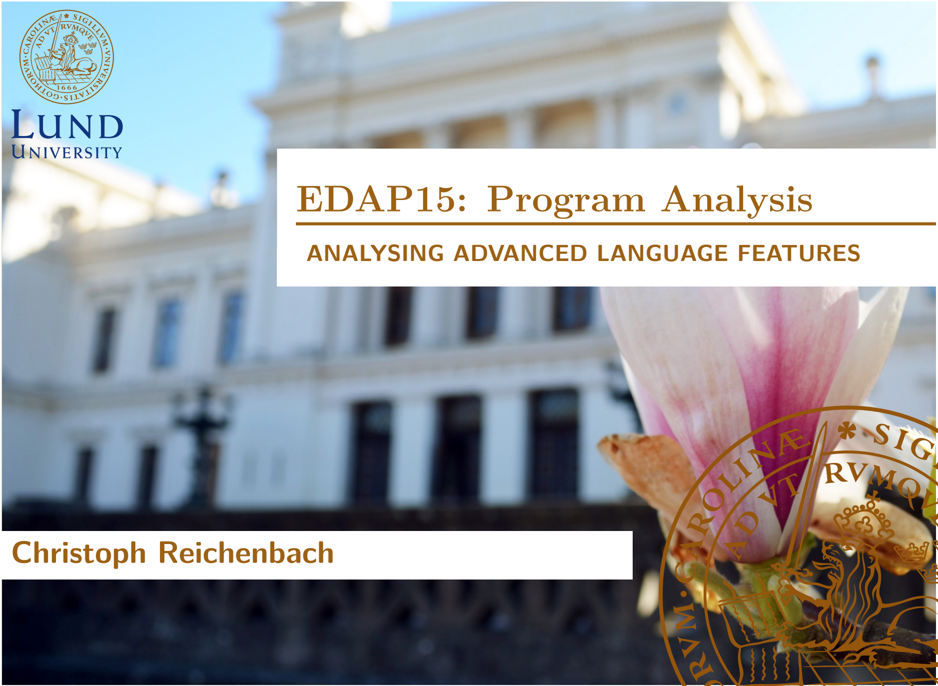


EDAP15: Program Analysis

ANALYSING ADVANCED LANGUAGE FEATURES

Christoph Reichenbach

Lund University

Welcome back!

- Quick presentation about CodeProber user studies in break by Anton
- Homework Exercise 1 update:
  - Can present in office hours today if you have already presented exercise 0
  - Can present in office hours next week if you have already presented exercises 0 & 2
- Homework Exercie 4 update:
  Will reqiure one of:
  - `podman` (available in Linux lab rooms in E-huset)
  - `docker`
  - Local installation & build of C programs on CLI (Linux, OS X, *BSD, WSL, any recent-ish Unix)

Questions?

Lecture Overview

Foundations	Static Analysis: Properties	Static Analysis: Control Flow	Dynamic Analysis
01 Foundations	03 04 Types		12 Instrumentation
02 Constructing Program Analyses in JastAdd	05 06 07 Data Flow	05 Intraprocedural	13 Analysis
	08 09 Memory	10 Interprocedural	
		11 Indirect	

14 Review

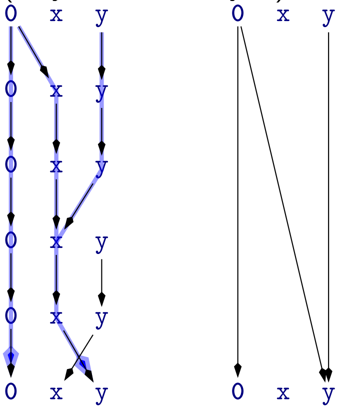
Composing Representation Relations

Representation Relations (*may be null* analysis):

```
x := null;
y := y;
```

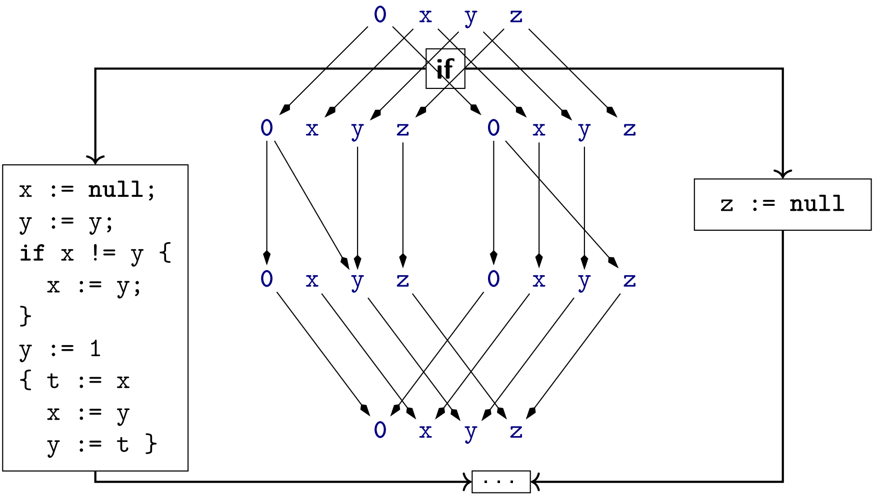
```
if x != y {
  x := y;
}
y := 1;
```

```
{ t := x;
  x := y;
  y := t; }
```

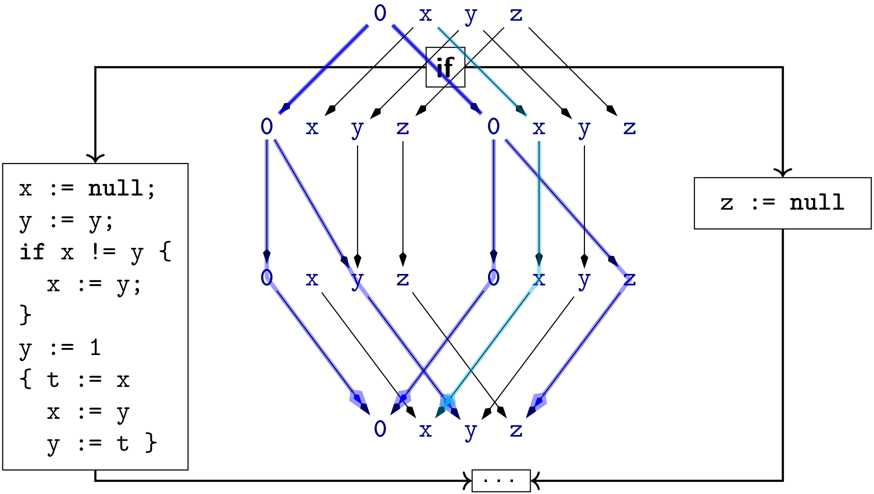


Composed representation relations are again representation relations

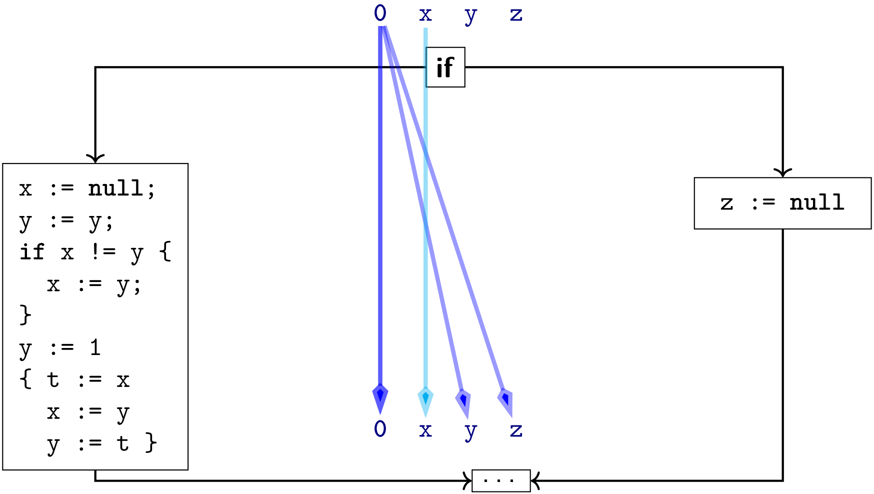
Joining Control-Flow Paths



Joining Control-Flow Paths



Joining Control-Flow Paths



Logical "Or"

Dataflow via Graph Reachability

$$n = \langle b, v \rangle$$

- Assume binary latice $(\{\top, \bot\}, \sqsubseteq, \sqcap, \sqcup)$
  - $\top \sqcup y = \top = x \sqcup \top$ and $\bot \sqcup \bot = \bot$
  - Typical for 'May' analysis ($P(x)$ = 'x may be `null`')

- Encode Dataflow problem as *Graph-Reachability*
- Graph nodes $n = \langle b, v \rangle$
  - b: CFG node
  - v: Variable or $\mathbf{0}$
    - $\mathbf{0}$: $\langle b_1, \mathbf{0} \rangle \longrightarrow \langle b_2, y \rangle$: $P(y)$ at b_2 holds always
    - Variable: $\langle b_1, x \rangle \longrightarrow \langle b_2, y \rangle$: $P(x)$ at $b_1 \implies P(y)$ at b_2

Dataflow via Graph Reachability

$$n = \langle b, v \rangle$$

- Assume binary latice $(\{\top, \bot\}, \sqsubseteq, \sqcap, \sqcup)$
  - $\top \sqcup y = \top = x \sqcup \top$ and $\bot \sqcup \bot = \bot$
  - Typical for 'May' analysis ($P(x) =$ 'x may be `null`')
  - Equivalently for 'Must' analysis:
    'x must be `null`' = not ('x may be `non-null`')
- Encode Dataflow problem as *Graph-Reachability*
- Graph nodes $n = \langle b, v \rangle$
  - b: CFG node
  - v: Variable or $\mathbf{0}$
    - $\mathbf{0}$: $\langle b_1, \mathbf{0} \rangle \longrightarrow \langle b_2, y \rangle$: $P(y)$ at b_2 holds always
    - Variable: $\langle b_1, x \rangle \longrightarrow \langle b_2, y \rangle$: $P(x)$ at $b_1 \implies P(y)$ at b_2

