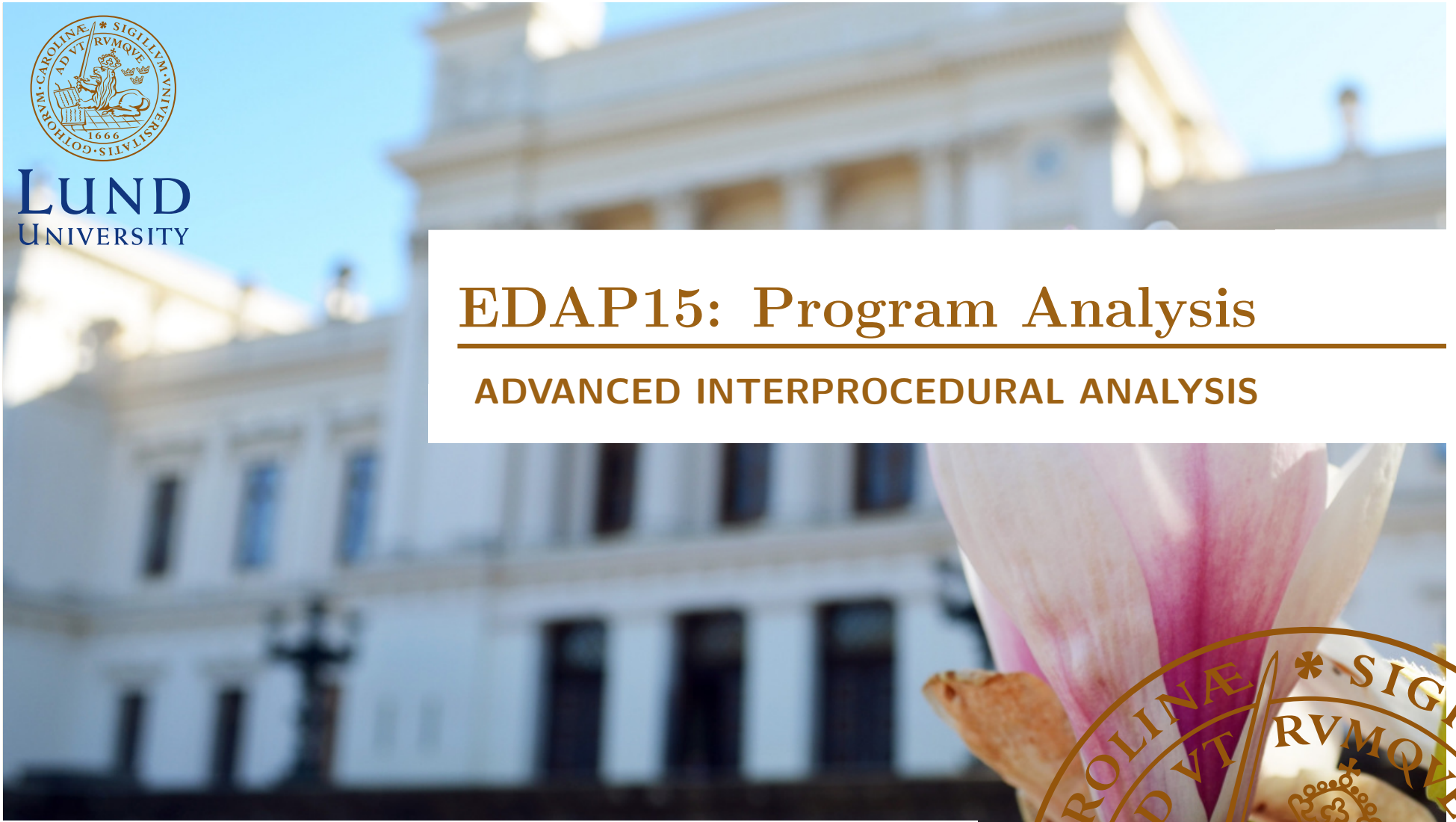


LUND UNIVERSITY

EDAP15: Program Analysis

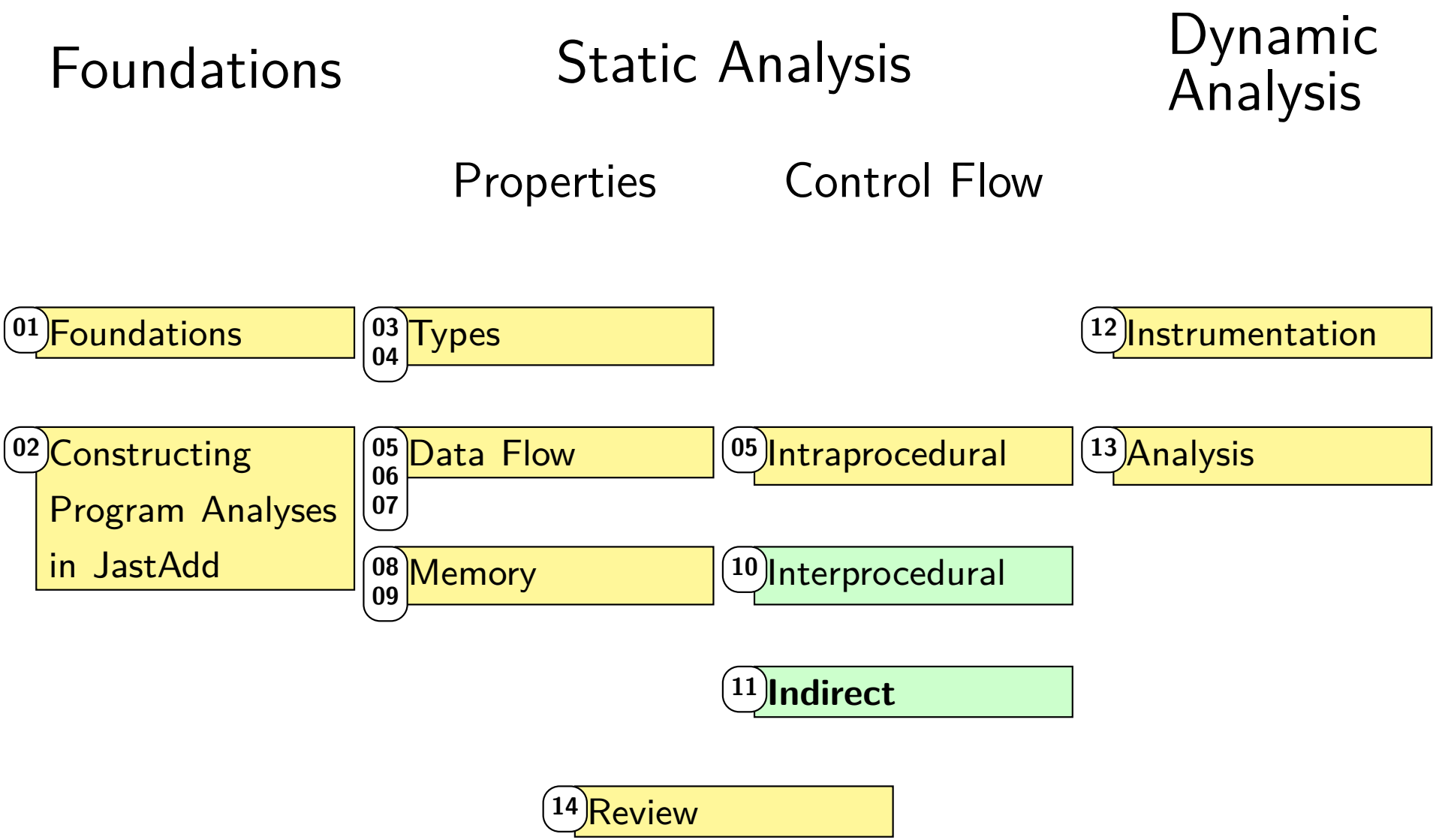
ADVANCED INTERPROCEDURAL ANALYSIS

Christoph Reichenbach

Welcome back!

- **Guest Lecture**: Second half of next Tuesday:
  - **Patrik Åberg** & **Magnus Templing**, Ericsson: Code Instrumentation with DMCE
  - Bring your own questions!

Lecture Overview



Procedure Summaries for Dataflow

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

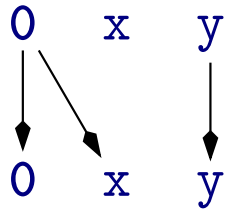
. . . *but only for Distributive Frameworks*

- Algorithm for reachability ("Gen/Kill") analyses: IFDS
- Algorithm for other analyses: IDE

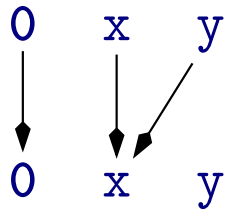
Representation Relations

Example procedure summary representation:

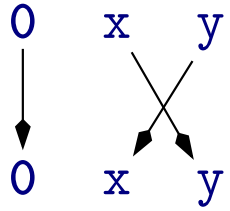
```
x := null;
y := y;
```




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




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



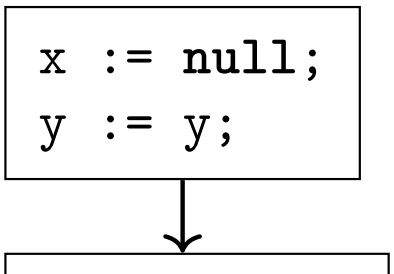
'May be null' analysis

- $c \longrightarrow d$:
  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 $\mathbf{out}_b$

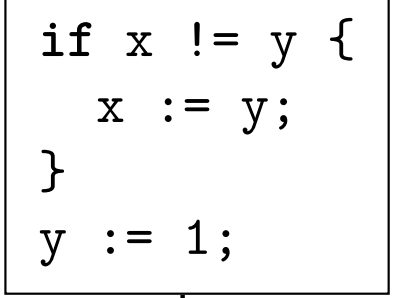
Composing Representation Relations

Representation Relations (*may be null* analysis):

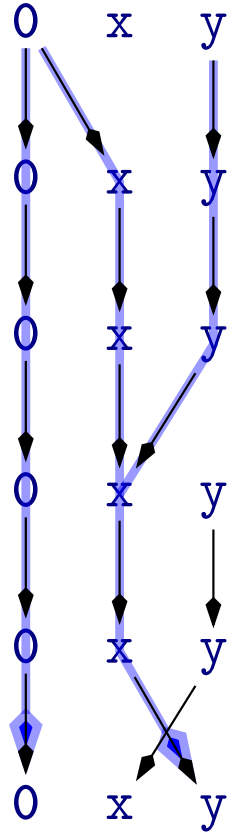
```
x := null;
y := y;
```

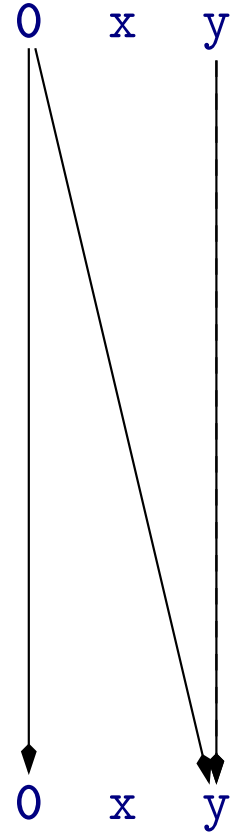


```
if x != y {
  x := y;
}
y := 1;
```



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

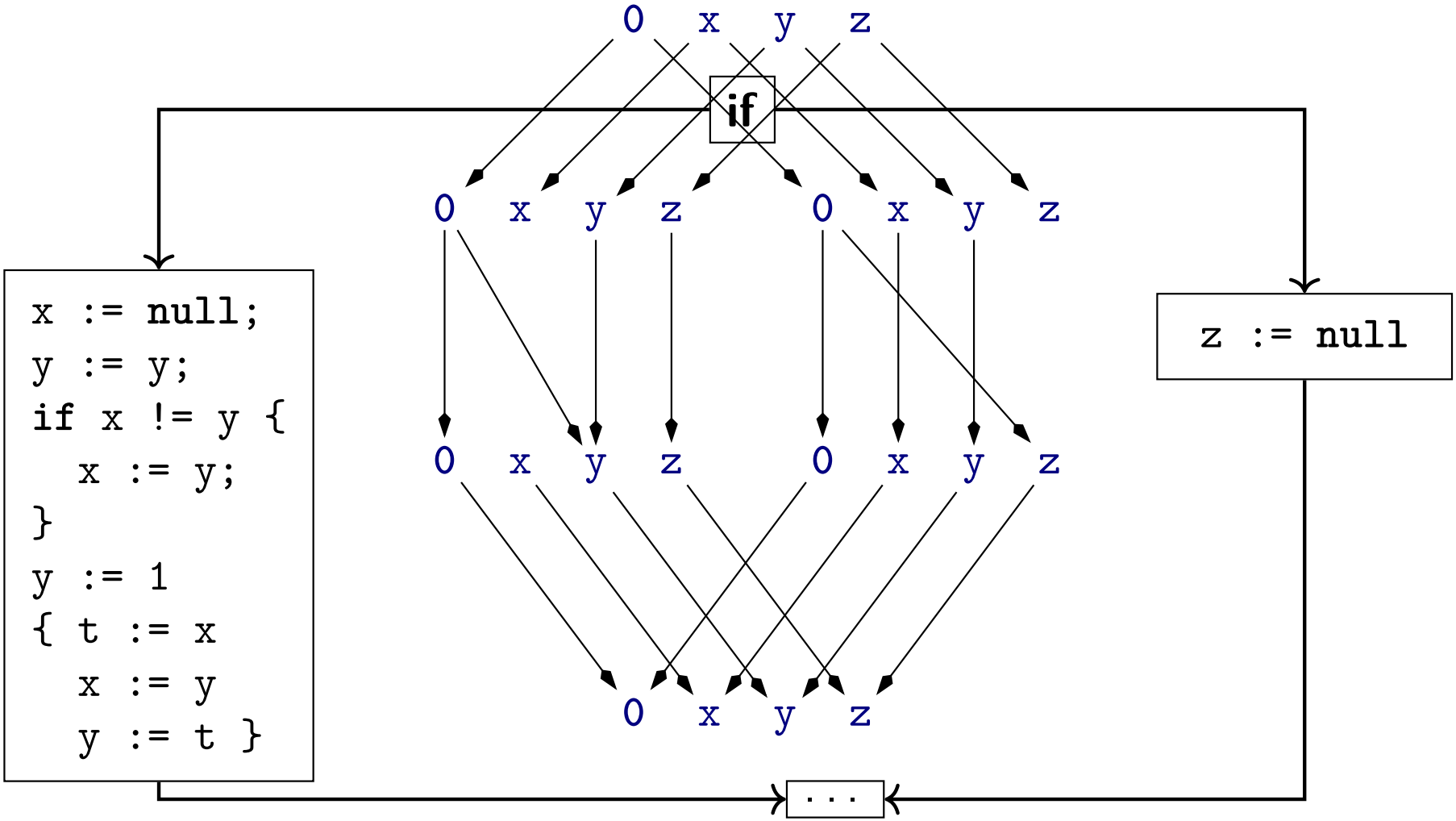




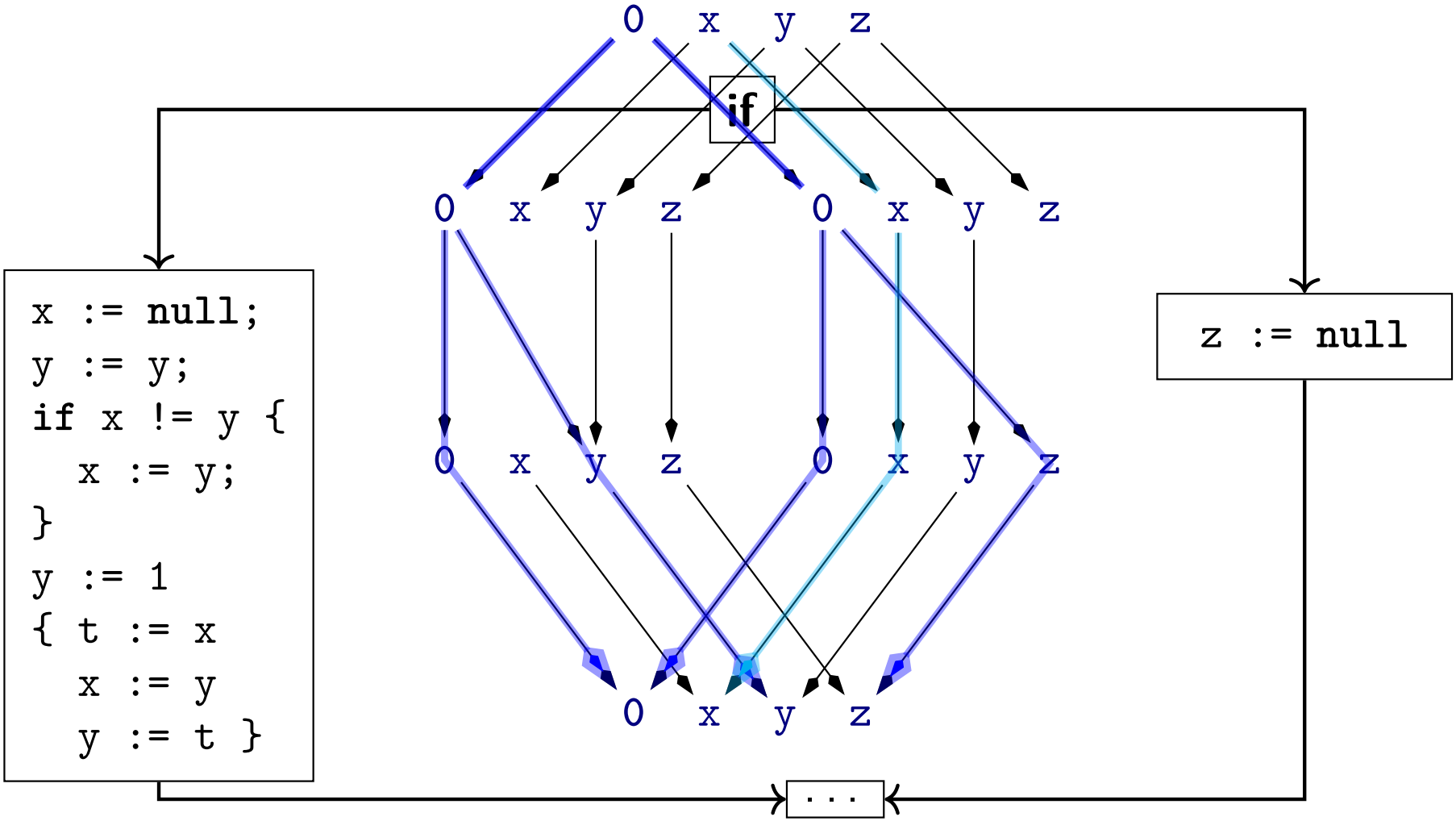
0 → y: always true, no need for $c \rightarrow y$

Composed representation relations are again representation relations

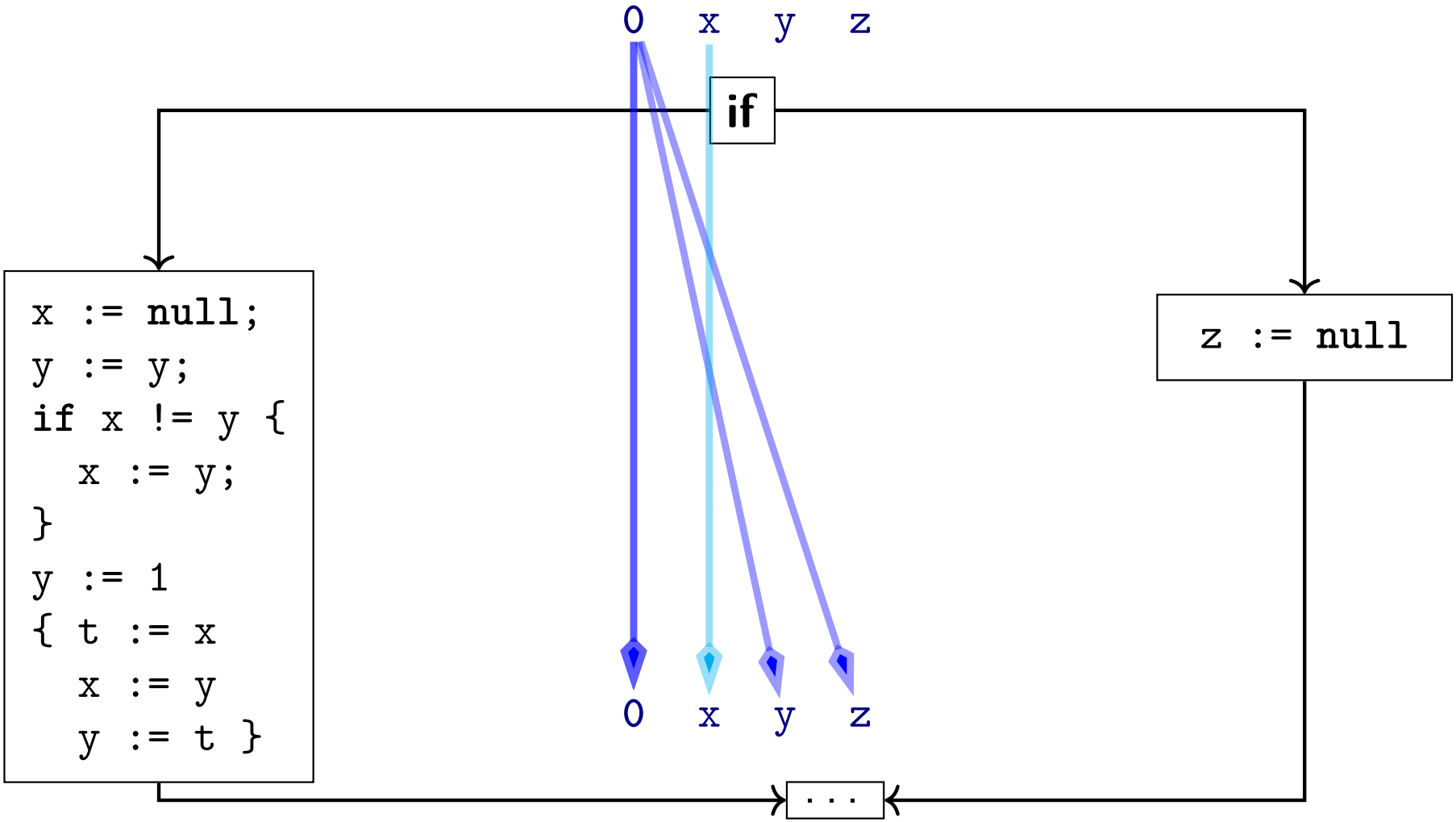
Joining Control-Flow Paths



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

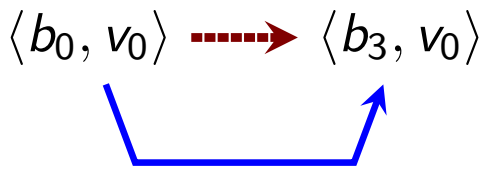
A Dataflow Worklist Algorithm: IFDS

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



PathEdge edge

All WorkList edges are also PathEdge edges

Result of our analysis

$N^\sharp$-edge

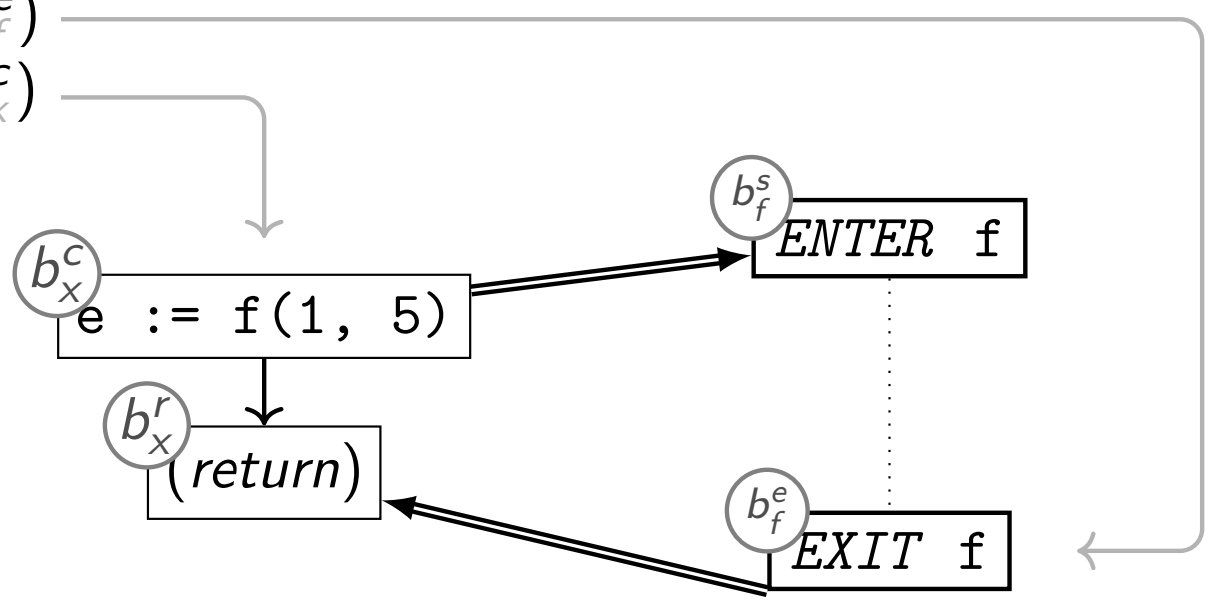
Transfer function as representational relation

