Introduction to Static Program Analysis

Mooly Sagiv

Challenges in Proving Correctness

- Specifying what the program is supposed to do
- Writing loop invariants
- Decision procedures for proving implications
  - Deduction

Static Analysis

- Automatically infer sound invariants from the code
- Prove the absence of certain program errors
- Prove user-defined assertions
- Report bugs before the program is executed

Simple Correct C code

```
main() {
    int i = 0,  *p =NULL, a[100];
    for (i=0 ;  i <100, i++) {
      a[i] = i;
      p = malloc(1, sizeof(int));
      *p = i;
      free(p); // not alloc(p)
      p = NULL; // no leak
     }
```

Simple Correct C code

```
main() {
    int i = 0,  *p=NULL, a[100];
    for (i=0 ;  i <100, i++) {
        { 0 <= i < 100}
        a[i] = i;
        { p == NULL:}
        p = malloc(1, sizeof(int));
        { alloc(p) }
        *p = i;
        {alloc(p)}
        free(p);
         {!alloc(p)}
        p = NULL;
         {p==NULL}
       }
```

Simple Incorrect C code

```
main() {
    int i = 0,  *p=NULL, a[100], j;
    for (i=0 ;  i <j , i++) {
      { 0 <= i < j}
      a[i] = i;
      p = malloc(1, sizeof(int));
      { alloc(p) }
     p = malloc(1, sizeof(int));
      { alloc(p) }
      free(p);
      free(p);
     }
```

Sound (Incomplete) Static Analysis

- It is undecidable to prove interesting program properties
- Focus on sound program analysis
  - When the compiler reports that the program is correct it is indeed correct for every run
  - The compiler may report spurious (false alarms)

A Simple False Alarm

```
int i, *p=NULL;
…
if (i >=5) {
    p = malloc(1, sizeof(int));
}
…
if (i >=5) {
    *p = 8;
}
…
if (i >=5) {
    free(p);
}
```

A Complicated False Alarm

```
int i, *p=NULL;
…
if (foo(i)) {
    p = malloc(1, sizeof(int));
}
…
if (bar(i )) {
   *p = 8;
}
…
if (zoo(i)) {
   free(p);
}
```

Foundation of Static Analysis

- Static analysis can be viewed as interpreting the program over an “abstract domain”
- Execute the program over larger set of execution paths
- Guarantee sound results
  - Whenever the analysis reports that an invariant holds it indeed hold

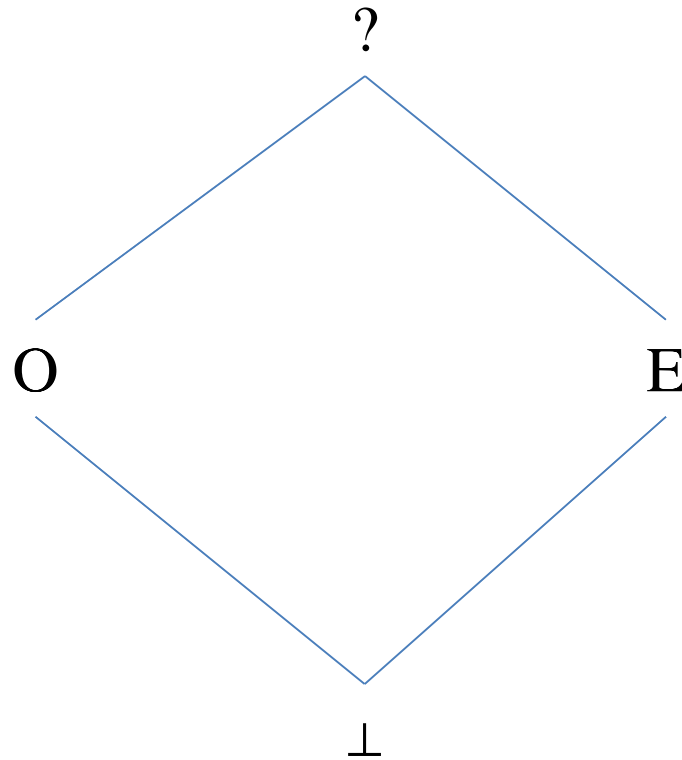
Even/Odd Abstract Interpretation

- Determine if an integer variable is even or odd at a given program point

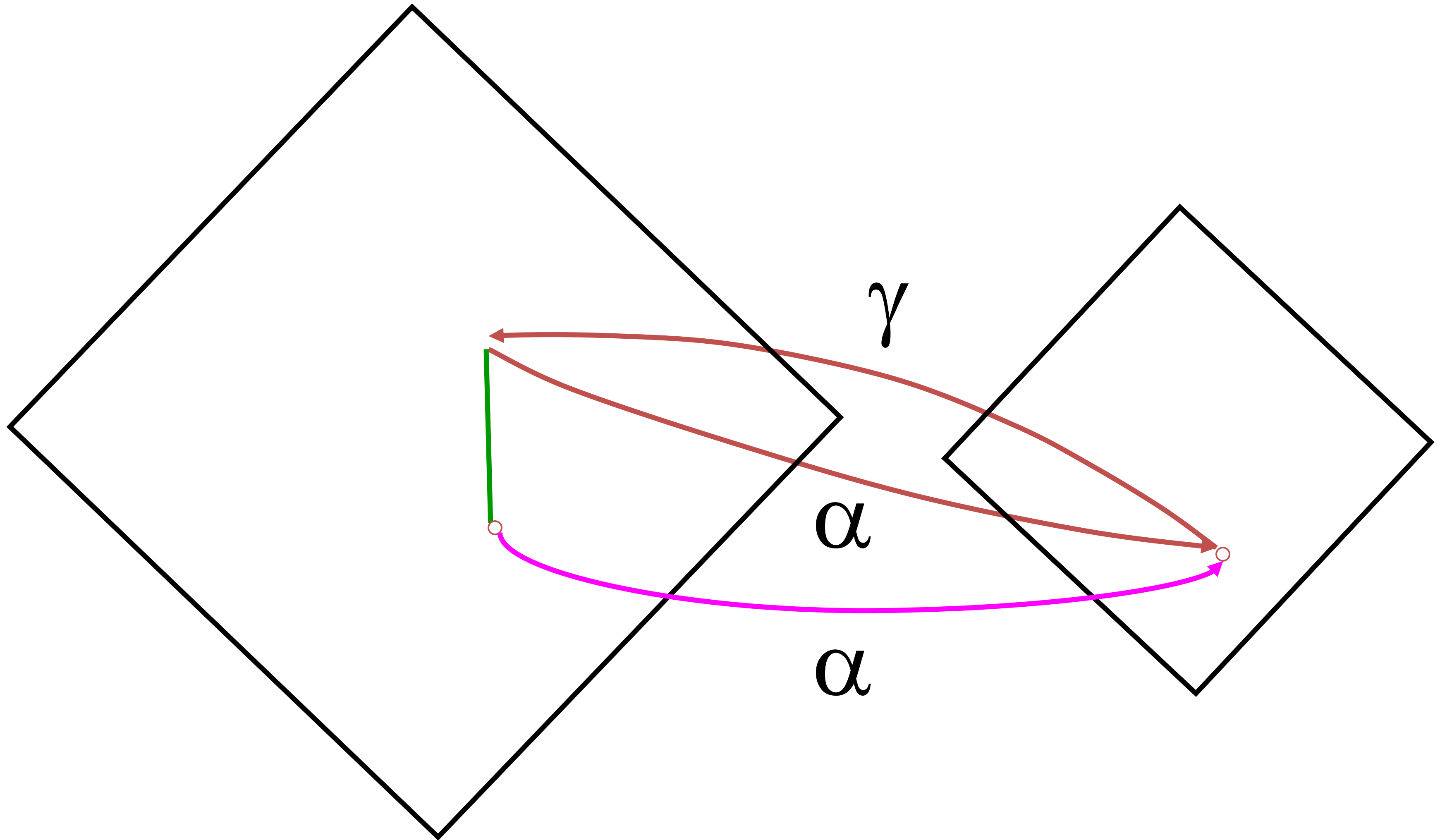
Example Program

```
/* x=? */

while  (x !=1)   do {  /* x=? */
        if   (x %2) == 0
/* x=E */            { x := x / 2; }        /* x=? */
              else
/* x=O */            { x := x * 3 + 1;
                       assert (x %2 ==0); }  /* x=E */
}

/* x=O*/
```

