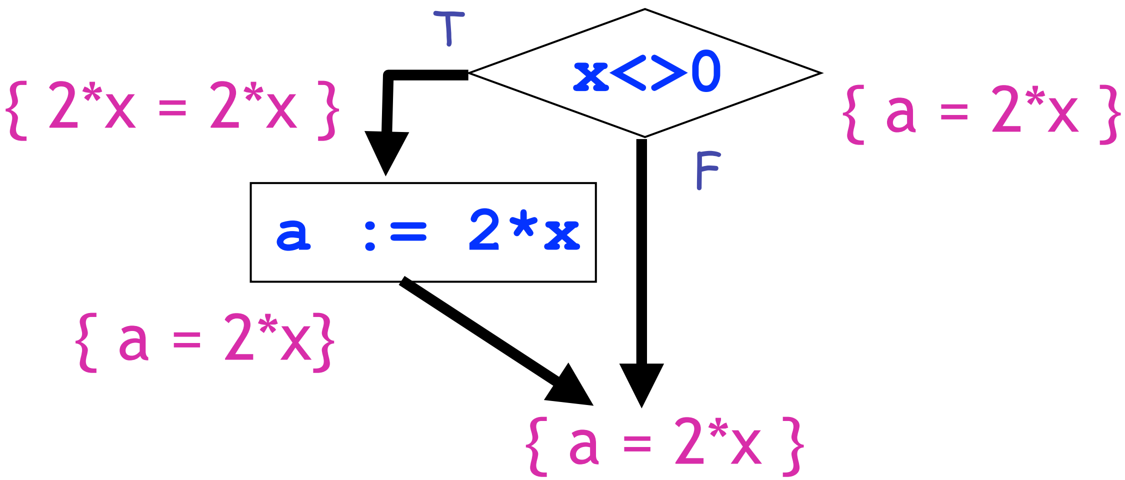
Conditionals+Joins: Backward



Conditionals+Joins: Backward

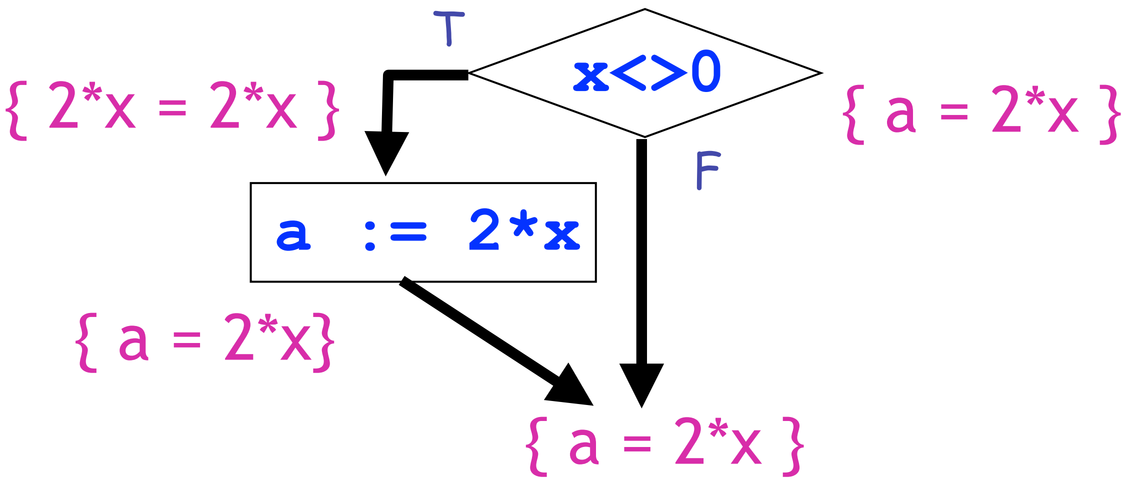


Conditionals+Joins: Backward



Conditionals+Joins: Backward

$\{ (x \neq 0 \;\&\; \text{true}) \;||\; (x = 0 \;\&\; a = 2{*}x) \}$

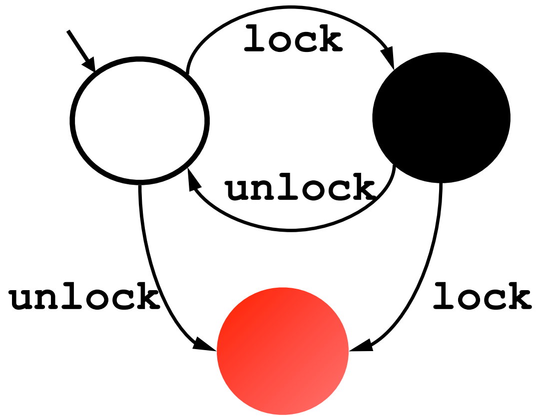


Forward or Backward ?

- 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

Locking Rules

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

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`

Locking Rules

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

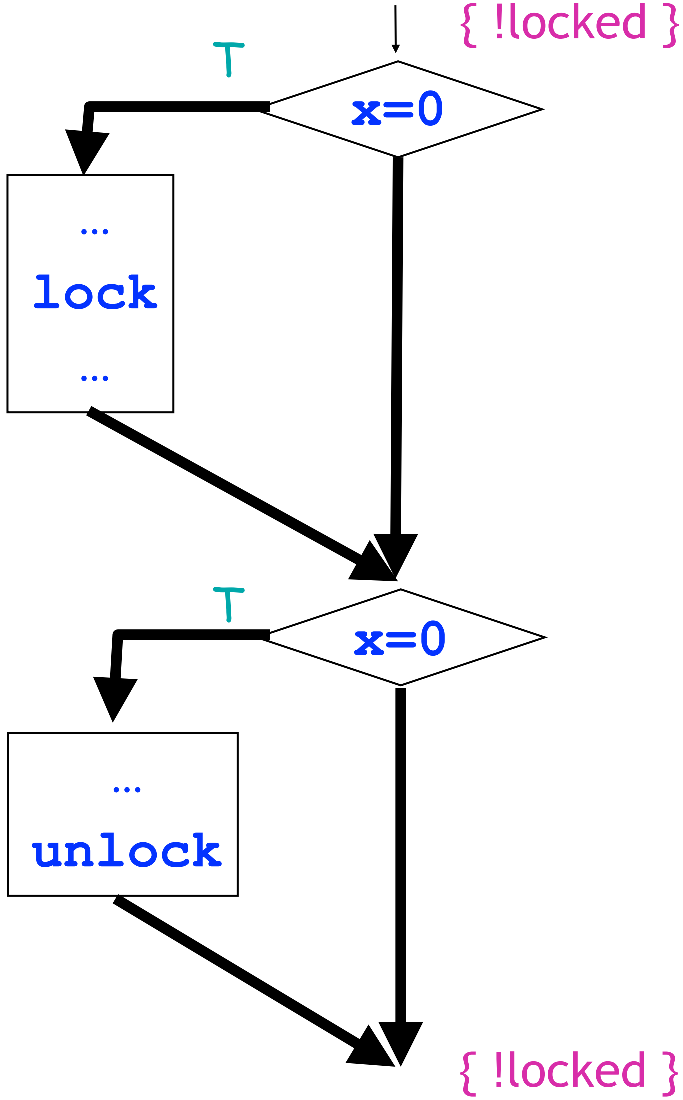
- {!locked & P[true/locked] } `lock` { P }

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

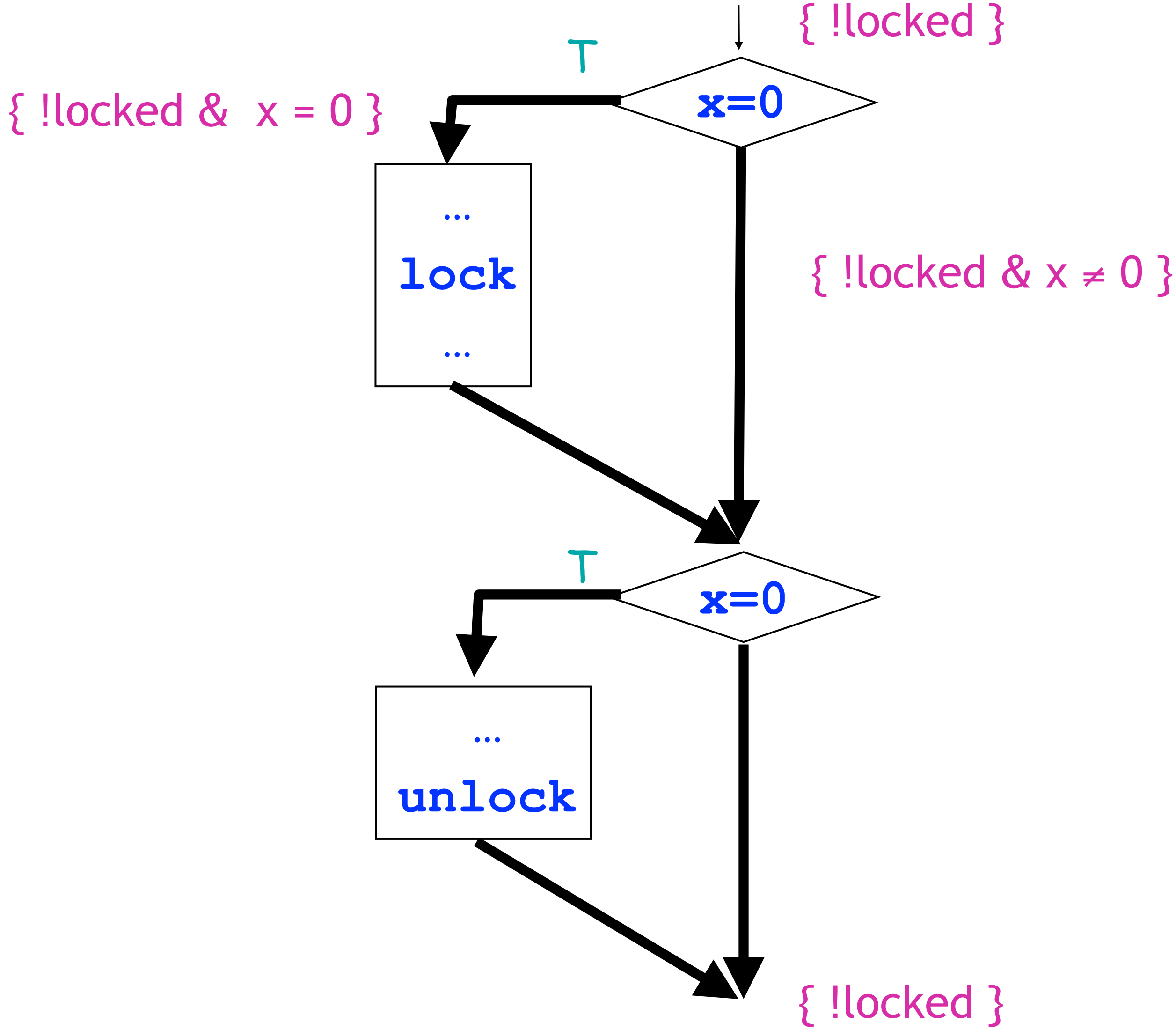
- { locked & P[false/locked] } `unlock` { P }

