cs264: Program Analysis

catalog name: Implementation of Programming Languages

Ras Bodik

WF 11-12:30

slides adapted from Mooly Sagiv

Topics

- static program analysis
  - **principles:** key ideas and connections
  - **techniques:** efficient algorithms
  - **applications:** from compilation to software engineering

- advanced topics, time permitting
  - dynamic program analysis
    - ex.: run-time bug-finding
  - program representations
    - ex.: Static Single Assignment form

Course Requirements

- Prerequisites
  - a compiler course (cs164)
- Useful but not mandatory
  - semantics of programming languages (cs263)
  - algorithms
  - discrete math

Course structure

- Source
  - Nielsen, Nielsen, Hankin, *Principles of Program Analysis*
  - research papers
- Format
  - lectures (roughly following the textbook)
  - discussions of research papers
    - you'll read the paper before lecture and send me a "mini-review"
    - 4 paragraphs
- Grade
  - 4-5 homeworks
  - take-home exam
  - project

Guest lectures

- Lectures may include several “guest lecturers”
  - expert’s view on a more advanced topic
- Guest lecture time usually <u>not</u> during class; instead
  - PS Seminar (Mondays 4pm in 320 Soda)
  - Faculty candidate talks (TBD)
  - CHESS Seminar (Tuesdays, 4pm, 540 Cory)
- First speaker
  - Shaz Qadeer, Monday 4pm 320 Soda
  - paper: *KISS: Keep it Simple and Sequential*
  - you’ll write a mini-review (instructions to come)

Outline

- What is static analysis
  - usage in compilers and other clients
- Why is it called abstract interpretation?
  - handling undecidability
  - soundness of abstract interpretation
- Relation to program verification
- Complementary approaches

Static Analysis

- Goal:
  - automatic derivation of properties that hold on every execution leading to a program location (label)
  - (without knowing program input)

- Usage:
  - compiler optimizations
  - code quality tools
    - Identify bugs
    - Prove absence of certain bugs

Example Static Analysis Problem

- Find variables with constant value at a given program location

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

```
int p(int x){
   return (x * x);
}
void main()
{
   int z;
   if  (getc()) z = p(3) + 1;
   else           z = p(-2) + 6;
   printf (z);
}
```

A more problematic program

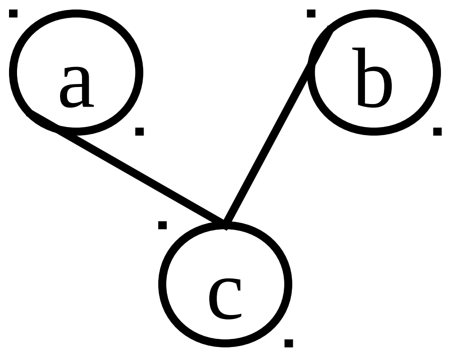
```
int x;
void p(a) {
        c = read();
        if (c > 0) {
                a = a -2;
                p(a);
                a = a + 2;
        }
        x = -2 * a + 5;
        print(x);
}
void main {
        p(7); print(x);
}
```

Another example static analysis problem

- Find variables which are live at a given program location
  - **Definition:** variable *x* is live at a program location *p* if *x*'s R-value can be used before *x* is set
  - **Corresponding property:** there exists an *x*-definition-free execution path from *p* to a use of *x*

