

An Overview on Static Program Analysis

Mooly Sagiv

<http://www.cs.tau.ac.il/~msagiv/courses/pa.html>

Tel Aviv University

640-6706

Textbook: Principles of Program Analysis

F. Nielson, H. Nielson, C.L. Hankin

Alternative Schedule

- ◆ Wednesday 2-5
- ◆ Wednesday 9-12

Course Requirements

- ◆ Course Notes 15%
- ◆ Assignments 35%
- ◆ Home exam 50%

Class Notes

- ◆ Prepare a document with (word,latex)
 - Original material covered in class
 - Explanations
 - Questions and answers
 - Extra examples
 - Self contained
- ◆ Send class notes by Monday night to msagiv@tau
- ◆ Incorporate changes
- ◆ Available next class

Static Analysis

- ◆ Automatic derivation of static properties which hold on every execution leading to a program location

Example Static Analysis Problem

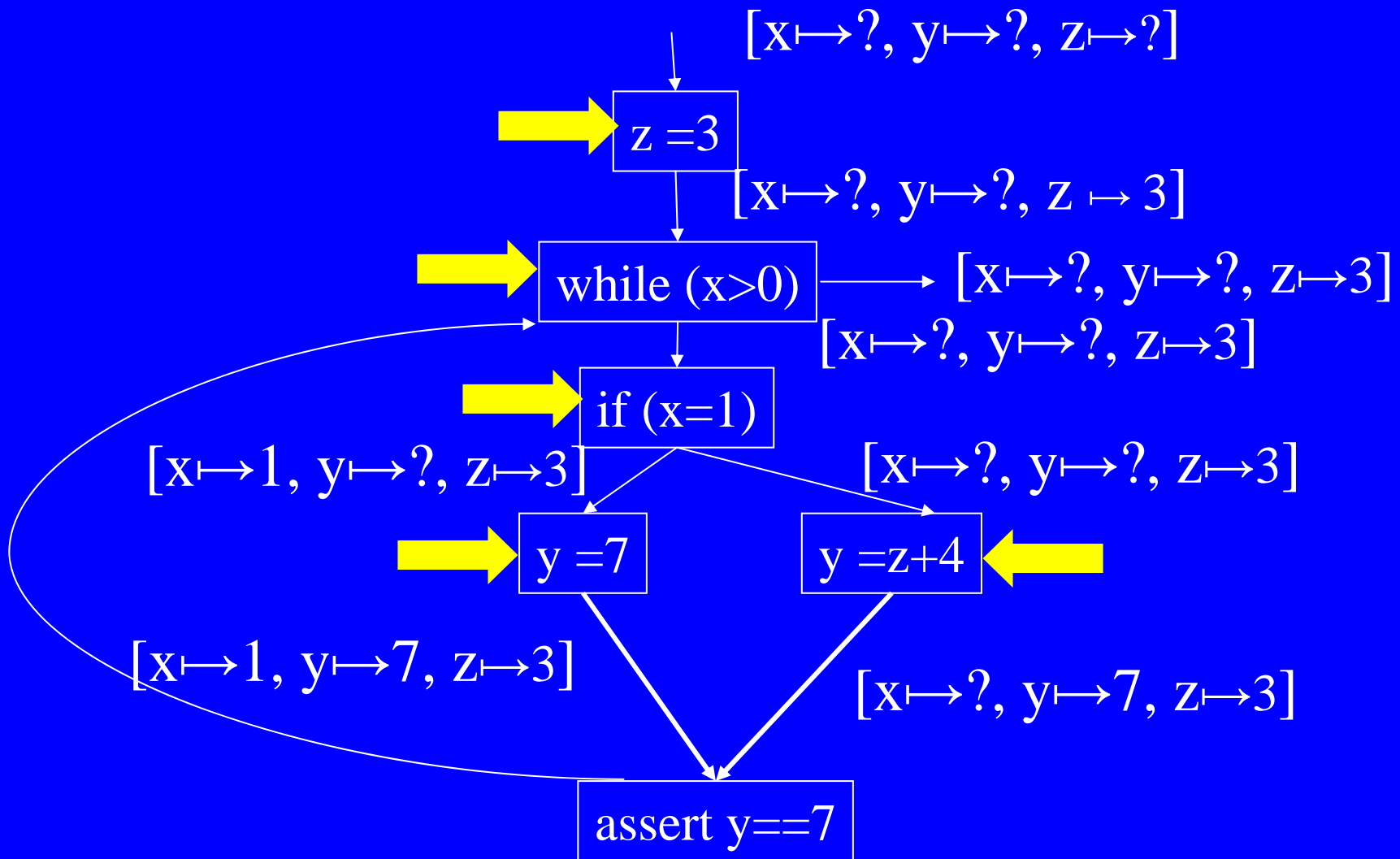
- ◆ Find variables with constant value at a given program location
- ◆ Example program

```
int p(int x){
    return x *x ;
void main()
{
int z;
if (getc())
    z = p(6) + 8;
    else z = p(-7) -5;
printf (z);
}          44
```

Recursive Program

```
int x
void p(a) {
    read (c);
    if c > 0 {
        a = a -2;
        p(a);
        a = a + 2;
    }
    x = -2 * a + 5;
    print (x);
}
void main {
    p(7);
    print(x);
}
```

Iterative Approximation



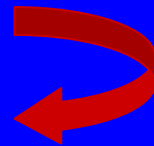
Memory Leakage

```
List reverse(Element *head)
{
    List rev, n;
    rev = NULL;
    while (head != NULL) {
        n = head →next;
        head → next = rev;
        head = n;
        rev = head;
    }
    return rev;
}
```

*potential leakage of address
pointed to by head*

Memory Leakage

```
Element* reverse(Element *head)
{
    Element *rev, *n;
    rev = NULL;
    while (head != NULL) {
        n = head → next;
        head → next = rev;
        rev = head;
        head = n;
    }
    return rev; }
```

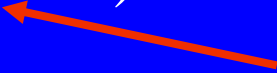


👍 No memory leaks

A Simple Example

```
void foo(char *s )  
{  
    while ( *s != ' ' )  
        s++;  
    *s = 0;  
}
```

Potential buffer overrun:
 $\text{offset}(s) \geq \text{alloc}(\text{base}(s))$



A Simple Example

```
void foo(char *s) @require string(s)
{
    while ( *s != ' ' && *s != 0)
        s++;
    *s = 0;
}
```

👍 No buffer overruns

Example Static Analysis Problem

- ◆ Find variables which are **live** at a given program location
- ◆ Used before set on some execution paths from the current program point

