

Applied Static Analysis - Code Slicing

11.07.2019 – Patrick Müller

What is slicing?

- Reduction of program to relevant parts
- Goals:
 - Debugging
 - Reconstruct runtime Values
 - Maintenance
 - Testing
 - Clone Detection
 - Parallelization
- Here: Only static forward Slicing

Overview

- We want the **relevant** part of the Program
 - Relevant for what?

=> We define a **slicing criterion (sc)**

In most cases: Location and Variable

- Slice consists of all code elements that may affect the value of the variable at the given location
- Can have several criteria

Example

```
def foo(input: Boolean) -> Unit {  
  val test = !input  
  var a: Int  
  var b: Int  
  if (test) {  
    a = 2  
    b = 3  
  } else {  
    a = 10  
    b = 20  
  }  
  b = 30  
sc: println(a)  
sc: println(b)  
}
```

println(a);a

println(b);b

```
def foo(input: Boolean) -> Unit {  
  val test = !input  
  var a: Int  
  if (test) {  
    a = 2  
  } else {  
    a = 10  
  }  
  println(a)  
}
```

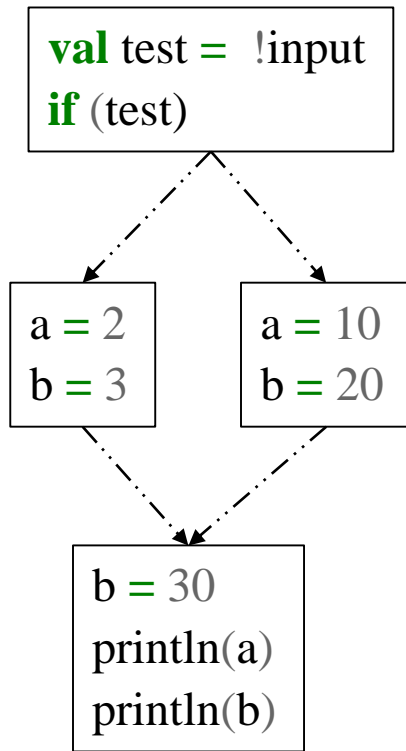
```
def foo() -> Unit {  
  var b: Int  
  b = 30  
  println(b)  
}
```

How to slice

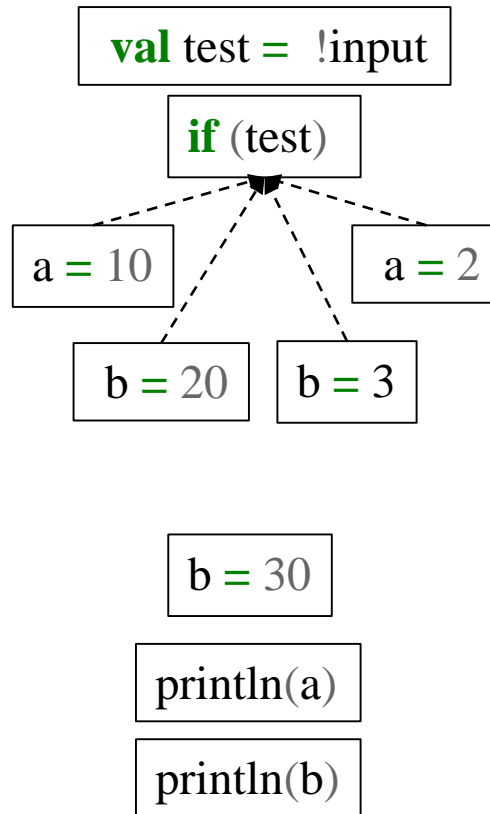
- We create a program dependence graph (pdg)
- Combination of
 - Control dependence graph
 - Data dependence graph
- Statements as vertices and 2 kinds of edges
 - Data dependency edges
 - Control dependency edges
- Slicing is backwards graph traversal from the slicing criterion

