Abstract Interpretation and Program Verification

Supratik Chakraborty
IIT Bombay

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

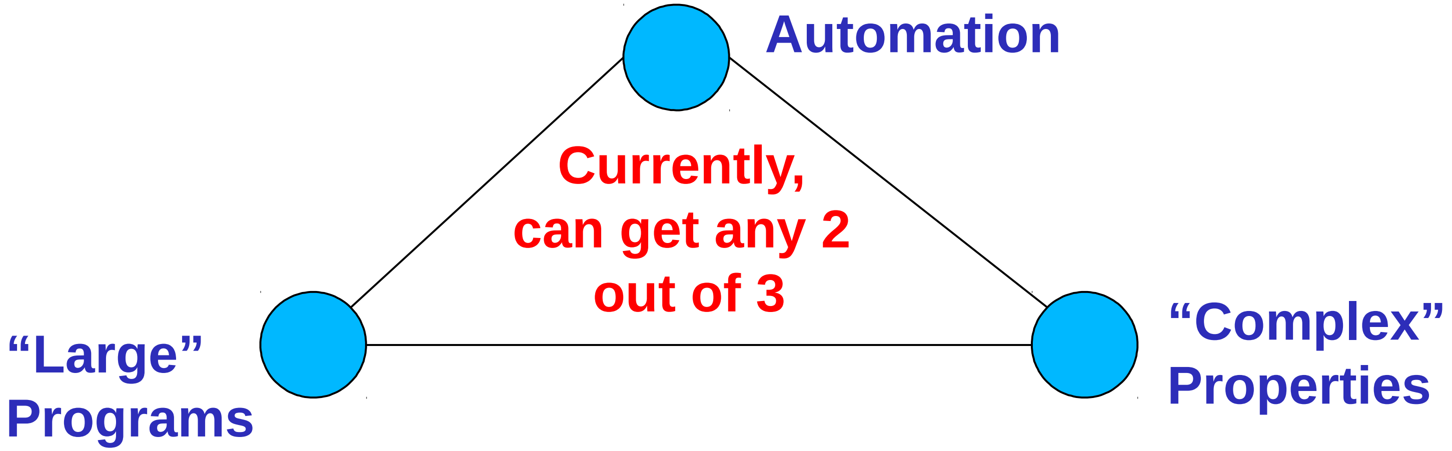
INVARIANT or PROPERTY

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**
- This doesn't reduce the importance of proving programs correct
  - Can we solve this in special (real-life) cases?

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



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, Horn clauses …
- 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

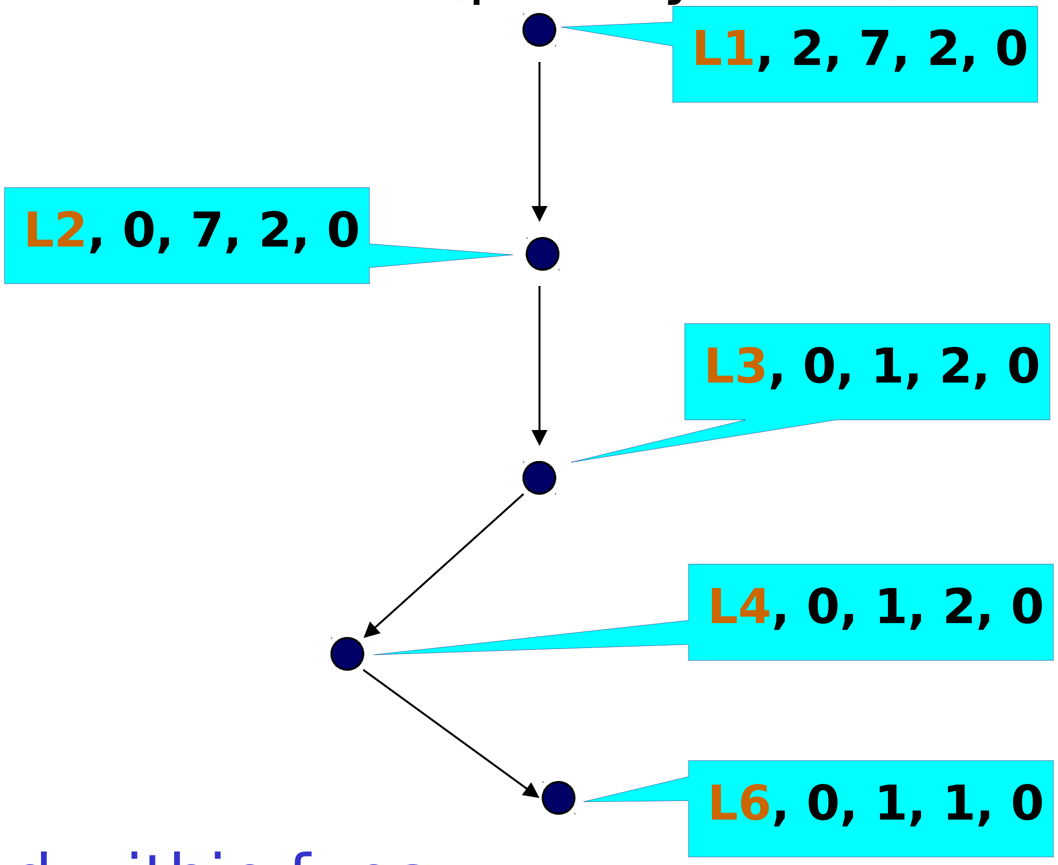
- Given sequential program P
  - State: information necessary to determine complete future behaviour
  - (pc, store, heap, call stack)
  - pc: program counter/location
  - store: map from program variables to values
  - heap: dynamically allocated/freed memory and pointer relations thereof
  - call stack: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

Programs as State Transition Systems

- A simple program:

```
int 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 (a-b);
}
```

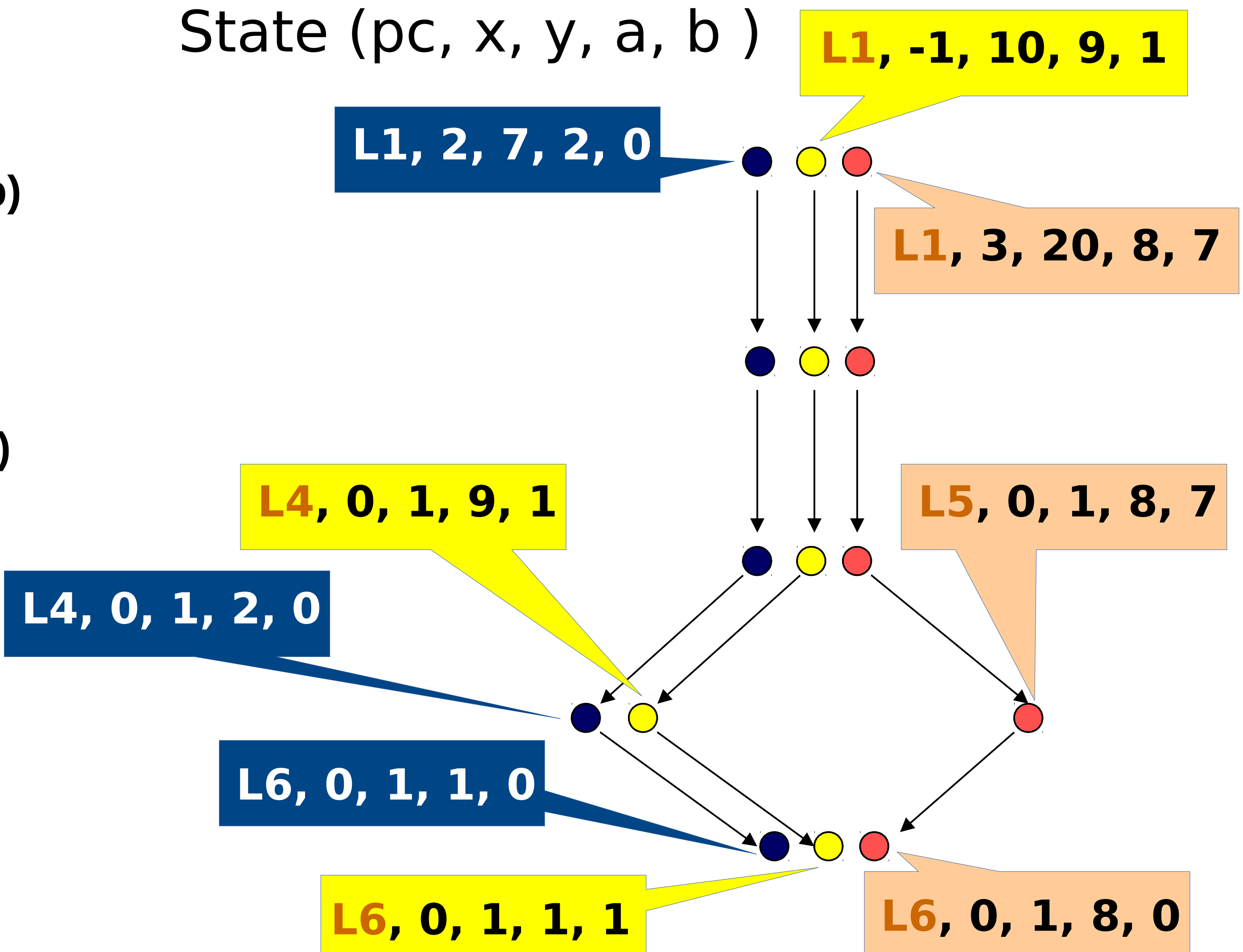
State (pc, x, y, a, b)



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

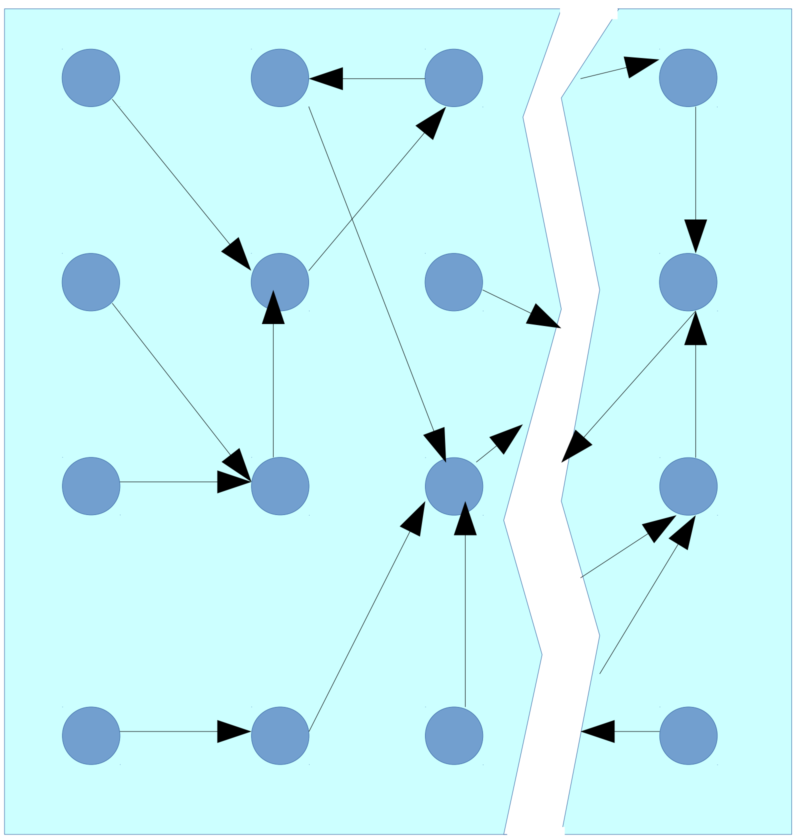
Programs as State Transition Systems

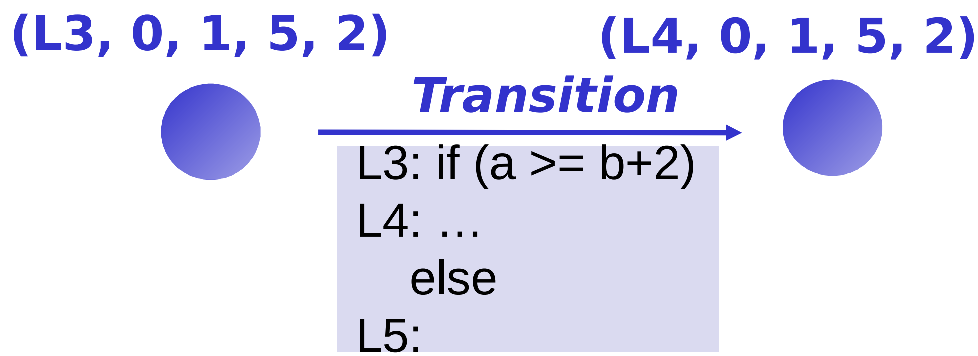
```
int func(int a, int b)
{ int x, y;

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



Programs as State Transition Systems

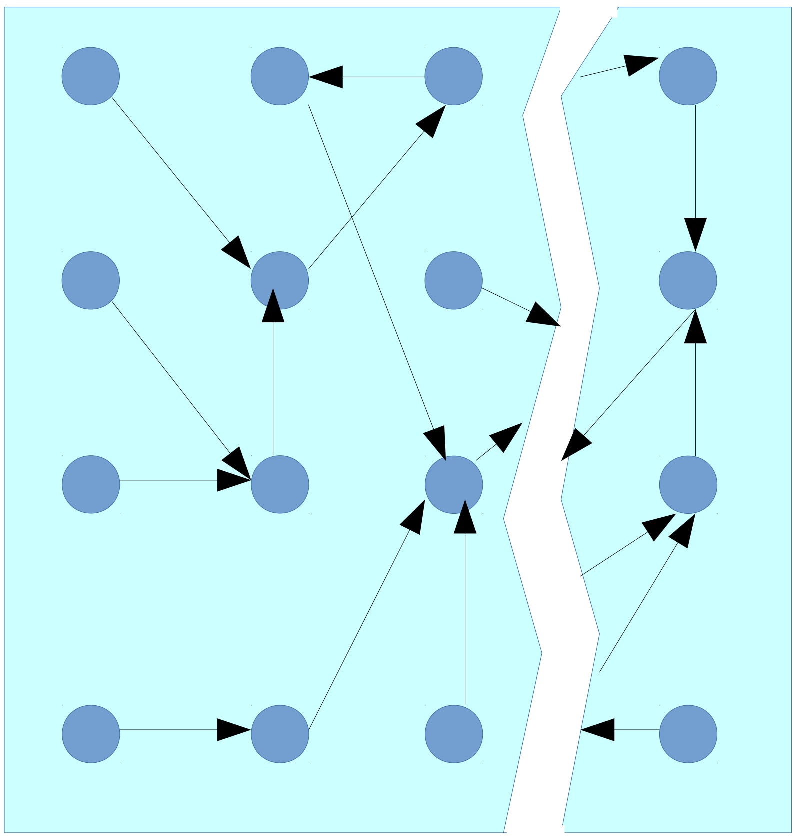
State: pc, x, y, a, b






```
int 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 (a-b);
}
```

Specifying Program Properties



State: pc, x, y, a, b

Pre-condition:
{ a + b >= 0 }

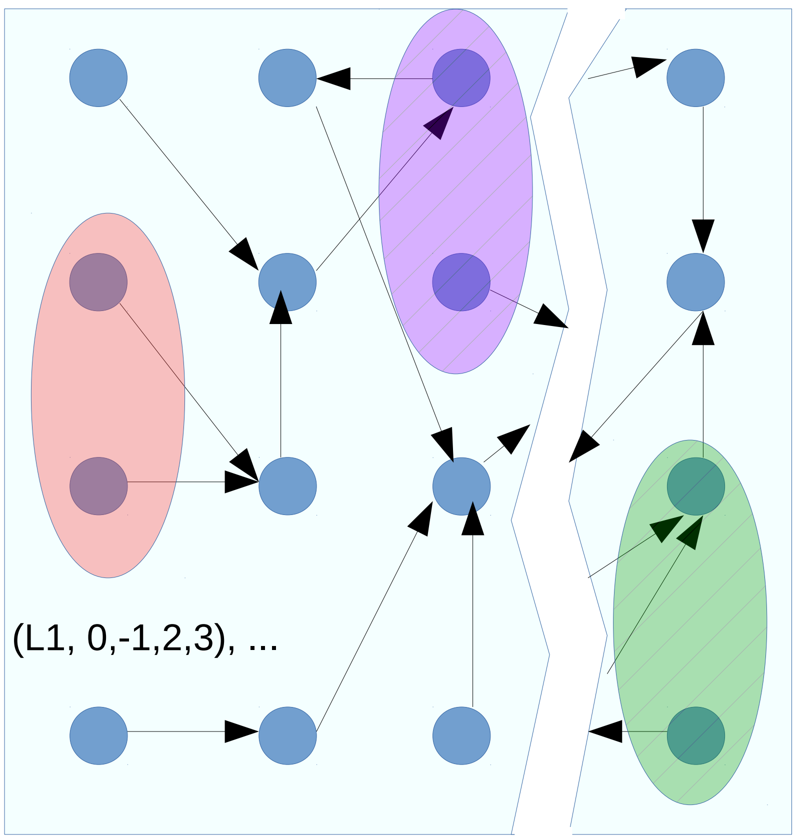
```
int func(int a, int b)
{ int x, y;

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

Post-condition:
{ ret_val <= 1 }

Specifying Program Properties

State: pc, x, y, a, b

(L4, 0,1, 5, 4), ...



(L1, 0,-1,2,3), ...

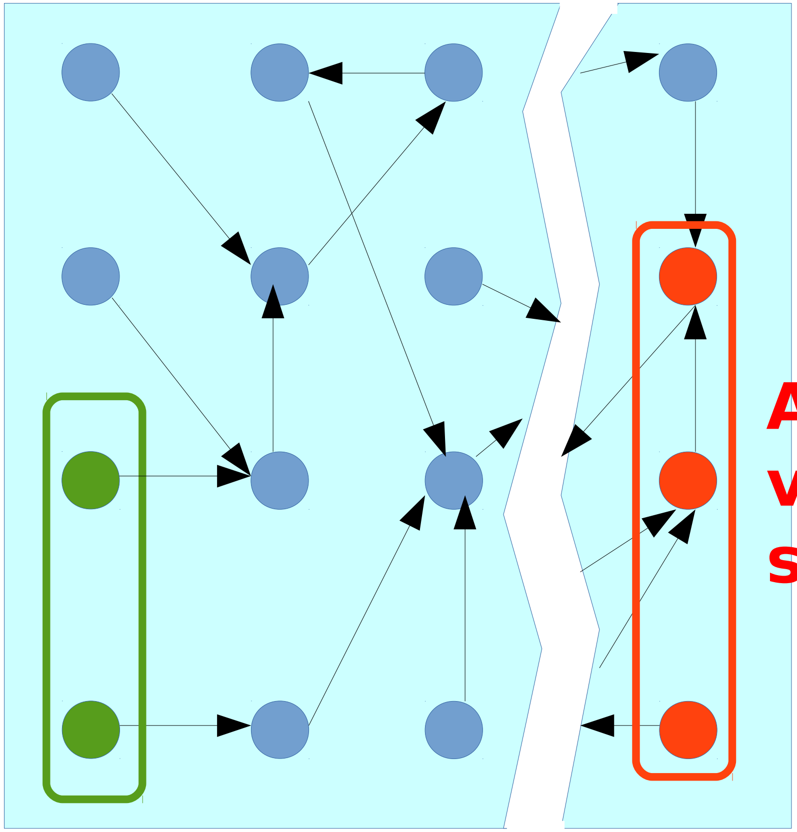
(L6, 0,1, 8, 4), ...