Lattice of Values

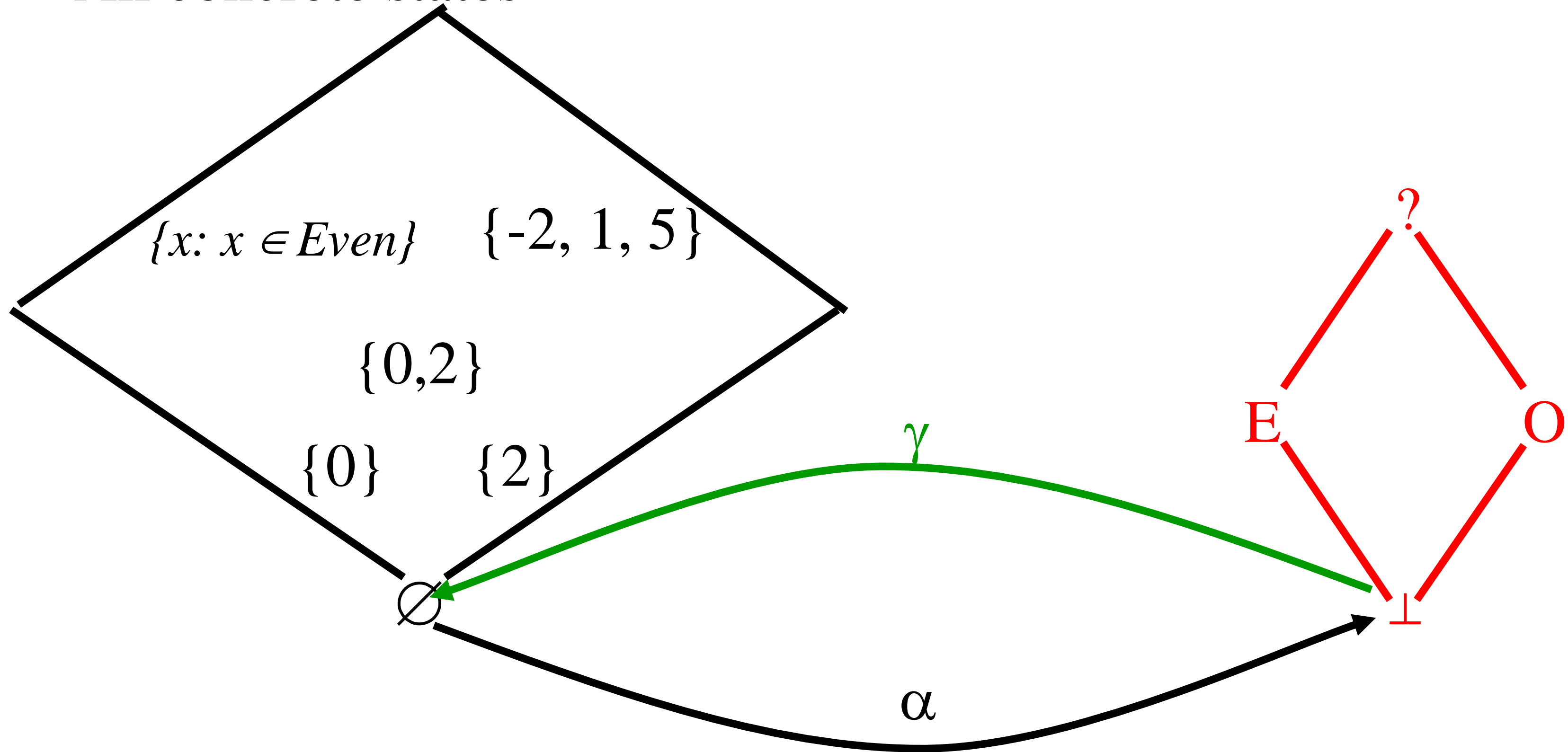


Abstract Interpretation

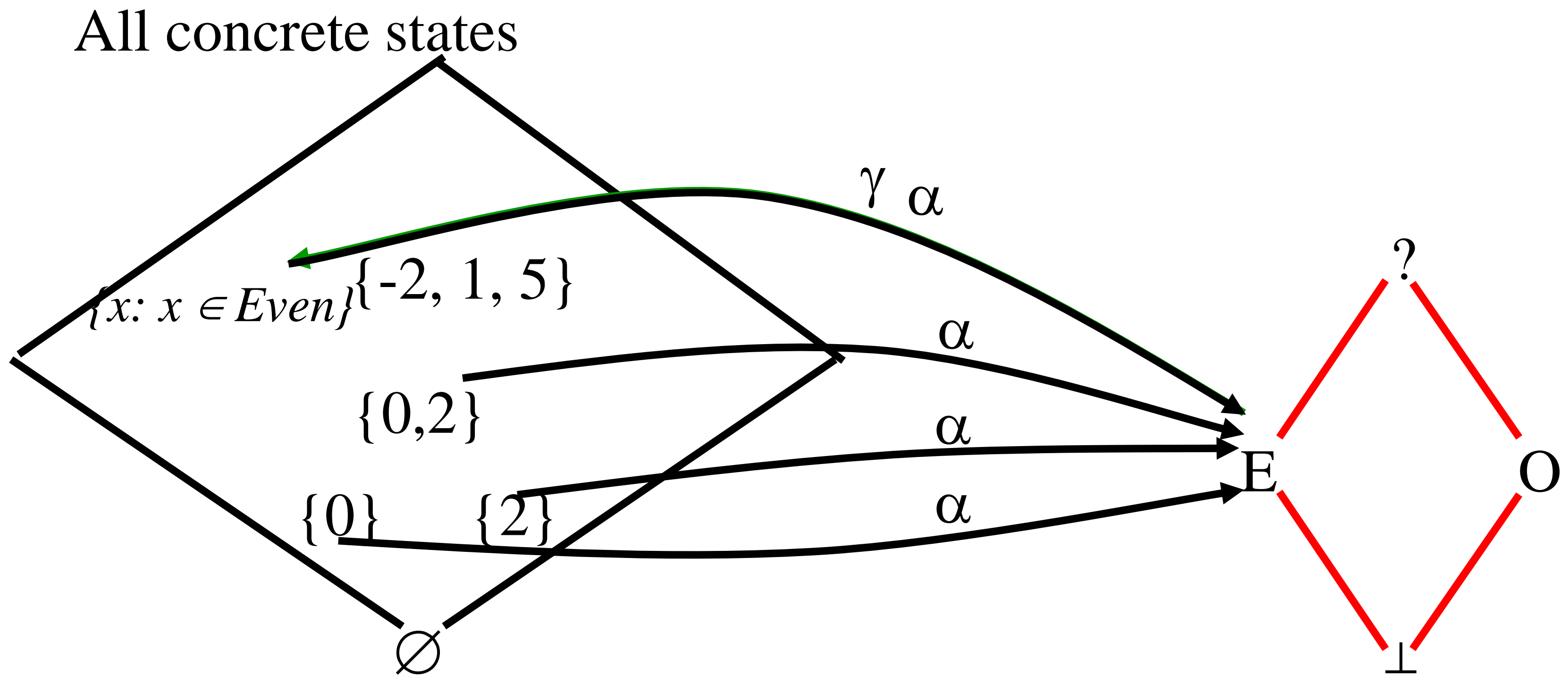


Sets of stores $\xrightarrow{\alpha}$ *Descriptors of sets of stores*

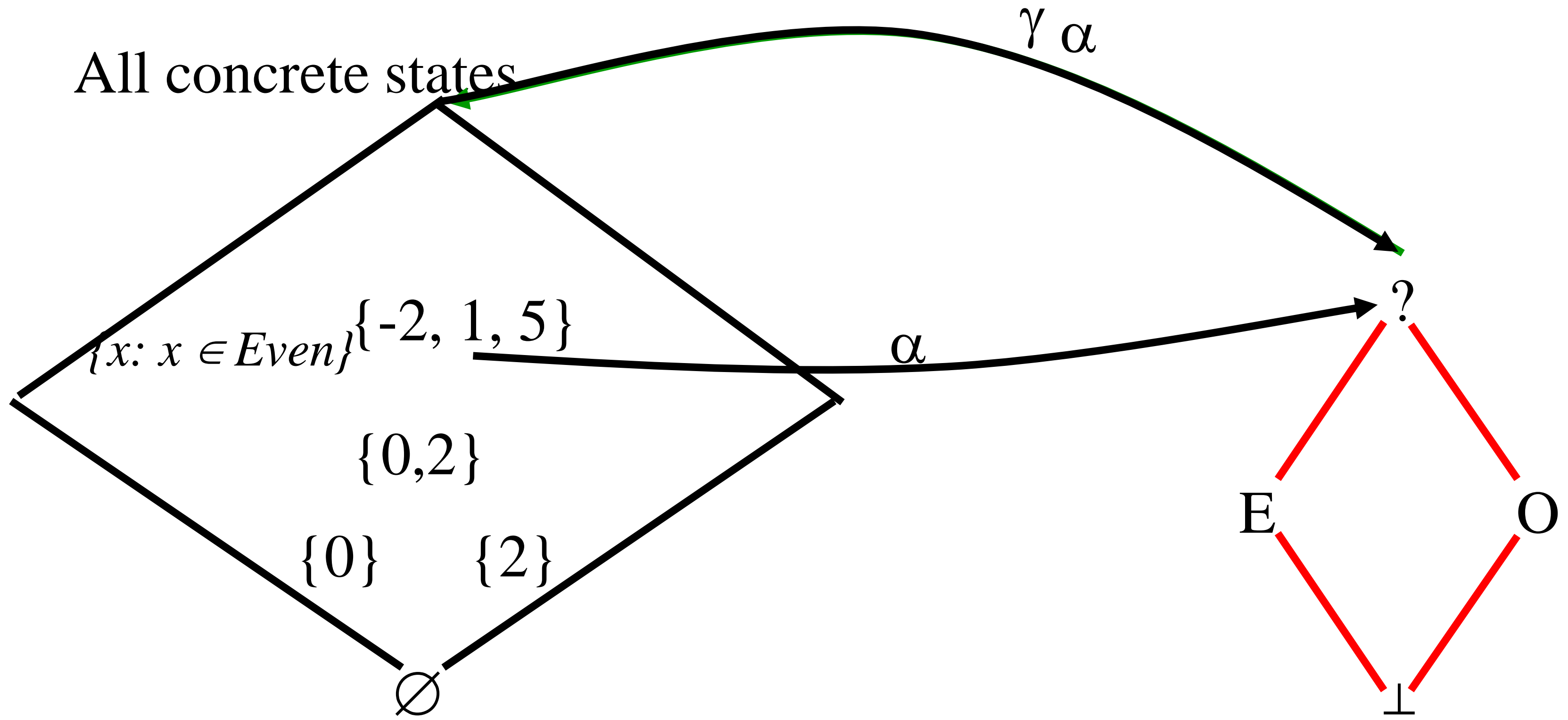
Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation



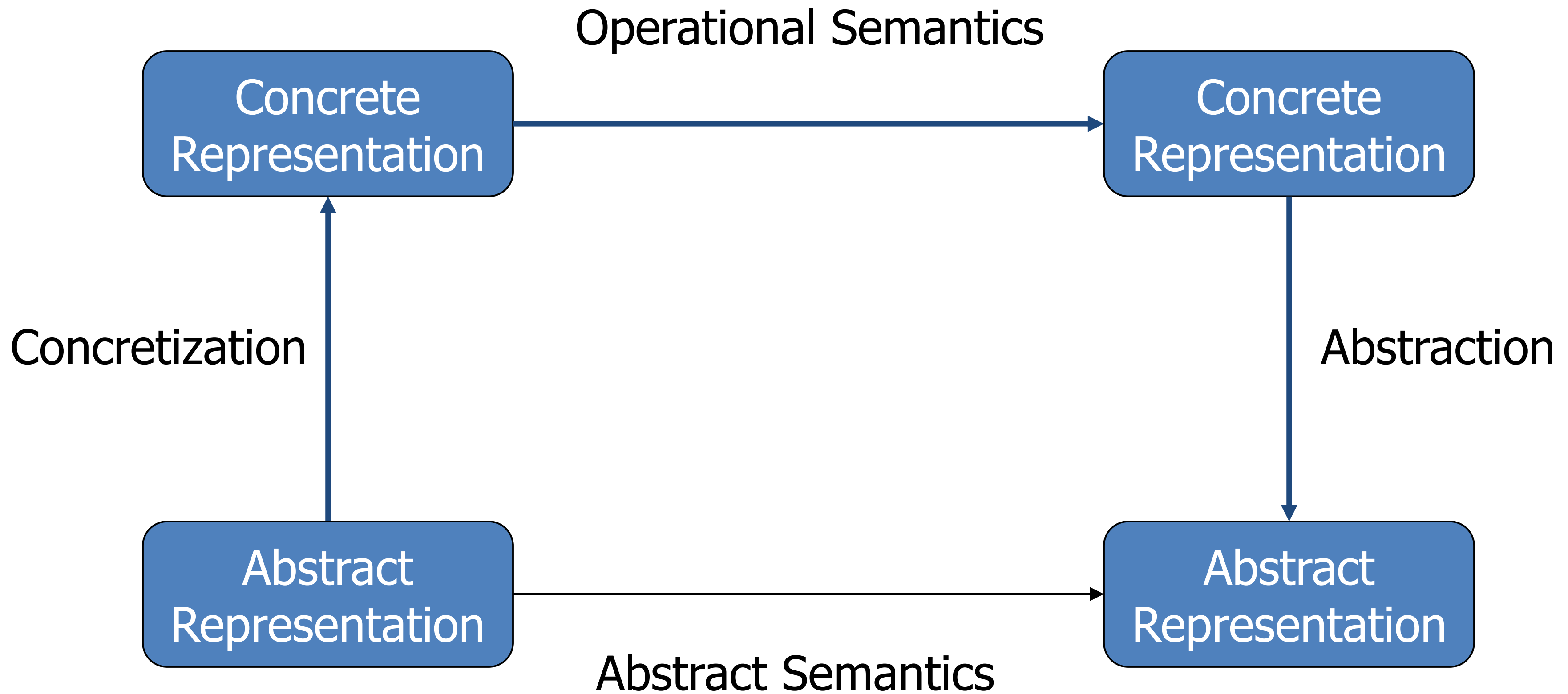
Odd/Even Abstract Interpretation



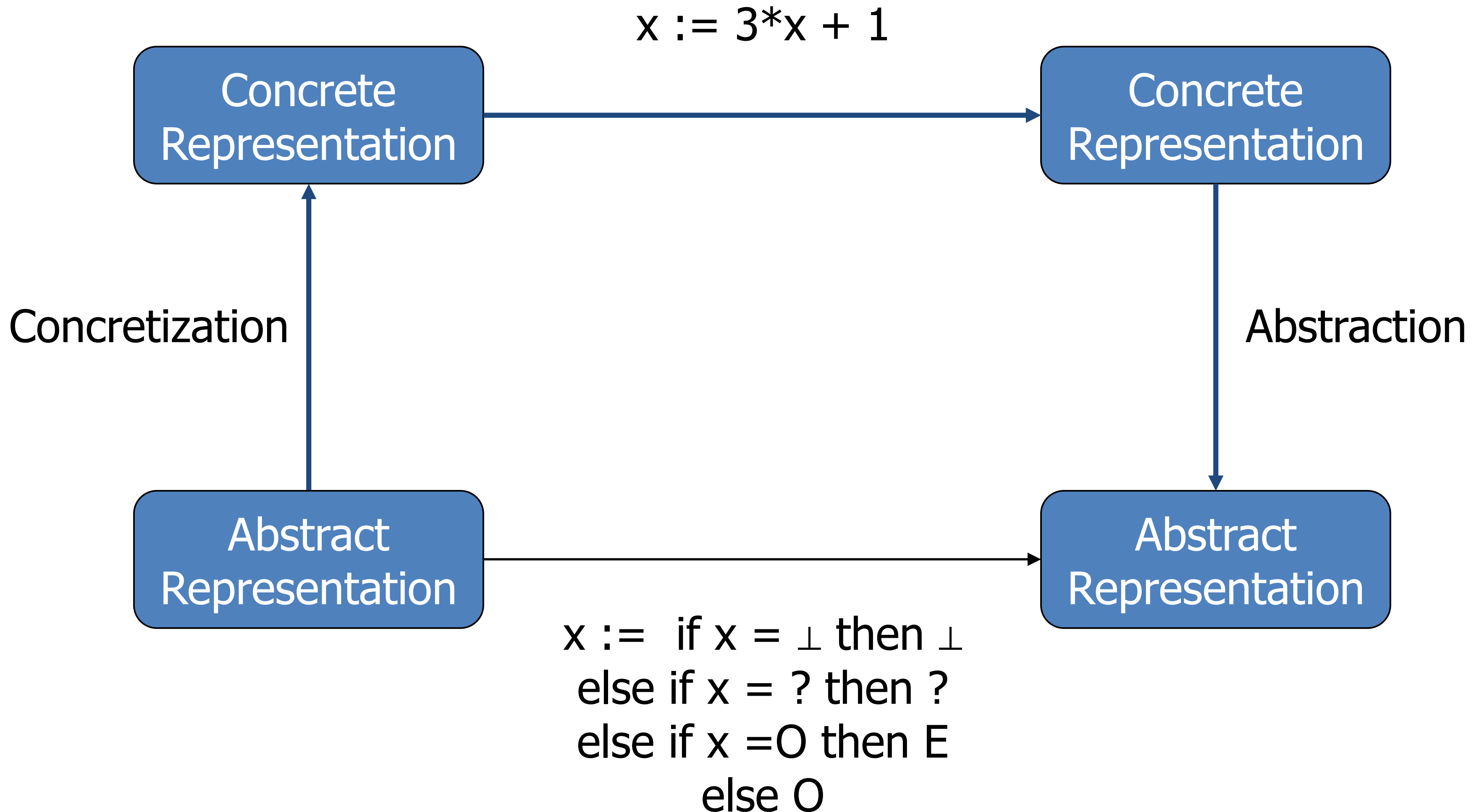
Example Program

```
while  (x !=1)   do {
          if   (x %2) == 0
                          { x := x / 2; }
                   else
   /* x=O */   { x := x * 3 + 1;        /* x=E */
                          assert (x %2 ==0); }
}
```

