Software Engineering

QA + Testing IV: Static Code Analysis

Software & Systems Engineering | Prof. Dr. Andreas Vogelsang | 13.12.2023

@andivogelsang
vogelsang@cs.uni-koeln.de

Learning Goals for Today

- Know the definition of static analysis.
- Explain the types of failures that static analysis targets.
- Differentiate between structural, control and data flow analyses

Static Analysis

Motivation

- Relevant errors may only occur on exceptional or hard to stimulate paths of a program
- Testing all possible paths through a program is impossible
- Wouldn't it be nice to have an analysis that checks if a property is true for ALL possible paths through a program?

Static Analysis

A static analysis tool S analyzes the source code of a program P to determine whether it satisfies a property φ.

Examples of Static Analysis

- P never accesses a variable that is null.
- P never uses inputs that are not validated.
- P never executes a division by zero.
- P will always close the DB connection.
- P will always return a value.

Safety and Liveness Properties

- *Liveness*: "something good eventually happens."
- *Safety*: "something bad never happens."

Practical Static Analysis

The Ultimate Property

Does P always terminate?

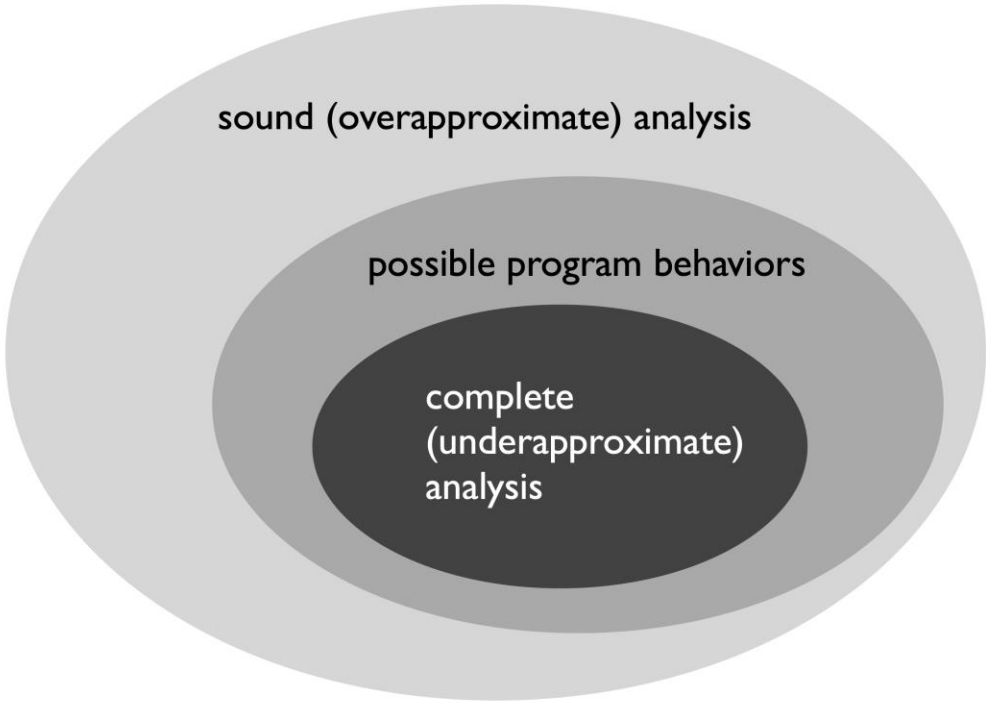
Practical Static Analysis

A static analysis tool S analyzes the source code of a program P to determine whether it satisfies a property φ, but it can be wrong in one of two ways:

- If S is **sound**, it will never miss any violations, but it may say that P violates φ even though it doesn't (resulting in **false positives**).
- If S is **complete**, it will never report false positives, but it may miss real violations of φ (resulting in **false negatives**).

Rice's theorem

For any nontrivial property φ, there is no general automated method to determine whether P satisfies φ. ☹



Concepts and Types of Static Analysis

Basic Concepts of Static Analysis

Abstraction: The possible state space of a program (i.e., the possible values of its variables) is reduced.

Programs as structures: Code is represented by basic structures such as trees or graphs.

Static analysis **systematically** checks whether some property holds in an **abstraction** of the state space of a program.

Types of Static Analysis

- Structural Analysis
- Control Flow Analysis
- Data Flow Analysis

Structural Analysis

Structural Analysis

Abstraction: Abstract Syntax Tree (AST)

An **abstract syntax tree** (AST) is a data structure to represent the structure of a program.

It is a **tree representation** of the abstract syntactic structure of source code.

Each node of the tree denotes a construct occurring in the code.

