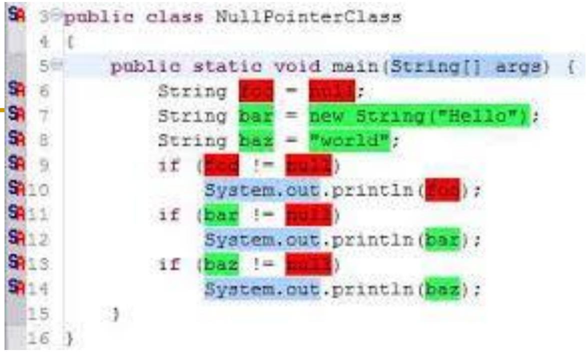
COMP5111 – Fundamentals of Software Testing and Analysis

Pointer Analysis & Abstract Interpretation

Shing-Chi Cheung

Computer Science & Engineering

HKUST



Pointer Analysis by Soot

Adapted from Charles Zhang's lecture notes

Pointer Operations are Common

	Referencing (Create location)	Dereferencing (Access location)	Aliasing (Copy pointer)
C:	`my_t *p = &var;` `p = malloc(8);`	`int x = *ptr;` `x = ptr2->field;`	`my_t *pa;` `pa = pb;`
Java:	`A a = new A();`	`int x = a.f;`	`A a = b;`

Pointer related bugs are also common

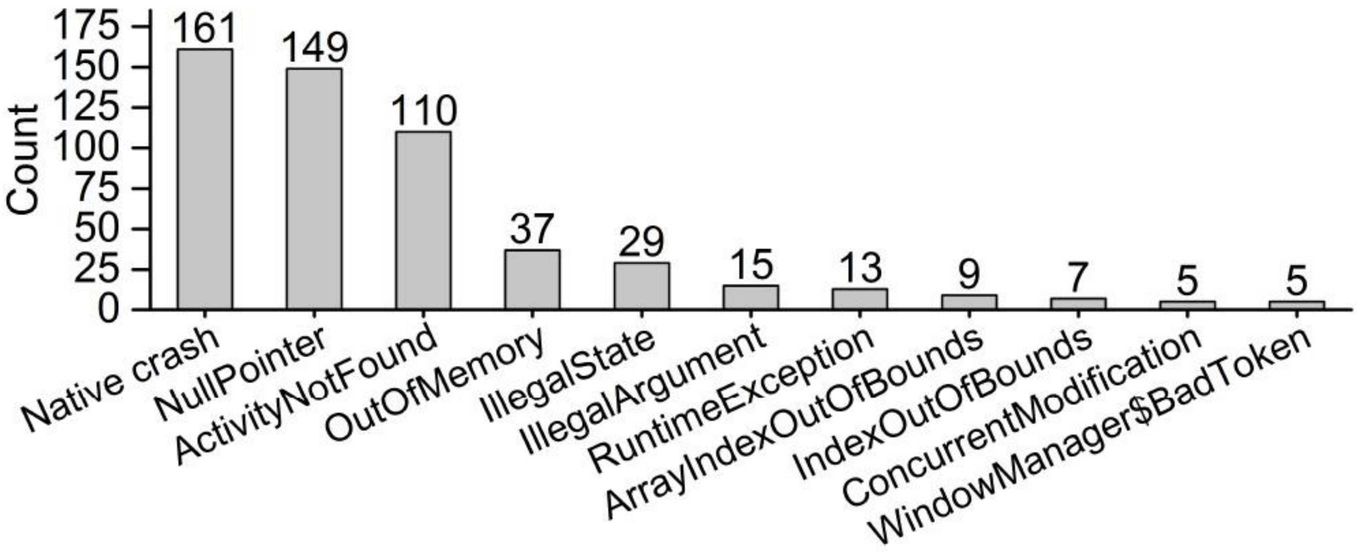
- Null pointer dereference
- Memory leaks
- Use after free / Double free
- Array index out of bounds
- Uninitialized pointers
- Mismatched malloc / free
- Buffer overflows

```
void foobar(int i) {
  char* p = new char[10];
  if ( i ) {
    p = 0;   // memory leak
  }
  if ( p->value == 0 ) … // null pointer
  delete[] p;
}
```

https://www.geeksforgeeks.org/common-memory-pointer-related-bug-in-c-programs/

1,340,561 (82.6%) out of the 1,622,375 code revisions of IF-clauses filed at GitHub as at Sept 2015 involve null-pointer checks.

Pointer related bugs dominate in Android applications



Main Crash Types on Google Play Subjects

Source: https://arstechnica.com/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz/

Pointers Complicate Compiler Optimization

- Example:

Compiler can determine the value of c at compile time

```
a = 1;          a = 1;
b = 1;   --->   b = 1;
c = a + b;      c = 2;
```

What if the program uses a pointer?

***p may modify the value of a or b. We may not pre-compute c.**

```
a = 1;          a = 1;
b = 1;   --->   b = 1;
*p = 2;         c = ?;
c = a + b;
```

Pointers Complicate Compiler Optimization

If we know p never points to a or b:

```
a = 1;                                          a = 1;
b = 1;      Program transformation:             b = 1;
*p = 2;     Avoid runtime a+b computation      *p = 2;
c = a + b;  ------------------------------->    c = 2;
```

If we know p must point to a or b:

```
a = 1;                                          a = 1;
b = 1;      Program transformation:             b = 1;
*p = 2;     Avoid runtime a+b computation      *p = 2;
c = a + b;  ------------------------------->    c = 3;
```

Sources of Aliases

- Function calls:

```
int foo(int *p, int *q) {
  *p = 1; *q = 2;
  return *p + *q;
}
```

What is the return value of foo()?

Note: p and q themselves are different variables according to the C language.

Sources of Aliases

- Function calls:

```
int foo(int *p, int *q) {
  *p = 1; *q = 2;
  return *p + *q;
}

int main() {
  int a = 1;
  printf("%d\n", foo(&a, &a));
  return 0;
}
```

4

Note: p and q themselves are different variables according to the C language.

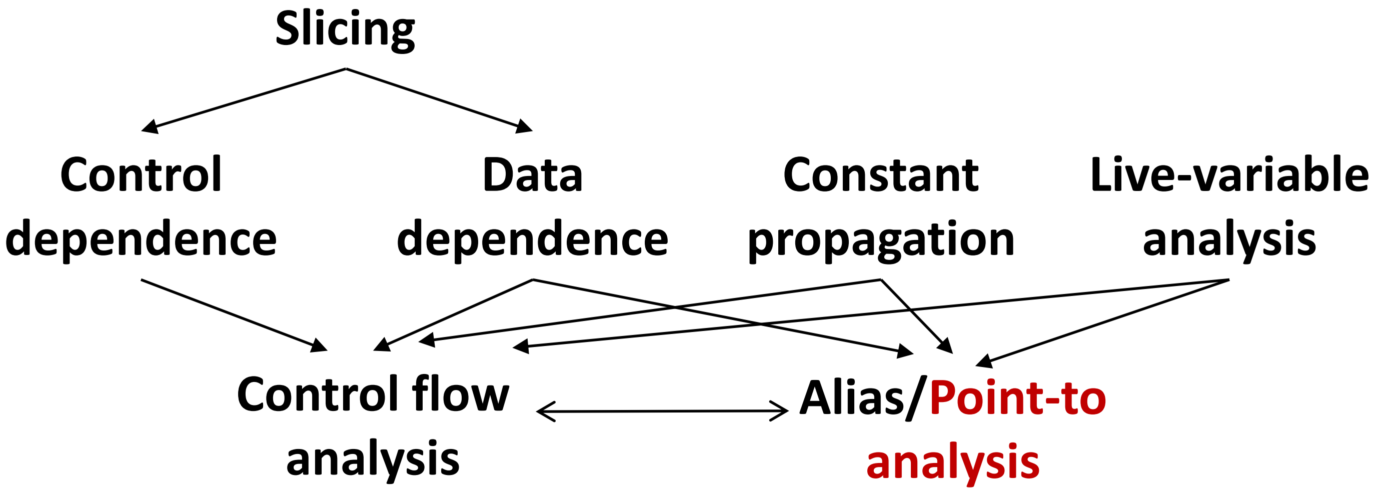
The expressions *p and *q access to the same memory location, thus *p is an alias of *q.

Sources of Aliases

- Address-of Operator:
  - int v;
  - int *p = &v;   // *p is an alias of v
- Dynamic Memory Allocation:
  - int *p = (int*) malloc(12);   // *p is an alias of a heap
- Array Arithmetic
  - int a[100];
  - int *p = a + x, *q = a + y; // *p is an alias of an array element

Pointer Analysis is important

- Alias information is a pre-requisite for many kinds of program analyses.



taken from Mary Jean Harrold's lecture notes

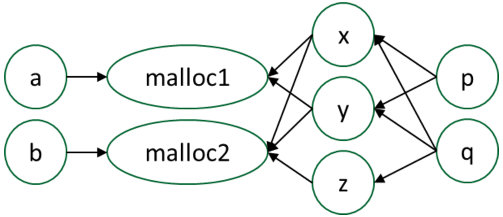
Many Uses of Pointer Analysis

- Basic compiler optimizations
  - Register allocation, dead code elimination, live variables, instruction scheduling, redundant load/store elimination
- Parallelization
  - Instruction-level parallelism, thread-level parallelism
- Error detection and program understanding
  - Memory leaks, security holes

Terminology

Let r_1 and r_2 represent two memory expressions. They can be the forms "x", "*p", "**p", "p->f", etc. We have the following relations:

- Alias: r_1 and r_2 are aliased if the memory locations accessed by r_1 and r_2 overlap, written as (r_1, r_2).
- Points-to: the value of memory location r_1 is the address of the memory location r_2, written as $r_1 \rightarrow r_2$.
- Points-to Set: the points-to set of r_1 contains all r_2 such that $r_1 \rightarrow r_2$, written as $pts(r_1)$. Two pointers p, q are said equivalent if $pts(p) = pts(q)$.
- Points-to Graph: A digraph where each node represents one or more memory locations; an edge from r_1 to r_2 means $r_1 \rightarrow r_2$.