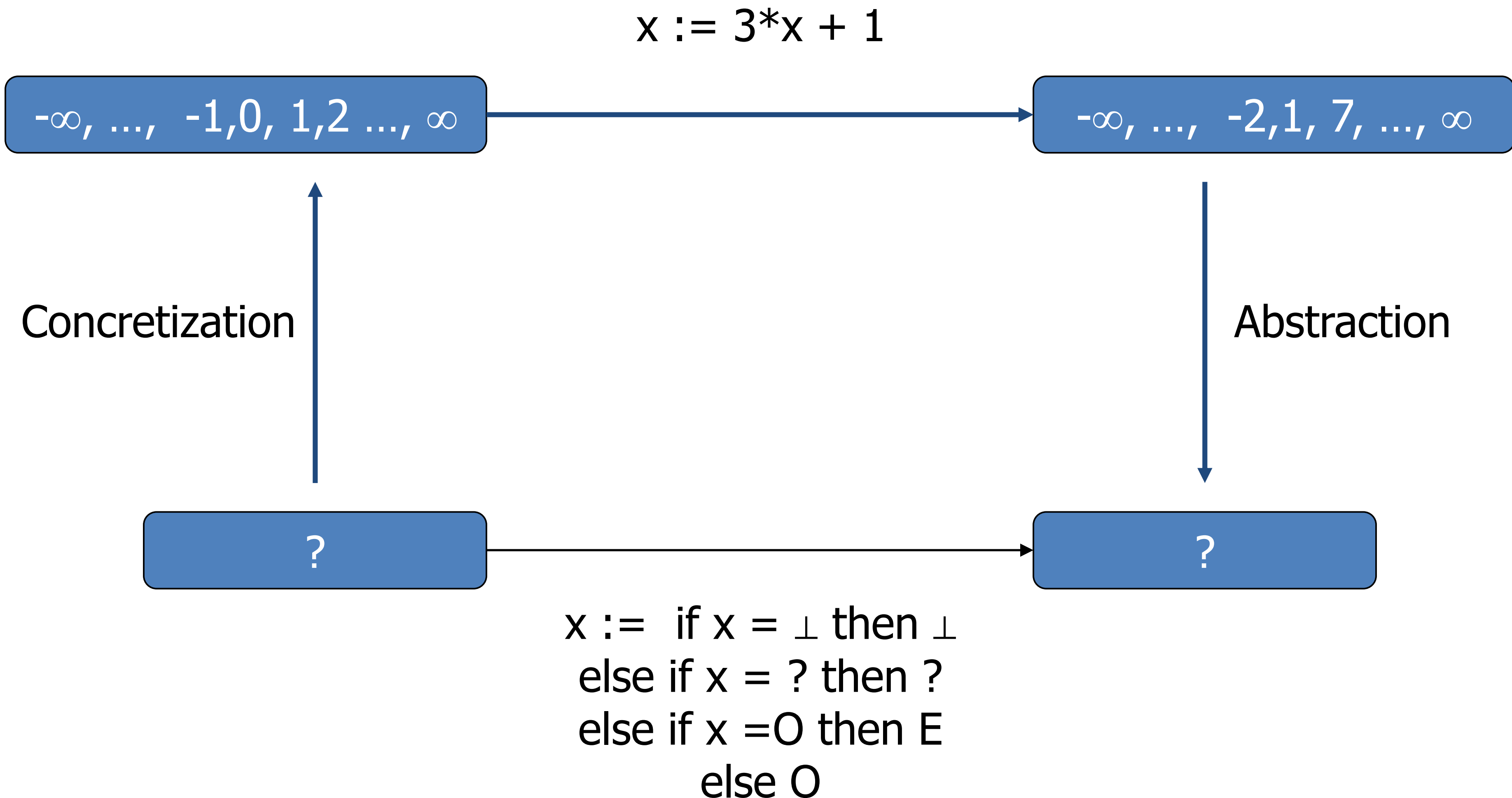
(Best) Abstract Transformer



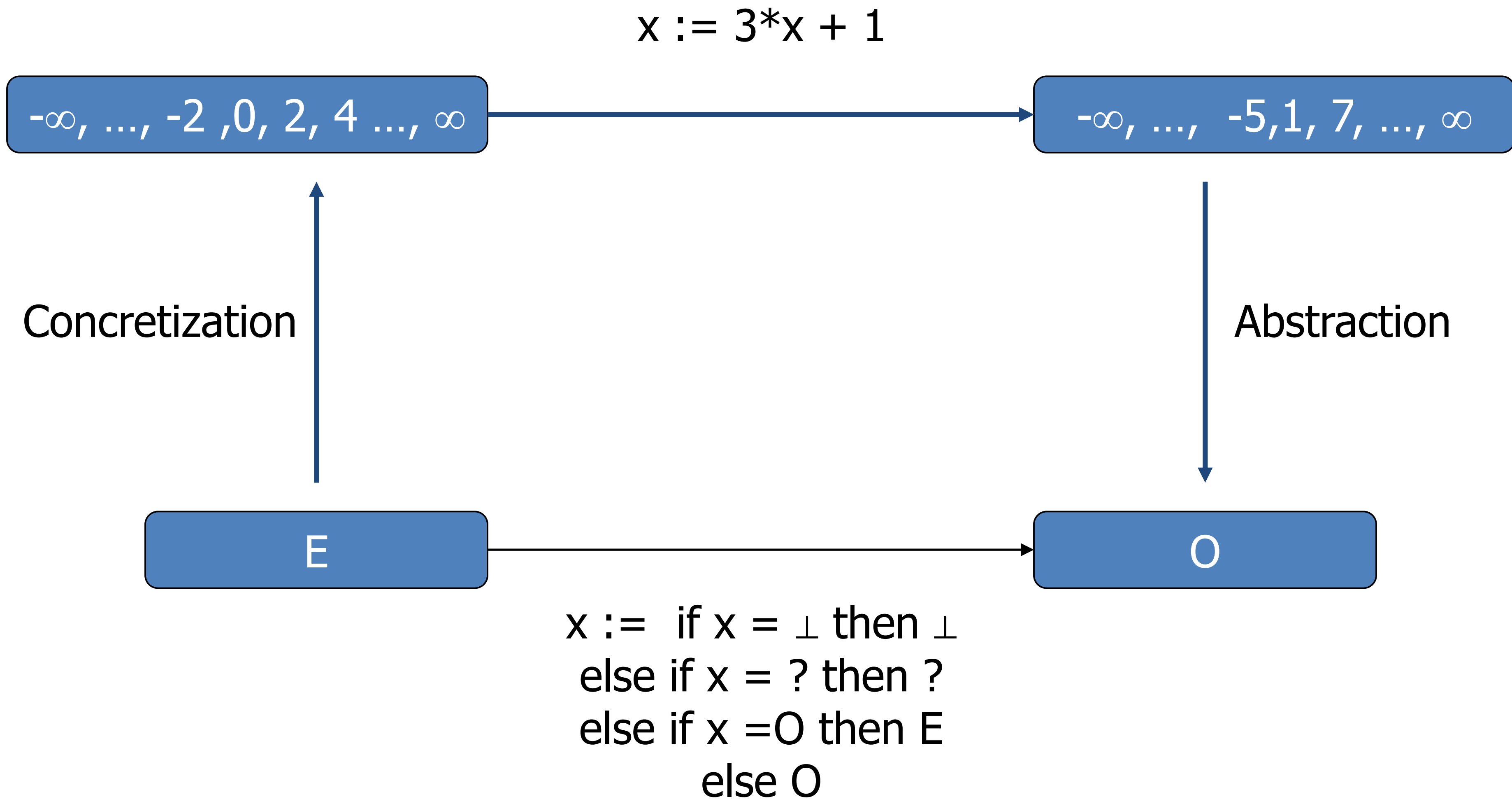
(Best) Abstract Transformer



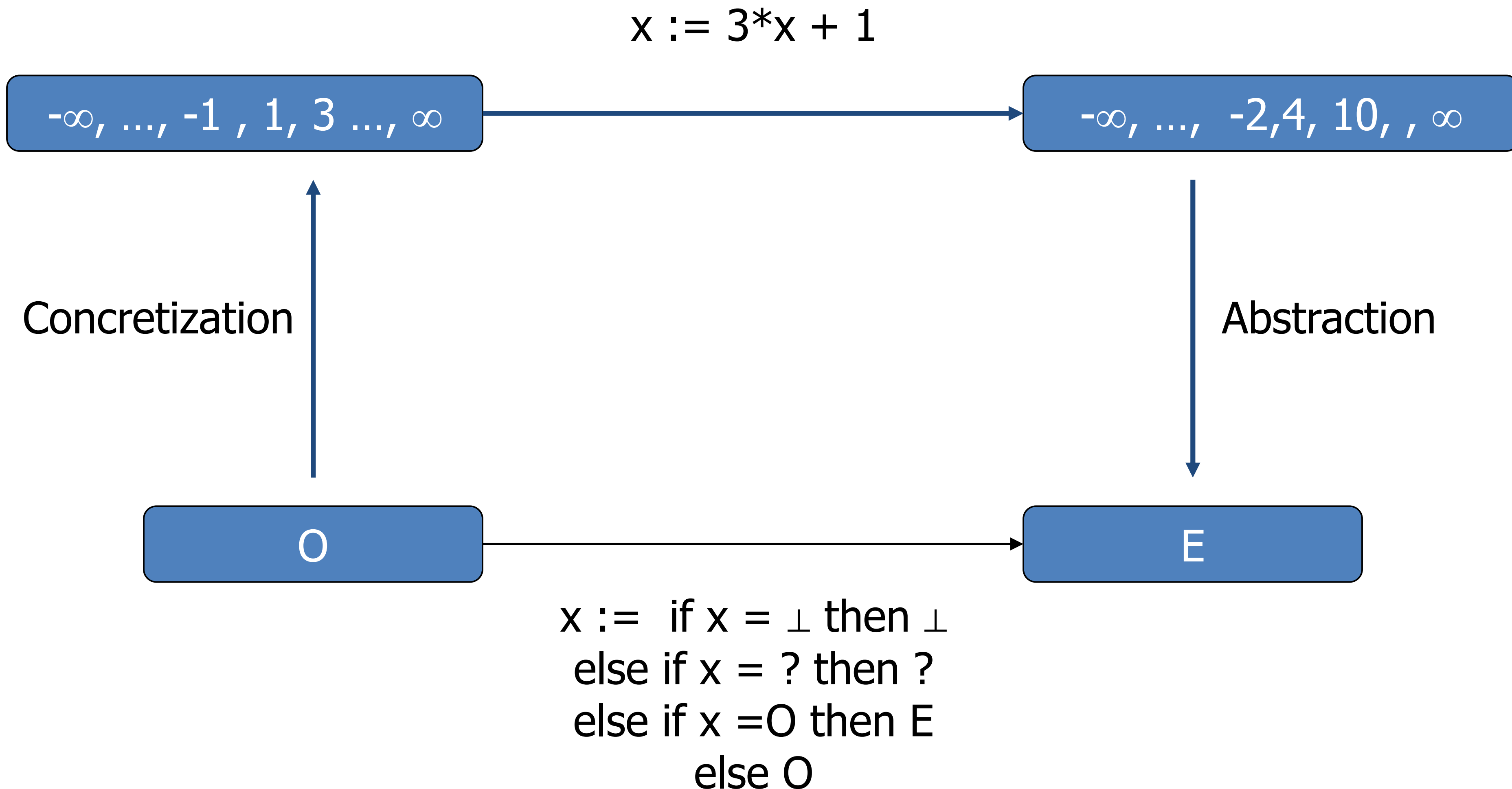
(Best) Abstract Transformer



(Best) Abstract Transformer



(Best) Abstract Transformer



Runtime vs. Static Testing

	Runtime	Static Analysis
Effectiveness	Missed Errors	False alarms
		Locate rare errors
Cost	Proportional to program's execution	Proportional to program's size
	No need to efficiently handle rare cases	Can handle limited classes of programs and still be useful

Static Analysis Algorithms

- Generate a control flow graph
- Collecting semantics define the reachable states
- Generate a system of equations over the abstract values at every node
- Iteratively compute the simultaneous least solution at every node
- The solution is guaranteed to be sound
  - Abstracts the set of reachable states
  - Computes an inductive invariant
    - May not be strong enough
- The correctness of the safety properties can be conservatively checked

Example Interval Analysis

- Find a lower and an upper bound of the value of a single variable
- Can be generalized to multiple variables

Simple Correct C code

```
main() {
    int i = 0, a[100];
    { [-minint, maxint] }
    for (i=0 ;  i <100, i++) {
        {[0, 99]}
        a[i] = i;
         {[0, 99]}
              }
{[100, 100]}
```

The Power of Interval Analysis

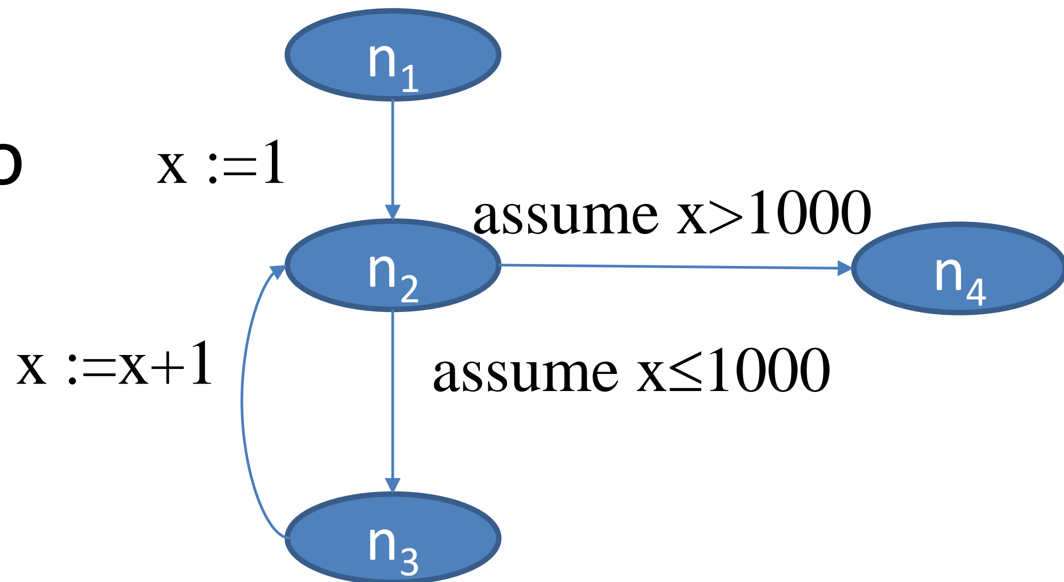
```
int f(x) {
 {[minint , maxint]}
  if (x > 100) {
      {[101, maxint]}
      return x -10 ;
      {[91, maxint-10];}
    }
    else {
    {[minint, 100] }
    return f(f(x+11))
    { [91, 91]}
    }
```

Example Program Interval Analysis

x := 1 ;

while x $\leq$ 1000 do

x := x + 1;

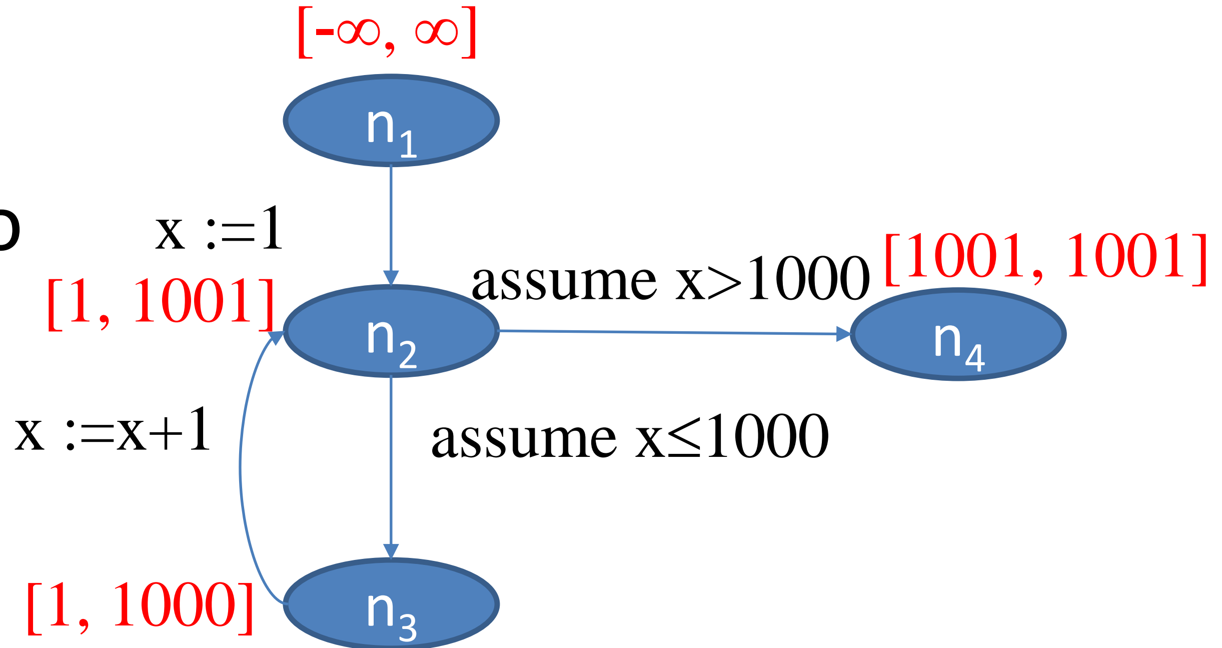


Example Program Interval Analysis

x := 1 ;

while x $\leq$ 1000 do

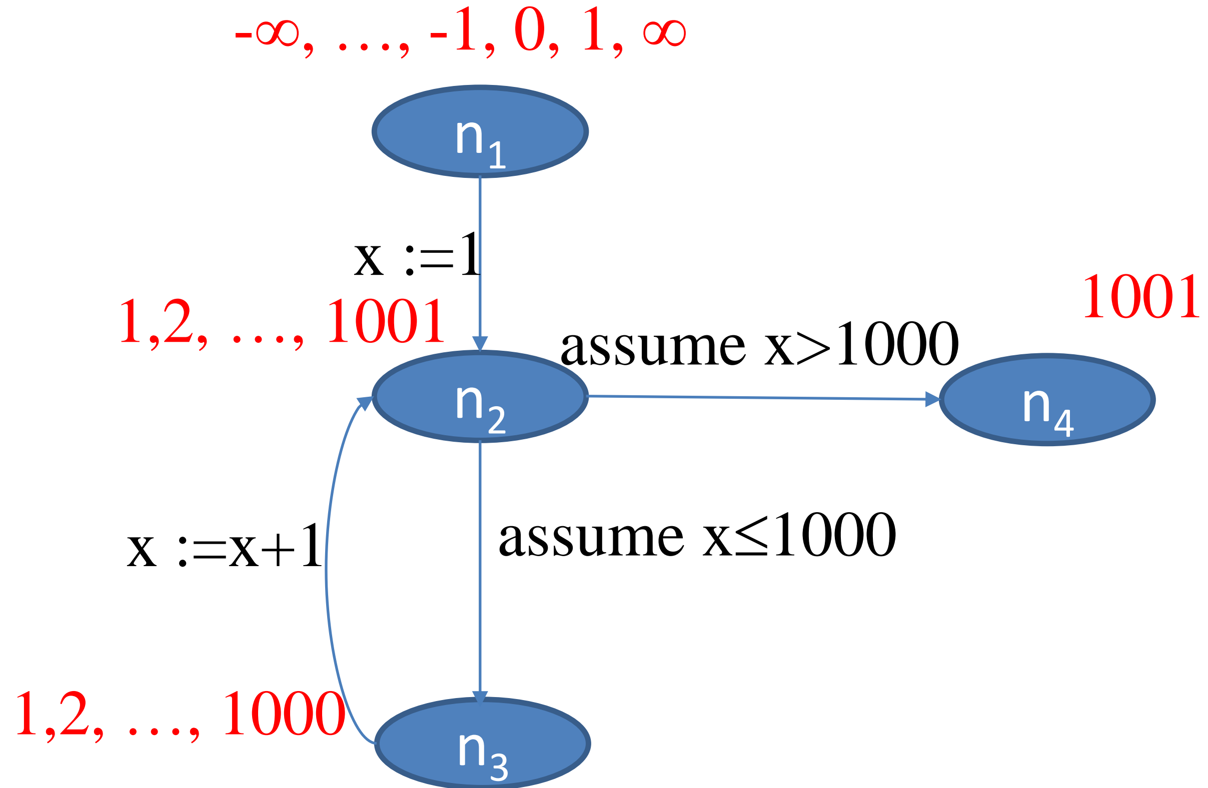
x := x + 1;



