Proving Programs Correct by Abstract Interpretation

Supratik Chakraborty
IIT Bombay

WEPL (POPL) 2015

Program Analysis: An Example

```
 int  x = 0, y = 0, z;
read(z);
while ( f(x, z)  > 0) {
   if ( g(z, y)  > 10) {
      x = x + 1;  y = y + 100;
   }
   else if ( h(z)  > 20) {
       if (x >= 4) {
           x = x + 1; y = y + 1;
       }
    }
}
```

IDEAS?
- ➢Run test cases
- ➢Get code analyzed by many people
- ➢Convince yourself by ad-hoc reasoning

What is the relation between x and y on exiting while loop?

Program Verification: An Example

```
int  x = 0, y = 0, z;
read(z);
while ( f(x, z)  > 0) {
   if ( g(z, y)  > 10) {
      x = x + 1;  y = y + 100;
   }
   else if ( h(z)  > 20) {
      if (x >= 4) {
         x = x + 1; y = y + 1;
      }
   }
}
assert( x < 4  OR y >= 2 );
```

IDEAS?

- Run test cases
- Get code analyzed by many people
- Convince yourself by ad-hoc reasoning

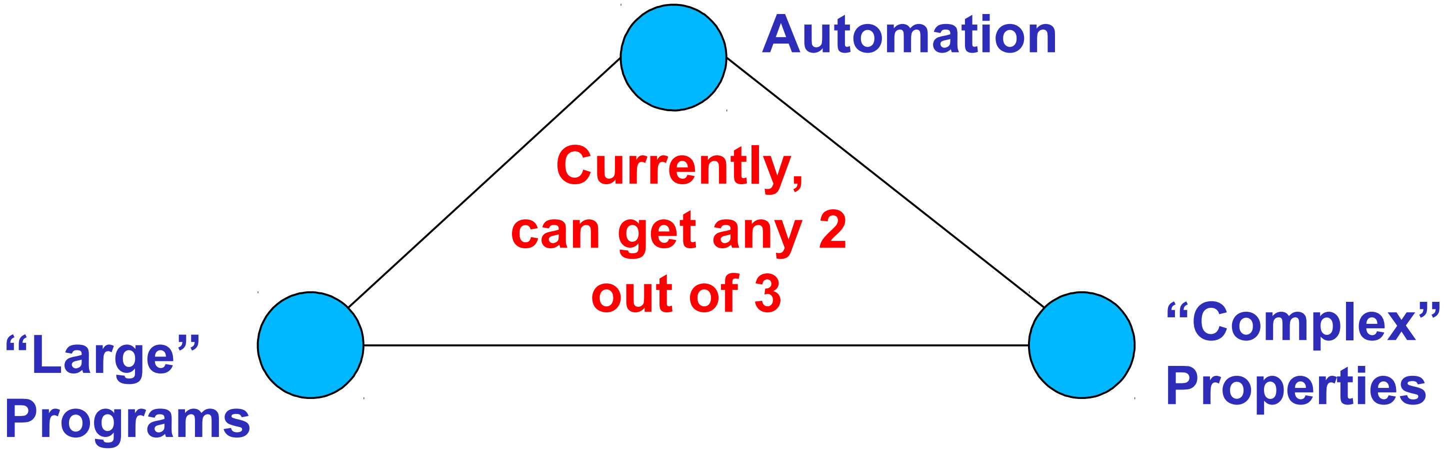
INVARIANT or PROPERTY

Verification & Analysis: Close Cousins

- Both investigate relations between program variables at different program locations
- Verification: A (seemingly) special case of analysis
  - Yes/No questions
  - No simpler than program analysis
- Both problems **undecidable** (in general) for languages with loops, integer addition and subtraction
  - Exact algorithm for program analysis/verification that works for all programs & properties: an impossibility
  - But why care about arbitrary programs?

Hope for Real-Life Software

- Certain classes of analyses/property-checking of real-life software feasible in practice
  - Uses domain specific techniques, restrictions on program structure…
  - "Safety" properties of avionics software, device drivers, …
- A practitioner's perspective


Automation

Currently, can get any 2 out of 3

"Large" Programs

"Complex" Properties

Some Driving Factors

- Compiler design and optimizations
  - Since earliest days of compiler design
- Performance optimization
  - Renewed importance for embedded systems
- Testing, verification, validation
  - Increasingly important, given criticality of software
- Security and privacy concerns
- Distributed and concurrent applications
  - Human reasoning about all scenarios difficult

Successful Approaches in Practical Software Verification

- Use of sophisticated abstraction and refinement techniques
  - Domain specific as well as generic
- Use of constraint solvers
  - Propositional, quantified boolean formulas, first-order theories, …
- Use of scalable symbolic reasoning techniques
  - Several variants of decision diagrams, combinations of decision diagrams & satisfiability solvers …
- Incomplete techniques that scale to real programs

Focus of today's talk

Abstract Interpretation Framework

- Elegant **unifying framework** for several program analysis & verification techniques
- Several success stories
  - Checking properties of avionics code in Airbus
  - Checking properties of device drivers in Windows
  - Many other examples
    - Medical, transportation, communication …
- But, **NOT a panacea**
- Often used in combination with other techniques

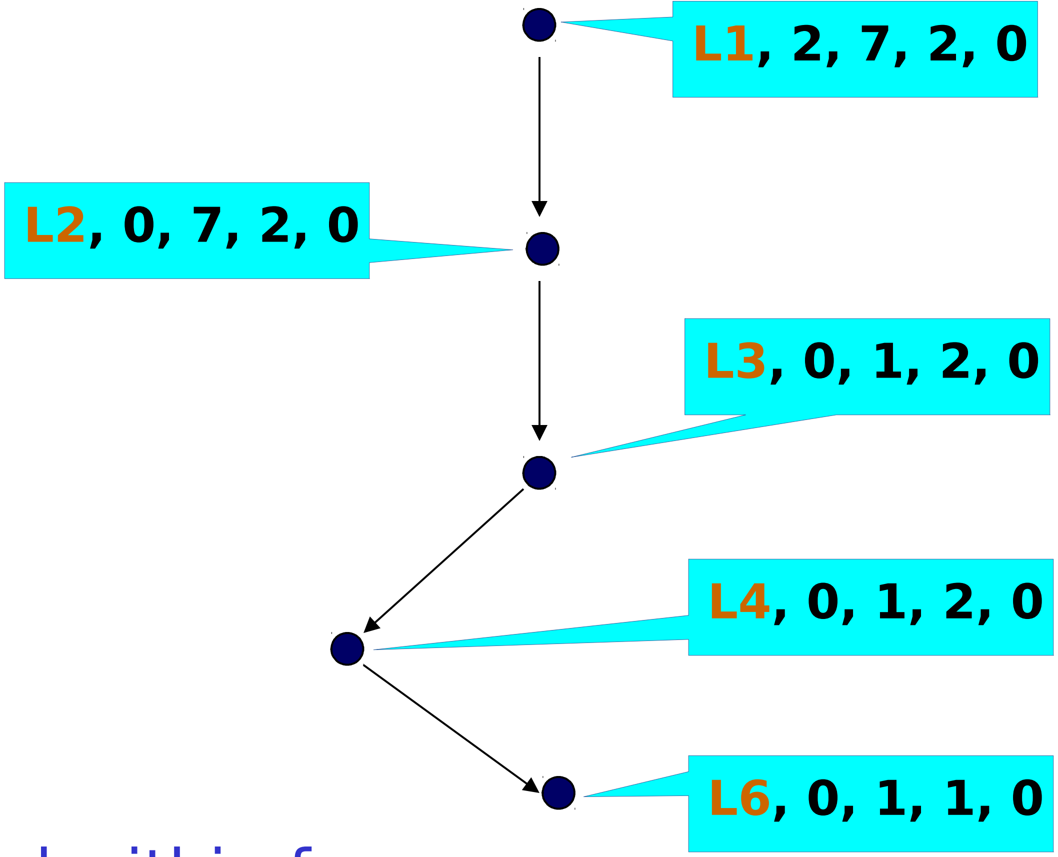
Sequential Program State

- Given sequential program P
  - State: information necessary to determine complete future behaviour
  - (pc, store, heap, call stack)
  - pc: program counter/location
  - store: map from program variables to values
  - heap: dynamically allocated/freed memory and pointer relations thereof
  - call stack: stack of call frames

Programs as State Transition Systems

- A simple program:

```
void func(int a, int b)
{ int x, y;

   L1: x = 0;
   L2: y = 1;
   L3: if (a >= b + 2)
   L4:    a = y;
          else
   L5:    b = x;
   L6: return;
}
```



State (pc, x, y, a, b)