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

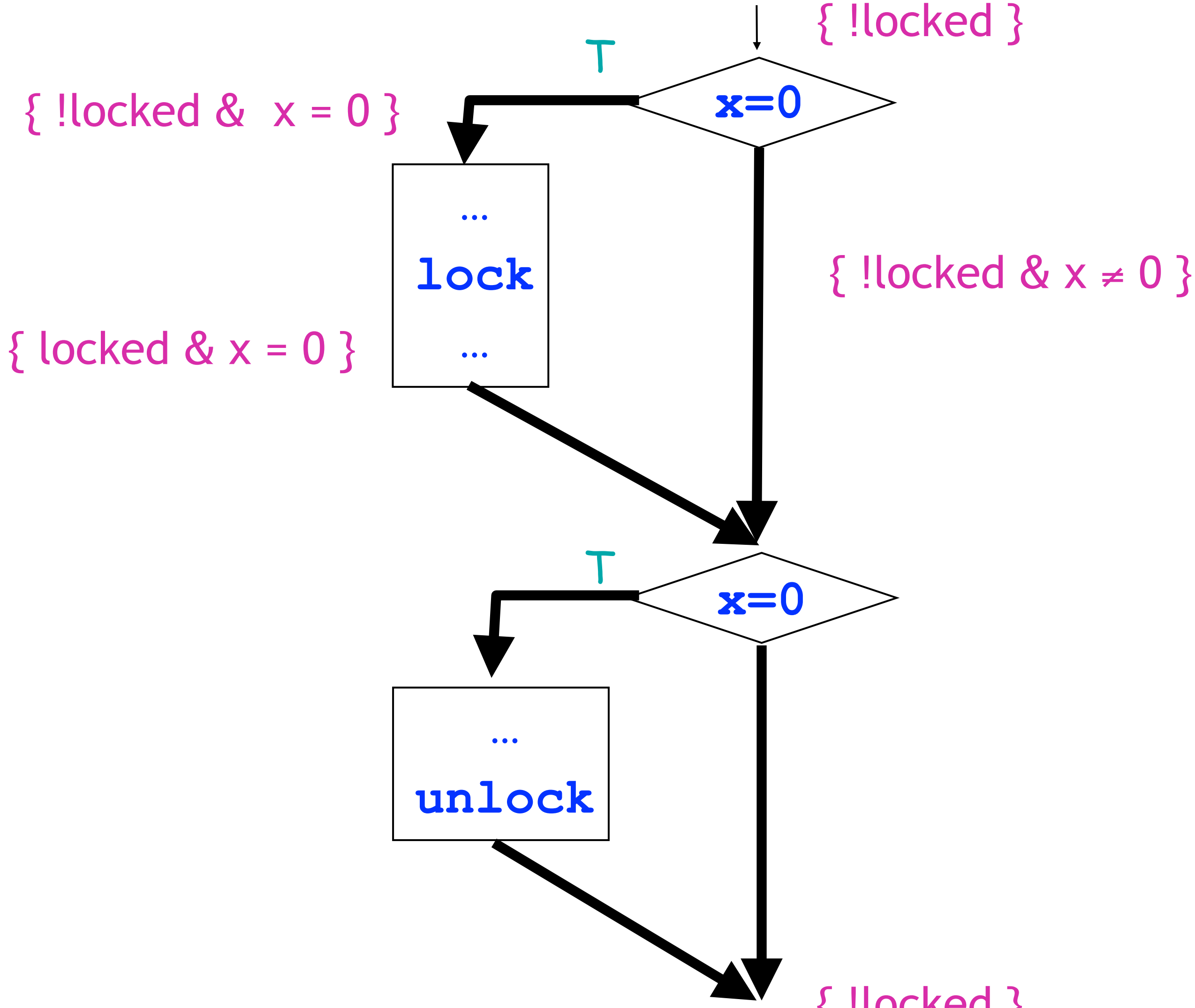
Locking Example



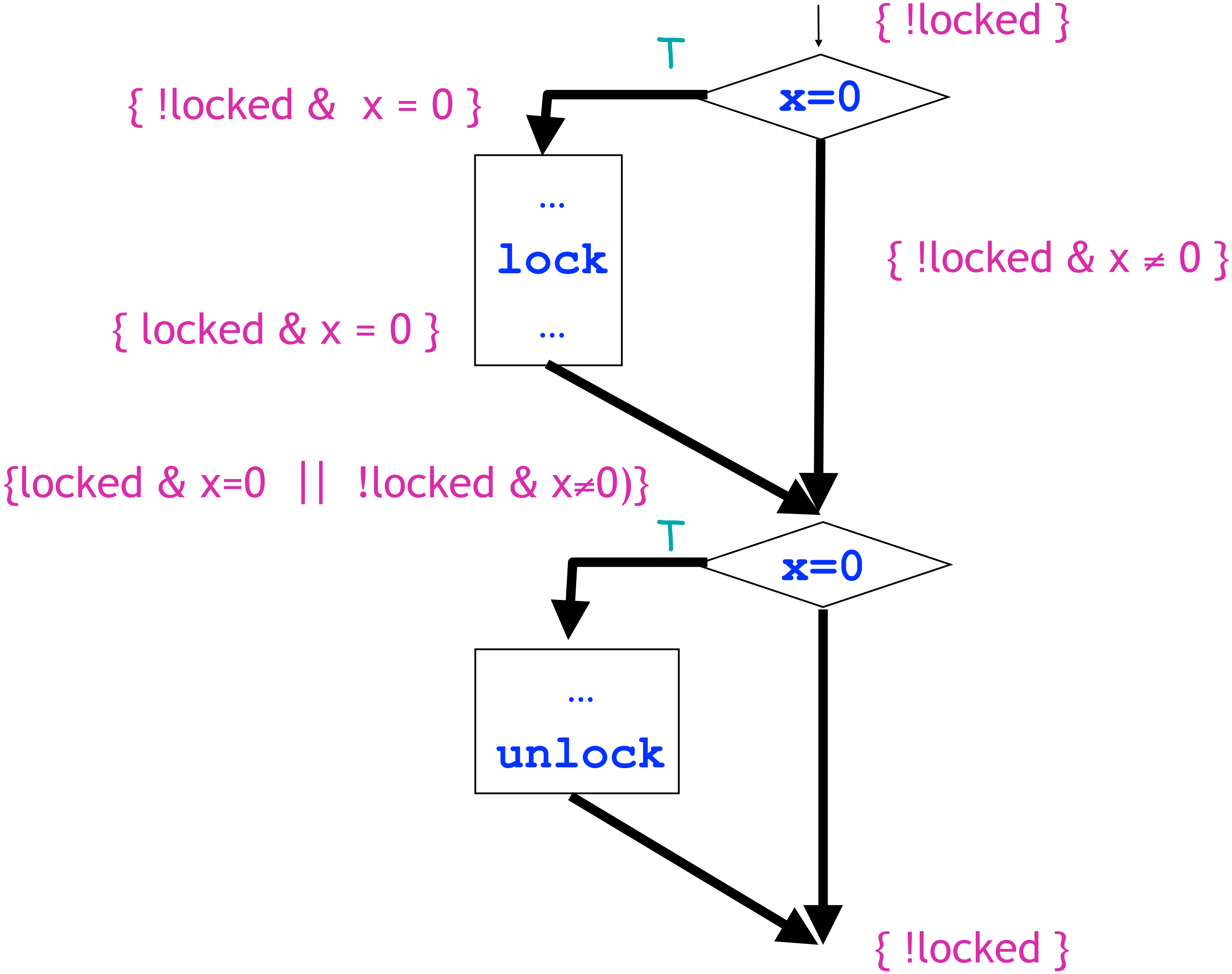
Locking Example



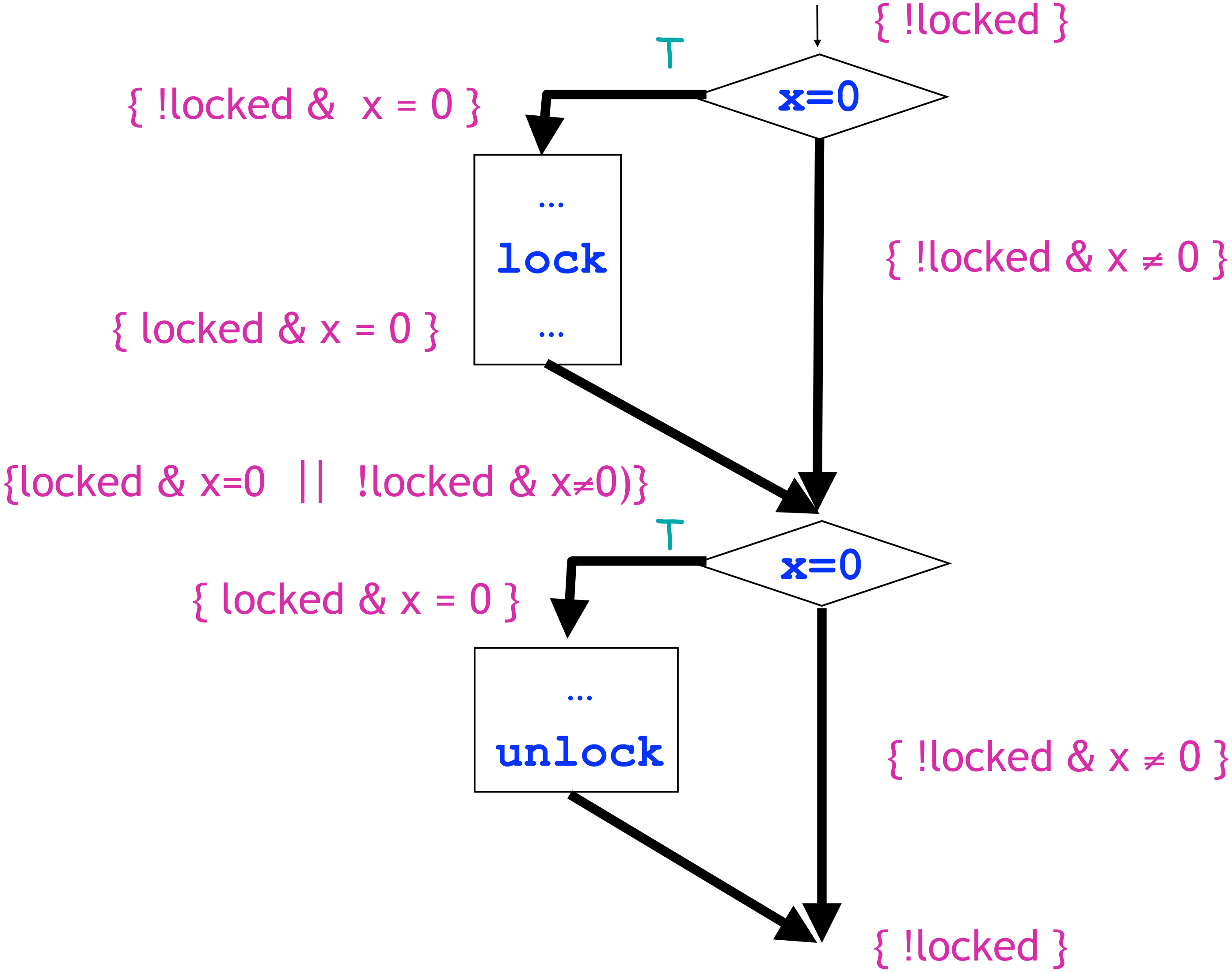
Locking Example



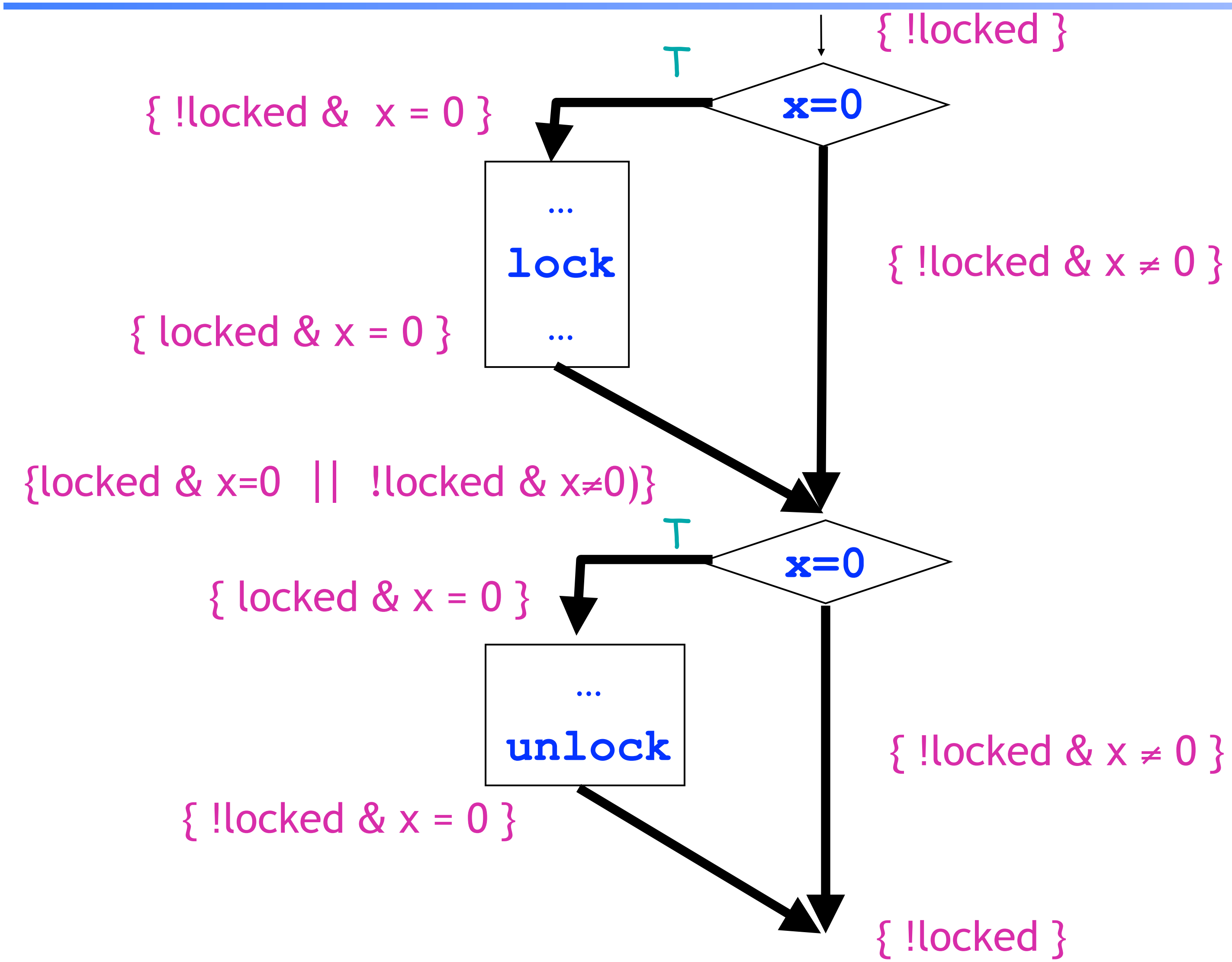
Locking Example



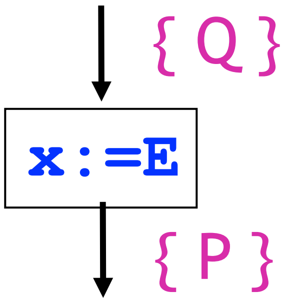
Locking Example



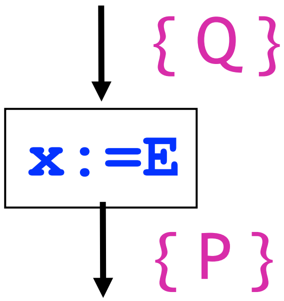
Locking Example



Review

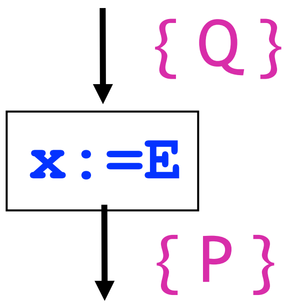


Review

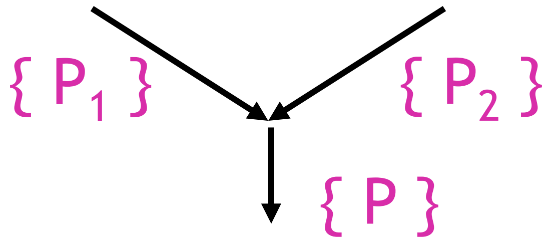


if Q => P[E\x] }

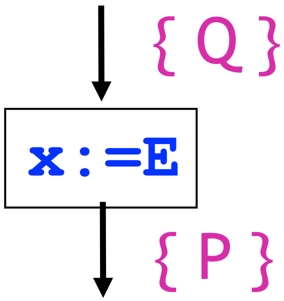
Review



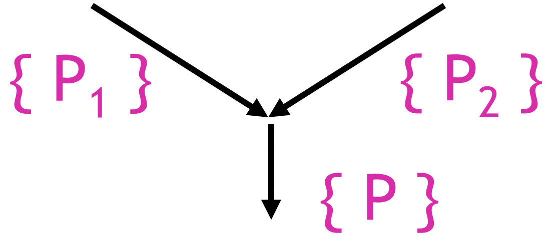
if Q => P[E\x] }



Review

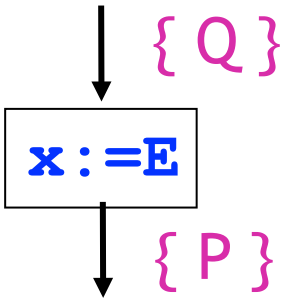


if $Q \Rightarrow P[E\backslash x]$ }

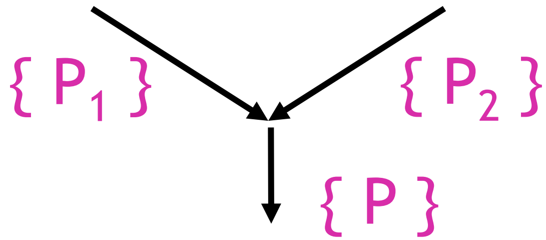


if $P_1 \Rightarrow P$ and $P_2 \Rightarrow P$

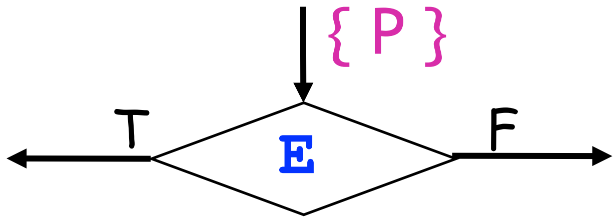
Review



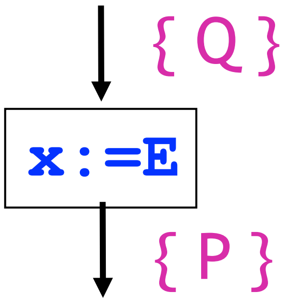
if $Q \Rightarrow P[E\backslash x]$ }



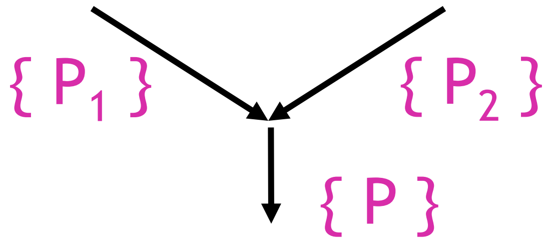
if $P_1 \Rightarrow P$ and $P_2 \Rightarrow P$



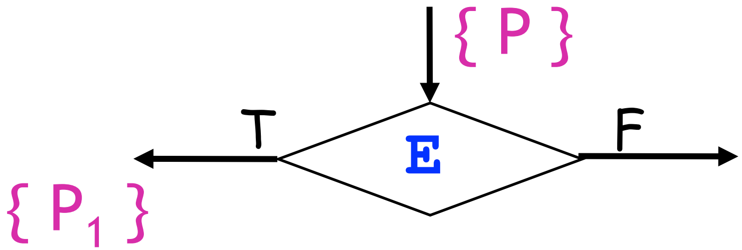
Review



if Q => P[E\x] }

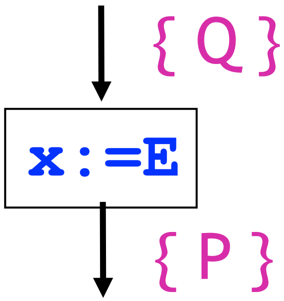


if P_1 => P and P_2 => P

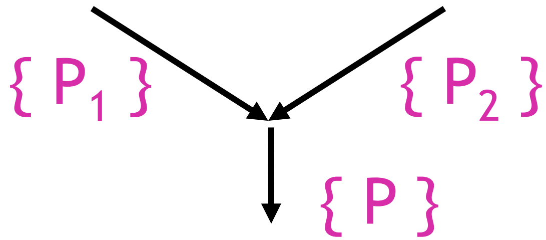


if P&E => P_1

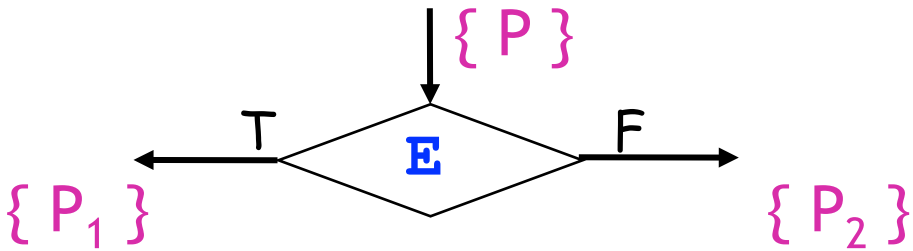
Review



if $Q => P[E\backslash x]$ }



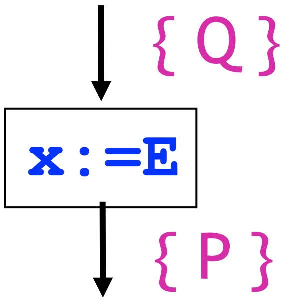
if $P_1 => P$ and $P_2 => P$



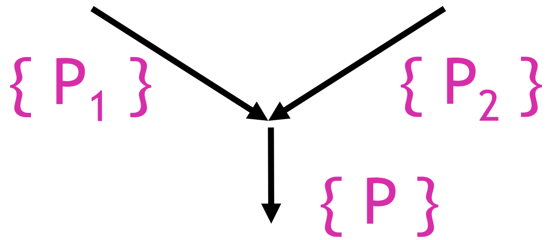
if $P\&E => P_1$

if $P\& !E => P_2$

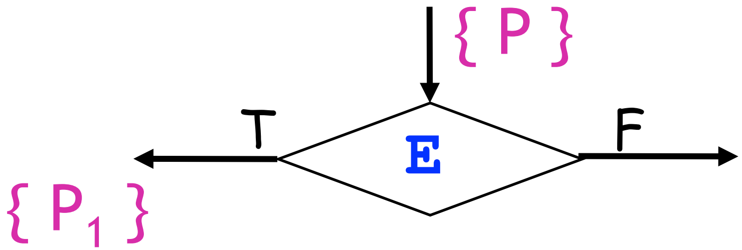
Review



if $Q => P[E\backslash x]$ }



if $P_1 => P$ and $P_2 => P$



if $P\&E => P_1$

if $P\& !E => P_2$

Implication is always in the direction of the control flow

What about real languages ?

- Loops
- Function calls
- Pointers

Reasoning about loops: Rules

Rewrite A with I : Loop Invariant

Rule of Consequence

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

Rule of Consequence

Reasoning about loops: Rules

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

Rewrite A with I : Loop Invariant

$$\frac{\vdash \{I \;\&\; b\}\ \texttt{c}\ \{I\}}{\vdash \{I\}\ \texttt{while b do c}\ \{I \;\&\; !b\}}$$

