# Invariant Generation

## CS560: Reasoning About Programs

Roopsha Samanta

PURDUE
UNIVERSITY

# Roadmap

Previously

- Semi-automating Hoare logic using verification condition generation

Today

- Automated invariant generation
- Forward propagation using strongest postconditions
- Abstract interpretation

# Invariant vs. Inductive Invariant

- A formula $I$ annotating location $L$ of a program $S$ is an **invariant** iff *it holds whenever control reaches location* $L$ :

  for each computation $s_0, s_1, s_2, \ldots$ of $S$ and each index $i$, $pc(s_i) = L$ implies $s_i \vDash I$.

Hard to prove an annotation is an invariant as there can be an infinite no. of computations

- The annotations of $S$ are invariant iff each annotation is invariant

- The annotations of $S$ are **inductive** (or, **inductive invariants**) iff all corresponding verification conditions are valid.

# Invariant vs. Inductive Invariant

- A formula $I$ annotating location $L$ of a program $S$ is an **invariant** iff *it holds whenever control reaches location* $L$ :

  for each computation $s_0, s_1, s_2, \ldots$ of $S$ and each index $i$, $pc(s_i) = L$ implies $s_i \vDash I$.

- The annotations of $S$ are invariant iff each annotation is invariant

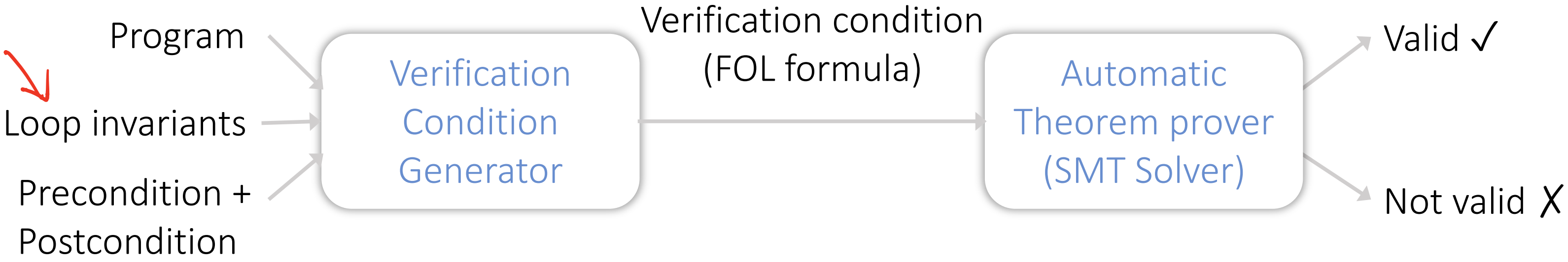
- The annotations of $S$ are **inductive** (or, **inductive invariants**) iff all corresponding verification conditions are valid.

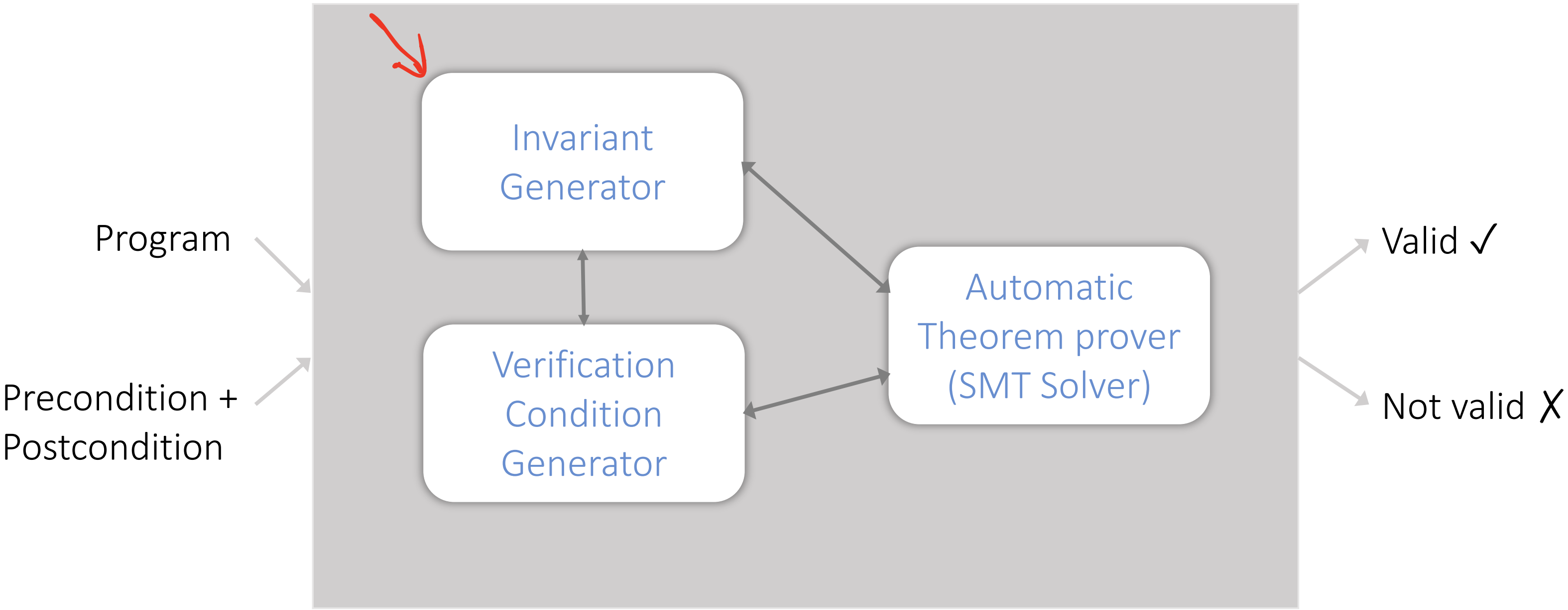
Hard to prove an annotation is an invariant as there can be an infinite no. of computations

Inductive invariants are the invariants we can *prove*

# Loop Invariant

- A **loop invariant** $I$ is an annotation for a loop that must hold at the beginning of every iteration.
  - $I$ must hold before the loop begins
  - $I$ must hold after every iteration


Program
Loop invariants
Precondition +
Postcondition
Verification
Condition
Generator
Verification condition
(FOL formula)
Automatic
Theorem prover
(SMT Solver)
Valid ✓
Not valid ✗


Program
Precondition +
Postcondition
Invariant Generator
Verification Condition Generator
Automatic Theorem prover (SMT Solver)
Valid ✓
Not valid ✗

# Inductive assertion method

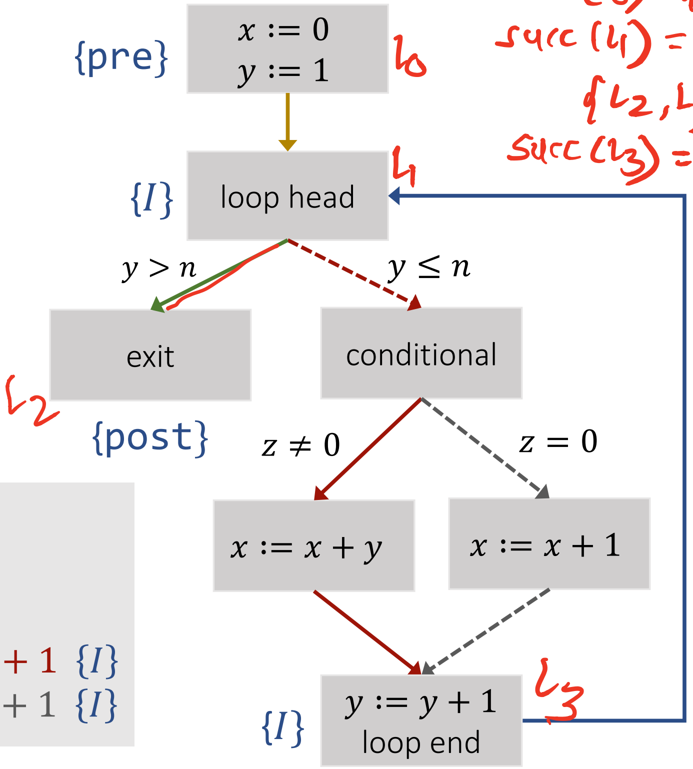
$\mathcal{L} = \{l_0, l_1, l_2, l_3\}$

$succ(l_0) = \{l_1\}$

$succ(l_1) = \{l_2, l_3\}$

$succ(l_3) = \{l_1\}$

- Represent program as control-flow graph with annotations/inductive assertions
- Identify basic paths
- For each basic path: check if corresponding Hoare triple is valid




VCs

{pre} $x := 0; y := 1$ {$I$}

{$I$} assume $y > n$ {post}

{$I$} skip {$I$}

{$I$} assume $y \leq n$; assume $z \neq 0$; $x := x + y; y := y + 1$ {$I$}

{$I$} assume $y \leq n$; assume $z = 0$; $x := x + 1; y := y + 1$ {$I$}

A cutset $\mathcal{L}$ is a set of locations (called cutpoints) such that every path between adjacent cutpoints is a basic path

**Invariant Generation:**

Find a map $\mu$: $\mathcal{L} \mapsto$ FOL such that for each basic path, its corresponding Hoare triple is valid

The map $\mu$ is called an inductive assertions map

ForwardPropagate(pre, $\mathcal{L}$)

$W$ := $\{L_{init}\}$

$\mu(L_{init})$ := pre

**foreach** $L \in \mathcal{L} \setminus \{L_{init}\}$ **do** $\mu(L)$ := $\bot$

**while** $W \neq \phi$ **do**

  pick $L_j \in W$; $W$ := $W \setminus \{L_j\}$

  **foreach** $L_k \in$ succ($L_j$) **do**

    **if** $\neg\left(sp\left(S_j; \ldots; S_k, \mu(L_j)\right) \Rightarrow \mu(L_k)\right)$

    **then**

      $\mu(L_k)$ := $\mu(L_k) \vee sp(S_j; \ldots; S_k, \mu(L_j))$

      $W$ := $W \cup \{L_k\}$

**return** $\mu$

```
ForwardPropagate(pre, 𝓛)
  W := {L_init}
  μ(L_init) := pre
  foreach L ∈ 𝓛\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W;  W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp(S_j; …; S_k, μ(L_j)) ⇒ μ(L_k))
      then
        μ(L_k) := μ(L_k) ∨ sp(S_j; …; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

$$
\begin{array}{l}
\mathtt{ForwardPropagate}(\mathtt{pre}, \mathcal{L}) \\
\quad W := \{L_{init}\} \\
\quad \mu(L_{init}) := \mathtt{pre} \\
\quad \mathtt{foreach}\ L \in \mathcal{L} \setminus \{L_{init}\}\ \mathtt{do}\ \mu(L) := \bot \\
\quad \mathtt{while}\ W \neq \phi\ \mathtt{do} \\
\qquad \mathtt{pick}\ L_j \in W;\ W := W \setminus \{L_j\} \\
\qquad \mathtt{foreach}\ L_k \in \mathtt{succ}(L_j)\ \mathtt{do} \\
\qquad\quad \mathtt{if}\ \neg\big(sp(S_j; \ldots; S_k, \mu(L_j)) \Rightarrow \mu(L_k)\big) \\
\qquad\quad \mathtt{then} \\
\qquad\qquad \mu(L_k) := \mu(L_k) \vee sp(S_j; \ldots; S_k, \mu(L_j)) \\
\qquad\qquad W := W \cup \{L_k\} \\
\quad \mathtt{return}\ \mu
\end{array}
$$

Set of locations in $\mathcal{L}$ that need to be processed

$\texttt{ForwardPropagate}(\texttt{pre}, \mathcal{L})$

$\quad W := \{L_{init}\}$

$\quad \mu(L_{init}) := \texttt{pre}$

$\quad \textbf{foreach } L \in \mathcal{L} \setminus \{L_{init}\} \textbf{ do } \mu(L) := \bot$

$\quad \textbf{while } W \neq \phi \textbf{ do}$

$\qquad \texttt{pick } L_j \in W;\ W := W \setminus \{L_j\}$

$\qquad \textbf{foreach } L_k \in \texttt{succ}(L_j) \textbf{ do}$

$\qquad\quad \textbf{if } \neg\big(sp\big(S_j; \ldots; S_k, \mu(L_j)\big) \Rightarrow \mu(L_k)\big)$

$\qquad\quad \textbf{then}$

$\qquad\qquad \mu(L_k) := \mu(L_k) \vee sp(S_j; \ldots; S_k, \mu(L_j))$

$\qquad\qquad W := W \cup \{L_k\}$

$\quad \textbf{return } \mu$

Set of locations in $\mathcal{L}$ that need to be processed

Initial map

```
ForwardPropagate(pre, ℒ)
  W := {L_init}
  μ(L_init) := pre
  foreach L ∈ ℒ\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W;  W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp(S_j; … ; S_k, μ(L_j)) ⇒ μ(L_k))
      then
        μ(L_k) := μ(L_k) ∨ sp(S_j; … ; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

Set of locations in $\mathcal{L}$ that need to be processed

Initial map

All locations in which basic paths originating at $L_j$ terminate

```
ForwardPropagate(pre, 𝓛)
  W := {L_init}
  μ(L_init) := pre
  foreach L ∈ 𝓛\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W; W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp(S_j; … ; S_k, μ(L_j)) ⇒ μ(L_k))
      then
        μ(L_k) := μ(L_k) ∨ sp(S_j; … ; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

Set of locations in $\mathcal{L}$ that need to be processed

Initial map

All locations in which basic paths originating at $L_j$ terminate

$sp(S_j; \ldots; S_k, \mu(L_j))$ contains states not contained in $\mu(L_k)$. Hence, update $\mu(L_k)$ with these new states

```
ForwardPropagate(pre, ℒ)
  W := {L_init}
  μ(L_init) := pre
  foreach L ∈ ℒ\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W; W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp(S_j; … ; S_k, μ(L_j)) ⇒ μ(L_k))
      then
        μ(L_k) := μ(L_k) ∨ sp(S_j; … ; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

Undecidable for FOL

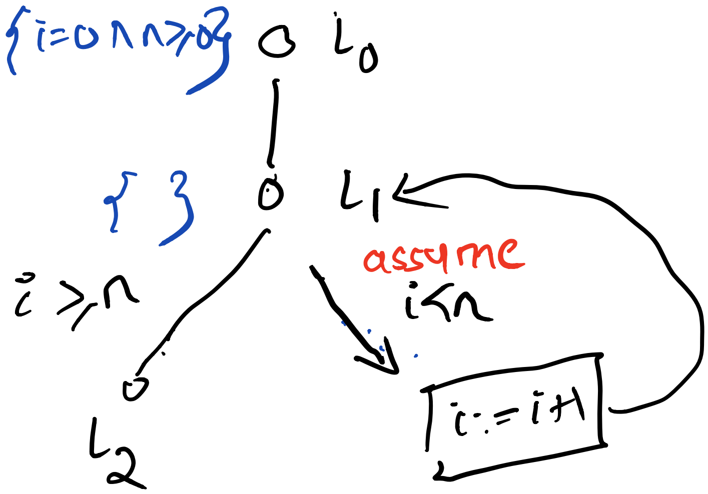
```
ForwardPropagate(pre, ℒ)
  W := {L_init}
  μ(L_init) := pre
  foreach L ∈ ℒ\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W;  W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp(S_j; …; S_k, μ(L_j)) ⇒ μ(L_k))
      then
        μ(L_k) := μ(L_k) ∨ sp(S_j; …; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

May not terminate

Undecidable for FOL

Pre : $i=0 \wedge n \geq 0$

while $\{I\}$ $(i<n)$
    $i := i+1$




$\mu(L_1) = \bot$
$\vee\ i=0 \wedge n \geq 0$
$\vee\ i=1 \wedge n > 0$
$\vee\ i=2 \wedge n > 1$
$\vdots$

Doesn't converge!
But "$0 \leq i \leq n$" is a loop invariant!

