Program Analysis

Mooly Sagiv

http://www.cs.tau.ac.il/~msagiv/courses/pa16.html

Formalities

- Prerequisites: Compilers or Programming Languages
- Course Grade
  - 10 % Lecture Summary (latex+examples within one week)
  - 45% 4 assignments
  - 45% Final Course Project (Ivy)

Motivation

- Compiler optimizations
  - Common subexpressions
  - Parallelization
- Software engineering
- Security

Class Notes

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

- A sailor on the U.S.S. Yorktown entered a 0 into a data field in a kitchen-inventory program
- The 0-input caused an overflow, which crashed all LAN consoles and miniature remote terminal units
- The Yorktown was dead in the water for about two hours and 45 minutes

Numeric static analysis can detect these errors when the ship is built!

- A sailor on the U.S.S. Yorktown entered a 0 into a data field in a kitchen-inventory program
- The 0-input caused an overflow, which crashed all LAN consoles and miniature remote terminal units
- The Yorktown was dead in the water for about two hours and 45 minutes

```
x = 3;
y = 1/(x-3);
```

need to track values other than 0

```
x = 3;
px = &x;
y = 1/(*px-3);
```

need to track pointers

```
for (x =5; x < y ; x++) {
    y = 1/ z - x
```

Need to reason about loops

Dynamic Allocation (Heap)

```
x = 3;
p = (int*)malloc(sizeof int);
*p = x;
q = p;
y = 1/(*q-3);
```

need to track heap-allocated storage

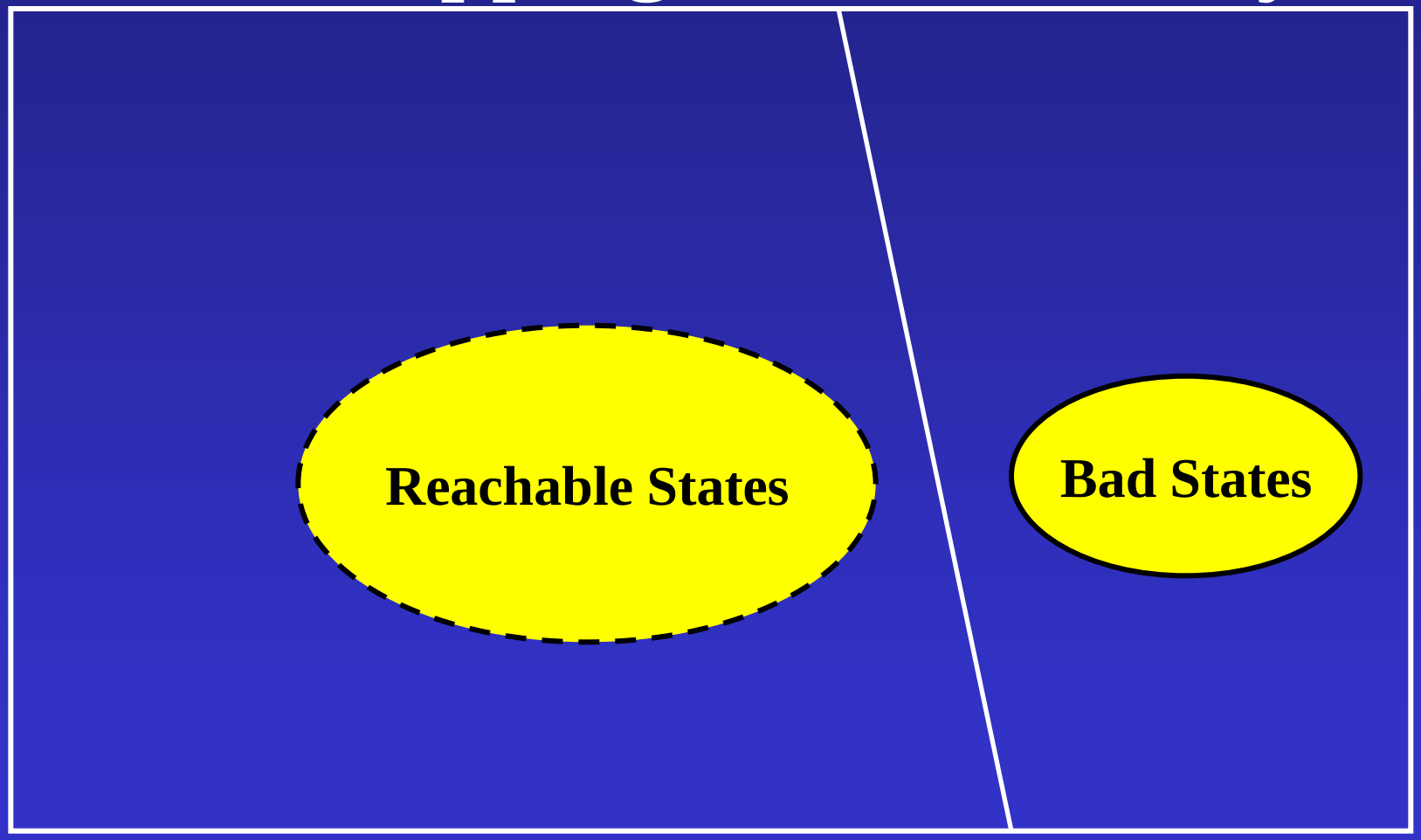
Why is Program Analysis Difficult?

- Undecidability
  - Checking if program point is reachable
    - The Halting Problem
  - Checking interesting program properties
    - Rice Theorem
  - Can the computer really perform inductive reasoning?

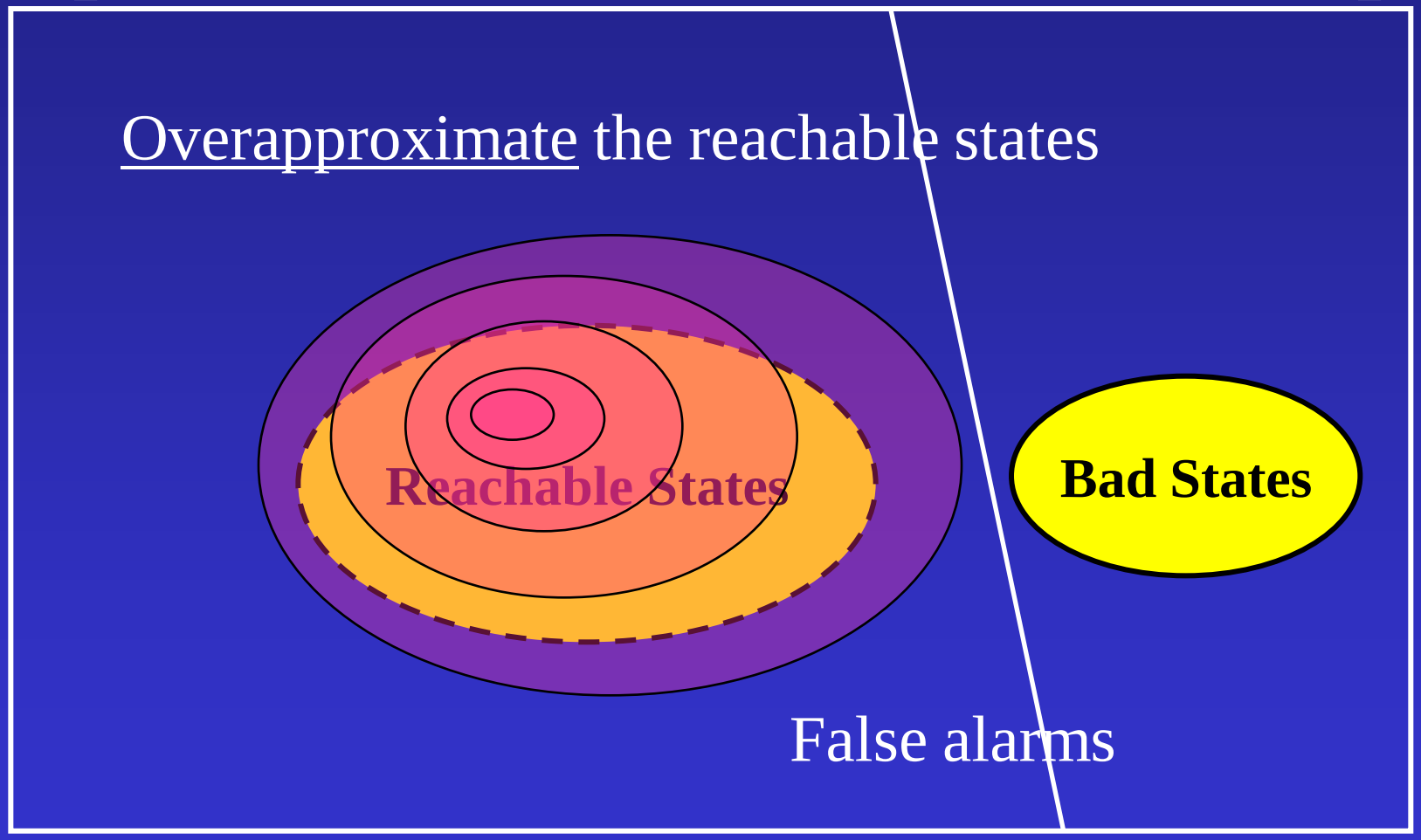
Why is Program Analysis Difficult?

- Complicated programming languages
  - Large/unbounded base types: **`int`**, **`float`**, **`string`**
  - Pointers/aliasing + unbounded #'s of heap-allocated cells
  - User-defined types/classes
  - Loops with unbounded number of iterations
  - Procedure calls/recursion/calls through pointers/dynamic method lookup/overloading
  - Concurrency + unbounded #'s of threads
- Conceptual
  - Which program to analyze?
  - Which properties to check?
- Scalability