Rule of Consequence

Reasoning about loops: Rules

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

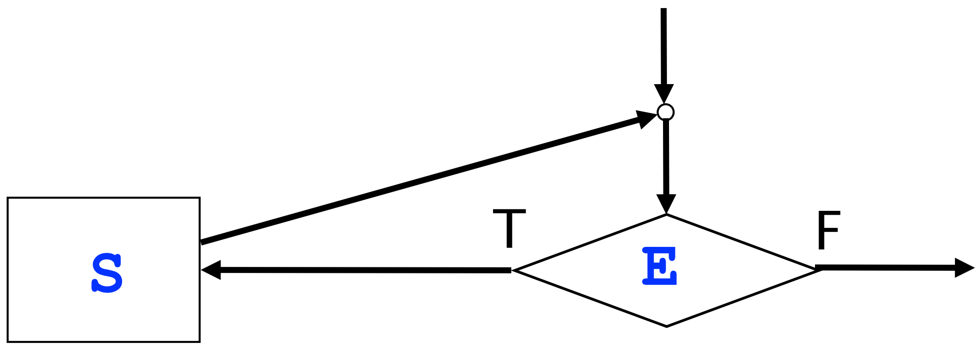
Rewrite A with I : Loop Invariant

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

Rule of Consequence

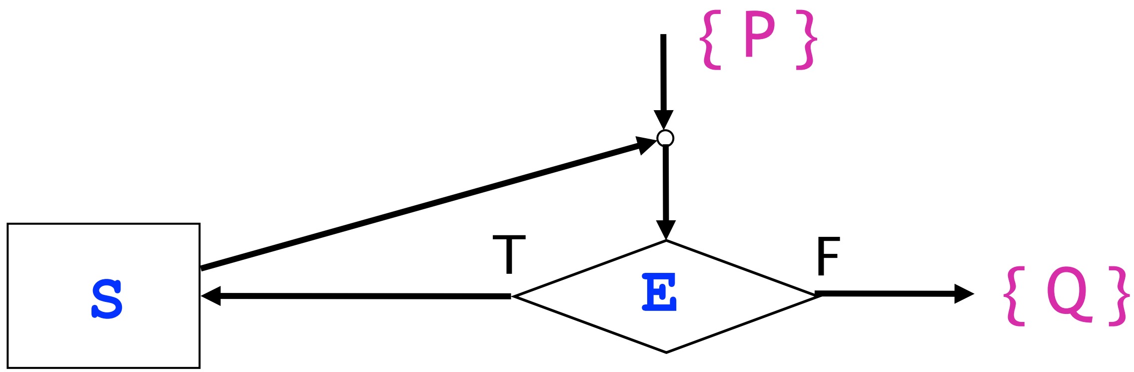
Reasoning about loops: Flow Graphs

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



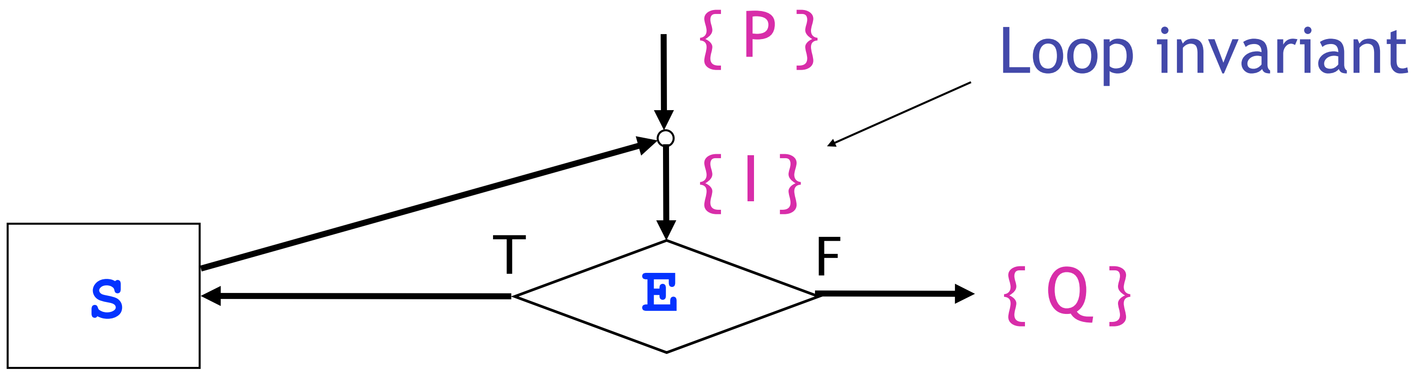
Reasoning about loops: Flow Graphs

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



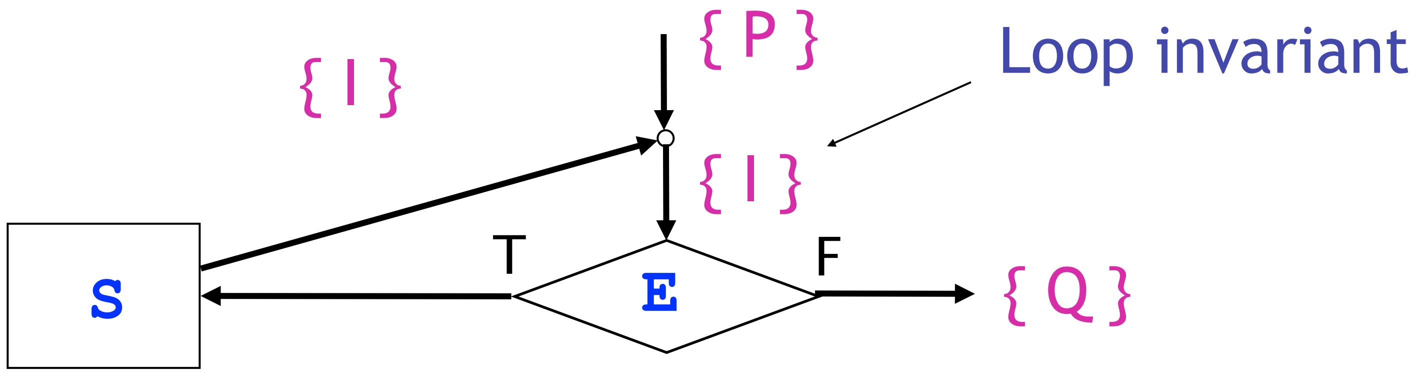
Reasoning about loops: Flow Graphs

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



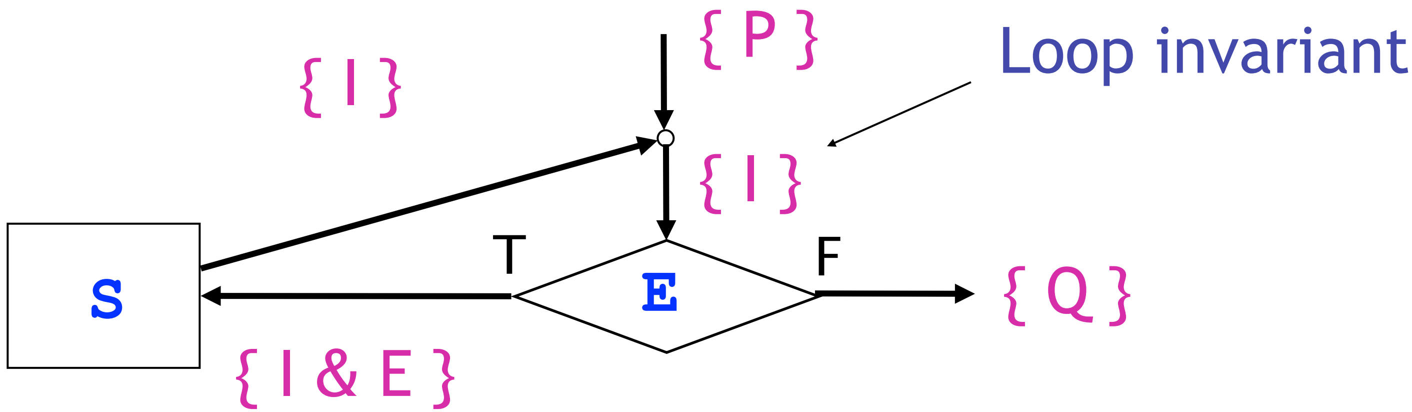
Reasoning about loops: Flow Graphs

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



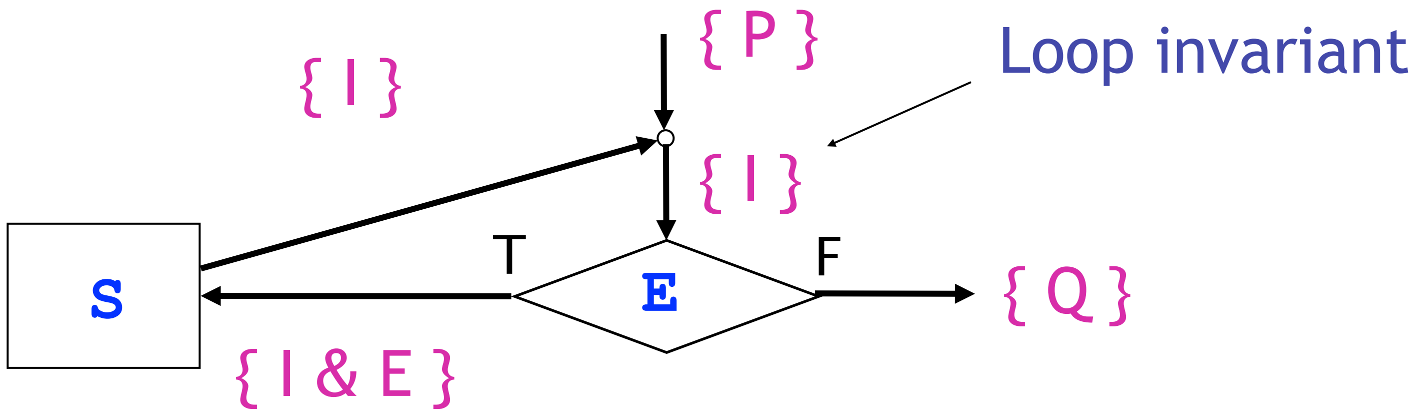
Reasoning about loops: Flow Graphs

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



Reasoning about loops: Flow Graphs

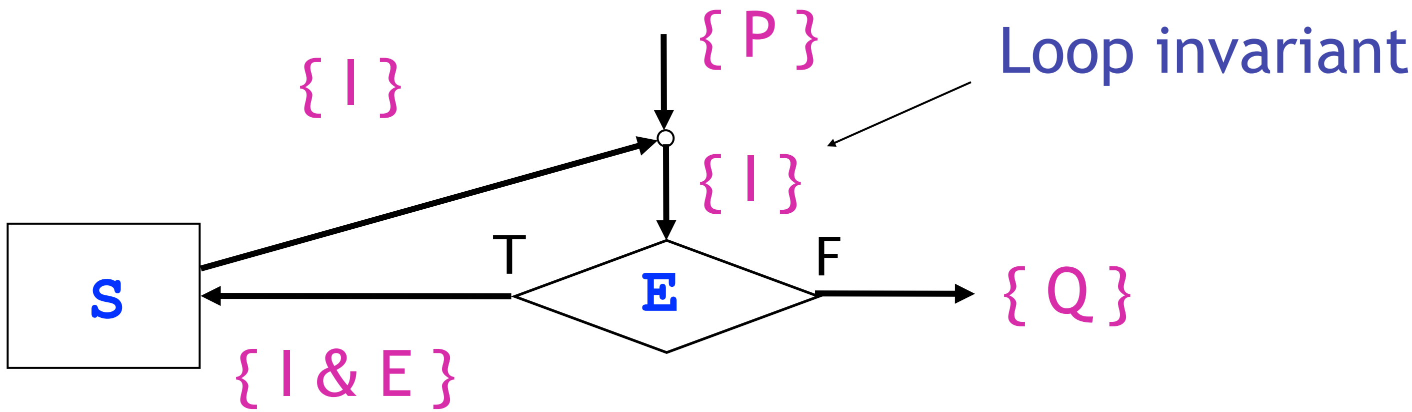
- Loops can be handled using conditionals and joins
- Consider the `while b do S` statement



if $P => I$ (loop invariant holds initially)

Reasoning about loops: Flow Graphs

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

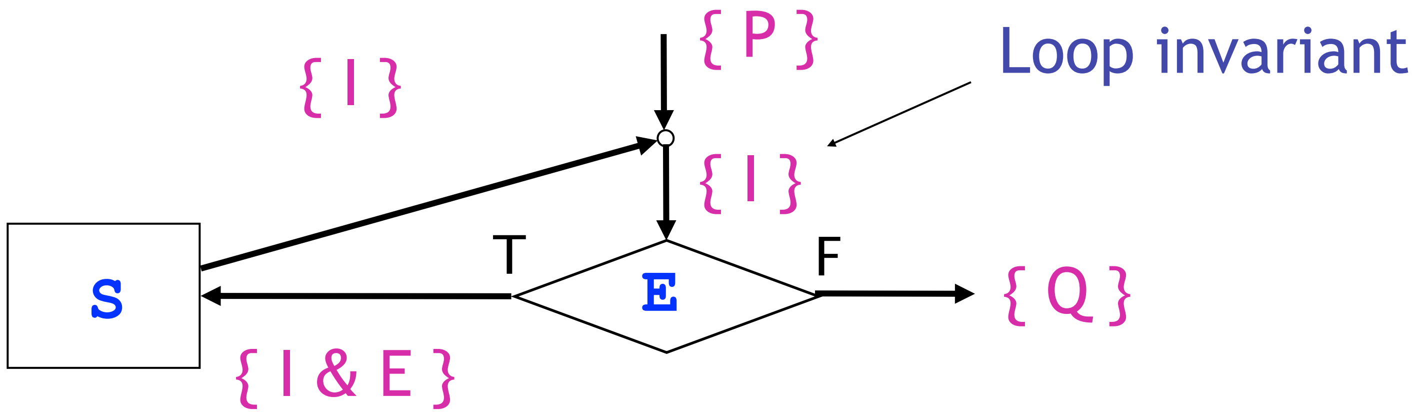


if P => I (loop invariant holds initially)

and I & !b => Q (loop establishes the postcondition)

Reasoning about loops: Flow Graphs

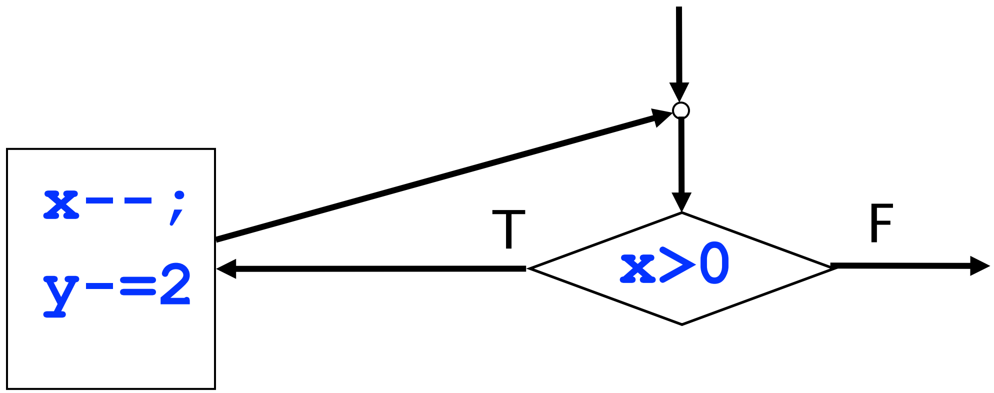
- Loops can be handled using conditionals and joins
- Consider the **`while b do S`** statement



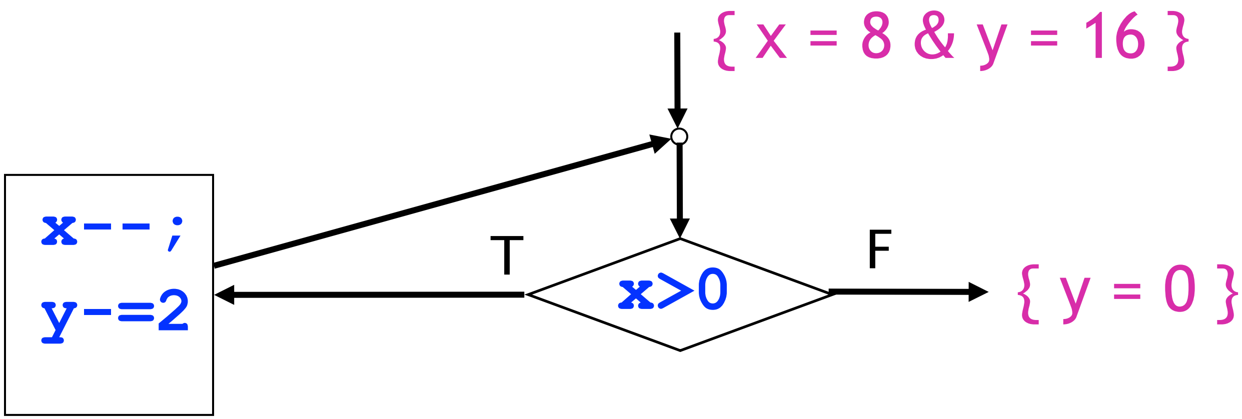
if	P => I	(loop invariant holds initially)
and	I & !b => Q	(loop establishes the postcondition)
and	{ I & b } **S** { I }	(loop invariant is preserved)

Loop Example

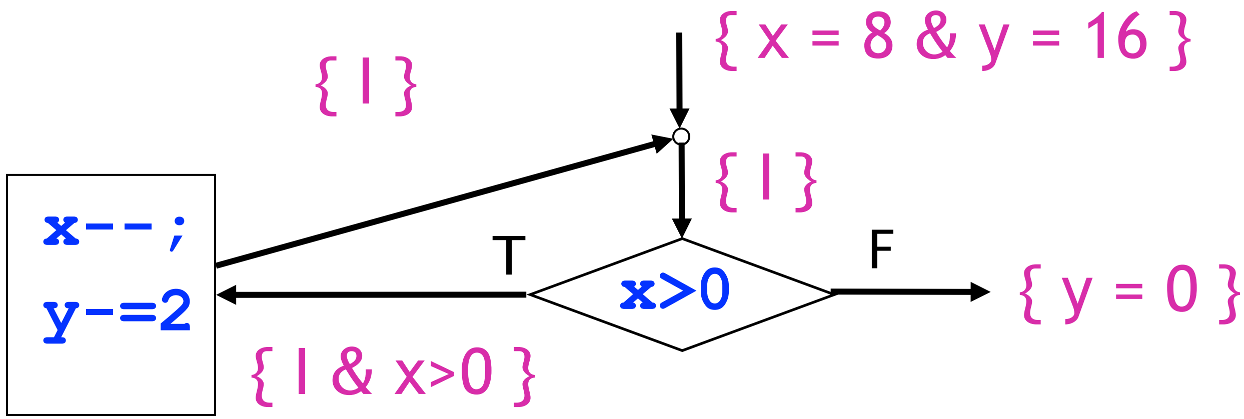
Loop Example



Loop Example



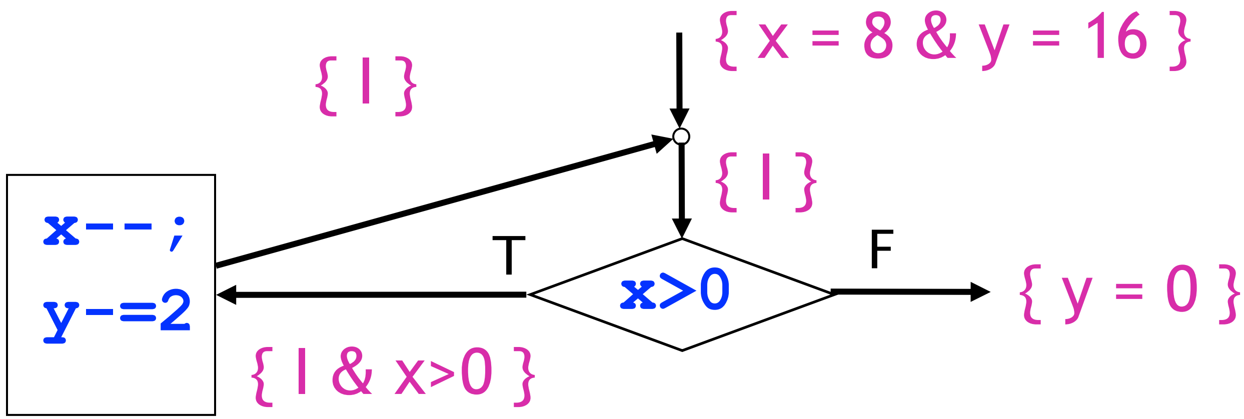
Loop Example



Loop Example

Verify:

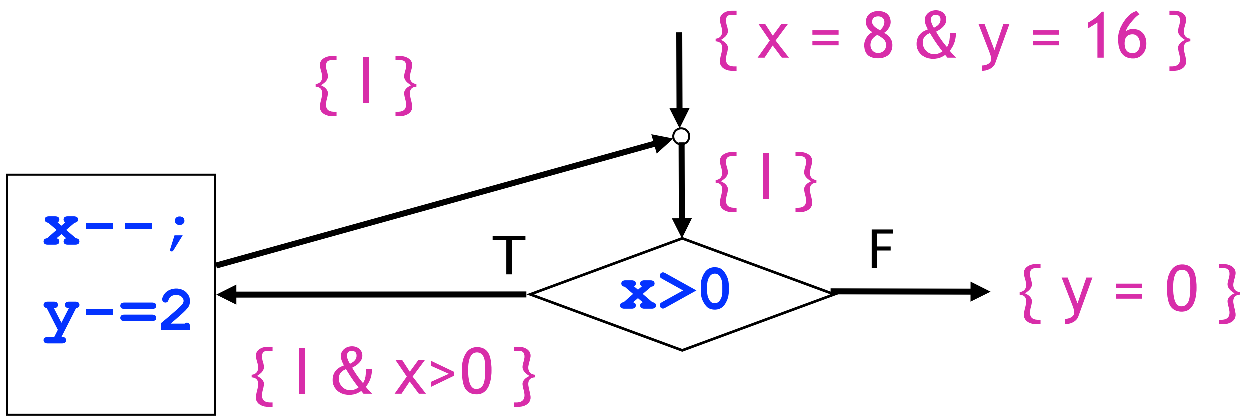
{x=8 & y=16} `while(x>0){x--; y-=2;}` {y = 0}



Loop Example

Verify:

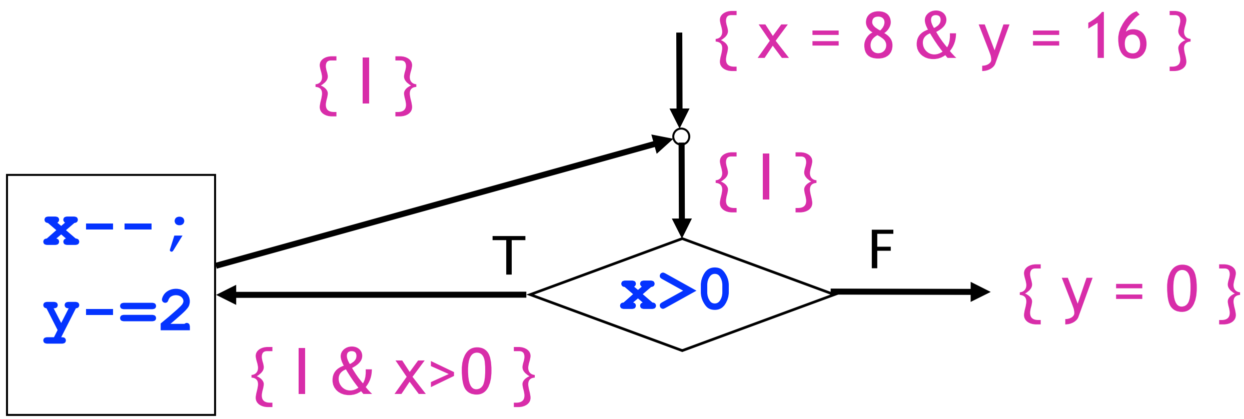
{x=8 & y=16} `while(x>0){x--; y-=2;}` {y = 0}



Loop Example

Verify:

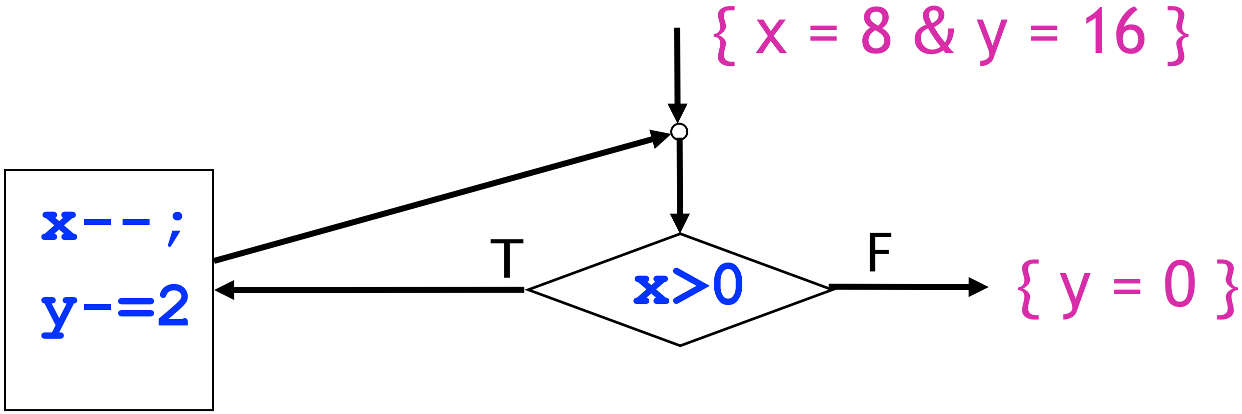
{x=8 & y=16} `while(x>0){x--; y-=2;}` {y = 0}



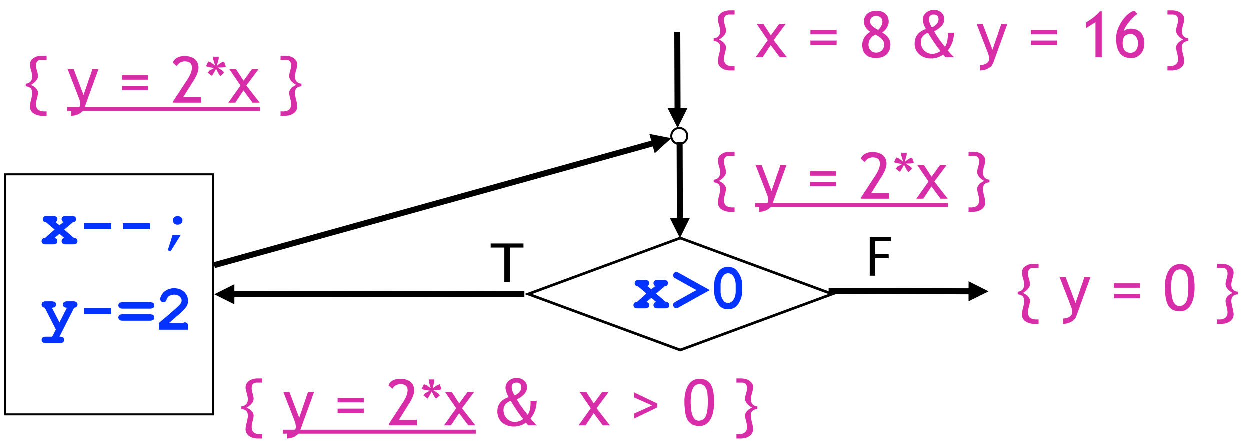
Find an appropriate invariant I

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

Loop Example (II)