L1, 2, 7, 2, 0

L2, 0, 7, 2, 0

L3, 0, 1, 2, 0

L4, 0, 1, 2, 0

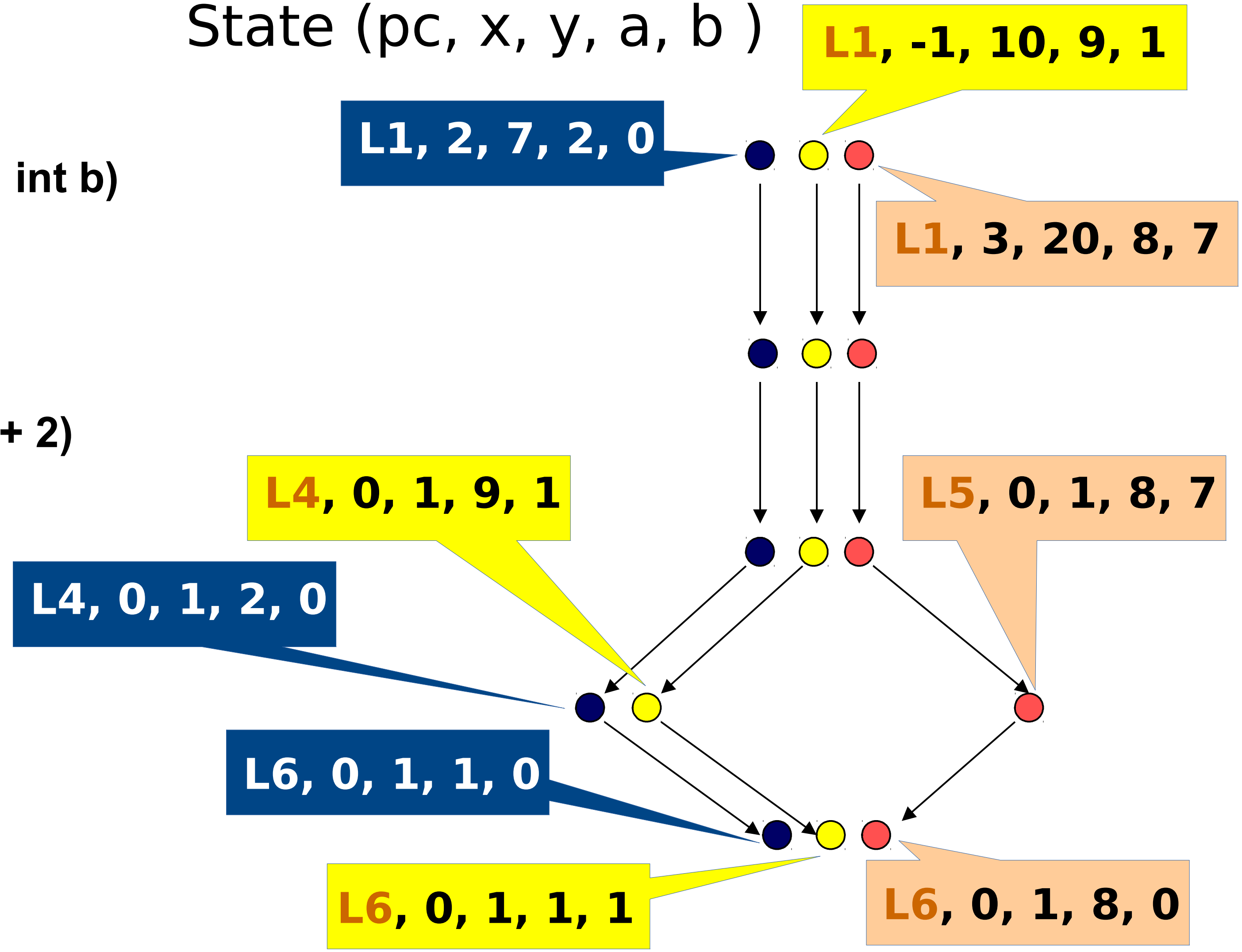
L6, 0, 1, 1, 0

State = (pc, store)
heap, stack unchanged within func

Programs as State Transition Systems

```
void func(int a, int b)
{ int x, y;

   L1: x = 0;
   L2: y = 1;
   L3: if (a >= b + 2)
   L4:   a = y;
         else
   L5:   b = x;
   L6: return;
}
```

State (pc, x, y, a, b)



L1, -1, 10, 9, 1

L1, 2, 7, 2, 0

L1, 3, 20, 8, 7

L4, 0, 1, 9, 1

L5, 0, 1, 8, 7

L4, 0, 1, 2, 0

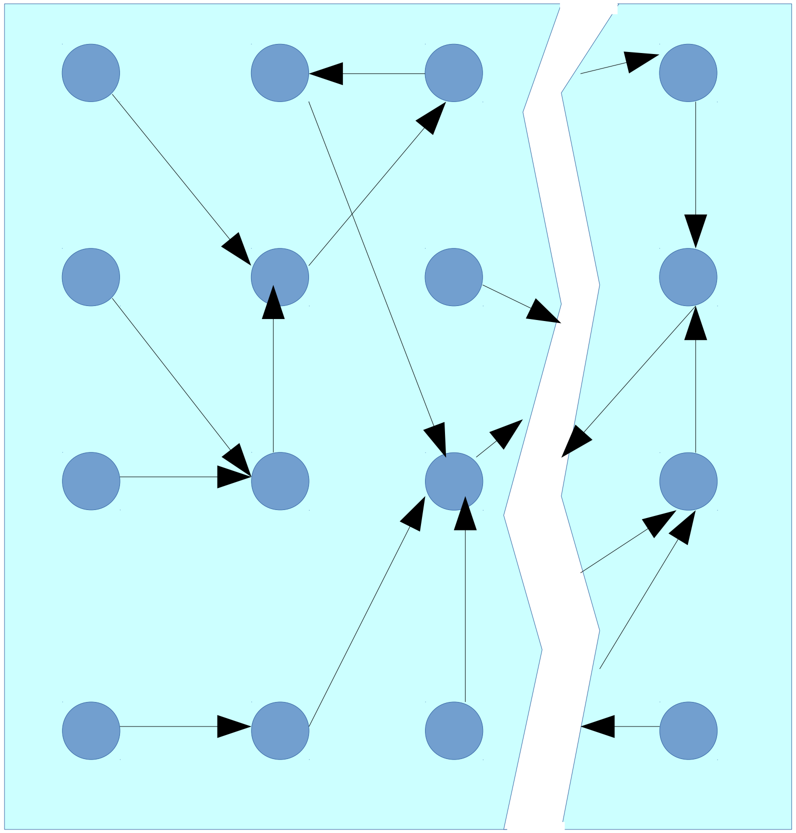
L6, 0, 1, 1, 0

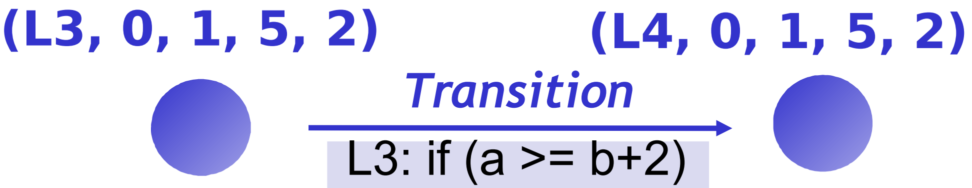
L6, 0, 1, 1, 1

L6, 0, 1, 8, 0

Programs as State Transition Systems

State: pc, x, y, a, b





(L3, 0, 1, 5, 2)

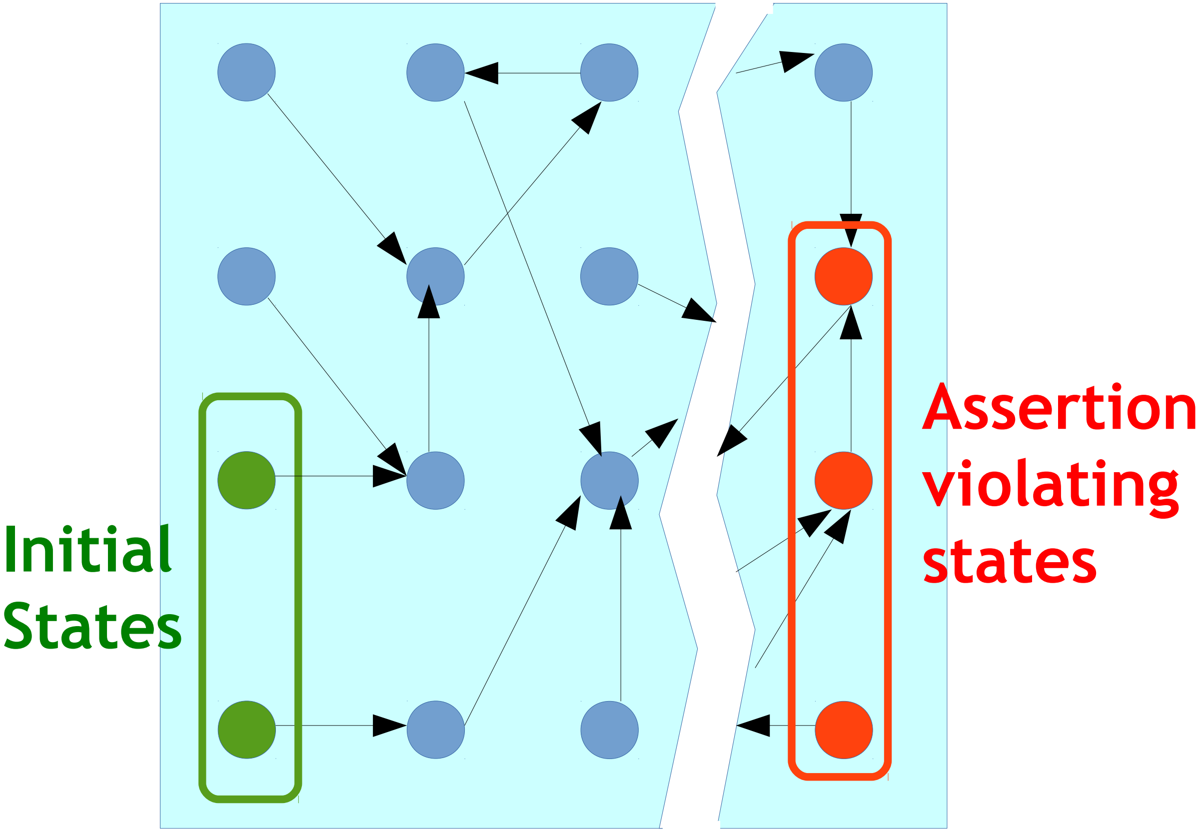
Transition

(L4, 0, 1, 5, 2)

```
L3: if (a >= b+2)
L4: …
    else
L5:
```

```
void func(int a, int b)
{ int x, y;

   L1: x = 0;
   L2: y = 1;
   L3: if (a >= b + 2)
   L4:   a = y;
       else
   L5:   b = x;
   L6: return;
}
```

Assertion Checking as Reachability



Path from an **initial** to an **assertion violating** state ?

- Absence of path: System cannot exhibit error
- Presence of path: System can exhibit error

What happens with procedure calls/returns?

State Space: How large is it?