Terminology

- Must Alias: The alias pair (r_1, r_2) holds in all program executions.
- May Alias: The alias pair (r_1, r_2) holds in some program execution.
- The must/may points-to relations are defined similarly.
- This lecture concerns May Points-to problem.

Terminology

- Alias Analysis:
  - Compute a set of ordered pairs $\{(r_i, r_j)\}$ denoting aliases that may hold during runtime
- Points-to Analysis:
  - For each pointer variable p, compute the set of objects pts(p) that p may point to during runtime

Points-to set

What's the difference between alias and points-to analysis?

Difference between Alias and Points-to

Example:

```
p = &a; q = &b;
if (…)
  p = &c;
else
  q = &c;
*p = *q + d;
```

- Alias emphasizes the simultaneity.
  - (p, q) is an alias pair if p and q refer to the same memory location simultaneously after executing a set of program instructions.
- Points-to emphasizes individuality.
  - p→c and q→c are two independent events.
  - pts(p)∩pts(q)≠Φ does not mean (p, q) is a true alias pair. For example, in the snippet on the left, *p never alias to *q.

pts(p) = {a, c}, pts(q) = {b, c}

Basics of Points-to Analysis

- A kind of static analysis
- All executable assumption:
  - All the *if* branches are considered to be executable, and we do not care about when the branch conditions are satisfied.
- More precise (path-sensitive) algorithms consider when the predicates are true, but this is not studied in this course.

Soundness

Let A be an analysis that deduces a property p, and $\models p$ denotes p holds in real program executions.

- Sound: $\models p \Rightarrow A \vdash p$ // no false negatives
- Exact: $A \vdash p \Leftrightarrow \models p$ // no false positives and negatives
- Precise: $A \vdash p \Rightarrow \models p$ // no false positives

We say an algorithm is sound in the detection of a property p when it always detects p if p exists.

http://www.pl-enthusiast.net/2017/10/23/what-is-soundness-in-static-analysis/

Basics of Points-to Analysis

- Safety property to be deduced
  - Whether a pointer NEVER (i.e., may not) points to a memory location.
- Sound:
  - The conclusion is sound if all the points-to relations that could occur in some real executions are included in the analysis result. It over-approximates the true points-to relation.

may relations

Basics of Points-to Analysis

What happens when executing *p = *q under different points-to analyses?

- Exact points-to:
  - a = d; b = c;
- Sound points-to:
  - a = b; a = d; c = b; c = d;
- Exact points-to is expensive; most points-to analyses aim to be sound.

```
p = &a; q = &b;
if (t > 0)
  p = &c;
else
  q = &d;
*p = *q;
```

pts(p) = {a, c},
pts(q) = {b, d}

Basics of Points-to Analysis

Program abstraction:

- Program abstraction is a static mechanism to approximate runtime memory.
- Since the runtime memory size is essentially unbounded (e.g., malloc, recursive callstacks), we define a function to map every runtime memory location to an abstract memory location. And the number of abstract memory locations is bounded.

Basics of Points-to Analysis

```
int add(int a, int b) {
  return a + b;
}
int main() {
  int x, y, t; scanf("%d", &t);
  while (t--) {
    scanf("%d %d", &x, &y);
    int m += add(x, y)
  }
  return 100 div m;
}
```

Program abstraction:

- We don't know how many times *add* will execute. Therefore, variables *a* and *b* have infinite runtime instances.
- R = {All runtime local variable locations};
- A = {a, b, x, y, t, m}
- *a* in A represents all runtime instances of local variable *a*.

abstract memory

Basics of Points-to Analysis

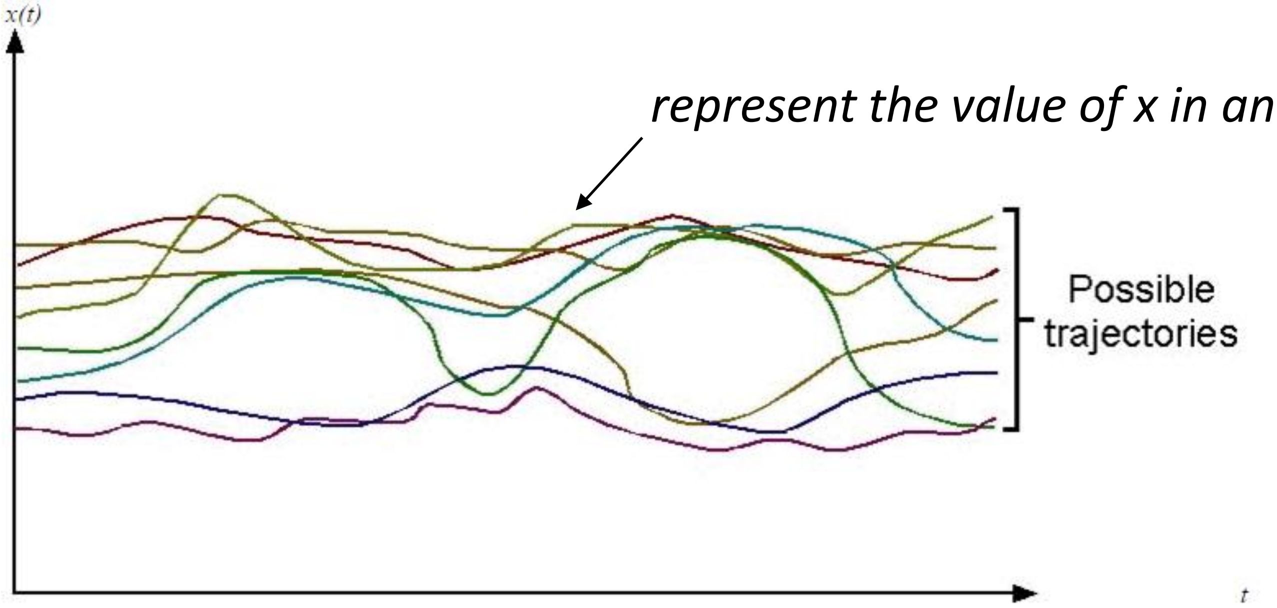
Points-to analysis has two parts:

- Abstract the given program (build the abstract domains of pointers and memories)
- Process the program constructs such as assignment “p = q;”

Program Abstraction (or Abstract Interpretation)

- Concrete program semantics

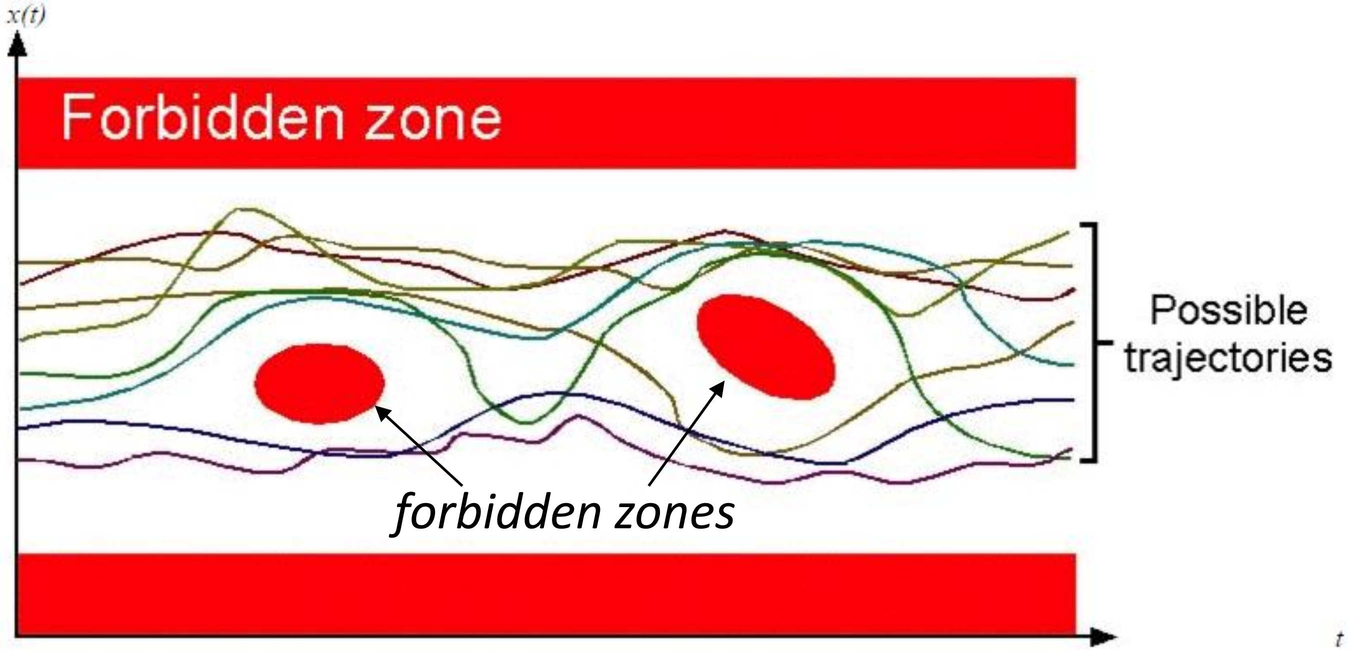
represent the value of x in an execution across time



extracted from http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Program Abstraction (or Abstract Interpretation)

