Static Analysis

Slides adapted from Claire Le Goues, Jonathan Aldrich, Phillip Gibbons

Testing approaches

```
void function(uint8_t* ptr, int x) {
  int n = x + 50;
  ...
  for (int i = 0; i < n; i++) {
      *(ptr + i) = i;
  }
  ...
}

uint8_t arr[128];
function(str, 128);
```

How to determine if this piece of code has a bug?

- Dynamic execution
- Symbolic execution
- **Static Analysis**

What is static analysis

- Static program analysis aims at discovering semantic properties or runtime behavior of programs without running them

- Key ideas
  - Abstraction
    - Capture semantically relevant details and elide non-relevant details
  - Treat programs as data:
    - Programs are just graphs (CFG, callgraph)

Defects Static Analysis can Catch

- Defects that result from inconsistently following simple, mechanical design rules.
  - Security: Buffer overruns, improperly validated input.
  - Memory safety: Null dereference, uninitialized data.
  - Resource leaks: Memory, OS resources.
  - API Protocols: Device drivers; real time libraries; GUI frameworks.
  - Exceptions: Arithmetic/library/user-defined
  - Encapsulation: Accessing internal data, calling private functions.
  - Data races: Two threads access the same data without synchronization

Static Analysis Example

- Consider the following program:

```
x = 10;
y = x;
z = 0;
while (y > -1) {
  x = x / y;
  y = y - 1;
  z = 5;
}
p = 100/x;
```

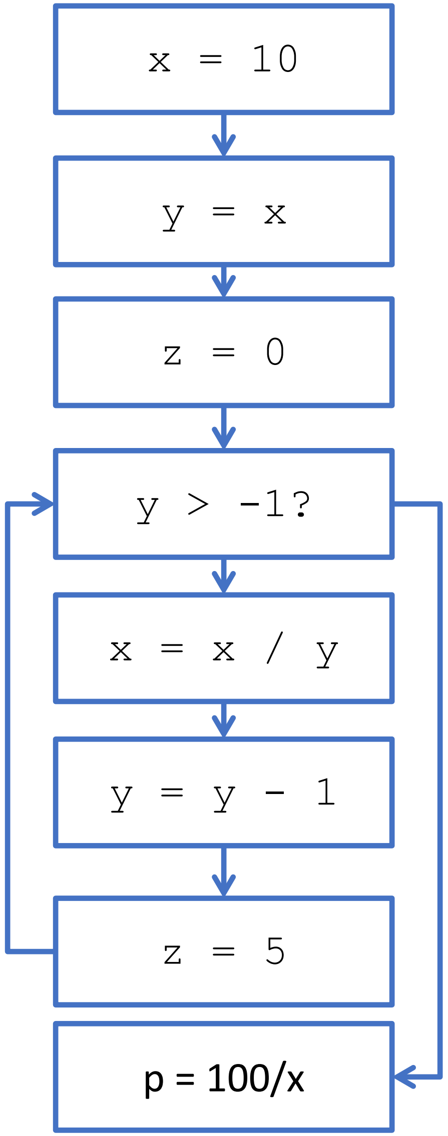
Can x ever be zero, leading to DIV by zero error?

- Use static analysis to determine if x is ever 0, without running the program
  - Semantic property of interest: Zero / Non-Zero