Example II

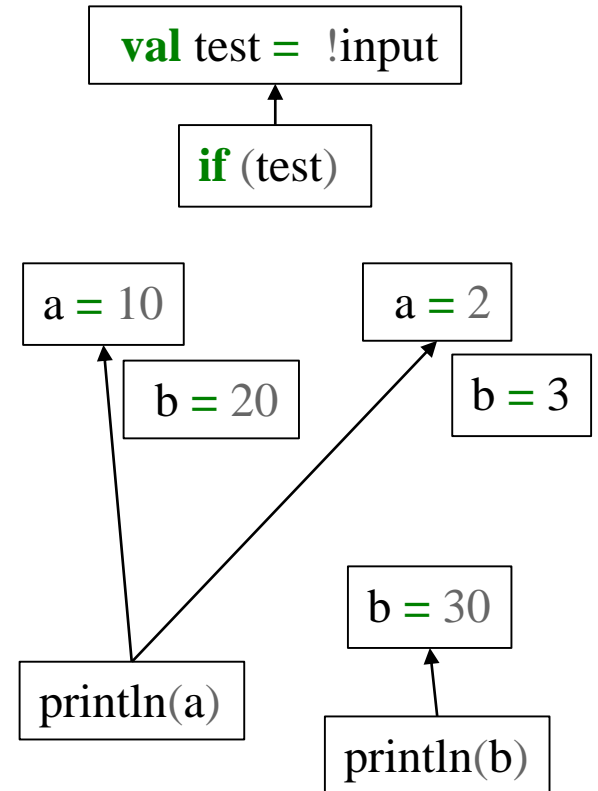
- control flow
- control dependency
- data dependency



control flow graph

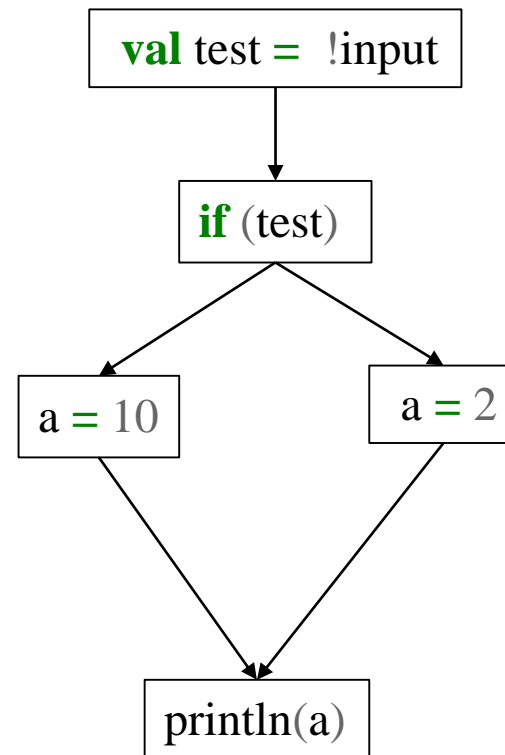
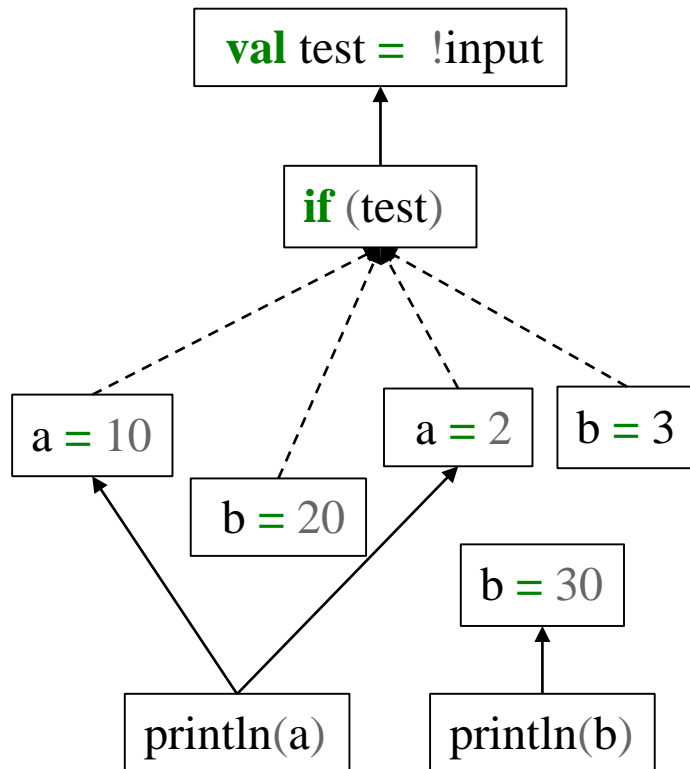


control dependence graph



data dependence graph

Program Dependence Graph



Program Dependence Graph II

- Only intraprocedural
- No Handling of Object Orientation

Reminder:

new **Object**()