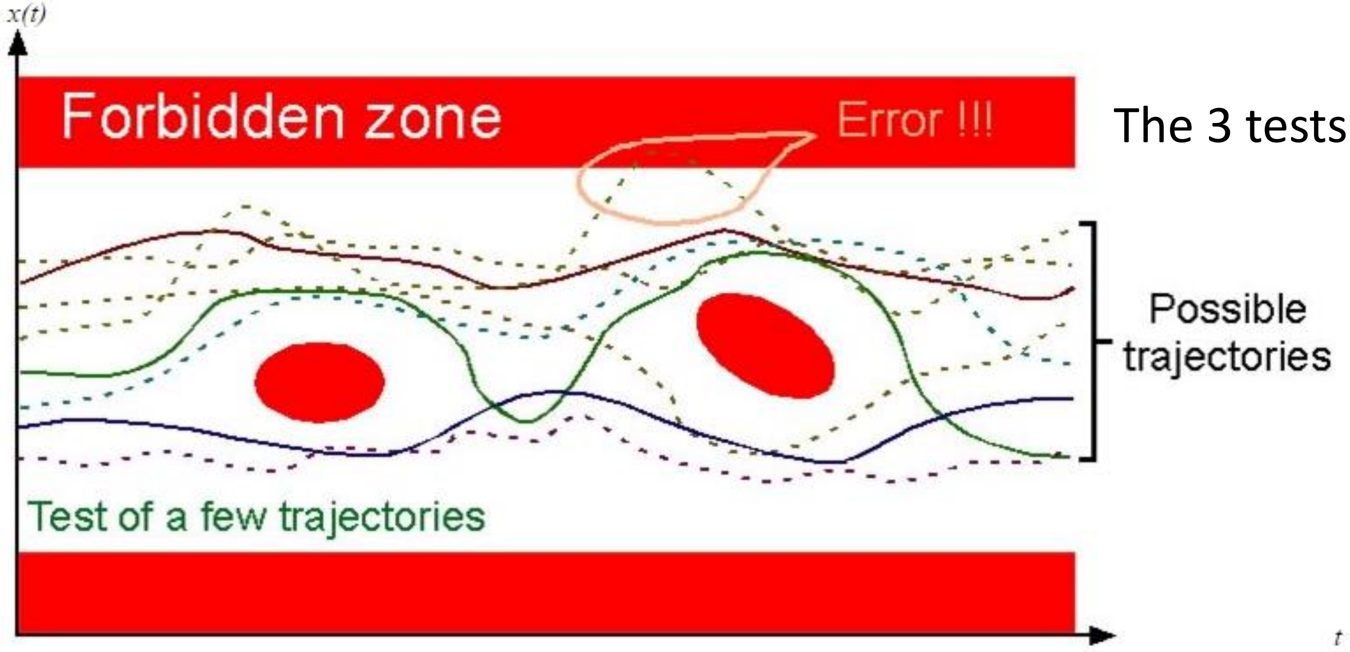
- Safety Properties



extracted from http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Program Abstraction (or Abstract Interpretation)

- Testing/Debugging



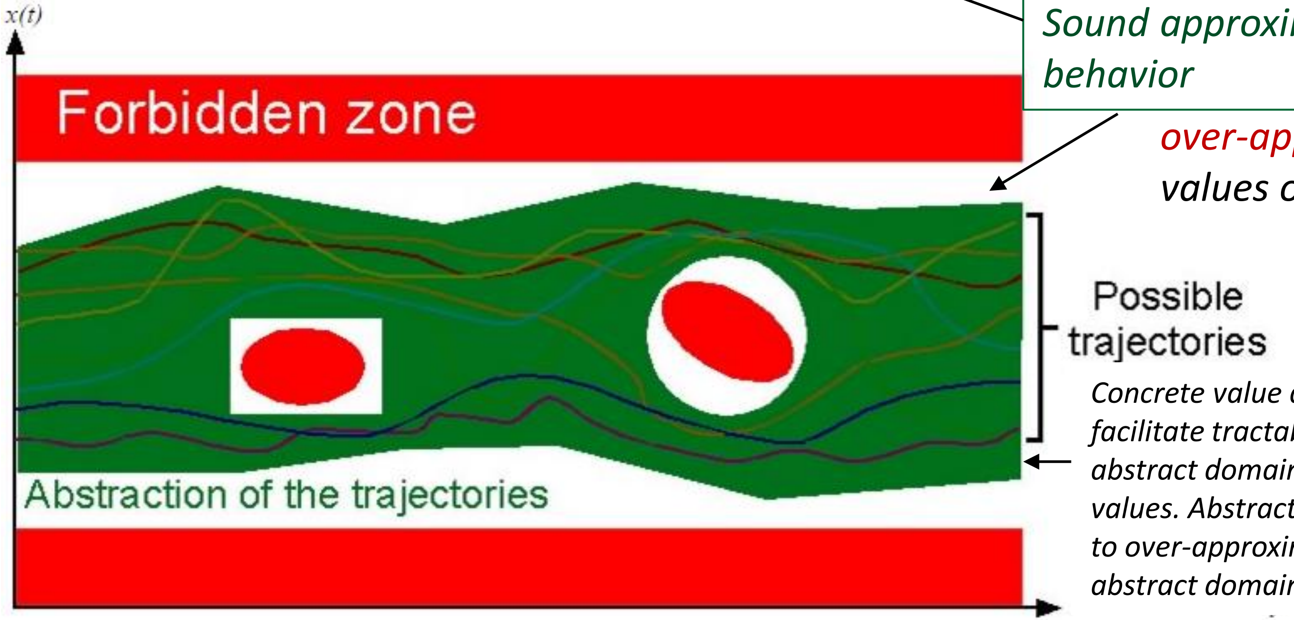
The 3 tests miss the bug!

extracted from http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Program Abstraction (or Abstract Interpretation)

- Sound Abstract Interpretation

Sound analysis ⇒
Sound approximation of a program's behavior



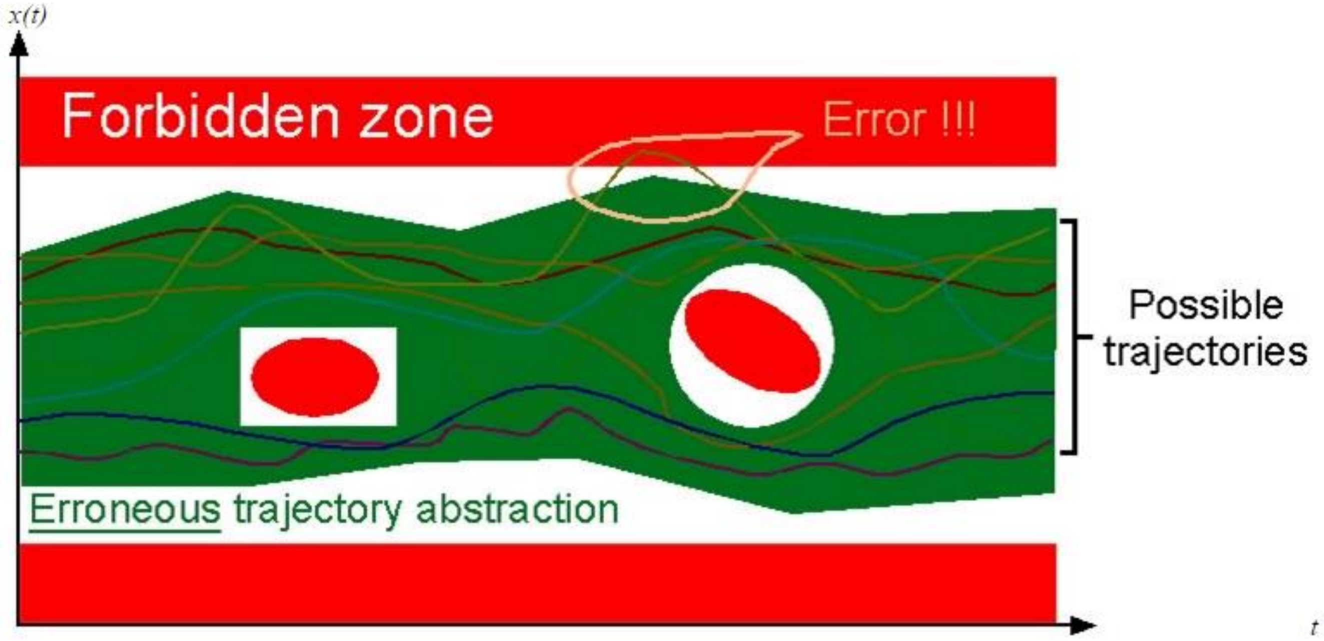
over-approximates possible values of x

Concrete value domain is often infinite. To facilitate tractable analysis, it is mapped to an abstract domain with finite number of abstract values. Abstract interpretation provides a theory to over-approximate program behavior under the abstract domain.

extracted from http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Program Abstraction (or Abstract Interpretation)

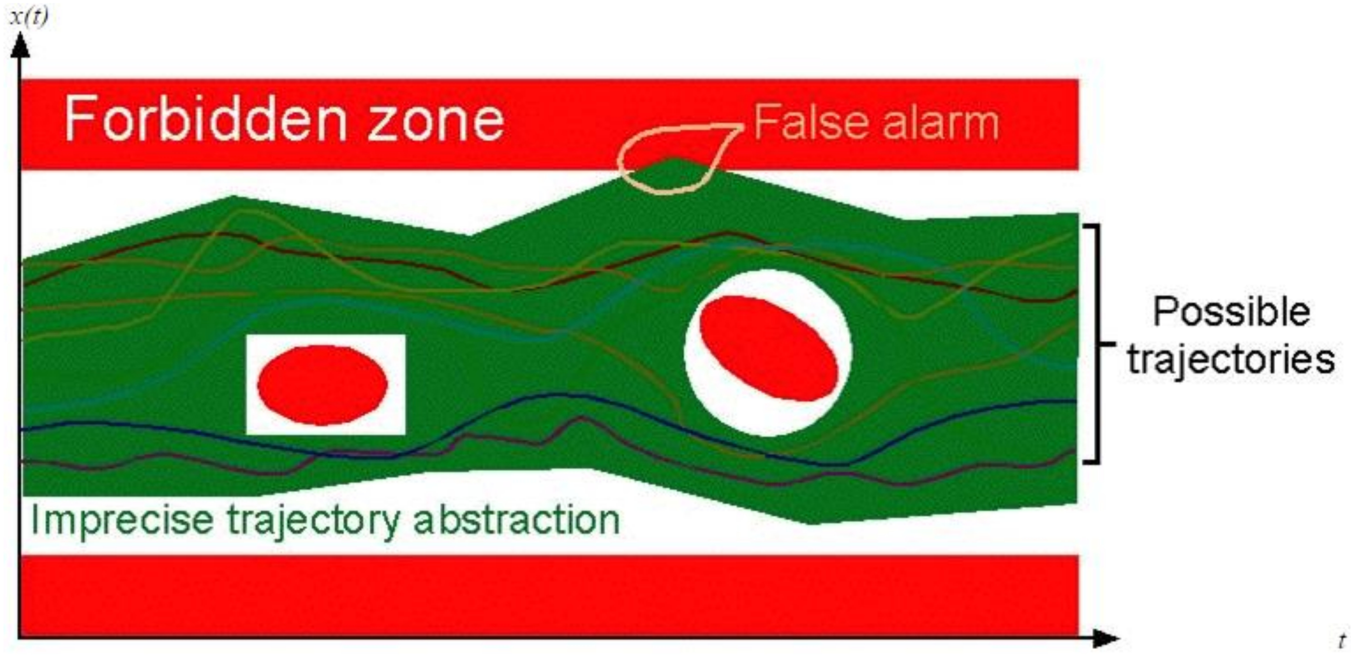
- Unsound Abstract Interpretation → false negatives



extracted from http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Program Abstraction (or Abstract Interpretation)