A Simple Example

/ c */*

L0: a := 0

/ ac */*

L1: b := a + 1

/ bc */*

 c := c + b

/ bc */*

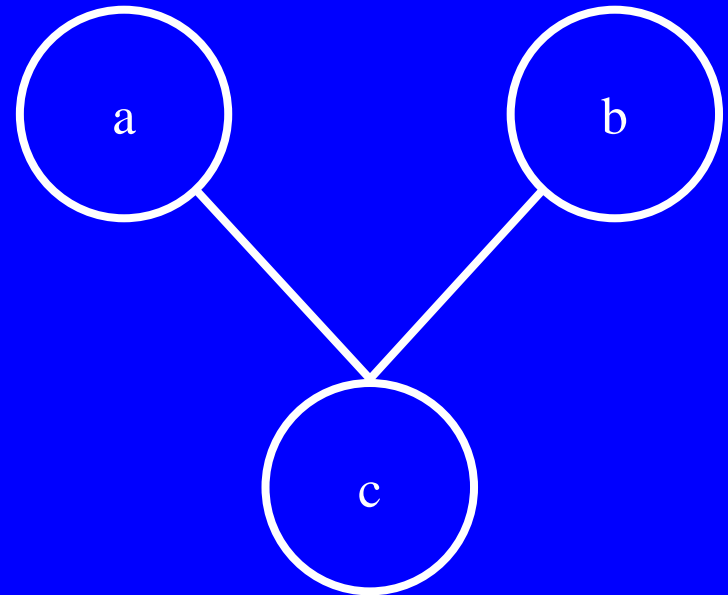
 a := b * 2

/ ac */*

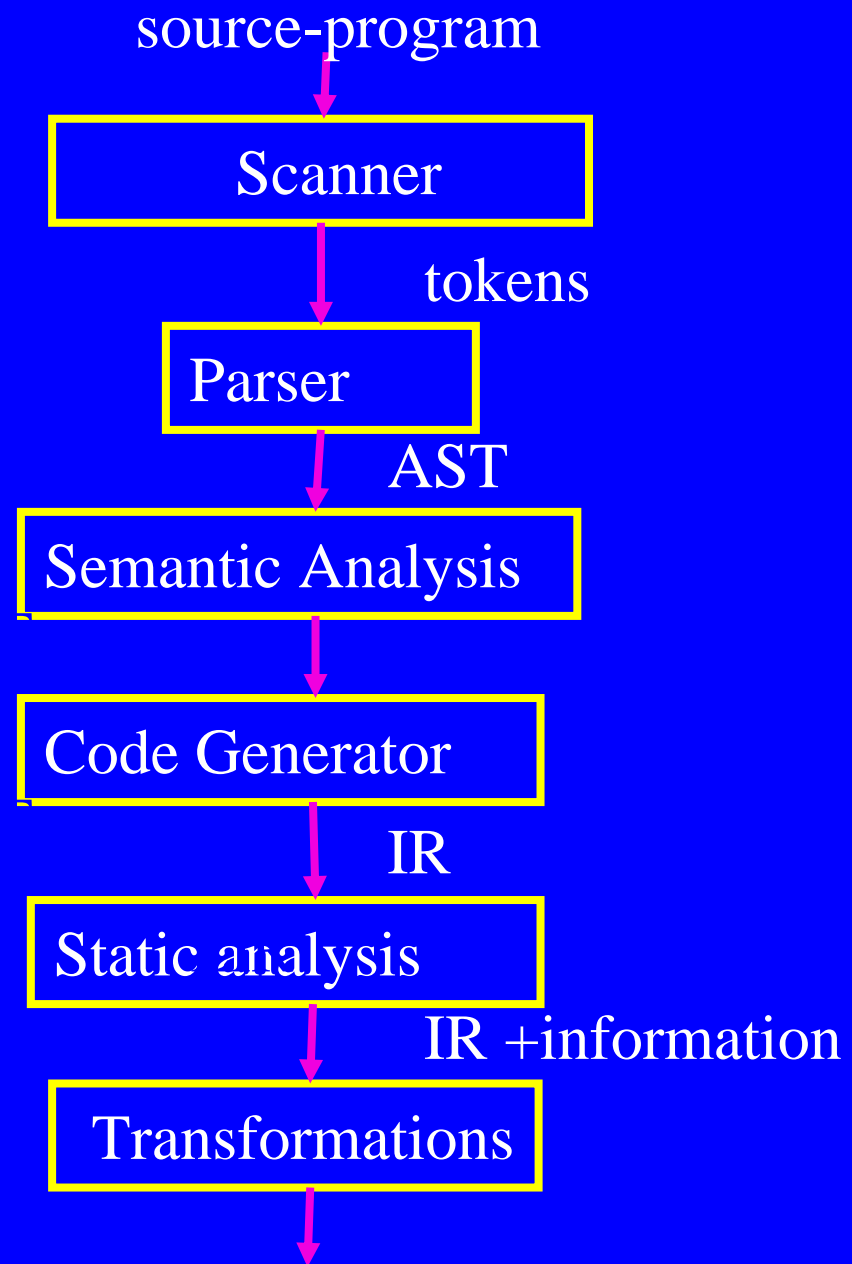
 if c < N goto L1

/ c */*

 return c



Compiler Scheme



Other Example Program Analyses

- ◆ Reaching definitions
- ◆ Expressions that are "available"
- ◆ Dead code
- ◆ Pointer variables never point into the same location
- ◆ Points in the program in which it is safe to free an object
- ◆ An invocation of virtual method whose address is unique
- ◆ Statements that can be executed in parallel
- ◆ An access to a variable which must be in cache
- ◆ Integer intervals

The Need for Static Analysis

◆ Compilers

- Advanced computer architectures
(Superscalar pipelined, VLIW, prefetching)
- High level programming languages
(functional, OO, garbage collected, concurrent)

◆ Software Productivity Tools

- Compile time debugging
 - » Stronger type Checking for C
 - » Array bound violations
 - » Identify dangling pointers
 - » Generate test cases
 - » Generate certification proofs

◆ Program Understanding

Challenges in Static Analysis

- ◆ Non-trivial
- ◆ Correctness
- ◆ Precision
- ◆ Efficiency of the analysis
- ◆ Scaling

C Compilers

- ◆ The language was designed to reduce the need for optimizations and static analysis
- ◆ The programmer has control over performance (order of evaluation, storage, registers)
- ◆ C compilers nowadays spend most of the compilation time in static analysis
- ◆ Sometimes C compilers have to work harder!

Software Quality Tools

- ◆ Detecting hazards (lint)

- Uninitialized variables

```
a = malloc() ;
```

```
b = a;
```

```
cfree (a);
```

```
c = malloc ();
```

```
if (b == c)
```

```
printf(“unexpected equality”);
```

- ◆ References outside array bounds

- ◆ Memory leaks (occurs even in Java!)

