correctness proof via forward derivation

- given a Hoare triple $\{\phi\}\ C\ \{\psi\}$,

correctness proof via forward derivation

- given a Hoare triple $\{\phi\}\ C\ \{\psi\}$,
- construct a *forwards* derivation

correctness proof via forward derivation

- given a Hoare triple $\{\phi\}$ C $\{\psi\}$,
- construct a *forwards* derivation
- derivation $=$ sequence of Hoare triples,
  each Hoare triple is an axiom (skip, update)
  or it is inferred by one of the inference rules (seq, cond, while)

correctness proof via forward derivation

- given a Hoare triple $\{\phi\}$ C $\{\psi\}$,
- construct a *forwards* derivation
- derivation $=$ sequence of Hoare triples,
  each Hoare triple is an axiom (skip, update)
  or it is inferred by one of the inference rules (seq, cond, while)
- Hoare triples with ψ and *strongest postcondition*
  for larger and larger program fragments

correctness proof via forward derivation

- given a Hoare triple $\{\phi\}$ C $\{\psi\}$,
- construct a *forwards* derivation
- derivation $=$ sequence of Hoare triples,
  each Hoare triple is an axiom (skip, update)
  or it is inferred by one of the inference rules (seq, cond, while)
- Hoare triples with ψ and *strongest postcondition*
  for larger and larger program fragments
- verification condition:
  strongest postcondition of ϕ under C entails ψ
  ($+$ special treatment of while)

strongest postcondition post(C, ψ)

- post(**skip**$, \phi) \equiv$

strongest postcondition post(C, ψ)

- post(**skip**, ϕ) $\equiv$ ϕ
- post($x := e, \phi$) $\equiv$

strongest postcondition $\mathrm{post}(C, \psi)$

- $\mathrm{post}(\textbf{skip}, \phi) \;\equiv\; \phi$
- $\mathrm{post}(x := e, \phi) \;\equiv\; \phi[x_{old}/x] \wedge x = e[x_{old}/x]$
- $\mathrm{post}(C_1 \; ; \; C_2, \phi) \;\equiv$

strongest postcondition post(C, ψ)

- post(**skip**, ϕ) $\equiv$ ϕ
- post($x := e, \phi$) $\equiv$ $\phi[x_{old}/x] \wedge x = e[x_{old}/x]$
- post(C_1 ; C_2, ϕ) $\equiv$ post(C_2, post(C_1, ϕ))
- post(**if** b **then** C_1 **else** C_2, ϕ) $\equiv$

strongest postcondition $\mathrm{post}(C, \psi)$

- $\mathrm{post}(\textbf{skip}, \phi) \;\equiv\; \phi$
- $\mathrm{post}(x := e, \phi) \;\equiv\; \phi[x_{old}/x] \wedge x = e[x_{old}/x]$
- $\mathrm{post}(C_1 \;;\; C_2, \phi) \;\equiv\; \mathrm{post}(C_2, \mathrm{post}(C_1, \phi))$
- $\mathrm{post}(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2, \phi) \;\equiv$
  $\mathrm{post}(C_1, b \wedge \phi) \vee \mathrm{post}(C_2, \neg b \wedge \phi)$
- $\mathrm{post}(\textbf{while } b \textbf{ do } \{\theta\}\; C_0, \phi) \;\equiv$

strongest postcondition $\text{post}(C, \psi)$

- $\text{post}(\textbf{skip}, \phi) \;\equiv\; \phi$
- $\text{post}(x := e, \phi) \;\equiv\; \phi[x_{old}/x] \wedge x = e[x_{old}/x]$
- $\text{post}(C_1 \;;\; C_2, \phi) \;\equiv\; \text{post}(C_2, \text{post}(C_1, \phi))$
- $\text{post}(\textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2, \phi) \;\equiv$
  $\text{post}(C_1, b \wedge \phi) \vee \text{post}(C_2, \neg b \wedge \phi)$
- $\text{post}(\textbf{while } b \textbf{ do } \{\theta\}\; C_0, \phi) \;\equiv\; \theta \wedge \neg b$

- next:
  static analysis constructs candidate for θ via forward analysis
  “reachability analysis”

program code for specifications

validity of Hoare triple:

```
{y >= z}
while (x < y) {
  x++;
}
{x >= z}
```

≡ safety of program:

```
assume(y >= z);
while (x < y) {
  x++;
}
assert(x >= z);
```

program with **assume** () and **assert** ()

- **assume** (e) $\equiv$ **if** e **then skip else halt**

program with **assume** () and **assert** ()