Pre-condition:
{ a + b >= 0 }

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

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

Post-condition:
{ ret_val <= 1 }

Assertion Checking as Reachability



Initial States

Assertion violating states

Path from **initial** to **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

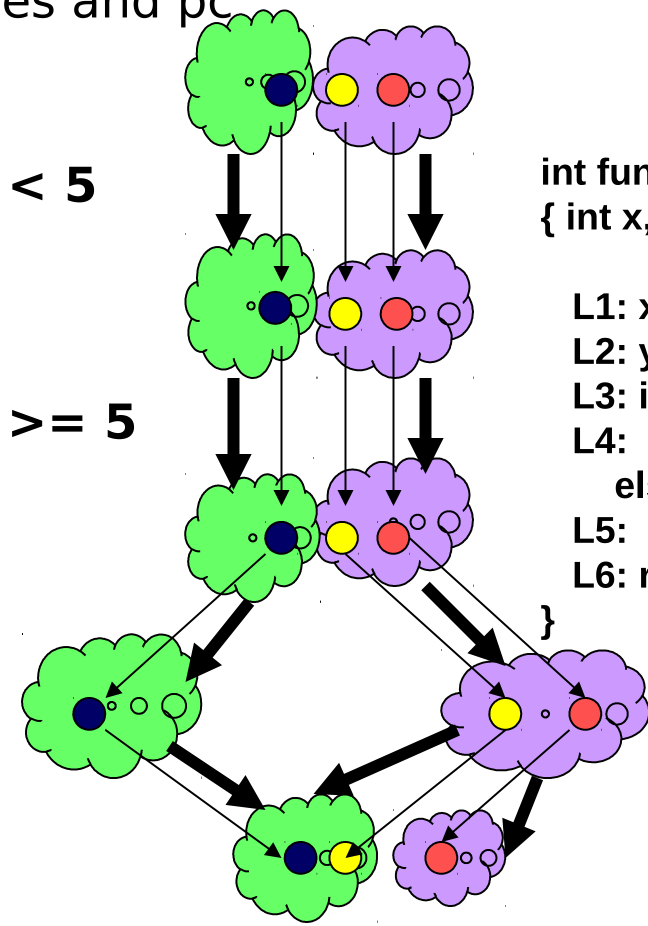
Abstract states

Simple Abstractions

Group states according to values of variables and pc

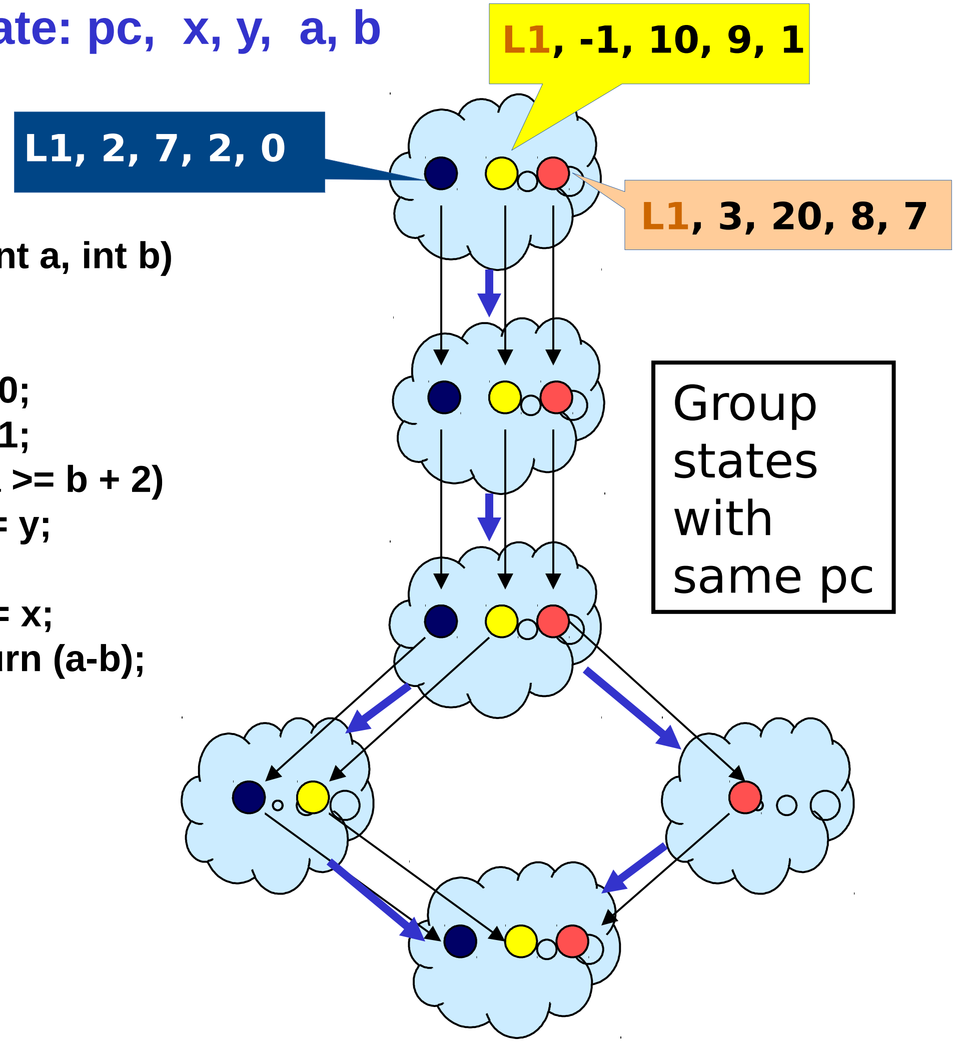
a < 5

a >= 5



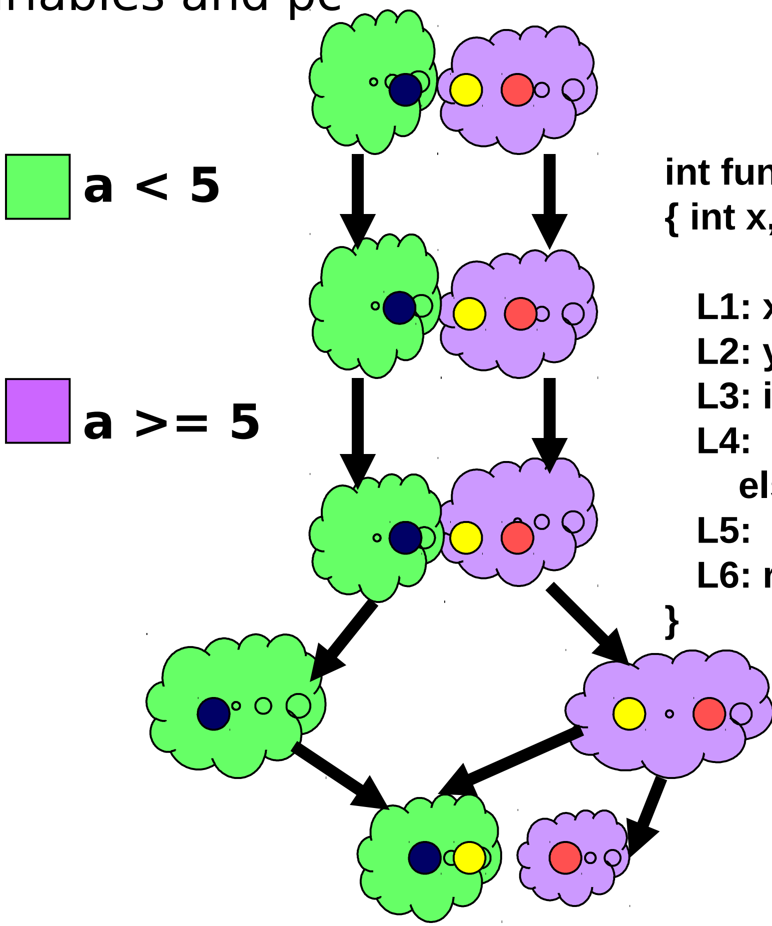
```
int func(int a, int b)
{ int x, y;

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

State: pc, x, y, a, b

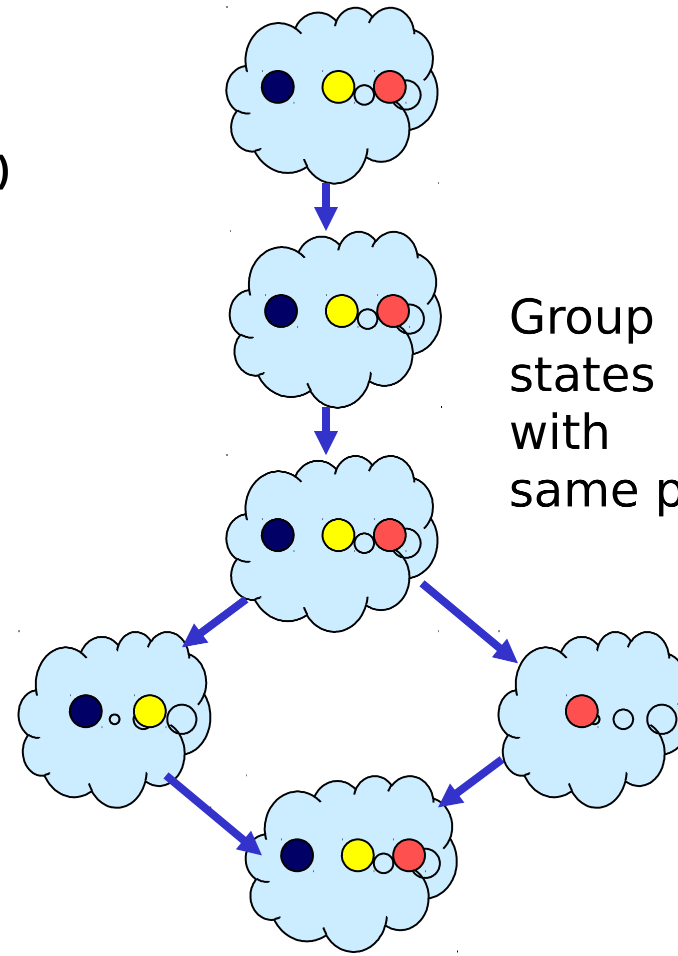


Programs as State Set Transformers

Group states according to values of variables and pc



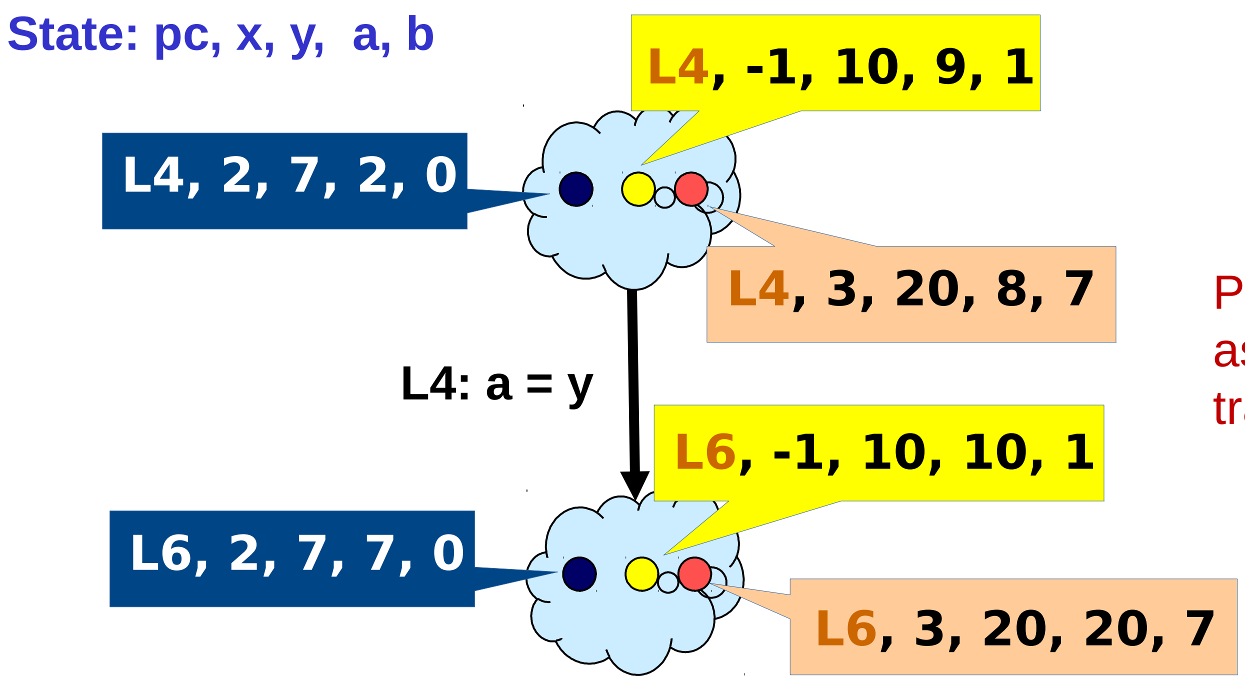

```
int func(int a, int b)
{ int x, y;

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



Group states with same pc

Programs as Abstr State Transformers

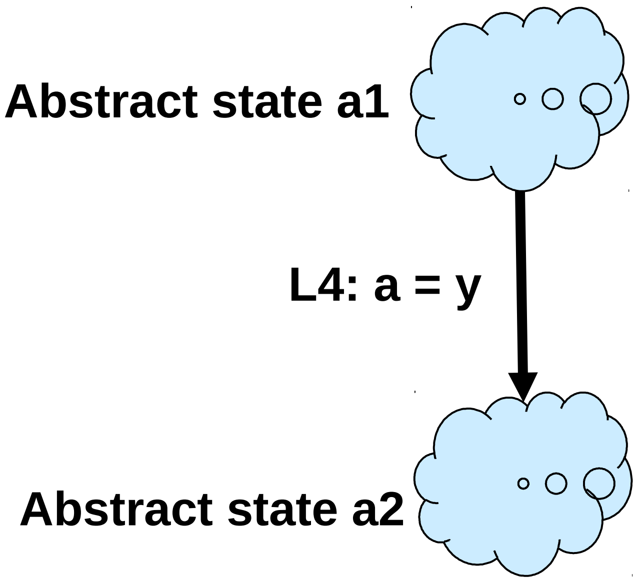
- Recall: Set of (potentially infinite) concrete states is an abstract state
- **Think of program as abstract state transformer**



Program statement as concrete state transformer

Programs as Abstr State Transformers

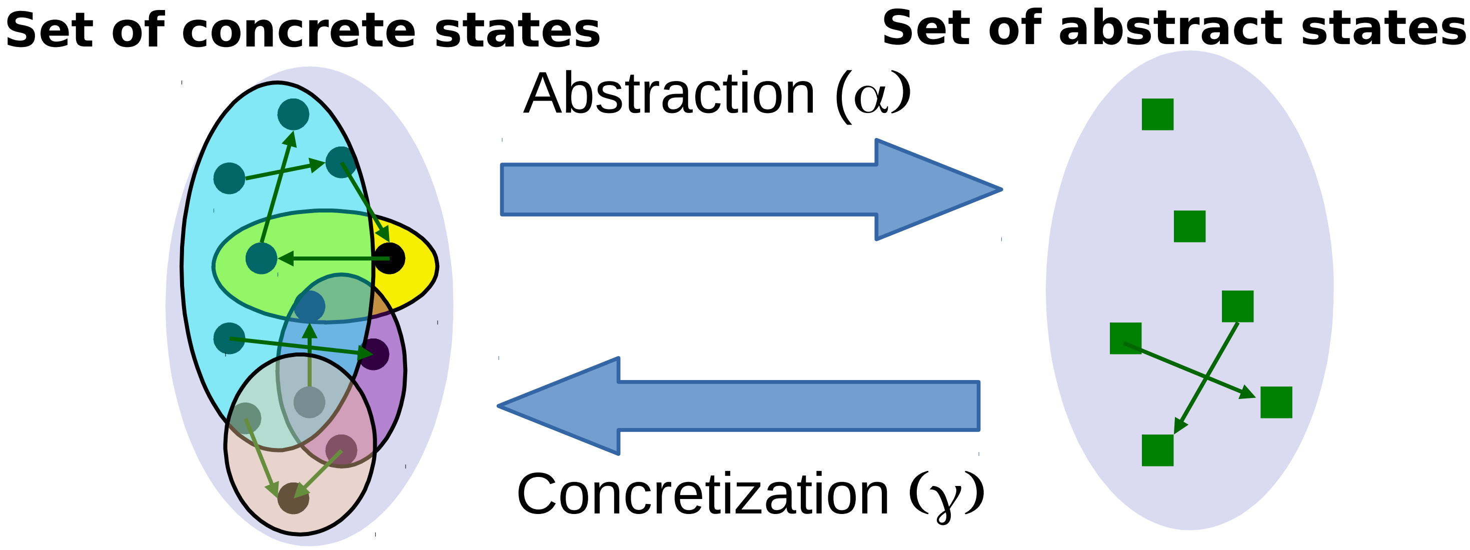
- Recall: Set of (potentially infinite) concrete states is an abstract state
- **Think of program as abstract state transformer**