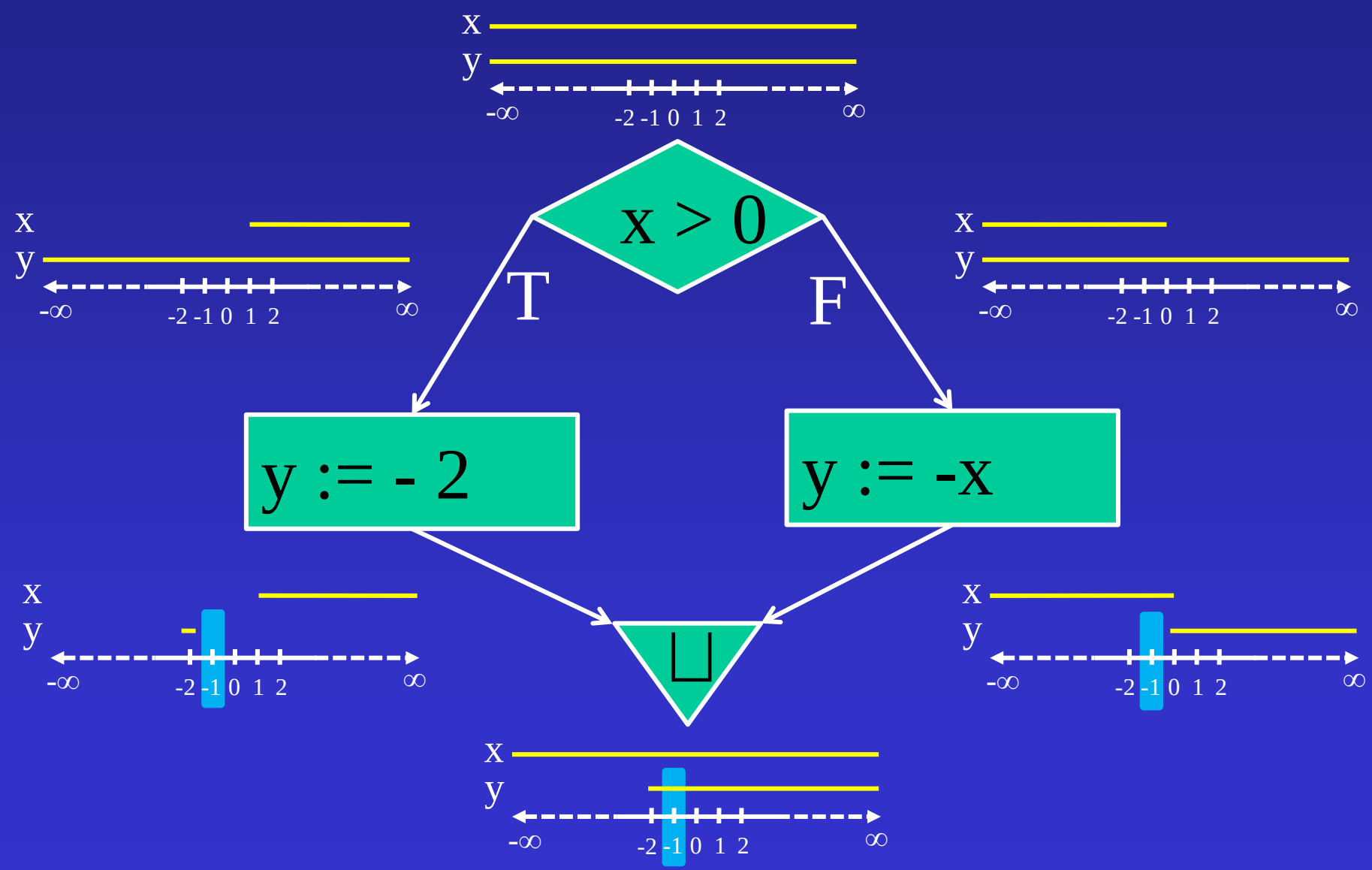
Sidestepping Undecidability



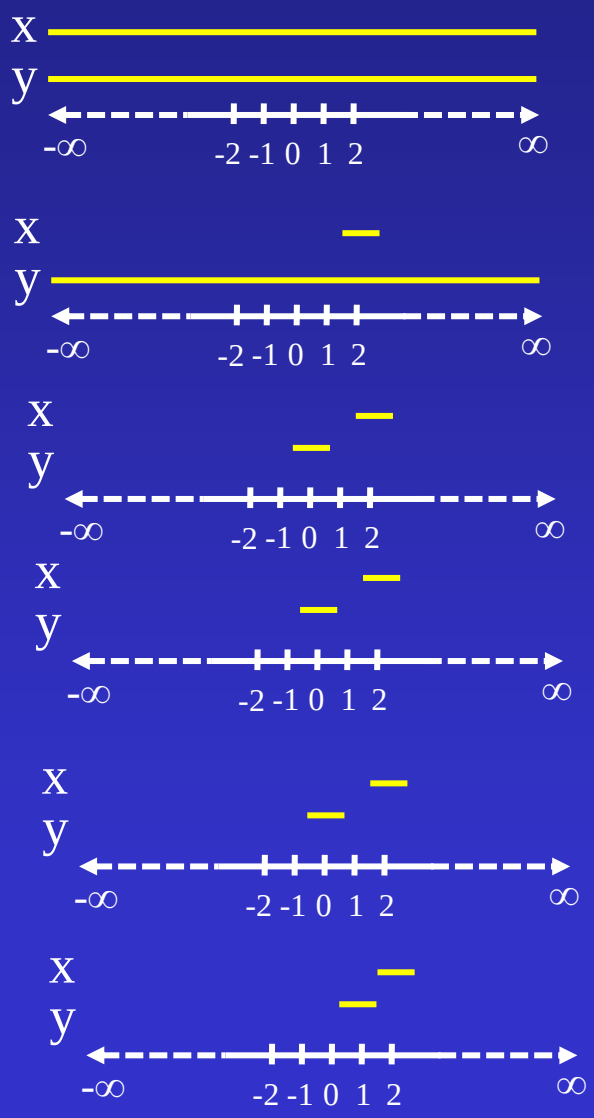
Sidestepping Undecidability [Cousot & Cousot POPL77-79]