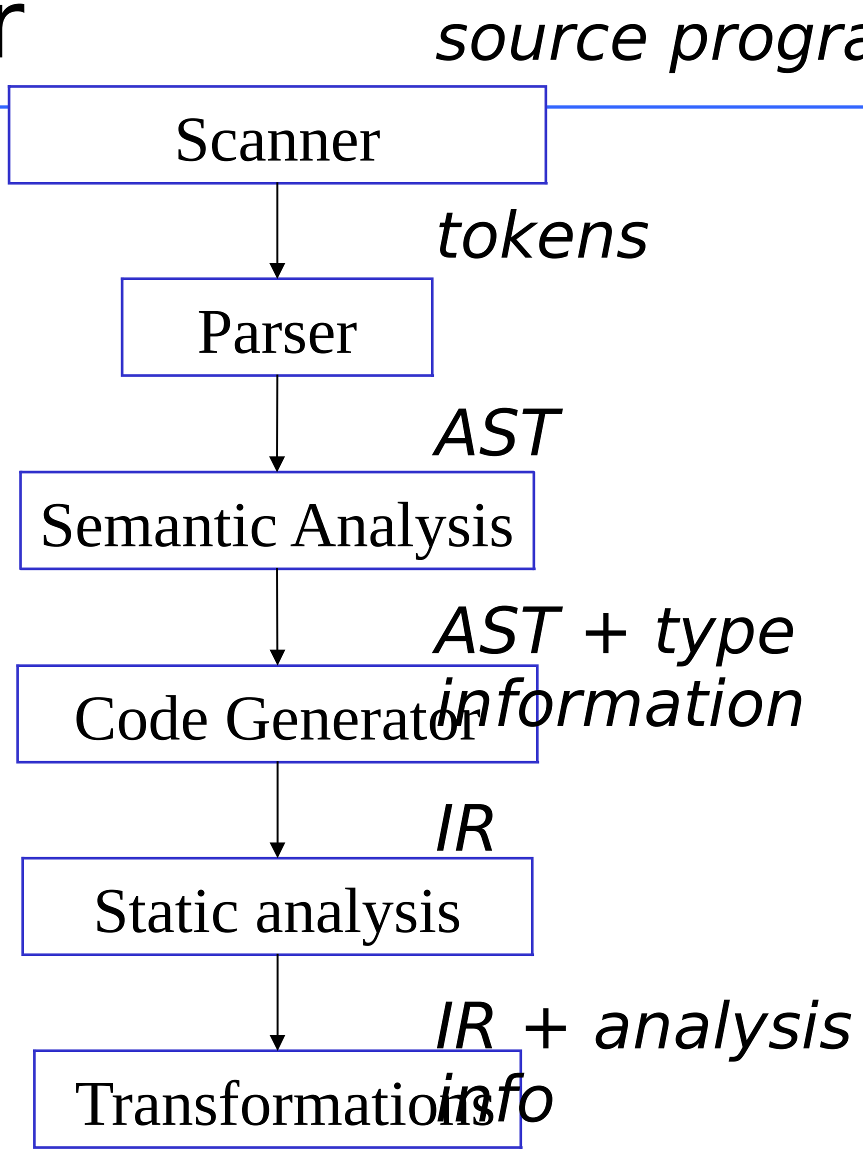
A simple liveness example

```
/* 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
```



register interference graph

Typical compiler



Some Static Analysis Problems

- Live variables
- Reaching definitions
- Available expressions
- Dead code
- Pointer variables that never point to same location
- Points in the program in which it is safe to free an object
- A virtual method call whose target method is unique
- Statements that can be executed in parallel
- An access to a variable that must reside in the 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
    - Strengthen type checking for C
    - Detect Array-bound violations
    - Identify dangling pointers
    - Generate test cases
    - Prove absence of runtime exceptions
    - Prove pre- and post-conditions

Software Quality Tools. Detecting Hazards

- Uninitialized variables:

```
a = malloc() ;
b = a;
cfree (a);
c =  malloc ();
if  (b == c)
   //  unexpected equality
```

- References outside array bounds
- Memory leaks

Memory leakage example

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

```
typedef struct List {
    int d;
    struct List* next;
} List;
```

leakage of address pointed to by head

Challenges in Static Analysis

- Correctness
- Precision
- Efficiency
- Scaling

Foundation of Static Analysis

- Static analysis can be viewed as
  - interpreting the program over an “abstract domain”
  - executing the program over larger set of execution paths
- Guarantee sound results, ex.:
  - Every identified constant is indeed a constant
  - But not every constant is identified as such

Example Abstract Interpretation. Casting Out Nines

- A (weak) sanity check of decimal arithmetic using 9 values
  - 0, 1, 2, 3, 4, 5, 6, 7, 8
- The casting-out-nine rule:
  - whenever an intermediate result exceeds 8, replace by the sum of its digits (recursively)
- Example "123 * 457 + 76543 = 132654?"
  - 123*457 + 76543 = 132654?
  - 6 * 7 + 7 = 21?
  - 6 + 7 = 3?
  - 4 = 3? NO. Report an error.
- Why this rule produces no false alarms:
  - (10a + b) mod 9 = (a + b) mod 9
  - (a+b) mod 9 = (a mod 9) + (b mod 9)
  - (a*b) mod 9 = (a mod 9) * (b mod 9)