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 $n_1 \to n_2 \in$ PathEdge **then**
    return
  PathEdge := PathEdge $\cup \{n_1 \to n_2\}$
  WorkList := WorkList $\cup \{n_1 \to n_2\}$
end

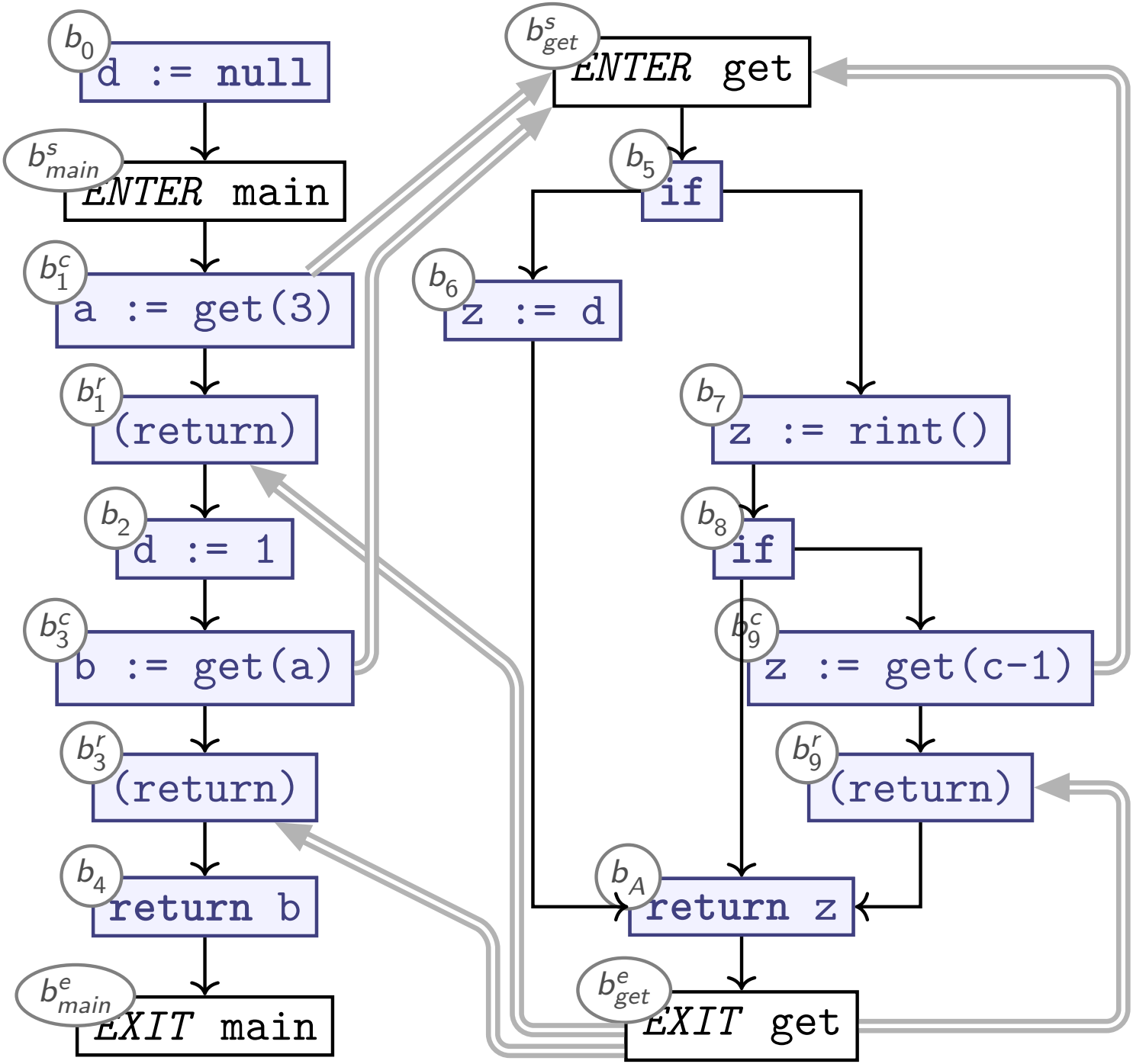
Running Example

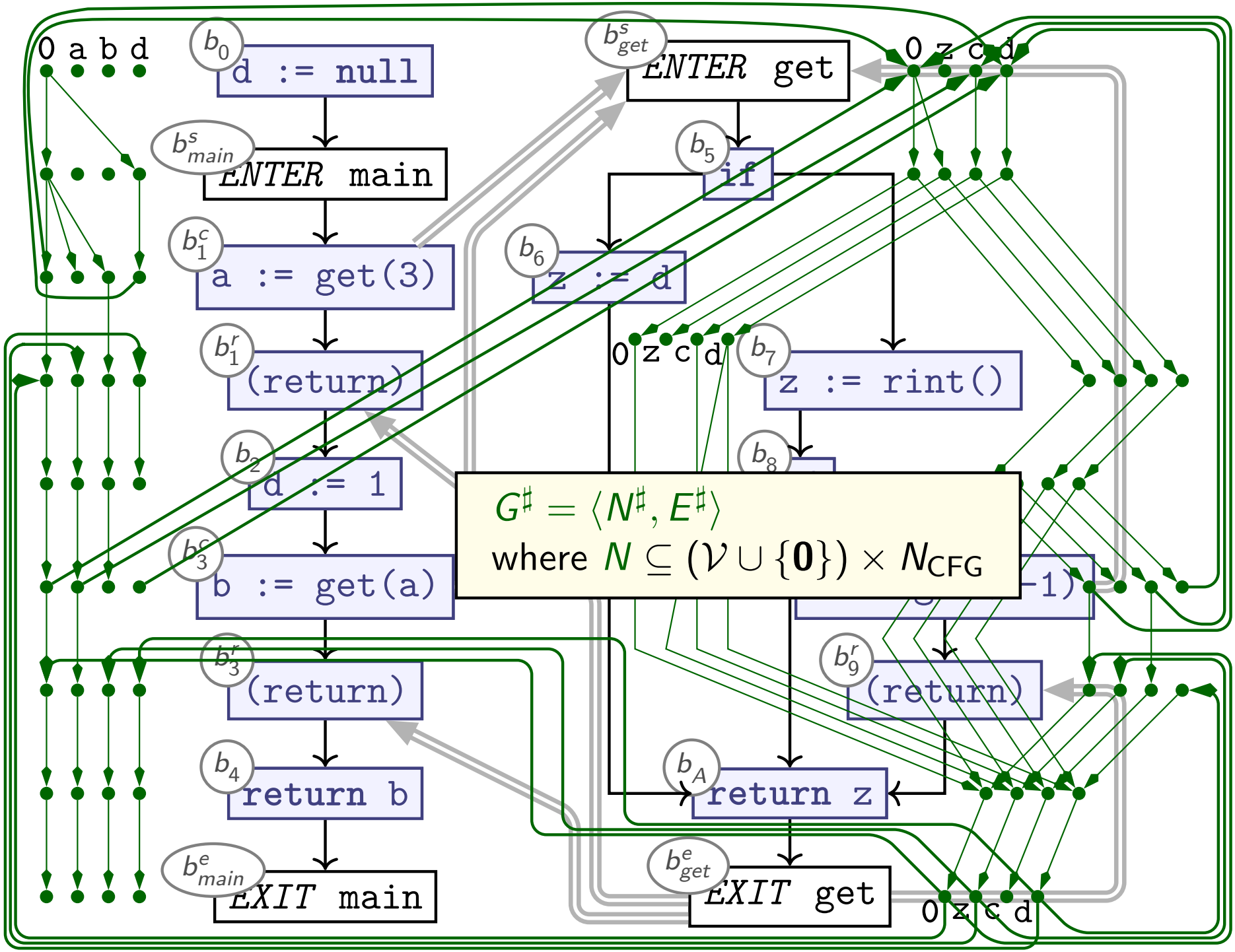
Teal-0: *main()*

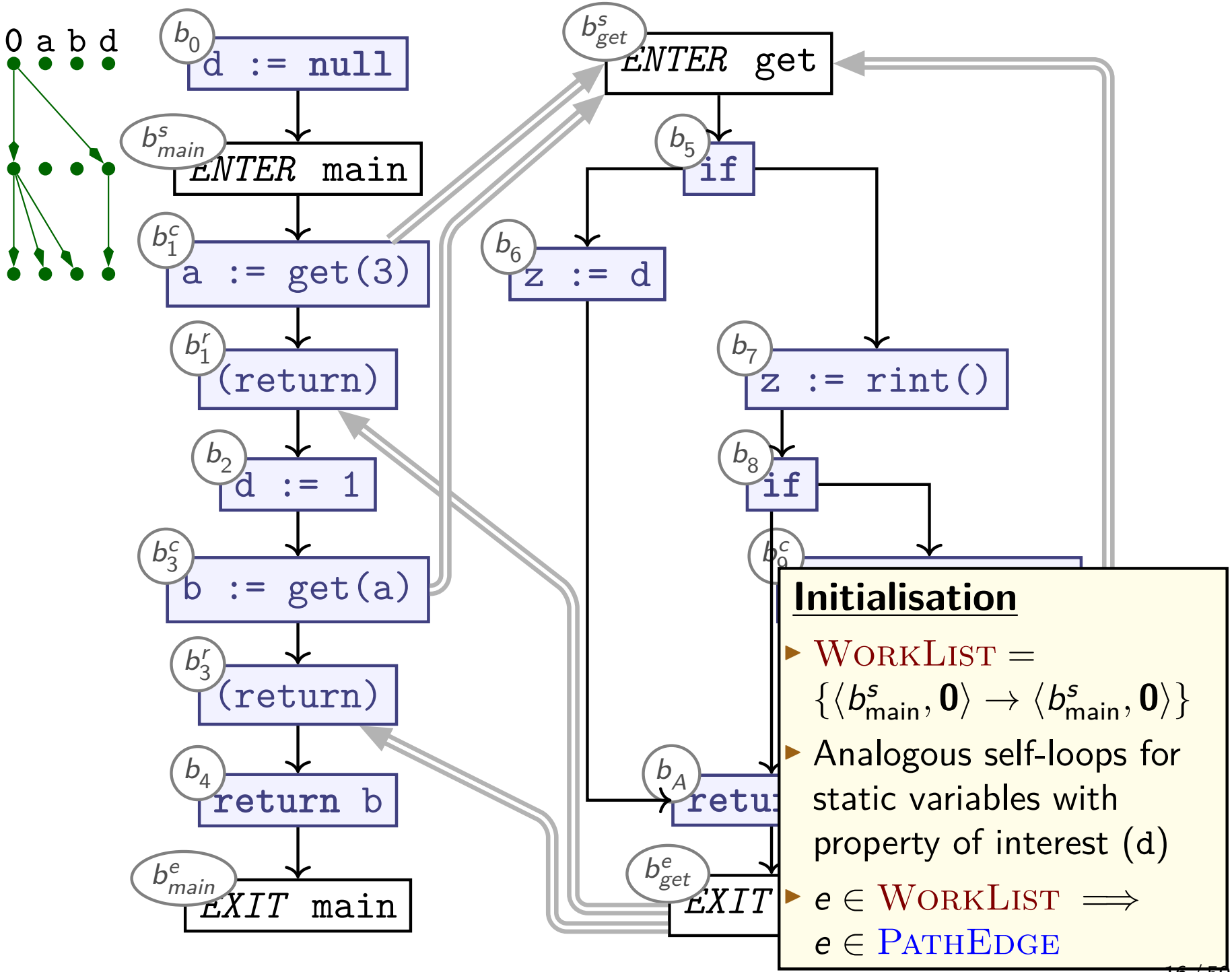
```
var default := null;
fun main() = {
  var a := get(3);
  default := 1;
  var b := get(a);
  return b;
}
```

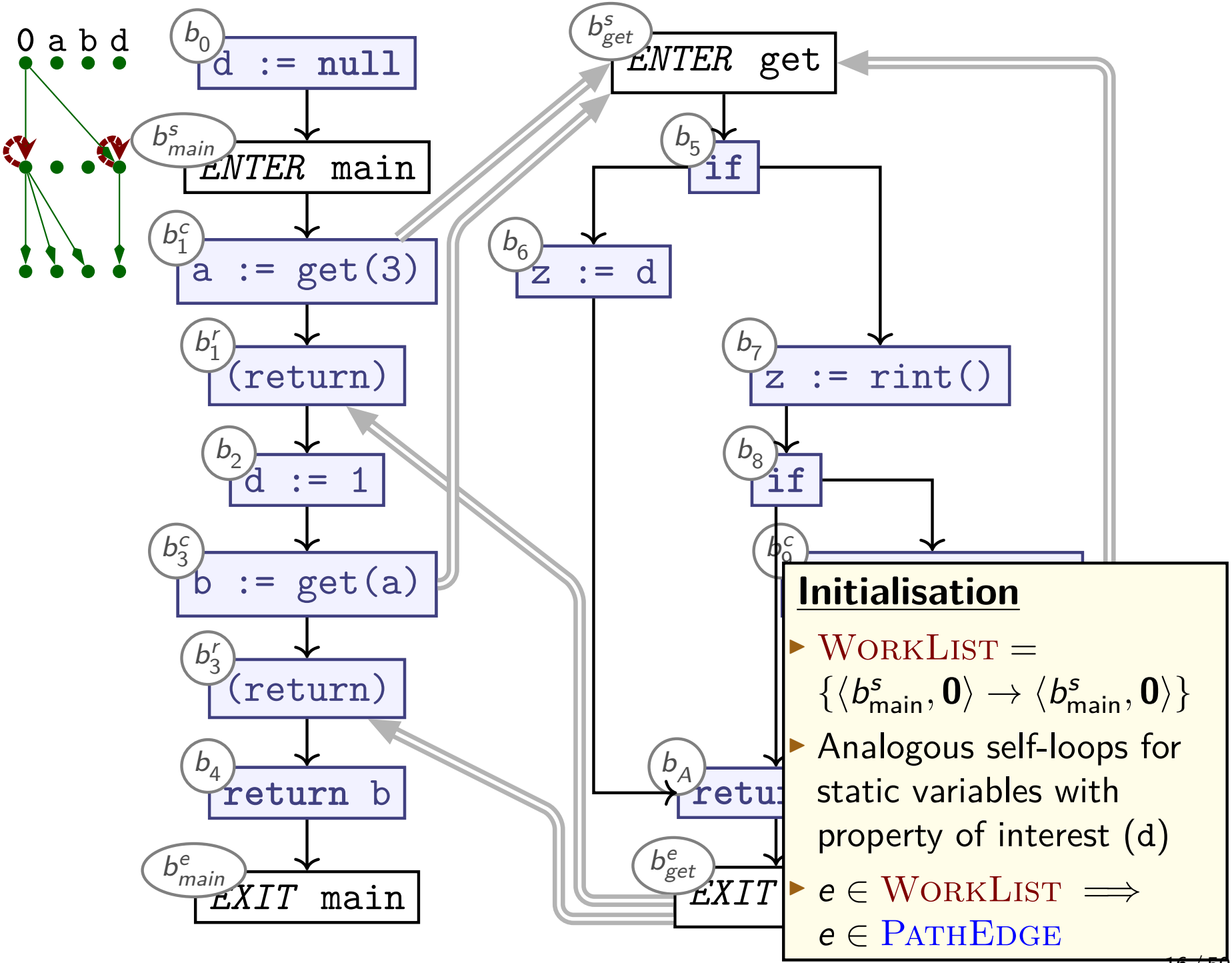
Teal-0: *get()*

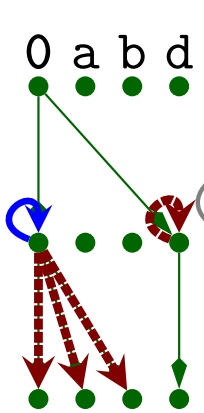
```
fun get(c) = {
  if c == 0 {
    z := default;
  } else {
    z := read_int();
    if z < 0 {
      z := get(c - 1);
    }
  }
  return z;
}
```

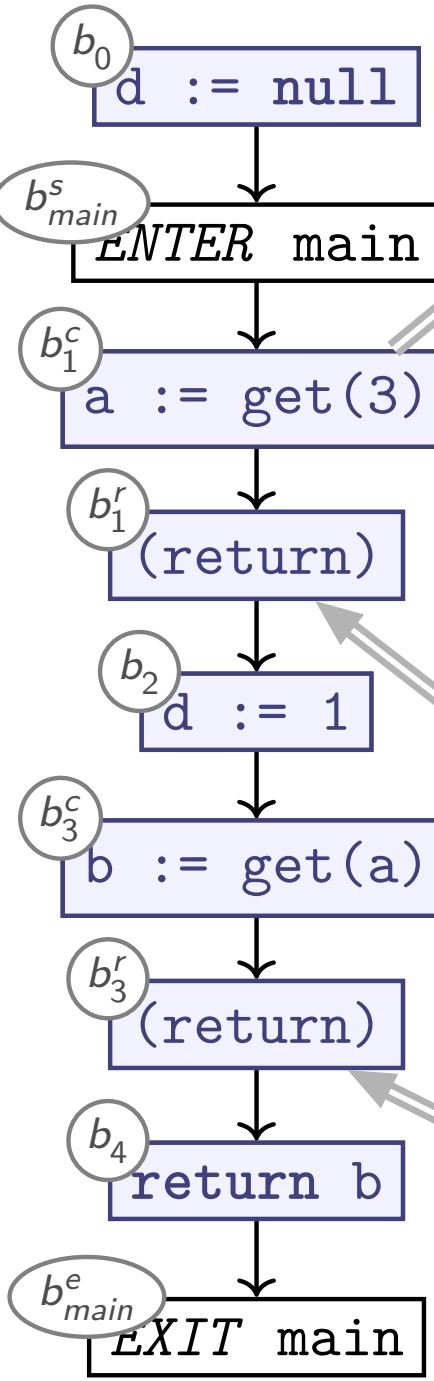

b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z := d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get

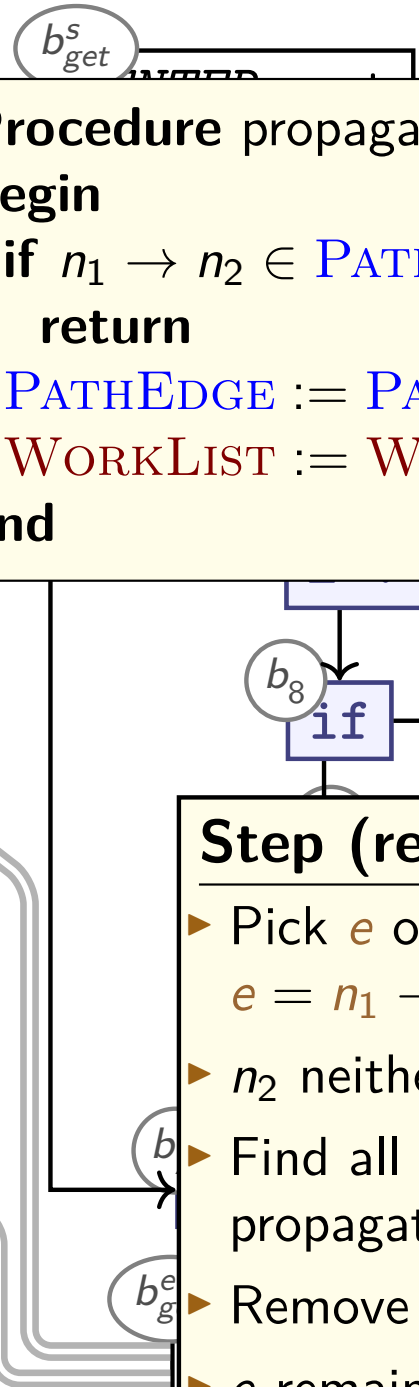

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
0 z c d
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
$G^\sharp = \langle N^\sharp, E^\sharp \rangle$
where $N \subseteq (\mathcal{V} \cup \{\mathbf{0}\}) \times N_{\mathsf{CFG}}$
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
b_5
if
b_6
z := d
b_7
z := rint()
b_8
if
b^c_9
b_A
retu
b^e_{get}
EXIT
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 $\Longrightarrow$ $e \in$ 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 := rint()
b_8
if
b^c_9
b_A
retu
b^e_{get}
EXIT
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 $\Longrightarrow$ $e \in$ PathEdge