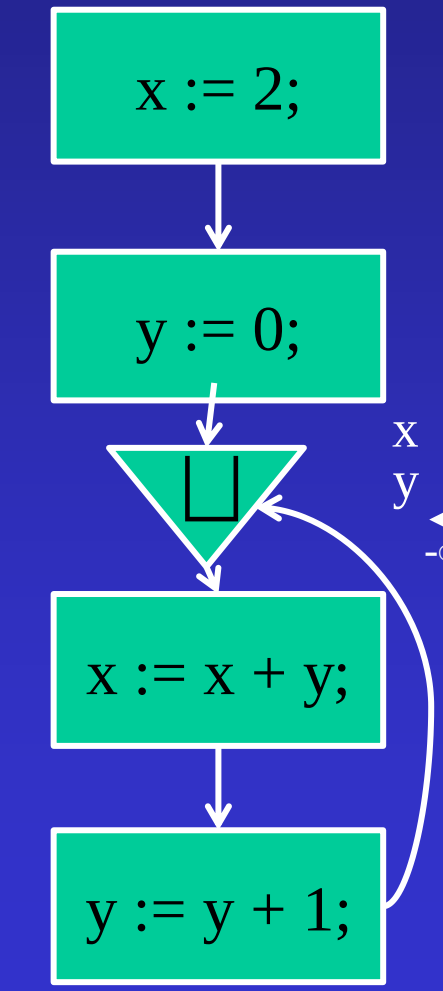


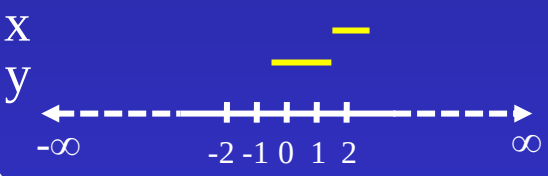
Abstract Interpretation



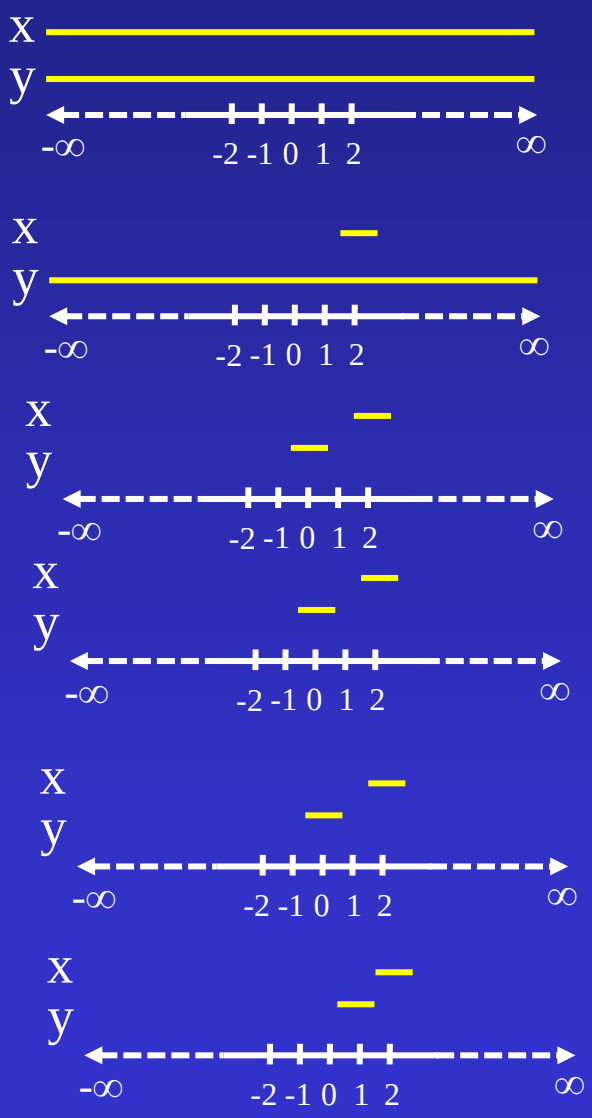
Infer Inductive Invariants via AI

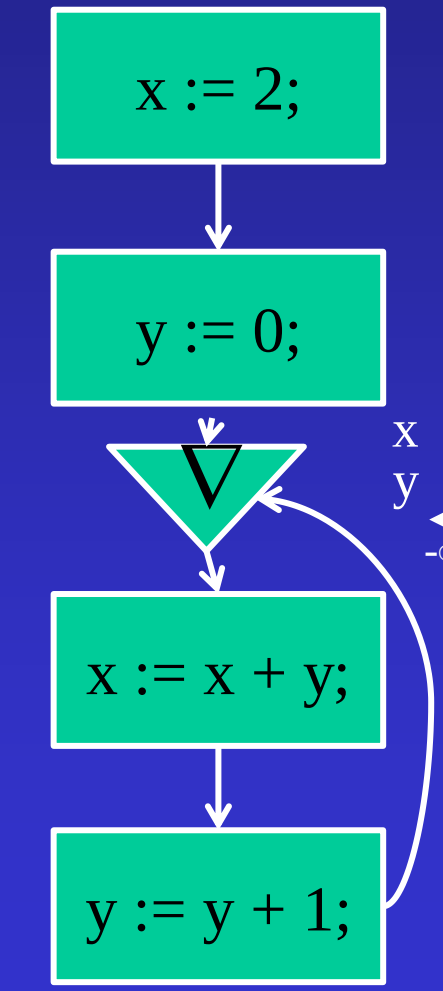


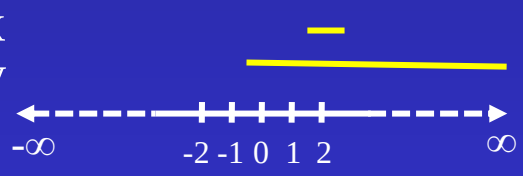




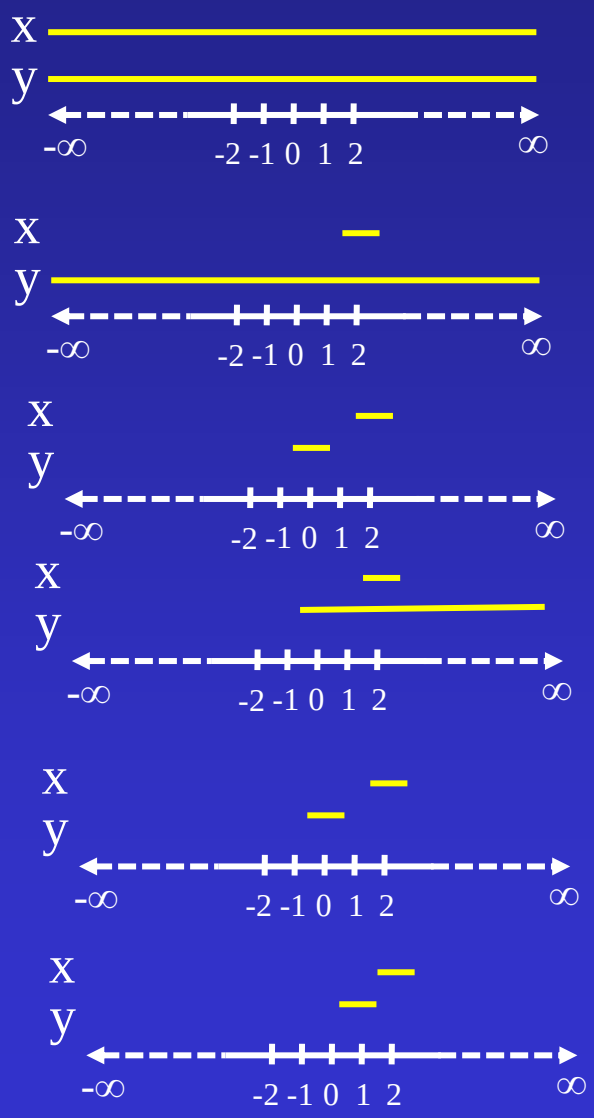
Infer Inductive Invariants via AI

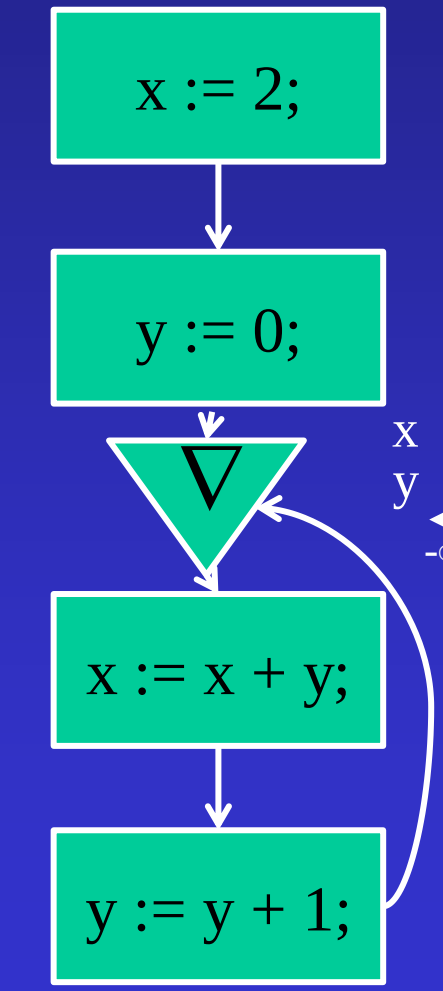


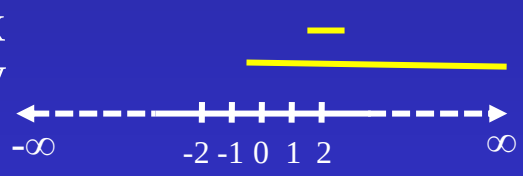




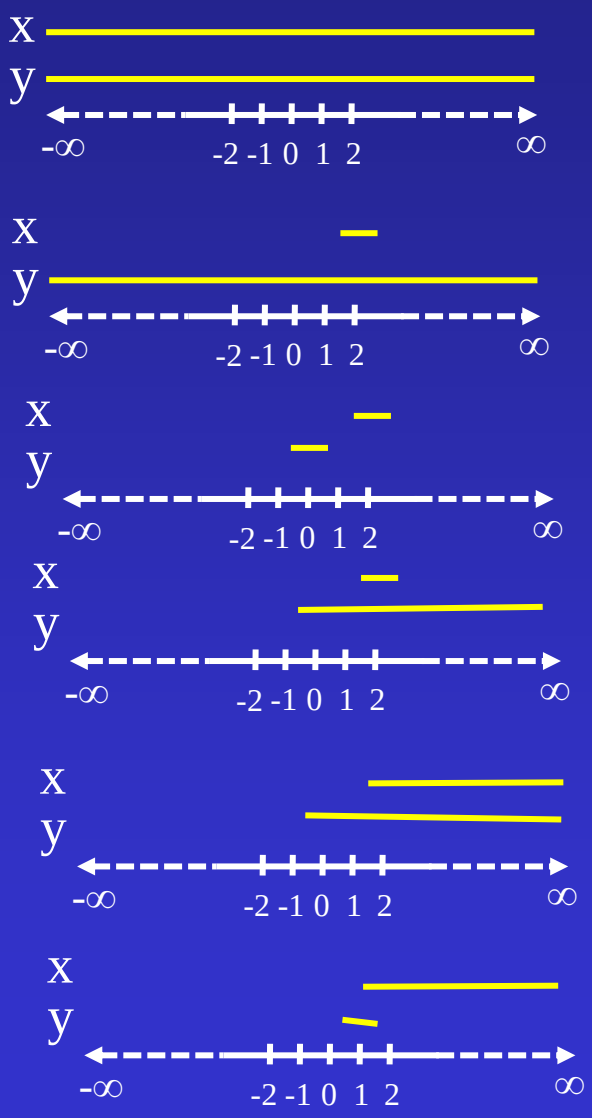
Infer Inductive Invariants via AI

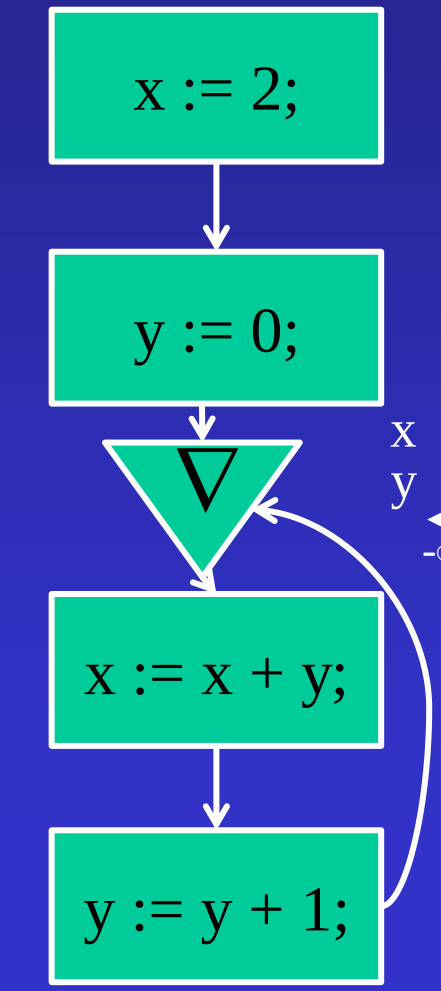


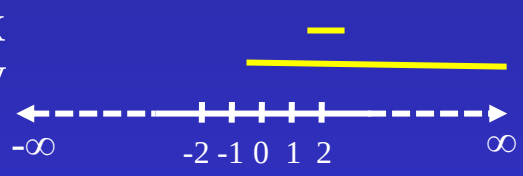




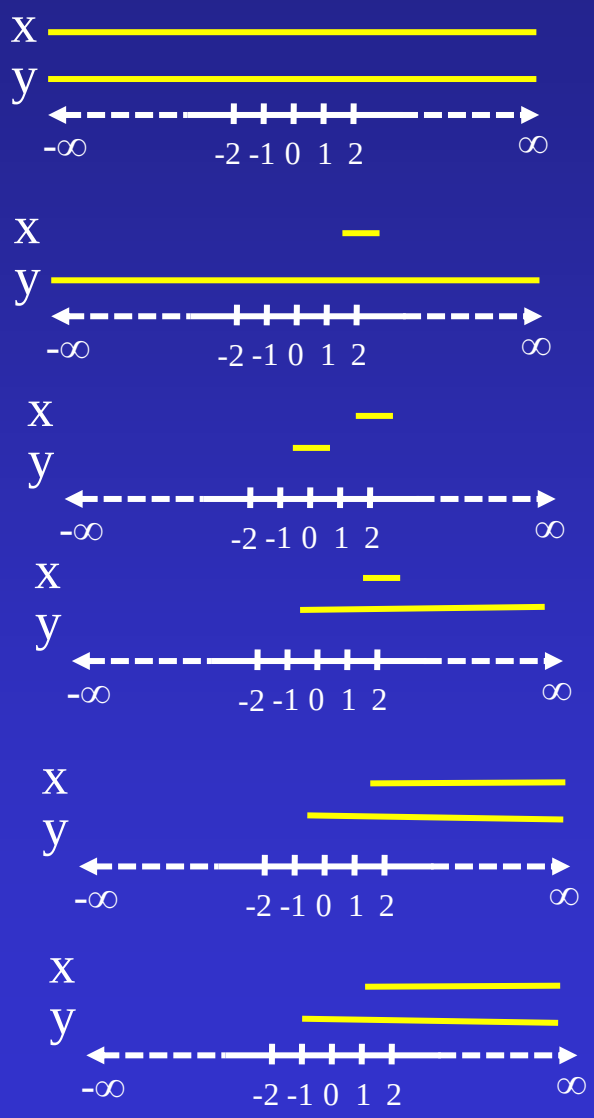
Infer Inductive Invariants via AI

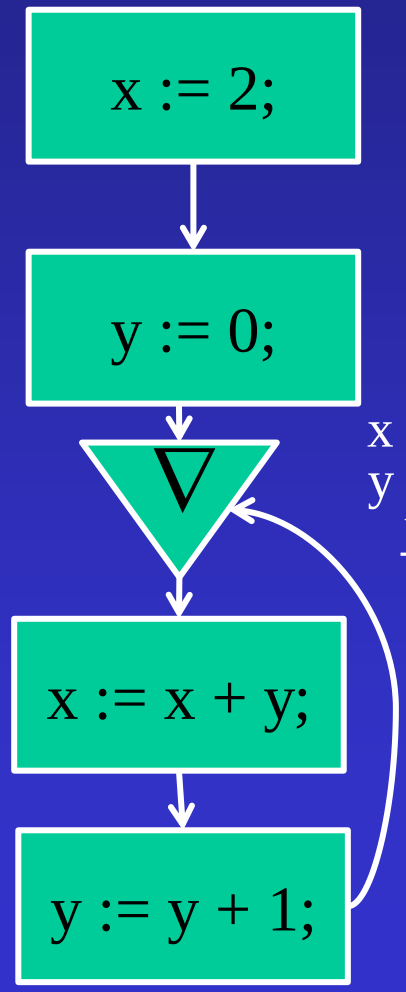


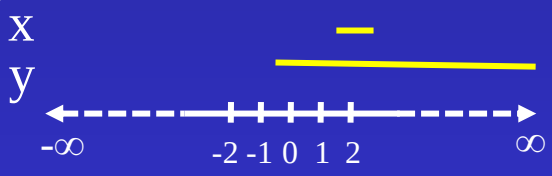




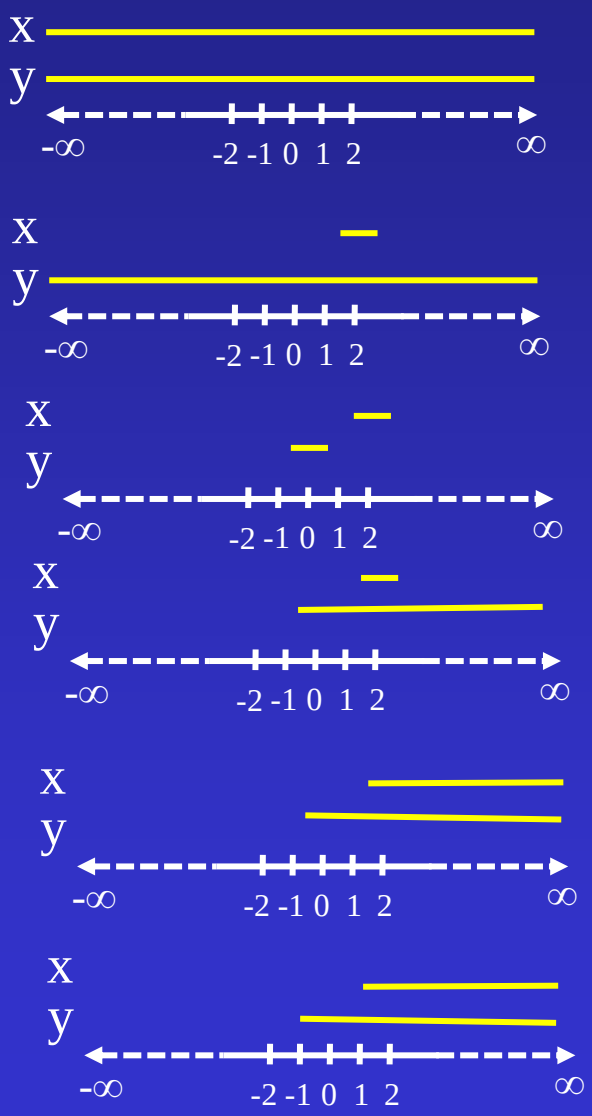
Infer Inductive Invariants via AI

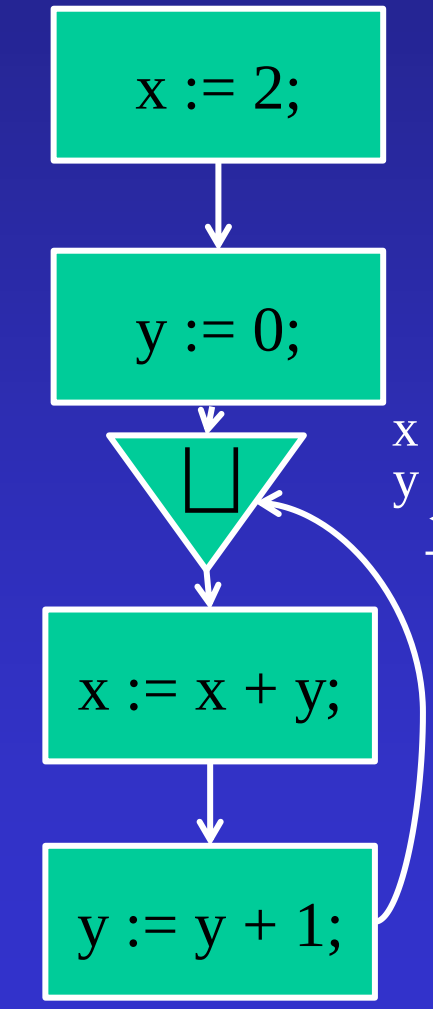


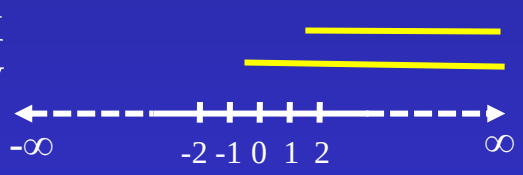




Infer Inductive Invariants via AI

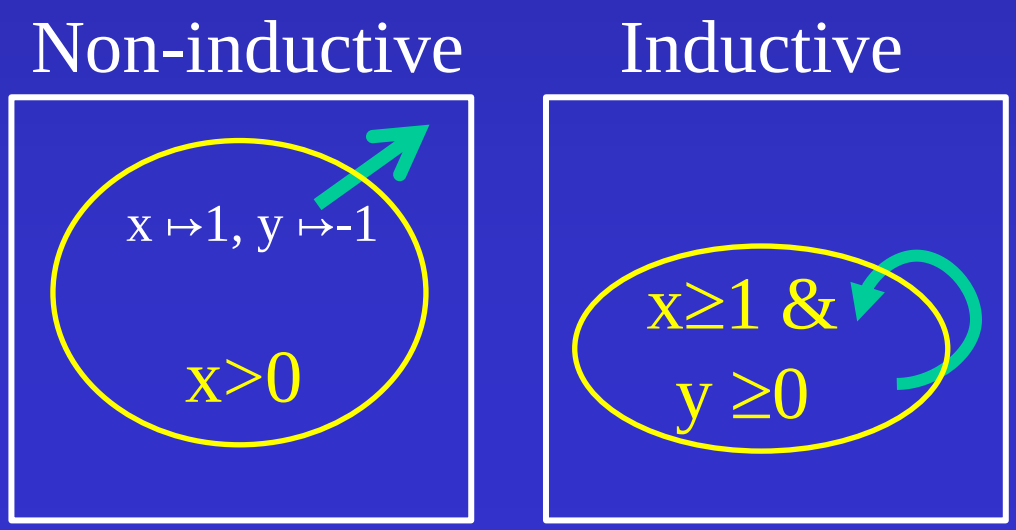






AI Infers Inductive Invariants

```
x := 2; y := 0;
while true do
        assert x > 0 ;
        x := x + y;
        y := y + 1
```



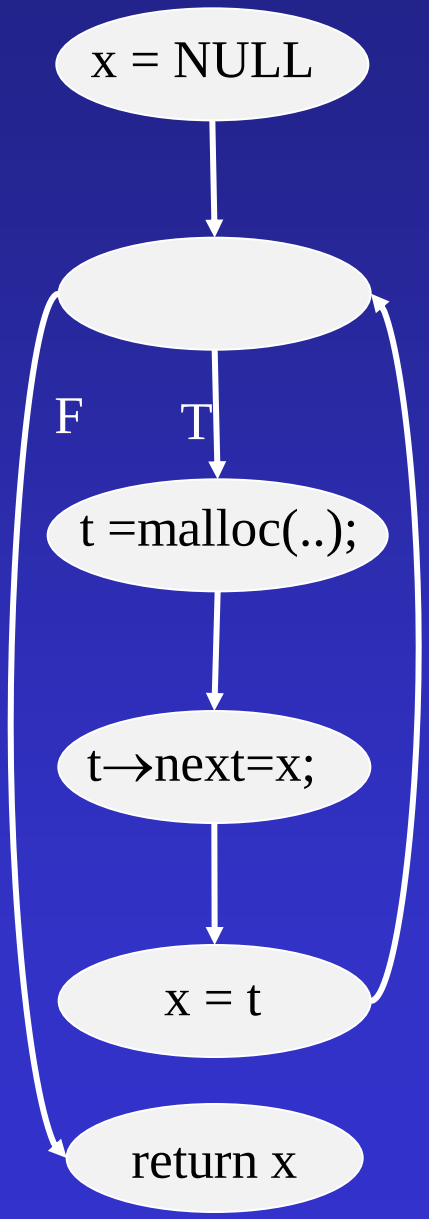
Original Problem: Shape Analysis (Jones and Muchnick 1981)