Program statement as abstract state transformer

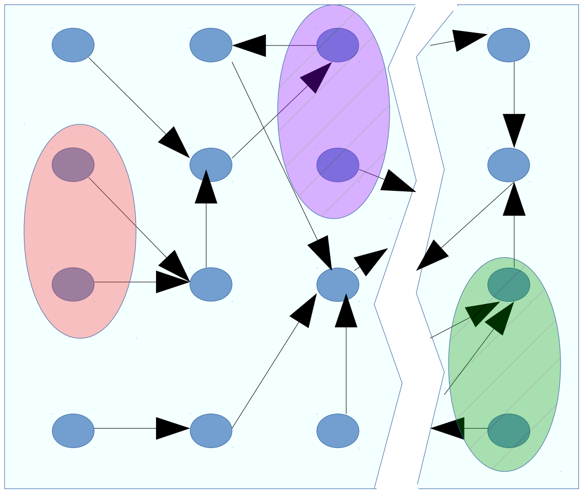
Central problem:
Compute a2 from a1 and prog stmt (abstract state transitions)

A Generic View of Abstraction

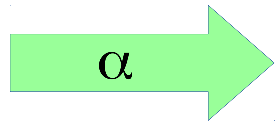


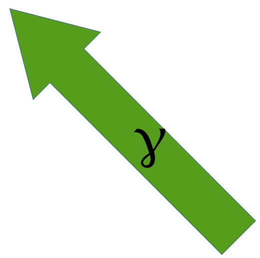
- Every subset of concrete states mapped to unique abstract state
- Desirable to capture containment relations
- Transitions between state sets (abstract states)

The Game Plan

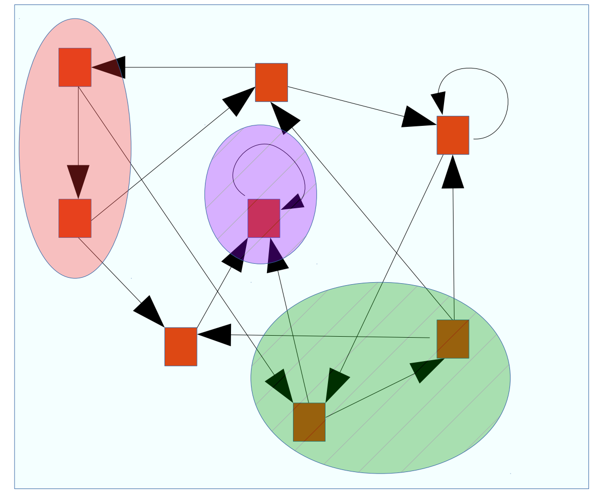


CONCRETE STATES





ABSTRACT STATES

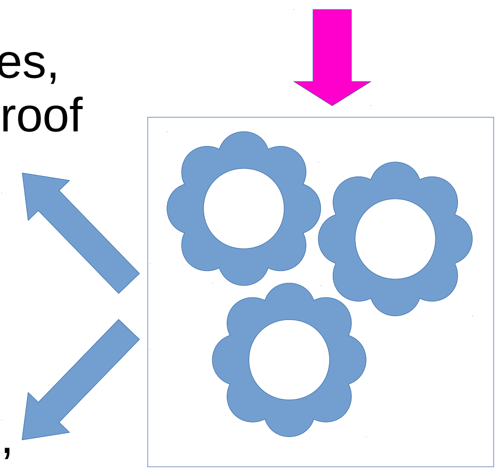


Pre-condition:
{ a + b >= 0 }

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

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

Post-condition:

{ ret_val <= 1 }

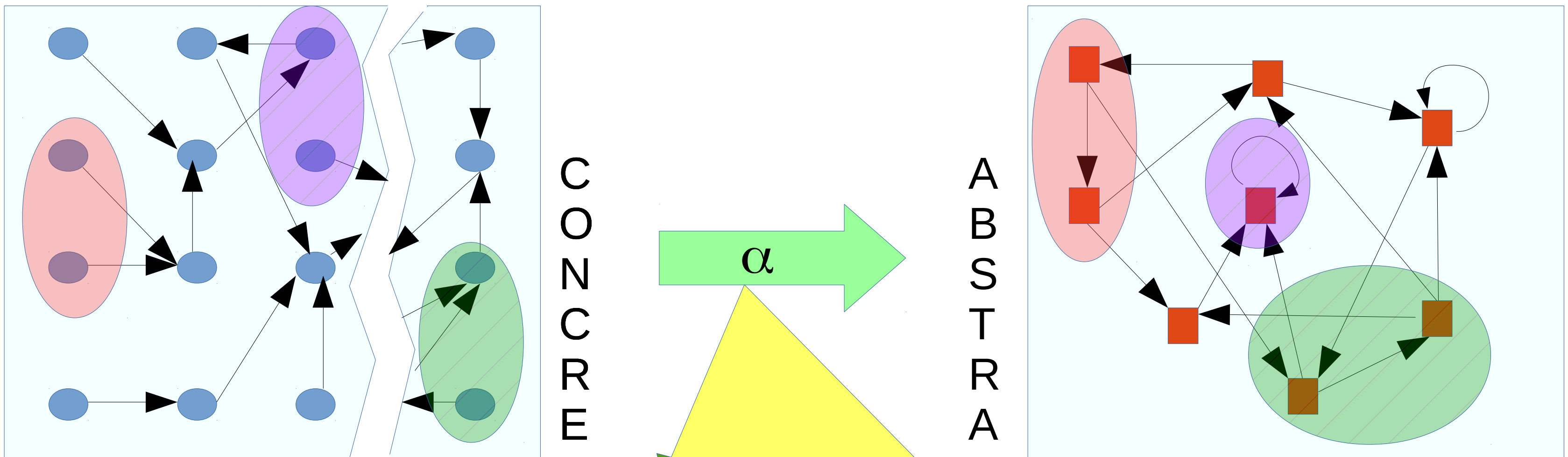


Yes, Proof

No, Counterexample

Abstract analysis engine

The Game Plan



Pre-condition:
{ a + b >= 0 }
int func(int
{ int x, y

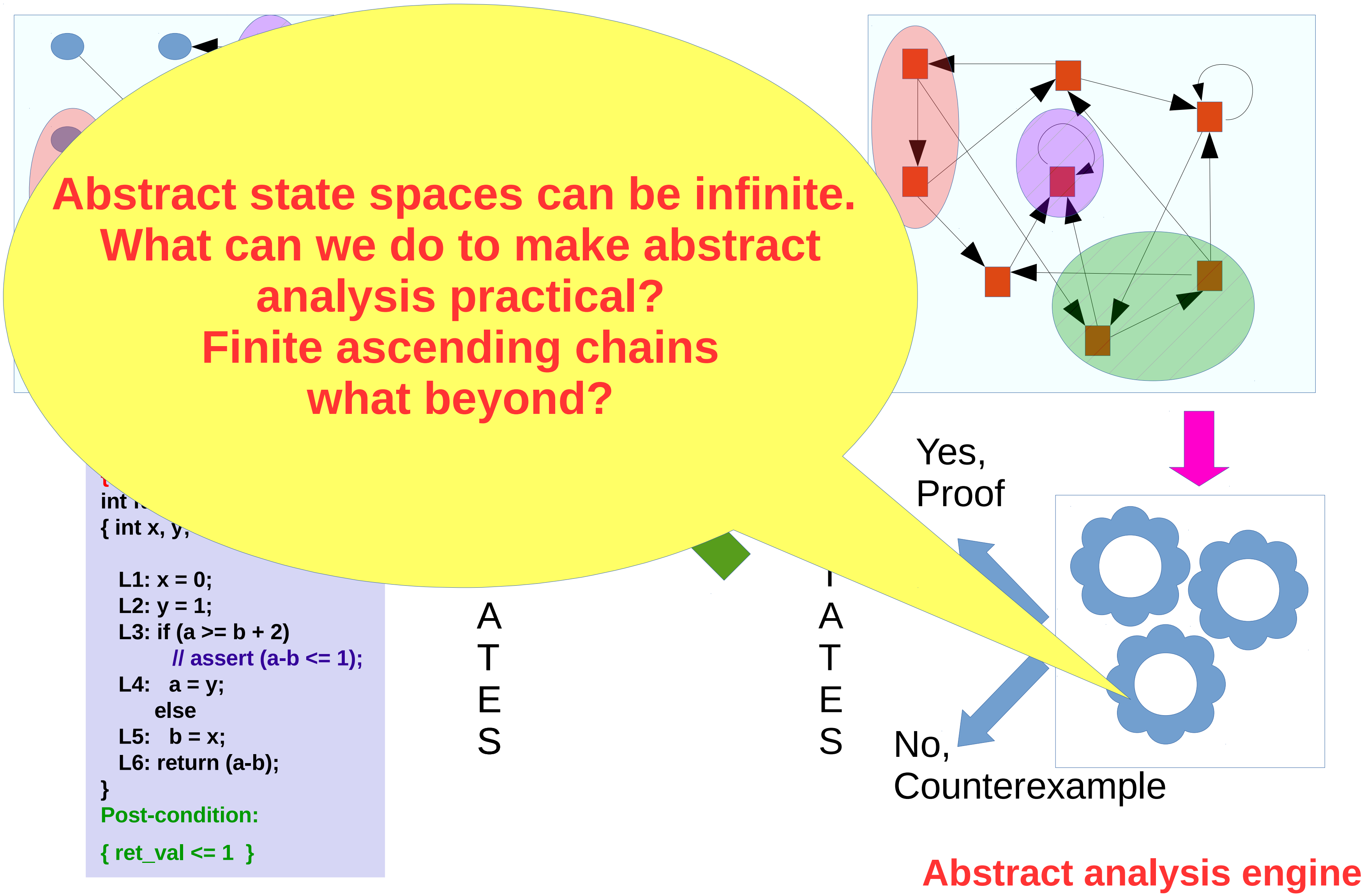
L4:
els
L5: b = x;
L6: return (a-b);
}
Post-condition:
{ ret_val <= 1 }

How do we choose the right abstraction?
Is there a method beyond domain expertise?
Can we learn from errors in abstraction to build better (refined) abstractions?
Can refinement be automated?

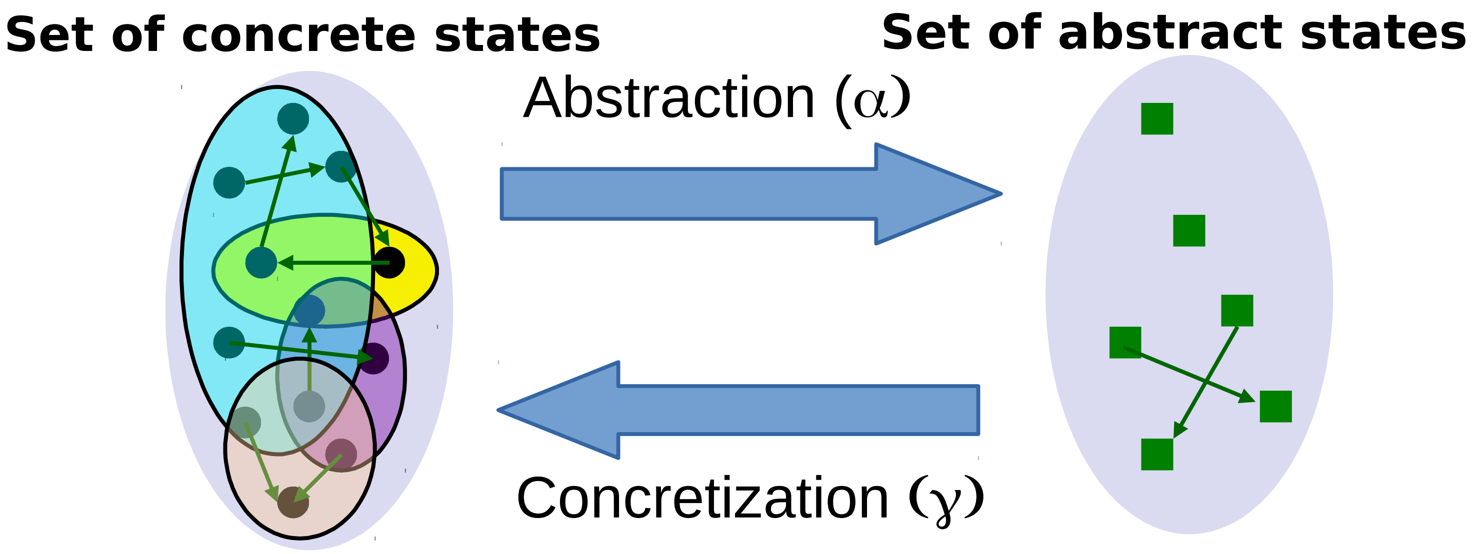
mple

Abstract analysis engine

The Game Plan



Desirable Properties of Abstraction



- Suppose $S_1 \subseteq S_2$: subsets of concrete states
  - Any behaviour starting from S_1 can also happen starting from S_2
  - If $\alpha(S_1) = a_1, \alpha(S_2) = a_2$ we want this monotonicity in behaviour in abstr state space too
    - Need ordering of abstract states, similar in spirit to $S_1 \subseteq S_2$

Structure of Concrete State Space

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

Powerset of S

Partial order

Least upper bound

Greatest lower bound

Top element

Bottom element

Structure of Abstract State Space

- Abstract lattice $\mathcal{A} = (\mathcal{A}, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$
- 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 $a_1, a_2 \in \mathcal{A}$
  - $\gamma(\top) = S, \quad \gamma(\bot) = \emptyset$

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: intervals of values of x, pc implicit
  - [-10, 7]: { (x, y) | -10 <= x <= 7 }
  - $(-\infty, 20]$: { (x, y) | x <= 20 }

- $\sqsubseteq$ relation: Inclusion of intervals
  - $[-10, 7] \sqsubseteq [-20, 9]$
- $\sqcup$ and $\sqcap$: union and intersection of intervals
  - $[-10, 9] \sqcup [-20, 7] = [-20, 9]$
  - $[-10, 9] \sqcap [-20, 7] = [-10, 7]$
- $\bot$ is empty interval of x
- $\top$ is $(-\infty, +\infty)$

Interval Abstract Domain

- Abstract states: intervals of values of x, pc implicit

  [-10, 7]: { (x, y) | -10 <= x <= 7 }

  $(-\infty, 20]$: { (x, y) | x <= 20 }

- $\sqsubseteq$ relation: Inclusion of intervals

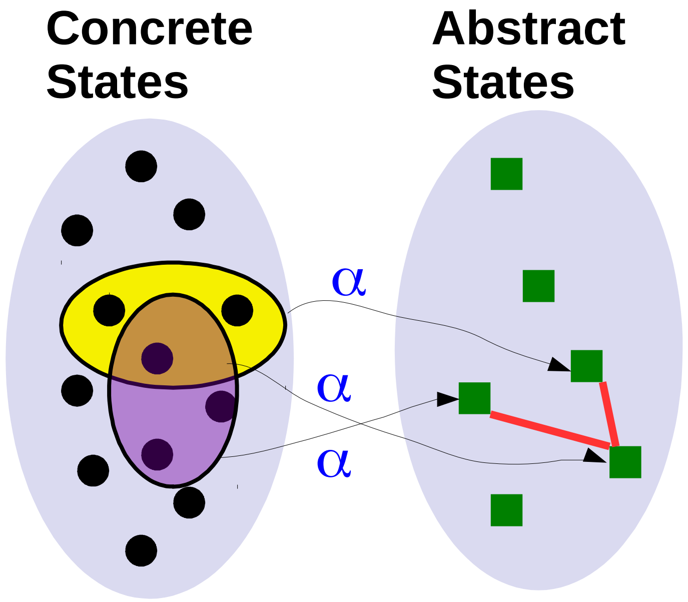
  $[-10, 7] \sqsubseteq [-20, 9]$

- $\sqcup$ and $\sqcap$: union and intersection

  $[-10, 9] \sqcup [-20, 7] = [-20, 9]$

  $[-10, 9] \sqcap [-20, 7] = [-10, 7]$

- $\perp$ is empty interval of x
- $\top$ is $(-\infty, +\infty)$



α({(L1, 1, 3), (L1, 2, 4), (L1, 5, 7)}) = [1, 5]

α({(L1, 5, 7), (L1, 7, 6), (L1, 9, 10)}) = [5, 9]

α({(L1, 5, 7)}) = [5, 5]

Interval Abstract Domain

- Abstract states: pairs of intervals (one for x, y), pc implicit
  - ([-10, 7] , (-∞, 20])
  - $\sqsubseteq$ relation: Inclusion of intervals

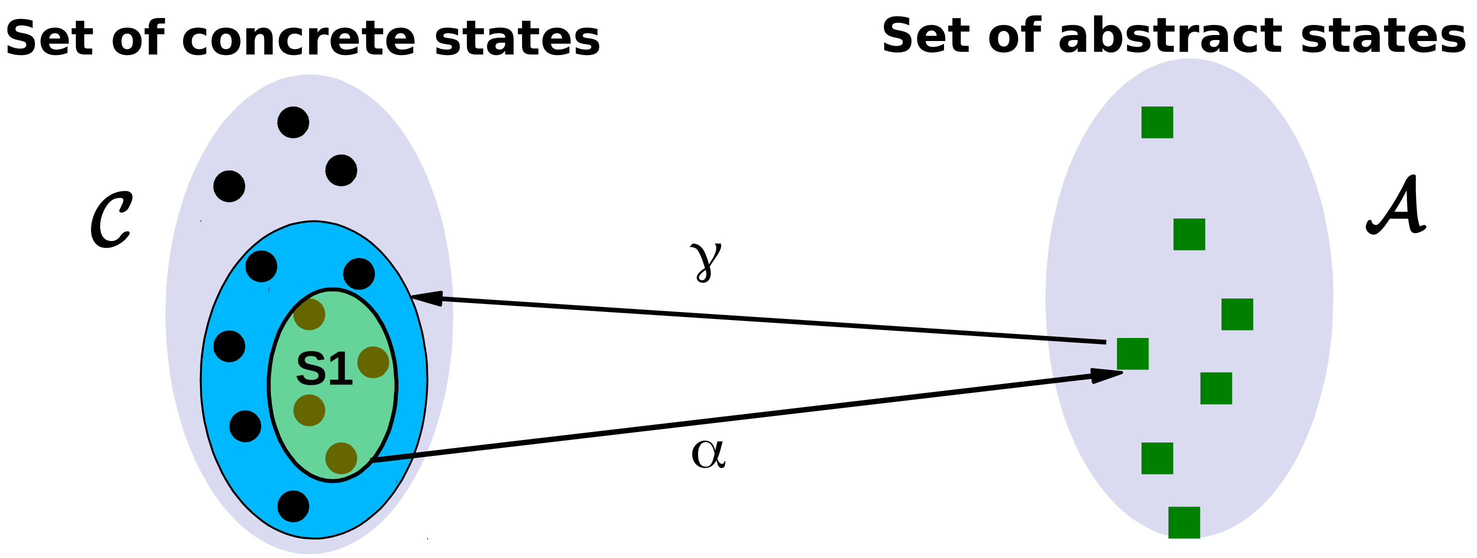
    ([-10, 7] , (-∞, 20]) $\sqsubseteq$ ([-20, 9], (-∞, +∞))
  - $\sqcup$ and $\sqcap$: union and intersection of intervals