A Dataflow Worklist Algorithm: IFDS

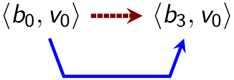
- Call-site sensitive interprocedural data flow algorithm
- IFDS = (**I**nterprocedural **F**inite **D**istributive **S**ubset problems)
- 'Exploded Supergraph': $G^\sharp = (N^\sharp, E^\sharp)$
  - $N^\sharp = N_{\mathsf{CFG}} \times (\mathcal{V} \cup \{0\})$
  - Plus parameter/return call edges
- Property-of-interest holds if reachable from $\langle b^s_{\mathsf{main}}, \mathbf{0} \rangle$
  - b^s_{main} is CFG *ENTER* node of main entry point
- **Key ideas**:
  - Worklist-based
  - Construct Representation Relations on demand
  - Construct 'Exploded Supergraph'
    - CFG of all functions $\times\ \mathcal{V} \cup \{\mathbf{0}\}$

IFDS Datastructures

Instead of $\langle\langle b_0, v_0\rangle, \langle b_3, v_0\rangle\rangle$ we also write:

$$\langle b_0, v_0\rangle \rightarrow \langle b_3, v_0\rangle$$

WorkList edge — All WorkList edges are also PathEdge edges



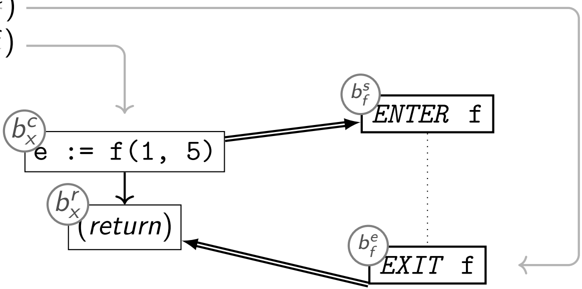
PathEdge edge — Result of our analysis

$N^\sharp$-edge

SummaryInst — Generated from summary nodes
Otherwise equivalent to $N^\sharp$-edges

IFDS Strategy

- Algorithm distinguishes between three types of nodes:
  - *Exit* nodes (b_f^e)
  - *Call* nodes (b_x^c)
  - Other nodes



On-demand processing

Procedure propagate($n_1 \to n_2$):
begin
  if $n_1 \to n_2 \in \textsc{PathEdge}$ **then**
    return
  $\textsc{PathEdge} := \textsc{PathEdge} \cup \{n_1 \to n_2\}$
  $\textsc{WorkList} := \textsc{WorkList} \cup \{n_1 \to n_2\}$
end

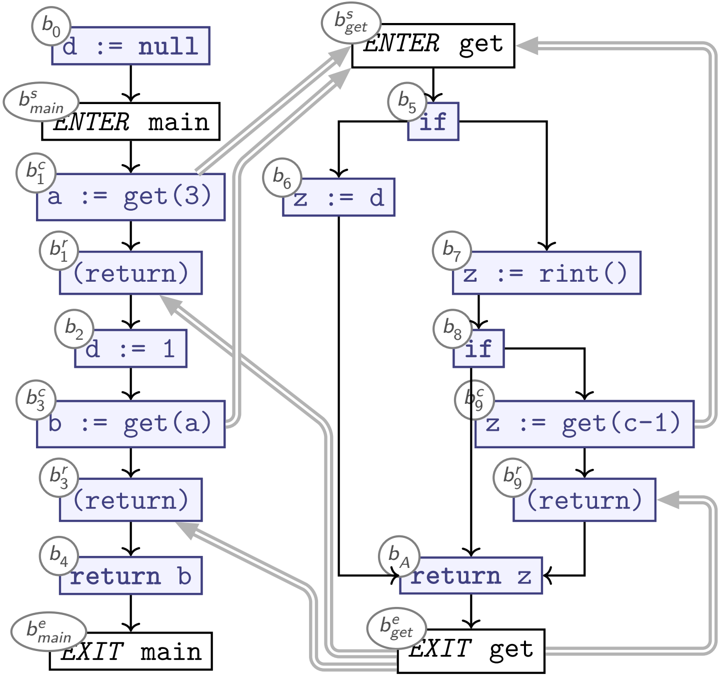
Running Example

Teal-0: *main()*

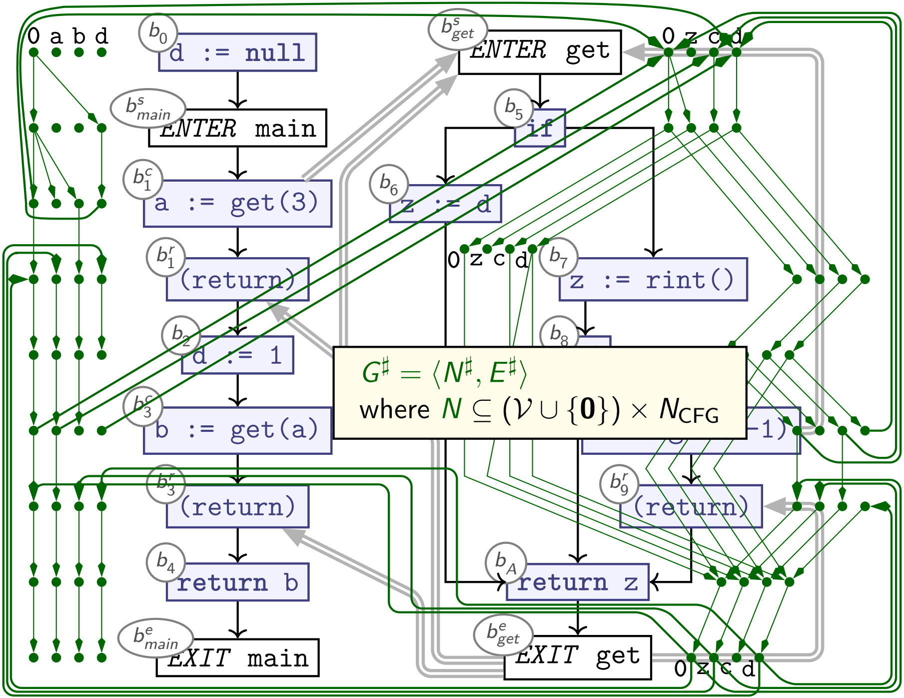
```
var default := null;
fun main() = {
  var a := get(3);
  default := 1;
  var b := get(3);
  return b;
}
```

Teal-0: *get()*

```
fun get(c) = {
  if c == 0 {
    z := default;
  } else {
    z := read_int();
    if z < 0 {
      z := get(c - 1);
    }
  }
  return z;
}
```

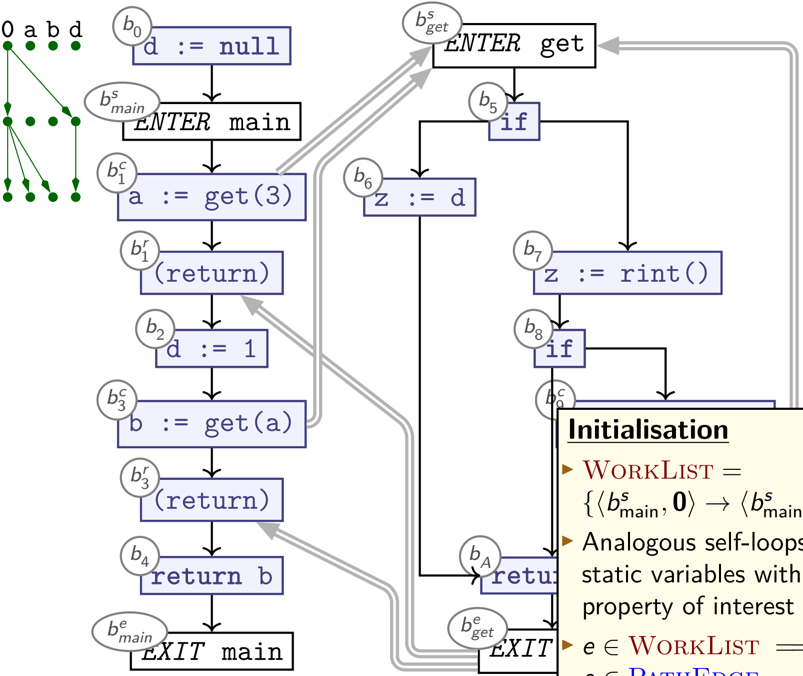


b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z := d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get



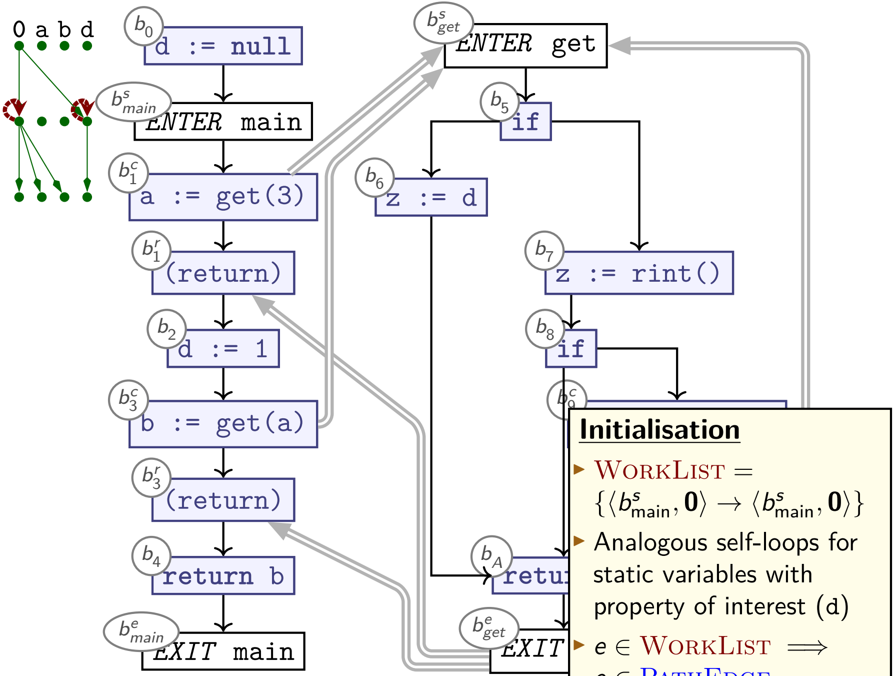
$G^\sharp = \langle N^\sharp, E^\sharp \rangle$