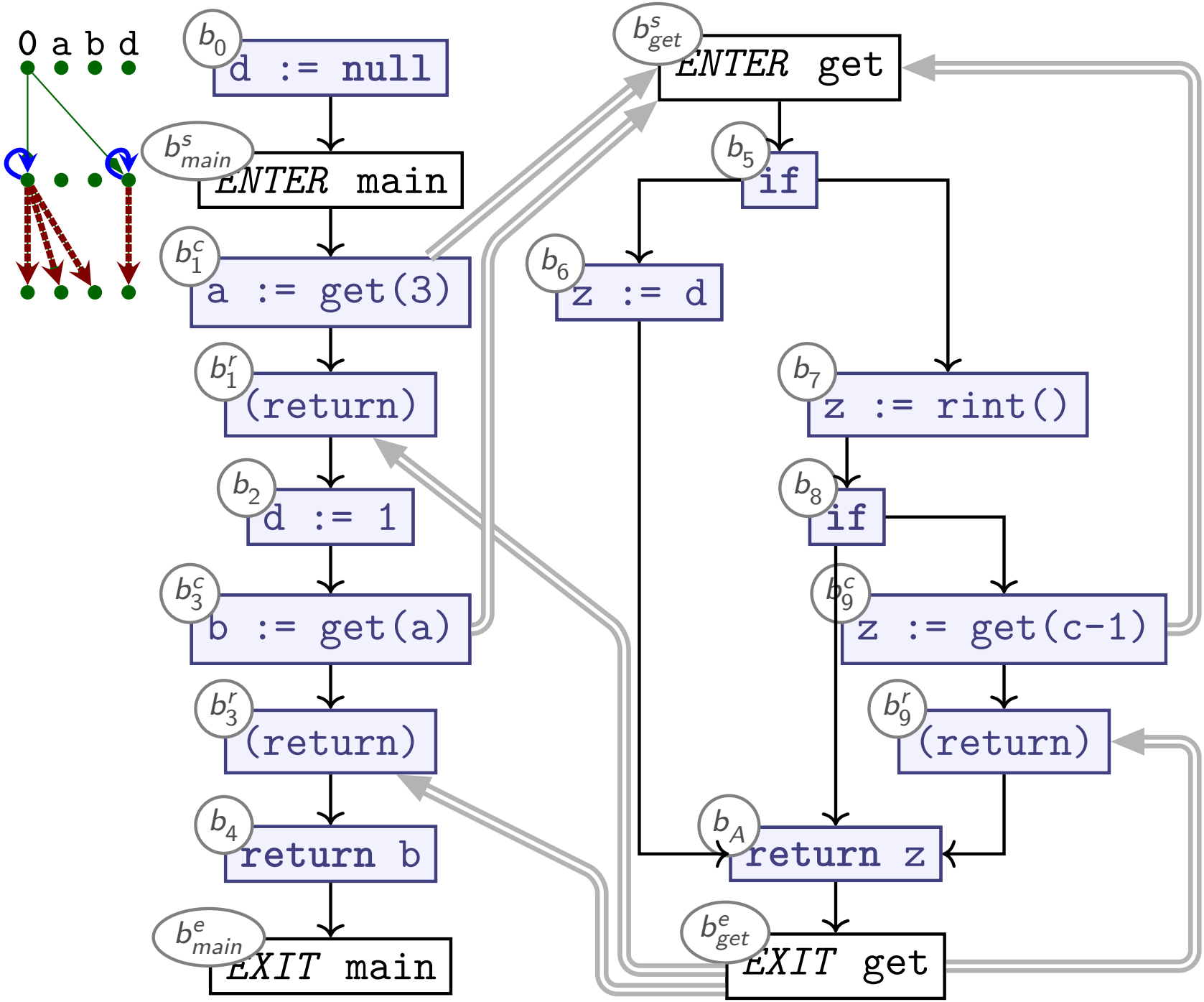


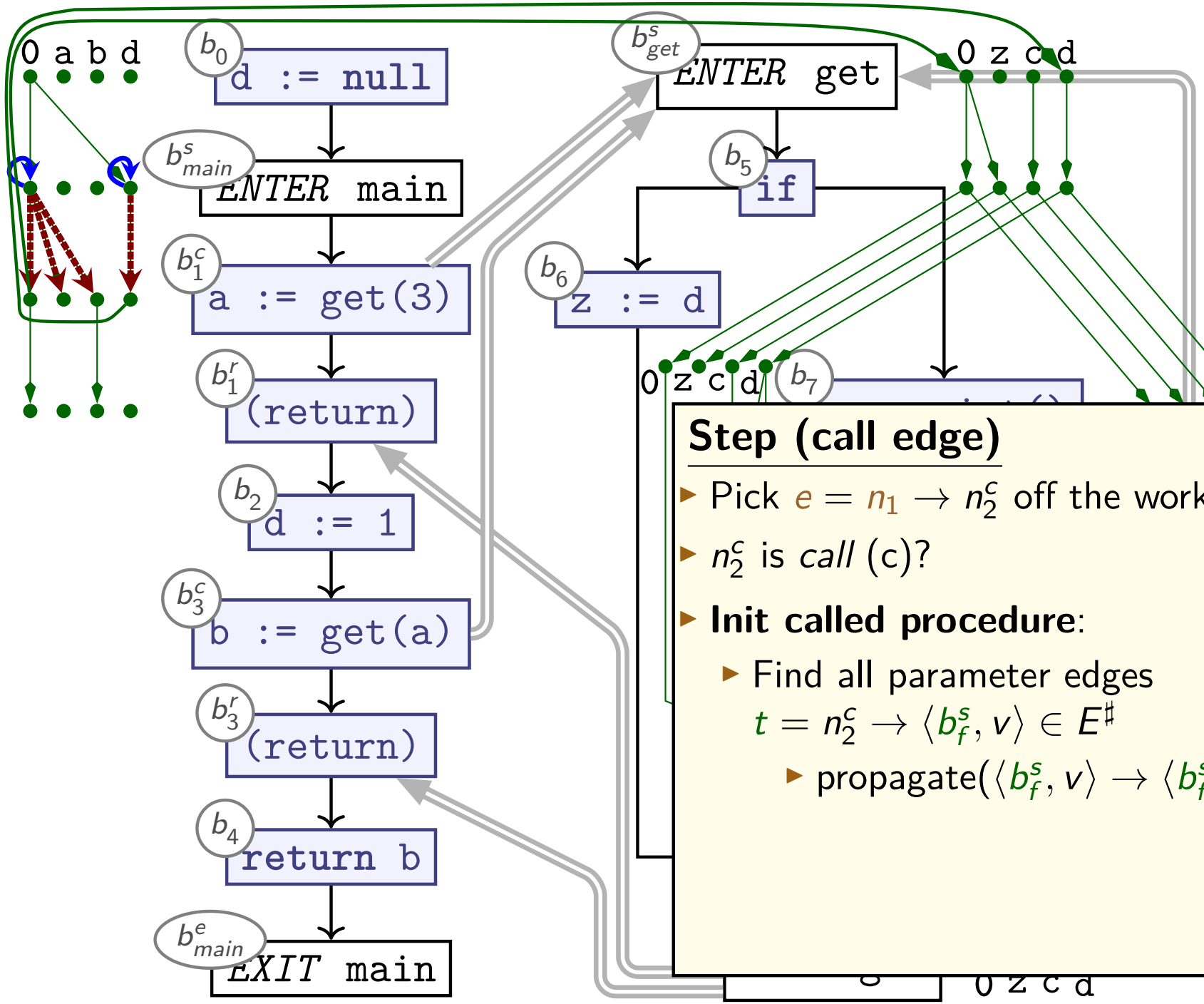


Procedure propagate($n_1 \to n_2$):
begin
  if $n_1 \to n_2 \in$ PathEdge **then**
    return
  PathEdge := PathEdge $\cup \{n_1 \to n_2\}$
  WorkList := WorkList $\cup \{n_1 \to n_2\}$
end

Step (regular edge)

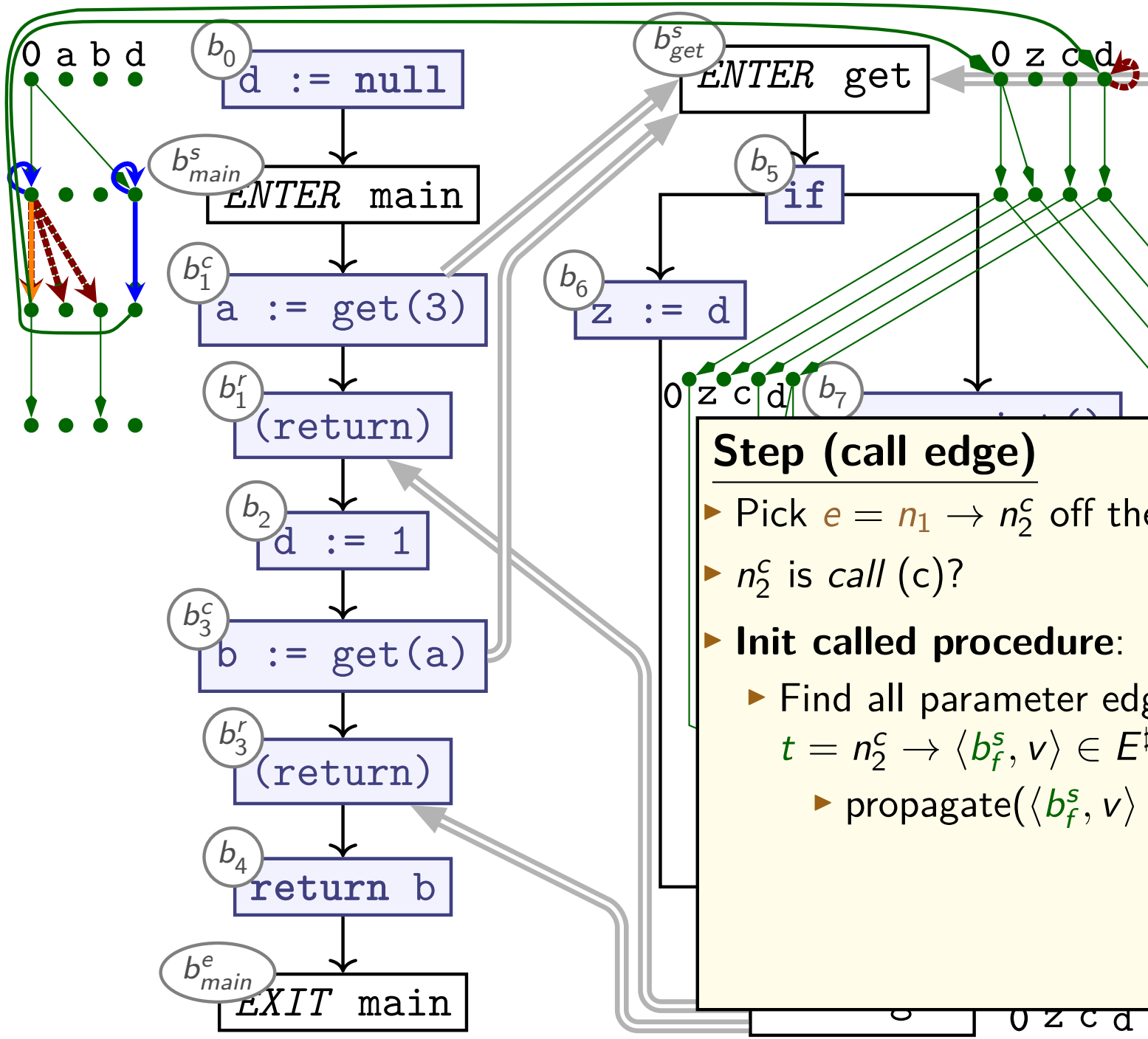
- Pick e off the work queue
  $e = n_1 \to n_2$
- n_2 neither call (c) nor exit (e)?
- Find all $n_2 \to n_3$
  propagate($n_1 \to n_3$)
- Remove e from WorkList
- e remains in PathEdge


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



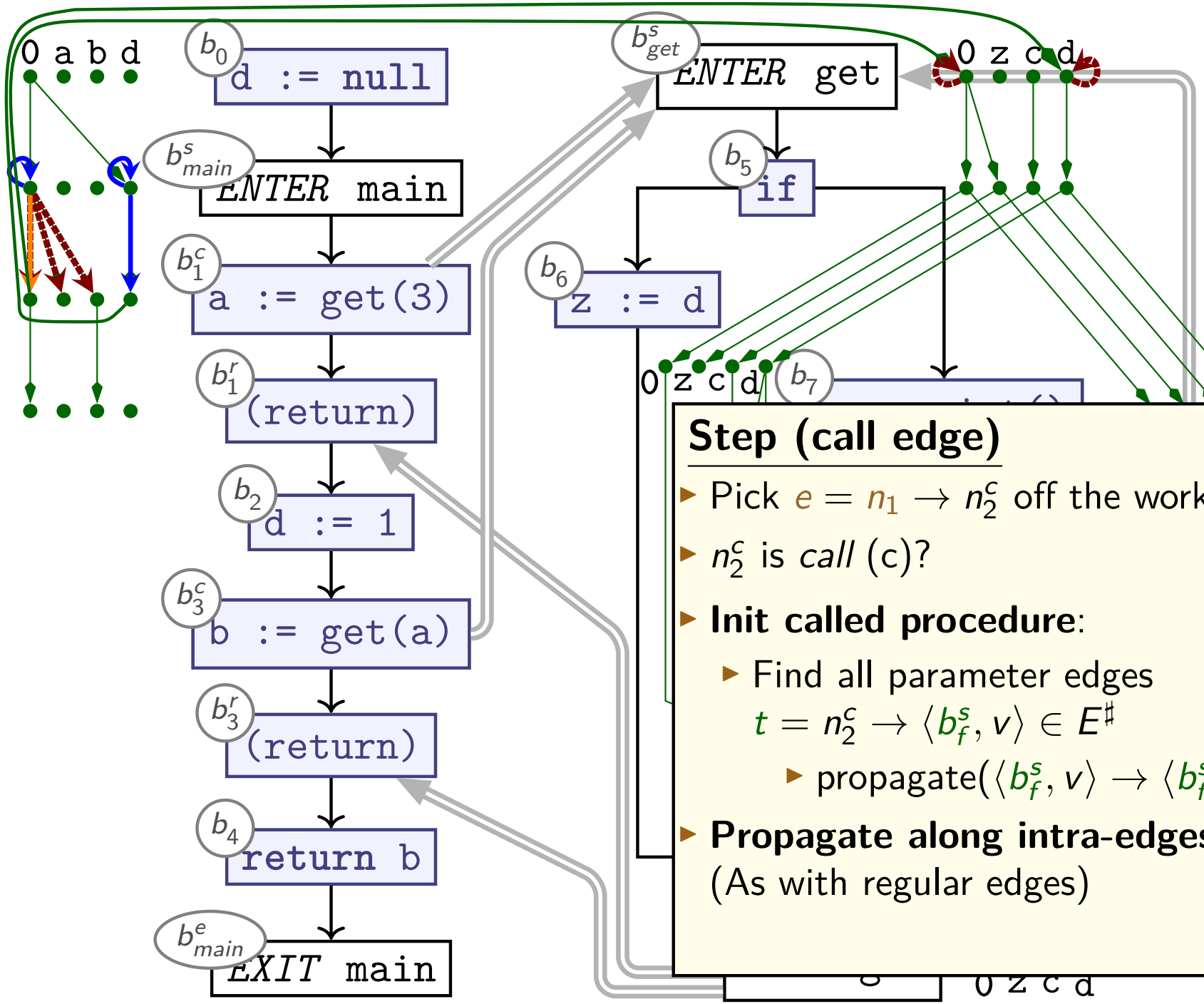
Step (call edge)

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



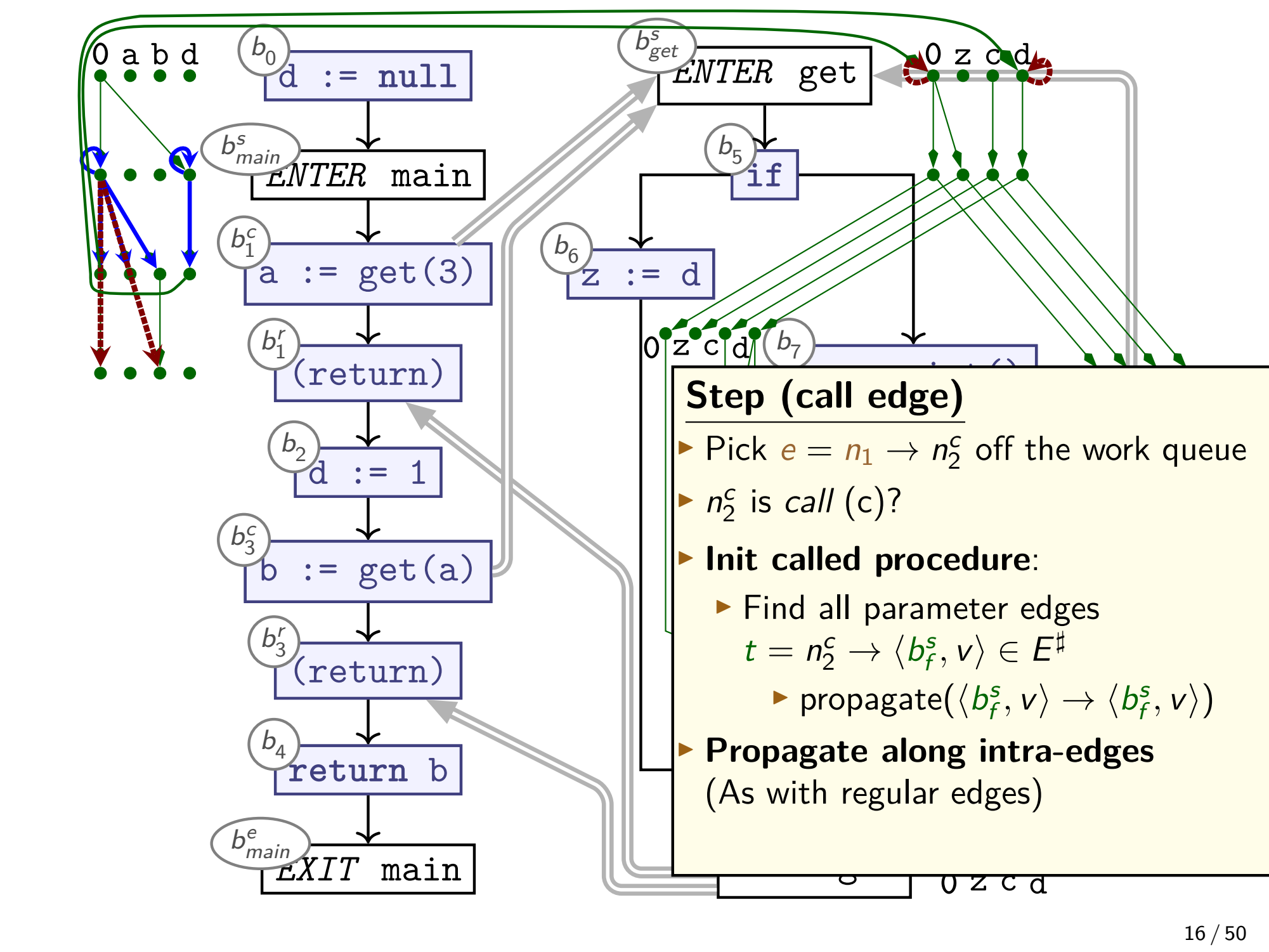
Step (call edge)

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



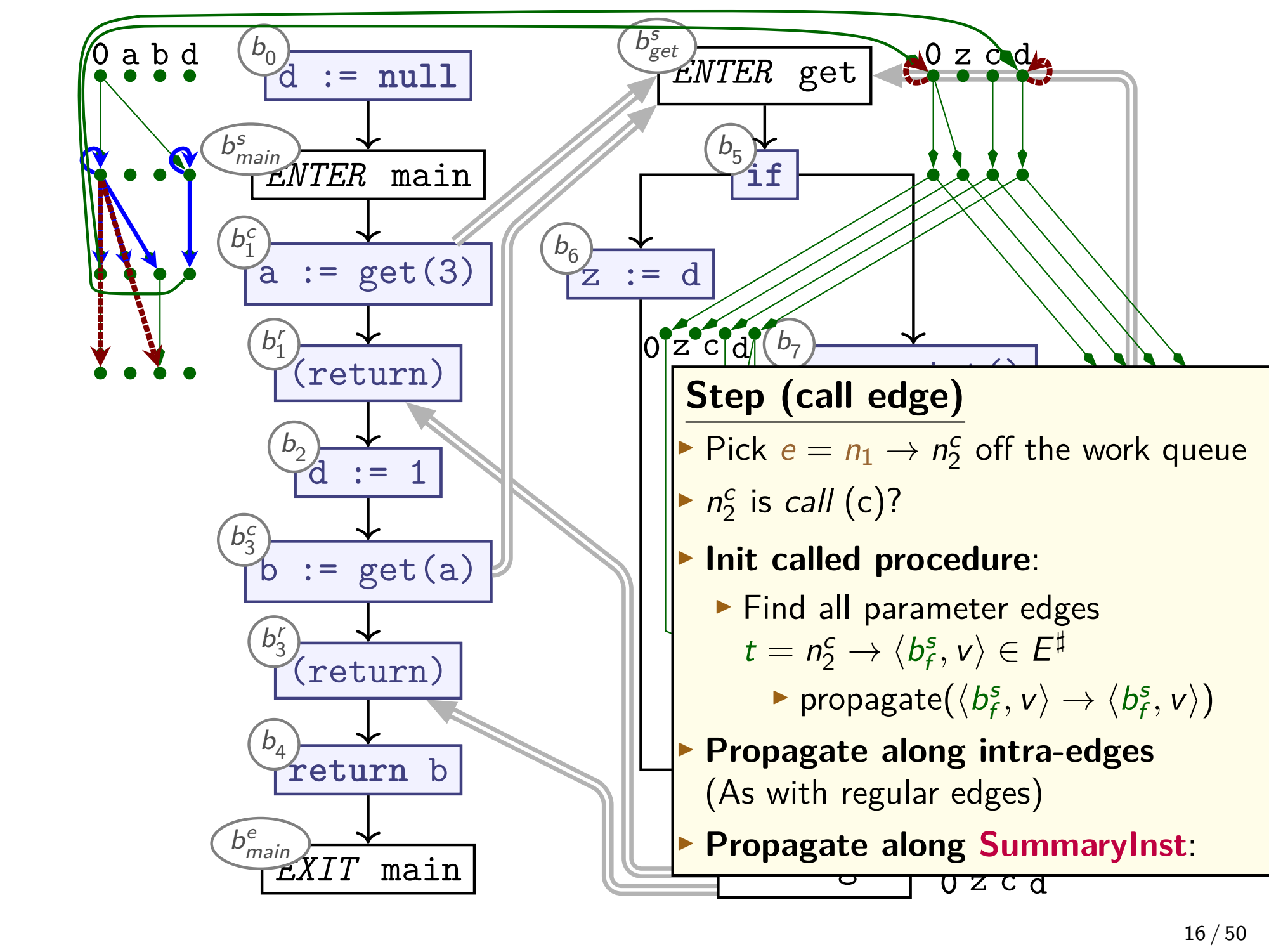
Step (call edge)

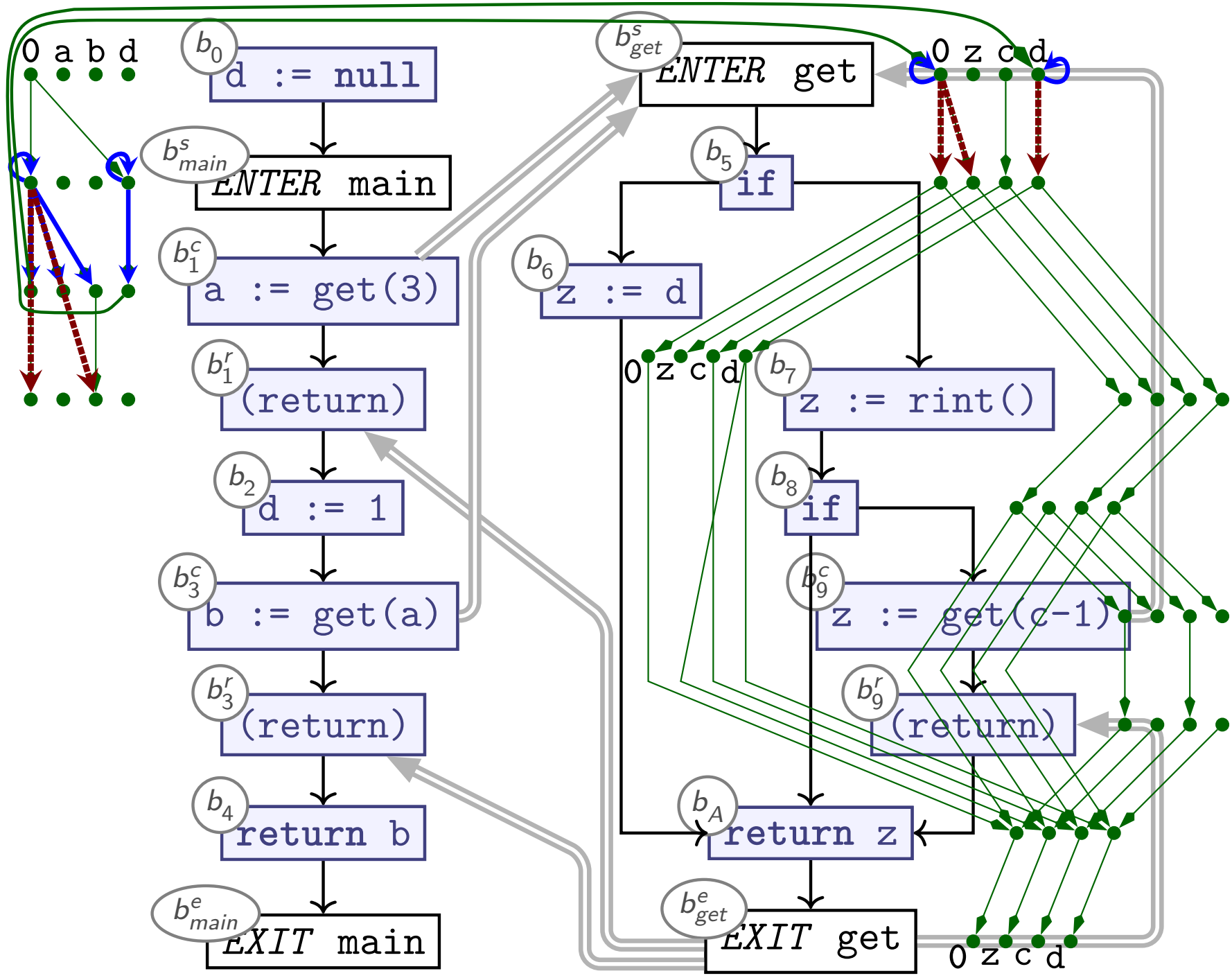
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \to \langle b_f^s, v \rangle$)
- **Propagate along intra-edges** (As with regular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