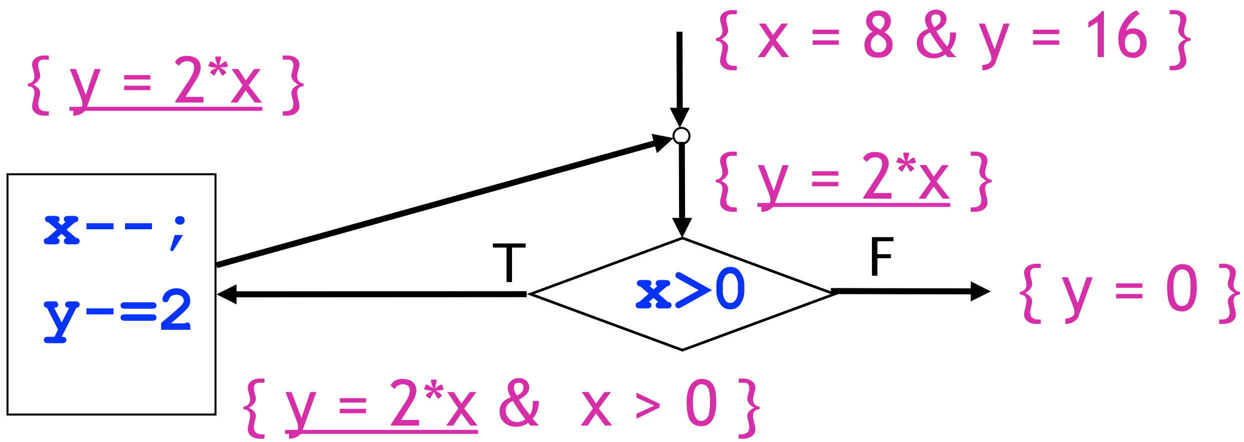


Loop Example (II)



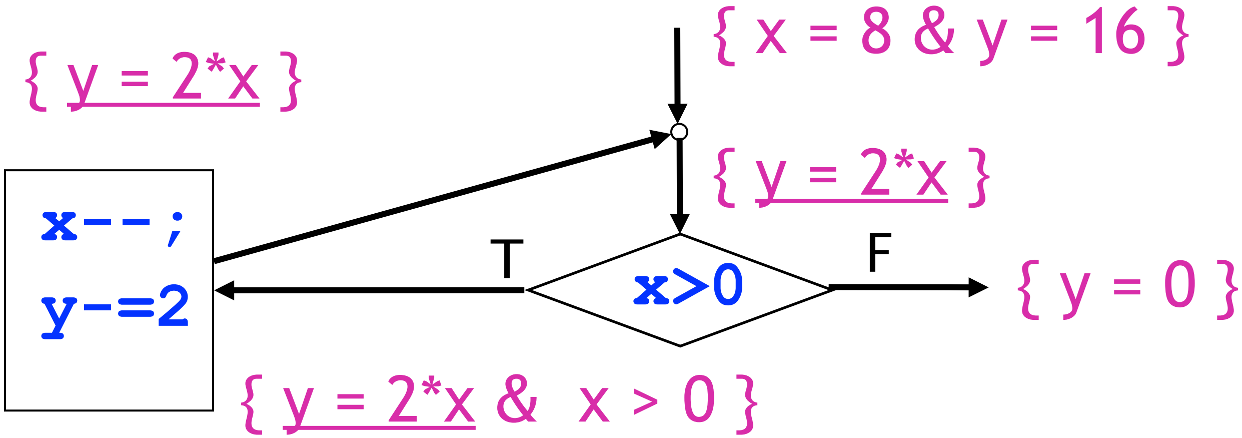
Loop Example (II)

Guess invariant y = 2*x



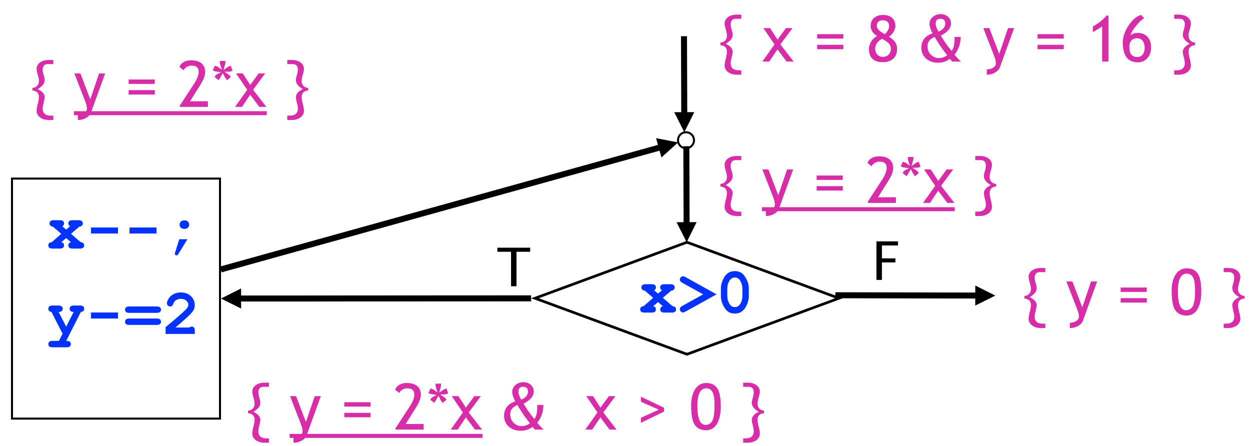
Loop Example (II)

Guess invariant y = 2*x



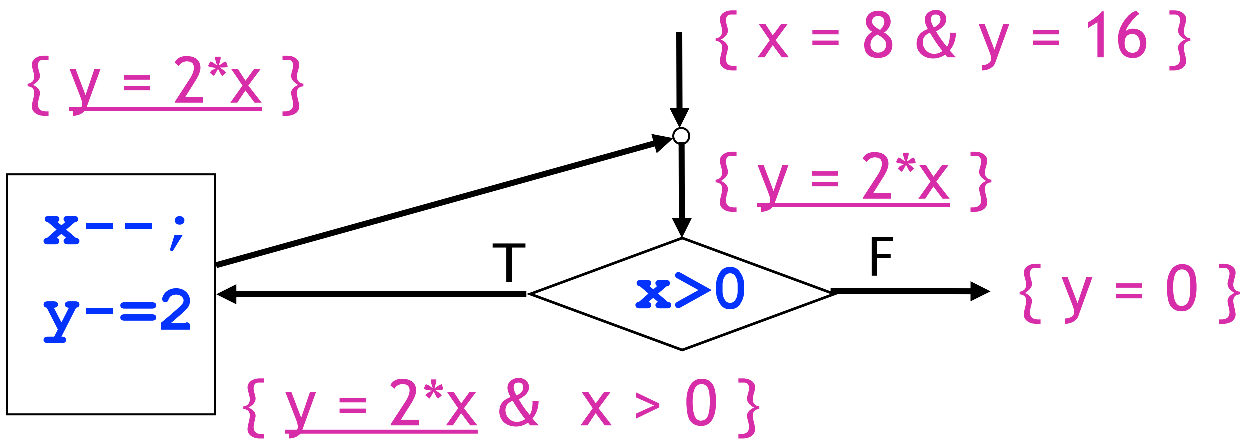
Loop Example (II)

Guess invariant y = 2*x



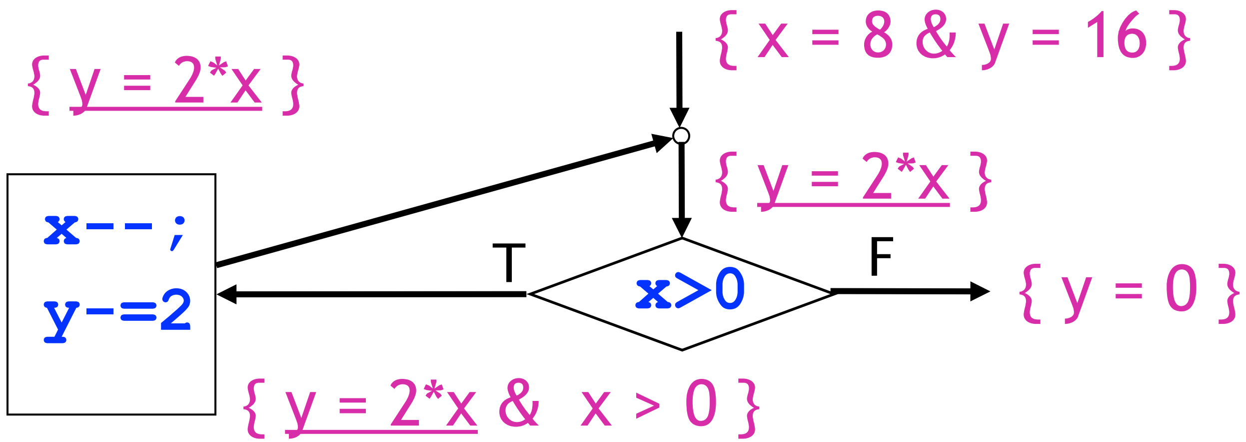
Loop Example (II)

Guess invariant <u>$y = 2*x$</u>



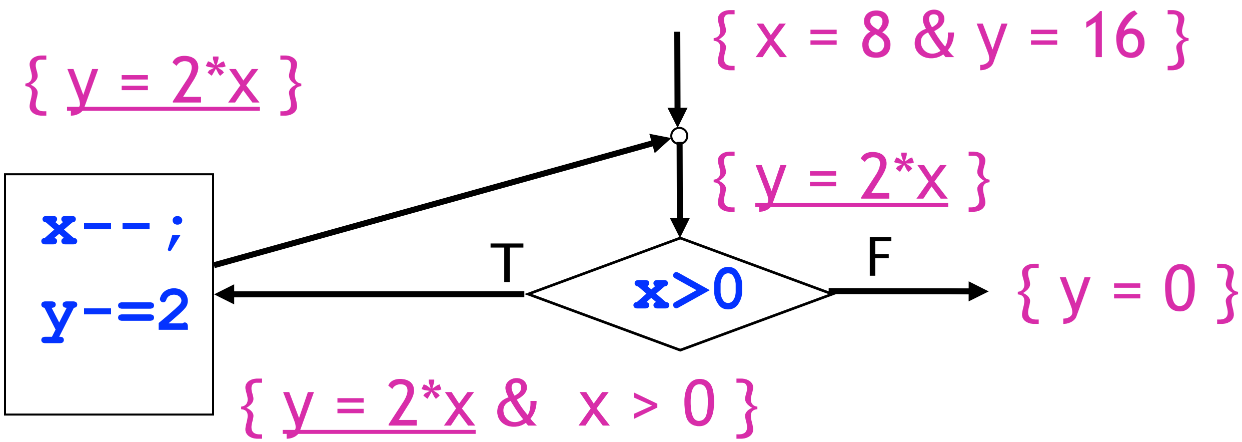
Loop Example (II)

Guess invariant y = 2*x



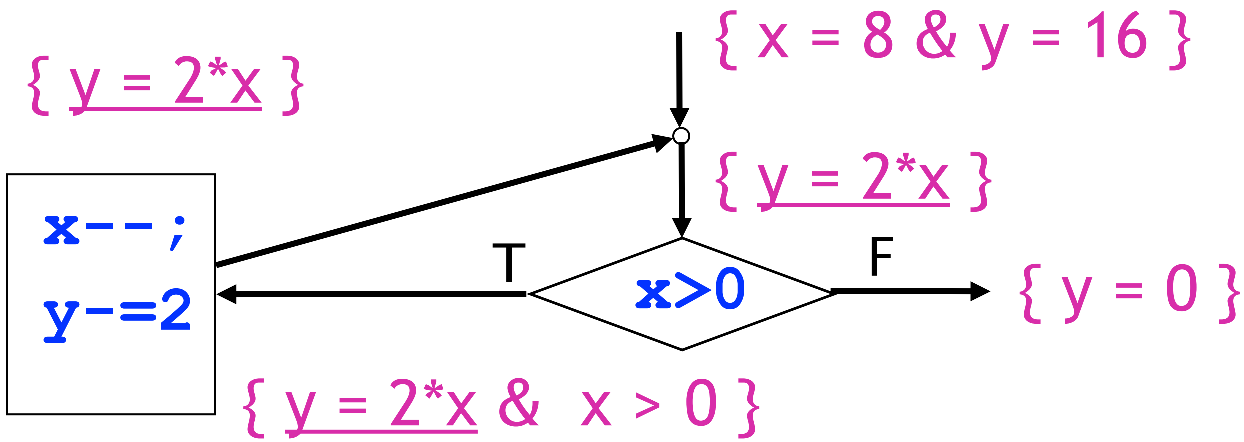
Loop Example (II)

Guess invariant y = 2*x



Loop Example (II)

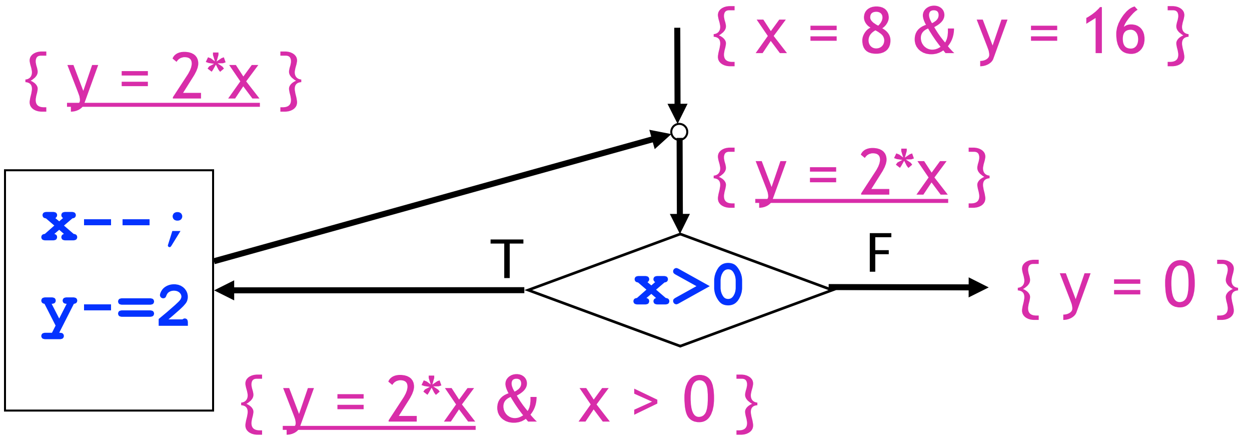
Guess invariant y = 2*x



Check :

Loop Example (II)

Guess invariant y = 2*x

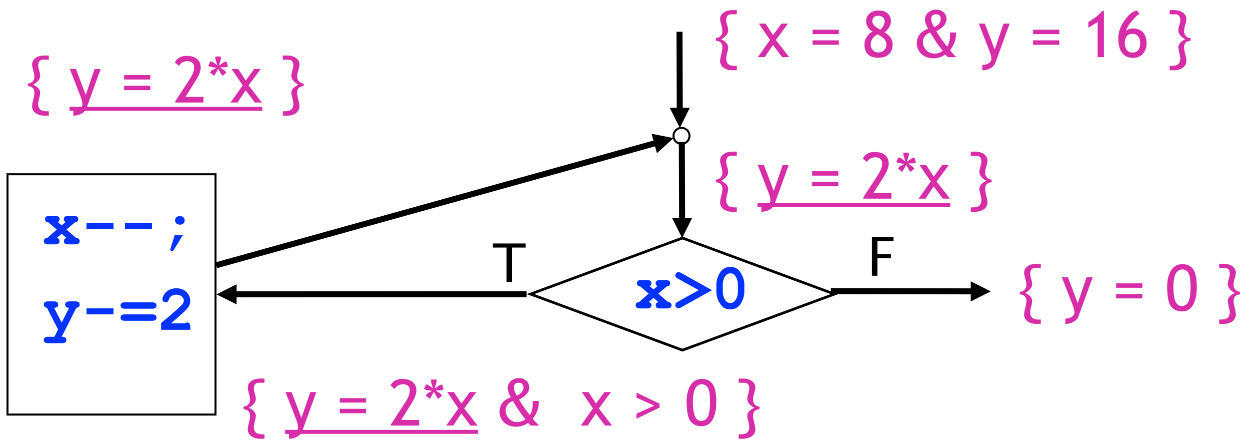


Check :

- Initial: x = 8 & y = 16 => y = 2*x

Loop Example (II)

Guess invariant y = 2*x

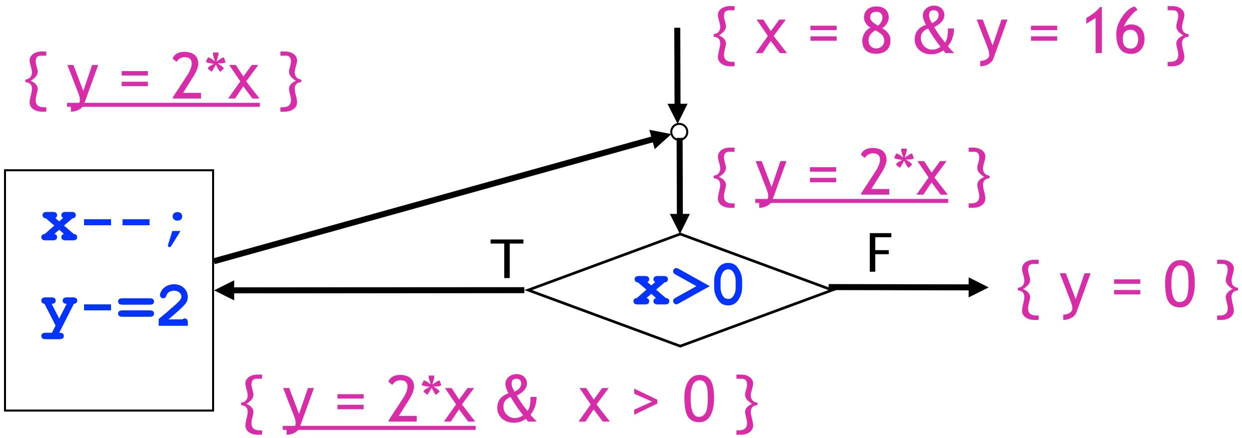


Check :

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

Loop Example (II)

Guess invariant y = 2*x

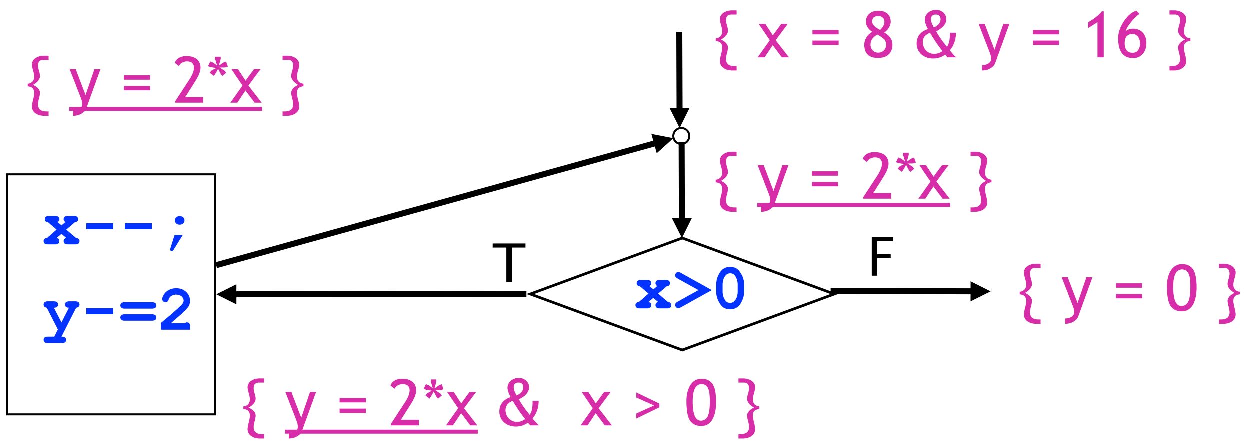


Check :

- 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

Loop Example (II)

