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

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);
        p = NULL;
       }
```

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

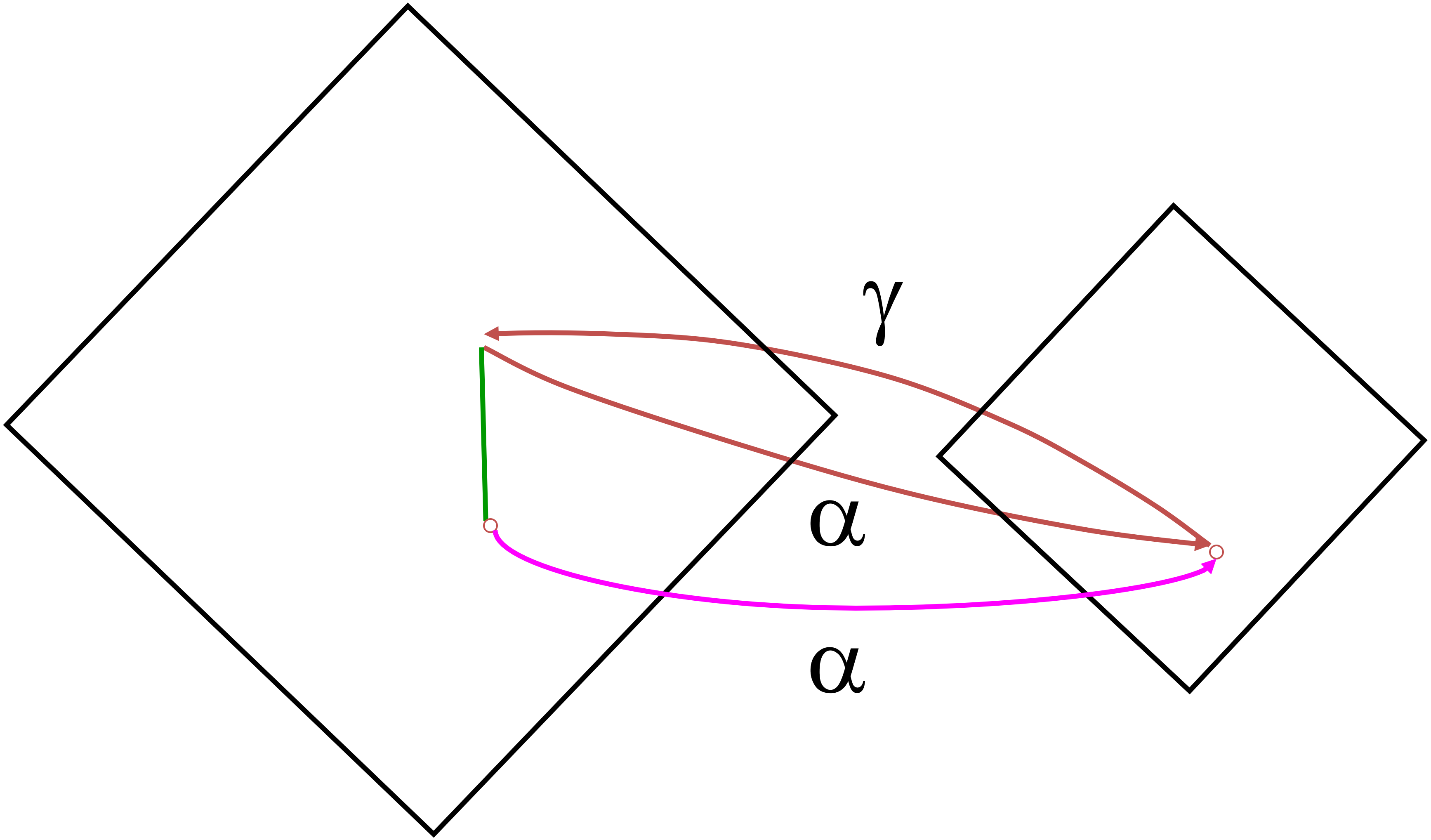