where $N \subseteq (\mathcal{V} \cup \{\mathbf{0}\}) \times N_{\mathsf{CFG}}$



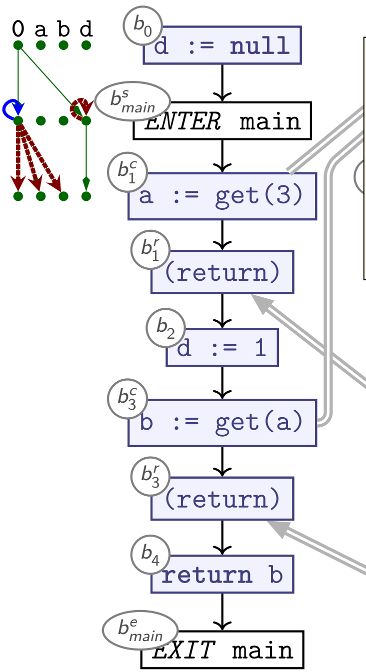
Initialisation

- WorkList = $\{\langle b^s_{\text{main}}, \mathbf{0}\rangle \rightarrow \langle b^s_{\text{main}}, \mathbf{0}\rangle\}$
- Analogous self-loops for static variables with property of interest (d)
- $e \in$ WorkList $\Longrightarrow$ $e \in$ PathEdge



Initialisation

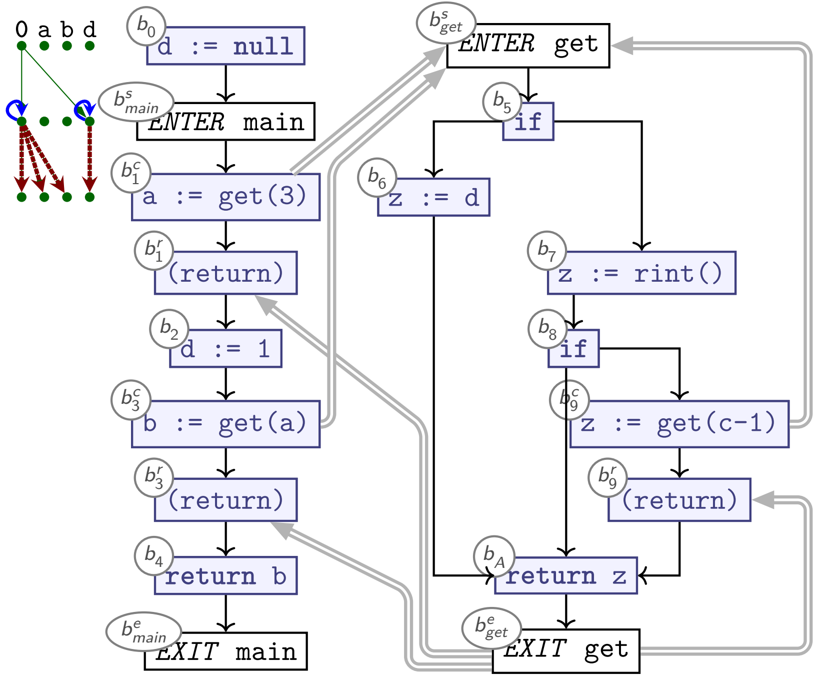
- WorkList = $\{\langle b^s_{\text{main}}, \mathbf{0}\rangle \rightarrow \langle b^s_{\text{main}}, \mathbf{0}\rangle\}$
- Analogous self-loops for static variables with property of interest (d)
- $e \in$ WorkList $\implies$ $e \in$ PathEdge



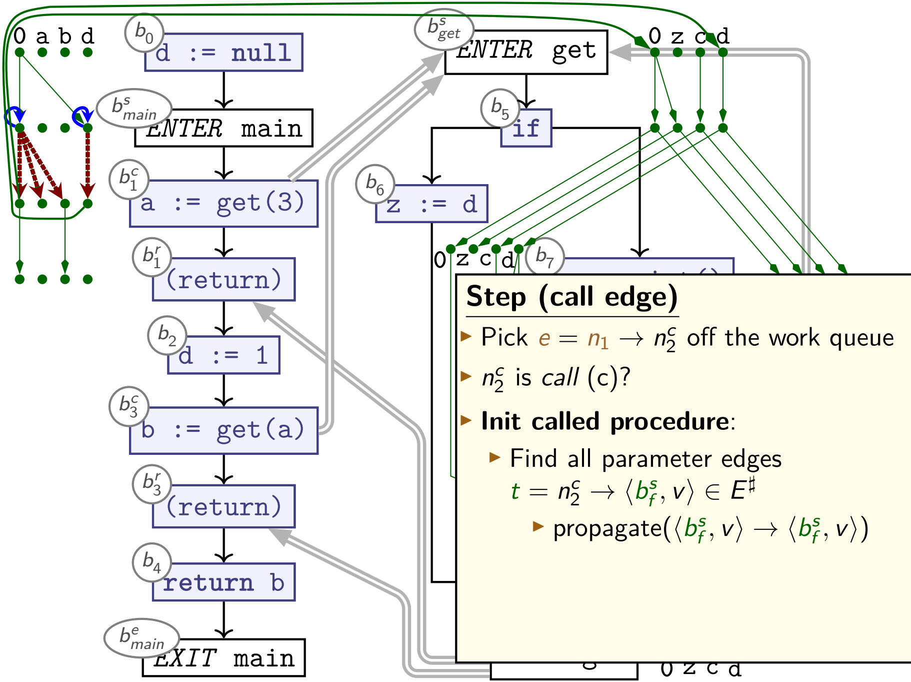
Procedure propagate($n_1 \to n_2$):
begin
 if $n_1 \to n_2 \in$ PathEdge **then**
  return
 PathEdge := PathEdge $\cup \{n_1 \to n_2\}$
 WorkList := WorkList $\cup \{n_1 \to n_2\}$
end

Step (regular edge)

- Pick e off the work queue
  $e = n_1 \to n_2$
- n_2 neither call (c) nor exit (e)?
- Find all $n_2 \to n_3$
  propagate($n_1 \to n_3$)
- Remove e from WorkList
- e remains in PathEdge

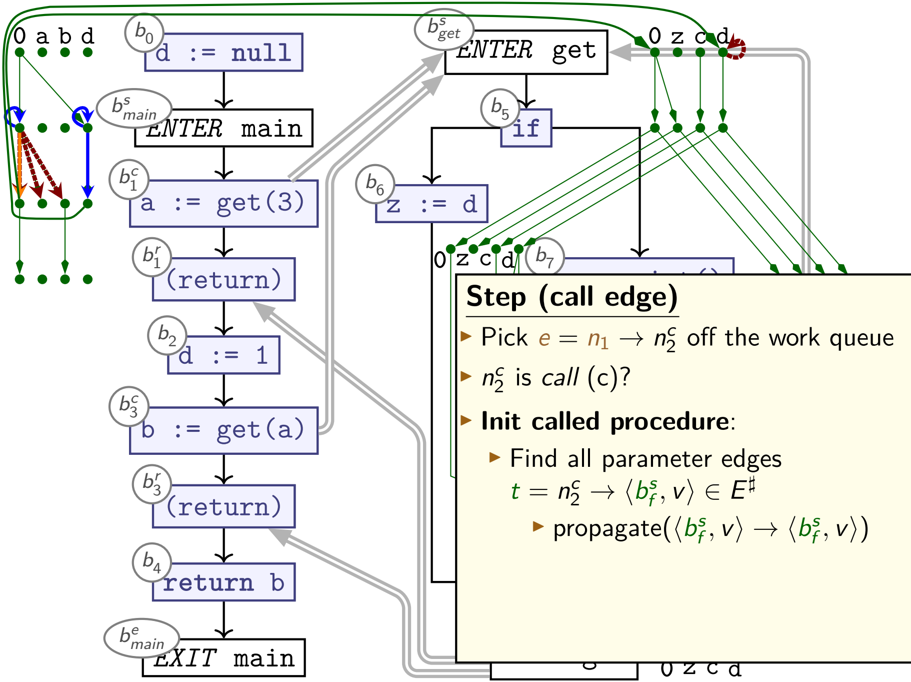


0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z := d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get



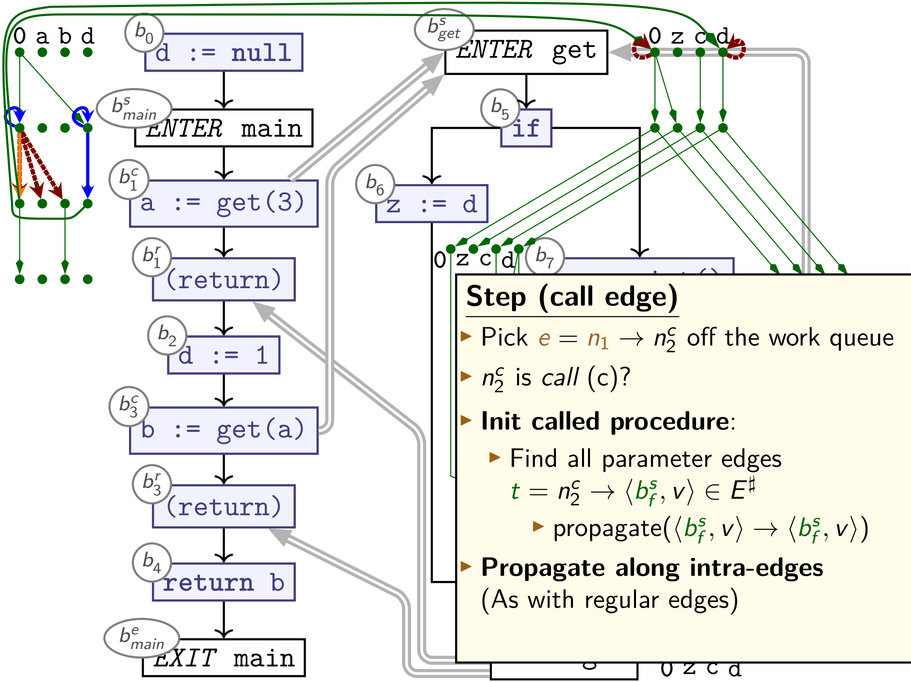
Step (call edge)

- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v\rangle \in E^\sharp$
    - propagate($\langle b_f^s, v\rangle \to \langle b_f^s, v\rangle$)



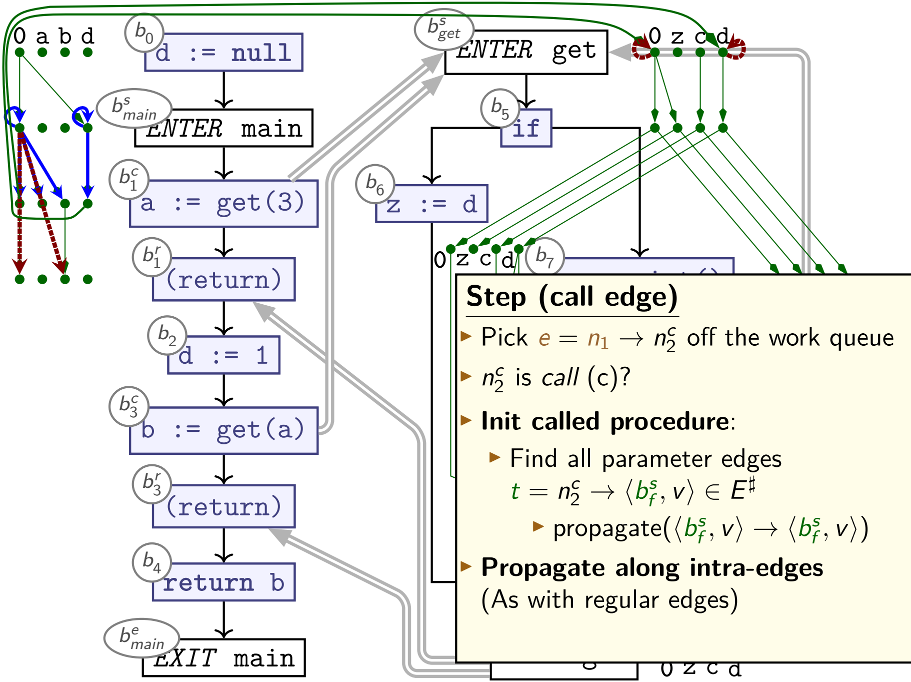
Step (call edge)

- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \to \langle b_f^s, v \rangle$)



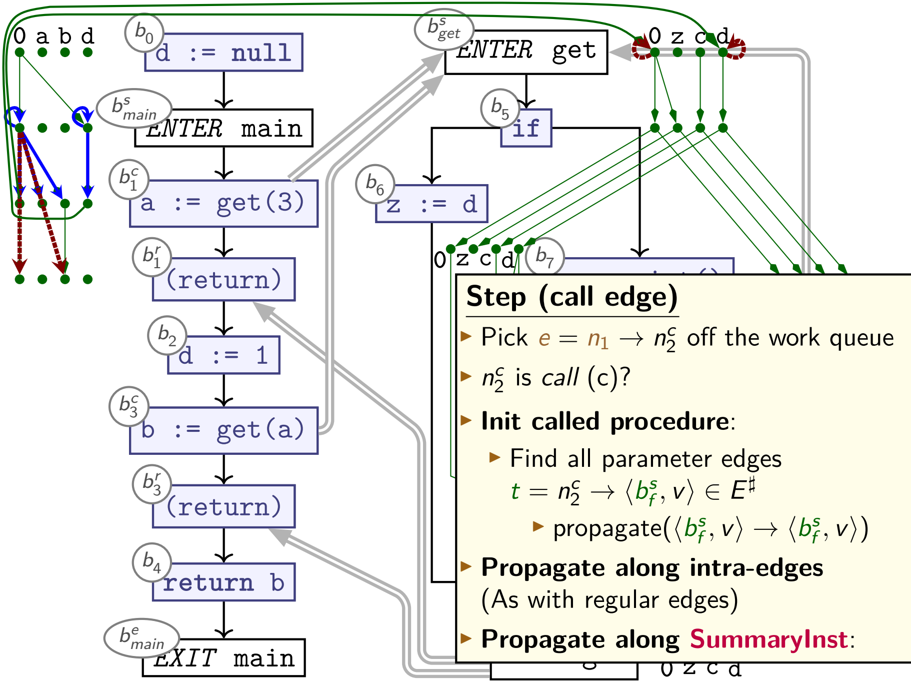
Step (call edge)

- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v\rangle \in E^\sharp$
    - propagate($\langle b_f^s, v\rangle \to \langle b_f^s, v\rangle$)
- **Propagate along intra-edges** (As with regular edges)



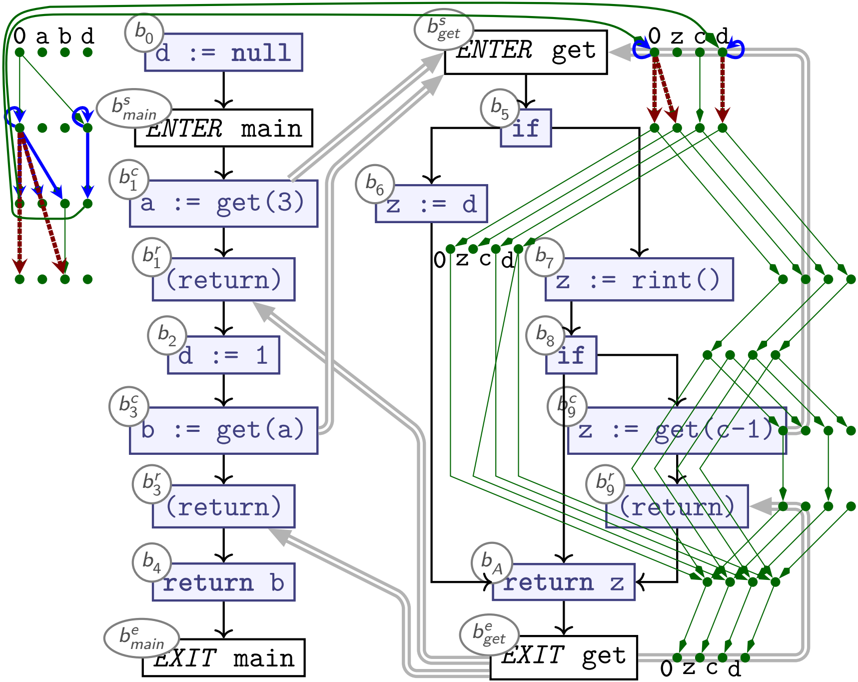
Step (call edge)

- Pick $e = n_1 \rightarrow n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \rightarrow \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \rightarrow \langle b_f^s, v \rangle$)
- **Propagate along intra-edges** (As with regular edges)

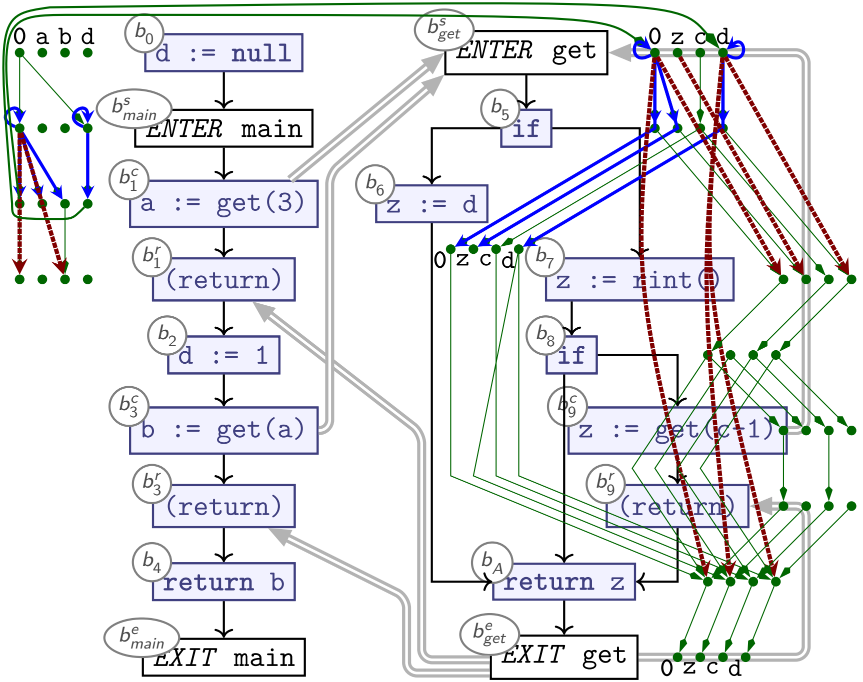


Step (call edge)