    ([-10, 9] , (-∞, 20]) $\sqcap$ ([-20, 7], [3,+∞)) = ([-10, 7], [3, 20])

    ([-10, 9], (-∞, 20]) $\sqcup$ ([-20, 7], [3,+∞)) = ([-20, 9],(-∞,+∞))
  - $\bot$ is empty interval of x and y
  - $\top$ is ((-∞, +∞), (-∞, +∞))

Desirable Properties of α and γ

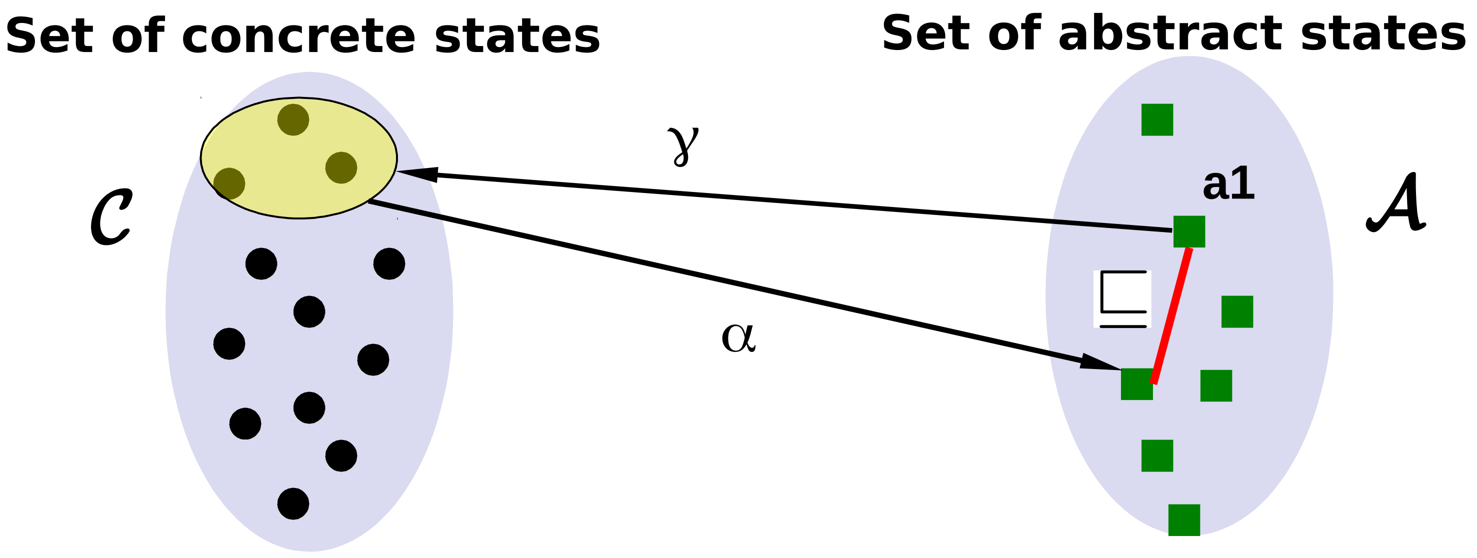
For all $S_1 \subseteq \mathcal{C}$ $\quad S_1 \subseteq \gamma(\alpha(S_1))$



Desirable Properties of α and γ

$$S_1 \subseteq \gamma(\alpha(S_1)) \quad \text{forall} \quad S_1 \subseteq \mathcal{C}$$

$$\alpha(\gamma(a_1)) \sqsubseteq a_1 \quad \text{forall} \quad a_1 \in \mathcal{A}$$

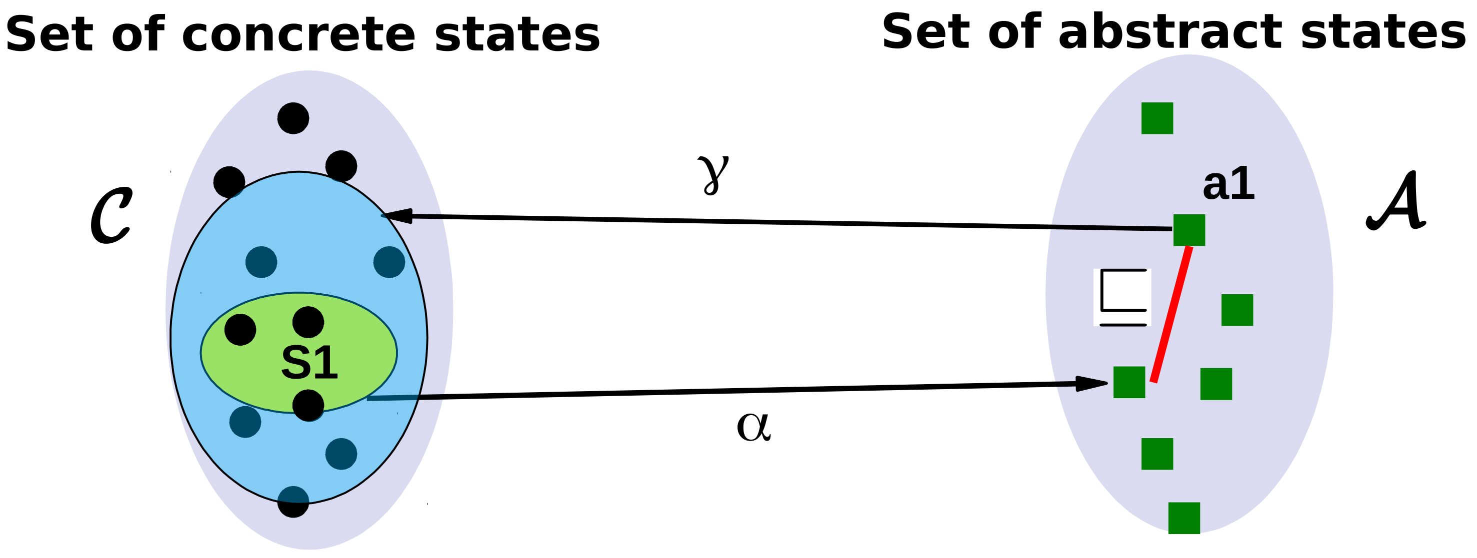


α and γ form a Galois connection

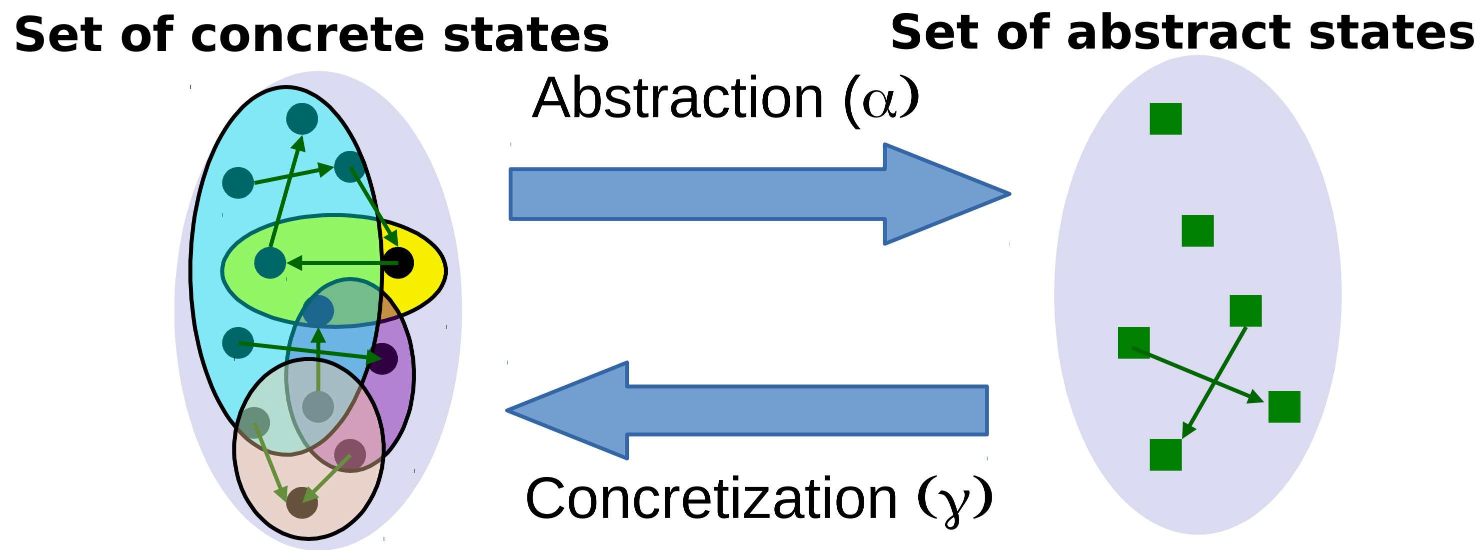
Desirable Properties of α and γ

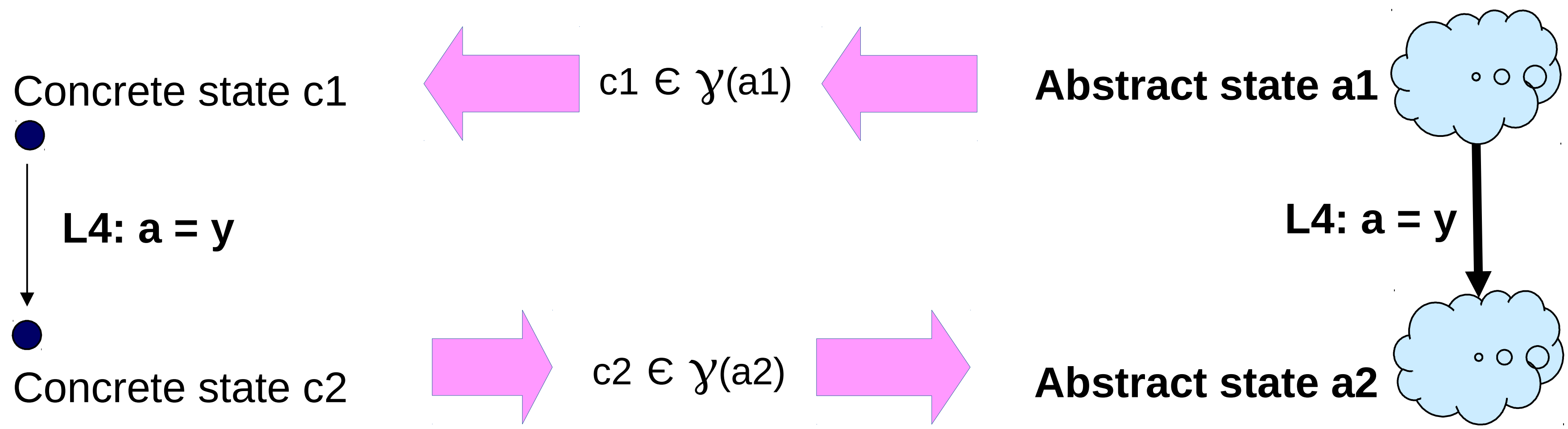
- α and γ form a Galois connection
  - Second (equivalent) view:

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



Computing Abstract State Transitions

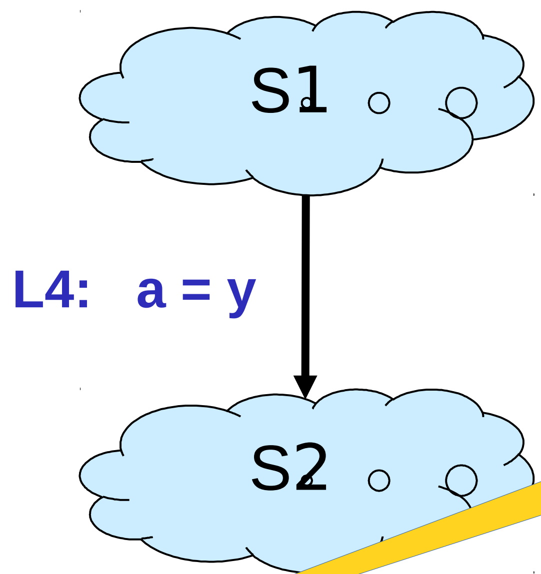




Computing Abstract State Transitions

- Concrete state set transformer function
  - Example:

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



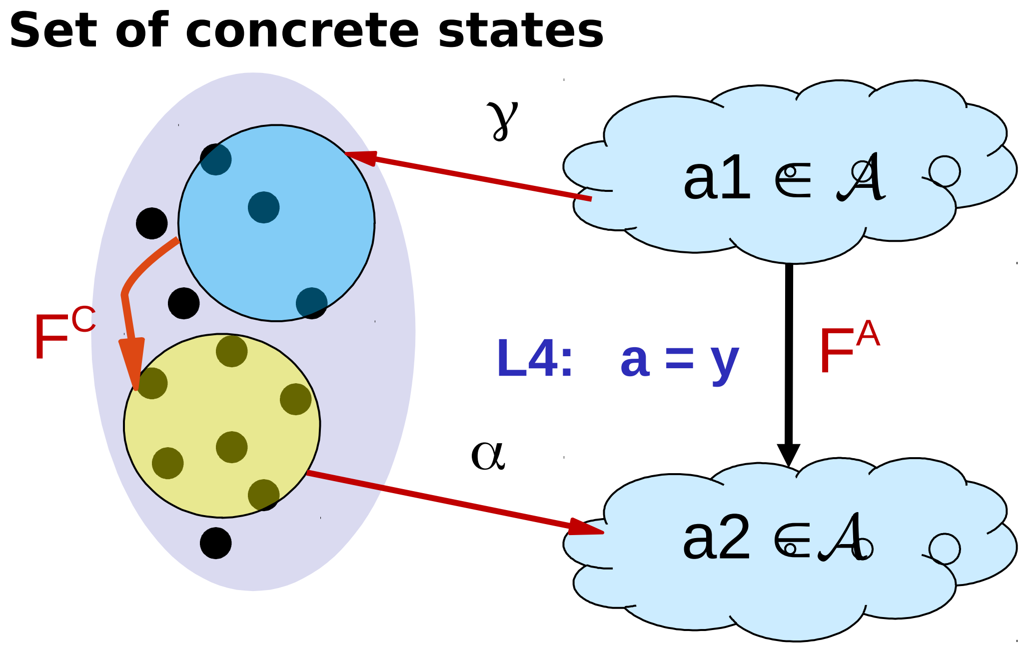
Monotone concrete state set transformer function for stmt at L4

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

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

Computing Abstract State Transitions

- Abstract state transformer function
  - Example:



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

Example Abstr State Transition

L0: x = 0; y = 0;

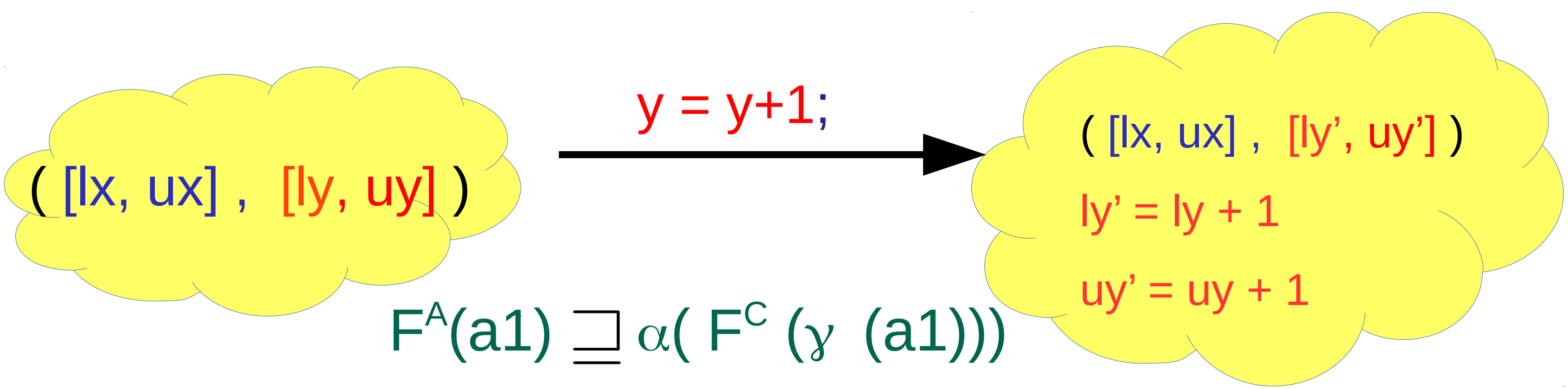
L1: while (x < 100) do

    L2: x = x+1;

    L3: y = y+1;

L4: end while

Abstract states: pairs of intervals (one for x, y), pc implicit



$$F^A(a1) \sqsupseteq \alpha(F^C (\gamma\ (a1)))$$

Example Abstr State Transition

L0: x = 0; y = 0;

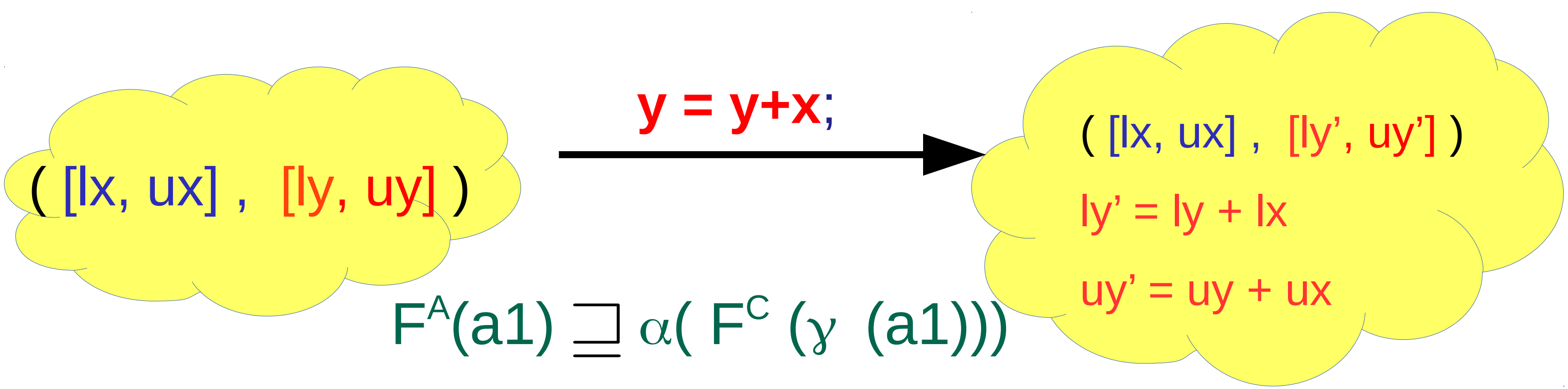
L1: while (x < 100) do

  L2: x = x+1;

