EDAP15: Program Analysis

ADVANCED ANALYSES

Christoph Reichenbach

Andersen's Points-To Analysis

- Asymptotic performance is $O(n^3)$
- More precise than Steensgaard's analysis
- *Subset-based* (a.k.a. *inclusion-based*)
- $\implies$ Flow-sensitive but *directed*
- Popular as basis for current points-to analyses

L. Andersen, "Program Analysis and Specialization for the C Programming Language", PhD. thesis, DIKU report 94/19, 1994

Collecting Constraints

- Collect constraints, resolve as needed
- For each statement in program, we record:
  - If **Referencing** (`x := new`$_{\ell_i}$ `A()`):

$$\ell_i \in pts(\mathtt{x}) \qquad (x \rightarrowtriangle \ell_i)$$

  - If **Aliasing** (`x := y`):

$$pts(\mathtt{x}) \supseteq pts(\mathtt{y})$$

  - If **Dereferencing read** (`x := y.`$\square$):

$$pts(\mathtt{x}) \supseteq pts(\mathtt{y}.\square)$$

  - If **Dereferencing write** (`x.`$\square$ `:= y`):

$$pts(\mathtt{x}.\square) \supseteq pts(\mathtt{y})$$

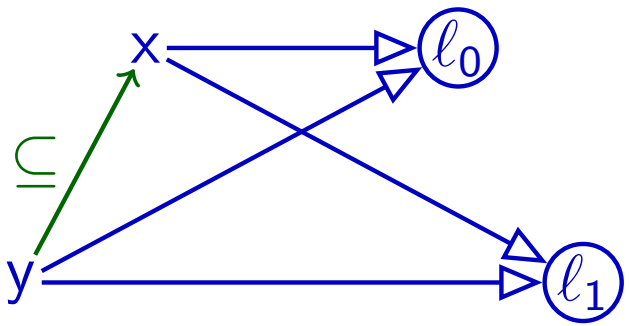
Solving Constraints

1. Fact extraction:
   - Initial points-to sets: $\ell \in pts(x)$, meaning $\ell \triangleleft\!\!- x$
   - Constraints:
     - $pts(x) \supseteq pts(y)$
     - $pts(x) \supseteq pts(y.\square)$
     - $pts(x.\square) \supseteq pts(y)$

Subset Constraints (1/2)

- Solving $pts(x) \supseteq pts(y)$

```
y := new_{ℓ0}();
while ...  {
  x := y;
  y := new_{ℓ1}();

}
```

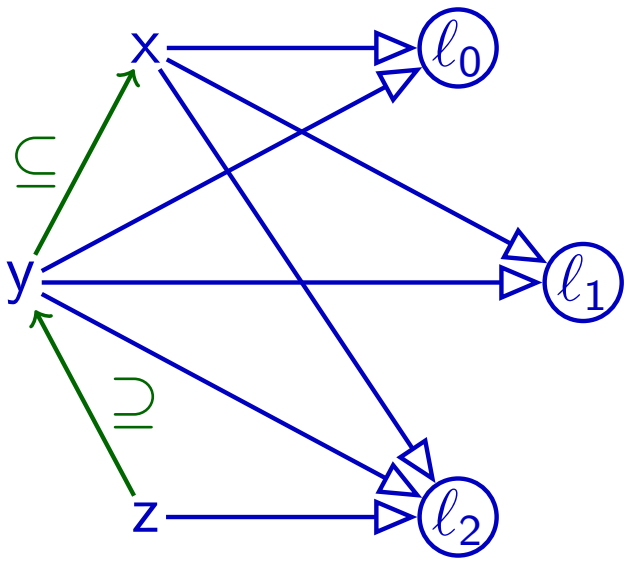


- $\ell \leftarrow y$ and $pts(x) \supseteq pts(y)$:
  $\Longrightarrow \ell \leftarrow x$
- *Flow insensitive*: can't distinguish before/after

Subset Constraints (1/2)

- Solving $pts(x) \supseteq pts(y)$

```
y := new_ℓ0();
while ...  {
  x := y;
  y := new_ℓ1();
  z := new_ℓ2();
  if ...  {
    y := z;
  }
}
```



- $\ell \leftarrow y$ and $pts(x) \supseteq pts(y)$:
  $\Longrightarrow \ell \leftarrow x$
- *Flow insensitive*: can't distinguish before/after