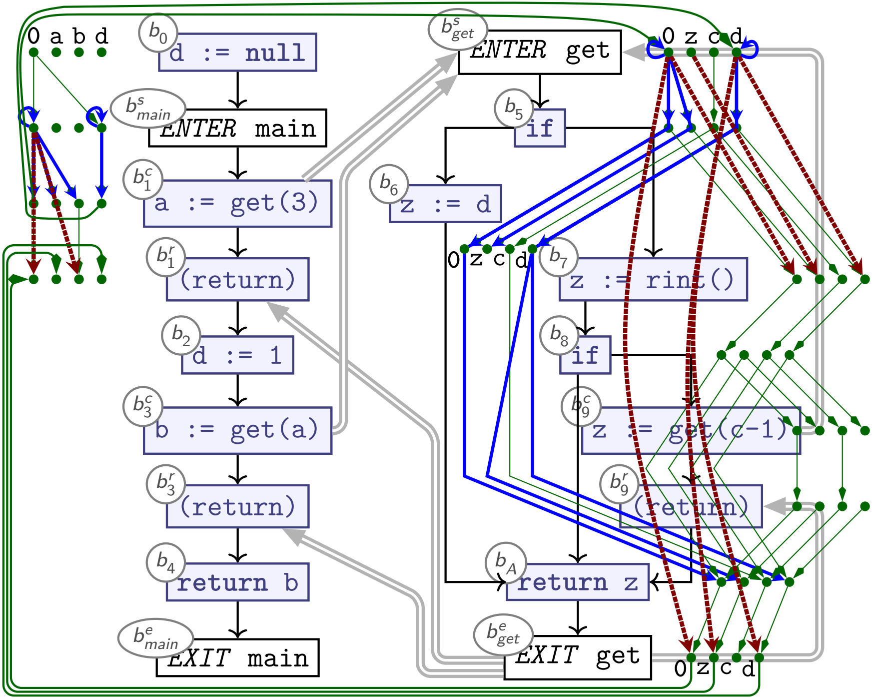
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \to \langle b_f^s, v \rangle$)
- **Propagate along intra-edges** (As with regular edges)
- **Propagate along SummaryInst**:



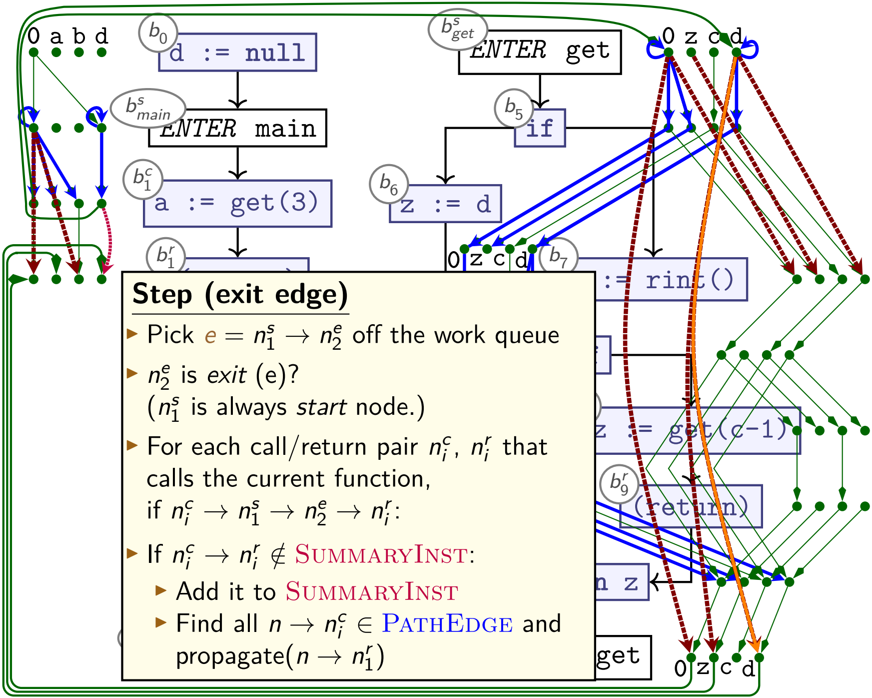
0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
0 z c d
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d



0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
0 z c d
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d

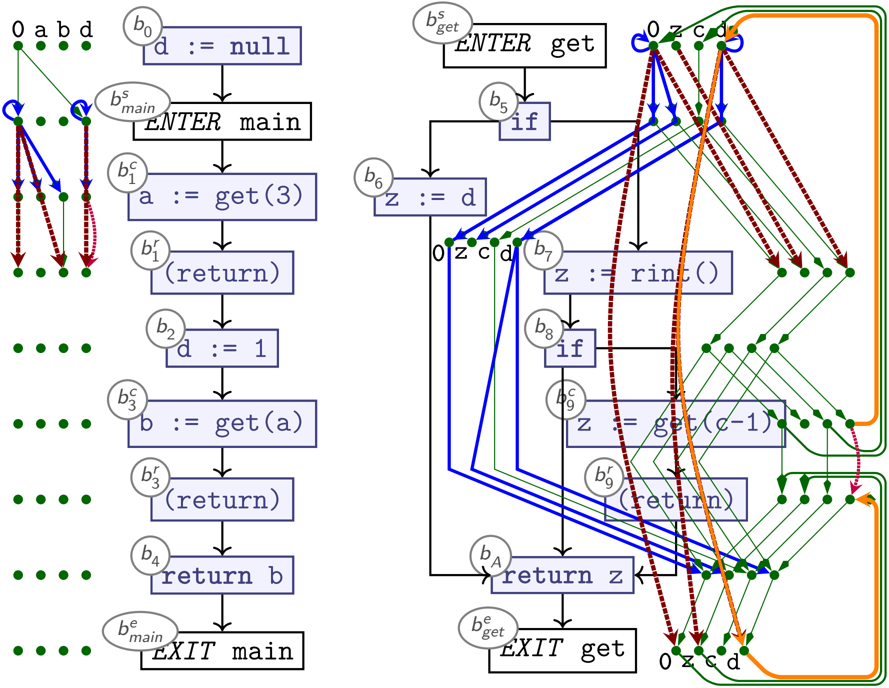


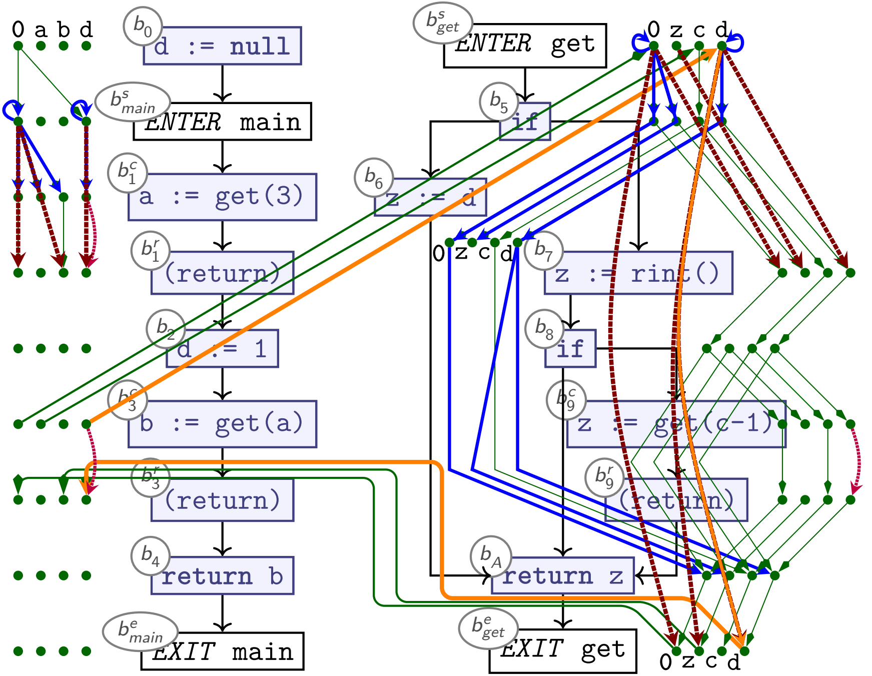
0 a b d
b_0 d := null
b^s_{main} ENTER main
b^c_1 a := get(3)
b^r_1 (return)
b_2 d := 1
b^c_3 b := get(a)
b^r_3 (return)
b_4 return b
b^e_{main} EXIT main
b^s_{get} ENTER get
0 z c d
b_5 if
b_6 z := d
0 z c d
b_7 z := rint()
b_8 if
b^c_9 z := get(c-1)
b^r_9 (return)
b_A return z
b^e_{get} EXIT get
0 z c d



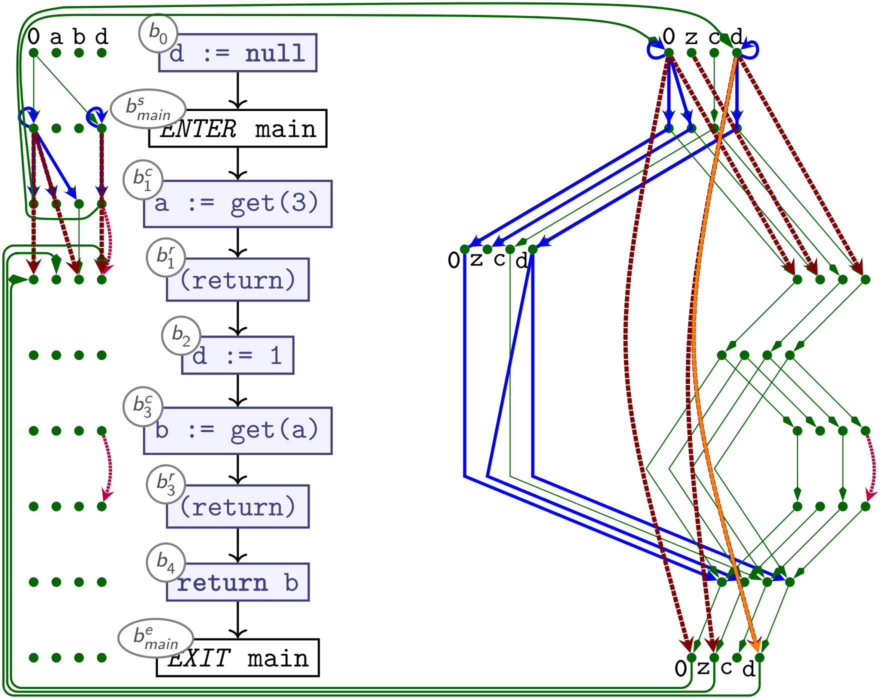
Step (exit edge)