  L3: **y = y+x**;

L4: end while

Abstract states: pairs of intervals (one for x, y), pc implicit



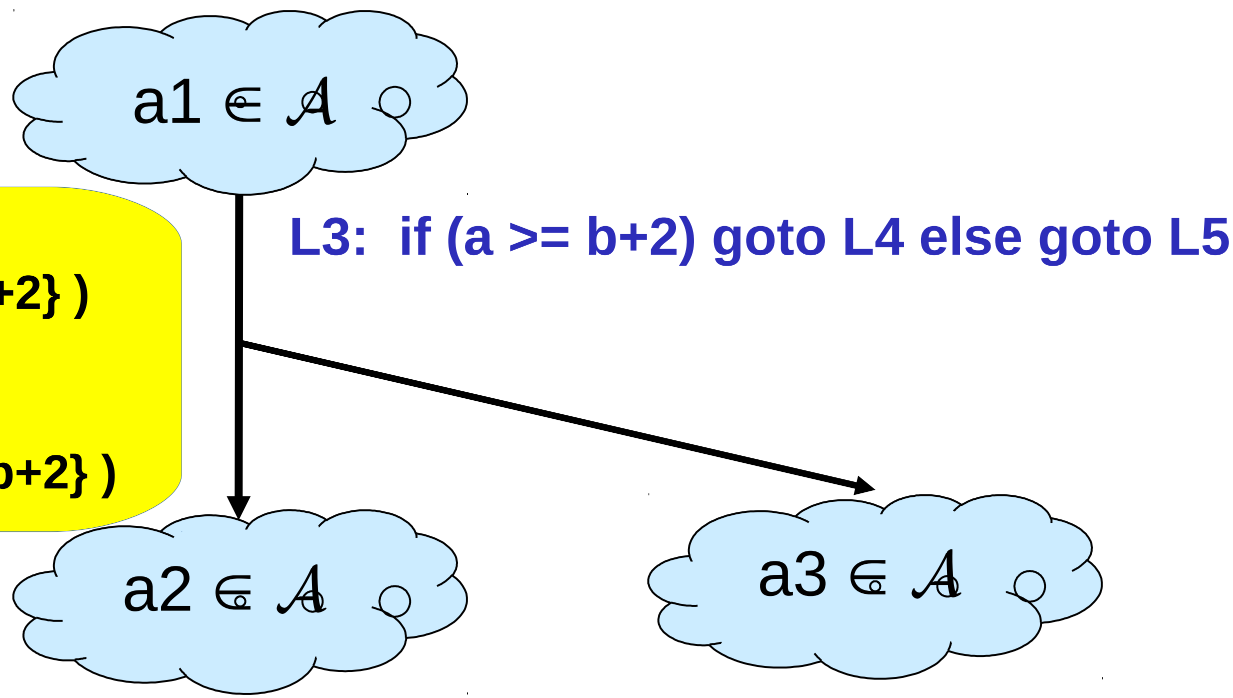
$F^A(a1) \sqsupseteq \alpha(F^C (\gamma\ (a1)))$

Computing Abstract State Transitions

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

acond =
α({(x, y, a, b) | a >= b+2})

acondb =
α ({(x, y, a, b) | a < b+2})



a2 = a1 ⊓ acond
pc in a2: L4

a3 = a1 ⊓ acondb
pc in a3: L5

Dealing with Loops

Abstract pre-cond: a0

L0: a = 0; b = 0;

L1: ……. ;

⋮

L7: while (a > b) do

L8: ….. ;

⋮

L19:…… ;

Loop Body

L20: end while

L21: ……;

⋮

L100: ……;

Dealing with Loops

Abstract state: a1 = F^{A}_{0} (a0)

L0: a = 0; b = 0;

L1: ……. ;

⋮

L7: while (a > b) do

L8: ….. ;

⋮

L19:….. ;

Loop Body

L20: end while

L21: ……;

⋮

L100: ……;

Dealing with Loops

Abstract state: $a7 = F^{A}_{1..7}(a1)$

L0: a = 0; b = 0;

L1: ……. ;

⋮

L7: while (a > b) do

L8: ….. ;

⋮

L19:…… ;

Loop Body

L20: end while

L21: ……;

⋮

L100: ……;

Dealing with Loops

L0: a = 0; b = 0;

L1: ……. ;

⋮

L7: while (a > b) do

L8: ….. ;

⋮

L19:….. ;

Loop Body

L20: end while

L21: ……;

⋮

L100: ……;

α(….) = acond

Abstract state a20 ?
Can't be computed as
$F^{A}_{8..19}$(a7 □ acond)

Loop may iterate 0,1,2,... times

Dealing with Loops

Calculate **abstract loop invariant a7*** at L7. Whenever L7 is reached in program, corresponding abstr state $\sqsubseteq$ **a7***

L0: a = 0; b = 0;

L1: ……. ;

⋮

α(not ...) = acondb

L7: while (a > b) do

L8: ….. ;

⋮

L19:….. ;

Loop Body

L20: end while

Abstract state a20 = (**a7*** $\sqcap$ acondb)

L21: ……;

⋮

L100: ……;

Dealing with Loops

L0: a = 0; b = 0;

L1: ……. ;

⋮

L7: while (a > b) do

L8: ….. ;

⋮

L19:…... ;

Loop Body

L20: end while

Abstract state: a21 = a20

L21: ……;

⋮

L100: ……;

Dealing with Loops

L0: a = 0; b = 0;

L1: ……. ;

⋮

L7: while (a > b) do

L8: ….. ;

⋮

L19:….. ;

Loop Body

L20: end while

L21: ……;

⋮

L100: ……;

Abstract state:
$a100 = F^{A}_{21..100}(a21)$

Loops can be handled if we know how to compute abstract loop invariants

Computing Abstract Loop Invariant

- Example: ....

  L7 : while (a > b) do

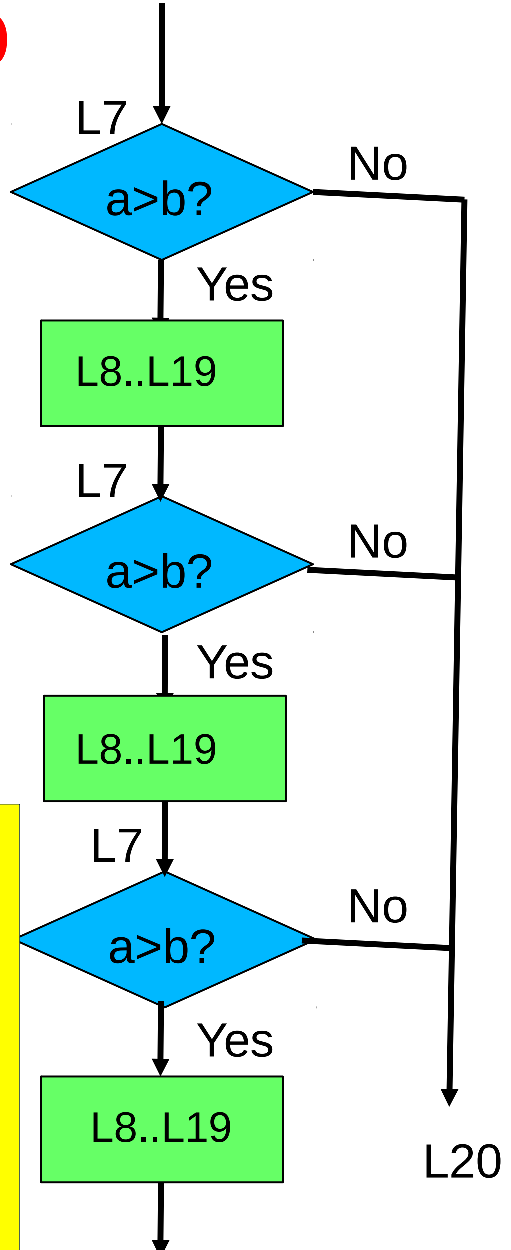
  L8: ......;

  ⋮

  L19: ......;

  Loop Body

  L20: end while



Given

F^A : abstr state transformer of loop body L8...L19

a : abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?

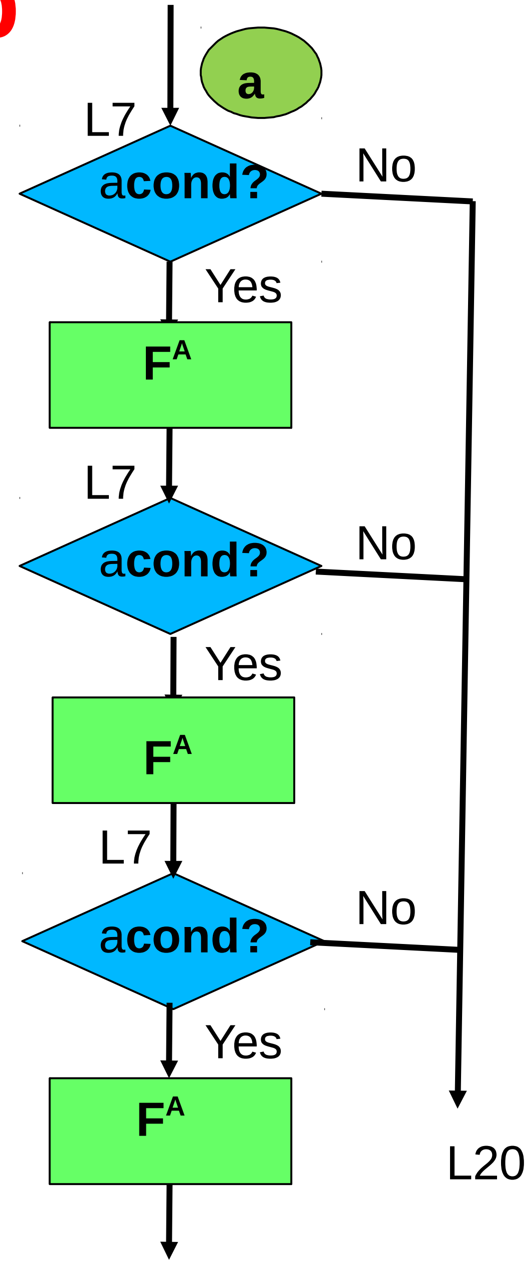
Computing Abstract Loop Invariant

Given

F^A : abstr state transformer of loop body,

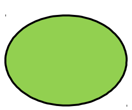
a : abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?



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

Current view of abstract loop invariant



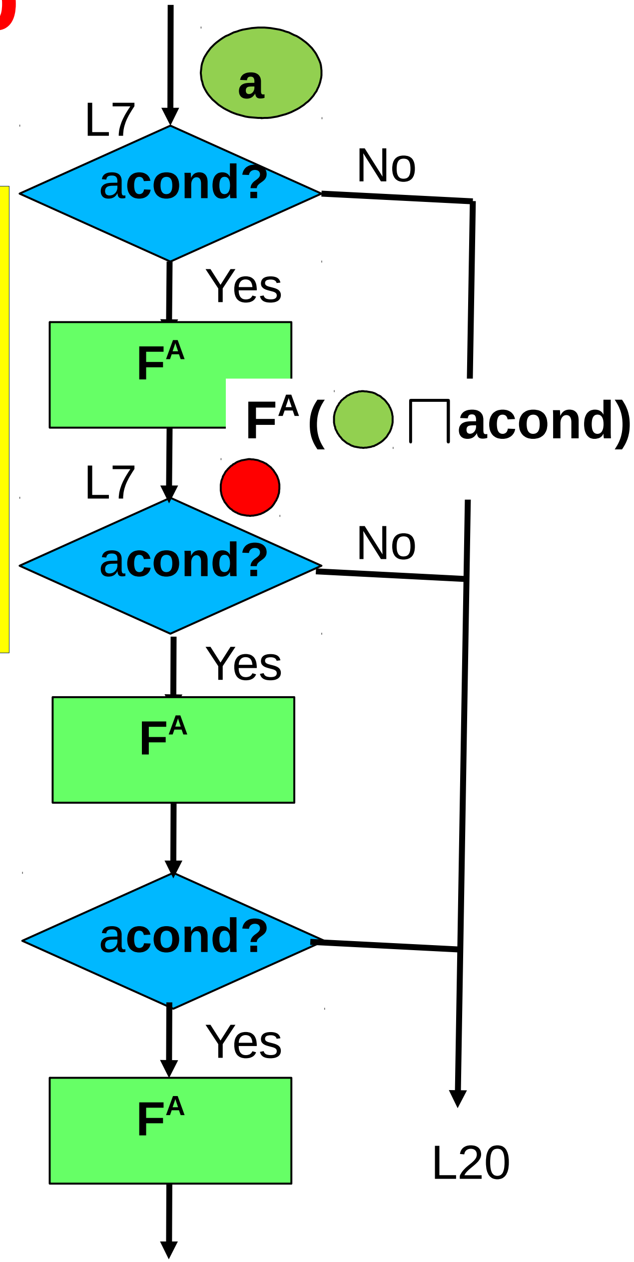
Computing Abstract Loop Invariant

Given

F^A : abstr state transformer of loop body,

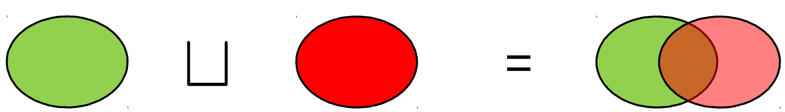
a : abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?



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

Current view of abstract loop invariant



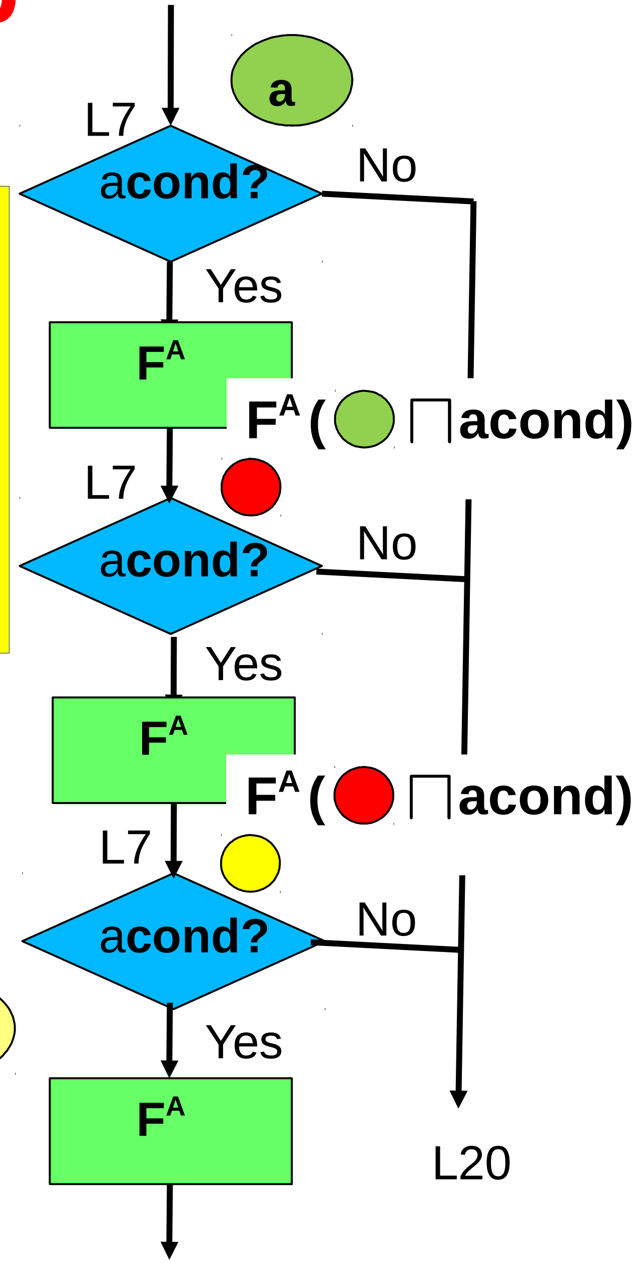
Computing Abstract Loop Invariant

Given

F^A : abstr state transformer of loop body,

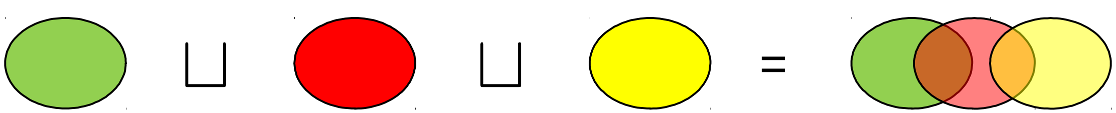
a : abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?