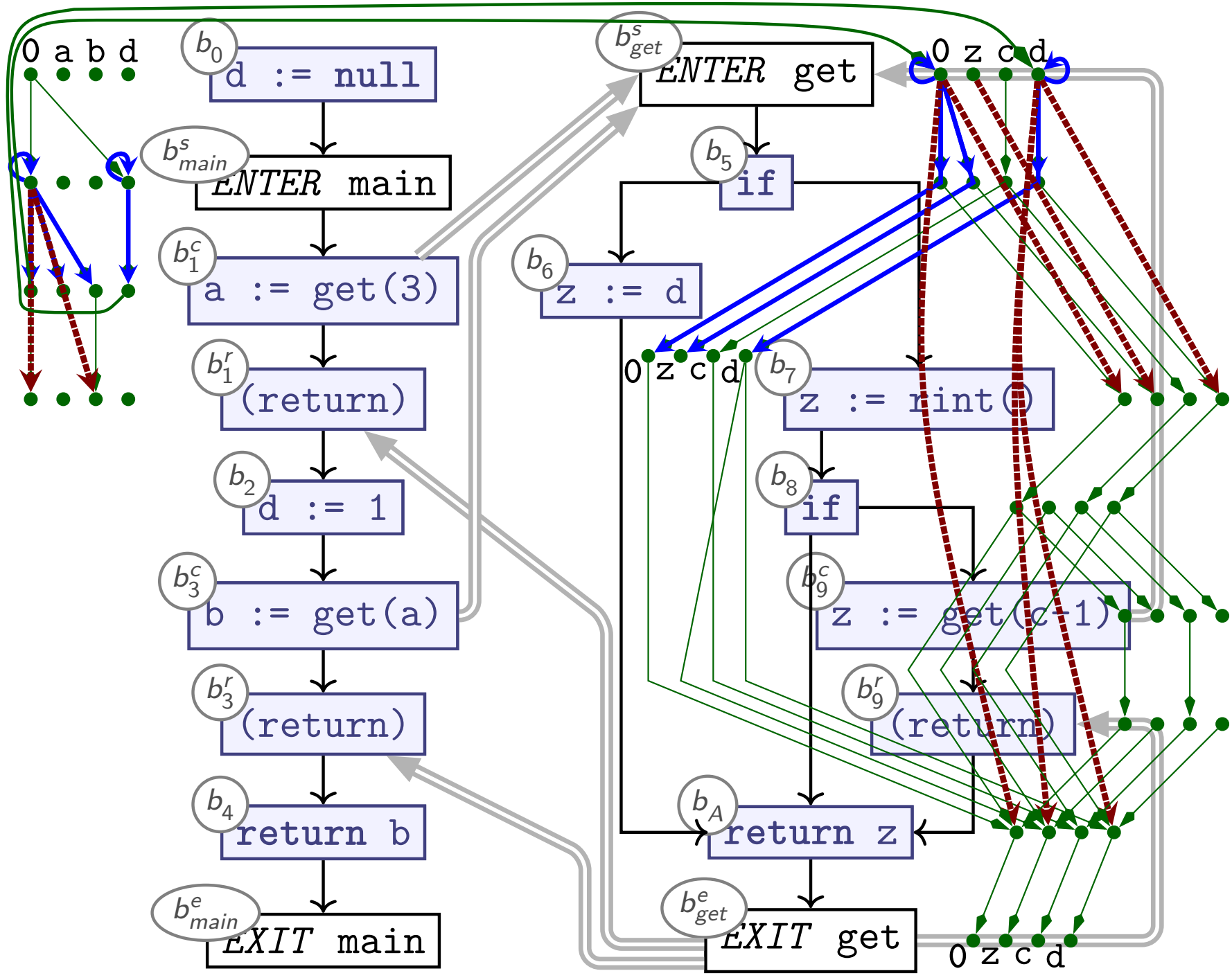


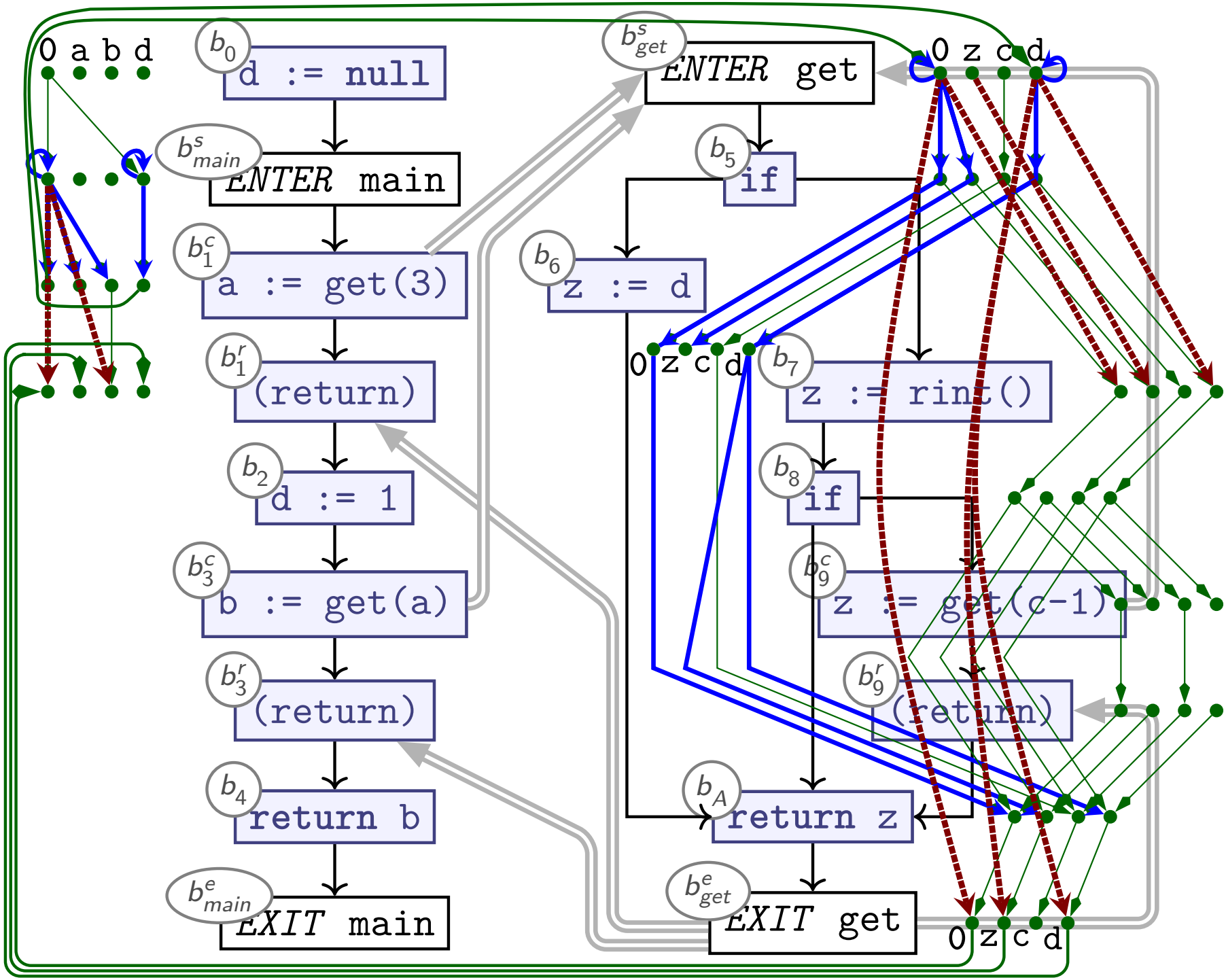
Step (call edge)

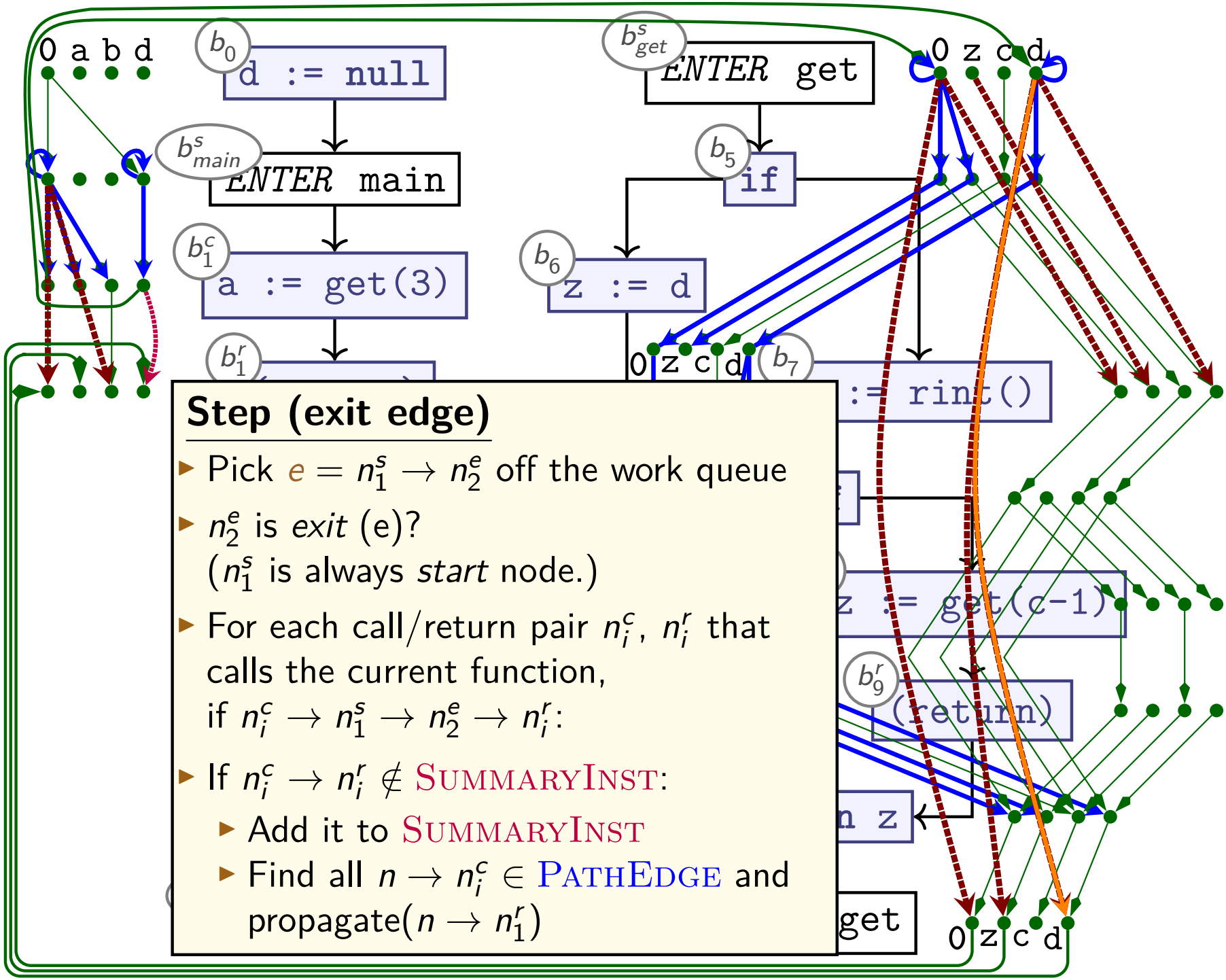
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \to \langle b_f^s, v \rangle$)
- **Propagate along intra-edges** (As with regular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


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
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:


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
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 z c d

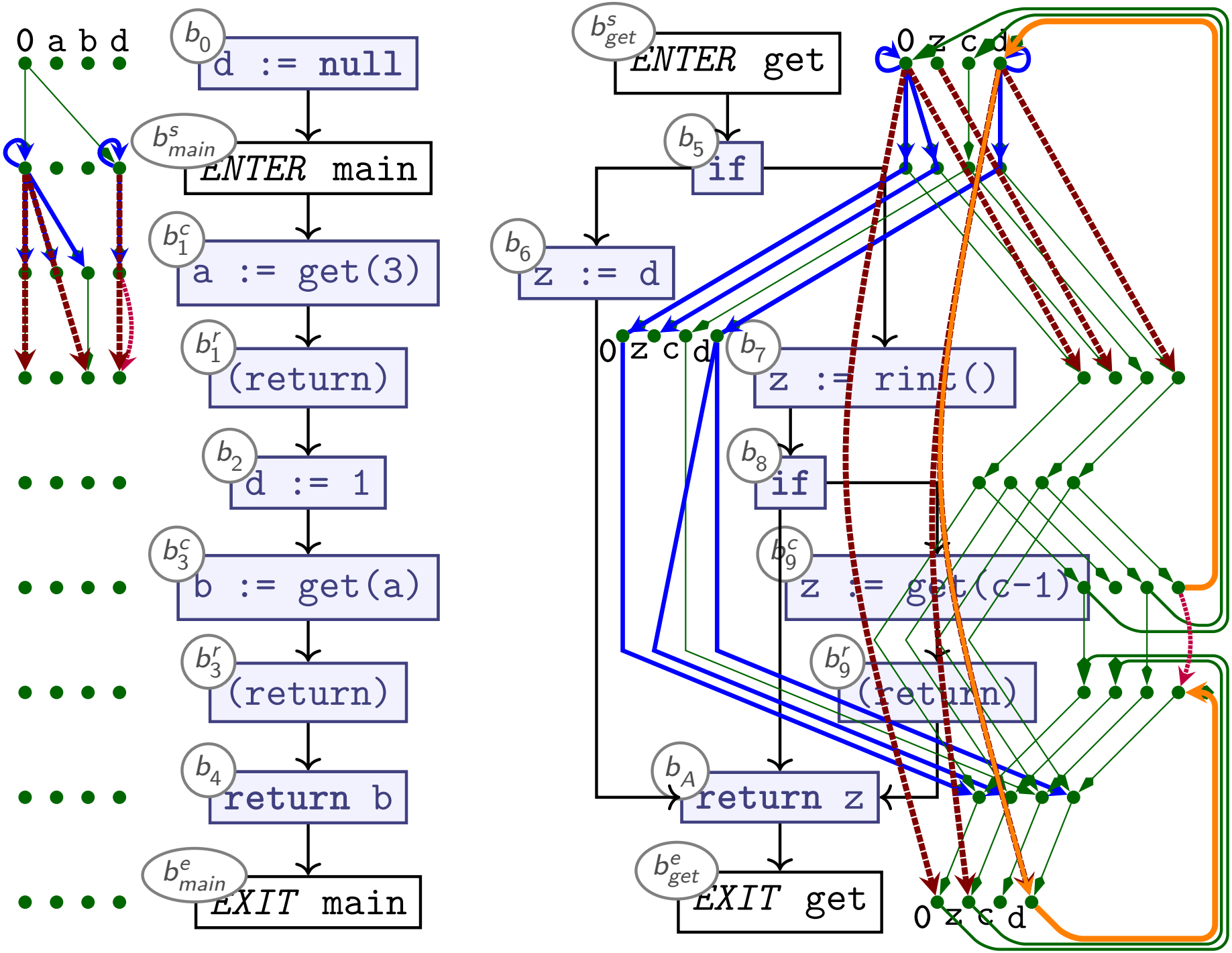

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d
0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

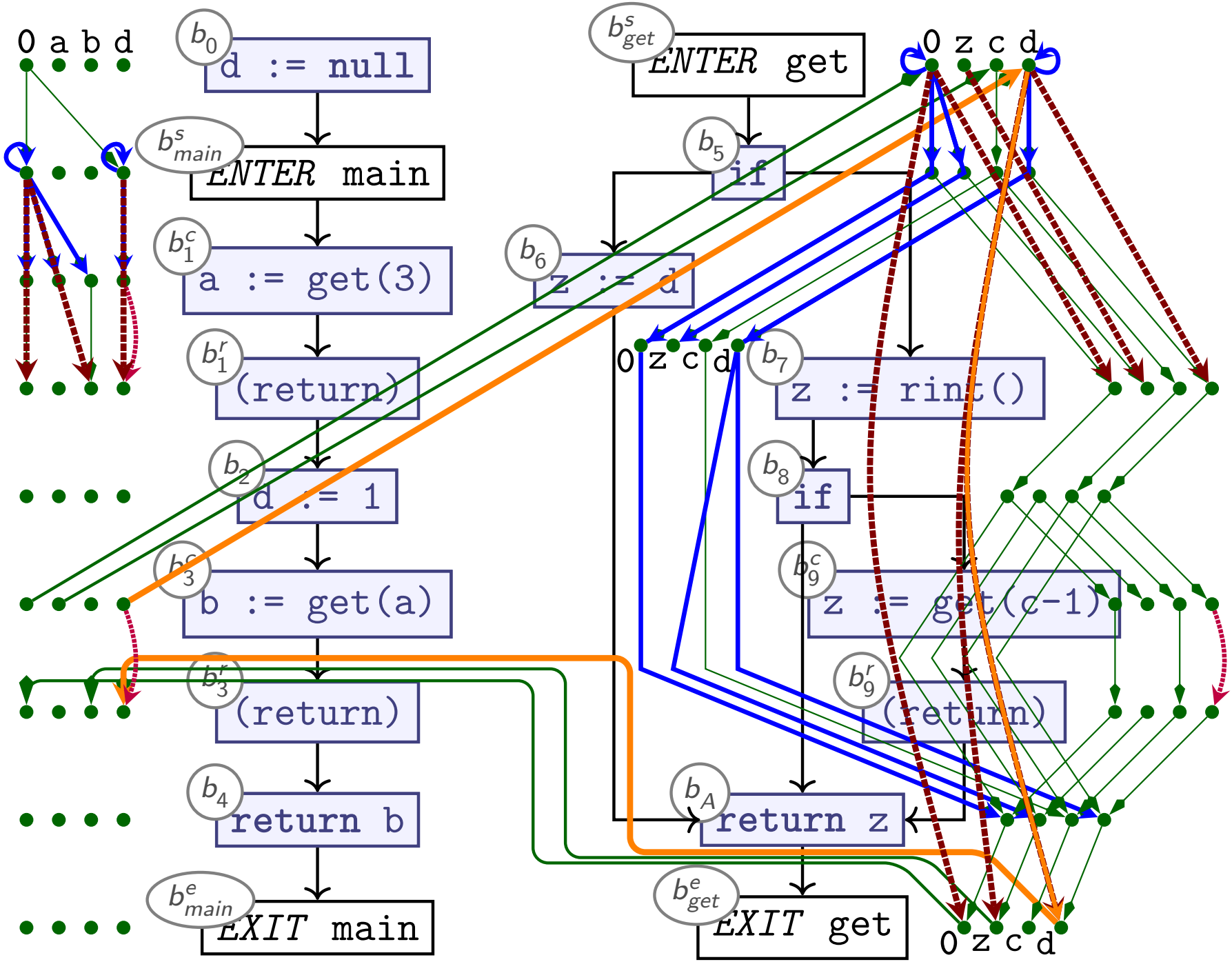

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d
0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

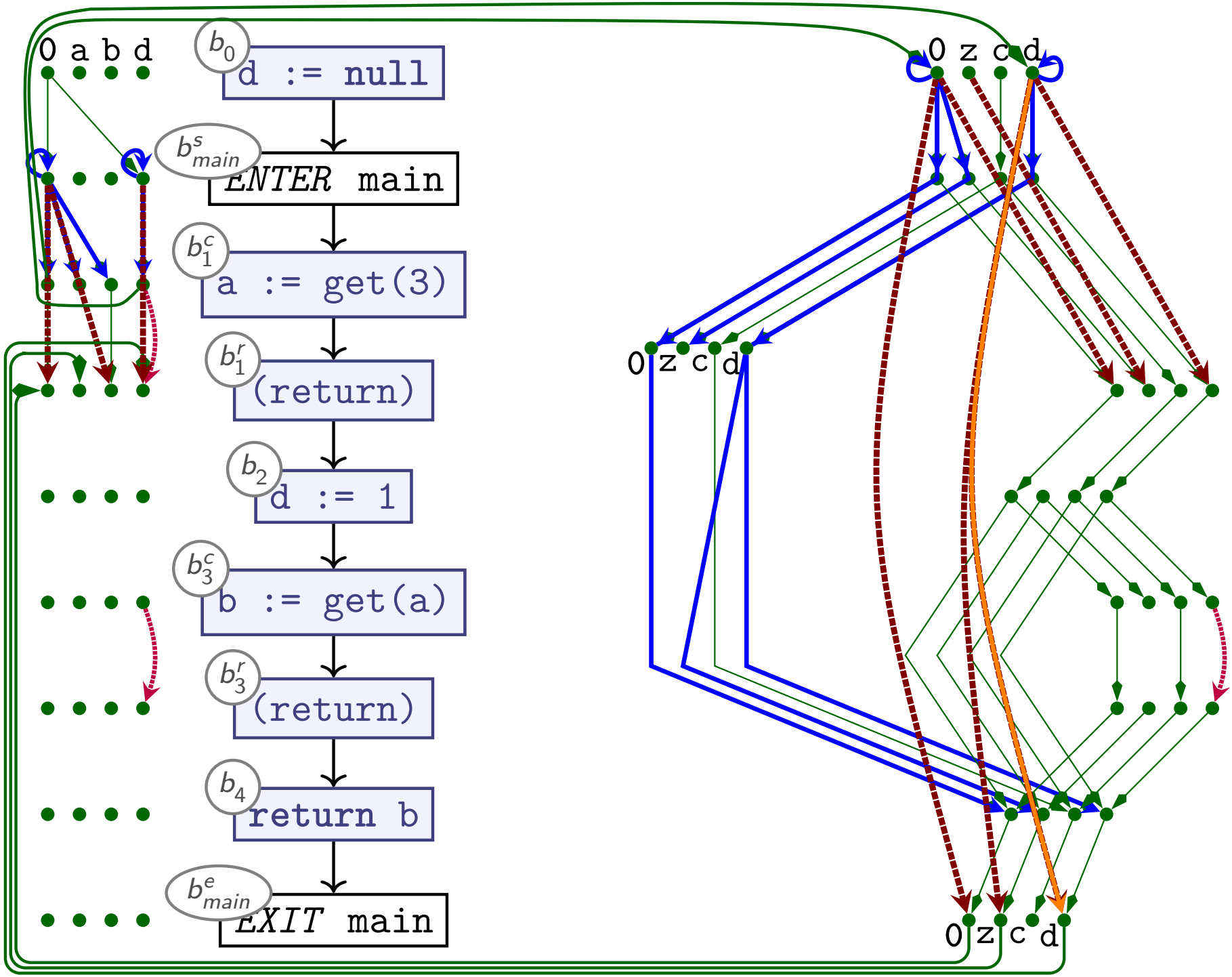


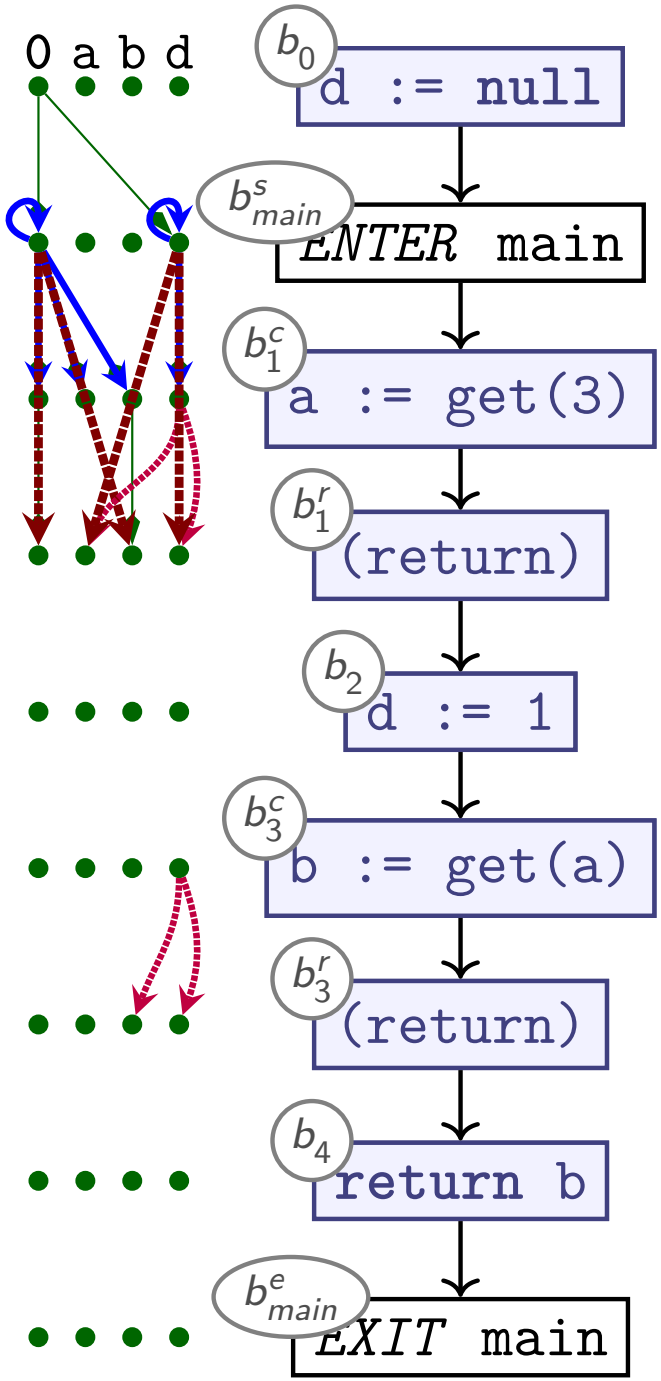
Step (exit edge)