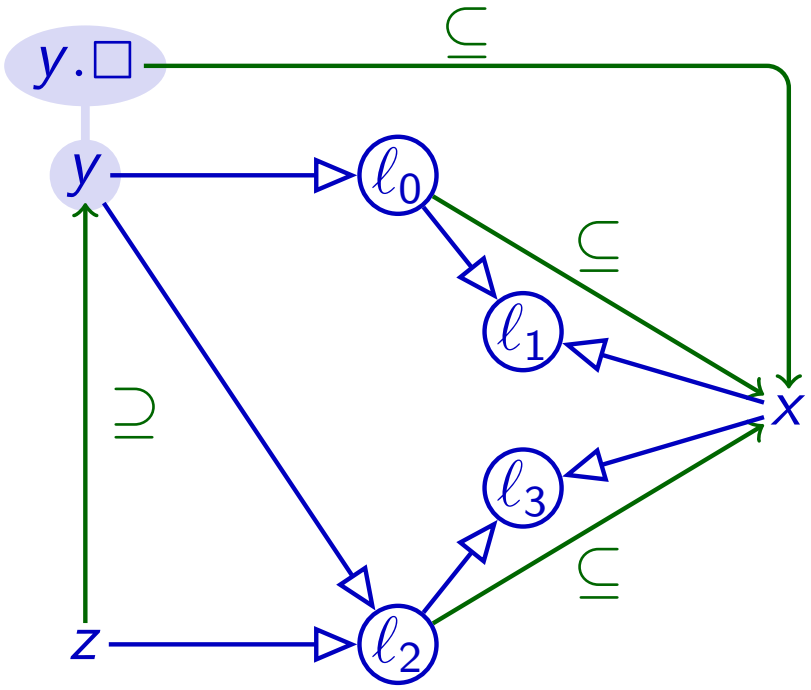
Solving one ($\supseteq$) can depend on all ($\leftarrow$) and ($\supseteq$) in program

Subset Constraints (2/2)

- Solving $pts(x) \supseteq pts(y.\square)$

```
y := new_ℓ0();
y.n := new_ℓ1();
z := new_ℓ2();
z.n := new_ℓ3();
if ...  {
  y := z;
}
x := y.n;
```

Simplified presentation (omitting ($\supseteq$) constraints)



- Recall:
  $\ell \leftarrow z$ and $pts(y) \supseteq pts(z)$:
  $\Longrightarrow \ell \leftarrow y$
- $\ell \leftarrow y$ and $pts(x) \supseteq pts(y.\square)$:
  $\Longrightarrow x \supseteq \ell$

Fresh Assignments to Fields

- Recall:
  `y.n := new`$_{\ell_1}$`();`
- No direct pattern for this code
- Can model as:
  `var tmp := new`$_{\ell_1}$`();`
  `y.n := tmp;`

Solving Constraints

1. Fact extraction:
   - Initial points-to sets: $\ell \in pts(x)$, meaning $\ell \triangleleft\!\!\leftarrow x$
   - Constraints:
     - $pts(x) \supseteq pts(y)$
     - $pts(x) \supseteq pts(y.\square)$
     - $pts(x.\square) \supseteq pts(y)$
2. Build directed *inclusion graph* $G_I = \langle MemLoc, E \rangle$
   - $x \leftarrow y$ represents $pts(x) \supseteq pts(y)$ ("x:=y")
3. Expand and propagate along inclusion graph:
   - Propagate points-to sets along E:
     - $\ell \triangleleft\!\!\leftarrow y$ and $x \leftarrow y$:
       $\Longrightarrow \ell \triangleleft\!\!\leftarrow x$
     - $v \triangleleft\!\!\leftarrow y$ and $x \leftarrow y.\square$:
       $\Longrightarrow x \leftarrow v$
     - $v \triangleleft\!\!\leftarrow x$ and $x.\square \leftarrow y$:
       $\Longrightarrow v \leftarrow y$

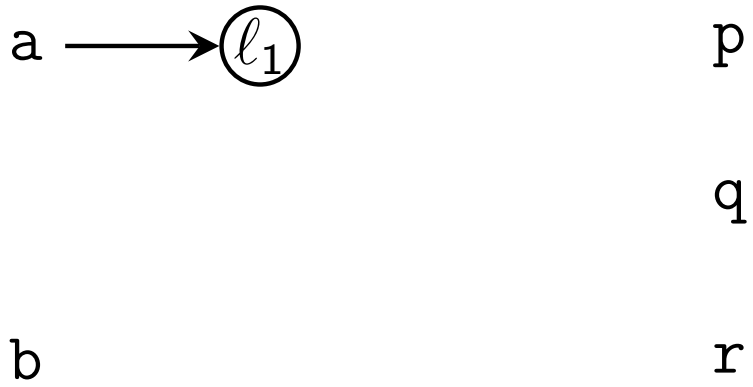
Example

⇒ `x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.`$\square$	$x \leftarrow y.\square$
`x.`$\square$ `:= y`	$x.\square \leftarrow y$