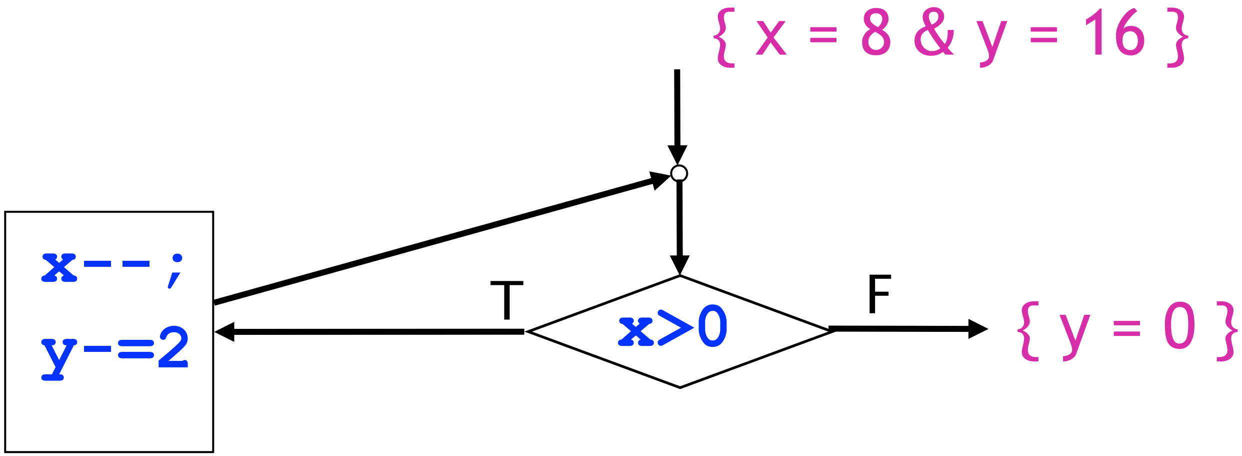
Guess invariant y = 2*x



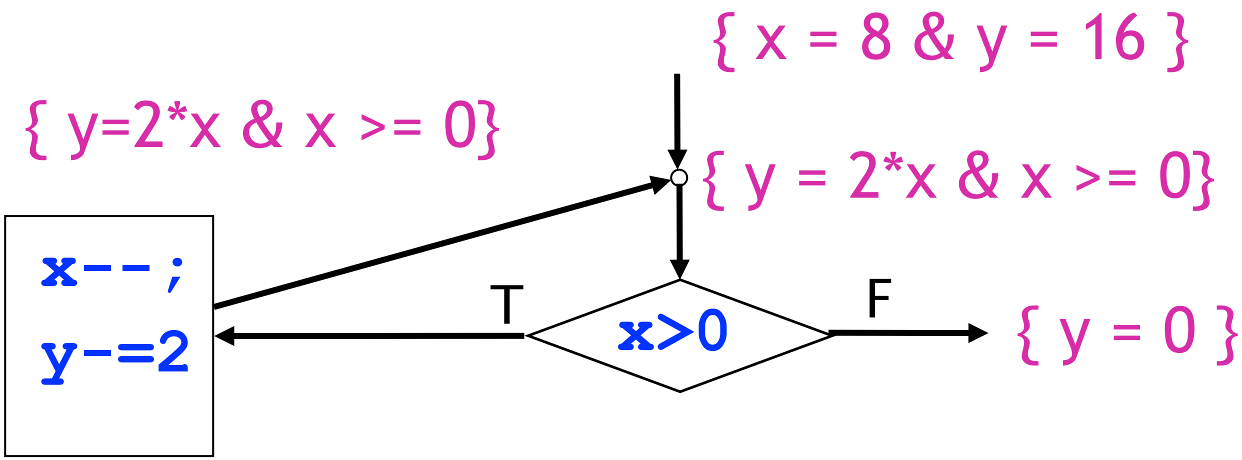
Check :

- 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)

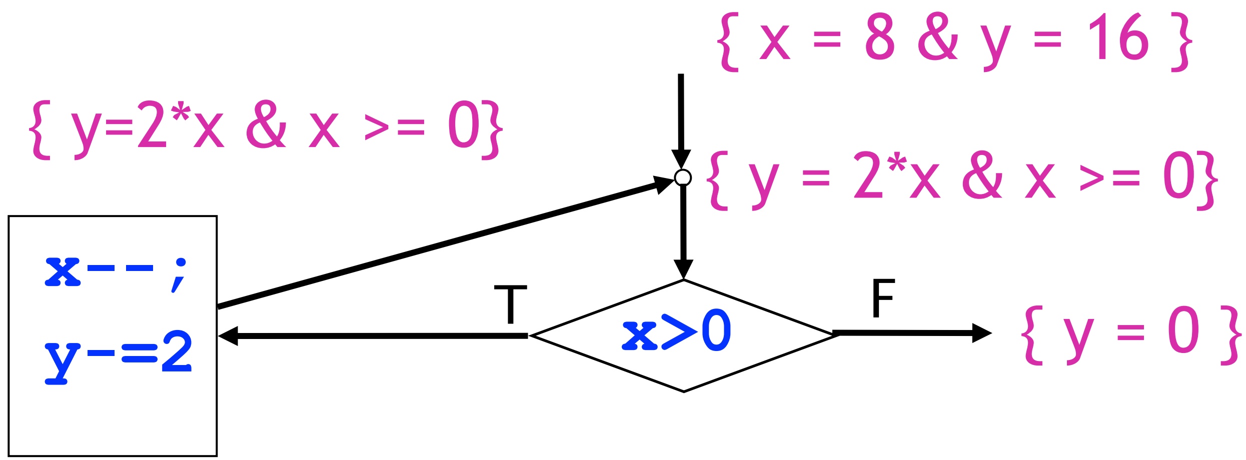


Loop Example (III)



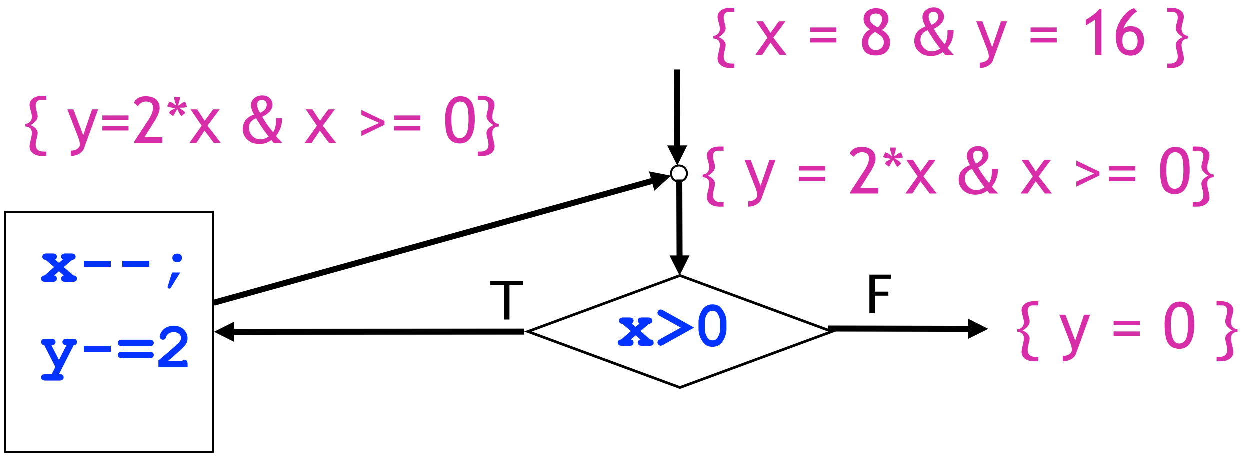
Loop Example (III)

Guess invariant y = 2*x & x >= 0



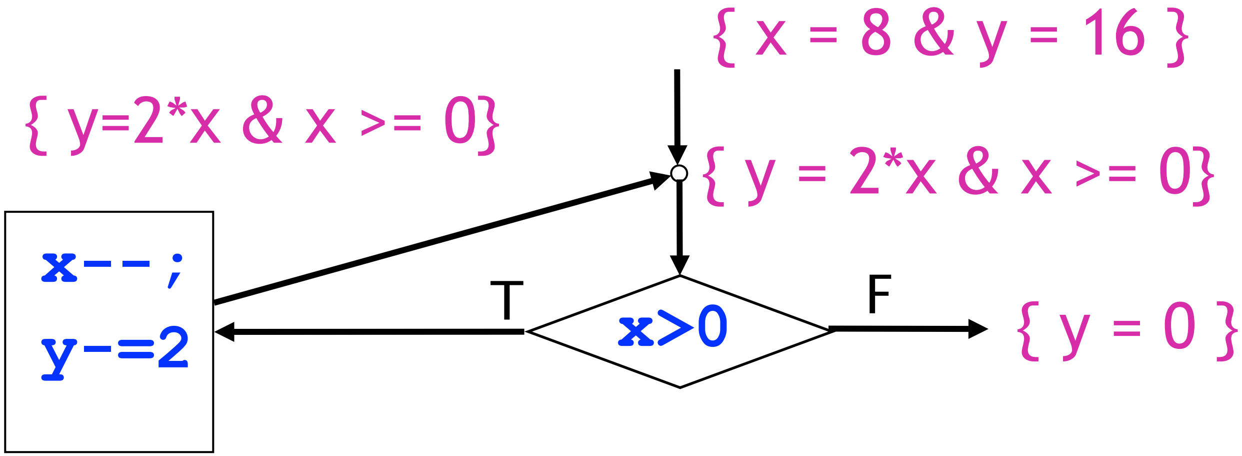
Loop Example (III)

Guess invariant y = 2*x & x >= 0



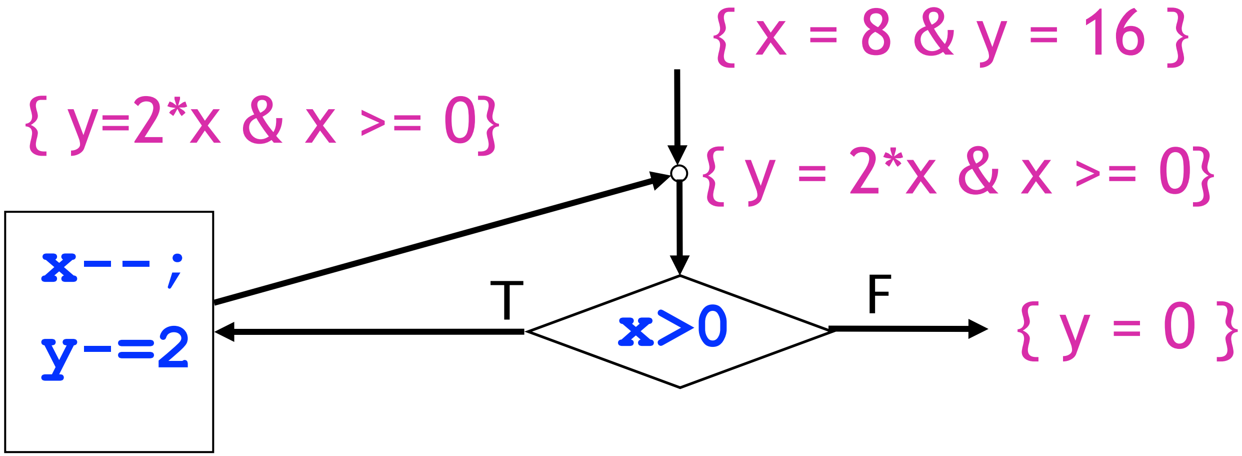
Loop Example (III)

Guess invariant y = 2*x & x >= 0



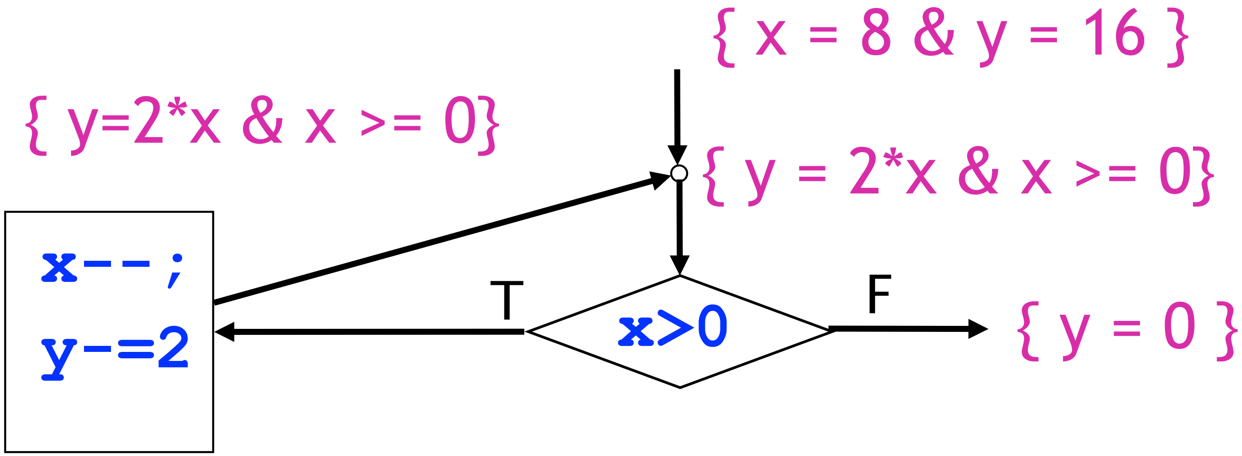
Loop Example (III)

Guess invariant y = 2*x & x >= 0



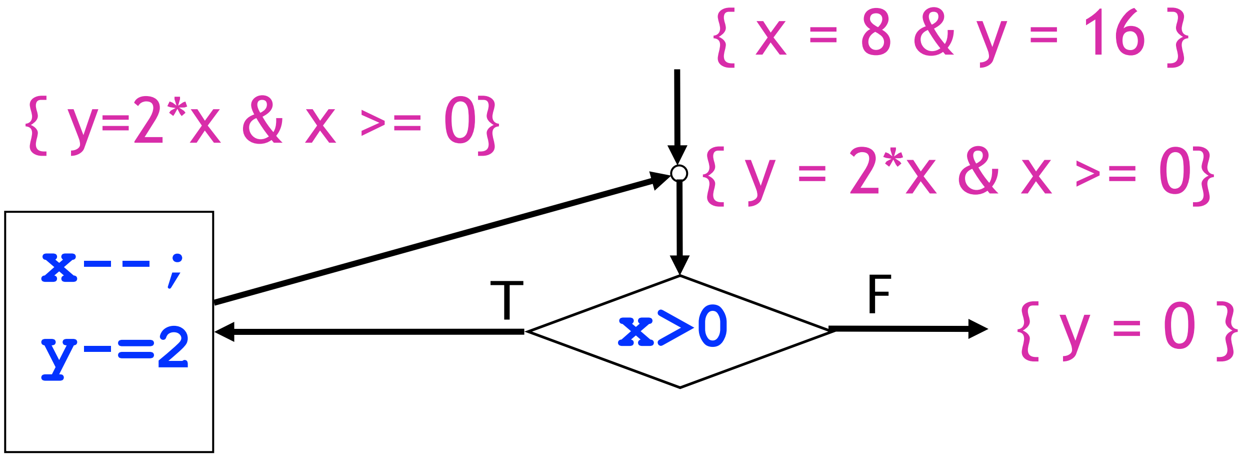
Loop Example (III)

Guess invariant y = 2*x & x >= 0



Loop Example (III)

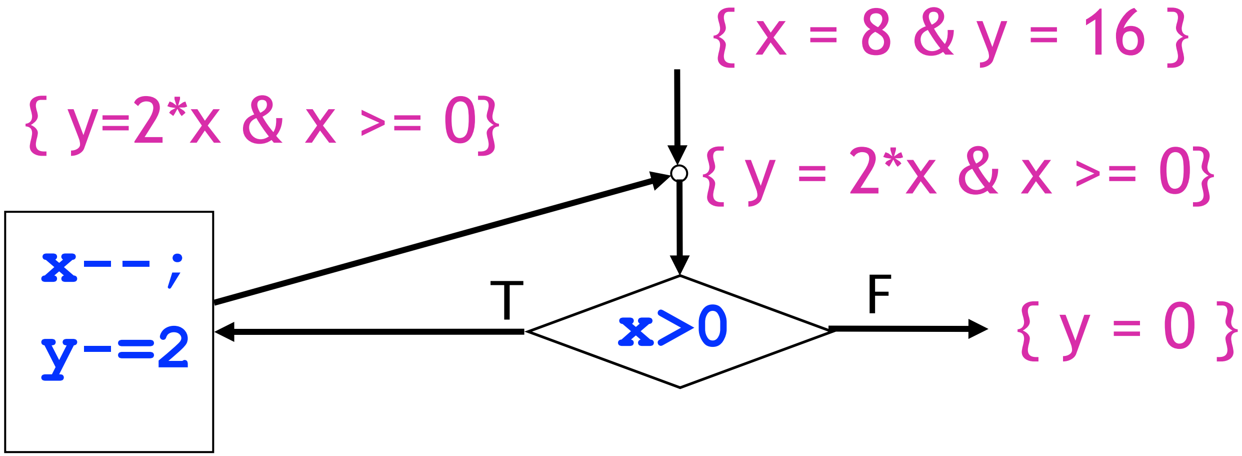
Guess invariant y = 2*x & x >= 0



Check

Loop Example (III)

Guess invariant y = 2*x & x >= 0

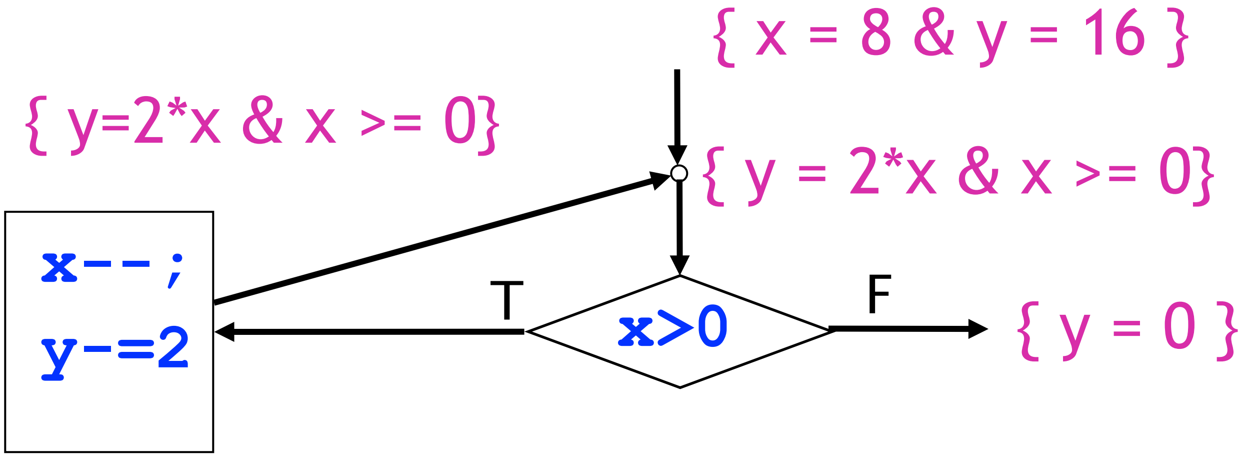


Check

- Initial : x = 8 & y = 16 => y = 2*x & x >= 0

Loop Example (III)

Guess invariant y = 2*x & x >= 0

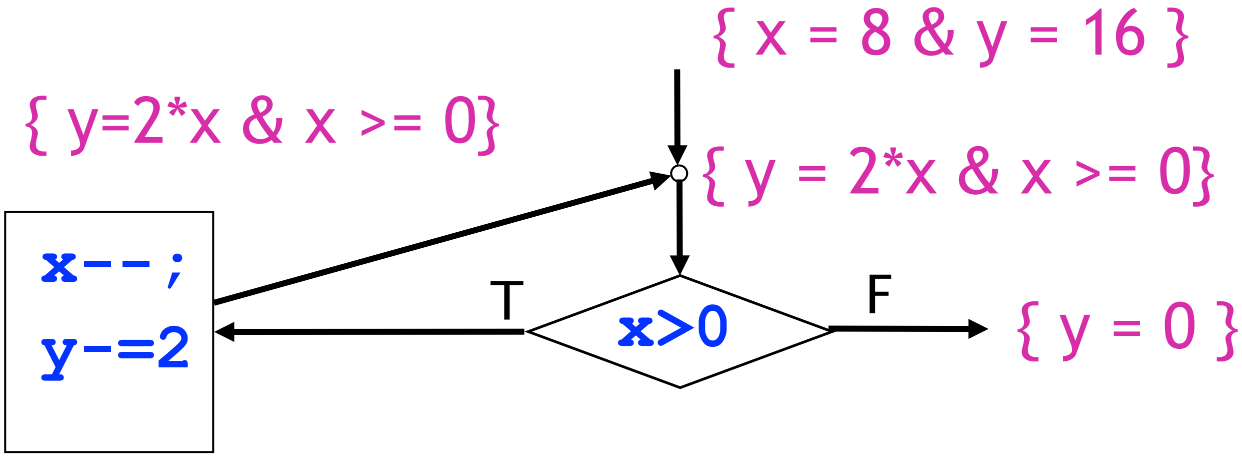


Check

- 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

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

Loops Discussion

- Simple forward/backward propagation fails

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)

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

```
        k = k + 1;
    }
    return r;
}
```

Verification Example

```
    }
    return r;
}
```

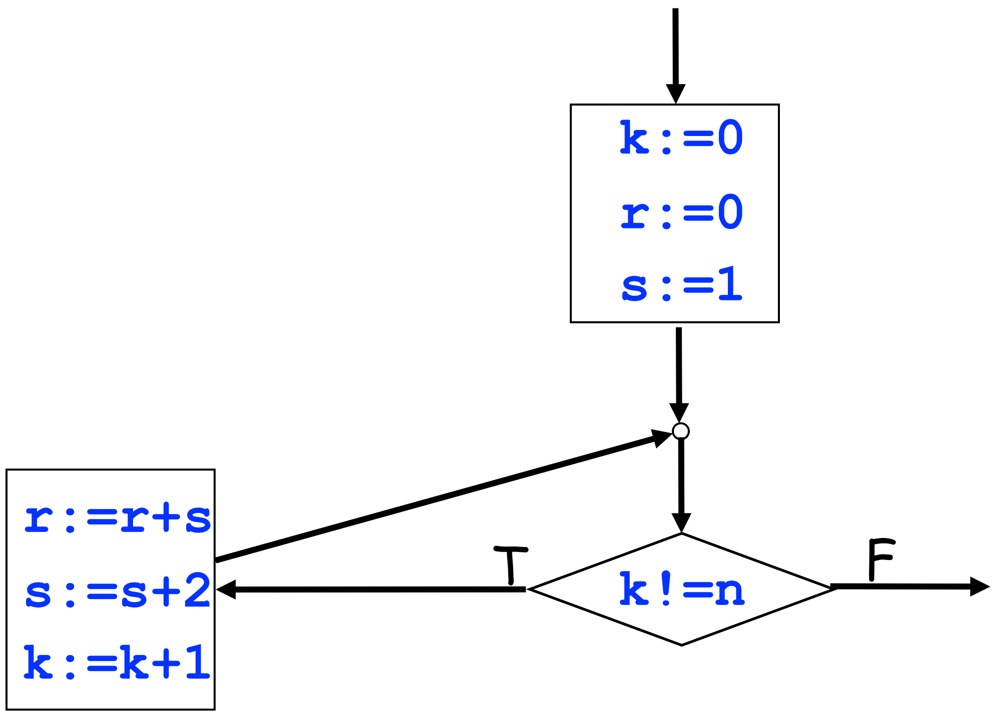
Verification Example

```
    return r;
  }
```