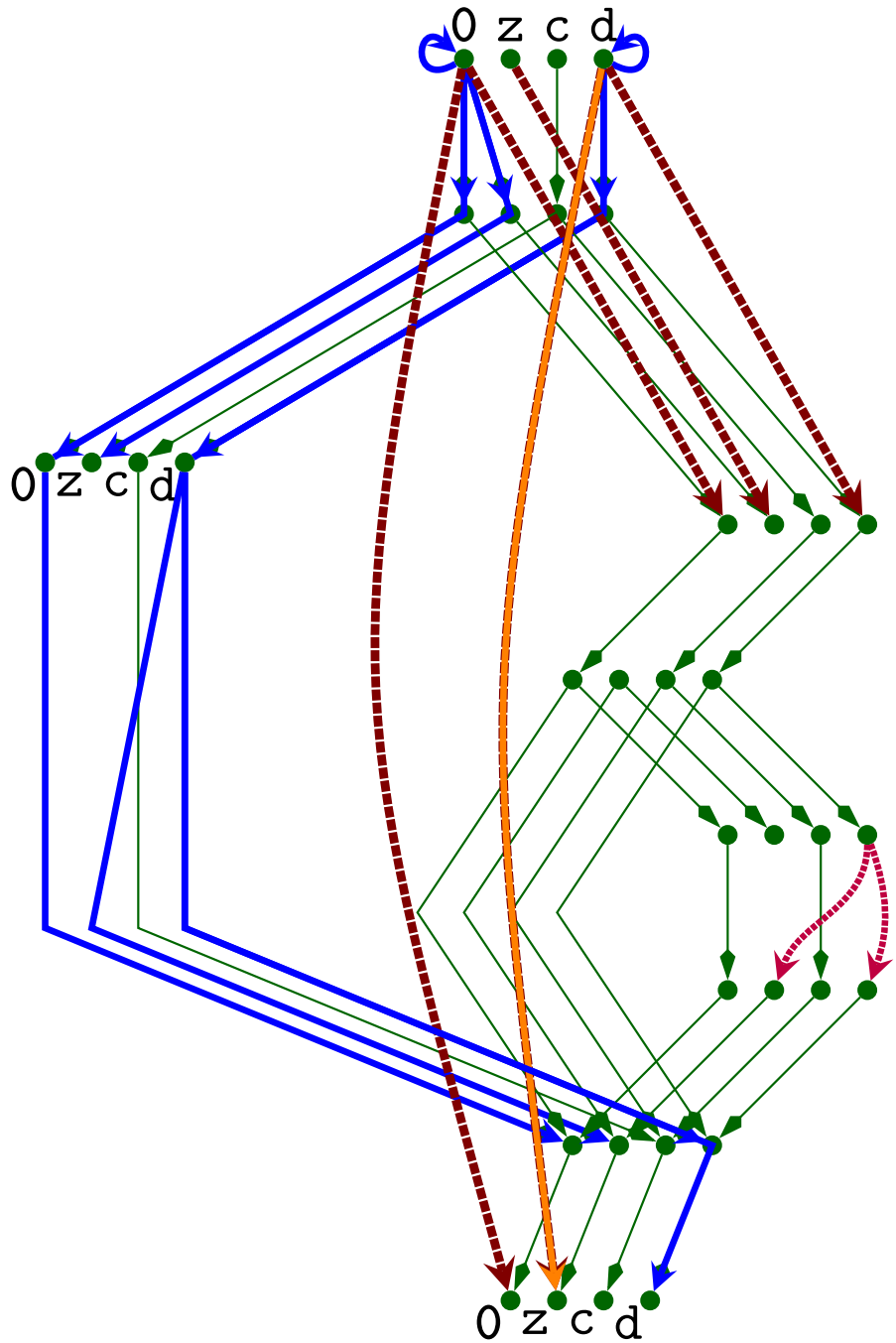
- Pick $e = n^s_1 \to n^e_2$ off the work queue
- n^e_2 is *exit* (e)?
  (n^s_1 is always *start* node.)
- For each call/return pair n^c_i, n^r_i that calls the current function,
  if $n^c_i \to n^s_1 \to n^e_2 \to n^r_i$:
- If $n^c_i \to n^r_i \notin$ SummaryInst:
  - Add it to SummaryInst
  - Find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

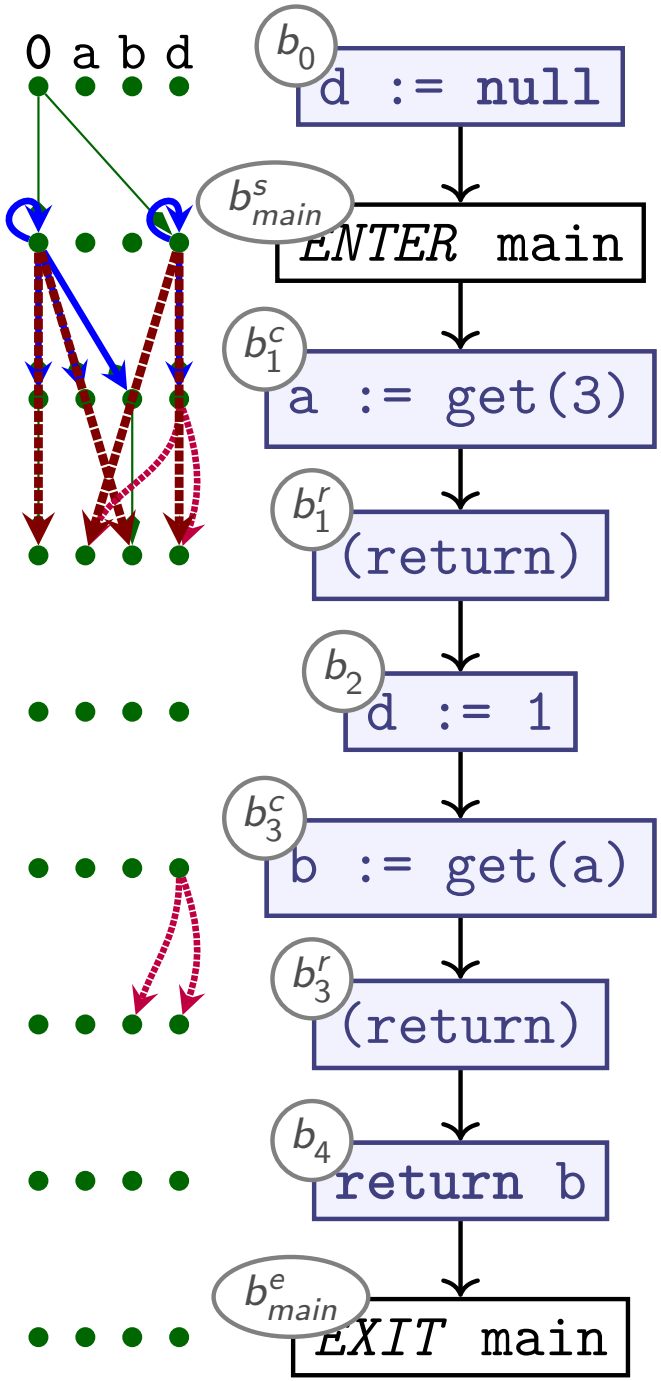

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z := d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d

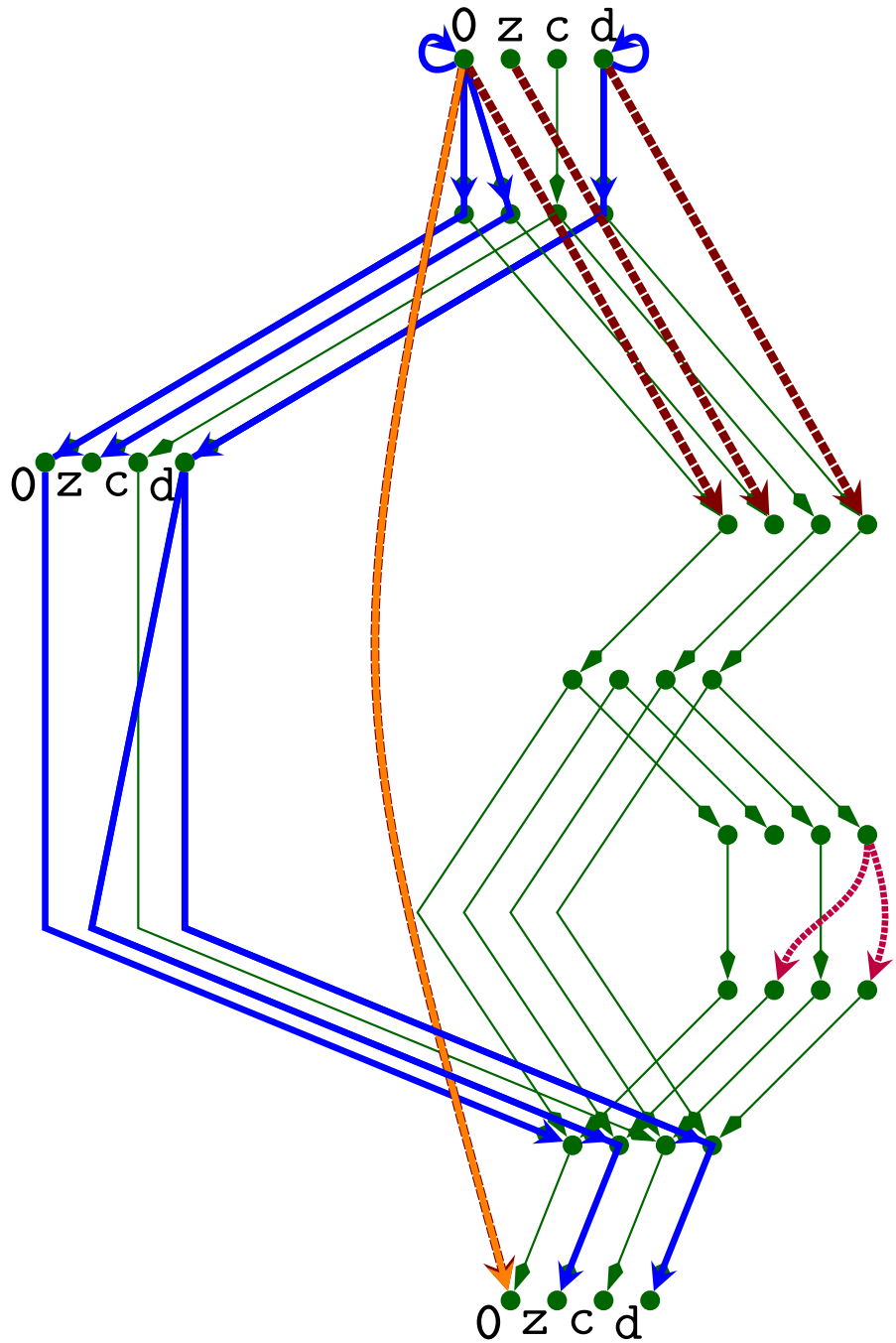

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
b^s_{get}
ENTER get
b_5
if
b_6
z := d
0 z c d
b_7
z := rint()
b_8
if
b^c_9
z := get(c-1)
b^r_9
(return)
b_A
return z
b^e_{get}
EXIT get
0 z c d
0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

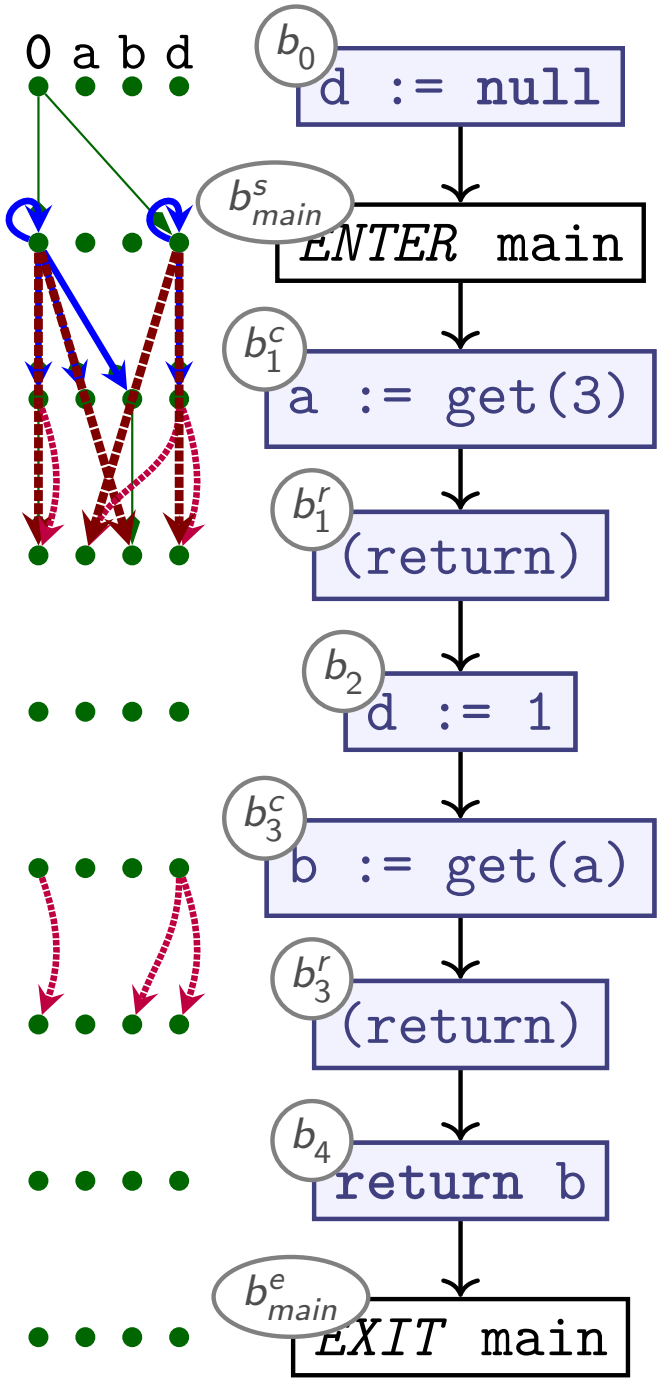

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main
0 z c d
0 z c d
0 z c d

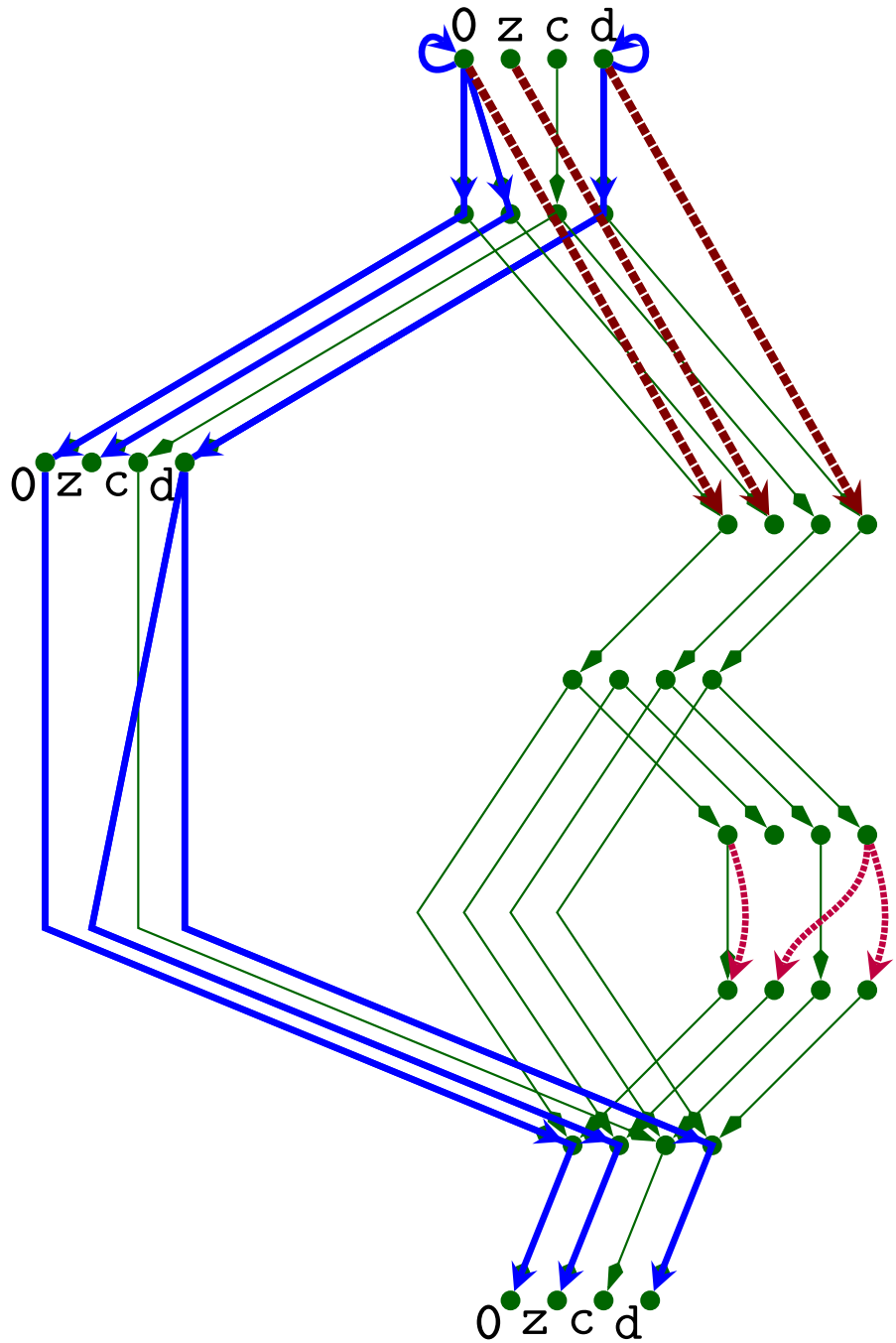

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main

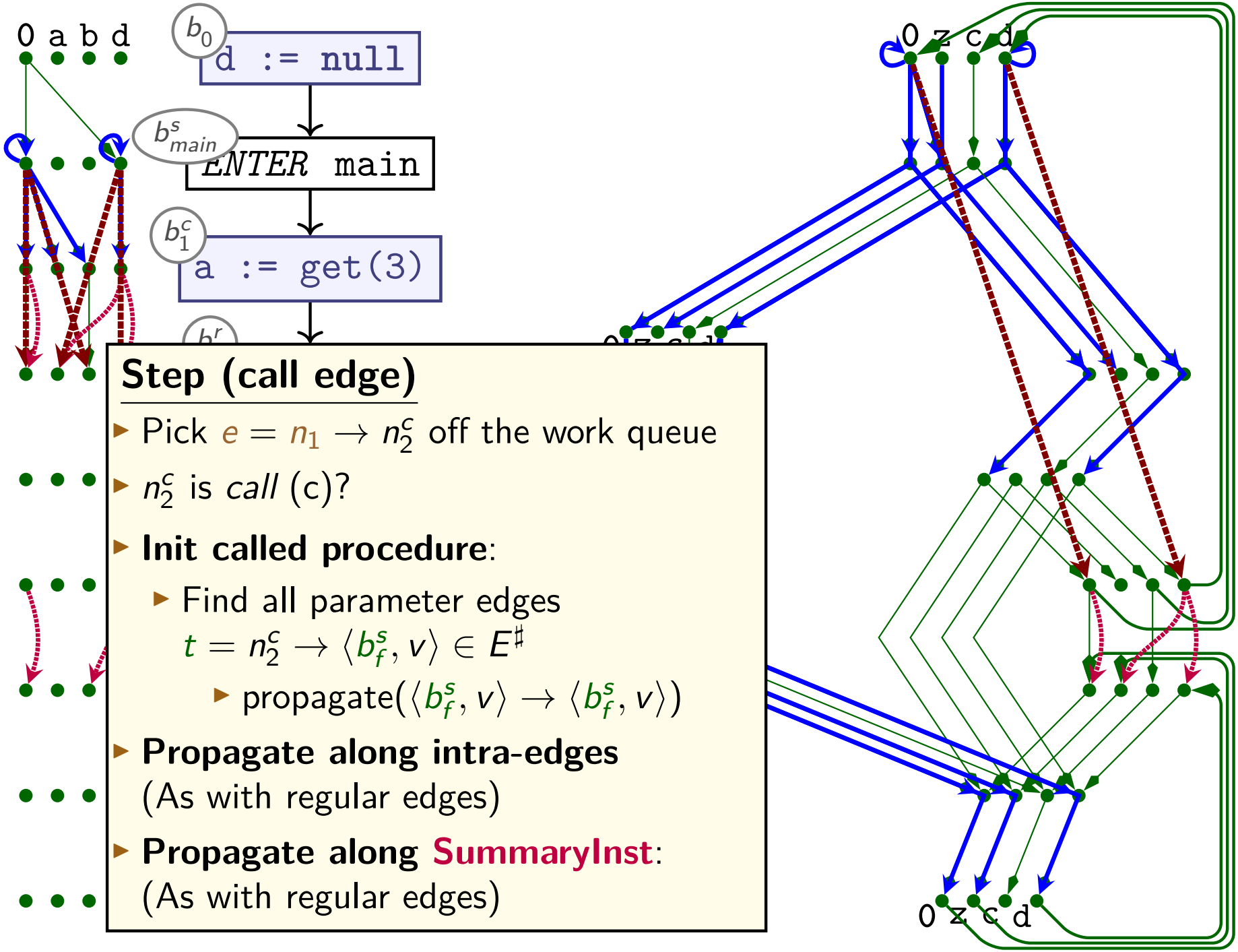

0 z c d
0 z c d
0 z c d


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

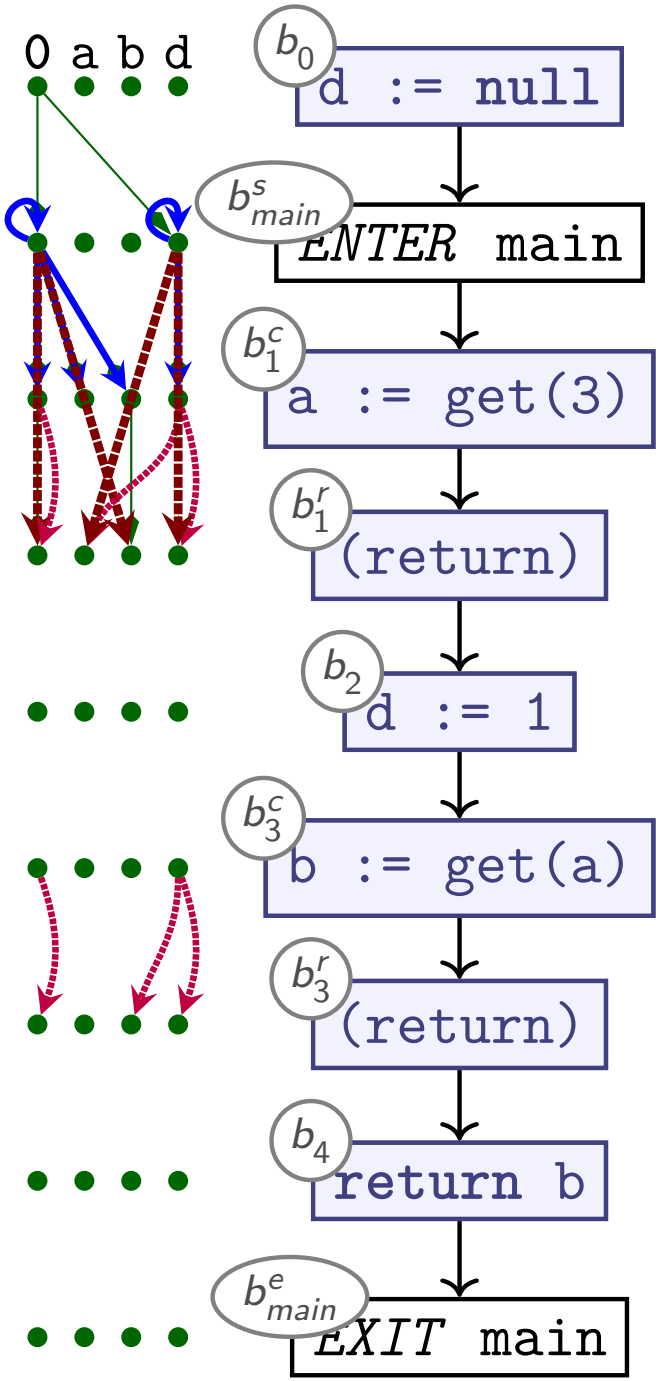

0 a b d
b_0
d := null
b^s_{main}
ENTER main
b^c_1
a := get(3)
b^r_1
(return)
b_2
d := 1
b^c_3
b := get(a)
b^r_3
(return)
b_4
return b
b^e_{main}
EXIT main