Even/Odd Abstract Interpretation

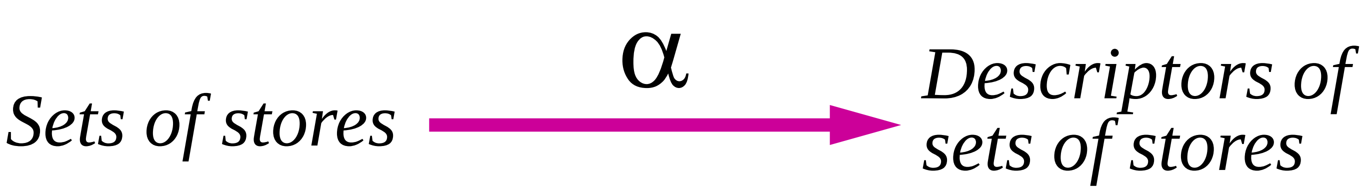
- Determine if an integer variable is even or odd at a given program point

Example Program

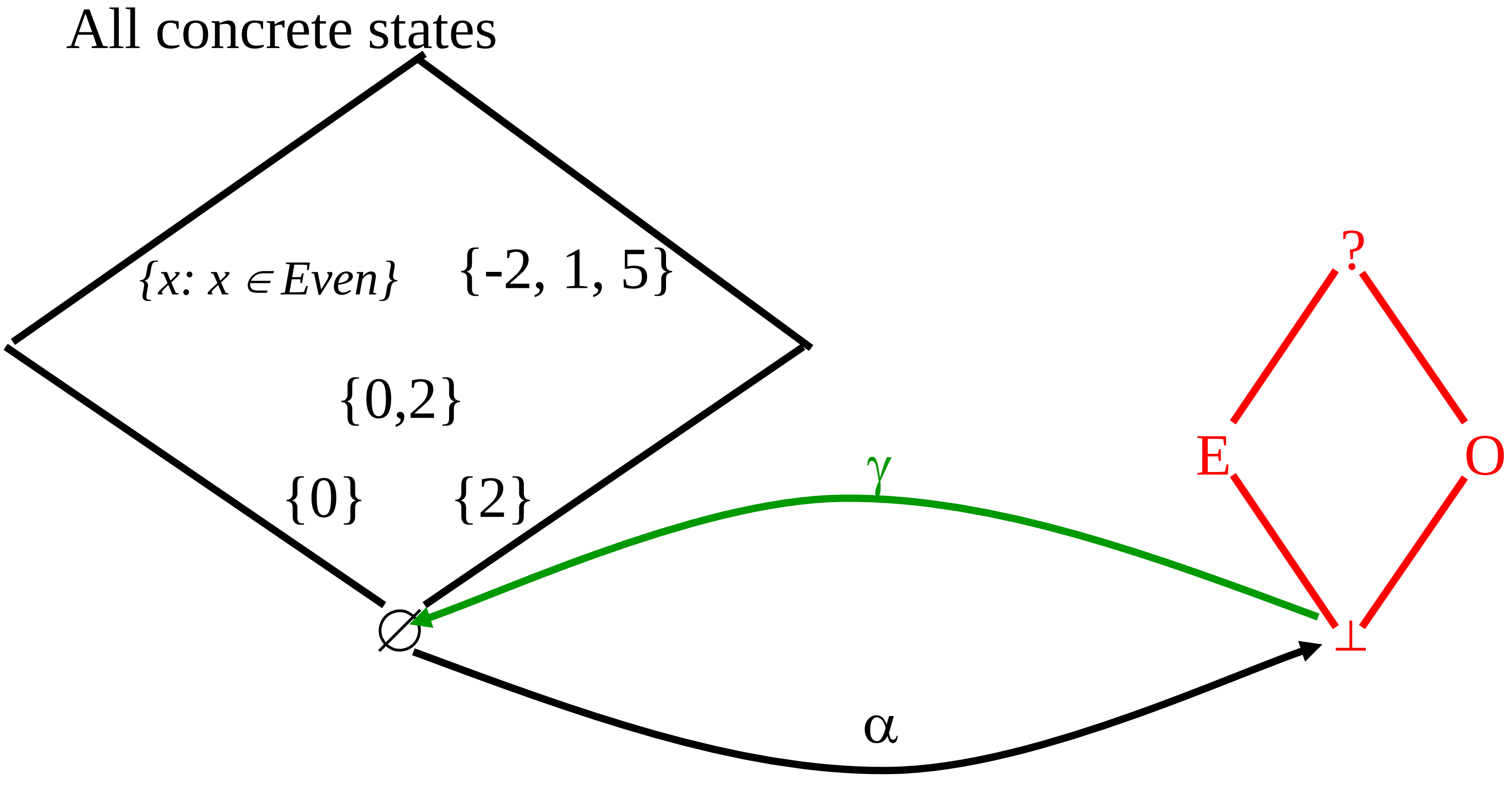
```
/* x=? */