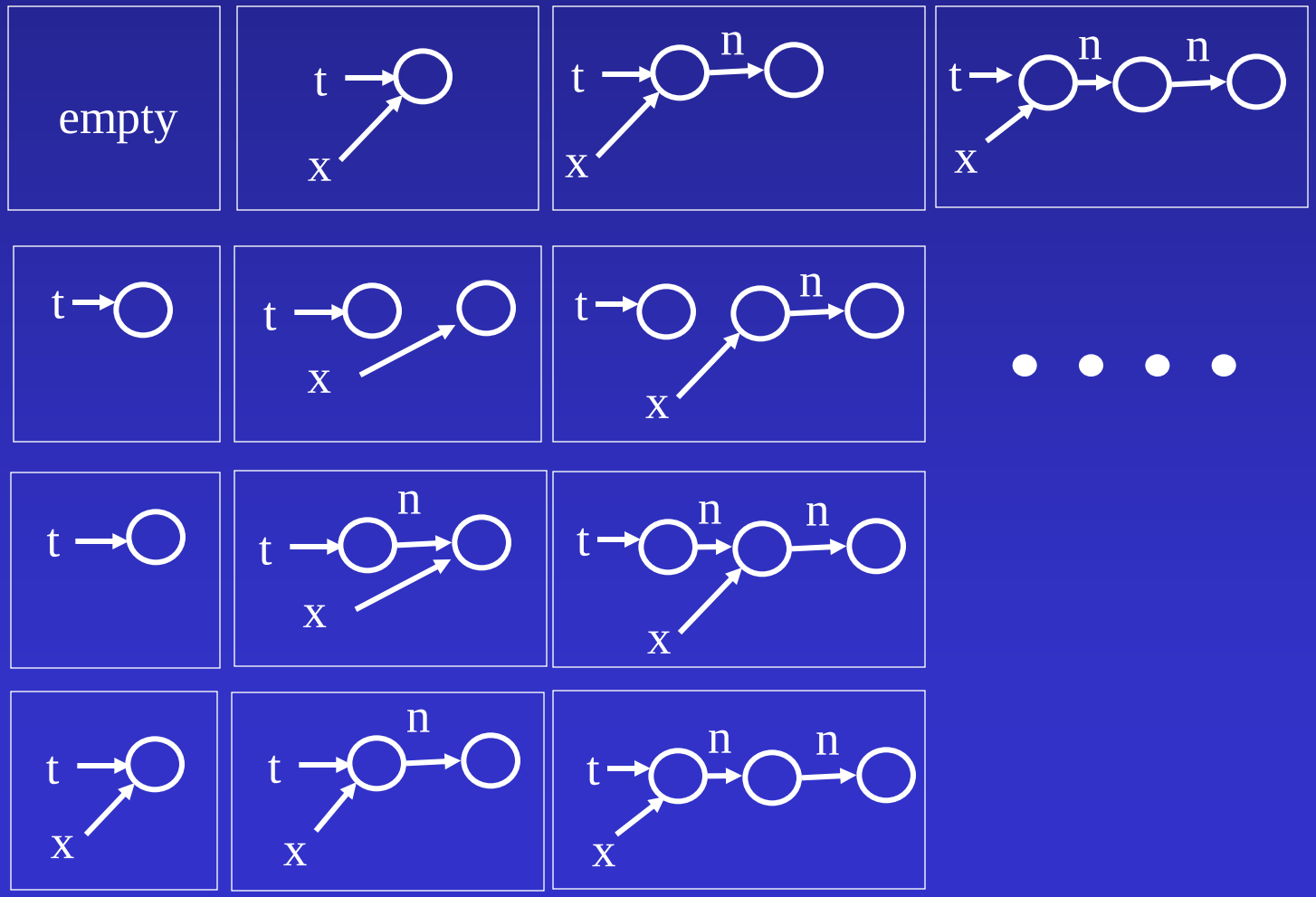
- Characterize dynamically allocated data
  - *x* points to an acyclic list, cyclic list, tree, dag, etc.
  - show that data-structure invariants hold
- Identify may-alias relationships
- Establish "disjointedness" properties
  - *x* and *y* point to structures that do not share cells
- Memory Safety
  - No null and dangling de-references
  - No memory leaks
- In OO programming
  - Everything is in the heap ➔ requires shape analysis

Why Bother?

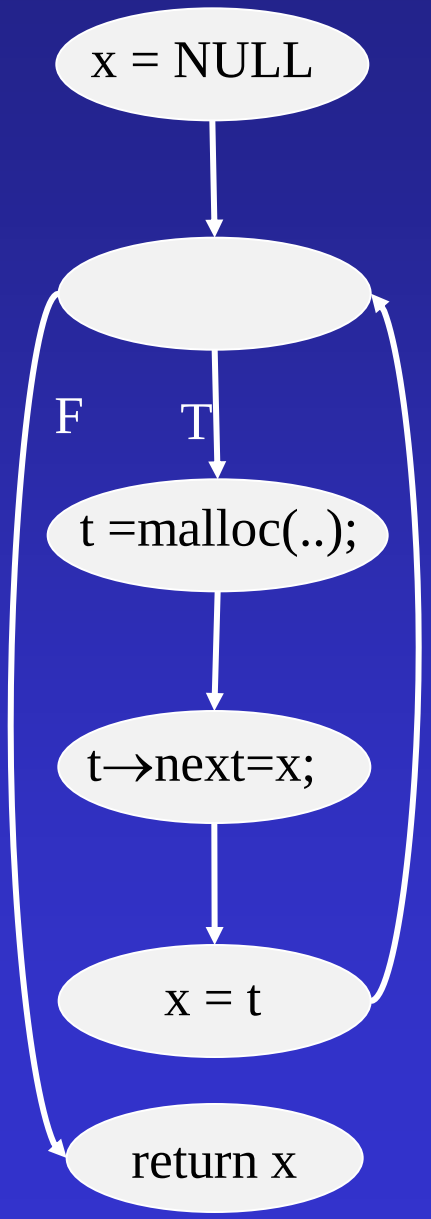
```
    int *p,  *q;
    q = (int *) malloc();
    p = q;
l1: *p = 5;
     p = (int *) malloc();
l2: printf(*q);  /* printf(5) */
```

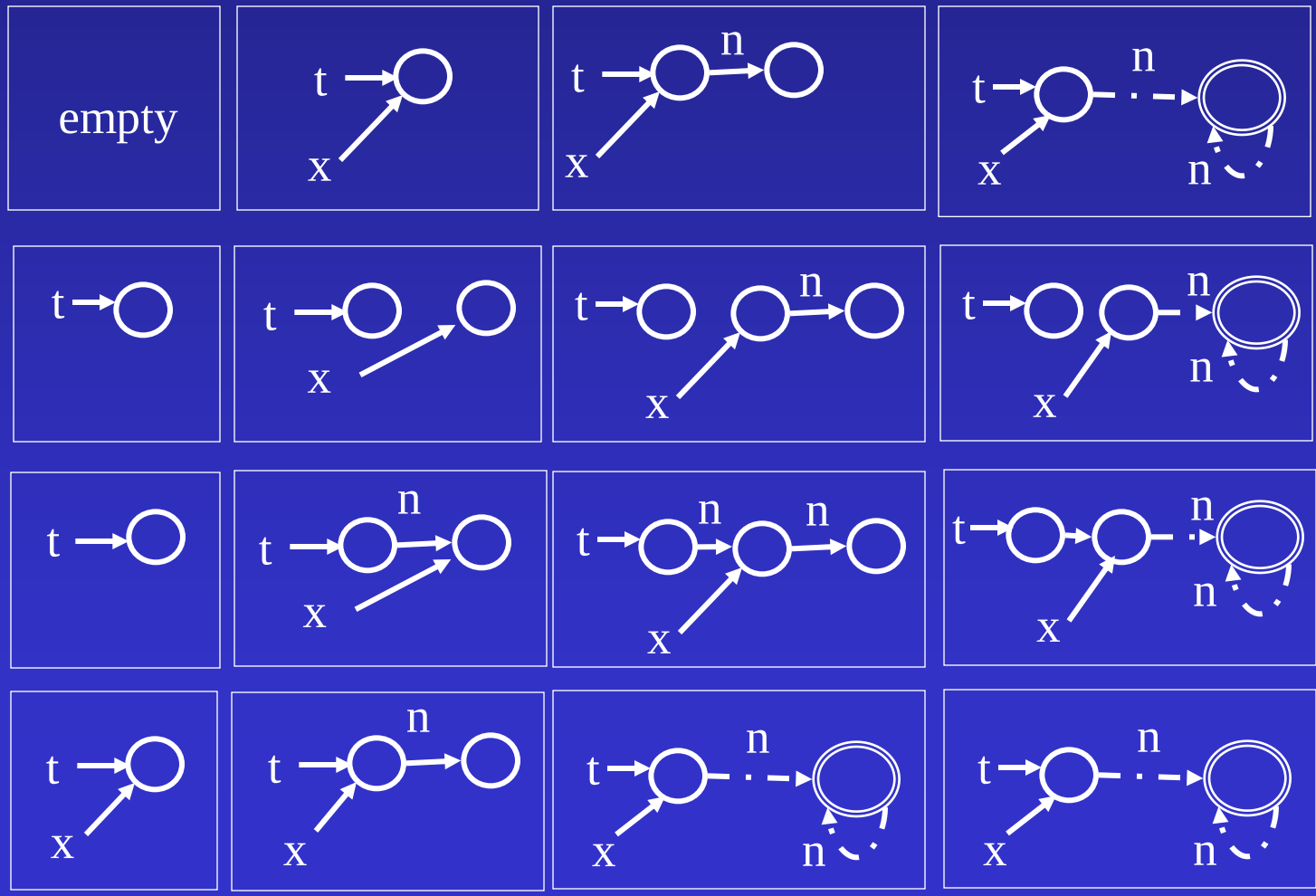
Example: Concrete Interpretation





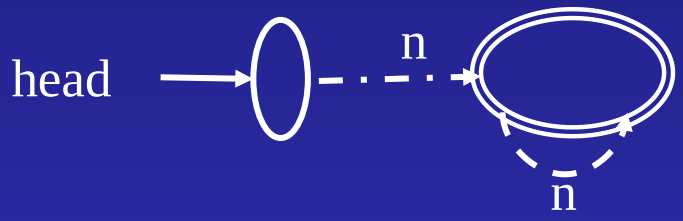
Example: Abstract Interpretation

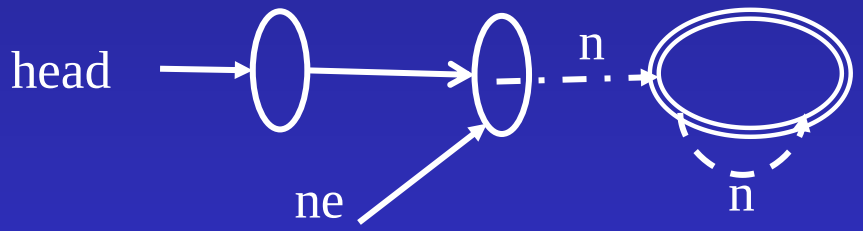


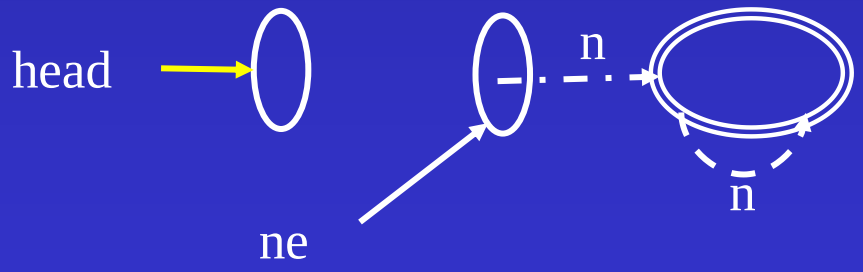


Memory Leakage

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



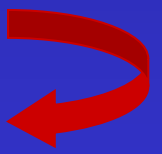




leakage of address pointed to by head

Memory Leakage

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



No memory leaks

Mark and Sweep

```
void Mark(Node root) {
  if (root != NULL) {
    pending = ∅
    pending = pending ∪ {root}
    marked = ∅
    while (pending ≠ ∅) {
      x = SelectAndRemove(pending)
      marked = marked ∪ {x}
      t = x → left
      if (t ≠ NULL)
        if (t ∉ marked)
          pending = pending ∪ {t}
      t = x → right
      if (t ≠ NULL)
        if (t ∉ marked)
          pending = pending ∪ {t}
    }
  }
  assert(marked = = Reachset(root))
}
```