- State = (pc, store, heap, call stack)
  - pc: finite valued
  - store: finite if all variables have finite types
  - Every program statement effects a state transition
  - enum {wait, critical, noncritical} pr_state (finite)
  - int a, b, c (infinite)
  - bool *p, *q (infinite)
  - heap: unbounded in general
  - call stack: unbounded in general
- **Bad news: State space infinite in general**

Dealing with State Space Size

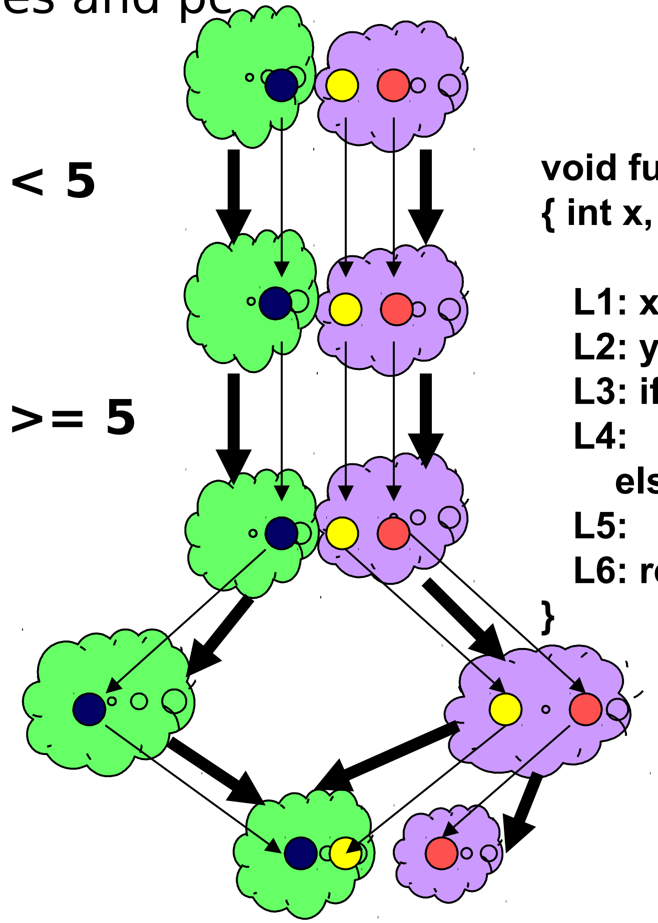
- Infinite state space
  - Difficult to represent using state transition diagram
  - Can we still do some reasoning?
- Solution: Use of abstraction
  - Naive view
    - Bunch sets of states together “intelligently”
    - Don't talk of individual states, talk of a representation of a set of states
    - Transitions between state set representations
  - Granularity of reasoning shifted
  - Extremely powerful general technique
    - Allows reasoning about large/infinite state spaces

Concrete states

Abstract states

Simple Abstractions

Group states according to values of variables and pc

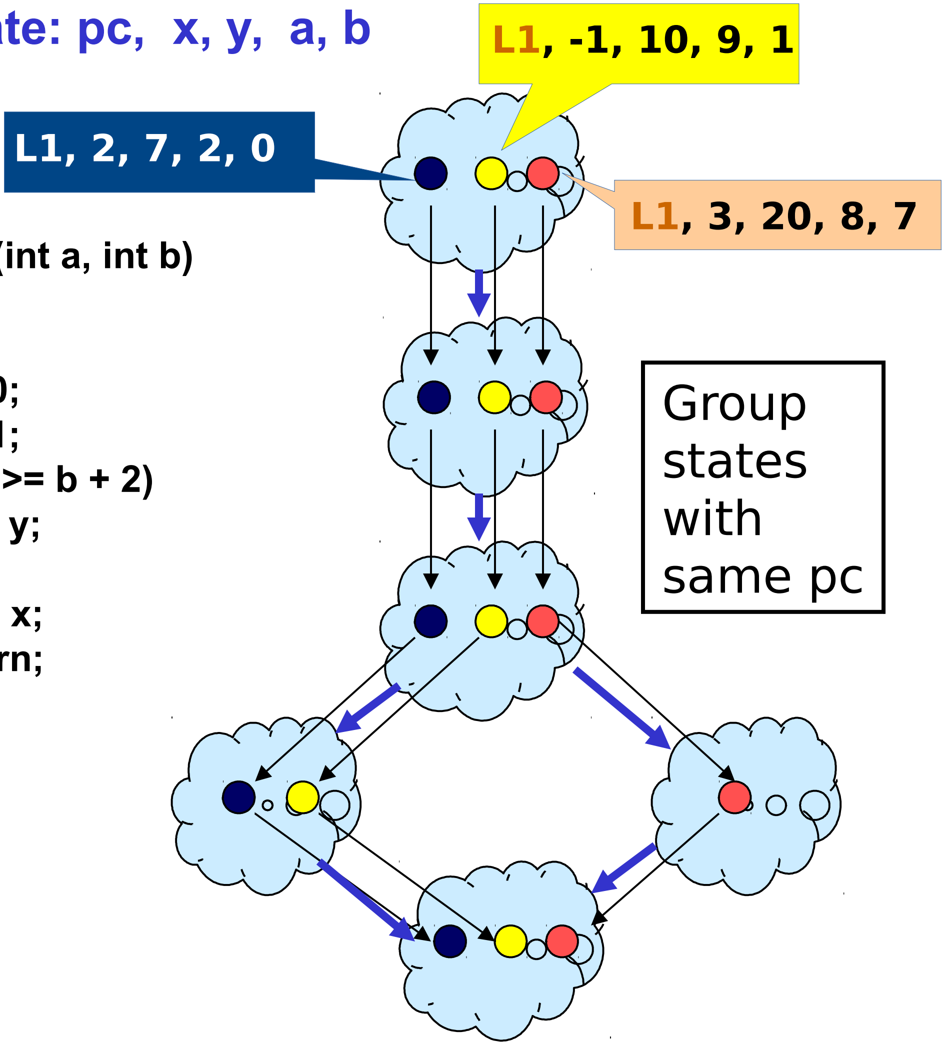


🟩 **a < 5**

🟪 **a >= 5**

```
void func(int a, int b)
{ int x, y;

   L1: x = 0;
   L2: y = 1;
   L3: if (a >= b + 2)
   L4:   a = y;
         else
   L5:   b = x;
   L6: return;
}
```

State: pc, x, y, a, b



L1, 2, 7, 2, 0

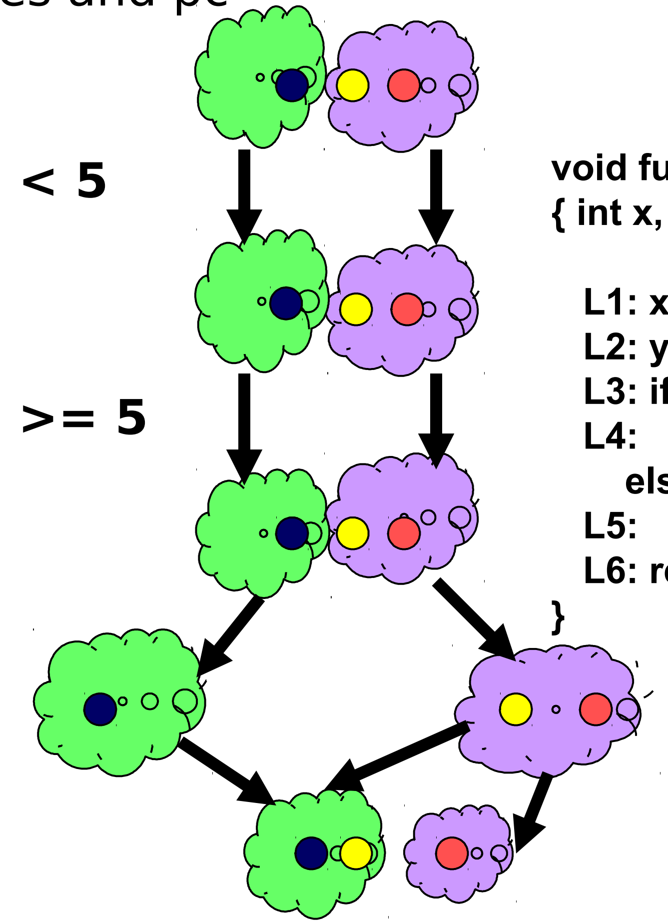
L1, -1, 10, 9, 1

L1, 3, 20, 8, 7

Group states with same pc

Programs as State Set Transformers

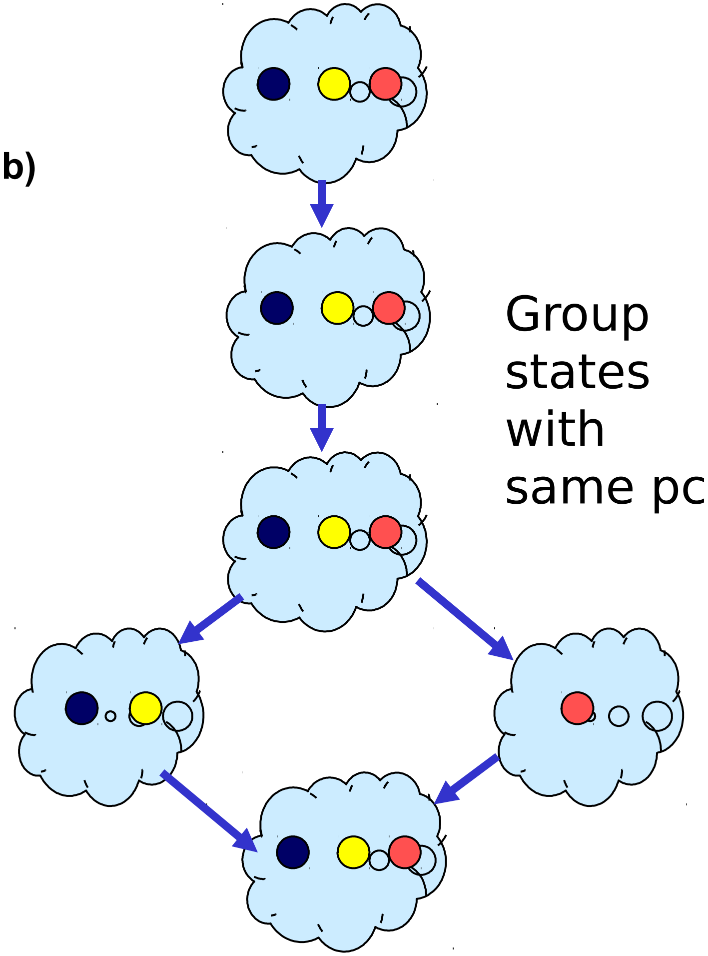
Group states according to values of variables and pc