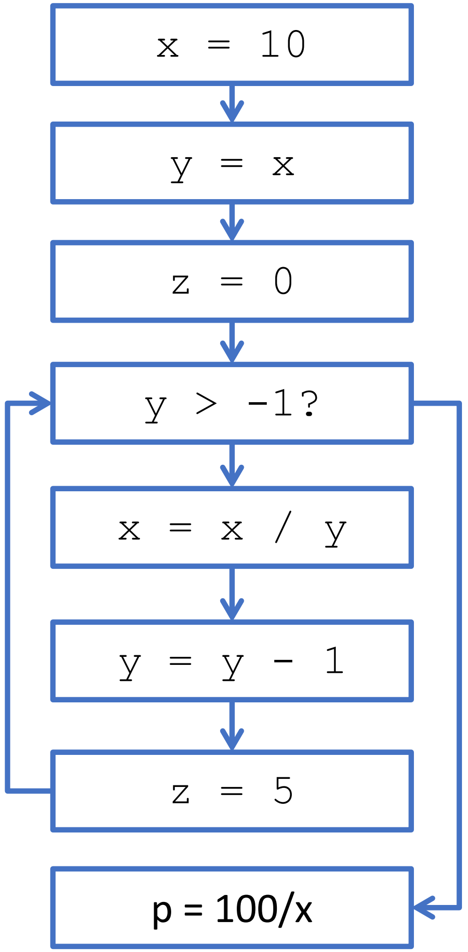
Applying Zero Analysis

Step 1: Convert program to a control flow graph

```
x = 10;
y = x;
z = 0;
while (y > -1) {
  x = x / y;
  y = y - 1;
  z = 5;
}
p = 100/x;
```



Applying Zero Analysis



Step 1: Apply abstraction: Zero/Non-Zero/Maybe-Zero

We don't care about the actual values (real or symbolic) of x, y, and z

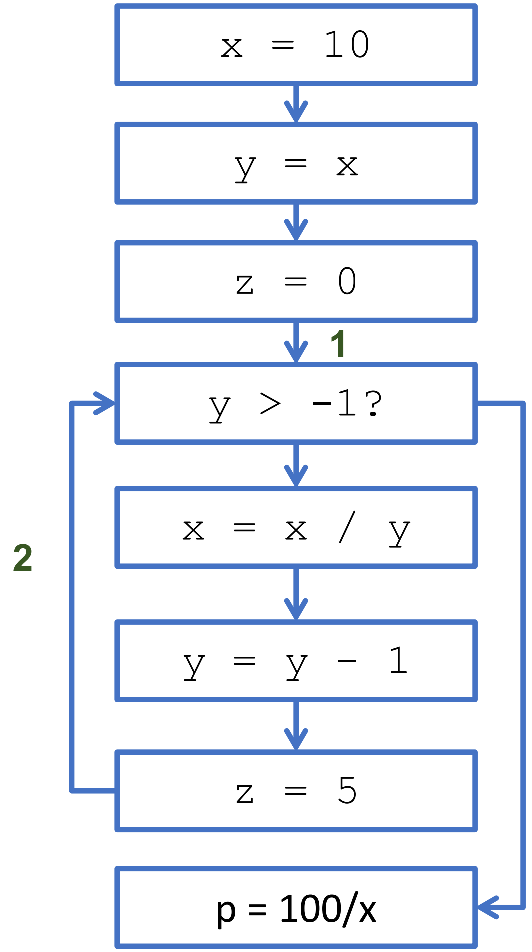
Operation	Operand 1	Operand 2	Result
+	Z	Z	Z
+	Z	NZ	NZ
+	NZ	NZ	NZ
+	NZ	Z	NZ
-	Z	Z	Z
-	Z	NZ	NZ
-	NZ	Z	NZ
-	NZ	NZ	**MZ**

By abstracting the actual values, we have simplified the analysis but lost precision

Applying Zero Analysis

Step 1: Apply abstraction: Zero/Non-Zero/Maybe-Zero

We don't care about the actual values of x, y, and z



Iteration 1	Iteration 2	Iteration 3
x:NZ		
x:NZ, y:NZ		
x:NZ, y:NZ, z:Z		
x:NZ, y:NZ, z:Z	x:NZ, y:MZ, z:MZ	x:NZ, y:MZ, z:MZ
x:NZ, y:NZ, z:Z	x:NZ, y:MZ, z:MZ	x:NZ, y:MZ, z:MZ
x:NZ, y:MZ, z:Z	x:NZ, y:MZ, z:MZ	x:NZ, y:MZ, z:MZ
x:NZ, y:MZ, z:NZ	x:NZ, y:MZ, z:NZ	x:NZ, y:MZ, z:NZ
x:NZ, y:MZ, z:NZ	x:NZ, y:MZ, z:NZ	x:NZ, y:MZ, z:NZ