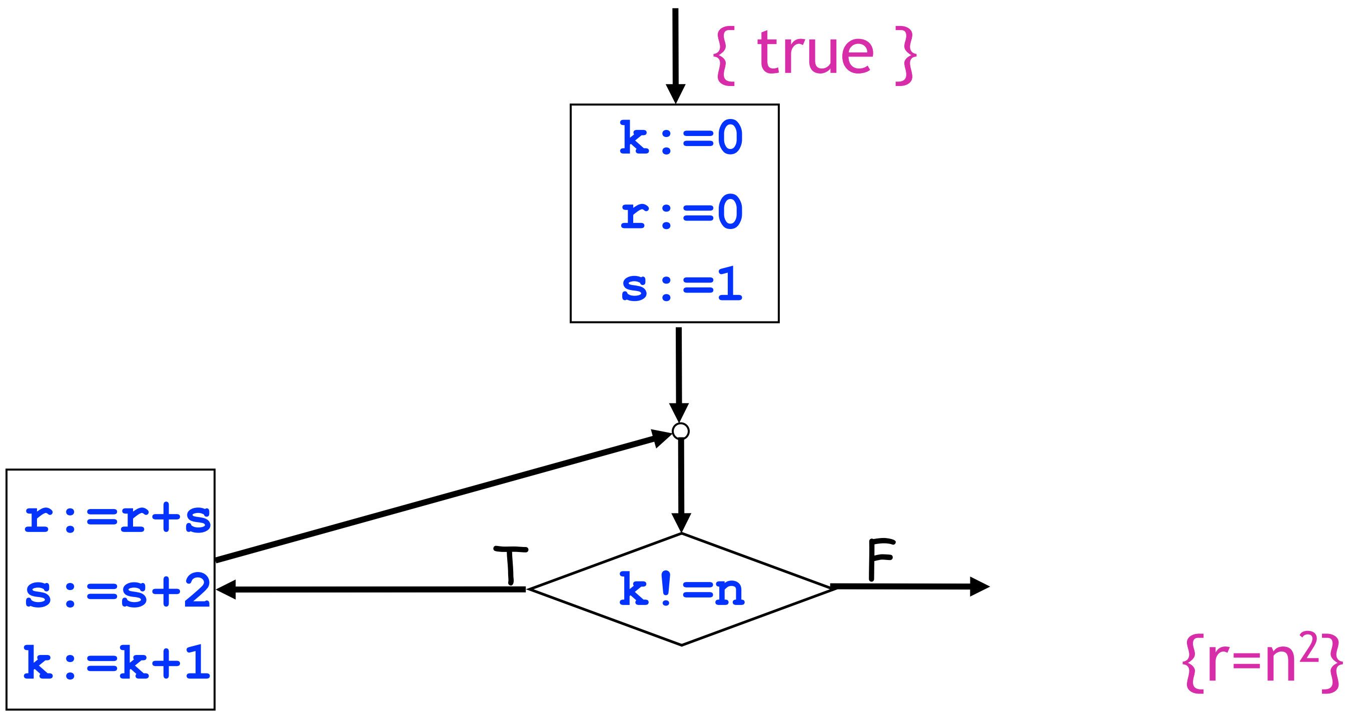
Verification Example

```
}
```

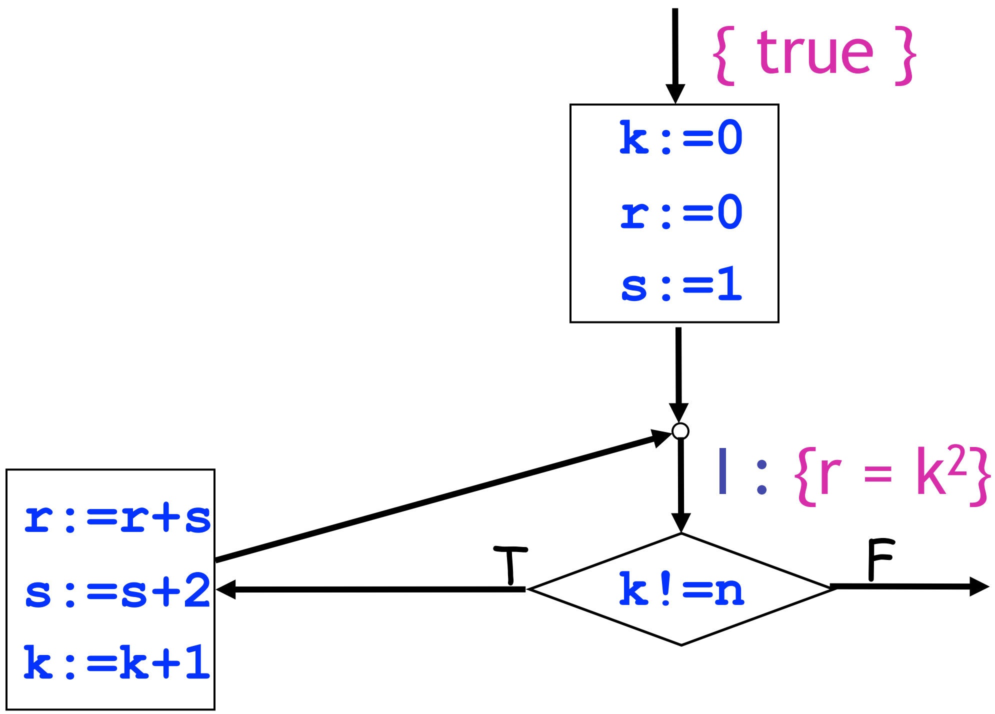
Verification Example



Verification Example



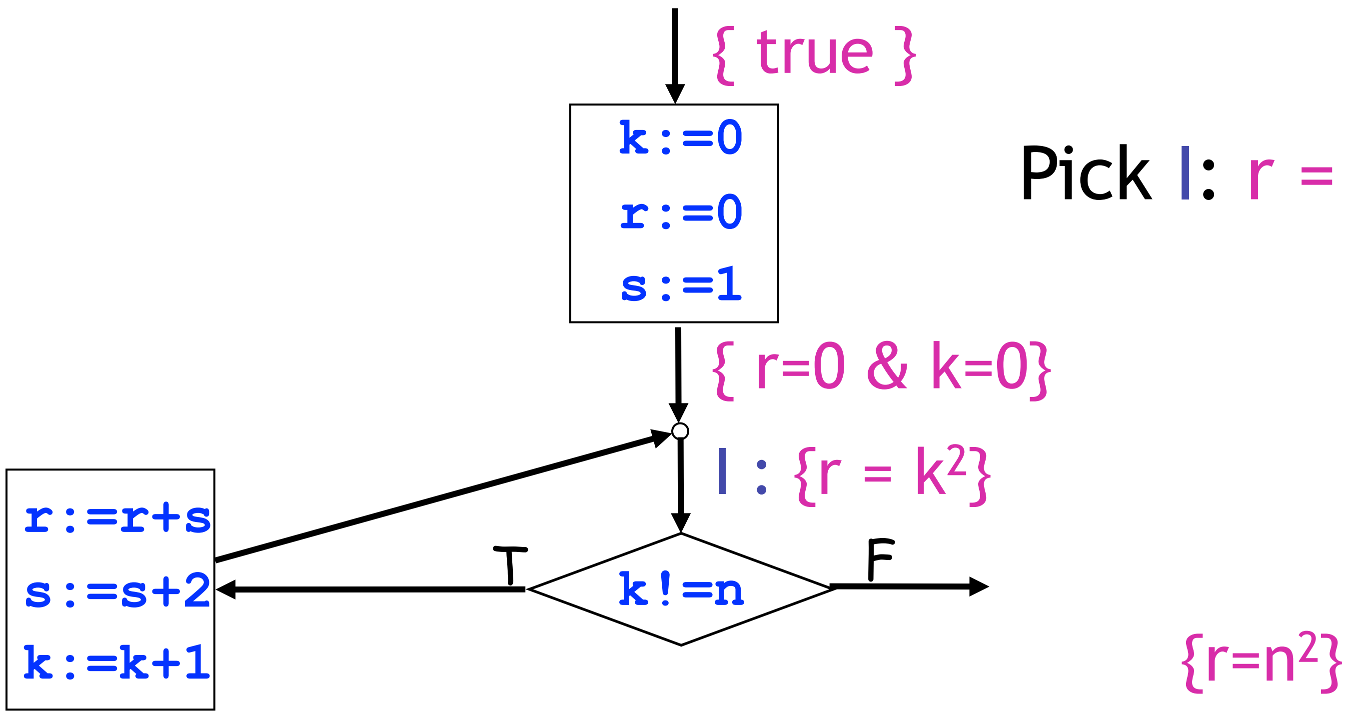
Verification Example



Pick I: $r = k^2$

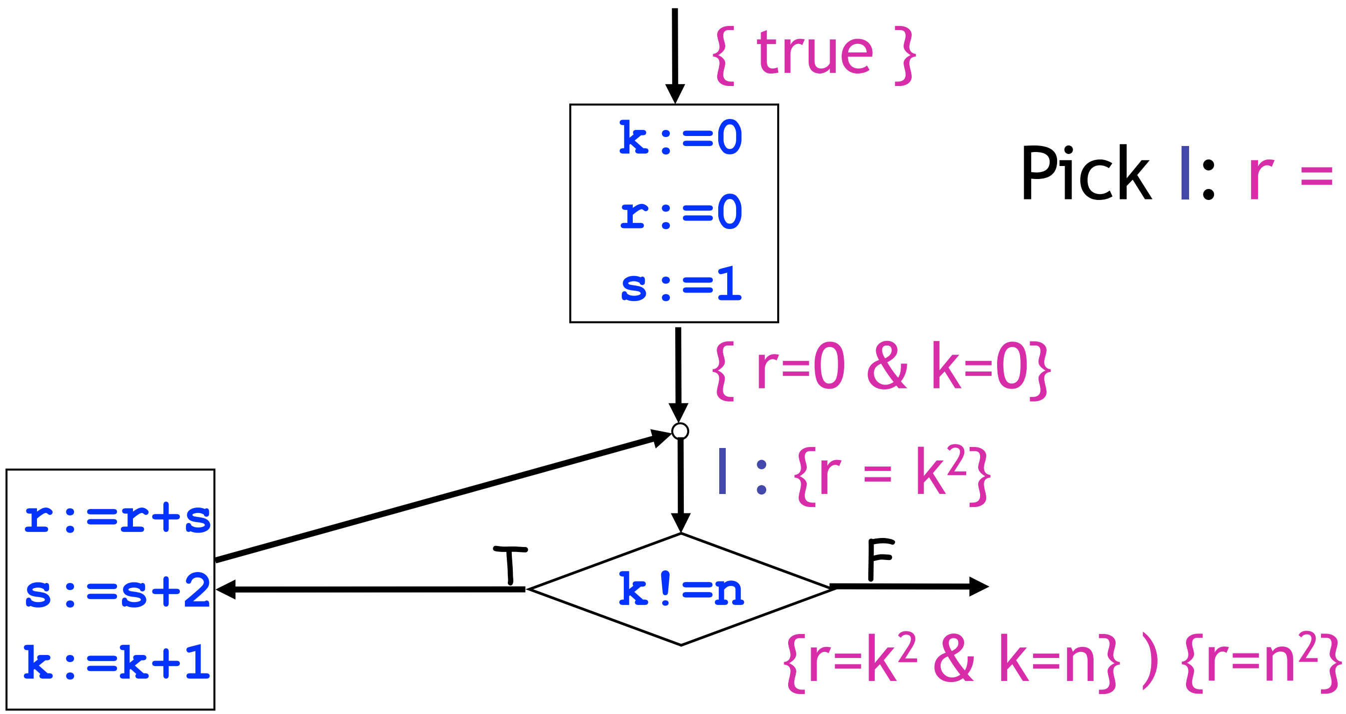
$\{r=n^2\}$

Verification Example



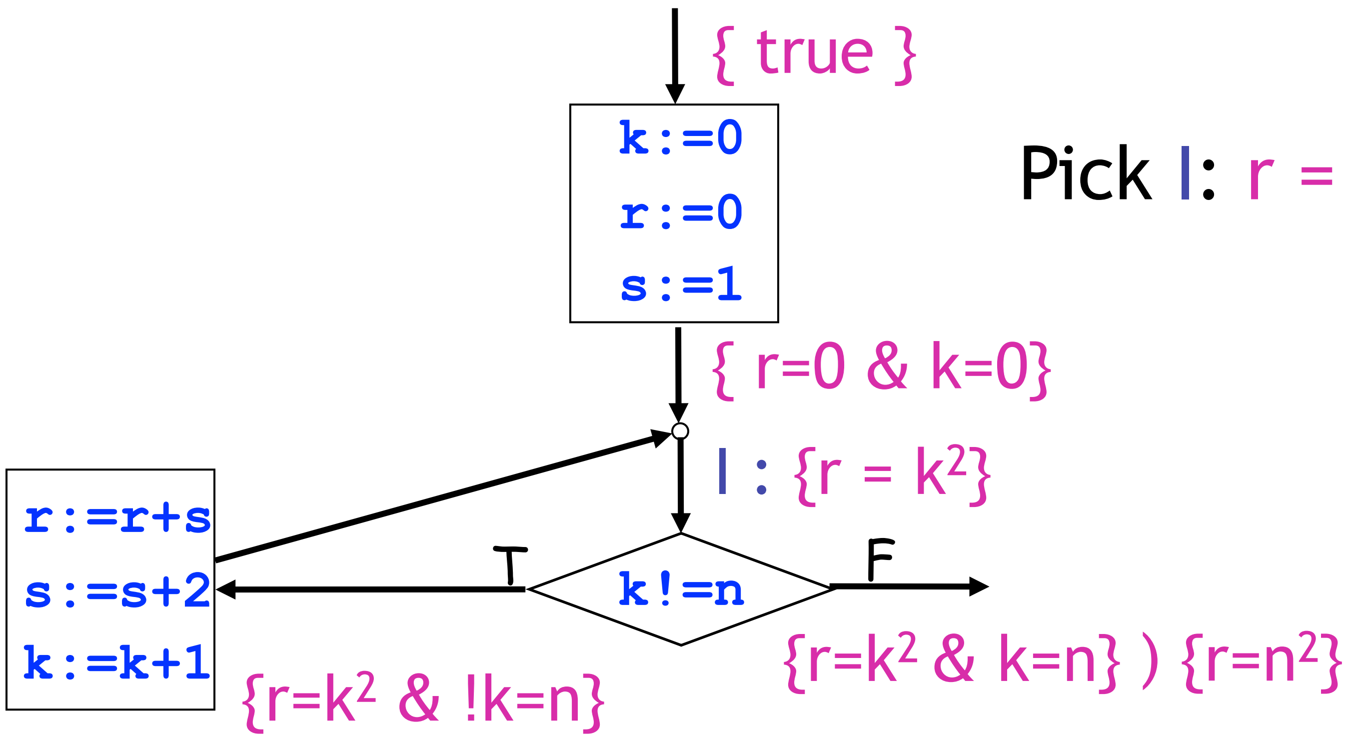
Pick I: $r = k^2$

Verification Example



Pick I: $r = k^2$

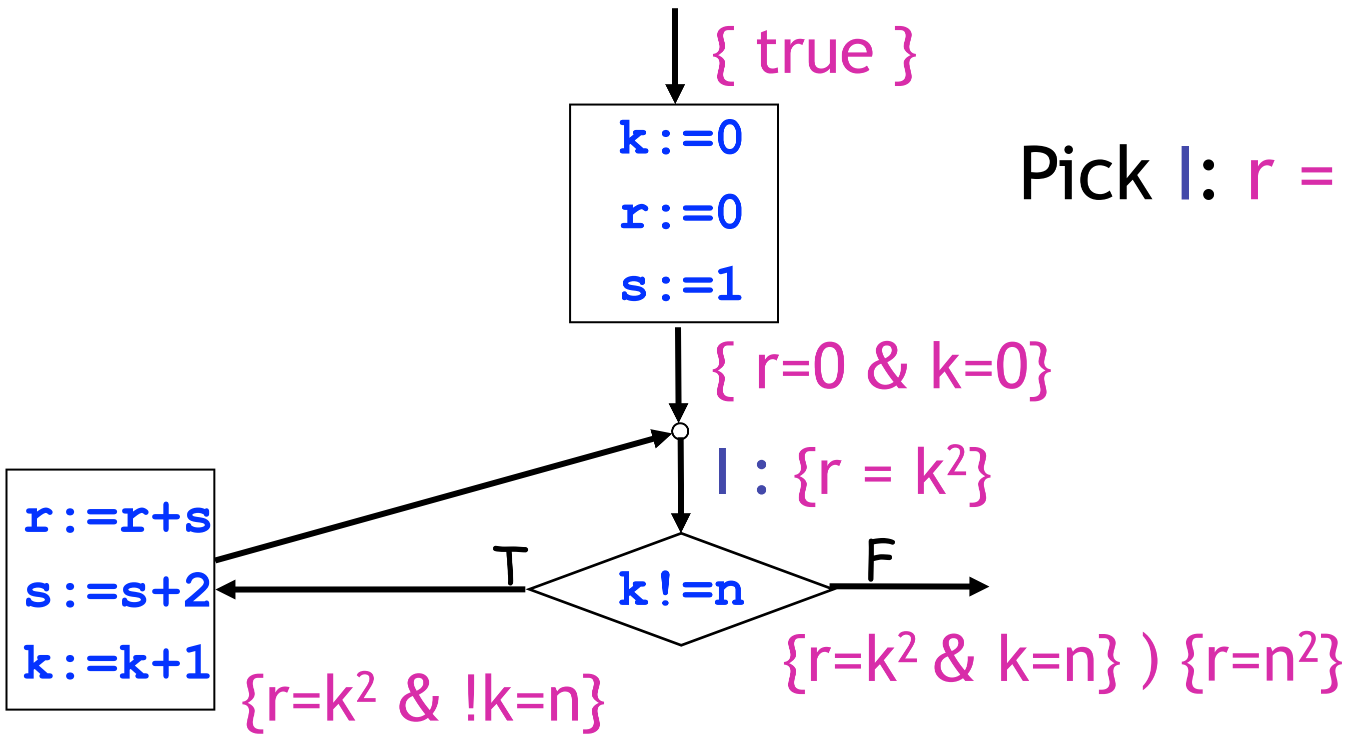
Verification Example



Pick I: $r = k^2$

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

Verification Example

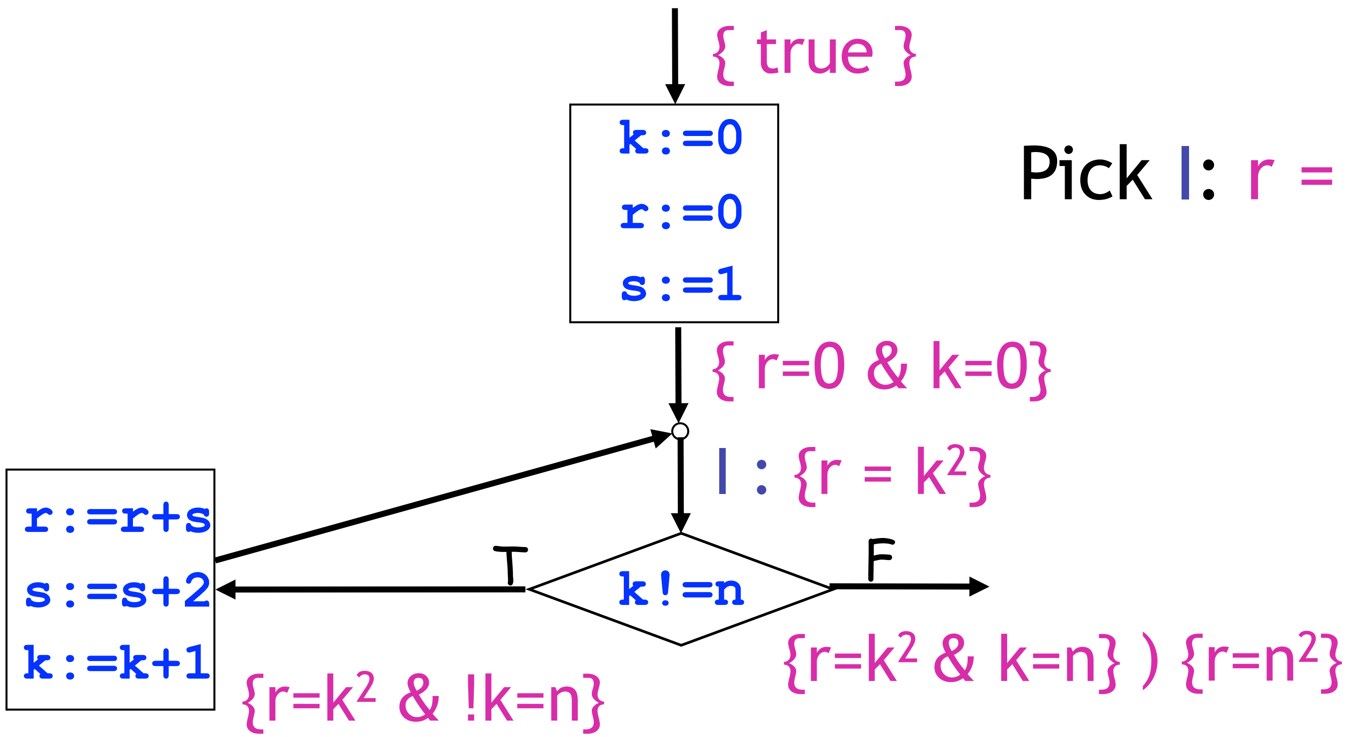


