Rigorous Software Development
CSCI-GA 3033-009

Instructor: Thomas Wies

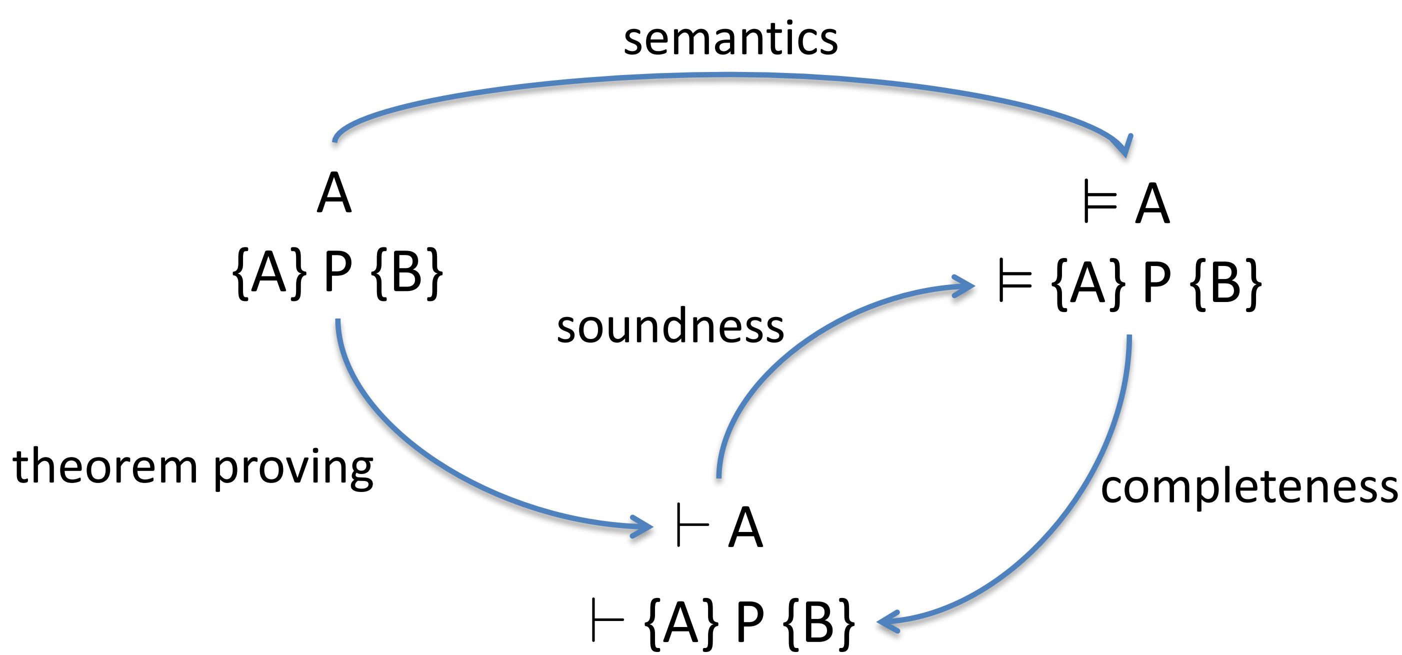
Spring 2013

Lecture 13

Invariant Generation

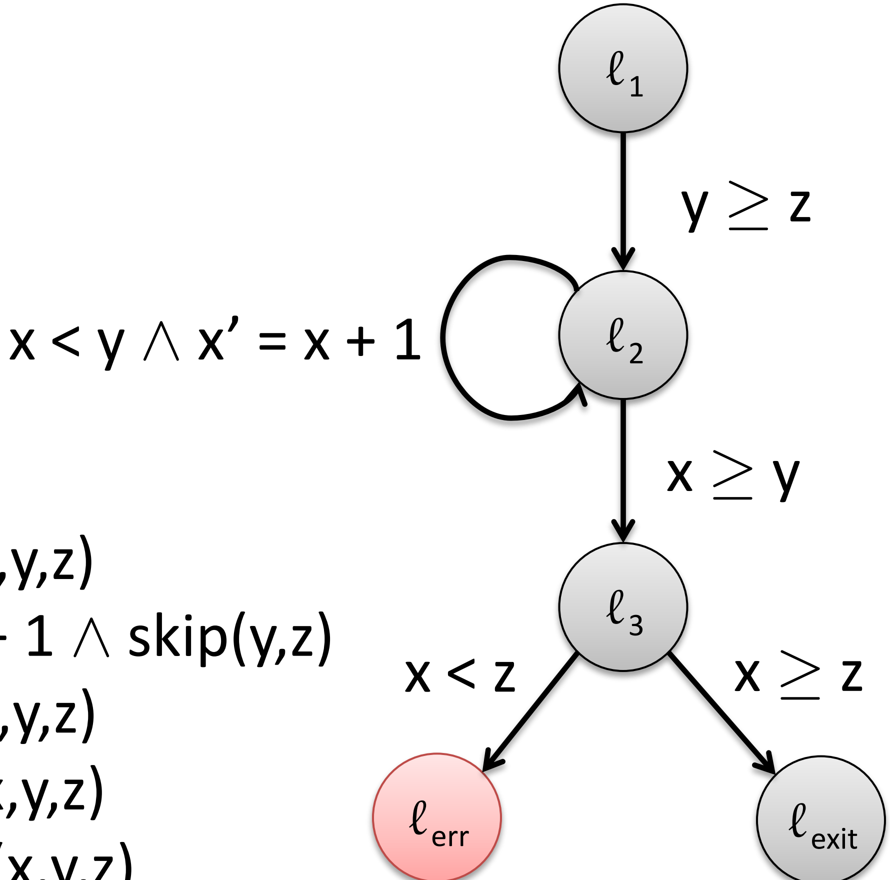
- Tools such as Dafny enable automated program verification by
  - automatically generating verification conditions and
  - automatically checking validity of the generated VCs.
- The user still needs to provide the invariants.
  - This is often the hardest part.
- Can we generate invariants automatically?

Axiomatic vs. Operational Semantics



Programs as Systems of Constraints

```
1:  assume y ≥ z;
2:  while x < y do
        x := x + 1;
3:  assert x ≥ z
```



$\rho_1 = \text{move}(\ell_1, \ell_2) \wedge y \geq z \wedge \text{skip}(x,y,z)$

$\rho_2 = \text{move}(\ell_2, \ell_2) \wedge x < y \wedge x' = x + 1 \wedge \text{skip}(y,z)$

$\rho_3 = \text{move}(\ell_2, \ell_3) \wedge x \geq y \wedge \text{skip}(x,y,z)$

$\rho_4 = \text{move}(\ell_3, \ell_{err}) \wedge x < z \wedge \text{skip}(x,y,z)$

$\rho_5 = \text{move}(\ell_3, \ell_{exit}) \wedge x \geq z \wedge \text{skip}(x,y,z)$