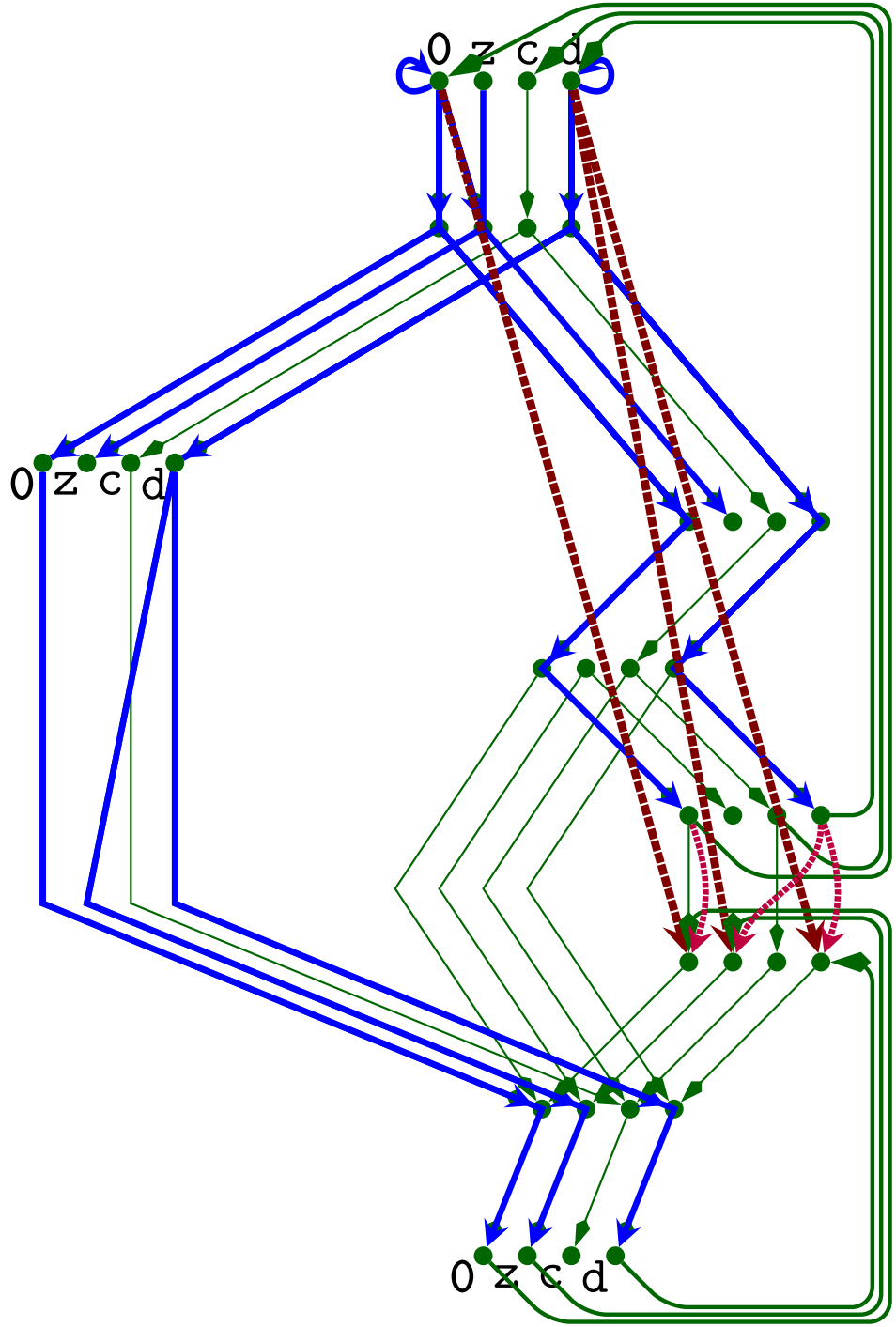

0 z c d
0 z c d
0 z c d

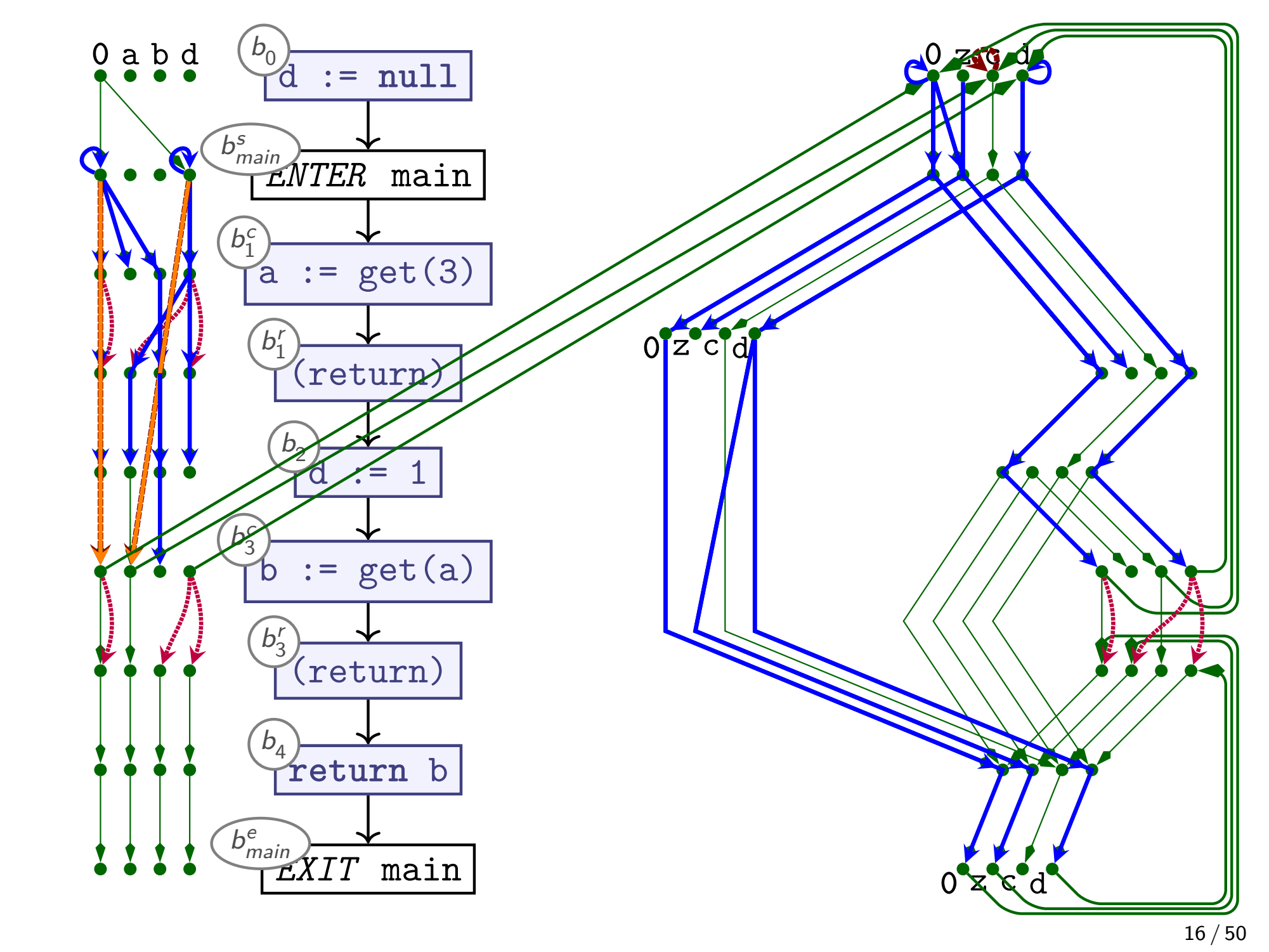


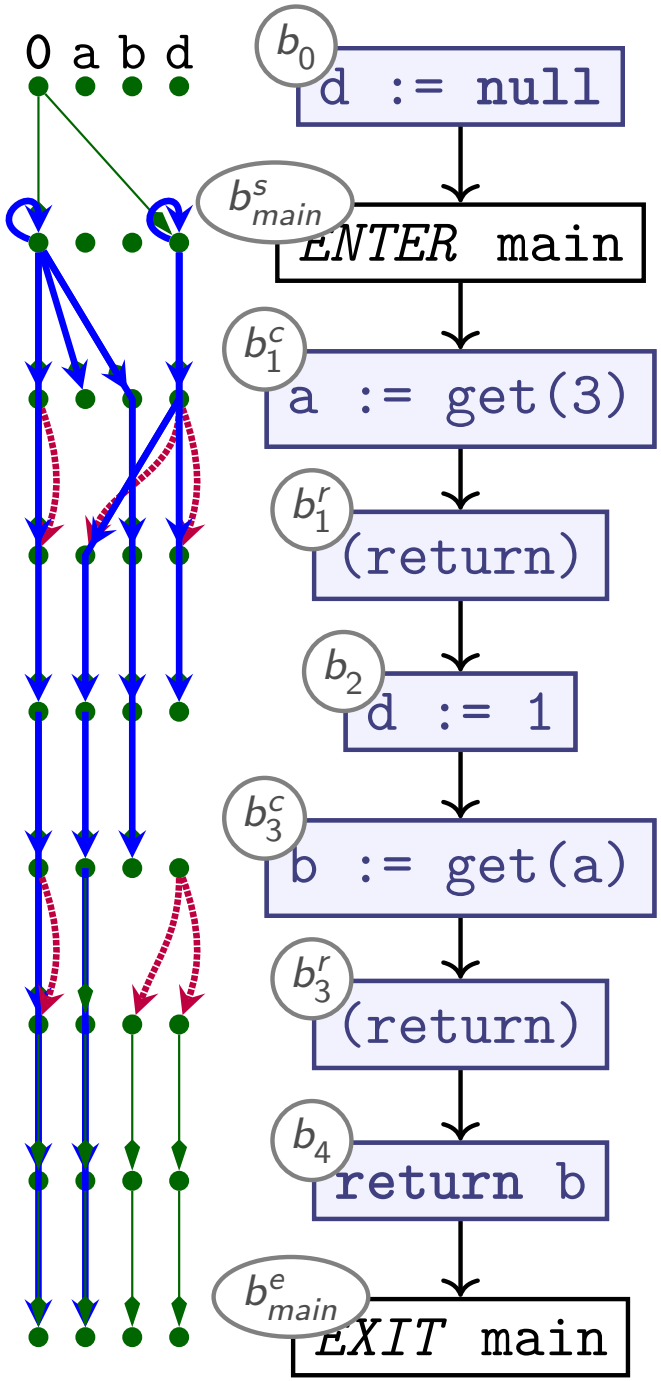
Step (call edge)

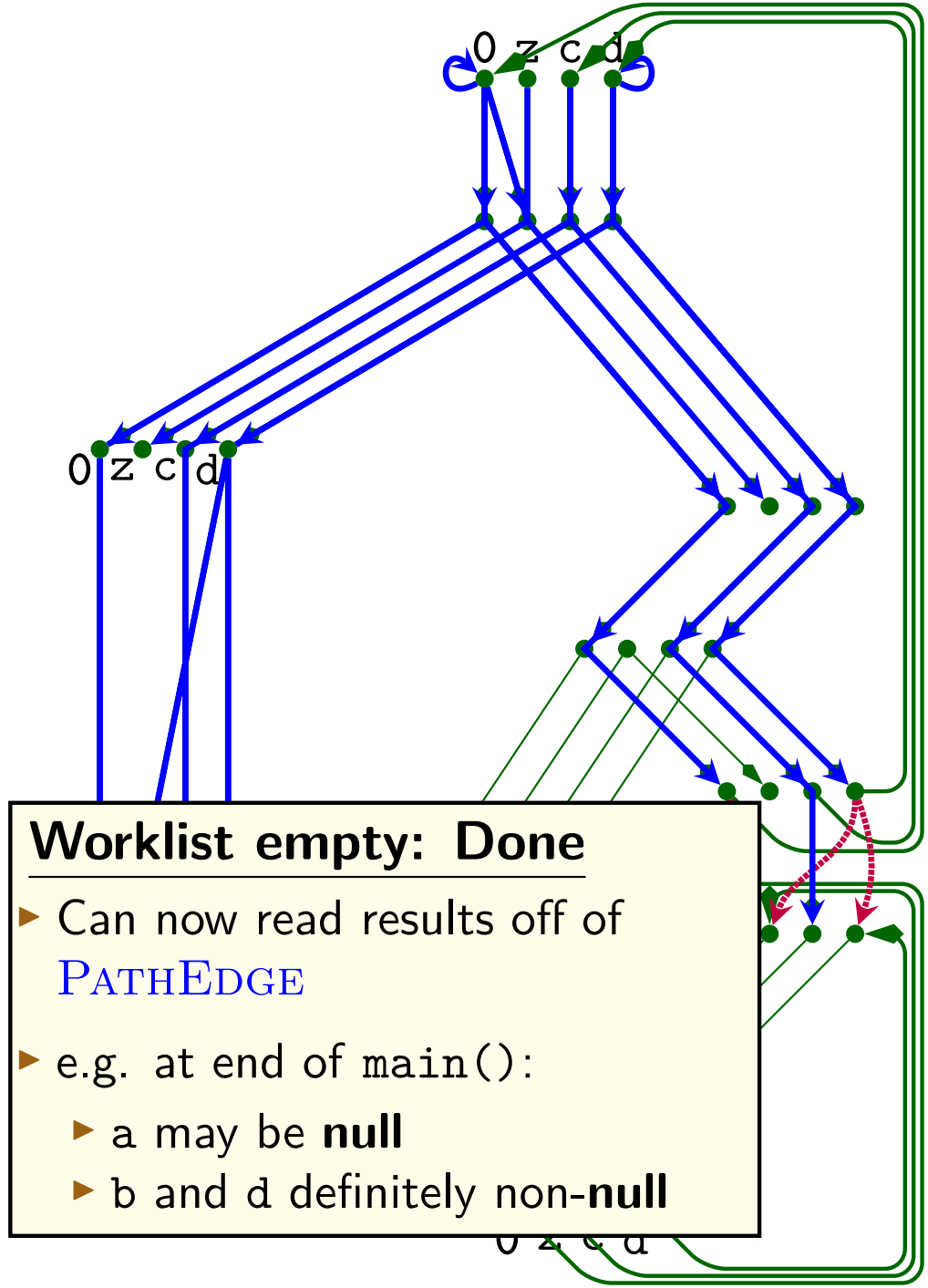
- Pick $e = n_1 \to n_2^c$ off the work queue
- n_2^c is *call* (c)?
- **Init called procedure**:
  - Find all parameter edges $t = n_2^c \to \langle b_f^s, v \rangle \in E^\sharp$
    - propagate($\langle b_f^s, v \rangle \to \langle b_f^s, v \rangle$)
- **Propagate along intra-edges** (As with regular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

IFDS: Forward Tabulation

Procedure ForwardTabulate():
begin
  while $n_0 \to n_1 \in$ 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 $n_1 \to n_2 \in E^\sharp \cup$ 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** $n_s \to n_r \in$ SummaryInst **then**
          SummaryInst := SummaryInst $\cup \{n_s \to n_r\}$
          foreach $n_z \to n_s \in$ 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**

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`

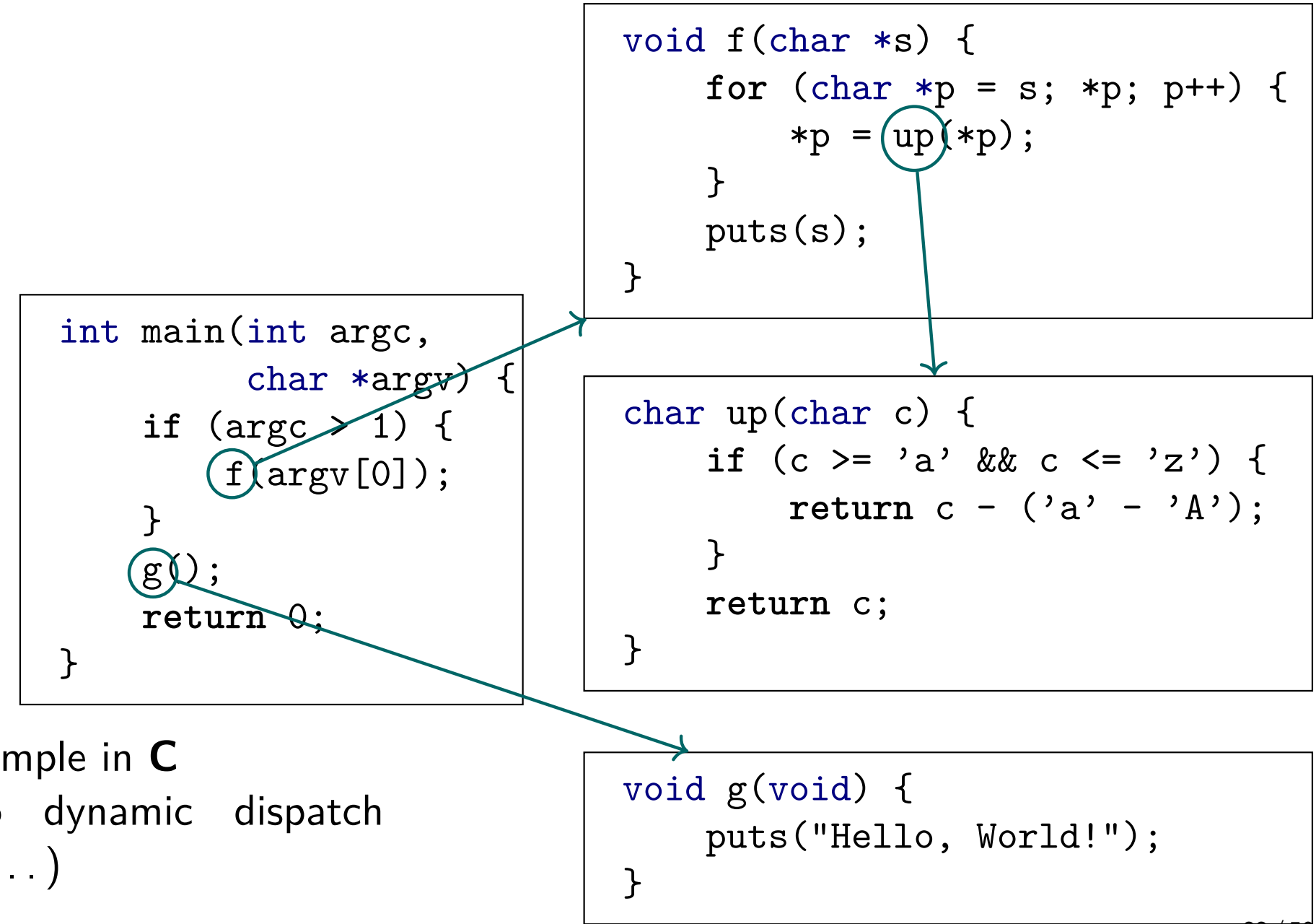
Challenges

- **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?

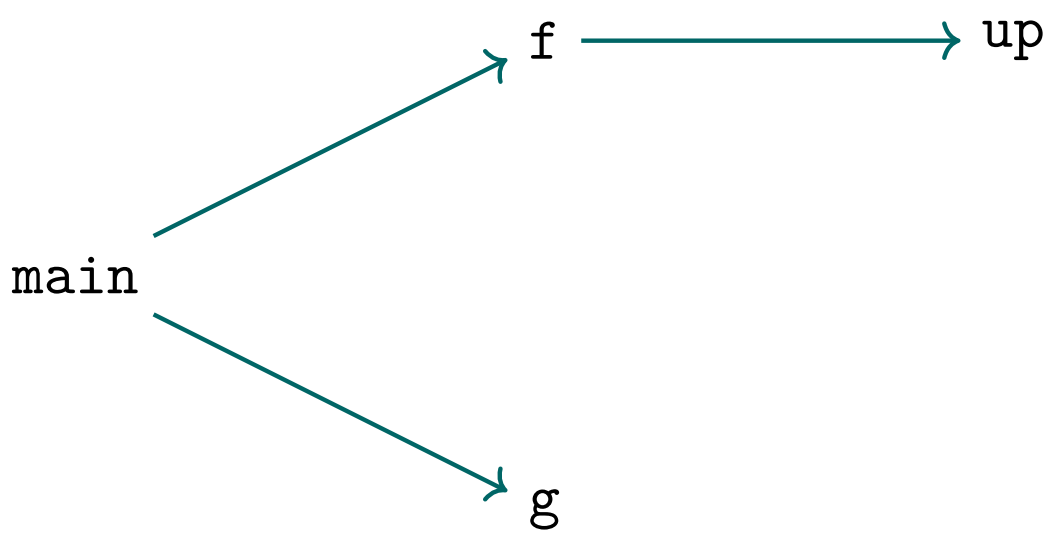
The Call Graph



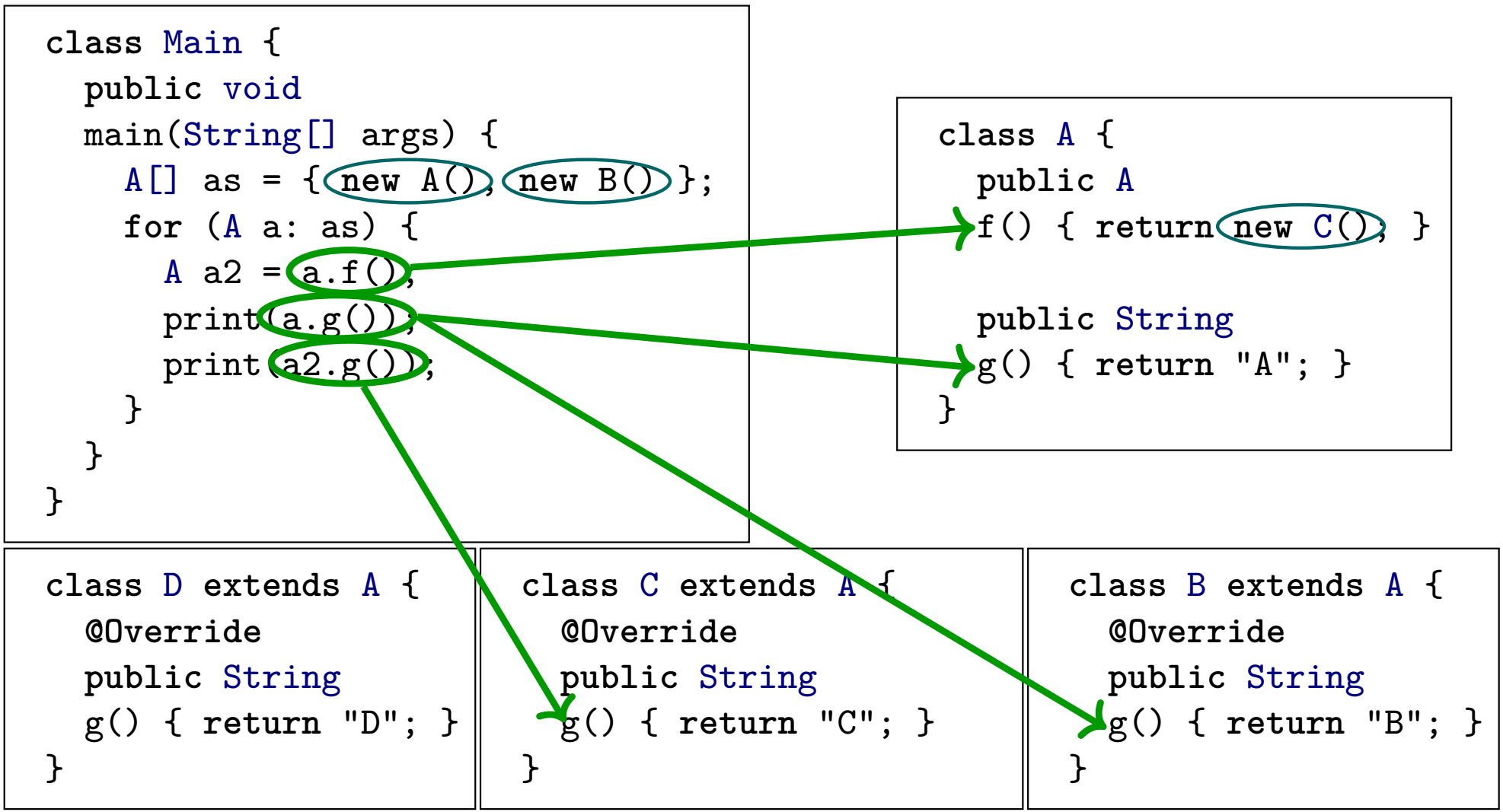
Example in **C**
(No dynamic dispatch yet...)