Pick I: $r = k^2$

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

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

Verification Example



Pick I: r = k^2

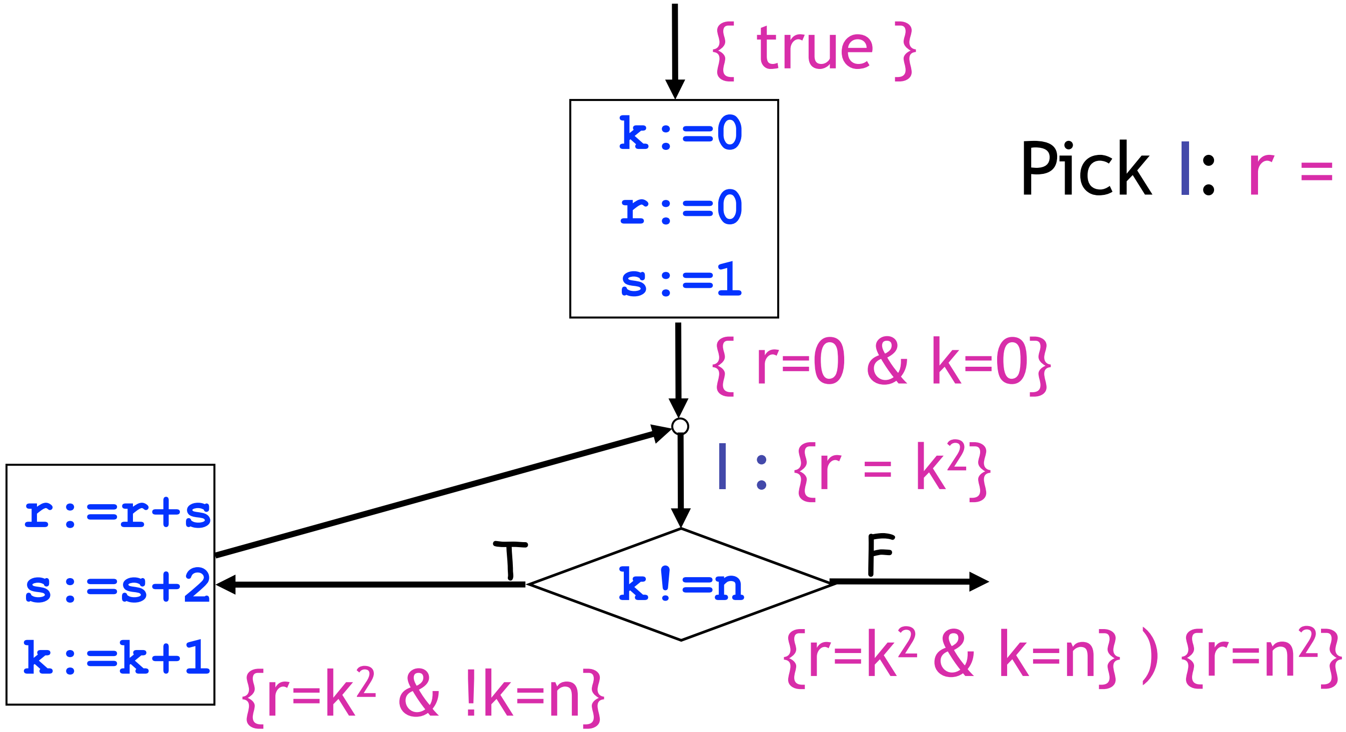
Need: {r=k^2 & !k=n} **c** {r=k^2}

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

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

Verification Example

Pick I: $r = k^2$



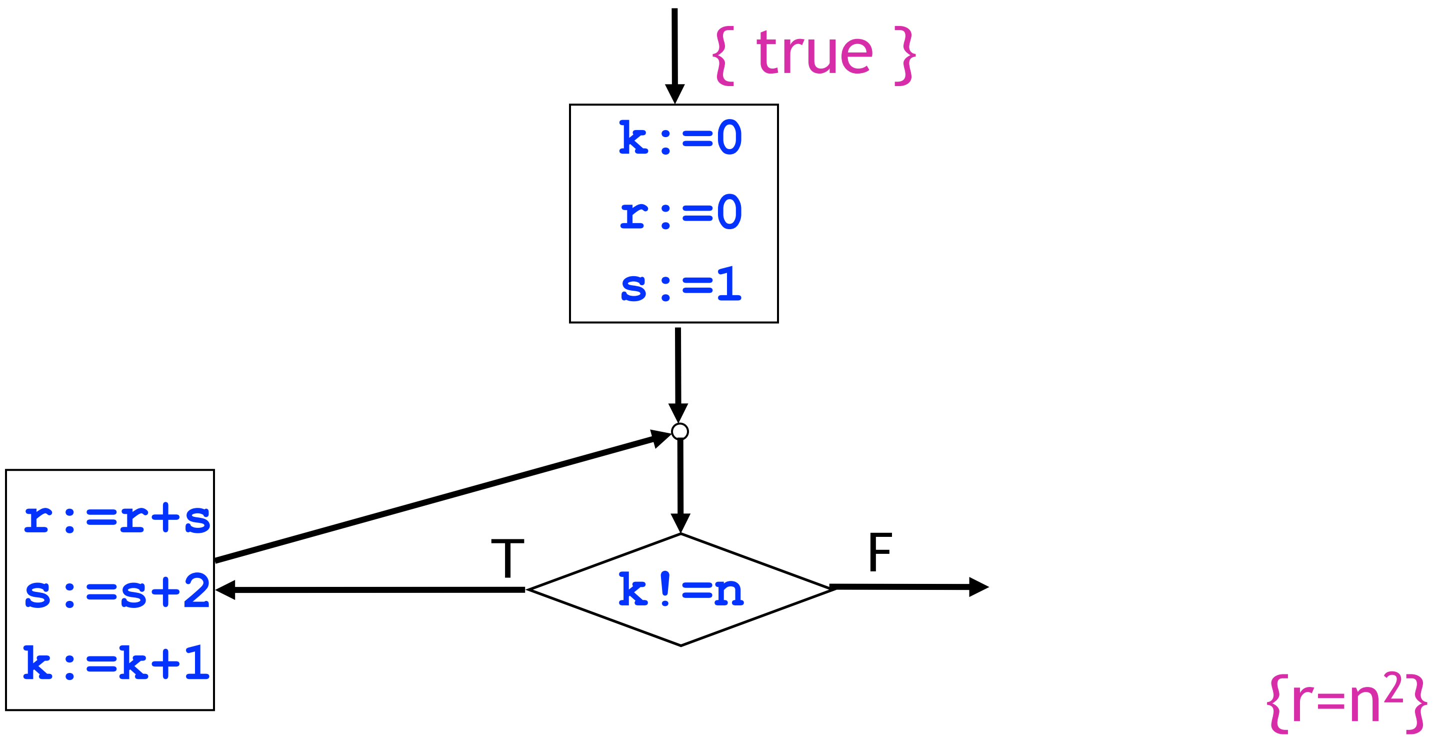
Need: $\{r=k^2 \;\&\; !k=n\}$ **c** $\{r=k^2\}$

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

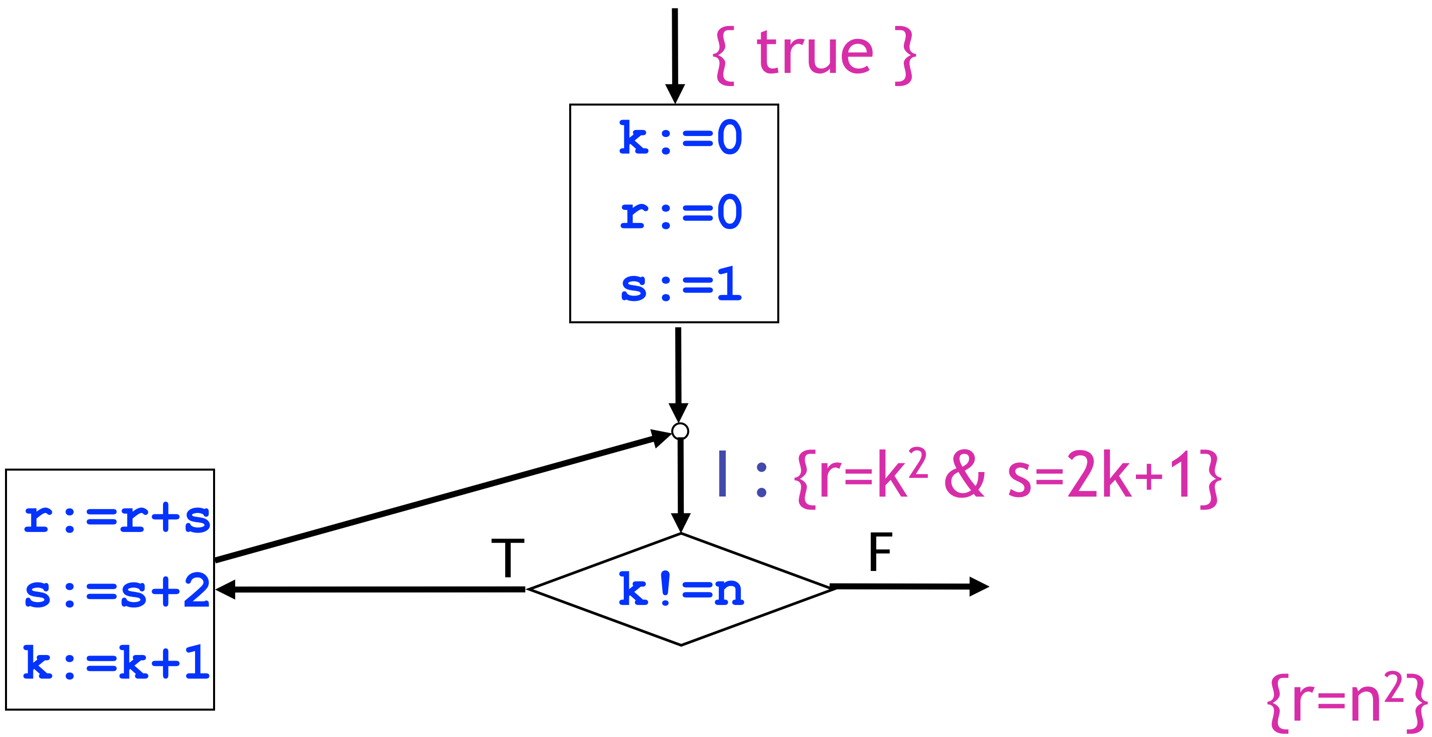
i.e. $\{r=k^2 \;\&\; !k=n\}$ => $\{r+s=(k+1)^2\}$