Termination

- Analysis values will not change, no matter how many times loop executes
  - Proof: our analysis is deterministic
    - We run through the loop with the current analysis values, none of them change. Therefore, no matter how many times we run the loop, the results will remain the same
  - Therefore, we have computed the zero analysis results for any number of loop iterations
- Important: We don't care about how many iterations the loop makes

Example final result: `x:NZ, y:MZ, z:MZ`

Abstraction at Work

- Number of possible states gigantic
  - n 32 bit variables results in 2^{32*n} states
    - $2^{(32*3)} = 2^{96}$
  - With loops, states can change indefinitely
- Zero Analysis narrows the state space
  - Zero or not zero
  - $2^{(2*3)} = 2^{6}$
  - When this limited space is explored, then we are done
    - Extrapolate over all loop iterations
- Improves scalability but lose precision (Maybe-Zero)

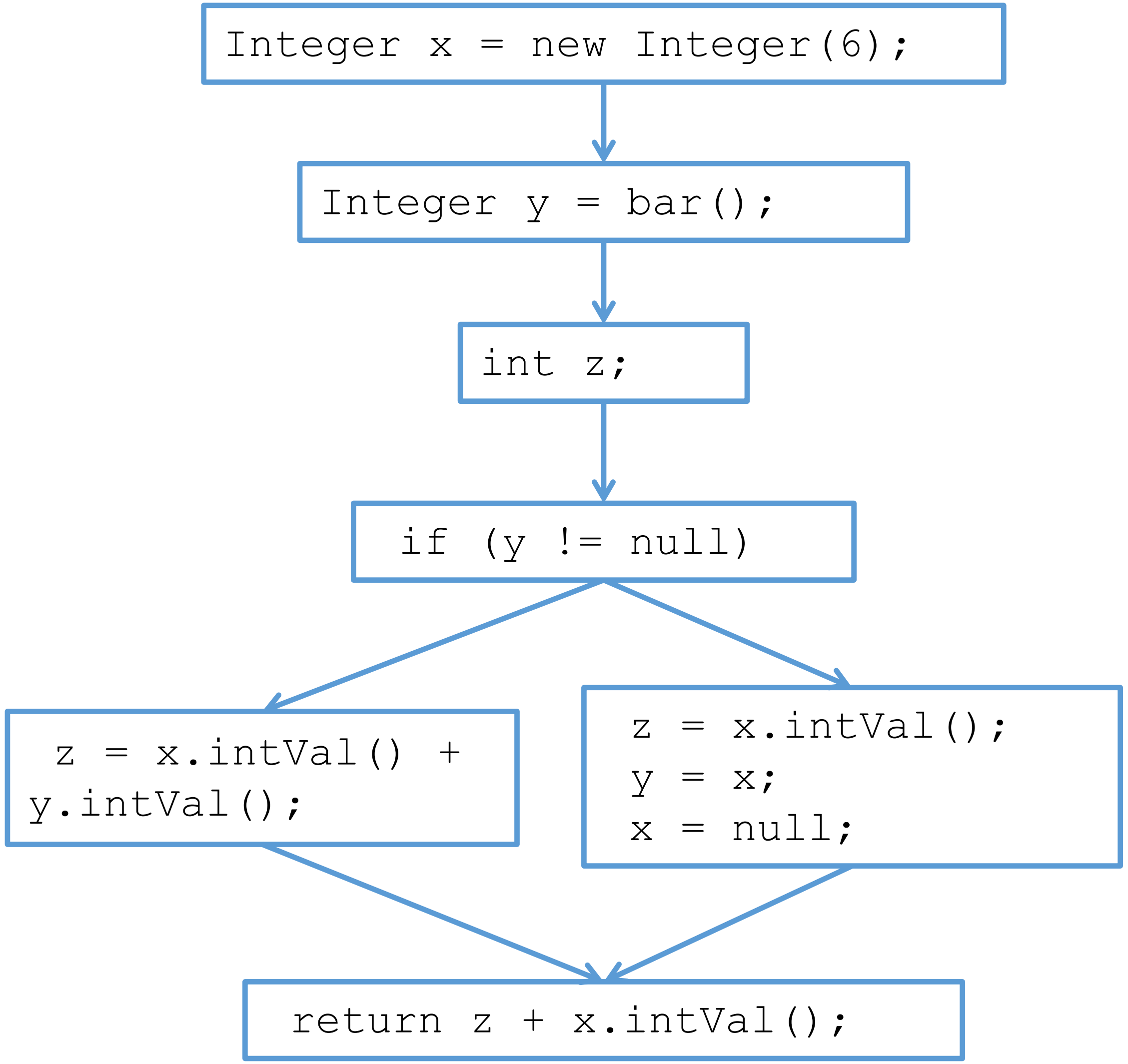
Exercise

```
int foo() {
    Integer x = new Integer(6);
    Integer y = bar(); // external
                       // library
    int z;
    if (y != null)
        z = x.intVal() + y.intVal();
    else {
        z = x.intVal();
        y = x;
        x = null;
    }
    return z + x.intVal();
}
```

