Static Analysis – part 2

Control/Dataflow analysis

- **Reason** about all possible executions, via paths through a *control flow graph*.
  - Track information relevant to a property of interest at every *program point*.
  - Including exception handling, function calls, etc
- Define an **abstract domain** that captures only the values/states relevant to the property of interest.
- **Track** the abstract state, rather than all possible concrete values, for all possible executions (paths!) through the graph.

Example

- Consider the following program:

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

- Use **zero analysis** to determine if y could be zero at the division.

Zero/Null-pointer Analysis

- Could a variable x ever be 0?
  - (what kinds of errors could this check for?)
- Original domain: N maps every variable to an integer.
- Abstraction: every variable is non zero (NZ), zero(Z), or maybe zero (MZ)

Zero analysis transfer

- What operations are relevant?

Zero analysis join

- Join(zero, zero) → zero
- Join(not-zero, not-zero) → not-zero
- Join(zero, not-zero) → maybe-zero
- Join(maybe-zero, *) → maybe-zero

Example

- Consider the following program:

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

- Use **zero analysis** to determine if y could be zero at the division.

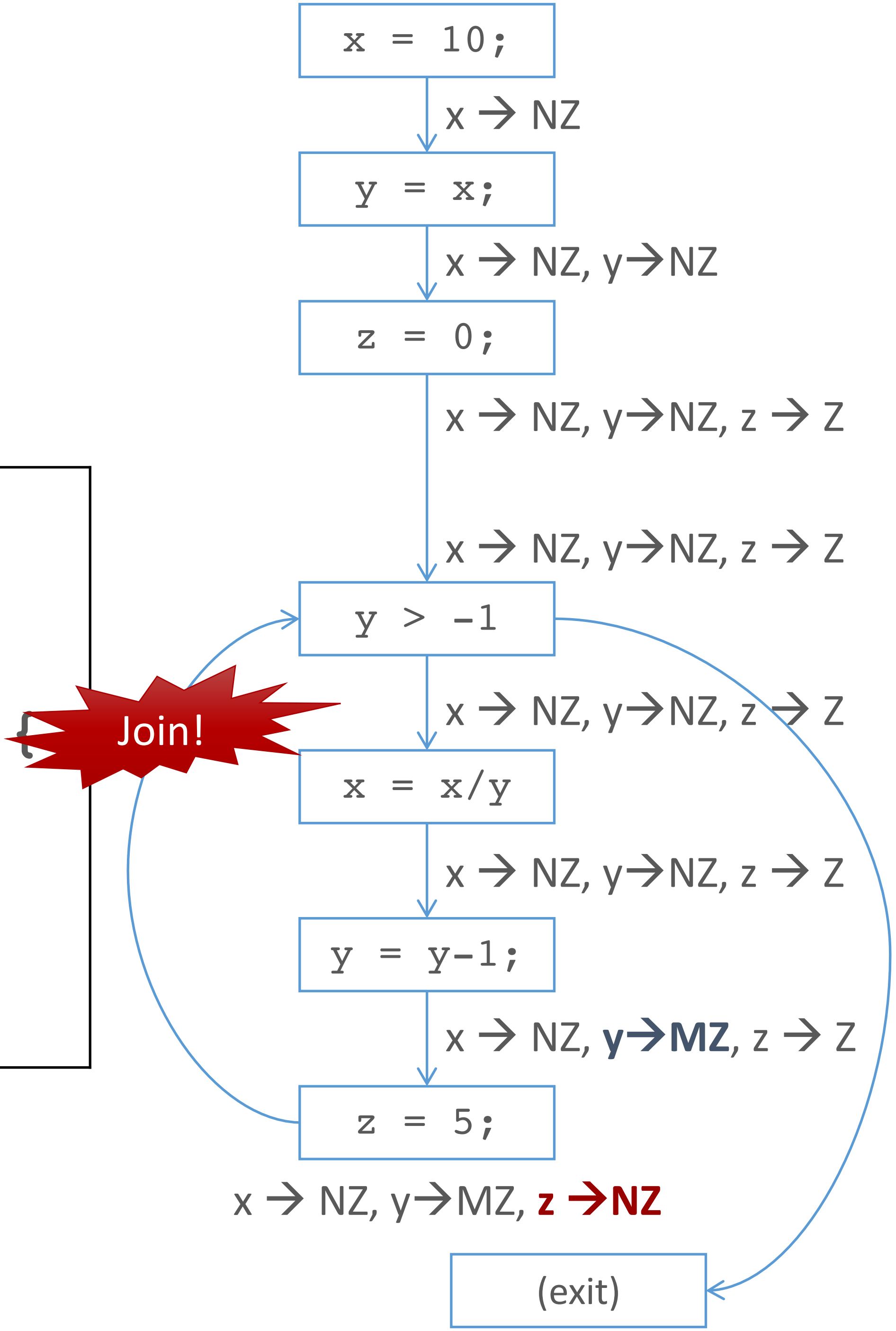
Reminder:

x: Join(NZ,NZ) → NZ
y: Join(MZ,NZ) → MZ
Z: Join(NZ, Z) → MZ

Reminder:

x: Join(NZ,NZ) → NZ
y: Join(MZ,NZ) → MZ
Z: Join(NZ, Z) → MZ

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



`x = 10;`

x → NZ

`y = x;`

x → NZ, y→NZ

`z = 0;`

x → NZ, y→NZ, z → Z

x → NZ, y→NZ, z → Z

`y > -1`

x → NZ, y→NZ, z → Z

`x = x/y`

x → NZ, y→NZ, z → Z

`y = y-1;`

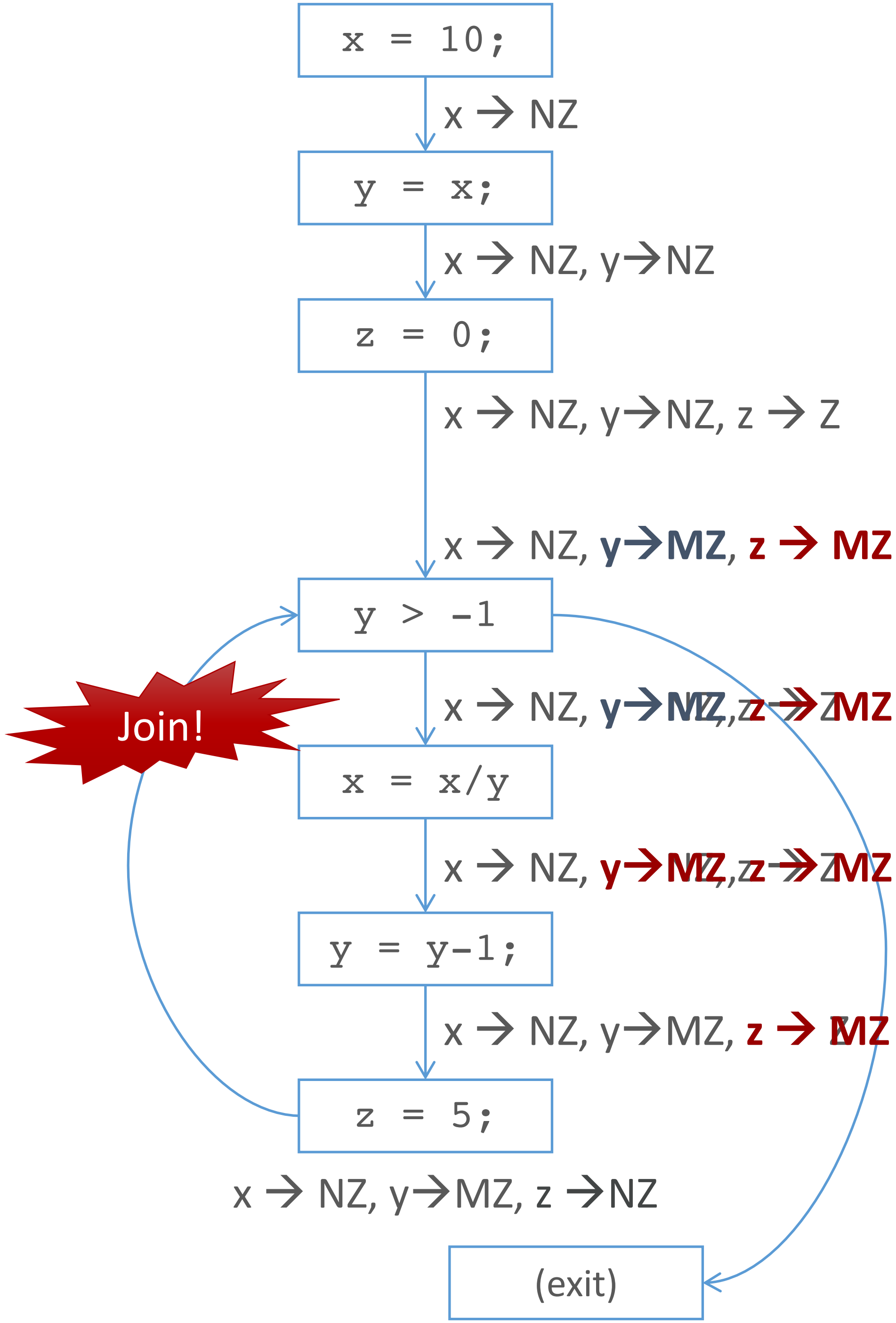
x → NZ, **y→MZ**, z → Z

`z = 5;`

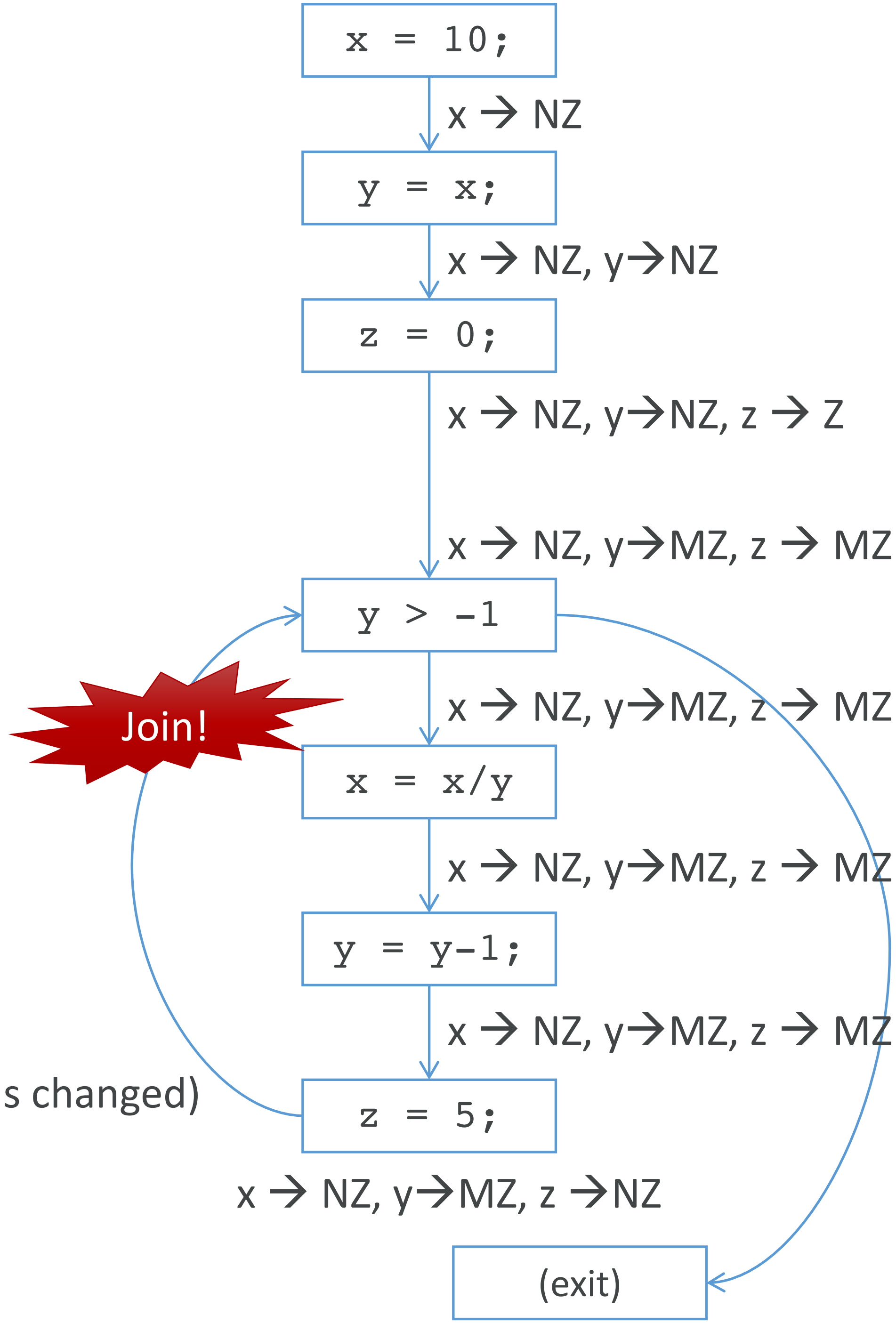
x → NZ, y→MZ, **z →NZ**

(exit)

Join!



(end of iteration 2)



`x = 10;`

x → NZ

`y = x;`

x → NZ, y→NZ

`z = 0;`

x → NZ, y→NZ, z → Z

x → NZ, y→MZ, z → MZ

`y > -1`

Join!