NEW Object

DUP

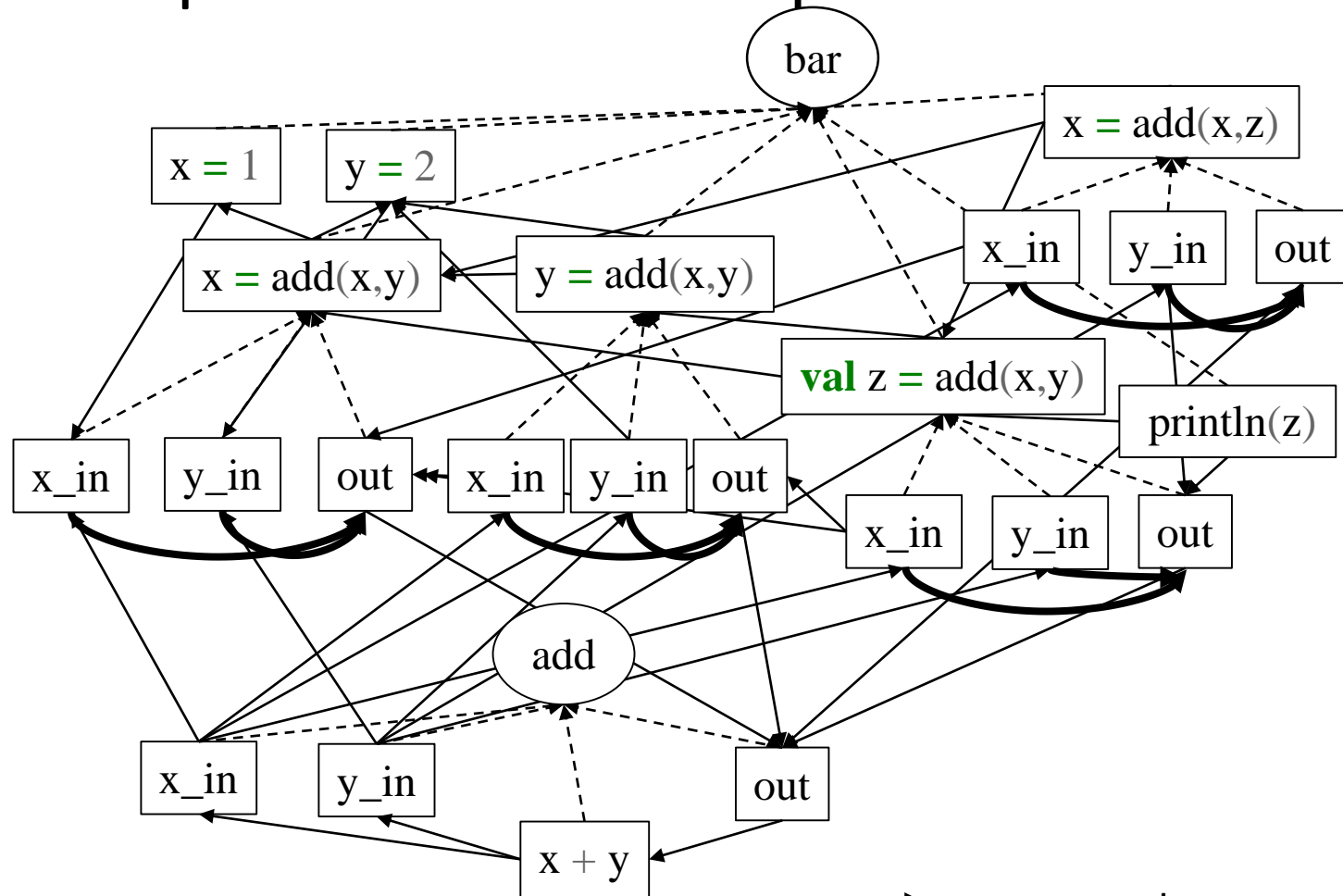
SPECIALINVOKE java/lang/**Object**.init()