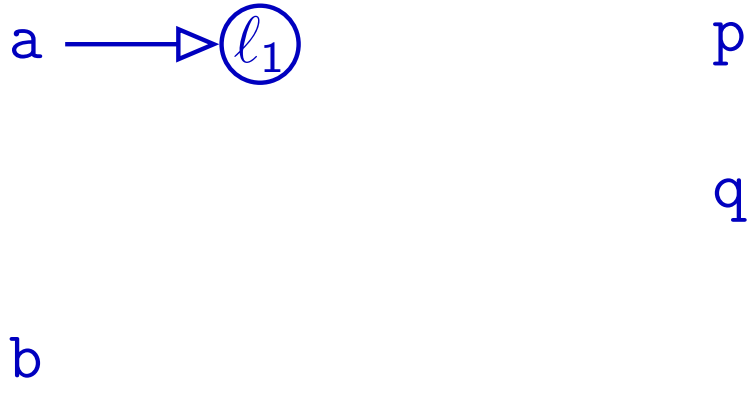
Teal

```
var a := new_ℓ1(); //⇐
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

- **Actual**:



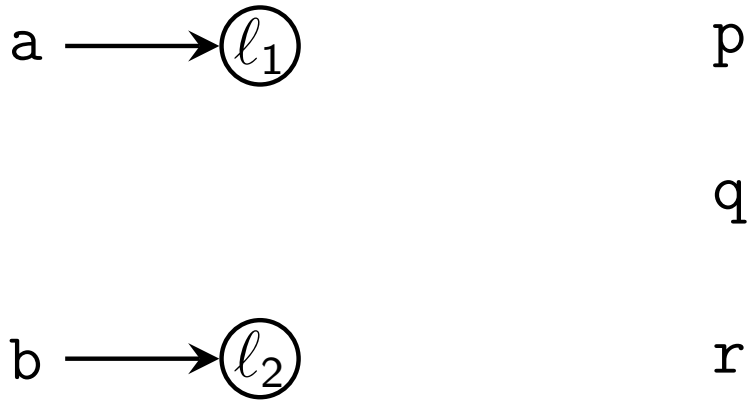
- **Andersen**:



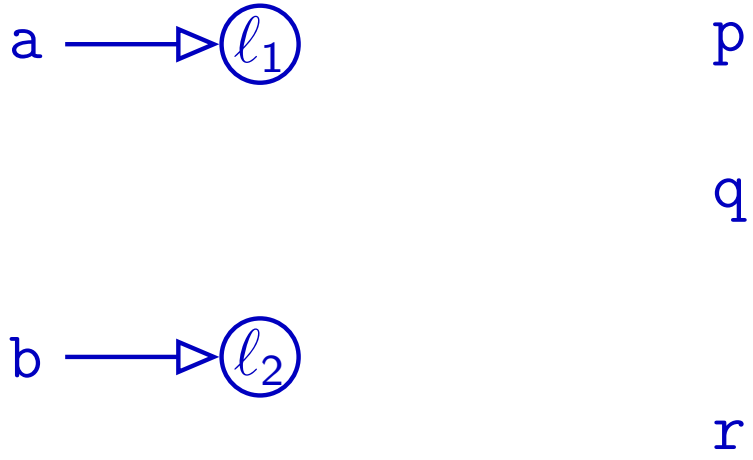
Example

⇒ `x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.□`	$x \leftarrow y.\square$
`x.□ := y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



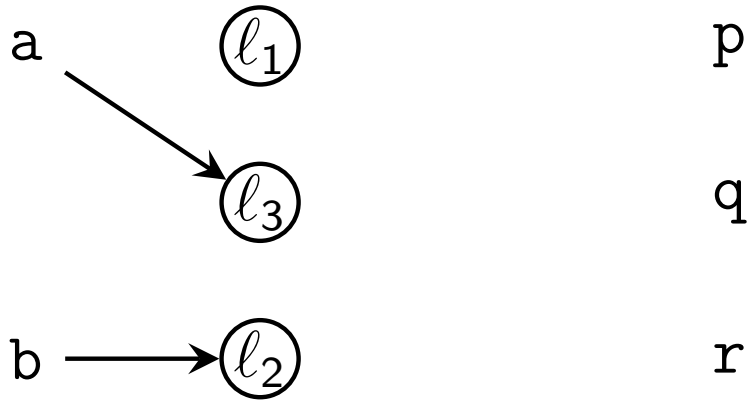
Teal