$\text{move}(\ell_1, \ell_2) = pc = \ell_1 \wedge pc' = \ell_2$

$\text{skip}(x_1, \ldots, x_n) = x_1' = x_1 \wedge \ldots \wedge x_n' = x_n$

Program $P = (V, init, R, error)$

- V : finite set of program variables
- *init* : initiation condition given by a formula over V
- R : a finite set of transition constraints
  - transition constraint $\rho \in R$ given by a formula over V and their primed versions V'
  - we often think of R as disjunction of its elements

$$R = \rho_1 \vee \dots \vee \rho_n$$

- *error* : error condition given by a formula over V

Programs as Systems of Constraints

$P = (V, init, R, error)$

$V = \{pc, x, y, z\}$

$init = \; pc = \ell_1$

$R = \{\rho_1, \rho_2, \rho_3, \rho_4, \rho_5\}$ where

$\rho_1 = move(\ell_1, \ell_2) \wedge y \geq z \wedge skip(x,y,z)$

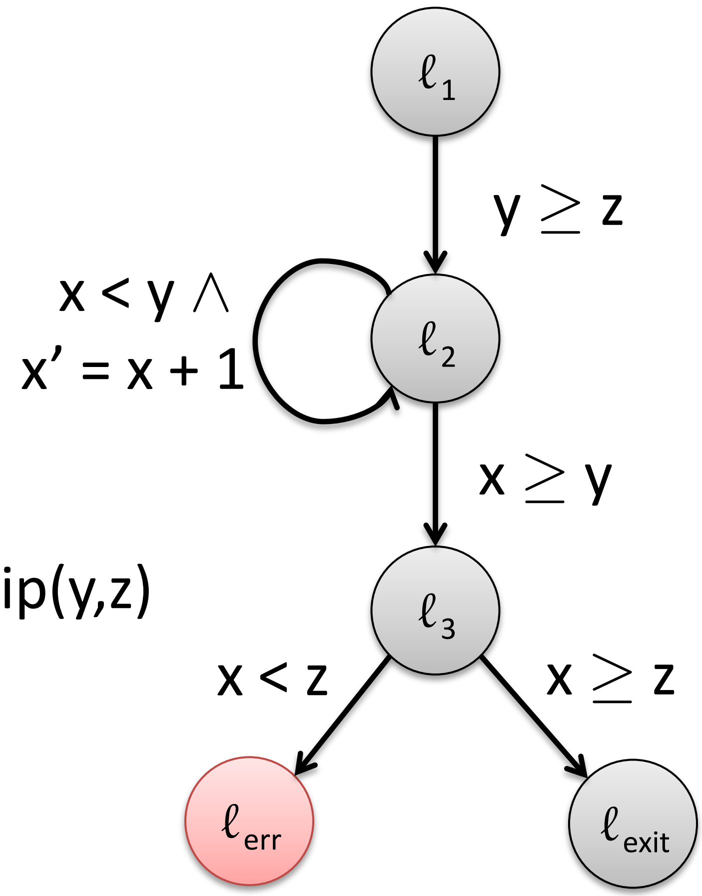
$\rho_2 = move(\ell_2, \ell_2) \wedge x < y \wedge x' = x + 1 \wedge skip(y,z)$

$\rho_3 = move(\ell_2, \ell_3) \wedge x \geq y \wedge skip(x,y,z)$

$\rho_4 = move(\ell_3, \ell_{err}) \wedge x < z \wedge skip(x,y,z)$

$\rho_5 = move(\ell_3, \ell_{exit}) \wedge x \geq z \wedge skip(x,y,z)$

$error = \; pc = \ell_{err}$



Programs as Transition Systems

- states Q are valuations of program variables V
- initial states Q_{init} are the states satisfying the initiation condition *init*

$$Q_{init} = \{q \in Q \mid q \models init\}$$

- transition relation $\rightarrow$ is the relation defined by the transition constraints in R

$$q_1 \rightarrow q_2 \quad \text{iff} \quad q_1, q_2' \models R$$

- error states Q_{err} are the states satisfying the error condition *error*

$$Q_{err} = \{q \in Q \mid q \models error\}$$

Partial Correctness of Programs

- a state q is reachable in P if it occurs in some computation of P

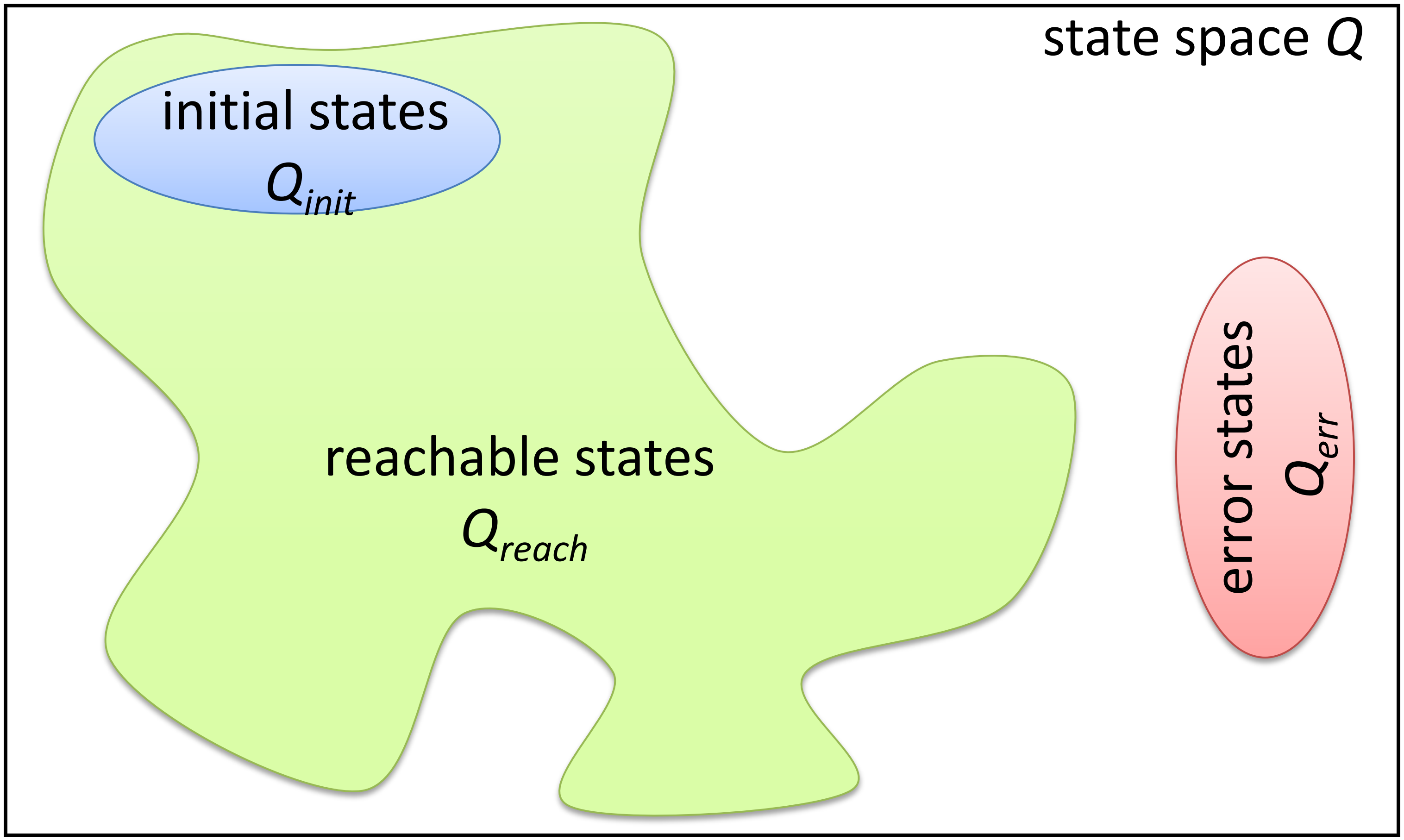
  $q_0 \rightarrow q_1 \rightarrow q_2 \rightarrow \ldots \rightarrow q$ where $q_0 \in Q_{init}$
- denote by Q_{reach} the set of all reachable states of P
- a program P is safe if no error state is reachable in P

