LUND UNIVERSITY

EDA045F: Program Analysis

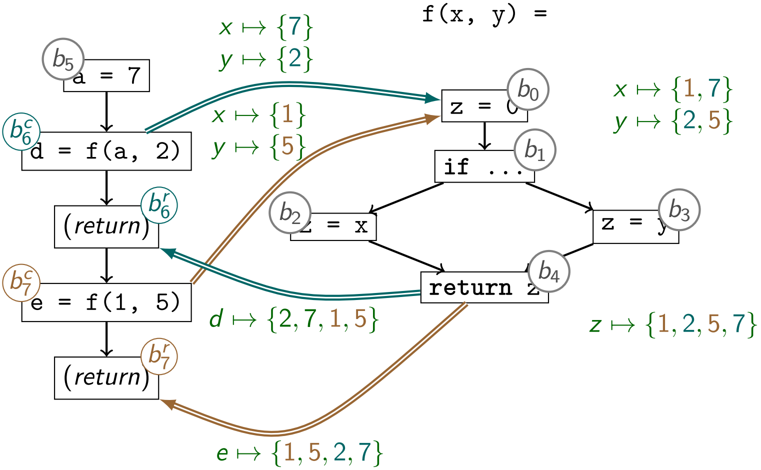
LECTURE 4: DATAFLOW ANALYSIS 3

Christoph Reichenbach

In the last lecture...

- Gen/Kill analyses
- Lattices
- Monotone Frameworks
- MFP algorithm
- MOP algorithm
- Distributive Frameworks
- Interprocedural Analysis
  - Context-sensitive vs. Context-insensitive analysis
  - Inlining for analysis

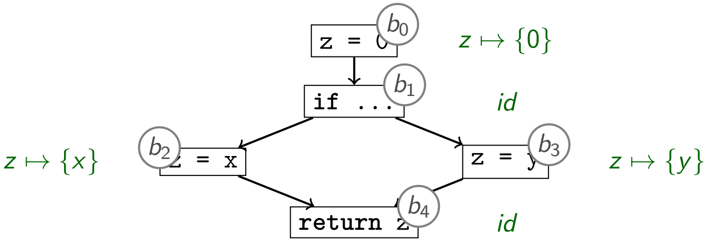
Interprocedural Data Flow Analysis



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

Alternative to Inlining: Summarise Procedure (Here: Reaching Defs.)

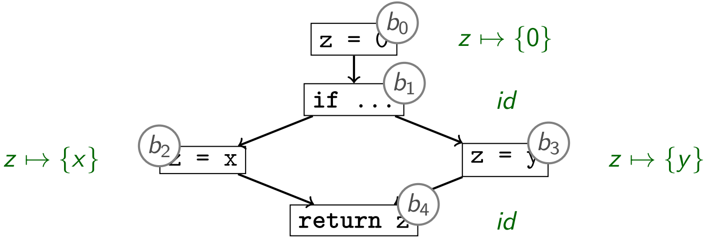
`f(x, y) =`



- *Compose* transfer functions:

Alternative to Inlining: Summarise Procedure (Here: Reaching Defs.)

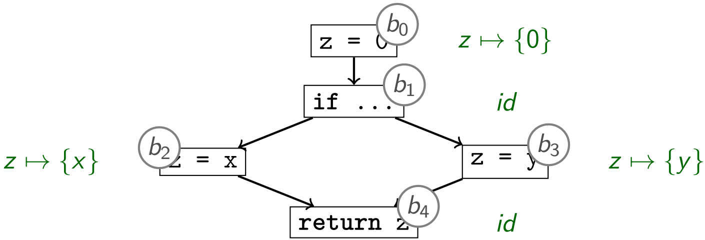
`f(x, y) =`



- *Compose* transfer functions:
  - $trans_{b_0} \circ trans_{b_1} = [z \mapsto 0]$

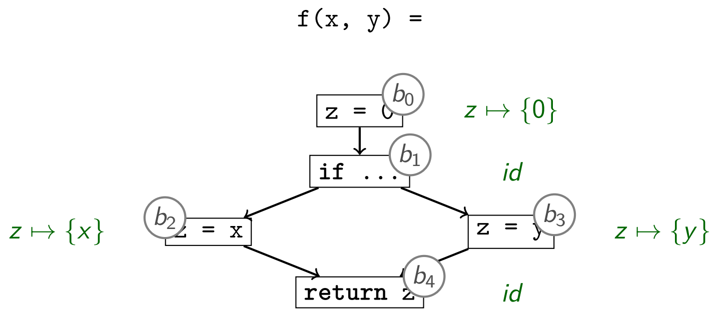
Alternative to Inlining: Summarise Procedure (Here: Reaching Defs.)

`f(x, y) =`



- *Compose* transfer functions:
  - $trans_{b_0} \circ trans_{b_1} = [z \mapsto 0]$
  - $trans_{b_0} \circ trans_{b_1} \circ trans_{b_2} = [z \mapsto \{x\}]$

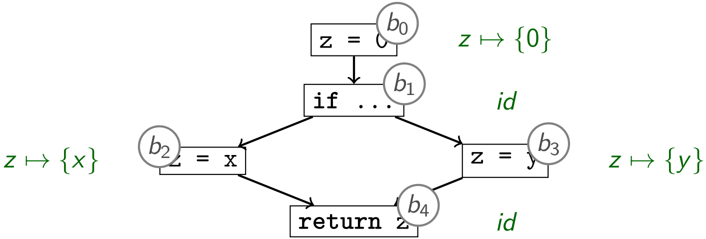
Alternative to Inlining: Summarise Procedure (Here: Reaching Defs.)



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

Alternative to Inlining: Summarise Procedure (Here: Reaching Defs.)

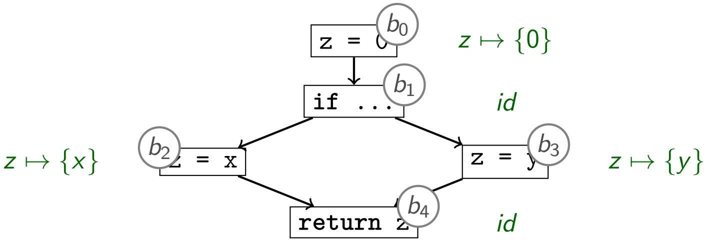
`f(x, y) =`



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

Alternative to Inlining: Summarise Procedure (Here: Reaching Defs.)

`f(x, y) =`



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

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

Procedure Summaries

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

$$trans_f = [\textbf{return} \mapsto \{x, y\}]$$

- Use $trans_f$ as transfer function for `f()`, discard `f`'s body

Procedure Summaries

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

$$trans_f = [\textbf{return} \mapsto \{x, y\}]$$

- Use $trans_f$ as transfer function for `f()`, discard `f`'s body
- **Advantages:**
  - Can yield compact subroutine descriptions
  - Can speed up call site analysis dramatically

Procedure Summaries

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

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

- Use trans_f as transfer function for `f()`, discard `f`'s body
- **Advantages:**
  - Can yield compact subroutine descriptions
  - Can speed up call site analysis dramatically
- **Disadvantages:**
  - More complex to implement
  - Recursion is challenging

Procedure Summaries

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

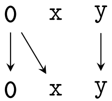
$$trans_f = [\textbf{return} \mapsto \{x, y\}]$$

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

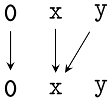
Representation Relations

Example procedure summary representation:

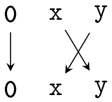
```
x = null
y = y
```




```
x = (y == null)?y:x
y = 1
```




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



'May be null' analysis

- 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 $\mathbf{out}_b$

Summary

- *Procedure summaries* built from composed transfer functions
- Can speed up context-sensitive analysis of popular functions, compared to inlining
- Needs some suitably abstract analysis *for the given program*
  - Example: IFDS-style **Representation Relations**
- Recursion is nontrivial:
  - Analyse function calls (*call graph*)
  - Analyse strongly connected components together

Composing Representation Relations

Recall Representation Relations (*may be null* analysis):

```
x = null
y = y
```

$$\begin{tikzcd} 0 \arrow[d] \arrow[dr] & x & y \arrow[d] \\ 0 & x & y \end{tikzcd}$$

```
x = (y == null)?y:x
y = 1
```

$$\begin{tikzcd} 0 \arrow[d] & x \arrow[d] & y \arrow[dl] \\ 0 & x & y \end{tikzcd}$$

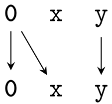
```
{ t = x
  x = y
  y = t }
```

$$\begin{tikzcd} 0 \arrow[d] & x \arrow[dr] & y \arrow[dl] \\ 0 & x & y \end{tikzcd}$$

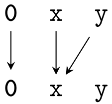
Composing Representation Relations

Recall Representation Relations (*may be null* analysis):

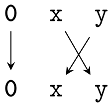
```
x = null
y = y
```




```
x = (y == null)?y:x
y = 1
```




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



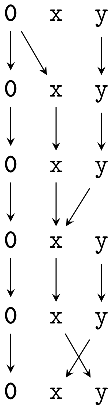
Composing Representation Relations

Recall Representation Relations (*may be null* analysis):

```
x = null
y = y
```

```
x = (y == null)?y:x
y = 1
```

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



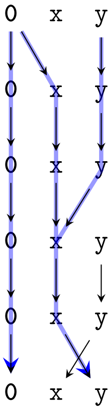
Composing Representation Relations

Recall Representation Relations (*may be null* analysis):

```
x = null
y = y
```

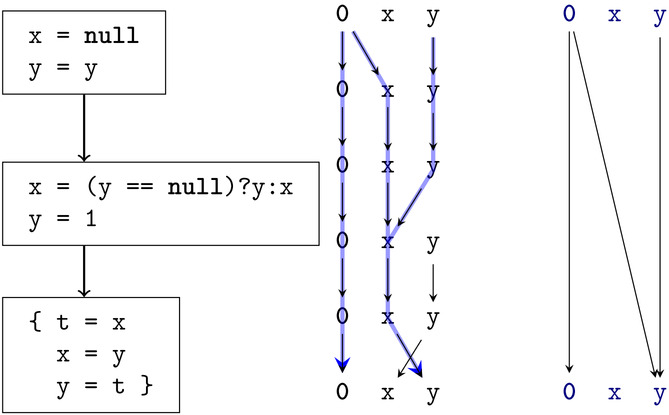
```
x = (y == null)?y:x
y = 1
```

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



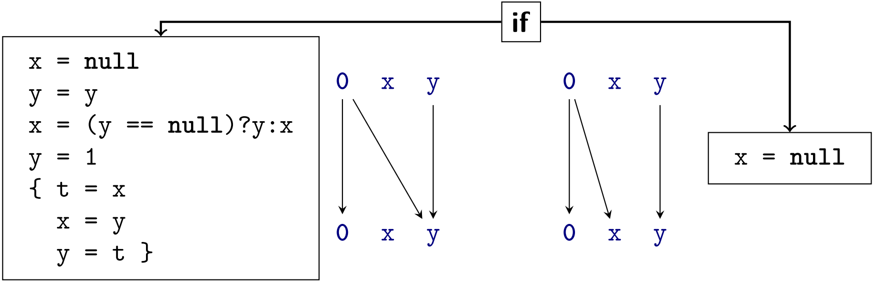
Composing Representation Relations

Recall Representation Relations (*may be null* analysis):

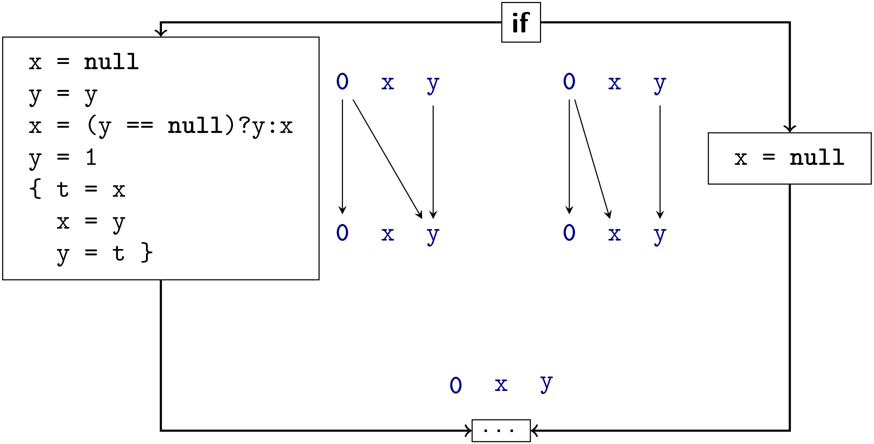


Composed representation relations are again representation relations

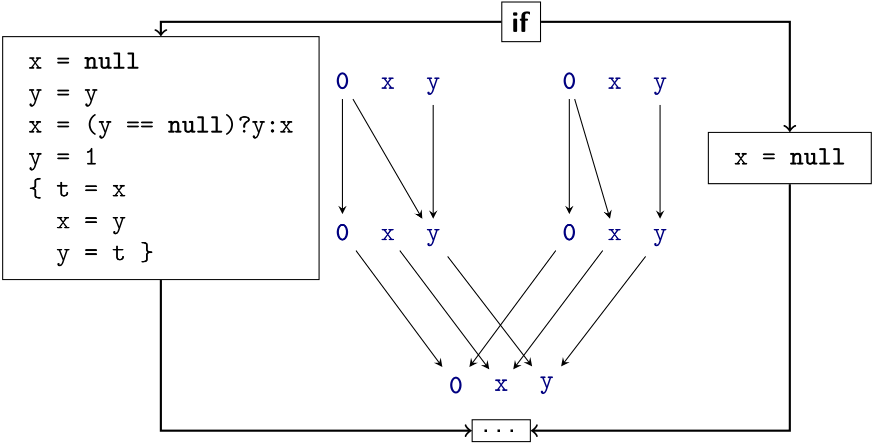
Merging Control-Flow Paths



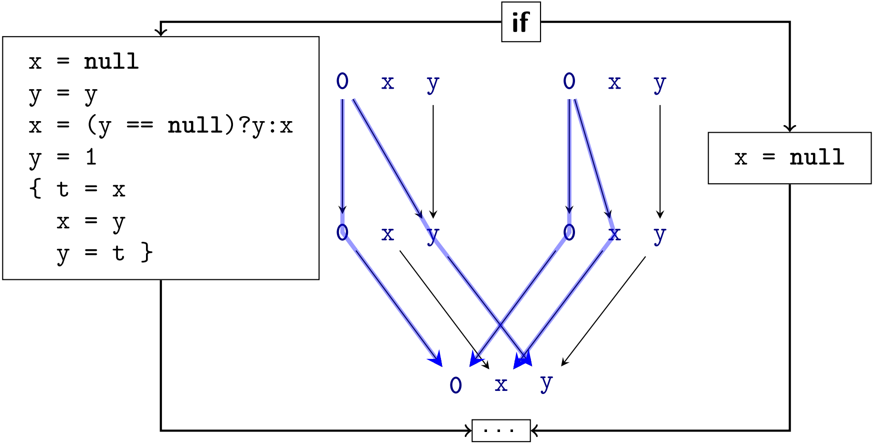
Merging Control-Flow Paths



Merging Control-Flow Paths



Merging Control-Flow Paths



Behaves analogous to disjunction / ‘May’ analysis: if reachable from 0 then *may be true*

Dataflow via Graph Reachability

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

- Assume binary latice $(\{\top, \bot\}, \sqsubseteq, \sqcap, \sqcup)$
  - $a \sqcap b = \bot$ iff $a = \bot$ and $b = \bot$, otherwise $a \sqcap b = \top$
  - Typical for 'May' analysis ('may be `null`')

- We can encode Dataflow problem as *Graph-Reachability*
- Graph nodes $n = \langle b, v \rangle$
  - b: CFG node
  - v: Variable or $\mathbf{0}$
    - Variable: Property of interest connected to variable
    - $\mathbf{0}$: Property of interest connected to executing this statement/block

Dataflow via Graph Reachability

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

- Assume binary latice $(\{\top, \bot\}, \sqsubseteq, \sqcap, \sqcup)$
  - $a \sqcap b = \bot$ iff $a = \bot$ and $b = \bot$, otherwise $a \sqcap b = \top$
  - Typical for 'May' analysis ('may be `null`')
  - Equivalently for 'Must' analysis:
    'must be `null`' = not ('may be `non-null`')
- We can encode Dataflow problem as *Graph-Reachability*
- Graph nodes $n = \langle b, v \rangle$
  - b: CFG node
  - v: Variable or $\mathbf{0}$
    - Variable: Property of interest connected to variable
    - $\mathbf{0}$: Property of interest connected to executing this statement/block

A Dataflow Worklist Algorithm: IFDS

- Context-sensitive interprocedural dataflow algorithm
- Historical name: IFDS
  (**I**nterprocedural **F**inite **D**istributive **S**ubset problems)
- 'Exploded Supergraph': $G^\sharp = (N^\sharp, E^\sharp)$
  - $N^\sharp = N_{\text{CFG}} \times \mathcal{V} \cup \{0\}$
  - Plus parameter/return call edges
- b^s_{main} is the CFG *ENTER* node of the main entry point
- Property-of-interest holds if reachable from $\langle b^s_{\text{main}}, \mathbf{0} \rangle$
- **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