```
var a := new_{ℓ1}();
var b := new_{ℓ2}(); //⇐
a := new_{ℓ3}();
var p := new_{ℓ4}();
p.n := a;
var q := new_{ℓ6}();
q.n := b;
p := q;
var r := q.n;
```

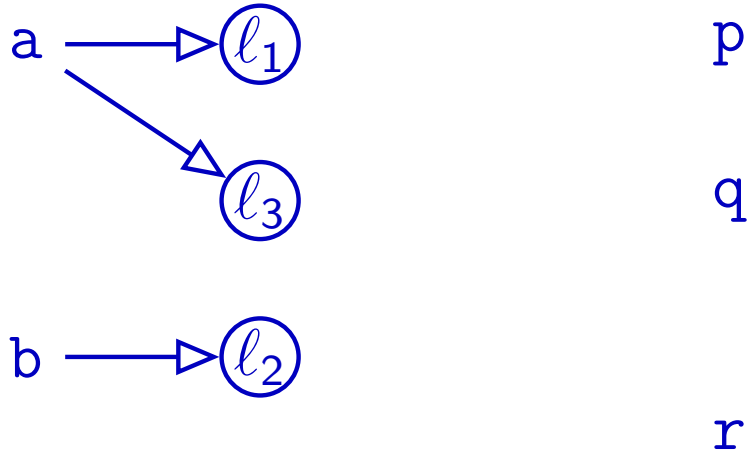
Example

⇒ `x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.□`	$x \leftarrow y.\square$
`x.□ := y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



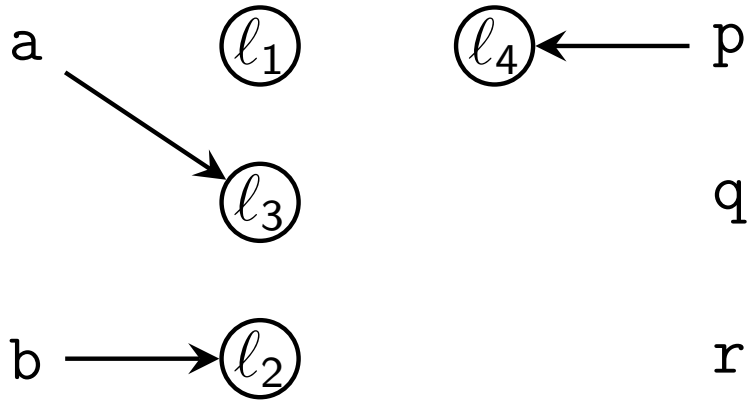
Teal

```
var a := new_{ℓ1}();
var b := new_{ℓ2}();
a := new_{ℓ3}();        //⇐
var p := new_{ℓ4}();
p.n := a;
var q := new_{ℓ6}();
q.n := b;
p := q;
var r := q.n;
```

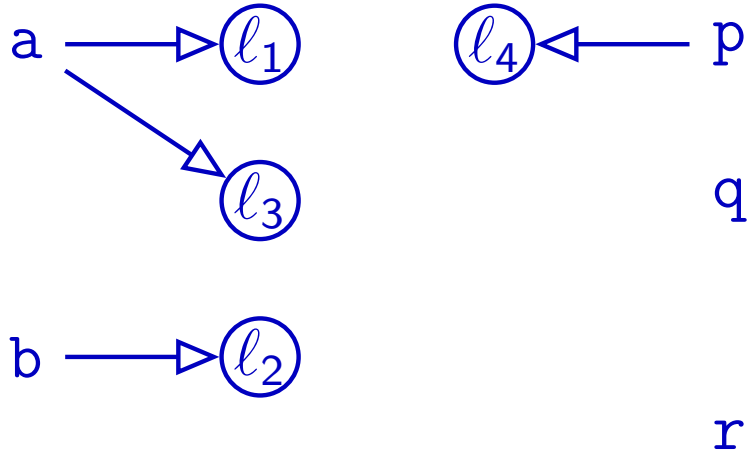
Example

⇒ `x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.□`	$x \leftarrow y.\square$
`x.□ := y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



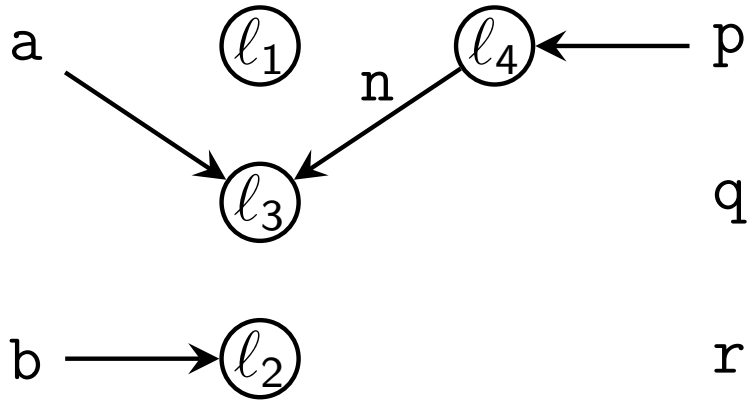
Teal