🟩 **a < 5**

🟪 **a >= 5**

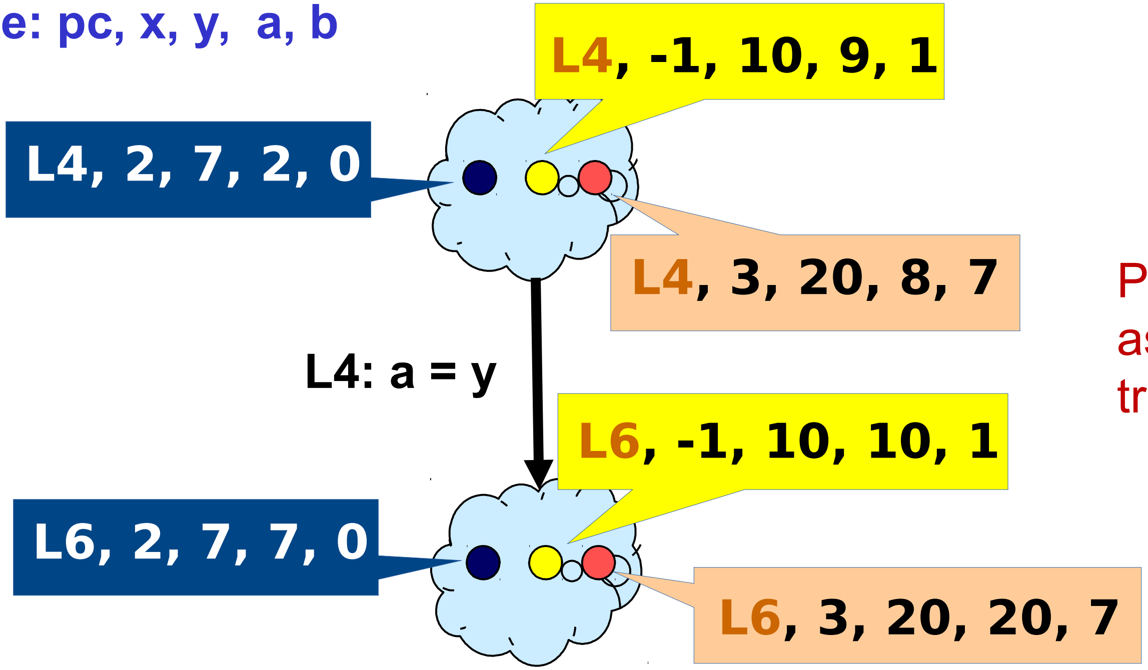
```
void func(int a, int b)
{ int x, y;

   L1: x = 0;
   L2: y = 1;
   L3: if (a >= b + 2)
   L4:   a = y;
          else
   L5:   b = x;
   L6: return;
}
```



Group states with same pc

Programs as Abstr State Transformers

- Recall: Set of (potentially infinite) concrete states is an abstract state
- **Think of program as abstract state transformer**



State: pc, x, y, a, b

L4, -1, 10, 9, 1

L4, 2, 7, 2, 0

L4, 3, 20, 8, 7

L4: a = y

L6, -1, 10, 10, 1

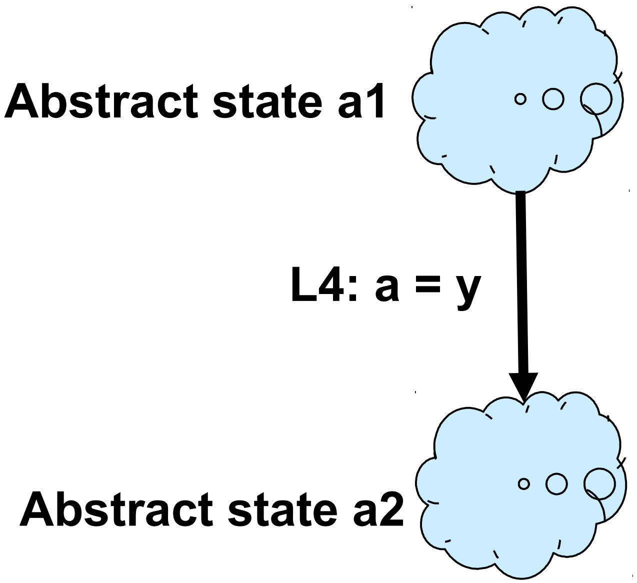
L6, 2, 7, 7, 0

L6, 3, 20, 20, 7

Program statement as concrete state transformer

Programs as Abstr State Transformers

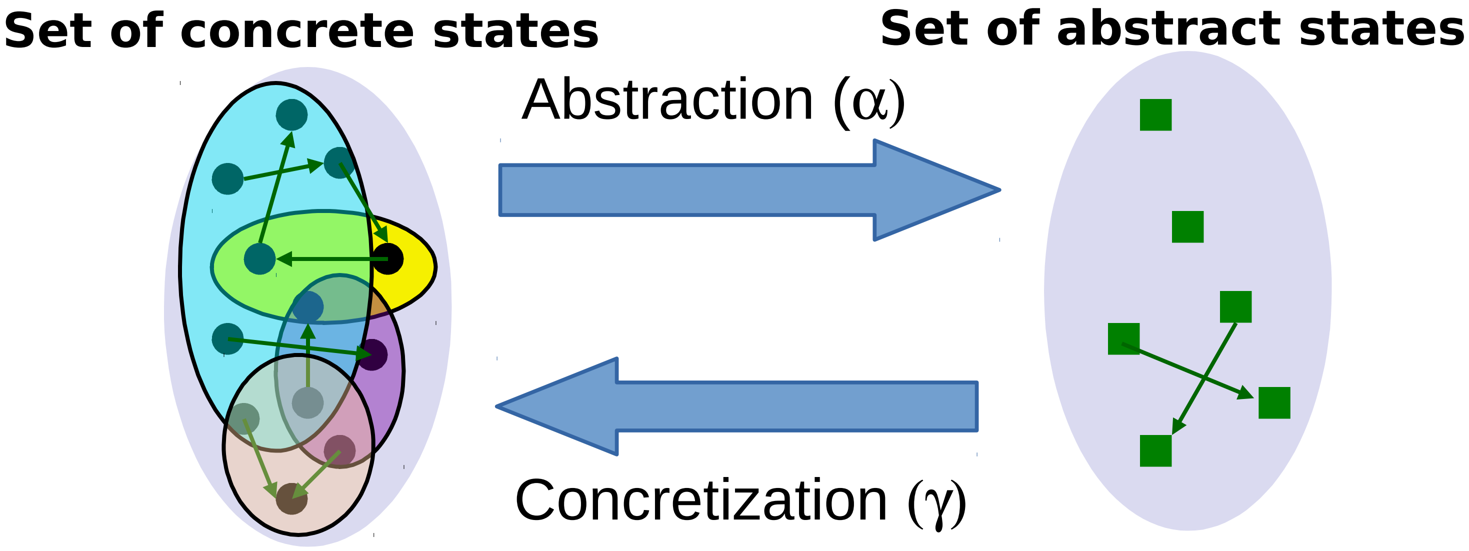
- Recall: Set of (potentially infinite) concrete states is an abstract state
- **Think of program as abstract state transformer**



Program statement as abstract state transformer

Central problem:
Compute a2 from a1 and prog stmt (abstract state transitions)

A Generic View of Abstraction



Set of concrete states

Set of abstract states

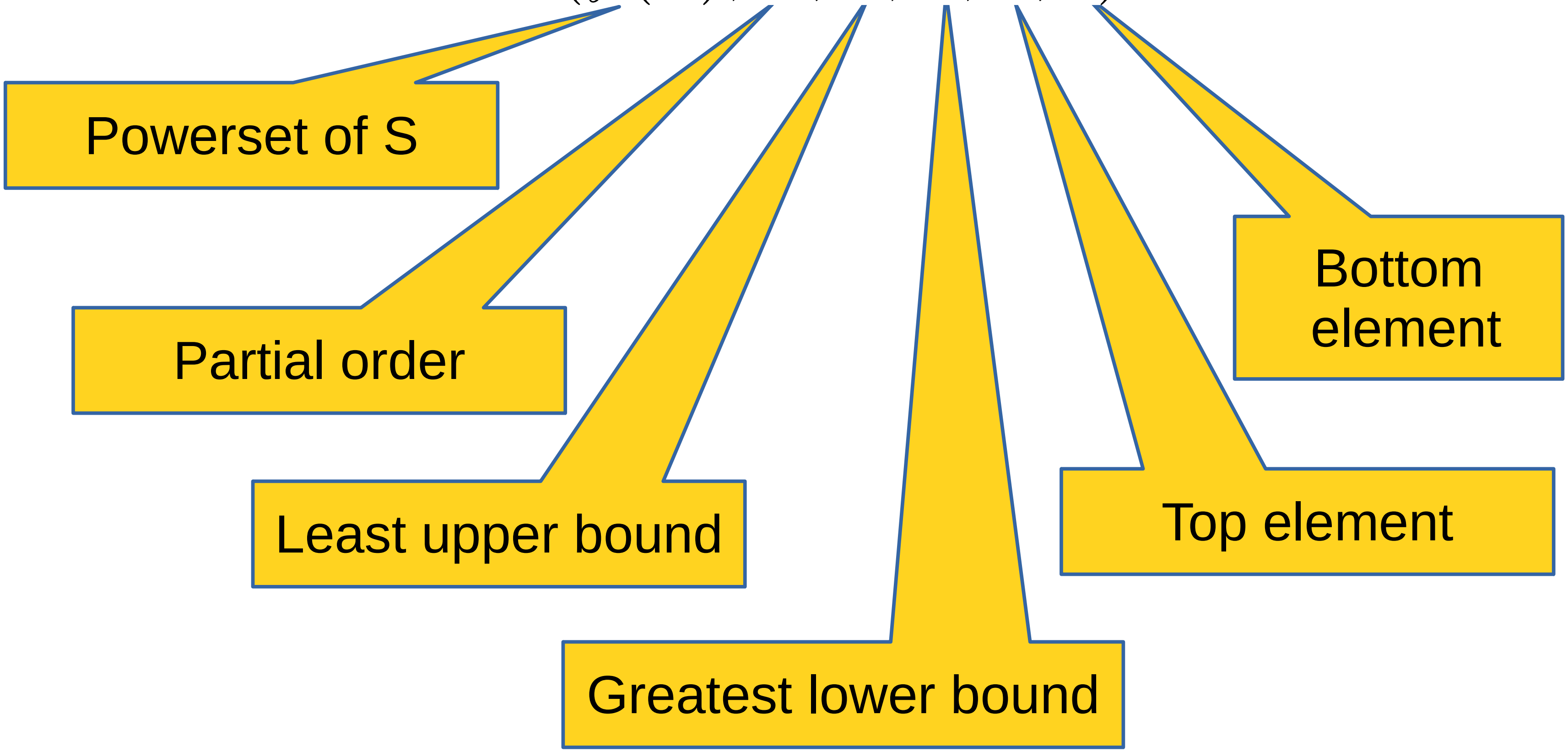
Abstraction (α)