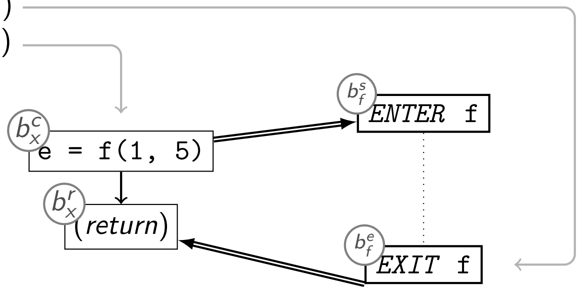
$N^\sharp$-edge

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

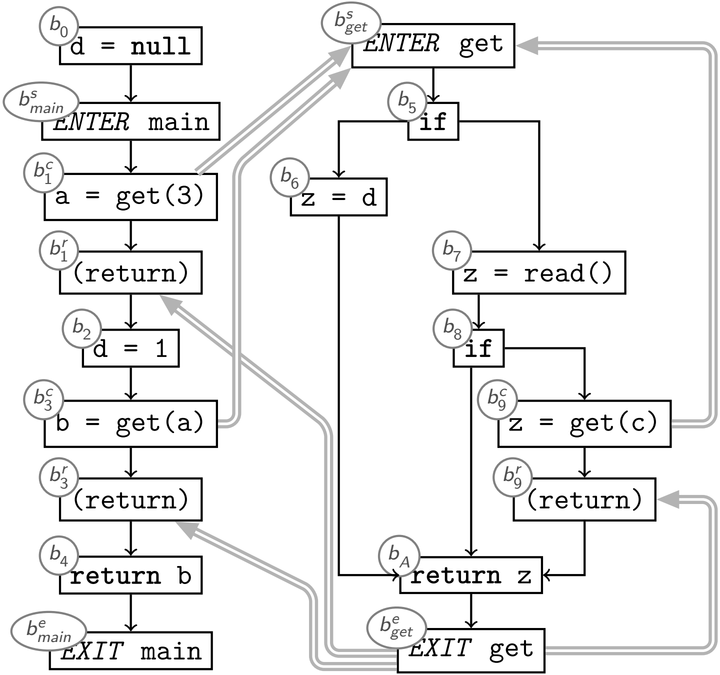
Running Example

ATL: *main()*

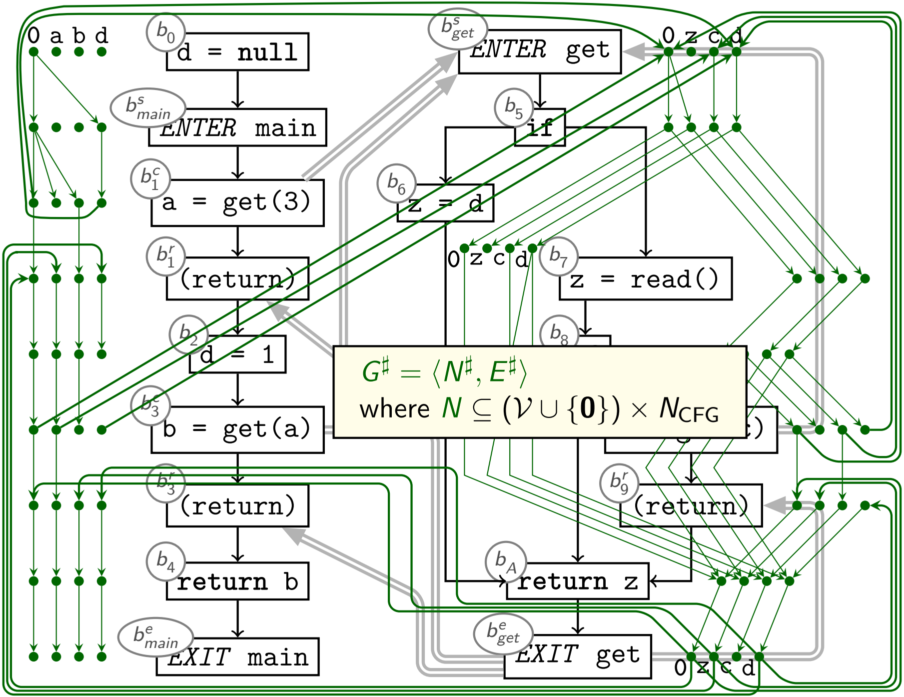
```
global default = null
proc main() {
  a = get(3)
  default = 1
  b = get(3)
  return b
}
```

ATL: *get()*

```
proc get(c) {
  if c == 0 {
    z = default
  } else {
    z = read()
    if z < 0 {
      z = get(c + -1)
    } else skip
  }
  return z
}
```

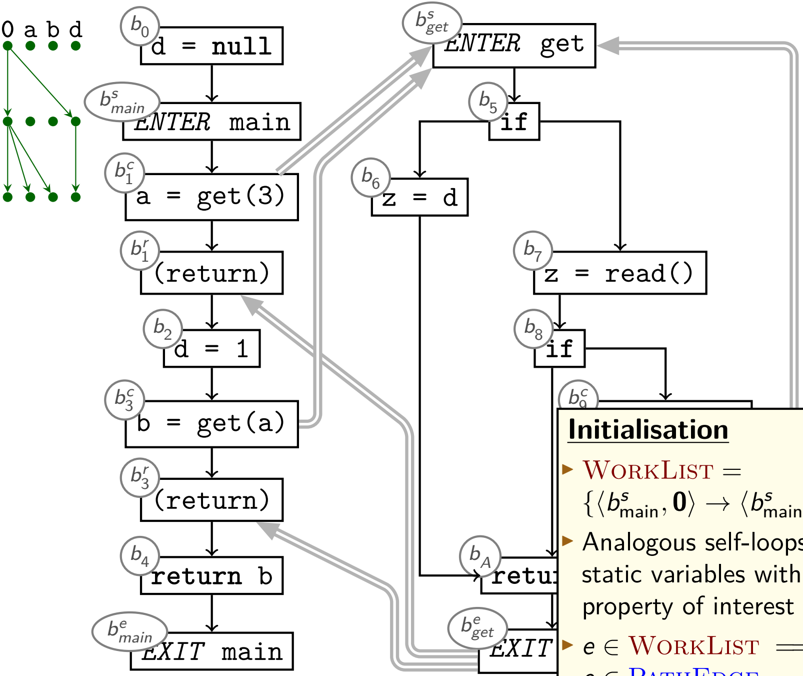


b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z = d
b_7
z = read()
b_8
if
b^c_9
z = get(c)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get



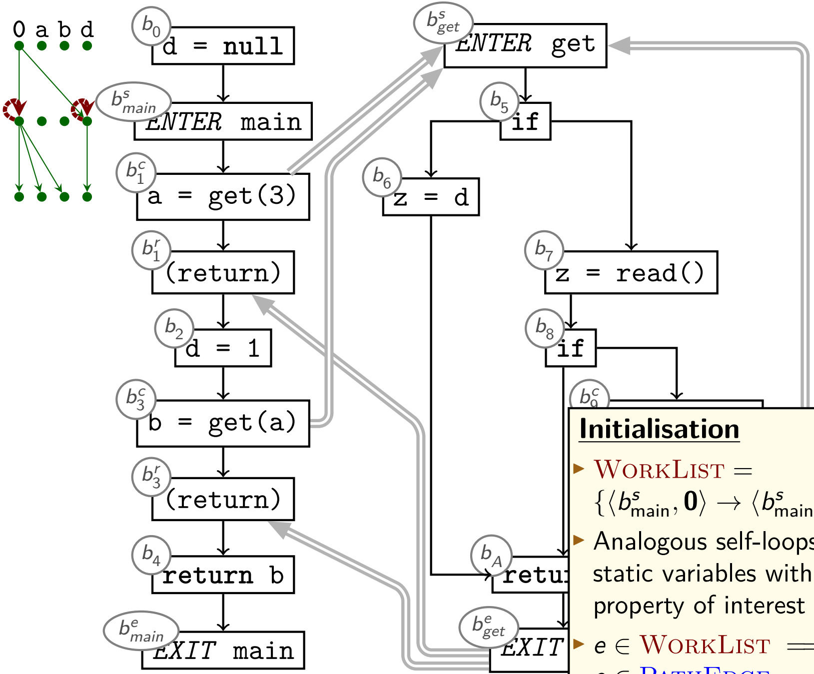
$G^\sharp = \langle N^\sharp, E^\sharp \rangle$

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



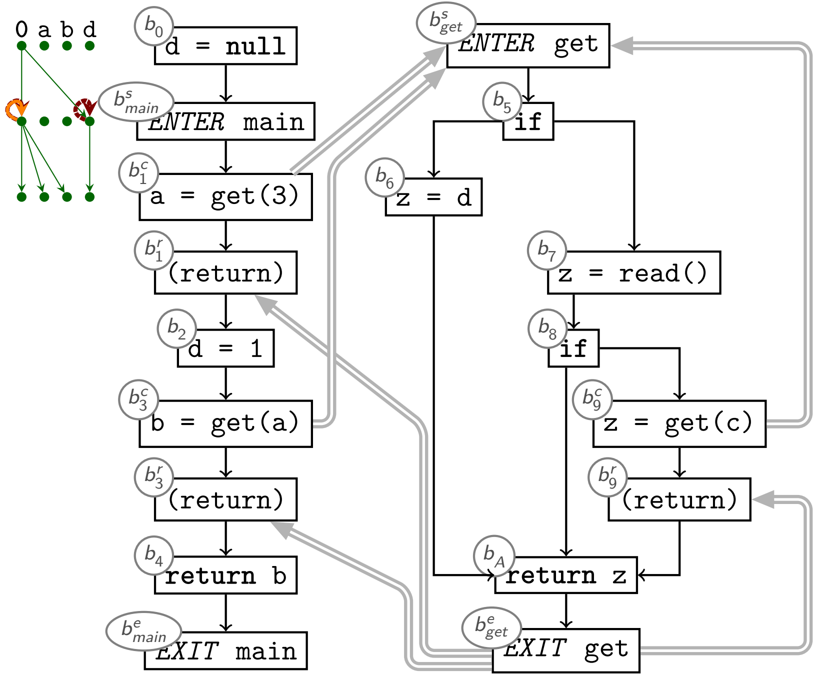
Initialisation

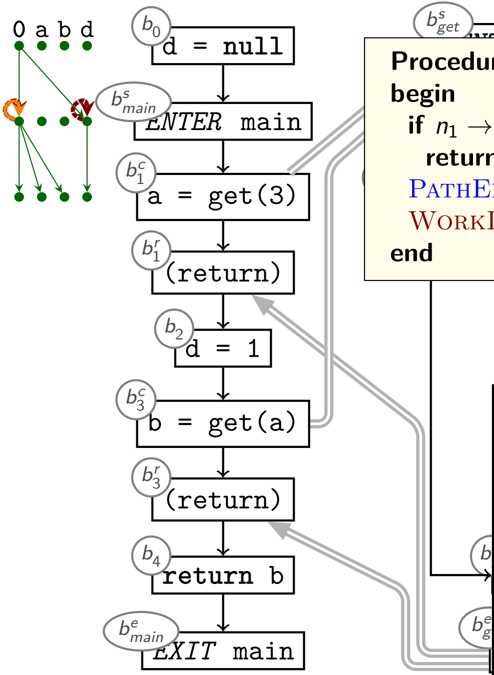
- WorkList = $\{\langle b^s_{\mathrm{main}}, \mathbf{0}\rangle \to \langle b^s_{\mathrm{main}}, \mathbf{0}\rangle\}$
- Analogous self-loops for static variables with property of interest (d)
- $e \in$ WorkList $\implies e \in$ PathEdge



Initialisation

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

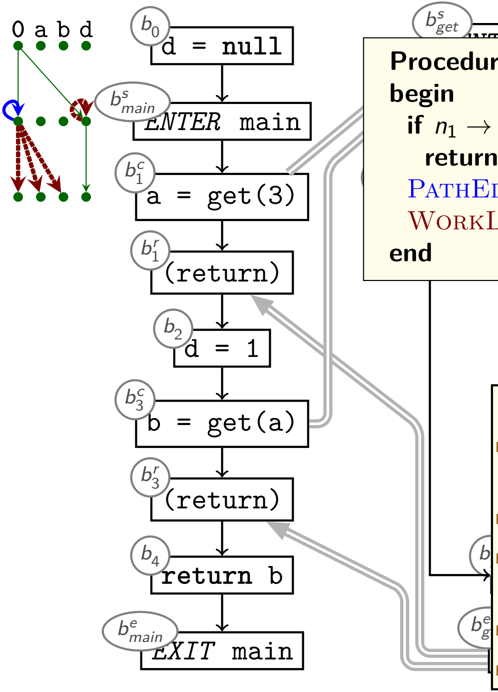

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z = d
b_7
z = read()
b_8
if
b^c_9
z = get(c)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get



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)

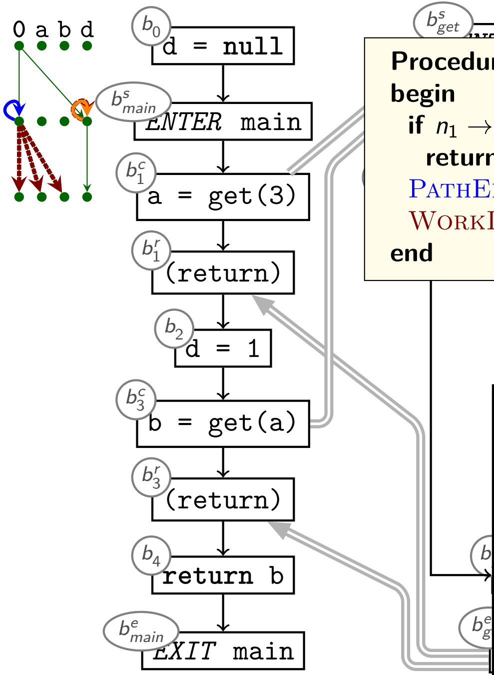
- Pick e off the work queue
  $e = n_1 \to n_2$
- n_2 neither call (c) nor exit (e)?
- Find all $n_2 \to n_3$
  propagate($n_1 \to n_3$)
- Remove e from WORKLIST
- e remains in PATHEDGE



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

Step (regular edge)

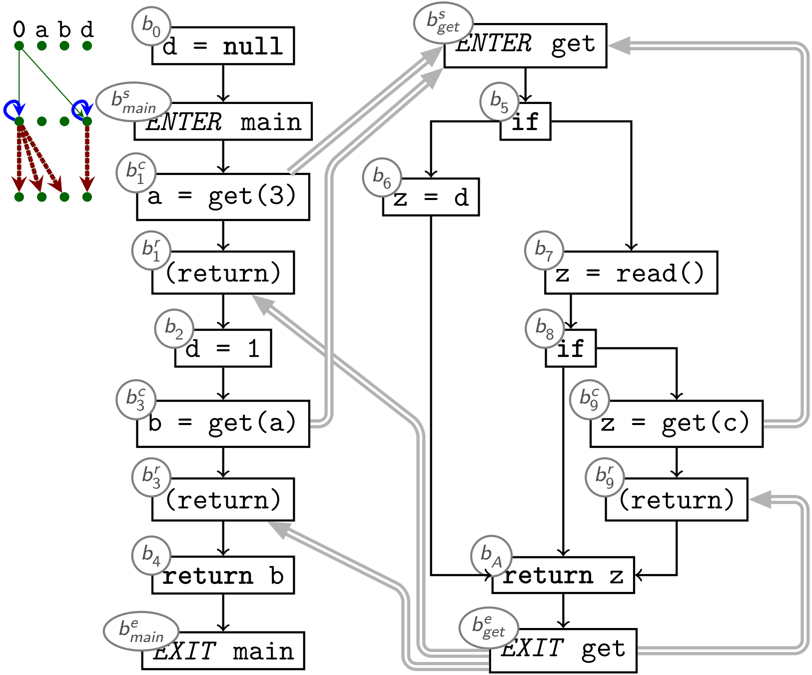
- Pick e off the work queue
  $e = n_1 \to n_2$
- n_2 neither call (c) nor exit (e)?
- Find all $n_2 \to n_3$
  propagate($n_1 \to n_3$)
- Remove e from WorkList
- e remains in PathEdge



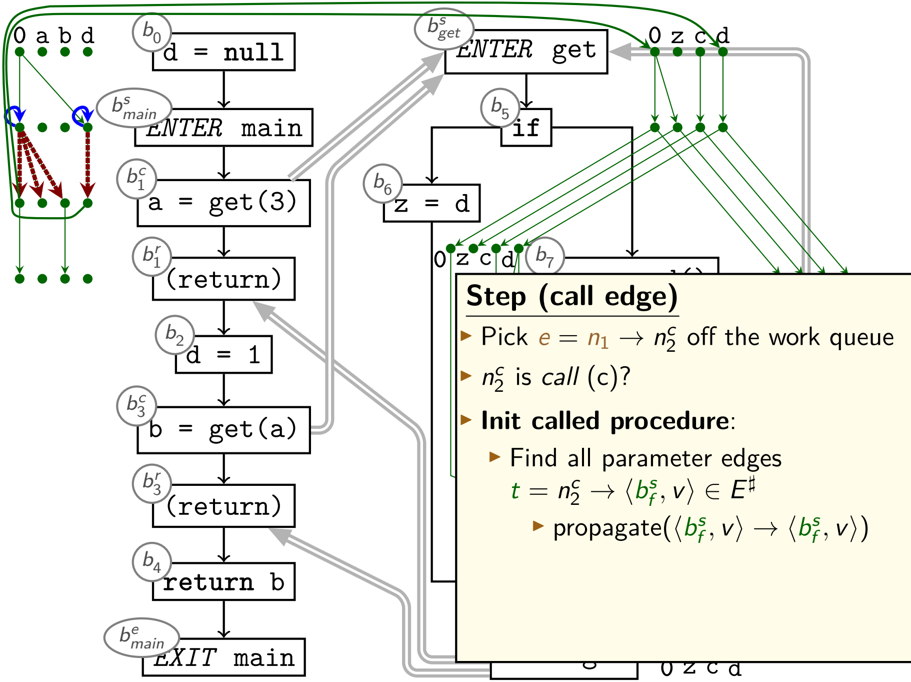
Procedure propagate($n_1 \to n_2$):
begin
  if $n_1 \to n_2 \in \textsc{PathEdge}$ **then**
    return
  $\textsc{PathEdge} := \textsc{PathEdge} \cup \{n_1 \to n_2\}$
  $\textsc{WorkList} := \textsc{WorkList} \cup \{n_1 \to n_2\}$
end