```
var a := new_{ℓ1}();
var b := new_{ℓ2}();
a := new_{ℓ3}();
var p := new_{ℓ4}(); //⇐
p.n := a;
var q := new_{ℓ6}();
q.n := b;
p := q;
var r := q.n;
```

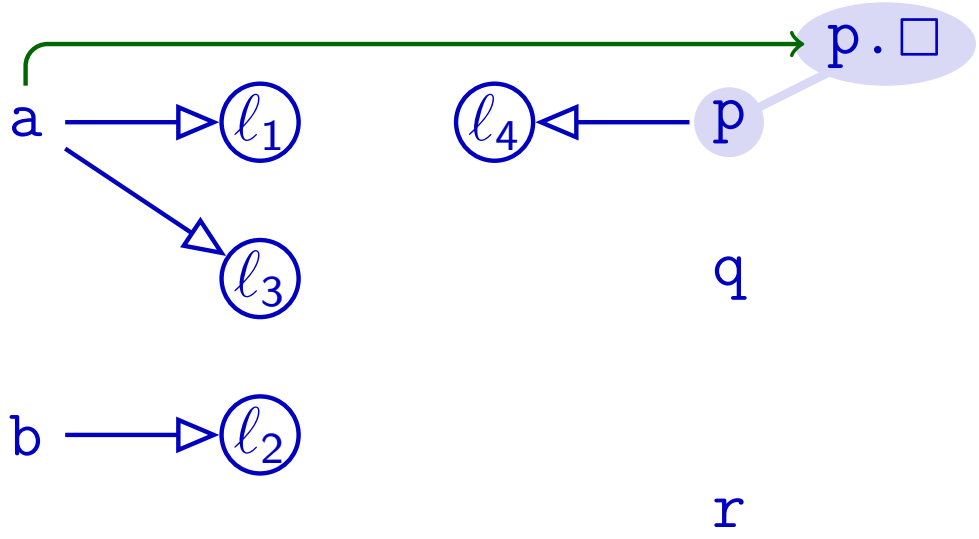
Example

x := new$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
x := y	$x \leftarrow y$
x := y.□	$x \leftarrow y.\square$
⇒ x.□ := y	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



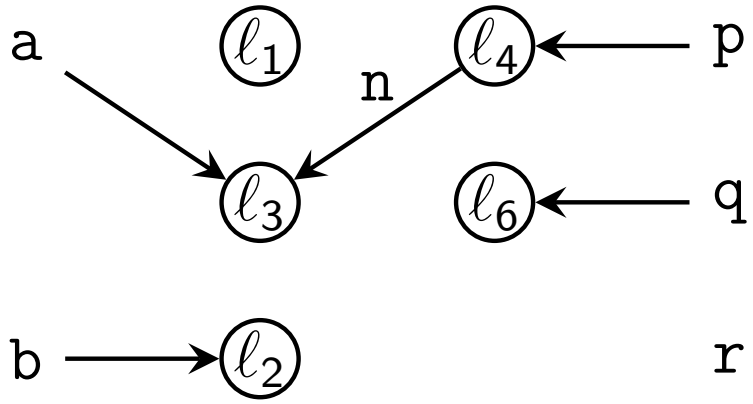
Teal