Invalid

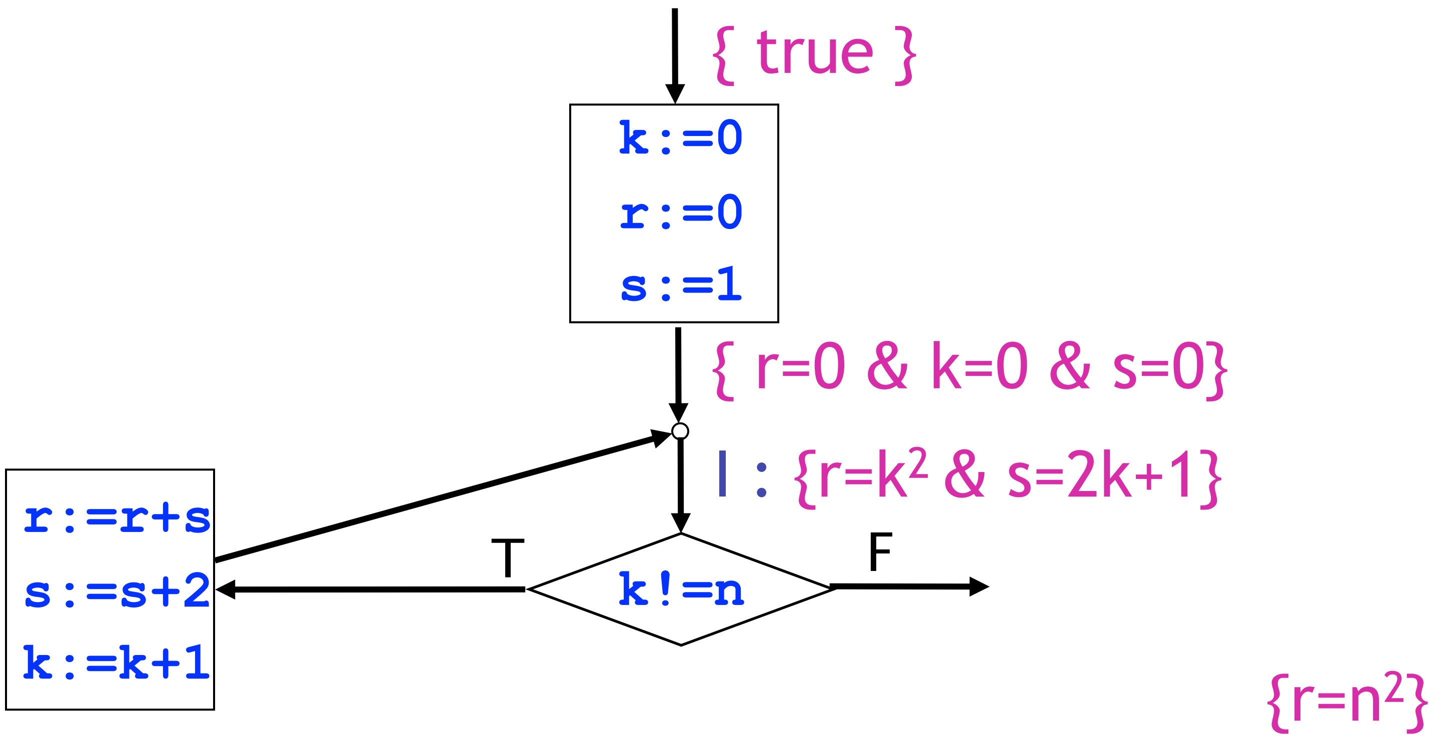
Verification Example



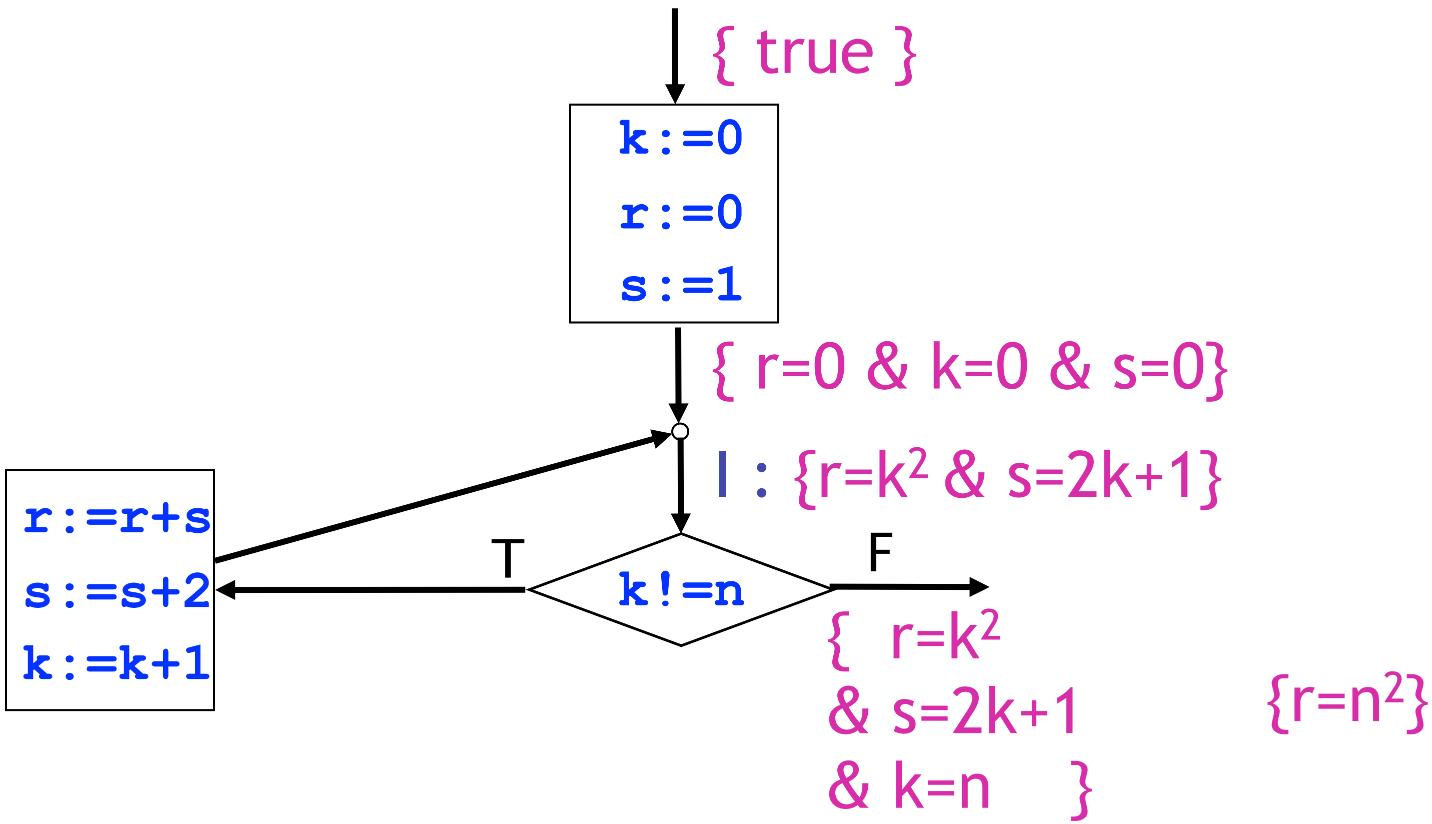
Verification Example



Verification Example

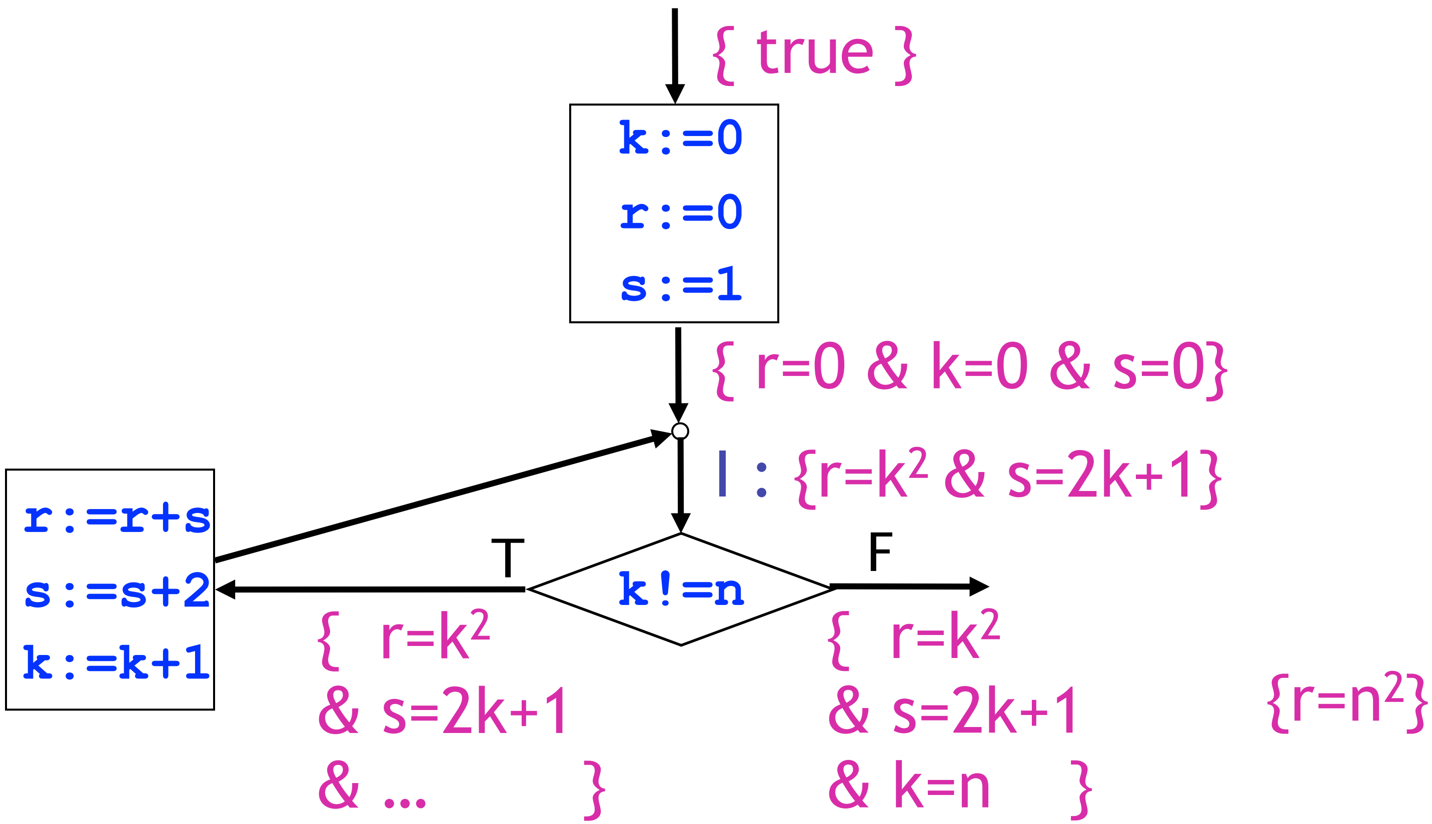


Verification Example



Verification Example

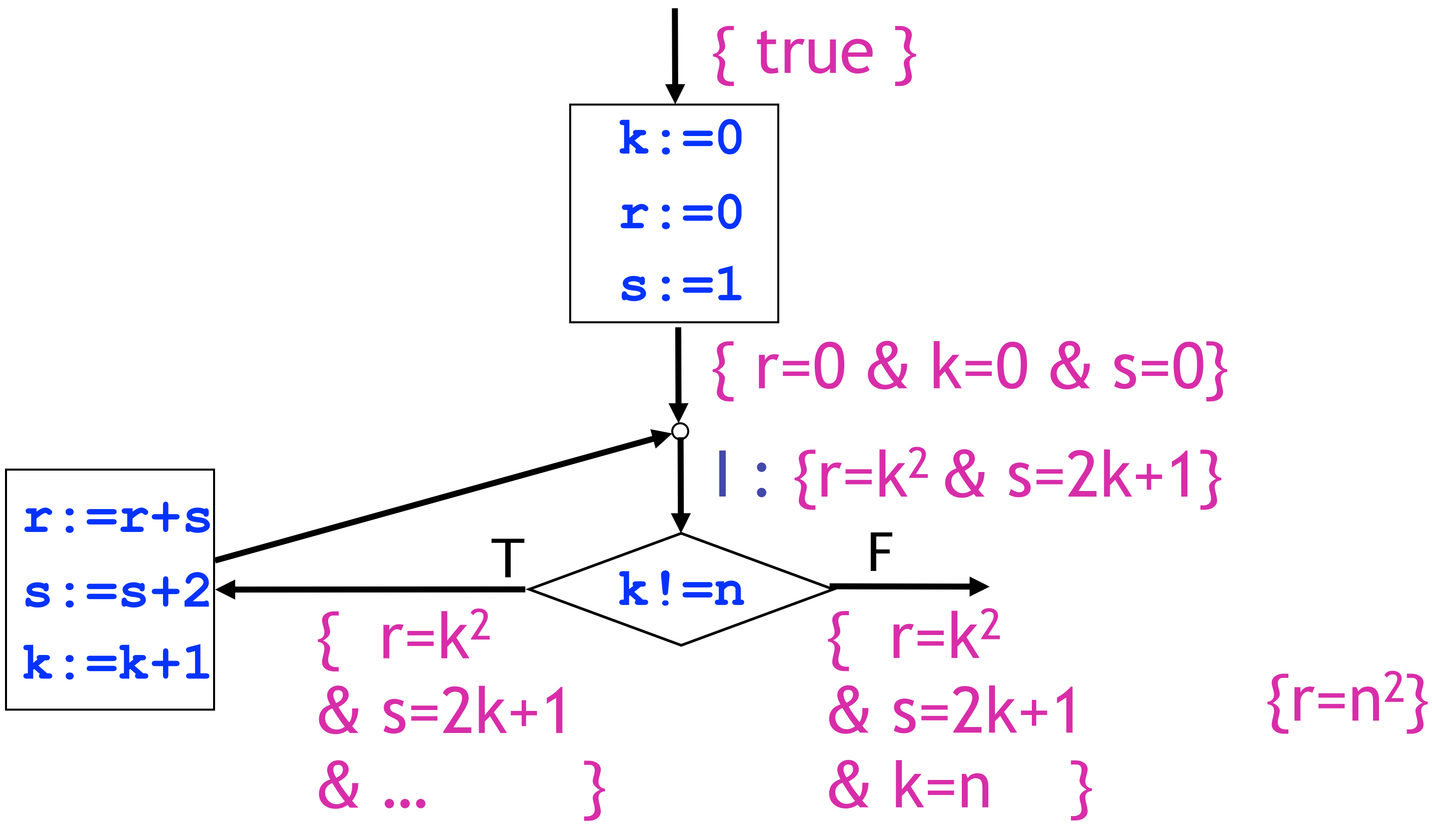
Need: {$r=k^2$ & $s=2k+1$ & …} **c** {$r=k^2$ & $s=2k+1$}



Verification Example

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

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

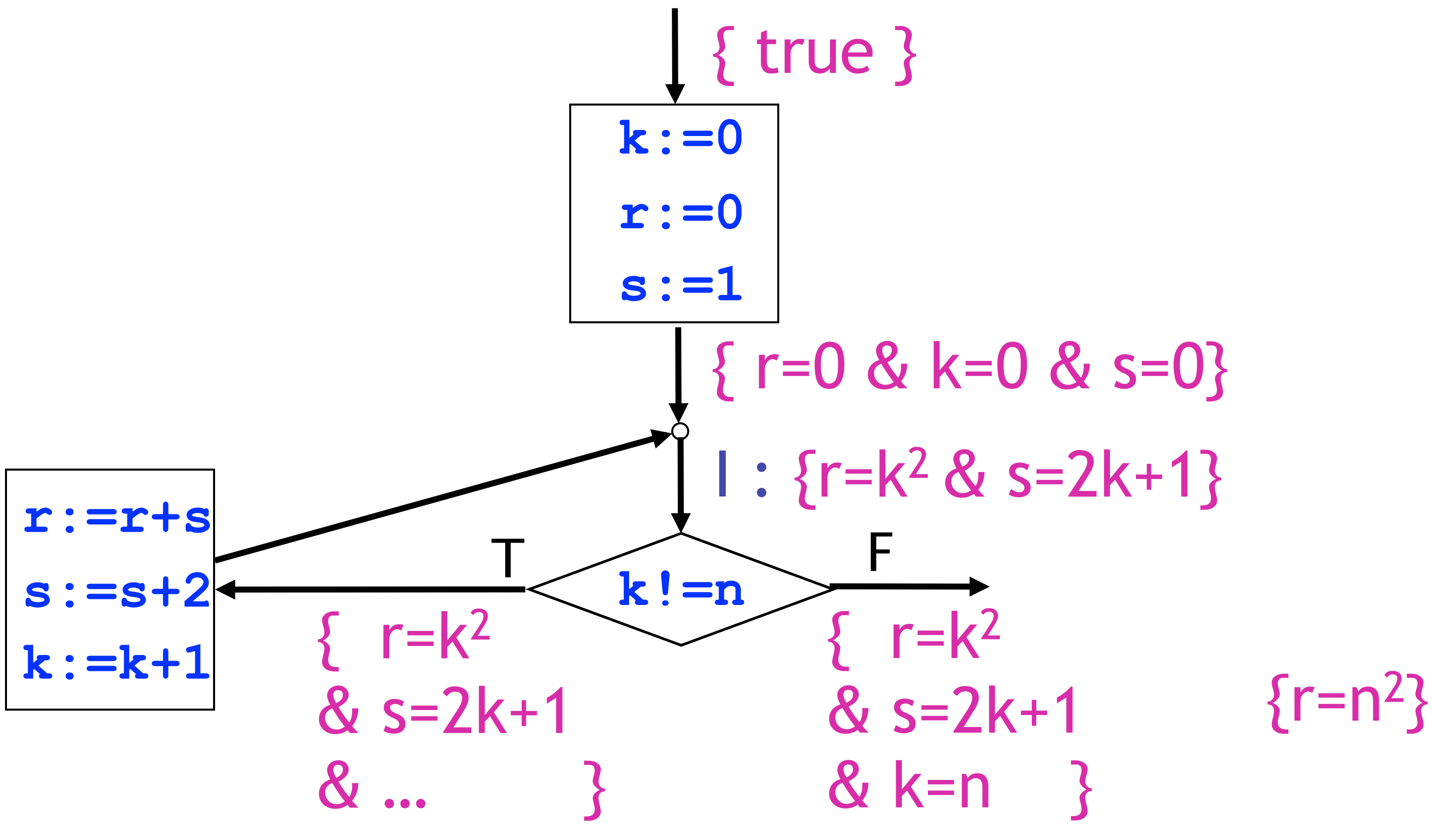


Verification Example

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 $(s+2) = 2(k+1)+1\}$

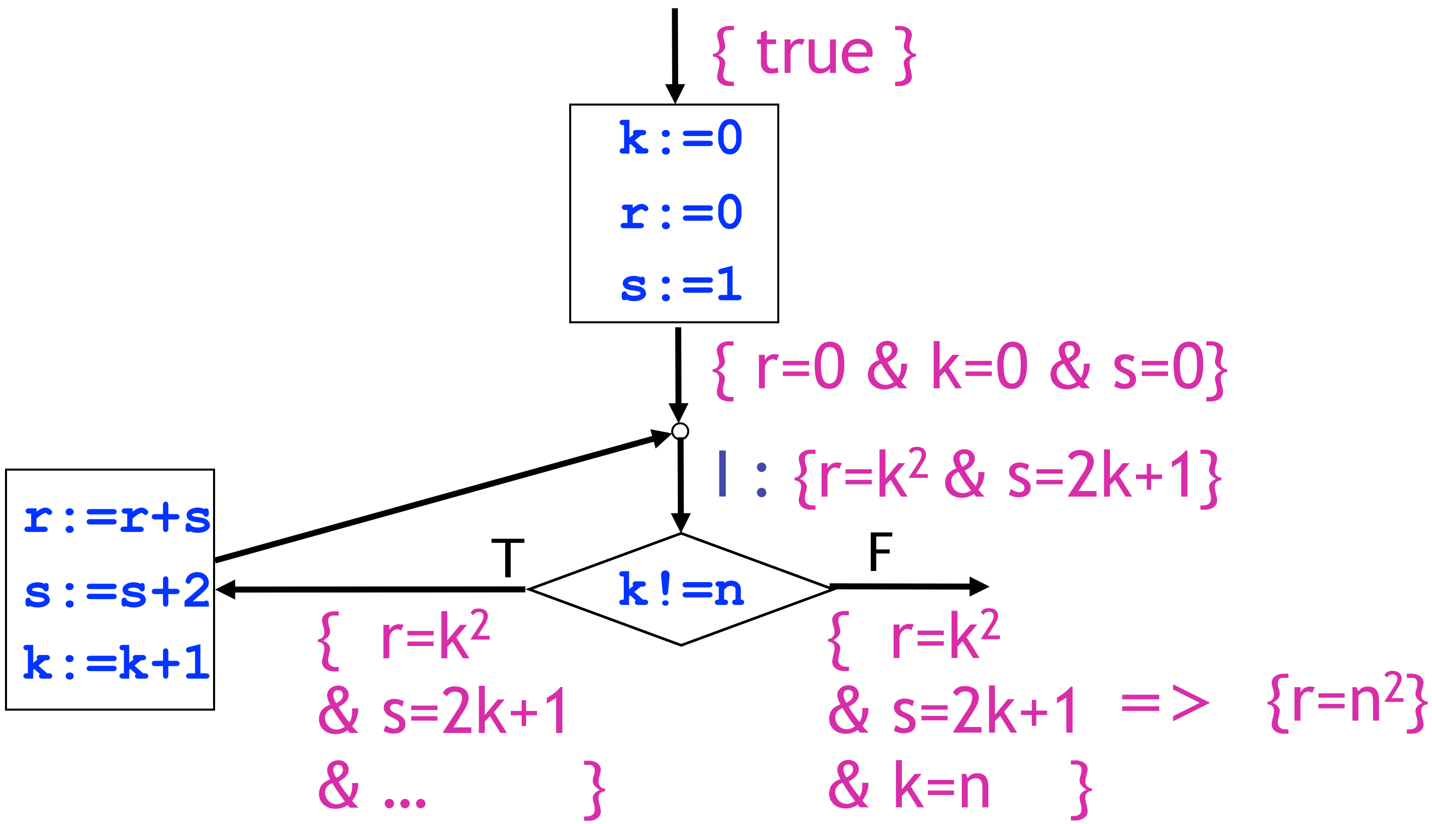


Verification Example

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 $(s+2) = 2(k+1)+1\}$

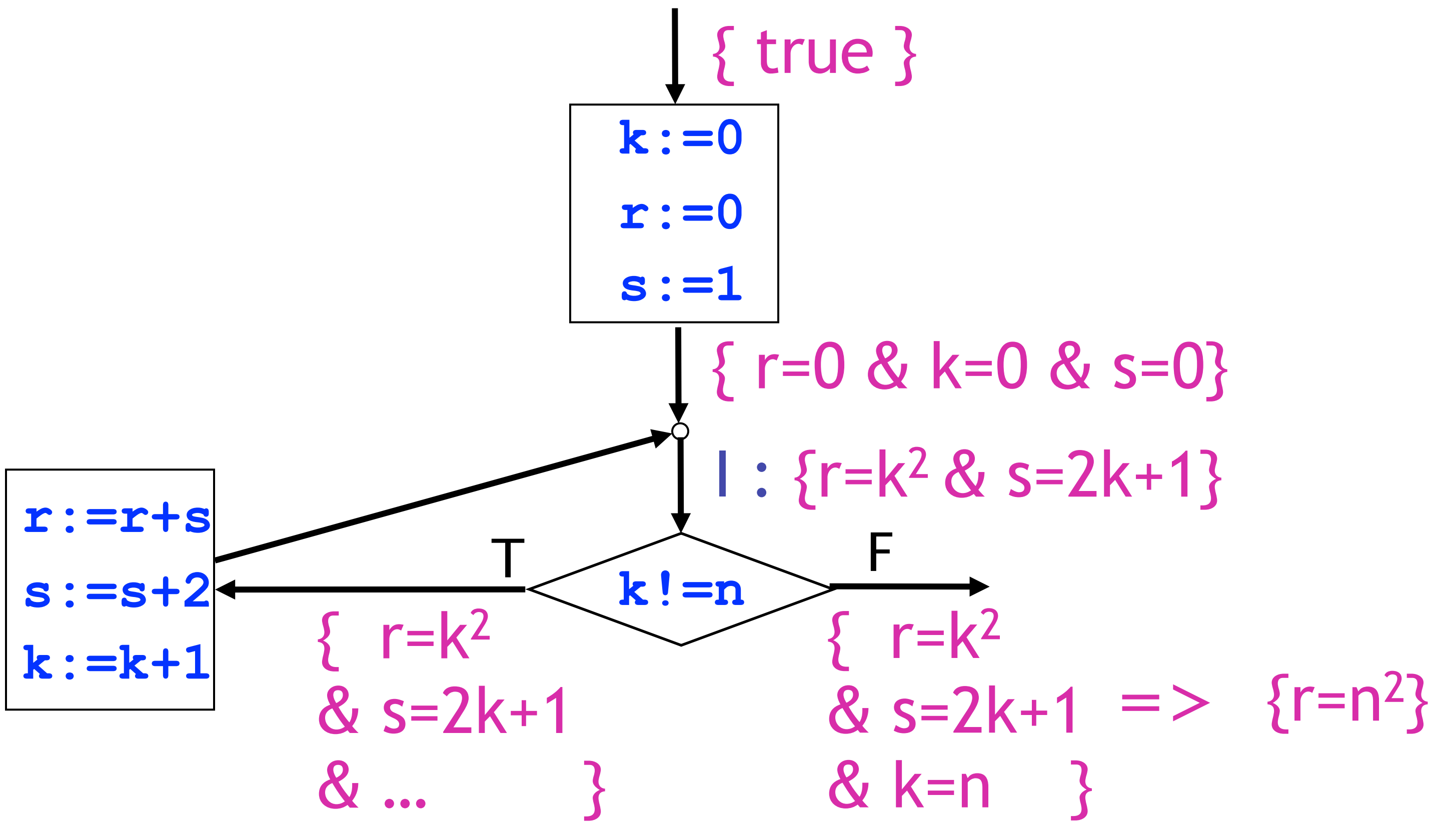


Verification Example

Need: {r=k² & s=2k+1 & …} c {r=k² & s=2k+1}

i.e. {r=k² & s=2k+1 …} => WP(c, {r=k² & s=2k+1})

i.e. {r=k² & 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

Functions are big instructions

Suppose we have verified `bsearch`

Postcondition
"Ensures"

Functions are big instructions

Suppose we have verified `bsearch`

Precondition
“Requires”

Postcondition
“Ensures”

Functions are big instructions

Suppose we have verified **`bsearch`**

```
int bsearch(int a[], int p) {
```

{ sorted(a) }

Precondition
"Requires"

Postcondition
"Ensures"

Functions are big instructions

Suppose we have verified **`bsearch`**

```
int bsearch(int a[], int p) {
    { sorted(a) }
    …
```

Precondition
"Requires"

Postcondition
"Ensures"

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)}
```

Precondition
"Requires"

Postcondition
"Ensures"

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"

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”

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"

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

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:

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 $P \Rightarrow Pre[E/x]$

$\boxed{\texttt{y:=f(e)}}$

$\{ Q \}$ and $Post[E/x,y/r] \Rightarrow Q$

Function Call: Example

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

Function Call: Example

Consider the call

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

Function Call: Example

Consider the call

{sorted(arr) }

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

Function Call: Example

Consider the call

{sorted(arr) }

```
y:=bsearch(arr,5)
```

{y=-1 || arr[y]=5}

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

Function Call: Example

Consider the call

{sorted(arr) }

```
y:=bsearch(arr,5)
```

{y=-1 || arr[y]=5}

```
if(y!=-1){
```

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

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}

Function Call: Example

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}

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

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}
```

Function Call: Example

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}

```
int bsearch(int a[],int p) {
```

{ sorted(a) }

…

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

```
  return r;
}
```

- sorted[array] => Pre[a := arr]

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)`

$\{y=-1 \mid\mid arr[y]=5\}$

`if(y!=-1){`

$\{y!=-1 \ \& \ (y=-1 \mid\mid arr[y]=5\}$

$\{arr[y]=5\}$

- 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

Assignment and Aliasing

Does assignment rule work with aliasing ?

Assignment and Aliasing

Does assignment rule work with aliasing ?

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

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

Hoare Rules: Assignment and References

- When is the following Hoare triple valid?

  { A } `*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

Hoare Rules: Assignment and References

Modeling writes with memory expressions

- Treat memory as a **whole** with memory variables (M)

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

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

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

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

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

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

Memory Aliasing

Memory Aliasing

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

Memory Aliasing

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

Memory Aliasing

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

  A = [upd(M, x, 5)/M] (*x+*y=10)

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)

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

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

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

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)

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

Algorithmic Program Verification

…or how does ESC/Java work ?

Algorithmic Program Verification

…or how does ESC/Java work ?

Q: How to algorithmically prove {P} `c` {Q} ?

If no loops:

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)

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)

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

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

VC Generation for Loops

Suppose all loops annotated with Invariant

$\texttt{while}_{I}\ \texttt{b do c}$

VC Generation for Loops

Suppose all loops annotated with Invariant

$$\texttt{while}_{I}\ \texttt{b do c}$$

VC Generation for Loops

Suppose all loops annotated with Invariant

$\texttt{while}_{I}\ \texttt{b do c}$

Compute VC:

VC Generation for Loops

Suppose all loops annotated with Invariant

`while`$_I$ `b do c`

Compute VC:

SMTValid(VC) **implies** |- {P} `c` {Q}

VC Generation for Loops

Suppose all loops annotated with Invariant

$$\texttt{while}_{I}\ \texttt{b do c}$$

Compute VC:

SMTValid(VC) **implies** |- {P} `c` {Q}

VC Generation for Loops

Suppose all loops annotated with Invariant

$\texttt{while}_I\ \texttt{b do c}$

Compute VC:

SMTValid(VC) **implies** |- {P} `c` {Q}

Q: Why not iff ?

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…

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