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 $\textsc{WorkList}$
- e remains in $\textsc{PathEdge}$

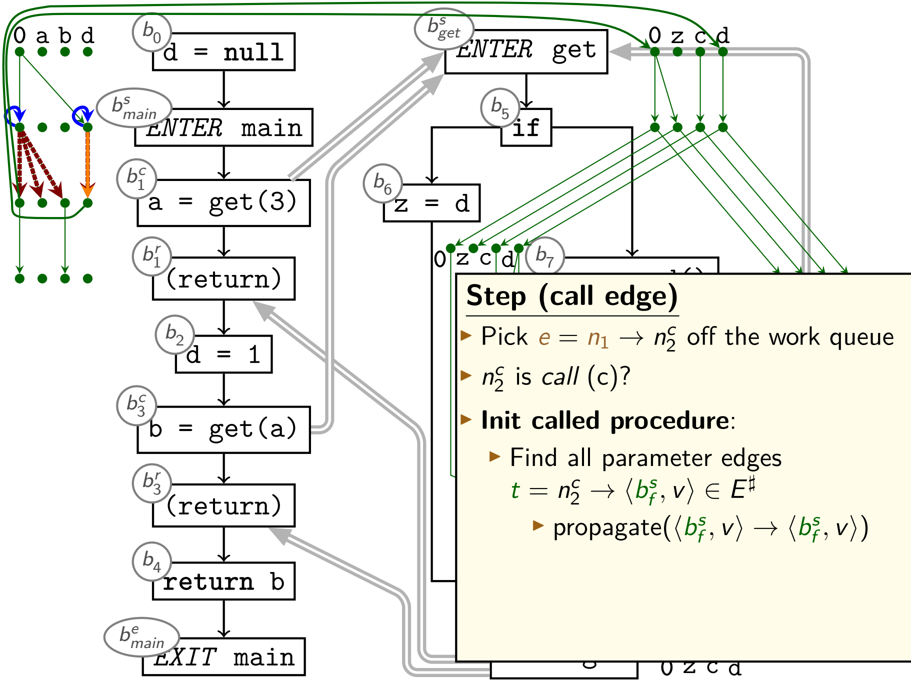


0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z = d
b_7
z = read()
b_8
if
b^c_9
z = get(c)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get



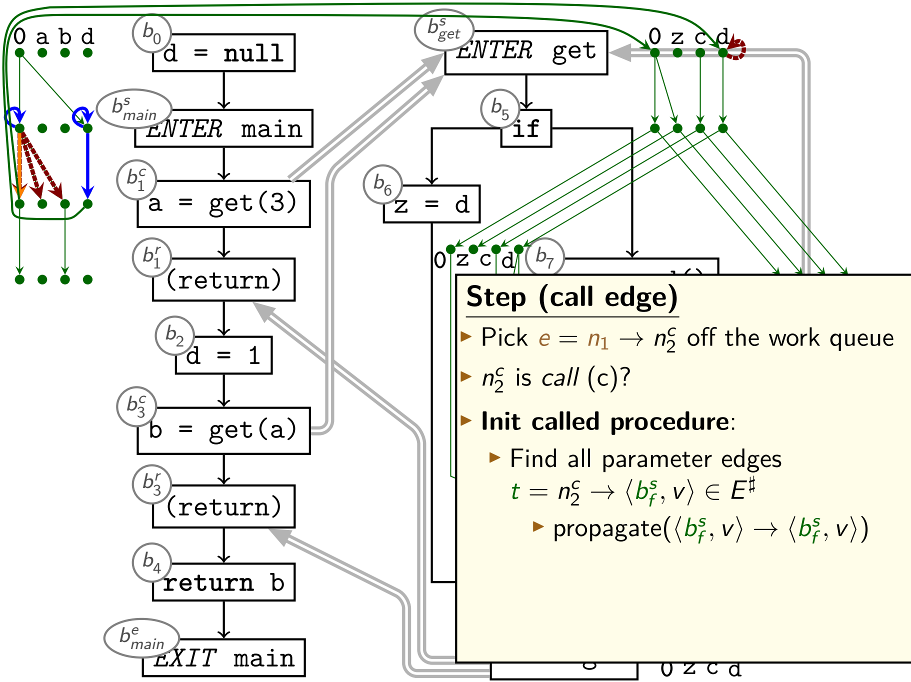
Step (call edge)

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



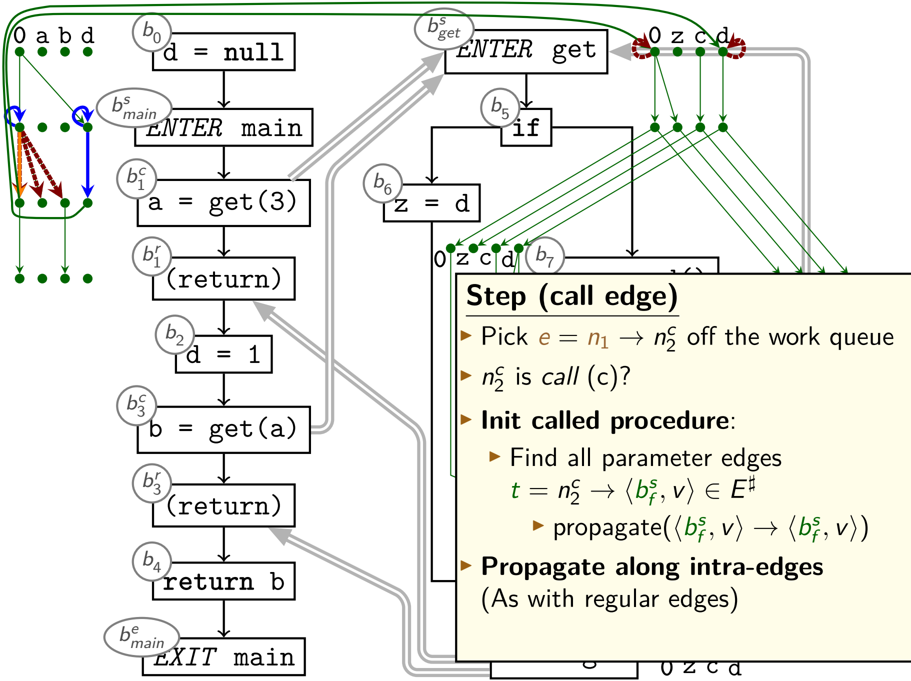
Step (call edge)

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



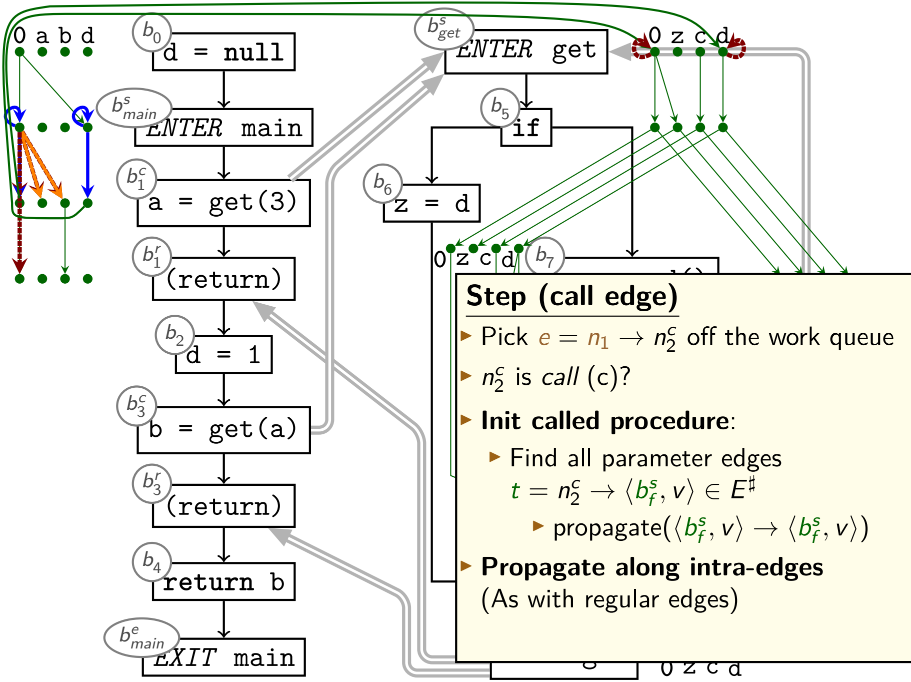
Step (call edge)

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



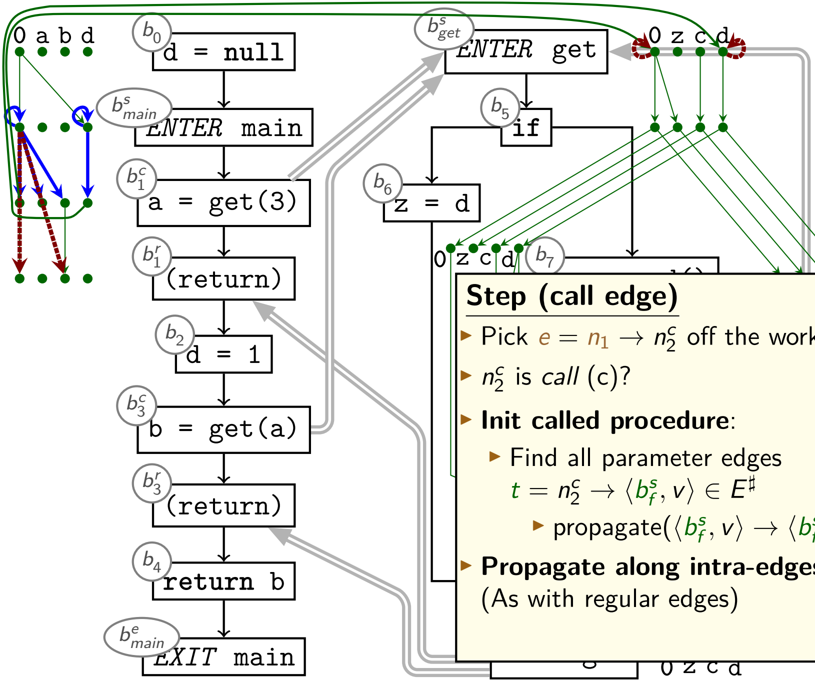
Step (call edge)

- 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)



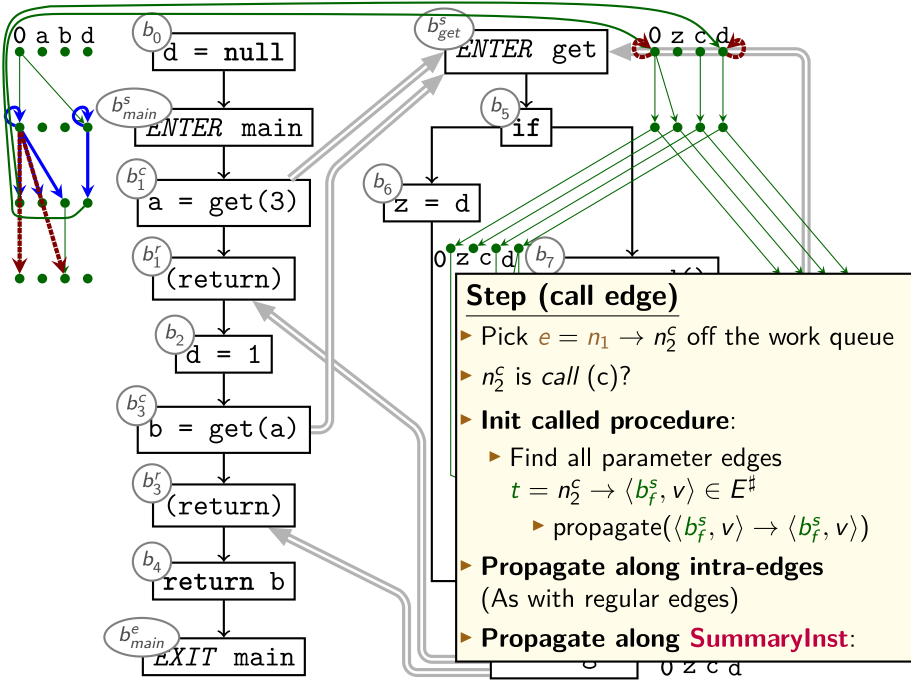
Step (call edge)

- Pick $e = n_1 \rightarrow n_2^c$ 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)



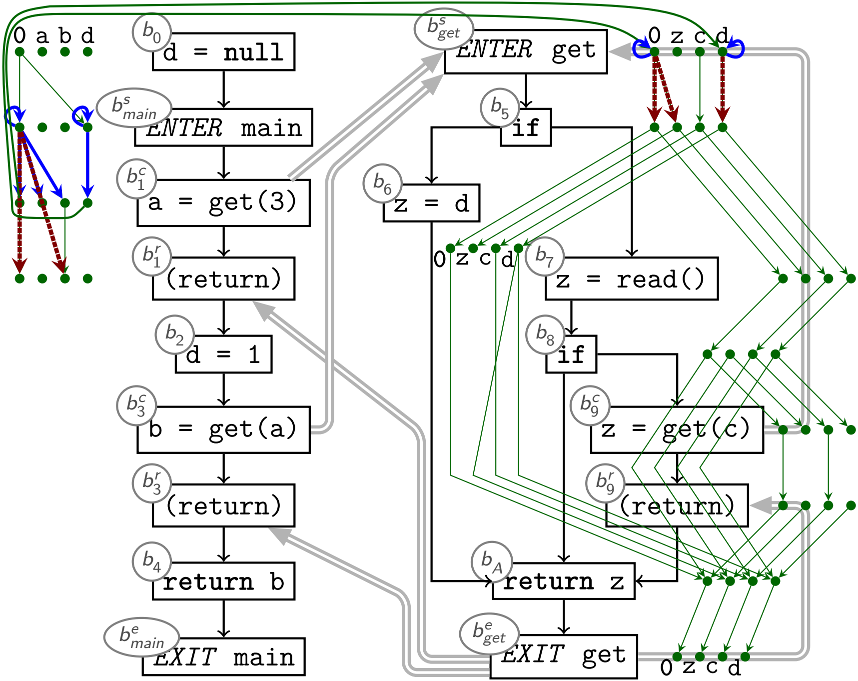
Step (call edge)

- 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)

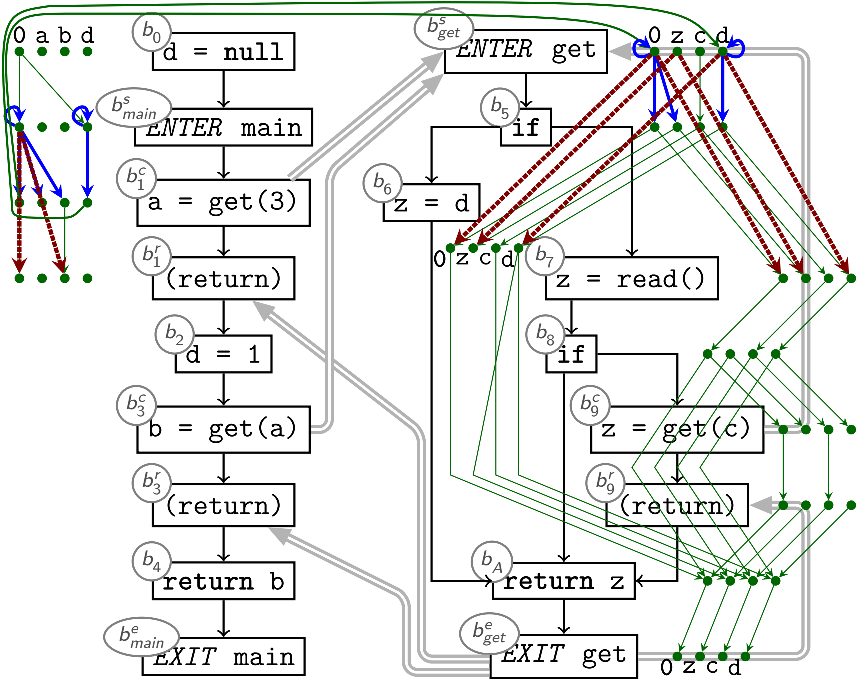


Step (call edge)