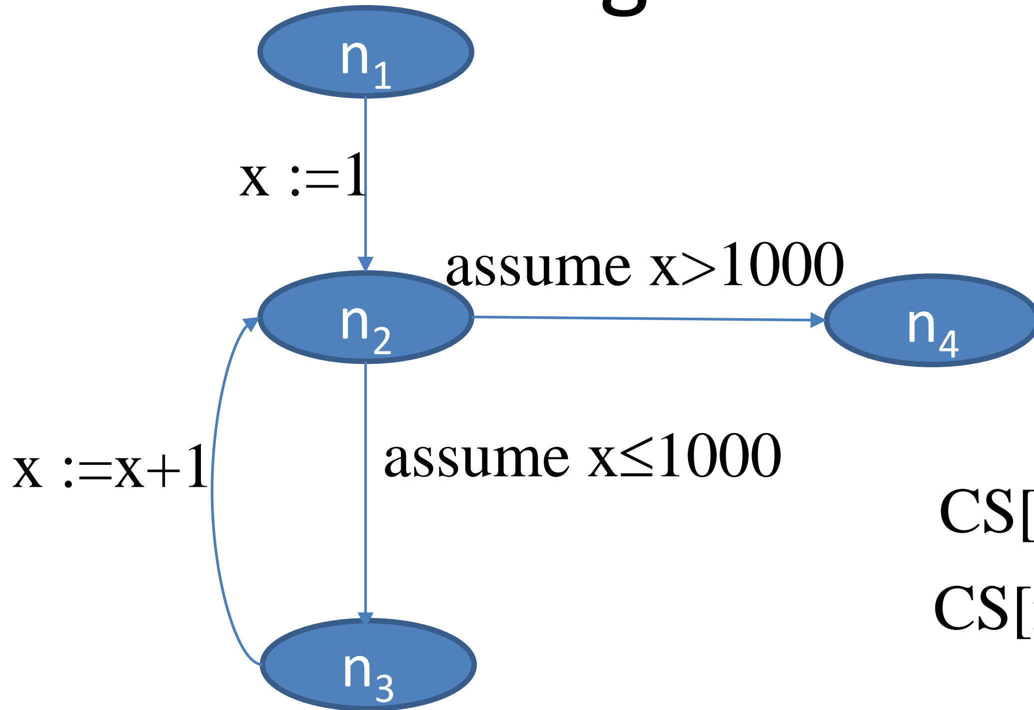
Collecting Interpretation

- Defines the set of reachable states as the least solution to a systems of equations
- Uniquely defined
- But not necessarily computable

Collecting Semantics (Example)



Collecting Semantics (Example)



$$CS[n_1] = Z$$

$$CS[n_2] = [\![x := 1]\!] CS[n_1] \cup [\![x := x+1]\!] CS[n_3]$$

$$CS[n3] = [\![\text{ assume } x \le 100]\!] CS[n_2]$$

$$CS[n4] = [\![\text{ assume } x > 100]\!] CS[n_2]$$

$$[\![]\!] : 2^Z \rightarrow 2^Z$$

$$[\![x := 1]\!] = \lambda x.\{1\}$$

$$[\![x := x+1]\!] = \lambda x.\{z+1 \mid z \in x\}$$

$$[\![\text{assume } x > 100]\!] = \lambda x.$$

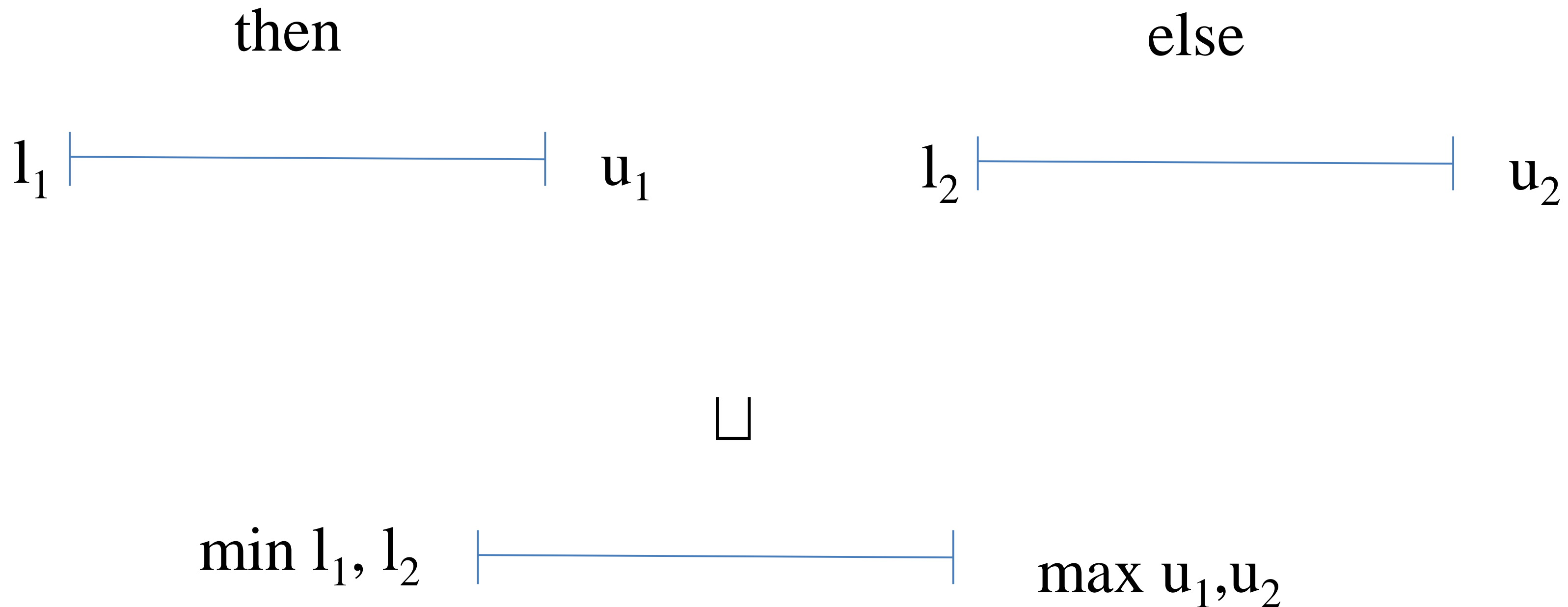
$$[\![\text{assume } x \le 100]\!] = \lambda x.$$

$CS[n_1]$	$CS[n_2]$	$CS[n_3]$	$CS[n_4]$
Z	{x |1 ≤x≤1001}	{x |1 ≤x≤1000}	{1001}
Z	{x |1 ≤x≤1000}	{x |1 ≤x≤1001}	{1002}
Z	{x |x≤1000}	{x |x≤1001}	{1001]