Are there any possible ***null pointer exceptions*** *in this code?*

Control flow graph

```
int foo() {
    Integer x = new Integer(6);
    Integer y = bar(); // external
                       // library
    int z;
    if (y != null)
        z = x.intVal() + y.intVal();
    else {
        z = x.intVal();
        y = x;
        x = null;
    }
    return z + x.intVal();
}
```



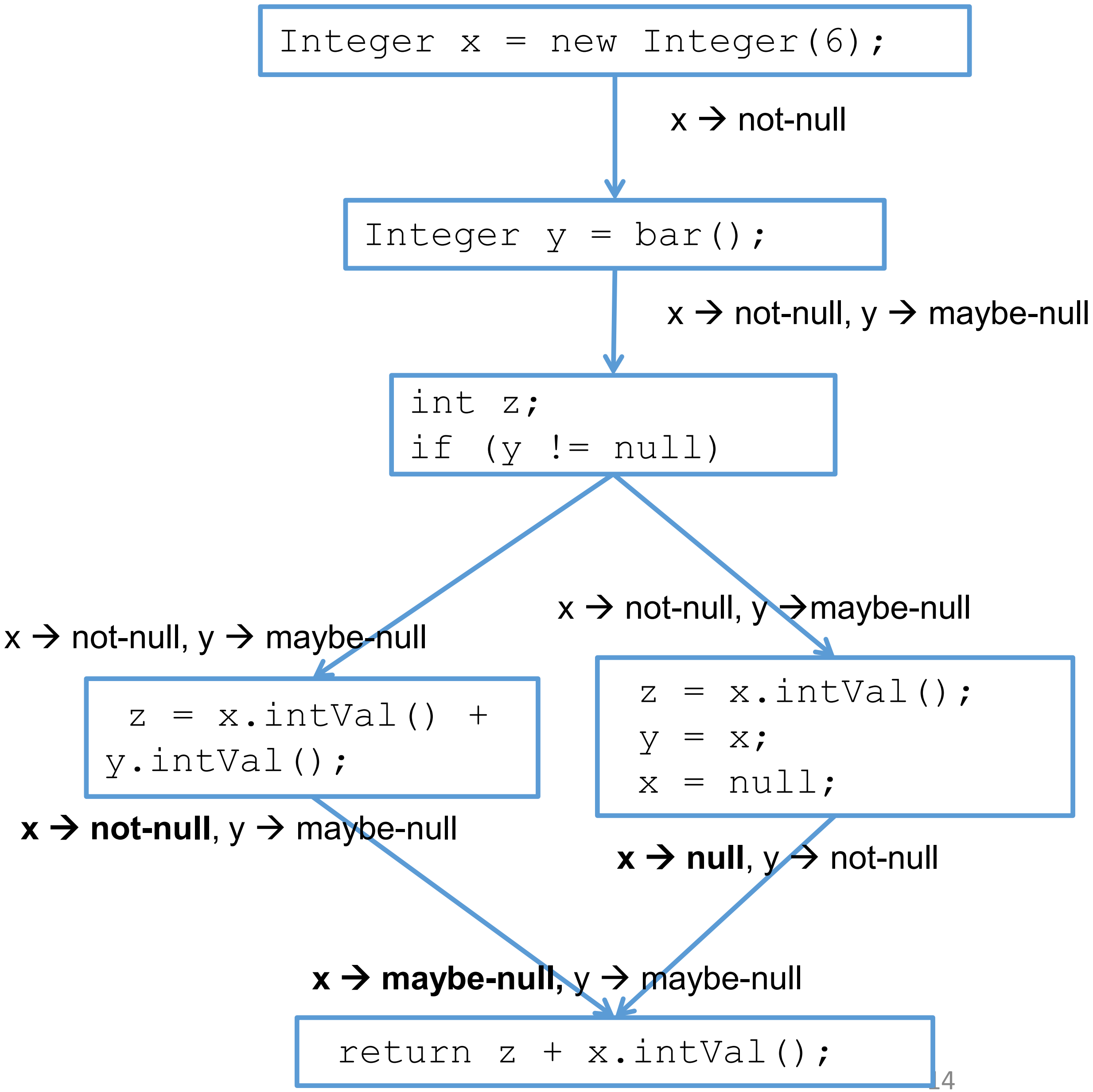
Null pointer analysis

- Track each variable in the program at all program points.
- Abstraction:
  - Program counter
  - 3 states for each variable: null, not-null, and maybe-null.
- Then check if, at each dereference, the analysis has identified whether the dereferenced variable is or might be null.

Control flow graph

```
1.  int foo() {
2.      Integer x = new Integer(6);
3.      Integer y = bar(); // external
                           // library
4.      int z;
5.      if (y != null)
6.          z = x.intVal() + y.intVal();
7.      else {
8.          z = x.intVal();
9.          y = x;
10.         x = null;
11.     }
12.     return z + x.intVal();
13. }