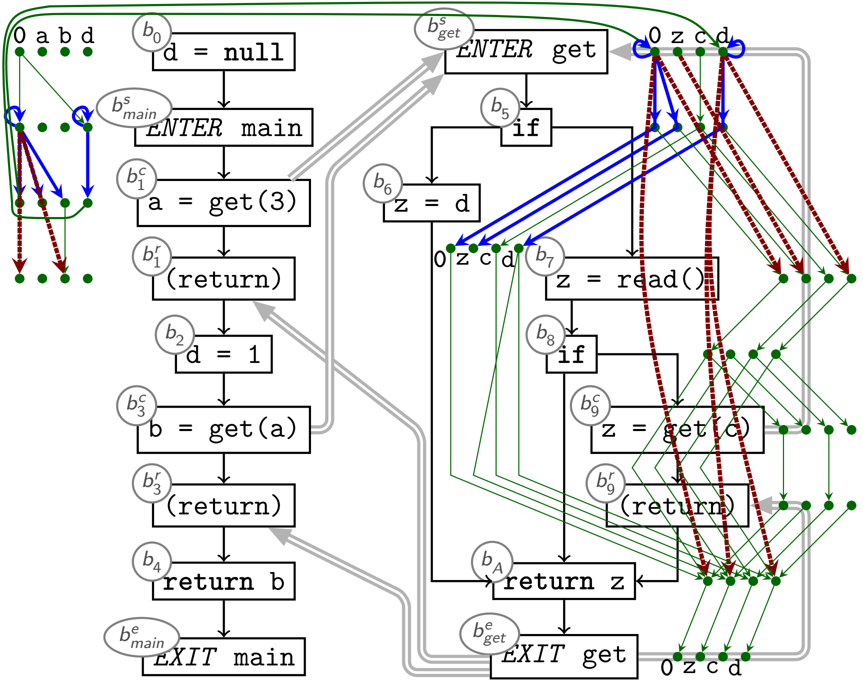
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v\rangle \in E^\sharp$
    - propagate($\langle b_f^s, v\rangle \to \langle b_f^s, v\rangle$)
- **Propagate along intra-edges** (As with regular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

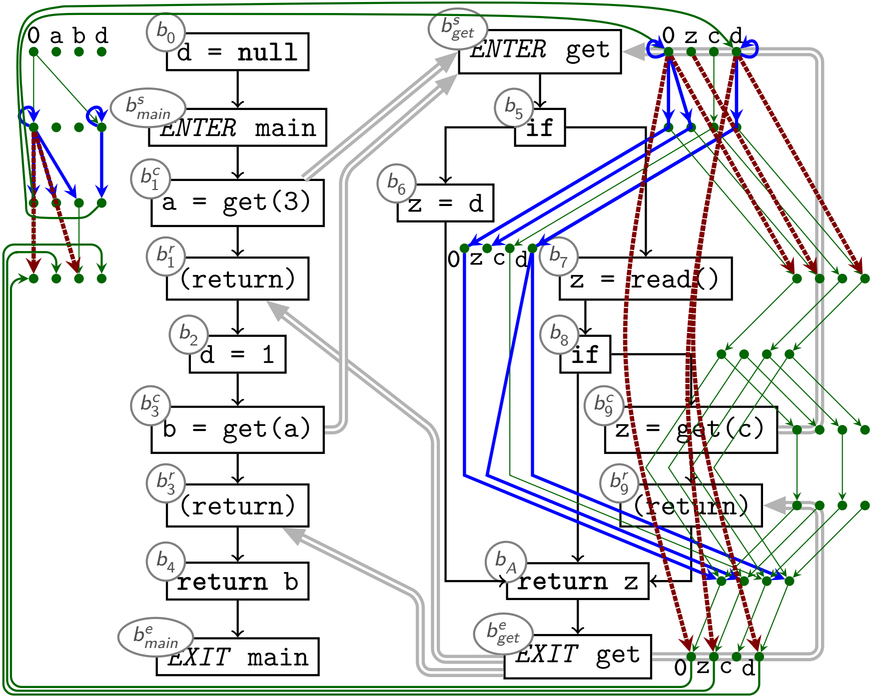


0 a b d
b_0 d = null
b^s_{main} ENTER main
b^c_1 a = get(3)
b^r_1 (return)
b_2 d = 1
b^c_3 b = get(a)
b^r_3 (return)
b_4 return b
b^e_{main} EXIT main
b^s_{get} ENTER get
0 z c d
b_5 if
b_6 z = d
0 z c d
b_7 z = read()
b_8 if
b^c_9 z = get(c)
b^r_9 (return)
b_A return z
b^e_{get} EXIT get
0 z c d

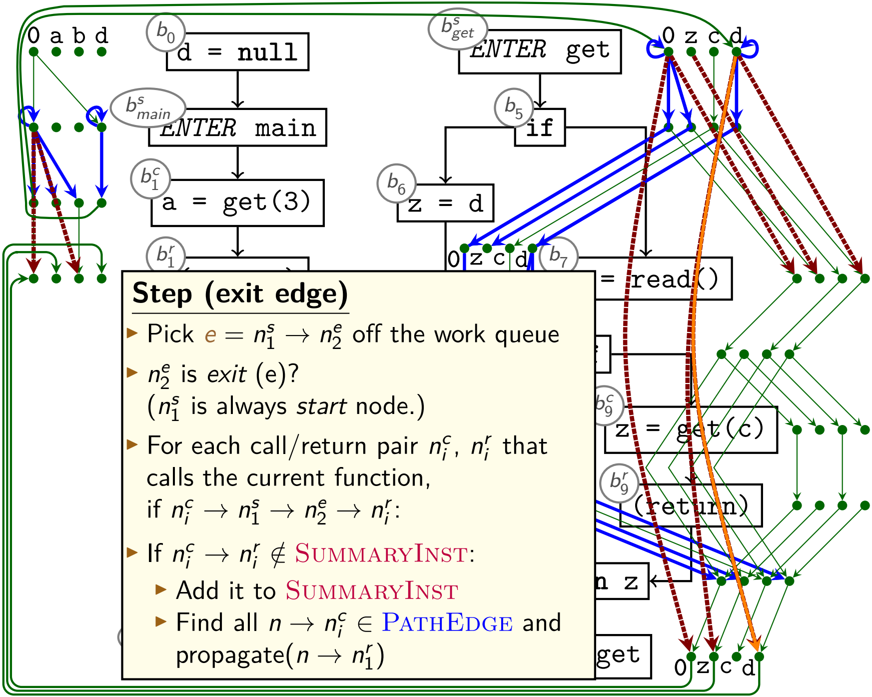


0 a b d
b_0 d = null
b^s_{main} ENTER main
b^c_1 a = get(3)
b^r_1 (return)
b_2 d = 1
b^c_3 b = get(a)
b^r_3 (return)
b_4 return b
b^e_{main} EXIT main
b^s_{get} ENTER get
0 z c d
b_5 if
b_6 z = d
0 z c d
b_7 z = read()
b_8 if
b^c_9 z = get(c)
b^r_9 (return)
b_A return z
b^e_{get} EXIT get
0 z c d


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 = read()
b_8
if
b^c_9
z = get(c)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d

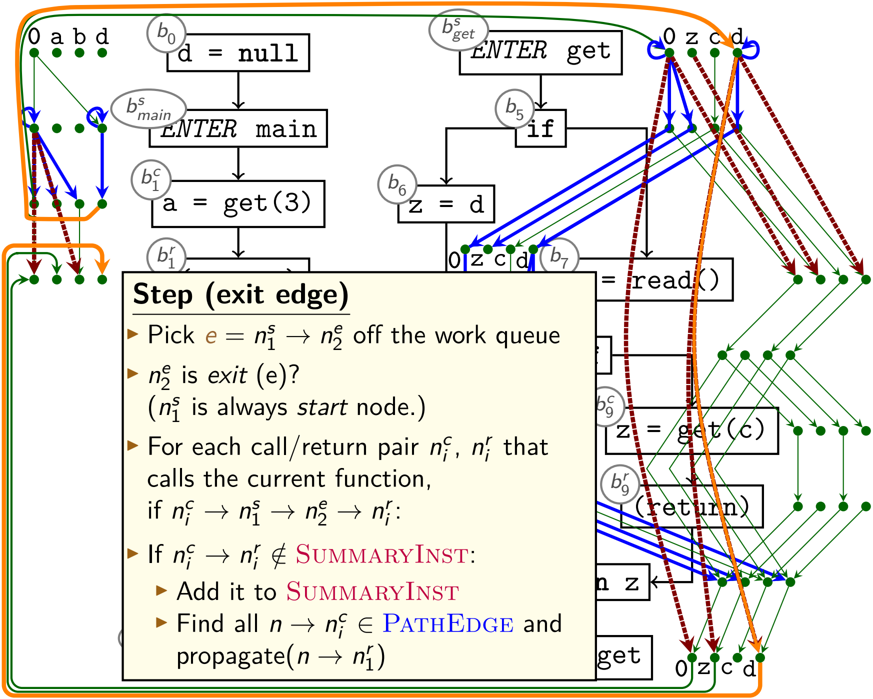


0 a b d
b_0 d = null
b^s_{main} ENTER main
b^c_1 a = get(3)
b^r_1 (return)
b_2 d = 1
b^c_3 b = get(a)
b^r_3 (return)
b_4 return b
b^e_{main} EXIT main
b^s_{get} ENTER get
0 z c d
b_5 if
b_6 z = d
0 z c d
b_7 z = read()
b_8 if
b^c_9 z = get(c)
b^r_9 (return)
b_A return z
b^e_{get} EXIT get
0 z c d



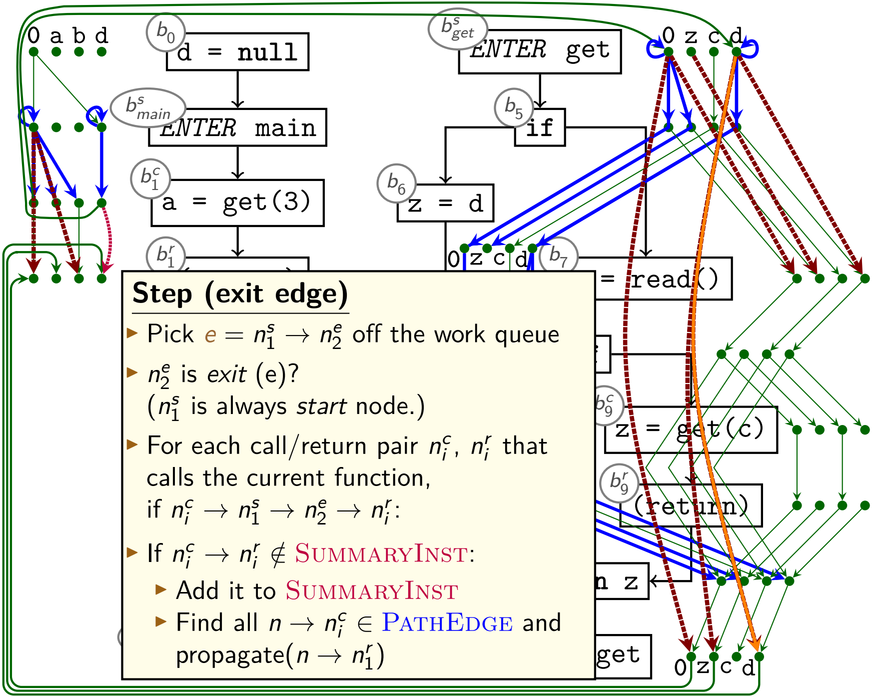
Step (exit edge)

- Pick $e = n^s_1 \to n^e_2$ off the work queue
- n^e_2 is *exit* (e)?
  (n^s_1 is always *start* node.)
- For each call/return pair n^c_i, n^r_i that calls the current function,
  if $n^c_i \to n^s_1 \to n^e_2 \to n^r_i$:
- If $n^c_i \to n^r_i \notin$ SUMMARYINST:
  - Add it to SUMMARYINST
  - Find all $n \to n^c_i \in$ PATHEDGE and propagate($n \to n^r_1$)



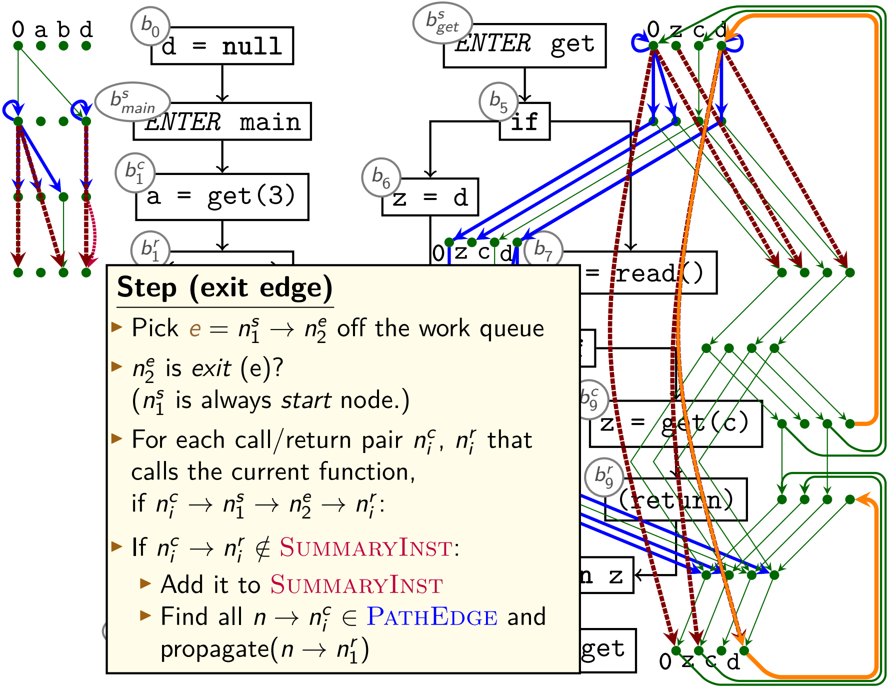
Step (exit edge)

- 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($n \to n_1^r$)



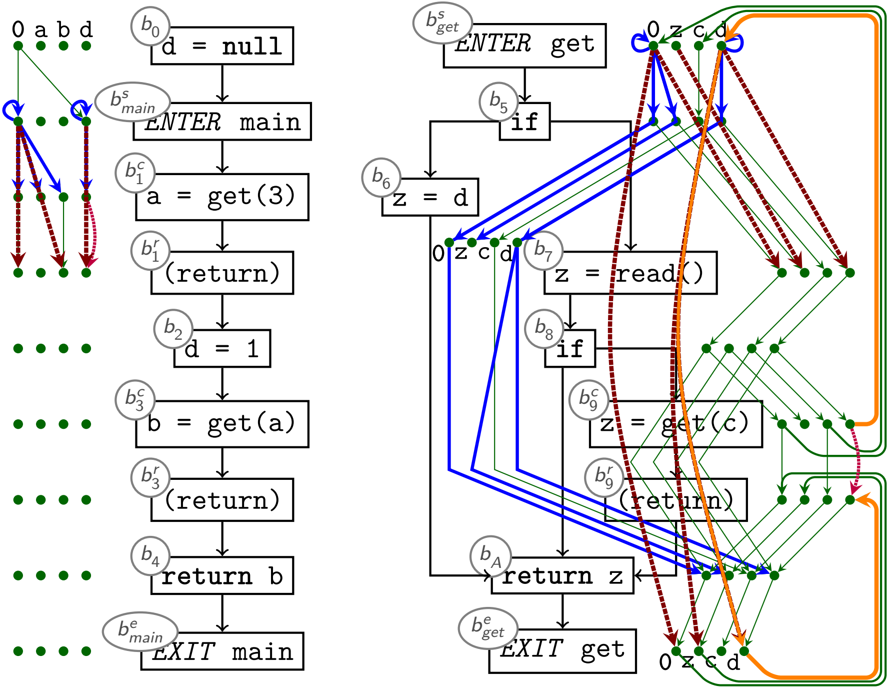
Step (exit edge)

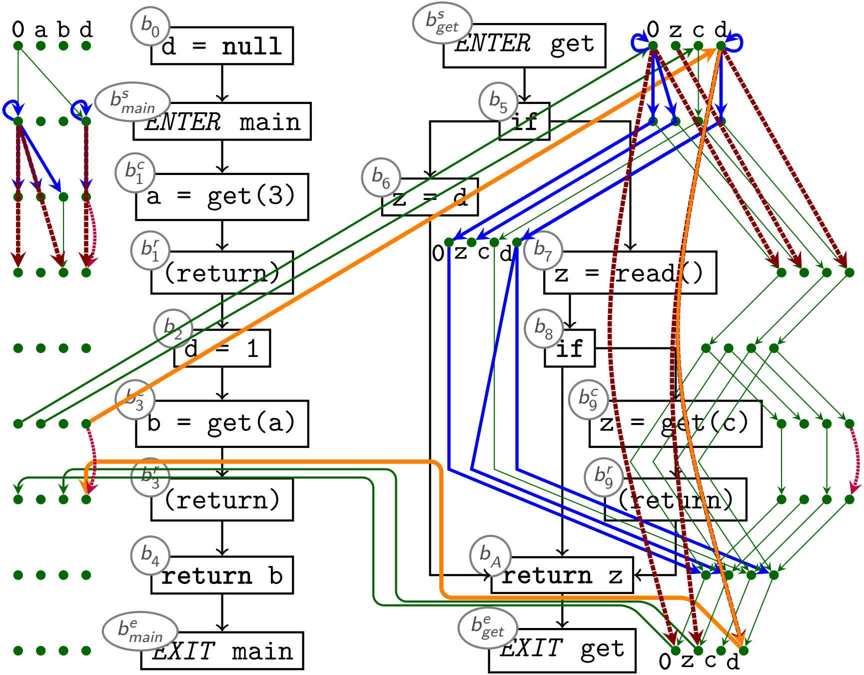
- Pick $e = n^s_1 \to n^e_2$ off the work queue
- n^e_2 is *exit* (e)?
  (n^s_1 is always *start* node.)
- For each call/return pair n^c_i, n^r_i that calls the current function, if $n^c_i \to n^s_1 \to n^e_2 \to n^r_i$:
- If $n^c_i \to n^r_i \notin$ SUMMARYINST:
  - Add it to SUMMARYINST
  - Find all $n \to n^c_i \in$ PATHEDGE and propagate($n \to n^r_1$)