```
ForwardPropagate(pre, 𝓛)
  W := {L_init}
  μ(L_init) := pre
  foreach L ∈ 𝓛\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W; W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp(S_j; … ; S_k, μ(L_j)) ⇒ μ(L_k))
      then
        μ(L_k) := μ(L_k) ∨ sp(S_j; … ; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

A state is reachable in a program if it appears in some computation of the program starting from a state satisfying the precondition

`ForwardPropagate` computes the **exact** set of reachable states using a **(least) fixpoint computation**: repeated symbolic execution of program starting from $\perp$ until equilibrium

```
ForwardPropagate(pre, 𝓛)
  W := {L_init}
  μ(L_init) := pre
  foreach L ∈ 𝓛\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W;  W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp(S_j; … ; S_k, μ(L_j)) ⇒ μ(L_k))
      then
        μ(L_k) := μ(L_k) ∨ sp(S_j; … ; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

A state is reachable in a program if it appears in some computation of the program starting from a state satisfying the precondition

Inductive assertions usually **over-approximate** the set of reachable states: every reachable state satisfies the annotation, but other unreachable states can also satisfy the annotation

**Abstract interpretation** can help force termination!

Ensures implication checking is decidable by manipulating **abstract states** --- an artificially constrained form of state sets

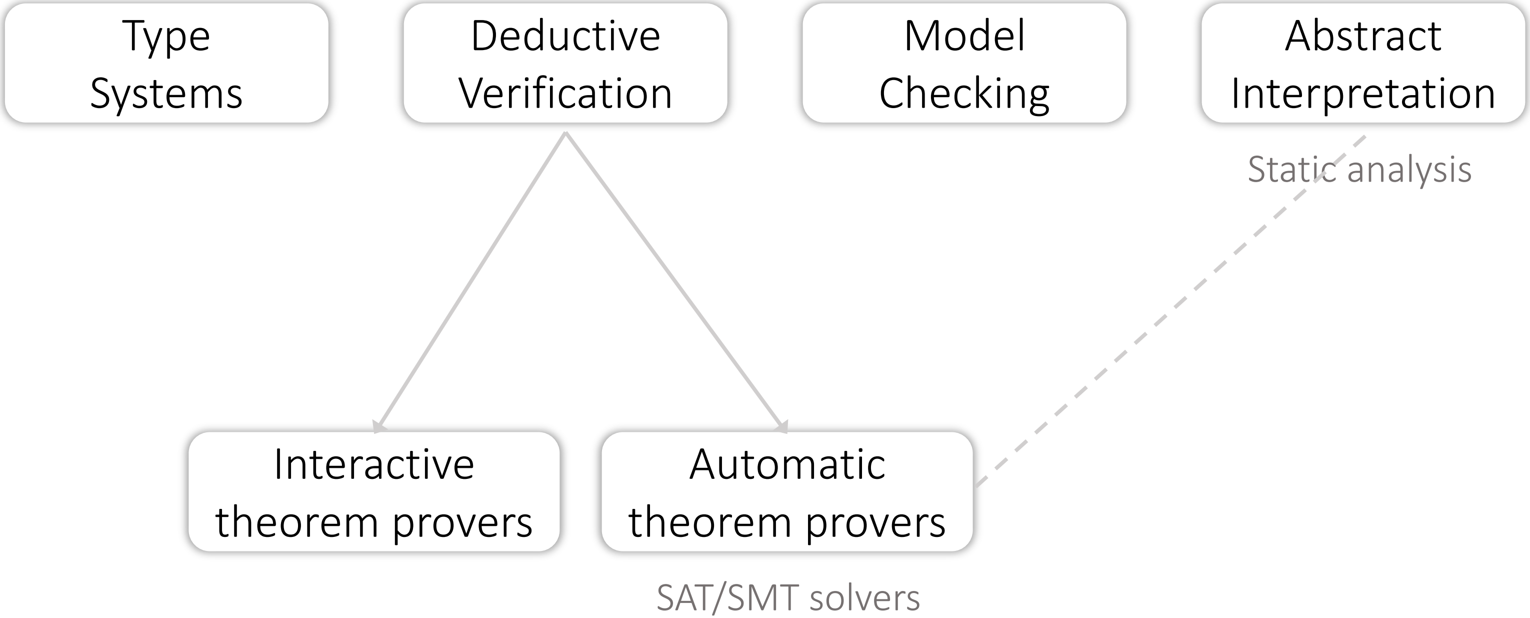
**Abstract interpretation** can help force termination!

Helps termination of main loop via **widening** --- guessing a limit overapproximation to a sequence of state sets.

$I_0 \vee I_1 \vee I_2 \vee I_3 \vee \ldots$

$\shortparallel \quad \vee \quad \shortparallel$

$\{true\} \quad \{true\}$


Type Systems
Deductive Verification
Model Checking
Abstract Interpretation
Static analysis
Interactive theorem provers
Automatic theorem provers
SAT/SMT solvers

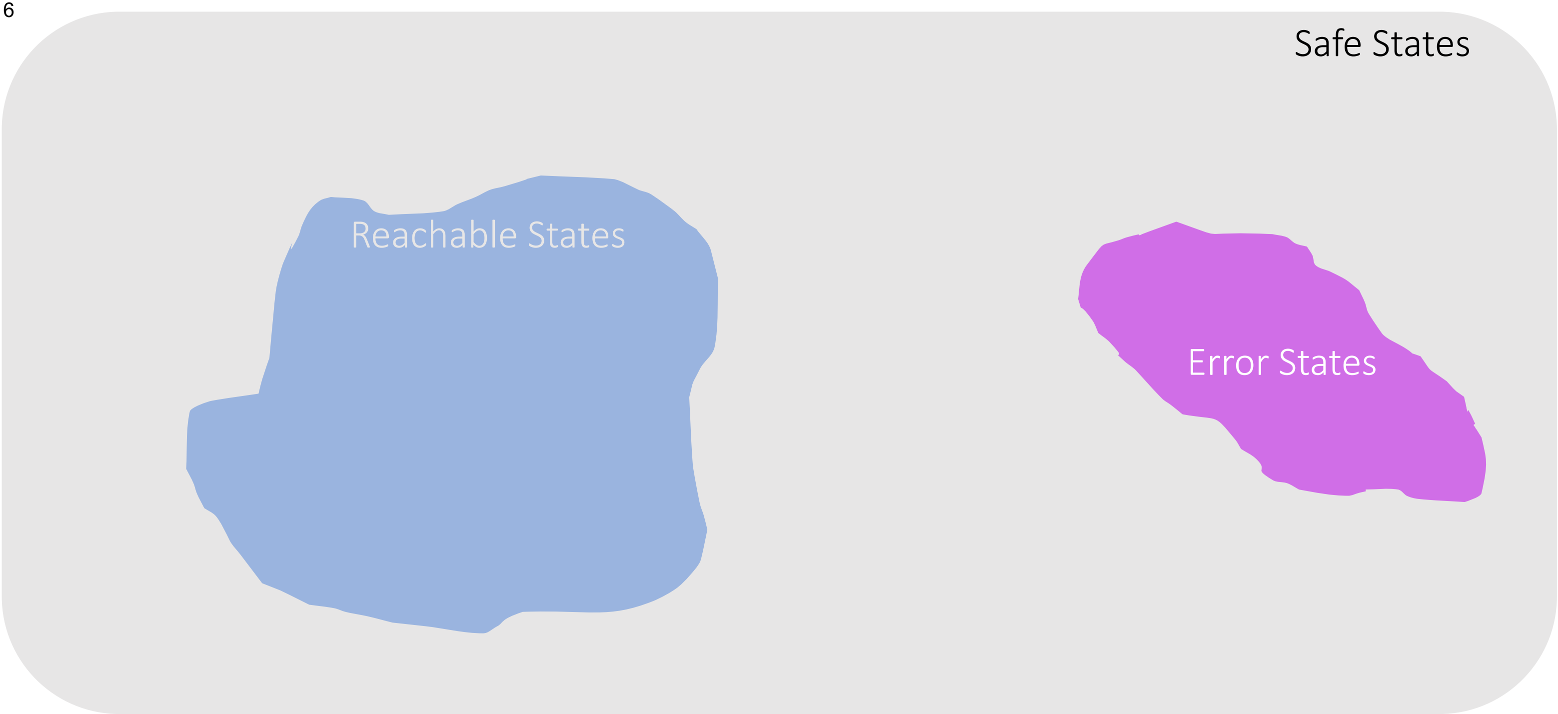
# Abstraction Interpretation

Patrick Cousot

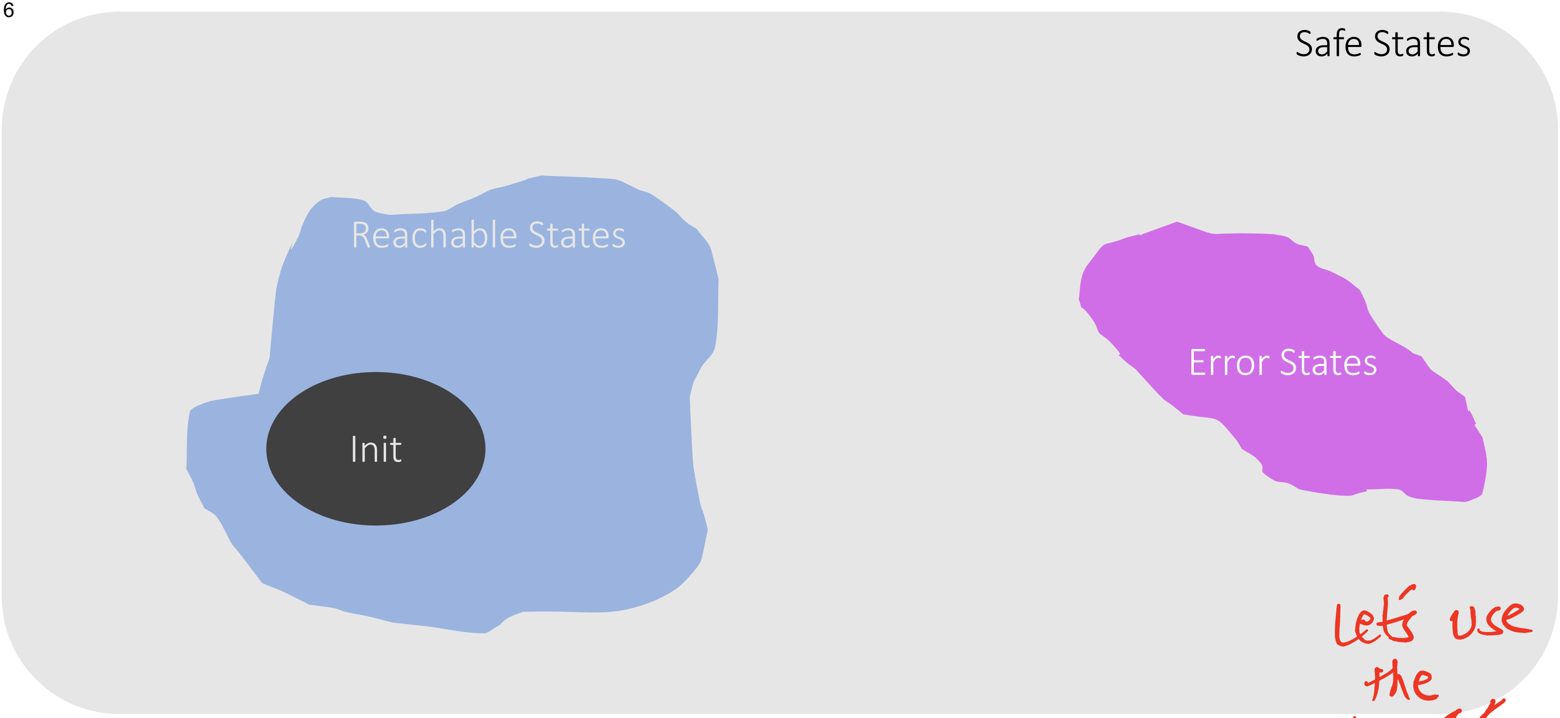
Radhia Cousot

Cousot & Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, 1977

A framework for designing sound-by-construction static analyses

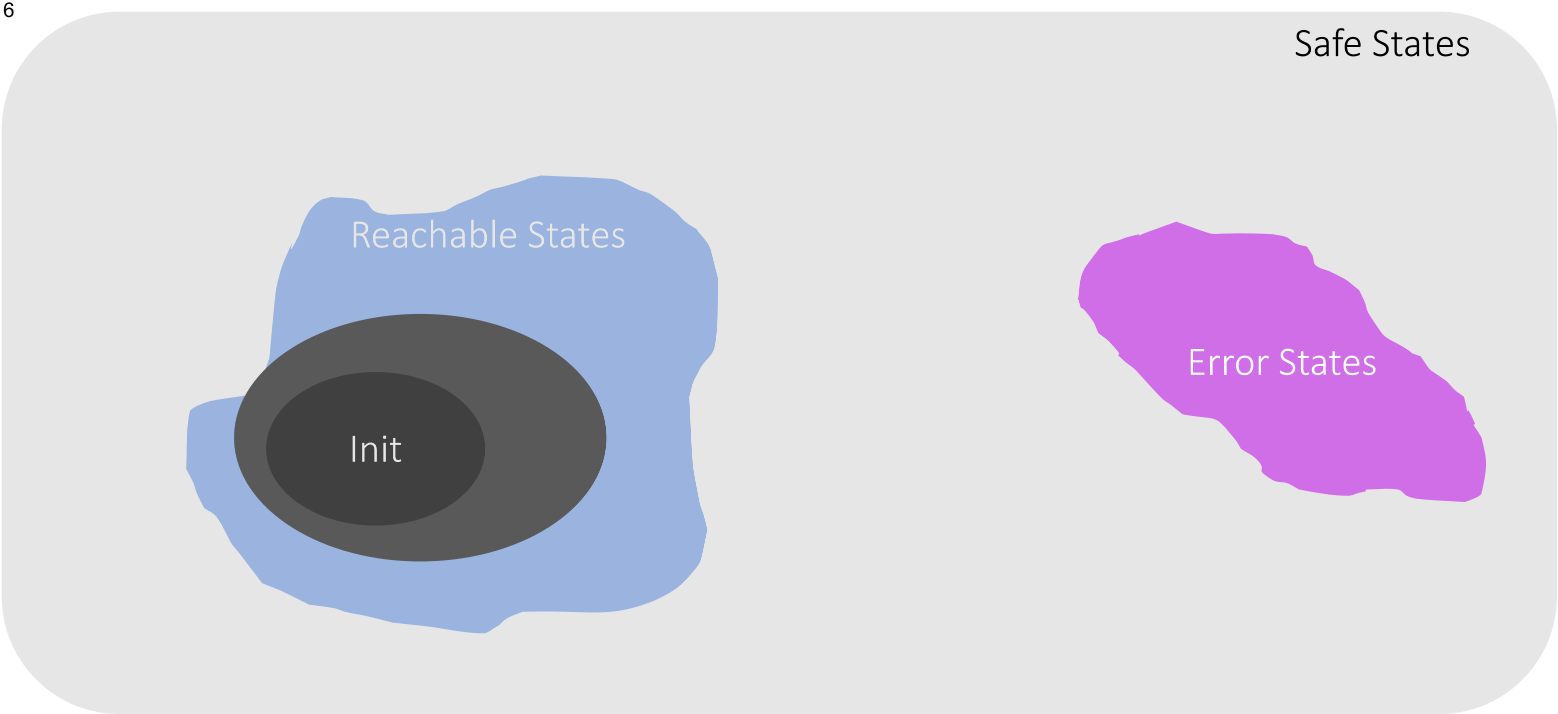



Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain

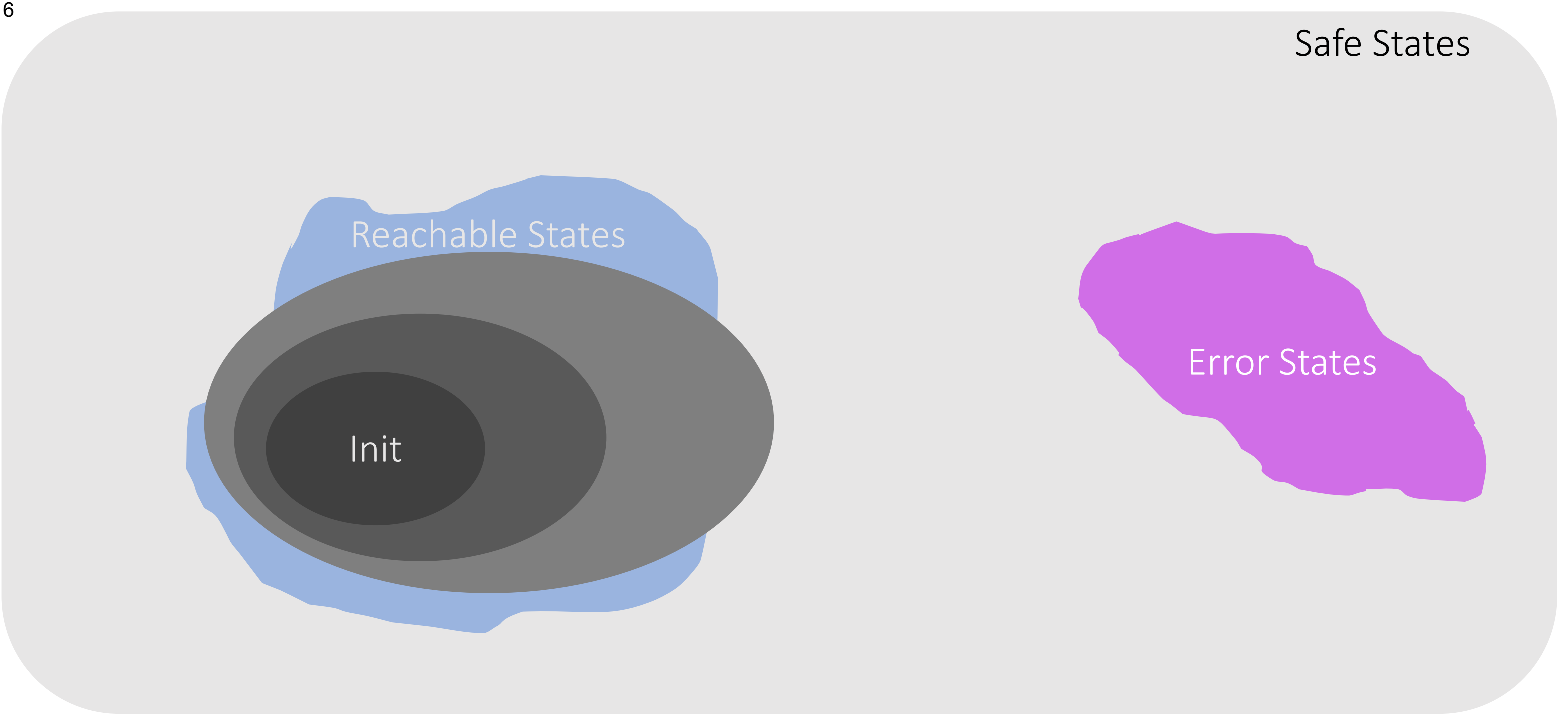



Let's use the "oval" abstract domain

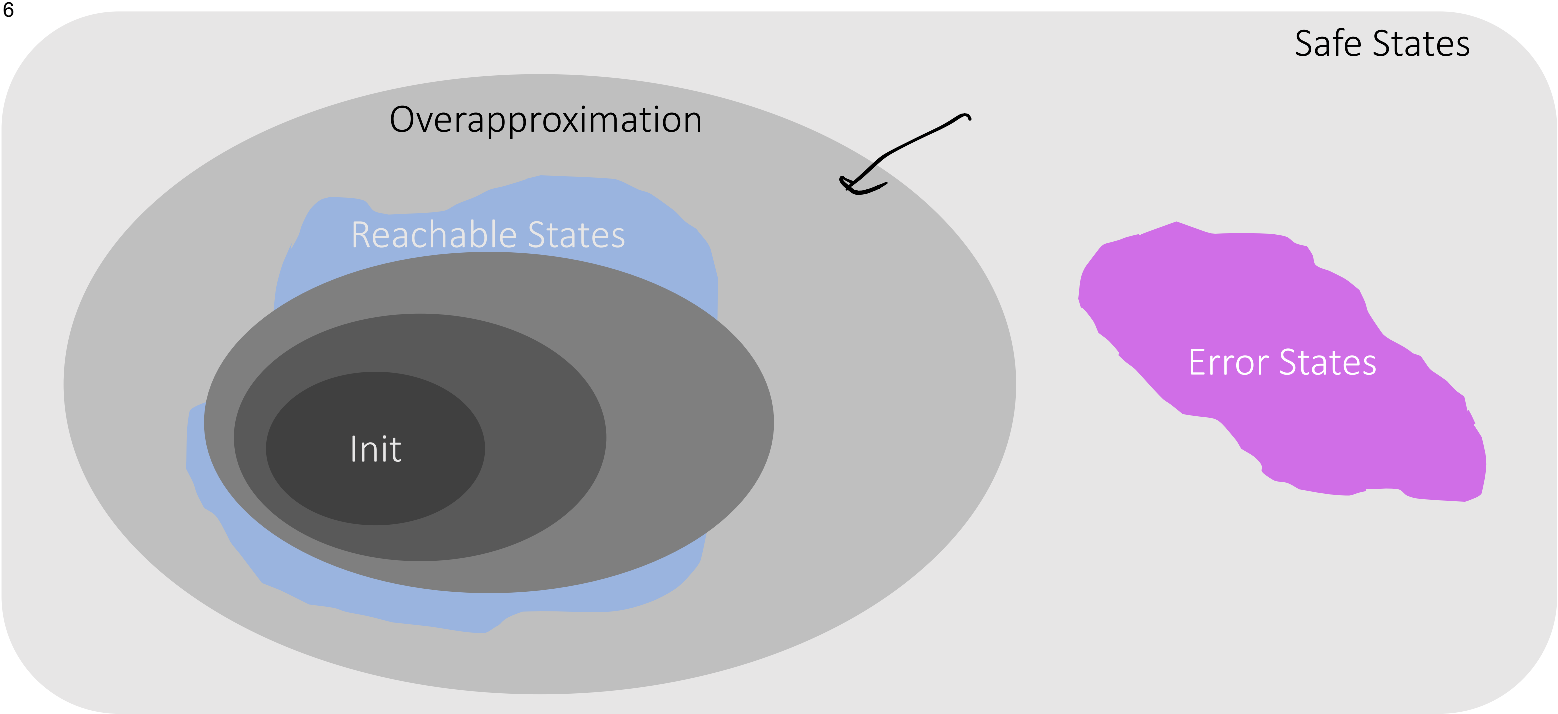
Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain



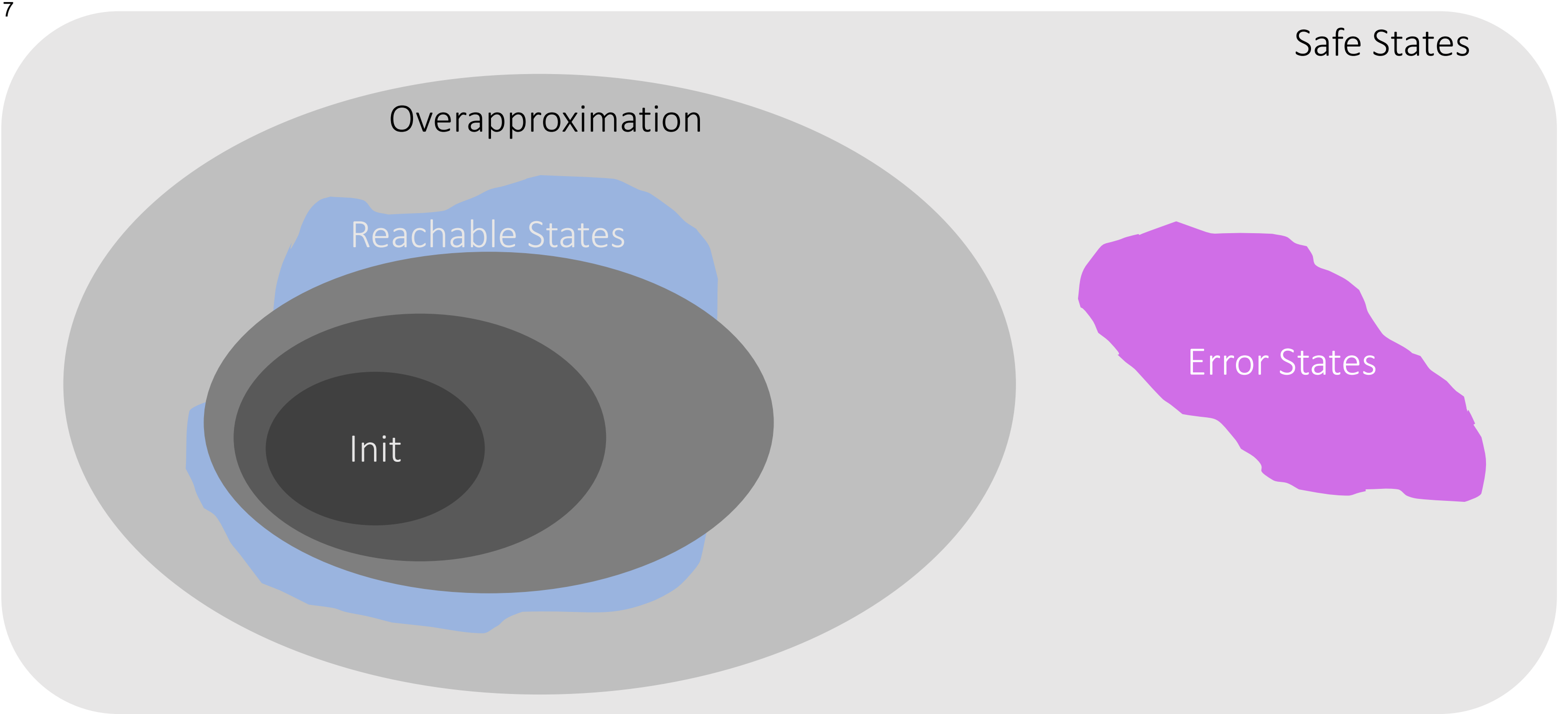

Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain



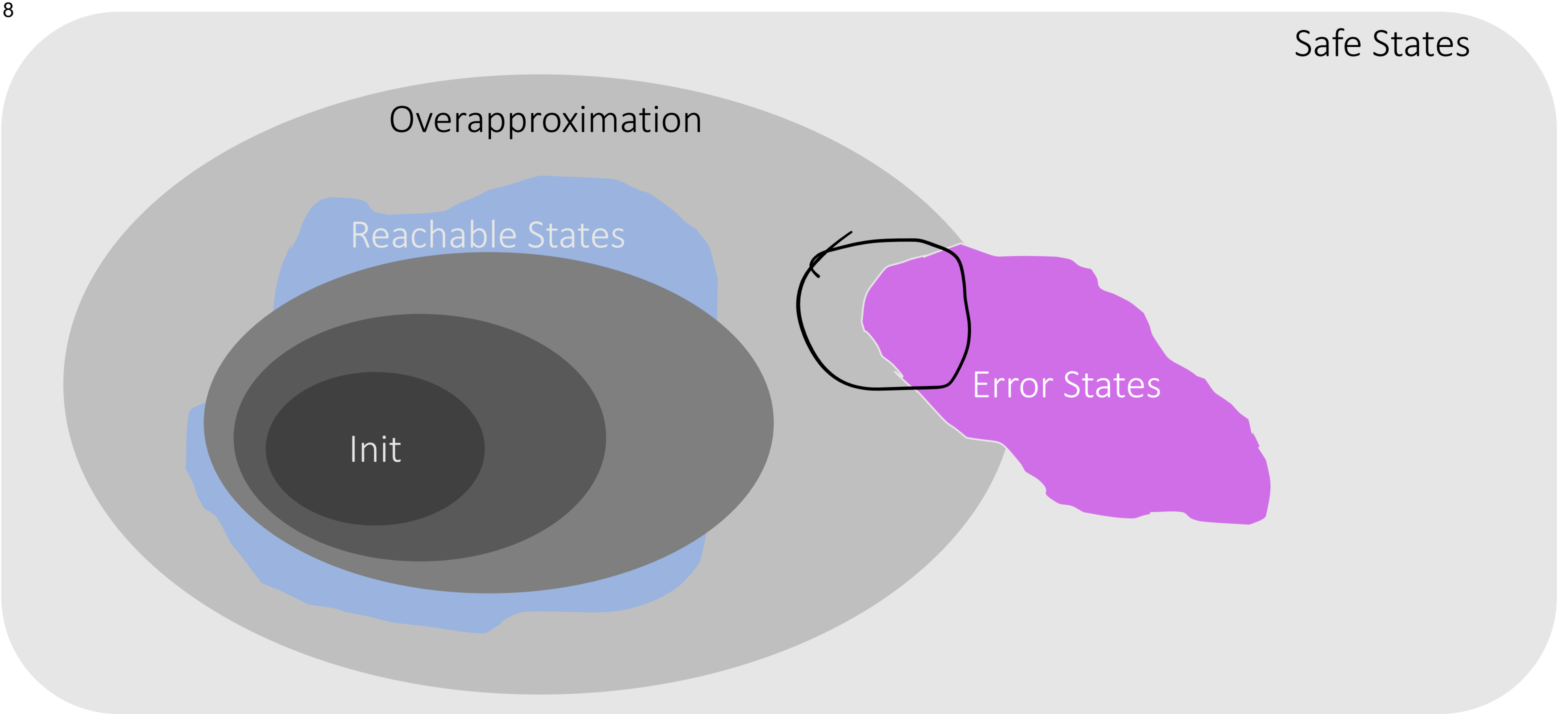

Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain



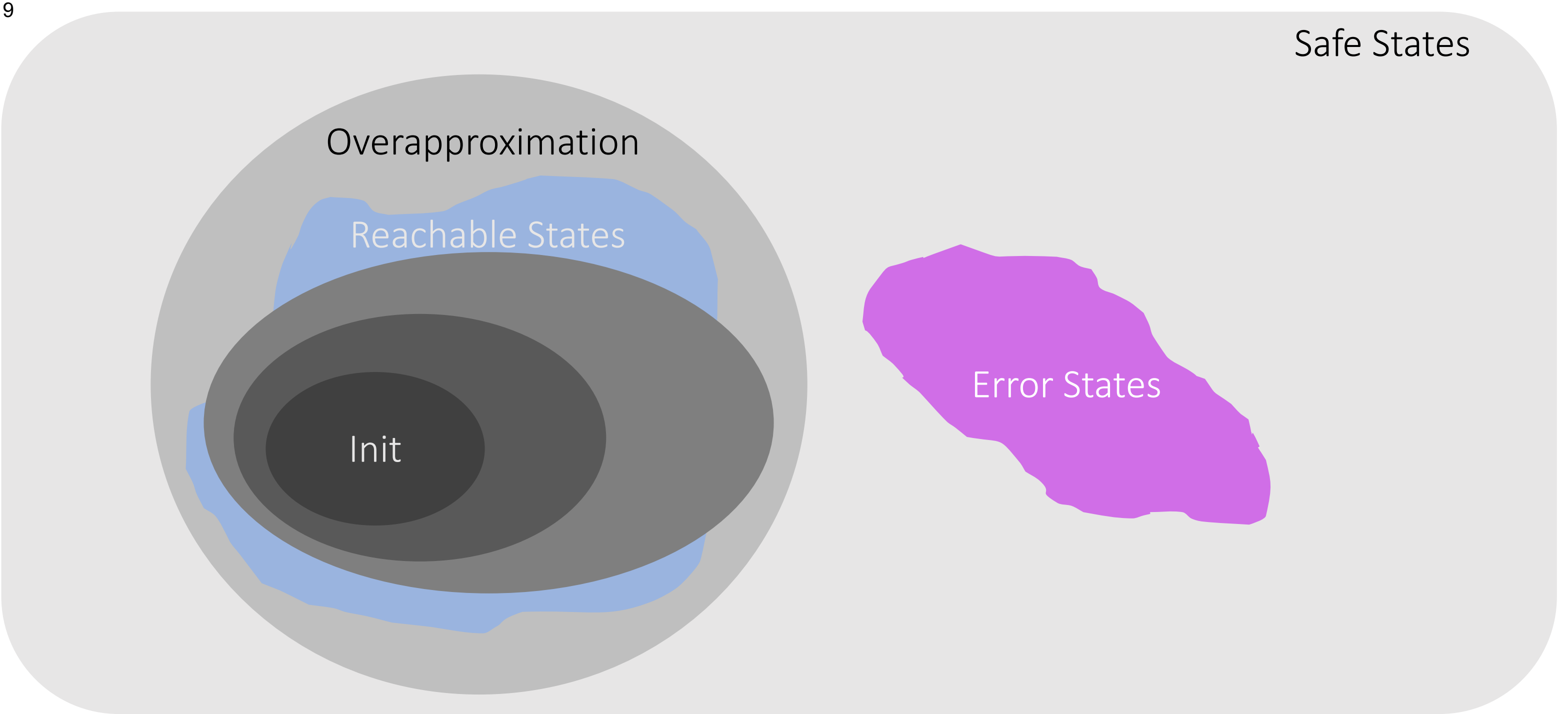

Key Idea: Overapproximate program behaviour by computing a (least) fixpoint under an abstract domain




Soundness: If abstract program is safe, then concrete program is safe
**No false negative** (no false declaration of absence of errors)




If abstract program is unsafe, then concrete program may still be safe.
**False positive** (false alarm)




Ideally, we want the most precise (strongest) overapproximation of reachable states expressible in the abstract domain of choice.

Choose abstract domain based on
what you want to prove about the program!

# Step 1: Abstract domain

*An abstract domain is just a set of abstract values we want to track*

Define an abstract domain $D$ as a syntactic class of $\Sigma_T$-formulas of some theory $T$

Each member $\Sigma_T$-formula represents a set of concrete states/elements

In addition, every abstract domain contains:

- ⊤ (top): “don't know", represents any value
- ⊥ (bottom): represents empty-set

# Step 2: Abstraction function

*Define functions that map sets of concrete states/elements to an element of the abstract domain*

Define abstraction function $\alpha$: FOL $\mapsto D$ to map a FOL formula $F$

to element $\alpha(F)$ of $D$ such that $F \Rightarrow \alpha(F)$

1. $D_I$ : Interval domain

$\bot$

$\top$

conjunctions of $\Sigma_Q$ - literals

$v \le c$ , $v \ge c$

var const

Interval arithmetic , $[l, u]$

2. $\alpha(F)$

literal

$\bot$ — $F : \bot$

$v \le \frac{b}{a}$ — $F: av \le b$

$v \ge \frac{b}{a}$ — $F: av \ge b$

$\left.\begin{array}{l} v \le \frac{b}{a} \\ \wedge \\ v \ge \frac{b}{a} \end{array}\right\}$ $F: av = b$

$\top$ otherwise

$a, b \in \mathbb{N}$, $a > 0$

# Step 3: Abstract transformer (SP)

*Define abstract semantics/transformers for each statement*

Define an abstract strongest postcondition $sp_D$ for assumption and assign statements such that

$$sp(S, P) \Rightarrow sp_D(S, P) \text{ and } sp_D(S, P) \in D$$

# Step 3: Abstract transformer (SP)

Recall: $sp(\mathtt{assume}\ \ C, P) = C \wedge P$

Define abstract conjunction $\sqcap_D$ such that for $F_1, F_2 \in D$

$$F_1 \wedge F_2 \Rightarrow \ \ F_1 \sqcap_D F_2 \text{ and } F_1 \sqcap_D F_2 \in D$$

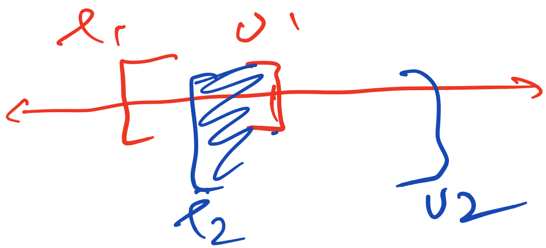
Then for $F \in D$: $sp_D(\mathtt{assume}\ \ C, F) = \alpha(C) \sqcap_D F$

Recall: $sp(x := \ E, P) = \exists x^0. x = E[x^0/x] \wedge p[x^0/x]$

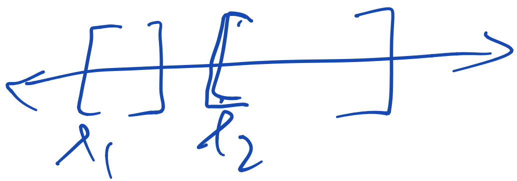
The existential quantification can lead to a quantifier alternation during implication checking and is undesirable

3. $\sqcap_I$ : exact

$$[l_1, u_1] \sqcap_I [l_2, u_2] = \begin{cases} [\infty, -\infty] & \text{if } \max(l_1, l_2) > \min(u_1, u_2) \\ [\max(l_1, l_2), \min(u_1, u_2)] & \text{otherwise} \end{cases}$$




$sp_I(\text{assume } c, F) = \alpha(c) \sqcap_I F$




$sp_I(x := a, F) = ??$

Interval arithmetic

# Step 4: Abstract disjunction

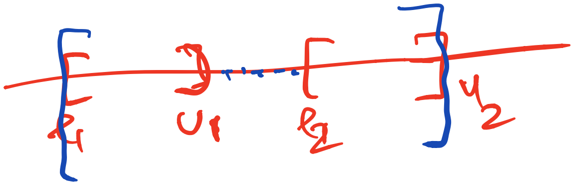
Define abstract disjunction $\sqcup_D$ such that for $F_1, F_2 \in D$

$$F_1 \vee F_2 \Rightarrow \; F_1 \sqcup_D F_2 \text{ and } F_1 \sqcup_D F_2 \in D$$

# Step 5: Abstract implication checking

When selecting $D$ ensure that the implication in the validity check is decidable

4. $\sqcup_I$ : interval hull




$$[l_1, u_1] \sqcup_I [l_2, u_2] = [\min(l_1, l_2), \max(u_1, u_2)]$$

5. $F, G \in D_I$

Validity of $F \Rightarrow G$ is decidable!

$$[l_1, u_1] \subseteq [l_2, u_2]$$

# Step 6: Widening

$F_1, \quad F_2 \equiv F_1 \vee H_2, \quad F_3 \equiv F_2 \vee H_3, \dots$

$G_1, \quad G_2 \equiv G_1 \nabla_D F_2, \quad G_3 \equiv G_2 \nabla_D F_3$

*Defining an abstraction does not suffice to guarantee termination. Some abstract domains need to be equipped with a widening operator.*

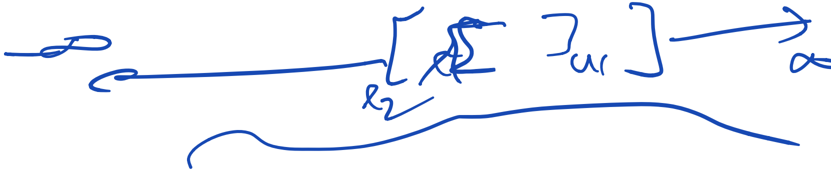
Define a widening operator $\nabla_D : D \times D \mapsto D$ such that

1. for all $F_1, F_2 \in D$, $F_1 \vee F_2 \Rightarrow F_1 \nabla_D F_2$ and
2. for all infinite sequences $F_1, F_2, \dots$ with $F_1 \Rightarrow F_2 \Rightarrow \cdots$,
   the infinite sequence $G_1 = F_1, \dots, G_{i+1} = G_i \nabla_D F_{i+1}, \dots$ converges.

Thus, the sequence $G_i$ converges even if the sequence $F_i$ does not.

6. $F: [\ell_1, u_1]$

$G \quad [\ell_2, u_2]$



$$F \nabla_I G = [\ell, u] \supseteq [\ell_1, u_1] \cup [\ell_2, u_2]$$

$$\ell = \begin{cases} -\infty & \text{, if } \ell_2 < \ell_1 \\ \ell_1 & \text{, otherwise} \end{cases}$$

$$u = \begin{cases} \infty & \text{, if } u_2 > u_1 \\ u_1 & \text{, otherwise.} \end{cases}$$

$F \nabla_I G$ "drops" bounds that grow from $F$ to $G$