- **assume** (e) $\equiv$ **if** e **then skip else halt**
- **assert** (e) $\equiv$ **if** e **then skip else error**

program with **assume** () and **assert** ()

- **assume** $(e) \equiv$ **if** e **then skip else halt**
- **assert** $(e) \equiv$ **if** e **then skip else error**
- generalize *partial correctness*:

program with **assume** () and **assert** ()

- **assume** $(e) \;\equiv$ **if** e **then skip else halt**
- **assert** $(e) \;\equiv$ **if** e **then skip else error**
- generalize *partial correctness*:
  correctness of program wrt. Hoare triple:

$$\{\phi\}\ C\ \{\psi\}$$

$\equiv$

program with **assume** () and **assert** ()

- **assume** (e) $\equiv$ **if** e **then skip else halt**
- **assert** (e) $\equiv$ **if** e **then skip else error**
- generalize *partial correctness*:
  correctness of program wrt. Hoare triple:

  $$\{\phi\}\ C\ \{\psi\}$$

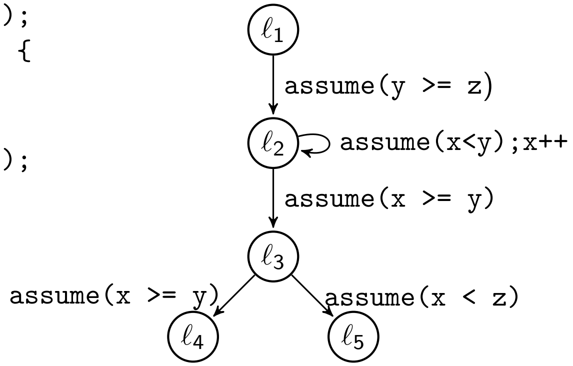
  $\equiv$ *safety* of program: **assume** (ϕ) ; C ; **assert** (ψ)
- safety $=$ non-reachability of **error**
  (no execution of **error** branch)

control flow graph

source code

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```

control flow graph

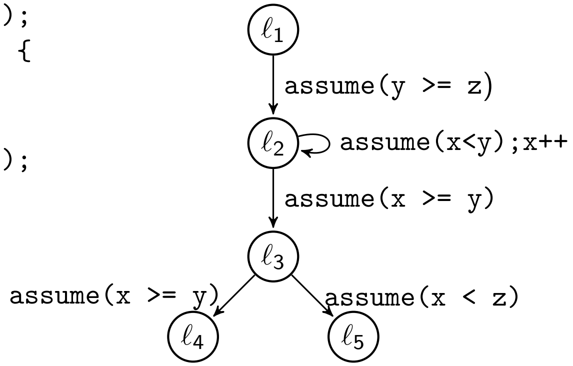


control flow graph

source code

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```

control flow graph



encode transition as logical formula

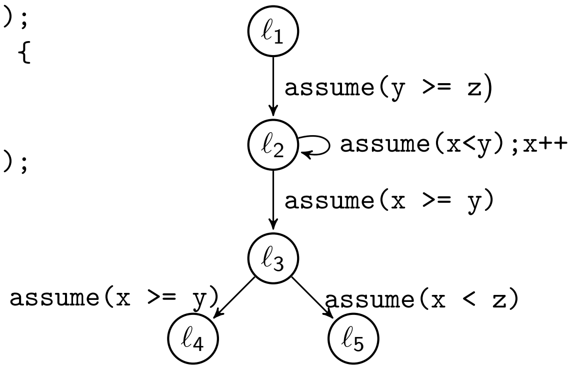
`assume( y >= z)` $\rightsquigarrow$

control flow graph

source code

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```

control flow graph



encode transition as logical formula

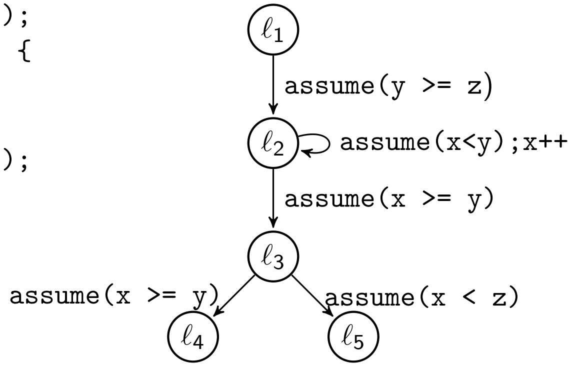
`assume( y >= z)` $\rightsquigarrow$ $y >= z$

