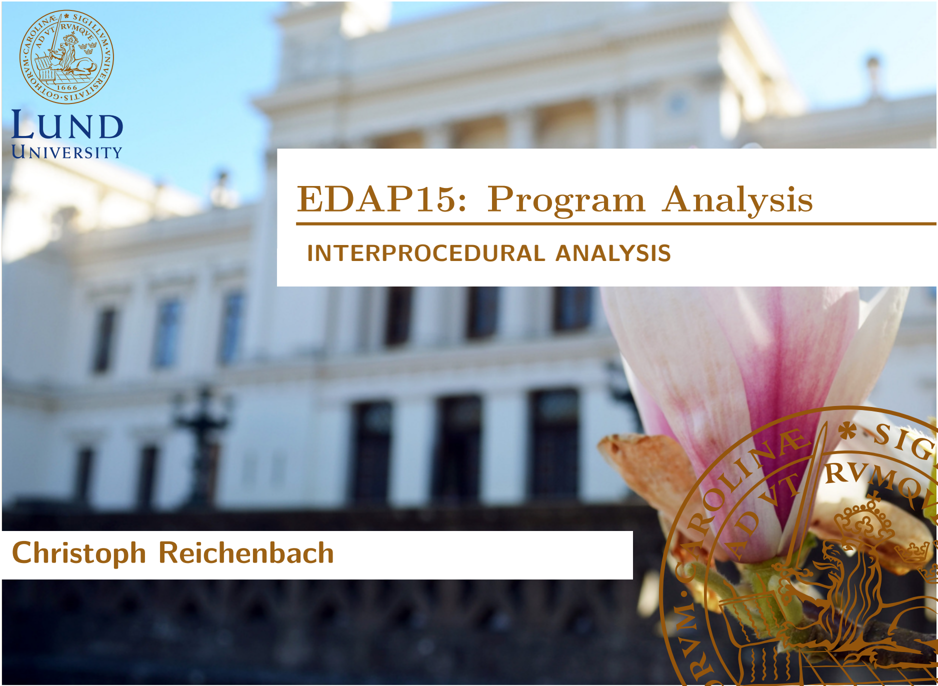


EDAP15: Program Analysis

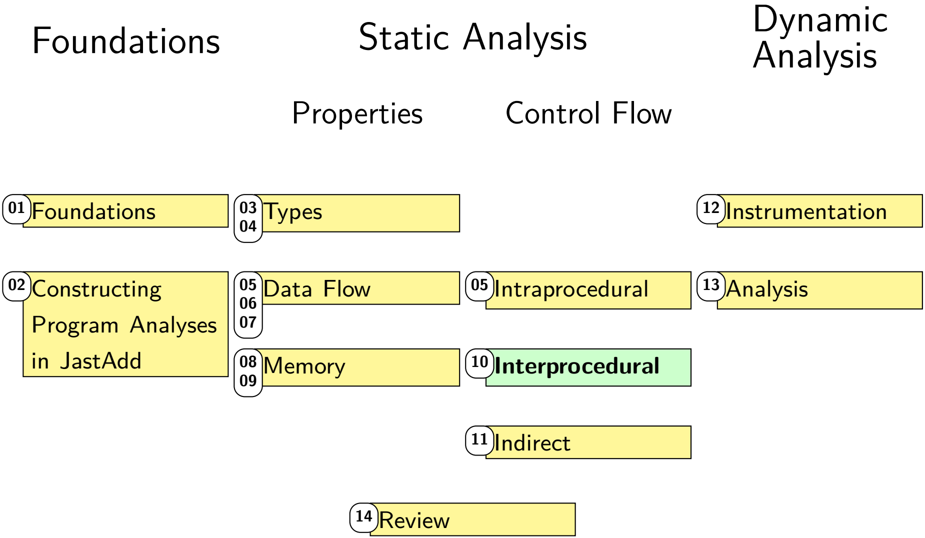
INTERPROCEDURAL ANALYSIS

Christoph Reichenbach

Welcome back!

Questions?