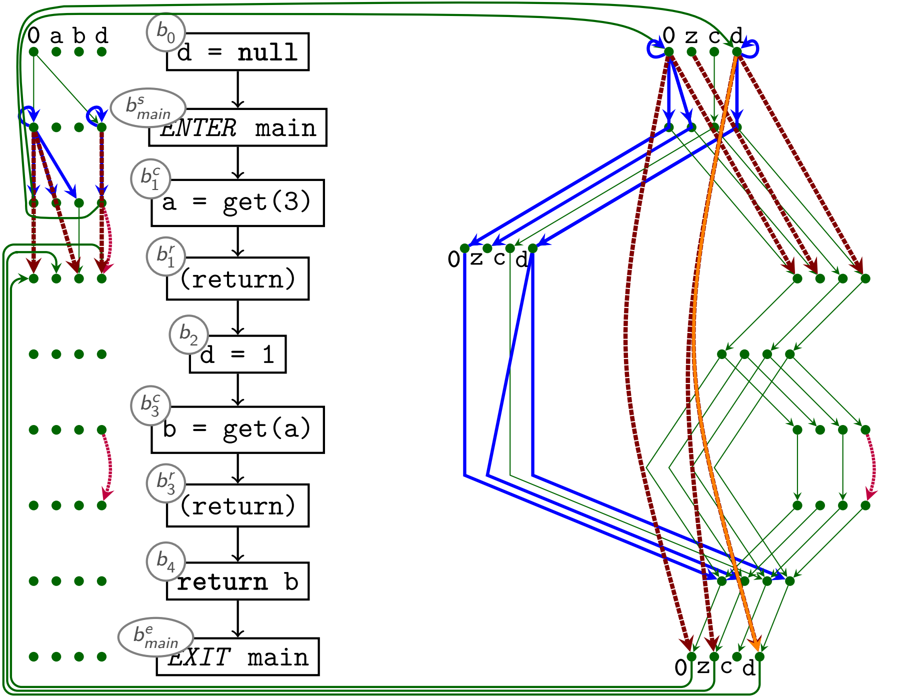


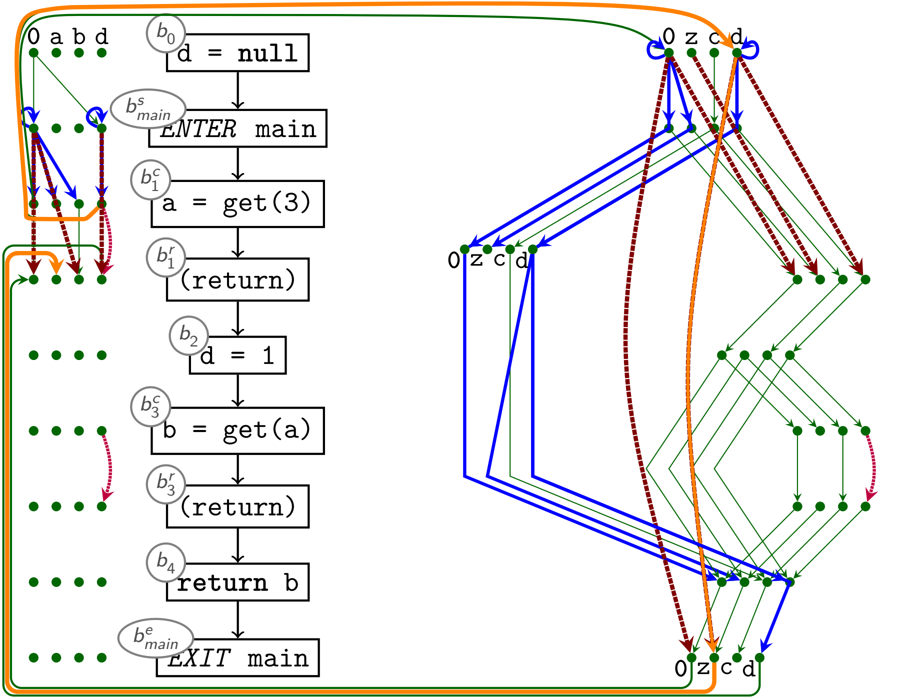
Step (exit edge)

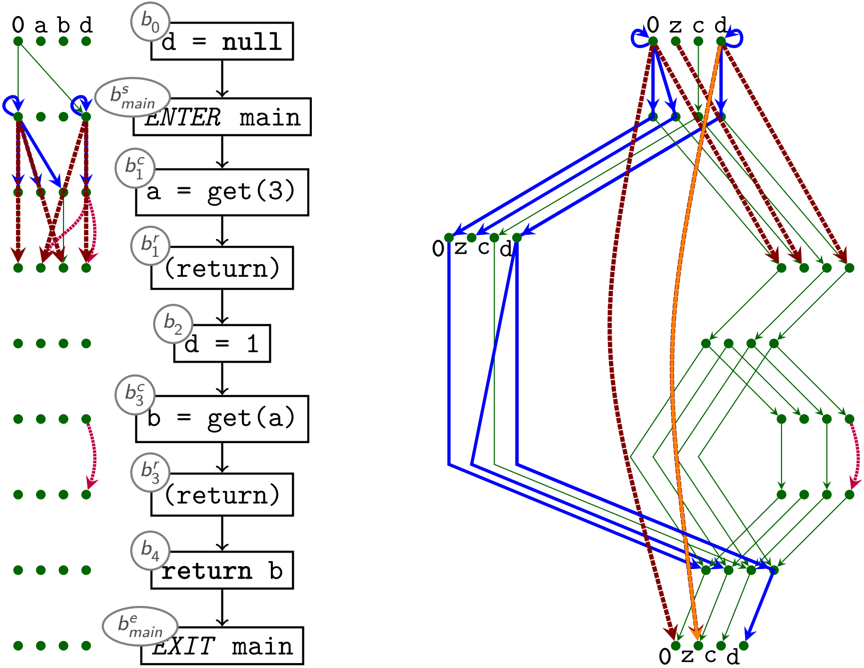
- Pick $e = n^s_1 \rightarrow n^e_2$ off the work queue
- n^e_2 is *exit* (e)?
  (n^s_1 is always *start* node.)
- For each call/return pair n^c_i, n^r_i that calls the current function,
  if $n^c_i \rightarrow n^s_1 \rightarrow n^e_2 \rightarrow n^r_i$:
- If $n^c_i \rightarrow n^r_i \notin$ SUMMARYINST:
  - Add it to SUMMARYINST
  - Find all $n \rightarrow n^c_i \in$ PATHEDGE and propagate($n \rightarrow n^r_1$)


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 = read()
b_8
if
b^c_9
z = get(c)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d

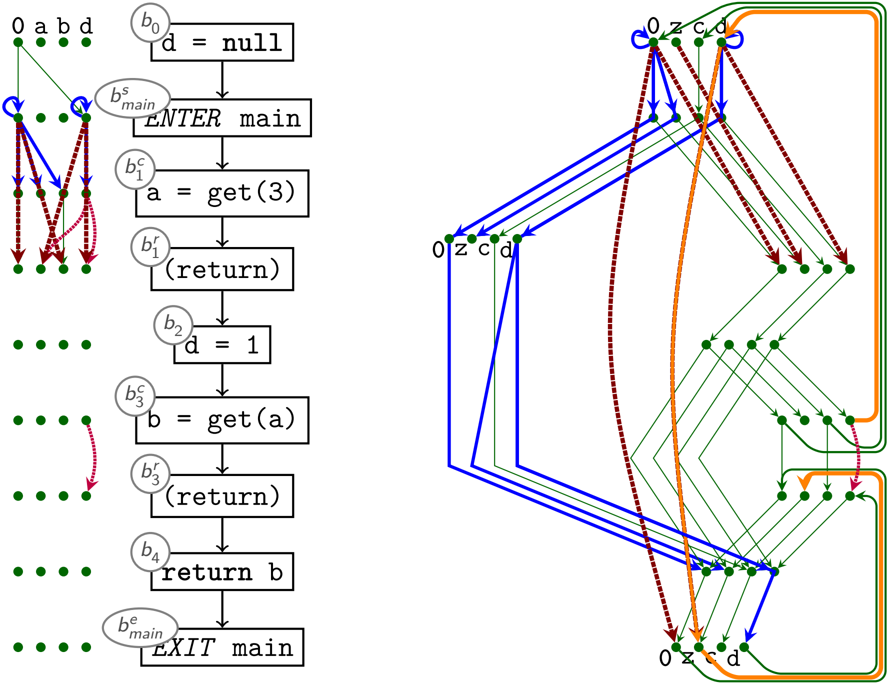

0 a b d
d = null
ENTER main
a = get(3)
(return)
d = 1
b = get(a)
(return)
return b
EXIT main
ENTER get
if
z = d
0 z c d
z = read()
if
z = get(c)
(return)
return z
EXIT get
0 z c d
0 z c d

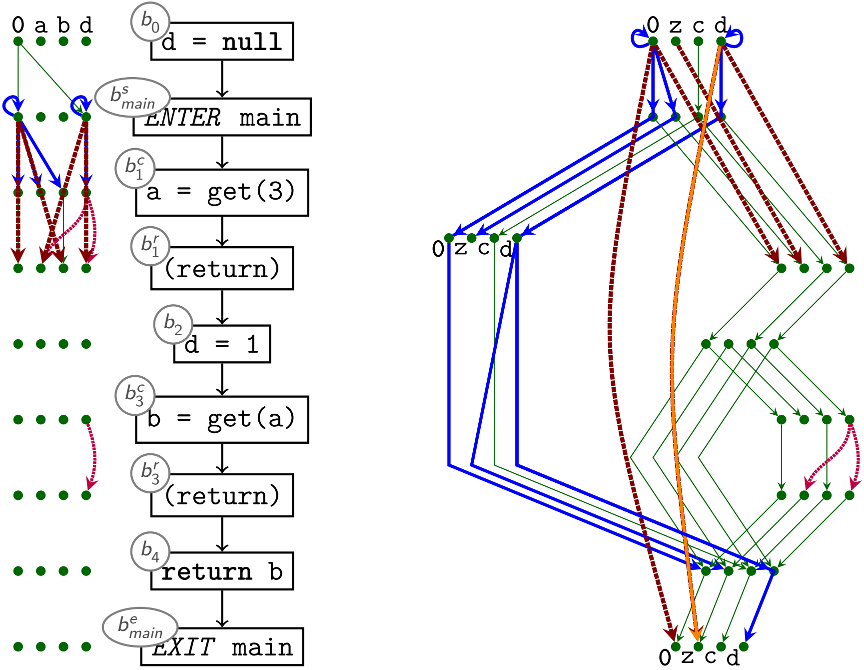

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d


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

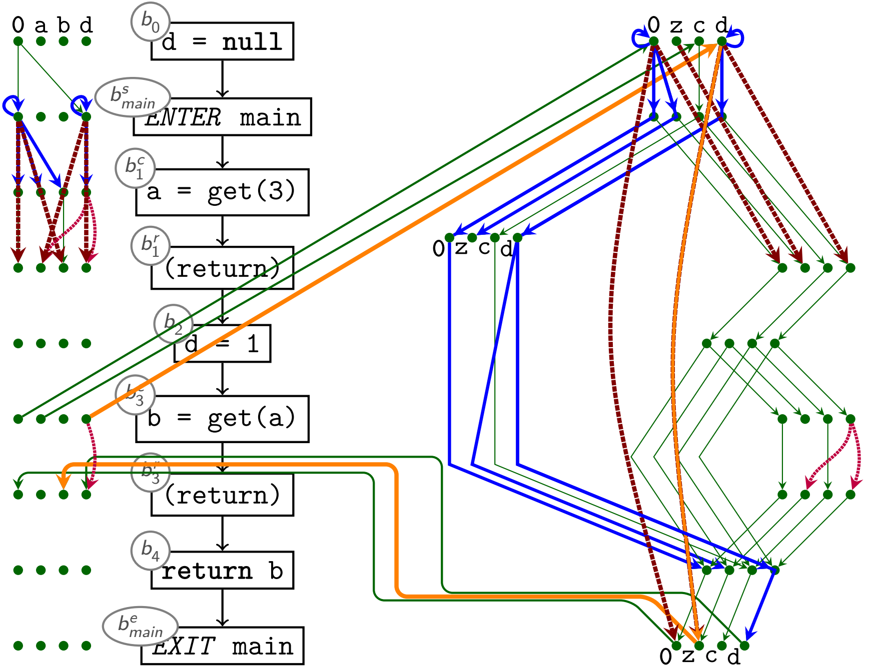


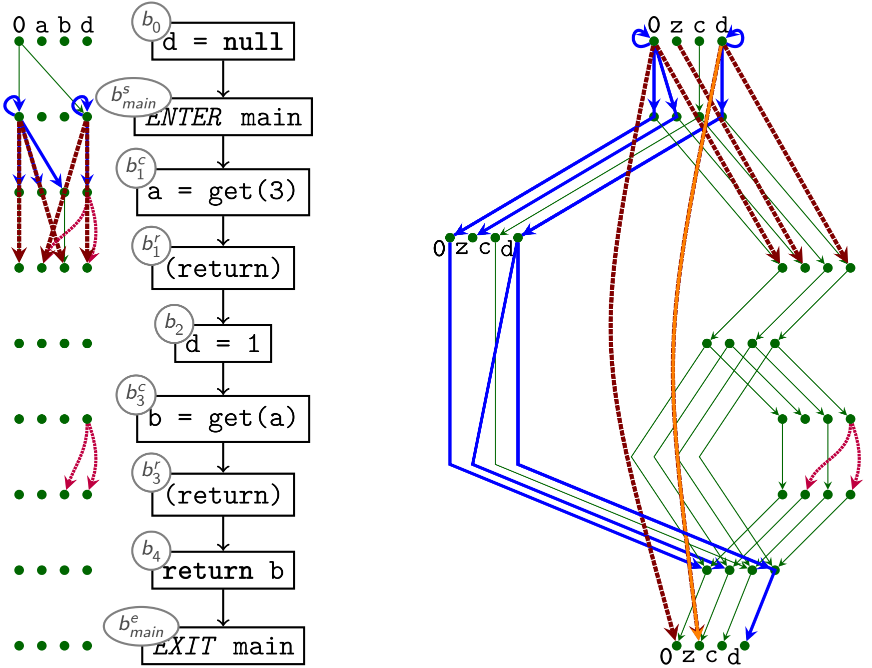
0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d


0 a b d
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

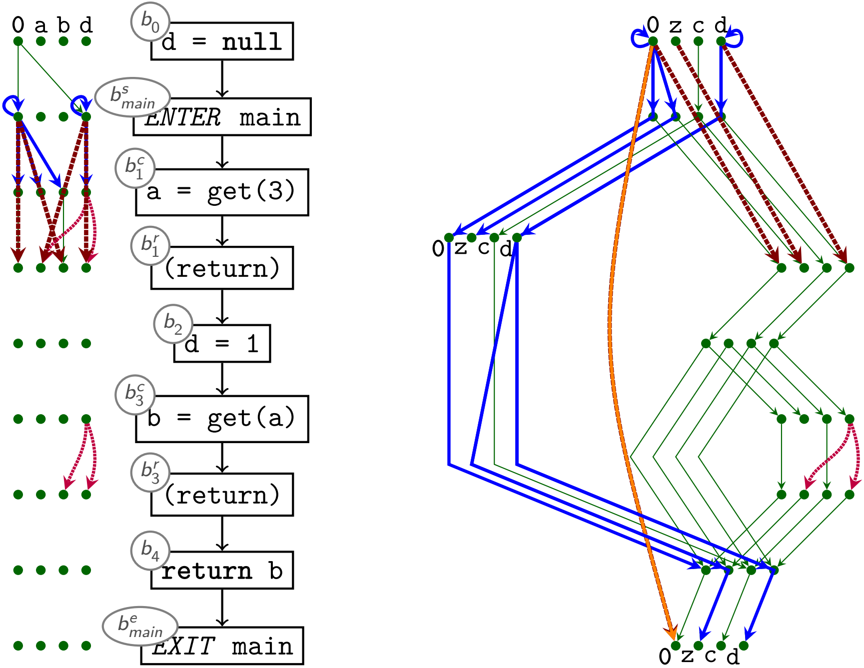


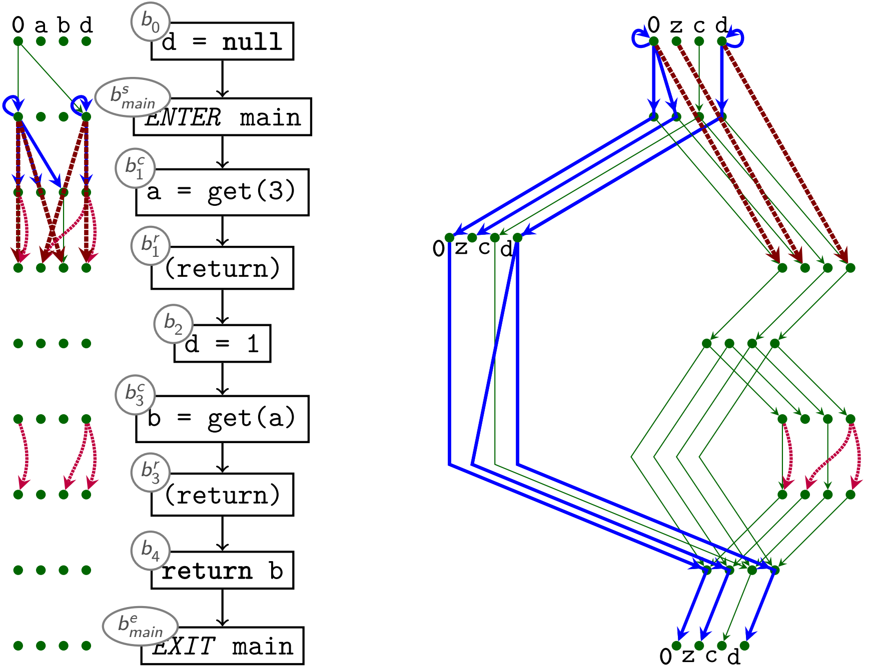
0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