control flow graph

source code

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```

control flow graph



encode transition as logical formula

`assume( y >= z)` $\rightsquigarrow$ $y >= z$

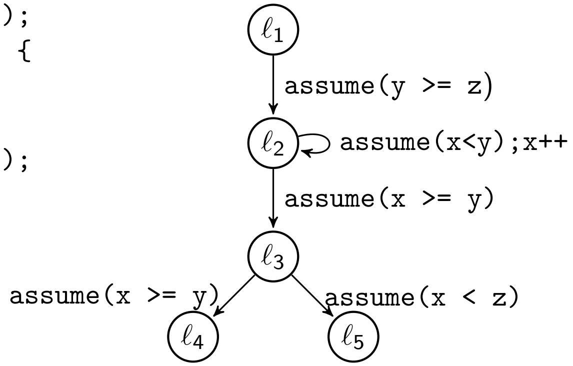
`x++` $\rightsquigarrow$

control flow graph

source code

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```

control flow graph



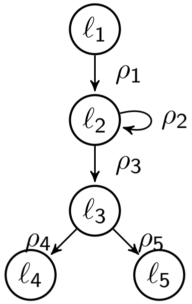
encode transition as logical formula

`assume( y >= z)` $\rightsquigarrow$ $y >= z$

`x++` $\rightsquigarrow$ $x' = x + 1$

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```

.



$$\begin{aligned}
\rho_1 &= (\mathit{move}(\ell_1, \ell_2) \land y \geq z \land \mathit{skip}(x, y, z))\\
\rho_2 &= (\mathit{move}(\ell_2, \ell_2) \land x + 1 \leq y \land x' = x + 1 \land \mathit{skip}(y, z))\\
\rho_3 &= (\mathit{move}(\ell_2, \ell_3) \land x \geq y \land \mathit{skip}(x, y, z))\\
\rho_4 &= (\mathit{move}(\ell_3, \ell_4) \land x \geq z \land \mathit{skip}(x, y, z))\\
\rho_5 &= (\mathit{move}(\ell_3, \ell_5) \land x + 1 \leq z \land \mathit{skip}(x, y, z))
\end{aligned}$$

transition relation ρ expressed by logica formula

$$\begin{aligned}
\rho_1 &\equiv (move(\ell_1, \ell_2) \wedge y \geq z \wedge skip(x, y, z)) \\
\rho_2 &\equiv (move(\ell_2, \ell_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge skip(y, z)) \\
\rho_3 &\equiv (move(\ell_2, \ell_3) \wedge x \geq y \wedge skip(x, y, z)) \\
\rho_4 &\equiv (move(\ell_3, \ell_4) \wedge x \geq z \wedge skip(x, y, z)) \\
\rho_5 &\equiv (move(\ell_3, \ell_5) \wedge x + 1 \leq z \wedge skip(x, y, z))
\end{aligned}$$

abbreviations:

$$\begin{aligned}
move(\ell, \ell') &\equiv (pc = \ell \wedge pc' = \ell') \\
skip(v_1, \ldots, v_n) &\equiv (v_1' = v_1 \wedge \ldots \wedge v_n' = v_n)
\end{aligned}$$

program $\mathbf{P} = (V, pc, \varphi_{init}, \mathcal{R}, \varphi_{err})$

- V - finite tuple of *program variables*
- pc - *program counter variable* (pc included in V)
- φ_{init} - *initiation condition* given by formula over V
- $\mathcal{R}$ - a finite set of *transition relations*
- φ_{err} - an *error condition* given by a formula over V

- transition relation $\rho \in \mathcal{R}$ given by
  formula over the variables V and their primed versions V'

states, sets, and relations

- each program variable is assigned a *domain* of values

states, sets, and relations

- each program variable is assigned a *domain* of values
- *program state* = function that assigns each program variable a value from its respective domain

states, sets, and relations

- each program variable is assigned a *domain* of values
- *program state* = function that assigns each program variable a value from its respective domain
- Σ = set of program states

states, sets, and relations

- each program variable is assigned a *domain* of values
- *program state* = function that assigns each program variable a value from its respective domain
- Σ = set of program states
- formula with free variables in V = set of program states

states, sets, and relations

- each program variable is assigned a *domain* of values
- *program state* = function that assigns each program variable a value from its respective domain
- Σ = set of program states
- formula with free variables in V = set of program states
- formula with free variables in V and V' = binary relation over program states
    - first component of each pair assigns values to V
    - second component of the pair assigns values to V'