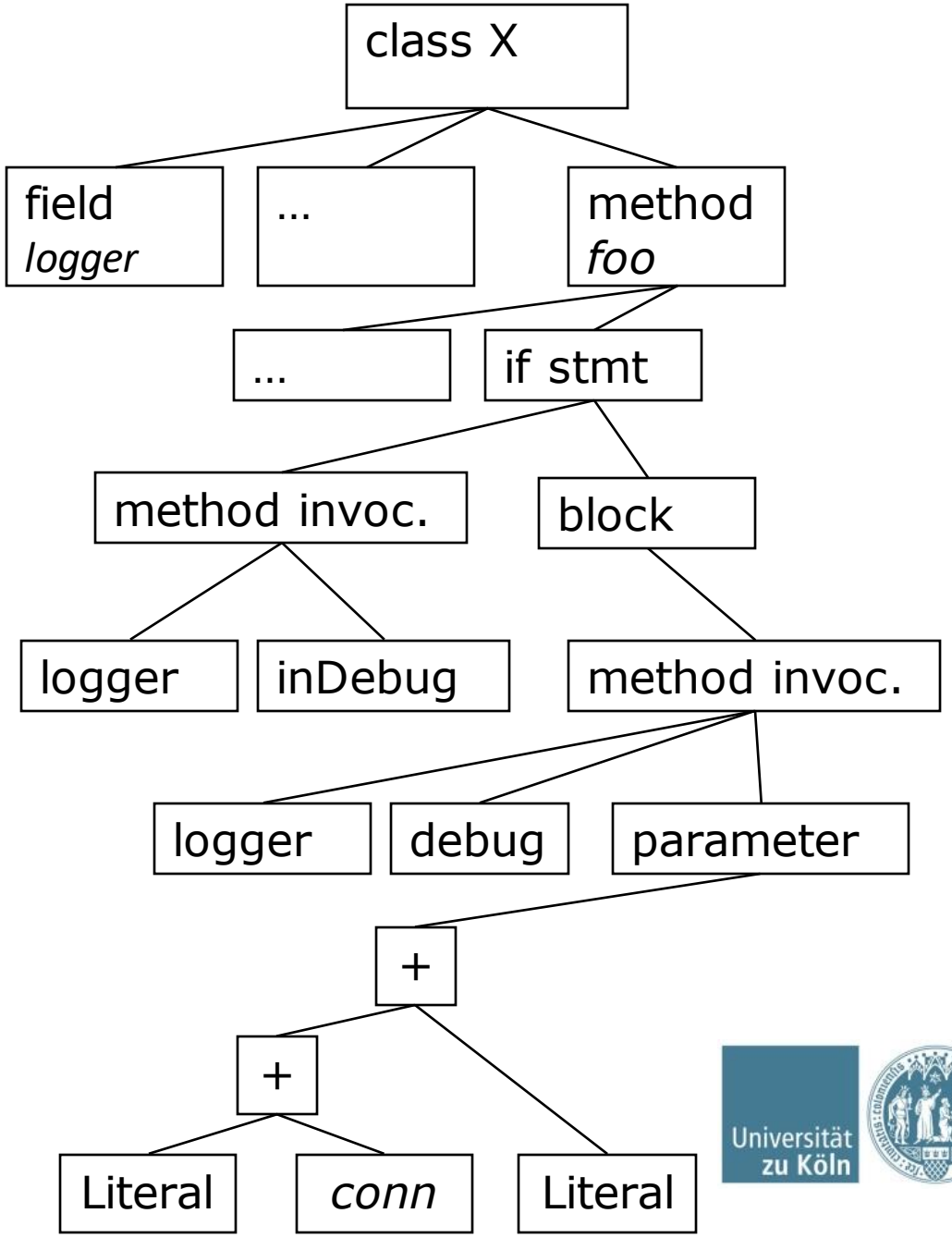
The syntax is “abstract” in the sense that it does not represent every detail appearing in the real syntax, but rather just the **structural or content-related** details.

For instance, grouping parentheses are implicit in the tree structure, so these do not have to be represented as separate nodes. Likewise, a syntactic construct like an if-condition-then statement may be denoted by means of a single node with three branches.

Universität zu Köln

Abstract Syntax Tree

```
class X {
  Logger logger;
  …
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " + conn +
                   "connections.");
    }
  }
}
```



ASTs for Real Programs

"Real" ASTs are way more detailed:

https://astexplorer.net

Types of Structural Analysis

Static Type Checking

- The process of verifying and enforcing the **constraints of types**.
- **Static type checking** is the process of verifying the type safety of a program based on analysis of a program's text (source code)

Code Style Checks

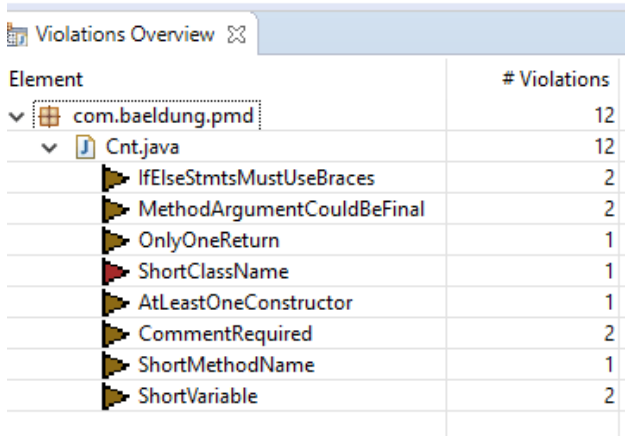
Analysis that check conformance to certain coding styles

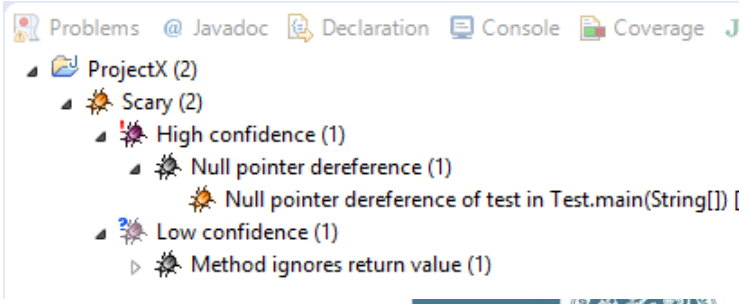
Bug Finding

Analysis that checks the code for typical bug patterns

```
public void foo() {
    int a = computeSomething();

    if (a == "5")
        doMoreStuff();
}
```