Concretization (γ)

- Every subset of concrete states mapped to unique abstract state
- Desirable to capture containment relations
- Transitions between state sets (abstract states)

Mathematical Foundations of Abstract Interpretation

- Set of concrete states: S
  - Concrete lattice C = $(\wp(S), \subseteq, \cup, \cap, S, \emptyset)$



Powerset of S

Partial order

Least upper bound

Greatest lower bound

Top element

Bottom element

Mathematical Foundations of Abstract Interpretation

- Abstract lattice A $= (\mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$
-
- Abstraction function $\alpha : \wp(S) \rightarrow \mathcal{A}$

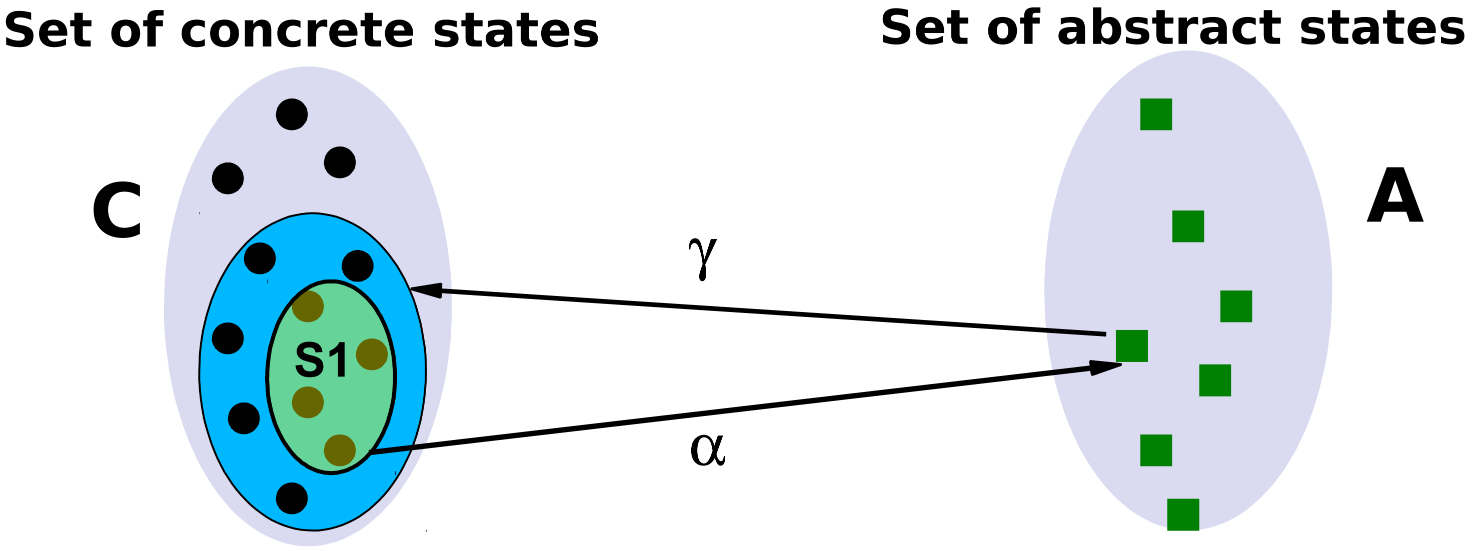
  - Monotone: $S_1 \subseteq S_2 \Rightarrow \alpha(S_1) \sqsubseteq \alpha(S_2)$ for all $S_1, S_2 \subseteq S$
  - $\alpha(S) = \top, \quad \alpha(\emptyset) = \bot$

- Concretization function $\gamma : \mathcal{A} \rightarrow \wp(S)$

  - Monotone: $a_1 \sqsubseteq a_2 \Rightarrow \gamma(a_1) \subseteq \gamma(a_2)$ for all $a_1, a_1 \in \mathcal{A}$
  - $\gamma(\top) = S, \quad \gamma(\bot) = \emptyset$

Mathematical Foundations of Abstract Interpretation

- α and γ form a ***Galois connection***
  - First view: $S_1 \subseteq \gamma(\alpha(S_1))$ for all $S_1 \subseteq S$



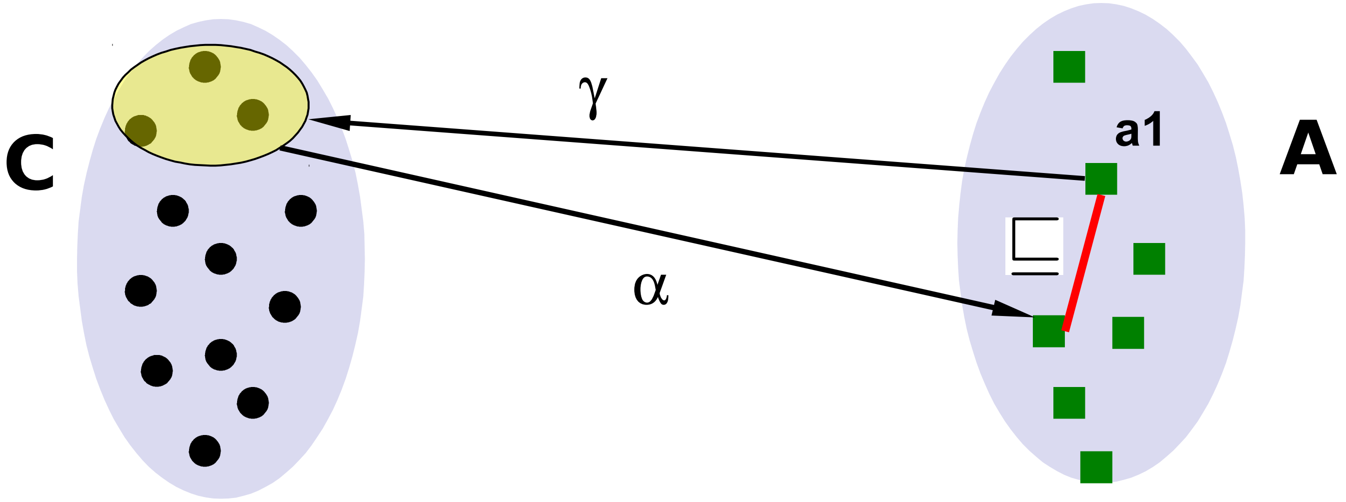
Mathematical Foundations of Abstract Interpretation

- α and γ form a ***Galois connection***
  - First view: $S_1 \subseteq \gamma(\alpha(S_1))$ for all $S_1 \subseteq S$

    $\alpha(\gamma(a_1)) \sqsubseteq a_1$ for all $a_1 \in \mathcal{A}$

Set of concrete states

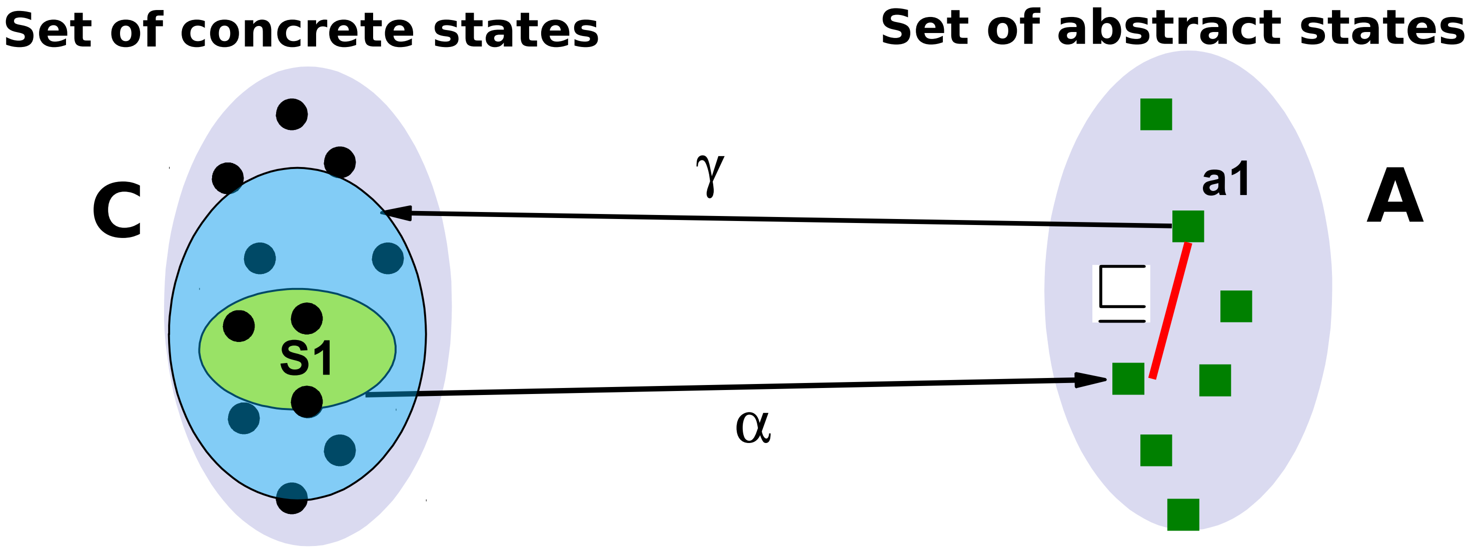
Set of abstract states



Mathematical Foundations of Abstract Interpretation

- α and γ form a ***Galois connection***
  - Second (equivalent) view:

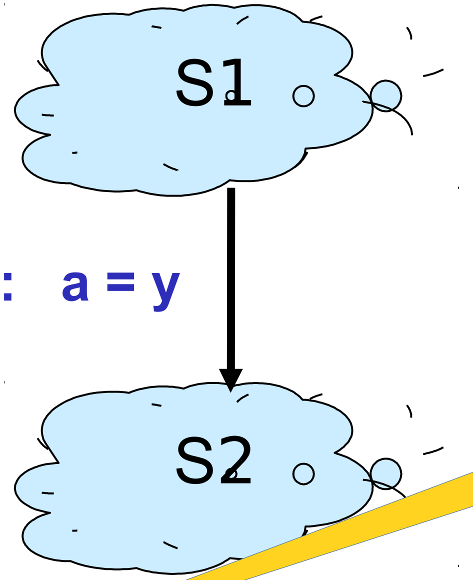
$$\alpha(S_1) \sqsubseteq a_1 \Leftrightarrow S_1 \subseteq \gamma(a_1) \text{ for all } S_1 \subseteq S,\, a_1 \in \mathcal{A}$$