```
var a := new_{ℓ1}();
var b := new_{ℓ2}();
a := new_{ℓ3}();
var p := new_{ℓ4}();
p.n := a;          //⇐
var q := new_{ℓ6}();
q.n := b;
p := q;
var r := q.n;
```

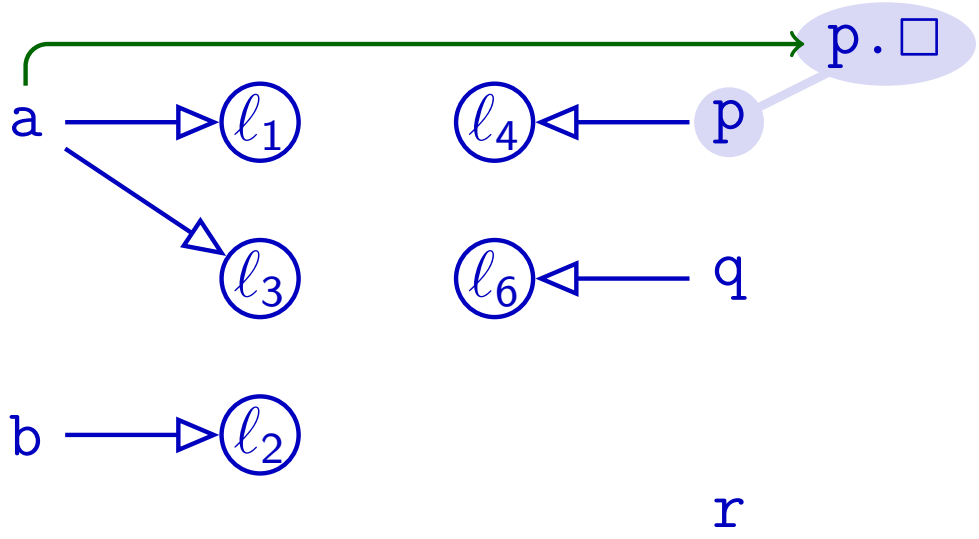
Example

⇒ `x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.□`	$x \leftarrow y.\square$
`x.□ := y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



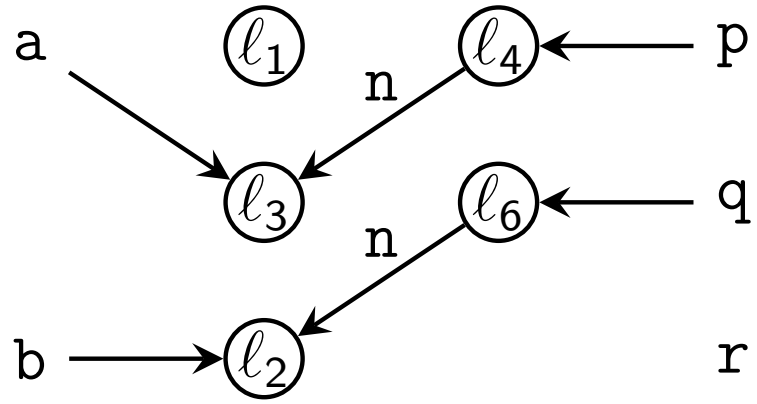
Teal