The Lattice of Intervals

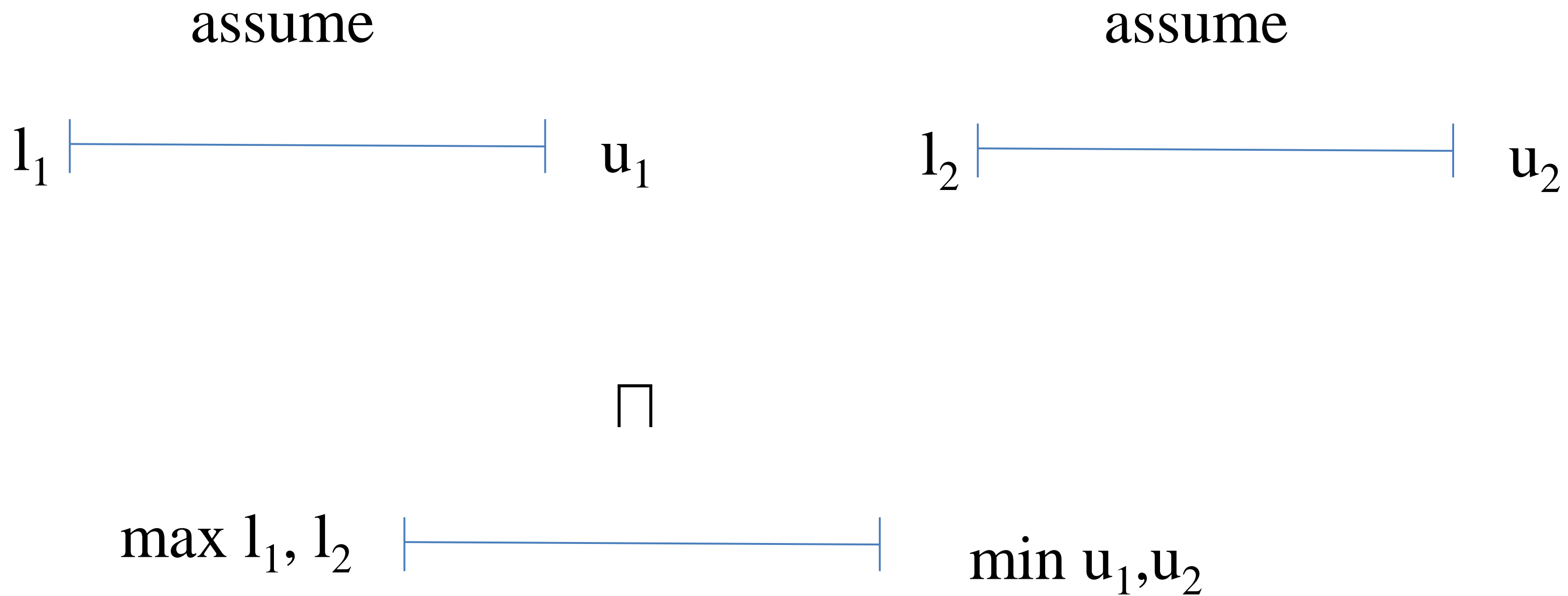
Galois Connection

Abstract Interpretation of Joins



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

Abstract Interpretation of Meets



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

Abstract Interpretation of Atomic Statements

$$[\![\text{ skip}]\!]^{\#}[l, u] = [l, u]$$

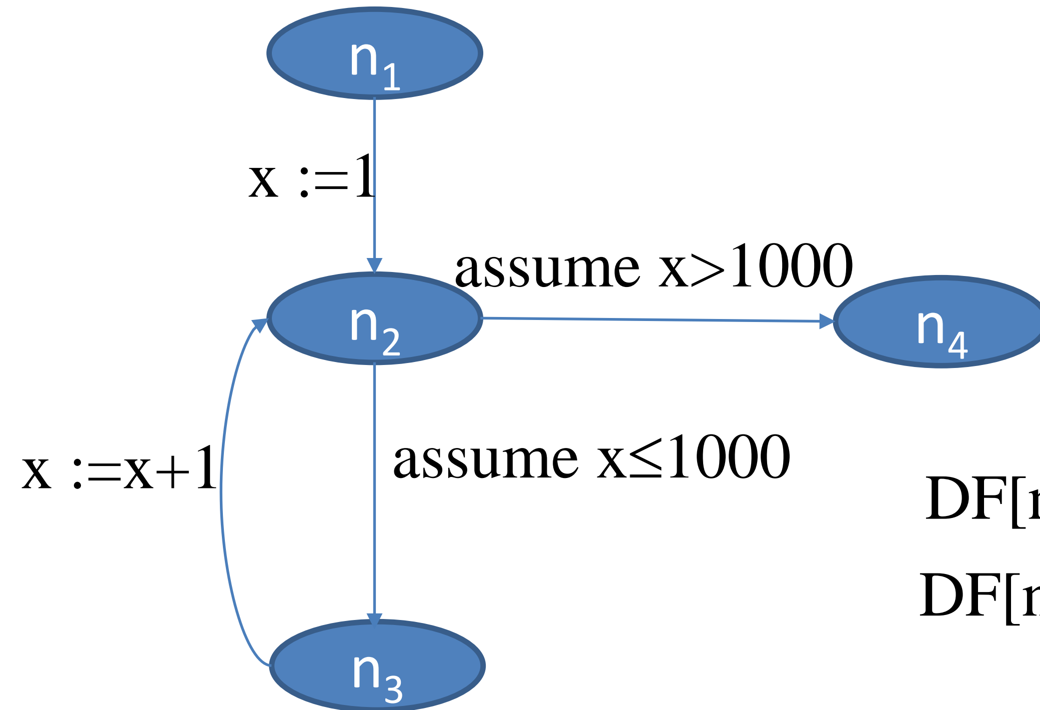
$$[\![x := 1]\!]^{\#}[l, u] = [1, 1]$$

$$[\![\ x := x + 1]\!]^{\#}[l, u] = [l, u] + [1, 1] = [l + 1, u + 1]$$

$$[\![\ \text{assume } x \leq k]\!]^{\#}[l, u] =$$

$$[\![\ \text{assume } x \geq k]\!]^{\#}[l, u] =$$

Interval Analysis



$DF[n_1] = [-\infty, \infty]$

$DF[n_2] = [\![x := 1]\!]^{\#} DF[n_1]$

$\cup$

$[\![x := x+1]\!]^{\#} DF[n_3]$

$DF[n3] = [\![\text{assume } x \leq 100]\!]^{\#} DF[n_2]$

$DF[n4] = [\![\text{assume } x > 100]\!]^{\#} DF[n_2]$

$[\![\,]\!]^{\#}: Z \times Z \rightarrow Z \times Z$

$[\![x := 1]\!]^{\#} = \lambda x.[1, 1]$

$[\![x := x+1]\!]^{\#} = \lambda[l, u]. [l+1, u+1]$

$[\![\text{assume } x > 100]\!]^{\#} = \lambda x.$

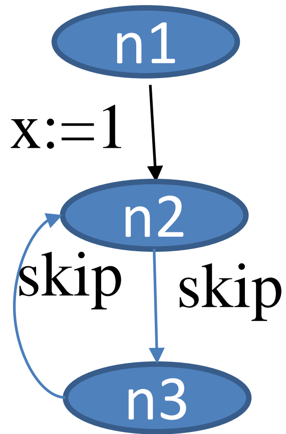
$[\![\text{assume } x \leq 100]\!]^{\#} = \lambda x.$

$DF[n_1]$	$DF[n_2]$	$DF[n_3]$	$DF[n_4]$
$[-\infty, \infty]$	[1,1001]	[1, 1000]	[1001, 1001]
$[-\infty, \infty]$	$\{x \mid 1 \leq x \leq 1000\}$	$\{x \mid 1 \leq x \leq 1001\}$	{1002}
Z	$\{x \mid x \leq 1000\}$	$\{x \mid x \leq 1001\}$	{1001}

Solving the Equations

- For programs with loops the equations have many solutions
- Every solution is sound
- Compute a minimal solution

An Example with Multiple Solutions



$DF(n_1) = [-\infty, \infty]$

$DF(n_2) = [\![x:=1]\!]^\# DF(n_1) \sqcup [\![skip]\!]^\# DF(n_3)$

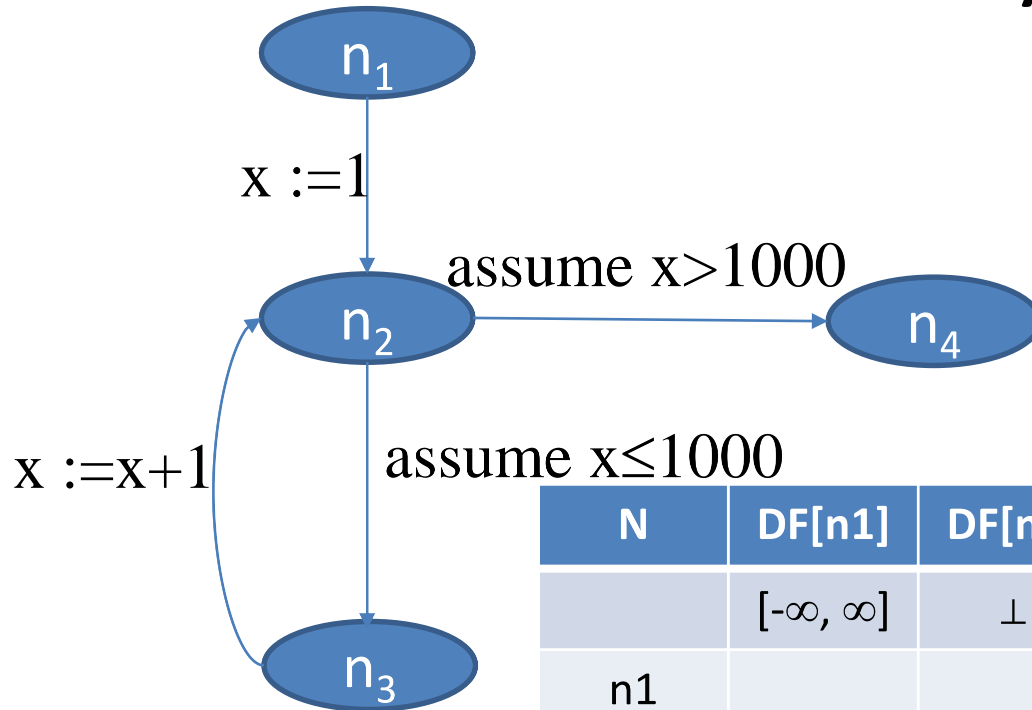
$DF(n3) = [\![skip]\!]^\# DF(n_2)$

DF[n1]	DF[n2]	DF[n3]	Comments
[-∞, ∞]	[-∞, ∞]	[-∞, ∞]	Maximal
[-∞, ∞]	[1, 1]	[1, 1]	Minimal
[-∞, ∞]	[1, 2]	[1, 2]	Solution
[-∞, ∞]	[1, 1]	[1, 2]	Not a solution

Computing Minimal Solution

- Initialize the interval at the entry according to program semantics
- Initialize the rest of the intervals to empty
- Iterate until no more changes

Iterations Interval Analysis

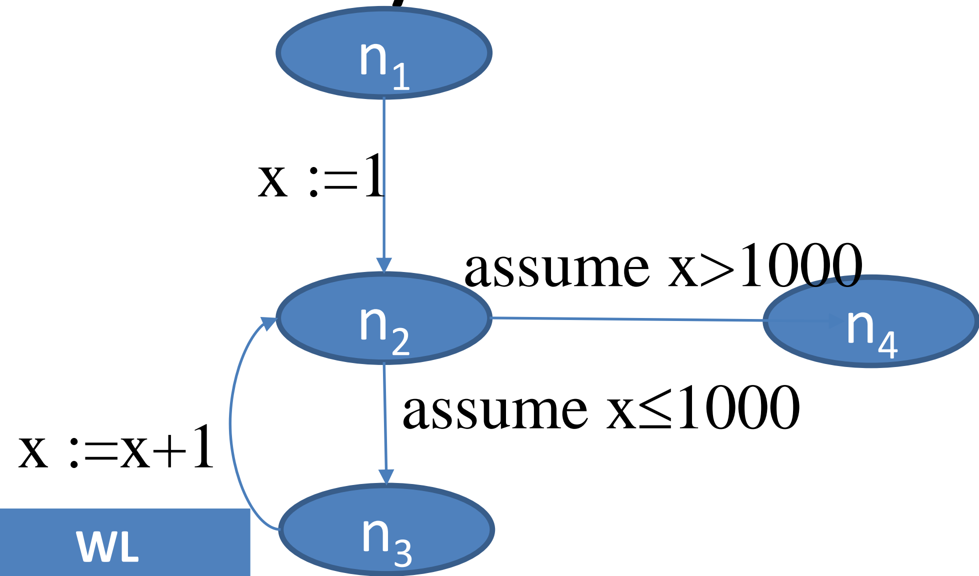


N	DF[n1]	DF[n2]	DF[n3]	DF[n4]
	$[-\infty, \infty]$	$\bot$	$\bot$	$\bot$
n1				
n2		[1,1]		
n3			[1, 1]	
n2		[1, 2]		
n3			[1, 2]	

Iterative Algorithm

```
Chaotic(G(V, E): Graph, s: Node, L: Lattice, ι: L, f: E →(L →L) ){
   for each v in V to n do DF[v] := ⊥
  df[v] = ι
   WL = {s}
   while (WL ≠   ∅ )  do
      select and remove an element u ∈ WL
      for each v, such that. (u, v) ∈E do
              temp = f(e)(DF[u])
              new := DF[v]⊔ temp
              if (new ≠ DF[V]) then
                     DF[v] := new;
                     WL := WL ∪{v}
```