- Imprecise Abstract Interpretation → false positives



extracted from http://www.di.ens.fr/~cousot/AI/IntroAbsInt.html

Nature of Program Analysis for Testing and Verification

	Sound	**Complete**
Over-approximation Applicable to most static analysis: *Abstract Interpretation, Software Model Checking with Predicate Abstraction*	A program proven safe is actually safe (finds only real proofs)	Successfully proves safety of every safe program (finds all real proofs)
Under-approximation Applicable to most dynamic analysis: *Testing, Dynamic Symbolic Execution, Dynamic Software Model Checking*	A program reported unsafe is actually unsafe, i.e., no false positives (finds only real bugs)	Successfully reports all unsafe programs, i.e., no false negatives (finds all real bugs)

To avoid confusion, dynamic analysis nowadays often uses precision and recall to describe the nature of its results instead of soundness and completeness

Nature of Program Analysis for Testing and Verification

Alternative

	Sound	Complete
Desirable Analysis Applicable to most static analysis: *Abstract Interpretation, Software Model Checking with Predicate Abstraction*	A program proven safe is actually safe (finds only real proofs)	Successfully proves safety of every safe program (finds all real proofs)
Violation Analysis Applicable to most dynamic analysis: *Testing, Dynamic Symbolic Execution, Dynamic Software Model Checking*	A program reported unsafe is actually unsafe, i.e., no false positives (finds only real bugs)	Successfully reports all unsafe programs, i.e., no false negatives (finds all real bugs)

To avoid confusion, dynamic analysis nowadays often uses precision and recall to describe the nature of its results instead of soundness and completeness

Soundness, Completeness (Desirable Analysis)

Property	Definition (Premise: Is X true?)
Soundness	“**Sound for reporting a desirable property X**” It says X is true $\rightarrow$ X is true *It says P is safe $\rightarrow$ P is safe (from correctness perspective)* *or equivalently* *P violates X $\rightarrow$ It reports a warning (from error perspective)*
Completeness	“**Complete for reporting a desirable property X**” X is true $\rightarrow$ It says X is true *P is safe $\rightarrow$ It says P is safe (from correctness perspective)* *or equivalently* *It reports a warning $\rightarrow$ P violates X (from error perspective)*

Fact from logic: $A \rightarrow B$ is equivalent to $(\neg B) \rightarrow (\neg A)$

(for the desirable analysis of an error-free property)

	Complete	**Incomplete**
Sound	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Unsound	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

(for the violation analysis of errors)

	Sound	**Unsound**
Complete	Reports all errors Reports no false alarms **Undecidable**	Reports all errors May report false alarms **Decidable**
Incomplete	May not report all errors Reports no false alarms **Decidable**	May not report all errors May report false alarms **Decidable**

Basics

Program abstraction has two parts:

- Space abstraction: how program points and memories are abstracted
- Operation abstraction: how the program constructs (such as assignment “p = q;”) are processed

Space Abstraction

Program Point:

- Every statement *s* in the program has two program points:
  - the point before executing *s*
  - the point after executing *s*
- Unless otherwise specified, our discussion refers to the point after executing a statement

Space Abstraction

Able to distinguish one function call from another

```
public Object foo () {
    Object p1 = new Integer () ;   // o1
    Object q1 = new Integer () ;  // o2
    Object p2 = bar ( p1 ) ;         // c1
    Object q2 = bar ( q1 ) ;         // c2
}

public Object bar ( Object r ) {
    return r ;
}
```

Context Sensitivity:

- Function bar has two invocations, which creates two instances of r;
- If we distinguish the two invocations of bar with the callsite labels c1 and c2, we can distinguish the two instances of r by r^{c1} and r^{c2}.

Space Abstraction - Context Sensitive

- Whether different calling contexts are distinguished

```
void yellow()
{
1. red(1);
2. red(2);
3. green();
}
```

```
void red(int x)
{
..
}
```

```
void green()
{
  green();
  yellow();
}
```

Context sensitive distinguishes 2 different calls to red( )

Space Abstraction - Context Sensitive

```
a = id(4);          void id(int z)
                    { return z; }
b = id(5);
```

Context-Sensitive
(color denotes matching call/return)

Context sensitive analysis can tell one call returns 4, the other 5

```
a = id(4);          void id(int z)
                    { return z; }
b = id(5);
```

Context-Insensitive
(note: merging)

Context insensitive analysis will say both calls return {4, 5}

Space Abstraction – Context Sensitive

```
public Object foo () {
    Object p1 = new Integer () ;   // o1
    Object q1 = new Integer () ;   // o2
    Object p2 = bar ( p1 ) ;         // c1
    Object q2 = bar ( q1 ) ;        // c2
}

public Object bar ( Object r ) {
    return r ;
}
```

Context Sensitive:

- $pts(r^{c1}) = \{o1\}$, $pts(r^{c2}) = \{o2\}$
- $pts(p2) = \{o1\}$, $pts(q2) = \{o2\}$

Context insensitive:

- $pts(r) = \{o1, o2\}$
- $pts(p2) = \{o1, o2\}$
- $pts(q2) = \{o1, o2\}$

Space Abstraction – Field Sensitive

Field Sensitivity

- Distinguish fields in a class/structure
- In theory, field sensitivity is unsound for C programs and requires exponential time to complete

```
struct T {
    int *p, *q;
};
```

Space Abstraction – Field Sensitive

Example:

```
struct T {
    int *p, *q;
};
int main() {
    int &a, &b;
    struct T pt;
    pt.p = &a;
    pt.q = &b;
    return 0;
}
```

Field sensitive:

- **pts(pt.p) = {a};**
- **pts(pt.q) = {b};**

Field insensitive:

- **pts(pt.p) = {a, b};**
- **pts(pt.q) = {a, b};**

In field insensitive analysis, whatever assigned to a field are also assigned to other fields in the same structure.

Space Abstraction – Field Sensitive

- Field sensitivity for C is unsound because C permits access to a field via pointer arithmetic.

```
struct T { int *p, *q; };

int main() {
    int offset;
    struct T* pt = malloc(…);
    scanf( “%d”, &offset);
    pt + offset = malloc(…);
    return 0;
}
```

We cannot determine at compile time the value of “pt+offset”.

Therefore, we can only assume both pt->p and pt->q point to the same allocated memory, which is essentially the field insensitive treatment.

Space Abstraction – Types

Type information:

- C is a weakly typed language that we cannot say the pointers declared "int*" only store the addresses of integer variables.
- Ignoring types may produce many large points-to sets (e.g., size > 500).
- Java is strongly typed. We can use the type information to remove spurious points-to results.
- This explains why Java points-to analysis is much more precise.

Basics

Program abstraction has two parts:

- Space abstraction: how program points and memories are abstracted
- Operation abstraction: how the program constructs (such as assignment “p = q;”) are processed

Flow Sensitive

- A *flow* sensitive analysis considers the order (flow) of statements
  - Flow insensitive = usually linear-type algorithm
  - Flow sensitive = usually at least quadratic (dataflow)
- Examples:
  - Type checking is flow insensitive since a variable has a single type regardless of the order of statements
  - Detecting uninitialized variables requires flow sensitivity

```
   x = 4;
6. ....
   x = 5;
```

Flow sensitive analysis distinguishes values of x before and after line 6, flow insensitive analysis cannot.

Flow Sensitive Example

```
1. x = 4;
....
9. x = 5;
```

Flow sensitive:
x is constant 4 at line 1,
x is constant 5 at line 9

Flow insensitive:
x is not a constant

Handling Program Constructs

Flow Sensitivity:

- Analyze program along the Control Flow Graph (CFG).
  - For example, if the programmer writes two statements:
    “a=1; b=2;”,
    we analyze “a=1” before considering the effects of “b=2”.
  - We associate analysis result to every program point.

Handling Program Constructs

Solution for each program point

Flow Sensitive:

```
p = &a;               // pts(p) = {a}, pts(q) = Φ
q = &b;               // pts(p) = {a}, pts(q) = {b}
if ( t > 0 )
    p = &c;           // pts(p) = {c}, pts(q) = {b}
else
    q = &d;           // pts(p) = {a}, pts(q) = {d}
*p = *q;              // pts(p) = {b, d}, pts(q) = {b, d}
```

Handling Program Constructs

Flow Insensitive:

- Does not analyze the program statements in their appearance order.
  - We can view flow insensitivity as a special case of flow sensitivity, where CFG is a complete digraph (i.e., there is a directed edge between any two statements).
- A single solution for the whole program is given. We don’t associate results to every program point.