  $Q_{reach} \cap Q_{err} = \emptyset$

  or, if Q_{reach} is expressed as a formula *reach* over V

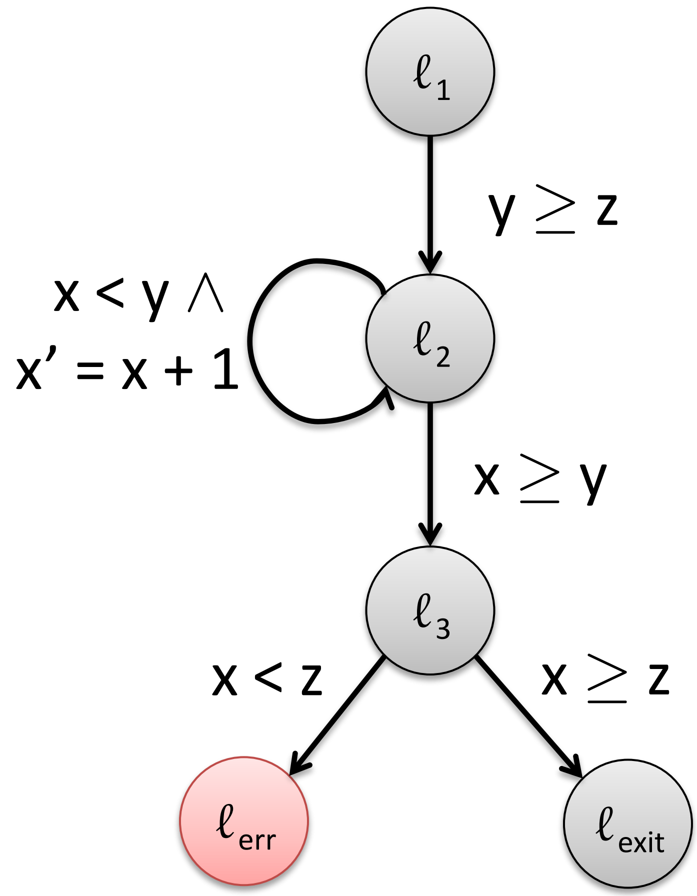
  $\models reach \wedge error \Rightarrow \text{false}$

Partial Correctness of Programs



Example: Reachable States of a Program

1: **assume** $y \geq z$;
2: **while** $x < y$ **do**
        $x := x + 1$;
3: **assert** $x \geq z$



Reachable states

$$
\begin{aligned}
\textit{reach} = \; & pc = \ell_1 \vee \\
& pc = \ell_2 \wedge y \geq z \vee \\
& pc = \ell_3 \wedge y \geq z \wedge x \geq y \vee \\
& pc = \ell_{exit} \wedge y \geq z \wedge x \geq y
\end{aligned}
$$

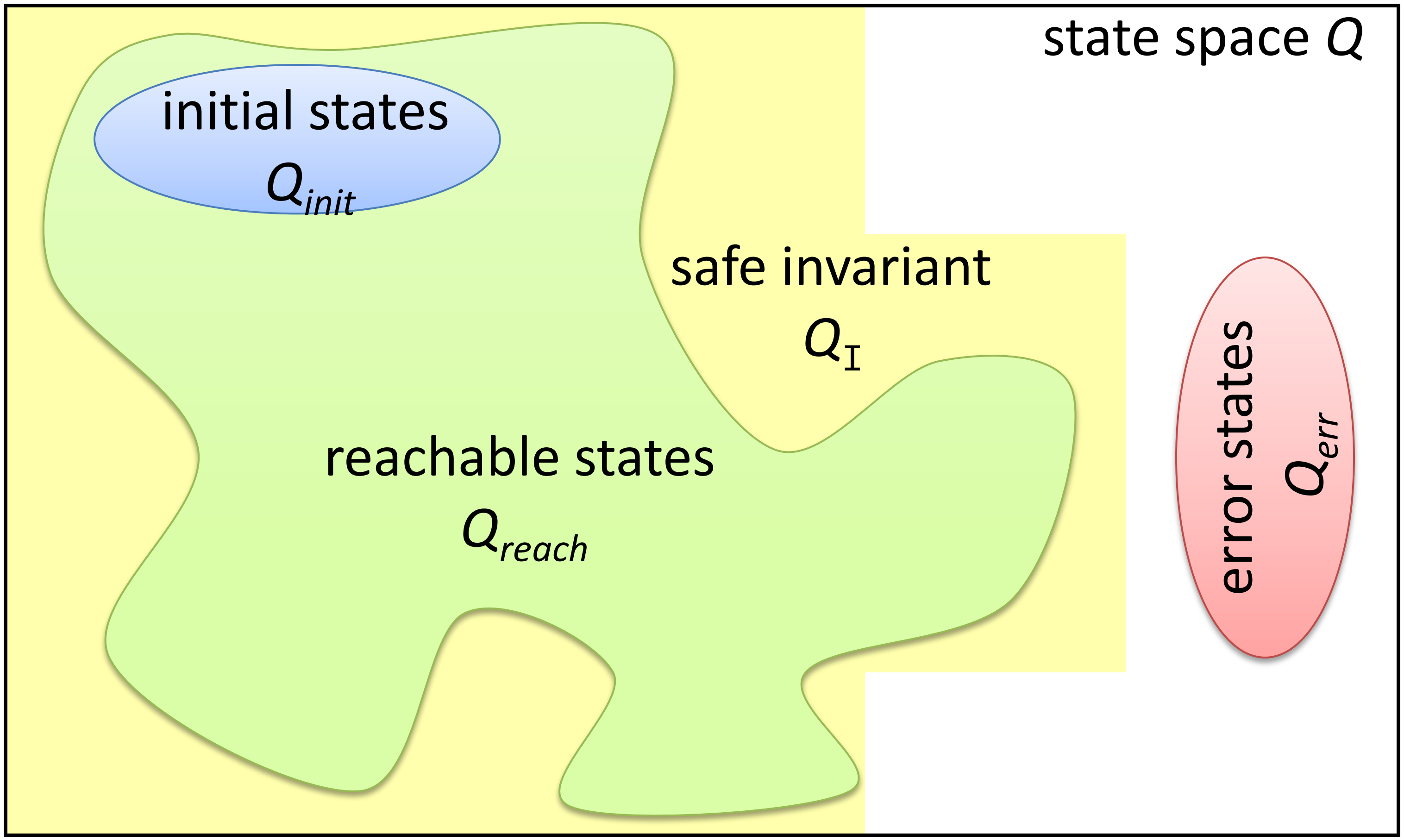
What is the connection with invariants?

Can we compute *reach*?

Invariants of Programs

- an invariant $Q_{\mathtt{I}}$ of a program *P* is a superset of its reachable states:
  $$Q_{reach} \subseteq Q_{\mathtt{I}}$$
- an invariant $Q_{\mathtt{I}}$ is safe if it does not contain any error states:
  $$Q_{\mathtt{I}} \wedge Q_{err} = \emptyset$$
  or if $Q_{\mathtt{I}}$ is expressed as a formula $\mathtt{I}$ over V
  $$\models \mathtt{I} \wedge error \Rightarrow \text{false}$$
- *reach* is the "smallest" invariant of *P.*
- In particular, if *P* is safe then so is *reach*.