Lecture Overview



Foundations

Static Analysis

Dynamic Analysis

Properties

Control Flow

01 Foundations

02 Constructing Program Analyses in JastAdd

03 04 Types

05 06 07 Data Flow

08 09 Memory

05 Intraprocedural

10 **Interprocedural**

11 Indirect

12 Instrumentation

13 Analysis

14 Review

What about subroutines?

Teal

```
var x := max(0, 5);
print(10 / x); // Division by zero?
```

- Understanding code usually requires understanding subroutines like `max`

Inter- vs. Intra-Procedural Analysis

- **Intra**procedural: Within one procedure
  - Data flow analysis so far
- **Inter**procedural: Across multiple procedures
  - Type Analysis, especially. with polymorphic type inference

Limitations of Intra-Procedural Analysis

Teal-0

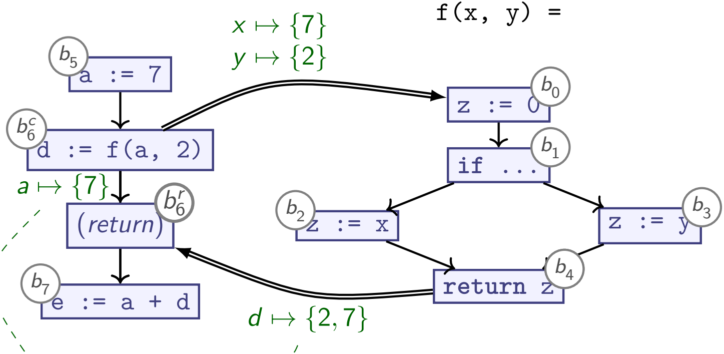
```
a := 7;
d := f(a, 2);
e := a + d;
```

Teal-0

```
fun f(x, y) = {
  var z := 0;
  if x > y {
    z := x;
  } else {
    z := y;
  }
  return z;
}
```

How can we compute Reachable Definitions here?

A Naïve Inter-Procedural Analysis



f(x, y) =

$x \mapsto \{7\}$

$y \mapsto \{2\}$

b_5 a := 7

b_6^c d := f(a, 2)

$a \mapsto \{7\}$

b_6^r *(return)*

b_7 e := a + d

b_0 z := 0

b_1 if ...

b_2 z := x

b_3 z := y

b_4 return z

$d \mapsto \{2, 7\}$

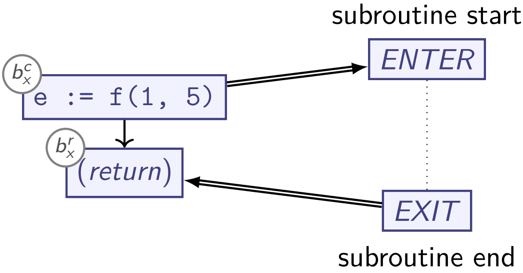
omitting ‘obvious’ transfer functions

$$\mathbf{in}_{b_6^r} = \mathbf{out}_{b_6} \sqcup \mathbf{out}_{b_4} = \mathbf{out}_{b_6^r}$$

- $\mathbf{out}_{b_7}$: $e \mapsto \{9, 14\}$

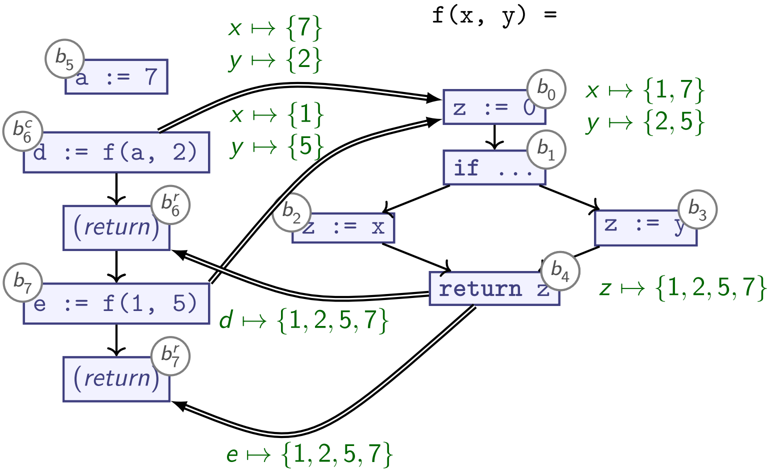
Works rather straightforwardly!