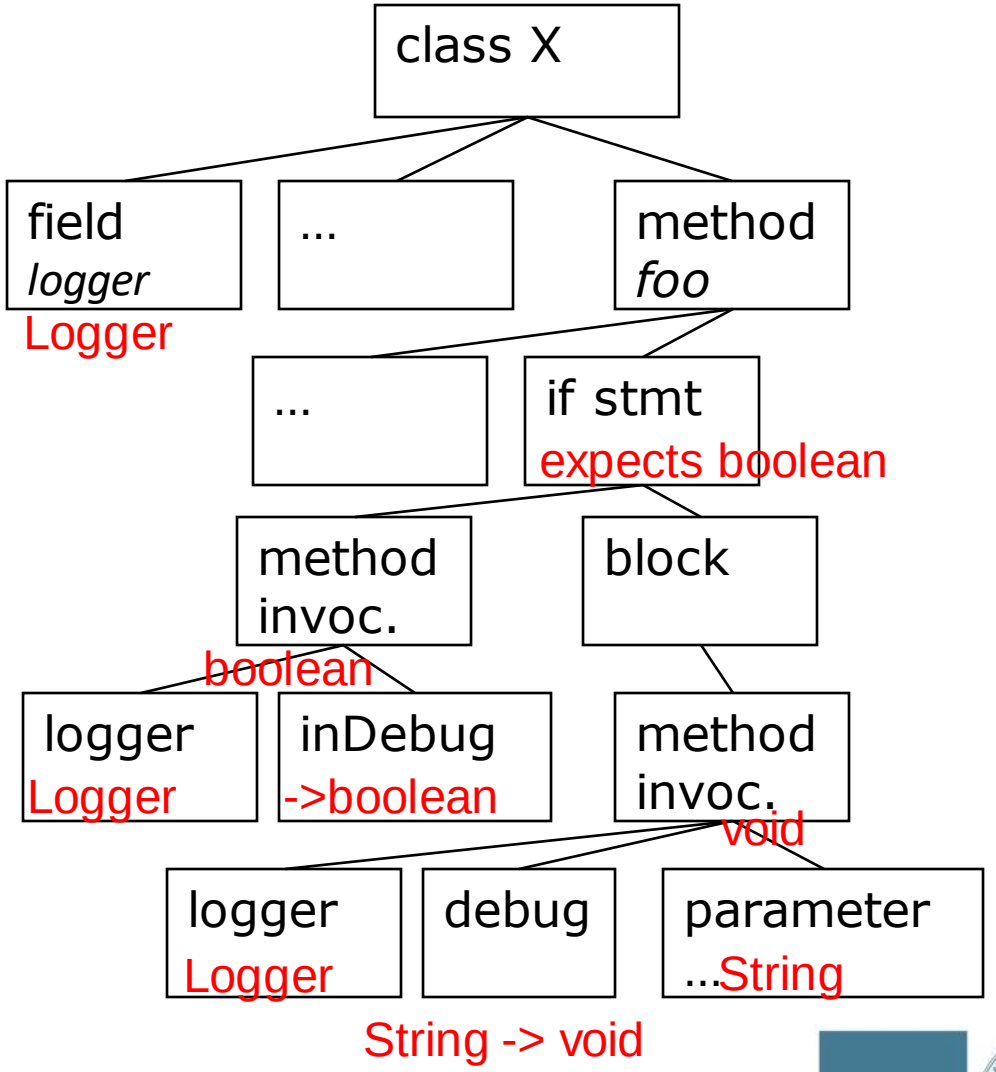


Universität zu Köln

Type Checking

```
class X {
  Logger logger;
  …
  public void foo() {
    …
    if (logger.inDebug()) {
      logger.debug("We have " + conn +
                   "connections.");
    }
  }
}

class Logger {
   boolean inDebug() {…}
   void debug(String msg) {…}
}
```



Code Style Checks

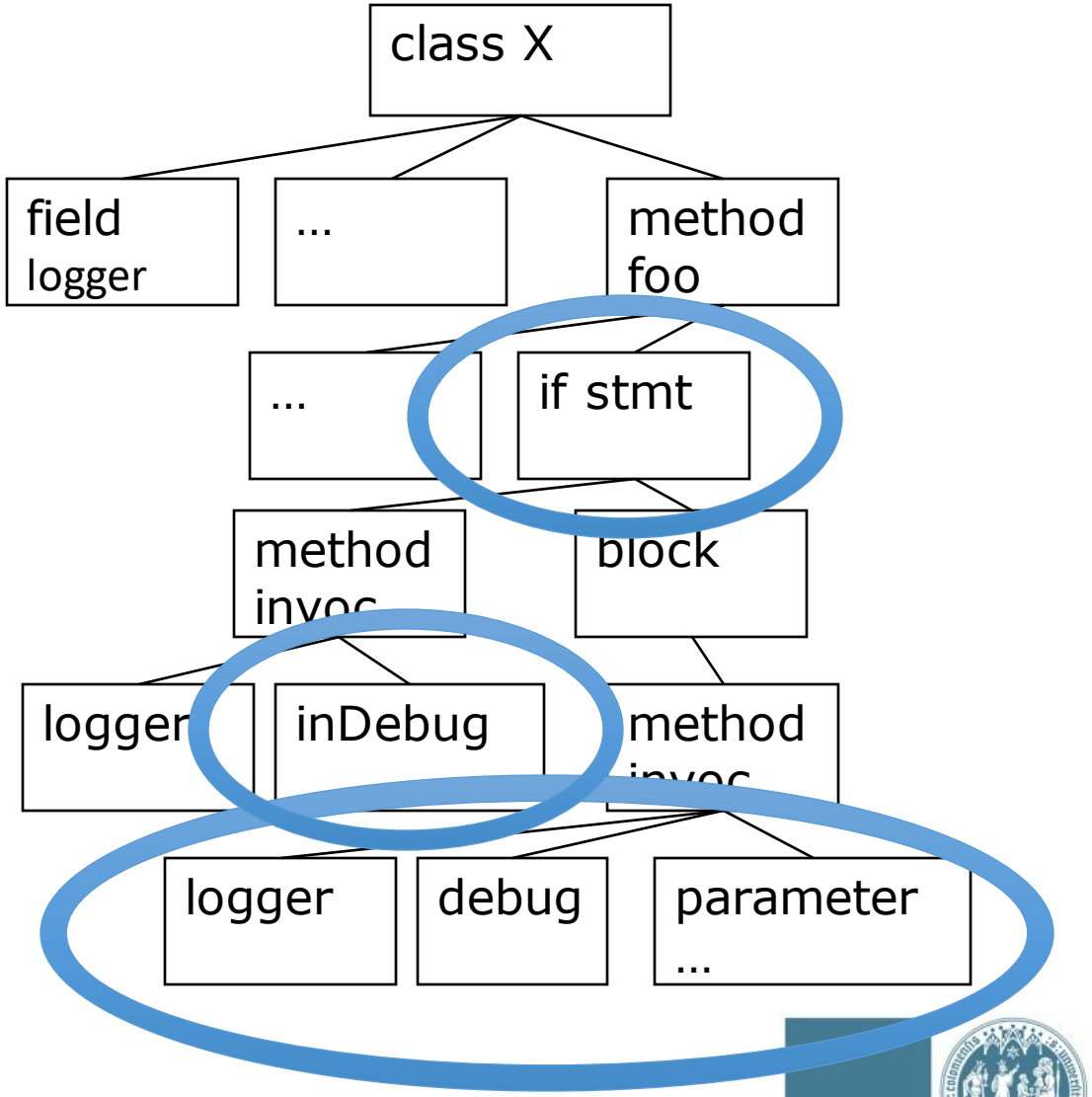
AST Walker

A check that traverses the AST to find violations of rules or properties

Example

No string shall be logged outside of `Logger.inDebug()` check.