Handling Program Constructs

Flow Insensitive:

```
p = &a;
q = &b;
if ( t > 0 )
    p = &c;
else
    q = &d;
*p = *q;
```

Single solution for all program points

pts(p) = {a, b, c, d}

pts(q) = {b, d}

Unordered: any statement can be executed after another

Handling Program Constructs

Path Sensitivity:

- A path sensitive analysis maintains branch conditions along each ***execution path***
  - Requires extreme care to make the analysis scalable
  - Subsumes flow sensitivity

Path Sensitive Example

```
1. if(x >= 0)
2.    y = x;
3. else
4.    y = -x;
```

path sensitive:
y >= 0 at line 2,
y > 0 at line 4

path insensitive:
y is not a constant

Path insensitive analysis ignores the predicate in if-condition

Precision

Path sensitive analysis approximates behavior due to:

- loops/recursion
- unrealizable paths

```
1. if(aⁿ + bⁿ == cⁿ && n>2 && a>0 && b>0 && c>0)
2.    x = 7;
3. else
4.    x = 8;
```

Unrealizable path.
x will always be 8

Handling Pointer Assignment: q = p

- Two categories of algorithms depend on how to handle the pointer assignment: q = p
- **Andersen's analysis: $pts(p) \subseteq pts(q)$**
  - Explanation: whatever p points-to would also be pointed by q
  - Complexity: $O(n^3)$, n is the number of pointers
- **Steensgard's analysis: pts(p) = pts(q)** ← Over-approximation of p
  - Explanation: p and q point to the same set of variables
  - Complexity: O(n*a(n)), a is the inverse Ackerman's function

Andersen's Analysis

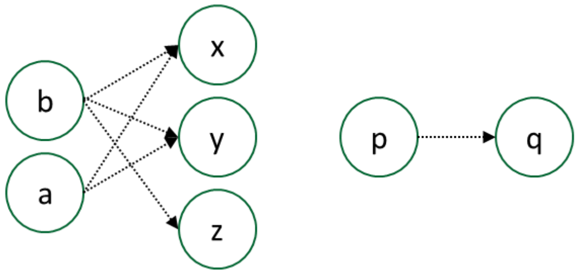
- Andersen's analysis is the most precise pointer analysis algorithm in the context insensitive, flow insensitive spectrum
- Steensgard's is the most imprecise one in the spectrum. Steensgard's is orders of magnitude faster than Andersen's
- Other algorithms with precision and performance in between
- Read the following two papers if you are interested.
  - PLDI 2000, Das, Unification-based Pointer Analysis with Directional Assignments
  - POPL 1997, Shapiro, Fast and accurate flow-insensitive points-to analysis

Andersen's Analysis

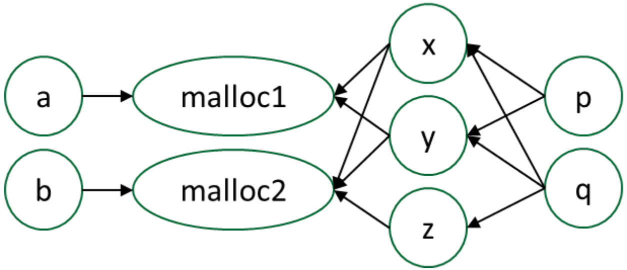
Data Structures:

- The Pointer Assignment Graph (PAG): the nodes in the graph represent the pointers with one-to-one corresponding. The directional edge, e.g., p→q, means that pts(p) ⊆ pts(q).
- The Points-to Graph: the nodes represent the pointers and the memory locations. The edges p→x represents p points to x.

Final Pointer Assignment Graph:



Final Points-to Graph:



Andersen's Analysis

- Evaluation rules for different constraints (statements):

Constraint Type	Symbolic Form	Evaluation Rule
Base	u = &e	$pts(u) = pts(u) \cup \{e\}$
Simple	u = v	$pts(u) = pts(u) \cup pts(v)$
Store	*(u+c) = v	$\forall e \in pts(u), pts(e) = pts(e) \cup pts(v)$
Load	u = *(v+c)	$\forall e \in pts(v), pts(u) = pts(u) \cup pts(e)$

Over-approximation

1. c is a constant
2. The store and load constraints are also called complex constraints.

Andersen’s Analysis

- Evaluation rules for different constraints (statements):

Constraint Type	Symbolic Form	Evaluation Rule
Base	u = &e	pts(u) = pts(u) ∪ {e}
Simple	u = v	pts(u) = pts(u) ∪ pts(v)
Store	*(u+c) = v	∀ e ∈ pts(u), pts(e) = pts(e) ∪ pts(v)
Load	u = *(v+c)	∀ e ∈ pts(v), pts(u) = pts(u) ∪ pts(e)

Over-approximation

Questions:

1. Why do we only consider these four types of constraints?
2. Is the analysis field sensitive?

Andersen’s Analysis

- Q: Why do we only consider these four types of constraints?
- A: Complex constraints are a combination of the four basic statements.
- Example:

Constraint Type	Symbolic Form
Base	u = &x
Simple	u = v
Store	*(u+c) = v
Load	u = *(v+c)

```
**p = (*q)->f;
```

$\xrightarrow{\textit{transform}}$

```
a = *q;
b = *(a+f);
c = *p;
*c = b;
```

Andersen’s Analysis

- Q: Is the analysis field sensitive?
- A: No, it is field insensitive because, when we process *(u+c)=v and p=*(q+c), we ignore the offset c.

Field insensitive rules:

Constraint Type	Symbolic Form	Evaluation Rule
Store	*(u+c) = v	$\forall$ e $\in$ pts(u), pts(e) = pts(e) $\cup$ pts(v)
Load	u = *(v+c)	$\forall$ e $\in$ pts(v), pts(u) = pts(u) $\cup$ pts(x)

u+c is the abstract variable for field c.

Andersen's Analysis

Algorithm:

- Extract all the pointer relevant statements from the given program;
- Apply the four evaluations to these statements (or constraints) until the points-to results unchanged.

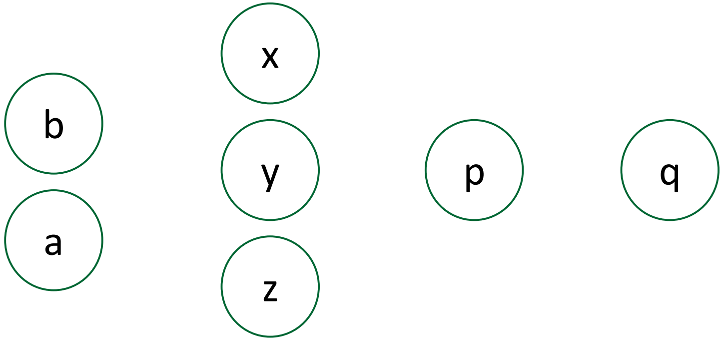
Andersen's Analysis

- Example:

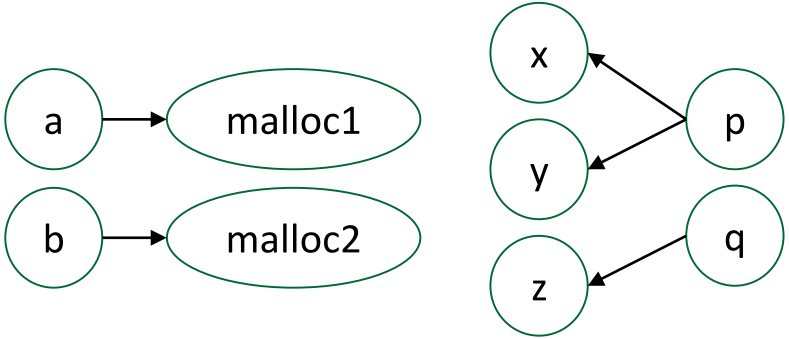
```
p = &x;
q = p;
p = &y;
q = &z;
*p = a;
*q = b;
a = malloc 1;
b = malloc 2;
```

Constraint Type	Symbolic Form	Evaluation Rule
Base	u = &e	pts(u) = pts(u) ∪ {e}
Simple	u = v	pts(u) = pts(u) ∪ pts(v)
Store	*(u+c) = v	∀ e ∈ pts(u), pts(e) = pts(e) ∪ pts(v)
Load	u = *(v+c)	∀ e ∈ pts(v), pts(u) = pts(u) ∪ pts(e)

Initial Pointer Assignment Graph:



Initial Points-to Graph:

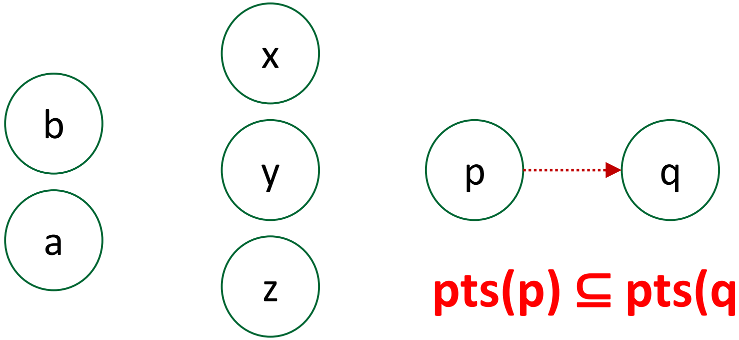


Andersen's Analysis

- Evaluate q = p:

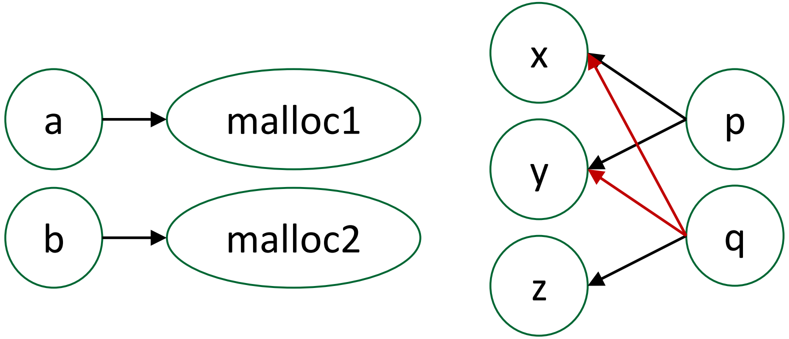
```
p = &x;
q = p;
p = &y;
q = &z;
*p = a;
*q = b;
a = malloc 1;
b = malloc 2;
```

Updated Pointer Assignment Graph:



pts(p) ⊆ pts(q)

Updated Points-to Graph:



Constraint Type	Symbolic Form	Evaluation Rule
Base	u = &e	$pts(u) = pts(u) \cup \{e\}$
Simple	u = v	$pts(u) = pts(u) \cup pts(v)$
Store	*(u+c) = v	$\forall e \in pts(u), pts(e) = pts(e) \cup pts(v)$
Load	u = *(v+c)	$\forall e \in pts(v), pts(u) = pts(u) \cup pts(e)$