Inter-Procedural Control Flow Graph



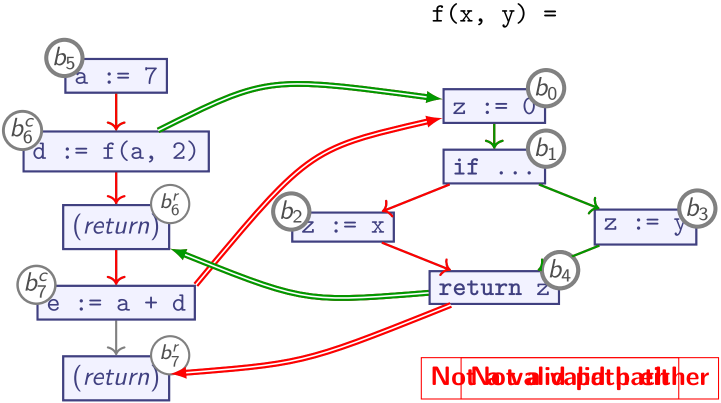
- Split call sites b_x into *call* (b_x^c) and *return* (b_x^r) nodes
- Intra-procedural edge $b_x^c \longrightarrow b_x^r$ carries environment/store
- Inter-procedural edge ($\Rrightarrow$):
  - Call site $\Rrightarrow$ callee: substitutes parameters
  - Call site $\Lleftarrow$ return: substitutes result
  - Otherwise like intra-procedural data flow edge

A Naïve Inter-Procedural Analysis



Imprecision!

Valid Paths



Not a valid path

Not valid path either

- $[b_5, b_6^c, b_0, b_1, b_3, b_4, b_6^r]$

Context-sensitive interprocedural analyses consider only valid paths

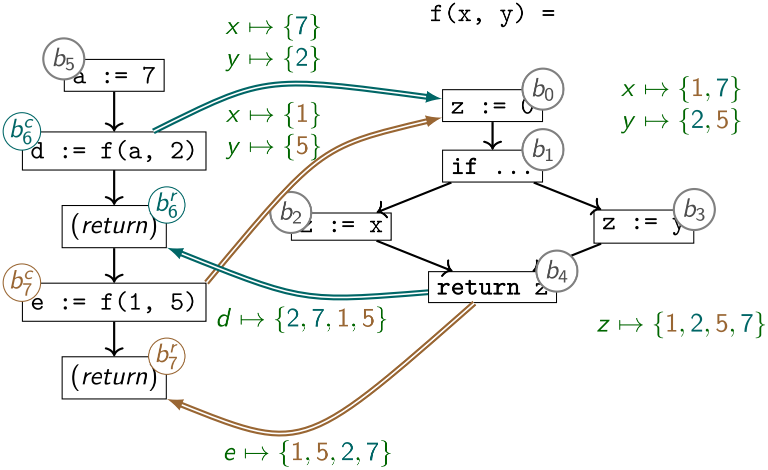
Summary

- **Intraprocedural** Analysis:
    - Considers one subroutine at a time
    - Calls to other subroutines treated as “worst-case” (e.g., $\top$ for dataflow analysis)
- **Interprocedural** Analysis:
    - Analyses calls to subroutines
    - For Dataflow analysis: uses **Interprocedural CFG** (ICFG)
        - ICFG represents subroutine calls as two nodes: **call** and **return**
        - Special Call/Return edges caller $\Leftrightarrow$ callee
        - Naïve interpretation of ICFG call/return edges “spills” analysis results across call sites

Interprocedural Data Flow Analysis

- **Call-site insensitive**
  - Use same abstraction for each call site
  - Examples for dataflow analysis:
    - Treat ICFG call/return edges like "regular" call/return edges
    - Use same transfer function everywhere (e.g., for builtin functions)
- **Call-site sensitive**
  - Use different abstractions at different call sites