Even/Odd Abstract Interpretation

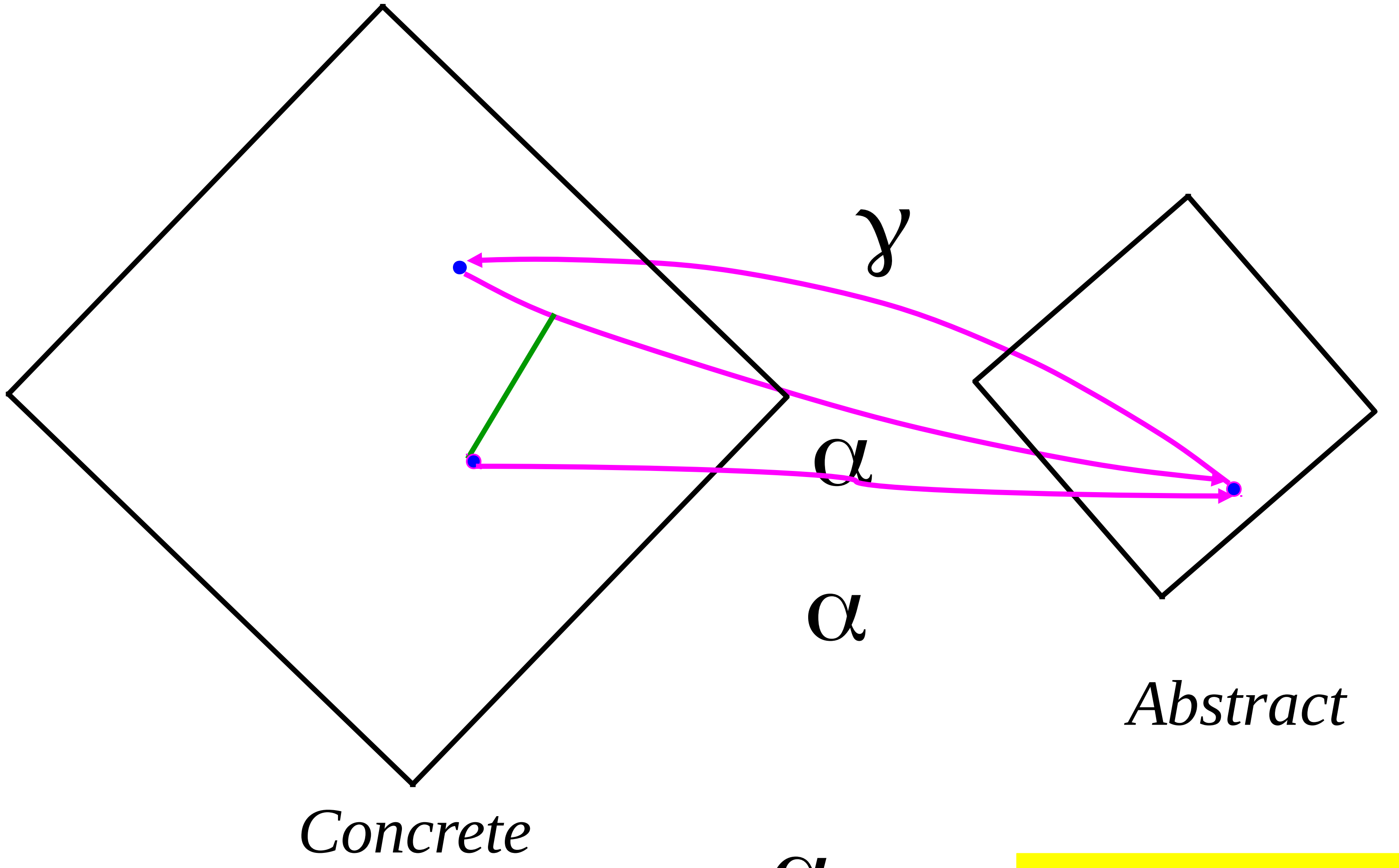
- Determine if an integer variable is even or odd at a given program point