x → NZ, y→MZ, z → MZ

`x = x/y`

x → NZ, y→MZ, z → MZ

`y = y-1;`

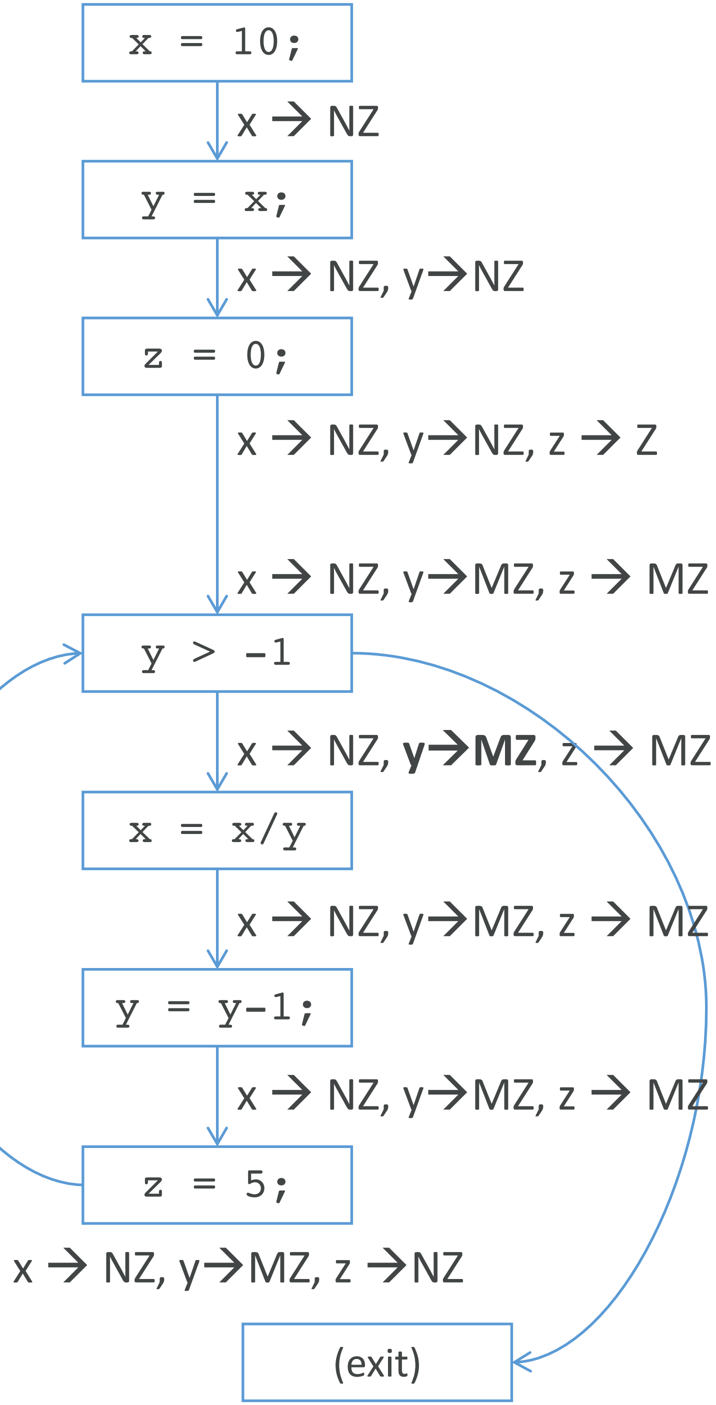
x → NZ, y→MZ, z → MZ

`z = 5;`

x → NZ, y→MZ, z →NZ

(exit)

(end of iteration 3; nothing has changed)



`x = 10;`

x → NZ

`y = x;`

x → NZ, y→NZ

`z = 0;`

x → NZ, y→NZ, z → Z

x → NZ, y→MZ, z → MZ

`y > -1`

x → NZ, **y→MZ**, z → MZ

`x = x/y`

x → NZ, y→MZ, z → MZ

`y = y-1;`

x → NZ, y→MZ, z → MZ

`z = 5;`

x → NZ, y→MZ, z →NZ

(exit)

Warning! Possible division by zero error!

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

Termination intuition

- Can process instructions in whatever order we want, until the information doesn't change over the whole program.
    - Use a special value as the initial state of all uncomputed states.

- A **fixed point** of a function is a data value *v* that a function maps to itself:
    - $f(v) = v$
- The flow function is the mathematical function.
- The dataflow analysis state at each fix point is the data values.

The Bad News: Rice's Theorem

> "Any nontrivial property about the language recognized by a Turing machine is undecidable.“
>
> Henry Gordon Rice, 1953

Every static analysis is necessarily incomplete or unsound or undecidable (or multiple of these)

Computability theory says…

- **Halting problem:** the problem of determining whether a given program will halt/terminate on a given input.
- A *general* algorithm that solves this problem is impossible.
    - More specifically: it's undecidable (it's possible to get a *yes* answer, but not a *no* answer).
    - (sometimes you can use heuristics, but solving it generally for all programs is still out.)
- The proof here is very elegant. But trust me: this problem is extremely impossible.

OK, so?

- If you could always statically tell if any program had a non-trivial property (never dereferences null, always releases all file handles, etc, etc), you could also generally solve the halting problem.
- …but the halting problem is *definitely* impossible.
- So: no static analysis is perfect. They will always have false positives or false negatives (or both).
- *All tools make tradeoffs.*