System Dependence Graph

- Idea:
 - Augment PDG with functions and callsites
 - For each callsite add explicit vertices for parameters
 - Split graph traversal in two phases:
 - Upwards in the call stack
 - Downwards in the call stack

System Dependence Graph II

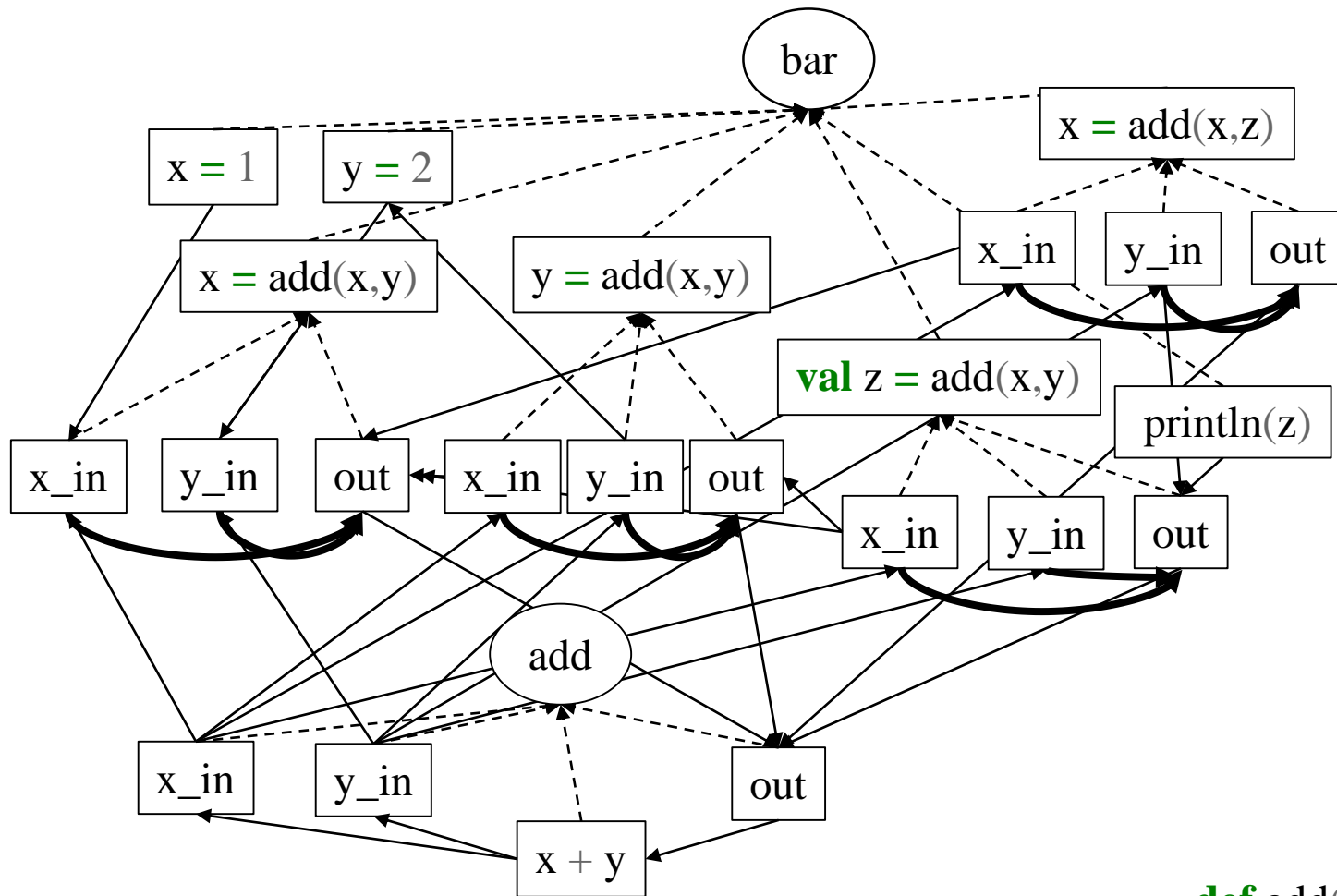
```
def bar() {
  var x = 1
  var y = 2
  x = add(x,y)
  y = add(x,y)
  val z = add(x,y)
  x = add(x,z)
  println(z)
}
```