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*/
```

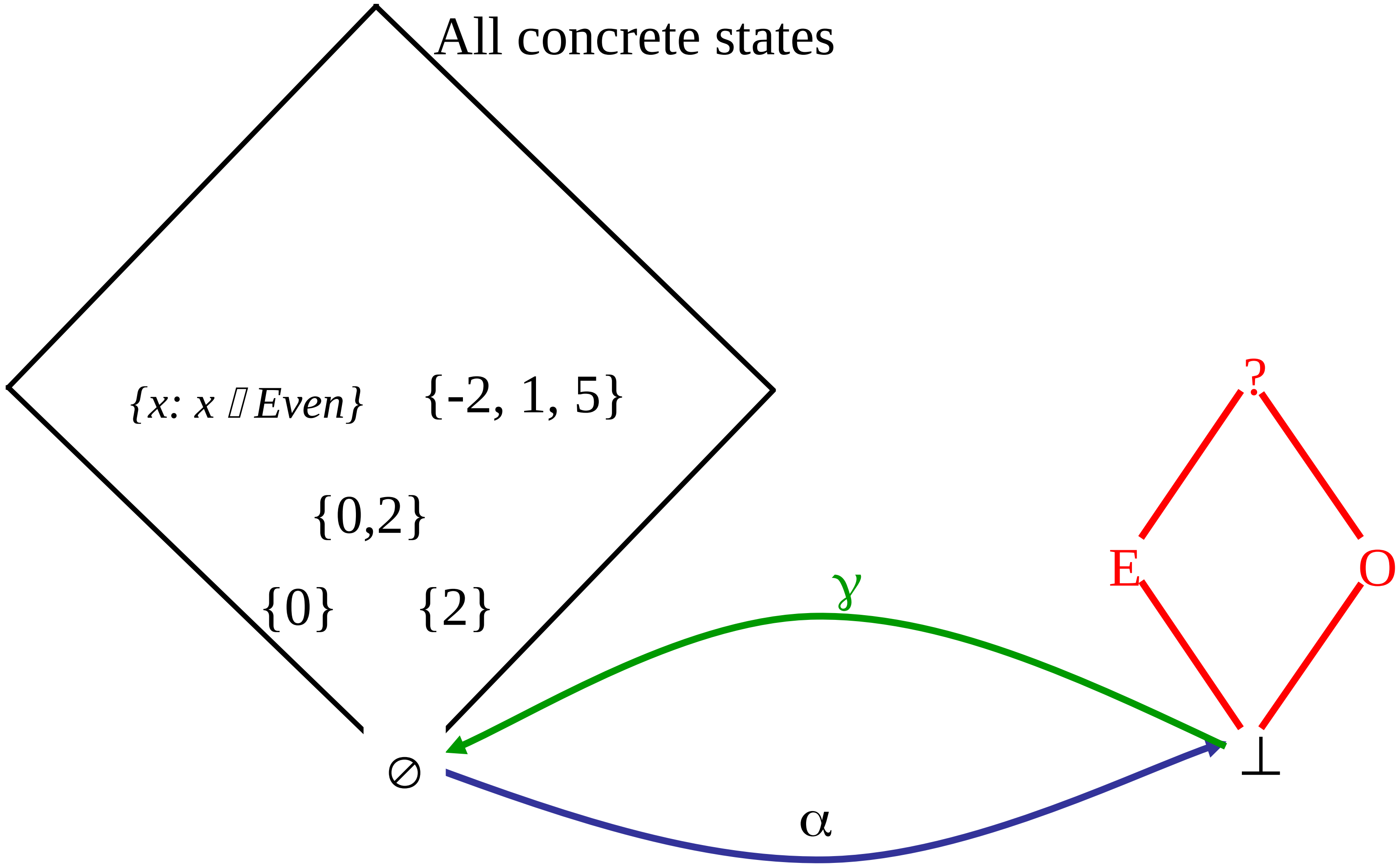