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
 - Every identified constant is indeed a constant
 - But not every constant is identified as such

Example Abstract Interpretation

Casting Out Nines

- ◆ Check soundness of arithmetic using 9 values
0, 1, 2, 3, 4, 5, 6, 7, 8
- ◆ Whenever an intermediate result exceeds 8, replace by the sum of its digits (recursively)
- ◆ Report an error if the values do not match
- ◆ Example query “123 * 457 + 76543 = 132654\$?”
 - Left $123 * 457 + 76543 = 6 * 7 + 7 = 6 + 7 = 4$
 - Right 3
 - Report an error

◆ Soundness

$$(10a + b) \bmod 9 = (a + b) \bmod 9$$

$$(a+b) \bmod 9 = (a \bmod 9) + (b \bmod 9)$$

$$(a*b) \bmod 9 = (a \bmod 9) * (b \bmod 9)$$

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=? */*

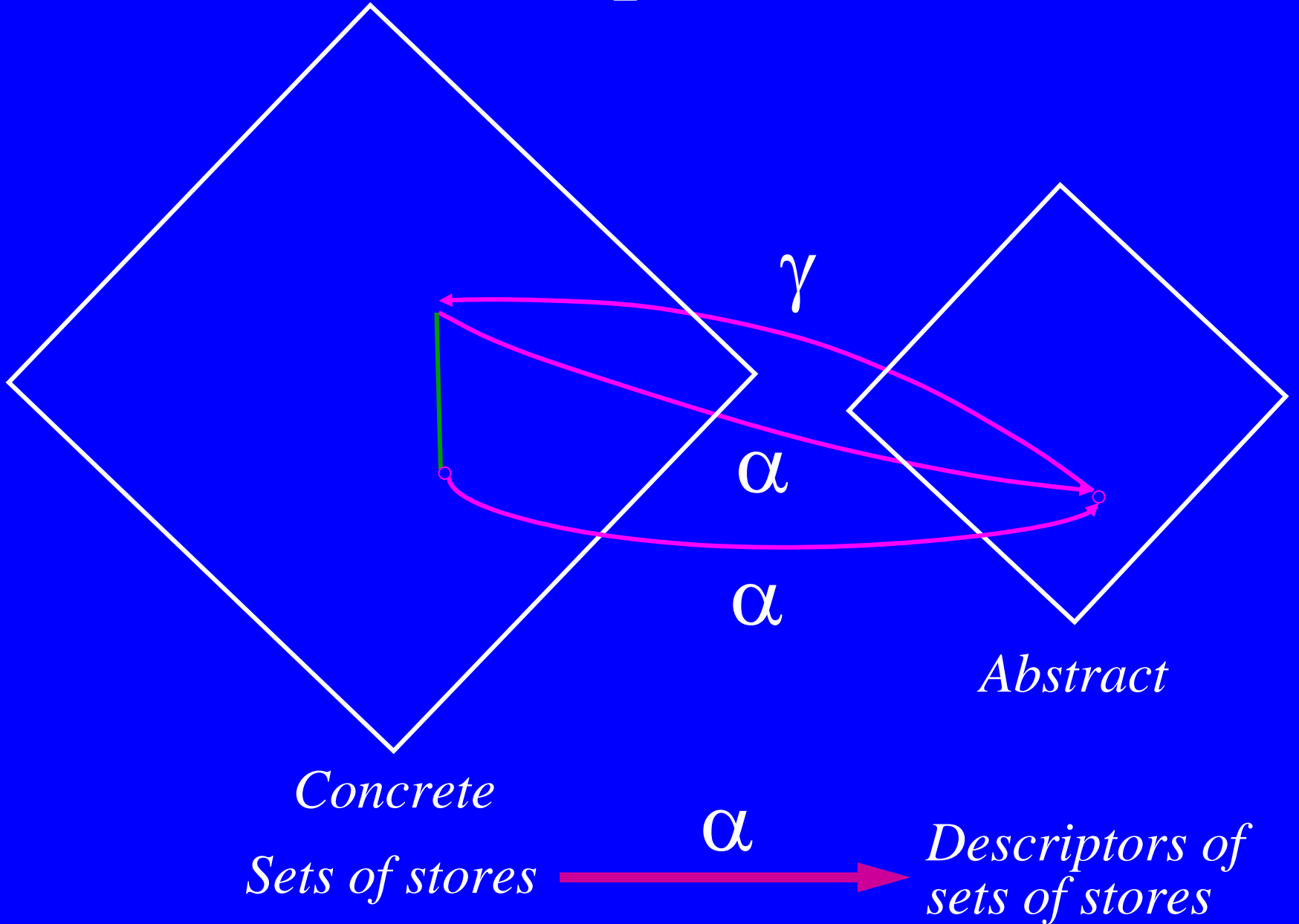
 else

/ x=O */* { x := x * 3 + 1; */* x=E */*
 assert (x %2 ==0); }

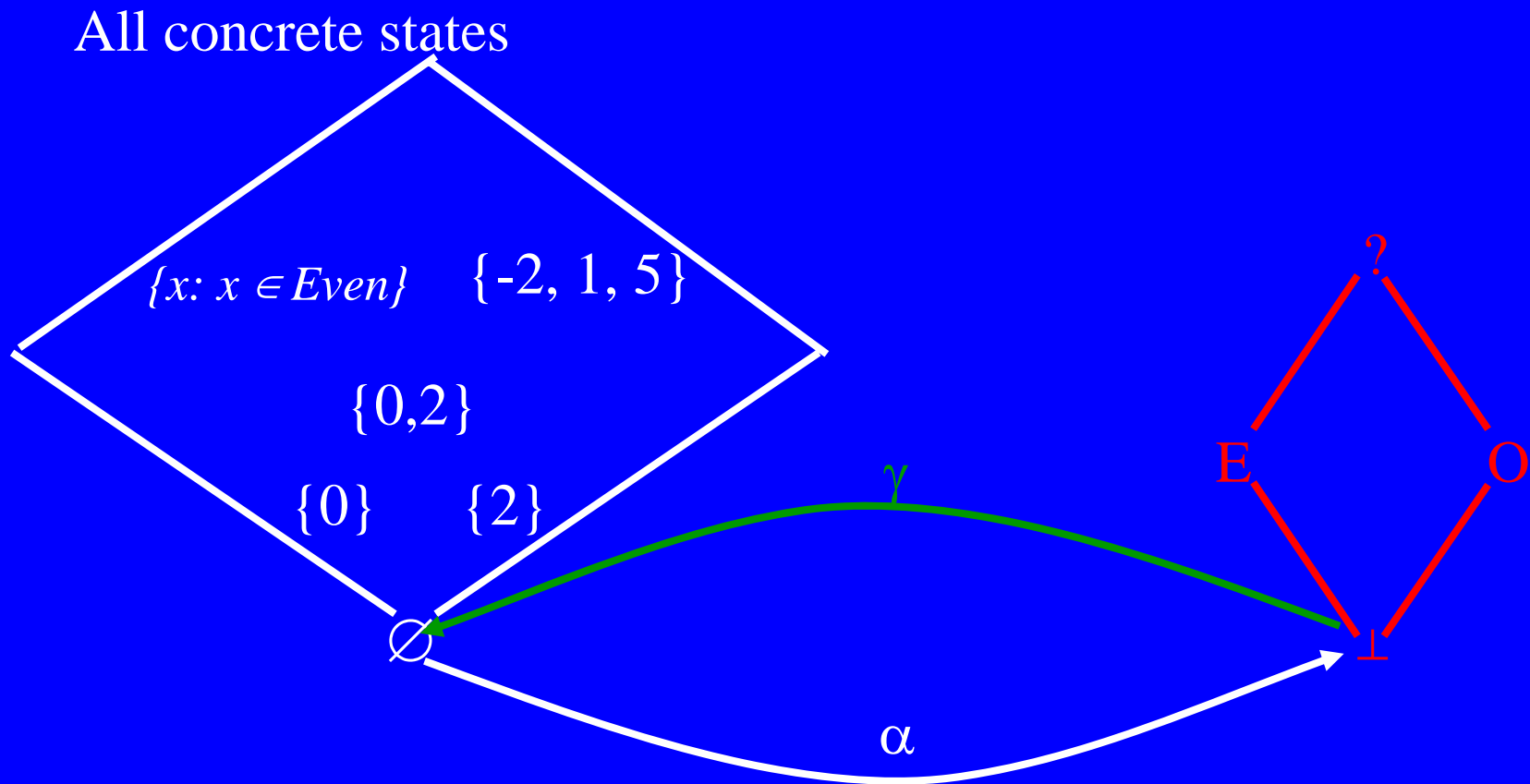
}

/ x=O*/*

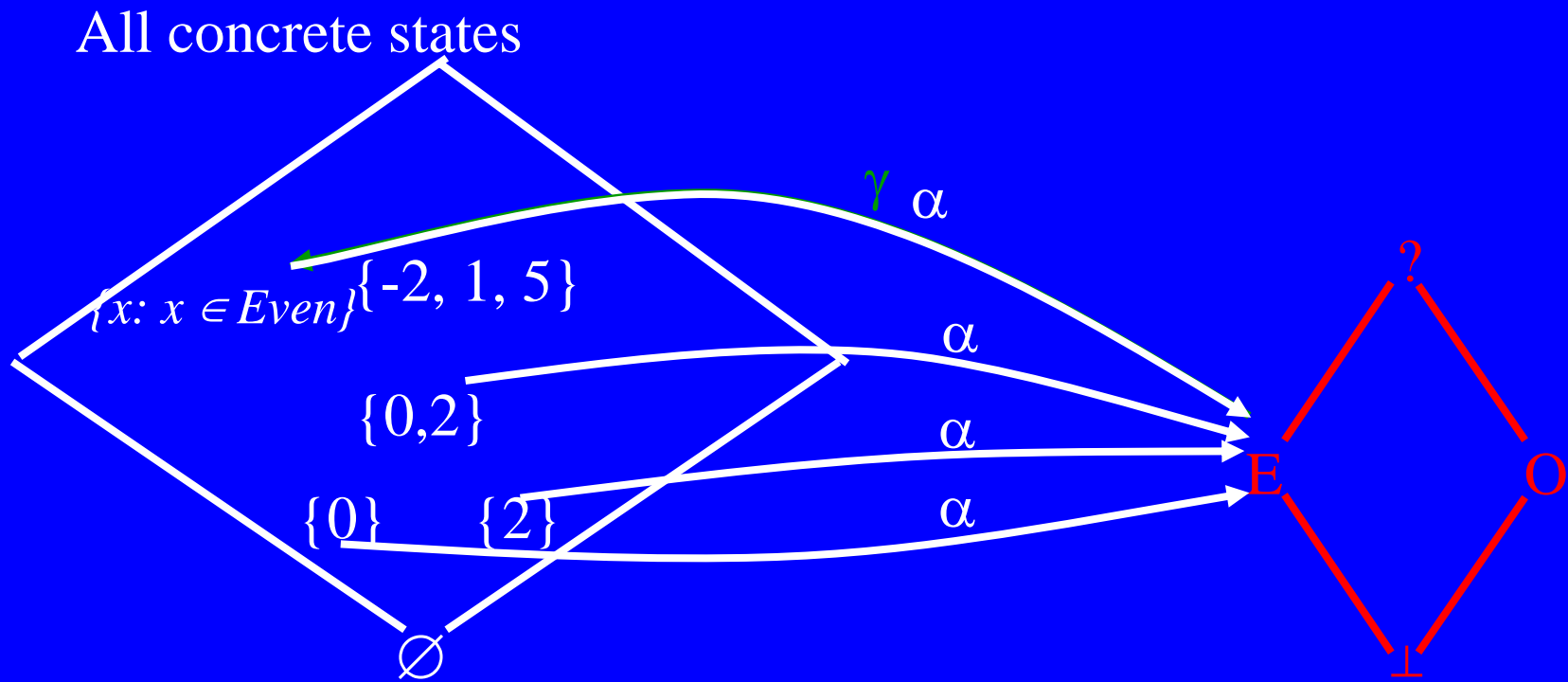
Abstract Interpretation



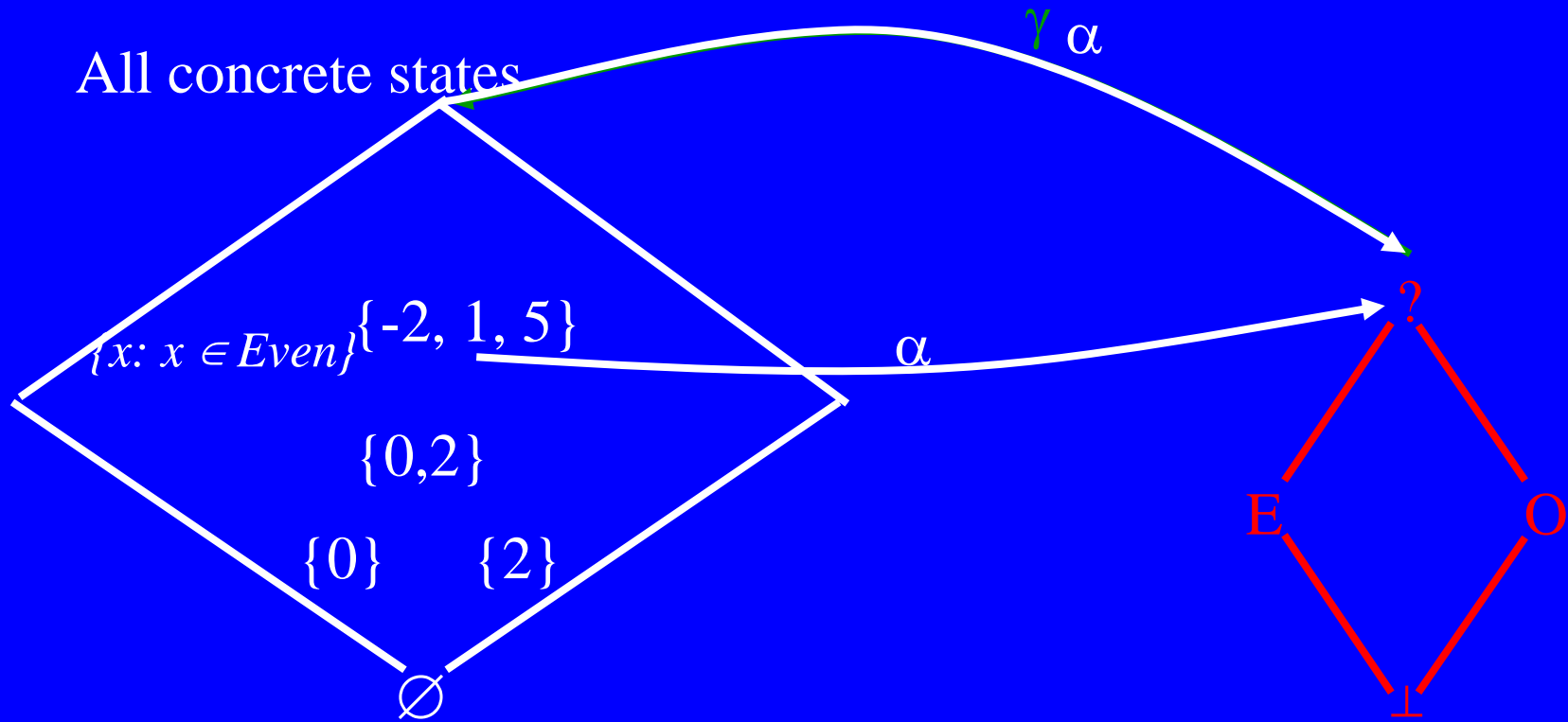
Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation



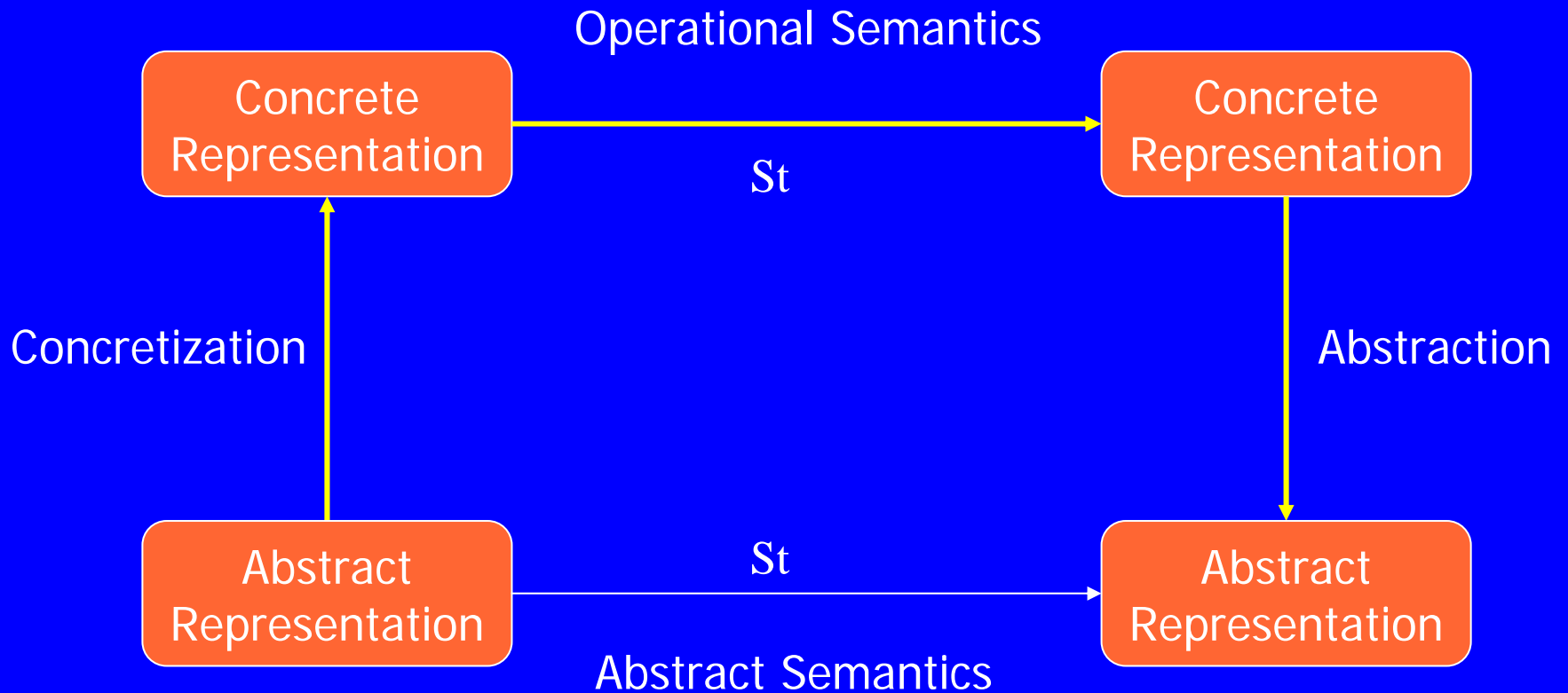
Odd/Even Abstract Interpretation



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



Concrete and Abstract Interpretation

+	0	1	2	3	...
0	0	1	2	3	...
1	1	2	3	4	...
2	2	3	4	5	...
3	3	4	5	6	...
M	M	M	M	M	

*	0	1	2	3	...
0	0	0	0	0	...
1	0	1	2	3	...
2	0	2	4	6	...
3	0	3	6	9	...
M	M	M	M	M	

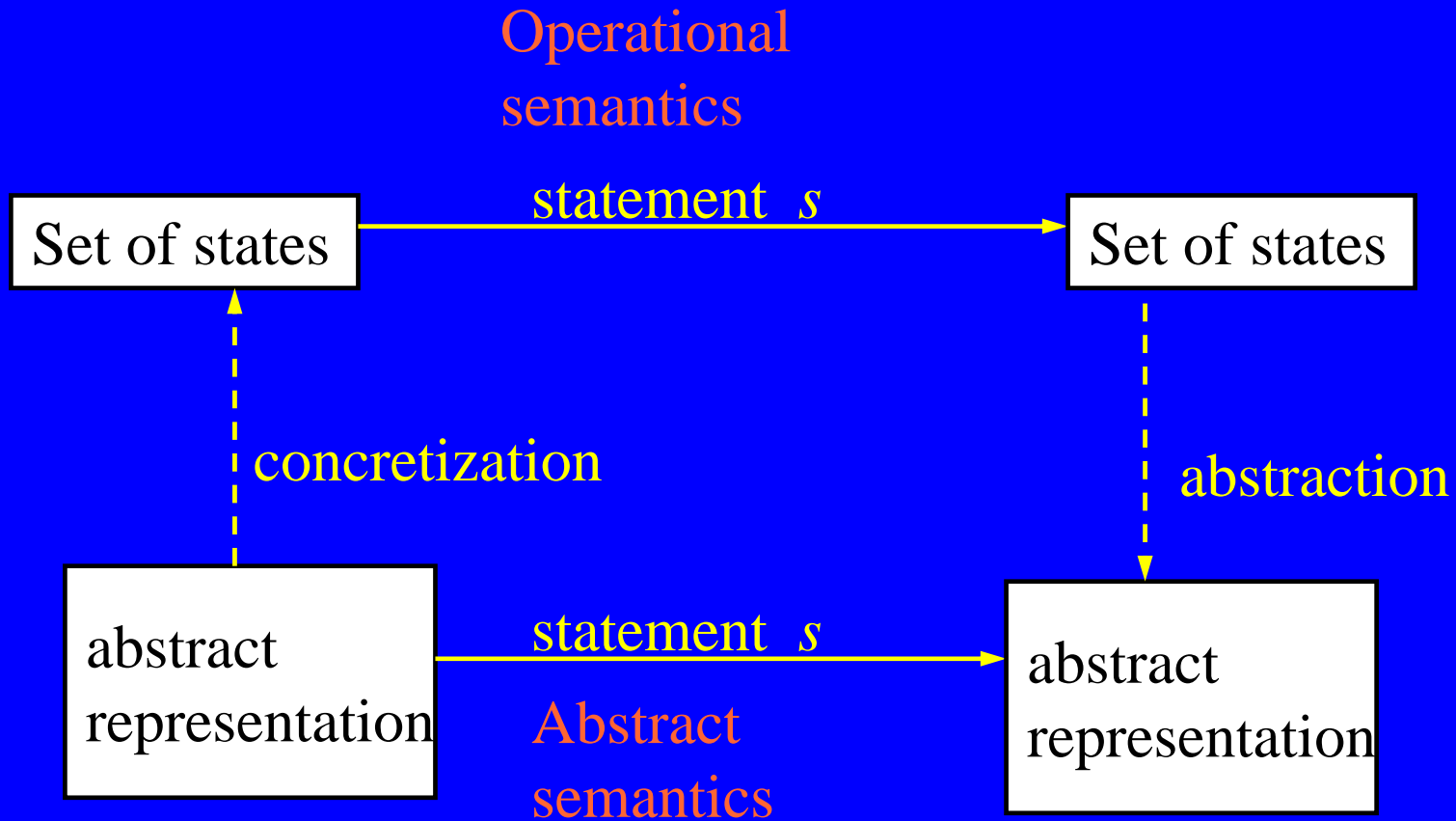
+'	?	O	E
?	?	?	?
O	?	E	O
E	?	O	E

*'	?	O	E
?	?	?	E
O	?	O	E
E	E	E	E

Runtime vs. Static Testing

	Runtime	Abstract
Effectiveness	Missed Errors	False alarms
		Locate rare errors
Cost	Proportional to program's execution	Proportional to program's size