```
AbstractForwardPropagate(pre, ℒ)
  W := {L_init}
  μ(L_init) := α(pre)
  foreach L ∈ ℒ\{L_init} do μ(L) := ⊥
  while W ≠ φ do
    pick L_j ∈ W; W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp_D(S_j; …; S_k, μ(L_j)) ⇒ μ(L_k))
      then if WIDEN()
        then μ(L_k) := μ(L_k) ∇_D (μ(L_k) ⊔_D sp_D(S_j; …; S_k, μ(L_j)))
        else μ(L_k) := μ(L_k) ⊔_D sp(S_j; …; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

```
AbstractForwardPropagate(pre, ℒ)
  W := {L_init}
  μ(L_init) := α(pre)
  foreach L ∈ ℒ\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W; W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp_D(S_j; …; S_k, μ(L_j)) ⇒ μ(L_k))
      then if WIDEN()
        then μ(L_k) := μ(L_k) ∇_D (μ(L_k) ⊔_D sp_D(S_j; …; S_k, μ(L_j)))
        else μ(L_k) := μ(L_k) ⊔_D sp(S_j; …; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

```
AbstractForwardPropagate(pre, ℒ)
  W := {L_init}
  μ(L_init) := α(pre)
  foreach L ∈ ℒ\{L_init} do μ(L) := ⊥
  while W ≠ ϕ do
    pick L_j ∈ W; W := W\{L_j}
    foreach L_k ∈ succ(L_j) do
      if ¬(sp_D(S_j; …; S_k, μ(L_j)) ⇒ μ(L_k))
      then if WIDEN()
        then μ(L_k) := μ(L_k) ∇_D (μ(L_k) ⊔_D sp_D(S_j; …; S_k, μ(L_j)))
        else μ(L_k) := μ(L_k) ⊔_D sp(S_j; …; S_k, μ(L_j))
        W := W ∪ {L_k}
  return μ
```

WIDEN() determines if widening should be applied

```
AbstractForwardPropagate(pre, ℒ)
   W := {L_init}
   μ(L_init) := α(pre)
   foreach L ∈ ℒ\{L_init} do μ(L) := ⊥
   while W ≠ ϕ do
      pick L_j ∈ W; W := W\{L_j}
      foreach L_k ∈ succ(L_j) do
         if ¬(sp_D(S_j; …; S_k, μ(L_j)) ⇒ μ(L_k))
         then if WIDEN()
            then μ(L_k) := μ(L_k) ∇_D (μ(L_k) ⊔_D sp_D(S_j; …; S_k, μ(L_j)))
            else μ(L_k) := μ(L_k) ⊔_D sp(S_j; …; S_k, μ(L_j))
            W := W ∪ {L_k}
   return μ
```

WIDEN() determines if widening should be applied

Least Fixpoint Computation under abstract domain

# Invariant generation

- Today: least fixed-point in an abstract domain
- Farkas' lemma (for linear invariants) [Colon_CAV_2003]
- Interpolation [McMillan_TACAS_2008]
- Abductive inference [Dillig_OOPSLA_2013]
- IC3 [Bradley_VMCAI_2011]
- Machine learning [Garg_CAV_2014]

# Abstraction Interpretation

Patrick Cousot

Radhia Cousot

Cousot & Cousot, *Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints*, 1977

A framework for designing sound-by-construction static analyses

# Lattices and Abstract Domains

Abstraction function $\alpha$: FOL $\mapsto D$ maps a FOL formula $F$ to element $\alpha(F)$ of $D$ such that $F \Rightarrow \alpha(F)$

Concretization function $\gamma$: $D \mapsto$ FOL to map an element of $D$ to a FOL formula $F$

# Lattices and Abstract Domains

Abstraction function $\alpha$: FOL $\mapsto D$ maps a FOL formula $F$ to element $\alpha(F)$ of $D$ such that $F \Rightarrow \alpha(F)$

Concretization function $\gamma$: $D \mapsto$ FOL to map an element of $D$ to a FOL formula $F$

- Concretization function defines partial order $\preccurlyeq$ on abstract values:
$$\mathrm{A}_1 \preccurlyeq \mathrm{A}_2 \text{ iff } \gamma(\mathrm{A}_1) \Rightarrow \gamma(\mathrm{A}_2)$$
- In an abstract domain $D$ , every pair of elements has a lub ($\sqcup_D$) and glb ($\sqcap_D$)

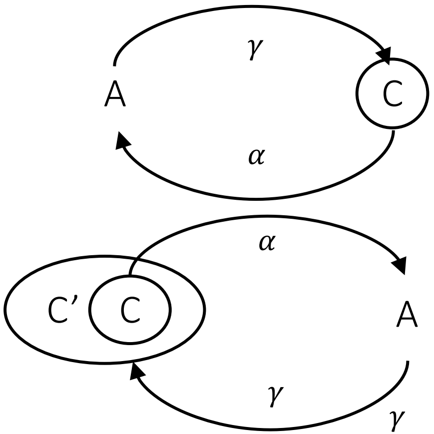
$\Rightarrow$ Mathematical Lattice

# Galois insertion

Important property of the abstraction and concretization function is that they are almost inverses:

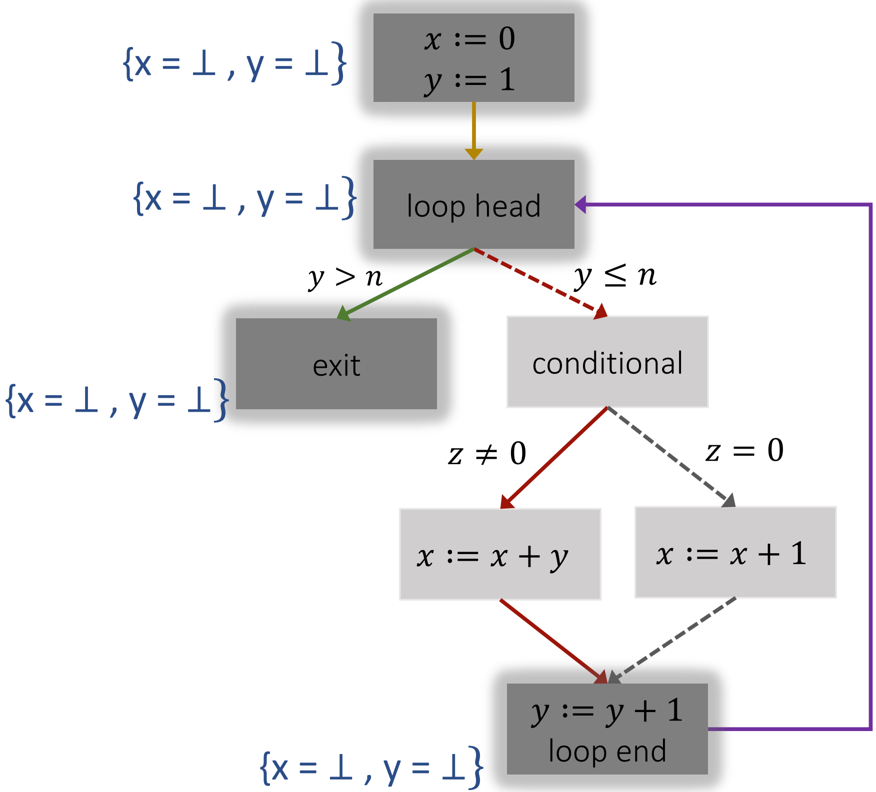
$$\alpha\big(\gamma(A)\big) = A$$

$$C \subseteq \gamma(\alpha(C))$$

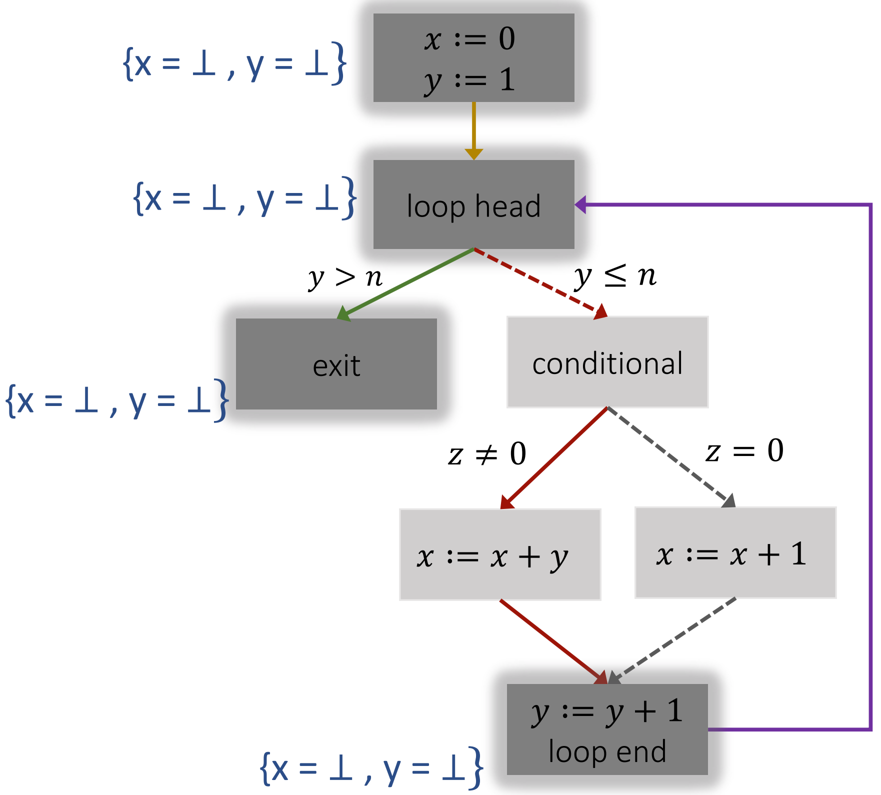



Captures soundness of the abstraction

