EDAP15: Program Analysis

INTERPROCEDURAL ANALYSIS

Christoph Reichenbach

Welcome back!

- Lab 2: Bugfix pushed
- Lab 3: Simplified (but still two parts)
- Office hours:
  - **Wednesdays** 14:30–15:30
  - **Thursdays** 13:15–14:00

Questions?

Lecture Overview

Foundations

- 01 Foundations
- 02 Constructing Program Analyses in JastAdd

Static Analysis

Properties

- 03 04 Types
- 05 06 07 Data Flow
- 08 09 Memory

Control Flow

- 05 Intraprocedural
- 10 **Interprocedural**
- 11 Indirect

Dynamic Analysis

- 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

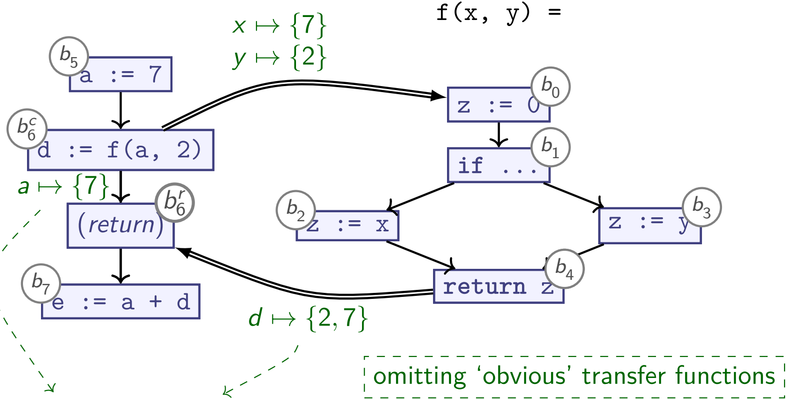
```
a := 7;
d := f(a, 2);
e := a + d;
```

Teal-0

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

How can we compute Constant Propagation here?

A Naïve Inter-Procedural Analysis

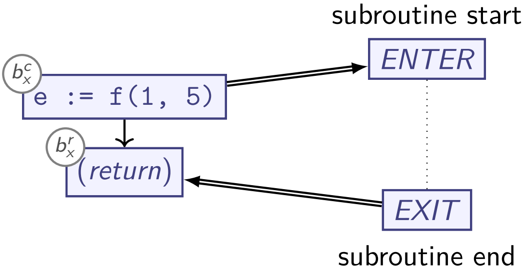


$$\mathbf{in}_{b_6^r} = \mathbf{out}_{b_6} \sqcup \mathbf{out}_{b_4} = \mathbf{out}_{b_6^r}$$

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

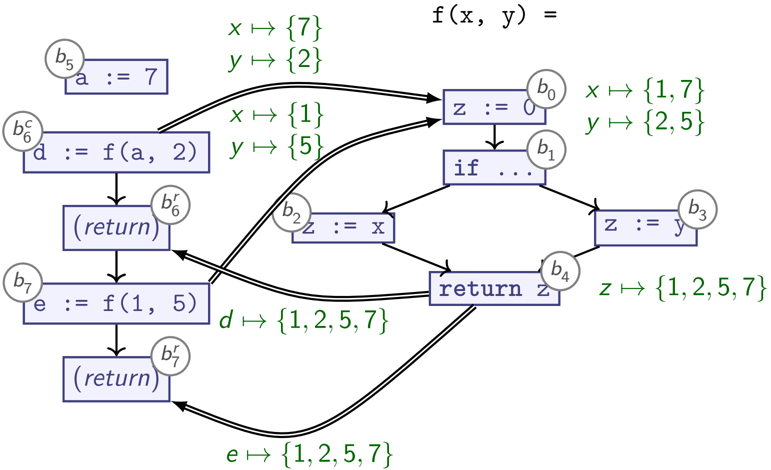
Works rather straightforwardly!

Inter-Procedural Control Flow Graph



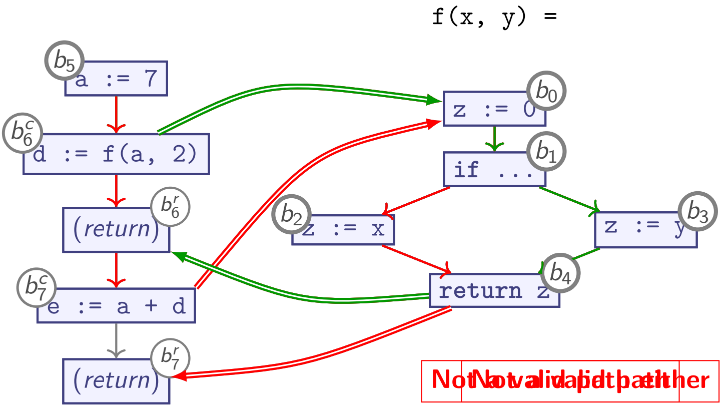
- Split call sites b_x into *call* (b_x^c) and *return* (b_x^r) nodes
- Intra-procedural edge $b_x^c \longrightarrow b_x^r$ carries environment/store
- Inter-procedural edge ($\Longrightarrow$):
  - Call site $\Longrightarrow$ callee: substitutes parameters
  - Call site $\Longleftarrow$ return: substitutes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

A Naïve Inter-Procedural Analysis



Imprecision!

Valid Paths



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

Context-sensitive interprocedural analyses consider only valid paths

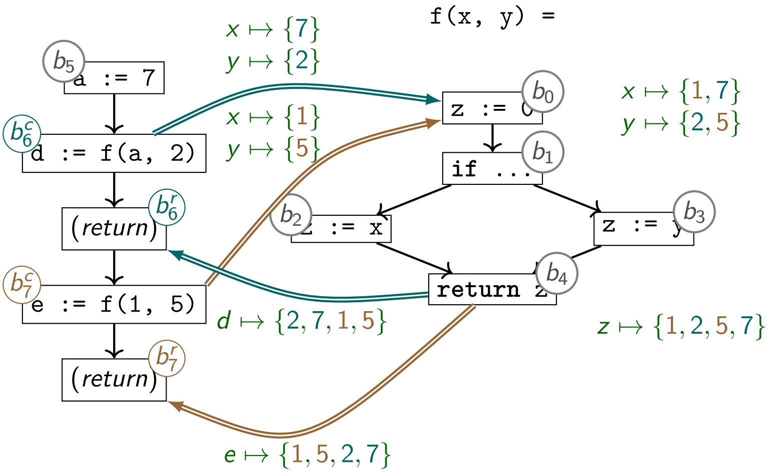
Summary

- **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



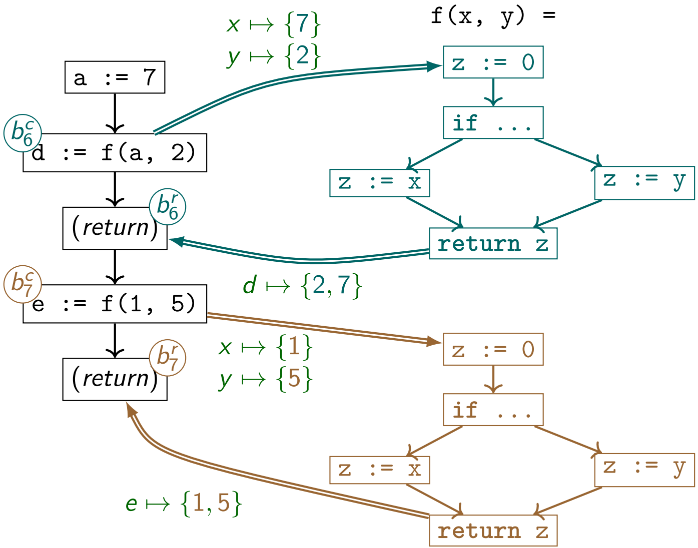
***Call-site insensitive*: analysis merges all callers to `f()`**

Precise Interprocedural Dataflow

- Precision via one of:

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

Inlining

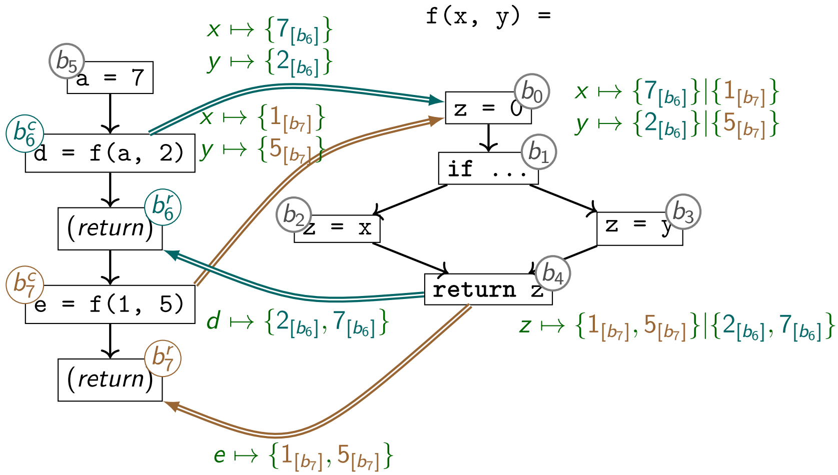


Clone subroutine IRs for each *calling context*

Precise Interprocedural Dataflow

- Precision via one of:
  1. Inlining or AST cloning
  2. **Call Strings**
  3. Procedure Summaries