Abstract (Conservative) interpretation

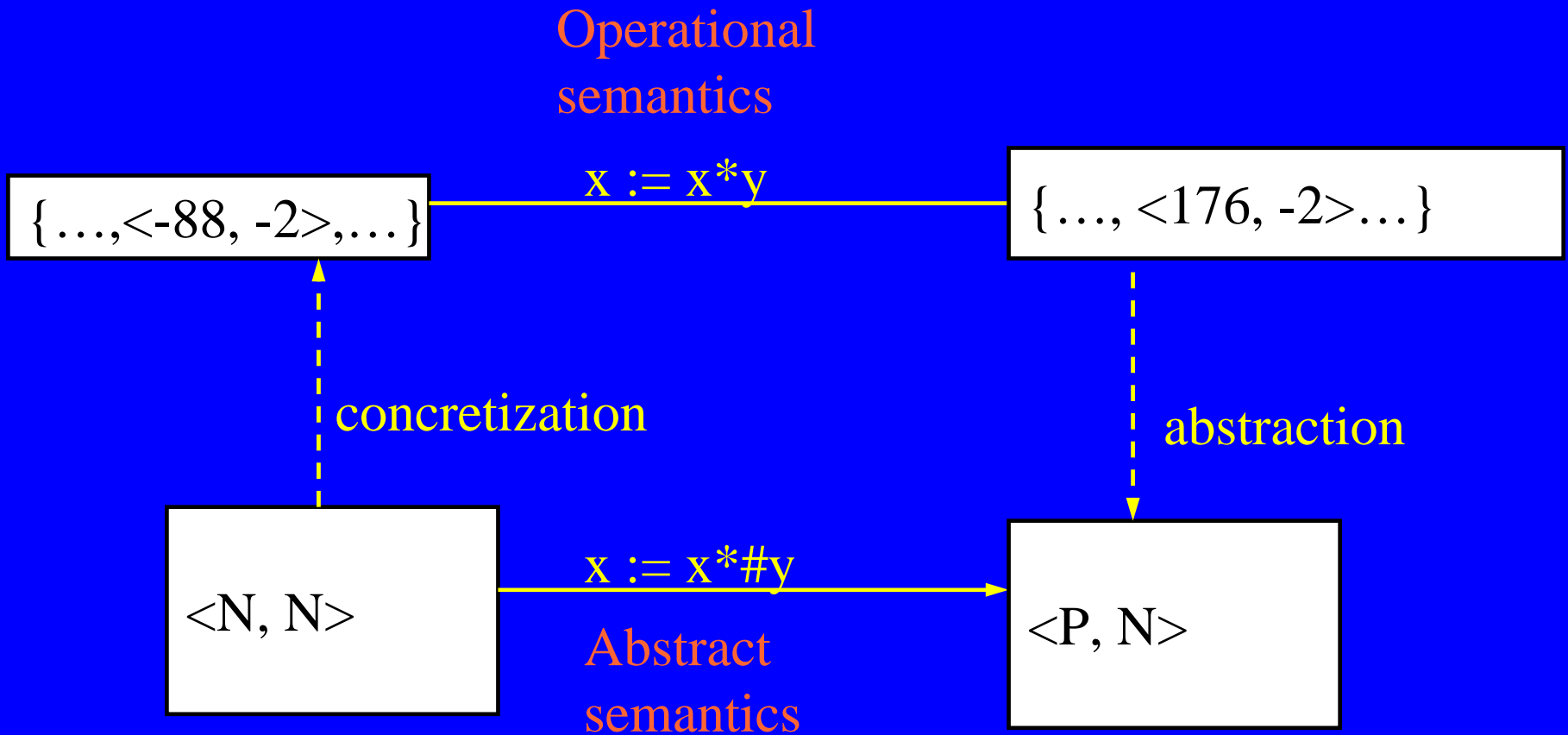


Example rule of signs

- ◆ Safely identify the sign of variables at every program location
- ◆ Abstract representation {P, N, ?}
- ◆ Abstract (conservative) semantics of *

*#	P	N	?
P	P	N	?
N	N	P	?
?	?	?	?

Abstract (conservative) interpretation



Example rule of signs (cont)

- ◆ Safely identify the sign of variables at every program location
- ◆ Abstract representation $\{P, N, ?\}$
- ◆ $\alpha(C) =$ if all elements in C are positive
then return P
else if all elements in C are negative
then return N
else return $?$
- ◆ $\gamma(a) =$ if $(a==P)$ then
return $\{0, 1, 2, \dots\}$
else if $(a==N)$
return $\{-1, -2, -3, \dots\}$
else return Z

Example Constant Propagation

- ◆ Abstract representation set of integer values and an extra value “?” denoting variables not known to be constants
- ◆ Conservative interpretation of +

+#	?	0	1	2
?	?	?	?	?
0	?	0	1	2
1	?	1	2	3
2	?	2	3	4

Example Constant Propagation(Cont)

◆ Conservative interpretation of *

*#	?	0	1	2
?	?	0	?	?
0	0	0	0	0
1	?	0	1	2
2	?	0	2	4

Example Program

```
x = 5;
```

```
y = 7;