Abstract Interpretation



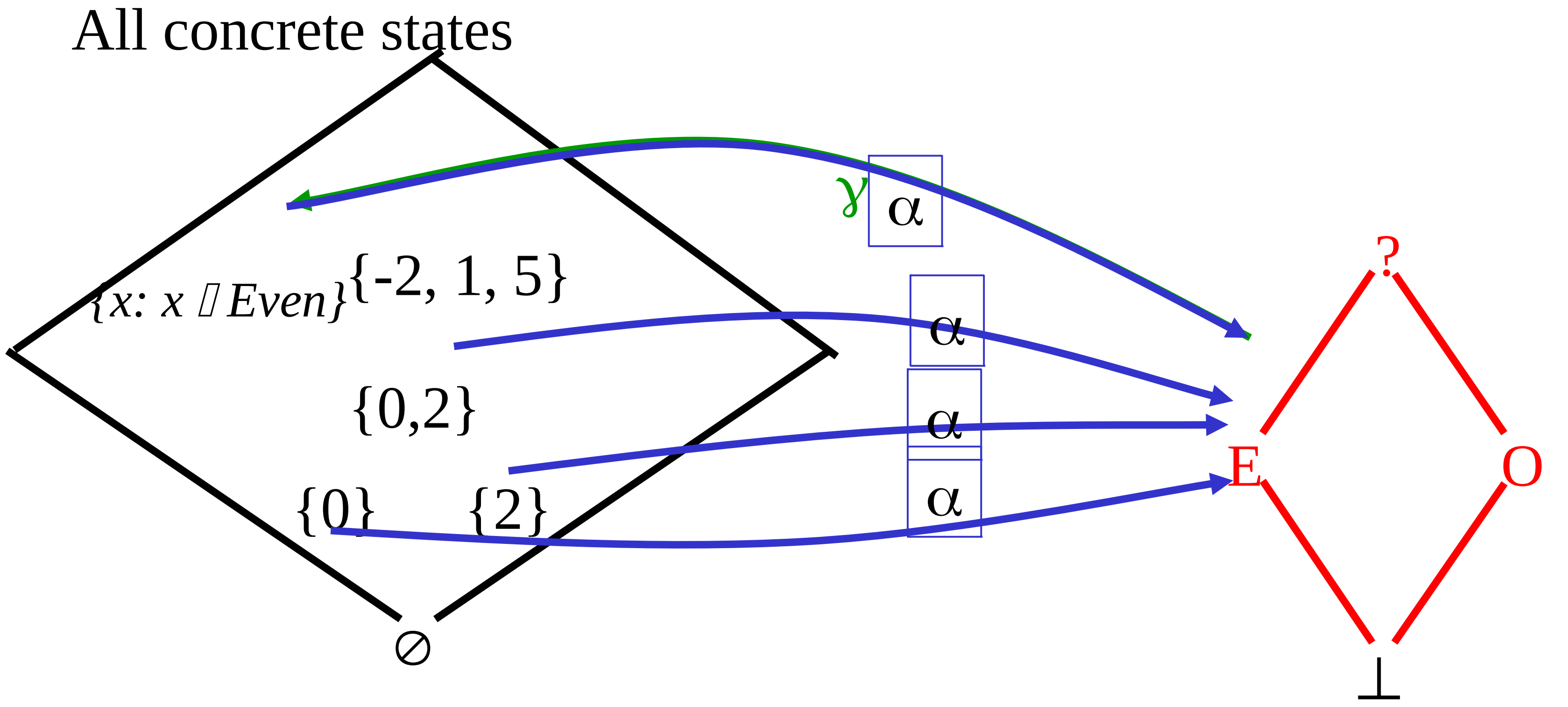
α

Sets of stores → Descriptors of sets of stores