- Look for `Logger.debug()` calls
- Check if these are children of an `if (Logger.inDebug())` node

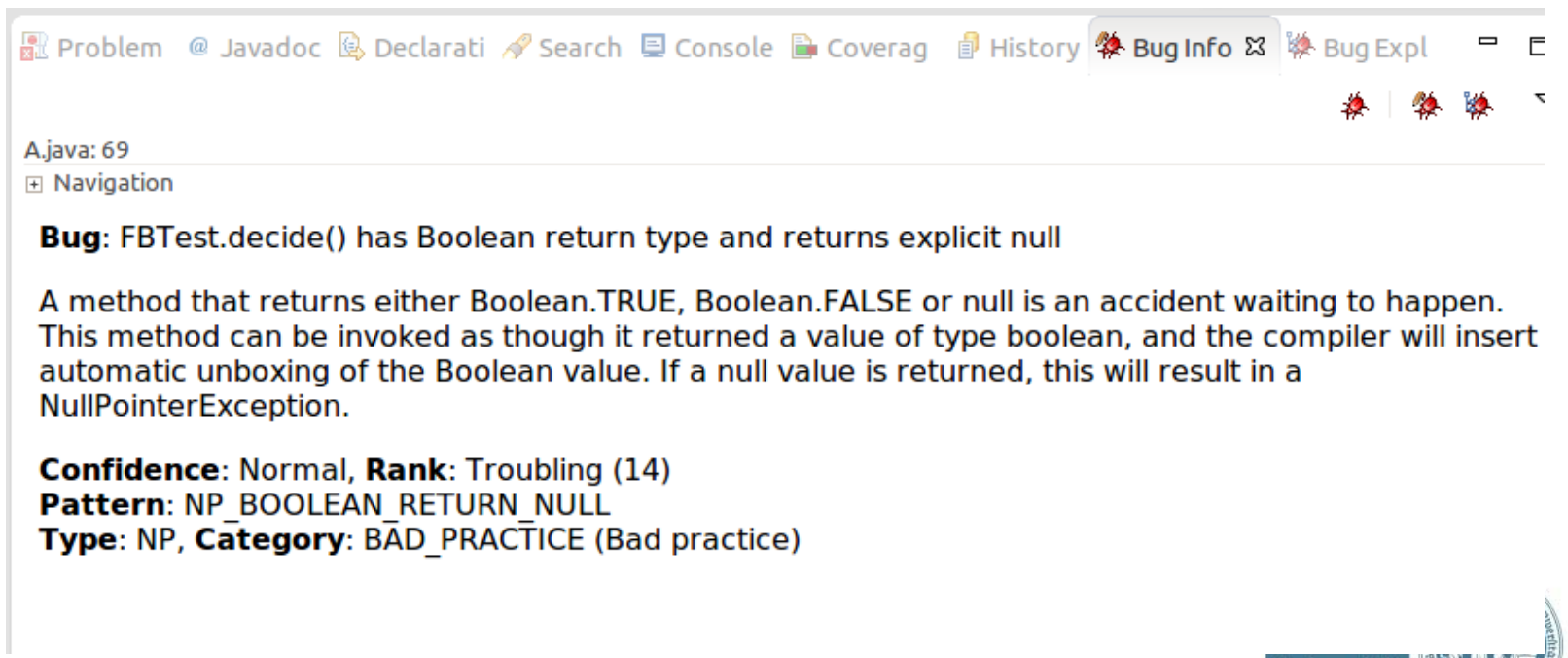


Bug Finding

AST Walker

Bug finding works like Code Style Checks. An AST walker searches for bug patterns.

```
public Boolean decide() {
    if (computeSomething()==3)
        return Boolean.TRUE;
    if (computeSomething()==4)
        return false;
    return null;
}
```



Universität zu Köln

Structural Analysis Summary

Structural Analysis

- Analysis of token streams (text) or code structures.
- Suitable for finding patterns.
- Checks local and structural properties that are independent from any execution path.

Tools for Java

Checkstyle: Checks coding style and conventions

PMD: Identifies bad practices

- Complicated statements
- Inefficient code
- …

Findbugs: Specialized on bug patterns

Control Flow Analysis

Control Flow Analysis

Idea

Analysis of **all possible executions** via paths in a **control flow graph**

- Checking specific properties at each **program point**.
- Including exception handling, function calls, etc.

Abstraction

- Definition of an **abstract domain** that considers **only** the values/states relevant to the property
- Testing the abstract state instead of any concrete values in all possible paths of the program

Universität zu Köln

Propagate state through the CFG

Program Points

Every edge in a control flow graph denotes a *program point*.

Program points characterize the possible conditions that hold **before entering** and **after leaving** a node in the CFG.

Control Flow Analysis

For every node in the CFG:

- Evaluate the state before entering the node
- What is the possible state after leaving the node (apply the transfer function)?

Iterate over all successor nodes in the CFG until no program point's state changes anymore.

Result: A state in every program point

Example: Check if DB connection is always closed after executing a method

Abstraction

- 3 abstract states of interest: `open`, `closed`, `maybe-open`
- Raise a warning if at the end of a method, the state is not `closed`.

Transfer function

Transfer functions specify how to evaluate program expressions on abstract values.