Call Strings of Length 1



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) // b6
       + g(5); // b7
}
...
  f(1); // b0
  f(2); // b1
```

Must bound length of call strings to ensure termination

Summary

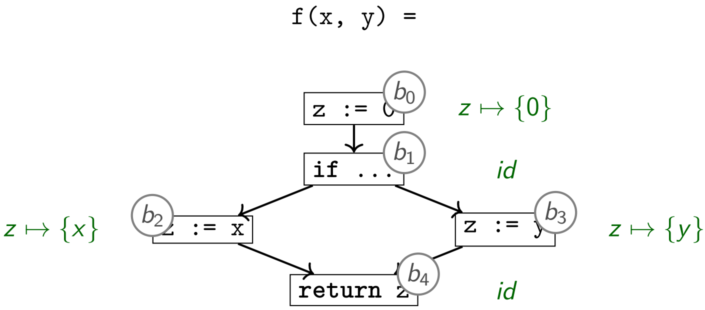
Strategies for call-site sensitive analysis:

- **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



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

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
- Loops/recursion require fixpoint computation over function composition!

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
  - Loops / recursion are 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

Procedure Summaries with Gen/Kill-sets

- Special case: Gen/Kill sets
- Transfer functions always consist of
  - Gen-set
  - Kill-set
- *No symbolic reasoning needed*:
  - Compose gen-sets, kill-sets, receive combined gen/kill-set for subroutine
- Maybe there are other such cases?

Greta Yorsh, Eran Yahav, Satish Chandra: "Generating Precise and Concise Procedure Summaries", in Principles of Programming Languages 2008

Procedure Summaries for Dataflow

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

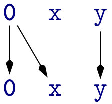
. . . *but only for Distributive Frameworks*

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