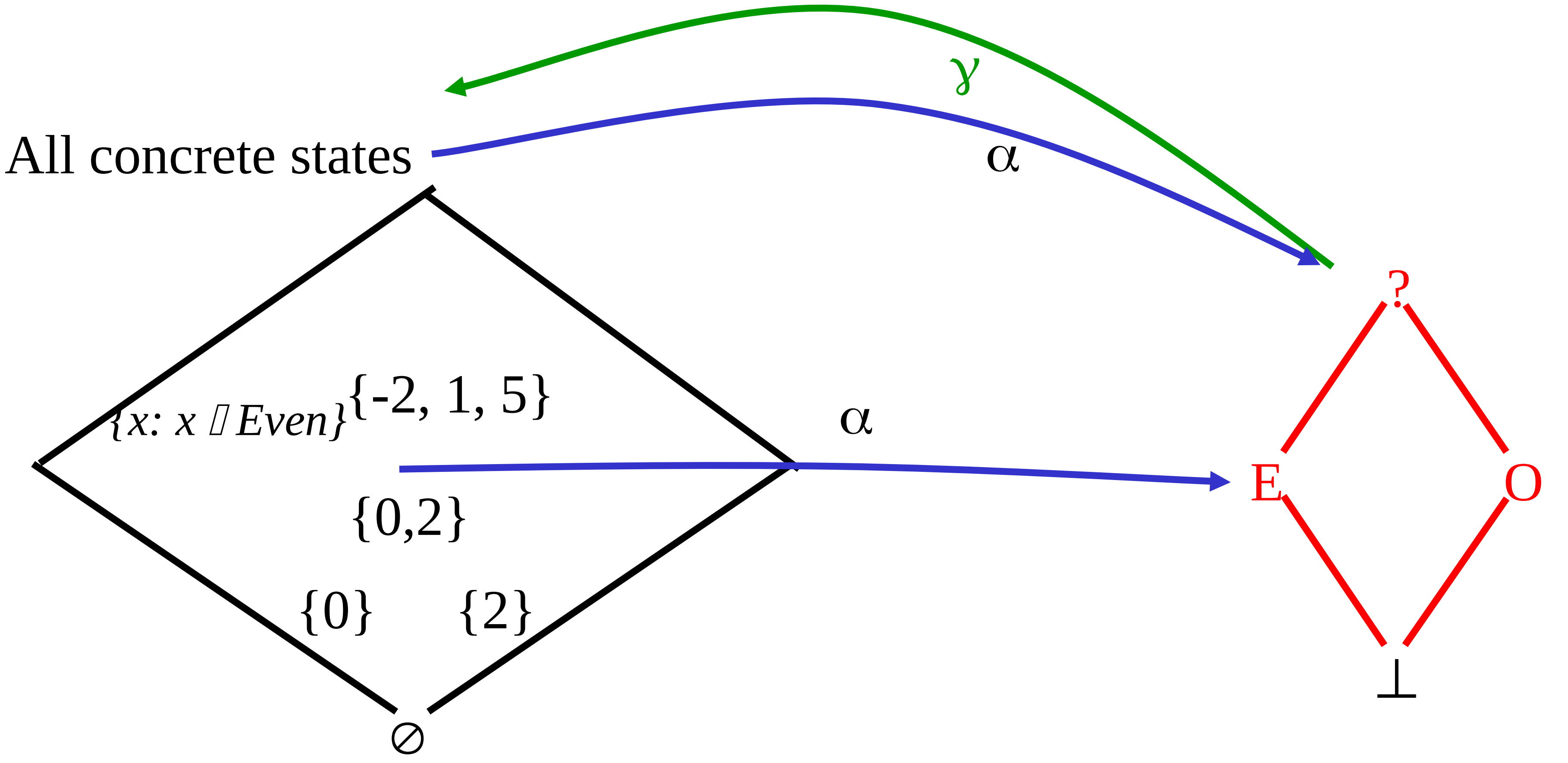
Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation

```
$\alpha(X)$ = if $X = \emptyset$  return $\perp$
        else if  for all z in X (z%2 == 0) return E
        else if  for all z in X (z%2 != 0) return O
        else return ?
```

```
$\gamma(a)$ = if $a = \perp$ return $\emptyset$
        else if a = E  return Even
        else if  a = O  return Odd
        else return Natural
```

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); }
{
```

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	…
⋮	⋮	⋮	⋮	⋮	

*	0	1	2	3	…
0	0	0	0	0	…
1	0	1	2	3	…
2	0	2	4	6	…
3	0	3	6	9	…
⋮	⋮	⋮	⋮	⋮	

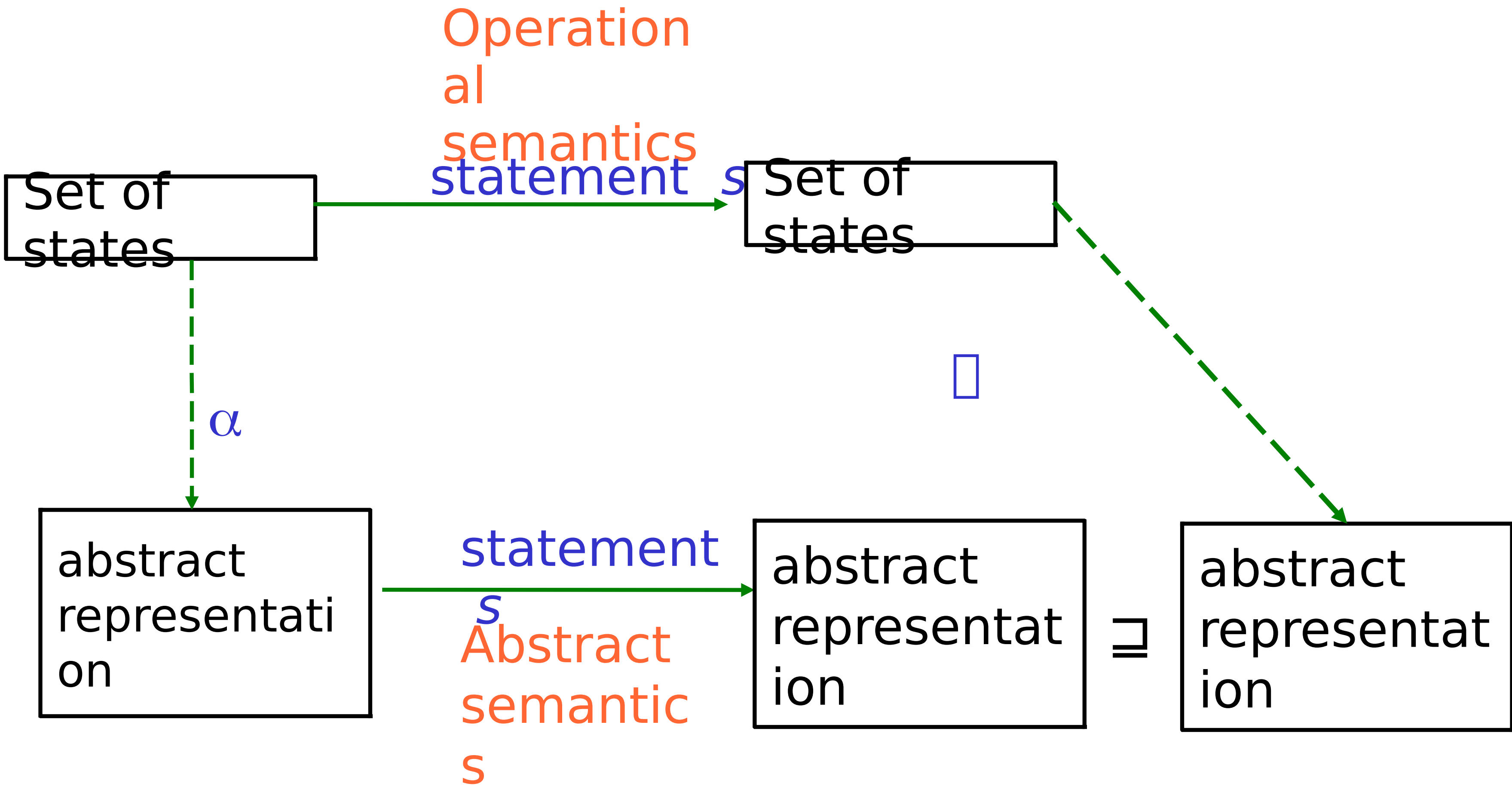
+′	?	O	E
?	?	?	?
O	?	E	O
E	?	O	E

*′	?	O	E
?	?	?	E
O	?	O	E
E	E	E	E

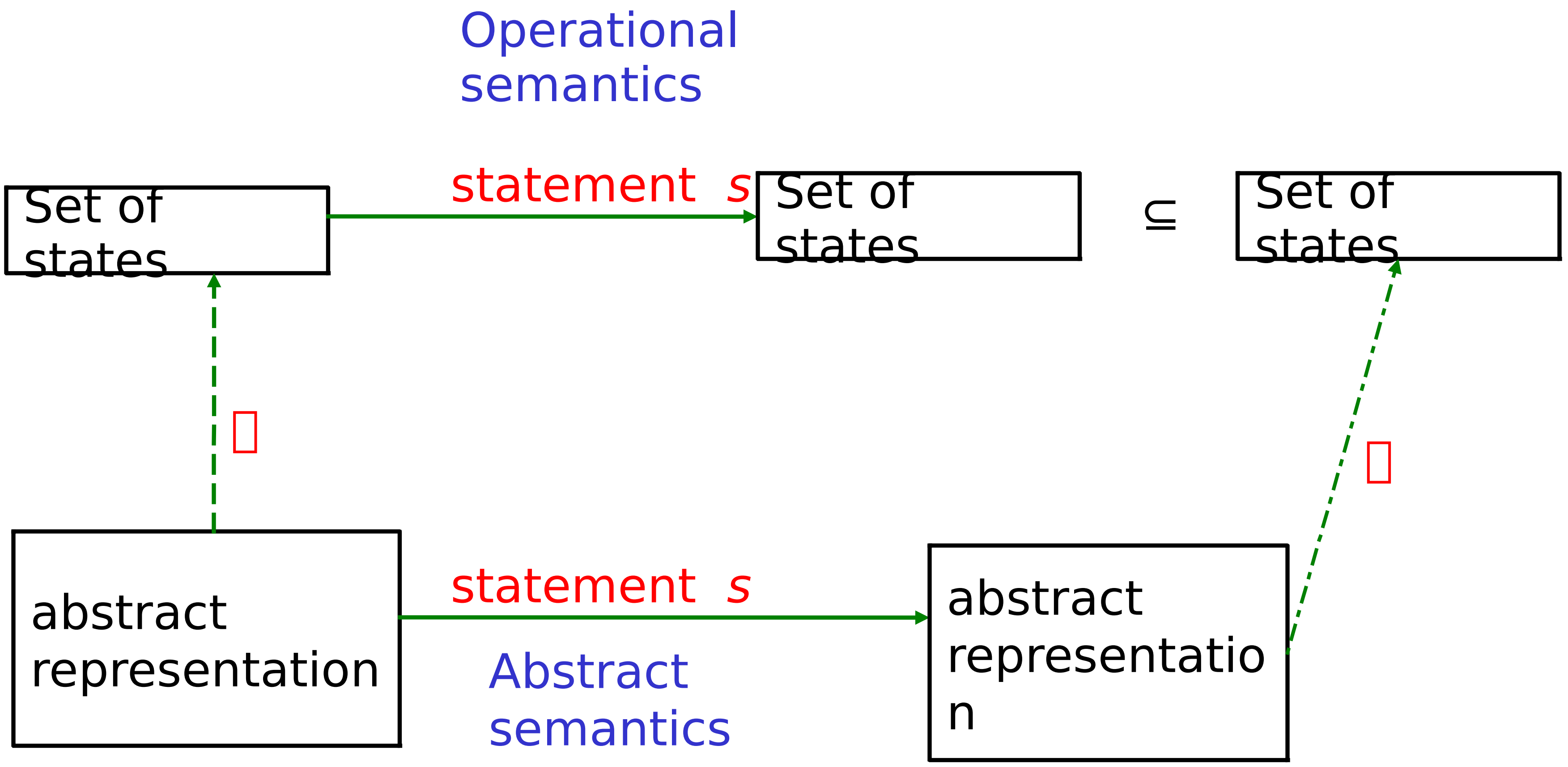
Abstract interpretation cannot be always precise

$$\begin{tikzcd}
\{16, 32\} \arrow[rr, "x := x/2", "\text{Operational semantics}"'] \arrow[d, dashed, "\alpha"] & & \{8, 16\} \arrow[d, dashed, "\alpha"] \\
E \arrow[r, "x := x /^{\#} 2", "\text{Abstract semantics}"'] & ? \arrow[r, phantom, "\sqsupseteq"] & E
\end{tikzcd}$$

Abstract (Conservative) interpretation



Abstract (Conservative) interpretation



Challenges in Abstract Interpretation

- Finding appropriate program semantics (runtime)
- Designing abstract representations
  - What to forget
  - What to remember
    - Summarize crucial information
  - Handling loops
  - Handling procedures
- Scalability
  - Large programs
  - Missing source code
- Precise enough

Runtime vs. Abstract Interpretation (Software Quality Tools)

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

Example Constant Propagation