```
{pre}
x := 0
y := 1
while {I} (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    y := y + 1
{post}
```



Let's do a static analysis of this program under the "sign" abstract domain



# Sign domain

1. $D$ : $\top$

$\bot$

P - pos : $\{x : x \in \mathbb{Z} \wedge x > 0\}$ -

N neg : $\{x : x \in \mathbb{Z} \wedge x < 0\}$

Z - zero : $\{0\}$

NN non-neg : $\{x : x \in \mathbb{Z} \wedge x \geq 0\}$

2. $\alpha(\{2, 5, 0\}) = NN$

$\alpha(\{2, 5\}) = P$

$\alpha(\{-2, 2\}) = \top$

3.

| $+$ | P | N | Z | NN | $\top$ | $\bot$ |
|---|---|---|---|---|---|---|
| P | P | $\top$ | P | | | |
| N | | | | | | |
| Z | | | | | | |
| NN | | | | | | |
| $\top$ | | | | | | |
| $\bot$ | | | | | | |

4. $\sqcup \equiv \cup$

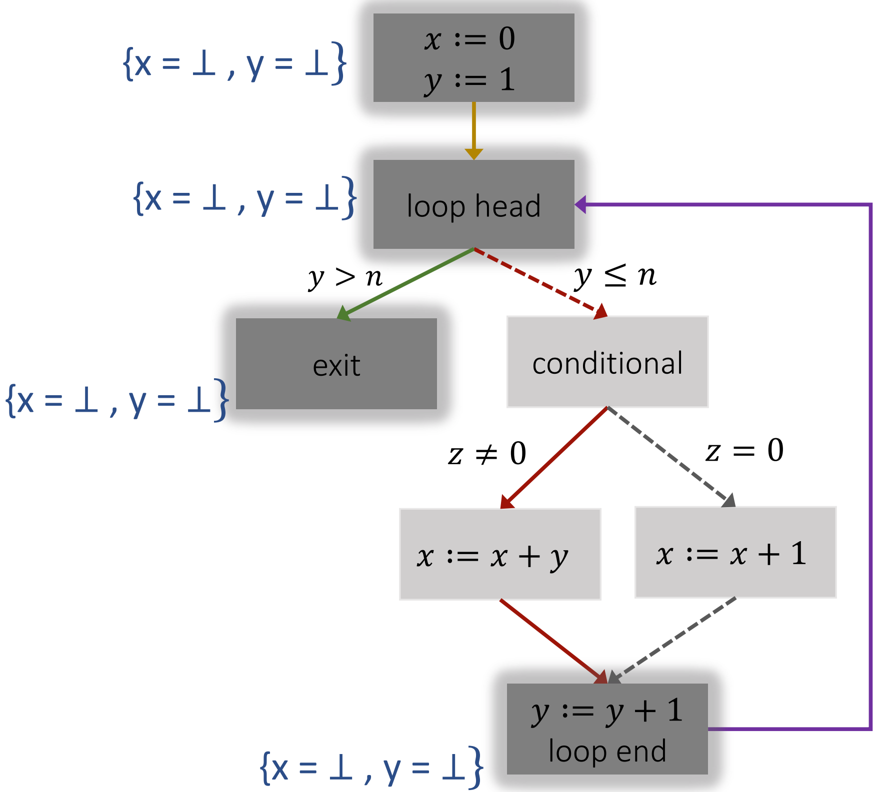
5. $\sqcap$ $\cap$

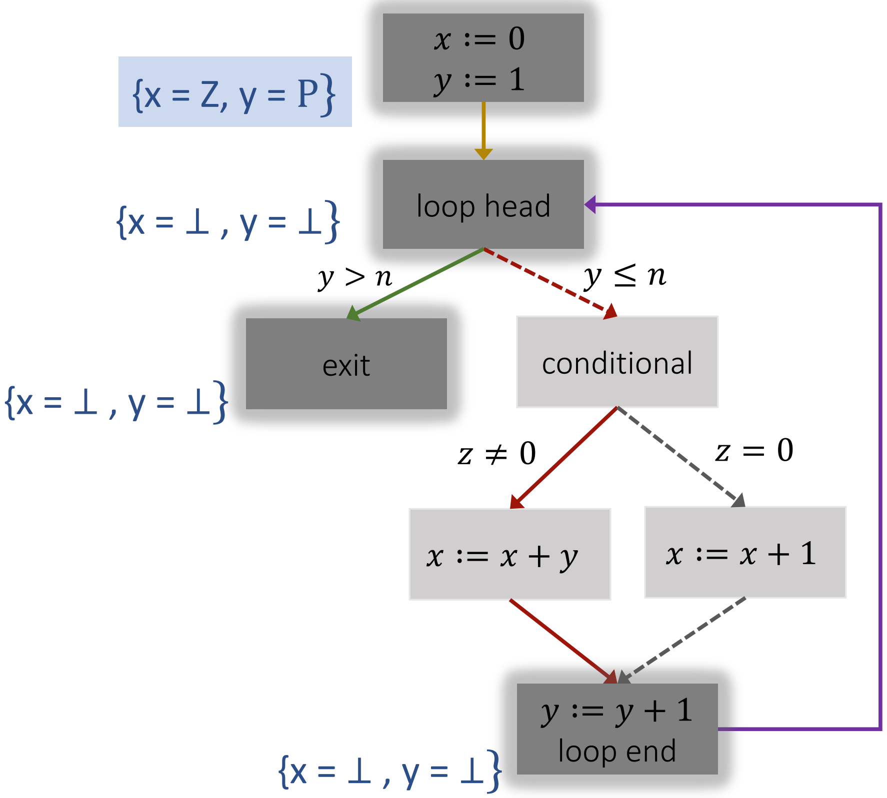
Let's do a static analysis of this program under the "sign" abstract domain

Here `pre` and `post` are not given.

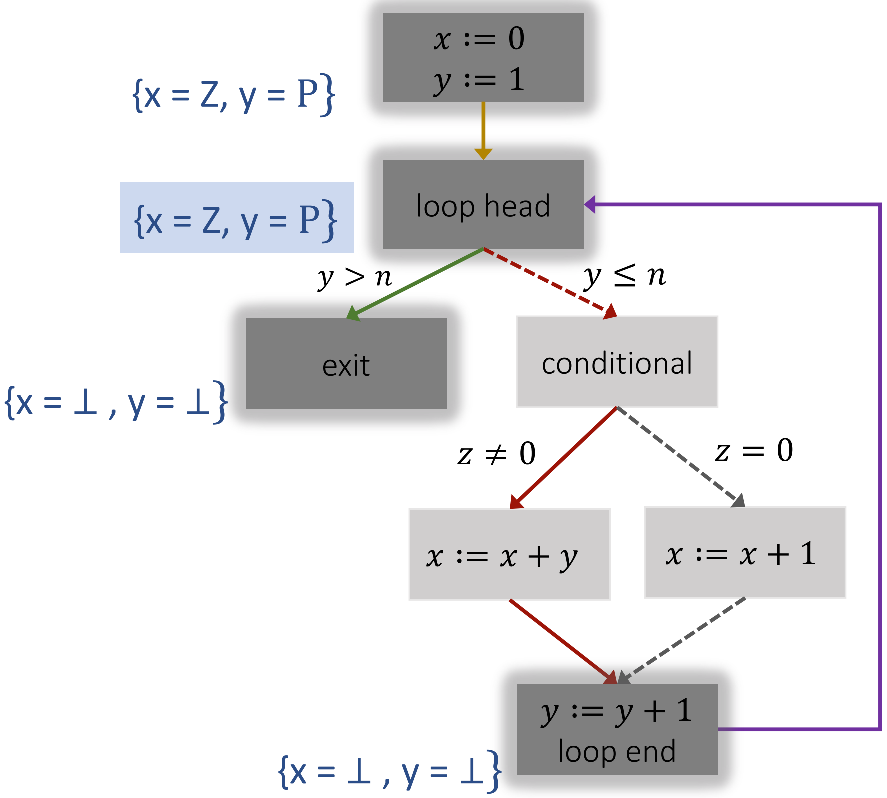
Assume `pre` is **`true.`** This analysis will find an overapproximation of reachable states at all cut-points.

One can then easily verify if some given `post` holds.

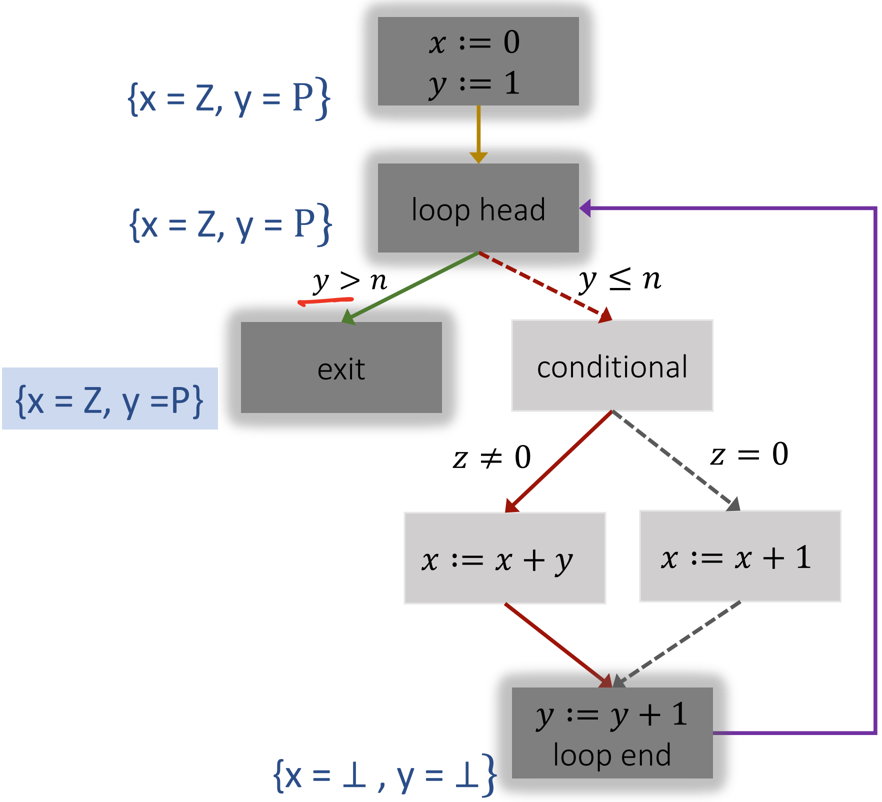




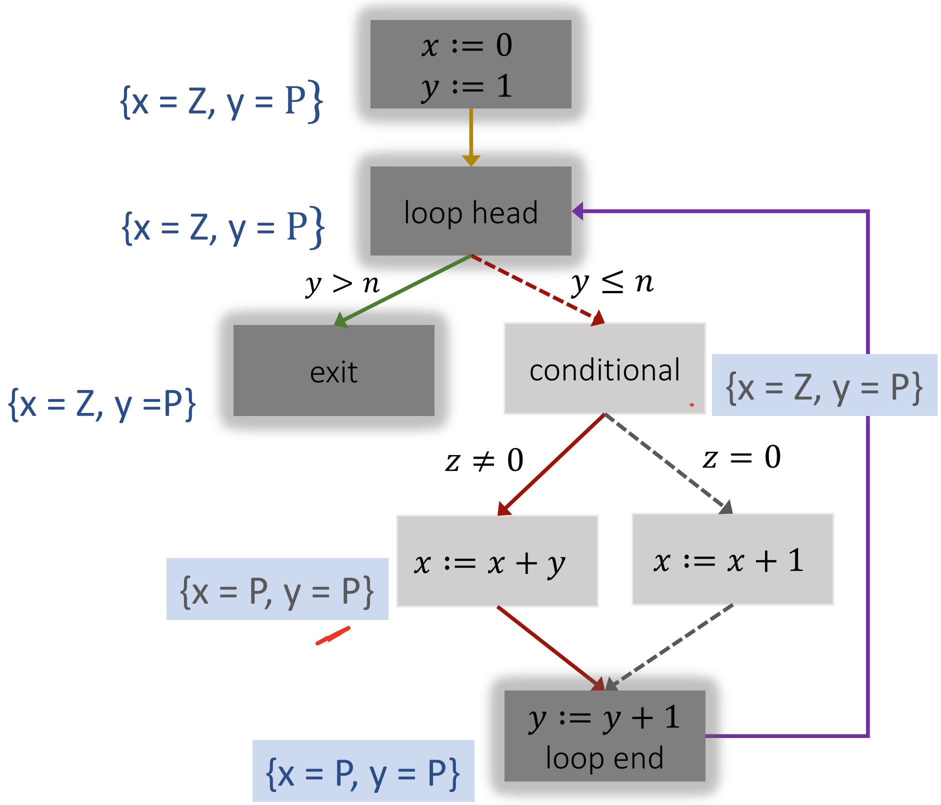
$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = ⊥ , y = ⊥}
$y > n$
$y \leq n$
exit
{x = ⊥ , y = ⊥}
conditional
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
$y := y + 1$
loop end
{x = ⊥ , y = ⊥}

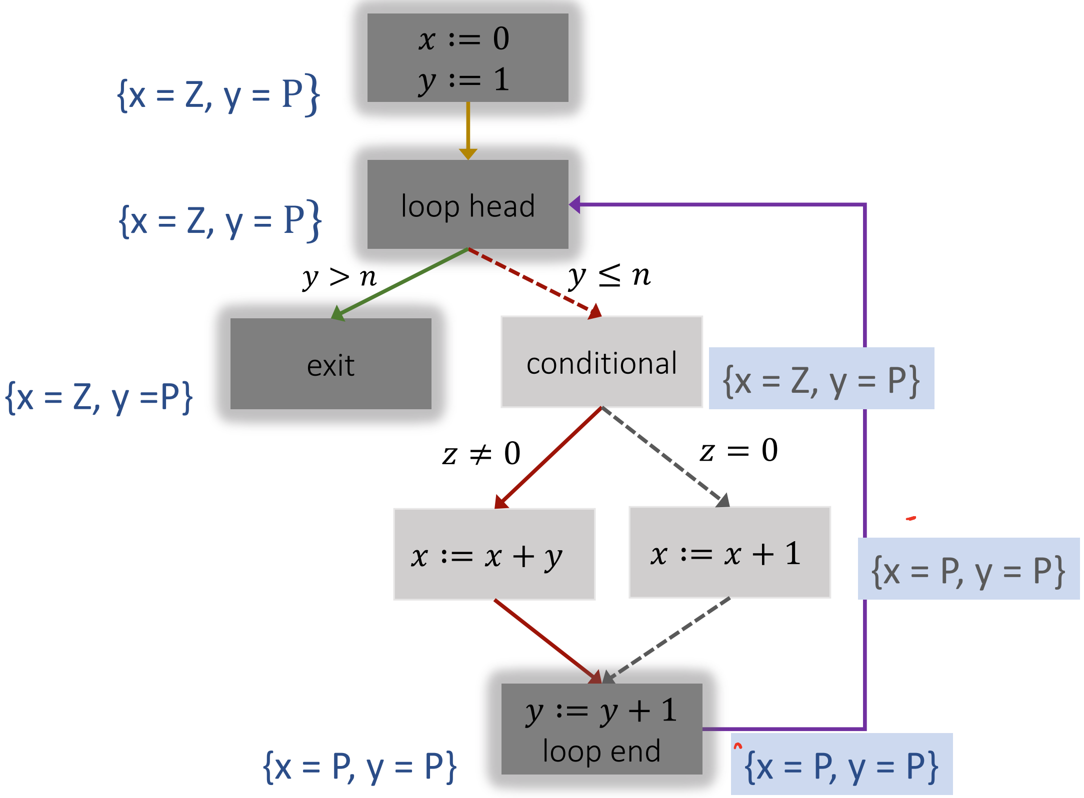


{x = Z, y = P}
$x := 0$
$y := 1$
loop head
{x = Z, y = P}
$y > n$
$y \leq n$
exit
conditional
{x = ⊥ , y = ⊥}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
$y := y + 1$
loop end
{x = ⊥ , y = ⊥}

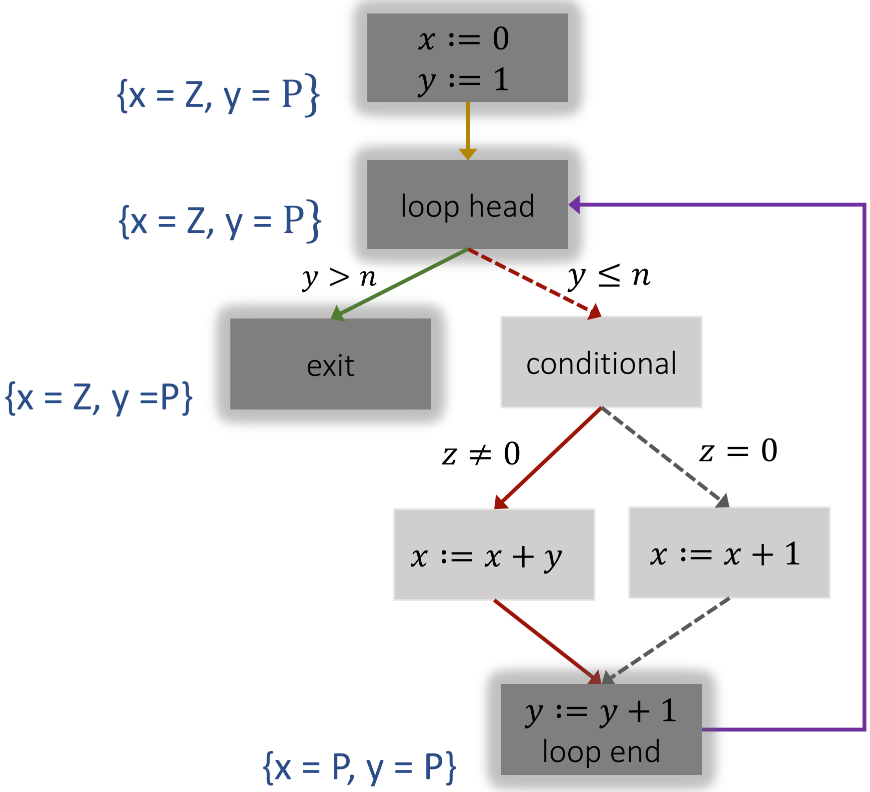


$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = Z, y = P}
$y > n$
$y \leq n$
exit
{x = Z, y =P}
conditional
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
$y := y + 1$
loop end
{x = ⊥ , y = ⊥}

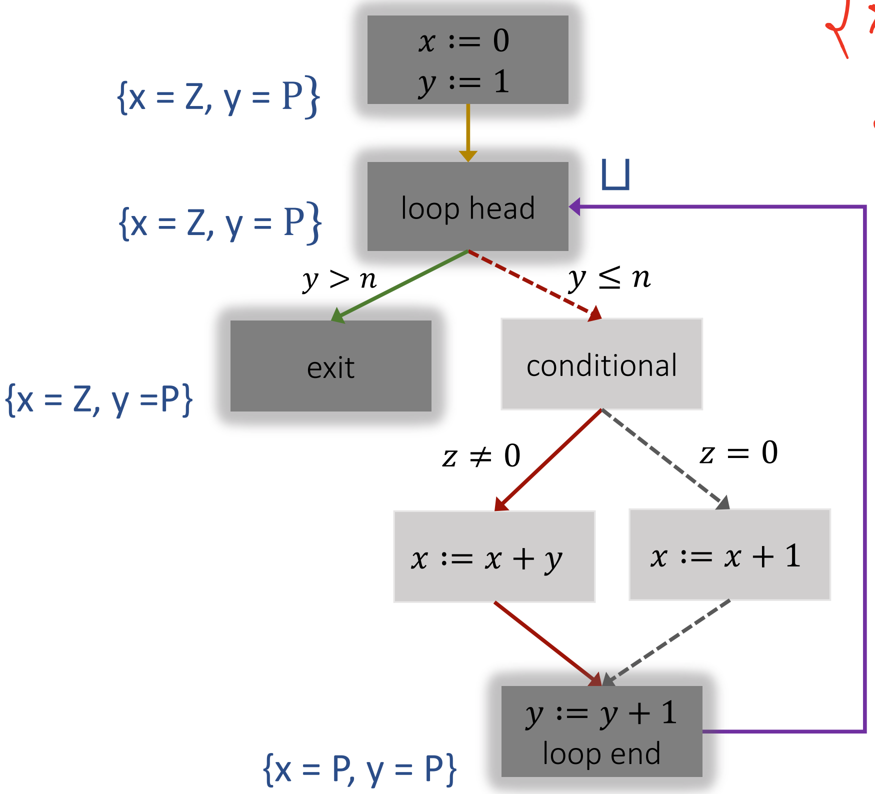

$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = Z, y = P}
$y > n$
$y \leq n$
exit
{x = Z, y =P}
conditional
{x = Z, y = P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}



$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = Z, y = P}
$y > n$
$y \leq n$
exit
{x = Z, y =P}
conditional
{x = Z, y = P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}
{x = P, y = P}

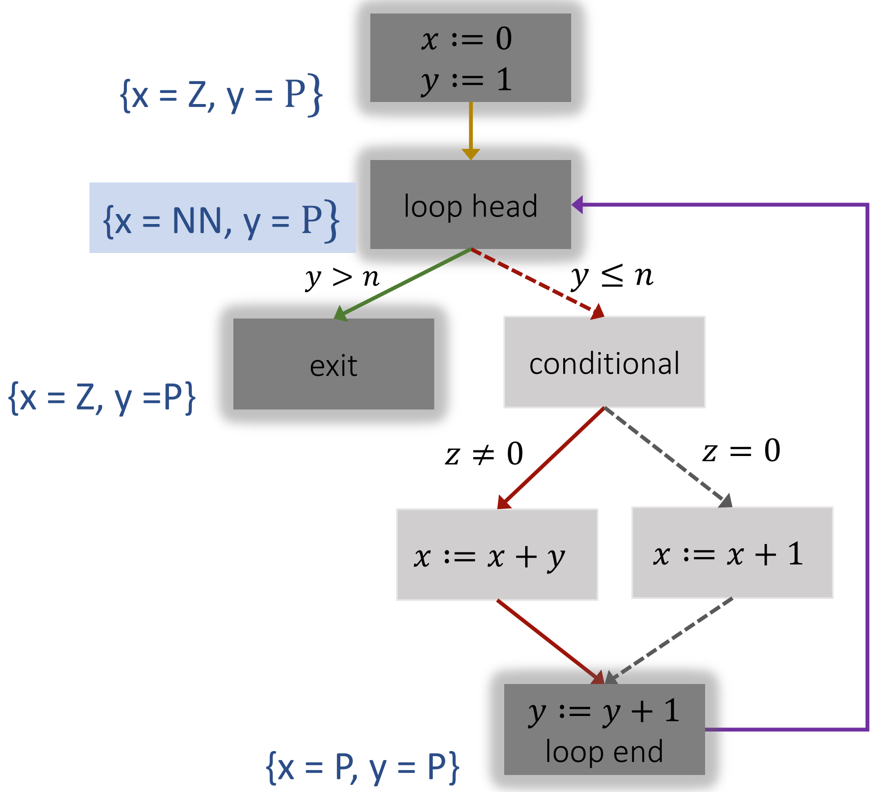


$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = Z, y = P}
$y > n$
$y \leq n$
exit
{x = Z, y =P}
conditional
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
$y := y + 1$
loop end
{x = P, y = P}

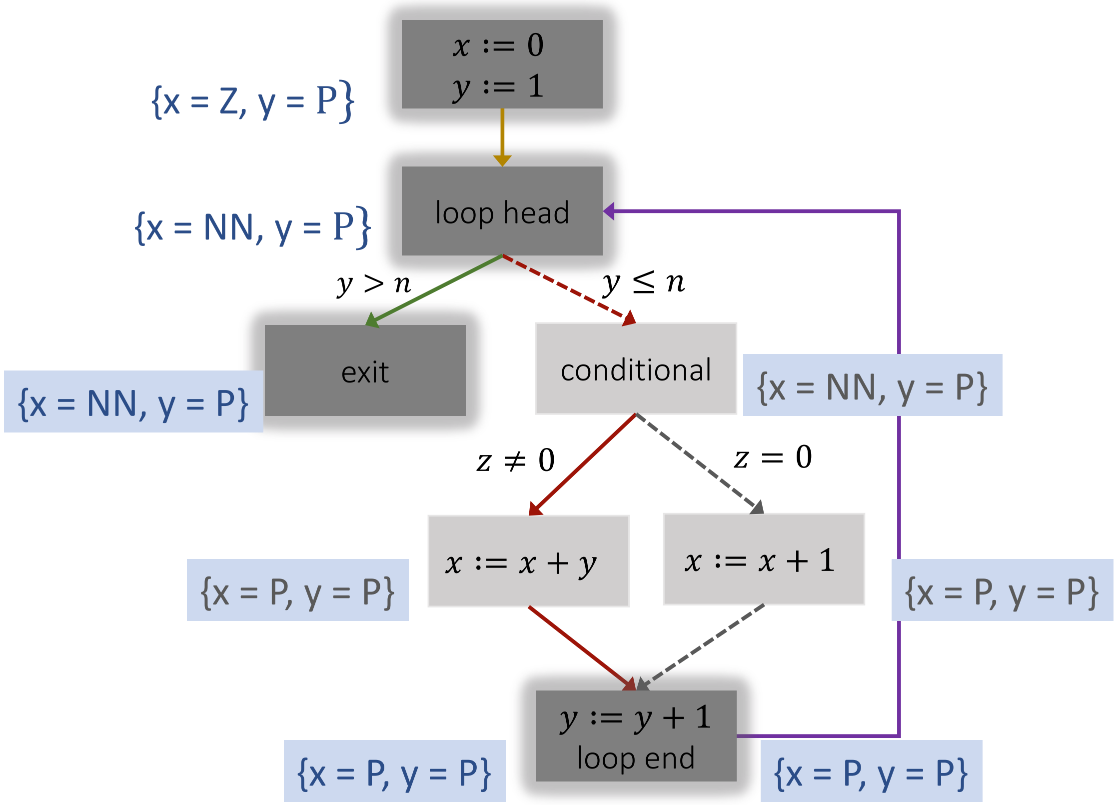



{x = Z, y = P}
$\sqcup$
{x = P, y = P}

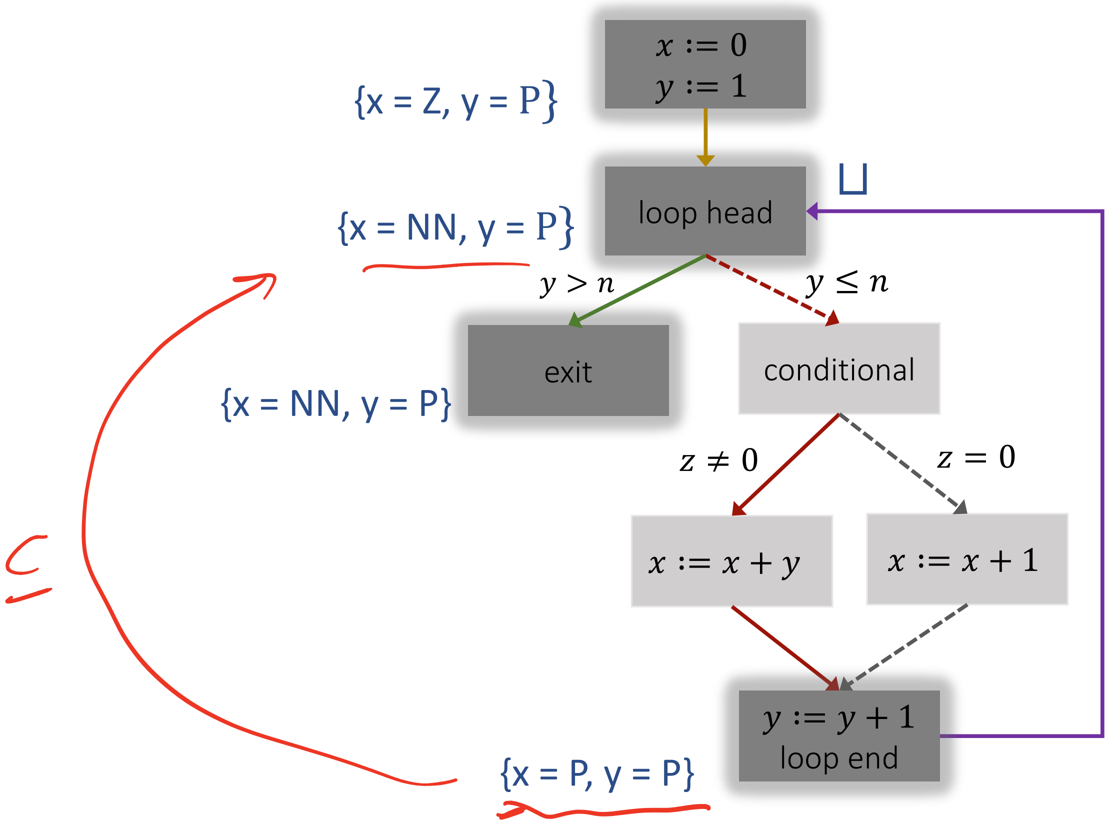
{x = NN, y = P}



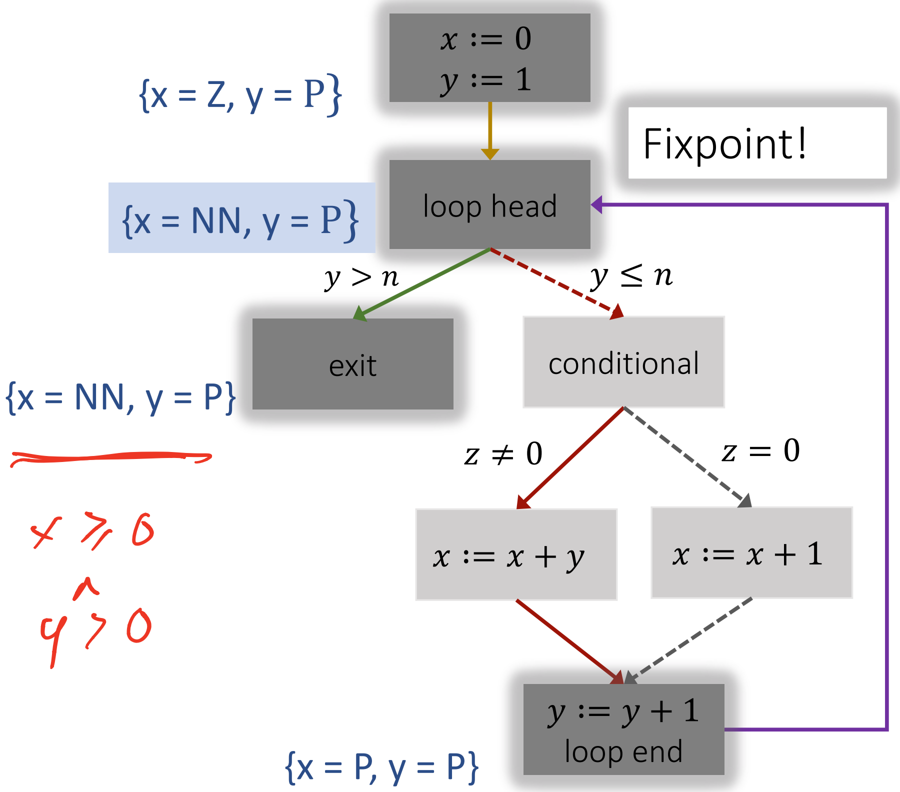
$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = NN, y = P}
$y > n$
$y \leq n$
exit
conditional
{x = Z, y =P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
$y := y + 1$
loop end
{x = P, y = P}



$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = NN, y = P}
$y > n$
$y \leq n$
exit
{x = NN, y = P}
conditional
{x = NN, y = P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
{x = P, y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}
{x = P, y = P}



$x := 0$
$y := 1$
{x = Z, y = P}
loop head
$\sqcup$
{x = NN, y = P}
$y > n$
$y \le n$
exit
conditional
{x = NN, y = P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
$\subseteq$
$y := y + 1$
loop end
{x = P, y = P}




$x \geq 0$

$\wedge$

$y > 0$

# Partial specifications

- Pre/postcondition pairs are also called functional specifications
  - Meant to be complete
- Writing complete specifications can be very hard
- Partial specifications are easier to write and, often, easier to check
- Partial specifications may be explicit or implicit

# Partial specifications

- Pre/postcondition pairs are also called functional specifications
  - Meant to be complete
- Writing complete specifications can be very hard
- Partial specifications are easier to write and, often, easier to check
- Partial specifications may be explicit or implicit

assertions

no division-by-zero
no null-pointer-dereference