Representation Relations

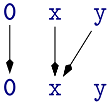
Example procedure summary representation:

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

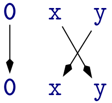


'May be null' analysis

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




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



- $c \longrightarrow d$:
  if $P(c) \in \mathbf{in}_b$ then $P(d) \in \mathbf{out}_b$
- Representation Relations relate $\mathbf{in}_b$ and $\mathbf{out}_b$ variables $\mathcal{V}$
- $R \subseteq (\mathcal{V} \cup \{\mathbf{0}\}) \times (\mathcal{V} \cup \{\mathbf{0}\})$
- if $\langle \mathbf{0}, X \rangle \in R$:
  X always 'may be null' in $\mathbf{out}_b$
- if $\langle Y, X \rangle \in R$:
  If Y 'may be null' in $\mathbf{in}_b$:
  $\Rightarrow X$ 'may be null' in $\mathbf{out}_b$

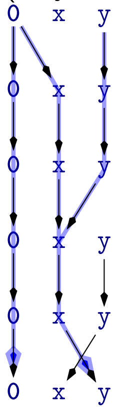
Composing Representation Relations

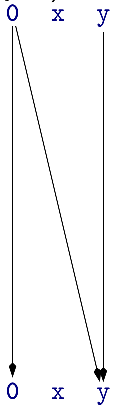
Representation Relations (*may be null* analysis):

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

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

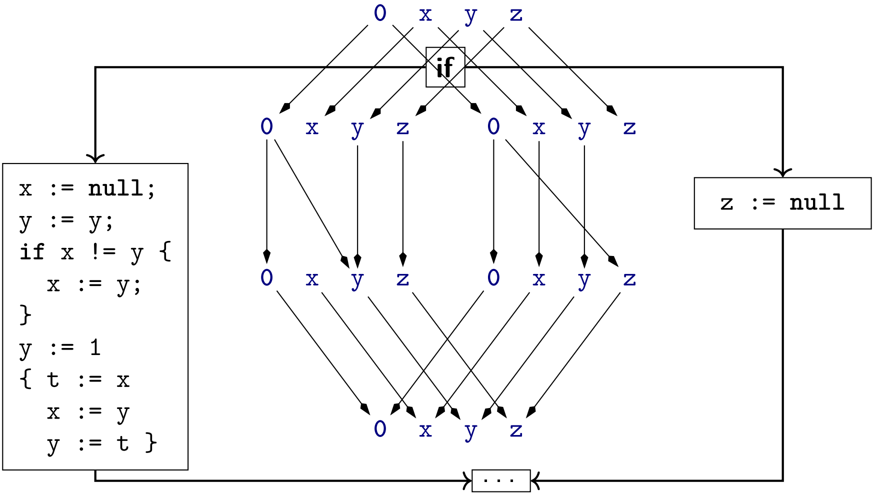
```
{ t := x;
  x := y;
  y := t; }
```



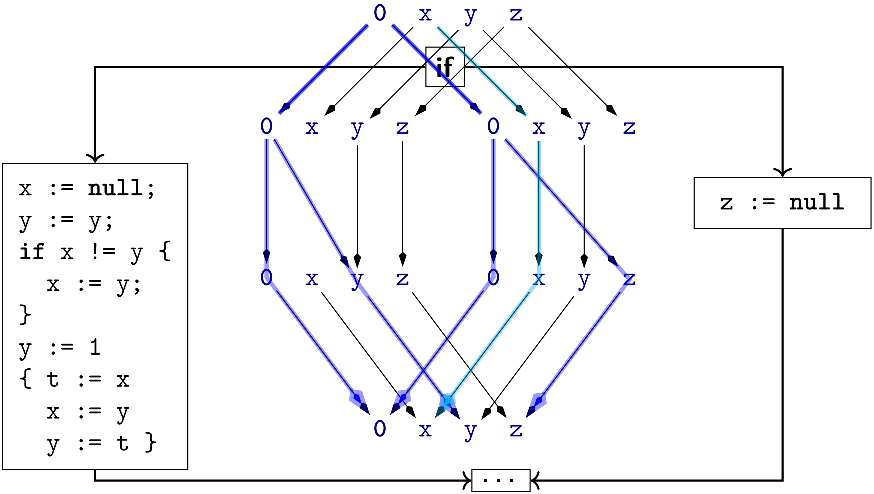


Composed representation relations are again representation relations