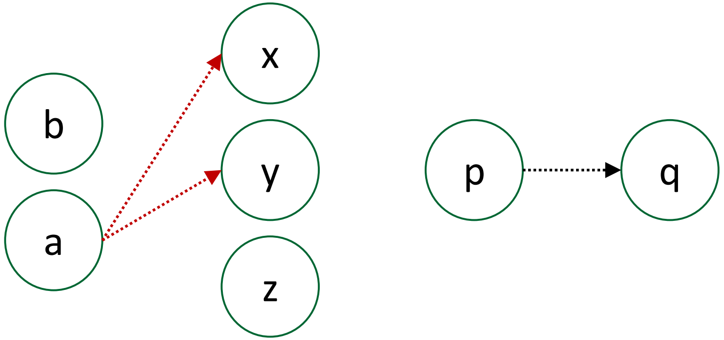
Andersen’s Analysis

- Evaluate *p = a:

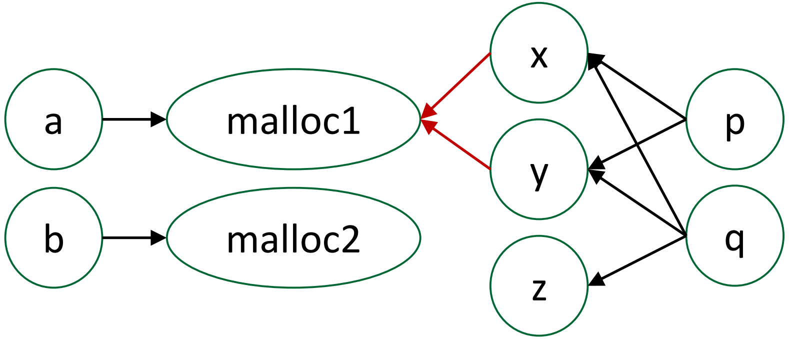
  p = &x;
  q = p;
  p = &y;
  q = &z;
  ☞ *p = a;
  *q = b;
  a = malloc 1;
  b = malloc 2;

pts{a} ⊆ pts(x)
pts{a} ⊆ pts(y)

Updated Pointer Assignment Graph:



Updated Points-to Graph:



Constraint Type	Symbolic Form	Evaluation Rule
Base	u = &e	pts(u) = pts(u) ∪ {e}
Simple	u = v	pts(u) = pts(u) ∪ pts(v)
☞ Store	*(u+c) = v	∀ e ∈ pts(u), pts(e) = pts(e) ∪ pts(v)
Load	u = *(v+c)	∀ e ∈ pts(v), pts(u) = pts(u) ∪ pts(e)

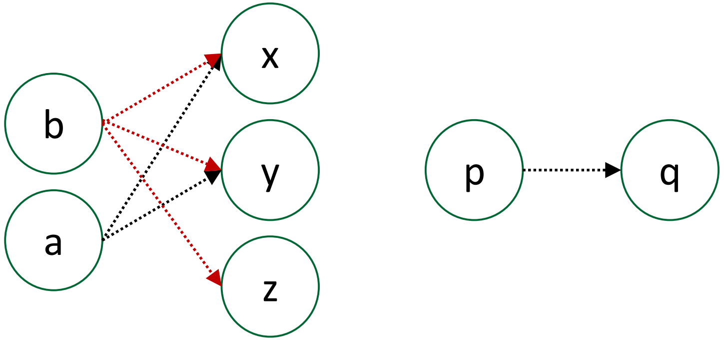
Andersen's Analysis

- Evaluate *q = b:

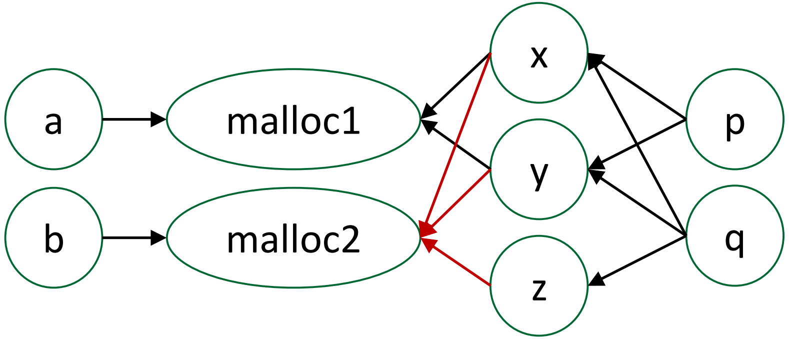
```
p = &x;
q = p;
p = &y;
q = &z;
*p = a;
*q = b;
a = malloc 1;
b = malloc 2;
```

Constraint Type	Symbolic Form	Evaluation Rule
Base	u = &e	pts(u) = pts(u) ∪ {e}
Simple	u = v	pts(u) = pts(u) ∪ pts(v)
Store	*(u+c) = v	∀ e ∈ pts(u), pts(e) = pts(e) ∪ pts(v)
Load	u = *(v+c)	∀ e ∈ pts(v), pts(u) = pts(u) ∪ pts(e)

Updated Pointer Assignment Graph:



Updated Points-to Graph:

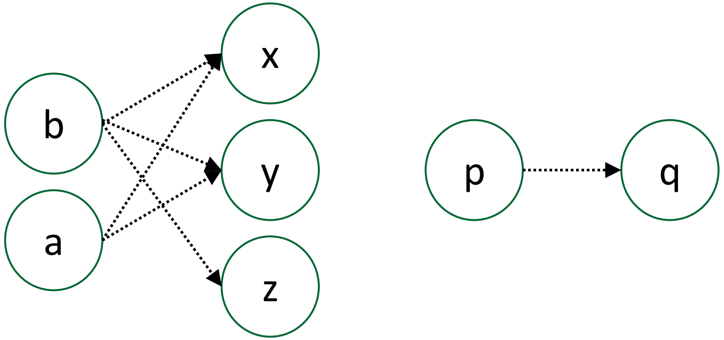


Andersen’s Analysis

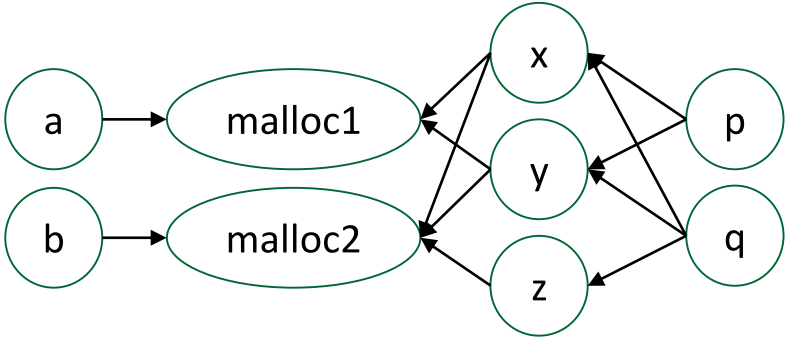
- The final result is irrelevant to the evaluation order of the statements. You can try other orders and will get the same result.

```
p = &x;
q = p;
p = &y;
q = &z;
*p = a;
*q = b;
a = malloc 1;
b = malloc 2;
```

Final Pointer Assignment Graph:



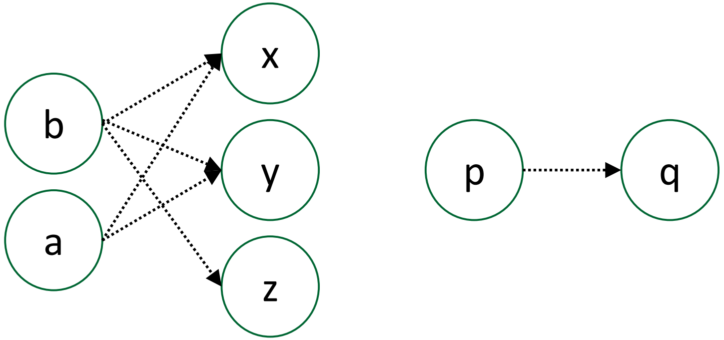
Final Points-to Graph:



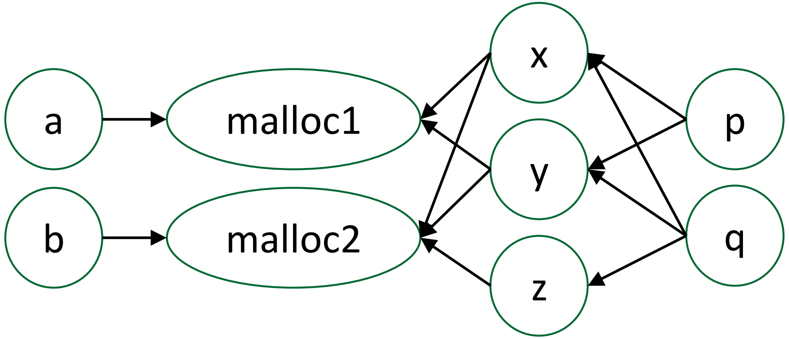
Andersen's Analysis

- The complexity is $O(n^3)$, where n is the number of pointers, and we have $O(n)$ statements. This is because we examine in each iteration $O(n)$ statements, and in the worst case we have $O(n^2)$ iterations.
- Recent work observes: Close to $O(n^2)$ if:
  - Few statements dereference each variable
  - Control flow graphs not too complex
  - Both observations are common in practice

Final Pointer Assignment Graph:



Final Points-to Graph:



Abstract Interpretation

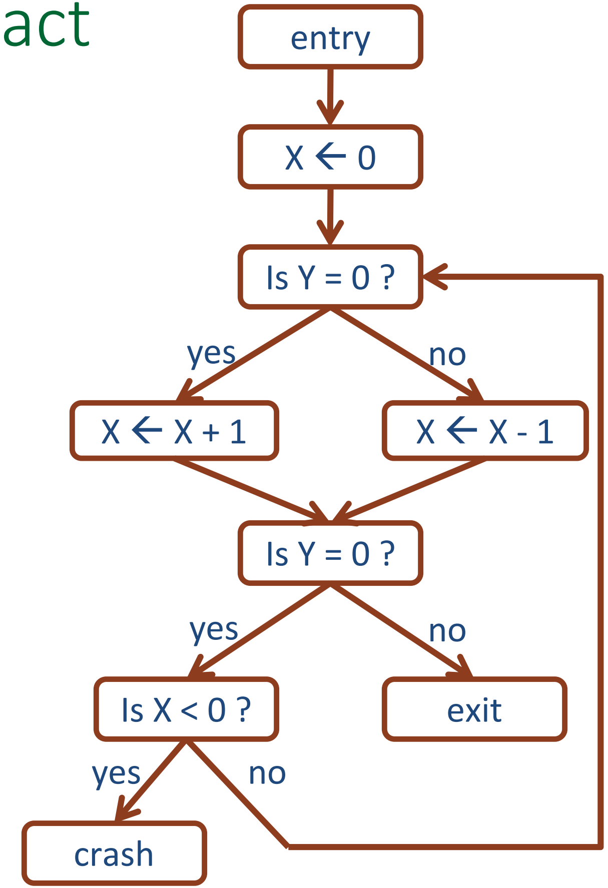
Illustration: Static analysis based on state abstraction

Why Abstract Interpretation?