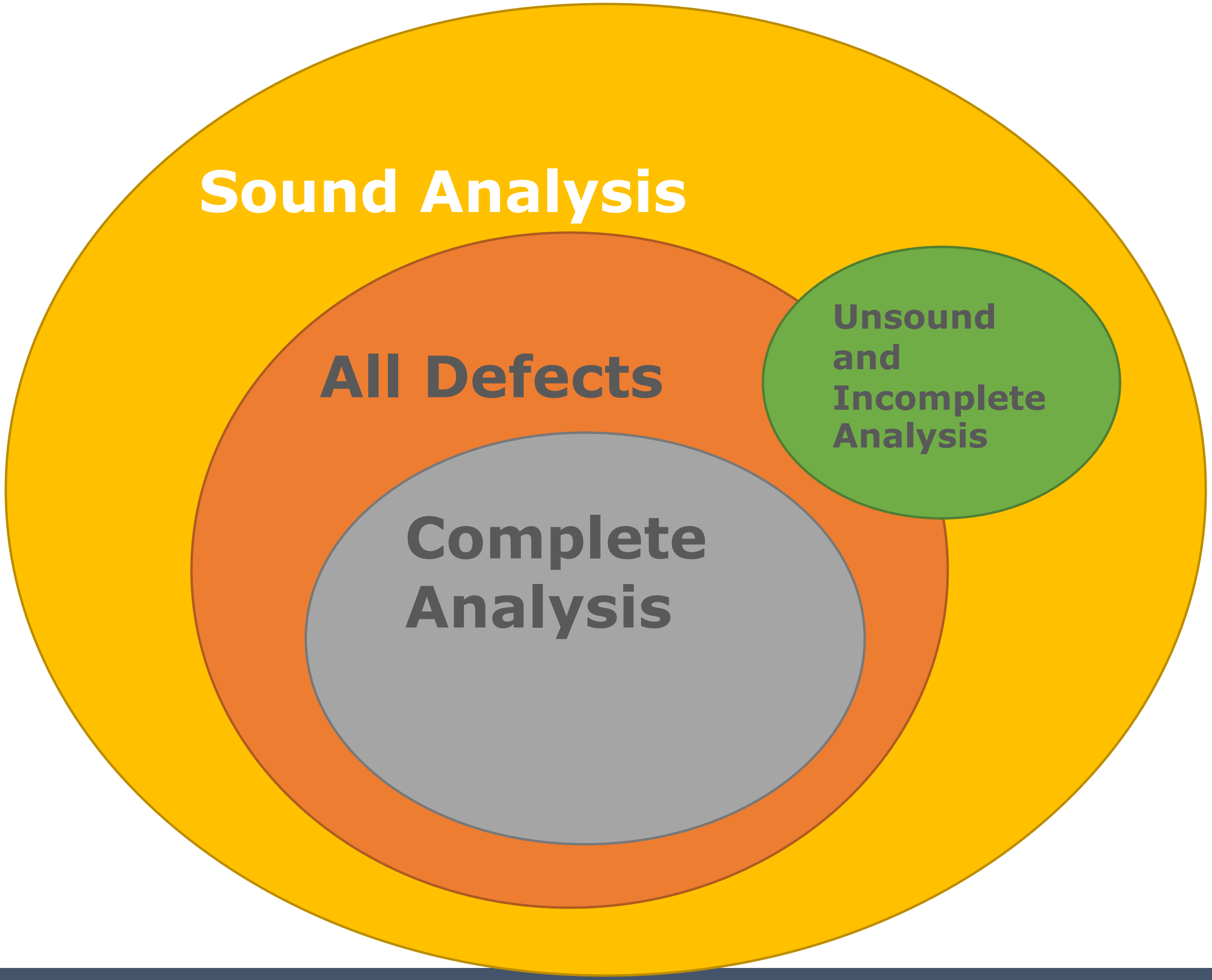
	Error exists	**No error exists**
Error Reported	True positive (correct analysis result)	False positive
No Error Reported	False negative	True negative (correct analysis result)

Sound Analysis:

reports all defects
-> no false negatives
typically overapproximated

Complete Analysis:

every reported defect is an actual defect
-> no false positives
typically underapproximated



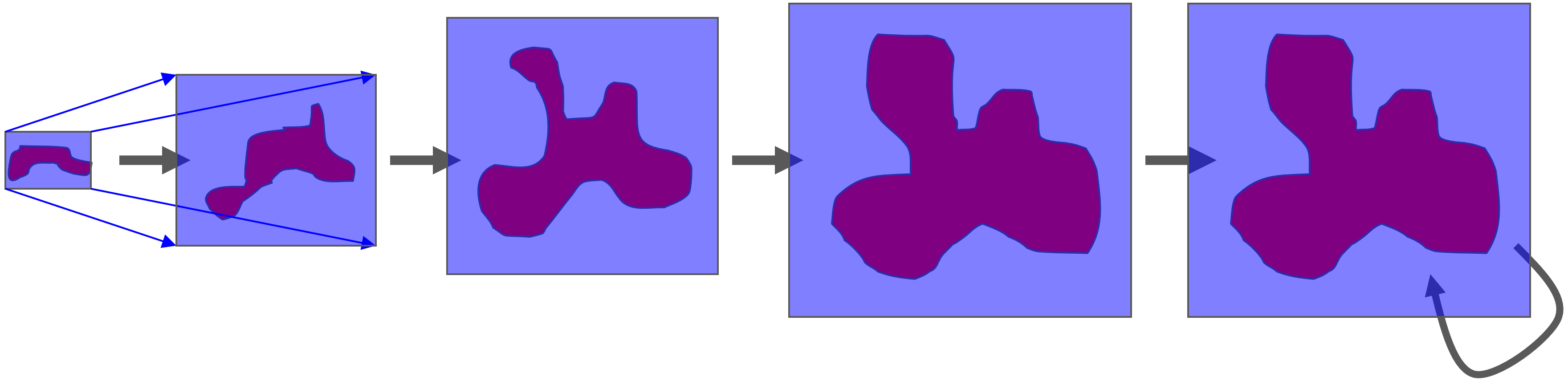
Sound Analysis

All Defects

Unsound and Incomplete Analysis

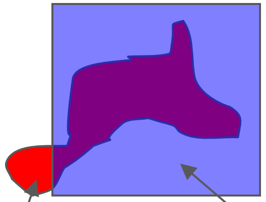
Complete Analysis

Soundness and precision



🟥 Program state covered in actual execution

🟦 Program state covered by abstract execution with analysis



unsound
(false negative)

imprecise
(false positive)