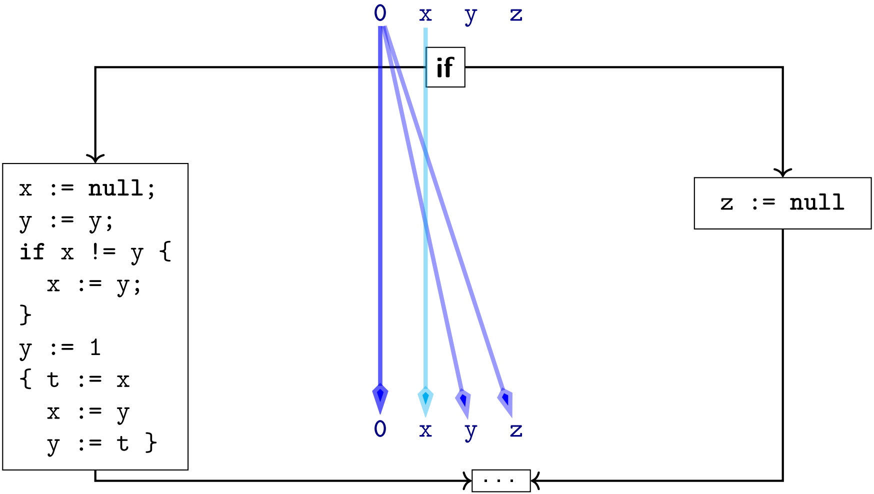
Joining Control-Flow Paths



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

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

PathEdge edge

All WorkList edges are also PathEdge edges

Result of our analysis

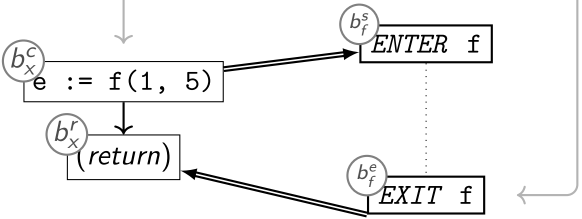
$N^\sharp$-edge

SummaryInst

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

IFDS Strategy

- Algorithm distinguishes between three types of nodes:
  - *Exit* nodes (b^e_f)
  - *Call* nodes (b^c_x)
  - 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

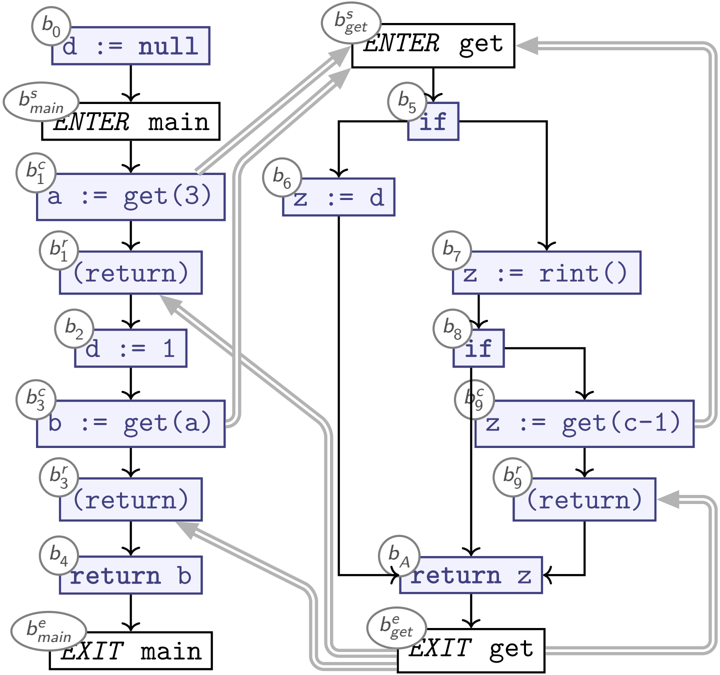
Running Example

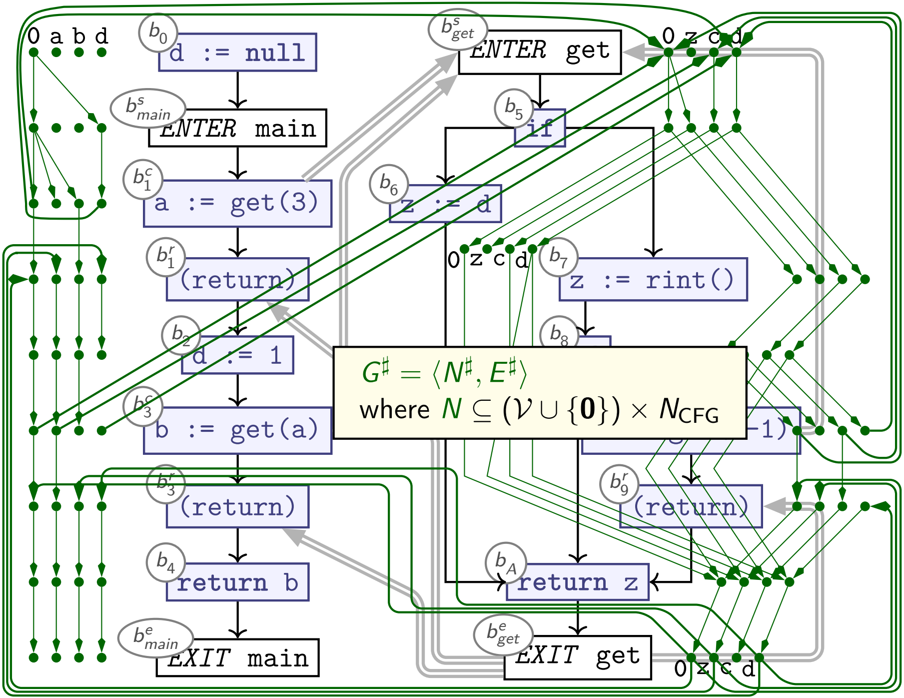
Teal-0: *main()*

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

Teal-0: *get()*

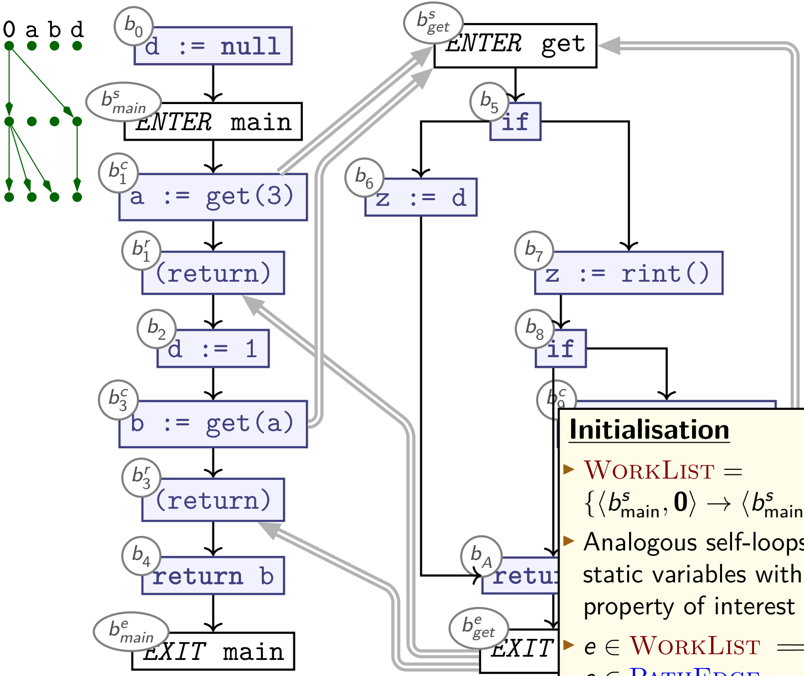
```
fun get(c) = {
  if c == 0 {
    z := default;
  } else {
    z := read_int();
    if z < 0 {
      z := get(c - 1);
    }
  }
  return z;
}
```


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



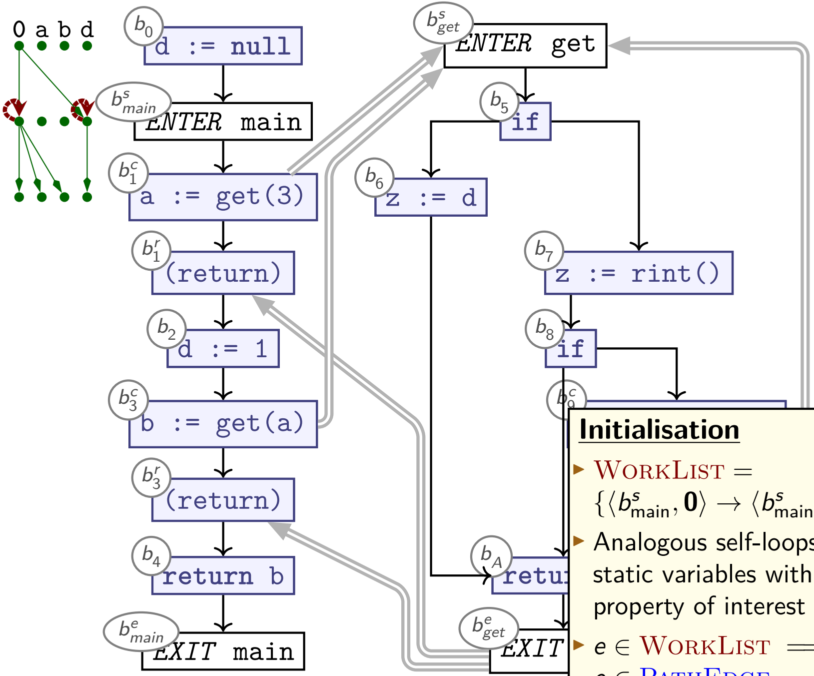
$G^\sharp = \langle N^\sharp, E^\sharp \rangle$

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



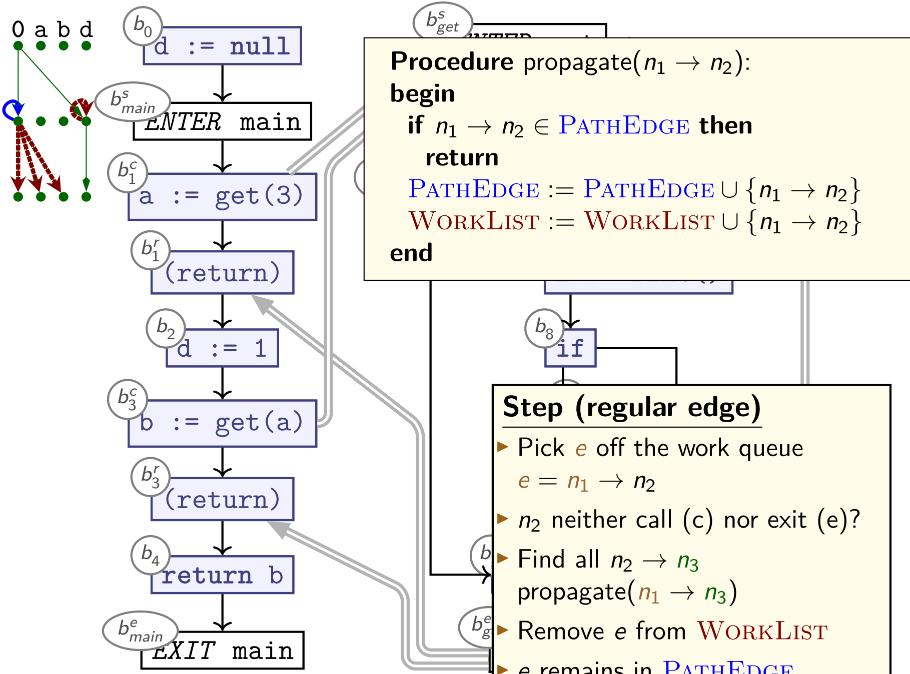
Initialisation

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



Initialisation