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

Current view of abstract loop invariant



Recall: Meet-over-paths

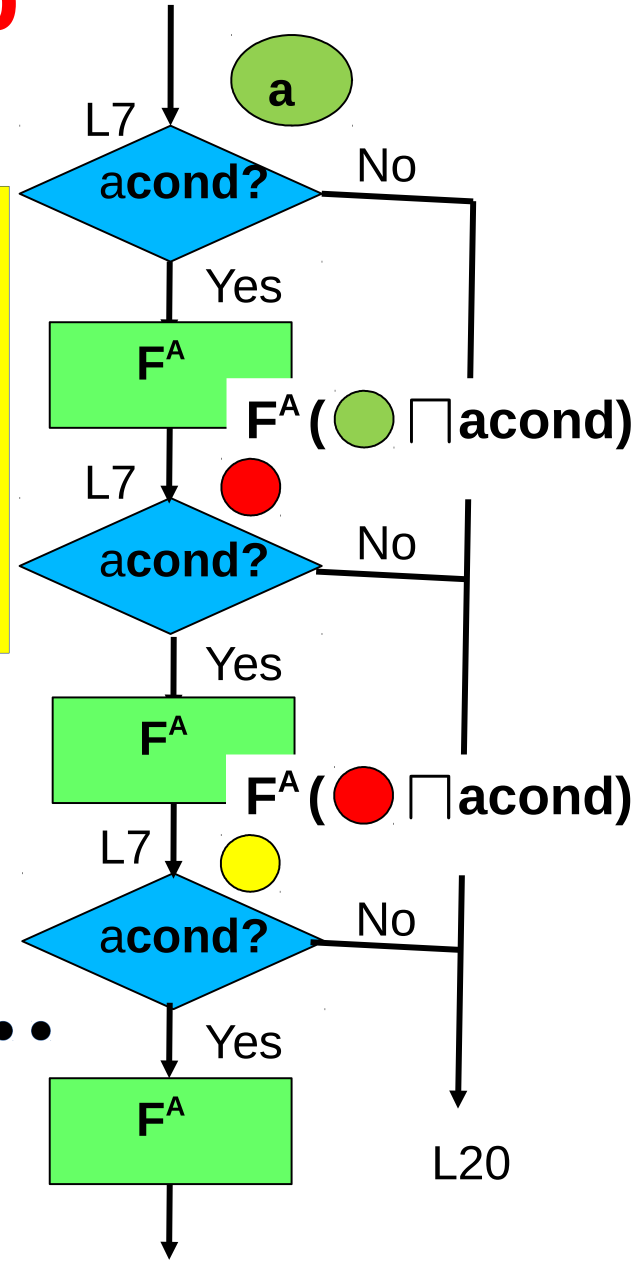
Computing Abstract Loop Invariant

Given

F^A : abstr state transformer of loop body,

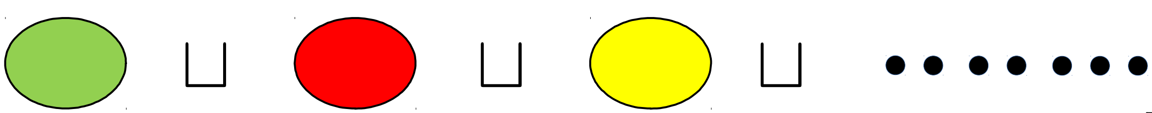
a : abstr state at L7 the first time L7 is reached

What is the abstract loop invariant at L7?



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

Abstract loop invariant



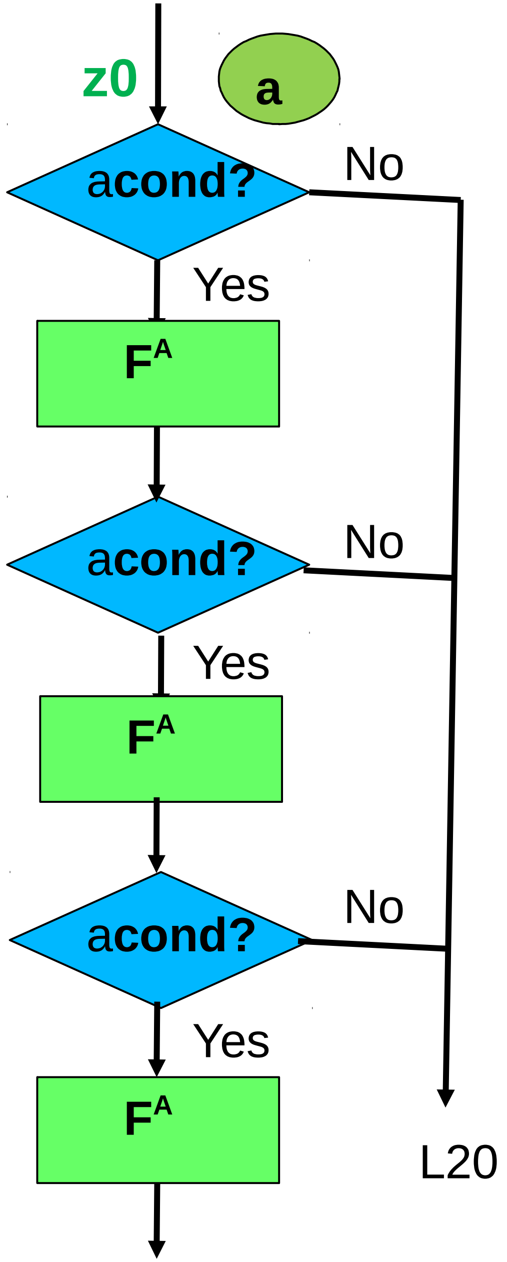
How do we calculate this effectively without knowing bound of loop iterations?

Abstract Loop Invariant: Another view

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

Successive views of of loop invariant at L7:

z0 = a



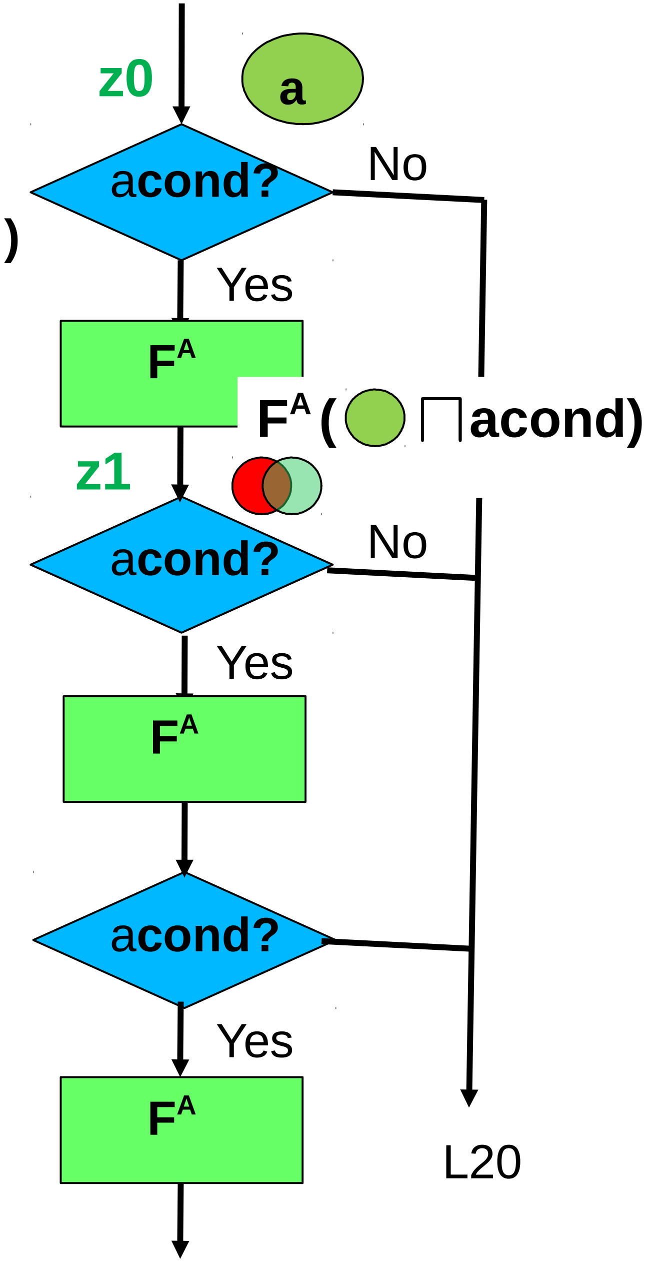
Abstract Loop Invariant: Another view

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

Successive views of of loop invariant at L7:

z0 = a

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

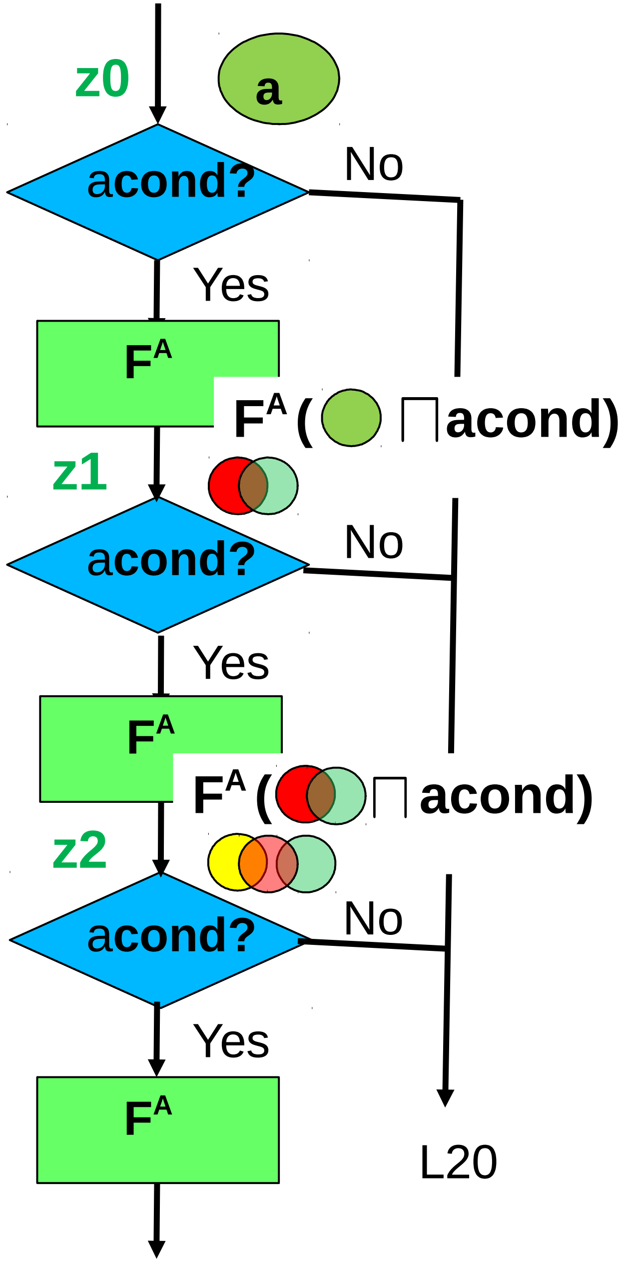


Abstract Loop Invariant: Another View

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

Successive views of of loop invariant at L7:

- $z0 = a$
- $z1 = a \sqcup F^A\ (z0 \sqcap \text{acond})$
- $z2 = a \sqcup F^A\ (z1 \sqcap \text{acond})$



Abstract Loop Invariant: Another View

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

Successive views of of loop invariant at L7:

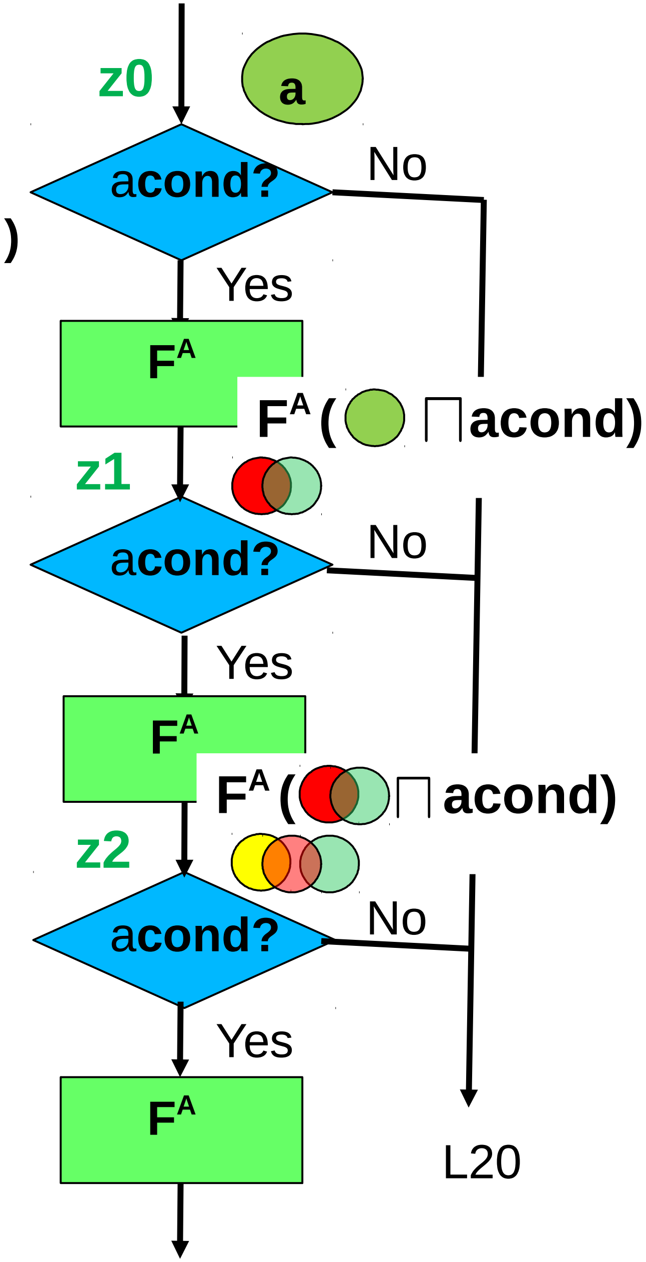
$z0 = a$

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

$z2 = a \sqcup F^A\ (z1 \sqcap acond)$

……

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



Abstract Loop Invariant: Another View

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

Successive views of of loop invariant at L7:

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

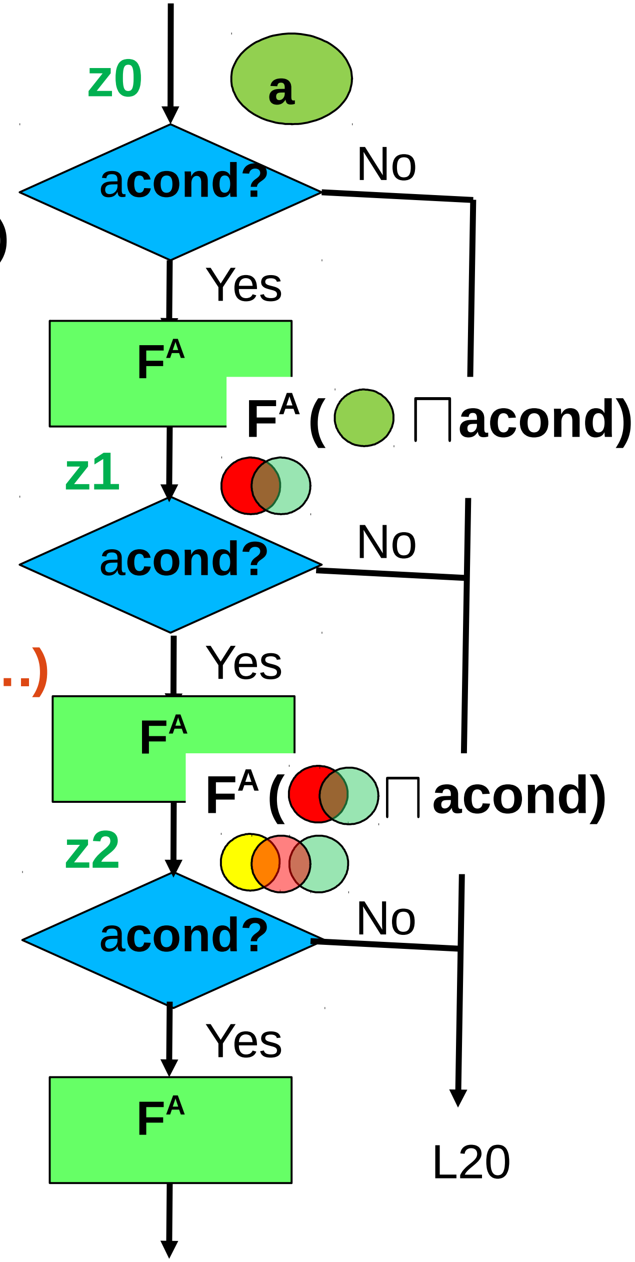
$z1 = a \sqcup F^A(z0 \sqcap acond)$ $= g(g(\bot))$

$z2 = a \sqcup F^A(z1 \sqcap acond)$ $= g(g(g(\bot)))$

....

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

$z0 \sqsubseteq z1 \sqsubseteq z2 \sqsubseteq \ldots$



Reasonable requirements:

$F^A(\bot) = \bot$

If $a1 \sqsubseteq a2$ then $F^A(a1) \sqsubseteq F^A(a2)$

$g(z) = a \sqcup F^A(z \sqcap acond)$

g( ) monotone

Abstract Loop Invariant: Another View

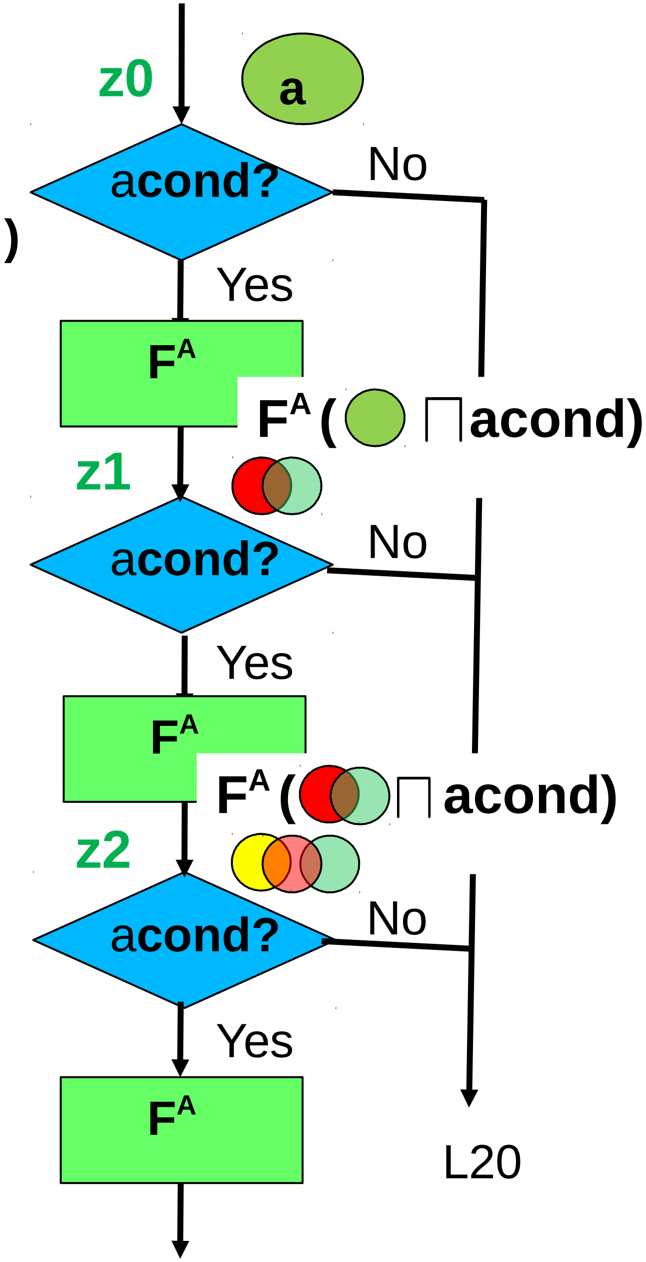
acond = α ({s | s is a concrete state with a > b })