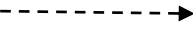
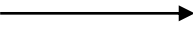


```
def add(x: Int, y: Int): Int =
  x + y
```

—————> summary edge
 - - - - -> control dependency
 —————> data dependency

System Dependence Graph III



 sum
 cont
 data

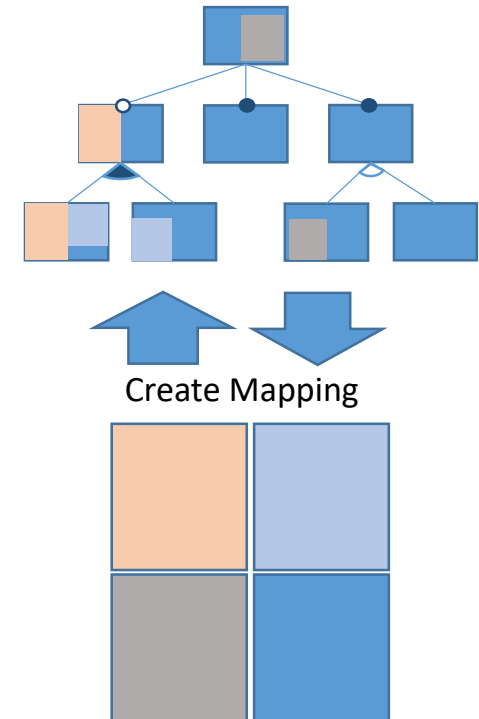
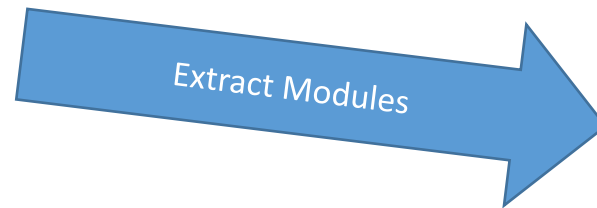
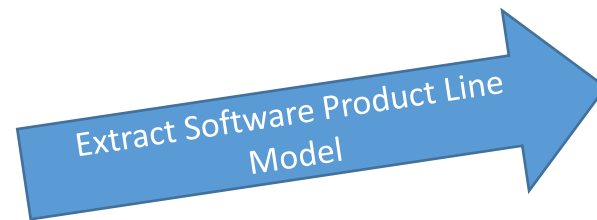
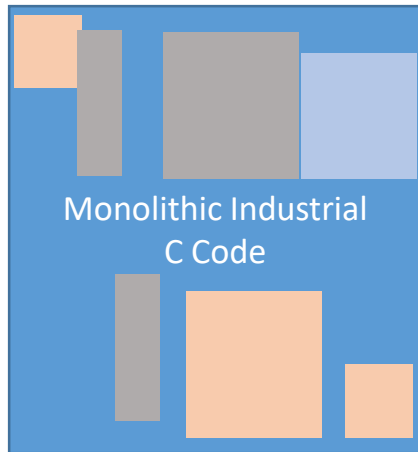
```
def bar() {
  var x = 1
  var y = 2
  x = add(x,y)
  y = add(x,y)
  val z = add(x,y)
  println(z)
}
```

```
def add(x: Int, y: Int): Int =
  x + y
```

Object Oriented Slicing

- Add nodes for classes to be able to represent dependency
- Add in/out vertices at callsites for fields and global variables
- Callsites can be polymorphic – use callgraph for candidate methods

Software Factory 4.0



Sources

- Susan Horwitz, Thomas Reps, and David Binkley. 1990. Interprocedural slicing using dependence graphs. ACM Trans. Program. Lang. Syst. 12, 1 (January 1990), 26-60. DOI=<http://dx.doi.org/10.1145/77606.77608>
- Loren Larsen and Mary Jean Harrold. 1996. Slicing object-oriented software. In Proceedings of the 18th international conference on Software engineering (ICSE '96). IEEE Computer Society, Washington, DC, USA, 495-505.
- Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. ACM Trans. Program. Lang. Syst. 9, 3 (July 1987), 319-349. DOI: <https://doi.org/10.1145/24039.24041>
- A. De Lucia, "Program slicing: methods and applications," Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, Florence, Italy, 2001, pp. 142-149.doi: 10.1109/SCAM.2001.972675