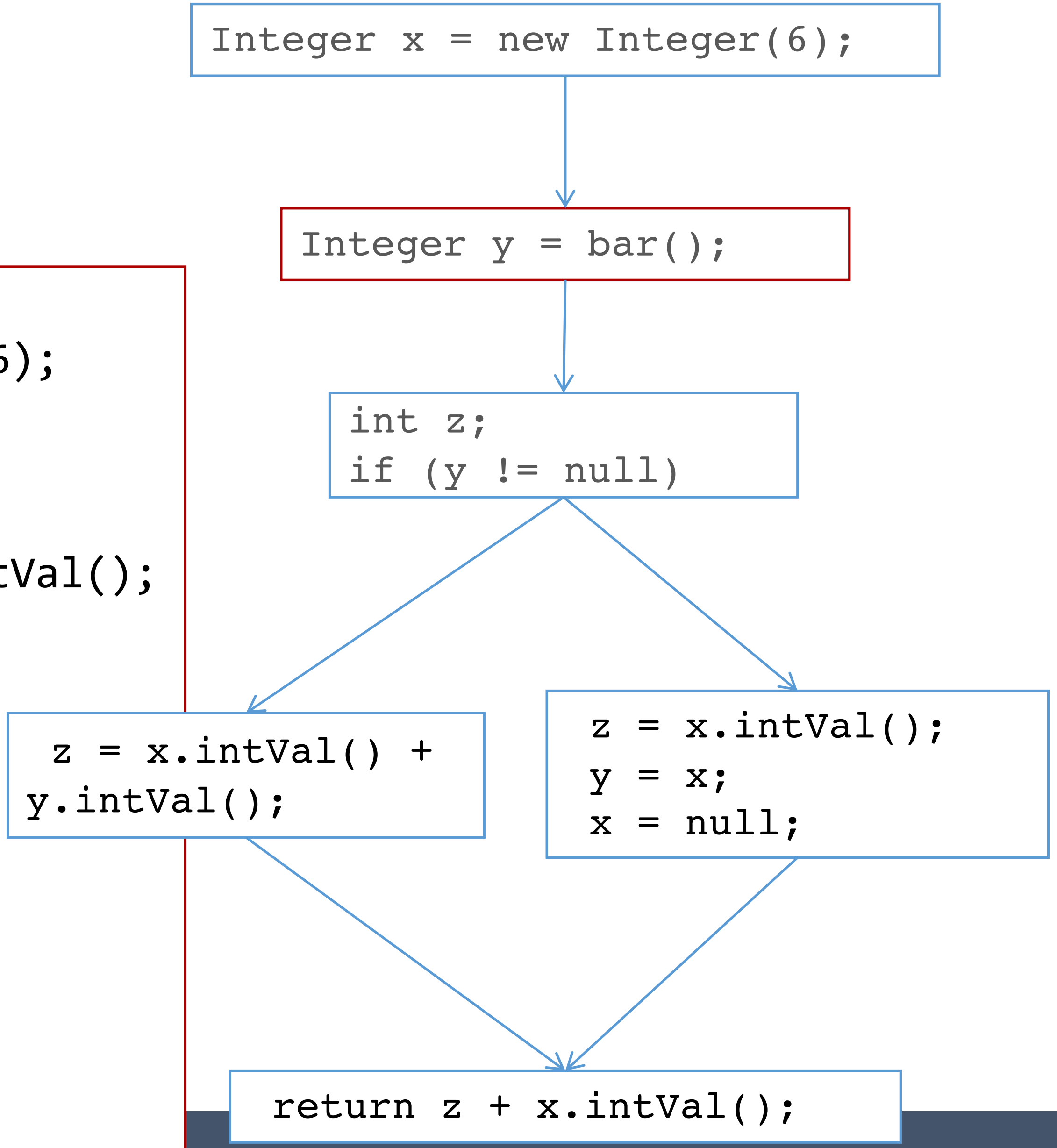
Sound vs. Heuristic Analysis vs. Reality

- Heuristic Analysis
  - FindBugs, coverity, checkstyle …
  - Follow rules, approximate, avoid some checks to reduce false positives
  - May report false positives and false negatives
- Sound Static Analysis
  - Type checking, Not-Null, … (specific fault classes)
  - Sound abstraction, precise analysis to reduce false positives

- But, in practice: languages are complicated, all tools need to make decisions about how to model what’s going on/actual abstraction under the hood.

Example: Null pointers

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



```
Integer x = new Integer(6);
```

```
Integer y = bar();
```

```
int z;
if (y != null)
```

```
 z = x.intVal() +
y.intVal();
```

```
z = x.intVal();
y = x;
x = null;
```

```
return z + x.intVal();
```

What about that function call?

- Some simple options:
  - If you’re worried about totally wacky control flow (exceptions, longjumps), they can be modeled in wackier/more complicated control flow graphs.
  - Ignore it by assuming that all functions return and tempering your claim: “assuming the program terminates, the analysis soundly computes…”
    - Most people don’t bother; this is basically assumed.