Successive views of of loop invariant at L7:

$z0 = g(\bot)$
$z1 = g(g(\bot))$
$z2 = g(g(g(\bot)))$
…….
$z_i = g(\ldots.g(\bot)\ldots.)$

Abstract loop invar = $\lim_{i \to \infty} g^{(i)}(\bot)$

Reasonable requirements:
$F^A(\bot) = \bot$
If a1 $\sqsubseteq$ a2 then $F^A(a1) \sqsubseteq F^A(a2)$



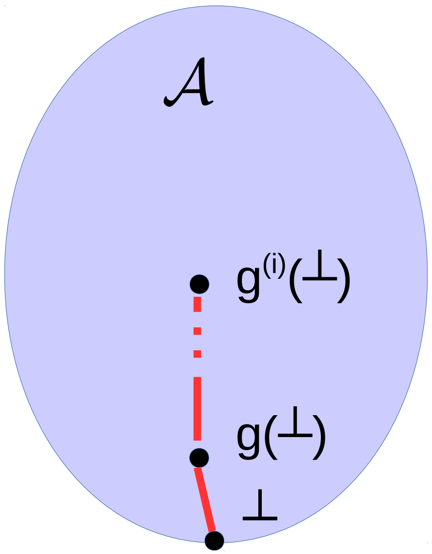
$g(z) = a \sqcup F^A(z \sqcap acond)$
g( ) monotone

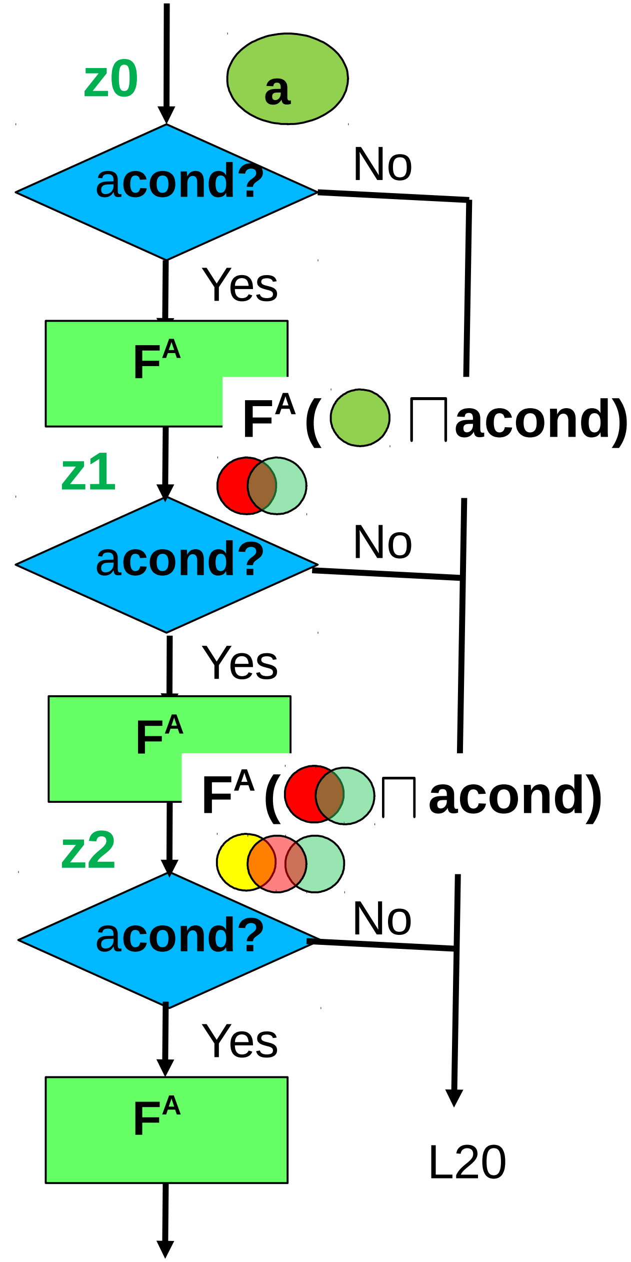
Abstract Loop Invariant: Another View

Abstract loop invar = $\lim_{i \to \infty} g^{(i)}(\bot)$

= smallest a* s.t. g(a*) = a*

= “least fixed point” of g( )





Reasonable requirements:

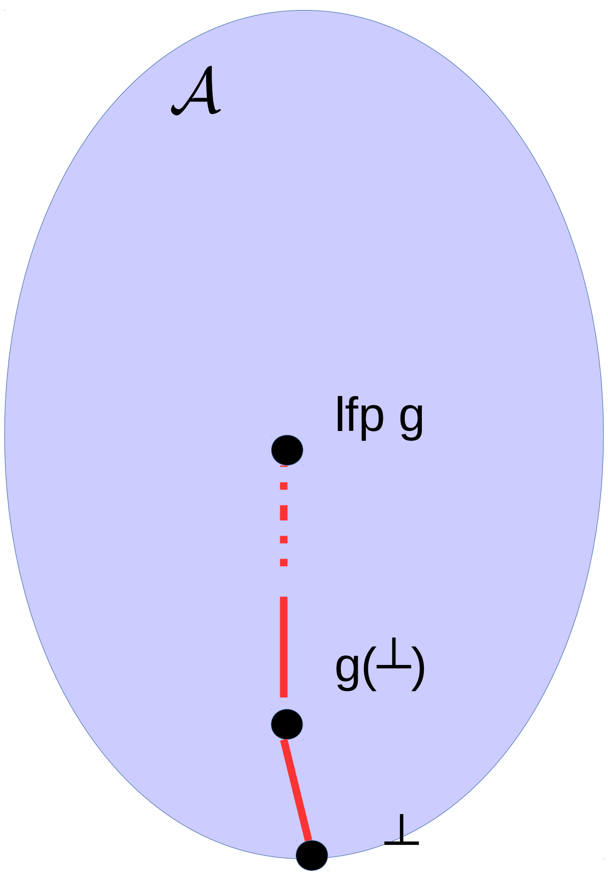
$F^A(\bot) = \bot$

If a1 $\sqsubseteq$ a2 then F^A(a1) $\sqsubseteq$ F^A(a2)

$g(z) = a \sqcup F^A(z \sqcap acond)$

g( ) monotone

Abstract Loop Invariant: Least Fixed Point View



Abstract loop invar a* computable if $\mathcal{A}$ has no infinite ascending chains

What if there are infinite ascending chains?
Can we at least compute an overapprox of a*?

Observe the sequence

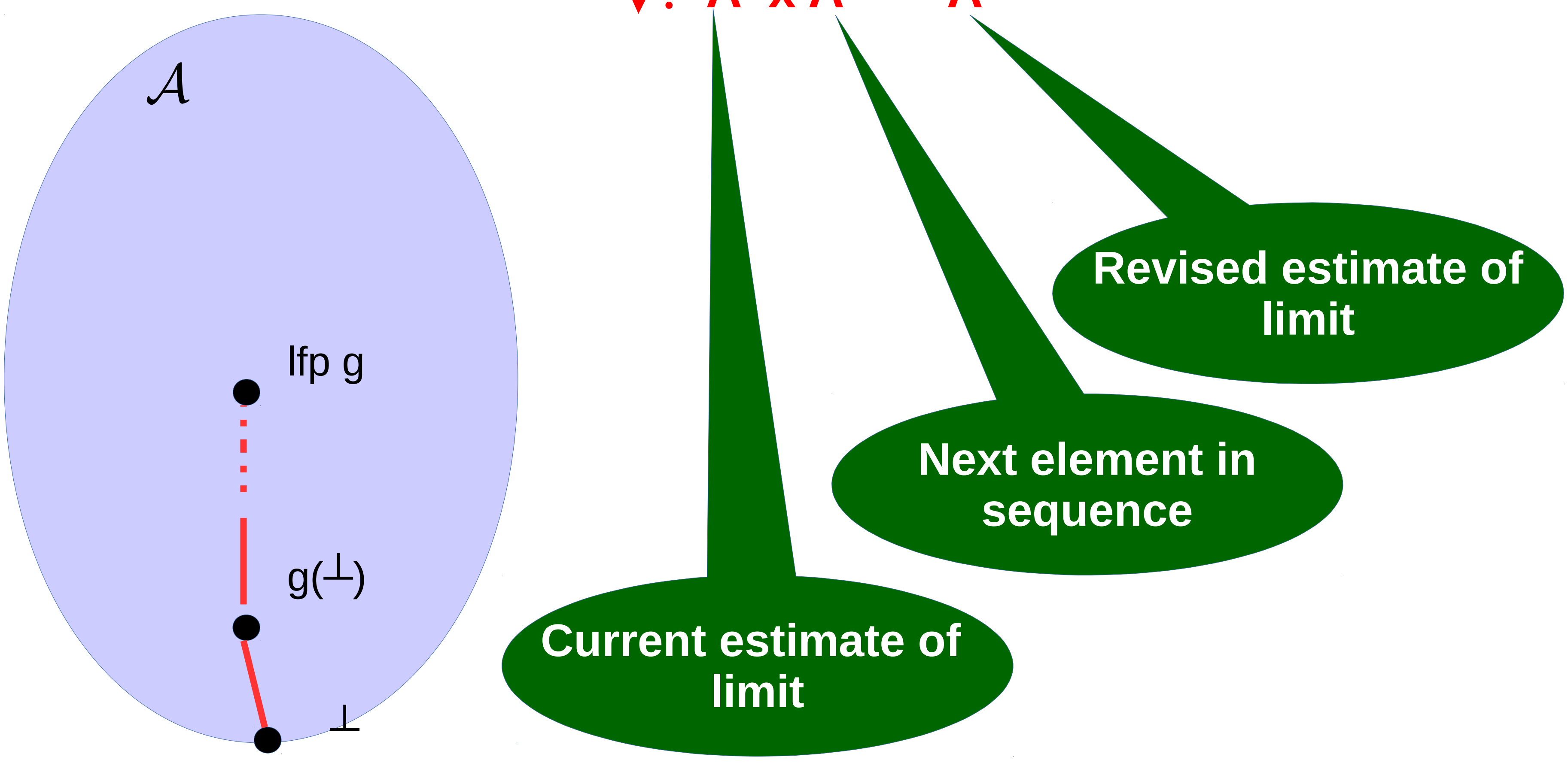
$g(\bot) \sqsubseteq g^2(\bot) \sqsubseteq \ldots \sqsubseteq g^{(i)}(\bot)$ upto i terms

and extrapolate ("informed guess") to a proposed overapprox of a*

Special extrapolation (widen) operator ∇

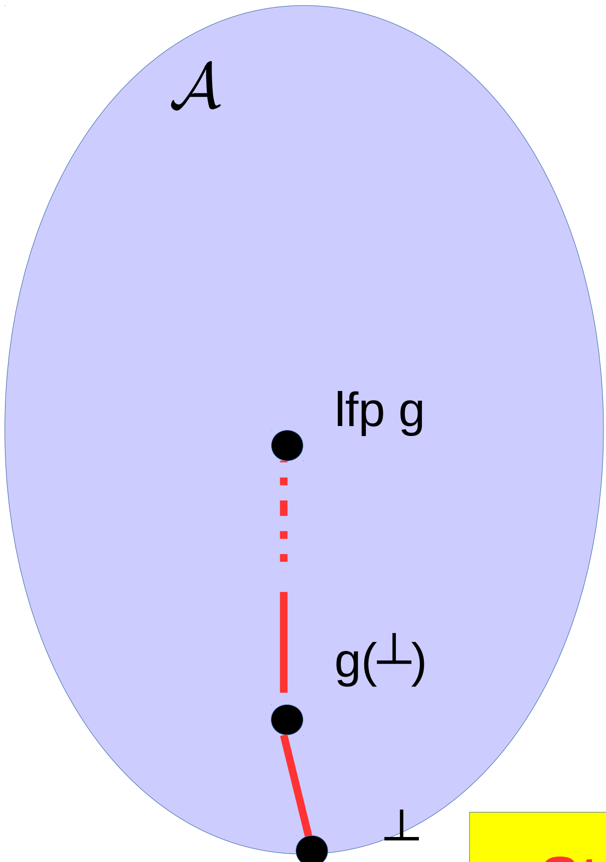
Abstract Loop Invariant: Widen Operator

$\nabla: A \times A \rightarrow A$



Abstract Loop Invariant: Widen Operator

$\nabla: A \times A \rightarrow A$



Required properties of ∇

For every a1, a2 in A
$a1 \nabla a2 \sqsupseteq a1$ and $a1 \nabla a2 \sqsupseteq a2$

For every $a0 \sqsubseteq a1 \sqsubseteq a2 \sqsubseteq \ldots$, the sequence

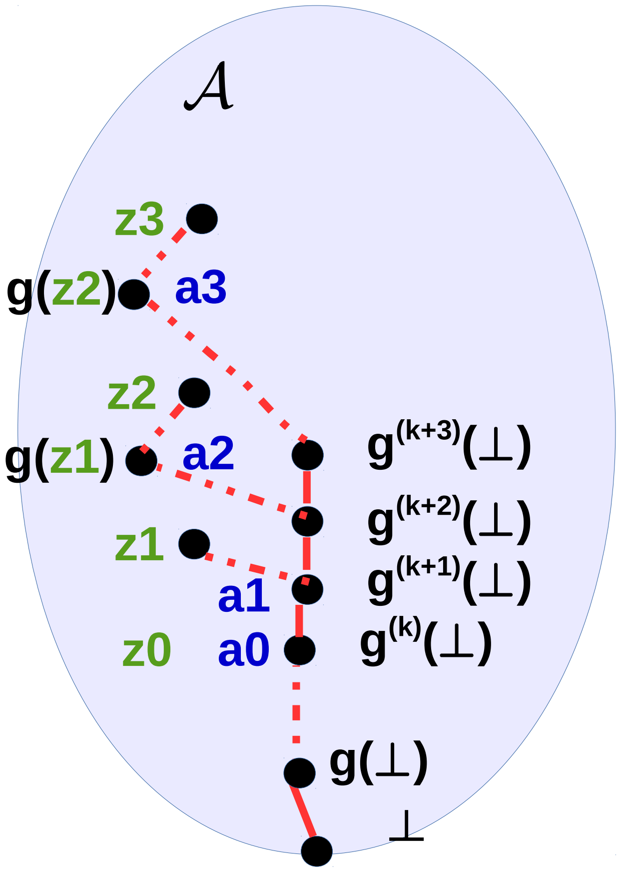
$z0 = a0$
$z1 = z0 \nabla a1$
$z2 = z1 \nabla a2$
....
$z_{i+1} = z_i \nabla a_{i+1}$

stabilizes, i.e.
There exists an $i >= 0$ s.t. $z_i = z_{i+1} = z_{i+2} = \ldots$

Stabilized value $z^* \sqsupseteq$ limit of a0, a1, a2, ….

Abstract Loop Invariant: Widen Operator

∇: A x A → A



Compute $g(\perp)$, $g^2(\perp)$, … $g^{(k)}(\perp)$ for parameter $k > 0$

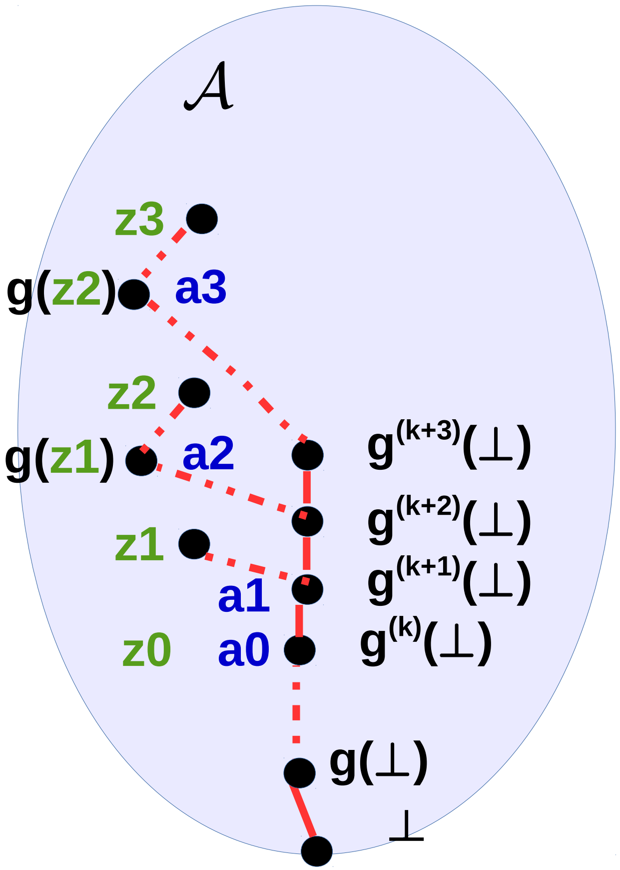
Define $a0 = g^{(k)}(\perp)$	$z0 = a0$
$a1 = g(z0)$	$z1 = z0 \nabla a1$
$a2 = g(z1)$	$z2 = z1 \nabla a2$
……..	…….
$a_i = g(z_{i-1})$	$z_i = z_{i-1} \nabla a_i$

Fact : $g^{(k+j)}(\perp) \sqsubseteq a_j \sqsubseteq a_{j+1}$ forall $j >= 0$

Recall g: A → A is monotone

Abstract Loop Invariant: Widen Operator

∇: A x A → A



Compute $g(\bot)$, $g^2(\bot)$, … $g^{(k)}(\bot)$ for parameter $k > 0$

Define $a0 = g^{(k)}(\bot)$	$z0 = a0$
$a1 = g(z0)$	$z1 = z0 \ \nabla \ a1$
$a2 = g(z1)$	$z2 = z1 \ \nabla \ a2$
……..	…….
$a_i = g(z_{i-1})$	$z_i = z_{i-1} \ \nabla \ a_i$