Computing Abstract State Transformers

- Concrete state set transformer function
  - Example:

S1 = { (L4, x, y, a, b) | ….. }: set of concr. states



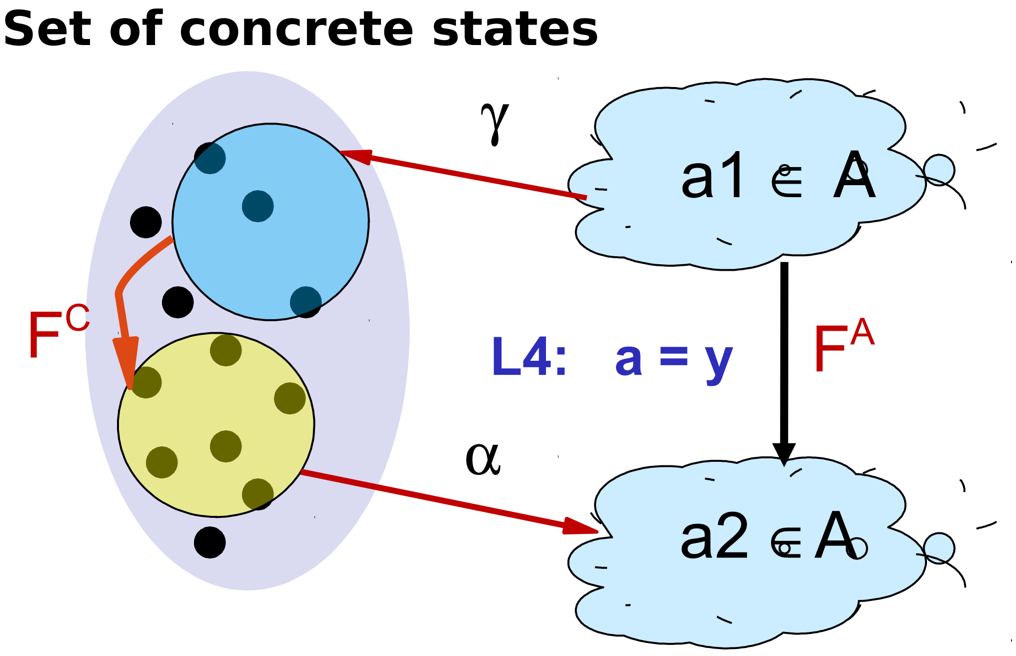
Monotone concrete state set transformer function for stmt at L4

$$S2 = \{ (L6, x, y, a', b) \mid \exists\, (L4, x, y, a, b) \in S1 \;\wedge\; a' = y\}$$

$$= F^{C}(S1)$$: set of concrete states

Computing Abstract State Transformers

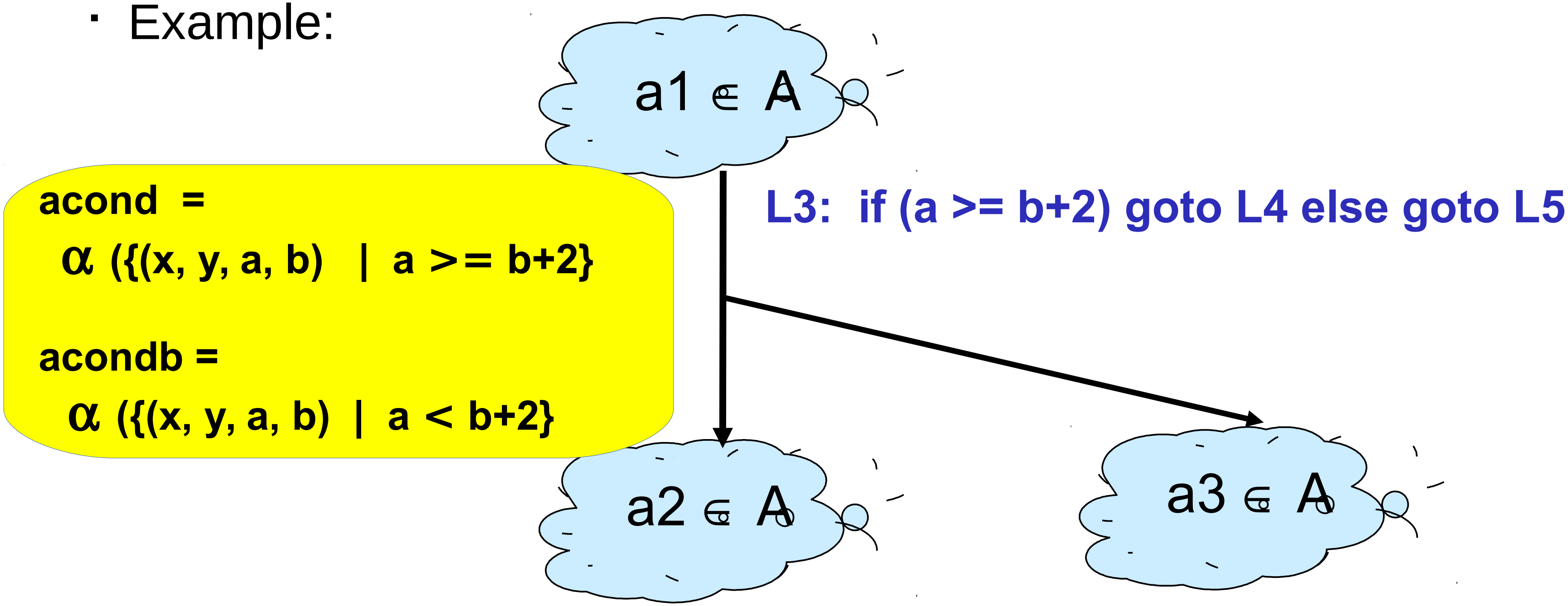
- Abstract state transformer function
  - Example:



$a2 = \alpha(F^C (\gamma(a1)))$ ideally, but $F^A(a1) \sqsupseteq \alpha(F^C (\gamma(a1)))$ often used

Computing Abstract State Transformers

- Abstract state transformer for if-then-else
  - Example:



a1 ∈ A

L3: if (a >= b+2) goto L4 else goto L5

acond =
α ({(x, y, a, b) | a >= b+2}

acondb =
α ({(x, y, a, b) | a < b+2}

a2 ∈ A

a3 ∈ A

a2 = a1 ⊓ acond
pc in a2: L4

a3 = a1 ⊓ acondb
pc in a3: L5

Dealing with Loops

- Example: ....

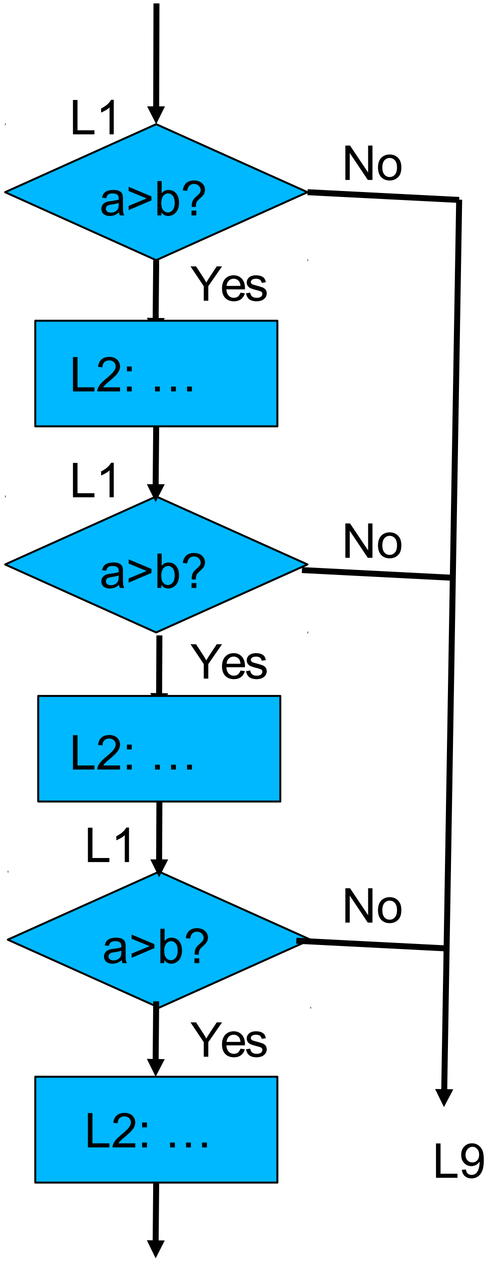
```
L1 :   while (a > b) do
         L2:  <loop body>
L9:    end while
```

Given

F^A : abstr state transformer of loop body,

a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



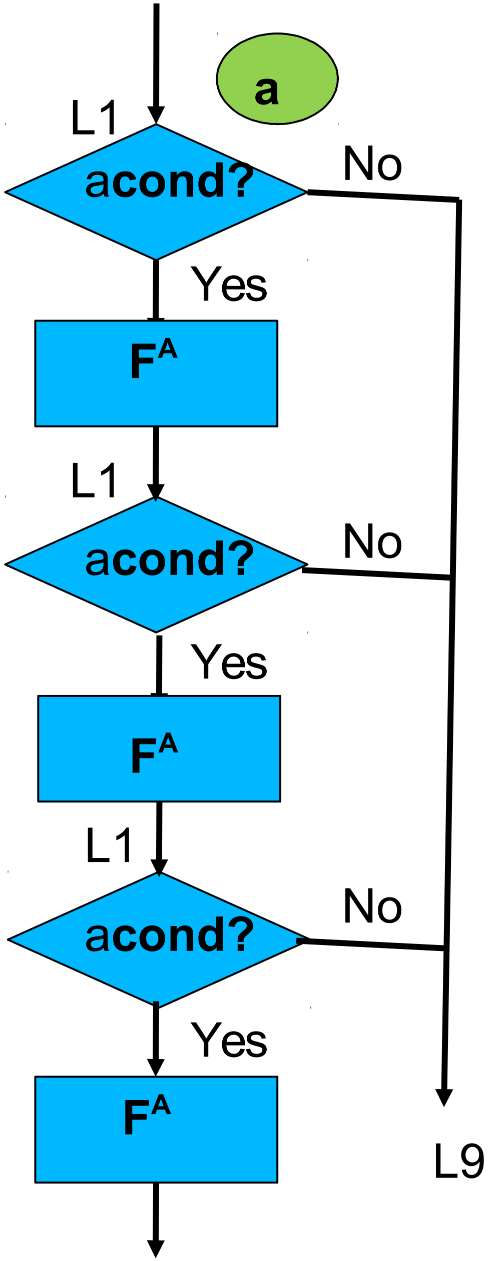
Dealing with Loops

Given

F^A : abstr state transformer of loop body,

a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



acond = α ({s | s is a concrete state with a > b})