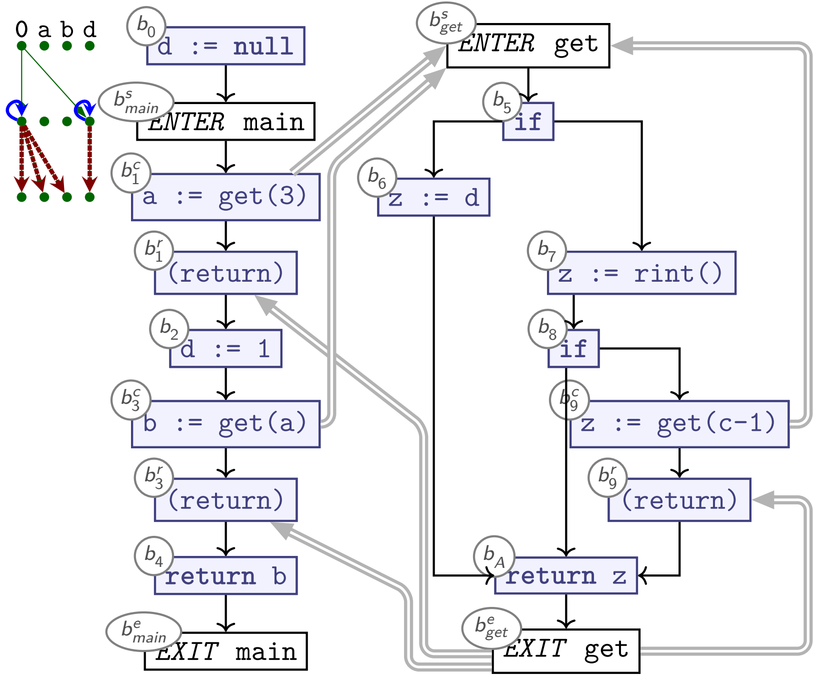
- WorkList = $\{\langle b^s_{\text{main}}, \mathbf{0}\rangle \rightarrow \langle b^s_{\text{main}}, \mathbf{0}\rangle\}$
- Analogous self-loops for static variables with property of interest (d)
- $e \in$ WorkList $\Longrightarrow$ $e \in$ PathEdge

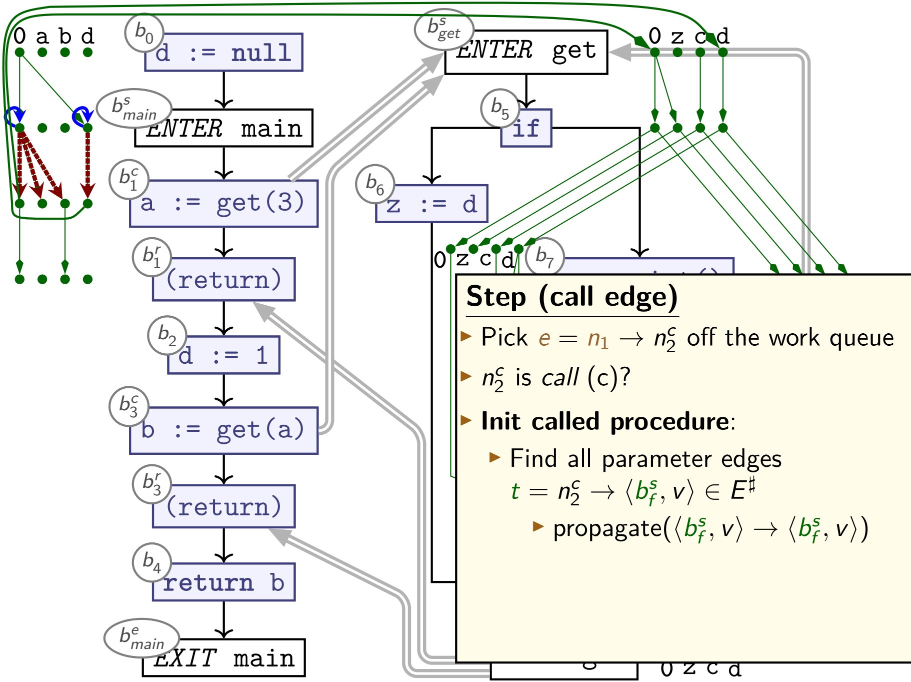


Procedure propagate($n_1 \to n_2$):
begin
  if $n_1 \to n_2 \in$ PathEdge **then**
    return
  PathEdge := PathEdge $\cup \{n_1 \to n_2\}$
  WorkList := WorkList $\cup \{n_1 \to n_2\}$
end

Step (regular edge)

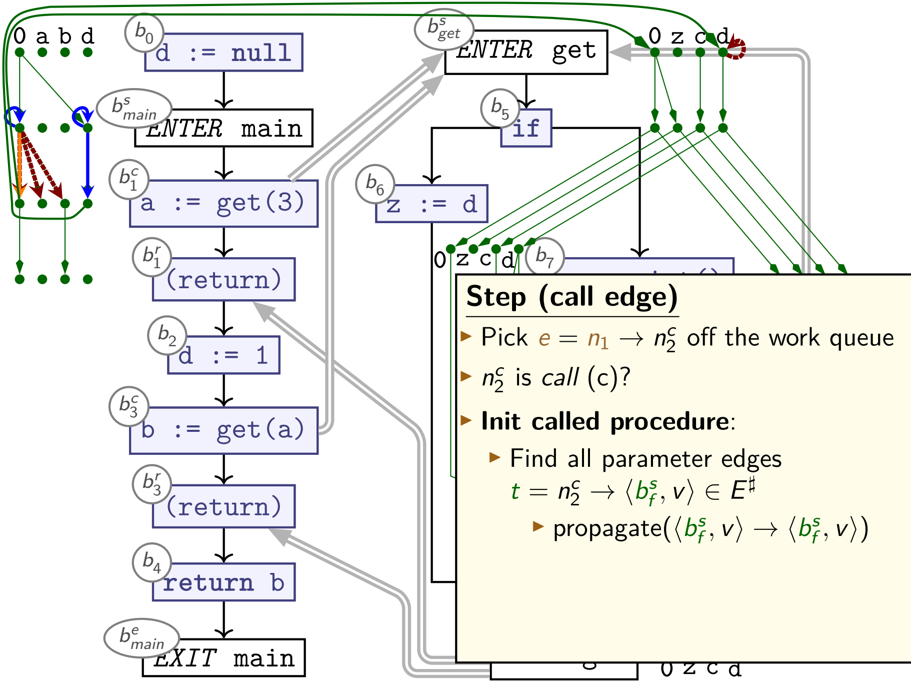
- Pick e off the work queue
  $e = n_1 \to n_2$
- n_2 neither call (c) nor exit (e)?
- Find all $n_2 \to n_3$
  propagate($n_1 \to n_3$)
- Remove e from WorkList
- e remains in PathEdge


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



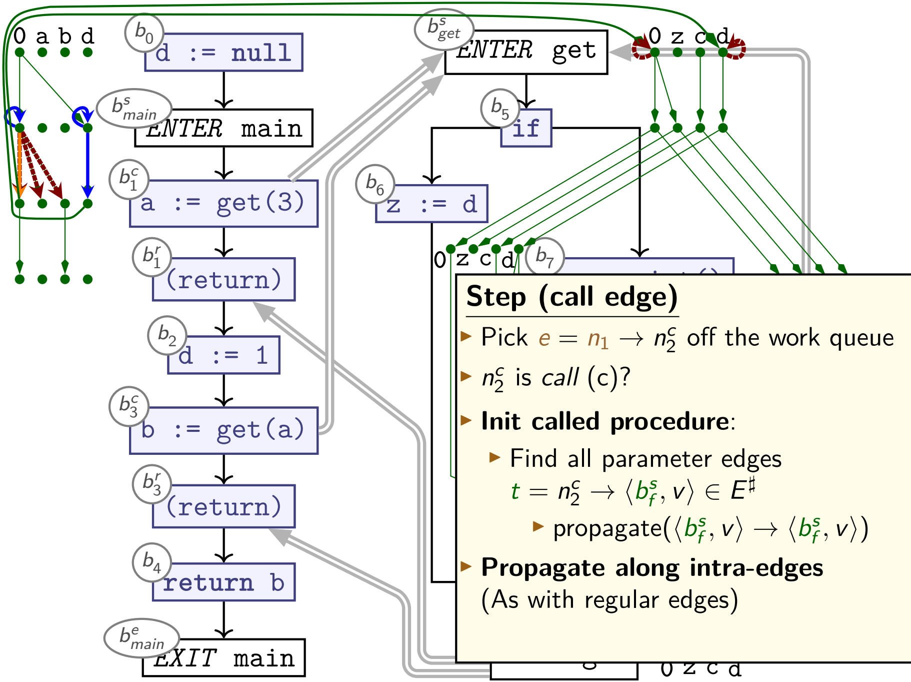
Step (call edge)

- 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$(\langle b_f^s, v \rangle \to \langle b_f^s, v \rangle)$



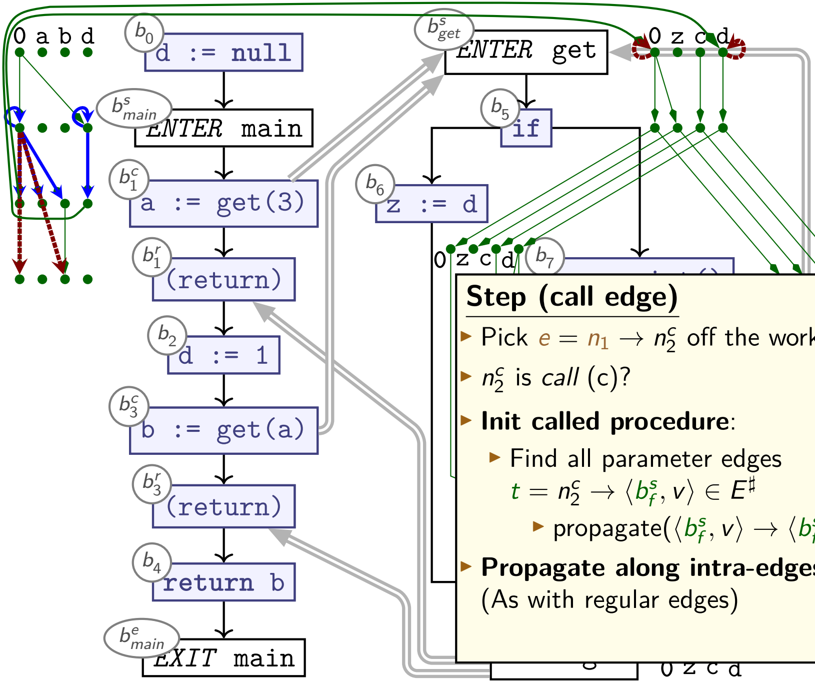
Step (call edge)

- 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($\langle b_f^s, v\rangle \to \langle b_f^s, v\rangle$)



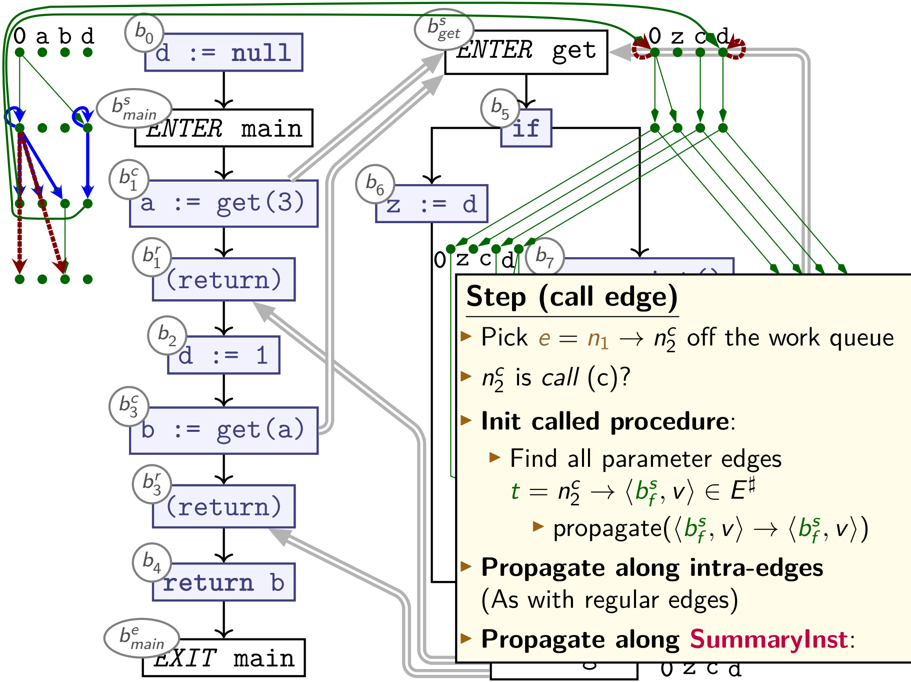
Step (call edge)

- 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)