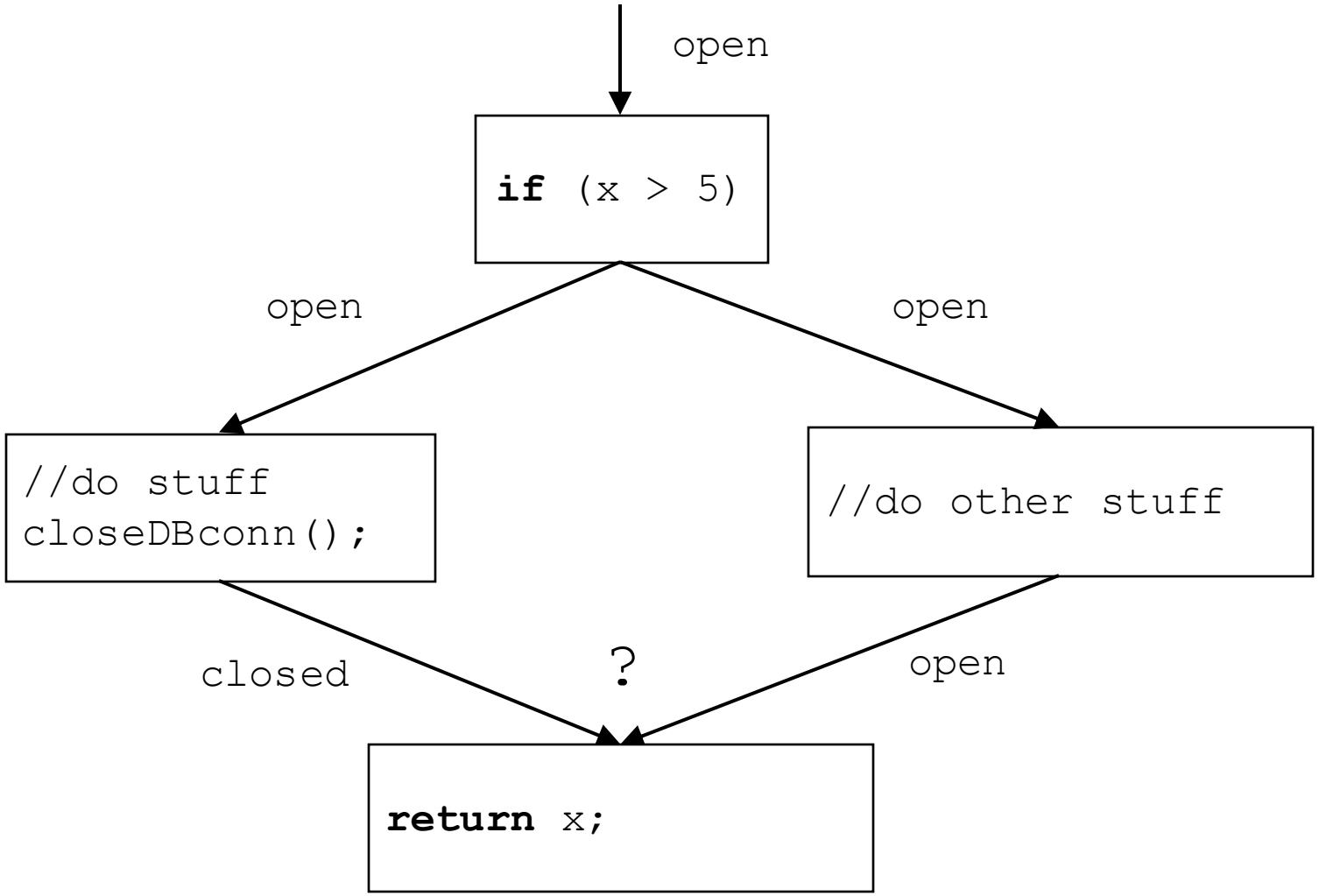
Transfer function for example

`openDBconn()` changes state to `open`

`closeDBconn()` changes state to `closed`

Universität zu Köln

What if control flow merges?



Join function

Join functions specify how to assess the state when the two control flows merge.

Usually, the merge must be resolved to include all possible preconditions.