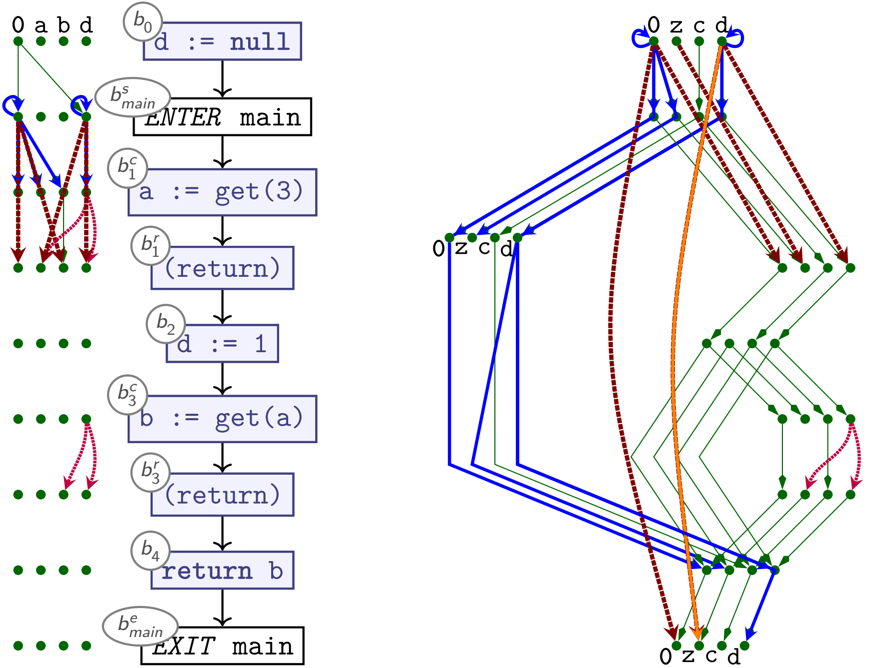
- Pick $e = n_1^s \rightarrow n_2^e$ off the work queue
- n_2^e is *exit* (e)?
  (n_1^s is always *start* node.)
- For each call/return pair n_i^c, n_i^r that calls the current function,
  if $n_i^c \rightarrow n_1^s \rightarrow n_2^e \rightarrow n_i^r$:
- If $n_i^c \rightarrow n_i^r \notin$ SummaryInst:
  - Add it to SummaryInst
  - Find all $n \rightarrow n_i^c \in$ PathEdge and propagate($n \rightarrow n_1^r$)


0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
0 z c d
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d

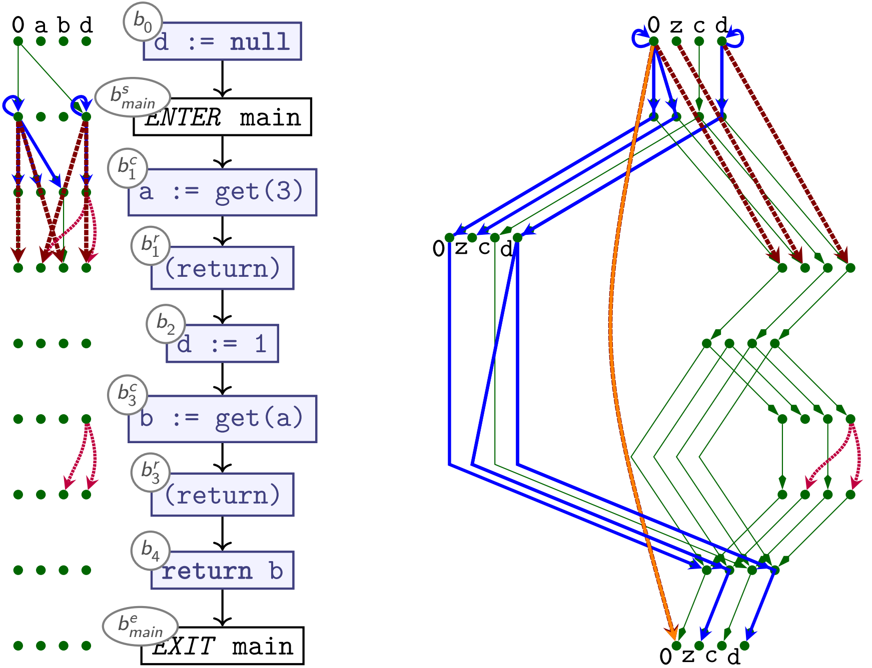


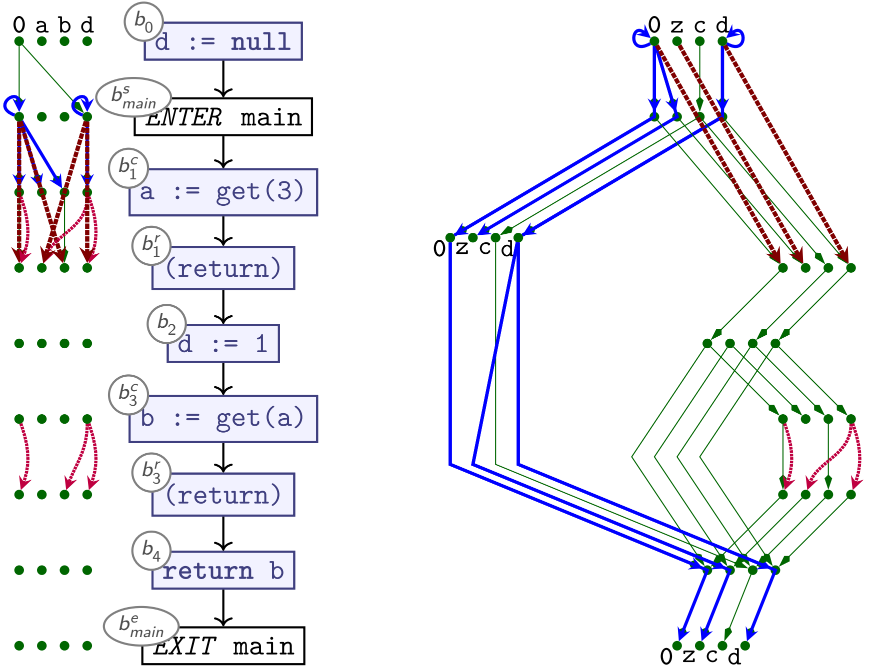
0 a b d
d := null
ENTER main
a := get(3)
(return)
d := 1
b := get(a)
(return)
return b
EXIT main
ENTER get
if
z := d
0 z c d
z := rint()
if
z := get(c-1)
(return)
return z
EXIT get
0 z c d
0 z c d

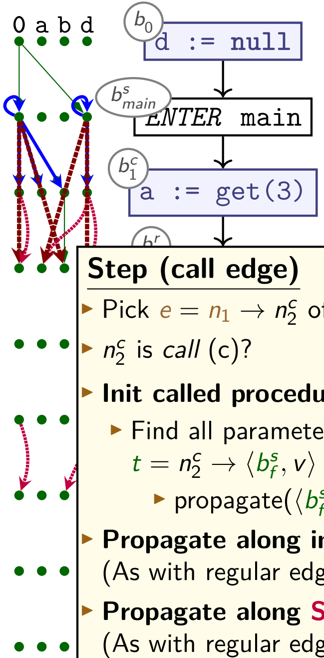

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

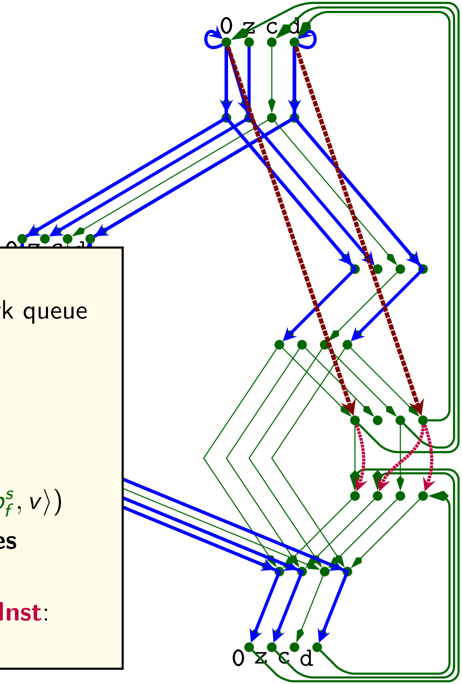


0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d


0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

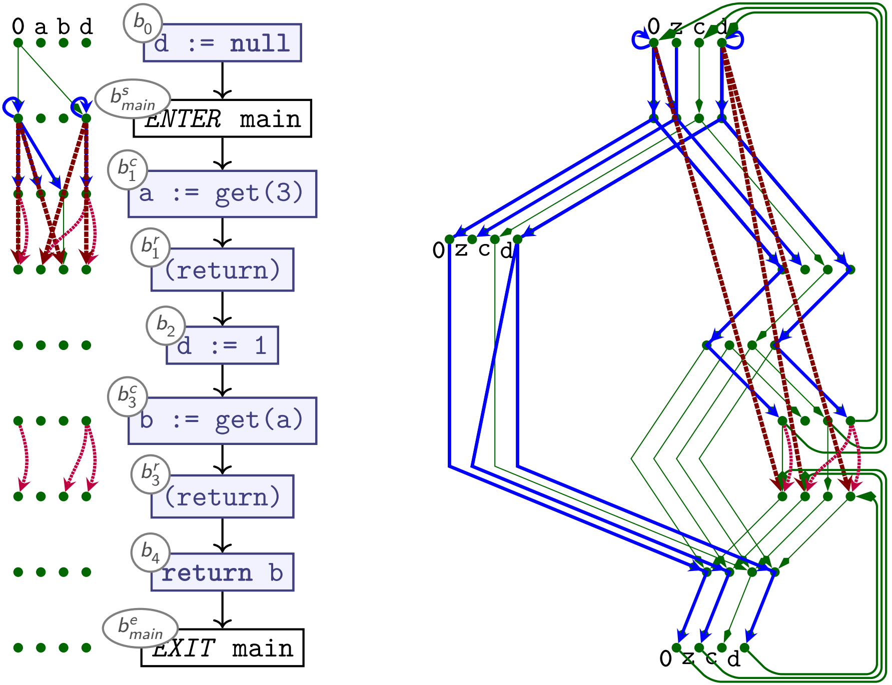

0 a b d
d := null
ENTER main
a := get(3)
(return)
d := 1
b := get(a)
(return)
return b
EXIT main
0 z c d
0 z c d
0 z c d