Fact : $g^{(k+j)}(\bot) \sqsubseteq a_j \sqsubseteq a_{j+1}$ forall $j >= 0$

If $z_i = z_{i+1}$, then

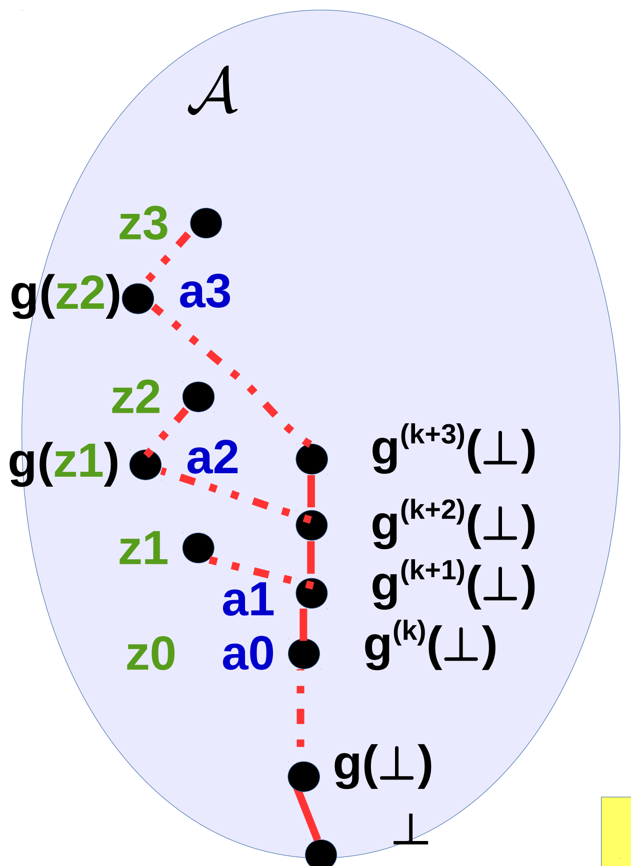
$a_{j+1} = a_{i+1}$ for all $j >= i$

$z_j = z_i$ for all $j >= I$

Can detect when sequence stabilizes

Abstract Loop Invariant: Widen Operator

∇: A x A $\rightarrow$ A



Compute $g(\perp)$, $g^2(\perp)$, … $g^{(k)}(\perp)$ for parameter $k > 0$

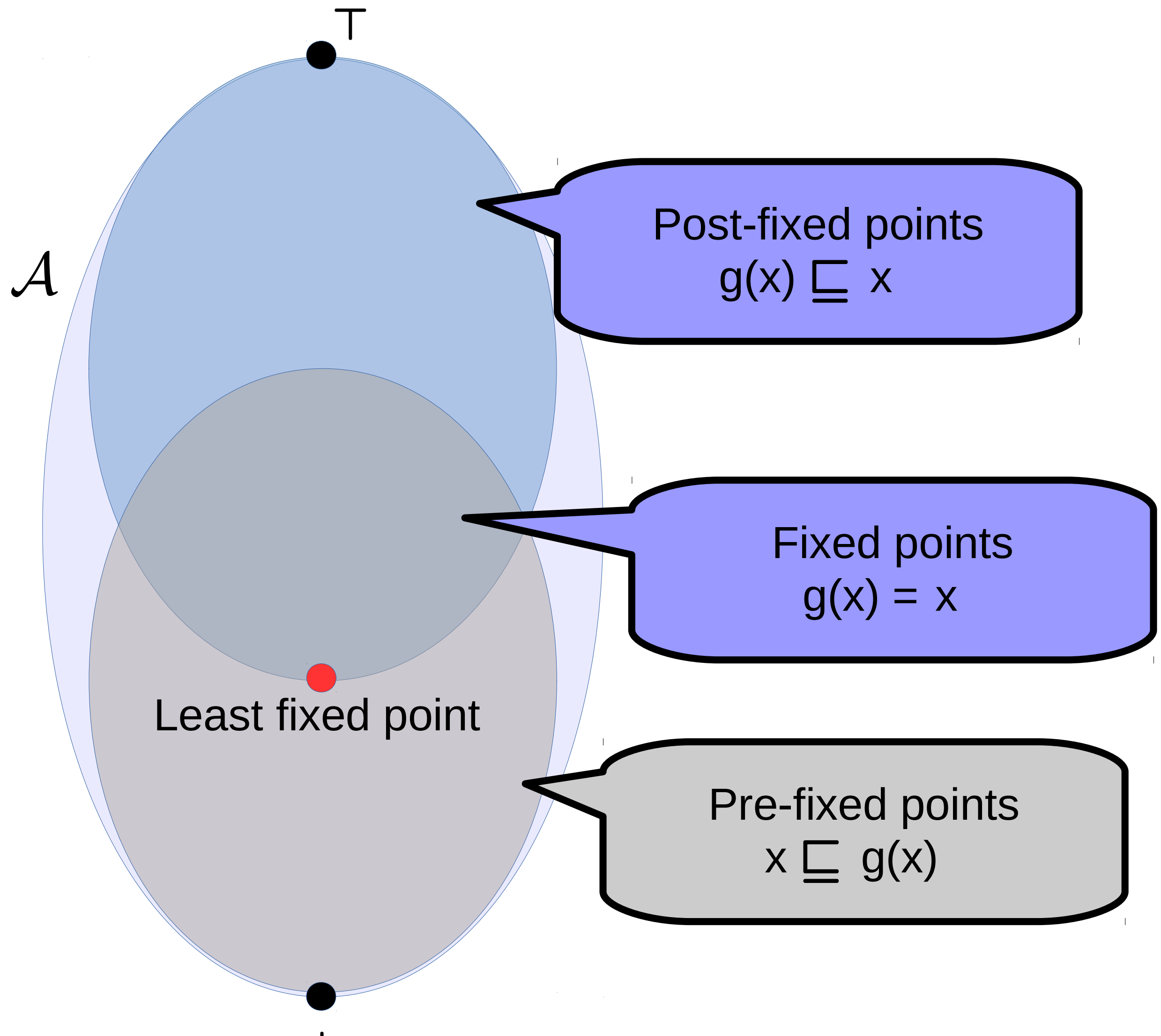
Define $a0 = g^{(k)}(\perp)$	$z0 = a0$
$a1 = g(z0)$	$z1 = z0 \; \nabla \; a1$
$a2 = g(z1)$	$z2 = z1 \; \nabla \; a2$
……..	…….
$a_i = g(z_{i-1})$	$z_i = z_{i-1} \; \nabla \; a_i$

Stabilized value z* overapproximates $g^{(i)}(\perp)$ for all i >= 0

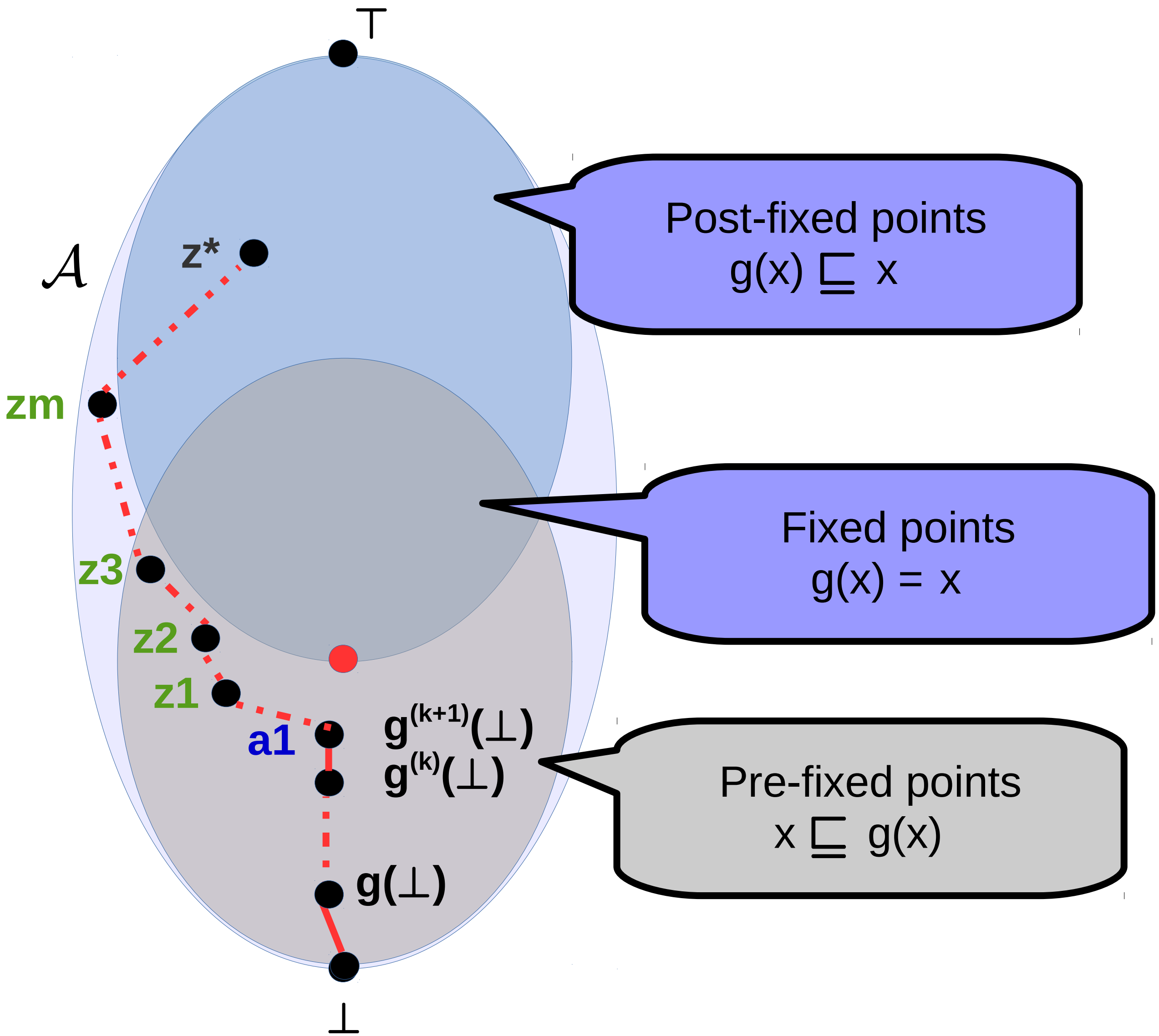
Abstract loop invariant

In fact, $g^{(r)}(z^*)$ also overapproximates $g^{(i)}(\perp)$ for all r >= 0

Another View of Widening



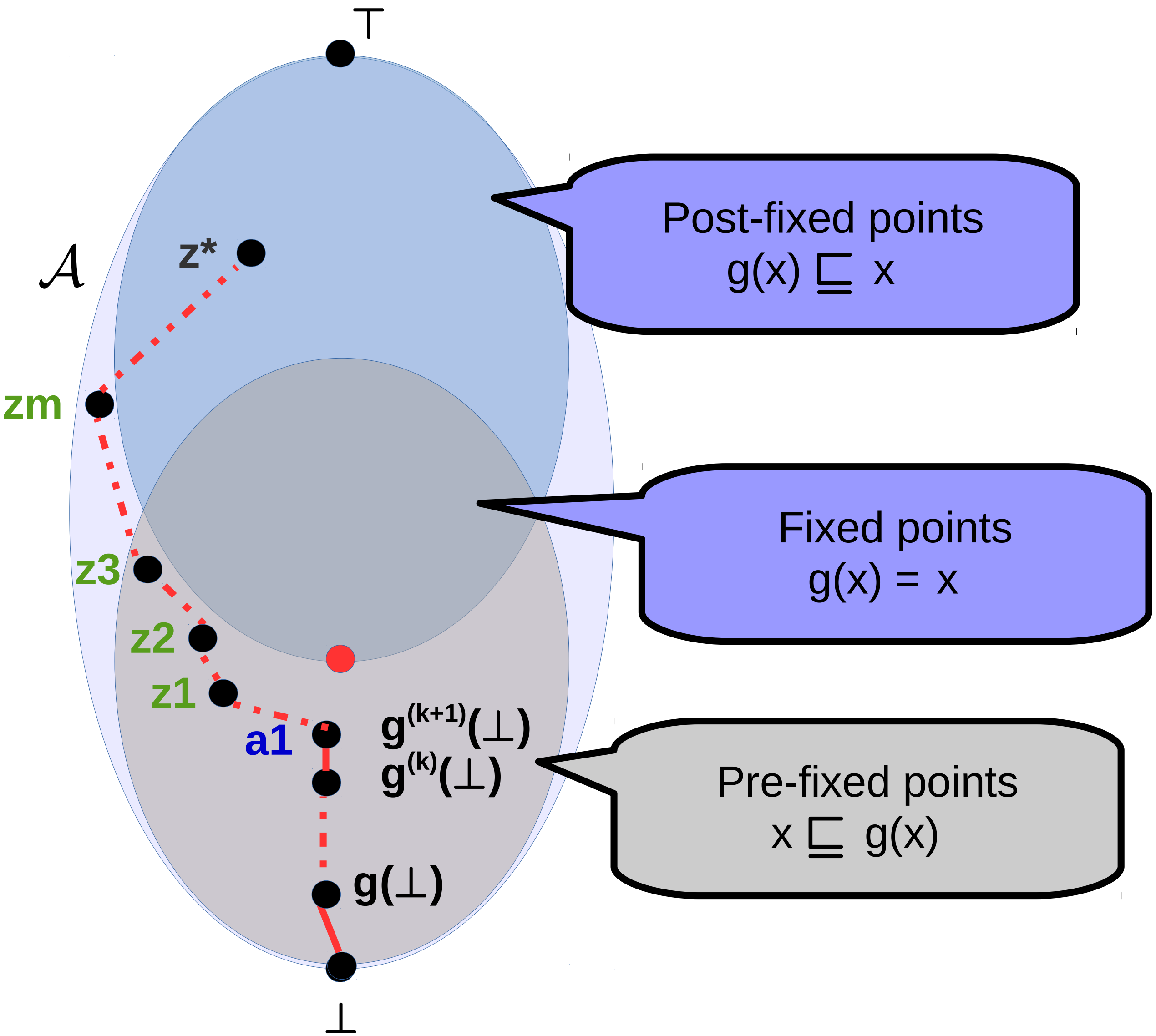
Another View of Widening



Another View of Widening

z* = z* ∇ g(z*)
implies
g(z*) ⊑ z*

z* is a
post-fixed point

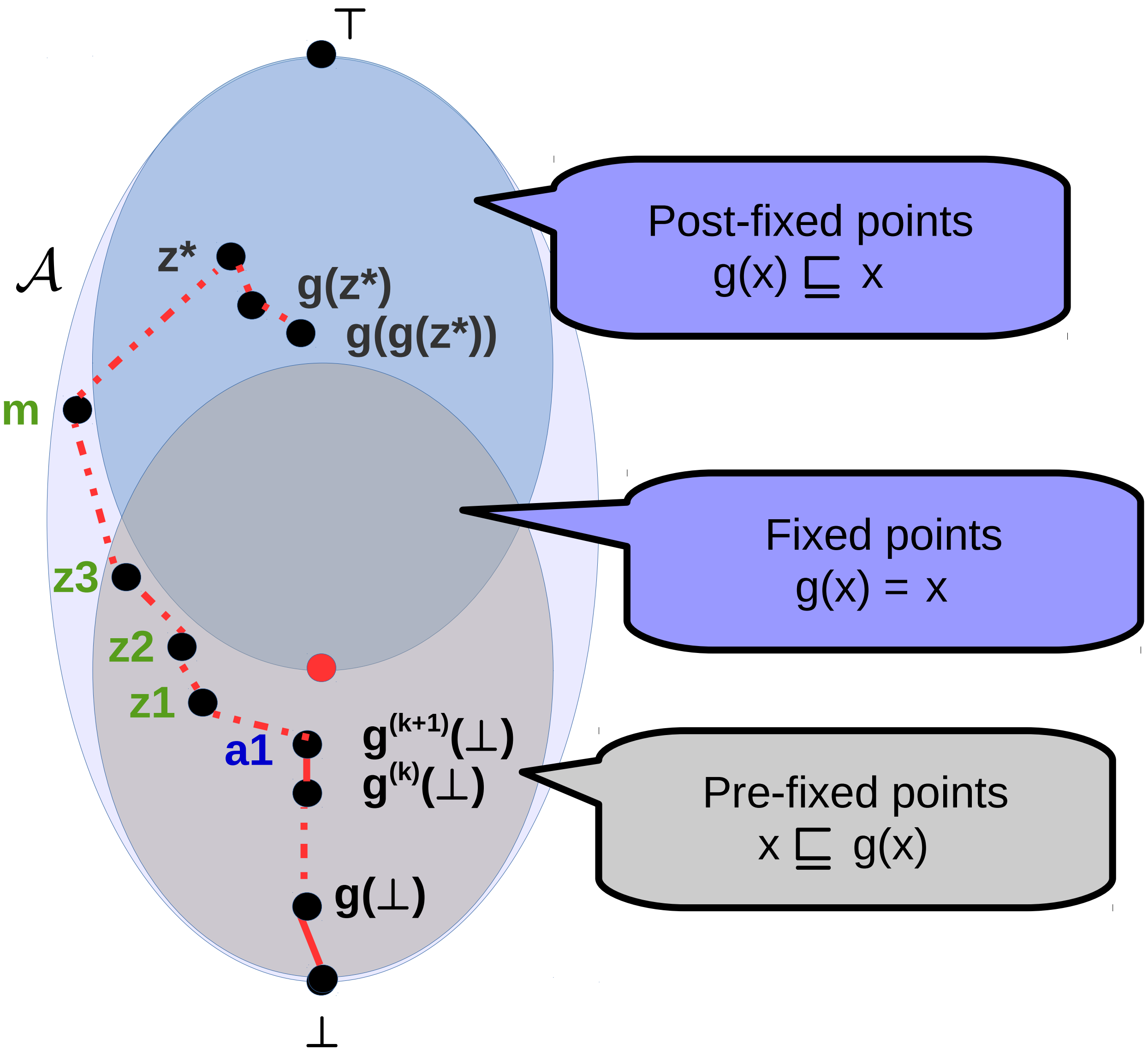


Another View of Widening

$z^* = z^* \nabla\ g(z^*)$
implies
$g(z^*) \sqsubseteq z^*$

z^* is a
post-fixed point

$g^{(r)}(z^*)$ is a
post-fixed point
and
$lfp \sqsubseteq g^{(r)}(z^*)$



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

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] , $(-\infty, 20]$
  - $\sqsubseteq$ relation: Inclusion of intervals
  - [-10, 7] , $(-\infty, 20] \sqsubseteq [-20, 9], (-\infty, +\infty)$
  - $\sqcup$ and $\sqcap$: union and intersection of intervals
  - $[a, b]\ \nabla x\ [c, d] = [e, f]$, where
    - $e = a$ if $c >= a$, and $e = -\infty$ otherwise
    - $f = b$ if $d <= b$, and $f = +\infty$ otherwise
  - ∇y similarly defined, and ∇ is simply $(\nabla x, \nabla y)$
  - $\bot$ 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