Join function for example

Join(`open`, `open`) → `open`

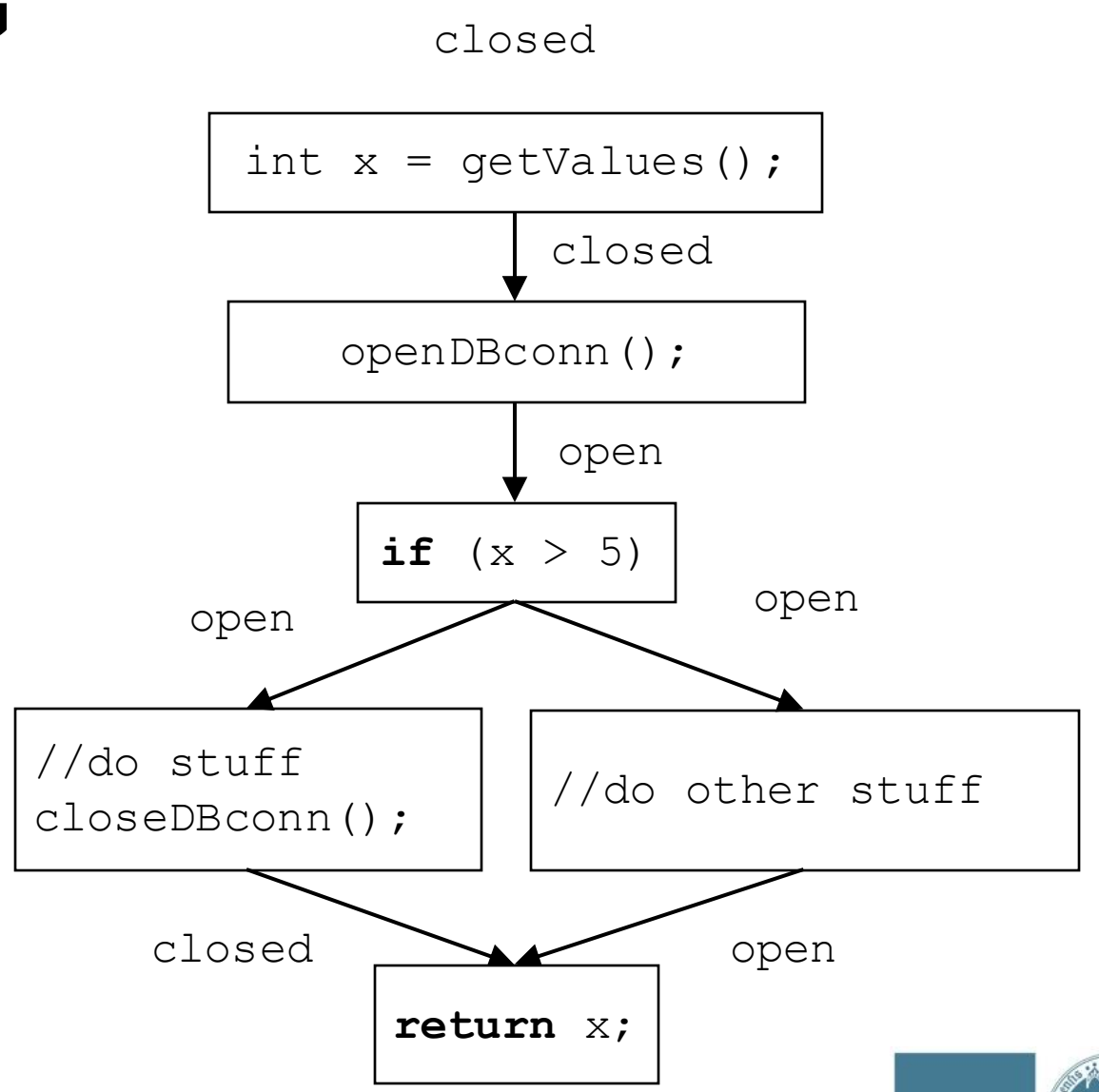
Join(`closed`, `closed`) → `closed`

Join(`open`, `closed`)
→ `maybe-open`

Join(`maybe-open`, `*`)
→ `maybe-open`

Iterate over the CFG

```
int foo() {
  int x = getValue();
  openDBconn();
  if (x > 5){
    //do stuff
    closeDBconn();
  }
  else{
    // do other stuff
  }
  return x;
}
```



Data Flow Analysis

Data Flow Analysis

Data Flow vs. Control Flow Analysis

Data flow: Tracks and manipulates abstract values for a program's variables

Control Flow: Tracks and manipulates the global state of a function.

The analysis itself works similar for both types, except for that, in DFA, a state for each variable must be maintained.

Example: Zero-Detection

Problem

Given a program P, determine which variables may be 0.

Why is this problem interesting?

- Check for division by 0
- Check for empty arrays
- Check for error codes
- …

Selecting an appropriate abstraction

Instead of evaluating all possible values/states of a program, we select an appropriate abstraction for numbers:

- We represent all non-zero numbers by the label `NZ`
- We represent 0 by the label `Z`
- We represent all potentially 0 numbers by the label `MZ` (maybe zero)
- We represent all undefined numbers by the label `NN`

Example: Zero detection

Working with the abstraction

```
x = 5 // label(x) = NZ
z = -5 // label(z) = NZ
p = 0 // label(p) = Z
x = b ? 1 : 0 // label(x) = MZ
x = y * 0 // label(x) = Z
```

concrete

```
a = 5;
b = -3;
c = a * b;
d = 0;
e = c * d;
f = 10 / e;
```

abstract

```
a = NZ;
b = NZ;
c = NZ;
d = Z;
e = Z;
f = NN;
```

Transfer function for example

- `NZ + NZ = MZ`
- `Z + Z = Z`
- `Z * NZ = Z`
- `NZ * NZ = NZ`
- `NZ / Z = NN`
- …

Join function for example

- Join(`Z`, `Z`) → `Z`
- Join(`NZ`, `NZ`) → `NZ`
- Join(`Z`, `NZ`) → `MZ`
- Join(`MZ`, `*`) → `MZ`