- Interprocedural analyses exist, but are challenging to scale and beyond the scope of this lecture.
  - E.g., Build single big graph or abstract at method level; often manual annotations to help

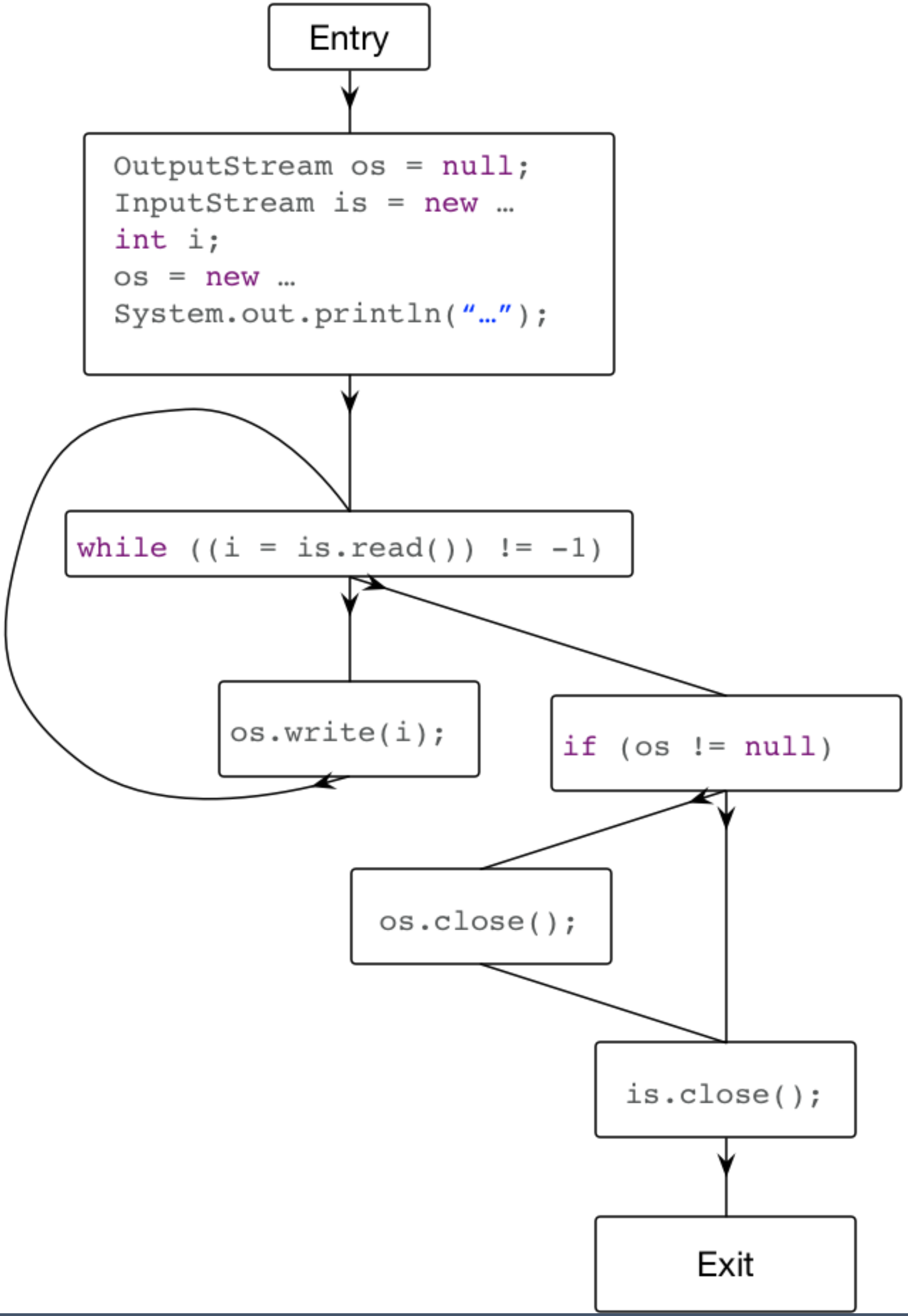
Exercise: File open/close

```
public class StreamDemo {
  public static void main(String[] args) throws Exception {
     OutputStream os = null;
     InputStream is = new FileInputStream("in.txt");
     int i;
     try {
      os = new FileOutputStream("out.txt");
      System.out.println("Copying in progress...");
      while ((i = is.read()) != -1) {
         os.write(i);
      }
      if (os != null) {
        os.close();
       }
     } catch (IOException e) {
        e.printStackTrace();
     }
      is.close();
    }
}
```

File open/close

- Abstract domain: file open, file closed, file maybe-open.
- Transfer and joins left as exercise to the reader…