```

Error: may have null pointer on line 12, because x may be null!



Sign Analysis

- Abstraction
  - Positive, Negative, Maybe-Positive
- Operations

Operation	Operand 1	Operand 2	Result
+	Positive	Positive	Positive
+	Positive	Negative	Maybe-Positive
+	Negative	Positive	Maybe-Positive
+	Negative	Negative	Negative
*	Positive	Positive	Positive
*	Positive	Negative	Negative
*	Negative	Positive	Negative
*	Negative	Negative	Positive

Sign Analysis (Negative Array Index)

```
1.  int arr[1000];

2.  int main(void) {
3.      int i = -2;
4.      int j = -10;
5.      int m = 20;

6.      for (int k = j; k < 200; k++) {
7.          i*=2;
8.          m-=4;
9.          arr[i+m] = k;
10.     }
11.     return 0;
12. }
```

Can the array index in statement 9 be negative?

Sign Analysis

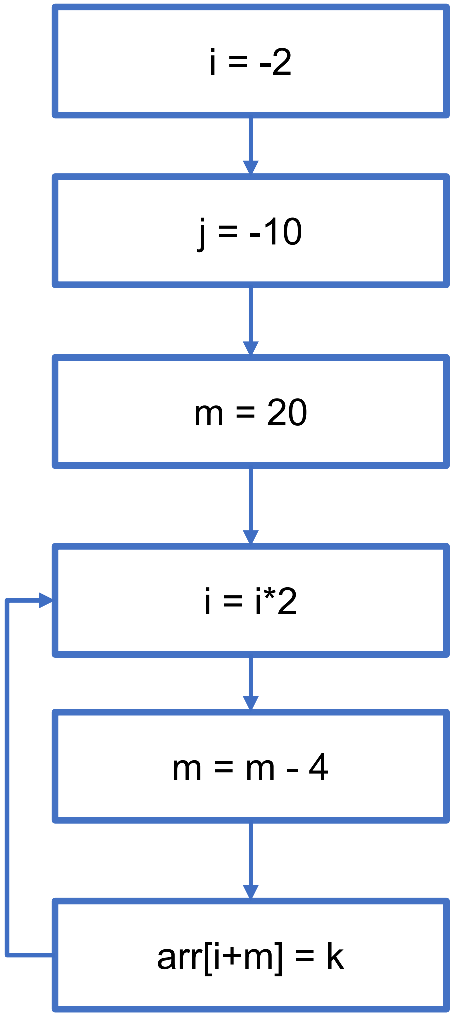
- Abstraction
  - Positive, Not-Positive, Maybe-Positiv
- Operations

Operation	Operand 1	Operand 2	Result
+	P	P	P
+	P	NP	**MP**
+	NP	P	**MP**
+	NP	NP	NP
-	P	P	**MP**
-	P	NP	P
-	NP	P	NP
-	NP	NP	**MP**
*	P	P	P
*	P	NP	**MP**
*	NP	P	**MP**
*	NP	NP	P

Can be zero!