```
x := 0
y := 1
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    assert y > 0
    y := y + 1
```

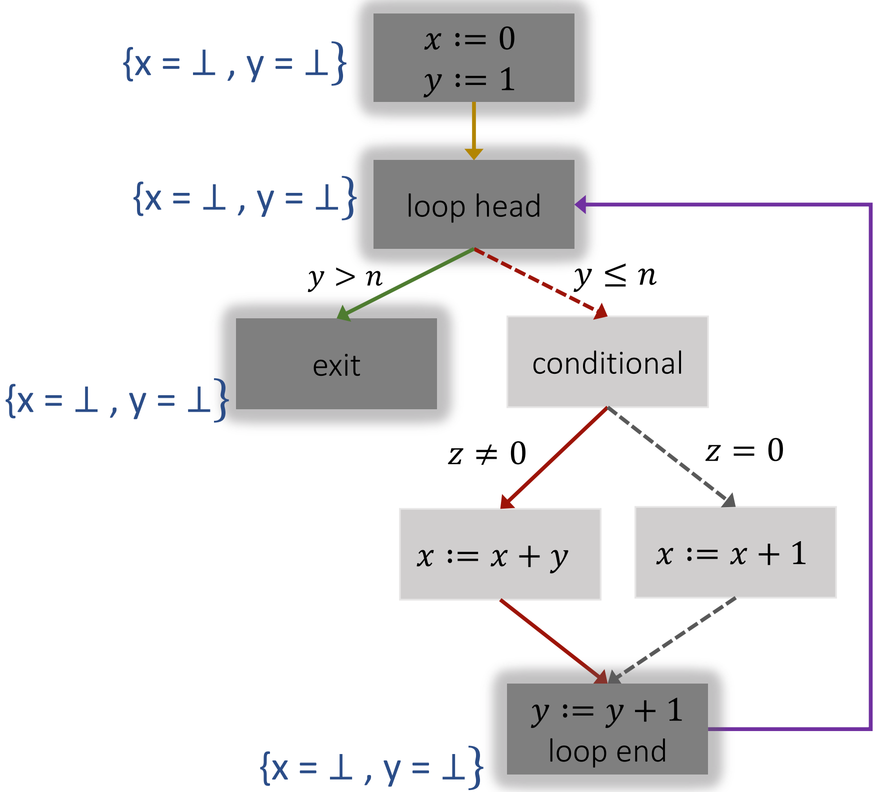
An assertion is a FOL formula $F$ at a program location $L$. It asserts that $F$ is true whenever program control reaches $L$. Equivalently, it asserts that the set of reachable states at $L$ satisfies $F$.