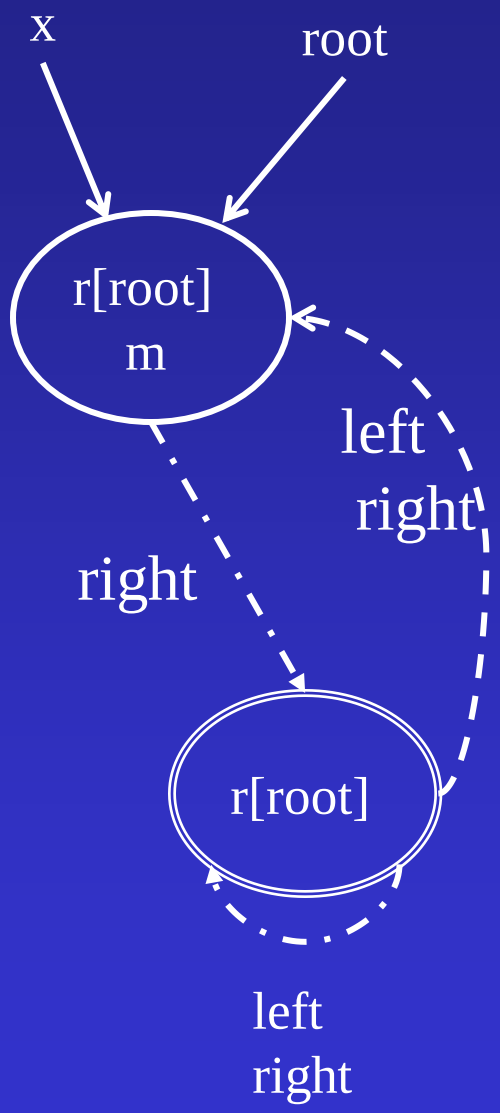
```
void Sweep() {
  unexplored = Universe
  collected = ∅
  while (unexplored ≠ ∅) {
    x = SelectAndRemove(unexplored)
    if (x ∉ marked)
      collected = collected ∪ {x}
  }
  assert(collected = =
          Universe – Reachset(root)
        )
}
```

$\forall v: marked(v) \Leftrightarrow reach[root](v)$

Example: Mark

```
void Mark(Node root) {
  if (root != NULL) {
    pending = ∅
    pending = pending ∪ {root}
    marked = ∅
    while (pending ≠ ∅) {
      x = SelectAndRemove(pending)
      marked = marked ∪ {x}
      t = x → left
      if (t ≠ NULL)
        if (t ∉ marked)
          pending = pending ∪ {t}
/*    t = x → right
 *    if (t ≠ NULL)
 *      if (t ∉ marked)
 *        pending = pending ∪ {t}
 */ }
  }
  assert(marked = = Reachset(root))
}
```

Bug Found



- There may exist an individual that is reachable from the root, but not marked

$$r[root](root) \wedge \neg p(root) \wedge m(root) \wedge$$
$$\exists e: r[root](e) \wedge \neg m(e) \wedge \neg root(e) \wedge \neg p(e)$$
$$\forall r, e: (root(r) \wedge r[root](r) \wedge \neg p(r) \wedge m(r) \wedge$$
$$r[root](e) \wedge \neg m(e)) \wedge \neg root(e) \wedge \neg p(e))$$
$$\rightarrow \neg left(r,e)$$

Properties Proved

Program	Properties	#Graphs	Seconds
LindstromScan	CL, DI	1285	8.2
LindstromScan	CL, DI, IS, TE	183564	2185
SetRemove	CL, DI, SO	13180	106
SetInsert	CL, DI, SO	299	1.75
DeleteSortedTree	CL, DI	2429	6.24
DeleteSortedTree	CL, DI, SO	30754	104
InsertSortedTree	CL, DI	177	0.85
InsertSortedTree	CL, DI, SO	1103	2.5
InsertAVLttree	CL, DI, SO	1855	27.4
RecQuickSot	CL, DI, SO	5585	9.2

CL=memory safety DI=data structure invariant TE=termination SO=sorted

Success Story: The SLAM/SDV Project MSR

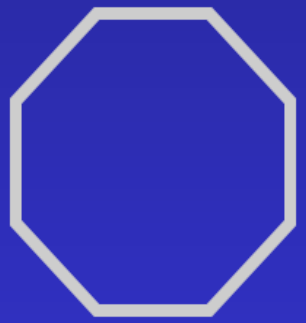
- Tool for finding possible bugs in Windows device drivers
- Complicated back-out protocols in driver APIs when events cancelled or interrupted

"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we're building tools that can do actual proof about the software and how it works in order to guarantee the reliability."
Bill Gates, April 18, 2002. Keynote address **at** WinHec 2002

[
E
[
Automatic Predicate Abstraction of
[POPL'04] T. A. Henzinger, R. Jhala, R. Majumdar, K. L. McMillan: Abstractions from proofs

Success Story: Astrée

- Developed at ENS
- A tool for checking the absence of runtime errors in Airbus flight software



[CC'00] R. Shaham, E.K. Kolodner, S. Sagiv:
Automatic Removal of Array Memory Leaks in Java
[WCRE'2001] A. Miné: The Octagon Abstract Domain
[PLDI'03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival: A static analyzer for large safety-critical software

Success: Panaya Making ERP easy

- Static analysis to detect the impact of a change for ERP professionals (slicing)
- Developed by N. Dor and Y. Cohen
- Acquired by Infosys

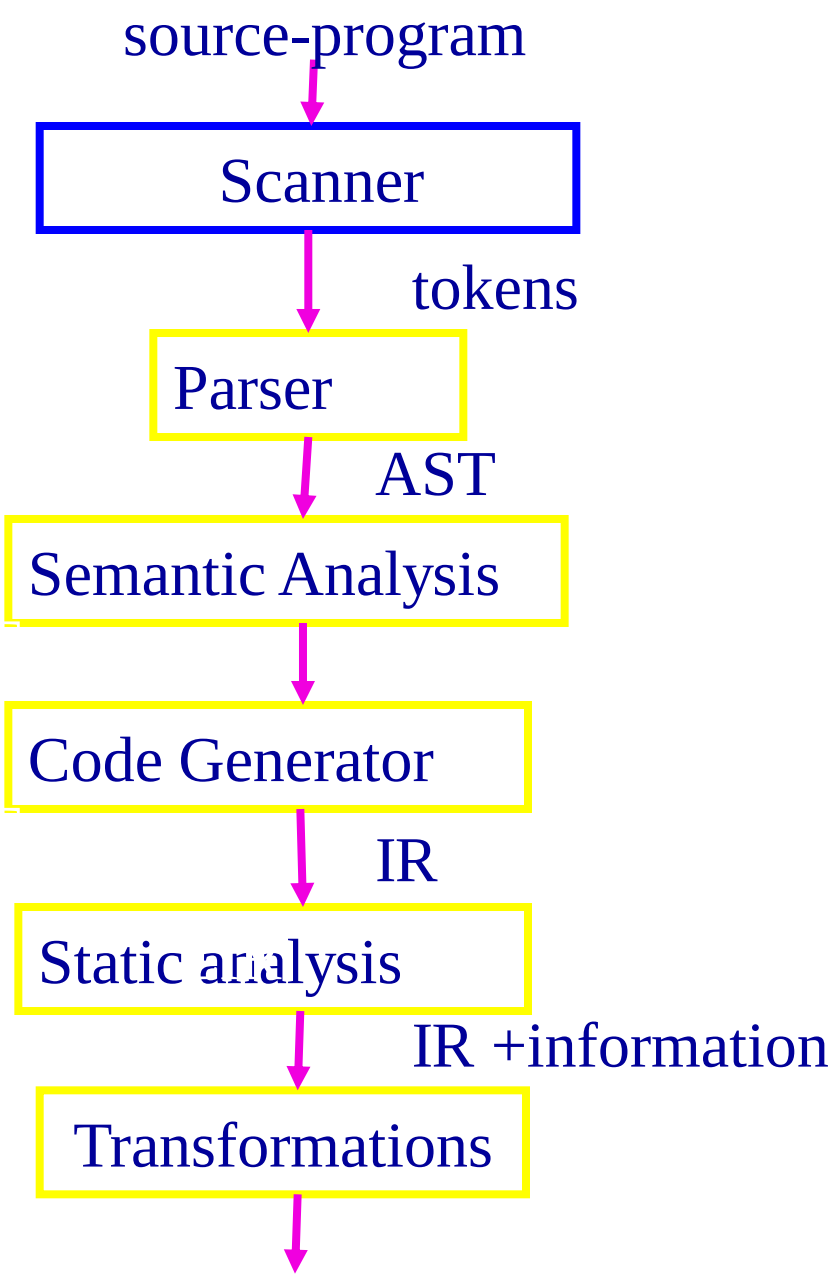
[ISSTA'08] N. Dor, T. Lev-Ami, S. Litvak, M. Sagiv, D. Weiss: Customization change impact analysis for erp professionals via program slicing

[FSE'10] S. Litvak, N. Dor, R. Bodík, N. Rinetzky, M. Sagiv: Field-sensitive program dependence analysi

Plan

- ✓ A bird's eye view of (program) static analysis
- Abstract Interpretation
- Tentative schedule

Compiler Scheme



Example Program Analyses

- Live variables
- 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 termination problem

The Program Termination Problem

- Determine if the program terminates on all possible inputs

Program Termination
Simple Examples

```
z := 3;
while  z > 0 do {
        if  (x == 1) z := z +3;
        else  z := z + 1;
```

```
while  z > 0 do {
        if  (x == 1) z := z  -1;
        else  z := z  -2;
```

Program Termination
Complicated Example

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

Summary Program Termination

- Very hard in theory
- Many programs terminate for simple reasons
- But termination may involve proving intricate program invariants
- Tools exist
  - MSR Terminator http://research.microsoft.com/en-us/um/cambridge/projects/terminator/
  - ARMC http://www.mpi-sws.org/~rybal/armc/

The Need for Static Analysis

- Compilers
  - Advanced computer architectures
  - 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) 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)