Partial Correctness of Programs



Strongest Postconditions

- The strongest postcondition *post*(ρ,A) holds for any state *q* that is a ρ-successor state of some state satisfying A:

  $q' \models post(\rho, A) \quad \text{iff} \quad \exists q \in Q.\ q \models A \land q, q' \models \rho$

  or equivalently

  $post(\rho, A) = (\exists\, V.\ A \land \rho)\ [V/V']$
- Compute *reach* by applying *post* iteratively to *init*

Example: Application of *post*

- A = pc $= \ell_2 \wedge$ y $\geq$ z
- ρ = move(ℓ_2, ℓ_2) $\wedge$ x < y $\wedge$ x' $=$ x + 1 $\wedge$ skip(y,z)
- *post*(ρ, A)

= ($\exists$ *V*. A $\wedge$ ρ) [*V*/*V'*]

= ($\exists$ pc x y z. pc$=\ell_2 \wedge$ y$\geq$z $\wedge$ pc$=\ell_2 \wedge$ pc'$=\ell_2 \wedge$ x<y $\wedge$
x'$=$ x+1 $\wedge$ y'$=$y $\wedge$ z'$=$z) [pc/pc', x/x', y/y', z/z']

= (y' $\geq$ z' $\wedge$ pc'$=\ell_2 \wedge$ x' - 1 < y') [pc/pc', x/x', y/y', z/z']

= y $\geq$ z $\wedge$ pc$=\ell_2 \wedge$ x $\leq$ y

Iterating *post*

- $reach^i(\rho, A) = \begin{cases} A, \text{ if } i = 0 \\ post(post^{i\text{-}1}(\rho, A)) \text{ if } i > 0 \end{cases}$

- *reach* = *init* $\vee$ *post*(*R*, *init*) $\vee$ *post*(*R*, *post*(*R*, *init*)) $\vee$ ...
  = $\bigvee_{i \geq 0} post^i(R, init)$

- i'th disjunct of *reach* represents all states reachable from Q_{init} in i computation steps.

Finite iteration of *post* may suffice

- Fixed point is reached after n steps if

$$\models \bigvee_{i=0}^{n+1} post^i(R, init) \Rightarrow \bigvee_{i=0}^{n} post^i(R, init)$$

Example Iteration

ρ_1 = move(ℓ_1, ℓ_2) $\wedge$ y $\geq$ z $\wedge$ skip(x,y,z)
ρ_2 = move(ℓ_2, ℓ_2) $\wedge$ x < y $\wedge$ x' = x + 1 $\wedge$ skip(y,z)
ρ_3 = move(ℓ_2, ℓ_3) $\wedge$ x $\geq$ y $\wedge$ skip(x,y,z)
ρ_4 = move(ℓ_3, ℓ_{err}) $\wedge$ x < z $\wedge$ skip(x,y,z)
ρ_5 = move(ℓ_3, ℓ_{exit}) $\wedge$ x $\geq$ z $\wedge$ skip(x,y,z)

$post^0(R, init) = init = \text{pc}{=}\ell_1$

Example Iteration

ρ_1 = move(ℓ_1, ℓ_2) $\wedge$ y $\geq$ z $\wedge$ skip(x,y,z)
ρ_2 = move(ℓ_2, ℓ_2) $\wedge$ x < y $\wedge$ x' = x + 1 $\wedge$ skip(y,z)
ρ_3 = move(ℓ_2, ℓ_3) $\wedge$ x $\geq$ y $\wedge$ skip(x,y,z)
ρ_4 = move(ℓ_3, ℓ_{err}) $\wedge$ x < z $\wedge$ skip(x,y,z)
ρ_5 = move(ℓ_3, ℓ_{exit}) $\wedge$ x $\geq$ z $\wedge$ skip(x,y,z)

$post^2(R, init)$
= $post(\rho_{2,}\ post(R, init)) \vee post(\rho_{3,}\ post(R, init))$
= $pc = \ell_2 \wedge y \geq z \wedge x \leq y \ \vee\ pc = \ell_3 \wedge y \geq z \wedge x \geq y$

$post^3(R, init)$ =
$post(\rho_{2,}\ post^2(R, init)) \vee post(\rho_{3,}\ post^2(R, init)) \vee$
$post(\rho_{4,}\ post^2(R, init)) \vee post(\rho_{5,}\ post^2(R, init))$
= $pc = \ell_2 \wedge y \geq z \wedge x \leq y \ \vee\ pc = \ell_3 \wedge y \geq z \wedge x = y \ \vee$
$pc = \ell_{exit} \wedge y \geq z \wedge x \leq y \vee \text{false}$

Example Iteration

$post^3(R, init)$ =

= $pc = \ell_2 \wedge y \geq z \wedge x \leq y \ \vee pc = \ell_3 \wedge y \geq z \wedge x \geq y \vee$

$pc = \ell_{exit} \wedge y \geq z \wedge x \leq y$

$post^4(R, init) = post^3(R, init)$

Fixed point:

reach

$= post^0(R, init) \vee post^1(R, init) \vee post^2(R, init) \vee post^3(R, init)$

$= pc = \ell_1 \ \vee$

$pc = \ell_2 \wedge y \geq z \vee$

$pc = \ell_3 \wedge y \geq z \wedge x \geq y \ \vee$

$pc = \ell_{exit} \wedge y \geq z \wedge x \leq y$

Checking Safety

- An inductive invariant $\mathtt{I}$ contains the initial states and is closed under successors:

$$\models \mathit{init} \Rightarrow \mathtt{I} \quad \text{and} \quad \models \mathit{post}(R, \mathtt{I}) \Rightarrow \mathtt{I}$$

- A program is safe if there exists a safe inductive invariant.

- *reach* is the strongest inductive invariant.

Inductive Invariants for Example Program

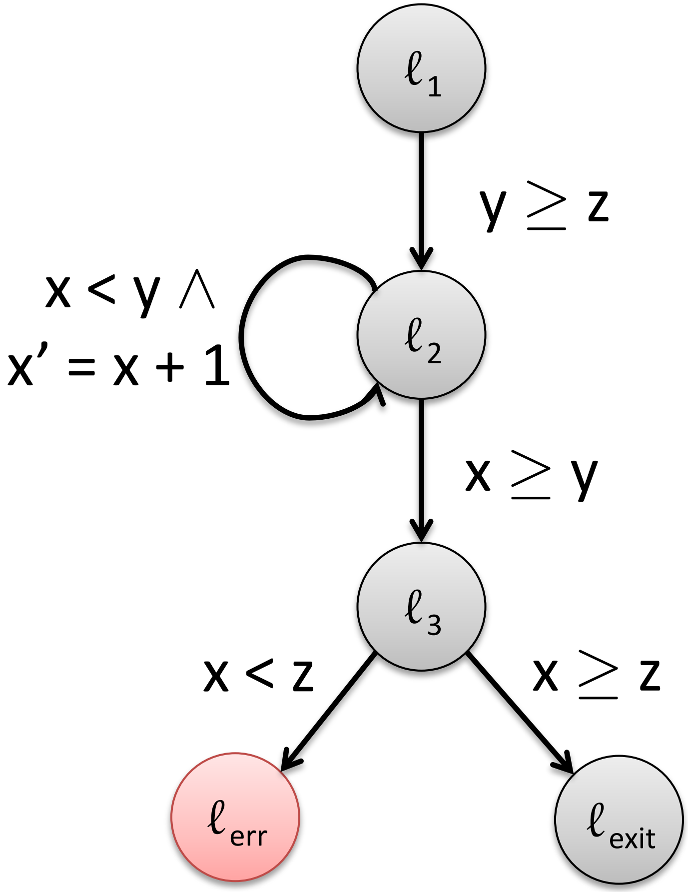
- weakest inductive invariant: true
  - set of all states
  - contains error states
- strongest inductive invariant: *reach*
  $pc = \ell_1 \lor pc = \ell_2 \land y \geq z \lor$
  $pc = \ell_3 \land y \geq z \land x \geq y \lor pc = \ell_{exit} \land y \geq z \land x \geq y$
- slightly weaker inductive invariant:
  $pc = \ell_1 \lor pc = \ell_2 \land y \geq z \lor$
  $pc = \ell_3 \land y \geq z \land x \geq y \lor pc = \ell_{exit}$
- Can we drop another conjunct in one of the disjuncts?