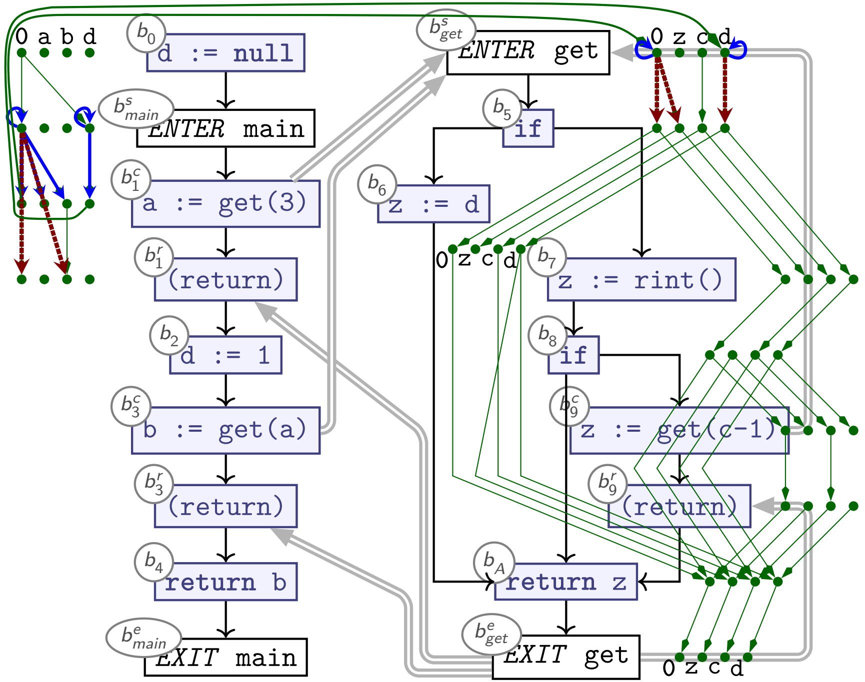
Step (call edge)

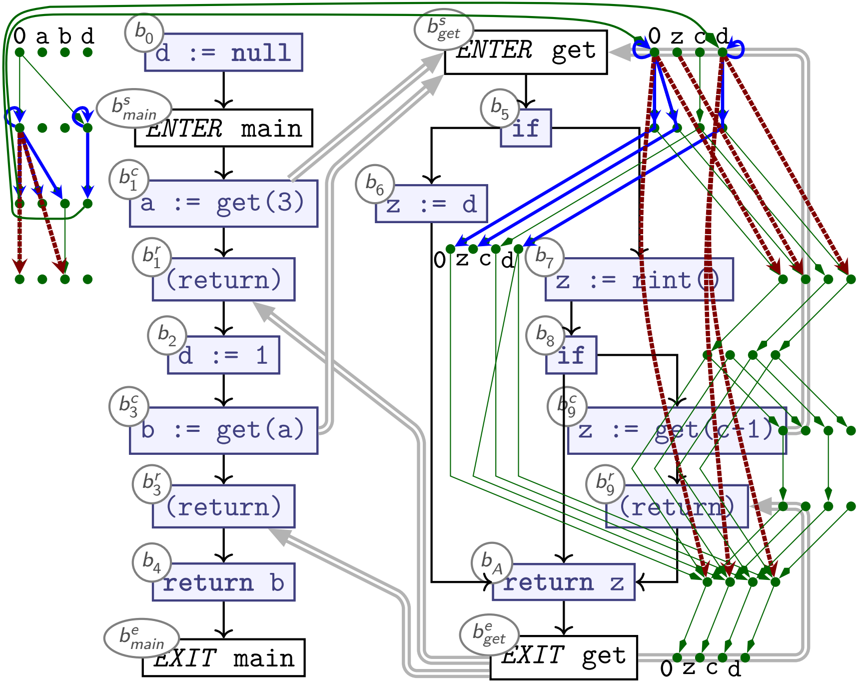
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \to \langle b_f^s, v \rangle$)
- **Propagate along intra-edges** (As with regular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



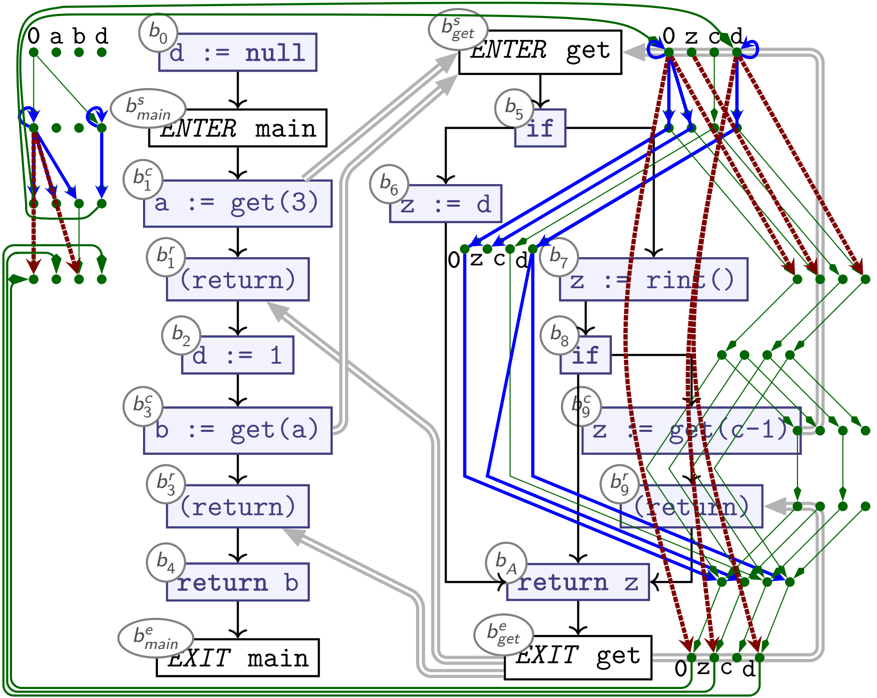
Step (call edge)

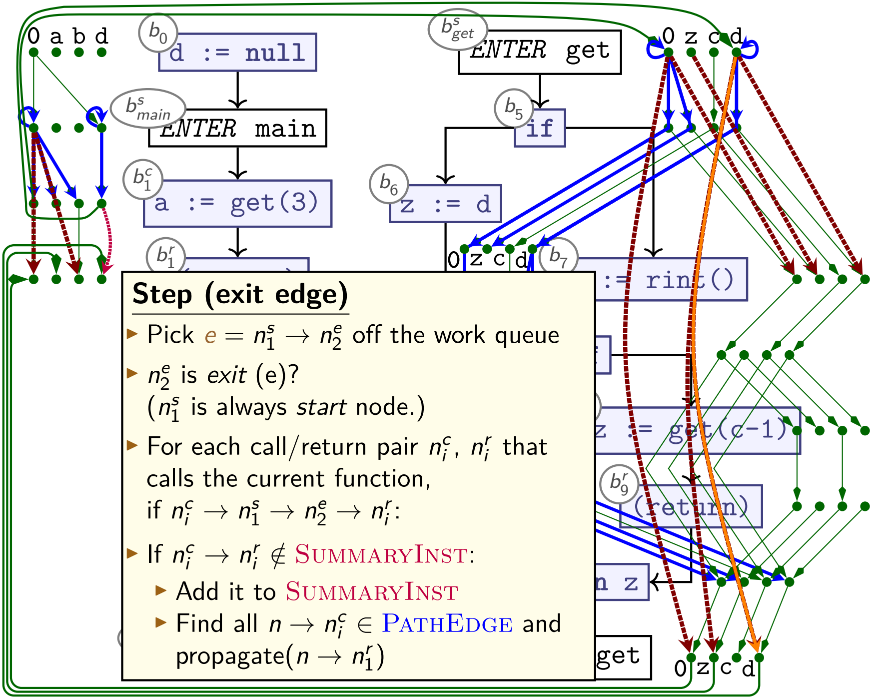
- Pick $e = n_1 \rightarrow n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \rightarrow \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \rightarrow \langle b_f^s, v \rangle$)
- **Propagate along intra-edges**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



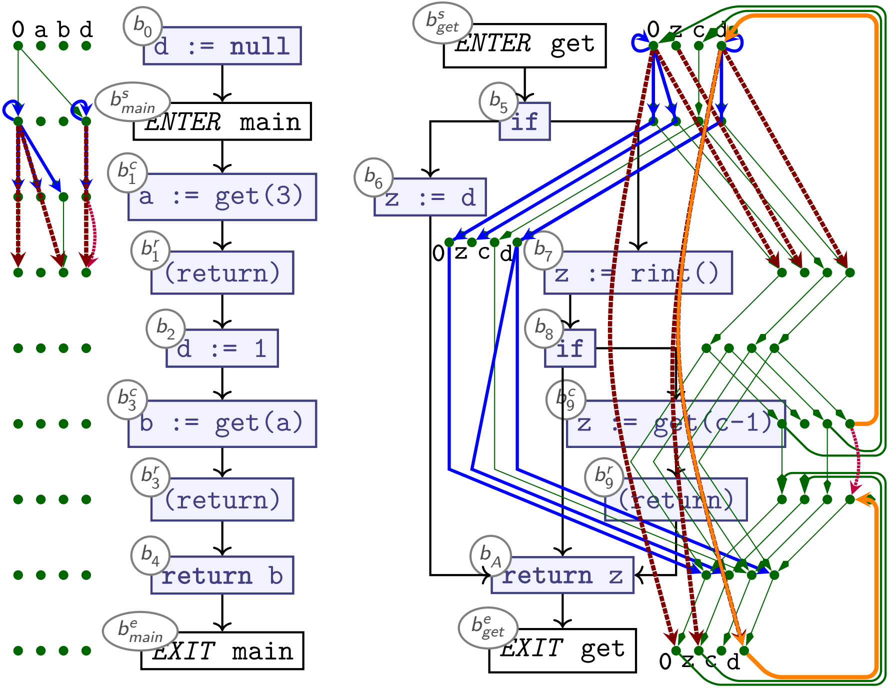
0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
0 z c d
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0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

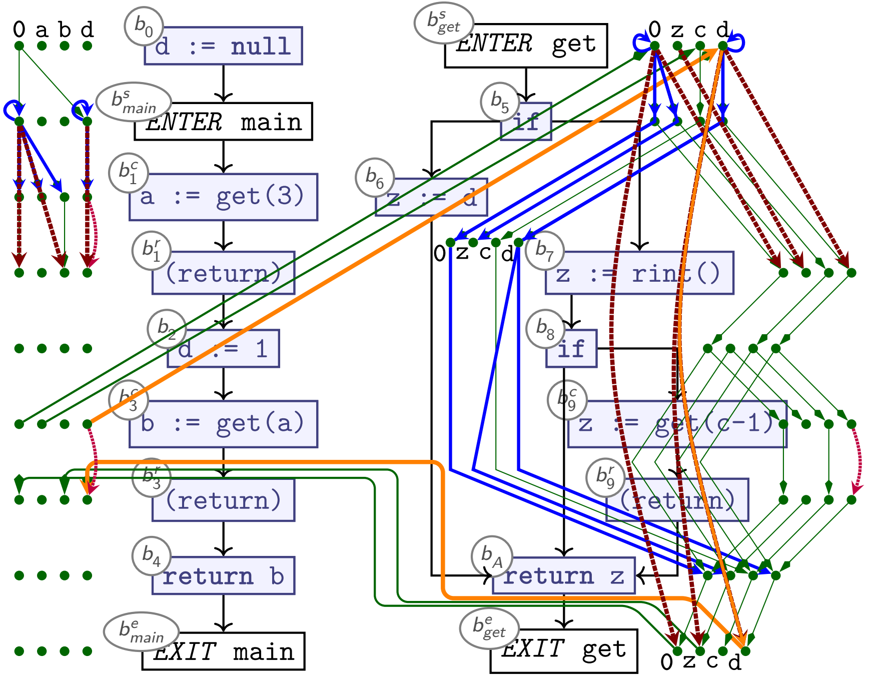


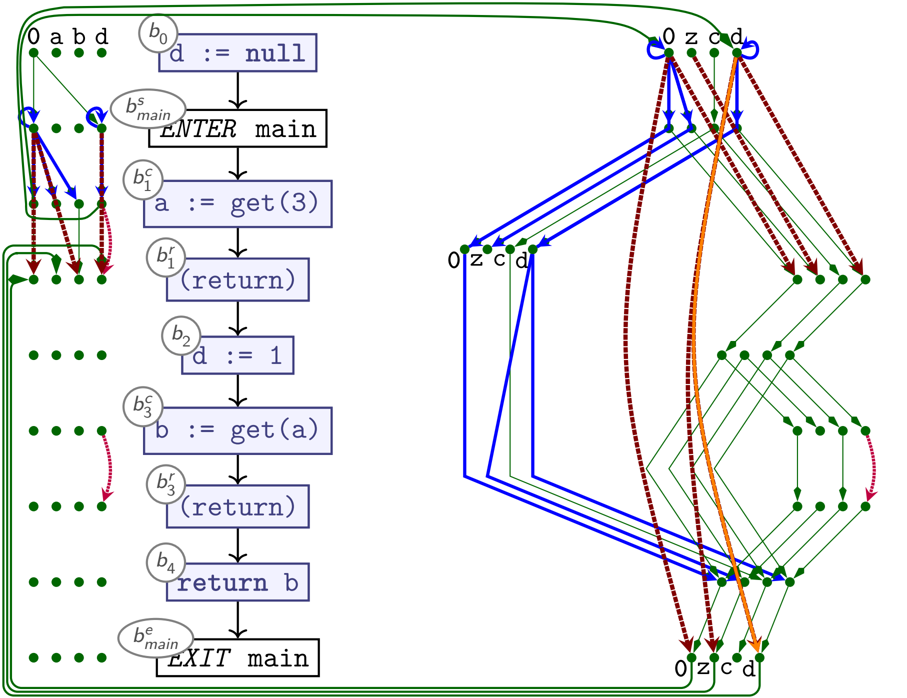
Step (exit edge)

- Pick $e = n_1^s \to n_2^e$ 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 $n_i^c \to n_1^s \to n_2^e \to n_i^r$:
- If $n_i^c \to n_i^r \notin$ SUMMARYINST:
  - Add it to SUMMARYINST
  - Find all $n \to n_i^c \in$ PATHEDGE and propagate($n \to n_1^r$)

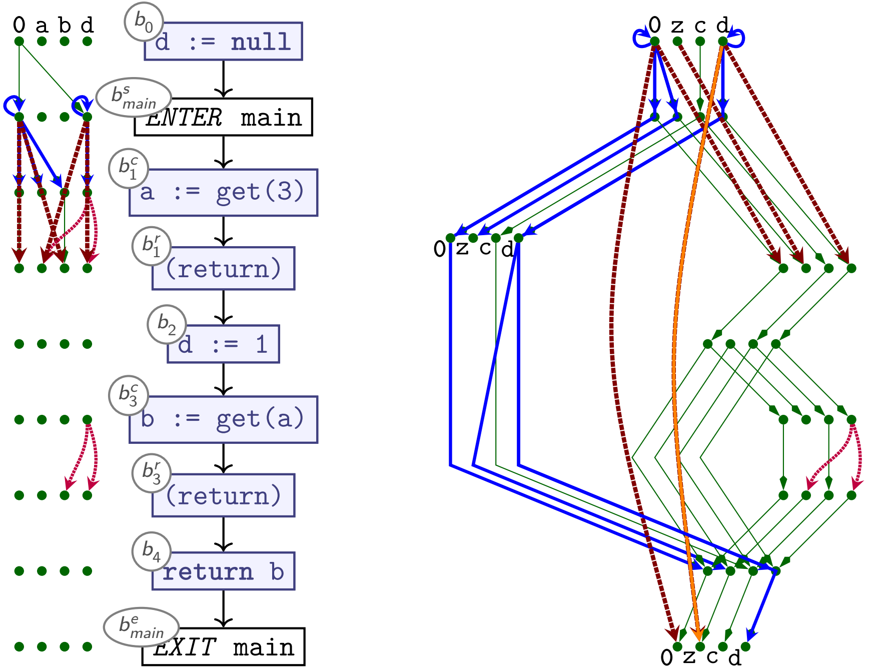


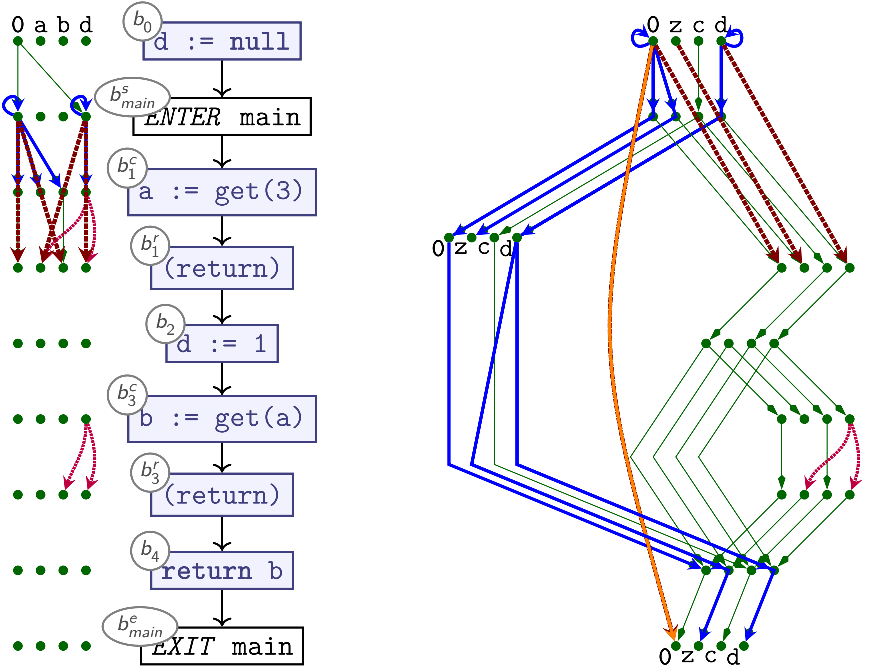