The Call Graph

- $G_{\text{call}} = \langle P, E_{\text{call}} \rangle$
- 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

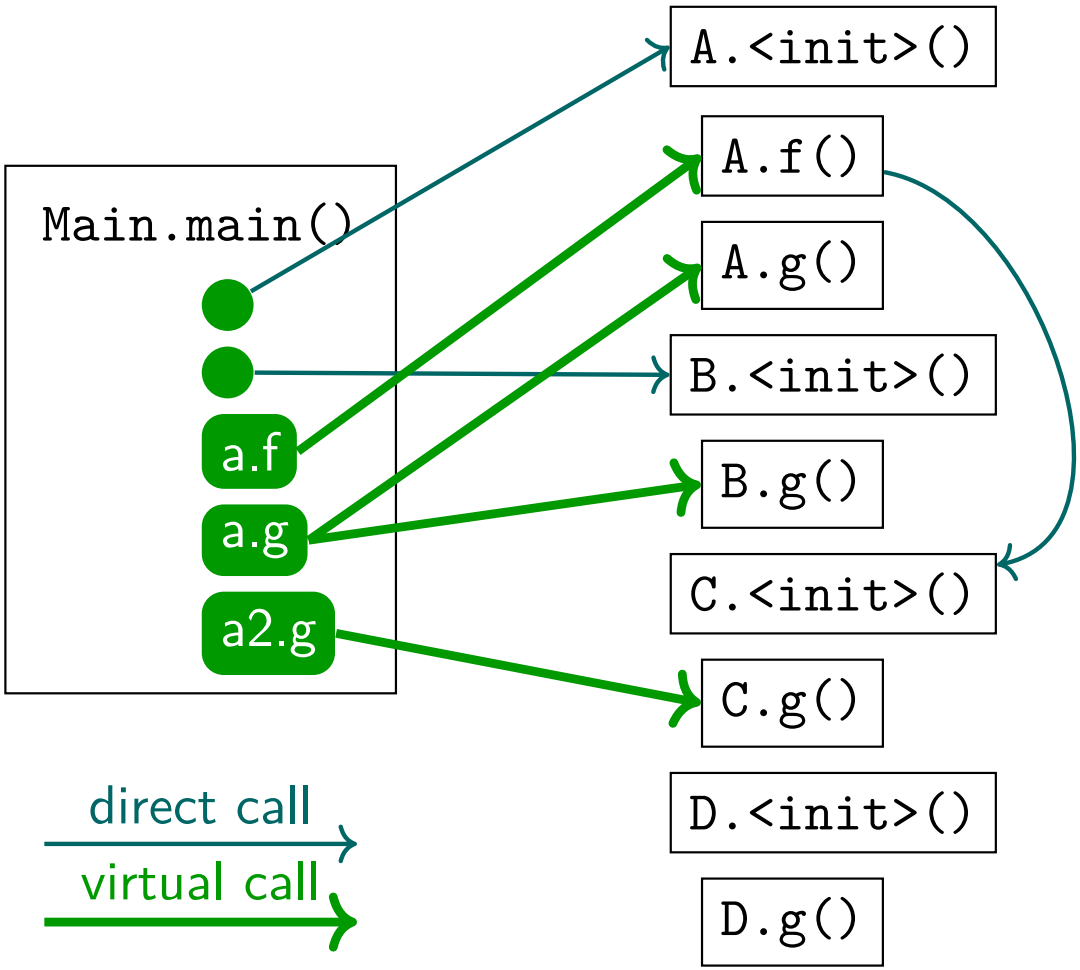


Finding Calls and Targets



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());
    }
  }
}
```

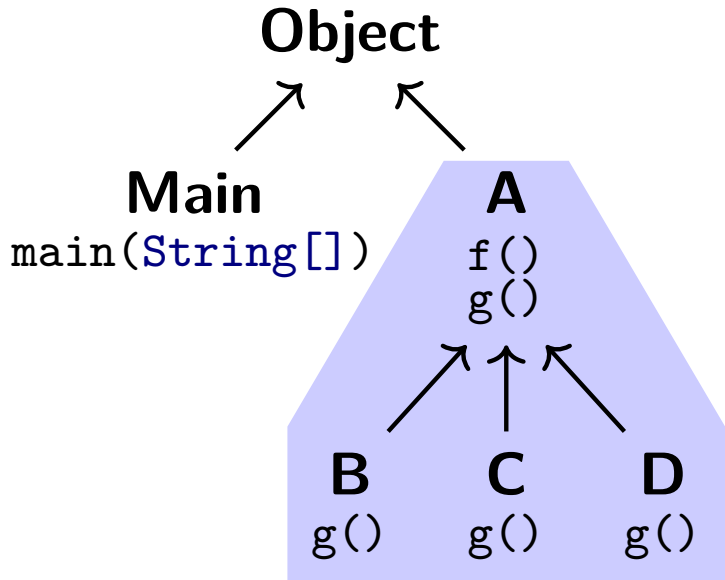
```
class A {
  public A
  f() { return new C(); }

  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"; }
}
```

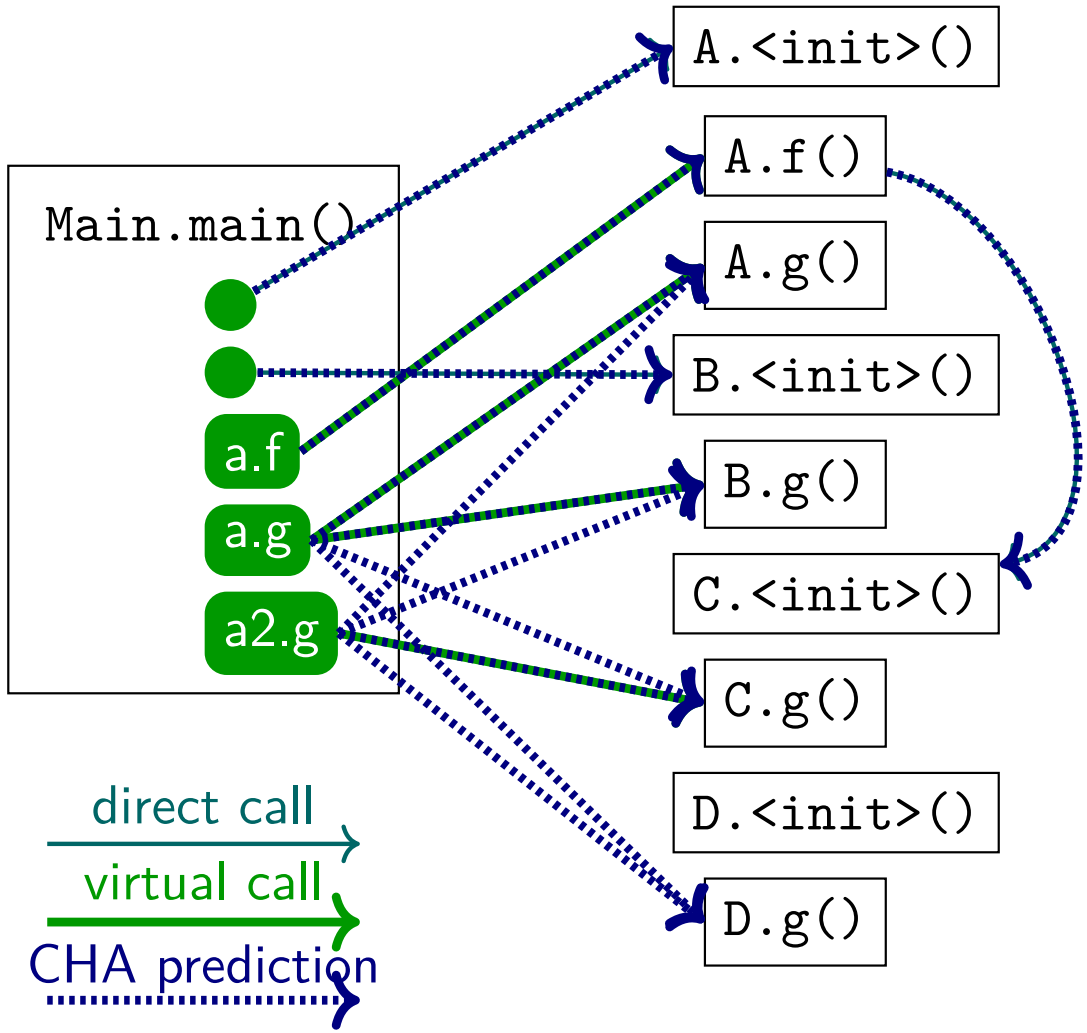
```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

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
  - Assuming whole-program knowledge (no dynamic classloading)
- 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());
    }
  }
}
```

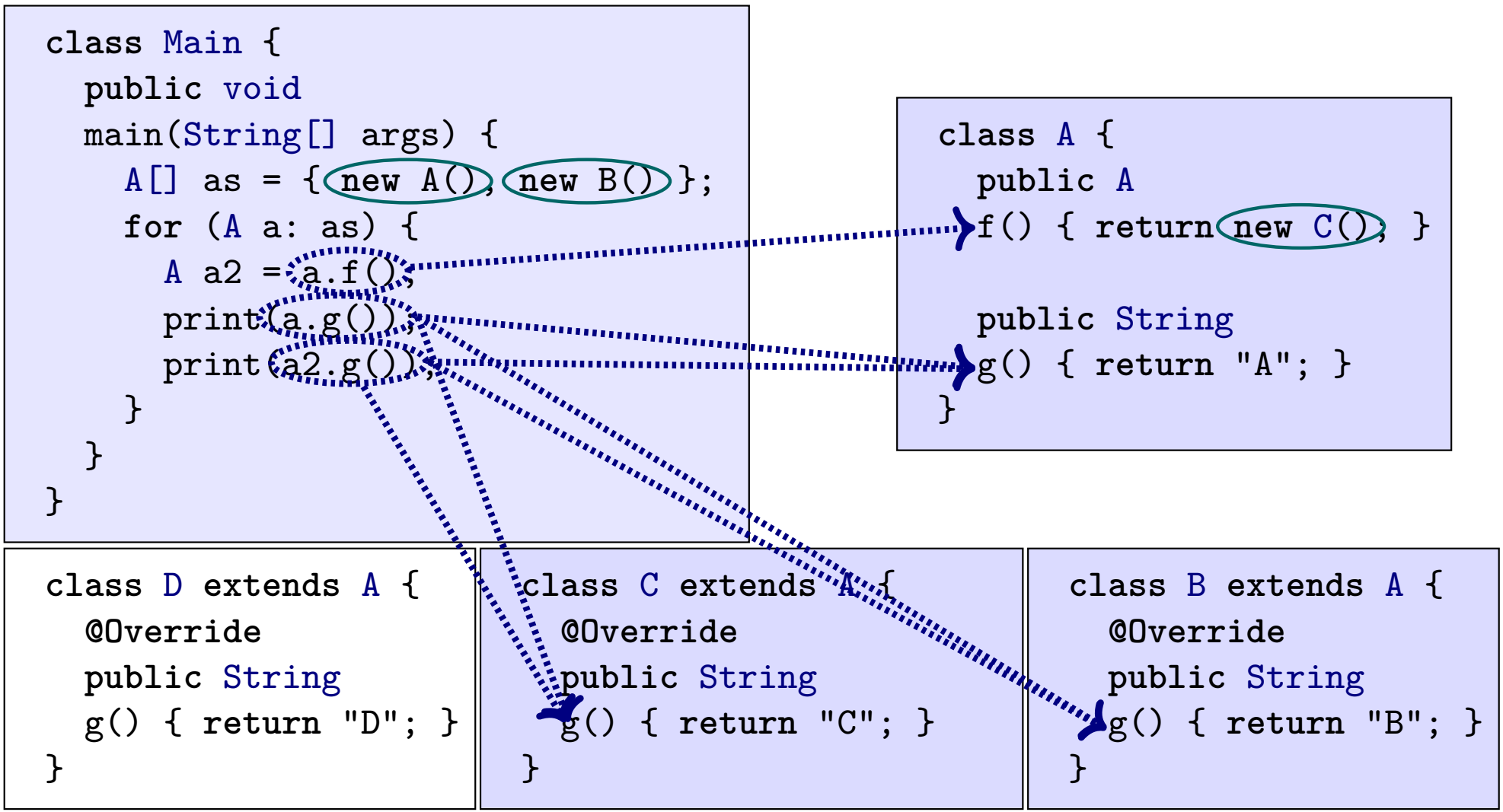
```
class A {
  public A
  f() { return new C(); }

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

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

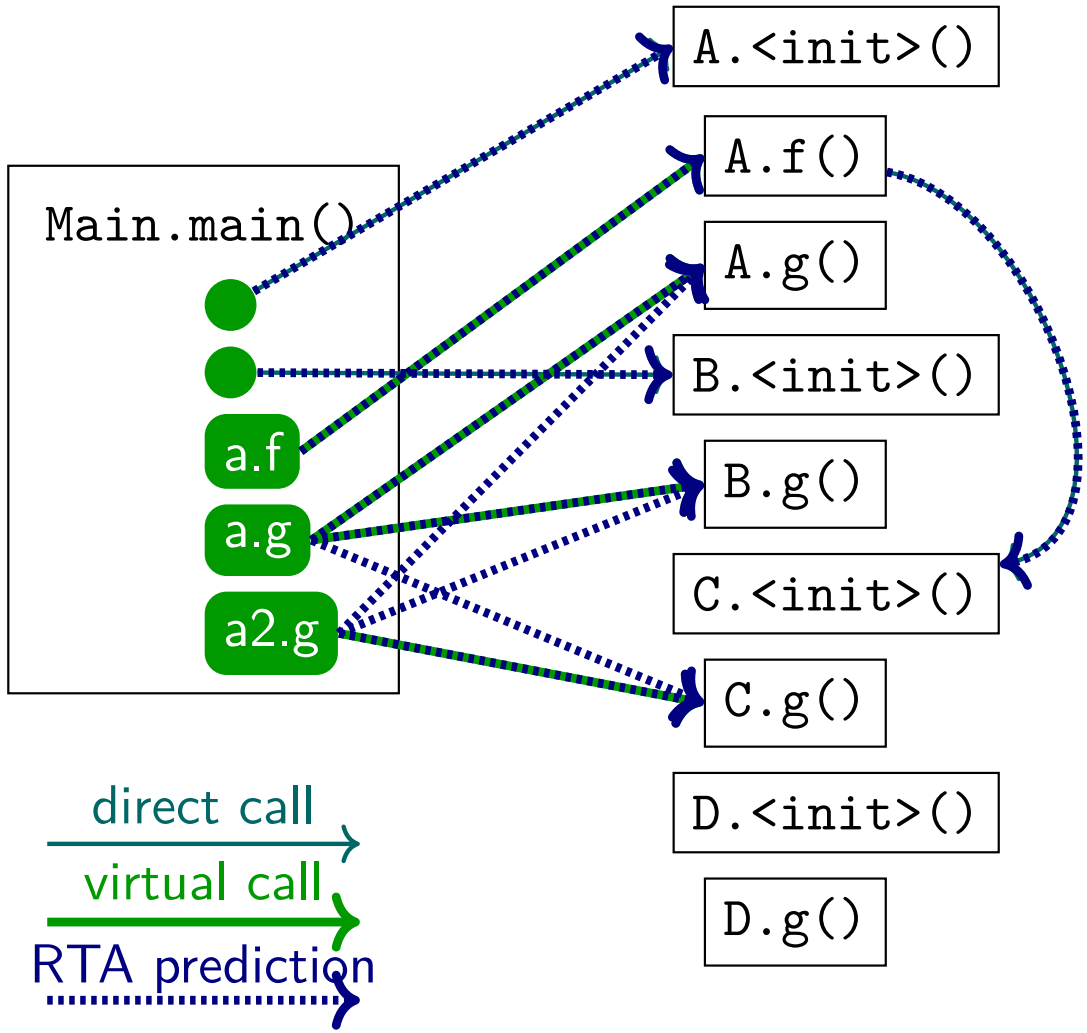
```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

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

Finding Calls and Targets



Rapid Type Analysis: Example



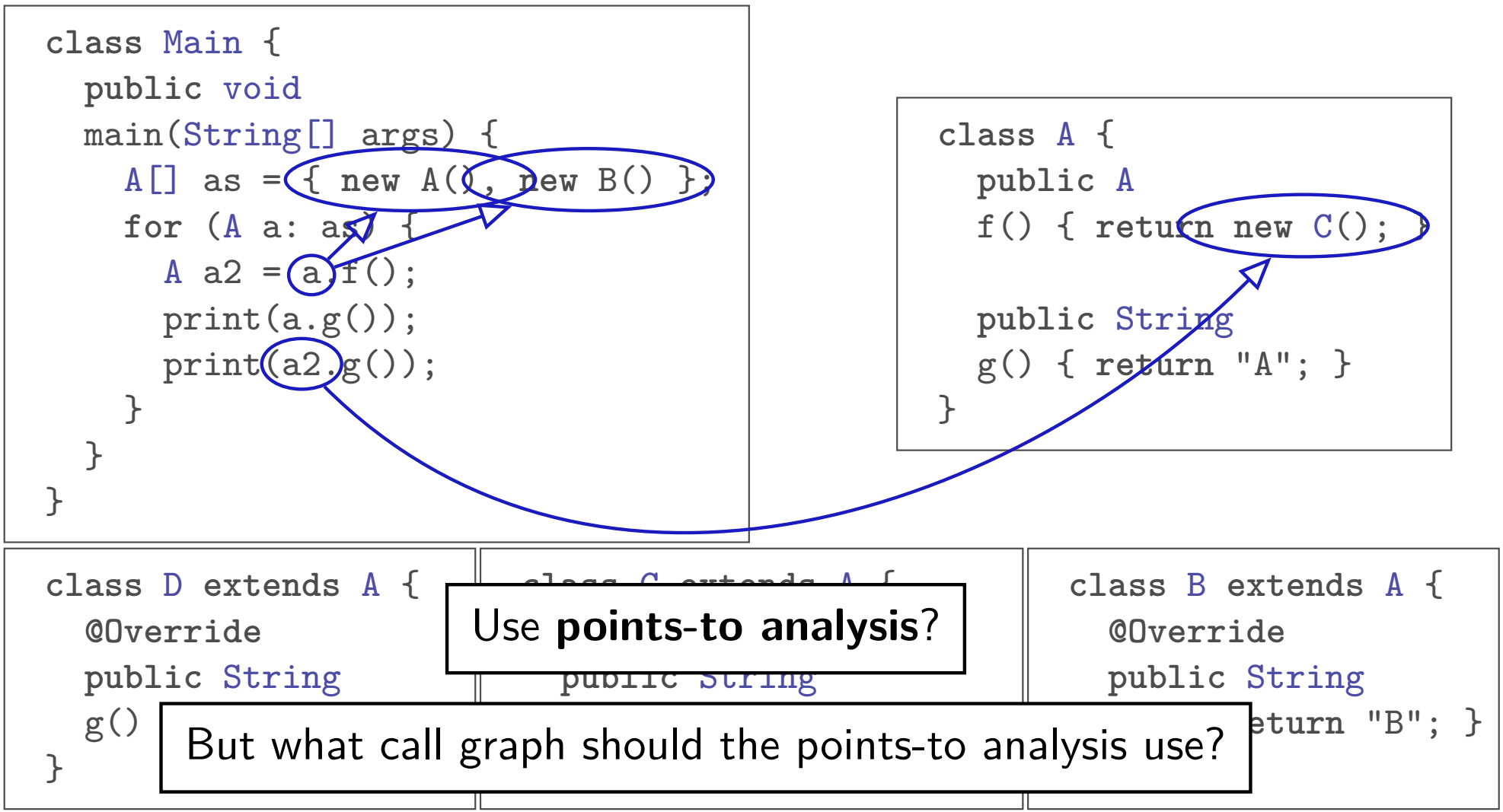
Rapid Type Analysis Algorithm Sketch

```
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
```

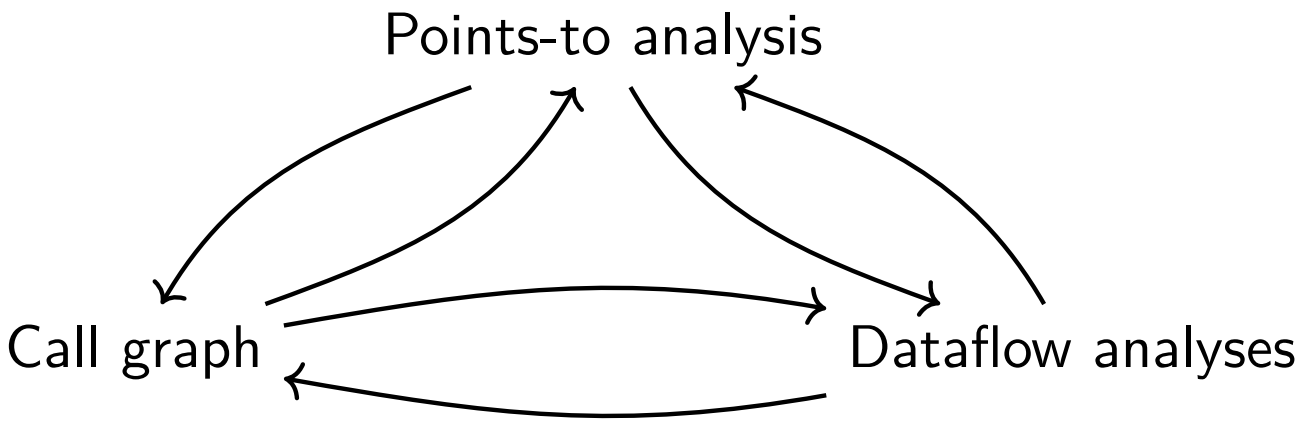
Summary

- **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



Dependencies



- Mutual dependencies across program analyses

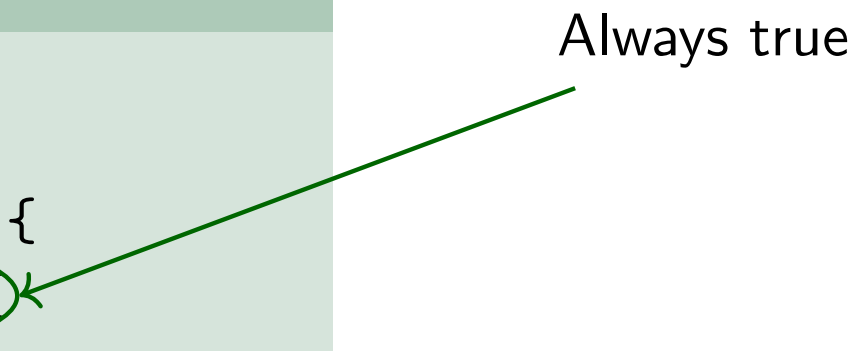
Analysis Composition

How do we handle mutual dependencies?

Analysis Composition: Example

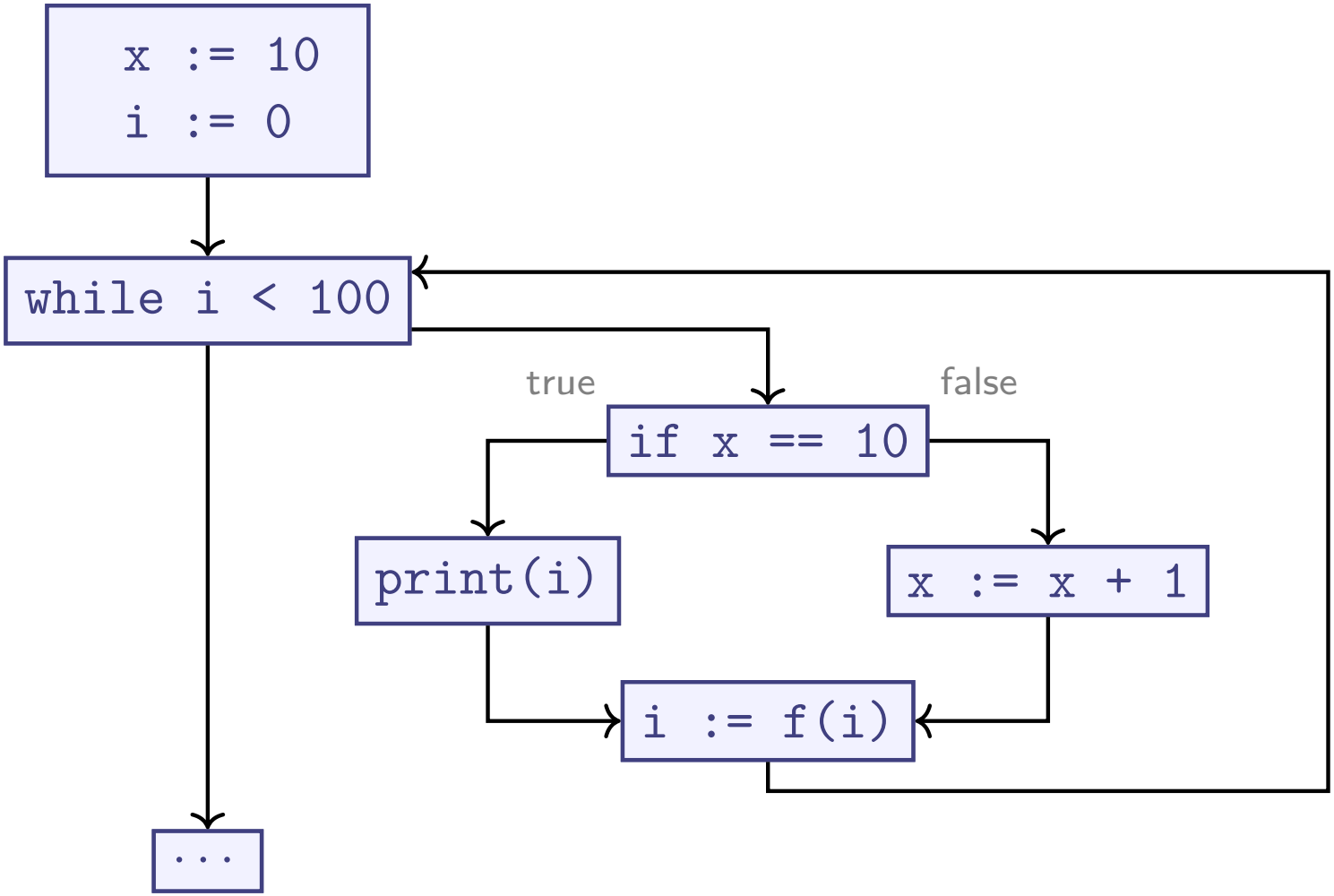
Teal