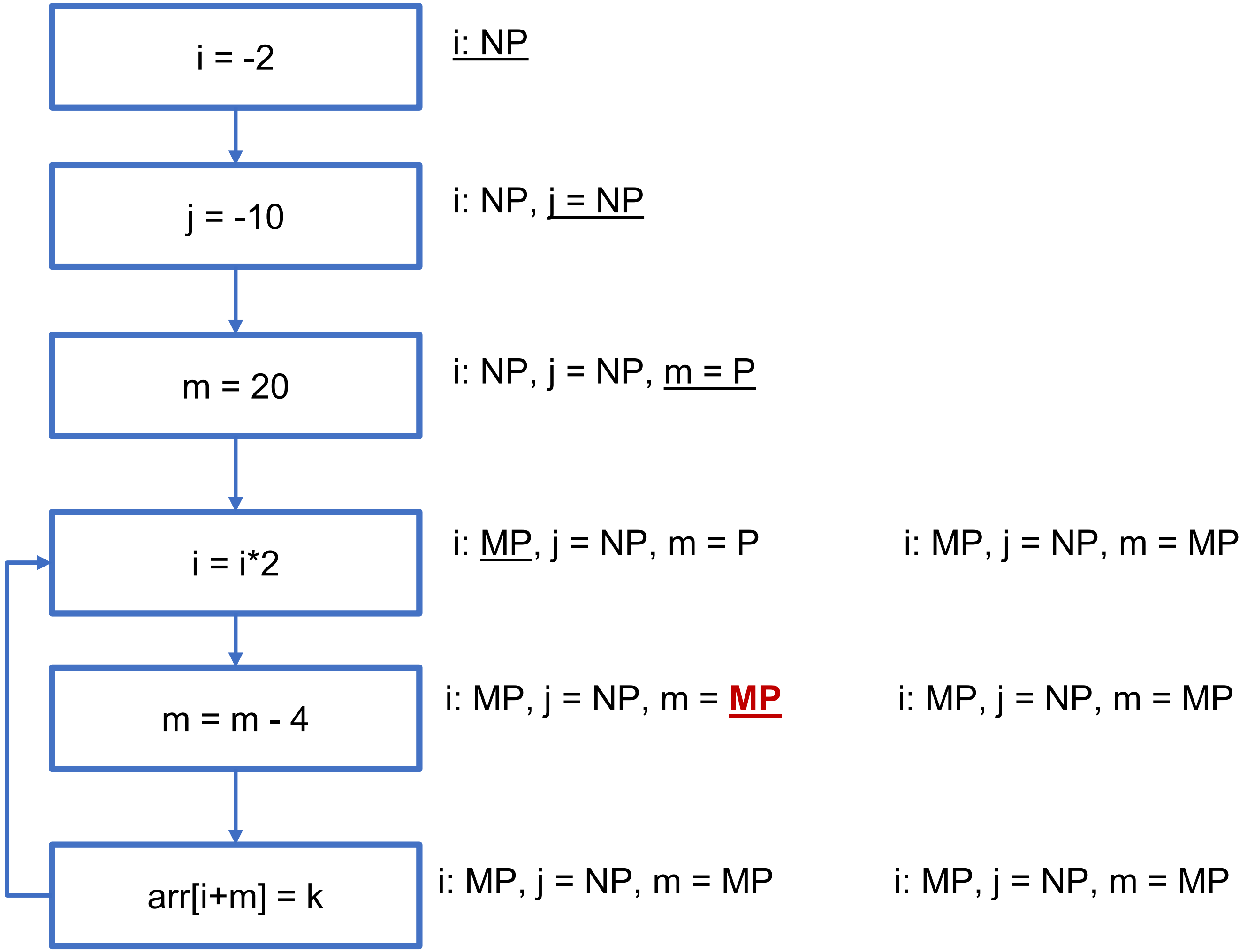
Sign Analysis: Control Flow Graph

```c
int arr[1000];

int main(void) {
    int i = -2;
    int j = -10;
    int m = 20;

    for (int k = j; k < 200; k++) {
        i*=2;
        m-=4;
        arr[i+m] = k;
    }
    return 0;
}
```