Current view of abstract loop invariant

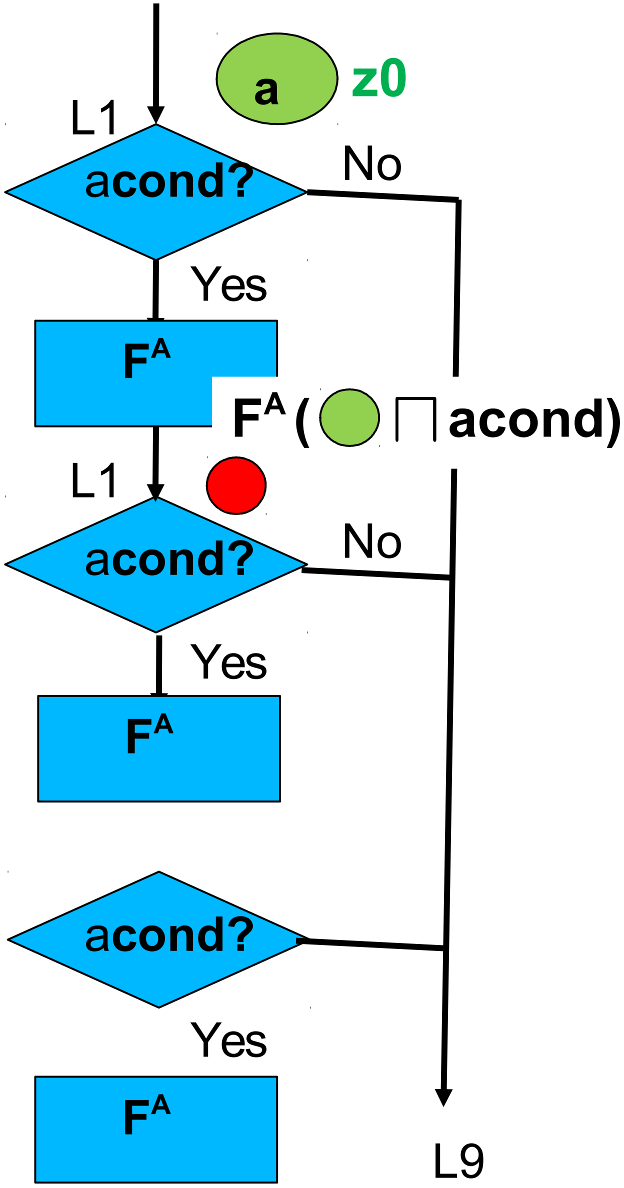
Dealing with Loops

Given

F^A : abstr state transformer of loop body,

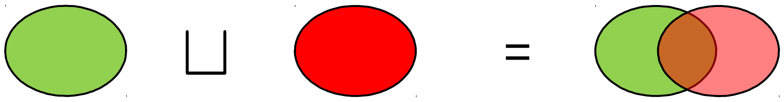
a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



acond = α ({s | s is a concrete state with a > b})

Current view of abstract loop invariant



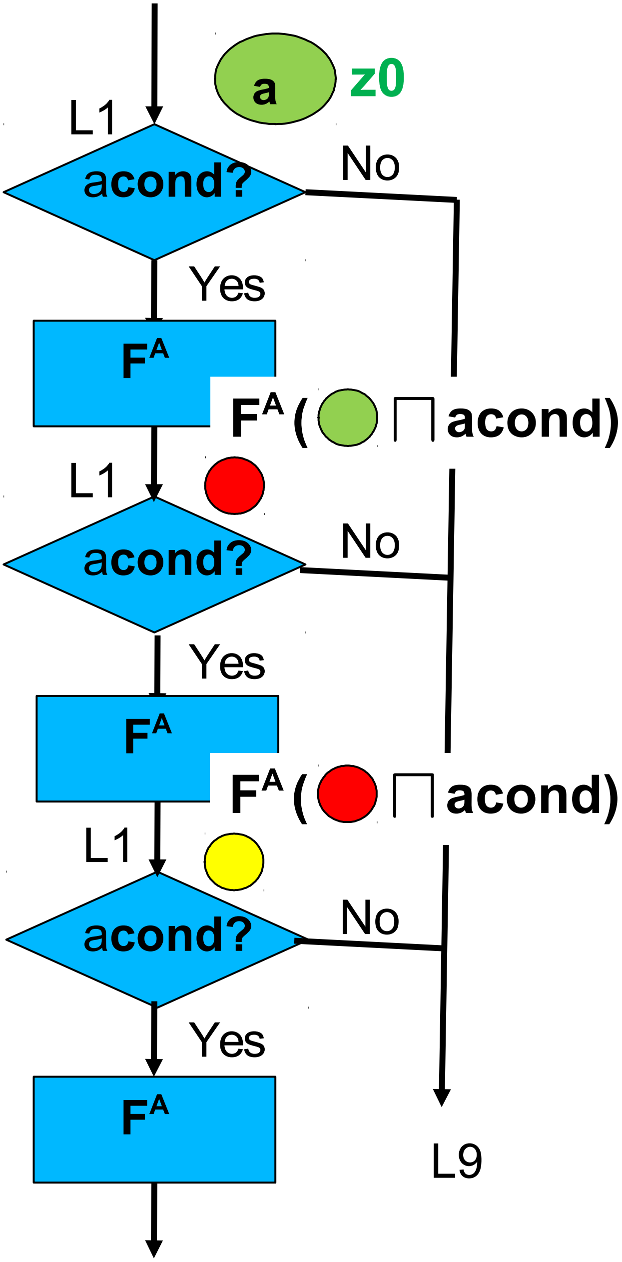
Dealing with Loops

Given

F^A: abstr state transformer of loop body,

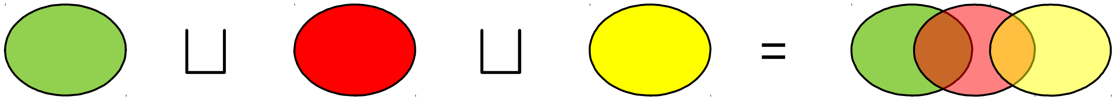
a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



acond = α ({s | s is a concrete state with a > b})

Current view of abstract loop invariant



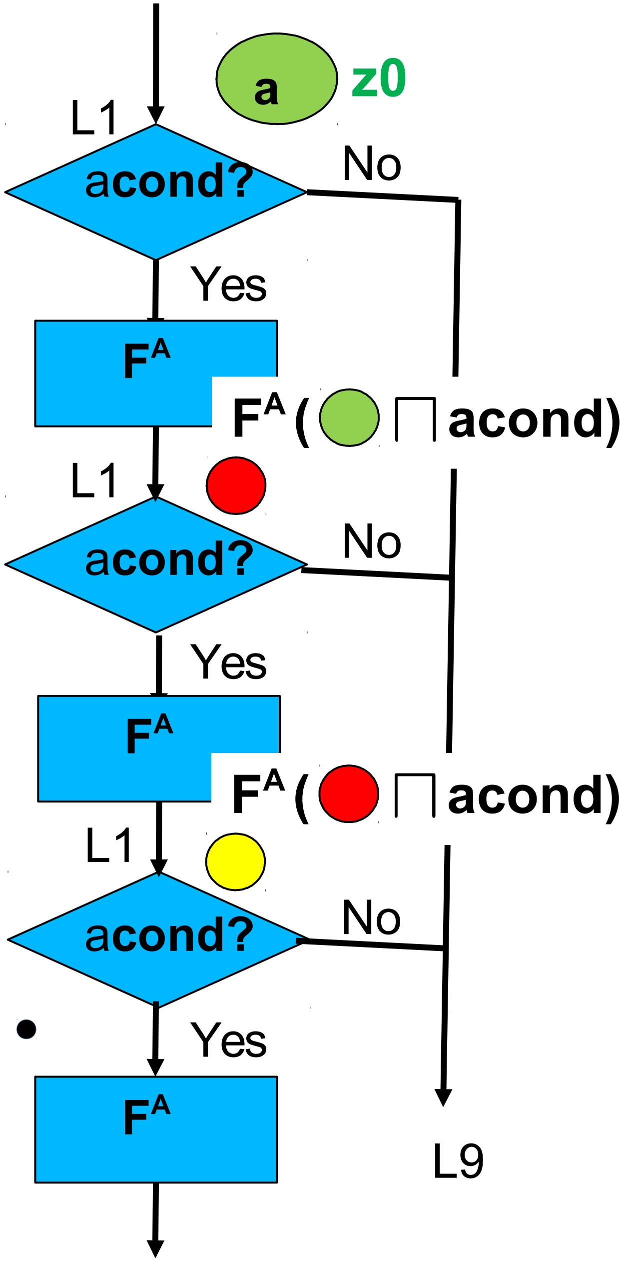
Dealing with Loops

Given

F^A: abstr state transformer of loop body,

a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



acond = α ({s | s is a concrete state with a > b})