```
var x := 10;
var i := 0;
while i < 100 {
  if x == 10 {
    print(i);
  } else {
    x := x + 1;
  }
  i := f(i);
}
```



Adapted from Sorin Lerner, David Grove, Craig Chambers: "Composing Dataflow Analyses and Transformations", ACM SIGPLAN Conference on Principles of Programming langauges (POPL 2002)
Partly attributed to Mark N. Wegman and F. Kenneth Zadeck: "Constant Propagation with Conditional Branches", TOPLAS vol. 13(2), April 1991, 181–210

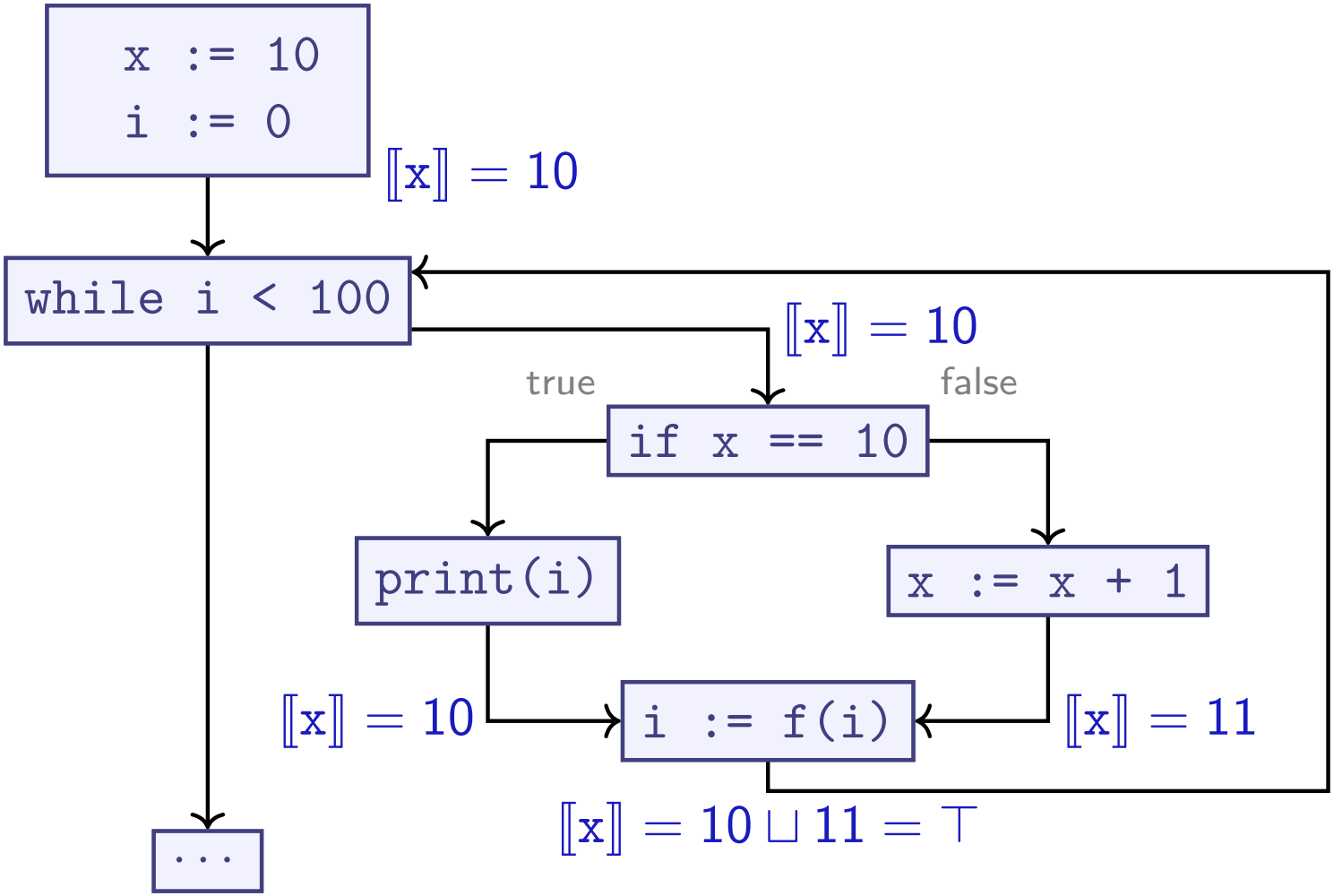
Analysis Composition: Loose (1/2)



- **First**: Unreachable Path Elimination
- **Second**: Constant Propagation / Constant Folding

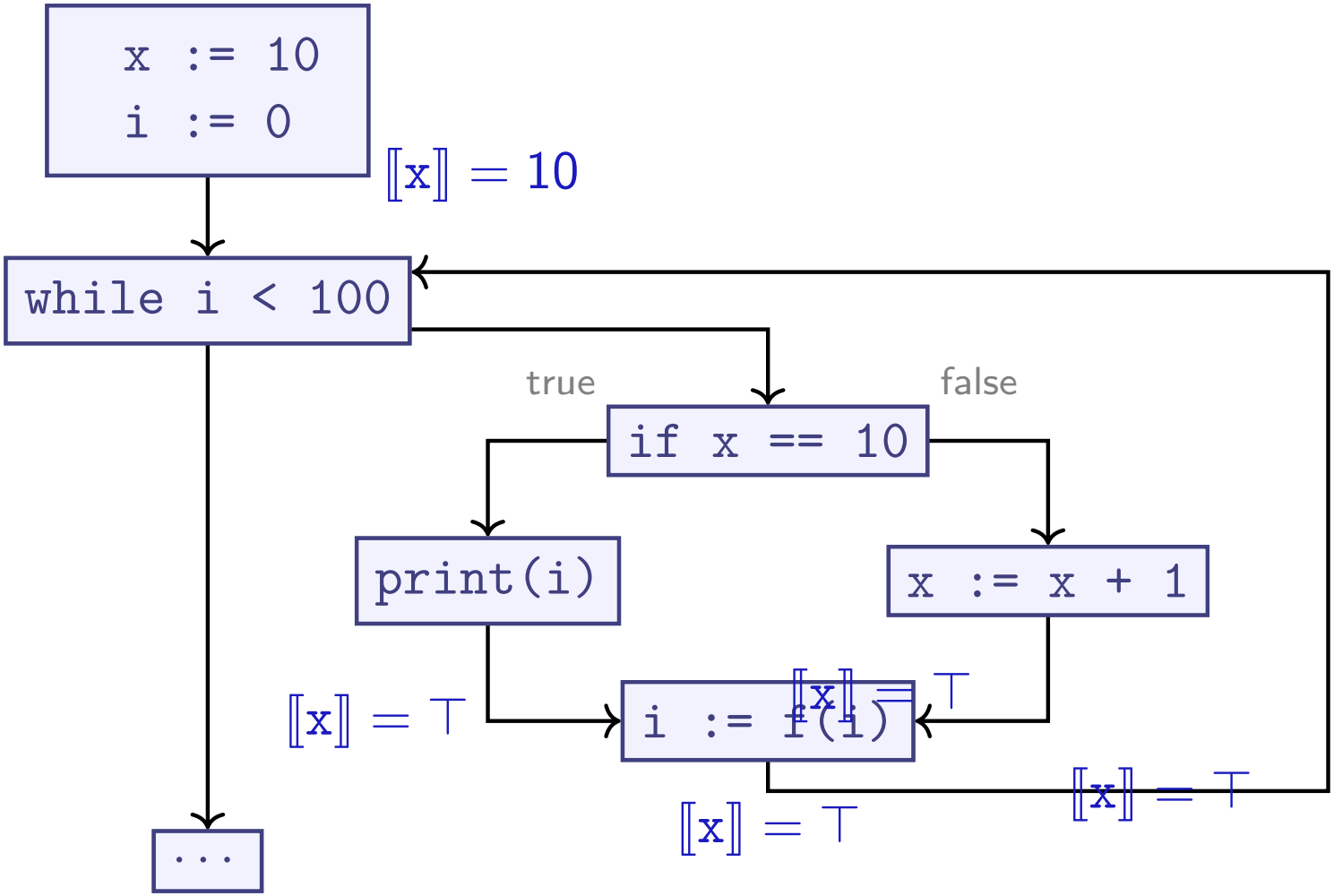
Unreachable Path Elimination can't evaluate any conditionals here

Analysis Composition: Loose (2/2)



- **First**: Constant Propagation / Constant Folding
- **Second**: Dead Path Elimination

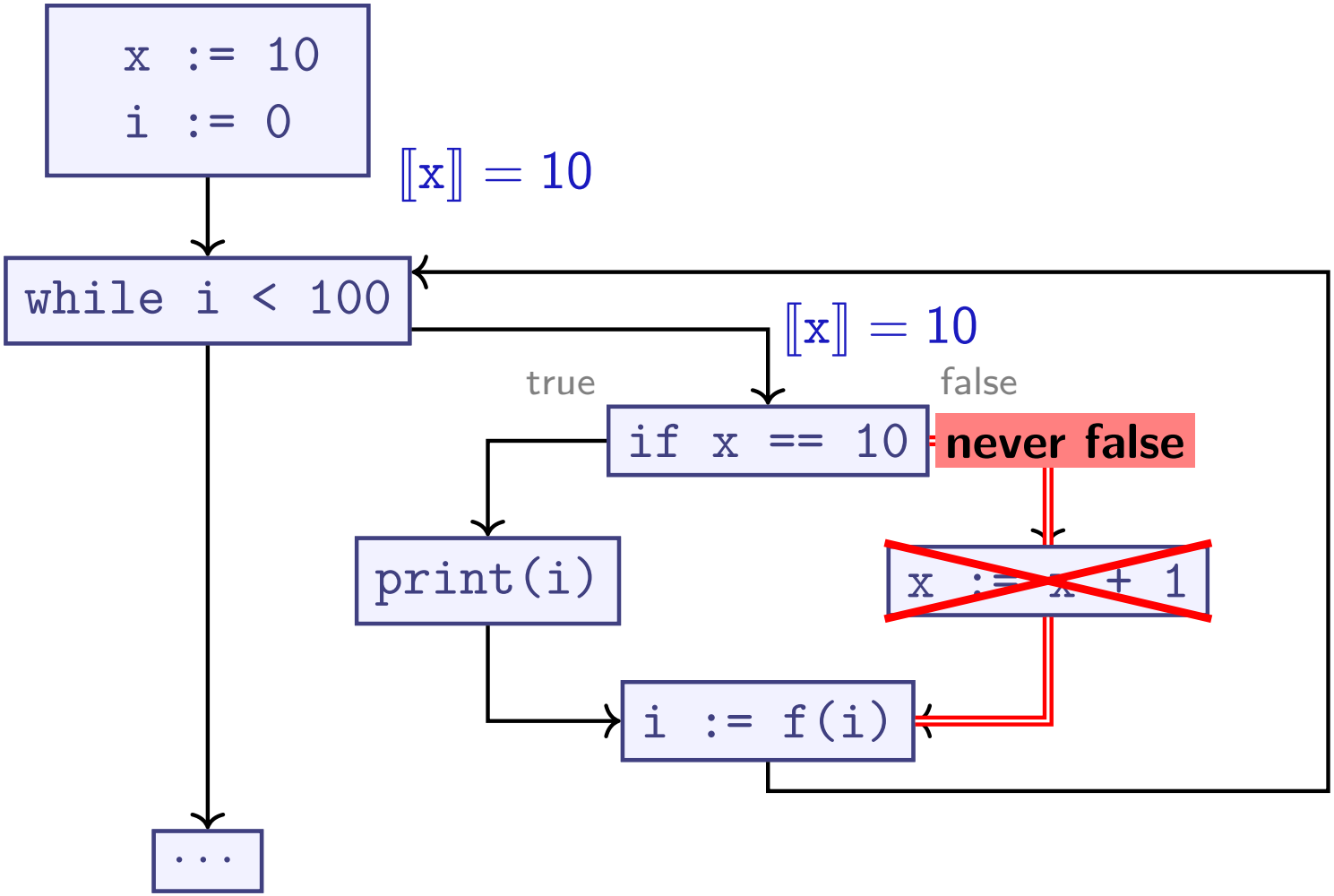
Analysis Composition: Loose (2/2)



- **First**: Constant Propagation / Constant Folding
- **Second**: Dead Path Elimination

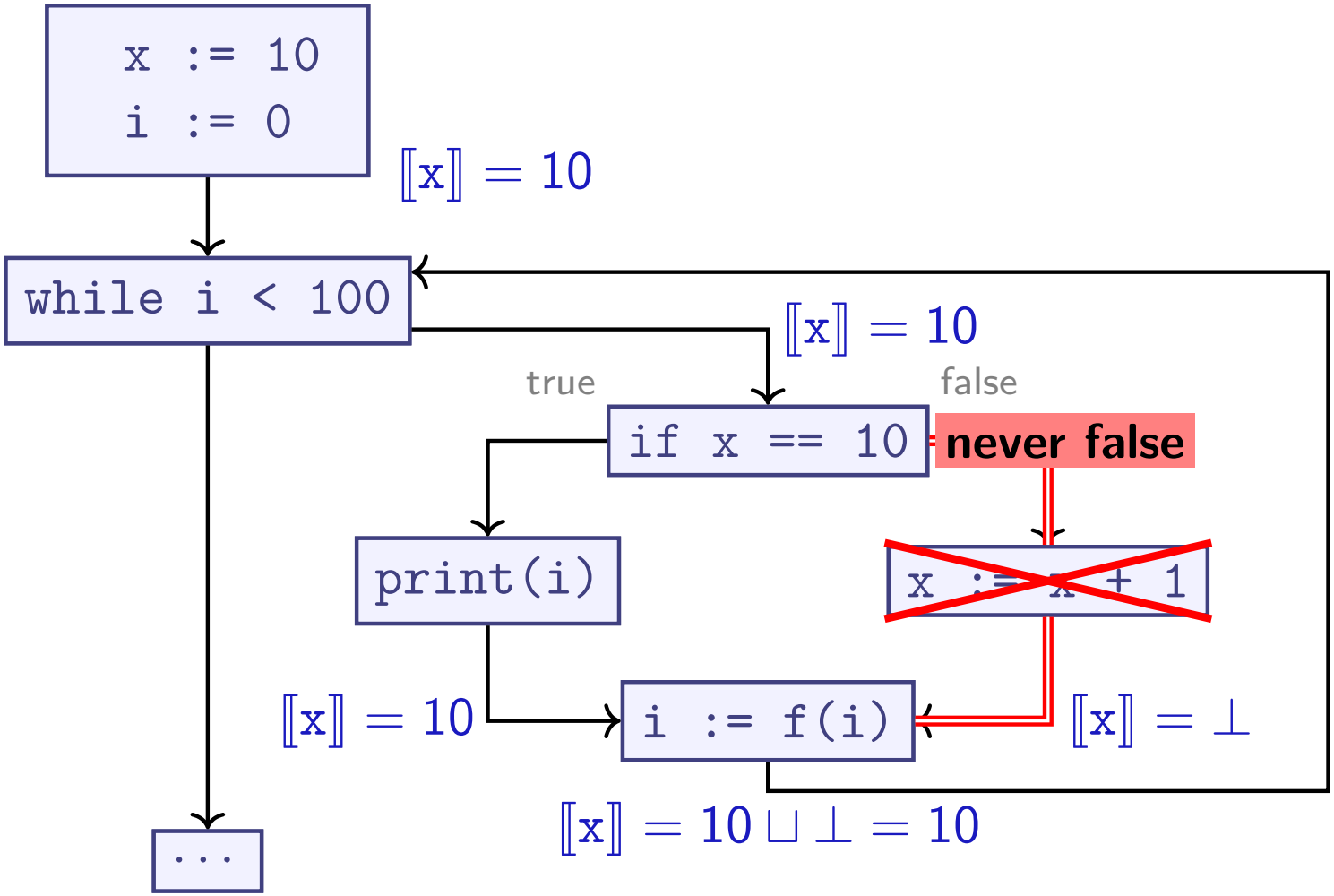
With $[\![x]\!] = \top$, Dead Path Elimination can't proceed

Analysis Composition: Tight



"Tight Composition": Run analyses *together*:
Constant Propagation / Folding & Dead Path Elimination

Analysis Composition: Tight



"Tight Composition": Run analyses *together*:

Constant Propagation / Folding & Dead Path Elimination

Executing at the same time gives correct result!

Loose Composition

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

- Each pass finishes before next pass starts
- Standard approach in compilers

Tight Composition

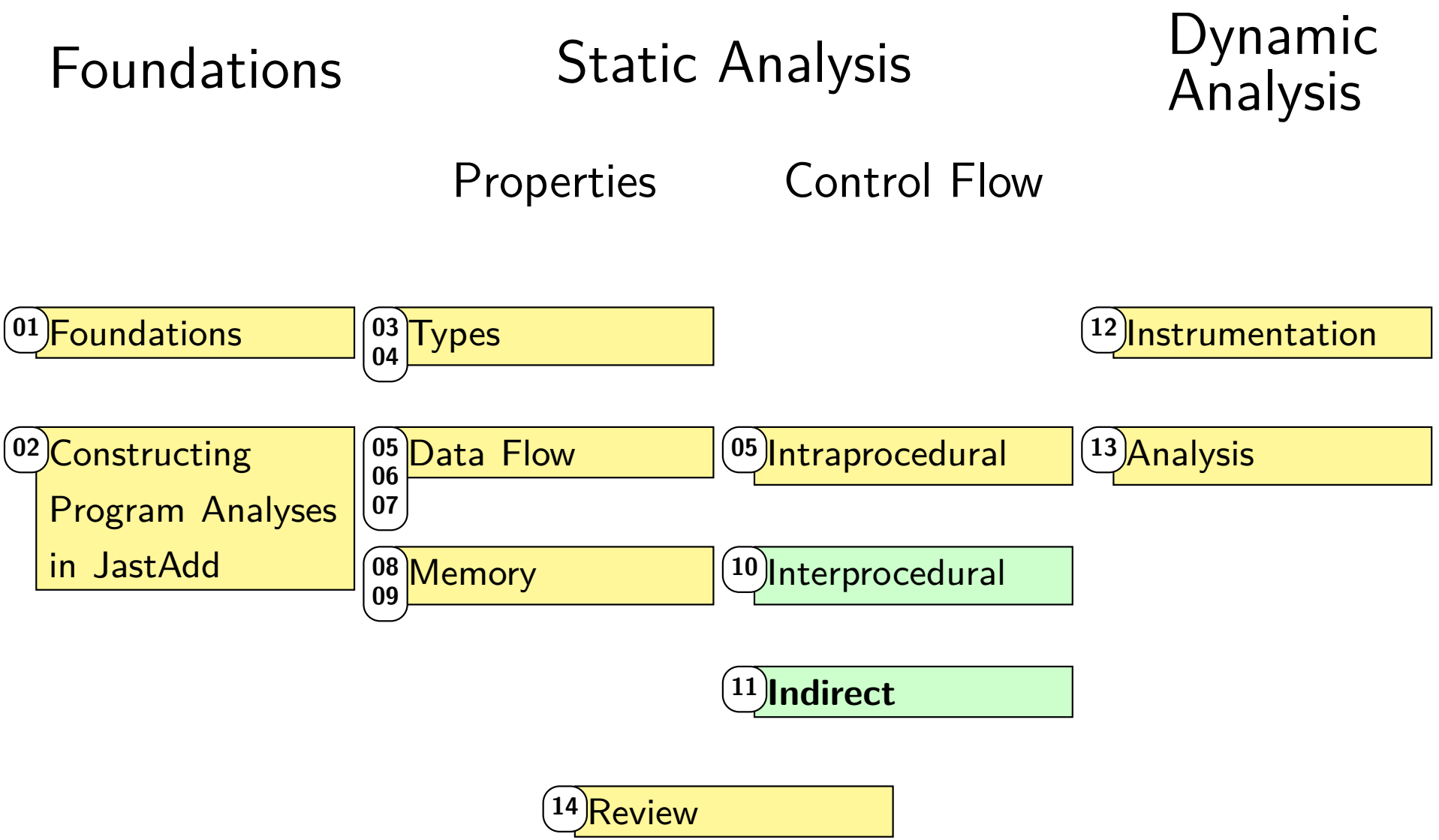
Tight Composition: **Analyses depend on each other's intermediate results**

- Analyses run "together"
- *Not widely supported*
- Systemic support:
  - Reference Attribute Grammars (JastAdd etc.) with circular attributes
  - Logic programming (Datalog, Prolog)
  - Term Rewriting (Vortex/Cyclone/)
- **Challenges**:
  - Traditional worklist algorithms:
    - Complex manual engineering needed
  - Declarative approaches (JastAdd, Logic Programming):
    - Must guarantee **Monotonicity**

Summary

- Mutual dependencies between program analyses are common
- Two approaches:
  - **Loose composition**:
    - One analysis after the other
    - May need to run analyses multiple times
    - Strictly less powerful than tight composition
  - **Tight composition**:
    - Analyses can use each other's intermediate results
    - Difficult to engineer for worklist algorithms
    - Easier with declarative approaches (attribute grammars, logic programming, term rewriting)
    - **Caveat**: Lattices must be "aligned": don't mix conservative + optimistic

Lecture Overview



Outlook

- Next lecture: Dynamic Program Analysis

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