```
x := 0
y := 1
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    assert y > 0
    y := y + 1
```

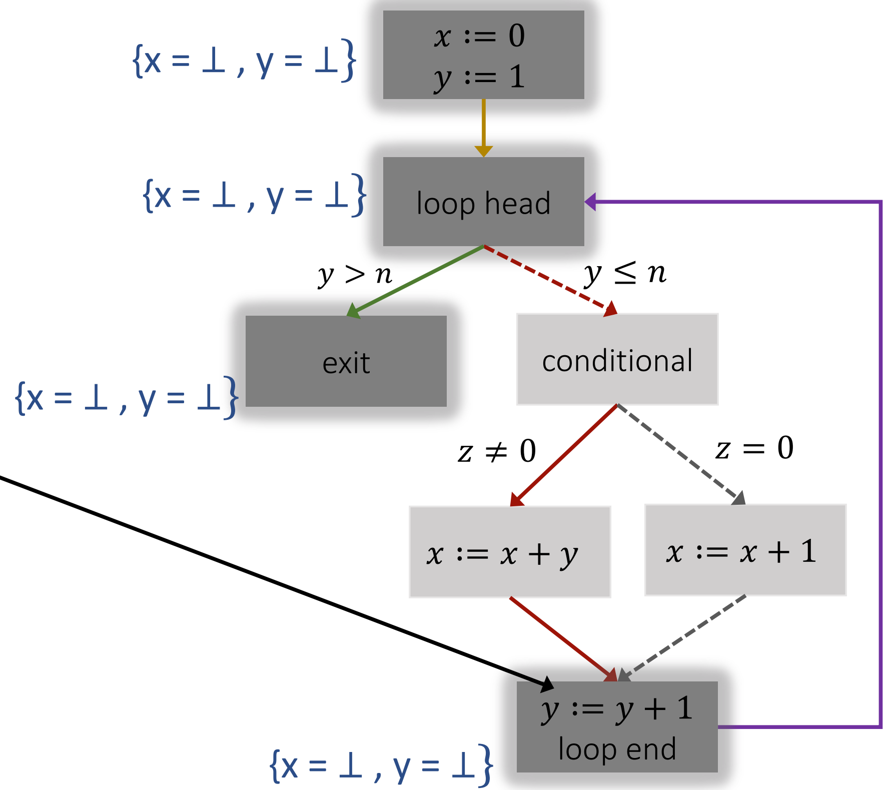
An assertion is a FOL formula $F$ at a program location $L$. It asserts that $F$ is true whenever program control reaches $L$. Equivalently, it asserts that the set of reachable states at $L$ satisfies $F$.

Can use forward propagation to also compute the exact/overapproximate set of reachable states at location $L$ (even if $L$ is not a cutpoint).

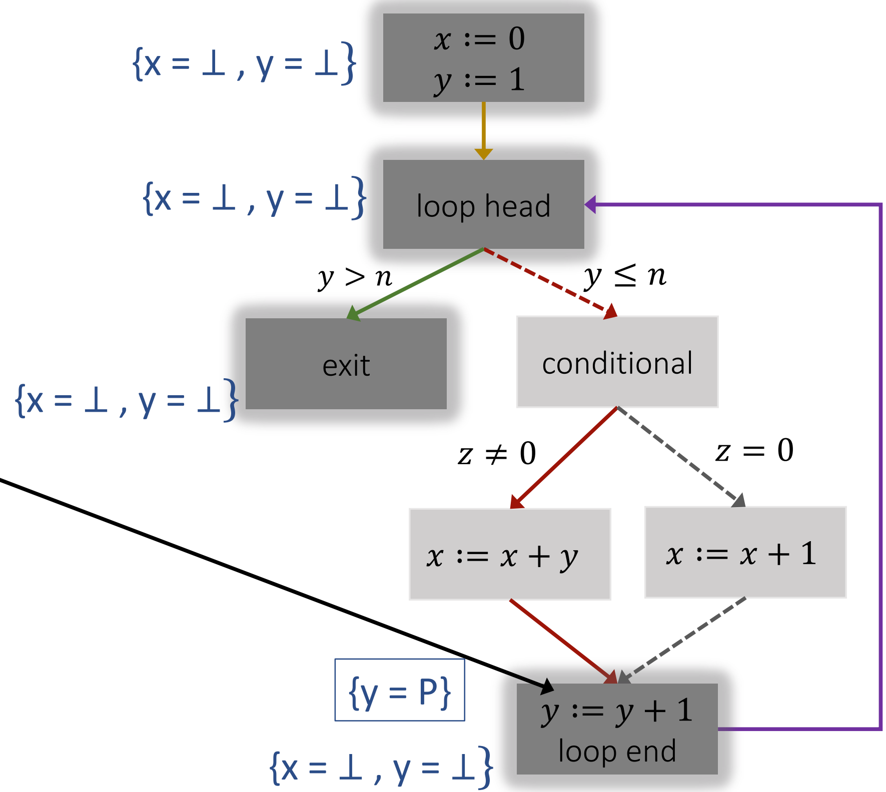
```
x := 0
y := 1
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
  assert y > 0
    y := y + 1
```

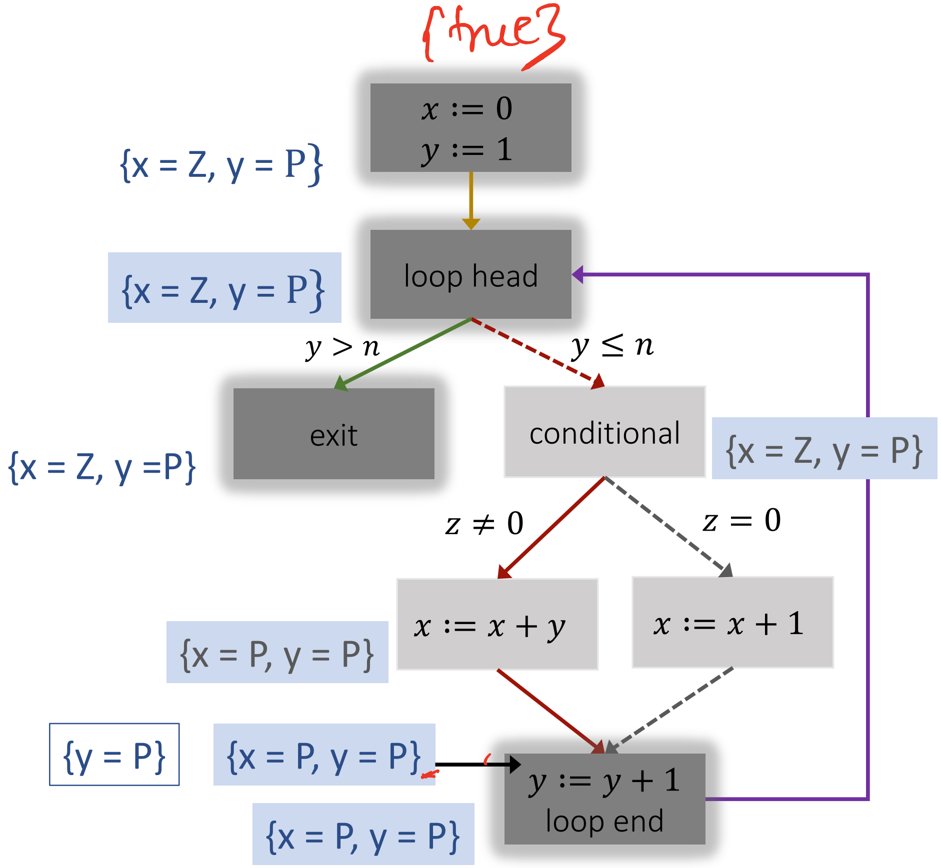


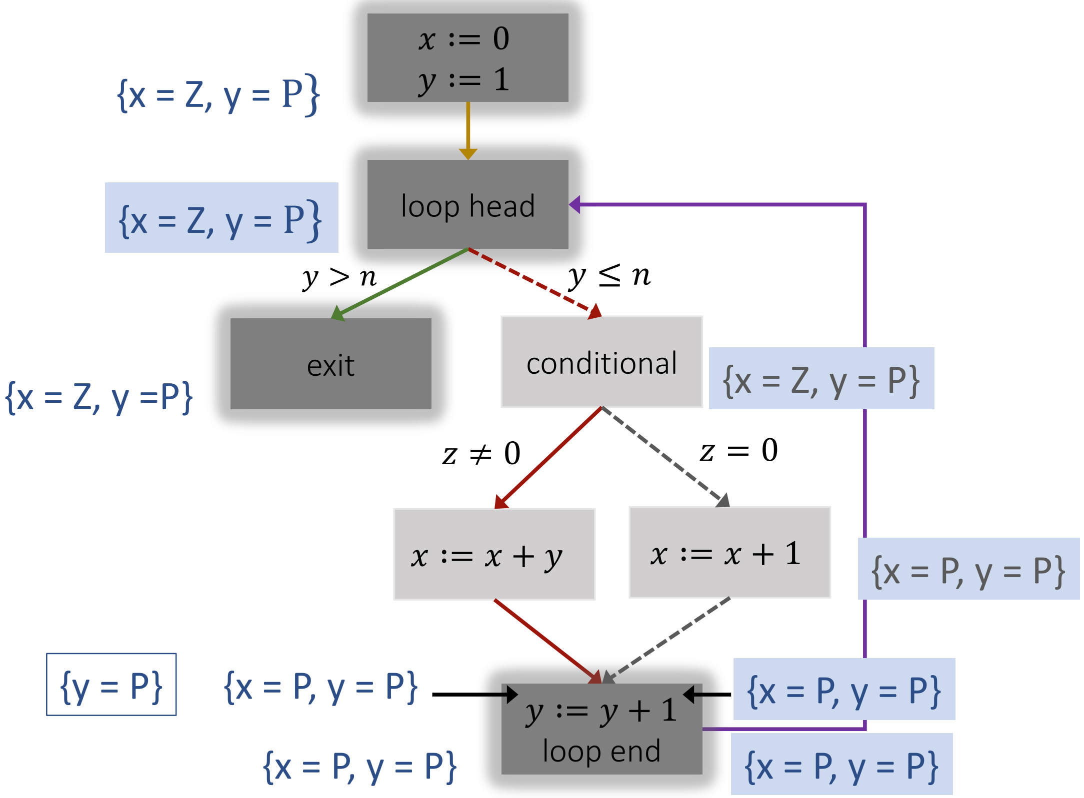
```
x := 0
y := 1
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    assert y > 0
    y := y + 1
```



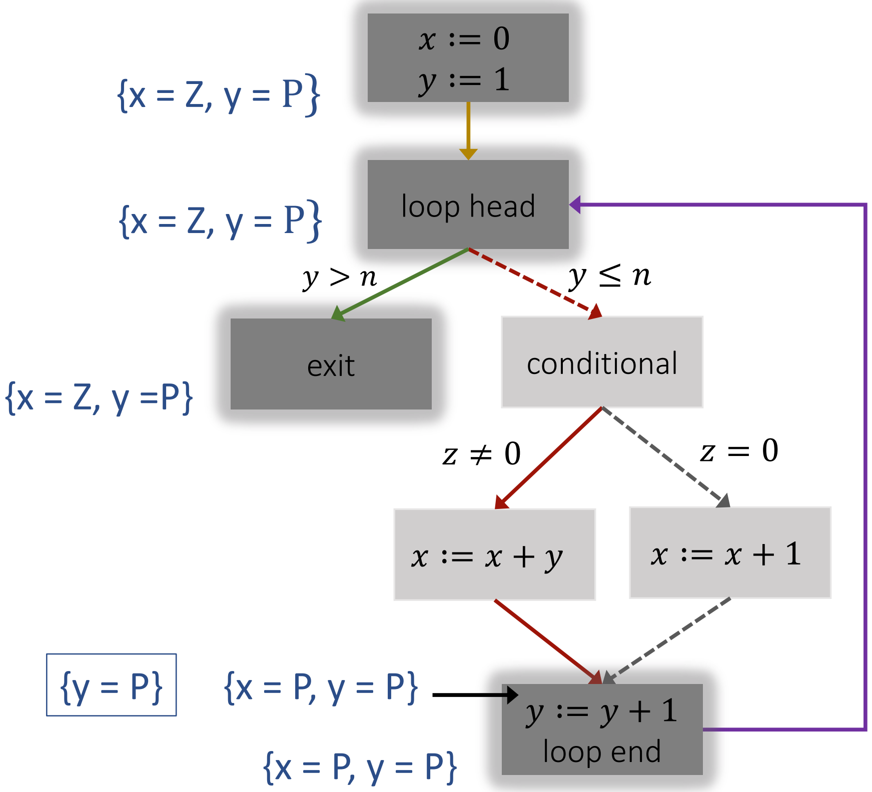
```
x := 0
y := 1
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
  assert y > 0
    y := y + 1
```



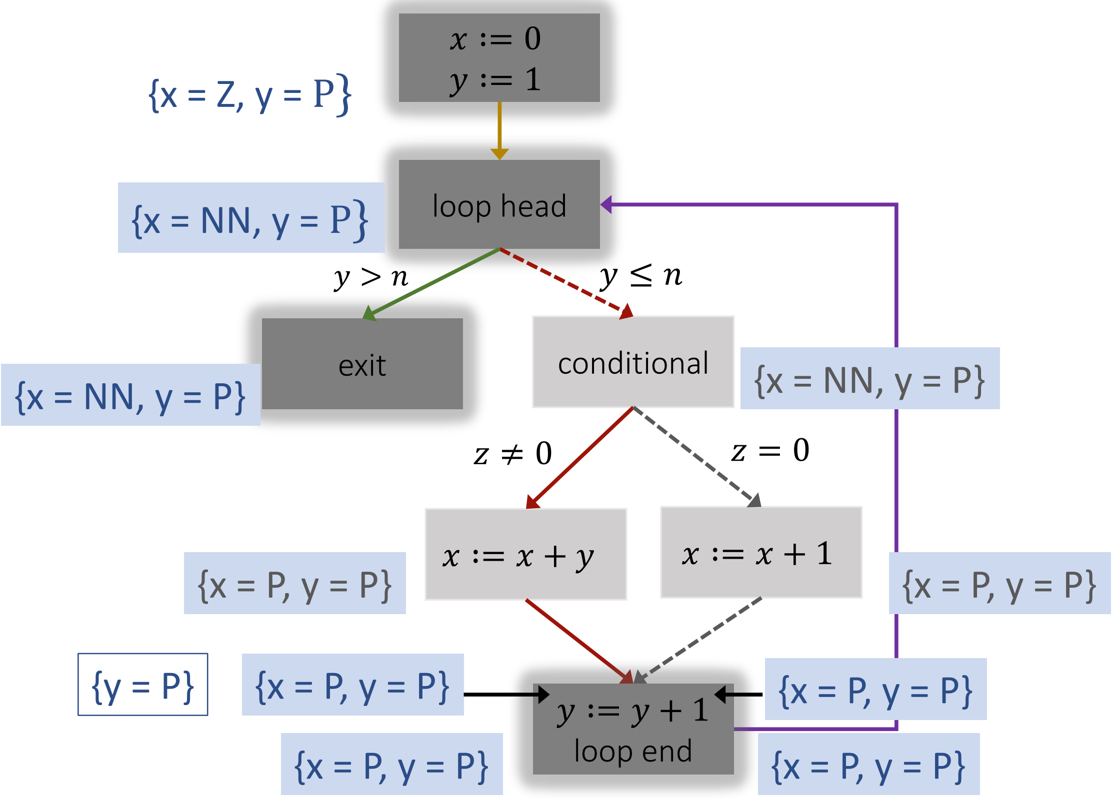

{true}
x := 0
y := 1
{x = Z, y = P}
loop head
{x = Z, y = P}
y > n
y ≤ n
exit
{x = Z, y =P}
conditional
{x = Z, y = P}
z ≠ 0
z = 0
x := x + y
x := x + 1
{x = P, y = P}
{y = P}
{x = P, y = P}
y := y + 1
loop end
{x = P, y = P}

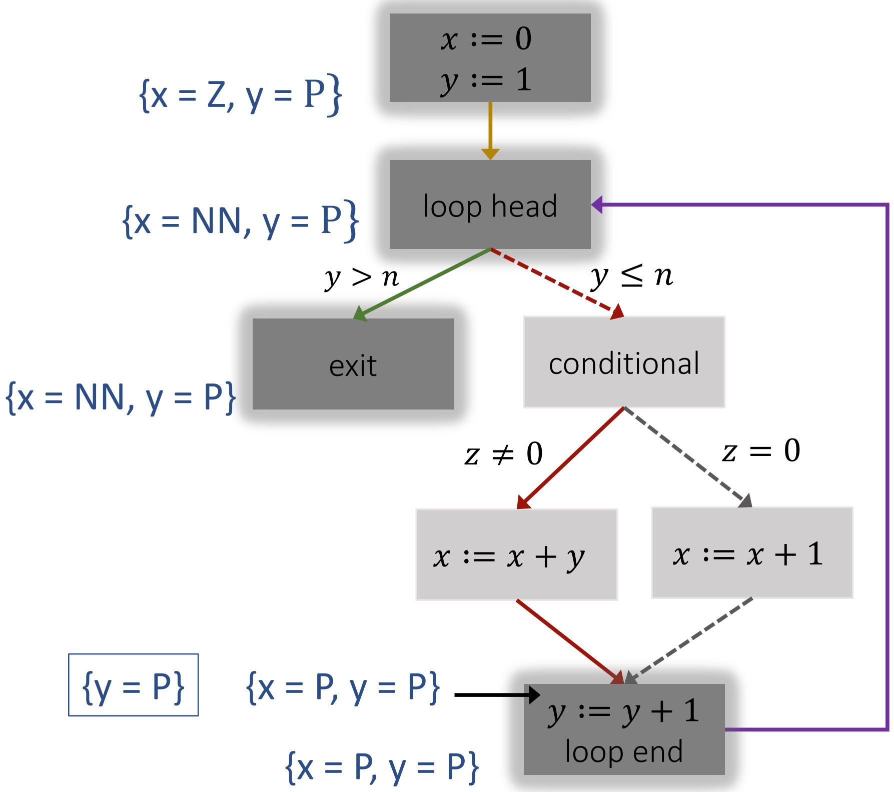


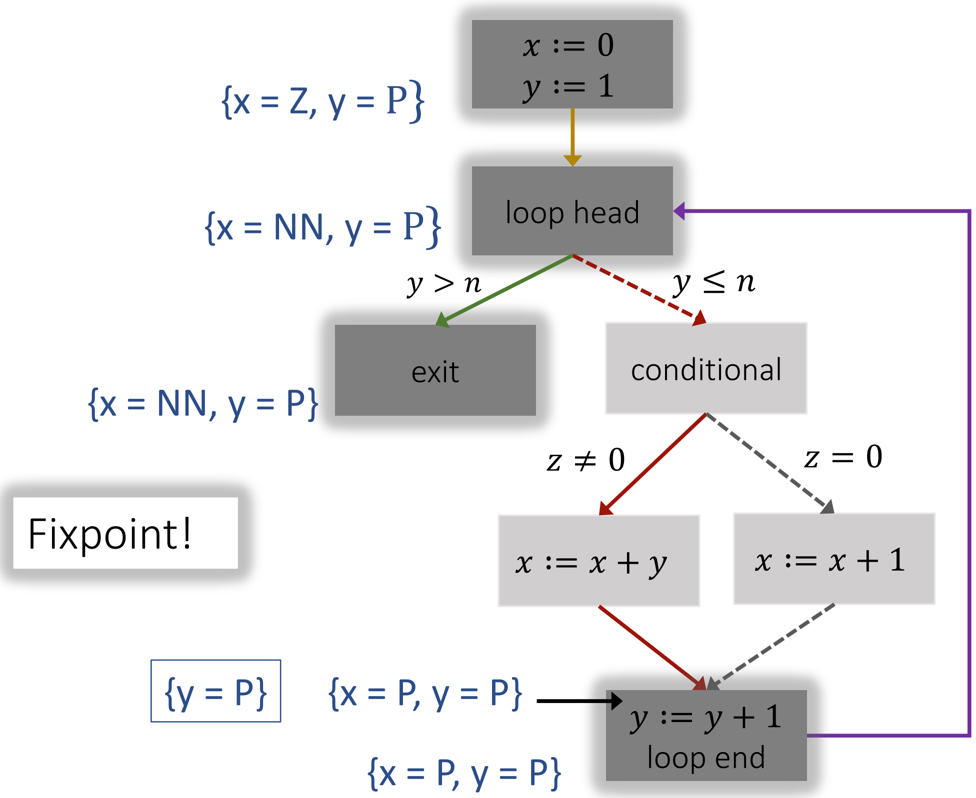
$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = Z, y = P}
$y > n$
$y \leq n$
exit
{x = Z, y =P}
conditional
{x = Z, y = P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
{x = P, y = P}
{y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}
{x = P, y = P}
{x = P, y = P}



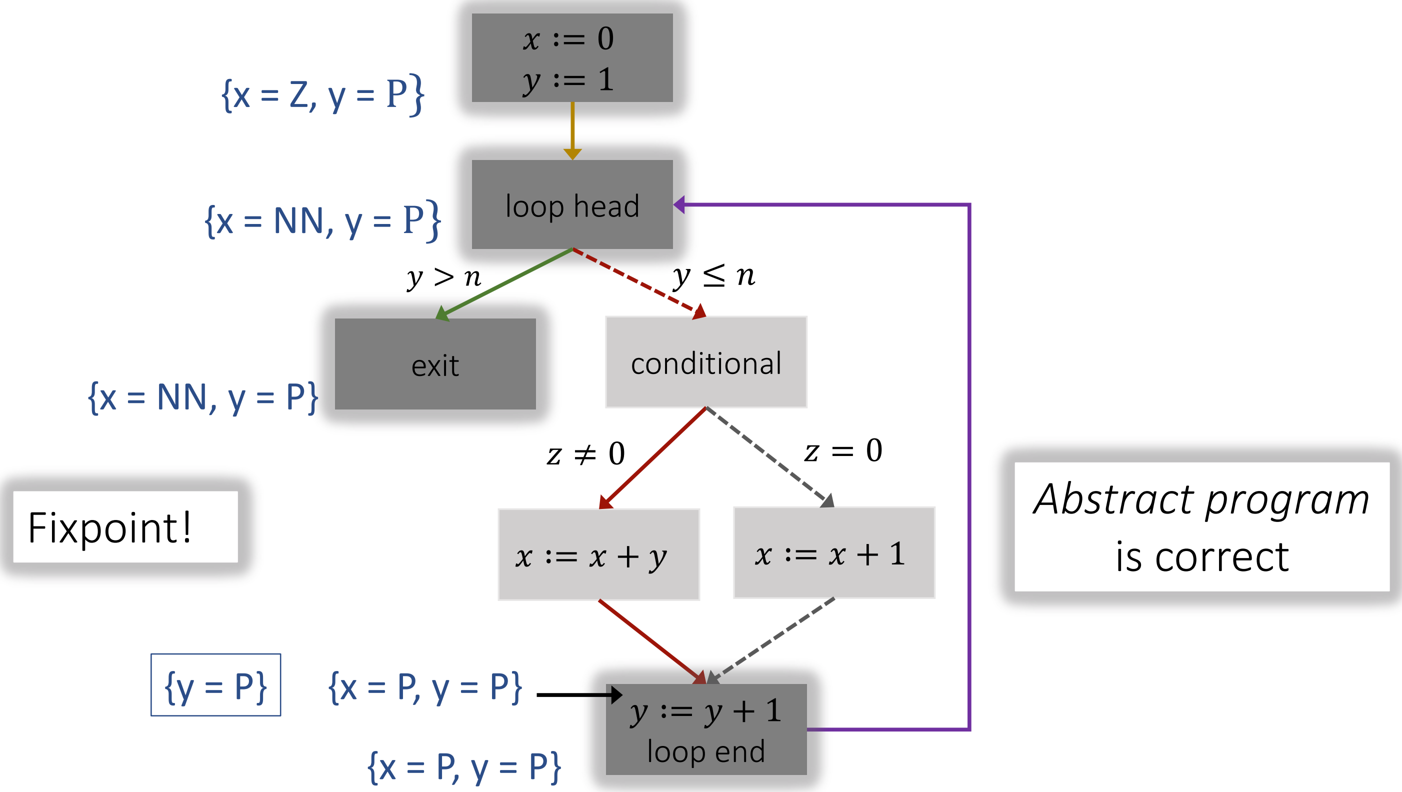
$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = Z, y = P}
$y > n$
$y \leq n$
exit
conditional
{x = Z, y =P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
{y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}


$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = NN, y = P}
$y > n$
$y \leq n$
exit
{x = NN, y = P}
conditional
{x = NN, y = P}
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
{x = P, y = P}
{x = P, y = P}
{y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}
{x = P, y = P}
{x = P, y = P}


$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = NN, y = P}
$y > n$
$y \leq n$
exit
{x = NN, y = P}
conditional
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
{y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}



$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = NN, y = P}
$y > n$
$y \leq n$
exit
conditional
{x = NN, y = P}
$z \neq 0$
$z = 0$
Fixpoint!
$x := x + y$
$x := x + 1$
{y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}



$x := 0$
$y := 1$
{x = Z, y = P}
loop head
{x = NN, y = P}
$y > n$
$y \leq n$
exit
{x = NN, y = P}
conditional
$z \neq 0$
$z = 0$
$x := x + y$
$x := x + 1$
Fixpoint!
{y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}
*Abstract program*
is correct