Call-Site Insensitive Analysis



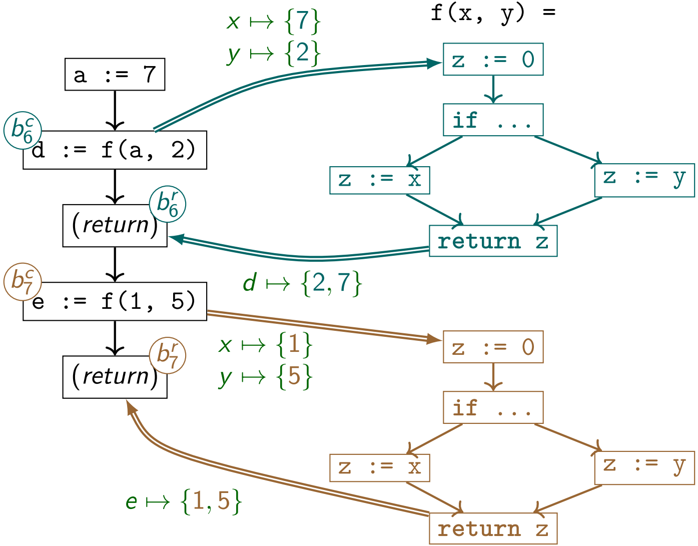
***Call-site insensitive*: analysis merges all callers to `f()`**

Precise Interprocedural Dataflow

- Precision via one of:

1. **Inlining** or **AST cloning**
2. Call Strings
3. Procedure Summaries

Inlining



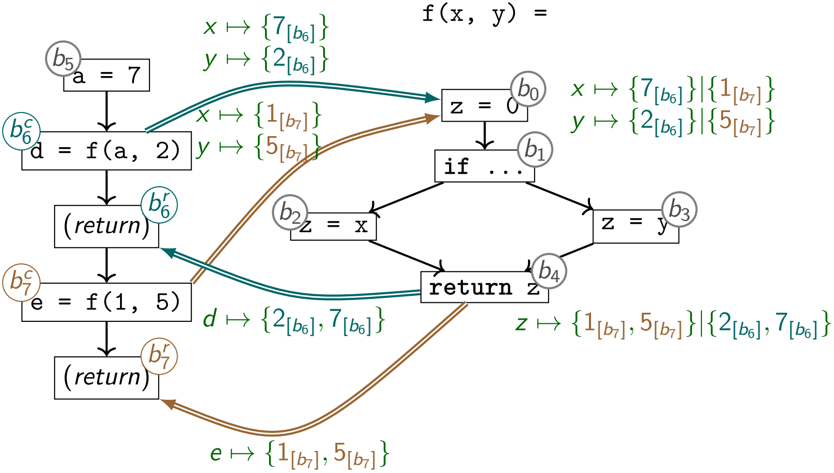
Clone subroutine IRs for each *calling context*

Precise Interprocedural Dataflow

- Precision via one of:

1. Inlining or AST cloning
2. **Call Strings**
3. Procedure Summaries

Call Strings of Length 1



f(x, y) =

b_5 a = 7

b_6^c d = f(a, 2)

b_6^r (*return*)

b_7^c e = f(1, 5)

b_7^r (*return*)

b_0 z = 0

b_1 if ...

b_2 z = x

b_3 z = y

b_4 return z

$x \mapsto \{7_{[b_6]}\}$
$y \mapsto \{2_{[b_6]}\}$

$x \mapsto \{1_{[b_7]}\}$
$y \mapsto \{5_{[b_7]}\}$

$x \mapsto \{7_{[b_6]}\}|\{1_{[b_7]}\}$
$y \mapsto \{2_{[b_6]}\}|\{5_{[b_7]}\}$

$d \mapsto \{2_{[b_6]}, 7_{[b_6]}\}$

$z \mapsto \{1_{[b_7]}, 5_{[b_7]}\}|\{2_{[b_6]}, 7_{[b_6]}\}$

$e \mapsto \{1_{[b_7]}, 5_{[b_7]}\}$

Degrees of Call-Site Sensitivity

- We used *call strings* to make call sites explicit:
  - $[b_6]$ in $2_{[b_6]}$
- “Strings” because this idea generalises:
  - Can keep track of *multiple* callers
  - Example: *2-call-site sensitivity*: $[b_0, b_6]$ vs $[b_1, b_6]$