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*/
```

Abstract Interpretation

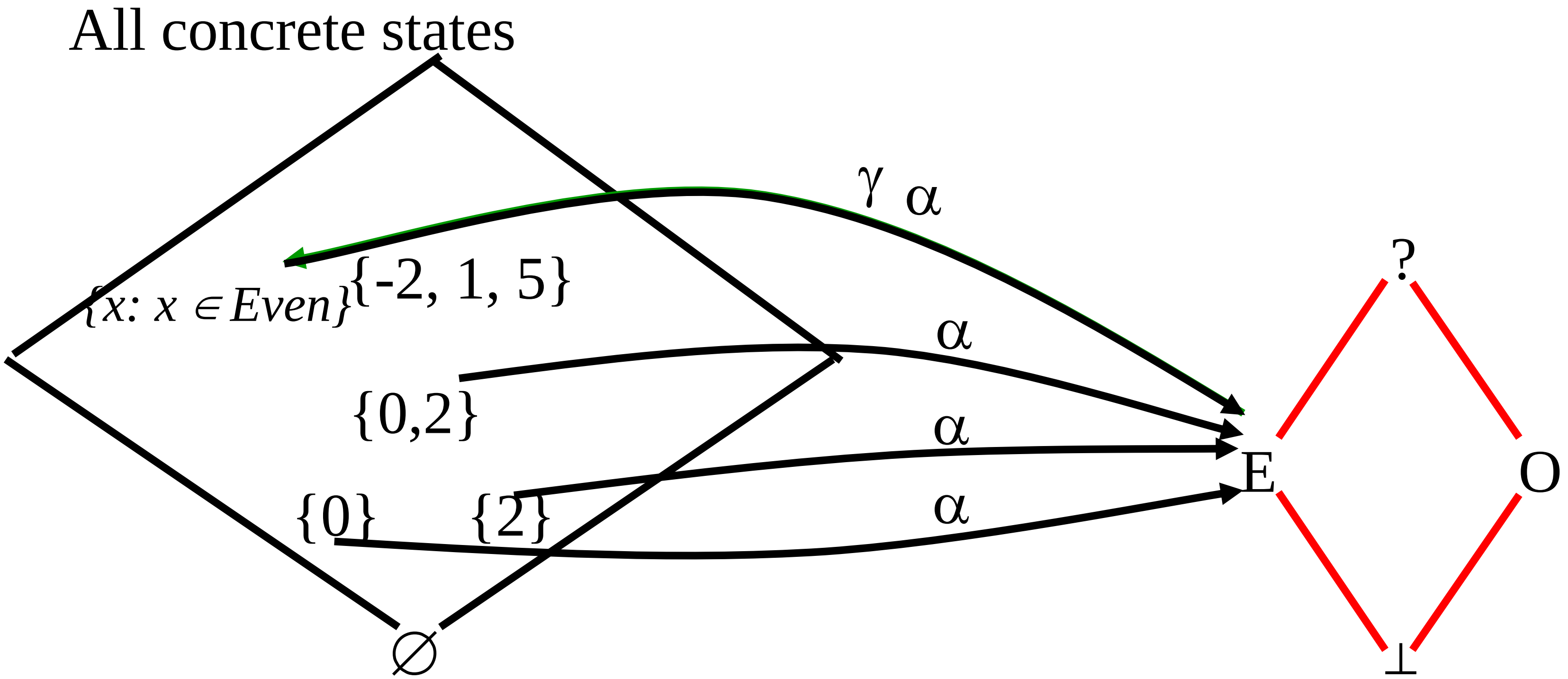




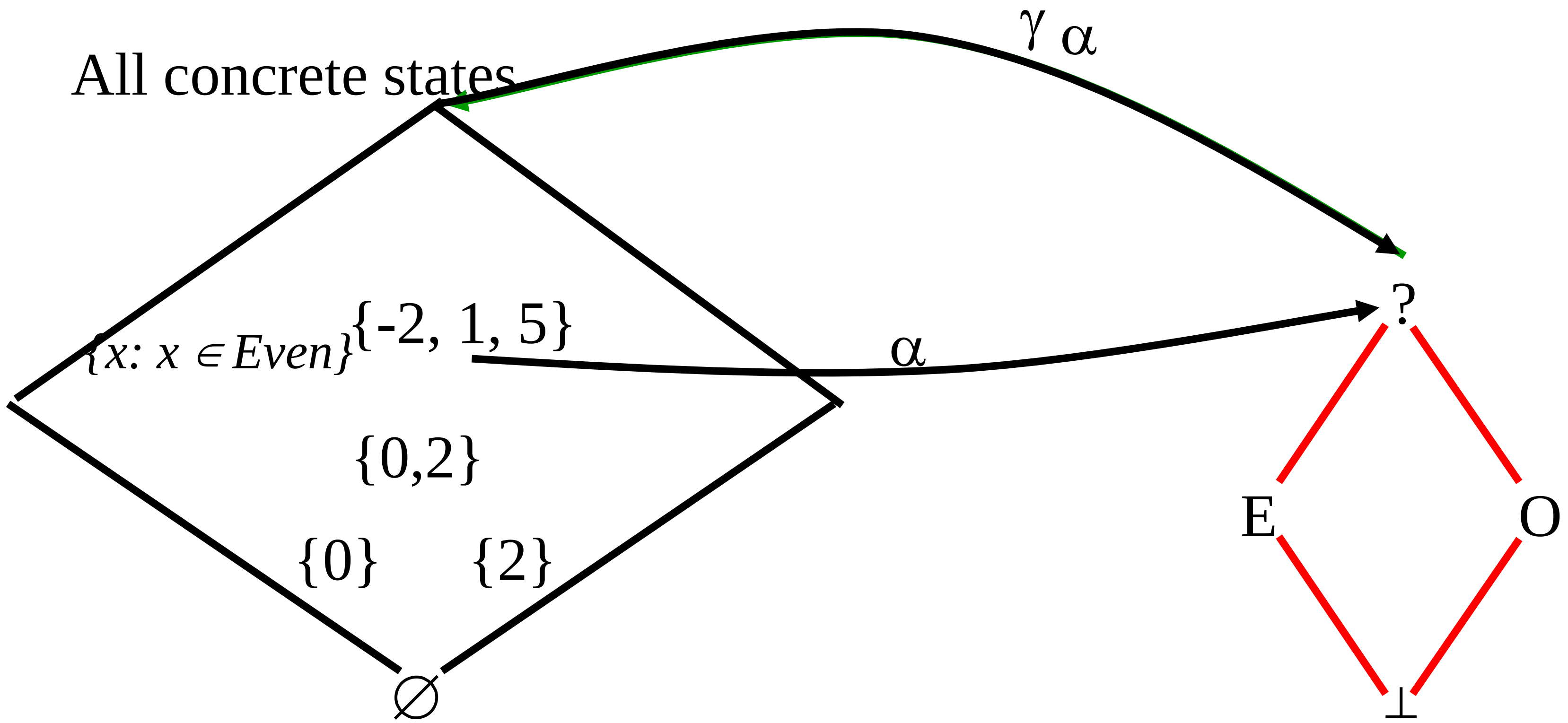
Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation



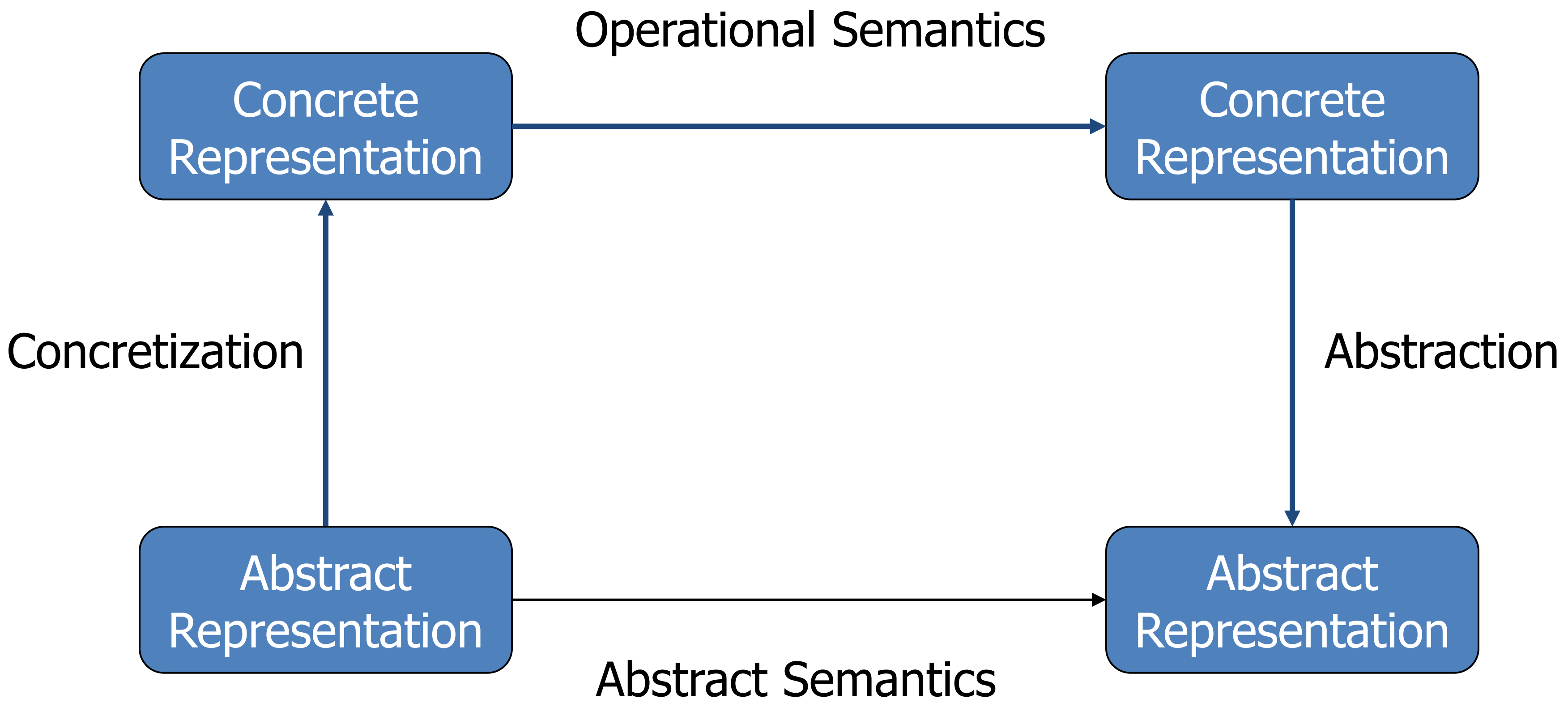
Odd/Even Abstract Interpretation



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



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 system of equations over the abstract values
- Iteratively compute the least solution to the system
- The solution is guaranteed to be sound
- The correctness of the invariants 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]}
    }
```