```

```
if (getc())
```

```
    y = x + 2;
```

```
z = x + y;
```

Example Program (2)

```
if (getc())  
    x = 3 ; y = 2;  
  
    else  
  
        x = 2; y = 3;  
  
z = x + y;
```

Undecidability Issues

- ◆ It is undecidable if a program point is reachable in some execution
- ◆ Some static analysis problems are undecidable even if the program conditions are ignored

The Constant Propagation Example

```
while (getc()) {  
    if (getc()) x_1 = x_1 + 1;  
    if (getc()) x_2 = x_2 + 1;  
    ...  
    if (getc()) x_n = x_n + 1;  
}  
y = truncate (1/ (1 + p2(x_1, x_2, ..., x_n))  
/* Is y=0 here? */
```

Coping with undecidability

- ◆ Loop free programs
- ◆ Simple static properties
- ◆ Interactive solutions
- ◆ Conservative estimations
 - Every enabled transformation cannot change the meaning of the code but some transformations are not enabled
 - Non optimal code
 - Every potential error is caught but some “false alarms” may be issued

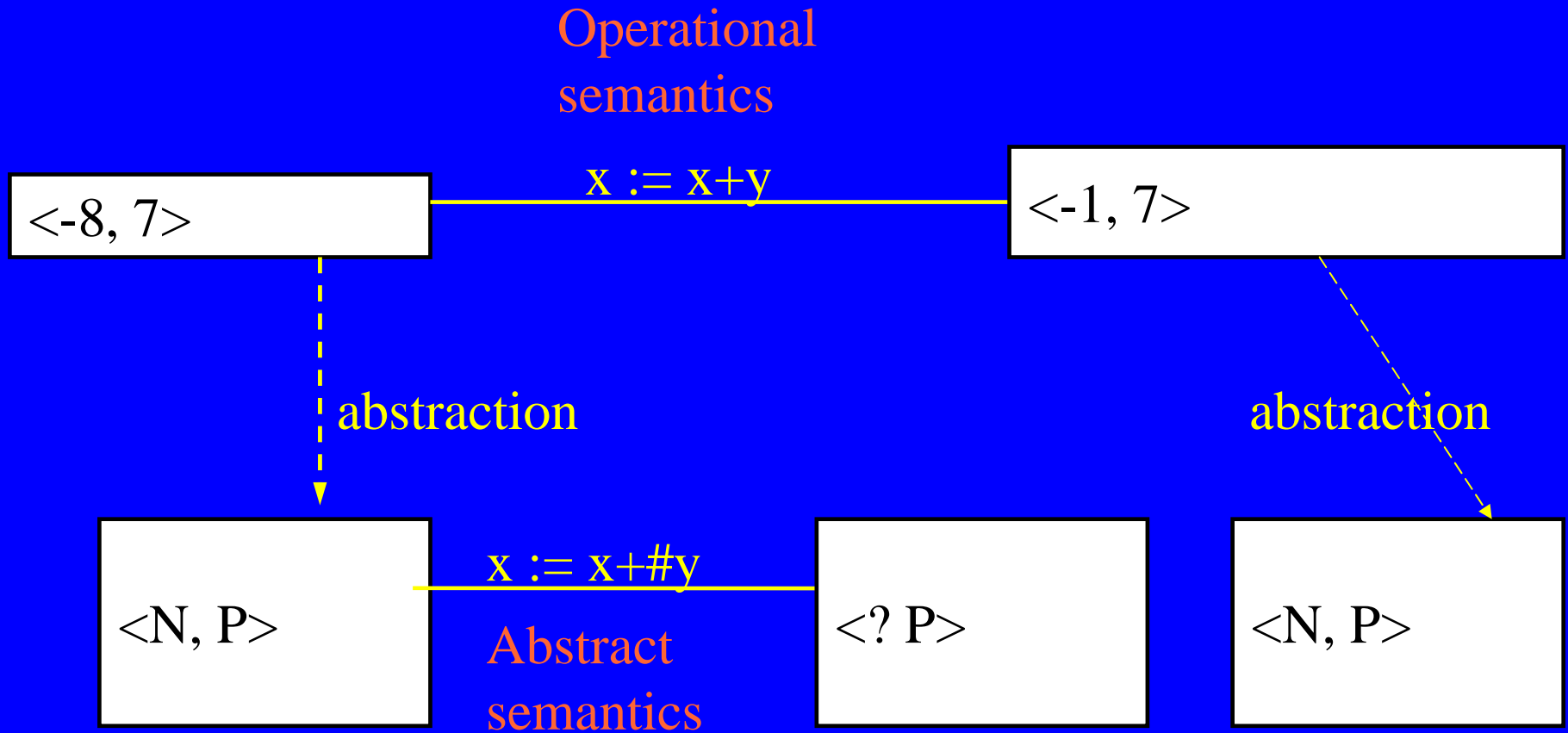
Analogies with Numerical Analysis

- ◆ Approximate the exact semantics
- ◆ More precision can be obtained at greater
- ◆ computational costs