```
x := 0
y := 1
while (y <= n) do
   if (z == 0) then
      x := x + 1
   else
      x := x + y
   assert y > 0
   y := y + 1
```

✓

```
x := 0
y := 1
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    y := y + 1
```

Always $y > 0$

$$
\begin{array}{l}
x := 0 \\
y := 1 \\
\texttt{while}\ (y <= n)\ \texttt{do} \\
\quad \texttt{if}\ (z == 0)\ \texttt{then} \\
\quad\quad x := x + 1 \\
\quad \texttt{else} \\
\quad\quad x := x + y \\
\quad y := y + 1
\end{array}
$$

Always $y > 0$

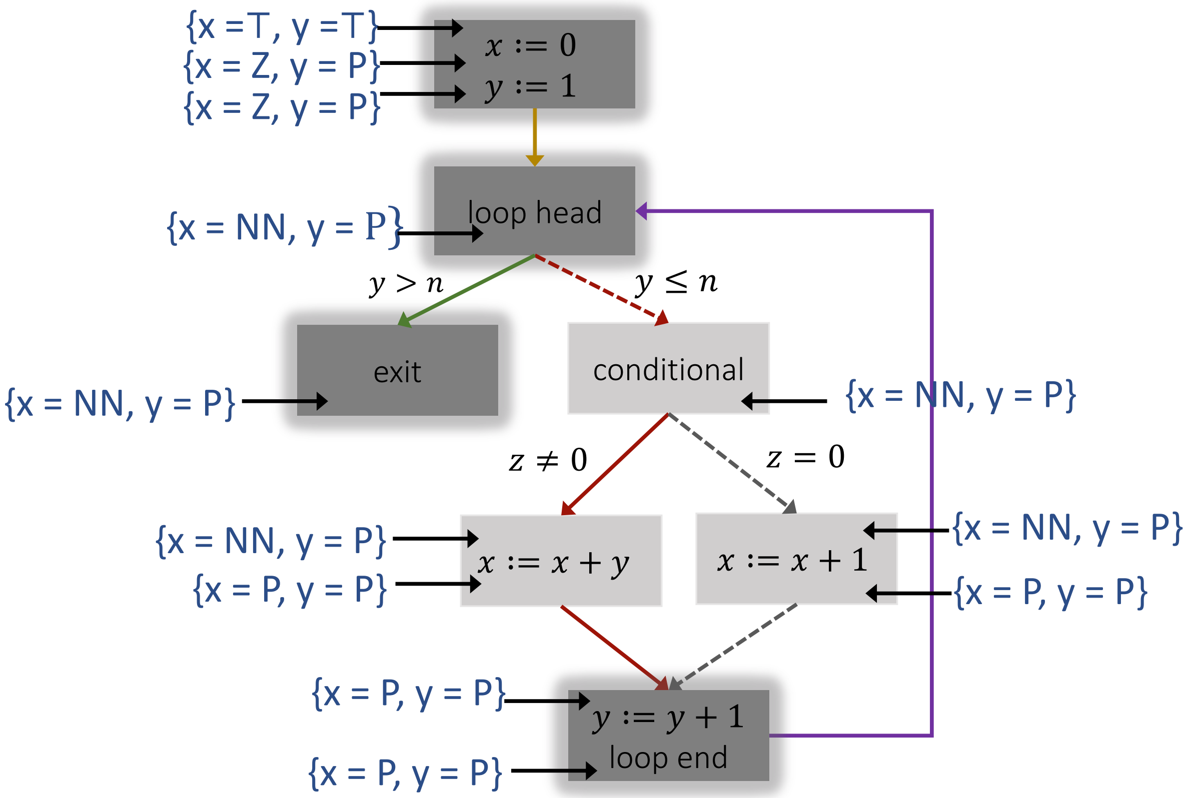
A safety property $F$ asserts that $F$ is true at every program location in every program execution. Equivalently, it asserts that the set of reachable states at any location satisfies $F$.

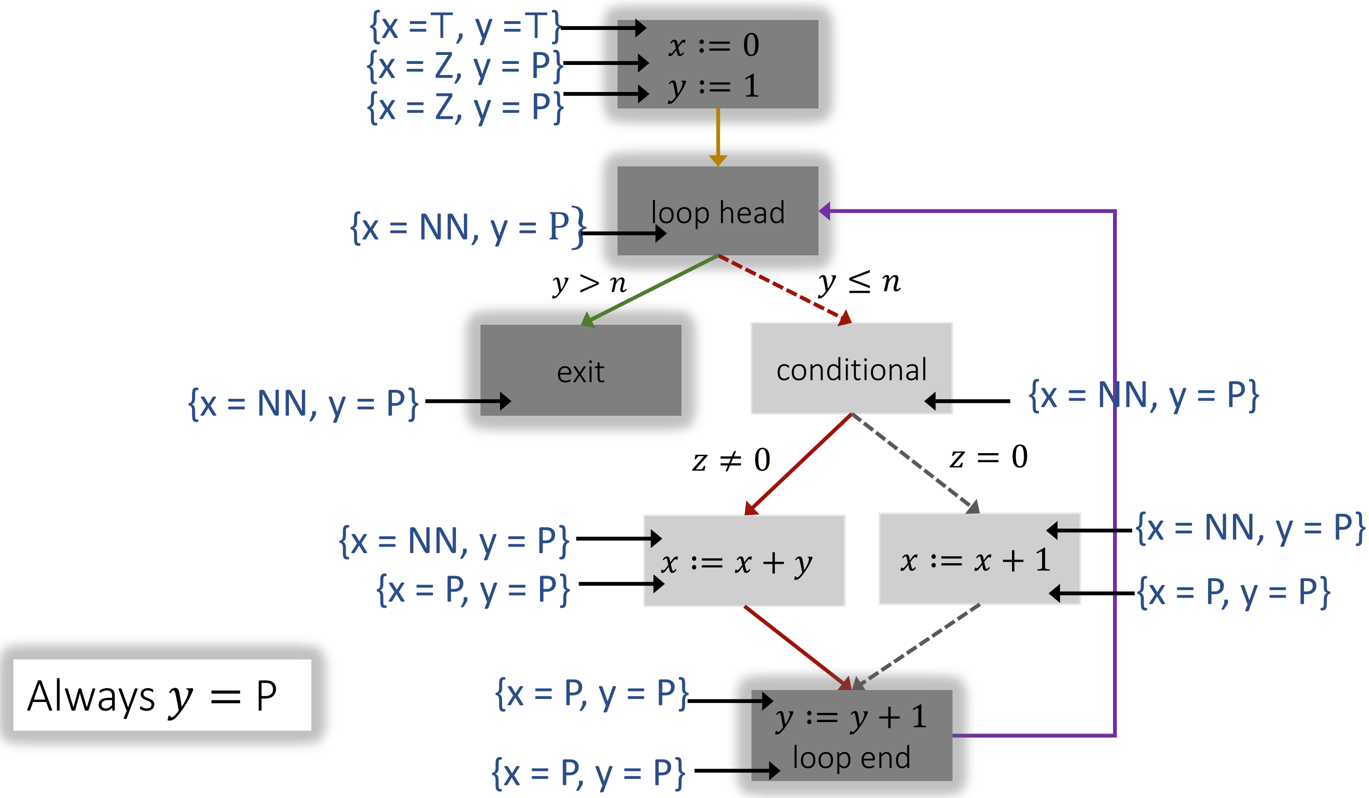
```
x := 0
y := 1
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    y := y + 1
```

Always $y > 0$

A safety property $F$ asserts that $F$ is true at every program location in every program execution. Equivalently, it asserts that the set of reachable states at any location satisfies $F$.

Can use forward propagation to compute the exact/overapproximate set of reachable states at every location $L$.


{x =T, y =T}
{x = Z, y = P}
{x = Z, y = P}
x := 0
y := 1
loop head
{x = NN, y = P}
y > n
y ≤ n
exit
conditional
{x = NN, y = P}
{x = NN, y = P}
z ≠ 0
z = 0
{x = NN, y = P}
{x = P, y = P}
x := x + y
x := x + 1
{x = NN, y = P}
{x = P, y = P}
{x = P, y = P}
y := y + 1
loop end
{x = P, y = P}


{x =T, y =T}
{x = Z, y = P}
{x = Z, y = P}
$x := 0$
$y := 1$
loop head
{x = NN, y = P}
$y > n$
$y \leq n$
exit
{x = NN, y = P}
conditional
{x = NN, y = P}
$z \neq 0$
$z = 0$
{x = NN, y = P}
{x = P, y = P}
$x := x + y$
$x := x + 1$
{x = NN, y = P}
{x = P, y = P}
{x = P, y = P}
$y := y + 1$
loop end
{x = P, y = P}
Always $y$ = P

```
x := 0
y := 0
while (y <= n) do
    if (z == 0) then
        x := x + 1
    else
        x := x + y
    y := y + 1
```

Always $y > 0$

✓

# Summary

Vechev ← $AI^2$

Cousot ← $AI^2$

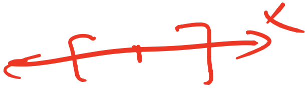
BDDs

Today

- Automated invariant generation
- Forward propagation using strongest postconditions
- Abstract interpretation

Next

- Abstraction-refinement and predicate abstraction


$\leq x \leq U_x$

$\leq y \leq U_y$

$\leq z \leq U_z$