0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
0 z c d
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0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

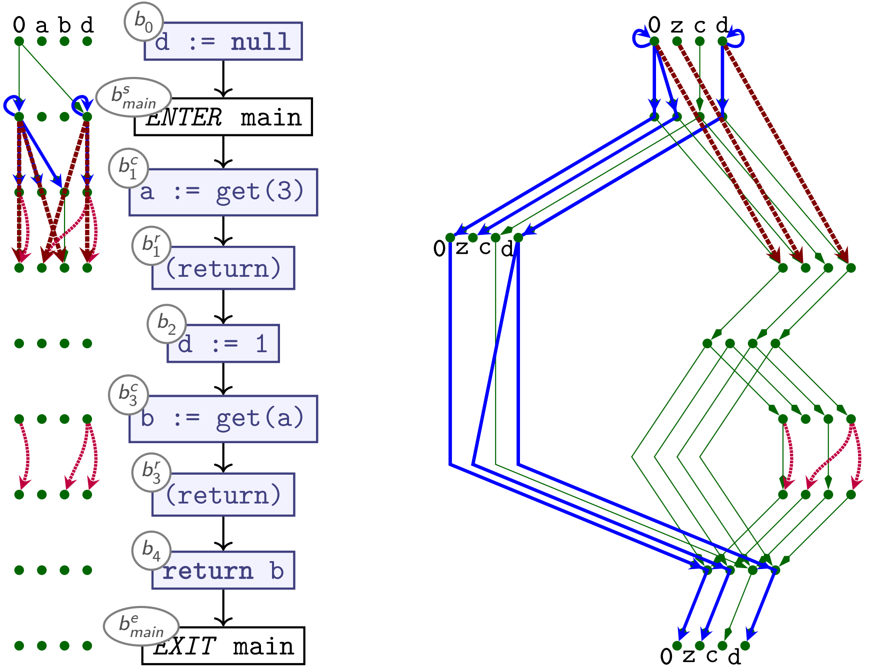

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
0 z c d
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0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



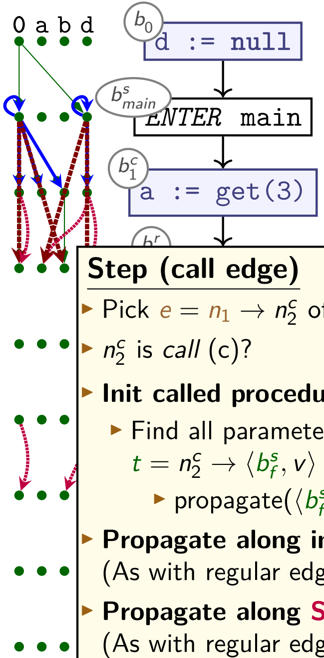
0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

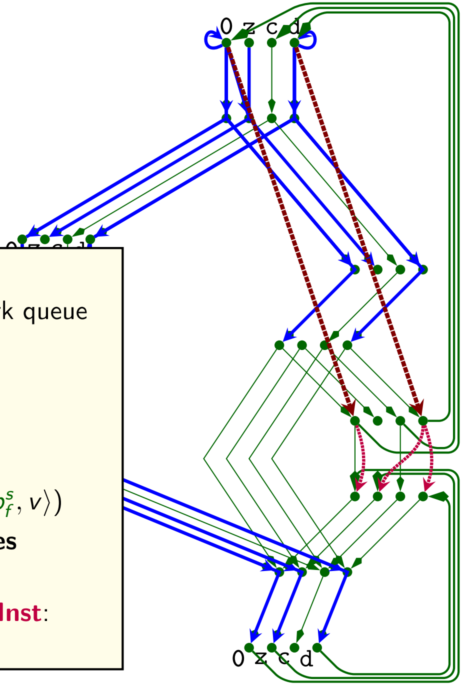

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d


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



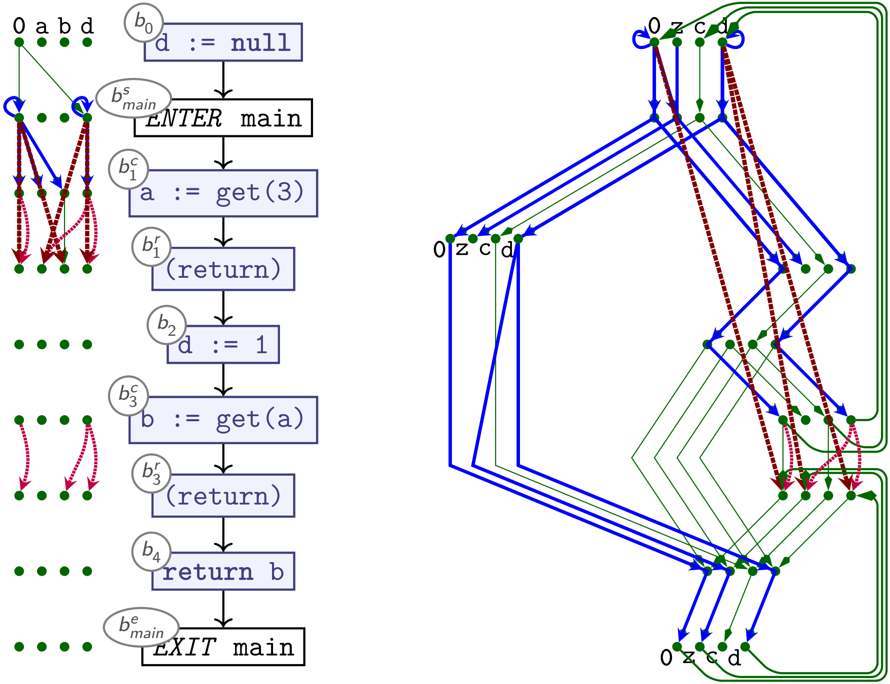
0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d



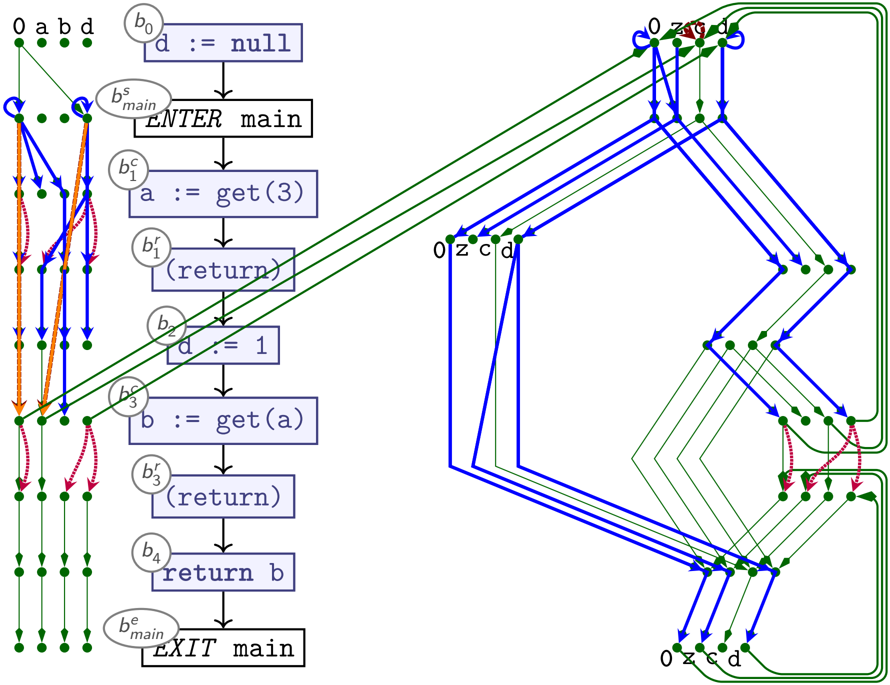


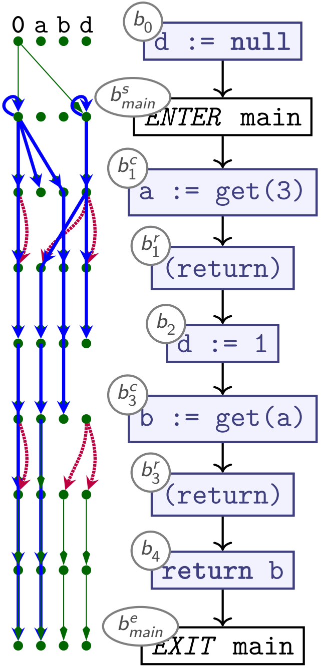
Step (call edge)

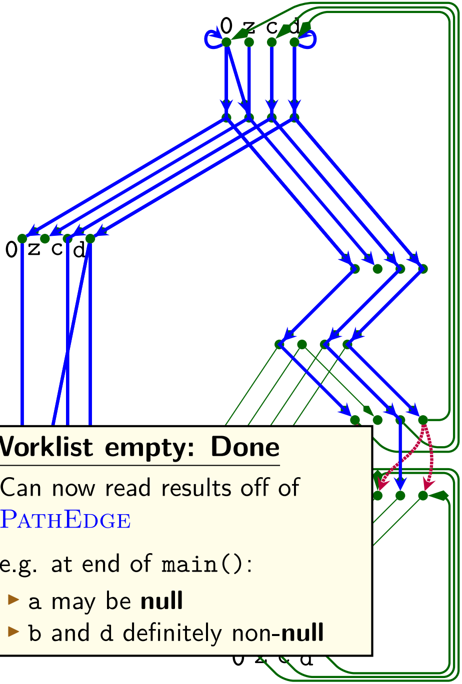
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v\rangle \in E^\sharp$
    - propagate$(\langle b_f^s, v\rangle \to \langle b_f^s, v\rangle)$
- **Propagate along intra-edges** (As with regular 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($\langle b^s_{\text{main}}, \mathbf{0}\rangle \to \langle b^s_{\text{main}}, \mathbf{0}\rangle$)
  ForwardTabulate()
end

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