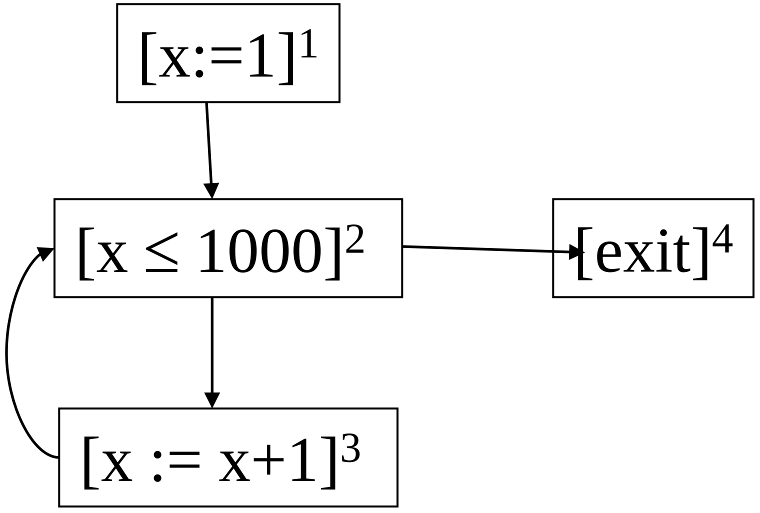
Example Program Interval Analysis

[x := 1]^1 ;

while [x $\leq$ 1000]^2

do

[x := x + 1;]^3



Abstract Interpretation of Atomic Statements

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

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

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

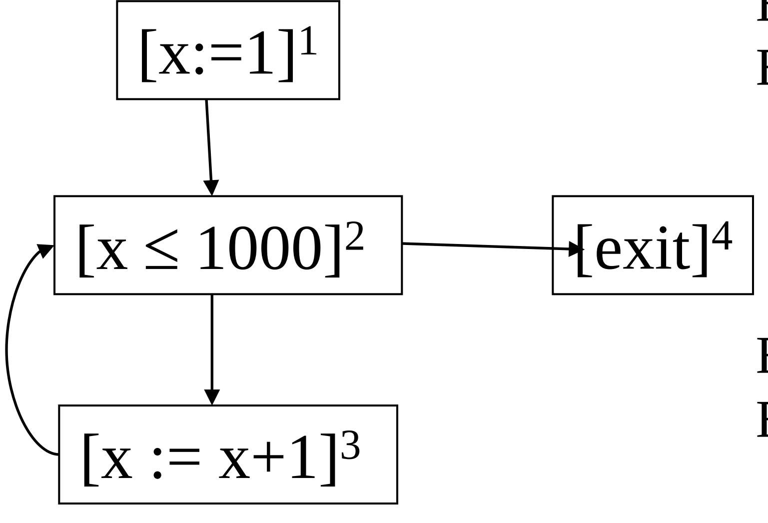
Equations Interval Analysis

[x := 1][1] ;

while [x $\leq$ 1000][2]

do

    [x := x + 1;][3]



En(1) = [minint,maxint]

Ex(1) = [1,1]

In(2) =

Ex(2) = In(2)