states, sets, and relations

- each program variable is assigned a *domain* of values
- *program state* = function that assigns each program variable a value from its respective domain
- Σ = set of program states
- formula with free variables in V = set of program states
- formula with free variables in V and V' = binary relation over program states
  - first component of each pair assigns values to V
  - second component of the pair assigns values to V'
- identify formulas with sets and relations that they represent

states, sets, and relations

- each program variable is assigned a *domain* of values
- *program state* = function that assigns each program variable a value from its respective domain
- Σ = set of program states
- formula with free variables in V = set of program states
- formula with free variables in V and V' = binary relation over program states
  - first component of each pair assigns values to V
  - second component of the pair assigns values to V'
- identify formulas with sets and relations that they represent
- identify the logical consequence relation between formulas $\models$ with set inclusion $\subseteq$

states, sets, and relations

- each program variable is assigned a *domain* of values
- *program state* = function that assigns each program variable a value from its respective domain
- Σ = set of program states
- formula with free variables in V = set of program states
- formula with free variables in V and V' = binary relation over program states
  - first component of each pair assigns values to V
  - second component of the pair assigns values to V'
- identify formulas with sets and relations that they represent
- identify the logical consequence relation between formulas $\models$ with set inclusion $\subseteq$
- identify the satisfaction relation $\models$ between valuations and formulas, with the membership relation $\in$

example: states, sets, and relations

- formula $y \geq z$ = set of program states in which the value of the variable y is greater than the value of z

example: states, sets, and relations

- formula $y \geq z$ = set of program states in which the value of the variable y is greater than the value of z
- formula $y' \geq z$ = binary relation over program states, = set of pairs of program states (s_1, s_2) in which the value of the variable y in the second state s_2 is greater than the value of z in the first state s_1

example: states, sets, and relations

- formula $y \geq z$ = set of program states in which the value of the variable y is greater than the value of z
- formula $y' \geq z$ = binary relation over program states, = set of pairs of program states (s_1, s_2) in which the value of the variable y in the second state s_2 is greater than the value of z in the first state s_1
- if program state s assigns 1, 3, 2, and ℓ_1
  to program variables x, y, z, and pc, respectively,
  then $s \models y \geq z$

example: states, sets, and relations

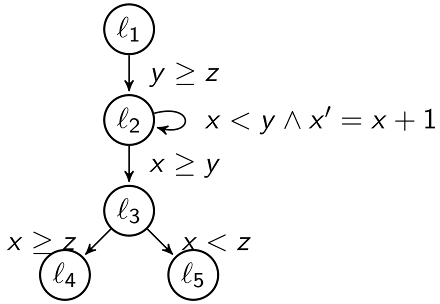
- formula $y \geq z$ = set of program states in which the value of the variable y is greater than the value of z
- formula $y' \geq z$ = binary relation over program states, = set of pairs of program states (s_1, s_2) in which the value of the variable y in the second state s_2 is greater than the value of z in the first state s_1
- if program state s assigns 1, 3, 2, and ℓ_1 to program variables x, y, z, and pc, respectively, then $s \models y \geq z$
- logical consequence: $y \geq z \models y + 1 \geq z$

example program $\mathbf{P} = (V, pc, \varphi_{init}, \mathcal{R}, \varphi_{err})$

- program variables $V = (pc, x, y, z)$
- program counter pc
- program variables x, y, and z range over integers
- set of control locations $\mathcal{L} = \{\ell_1, \ldots \ell_5\}$
- initiation condition $\varphi_{init} = (pc = pc = \ell_1)$
- error condition $\varphi_{err} = (pc = pc = \ell_5)$
- program transitions $\mathcal{R} = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$

$$\begin{aligned}
\rho_1 &= (move(\ell_1, \ell_2) \wedge y \geq z \wedge skip(x, y, z)) \\
\rho_2 &= (move(\ell_2, \ell_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge skip(y, z)) \\
\rho_3 &= (move(\ell_2, \ell_3) \wedge x \geq y \wedge skip(x, y, z)) \\
\rho_4 &= (move(\ell_3, \ell_4) \wedge x \geq z \wedge skip(x, y, z)) \\
\rho_5 &= (move(\ell_3, \ell_5) \wedge x + 1 \leq z \wedge skip(x, y, z))
\end{aligned}$$

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```



.