- Reduce an intractable/undecidable analysis to a tractable/decidable analysis
- Procedures
  - Abstract a large, possibly infinite value space (**concrete set**) using a small finite value space (**abstract set**)
  - Approximate computation over the concrete set using computation over the abstract set
  - Interpret the program semantics based on the abstracted computation results at a fixed point

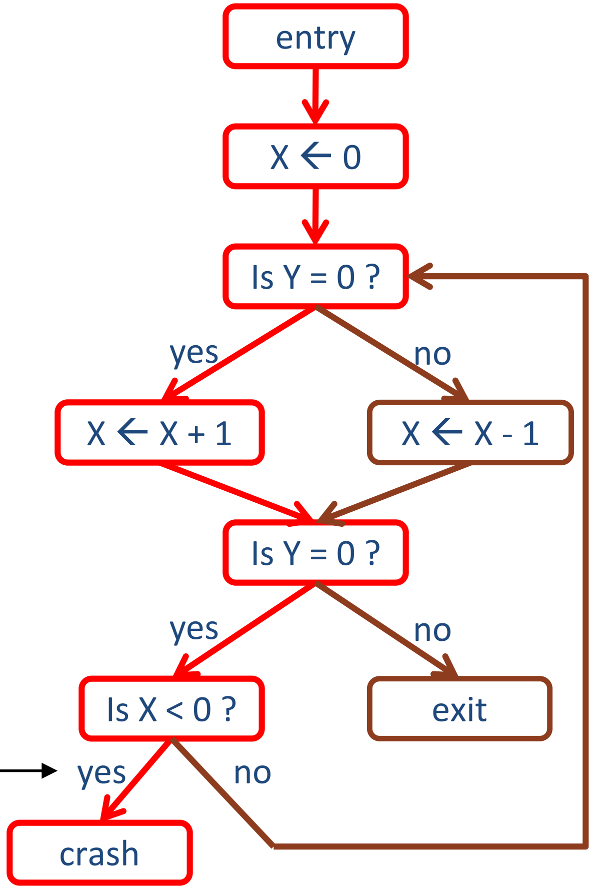
Note: To make the analysis sound after abstraction, the abstraction over-approximates the original behavior

Example – Abstract State Analysis



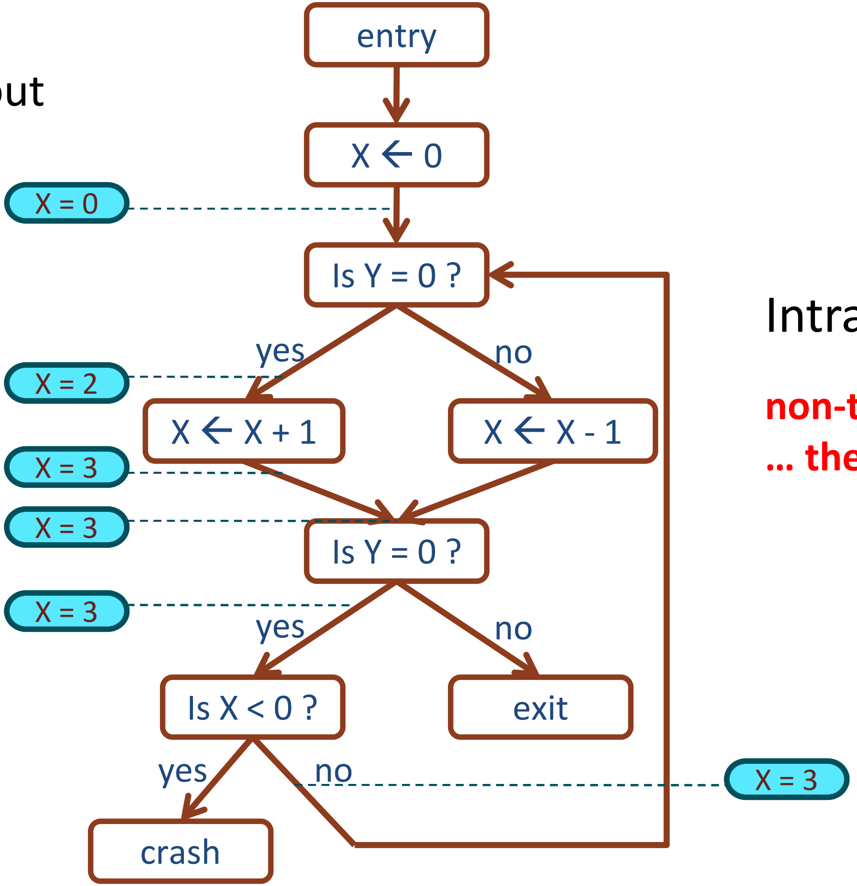
Does this program ever crash?

Does this program ever crash?



infeasible path!
… program will never crash

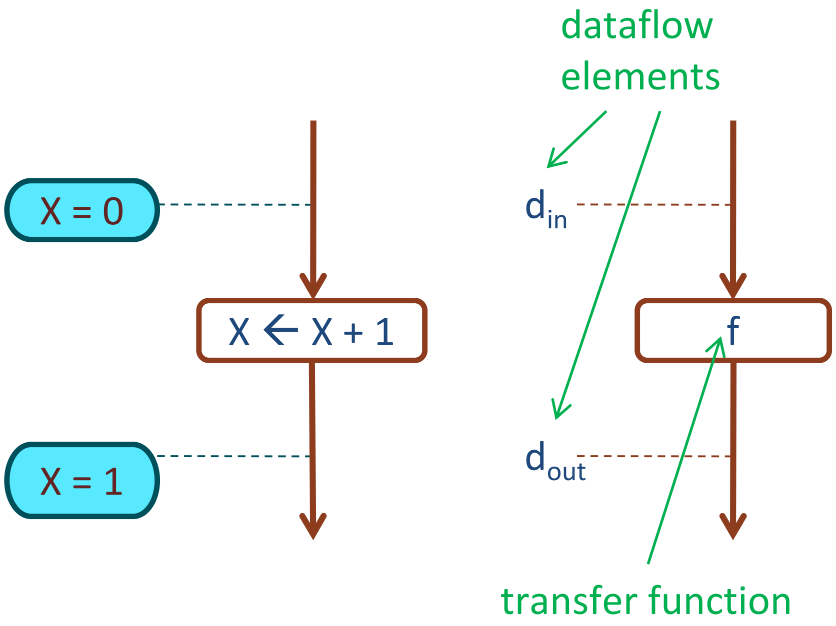
Try analyzing without approximating…



Intractable!

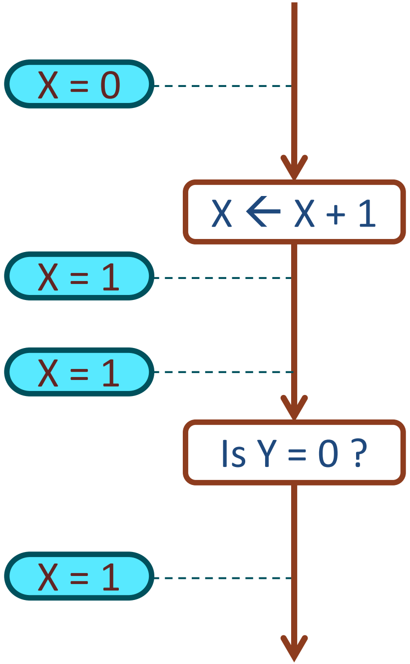
non-termination!

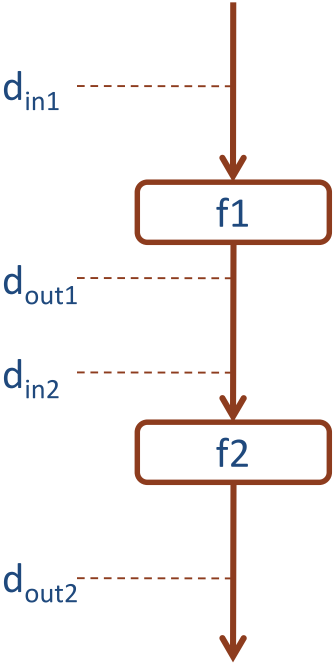
… therefore, need to approximate



$$d_{out} = f(d_{in})$$

dataflow equation

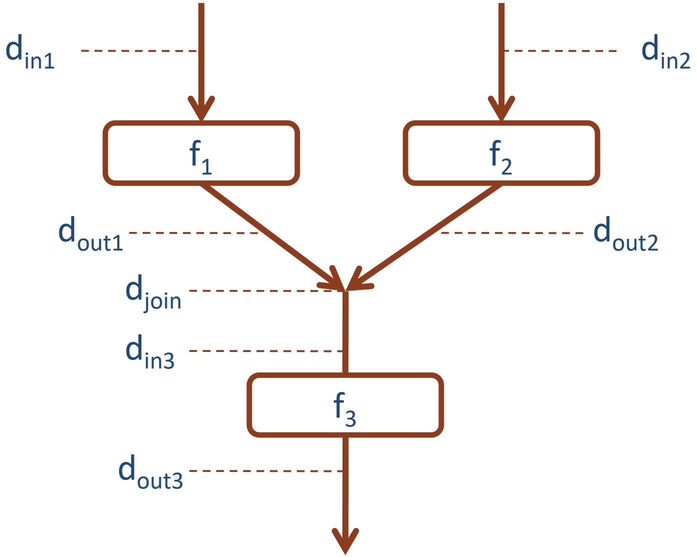




$$d_{out1} = f_1(d_{in1})$$

$$d_{in2} = d_{out1}$$

$$d_{out2} = f_2(d_{in2})$$



$$d_{out1} = f_1(d_{in1})$$

$$d_{out2} = f_2(d_{in2})$$

$$\mathbf{d_{join} = d_{out1} \sqcup d_{out2}}$$

$$d_{in3} = d_{join}$$

$$d_{out3} = f_3(d_{in3})$$

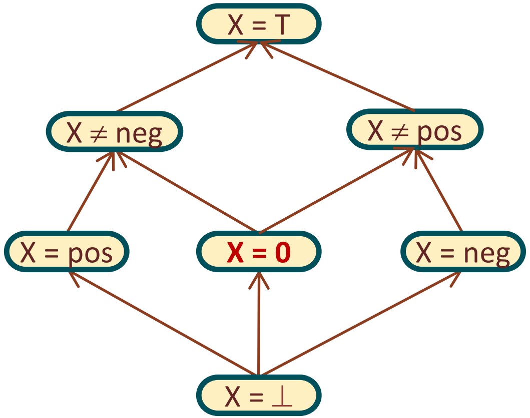
Source of precision loss; Need to design its semantics carefully!