Sign Analysis: Apply Abstraction Rules

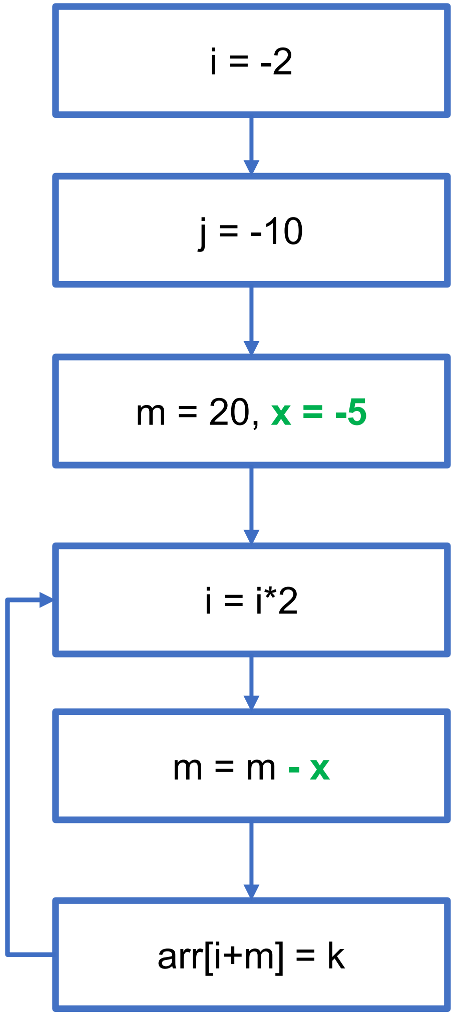
Op	V1	V2	Res
+	P	P	P
+	P	NP	**MP**
+	NP	P	**MP**
+	NP	NP	NP
-	P	P	**MP**
-	P	NP	P
-	NP	P	NP
-	NP	NP	**MP**
*	P	P	P
*	P	NP	**MP**
*	NP	P	**MP**
*	NP	NP	P



m = MP → array-index might be negative

Another Program

Op	V1	V2	Res
+	P	P	P
+	P	NP	**MP**
+	NP	P	**MP**
+	NP	NP	NP
-	P	P	**MP**
-	P	NP	P
-	NP	P	NP
-	NP	NP	**MP**
*	P	P	P
*	P	NP	**MP**
*	NP	P	**MP**
*	NP	NP	P