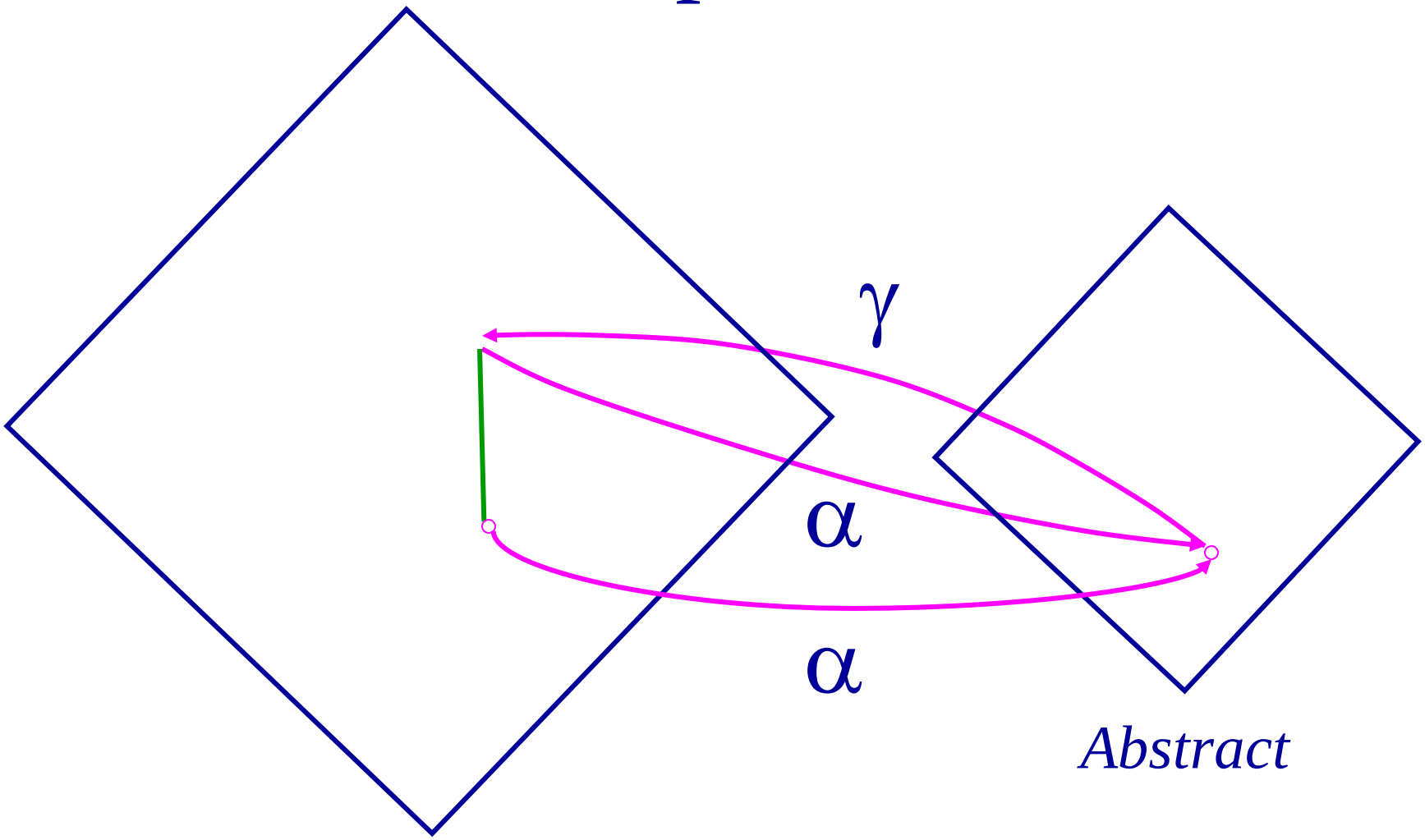
Even/Odd Abstract Interpretation

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

Example Program

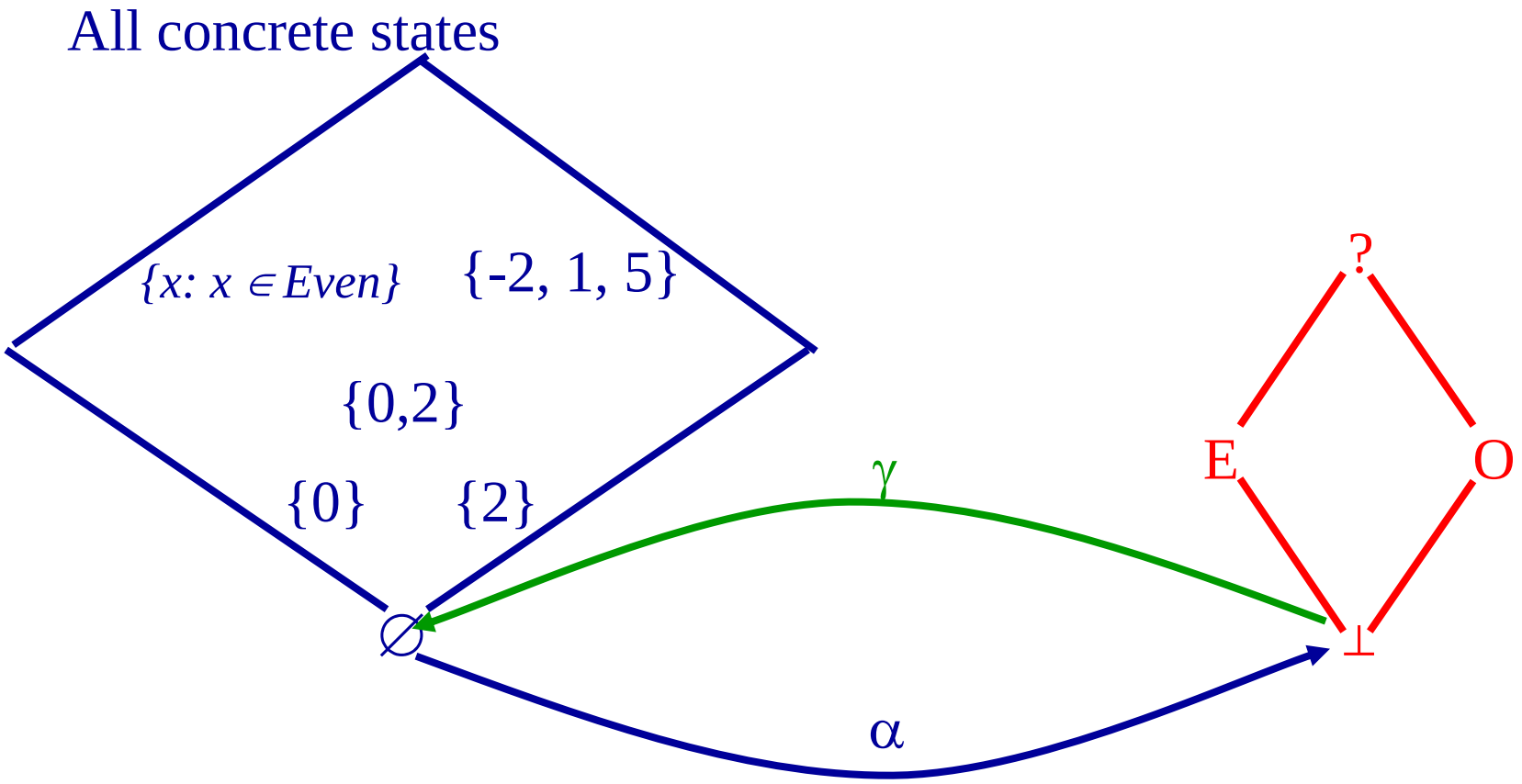
```
   /* x=? */

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

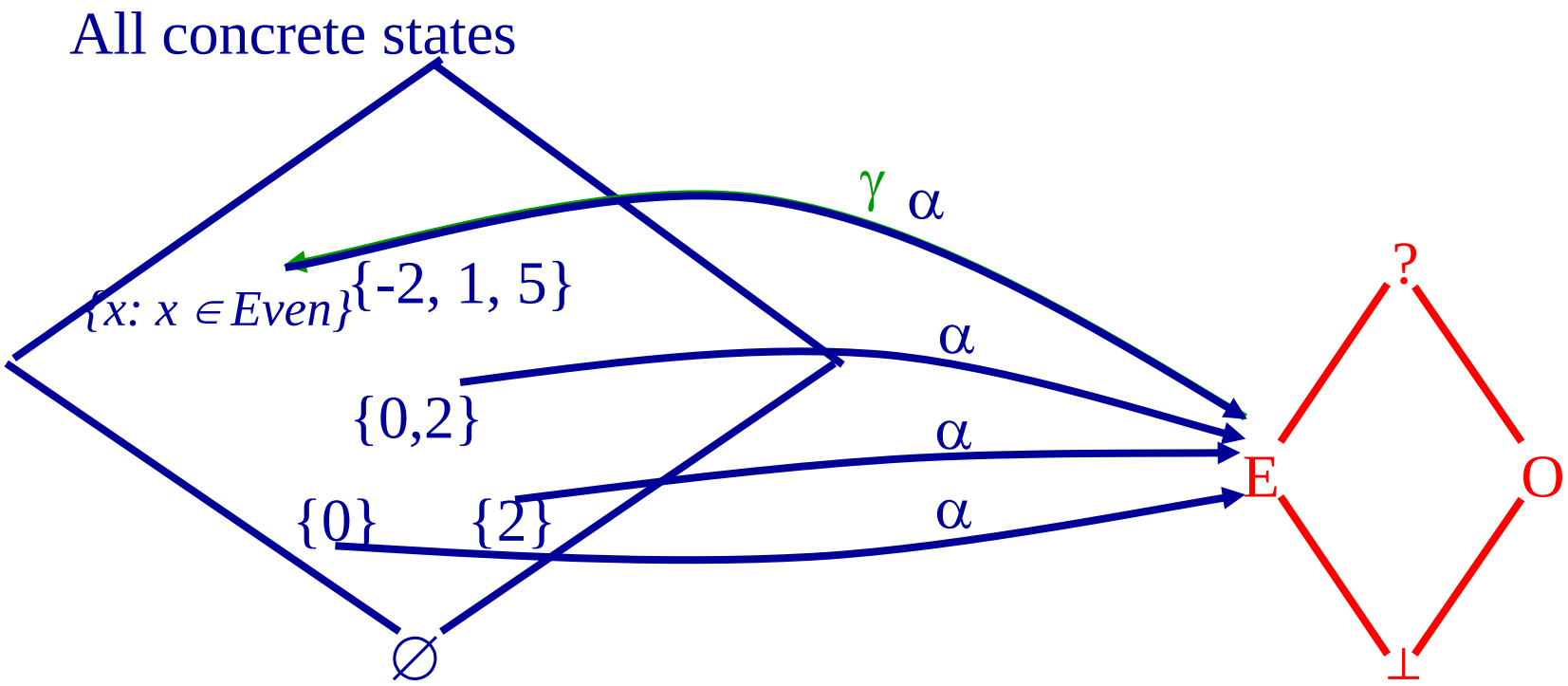


Sets of stores $\xrightarrow{\alpha}$ *Descriptors of sets of stores*

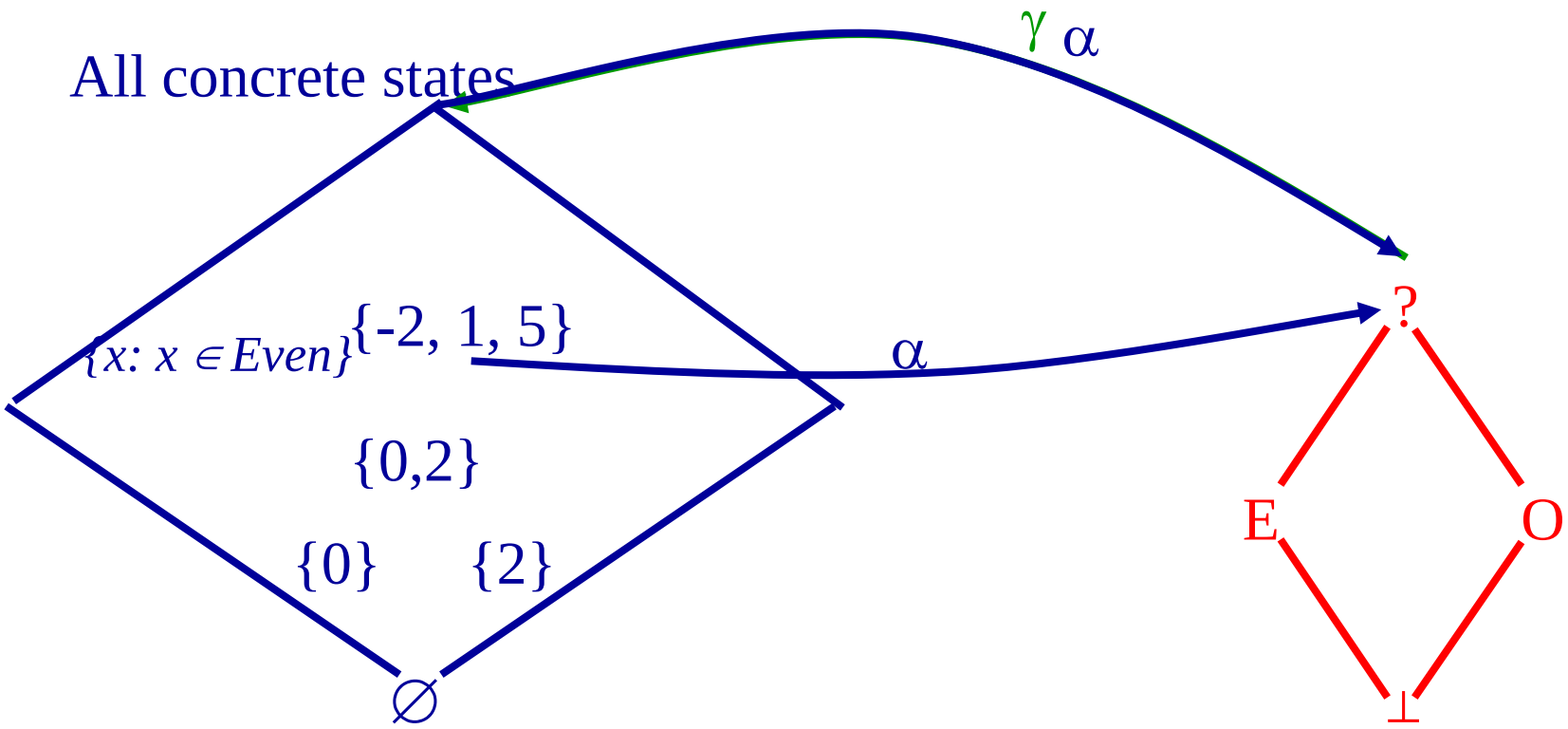
Odd/Even Abstract Interpretation



Odd/Even Abstract Interpretation



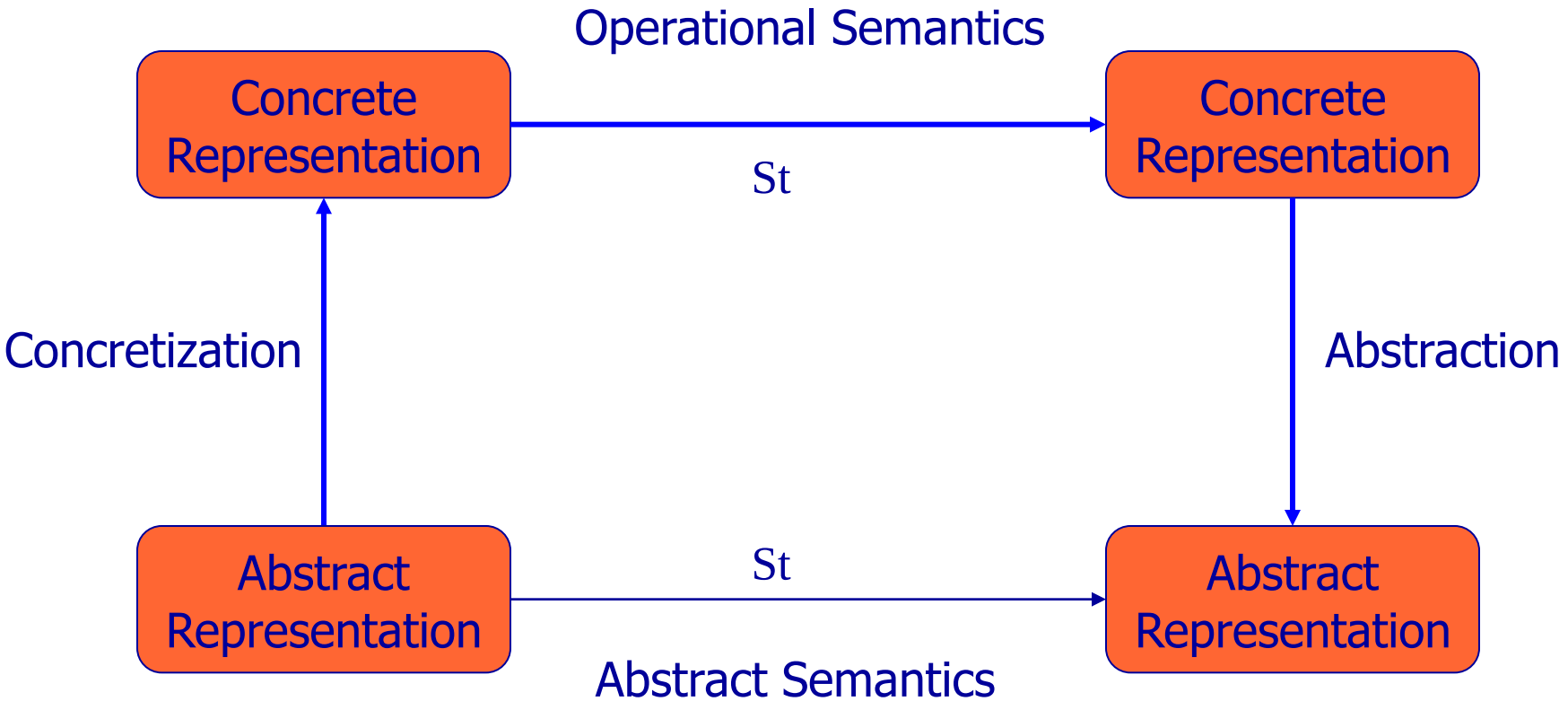
Odd/Even Abstract Interpretation



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



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	…
⋮	⋮	⋮	⋮	⋮	

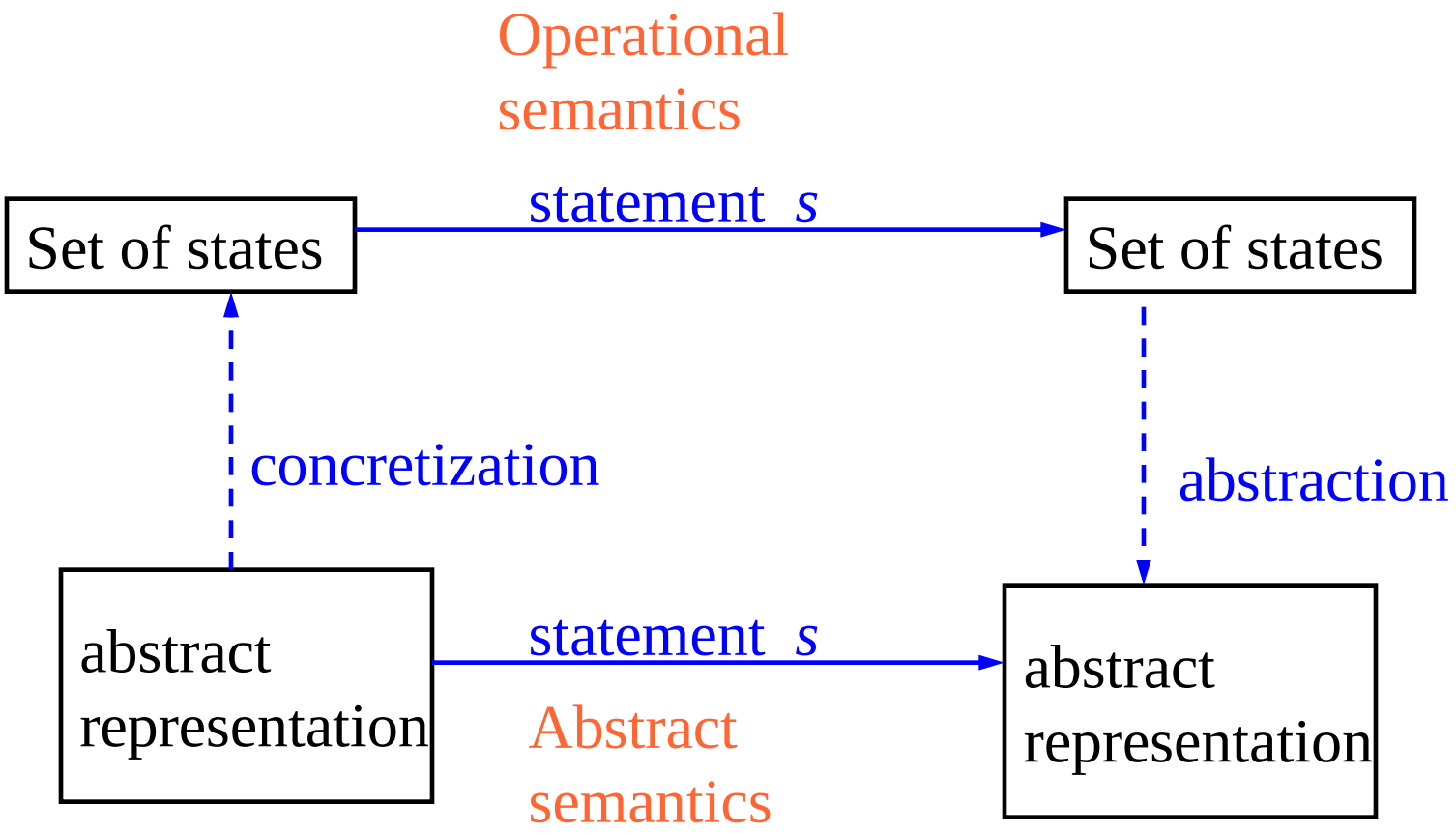
+′	?	O	E
?	?	?	?
O	?	E	O