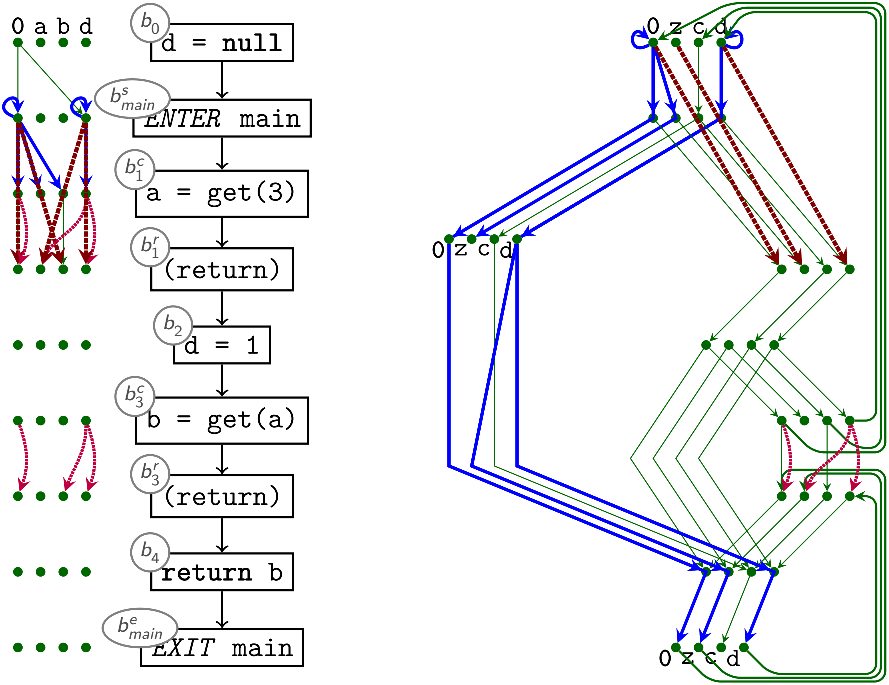

0 a b d
b0
d = null
bs main
ENTER main
bc1
a = get(3)
br1
(return)
b2
d = 1
bc3
b = get(a)
br3
(return)
b4
return b
be main
EXIT main
0 z c d
0 z c d
0 z c d

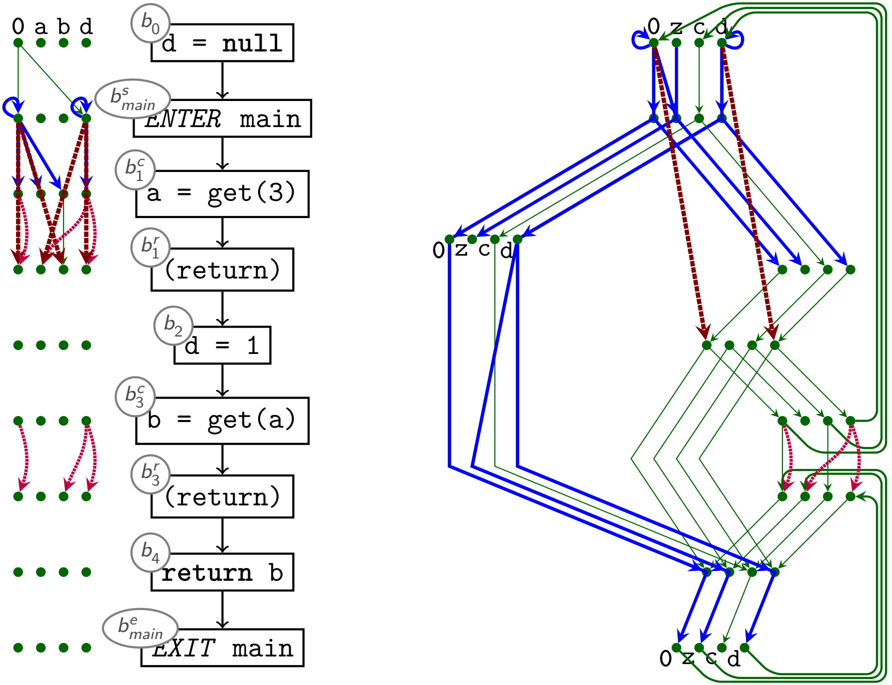


0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

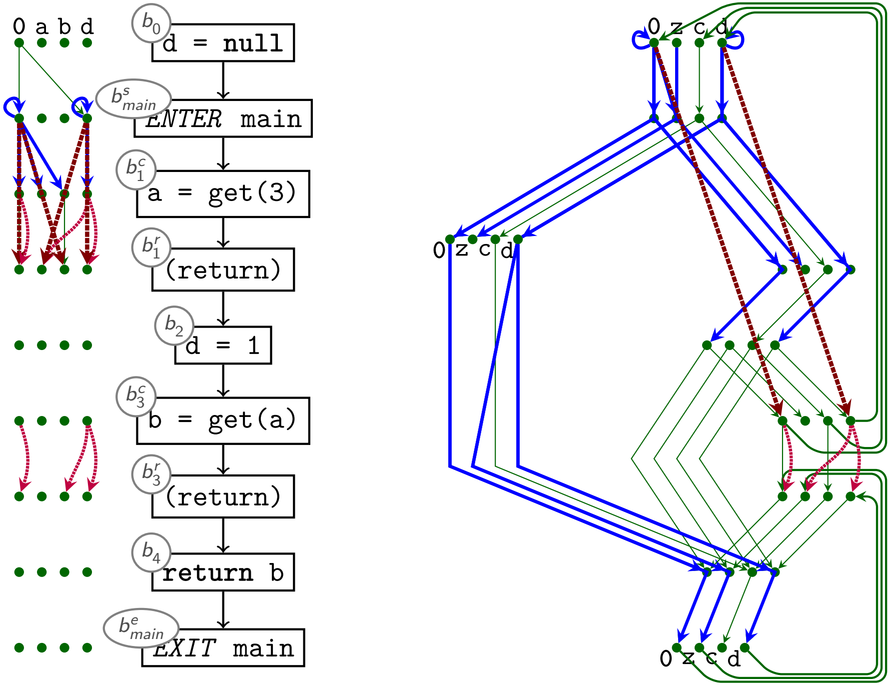

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

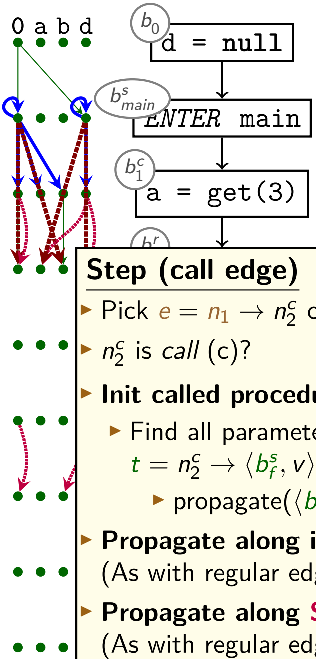

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d


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



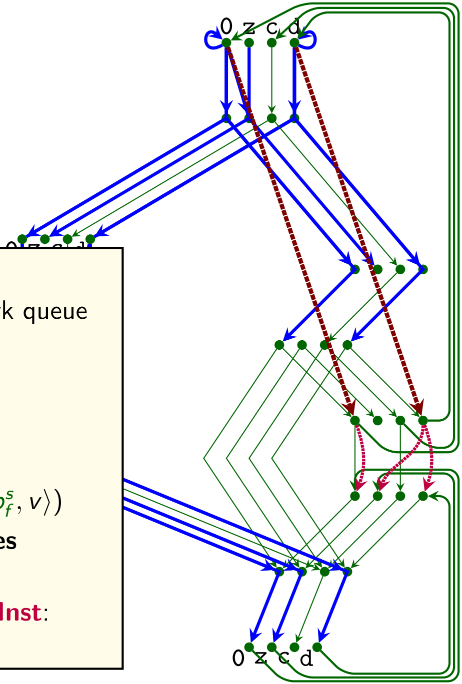
0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

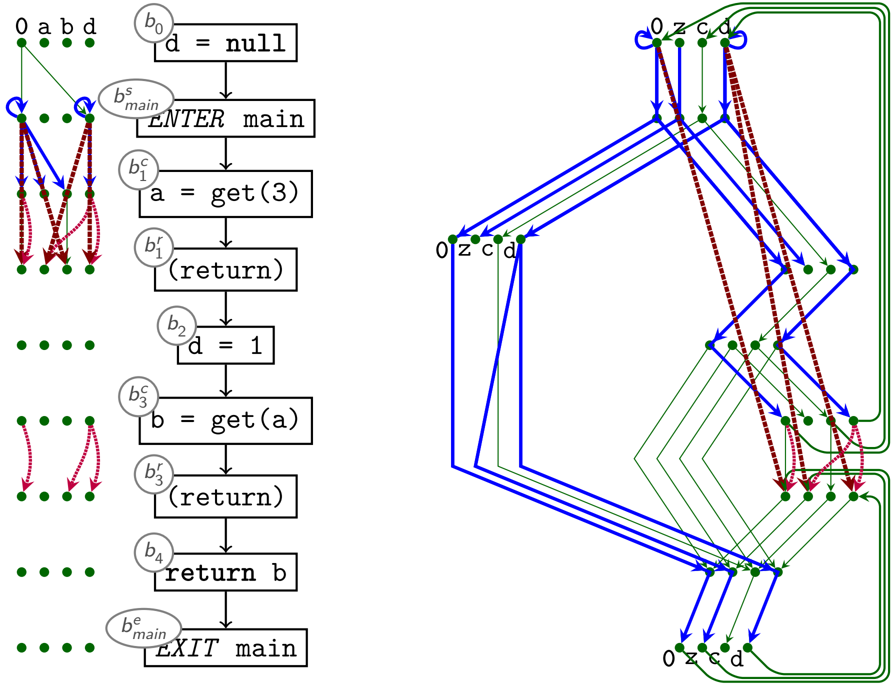

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

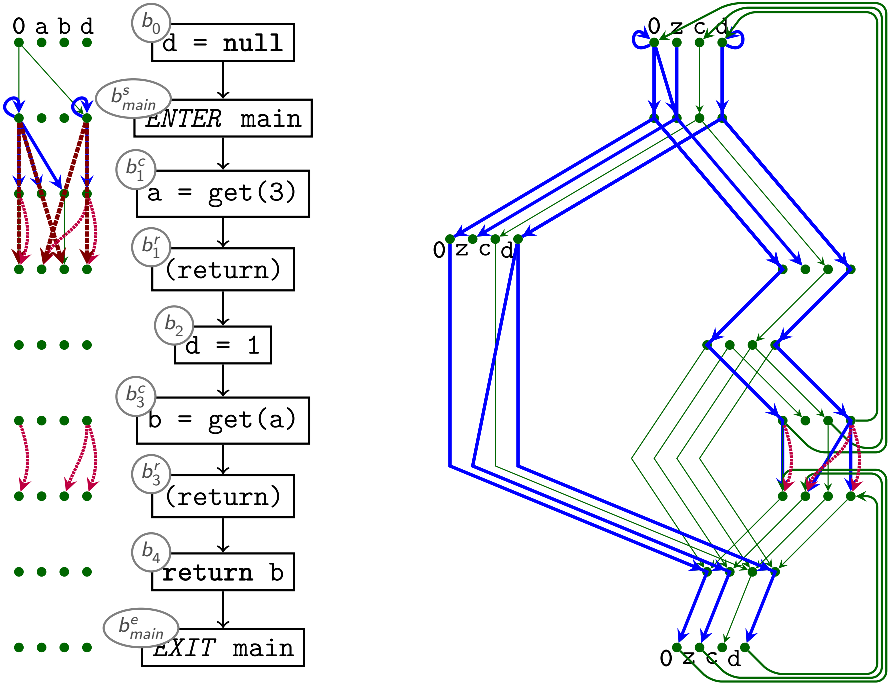


Step (call edge)

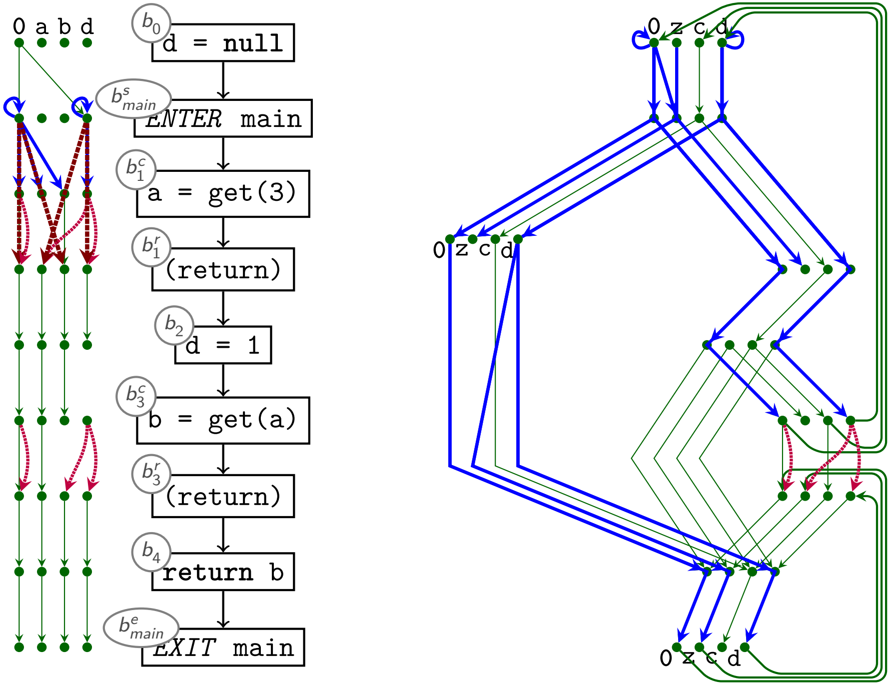
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v\rangle \in E^\sharp$
    - propagate$(\langle b_f^s, v\rangle \to \langle b_f^s, v\rangle)$
- **Propagate along intra-edges** (As with regular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