Inductive Invariants for Example Program

1: **assume** $y \geq z$;
2: **while** $x < y$ **do**
$\quad x := x + 1$;
3: **assert** $x \geq z$

Safe inductive invariant:

$pc = \ell_1 \;\vee$
$pc = \ell_2 \wedge y \geq z \;\vee$
$pc = \ell_3 \wedge y \geq z \wedge x \geq y \;\vee$
$pc = \ell_{exit}$



Computing Inductive Invariants

- We can compute the strongest inductive invariants by iterating *post* on *init.*
- Can we ensure that this process terminates?
- In general no: checking safety of programs is undecidable.
- But we can compute weaker inductive invariants
  - conservatively abstract the behavior of the program
  - iterate an abstraction of *post* that is guaranteed to terminate.

Abstracting *post*

- instead of iteratively applying post, use over-approximation *post*$^{\#}$ such that always

$$post(\rho,\ F) \models post^{\#}(\rho, F)$$

- decompose computation of *post*$^{\#}$ into two steps:
  - first, apply post and
  - then, over-approximate the result
- define abstraction function α such that

$$F \models \alpha(F)$$

- for a given abstraction function α define

$$post^{\#}(\rho, F) = \alpha\ (post(\rho, F))$$

Abstracting *reach* by *reach*#

- instead of computing *reach,* compute *reach*# such that

  $$reach \models reach^{\#}$$

- check whether *reach*# contains an error state
  if $\models reach^{\#} \land error \Rightarrow$ false then
  $\models reach \land error \Rightarrow$ false, i.e. program is safe

- compute *reach*# by applying iteration

  $$\begin{aligned} reach^{\#} = \ & \alpha(init) \lor \\ & post^{\#}(R, \alpha(init)) \lor \\ & post^{\#}(R, post^{\#}(R, \alpha(init))) \lor \ldots \\ = \ & \bigvee_{i \geq 0} (post^{\#})^{i}(R, init) \end{aligned}$$

- consequence: *reach* $\models$ *reach*#

Predicate Abstraction

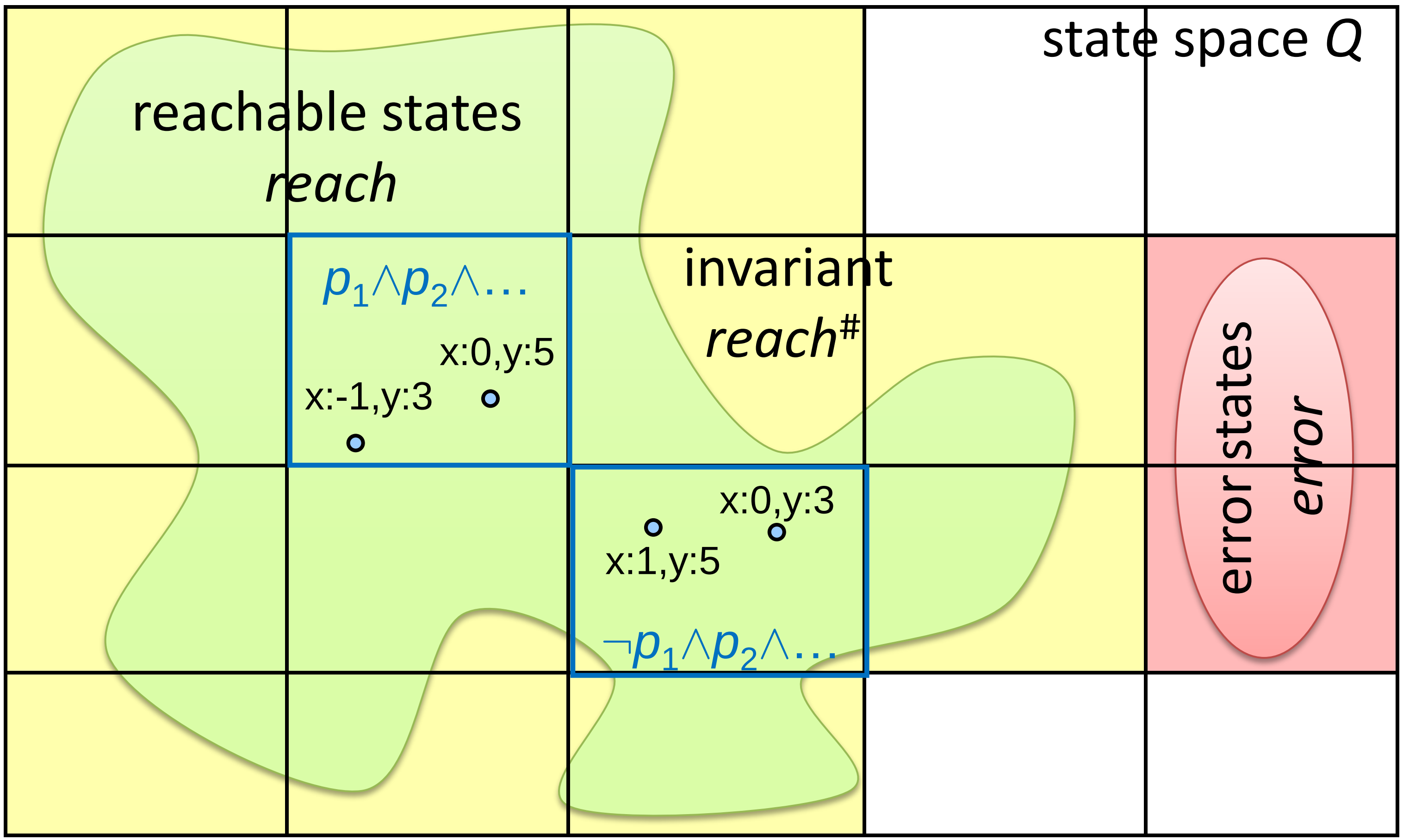
- construct abstraction α using a given set of building blocks, so-called predicates
- predicate = formula over program variables *V*
- fix finite set of predicates *Preds* = $\{p_1, ..., p_n\}$
- over-approximate F by conjunction of predicates in *Preds*

$$\alpha(F) = \bigwedge\{\, p \in Preds \mid F \vDash p \}$$

- computation of α(F) requires *n* theorem prover calls (*n* = number of predicates)

Predicate Abstraction

$p_1 \equiv x \leq 0 \quad p_2 \equiv y > 0 \quad \ldots$



Example: compute $\alpha(\text{pc} = \ell_2 \wedge y \geq z \wedge x + 1 \leq y)$

- *Preds* = $\{\text{pc} = \ell_1, \ldots, \text{pc} = \ell_{err}, y \geq z, x \leq y\}$

	$\text{pc} = \ell_1$	$\text{pc} = \ell_2$	$\text{pc} = \ell_3$	$\text{pc} = \ell_{exit}$	$\text{pc} = \ell_{err}$	$y \geq z$	$x \leq y$
$\text{pc} = \ell_2 \wedge y \geq z \wedge x + 1 \leq y$	$\not\models$	$\models$	$\not\models$	$\not\models$	$\not\models$	$\models$	$\models$

- result of abstraction = conjunction of implied predicates

$$\alpha(\text{pc} = \ell_2 \wedge y \geq z \wedge x + 1 \leq y) = \text{pc} = \ell_2 \wedge y \geq z \wedge x \leq y$$

Trivial Abstraction

- Result of applying predicate abstraction is *true* if none of the predicates is implied by F