i: NP

i: NP, j = NP

i: NP, j = NP, m = P, x = NP

i: NP, j = NP, m = P, x = NP   i: NP, j = NP, m = P, x = NP

i: NP, j = NP, m = P, x = NP   i: NP, j = NP, m = P, x = NP

i: NP, j = NP, m = P, x = NP   i: NP, j = NP, m = P, x = NP

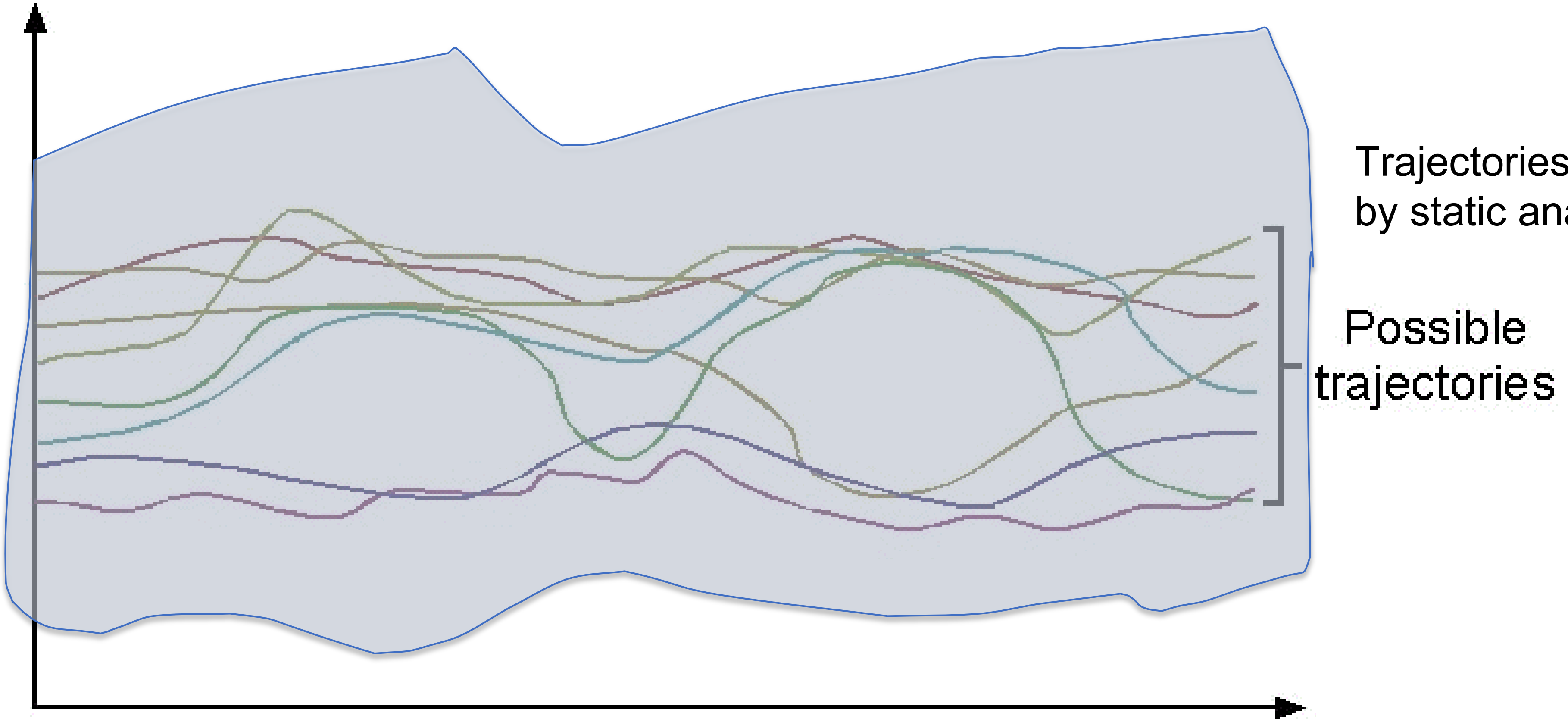
m = P → array-index must be positive

Exercise

- Create the control flow graph
- Apply the resolution rules till fixpoint reached

Static Analysis Soundness and Precision

Adapted from https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html



Static analysis covers all possible program paths (is sound), but has overapproximation (is imprecise)

Static Analysis Tools: Astree

- Based on abstract interpretation applied to the semantics of C language
  - Abstract interpretation is a mathematically formal way of doing the analysis we've been doing
  - More details: https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/10.ABSTRACT-INTERPRETATION.html
- Types of bugs detected:
  - Division by zero
  - Out of bounds array indexing
  - Erroneous pointer manipulation and dereferencing (NULL, unitialized and dangling pointers)
  - Integer and floating-point arithmetic overflow
  - Read access to uninitialized variables
- Used by Airbus France, Bosch automotive steering, Framatome (nucleur reactor safety)

Static Analysis Tools: CppCheck

- Automatic variable checking
- Bounds checking for array overruns
- Classes checking (e.g. unused functions, variable initialization and memory duplication)
- Usage of deprecated or superseded functions according to Open Group
- Exception safety checking, for example usage of memory allocation and destructor checks
- Memory leaks, e.g. due to lost scope without deallocation
- Resource leaks, e.g. due to forgetting to close a file handle