```
var a := new_{ℓ1}();
var b := new_{ℓ2}();
a := new_{ℓ3}();
var p := new_{ℓ4}();
p.n := a;
var q := new_{ℓ6}(); //⇐
q.n := b;
p := q;
var r := q.n;
```

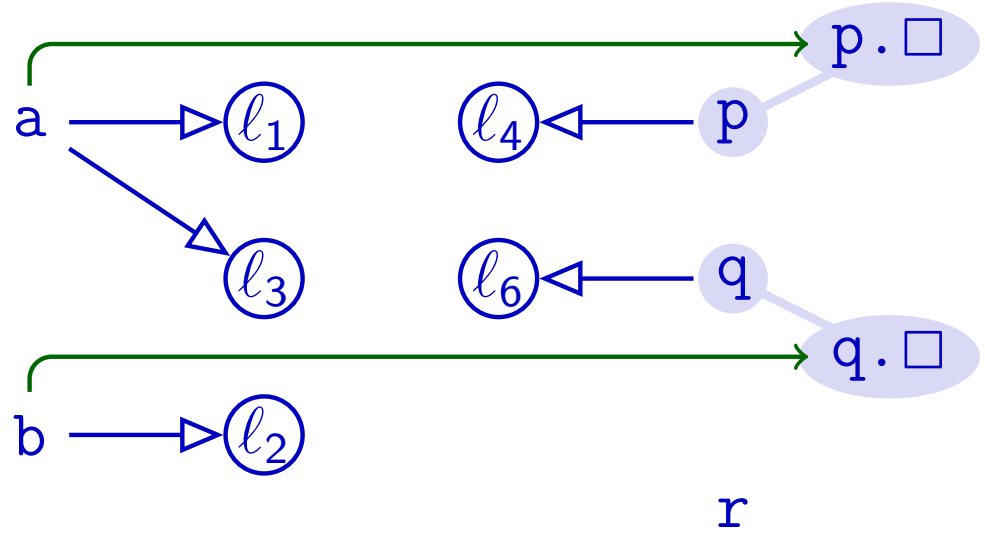
Example

x := new$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
x := y	$x \leftarrow y$
x := y.□	$x \leftarrow y.\square$
⇒ x.□ := y	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



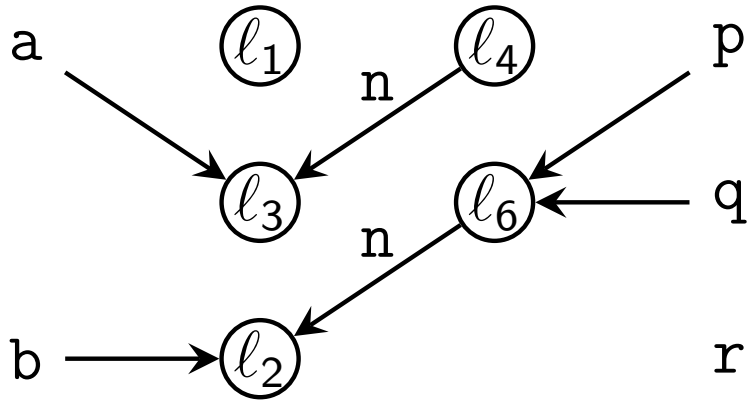
Teal

```
var a := new_{ℓ1}();
var b := new_{ℓ2}();
a := new_{ℓ3}();
var p := new_{ℓ4}();
p.n := a;
var q := new_{ℓ6}();
q.n := b;          //⇐
p := q;
var r := q.n;
```

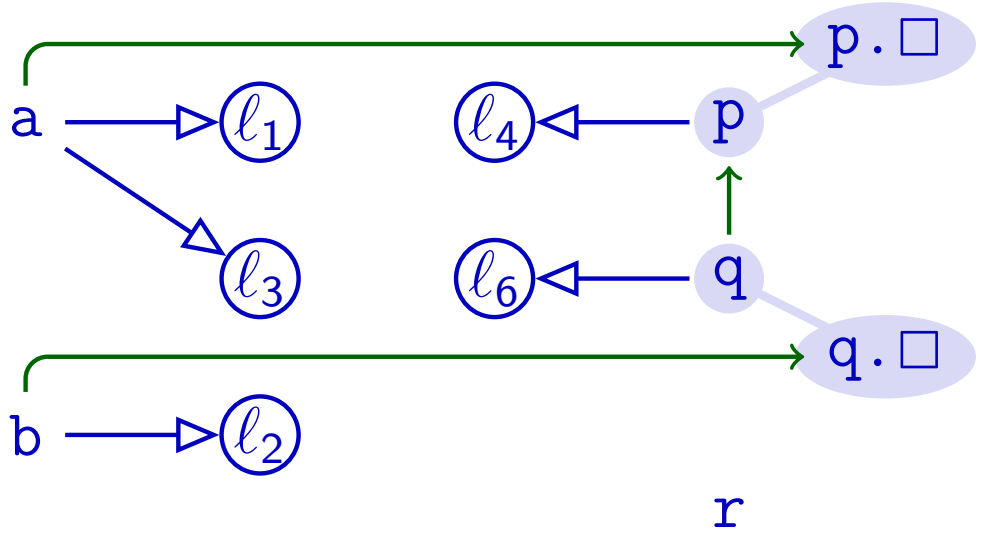
Example