E	?	O	E

*′	?	O	E
?	?	?	E
O	?	O	E
E	E	E	E

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
	No need to efficiently handle rare cases	Can handle limited classes of programs and still be useful

Abstract (Conservative) interpretation

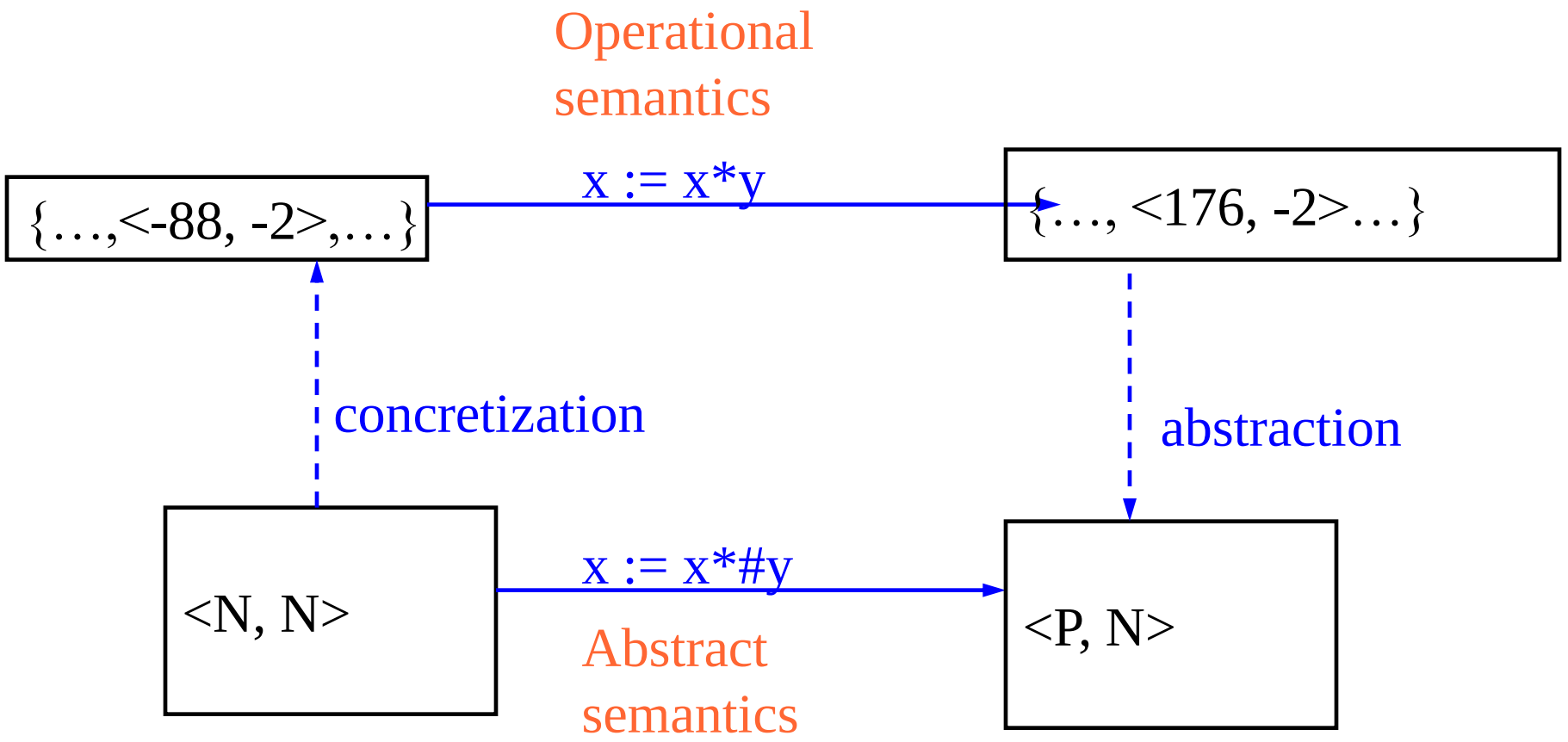


Example rule of signs

- 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, ?}
- α(C) = if all elements in C are positive
  then return P
  else if all elements in C are negative
  then return N
  else return ?
- γ(a) = if (a==P) then
  return{0, 1, 2, … }
  else if (a==N)
  return {-1, -2, -3, …, }
  else return Z

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

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 + p^2(x_1, x_2, ..., x_n))
/* Is y=0 here? */
```