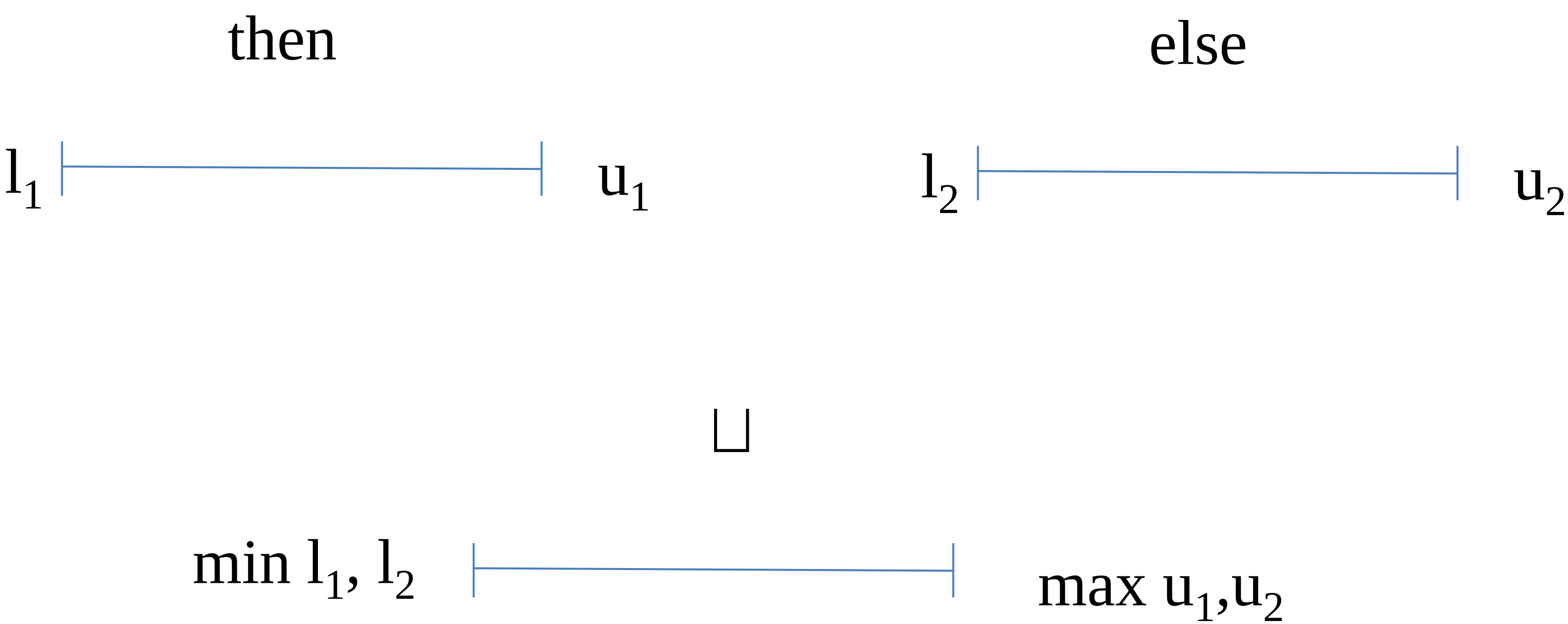
En(3) =

Ex(3) = In(3)+[1,1]

En(4) =

Ex(4) = In(4)

Abstract Interpretation of Joins



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

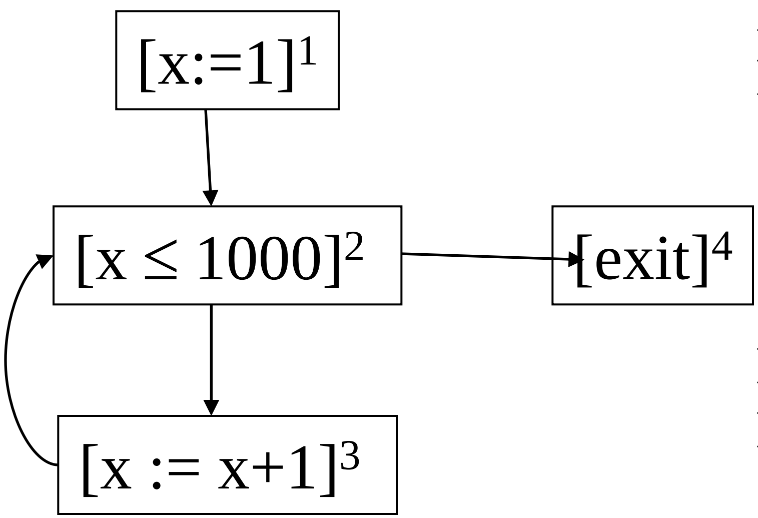
Equations Interval Analysis

[x := 1][1] ;

while [x $\leq$ 1000][2]

do

[x := x + 1;][3]



En(1) = [minint,maxint]
Ex(1) = [1,1]

En(2) = En(1) $\sqcup$ En(3)
Ex(2) = En(2)

En(3) =
Ex(3) = En(3)+[1,1]

En(4) =
Ex(4) = En(4)

Abstract Interpretation of Meets

assume $\qquad\qquad$ assume

l_1 $\vdash\!\!\!-\!\!\!-\!\!\!\dashv$ u_1 $\qquad$ l_2 $\vdash\!\!\!-\!\!\!-\!\!\!\dashv$ u_2

$\sqcap$

$\max l_1, l_2$ $\vdash\!\!\!-\!\!\!-\!\!\!\dashv$ $\min u_1,u_2$

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

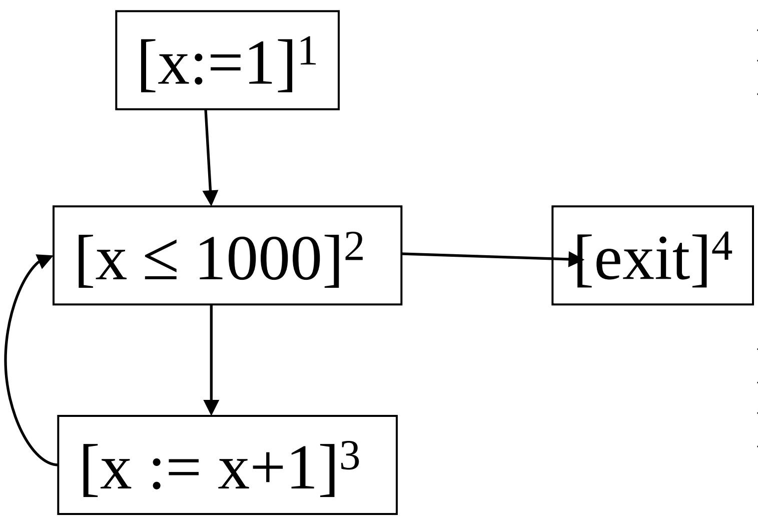
Equations Interval Analysis

[x := 1]^1 ;

while [x $\leq$ 1000]^2

do

[x := x + 1;]^3



En(1) = [minint, maxint]
Ex(1) = [1,1]