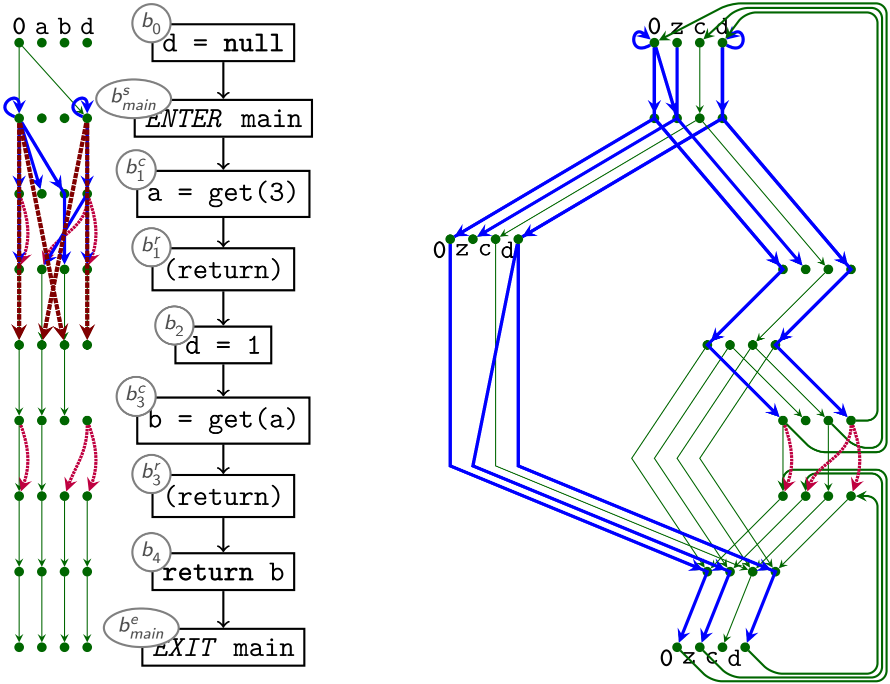


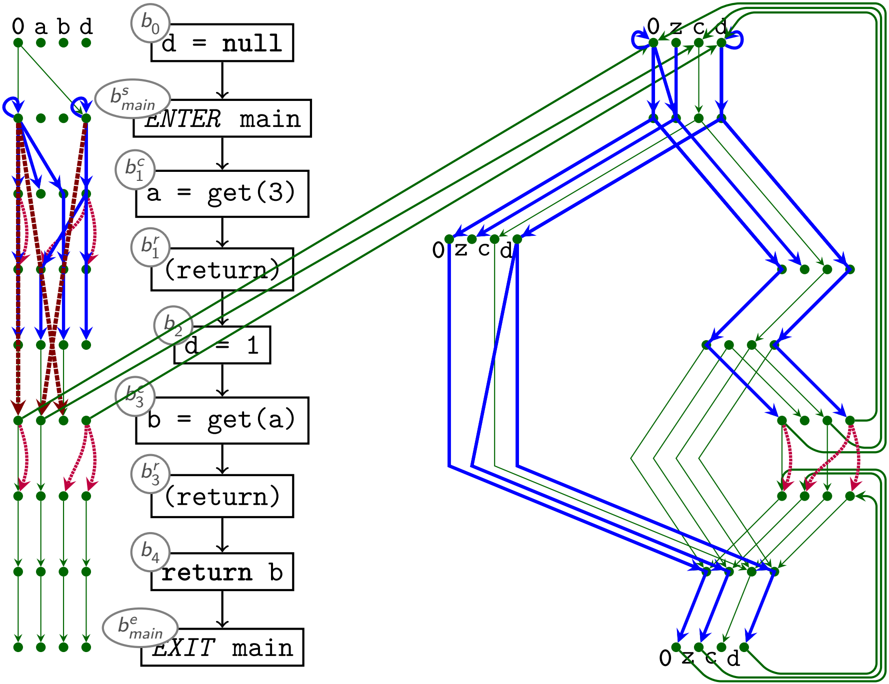

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

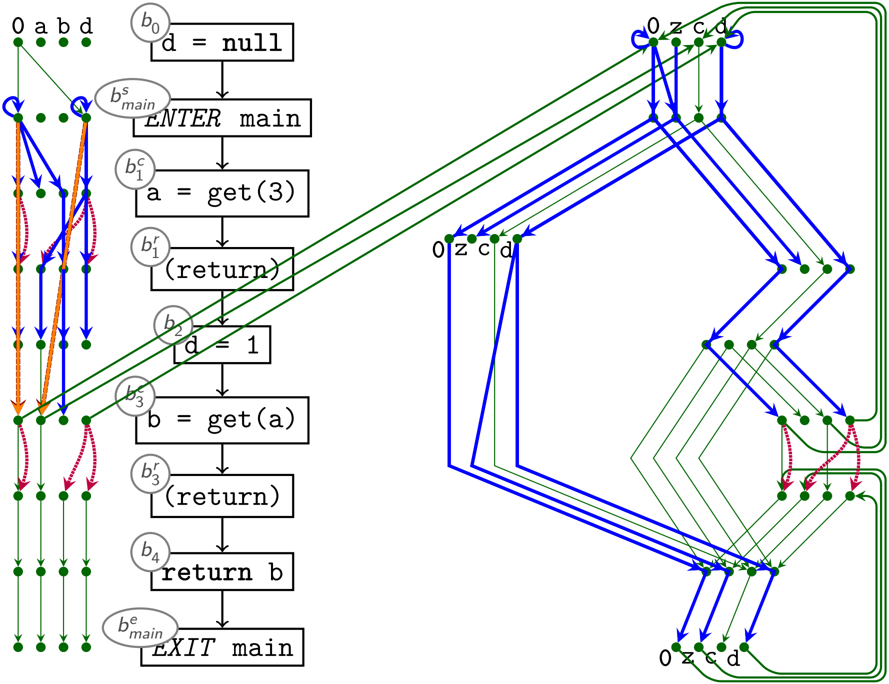


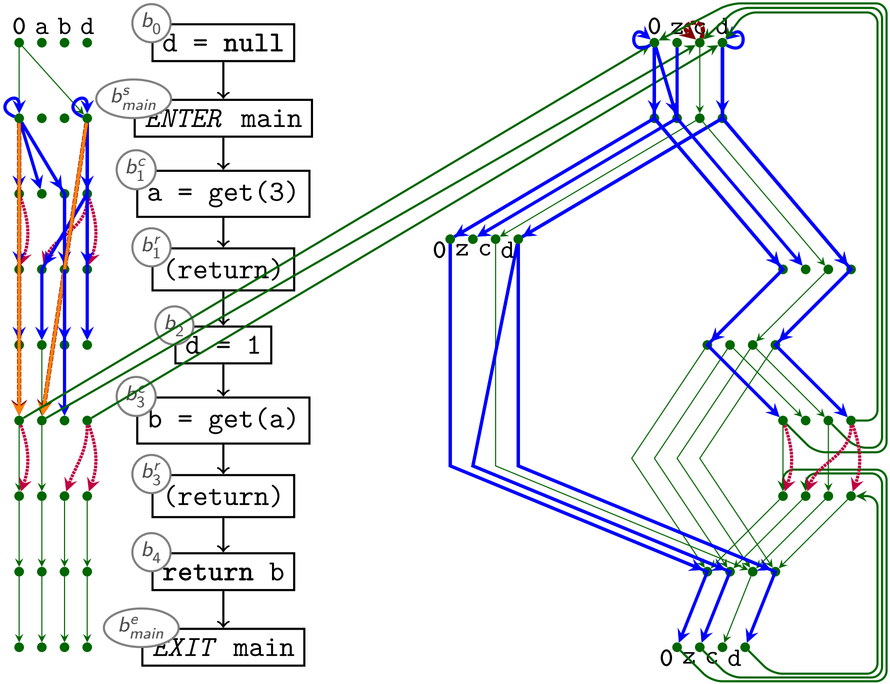
0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

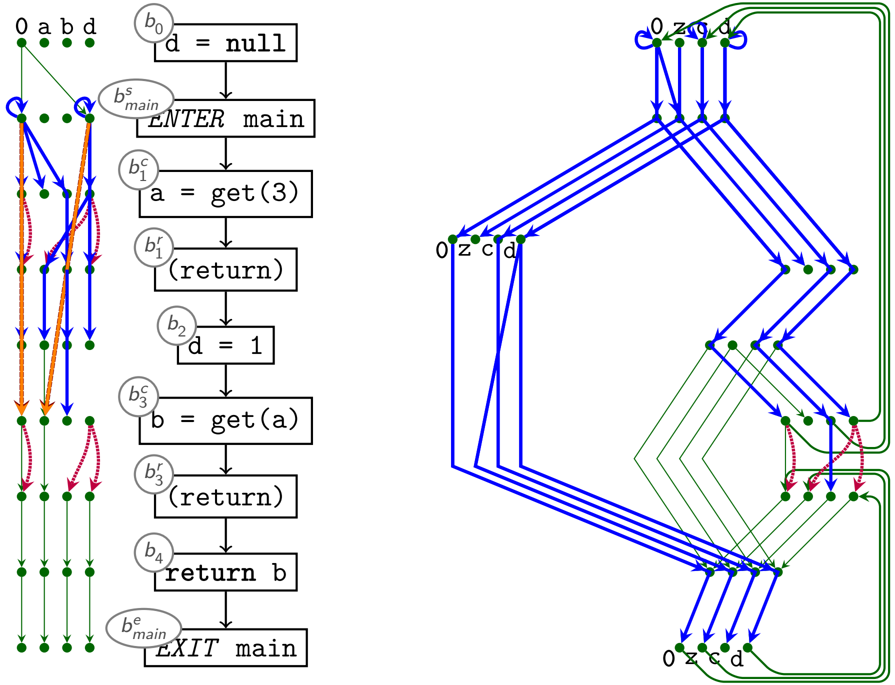

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

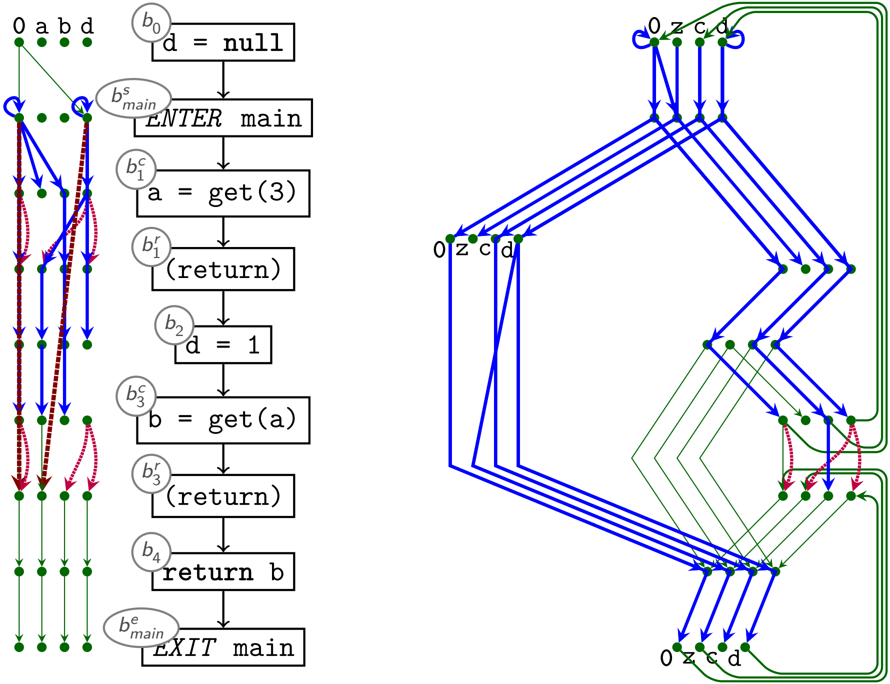

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

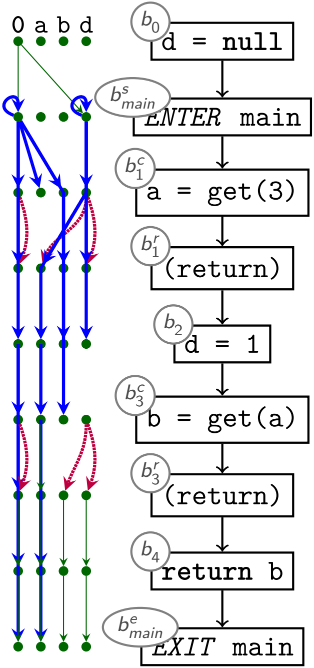

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

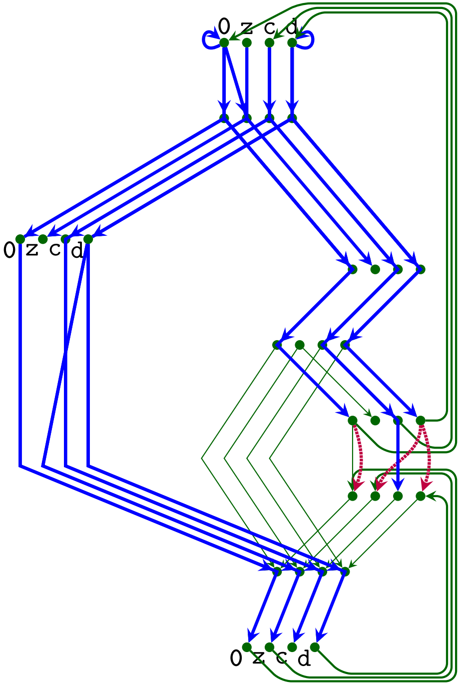

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

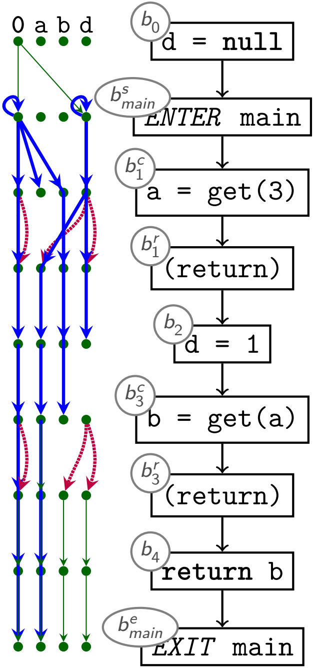

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

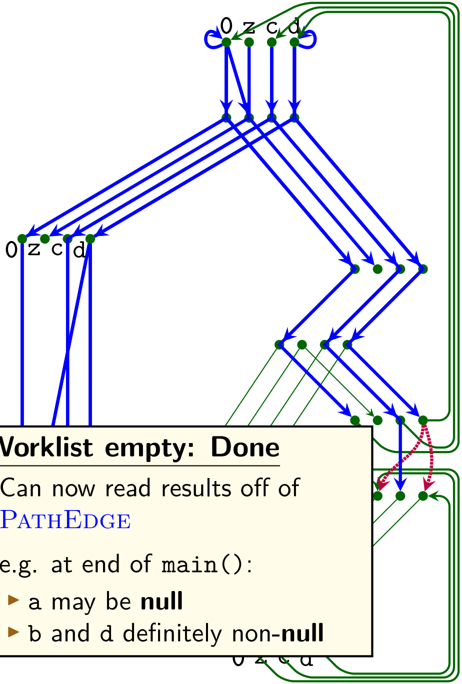

0 a b d
b_0
d = null
b^s_{main}
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b_2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d


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
b0
d = null
b^s_main
ENTER main
b^c_1
a = get(3)
b^r_1
(return)
b2
d = 1
b^c_3
b = get(a)
b^r_3
(return)
b4
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

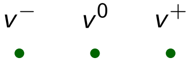
Summary: IFDS Algorithm

- Computes yes-or-no 'May' 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 Max. Fixpoint on distributive frameworks**

Beyond True and False

- What if abstract domain is not boolean?
  - e.g., $\{\top, A^+, A^-, A^0, \bot\}$

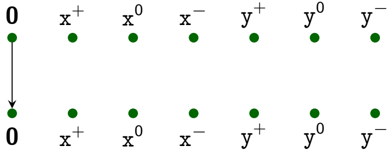
Beyond True and False



- What if abstract domain is not boolean?
  - e.g., $\{\top, A^+, A^-, A^0, \bot\}$
- Multiple boolean properties per variable
  - easy for powerset lattice $\mathcal{P}(\{+, -, 0\})$
- *Limitation*: Transfer functions only depend on one variable
- Some problems not representable, others must adapt lattice

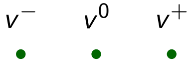
  Consider $b_1 =$ `y = 0 - x`:

$trans_{b_1} =$



This is how the algorithm was originally proposed

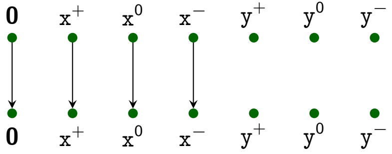
Beyond True and False



- What if abstract domain is not boolean?
  - e.g., $\{\top, A^+, A^-, A^0, \bot\}$
- Multiple boolean properties per variable
  - easy for powerset lattice $\mathcal{P}(\{+, -, 0\})$
- *Limitation*: Transfer functions only depend on one variable
- Some problems not representable, others must adapt lattice

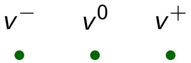
Consider $b_1 = \boxed{\texttt{y = 0 - x}}$:

$trans_{b_1} =$



This is how the algorithm was originally proposed

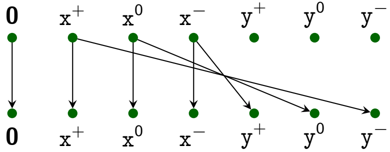
Beyond True and False



- What if abstract domain is not boolean?
  - e.g., $\{\top, A^+, A^-, A^0, \bot\}$
- Multiple boolean properties per variable
  - easy for powerset lattice $\mathcal{P}(\{+, -, 0\})$
- *Limitation*: Transfer functions only depend on one variable
- Some problems not representable, others must adapt lattice

  Consider $b_1 =$ `y = 0 - x`:

$trans_{b_1} =$



This is how the algorithm was originally proposed

Extending IFDS?

- Not all analyses map well to IFDS
- Core ideas are appealing:
  - Automatically compute procedure summaries
  - Exploit graph reachability + worklist for *dependency tracking*

Extending IFDS?

- Not all analyses map well to IFDS
- Core ideas are appealing:
  - Automatically compute procedure summaries
  - Exploit graph reachability + worklist for *dependency tracking*

It is possible to extend this to other classes of problems

Linear Reaching Values

Statement	$\mathbf{in}_b$	$\mathbf{out}_b$
`x = 42`	M	$\{[x \mapsto 42]\} \cup (M \setminus [x \mapsto \_])$
`x = y + 1`	$M = \{[y \mapsto c], \ldots\}$	$\{[x \mapsto c + 1]\} \cup (M \setminus [x \mapsto \_])$
`x = y * 7`	$M = \{[y \mapsto c], \ldots\}$	$\{[x \mapsto c \times 7]\} \cup (M \setminus [x \mapsto \_])$
`x = y + z`	M	$\{[x \mapsto \bot]\} \cup (M \setminus [x \mapsto \_])$

Linear Reaching Values

Statement	$\mathbf{in}_b$	$\mathbf{out}_b$
`x = 42`	M	$\{[x \mapsto 42]\} \cup (M \setminus [x \mapsto \_])$
`x = y + 1`	$M = \{[y \mapsto c], \ldots\}$	$\{[x \mapsto c + 1]\} \cup (M \setminus [x \mapsto \_])$
`x = y * 7`	$M = \{[y \mapsto c], \ldots\}$	$\{[x \mapsto c \times 7]\} \cup (M \setminus [x \mapsto \_])$
`x = y + z`	M	$\{[x \mapsto \bot]\} \cup (M \setminus [x \mapsto \_])$

- The above sketches a *distributive* reaching values analysis
  - Each annotation of form $v_1 \mapsto c_1 \times v_2 + c_2$
  - Tradeoff: no support for adding / multiplying / ... (multiple variables)
- Encode in IFDS?

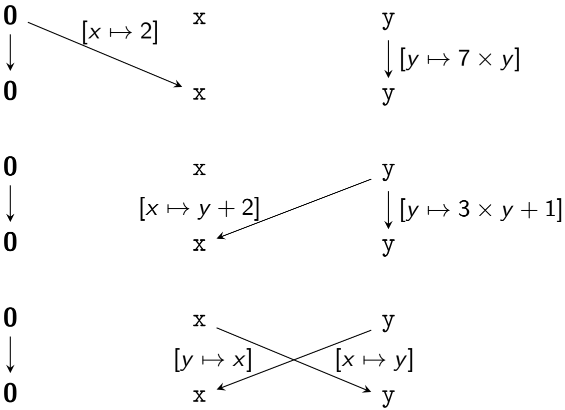
In the following, we consider *Linear Constant Propagation*, which is the **Must** analysis version of *Reaching Definitions*.