Coping with undecidabilty

- Loop free programs
- Simple static properties
- Interactive solutions
- Conservative estimations
  - Every enabled transformation cannot change the meaning of the code but some transformations are no enabled
  - Non optimal code
  - Every potential error is caught but some “false alarms” may be issued

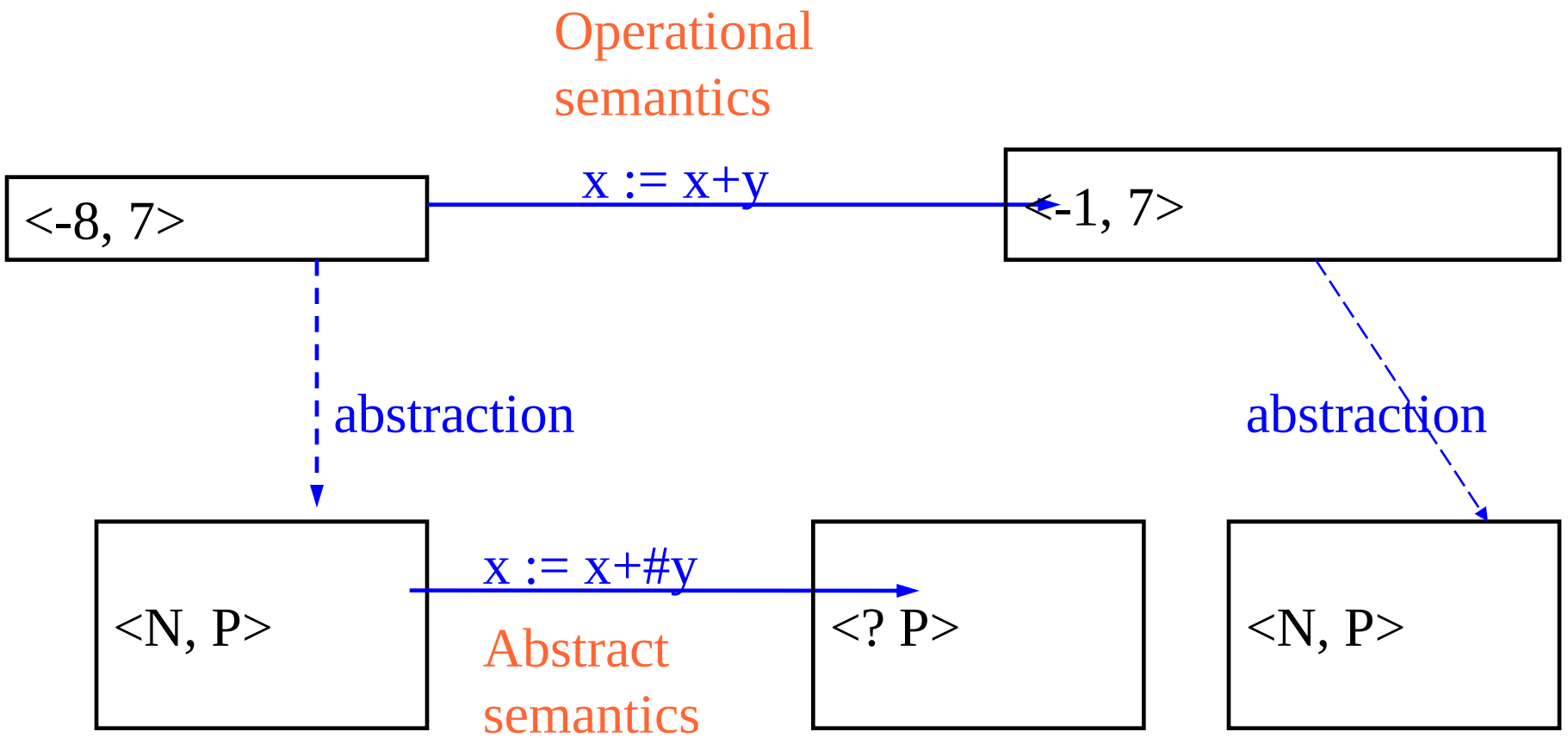
Analogies with Numerical Analysis

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

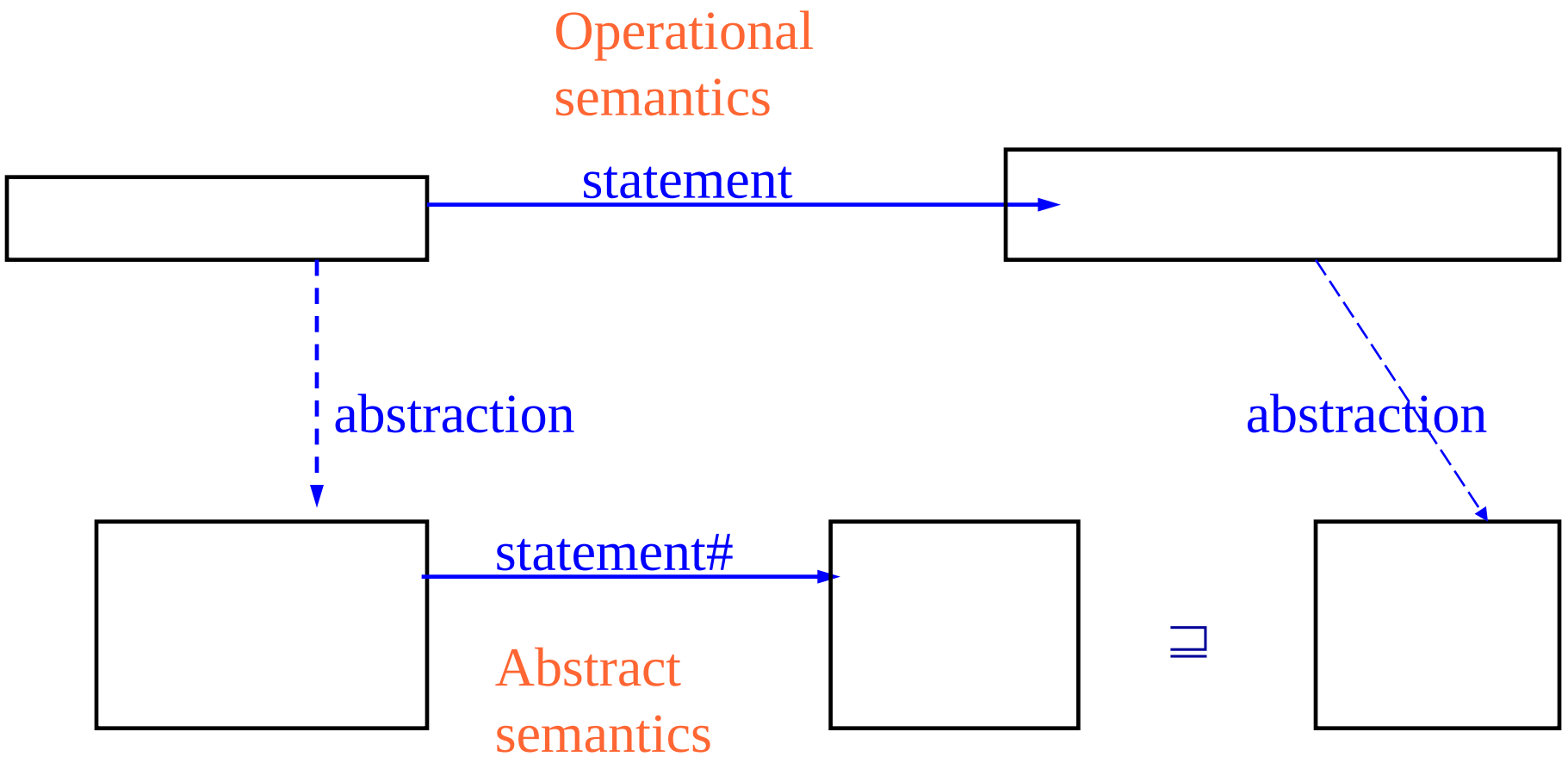
Violation of soundness

- Loop invariant code motion
- Dead code elimination
- Overflow
  ((x+y)+z) != (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

Complementary Techniques

- Dynamic Analysis
- Testing/Fuzzing
- Bounded Model Checking
- Deductive Verification
- Proof Assistance (Coq)

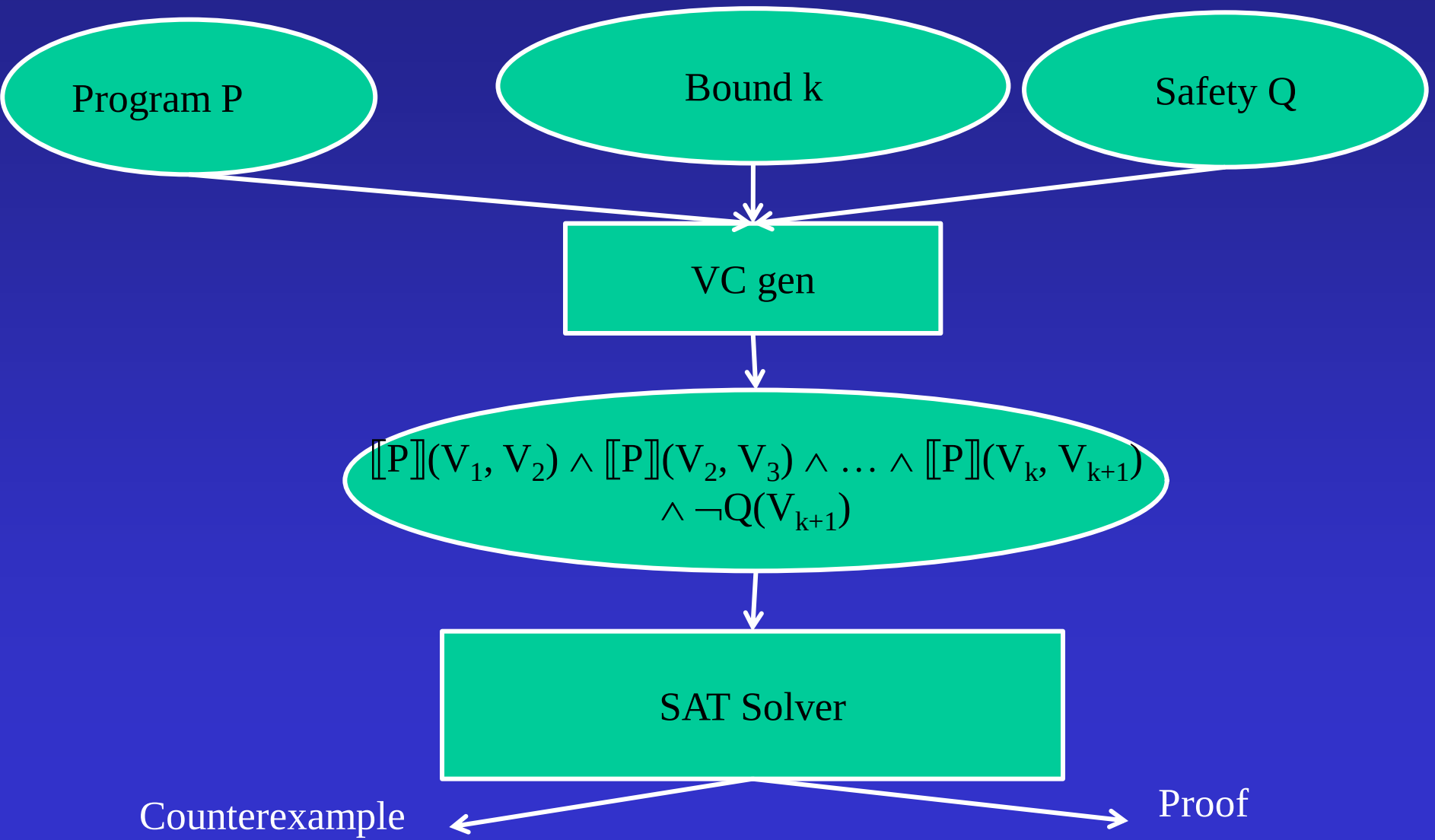
Fuzzing [Miller 1990]

- Test programs on random unexpected data
- Can be realized using black/white testing
- Can be quite effective
  - Operating Systems
  - Networks
- …
- Usually implemented via instrumentation
- Tricky to scale for programs with many paths

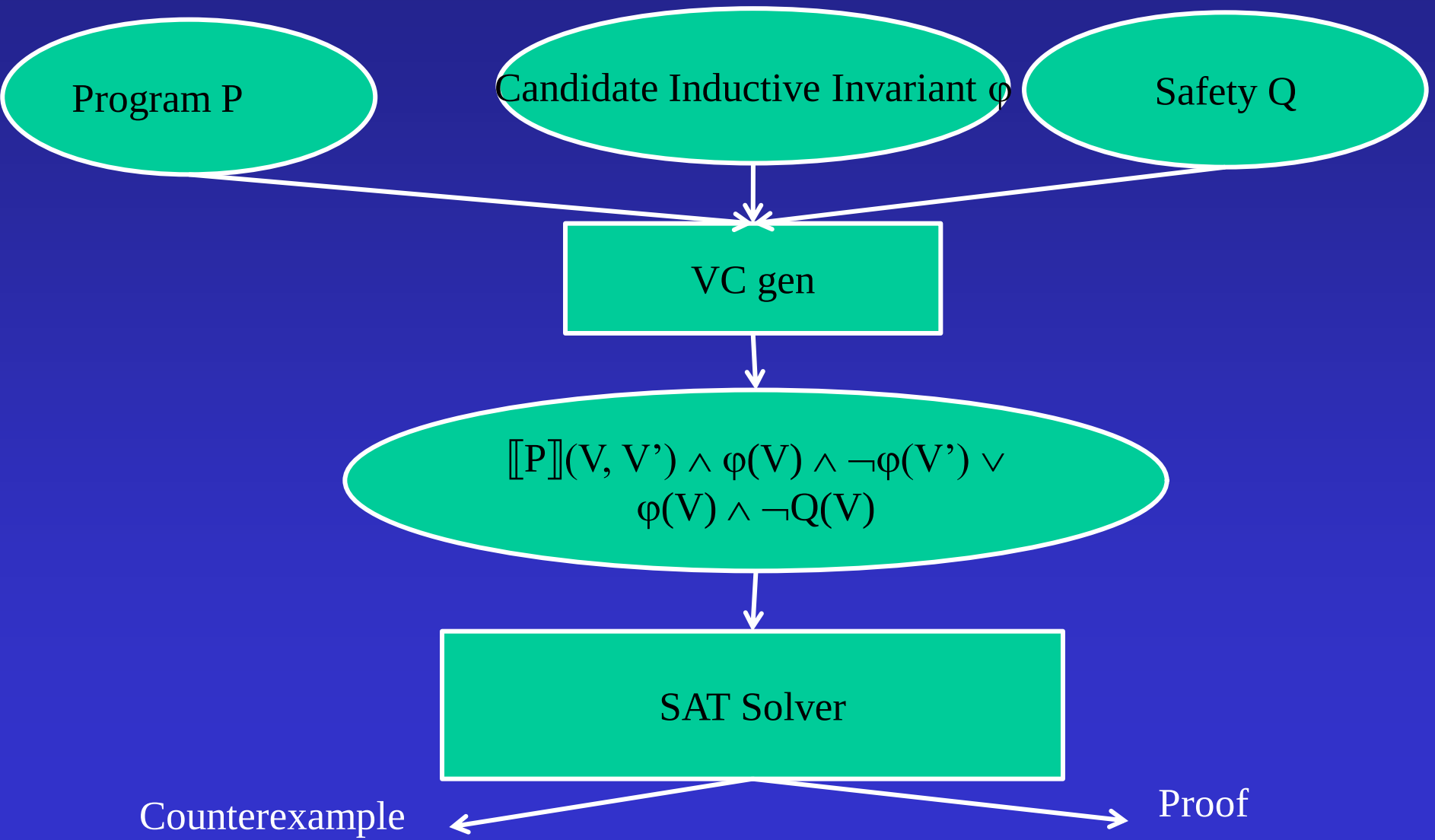
```
If (x == 10001) {

    ….
    if (f(*y) == *z) {
    ….
```

```
int f(int *p) {

  if (p !=NULL) {
    return q ;

}
```

Bounded Model Checking



Deductive Verification



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 foudedness 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 Scwartz]

Tentative Schedule

Date	Topic
25/10	Chaotic Iteration
1,8,15,22,29/11, 6/12	Theory and practice of AI (4 assignments)
20,27/12, 3, 10/1	ivy
17/1	Project Selection

Summary

- Static analysis is powerful
- Precision and scalability is an issue
- Static Analysis and Theorem Proving can be combined in many ways