En(2) = Ex(1) $\sqcup$ Ex(3)
Ex(2) = En(2)

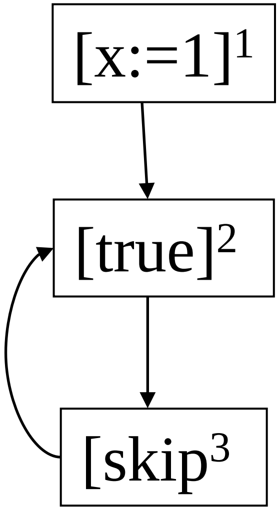
En(3) = Ex(2) $\sqcap$ [minint,1000]
Ex(3) = En(3)+[1,1]

En(4) = Ex(2) $\sqcap$ [1001,maxint]
Ex(4) = En(4)

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



En(1) = [minint,maxint]
Ex(1) = [1,1]

In(2) = Ex(1) $\sqcup$ Ex(3)
Ex(2) = In(2)

IntEntry(3) = IntExit(2)
IntExit(3) = IntEntry(3)

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	Comments
[-∞, ∞]	[1, 1]	[-∞, ∞]	[-∞, ∞]	[-∞, ∞]	[-∞, ∞]	Maximal
[-∞, ∞]	[1, 1]	[1, 1]	[1, 1]	[1, 1]	[1, 1]	Minimal
[-∞, ∞]	[1, 2]	[1, 2]	[1, 2]	[1, 2]	[1, 2]	Solution
[-∞, ∞]	⊥	[1, 1]	[1, 1]	[1, 2]	[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

IntEntry(1) = [minint,maxint]
IntExit(1) = [1,1]

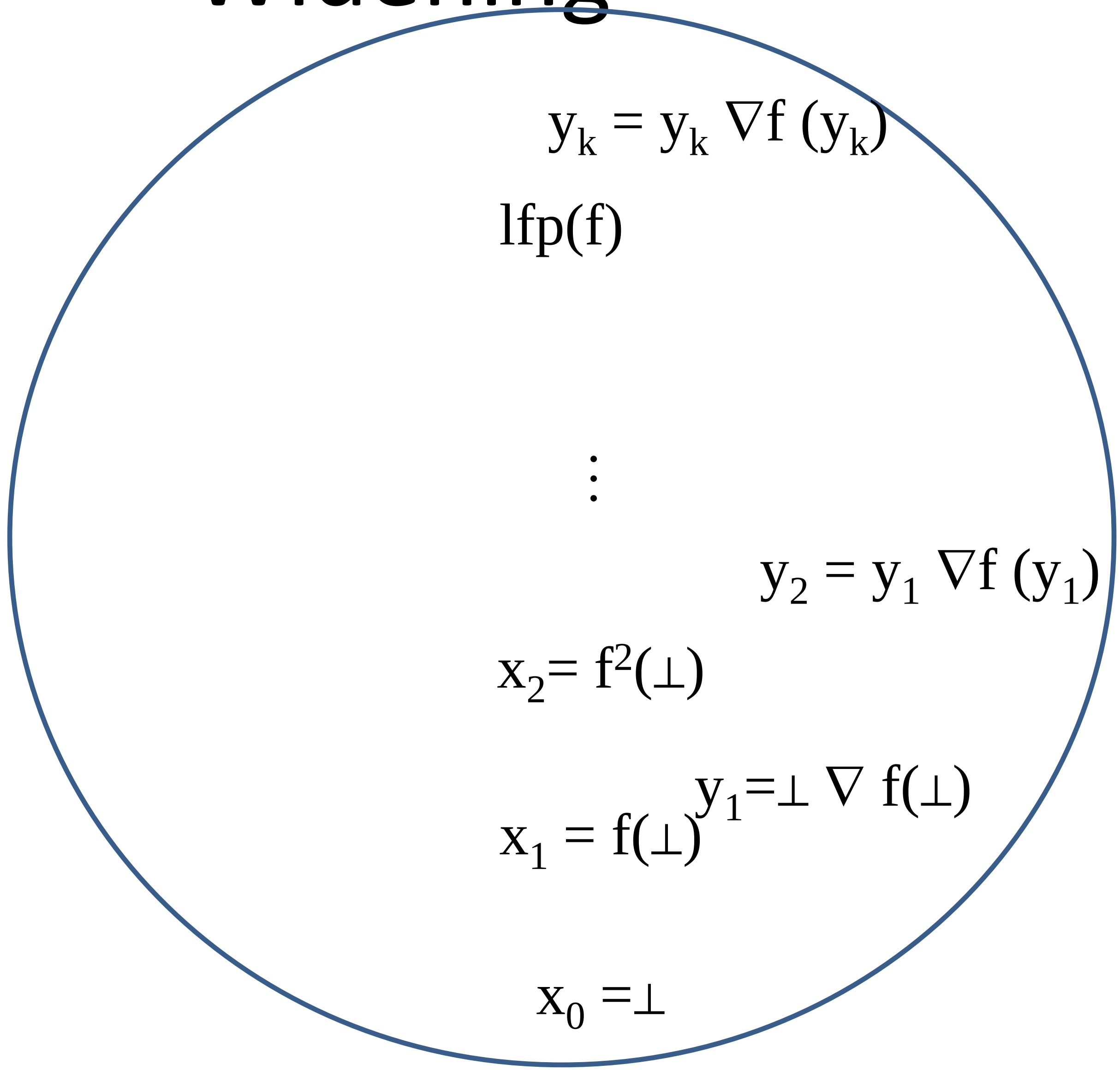
IntEntry(2) = IntExit(1) ⊔ IntExit(3)
IntExit(2) = IntEntry(2)