least upper bound operator
Example: union of possible values

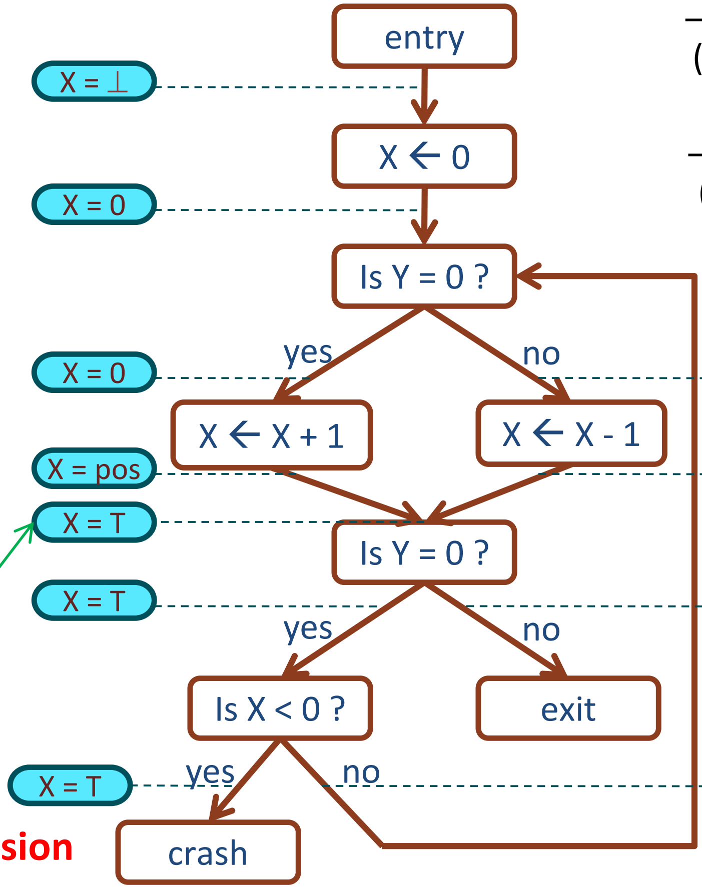
Need to answer two questions:

- **What is the space of dataflow elements, Δ?**
- **What is the least upper bound operator, ⊔?**

Abstract the integer value set using a sign value lattice …



lose precision



$$\frac{U = 0}{(V \leftarrow U + 1) \vdash V = pos}$$

$$\frac{U = 0}{(V \leftarrow U - 1) \vdash V = neg}$$

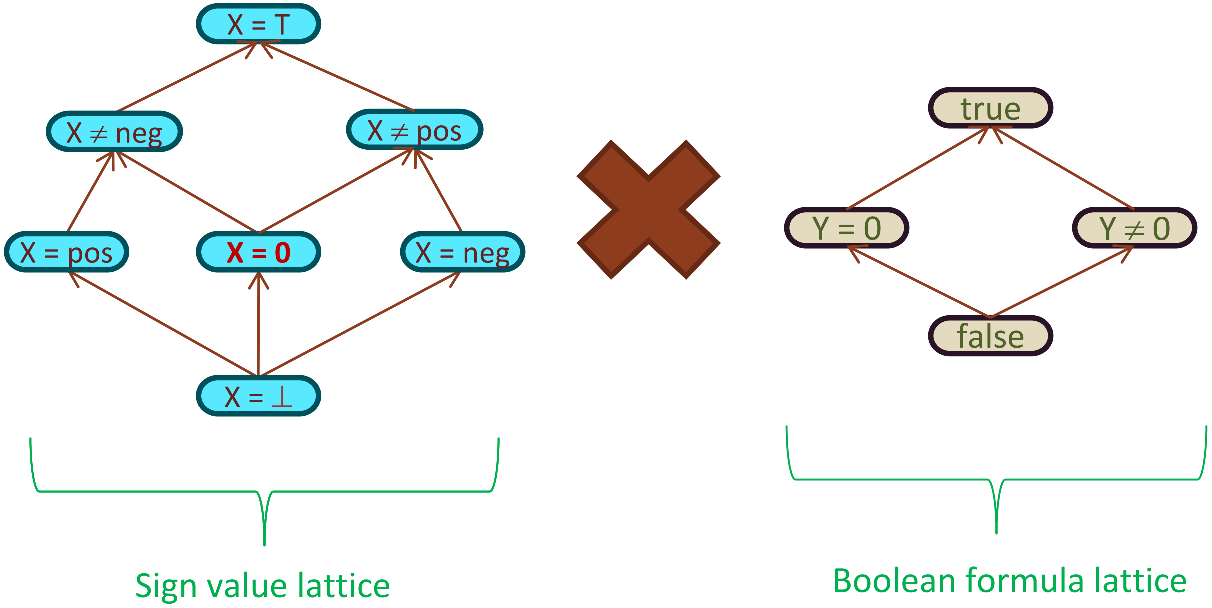
Abstracted computation

terminates...

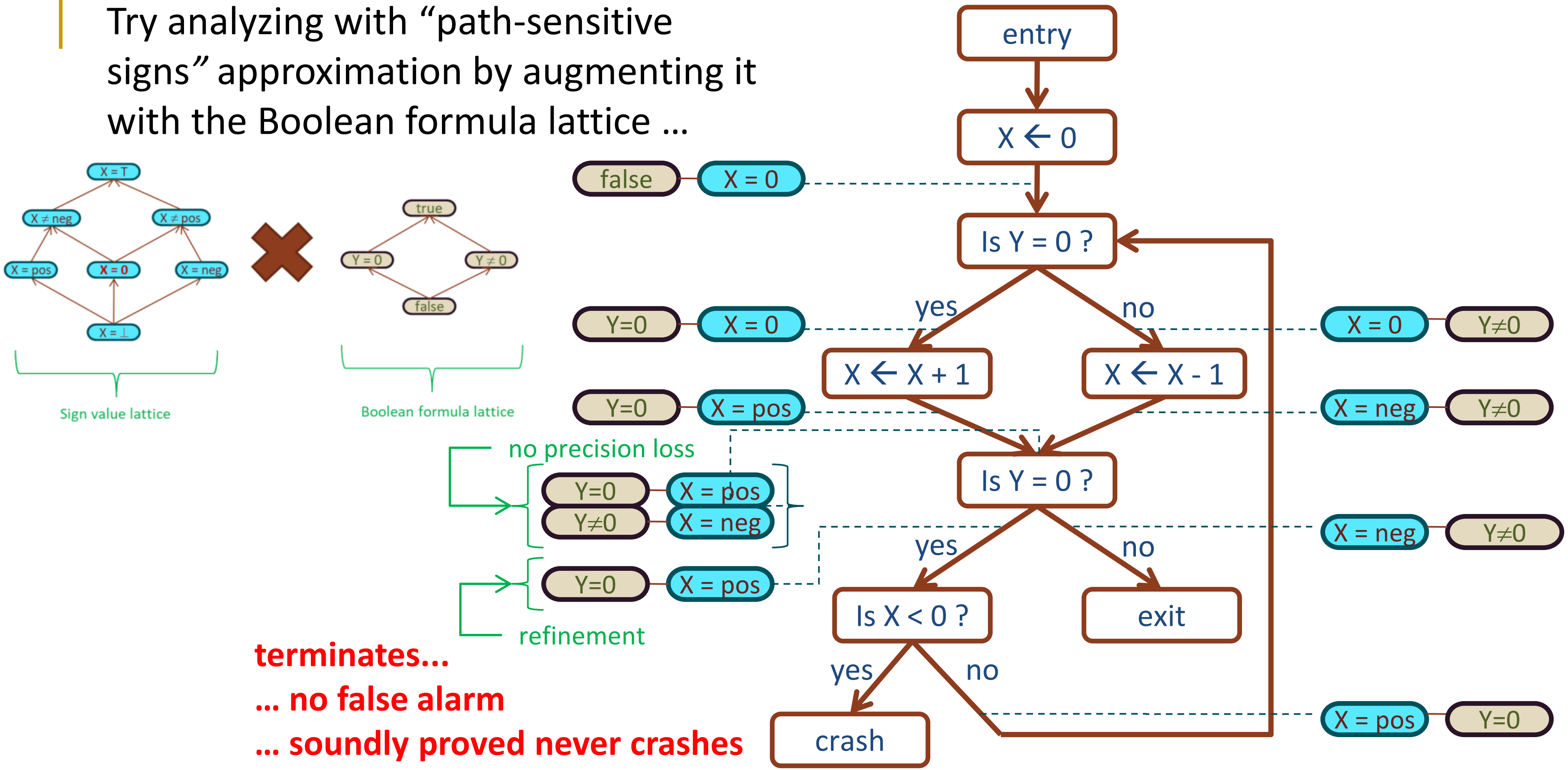
… but reports false alarm

… therefore, need more precision

Refined Lattice



Try analyzing with “path-sensitive signs” approximation by augmenting it with the Boolean formula lattice …



terminates...

… no false alarm

… soundly proved never crashes