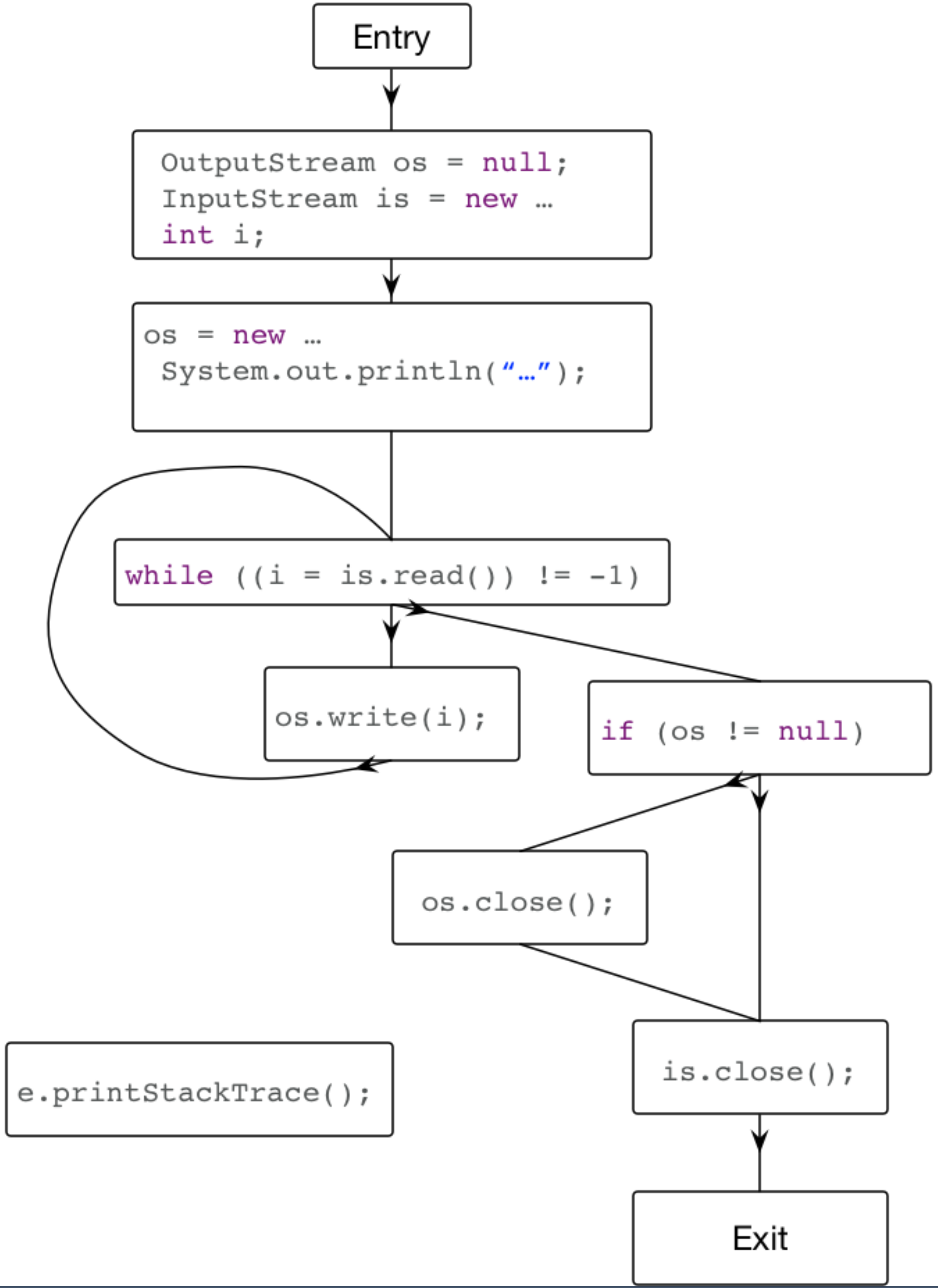
Try-Catch?


Entry
OutputStream os = null;
InputStream is = new …
int i;
os = new …
System.out.println("…");
while ((i = is.read()) != -1)
os.write(i);
if (os != null)
os.close();
is.close();
Exit

Design choices: representation and abstract domain

- What if we don't model the try/catch?
- If we do…how should we include it?

Try-Catch?


Entry
OutputStream os = null;
InputStream is = new …
int i;
os = new …
System.out.println("…");
while ((i = is.read()) != -1)
os.write(i);
if (os != null)
os.close();
e.printStackTrace();
is.close();
Exit

Design choices: representation and abstract domain

- What if we don’t model the try/catch?
- If we do…how should we include it?
- …what about non-IOExceptions?
- Broader question: How precisely should we model semantics?
    - E.g., Of instructions, of conditional checks, etc.

Upshot: analysis as approximation

- Analysis must approximate in practice
  - False positives: may report errors where there are really none
  - False negatives: may not report errors that really exist
  - All analysis tools have either false negatives or false positives
- Approximation strategy
  - Find a pattern P for correct code
    - which is feasible to check (analysis terminates quickly),
    - covers most correct code in practice (low false positives),
    - which implies no errors (no false negatives)
- Analysis can be pretty good in practice
  - Many tools have low false positive/negative rates
  - A sound tool has no false negatives
    - Never misses an error in a category that it checks

Tools

- Most commercial “static analysis tools”, bug detectors, incl. FindBugs
- Examples: Nullness, atomicity, information flow, …
- Many compiler optimizations…
- Most of the “code quality” tools on GitHub marketplace.

Summary

- Static analysis: systematic automated analysis of the program source without executing the program
- Structural analyses look for patterns in the code
- Control-flow analyses analyze all possible paths (global property)
- Data-flow analyses analyze possible (abstract) values of variables on all paths
  - Abstraction, transfer function, join
  - Fix point computation; termination
- Analyses unsound or incomplete or both