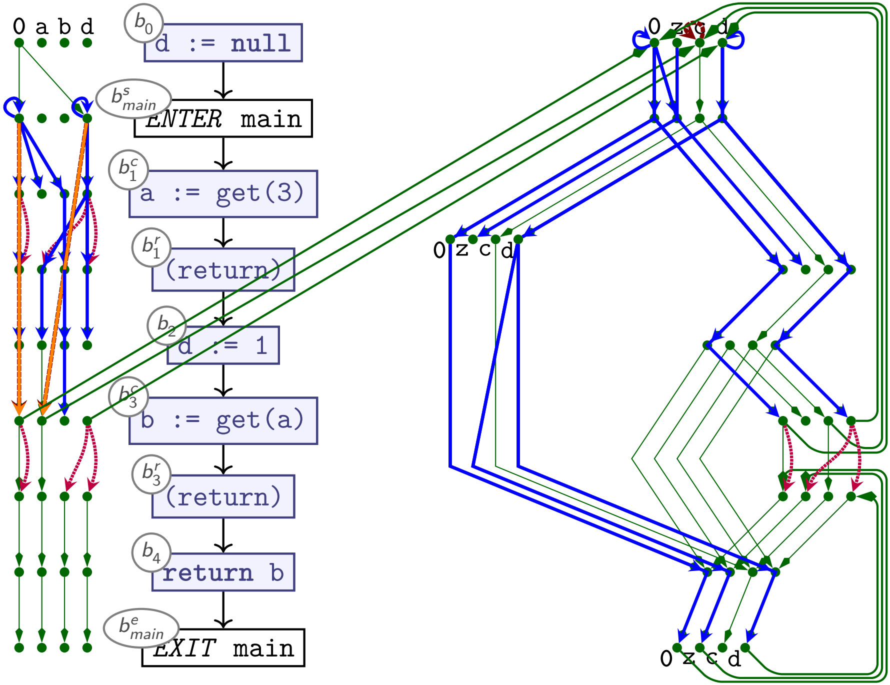


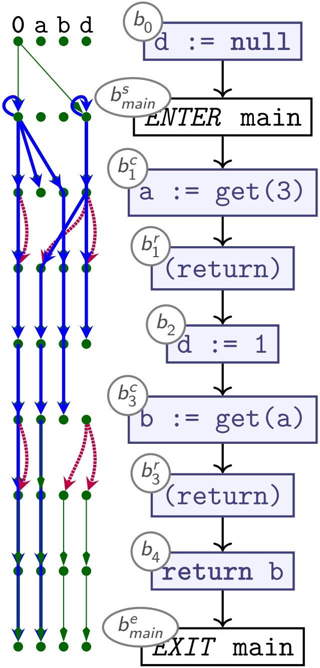


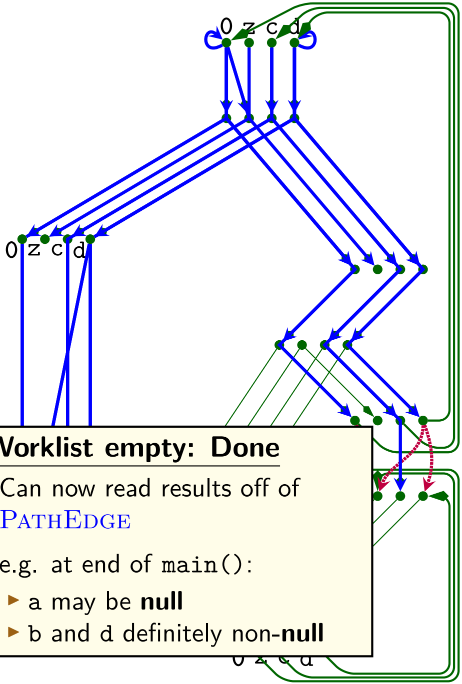
Step (call edge)

- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v\rangle \in E^\sharp$
    - propagate($\langle b_f^s, v\rangle \to \langle b_f^s, v\rangle$)
- **Propagate along intra-edges**
  (As with regular edges)
- **Propagate along SummaryInst**:
  (As with regular edges)


0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d


0 a b d
b_0 d := null
b^s_{main} ENTER main
b^c_1 a := get(3)
b^r_1 (return)
b_2 d := 1
b^c_3 b := get(a)
b^r_3 (return)
b_4 return b
b^e_{main} EXIT main
0 z c d
0 z c d
0 z c d





Worklist empty: Done

- Can now read results off of PathEdge
- e.g. at end of `main()`:
  - `a` may be **null**
  - `b` and `d` definitely non-**null**

The IFDS Algorithm: Initialisation and Propagation)

Procedure Init():
begin
  $\textsc{WorkList} := \textsc{PathEdge} := \emptyset$
  propagate($\langle b^s_{\mathsf{main}}, \mathbf{0}\rangle \rightarrow \langle b^s_{\mathsf{main}}, \mathbf{0}\rangle$)
  ForwardTabulate()
end

Procedure propagate($n_1 \rightarrow n_2$):
begin
  if $n_1 \rightarrow n_2 \in \textsc{PathEdge}$ **then**
    return
  $\textsc{PathEdge} := \textsc{PathEdge} \cup \{n_1 \rightarrow n_2\}$
  $\textsc{WorkList} := \textsc{WorkList} \cup \{n_1 \rightarrow n_2\}$
end

IFDS: Forward Tabulation

```
Procedure ForwardTabulate():
begin
  while n0 → n1 ∈ WorkList do
    WorkList := WorkList \ {n0 → n1}
    ⟨b0, v0⟩ = n0; ⟨b1, v1⟩ = n1
    if b1 is neither Call nor Exit node then
      foreach n1 → n2 ∈ E♯:
        propagate(n0 → n2)
    else if b1 is Call node then begin
      foreach call edge n1 → n2 ∈ E♯:
        propagate(n2 → n2)
      foreach non-call edge n1 → n2 ∈ E♯ ∪ SummaryInst:
        propagate(n0 → n2)
    end else if b1 is Exit node then begin
      foreach caller/return node pair b_i^c, b_i^r that calls b0 and vars v0, v1 do
        ns = ⟨b_i^c, v0⟩; nr = ⟨b_i^c, v1⟩
        if {ns → n0, n0 → n1, n1 → nr} ⊆ E♯ and not ns → nr ∈ SummaryInst then
          SummaryInst := SummaryInst ∪ {ns → nr}
          foreach nz → ns ∈ PathEdge:
            propagate(nz, nr)
end done end done end
```

Summary: IFDS Algorithm

- Computes yes-or-no analysis on all variables
  - Original notion of 'variables' is slightly broader)
- Represents facts-of-interest as nodes $\langle b, v \rangle$:
  - b is node (basic block) in CFG
  - v is variable that we are interested in
- Uses
  - *'Exploded Supergraph'* $G^\sharp$
    - All CFGs in program in one graph
    - Plus interprocedural call edges
  - *Representation relations*
  - *Graph reachability*
  - A *worklist*
- Distinguishes between *Call* nodes, *Exit* nodes, others
- **Demand-driven**: only analyses what it needs
- **Whole-program analysis**
- **Computes Least Fixpoint on distributive frameworks**

CodeProber study

Call for interviewees

Background

CodeProber is an active research project and we are curious of how you use CodeProber!

We would like to answer the following research questions by interviewing you:

- How is CodeProber used during the development of compilers and static analysis tools?
- What is the user perception of CodeProber?
- How does CodeProber compare to other tools during the development process (e.g debuggers, test cases, print-statements, AI, etc.)?

Interview

- We are looking for ~10 people
- 40-50 minutes long
- Swedish, English or Swenglish
- Mostly open questions, no “tests”, no need to prepare anything
- Interviews will be conducted in E building by me (Anton) and Niklas Fors.

Data and results management

- Interviews will be recorded for transcription purposes.
- Anonymized results will be discussed in the research team for this study (Anton, Niklas, Emma Söderberg).
- Anonymized results from interviews may be included in a publication.
- You can withdraw from the study up to 1 month after it takes place

Reward

- Drinks & snacks (“fika”) at the interview
- A small gift to bring home 🎁
- A feeling of contentment from having helped with research!
    - A quote from you during can become (anonymized &) published at a conference!

Interested?

Apply at https://book.ms/b/Intervju5@LundUniversityO365.onmicrosoft.com

(link & information will be mailed out after the lecture today)

Multiple time slots available next week (study week 7, 26/2→1/3)

- First come first served
- Please sign up as soon as possible, but at the latest Friday at 12
- Talk to me at the break if you want to register now!

Interprocedural Analysis in Java

Java

```
public static void main(String[] args) {
    Object obj = MyClass.getObj();
    System.err.println(obj.toString());
}
```

Subroutine call

- Analogous to Teal-0 calls
- …need to know `MyClass`

Method call

- **Dynamic Dispatch**
- Exact subroutine depends on *dynamic type* of `obj`

Challenges

- **Other modules**:
  - Must have access to analysable representation of module
  - *Not always available*
- **Dynamic Dispatch**:

```
obj.toString()
```

  - Which `toString` method are we calling?
  - Worst case assumption: *any* class (`Integer.toString()`, `HashSet.toString()`, ...)
  - Can we do better?