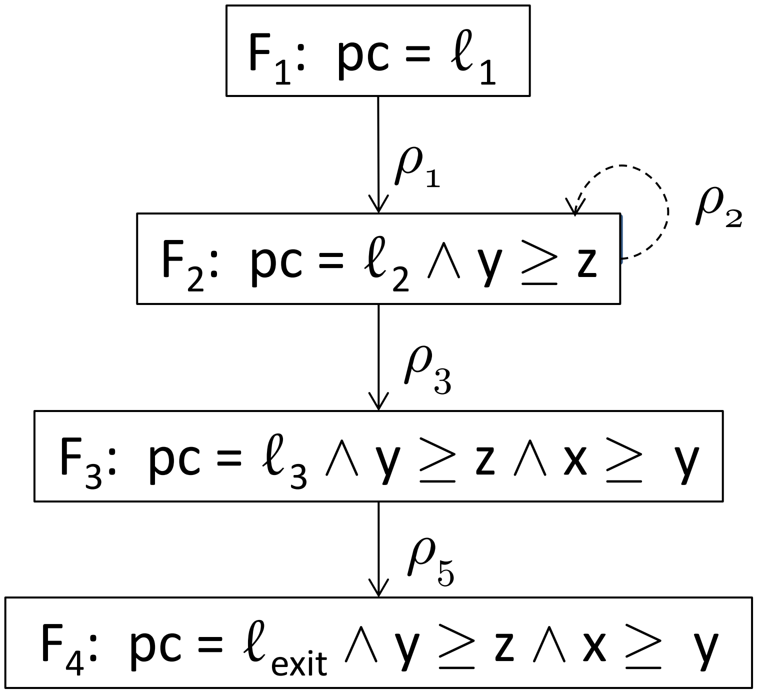
  $$\alpha(F) = \textit{true}$$

  "predicates are too specific"
- This is always the case if *Preds* = $\emptyset$

Algorithm `AbstReach`

begin
α := λ F. $\bigwedge\{ p \in Preds \mid \models F \Rightarrow p\}$
post$^\#$:= $\lambda\ \rho$ F. α(*post* (ρ, F))
reach$^\#$:= α(*init*)
Tree := $\emptyset$
Worklist := {*reach*$^\#$}
while *Worklist* $\neq \emptyset$ **do**
F := **choose from** *Worklist*
Worklist := *Worklist* \ {F}
for each $\rho \in$ R **do**
F' := *post#*(ρ, F)
if F' $\not\models$ *reach*$^\#$ **then**
reach$^\#$:= *reach*$^\#$ $\vee$ F'
Worklist := *Worklist* $\cup$ {F'}
Tree := *Tree* $\cup$ {(F', ρ, F)}
return (*reach*$^\#$, *Tree*)
end

Abstract Reachability Graph



$F_1 = \alpha(\mathit{init})$

$F_2 = \mathit{post}^{\#}(\rho_1, F_1)$

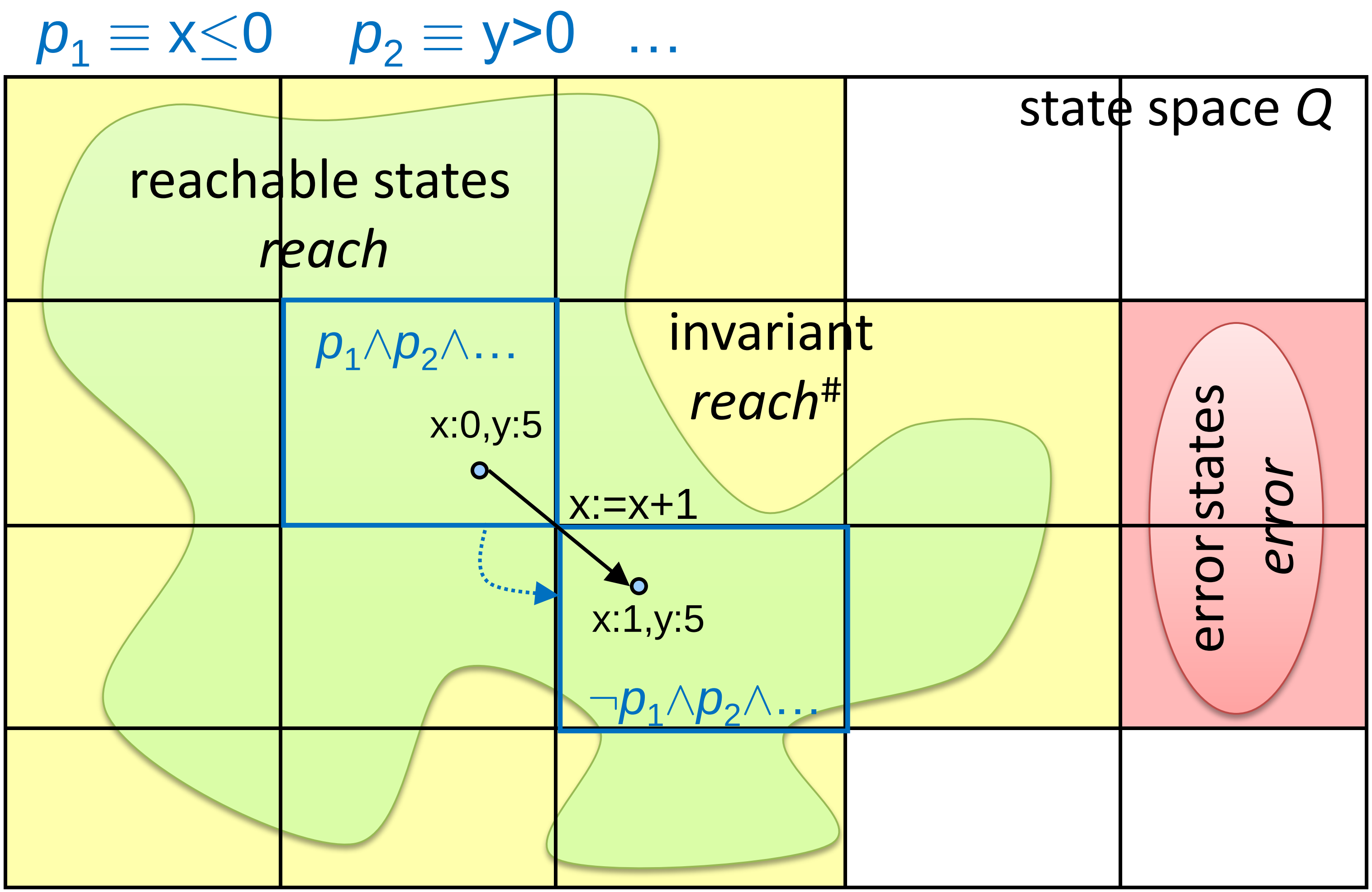
$\mathit{post}^{\#}(\rho_2, F_2) \models F_2$

$F_3 = \mathit{post}^{\#}(\rho_3, F_2)$

$F_4 = \mathit{post}^{\#}(\rho_5, F_3)$

- *Preds* = {*false*, pc = ℓ_1, ..., pc = ℓ_{err}, y $\geq$ z, x $\leq$ y}
- nodes $F_1, \ldots, F_4 \in Q^{\#}_{reach}$
- labeled edges $\in$ *Tree*
- dotted edge: entailment relation (here: $\mathit{post}^{\#}(\rho_2, F_2) \models F_2$

Abstract Reachability Graph



Example: Computing *reach*$^{\#}$

- *Preds* = {*false*, pc $= \ell_1$, ..., pc $= \ell_{err}$, y $\geq$ z, x $\leq$ y}
- over-approximation of the set of initial states *init*:

  $F_1 = \alpha(init) =$ pc $= \ell_1$

- apply *post*$^{\#}$ on F_1 and each program transition ρ_i

$$F_2 = post^{\#}(\rho_1, F_1) = \alpha(\underbrace{pc = \ell_2 \wedge y \geq z}_{post(\rho_1,\, F_1)}) = pc = \ell_2 \wedge y \geq z$$

$$post^{\#}(\rho_2, F_1) = \ldots = post^{\#}(\rho_5, F_1) = \bigwedge\{false, \ldots\} = false$$

Example: Computing *reach*$^{\#}$

- application of ρ_1, ρ_4, and ρ_5 on F_2 results in *false*
  (since ρ_1, ρ_4, ρ_5 are applicable only if pc = ℓ_1 or pc = ℓ_3 holds)
- for ρ_2 we obtain

  $post^{\#}(\rho_2, F_2) = \alpha(pc = \ell_2 \wedge y \geq z \wedge x \leq y) = pc = \ell_2 \wedge y \geq z \wedge x \leq y$

  result is F_2, which is already subsumed by *reach*$^{\#}$
- for ρ_3 we obtain

  $post^{\#}(\rho_3, F_2) = \alpha(pc = \ell_3 \wedge y \geq z \wedge x \geq y)$

  $= pc = \ell_3 \wedge y \geq z \wedge x \geq y$

  $= F_3$

  add new node F_3 to *reach*$^{\#}$, new edge to *Tree*

Example: Computing *reach*$^\#$

- application of ρ_1, ρ_2, and ρ_3 on F_3 results in *false*
- for ρ_5 we obtain

  post$^\#$ $(\rho_5, F_3) = \alpha(pc = \ell_{exit} \wedge y \geq z \wedge x \geq y)$

  $= pc = \ell_{exit} \wedge y \geq z \wedge x \geq y$

  $= F_4$

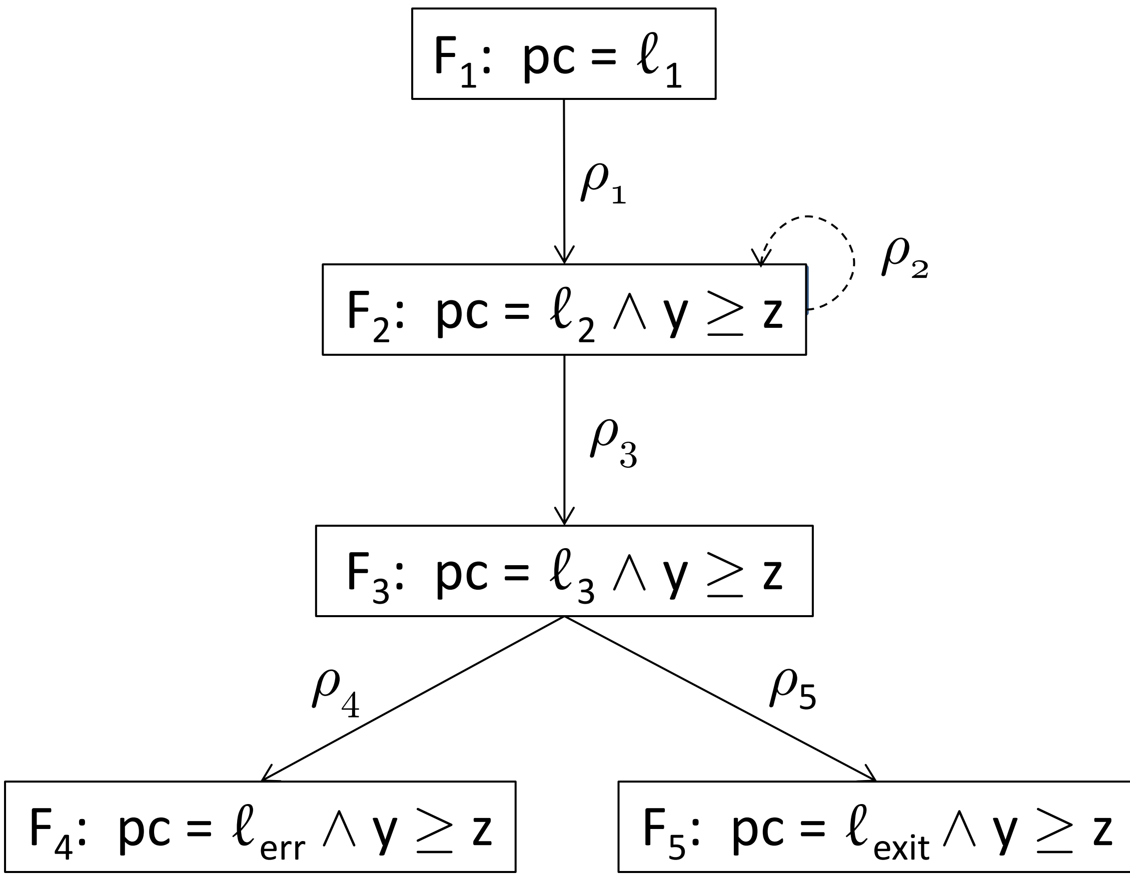
  new node F_4 in *reach*$^\#$, new edge in *Tree*

- for ρ_4 (assertion violation) we obtain

  post$^\#$ $(\rho_4, F_3) = \alpha(pc = \ell_{err} \wedge y \geq z \wedge x \geq y \wedge x < z)$ = *false*
- any further application of program transitions does not compute any additional reachable states
- thus, *reach*$^\#$ $= F_1 \vee F_2 \vee F_3 \vee F_4$
- since *reach*$^\#$ $\wedge\ pc = \ell_{err} \models$ *false* the program is proved safe.

Abstract Reachability Graph

with *Preds* = {*false*, $\mathrm{pc} = \ell_1$, ..., $\mathrm{pc} = \ell_{\mathrm{err}}$, $y \geq z$}