Teal

```
fun g(y:  int):  int = { return y }
fun f(x:  int):  int = {
  return g(x) // $b_6$
       + g(5); // $b_7$
}
...
  f(1); // $b_0$
  f(2); // $b_1$
```

Must bound length of call strings to ensure termination

Summary

Strategies for call-site sensitive analysis:

- **Inlining**
  - Copy subroutine bodies for each caller
  - Performance cost
  - Recursion: fall back to $\top$
- **Call Strings**
  - Call string length:
    - Unbounded: Maximum precision, may not terminate with recursion
    - Bounded to length k: k degrees of call site sensitivity (speed/precision trade-off)

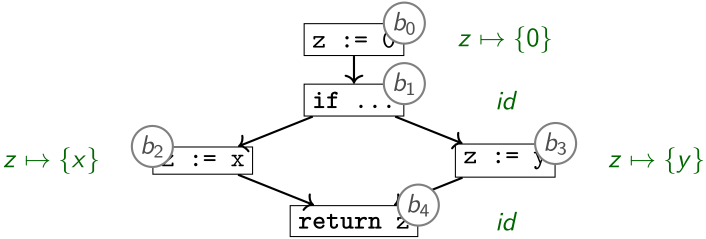
Precise Interprocedural Dataflow

- Precision via one of:

1. Inlining or AST cloning
2. Call Strings
3. **Procedure Summaries**

Summarising Procedures

`f(x, y) =`



- *Compose* transfer functions:
  - $\mathit{trans}_{b_0} \circ \mathit{trans}_{b_1} = [z \mapsto 0]$
  - $\mathit{trans}_{b_0} \circ \mathit{trans}_{b_1} \circ \mathit{trans}_{b_2} = [z \mapsto \{x\}]$
  - $\mathit{trans}_{b_0} \circ \mathit{trans}_{b_1} \circ \mathit{trans}_{b_3} = [z \mapsto \{y\}]$
  - $\mathit{trans}_{b_0} \circ \mathit{trans}_{b_1} \circ (\mathit{trans}_{b_2} \sqcup \mathit{trans}_{b_3}) = [z \mapsto \{x, y\}]$
  - $\mathit{trans}_{b_0} \circ \mathit{trans}_{b_1} \circ (\mathit{trans}_{b_2} \sqcup \mathit{trans}_{b_3}) \circ \mathit{trans}_{b_4} = [z \mapsto \{x, y\}]$

Procedure Summaries vs Recursion

`f` calls `g` calls `h` calls `f`

- Reqiures additional analysis to identify who calls whom
- Compute summaries of mutually recursive functions together
- Recursive call edges analogous to loops

Procedure Summaries

- Composing transfer functions yields a combined transfer function for `f()`:

$$\mathit{trans}_f = [\textbf{return} \mapsto \{x, y\}]$$

- Use trans_f as transfer function for `f()`, discard `f`'s body
- **Opportunities:**
  - Can yield compact subroutine descriptions
  - Can speed up call site analysis dramatically
- **Challenges:**
  - More complex to implement
  - Recursion remains challenging
- **Limitations:**
  - Requires suitable representation for summary
  - Requires mechanism for abstracting and applying summary
  - Worst cases:
    - trans_f is symbolic expression more complex than `f` itself

Procedure Summaries for Dataflow

- Procedure Summaries *can* be as precise as inlining/call strings

. . . *but only for Distributive Frameworks*

  - Algorithm for Gen/Kill analyses: IFDS
  - Algorithm for other analyses: IDE

Summary

Making interprocedural dataflow precise:

- **Call-site sensitive approaches:**
  - *Inlining*
  - *Call strings*
- **Call-site insensitive approaches:**
  - *Procedure Summaries*
    - Precise + compact summaries only possible for distributive frameworks

Outlook

- More static analysis on Monday
- Exercise 3 will go up tomorrow
- Exercise 4 (next week):
  - can run via *podman* (on lab computers)
  - will also offer Docker image

`http://cs.lth.se/EDAP15`