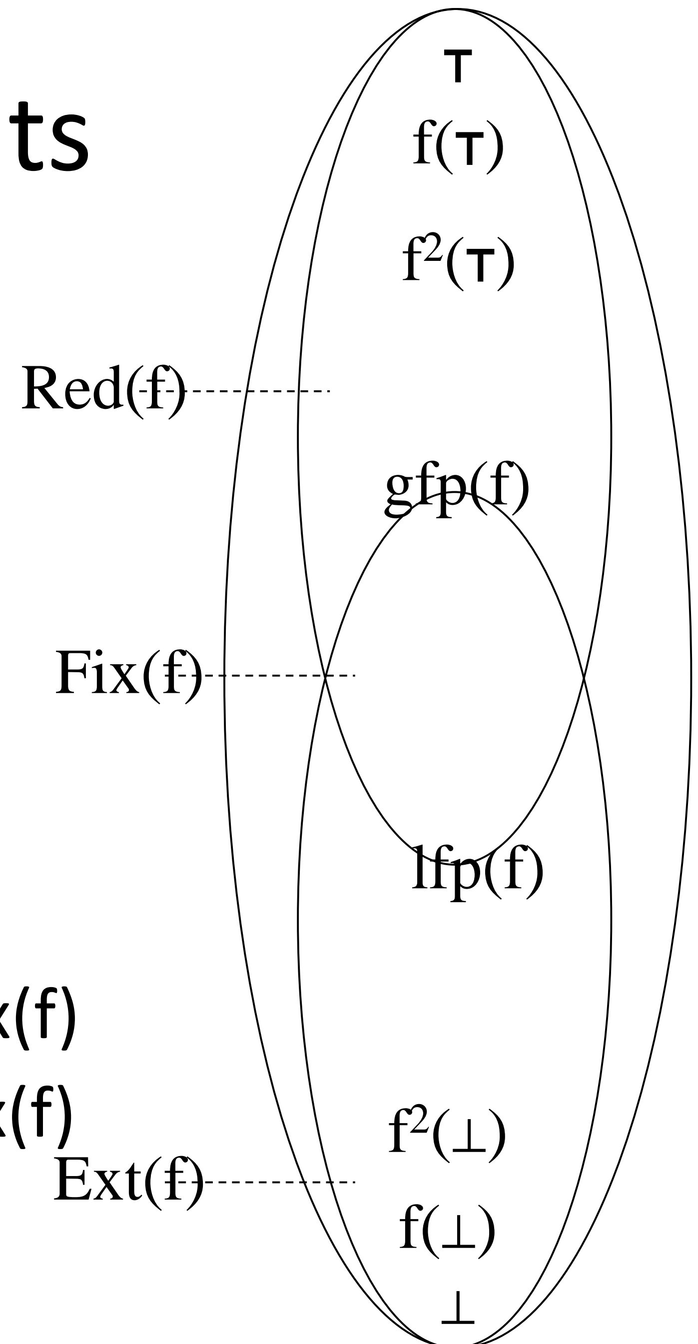
Iterations Interval Analysis

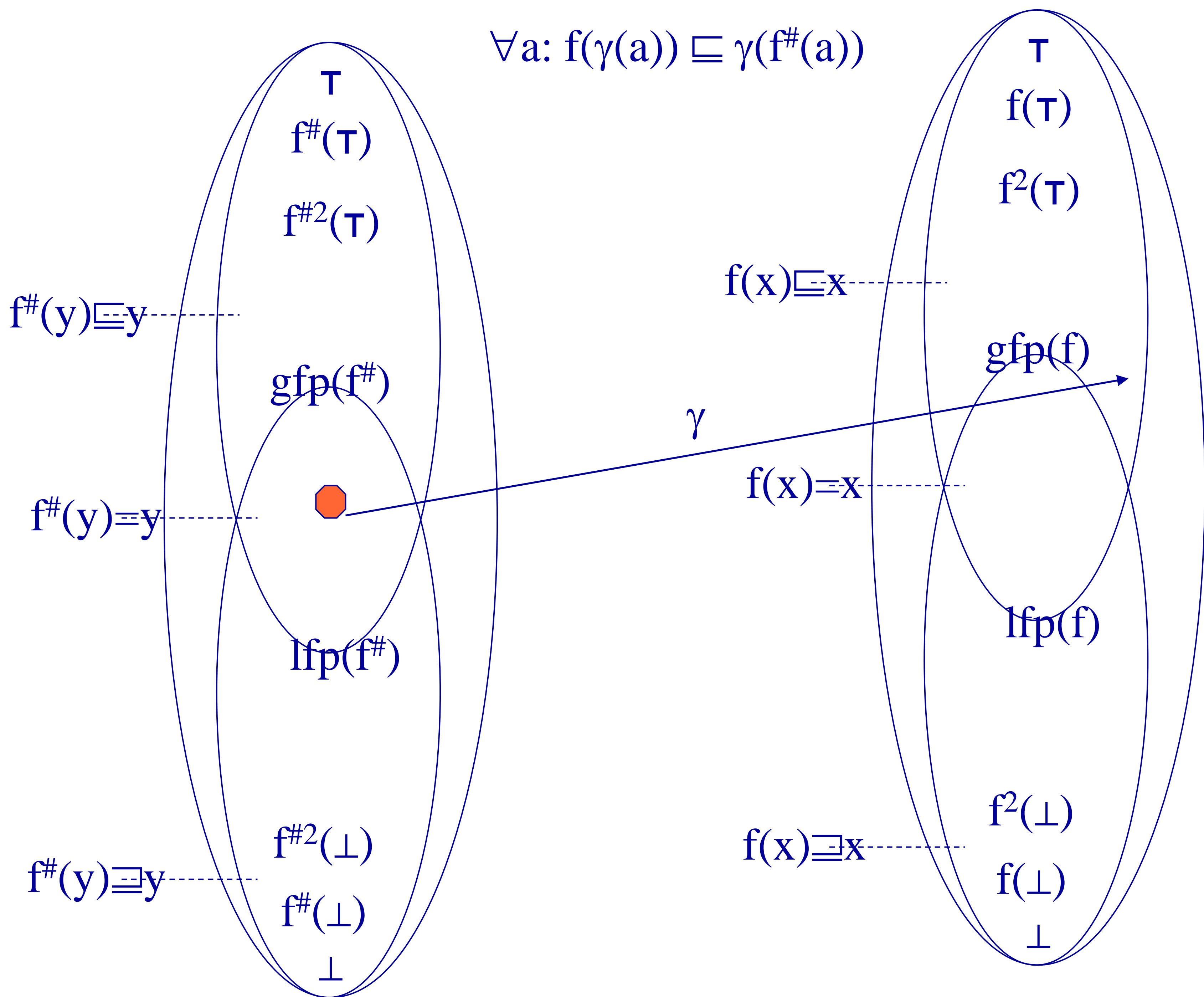


N	DF[n1]	DF[n2]	DF[n3]	DF[n4]	WL
	[-∞, ∞]	⊥	⊥	⊥	{n1, n2, n3, n4}
n1					
n2		[1,1]			{n2, n3, n4}
n3			[1, 1]		{n2, n4}
n2		[1, 2]			{n3, n4}
n3			[1, 2]		{n2, n4}

Fixed Points

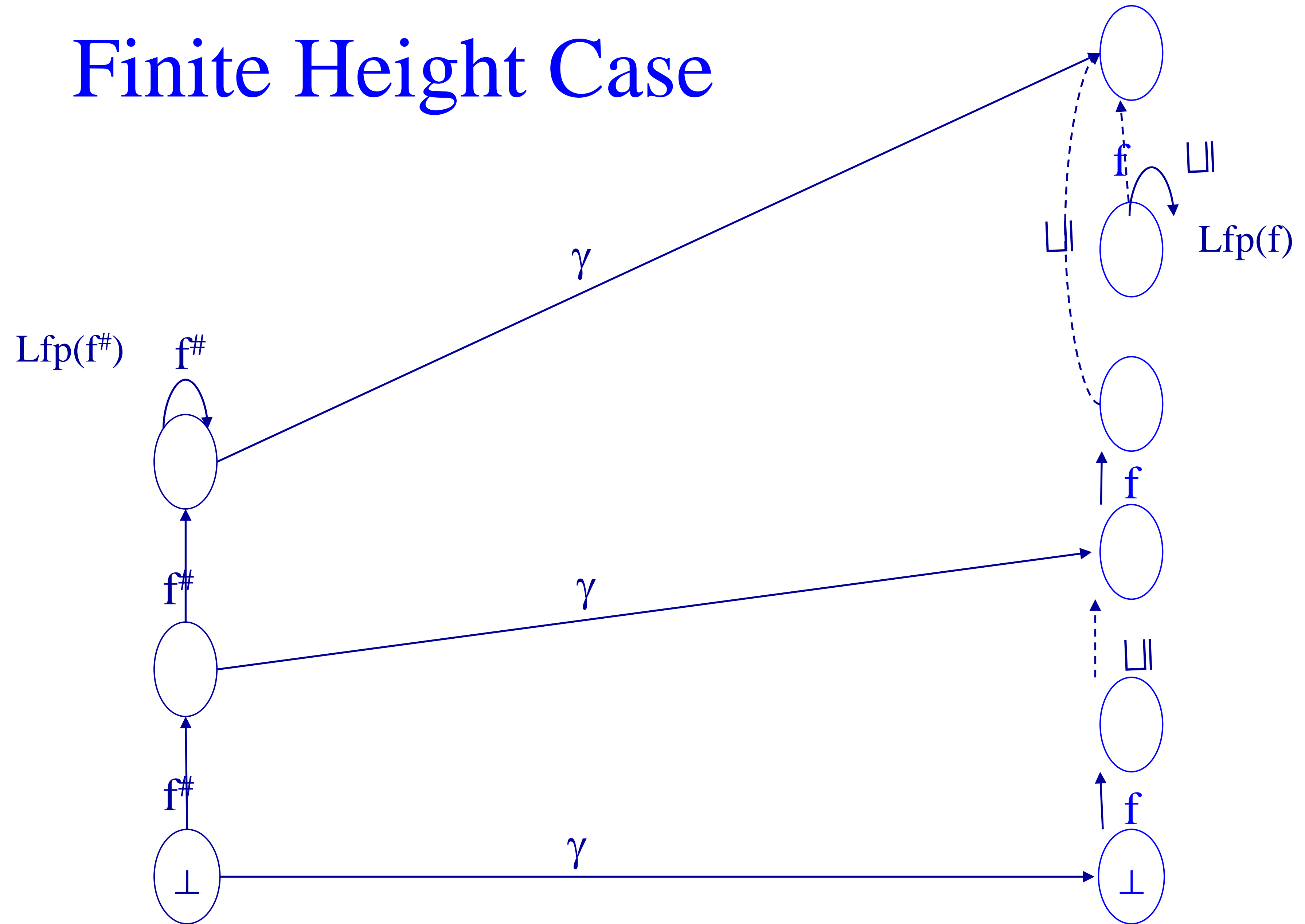
- A monotone function $f: L \to L$ where $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice
- $Fix(f) = \{ l: l \in L, f(l) = l\}$
- $Red(f) = \{l: l \in L, f(l) \sqsubseteq l\}$
- $Ext(f) = \{l: l \in L, l \sqsubseteq f(l)\}$
  - $l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$
- Tarski's Theorem 1955: if f is monotone then:
  - $lfp(f) = \sqcap Fix(f) = \sqcap Red(f) \in Fix(f)$
  - $gfp(f) = \sqcup Fix(f) = \sqcup Ext(f) \in Fix(f)$





∀a: f(γ(a)) ⊑ γ(f#(a))
T
f#(T)
f#2(T)
f#(y)⊑y
gfp(f#)
f#(y)=y
lfp(f#)
f#2(⊥)
f#(y)⊒y
f#(⊥)
⊥
γ
T
f(T)
f2(T)
f(x)⊑x
gfp(f)
f(x)=x
lfp(f)
f2(⊥)
f(x)⊒x
f(⊥)
⊥

Finite Height Case



Accelerating Convergence

- The Iterative algorithm can diverge when the domains contains infinite increasing chains
- Sometimes can take long time

Widening

- Accelerate the convergence of the iterative procedure by jumping to a more conservative solution
- Heuristic in nature
- But simple to implement