IntEntry(3) = IntExit(2) ⊓ [minint,1000]
IntExit(3) = IntEntry(3)+[1,1]

IntEntry(4) = IntExit(2) ⊓ [1001,maxint]
IntExit(4) = IntEntry(4)

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	In[4]	Ex[4]
[-∞, ∞]	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	[1, 1]						
		[1,1]					
			[1,1]				
				[1, 1]			
					[2, 2]		
		[1, 2]					

Widening



$y_k = y_k \nabla f\,(y_k)$
lfp(f)
⋮
$y_2 = y_1 \nabla f\,(y_1)$
$x_2 = f^2(\bot)$
$y_1 = \bot \nabla f(\bot)$
$x_1 = f(\bot)$
$x_0 = \bot$

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] ∇ [c, d] = [

    if a $\leq$ c

        then a

        else -∞,

    if b $\geq$ d

        then b

        else ∞

            ]

Iterations with widening

IntEntry(1) = [minint,maxint]

IntExit(1) = [1,1]

IntEntry(2) = IntEntry(2) ∇(IntExit(1) $\sqcup$ IntExit(3))

IntExit(2) = IntEntry(2)

IntEntry(3) = IntExit(2) $\sqcap$ [minint,1000]

IntExit(3) = IntEntry(3)+[1,1]

IntEntry(4) = IntExit(2) $\sqcap$ [1001,maxint]

IntExit(4) = IntEntry(4)

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	In[4]	Ex[4]
[-∞, ∞]	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	[1, 1]						
		[1,1]					
			[1,1]				
				[1, 1]			
					[2, 2]		
		[1, ∞]					
			[1, ∞]				
				[1, 1000]	[2, 1001]		

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

IntEntry(1) = [minint,maxint]
IntExit(1) = [1,1]

IntEntry(2) = IntEntry(2) $\triangle$(IntExit(1) $\sqcup$ IntExit(3))
IntExit(2) = IntEntry(2)

IntEntry(3) = IntExit(2) $\sqcap$ [minint,1000]
IntExit(3) = IntEntry(3)+[1,1]

IntEntry(4) = IntExit(2) $\sqcap$ [1001,maxint]
IntExit(4) = IntEntry(4)

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	In[4]	Ex[4]
[$-\infty$, ∞]	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$
	[1, 1]						
		[1,1]					
			[1,1]				
				[1, 1]			
					[2, 2]		
		[1, ∞]					
			[1, ∞]				
				[1, 1000]	[2, 1001]		

Iterations with narrowing after widening

IntEntry(1) = [minint,maxint]
IntExit(1) = [1,1]

IntEntry(2) = IntEntry(2) △(IntExit(1) ⊔ IntExit(3))
IntExit(2) = IntEntry(2)

IntEntry(3) = IntExit(2) ⊓ [minint,1000]
IntExit(3) = IntEntry(3)+[1,1]

IntEntry(4) = IntExit(2) ⊓ [1001,maxint]
IntExit(4) = IntEntry(4)

En[1]	Ex[1]	En[2]	Ex[2]	En[3]	Ex[3]	In[4]	Ex[4]
[-∞, ∞]	⊥	⊥	⊥	⊥	⊥	⊥	⊥
	[1, 1]						
		[1,1]					
			[1,1]				
				[1, 1]			
					[2, 2]		
		[1, 1001]					
			[1, ∞]				
				[1, 1000]	[2, 1001]		

Summary

- Static analysis is powerful
- Reach theory
- Can locate rear bugs
- Challenges
  - Specification
  - Scalability
  - False alarms
- Can be combined with decision procedures