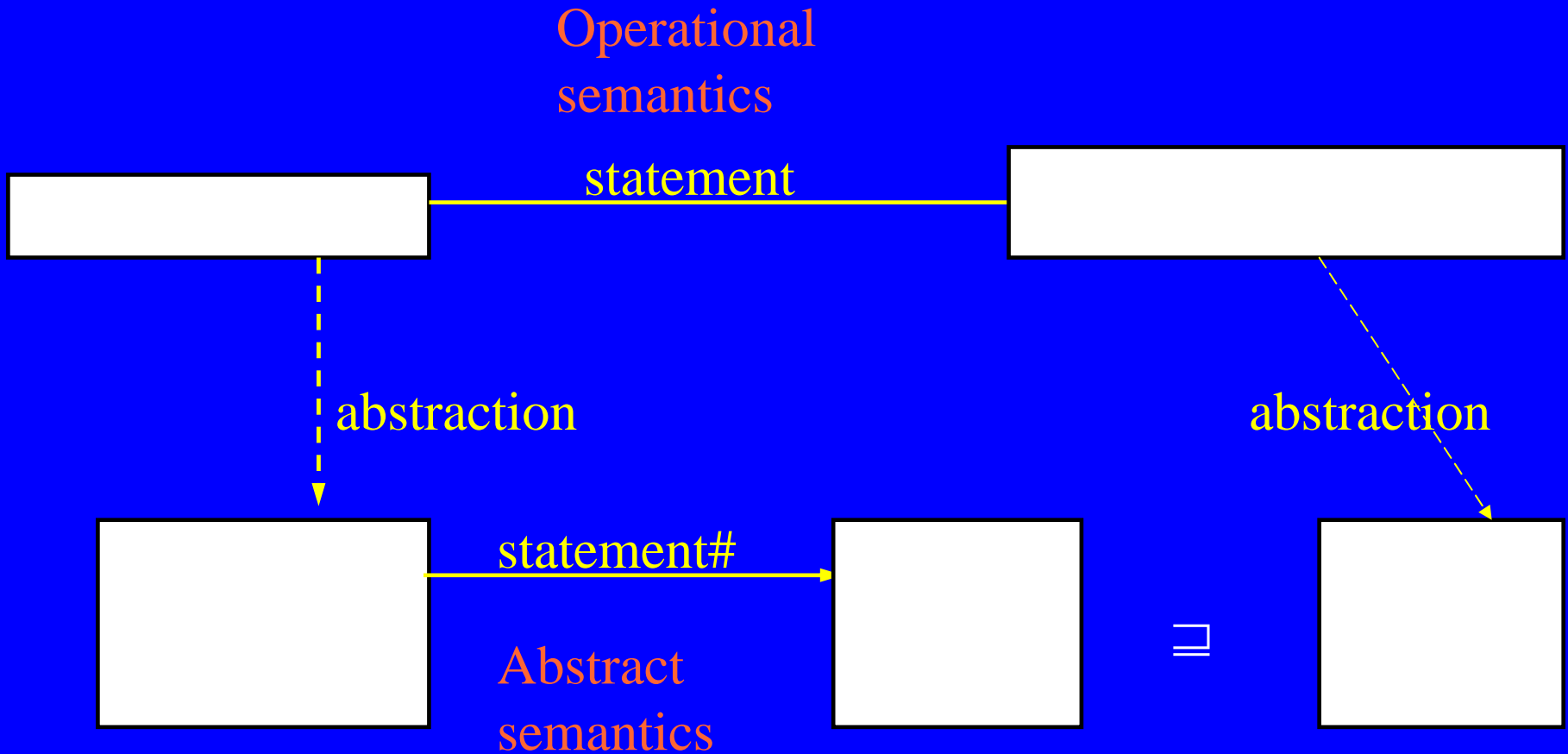
Violation of soundness

- ◆ Loop invariant code motion
- ◆ Dead code elimination
- ◆ Overflow
 $((x+y)+z) \neq (x + (y+z))$
- ◆ Quality checking tools may decide to ignore certain kinds of errors

Abstract interpretation cannot be always homomorphic (rules of signs)



Local Soundness of Abstract Interpretation



Optimality Criteria

- ◆ Precise (with respect to a subset of the programs)
- ◆ Precise under the assumption that all paths are executable (statically exact)
- ◆ Relatively optimal with respect to the chosen abstract domain
- ◆ Good enough

Relation to Program Verification

Program Analysis

- ◆ Fully automatic
- ◆ Applicable to a programming language
- ◆ Can be very imprecise
- ◆ May yield false alarms

Program Verification

- ◆ Requires specification and loop invariants
- ◆ Program specific
- ◆ Relative complete
- ◆ Provide counter examples
- ◆ Provide useful documentation
- ◆ Can be mechanized using theorem provers

Origins of Abstract Interpretation

- ◆ [Naur 1965] The Gier Algol compiler
“A process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not their value”
- ◆ [Reynolds 1969] Interesting analysis which includes infinite domains (context free grammars)
- ◆ [Syntzoff 1972] Well foundedness of programs and termination
- ◆ [Cousot and Cousot 1976,77,79] The general theory
- ◆ [Kamm and Ullman, Kildall 1977] Algorithmic foundations
- ◆ [Tarjan 1981] Reductions to semi-ring problems
- ◆ [Sharir and Pnueli 1981] Foundation of the interprocedural case
- ◆ [Allen, Kennedy, Cock, Jones, Muchnick and Schwartz]

Complementary Approaches

- ◆ Better programming language design
- ◆ Type checking
- ◆ Just in time and dynamic compilation
- ◆ Profiling
- ◆ Sophisticated hardware, e.g., Merced
- ◆ Runtime tests

Tentative Course Schedule

- ◆ Dataflow Algorithms
 - Iterative Dataflow Algorithms
 - Non-Iterative Dataflow Algorithms
 - Interprocedural Dataflow Algorithms
 - Flow insensitive algorithms
- ◆ Foundations of Dataflow Algorithms
 - Trace base program semantics
 - The Theory of Abstract Interpretation
 - » Galois Connections
 - » Widening and Narrowing
 - » Domain Constructors
 - Interesting Instances
 - » Pointer Analysis
 - » Shape Analysis
- ◆ Interesting Program Analyzers
 - SLAM
 - SAFE