Labelling Graph Edges

```
x = 2
y = y * 7
```

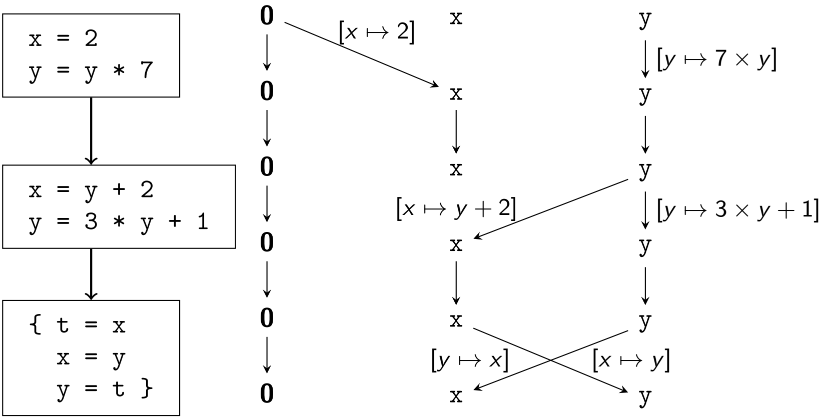
```
x = y + 2
y = 3 * y + 1
```

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



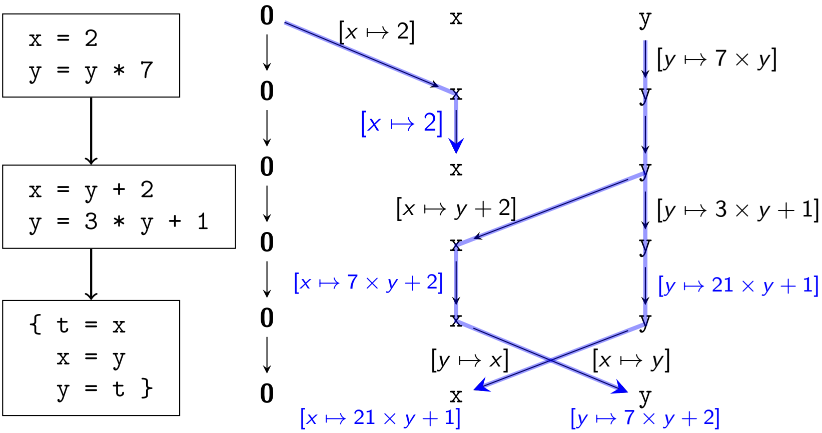
- Extending IFDS to support information processing
- Carrying over key techniques:
  - *Track dependencies*
  - *Generate procedure summaries on the fly*

Labelling Graph Edges



- Extending IFDS to support information processing
- Carrying over key techniques:
  - *Track dependencies*
  - *Generate procedure summaries on the fly*

Labelling Graph Edges



- Extending IFDS to support information processing
- Carrying over key techniques:
  - *Track dependencies*
  - *Generate procedure summaries on the fly*

Representation

$$\left\{\begin{array}{l}[x \mapsto c_{x,1} \times x + d_{x,1}] \\ [y \mapsto c_{y,1} \times y + d_{y,1}]\end{array}\right\} \circ \left\{\begin{array}{l}[x \mapsto c_{x,2} \times v_1 + d_{x,2}] \\ [y \mapsto c_{y,2} \times v_2 + d_{y,2}]\end{array}\right\}$$

$$=$$

$$\left\{\begin{array}{l}[x \mapsto (c_{x,2} \times c_{x,1}) \times v_1 + (d_{x,2} + c_{x_1} \times d_{x_1})] \\ [y \mapsto (c_{y,2} \times c_{y,1}) \times v_1 + (d_{y,2} + c_{y_1} \times d_{y_1})]\end{array}\right\}$$

- c_i, d_i: constants
- v_i: program variables

Representation

$$\left\{\begin{array}{l}[x \mapsto c_{x,1} \times x + d_{x,1}] \\ [y \mapsto c_{y,1} \times y + d_{y,1}]\end{array}\right\} \circ \left\{\begin{array}{l}[x \mapsto c_{x,2} \times v_1 + d_{x,2}] \\ [y \mapsto c_{y,2} \times v_2 + d_{y,2}]\end{array}\right\}$$
$$=$$
$$\left\{\begin{array}{l}[x \mapsto (c_{x,2} \times c_{x,1}) \times v_1 + (d_{x,2} + c_{x_1} \times d_{x_1})] \\ [y \mapsto (c_{y,2} \times c_{y,1}) \times v_1 + (d_{y,2} + c_{y_1} \times d_{y_1})]\end{array}\right\}$$

- c_i, d_i: constants
- v_i: program variables
- (Maps of) linear functions are closed under composition

Representation

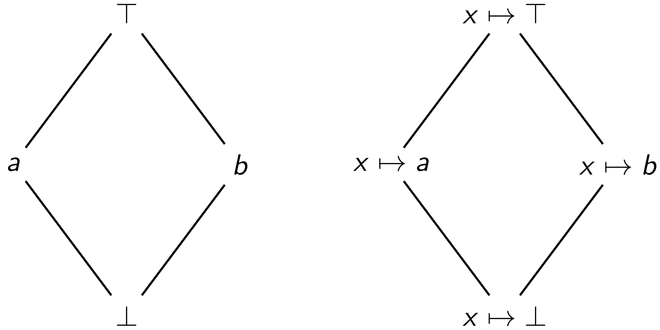
$$\left\{\begin{array}{l}[x \mapsto c_{x,1} \times x + d_{x,1}] \\ [y \mapsto c_{y,1} \times y + d_{y,1}]\end{array}\right\} \circ \left\{\begin{array}{l}[x \mapsto c_{x,2} \times v_1 + d_{x,2}] \\ [y \mapsto c_{y,2} \times v_2 + d_{y,2}]\end{array}\right\}$$
$$=$$
$$\left\{\begin{array}{l}[x \mapsto (c_{x,2} \times c_{x,1}) \times v_1 + (d_{x,2} + c_{x_1} \times d_{x_1})] \\ [y \mapsto (c_{y,2} \times c_{y,1}) \times v_1 + (d_{y,2} + c_{y_1} \times d_{y_1})]\end{array}\right\}$$

- c_i, d_i: constants
- v_i: program variables
- (Maps of) linear functions are closed under composition
- Must support $\sqcap$ to merge, map to $\perp$ on mismatch

$$\left\{\begin{array}{l}[x \mapsto c_{x,1} \times v_1 + d_{x,1}] \\ [y \mapsto c_{y,1} \times v_3 + d_{y,1}]\end{array}\right\} \sqcap \left\{\begin{array}{l}[x \mapsto c_{x,1} \times v_1 + d_{x,1}] \\ [y \mapsto c_{y,2} \times v_2 + d_{y,2}]\end{array}\right\}$$
$$=$$
$$\left\{\begin{array}{l}[x \mapsto c_{x,1} \times x + d_{x,1}] \\ [y \mapsto \perp]\end{array}\right\}$$

Micro-Functions and Lattices

- Extend lattices to such 'Micro-Functions':



Micro-Functions, Efficient Representation

- Micro-Functions must support:

Encoding	
Computation	$f(x)$
Equality testing	$f = f'$
Composition	$f \circ f'$
Meet	$f \sqcap f'$

- Other examples:
  - IFDS problems
  - Value bounds analysis

Micro-Functions, Efficient Representation

- Micro-Functions must support:

Encoding		$O(1)$ space
Computation	$f(x)$	$O(1)$ time
Equality testing	$f = f'$	$O(1)$ time
Composition	$f \circ f'$	$O(1)$ time
Meet	$f \sqcap f'$	$O(1)$ time

- Micro-functions are efficiently representable if they satisfy space / time constraints
  - Required for the algorithm's time bounds
- Other examples:
  - IFDS problems
  - Value bounds analysis

The IDE Algorithm (1/1)

- **I**nterprocedural **D**istributive **E**nvironments algorithm
- Extends IFDS to ‘labelled’ edges as described above
- Assumes distributive framework over micro-functions
- Algorithmic changes:
  - First phase analogous to IFDS
  - Second phase applies computed functions to read out results
- Maintain/update mapping from path edges to micro-functions f:

$$\textsc{PathEdge} = \{\langle b_0, v_0\rangle \xrightarrow{f_0} \langle b_1, v_1\rangle, \ldots\}$$

- ‘Missing edges’ equivalent to $x \mapsto \top$
- Initialise:

$$\textsc{PathEdge} = \{\langle b_0, v_0\rangle \xrightarrow{v_1 \mapsto \top} \langle b_1, v_1\rangle, \ldots\}$$

- Always exactly one f per $\{\langle b_0, v_0\rangle \xrightarrow{f} \langle b_1, v_1\rangle\} \in \textsc{PathEdge}$

The IDE Algorithm (2/2)

Procedure propagate($n_1 \rightarrow n_2$): `-- IFDS version`
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

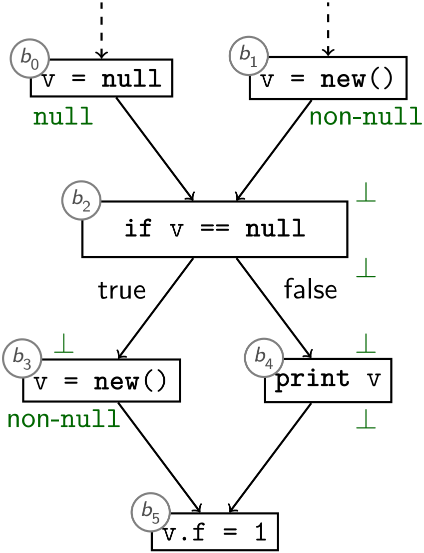
$$\Downarrow$$

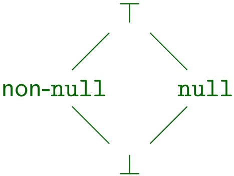
Procedure propagate$_{\mathsf{IDE}}$($n_1 \xrightarrow{f} n_2$): `-- IDE version`
begin
  let $n_1 \xrightarrow{f'} n_2 \in \textsc{PathEdge}$
  $f_{\mathsf{upd}} := f \sqcap f'$
  if $f_{\mathsf{upd}} = f'$ **then**
    return
  $\textsc{PathEdge} := (\textsc{PathEdge} \setminus \{n_1 \xrightarrow{f'} n_2\}) \cup \{n_1 \xrightarrow{f_{\mathsf{upd}}} n_2\}$
  $\textsc{WorkList} := \textsc{WorkList} \cup \{n_1 \rightarrow n_2\}$
end

Summary

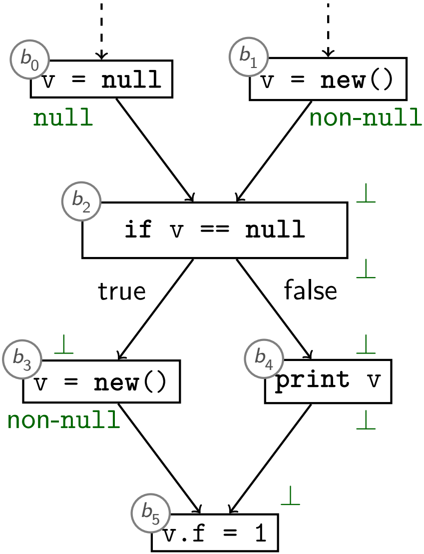
- IDE strictly generalises IFDS
- Utilises **Micro-Functions** to ensure efficient summaries:
  - Intra-procedural summaries via PATHEDGE
  - Inter-procedural procedure summaries via SUMMARYINST
- Runtime is $O(LED^3)$ if micro-functions are **efficiently representable**
  - L: Lattice height
    - IFDS: 1
    - IDE: length of longest descending chain
  - E: Number of control-flow edges
  - D: Number of variables
- IFDS supported by Soot, Phasar, WALA
- IDE supported by Soot, Phasar

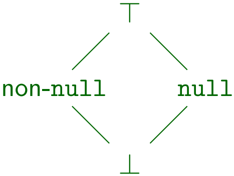
Path Sensitivity





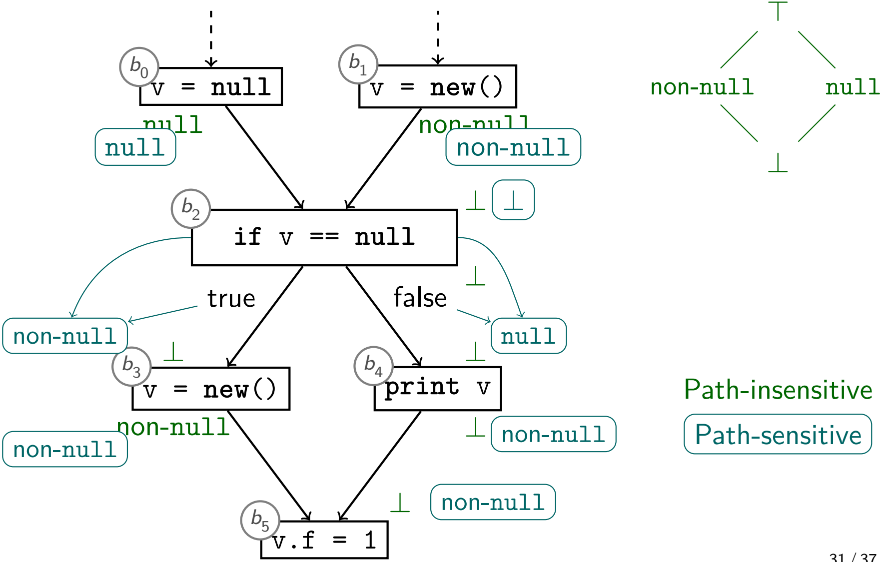
Path Sensitivity



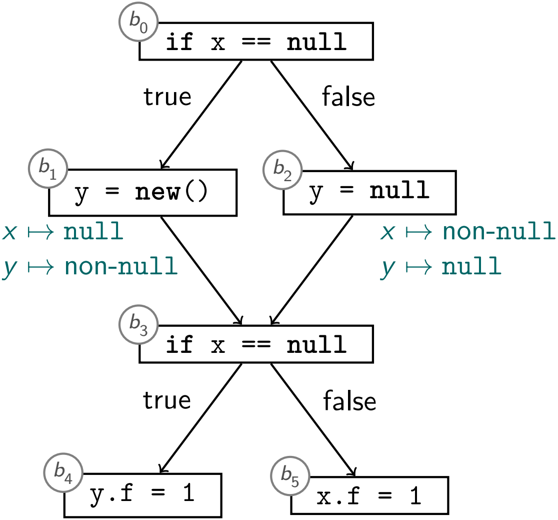


Path-insensitive

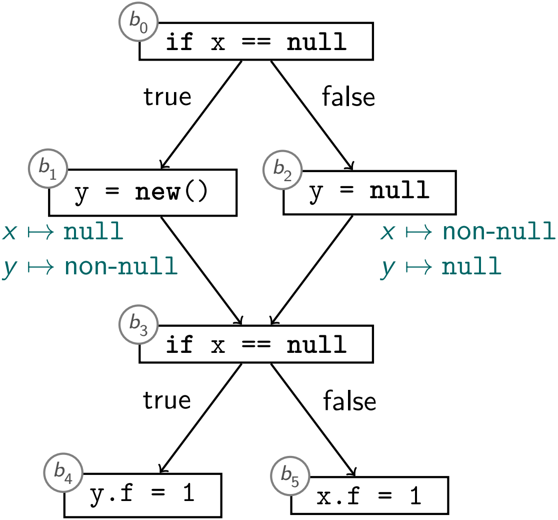
Path Sensitivity



Multiple Conditionals



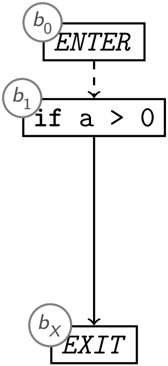
Multiple Conditionals



Should we carry path information across merge points?

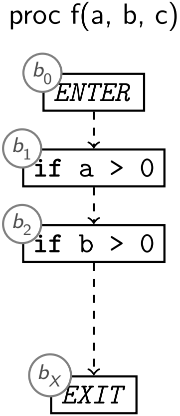
Paths

proc f(a, b, c)



2 paths

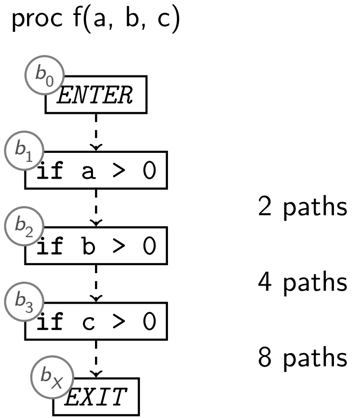
Paths



2 paths

4 paths

Paths



Number of paths grows exponentially

Summary

- **Path-sensitive** analysis considers conditionals:
  - May propagate different information along different paths
- Only for forward analyses
- Number of paths $O(\#\text{ of conditionals})$
  - Avoid exponential blow-up by merging (as before)
  - Path-sensitive procedure summaries might require exponential number of cases
- Exponential analyses/representations usually not practical

Homework 2

- IFDS analysis in Soot
- Flow Analysis on paper

Review

- Procedure Summaries
- IFDS algorithm
- IDE algorithm
- Path Sensitivity

To be continued...

Next week:

- Heap Analysis