IFDS: Forward Tabulation

Procedure ForwardTabulate():
begin
  while $n_0 \to n_1 \in \textsc{WorkList}$ **do**
    WorkList := **WorkList** $\setminus \{n_0 \to n_1\}$
    $\langle b_0, v_0 \rangle = n_0$; $\langle b_1, v_1 \rangle = n_1$
    if b_1 is neither *Call* nor *Exit* node **then**
      foreach $n_1 \to n_2 \in E^\sharp$:
        propagate($n_0 \to n_2$)
    else if b_1 is *Call* node **then begin**
      foreach call edge $n_1 \to n_2 \in E^\sharp$:
        propagate($n_2 \to n_2$)
      foreach non-call edge $n_1 \to n_2 \in E^\sharp \cup \textsc{SummaryInst}$:
        propagate($n_0 \to n_2$)
    end else if b_1 is *Exit* node **then begin**
      foreach caller/return node pair b_i^c, b_i^r that calls b_0 **and** vars v_0, v_1 **do**
        $n_s = \langle b_i^c, v_0 \rangle$; $n_r = \langle b_i^c, v_1 \rangle$
        if $\{n_s \to n_0, n_0 \to n_1, n_1 \to n_r\} \subseteq E^\sharp$ **and not** $n_s \to n_r \in \textsc{SummaryInst}$ **then**
          $\textsc{SummaryInst} := \textsc{SummaryInst} \cup \{n_s \to n_r\}$
          foreach $n_z \to n_s \in \textsc{PathEdge}$:
            propagate(n_z, n_r)
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**

Outlook

- More static analysis on Tuesday
- Lab 3 will go up today

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