⇦ Division by zero indicated by the `NN` label

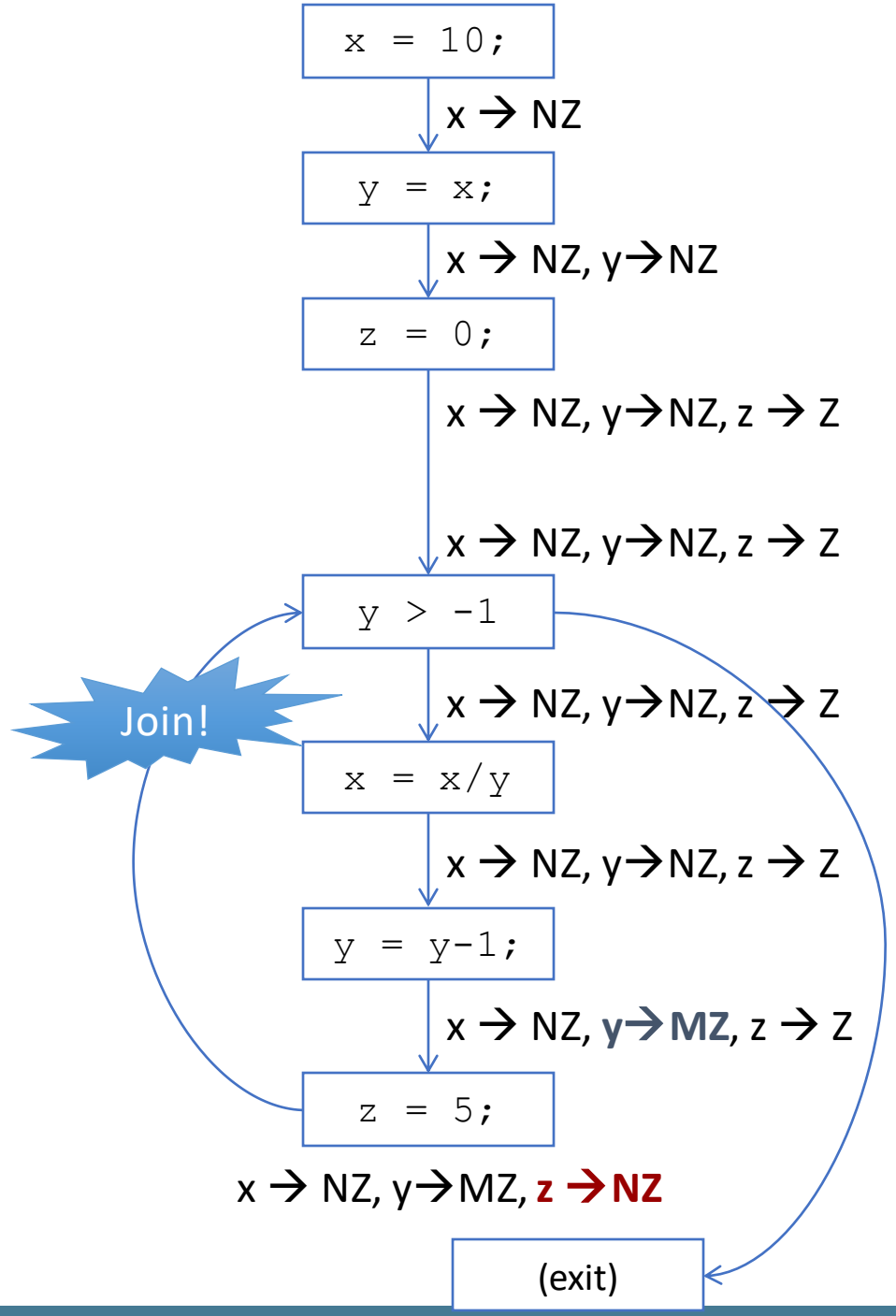
Join Function

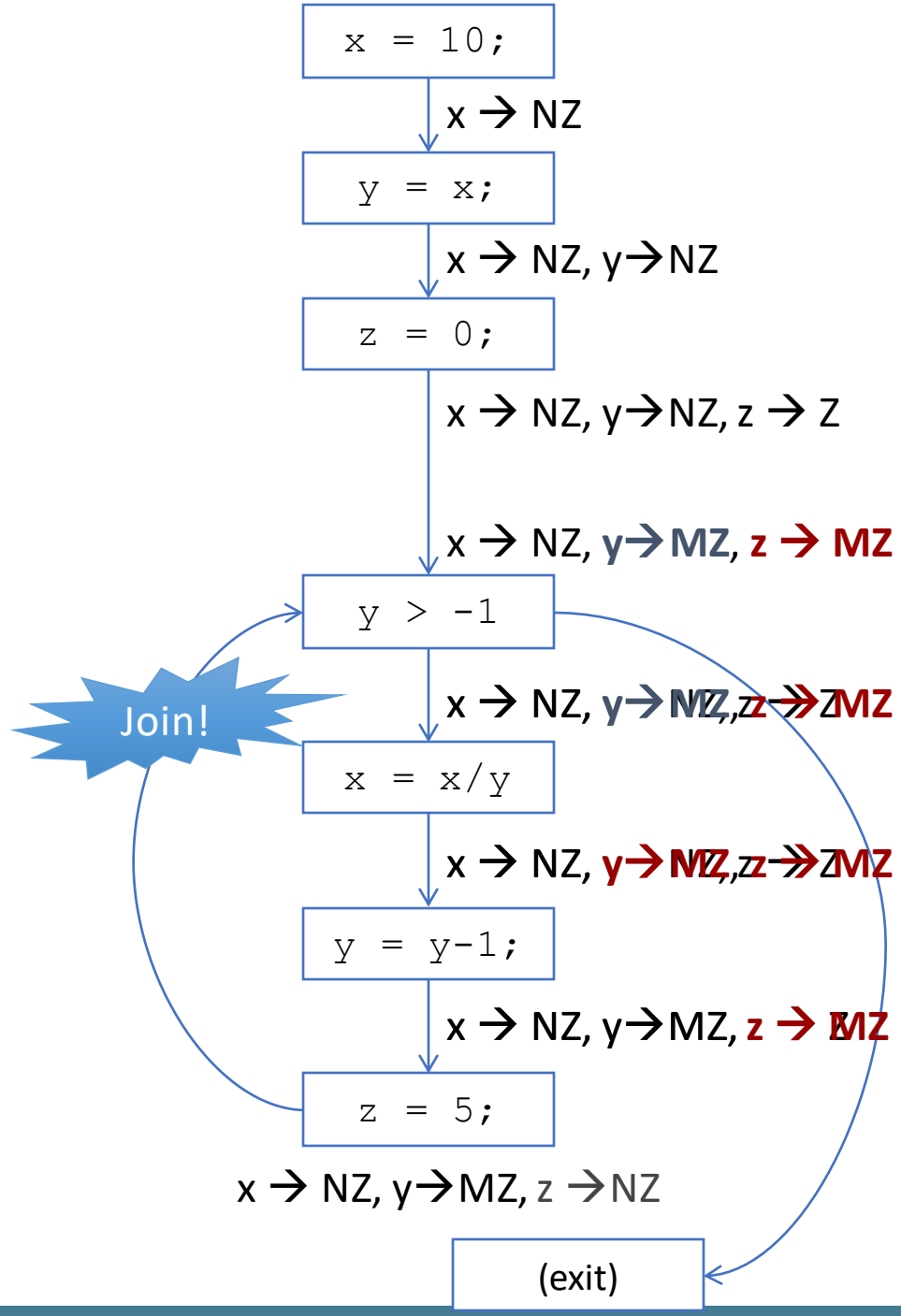
x: Join(NZ, NZ) → NZ

y: Join(MZ, NZ) → MZ

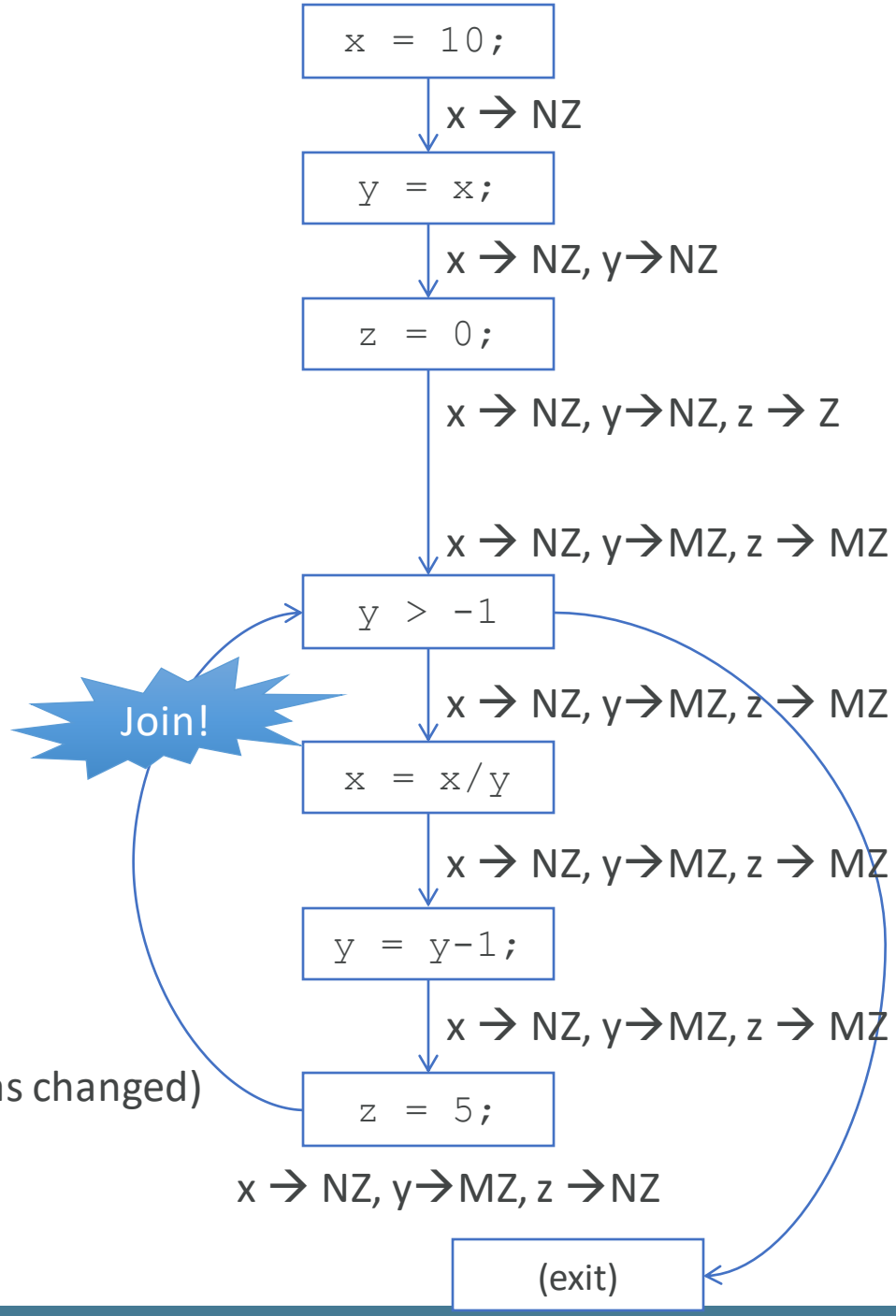
z: Join(NZ, Z) → MZ

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



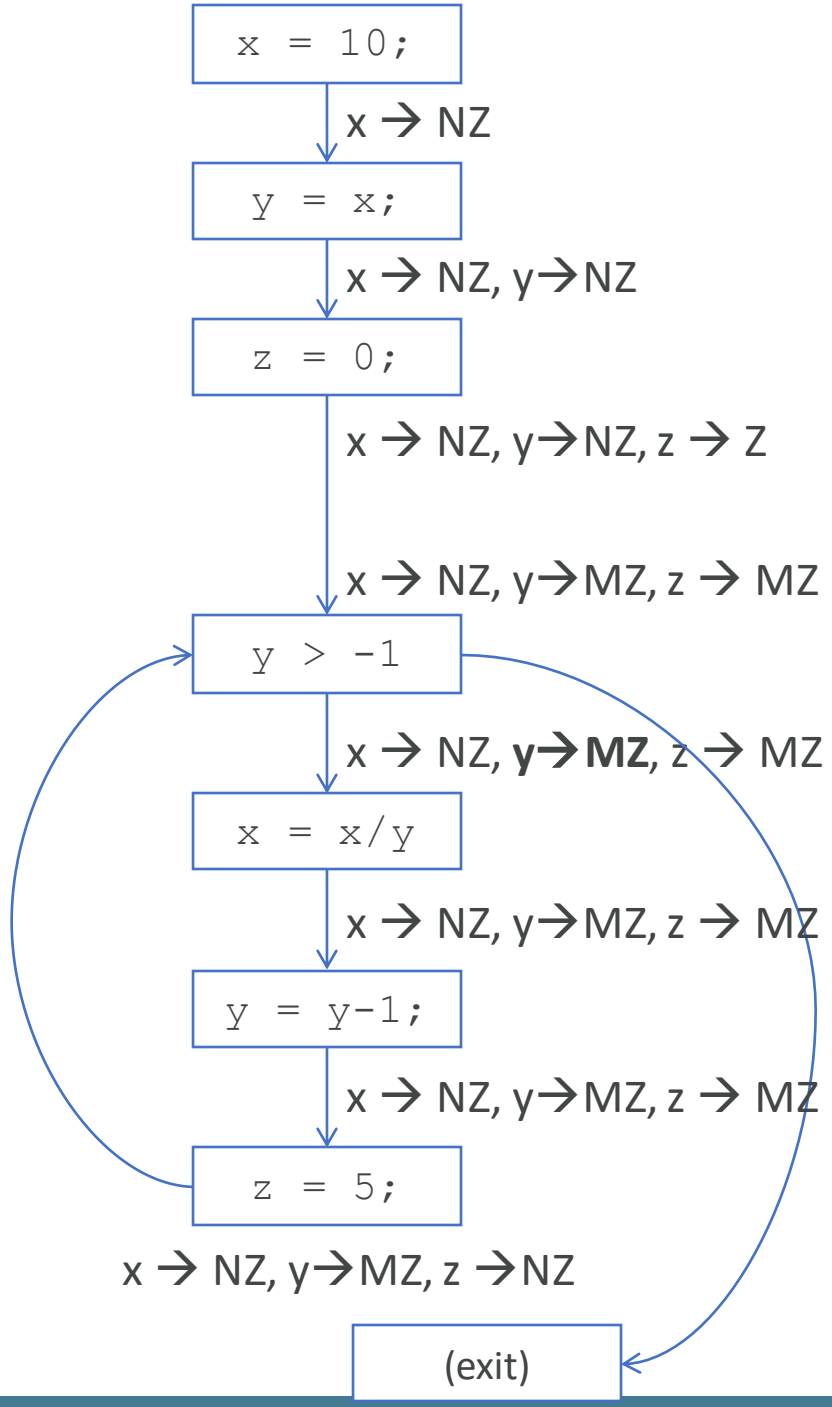


(end of iteration 2)



(end of iteration 3; nothing has changed)

Warning! Possible division by zero error!



Static Code Analysis

Summary

- Static analysis: systematic automated analysis of the code, without executing the program
  - Structural analysis: looking for patterns in code
  - Control Flow Analysis: Analyze all possible paths (global property)
  - Data Flow Analysis: Analyzing possible (abstract) values of variables on all paths
- All static analyses are *unsound* or *incomplete* or both