The Call Graph

Example in **C**
(No dynamic dispatch yet. . .)

```c
int main(int argc,
         char *argv) {
    if (argc > 1) {
        f(argv[0]);
    }
    g();
    return 0;
}
```

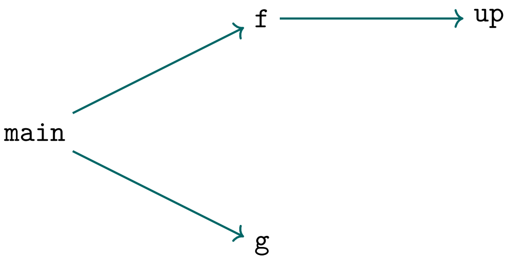
```c
void f(char *s) {
    for (char *p = s; *p; p++) {
        *p = up(*p);
    }
    puts(s);
}
```

```c
char up(char c) {
    if (c >= 'a' && c <= 'z') {
        return c - ('a' - 'A');
    }
    return c;
}
```

```c
void g(void) {
    puts("Hello, World!");
}
```

The Call Graph

- $G_{\mathsf{call}} = \langle P, E_{\mathsf{call}} \rangle$
- Connects procedures from P via call edges from E_{call}
- 'Which procedure can call which other procedure?'
- Often refined to:
  'Which *call site* can call which procedure?'
- Used by program analysis to find procedure call targets



Finding Calls and Targets

```java
class Main {
  public void
  main(String[] args) {
    A[] as = {new A(), new B()};
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```java
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

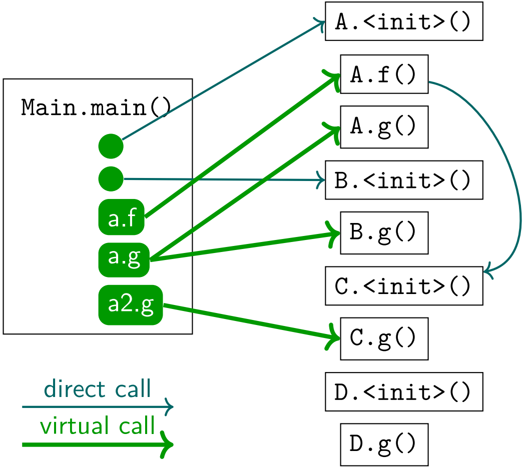
```java
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```java
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```java
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Dynamic Dispatch: Call Graph

Challenge: Computing the precise call graph:



Summary

- **Call Graphs** capture which procedure calls which other procedure
- For program analysis, further specialised to map:

$$\text{Callsite} \rightarrow \text{Procedure}$$

- Direct calls: straightforward
- Virtual calls (dynamic dispatch):
  - Multiple targets possible for call
  - No fully sound/precise solution in general

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

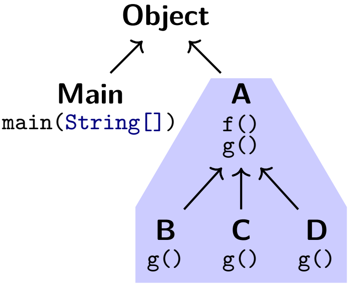
```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

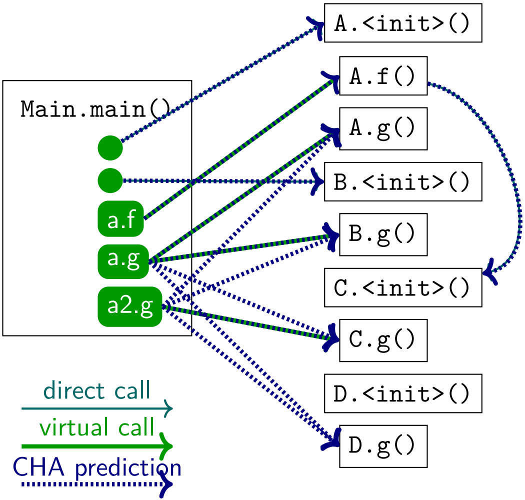
```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Class Hierarchy Analysis



- Use declared type to determine possible targets
- Must consider all possible subtypes
- In our example: assume a.f can call any of:
  `A.f()`, `B.f()`, `C.f()`, `D.f()`

Class Hierarchy Analysis: Example



Summary

- **Call Hierarchy Analysis** resolves virtual calls $a.f()$ by:
  - Examining static types T of receivers $(a : T)$
  - Finding all subtypes $S <: T$
  - Creating call edges to all $S.f$, if $S.f$ exists
- **Sound**
  - Assuming strongly and statically typed language with subtyping
- Not very **precise**
  - Java: `((Object) obj).toString()`:
    Will use *all* `toString()` methods *anywhere*

Rapid Type Analysis

- Intuition:
  - Only consider reachable code
  - Ignore unused classes
  - Ignore classes instantiated only by unused code

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = {new A(), new B()};
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

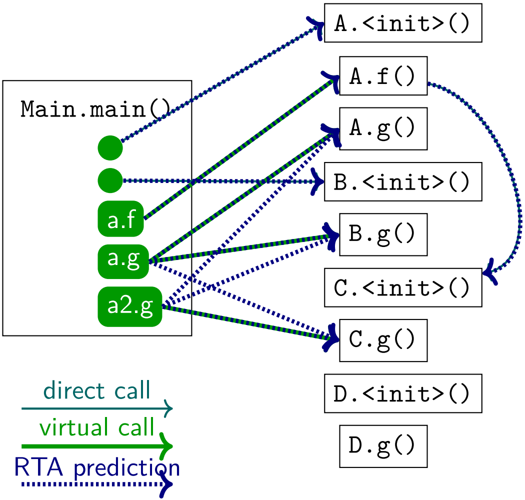
```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Rapid Type Analysis: Example



Rapid Type Analysis Algorithm Sketch

```
Procedure RTA(mainproc, <:):
begin
  WORKLIST := {mainproc}
  VIRTUALCALLS := ∅
  LIVECLASSES := ∅
  while s ∈ mainproc do
    foreach call c ∈ s do
      if c is direct call to p then
        addToWorklist(p)
        registerCallEdge(c → p)
      else if c = v.m() and v : T then begin
        VIRTUALCALLS := VIRTUALCALLS ∪ {c}
        foreach S <: T do
          addToWorklist(S.m)
          registerCallEdge(c → S.m)
        done
      end else if c = new C() and C ∉ LIVECLASSES then begin
        LIVECLASSES := LIVECLASSES ∪ {C}
        foreach v.m() ∈ VIRTUALCALLS with v : T and C <: T do
          addToWorklist(C.m)
          registerCallEdge(c → C.m)
        done
      end
done done end
```

Summary

- **Rapid Type Analysis** resolves virtual calls $a.f()$ as follows:
  - Find all classes that can be instantiated in reachable code
  - Expand reachable code:
    - For direct calls to p, add p as reachable
    - For all virtual calls to $v.m()$ with $v : T$:
      $\Rightarrow$ Add $S.m()$ as reachable
  - Iterate until we reach a fixpoint
- **Sound**
  - Assuming strongly and statically typed language with subtyping
- More **precise** than Class Hierarchy Analysis

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g()
}
```

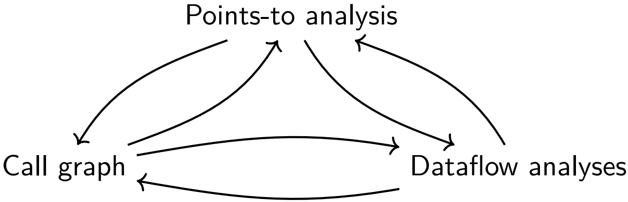
```
class C extends A {
  public String
```

```
class B extends A {
  @Override
  public String
  eturn "B"; }
```

Use **points-to analysis**?

But what call graph should the points-to analysis use?

Dependencies



- Mutual dependencies across program analyses

Loose Composition

Loose Composition: **Split analyses into multiple passes**

- Each pass finishes before next pass starts
- Example:
  1. **RTA**: compute initial call graph
  2. **Steensgaard** on RTA output: conservative points-to graph
  3. Build **pointer-based call graph** from Steensgaard's results
  4. **Andersen's analysis** with refined (smaller) call graph

Tight Composition

Tight Composition: **Analyses depend on each other's intermediate results**

- Analyses run "together"
- Example:
  - JastAdd circular attribute computations (Exercise 2)
  - Could combine data flow analysis with points-to or call-graph analysis
- **Challenges**:
  - Traditional worklist algorithms:
    - Complex manual engineering needed
  - Declarative approaches:
    - Must guarantee **Monotonicity**

Summary

- Mutual dependency between *points-to*, *data flow*, *call graph* analyses
- Two approaches:
  - **Loose composition**:
    - One analysis after the other
    - May need to run analyses multiple times
  - **Tight composition**:
    - Analyses can use each other's intermediate results
    - Difficult to engineer for worklist algorithms
    - Easier with declarative approaches (attribute grammars, logic programming)

Summary: Flow-Insensitive Analysis

- **Monomorphic type inference**
  - Free variables, occurs check, unification
  - Close to $O(\#\text{AST nodes})$
- Polymorphic type inference (Hindley-Damas-Milner)
  - Type schemas and instantiation
  - DEXPTIME-complete
- **Steensgaard's points-to analysis**
  - Similar to monomorphic type inference
  - Close to $O(\#\text{AST nodes})$
- **Andersen's points-to analysis**
  - Points-to edges and inclusion edges that generate new edges
  - $O(\#\text{nodes}^3)$

Summary: Data Flow Analyses

- **MFP**
  - Precise for distributive frameworks
  - $O(\#edges \times height(\mathcal{L}))$
- **MOP**
  - Precise for monotone frameworks
  - Undecidable
- **IFDS** / IDE
  - Interprocedural, precise for distributive frameworks
  - $O(\#\text{edges} \times \#\text{variables}^3)$
    (IDE: $O(\#\text{edges} \times \#\text{variables}^3 \times height(\mathcal{L}))$)

Summary: Call Graph Analyses

- **Class Hierarchy analysis**
  - Trivial
  - $O(\#\text{classes} \times \#\text{methods})$
- **Rapid Type Analysis**
  - Transitive reachability check
  - $O(\#\text{classes} \times \#\text{methods})$
- **Points-to-based call graph analysis**
  - Mutual dependency
  - Complexity and precision vary

Building Analyses: Considerations

- What level of soundness?
  - Conservative: sound, but can be imprecise
  - Optimistic: unsound, but can be more precise
- What performance needs?
  - Trade-off: soundness vs. precision vs. performance
  - More precise server analysis $\Longrightarrow$ faster client analysis
  - Some analyses can be split into:
    - fast/coarse "filter" pass
    - slow/precise main pass
  - Interactive use? Low latency, consider incremental analyses
  - High reliability need? (Integrate interactive tools?)

  ...
- What do we know?
  - Language semantics
  - External libraries of importance
  - User annotations / specs to help analysis

  ...