- Abstract representation set of integer values and and 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
…	?	0	…	…

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 undecidabilty

- Loop free programs
- Simple static properties
- Interactive solutions
- Effects of 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
  - But sometimes more precise can also be more efficient

Violation of soundness

- Loop invariant code motion
- Dead code elimination
- Overflow
  ((x+y)+z) != (x + (y+z))
- Quality checking tools may decide to ignore certain kinds of errors
  - Sound w.r.t different concrete semantics

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

Program Verification

- Mathematically prove the correctness of the program
- Requires formal specification

- Example. Hoare Logic {P} S {Q}
  - {x = 1} x++ ; {x = 2}
  - {x =1}
    {true} if (y >0) x = 1 else x = 2 {?}
  - {y=n} z = 1 while (y>0)   {z = z * y-- ; } {?}

Relation to Program Verification

Program Analysis

- Fully automatic
- But can benefit from specification
- Applicable to a programming language
- Can be very imprecise
- May yield false alarms
- Identify interesting bugs
- Establish non-trivial properties using effective algorithms

Program Verification

- Requires specification and loop invariants
- Not decidable
- Program specific

- Relative complete
- Must provide counter examples
- Provide useful documentation

Complementary Approaches

- Finite state model checking
- Unsound approaches
  - Compute underapproximation
- Better programming language design
- Type checking
- Proof carrying code
- Just in time and dynamic compilation
- Profiling
- Runtime tests