Abstract loop invariant

🟢 ⊔ 🔴 ⊔ 🟡 ⊔ • • • • • • •

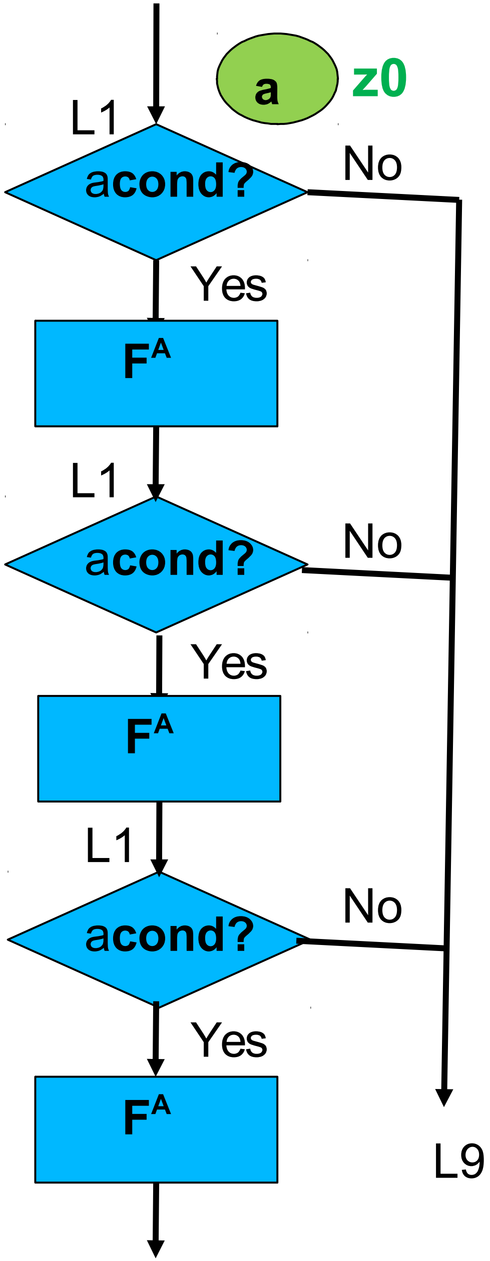
Dealing with Loops

Given

F^A: abstr state transformer of loop body,

a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



acond = α ({s | s is a concrete state with a > b })

Loop invariant at L1 is limit of the sequence:

z0 = a

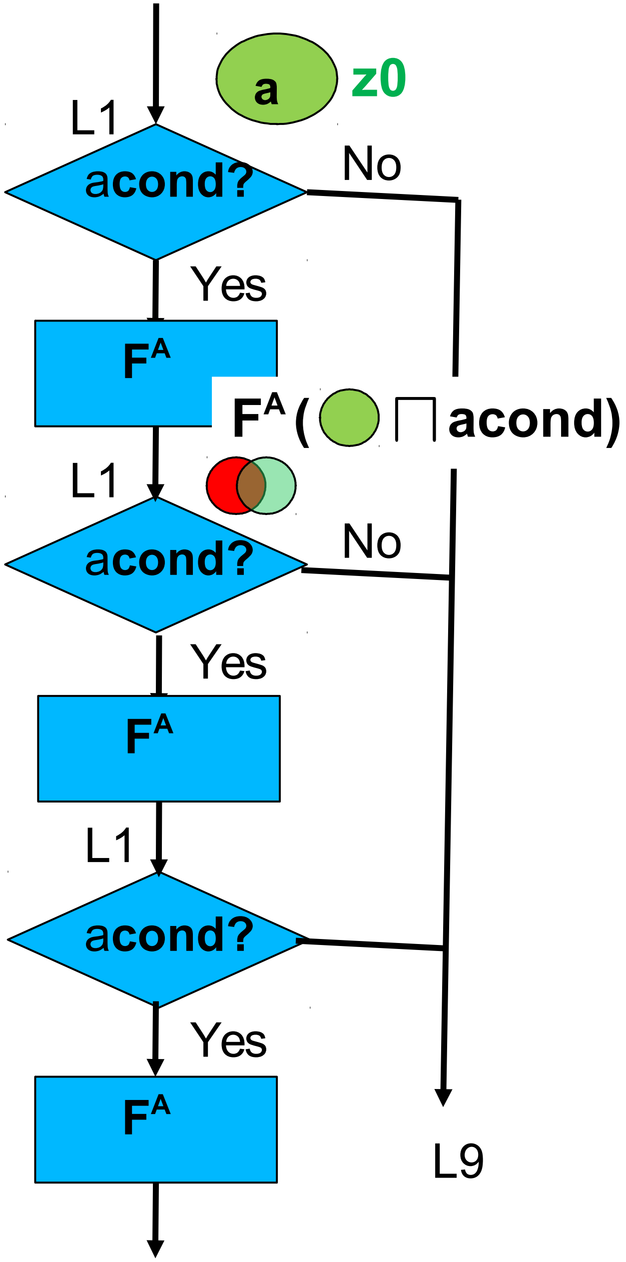
Dealing with Loops

Given

F^A : abstr state transformer of loop body,

a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



acond = α ({s | s is a concrete state with a > b })

Loop invariant at L1 is limit of the sequence:

z0 = a

$z1 = a \sqcup F^A (z0 \sqcap acond)$

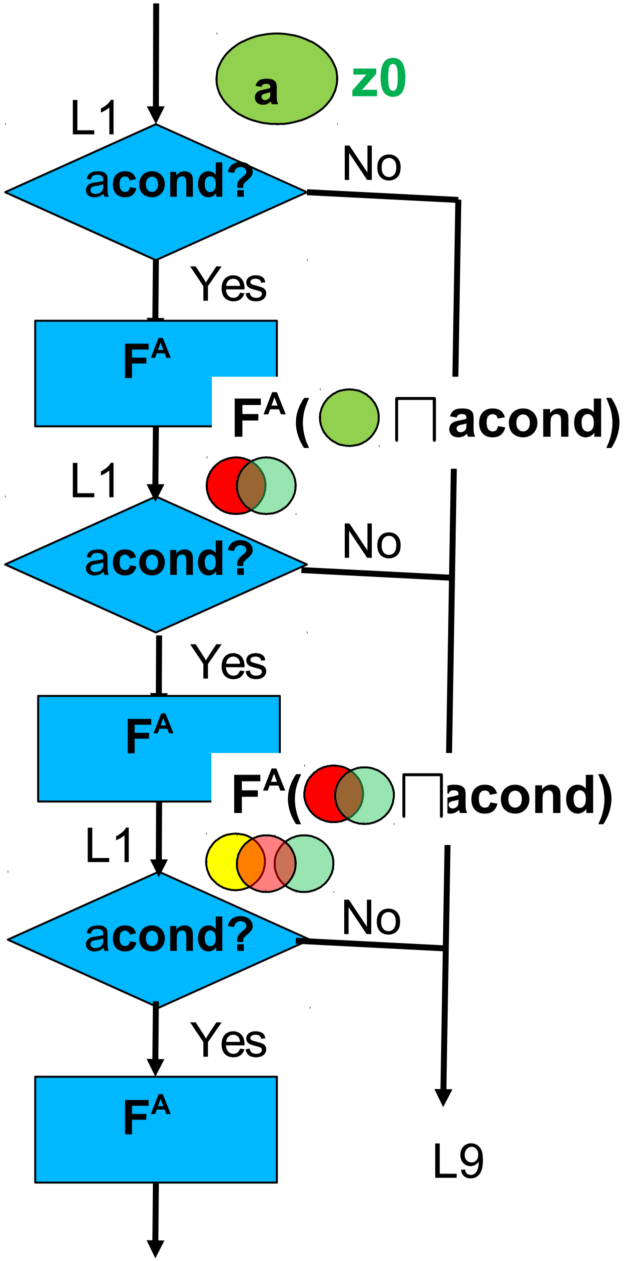
Dealing with Loops

Given

F^A: abstr state transformer of loop body,

a : abstr state at L1 the first time L1 is reached

What is the abstract loop invariant at L1?



acond = α ({s | s is a concrete state with a > b })

Loop invariant at L1 is limit of the sequence:

$$z0 = a$$

$$z1 = a \sqcup F^A\ (z0 \sqcap\ acond)$$

……

$$z_{i+1} = a\ \sqcup F^A\ (z_i \sqcap\ acond)$$

Dealing with loops

- Loop invariant at L1 is limit of the sequence:
  - $z_0 = a, \ldots, \ z_{i+1} = a \sqcup F^A (z_i \sqcap \text{acond})$
  - The limit exists and is the least fixpoint of g: $\mathcal{A} \to \mathcal{A}$
    where $g(z) = a \sqcup F^A(z \sqcap \text{acond})$
- Difficult to compute if A has infinite ascending chains
- Use an extrapolation (widen) operator r
  - $w_0 = z_0$, and $w_{i+1} = w_i \nabla z_{i+1}$ for all i >= 0
  - By definition of ∇,
    - Sequence of w_i ’s stationary after finitely many i’s
    - Stationary value w* overapproximates limit of sequence of z_i’s
  - Theory of abstract interpretation guarantees that
    γ(w*) overapproximates loop invariant at L1

Putting It All Together

- Given a program P and an assertion φ at location L
  - Choose an abstract lattice (domain) A with a ∇ operator
  - Compute abstract invariant at each location of P
  - If abstract invariant at L is a_L, check if $\gamma(a_L)$ satisfies φ
  - The theory of abstract interpretation guarantees that

$$\gamma(a_L) \supseteq \text{concrete invariant at } l$$

Bird's eye-view of program verification by abstract interpretation

A Simple Abstract Domain

Interval Abstract Domain

- Simplest domain for analyzing numerical programs
- Represent values of each variable separately using intervals
- Example:

```
L0:  x = 0; y = 0;
L1:  while (x < 100) do
         L2:  x = x+1;
         L3:  y = y+1;
L4:  end while
```

If the program terminates, does x have the value 100 on termination?

Interval Abstract Domain

- Abstract states: pairs of intervals (one for each of x, y)
  - [-10, 7] , (-1, 20]
  - $\sqsubseteq$ relation: Inclusion of intervals
  - [-10, 7] , (-1, 20] $\sqsubseteq$ [-20, 9], (-1, $+\infty$)
  - $\sqcup$ and $\sqcap$: union and intersection of intervals
  - [a, b] ∇x [c, d] = [e, f], where
    - e = a if c >= a, and e = - ∞ otherwise
    - f = b if d <= b, and f = $+\infty$ otherwise
  - ∇y similarly defined, and ∇ is simply (∇x, ∇y)
  - $\bot$ is empty interval of x and y
  - $\top$ is ($-\infty$, $+\infty$), ($-\infty$, $+\infty$)

Analyzing our Program

```
L0:  x = 0; y = 0;

L1:  while (x < 100) do

        L2:  x = x+1;

        L3:  y = y+1;

L4:  end while
```

Some Concluding Remarks

- Abstract interpretation: a fundamental technique for analysis of programs
- Choice of right abstraction crucial
- Often getting the right abstraction to begin with is very hard
    - Need automatic refinement techniques
- Very active area of research