Widening for Interval Analysis

- $\bot \nabla [c, d] = [c, d]$
- $[a, b] \nabla [c, d] = [$

  if $a \leq c$

  then a

  else $-\infty$,

  if $b \geq d$

  then b

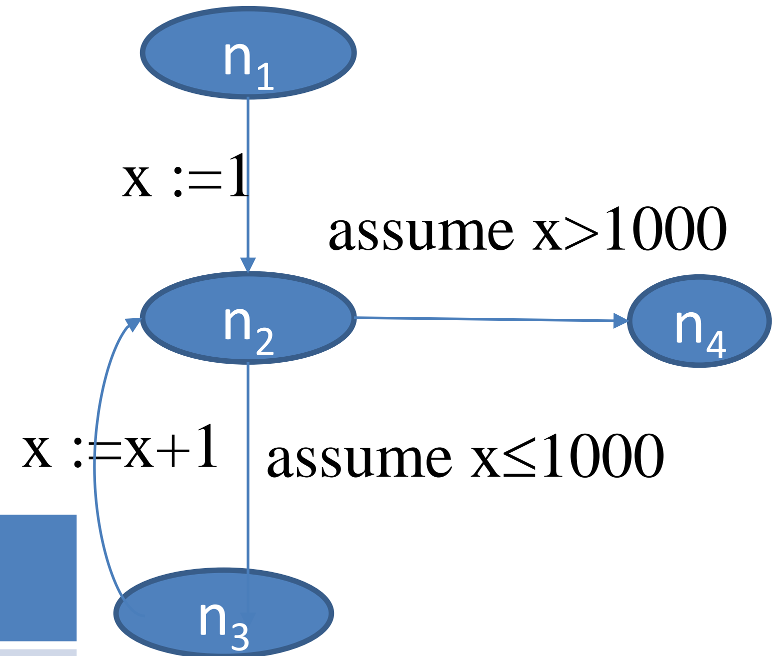
  else ∞

  ]

Iterations Interval Analysis with widening

$DF[n_1] = [-\infty, \infty]$

$DF[n_2] = DF[n2] \triangledown [\![x := 1]\!]^{\#} DF[n_1] \cup [\![x := x+1]\!]^{\#} DF[n_3]$



$DF[n3] = [\![\text{assume } x \leq 100]\!]^{\#} DF[n_2]$

$DF[n4] = [\![\text{assume } x > 100]\!]^{\#} DF[n_2]$

N	DF[n1]	DF[n2]	DF[n3]	DF[n4]	WL
	[-∞, ∞]	⊥	⊥	⊥	{n1, n2, n3, n4}
n1					
n2		[1,1]			{n2, n3, n4}
n3			[1, 1]		{n2, n4}
n2		[1, ∞]			{n3, n4}
n3			[1, 1000]		{ n4}

Widening

$y_k = y_k \nabla f(y_k)$

$lfp(f)$

$\vdots$

$y_2 = y_1 \nabla f(y_1)$

$x_2 = f^2(\perp)$

$y_1 = \perp \nabla f(\perp)$

$x_1 = f(\perp)$

$x_0 = \perp$

Narrowing

- Improve the precision of widened solution
- Heuristic in nature
- But simple to implement

Narrowing for Interval Analysis

- $[a, b] \triangle \bot = [a, b]$
- $[a, b] \triangle [c, d] = [$

  if $a = -\infty$

  then c

  else a,

  if $b = \infty$

  then d

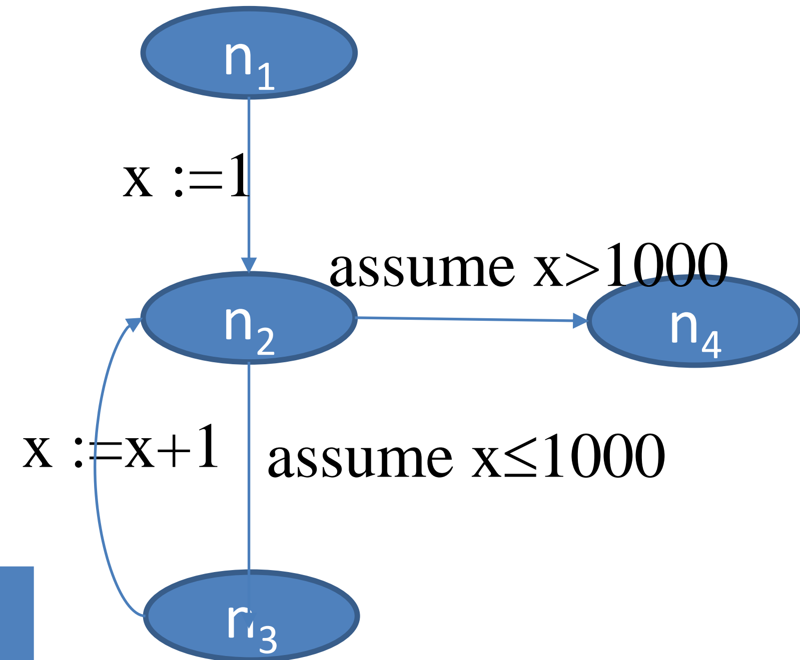
  else b

  ]

Iterations with narrowing after widening

$DF[n_1] = [-\infty, \infty]$

$DF[n_2] = \color{red}{DF[n2]\ \triangle}\ [\![x := 1]\!]^{\#} DF[n_1]$
$\cup$
$[\![x := x+1]\!]^{\#} DF[n_3]$

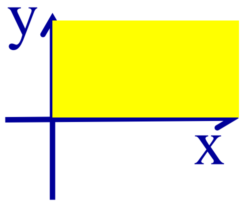


$DF[n3] = [\![\text{assume } x \leq 100\ ]\!]^{\#} DF[n_2]$

$DF[n4] = [\![\text{assume } x > 100\ ]\!]^{\#} DF[n_2]$

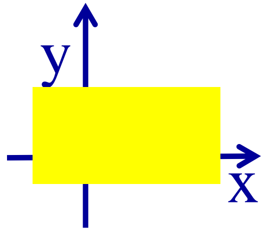
N	DF[n1]	DF[n2]	DF[n3]	DF[n4]	WL
	[-∞, ∞]	[1, ∞]	[1, 1000]	[1001, ∞]	{n1, n2, n3, n4}
n2		[2, 1001]			{n2, n3, n4}
n3			[1, 1]		{ n4}
				[1001, 1001]	{n3, n4}

Numeric Abstract Domain Examples



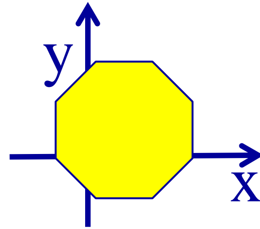
signs

$x \geq 0$



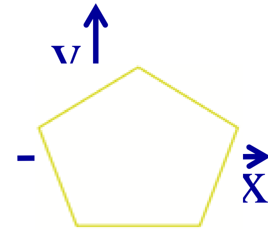
intervals

$x \in [a, b]$



octagons

$\pm x \pm y \leq c$



polyhedra

$\Sigma a_i \, x_i \leq c$

Pointer Language

$a ::= x \mid *x \mid \&x \,/\, \ldots$

$b ::=$ true | a = a| not b

assume b

$x := a$

$*x := y$

Collecting Semantics for Pointers

State1= $[Loc \rightarrow Loc \cup Z]$

Points-To Analysis

- Lattice L_{pt} =
- Galois connection
- Meaning of statements

```
t := &a;
y := &b;

z := &c;

 if x> 0
         then p:= &y;
         else p:= &z;

 *p := t;
```

```
/*  ∅ */ t := &a; /* {(t, a)}*/
 /* {(t, a)}*/  y := &b; /* {(t, a), (y, b) }*/

/* {(t, a), (y, b)}*/ z := &c; /* {(t, a), (y, b), (z, c) }*/

 if x> 0;
        then p:= &y; /* {(t, a),  (y, b), (z, c), (p, y)}*/


        else p:= &z; /* {(t, a),  (y, b), (z, c), (p, z)}*/
/* {(t, a),  (y, b), (z, c), (p, y), (p, z)}*/

 *p := t;

/* {(t, a), (y, b), (y, c), (p, y), (p, z), (y, a), (z, a)}*/
```

Abstract Transformers

$State^{\#} = P(Var^{*} \times Var^{*})$

$$[\![x := a]\!]^{\#}$$

$$[\![x := \&y]\!]^{\#}$$

$$[\![x := {*}y]\!]^{\#}$$

$$[\![x := y]\!]^{\#}$$

$$[\![{*}x := y]\!]^{\#}$$

$$[\![\text{assume } x == y]\!]^{\#}$$

$$[\![\text{assume } x \mathrel{!}= y]\!]^{\#}$$

```
/*  ∅ */ t := &a; /* {(t, a)}*/
 /* {(t, a)}*/  y := &b; /* {(t, a), (y, b) }*/

/* {(t, a), (y, b)}*/ z := &c; /* {(t, a), (y, b), (z, c) }*/

 if x> 0;
        then p:= &y; /* {(t, a),  (y, b), (z, c), (p, y)}*/


        else p:= &z; /* {(t, a),  (y, b), (z, c), (p, z)}*/
/* {(t, a),  (y, b), (z, c), (p, y), (p, z)}*/

 *p := t;

/* {(t, a), (y, b), (y, c), (p, y), (p, z), (y, a), (z, a)}*/
```

Flow insensitive points-to-analysis Steengard 1996

- Ignore control flow
- One set of points-to per program
- Can be represented as a directed graph
- Conservative approximation
  - Accumulate pointers
- Can be computed in almost linear time
  - Union find

```
t := &a;
y := &b;

z := &c;

 if x> 0;
         then p:= &y;
         else p:= &z;

 *p := t;
```

Precision

- We cannot usually have
  - $\alpha(CS) = DF$
    on all programs
- But can we say something about precision in all programs?

The Join-Over-All-Paths (JOP)

- Let paths(v) denote the potentially infinite set paths from start to v (written as sequences of edges)
- For a sequence of edges [e$_1$, e_2, ..., e_n] define $f^{\#}$[e$_1$, e_2, ..., e_n]: $L \rightarrow L$ by composing the effects of basic blocks
  $f^{\#}$[e$_1$, e_2, ..., e_n](l) $= f^{\#}(e_n)$ (… ($f^{\#}(e_2)$ ($f^{\#}(e_1)$ (l)) …)
- JOP[v] $= \sqcup\{f^{\#}[e_1, e_2, \ldots, e_n](\iota)$
  $[e_1, e_2, \ldots, e_n] \in$ paths(v)$\}$

JOP vs. Least Solution

- The df solution obtained by Chaotic iteration satisfies for every *v*:
  - $JOP[v] \sqsubseteq df[v]$
- A function $f^{\#}$ is additive (distributive) if
  - $f^{\#}(\sqcup\{x \mid x \in X\}) = \sqcup\{f^{\#}(x) \mid \in X\}$
- If every $f^{\#}_{(u,v)}$ is additive (distributive) for all the edges (u,v)
  - $JOP[v] = df[v]$
- Examples
  - Intervals
  - Points-to

Notions of precision

- $CS = \gamma(df)$
- $\alpha(CS) = df$
- Meet(Join) over all paths
- Using best transformers
- Good enough

Complexity of Chaotic Iterations

- Usually depends on the height of the lattice
- In some cases better bound exist
- A function f is fast if $f(f(l)) \sqsubseteq l \sqcup f(l)$
- For fast functions the Chaotic iterations can be implemented in O(nest * |V|) iterations
  - nest is the number of nested loop
  - |V| is the number of control flow nodes

Static Analysis Projects

- Implement static analysis for a toy set language inspired by Python
- Implement static analysis on top of LLVM
  - Flow sensitive points-to for programs w/o procedures
- Implement static analysis on top of SOOT
  - Flow sensitive points-to for programs w/o procedures
- Implement k-CNF based extension on points to analysis for a toy language
  - p = &q

Conclusion

- Static analysis is powerful technique
- But expensive
  - More efficient methods exist for structured programs
- Abstract interpretation relates runtime semantics and static information
- The concrete semantics serves as a tool in designing abstractions