$F_1 = \alpha(\mathit{init})$

$F_2 = \mathit{post}^{\#}(\rho_1, F_1)$

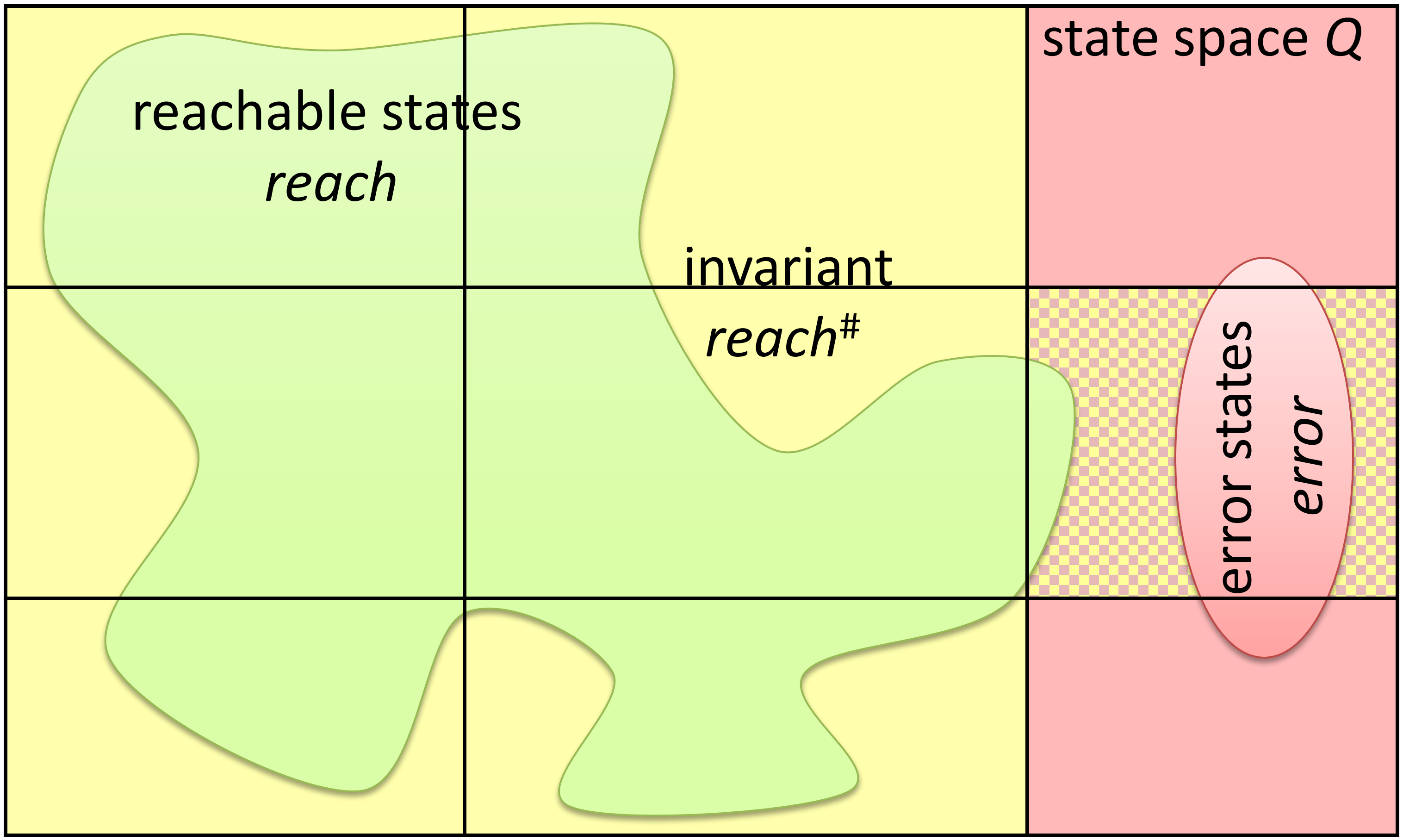
$\mathit{post}^{\#}(\rho_2, F_2) \models F_2$

$F_3 = \mathit{post}^{\#}(\rho_3, F_2)$

$F_4 = \mathit{post}^{\#}(\rho_4, F_3)$

$F_5 = \mathit{post}^{\#}(\rho_5, F_3)$

Too Coarse Abstraction



Finding the Right Predicates

- omitting just one predicate (in the example: x $\geq$ y) may lead to an over-approximation *reach*$^\#$ such that

  $$reach^\# \wedge error \not\models false$$

  that is, algorithm `AbstReach` fails to prove safety of the program without the predicate x $\geq$ y.
- How can we find the right predicates?

Counterexample Path

- Tree relation records sequence of transitions leading to F_4
  - apply ρ_1 to F_1 and obtain F_2
  - apply ρ_3 to F_2 and obtain F_3
  - apply ρ_4 to F_3 and obtain F_4
- counterexample path: sequence of transitions ρ_1, ρ_3, ρ_4
- Using this path and the functions α and *post*# for the current set of predicates we obtain

$$F_4 = post^{\#}(\rho_4, post^{\#}(\rho_3, post^{\#}(\rho_1, \alpha(init))))$$

- that is, F_4 is the over-approximation of the post-condition computed along the counterexample path.

Analysis of Counterexample Path

- check if the counterexample path also leads to the error states when no over-approximation is applied

- compute

  $post(\rho_4, post(\rho_3, post(\rho_1, init)))$

  $= post(\rho_4, post(\rho_3, pc = \ell_2 \wedge y \geq z))$

  $= post(\rho_4, pc = \ell_2 \wedge y \geq z \wedge x \geq y)$

  $= false$

- by executing the program transitions ρ_1, ρ_3, and ρ_4 it is not possible to reach any error state.
- conclude that the over-approximation is too coarse when dealing with the above path.

Refinement of Abstraction

- need a more precise over-approximation that will prevent *reach*$^\#$ from including error states.
- need a more precise over-approximation that will prevent α from including states that lead to error states along the path ρ_1, ρ_3, ρ_4.
- need a refined abstraction function and a corresponding *post*$^\#$ such that the execution of AbstReach along the counterexample path does not compute a set of states that contains some error states

$$post^\#(\rho_4, post^\#(\rho_3, post^\#(\rho_1, \alpha(init)))) \wedge \text{error} \models \text{false}$$

Over-Approximation along Counterexample Path

- goal: $post^{\#}(\rho_4, post^{\#}(\rho_3, post^{\#}(\rho_1, \alpha(init)))) \wedge \text{error} \models \text{false}$
- find formulas F_1, F_2, F_3, F_4 such that

$$init \models F_1$$
$$post(\rho_1, F_1) \models F_2$$
$$post(\rho_3, F_2) \models F_3$$
$$post(\rho_4, F_3) \models F_4$$
$$F_4 \wedge error \models false$$

- thus, F_1, ..., F_4 guarantee that no error state can be reached but may still approximate, i.e., allow additional states
- example choice for F_1, ..., F_4

$F_1 = pc = \ell_1$ $F_2 = pc = \ell_2 \wedge y \geq z$,
$F_3 = pc = \ell_3 \wedge x \geq z$ $F_4 = false$

Refinement of Predicate Abstraction

- given formulas F_1, F_2, F_3, F_4 such that

$$
\begin{aligned}
&\textit{init} \models F_1 \\
&\textit{post}(\rho_1, F_1) \models F_2 \\
&\textit{post}(\rho_3, F_2) \models F_3 \\
&\textit{post}(\rho_4, F_3) \models F_4 \\
&F_4 \wedge \textit{error} \models \textit{false}
\end{aligned}
$$

- add atoms of F_1, ..., F_4 to *Preds.*
- refinement guarantees that counterexample path ρ_1, ρ_3, ρ_4 is eliminated.

CEGAR: Counter-Example Guided Abstraction Refinement Loop

```
function AbstRefineLoop
  begin
    Preds := ∅;
    repeat
      (reach#, Tree) := AbstReach(Preds)
      if exists  F ∈ reach# such that  F ∧ error ⊭ false then
        path := MakePath(F, Tree)
        if FeasiblePath(path) then
          return "counterexample path: path"
        else
          Preds := Preds ∪ RefinePath(path)
      else
        return "program is safe"
  end
```

Path Computation

```
function MakePath
  input
    F_err - reachable abstract error state formula
    Tree – abstract reachability tree
  begin
    path := empty sequence
    F' := F_err
    while exist F and ρ such that (F, ρ, F') ∈ Tree do
        path := ρ . path
        F' := F
    return path
  end
```

Feasibility of a Path

function `FeasiblePath`

input ρ_1 … ρ_n - path

begin

F := *post*($\rho_1 \circ \ldots \circ \rho_n$, *init*)

if F $\wedge$ error $\models$ *false* **then**

return *true*

else

return *false*

end

Counterexample-Guided Predicate Discovery

function `RefinePath`

input

$\rho_1 \dots \rho_n$ – infeasible path

begin

$F_1, \dots, F_{n+1}$:= compute such that

init $\models F_1$ and

post(ρ_1, F_1) $\models F_2$ and ... *post*(ρ_n, F_n $\models F_{n+1}$ and

$F_{n+1} \wedge$ *error* $\models$ *false*

return $\{F_1, \dots, F_{n+1}\}$

end

omitted: particular algorithm for finding the $F_1, \dots, F_{n+1}$