$$
\begin{aligned}
\rho_1 &= (move(\ell_1, \ell_2) \wedge y \geq z \wedge skip(x, y, z))\\
\rho_2 &= (move(\ell_2, \ell_2) \wedge x + 1 \leq y \wedge x' = x + 1 \wedge skip(y, z))\\
\rho_3 &= (move(\ell_2, \ell_3) \wedge x \geq y \wedge skip(x, y, z))\\
\rho_4 &= (move(\ell_3, \ell_4) \wedge x \geq z \wedge skip(x, y, z))\\
\rho_5 &= (move(\ell_3, \ell_5) \wedge x + 1 \leq z \wedge skip(x, y, z))
\end{aligned}
$$

initial state, error state, transition relation $\mathcal{R}$

- each state that satisfies the initiation condition φ_{init} is called an *initial* state
- each state that satisfies the error condition φ_{err} is called an *error* state
- program transition relation $\rho_{\mathcal{R}}$ is the union of the "single-statement" transition relations, i.e.,

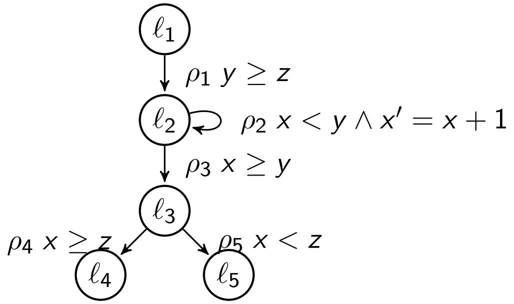
$$\rho_{\mathcal{R}} = \bigvee_{\rho \in \mathcal{R}} \rho \ .$$

- the state s has a transition to the state s' if the pair of states (s, s') lies in the program transition relation $\rho_{\mathcal{R}}$, i.e., if $(s, s') \models \rho_{\mathcal{R}}$

program computation $s_1, s_2, \ldots$

- the first element is an initial state, i.e., $s_1 \models \varphi_{init}$
- each pair of consecutive states (s_i, s_{i+1}) is connected by a program transition, i.e., $(s_i, s_{i+1}) \models \rho_{\mathcal{R}}$
- if the sequence is finite
  then the last element does not have any successors
  i.e., if the last element is s_n,
  then there is no state s such that $(s_n, s) \models \rho_{\mathcal{R}}$

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```



example of a computation:

$(\ell_1, 1, 3, 2),\ (\ell_2, 1, 3, 2),\ (\ell_2, 2, 3, 2),\ (\ell_2, 3, 3, 2),\ (\ell_3, 3, 3, 2),\ (\ell_4, 3, 3, 2)$

- sequence of transitions $\rho_1, \rho_2, \rho_2, \rho_3, \rho_4$
- state $=$ tuple of values of program variables pc, x, y, and z
- last program state does not any successors

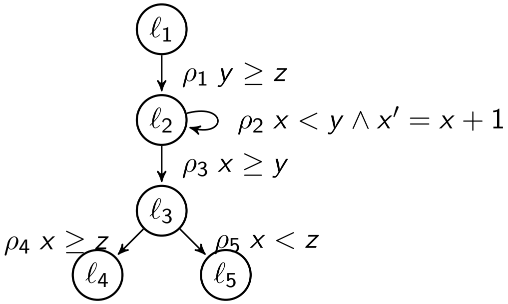
Correctness: Safety

- a state is *reachable* if it occurs in some program computation
- a program is *safe* if no error state is reachable
- . . . if and only if no error state lies in φ_{reach},

$$\varphi_{err} \wedge \varphi_{reach} \models \mathit{false} \ .$$

where φ_{reach} = set of reachable program states

```
1:  assume(y >= z);
2:  while (x < y) {
      x++;
    }
3:  assert(x >= z);
4: exit
5: error
```



set of reachable states:

$$\begin{aligned}\varphi_{reach} = (&pc = \ell_1 \vee \\ &pc = \ell_2 \wedge y \geq z \vee \\ &pc = \ell_3 \wedge y \geq z \wedge x \geq y \vee \\ &pc = \ell_4 \wedge y \geq z \wedge x \geq y)\end{aligned}$$

post operator

- let φ be a formula over V
- let ρ be a formula over V and V'
- define a *post-condition* function *post* by:

$$post(\varphi, \rho) \;=\; \exists V'' : \varphi[V''/V] \wedge \rho[V''/V][V/V']$$

  an application $post(\varphi, \rho)$ computes the image of the set φ under the relation ρ
- post distributes over disjunction wrt. each argument:

$$\begin{aligned} post(\varphi, \rho_1 \vee \rho_2) = (post(\varphi, \rho_1) \vee post(\varphi, \rho_2)) \\ post(\varphi_1 \vee \varphi_2, \rho) = (post(\varphi_1, \rho) \vee post(\varphi_2, \rho)) \end{aligned}$$