```
   x := new_{ℓz}     x→▷ℓz
⇒  x := y           x←y
   x := y.□         x←y.□
   x.□ := y         x.□←y
```

- **Actual**:



- **Andersen**:



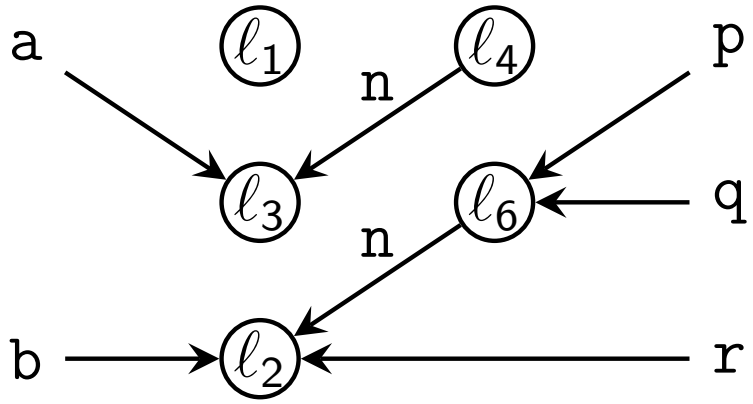
Teal

```
var a := new_{ℓ1}();
var b := new_{ℓ2}();
a := new_{ℓ3}();
var p := new_{ℓ4}();
p.n := a;
var q := new_{ℓ6}();
q.n := b;
p := q;              //⇐
var r := q.n;
```

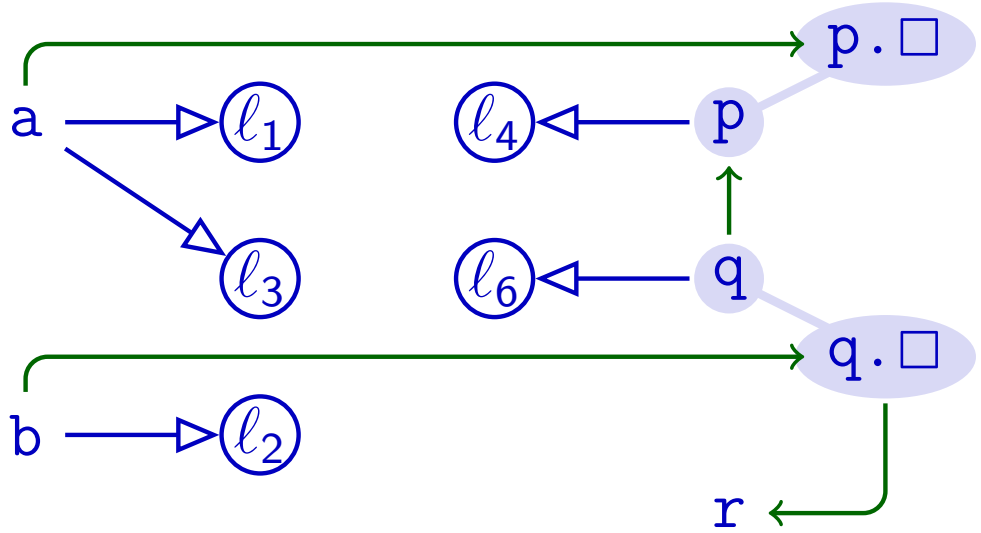
Example

`x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.`$\square$	$x \leftarrow y.\square$
`x.`$\square$ `:= y`	$x.\square \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

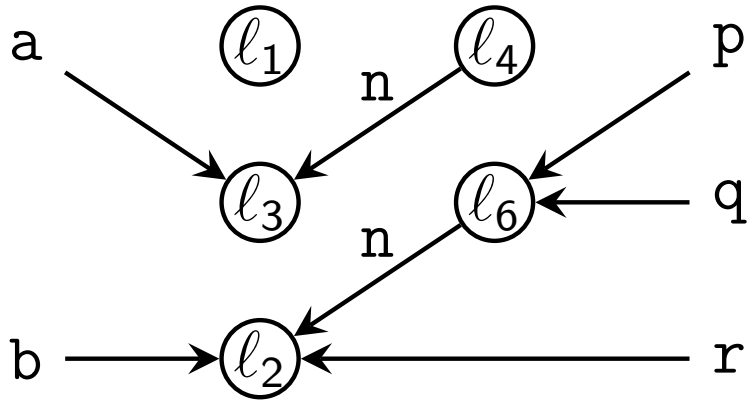
```
var a := new_{ℓ1}();
var b := new_{ℓ2}();
a := new_{ℓ3}();
var p := new_{ℓ4}();
p.n := a;
var q := new_{ℓ6}();
q.n := b;
p := q;
var r := q.n;      //⇐
```

Example

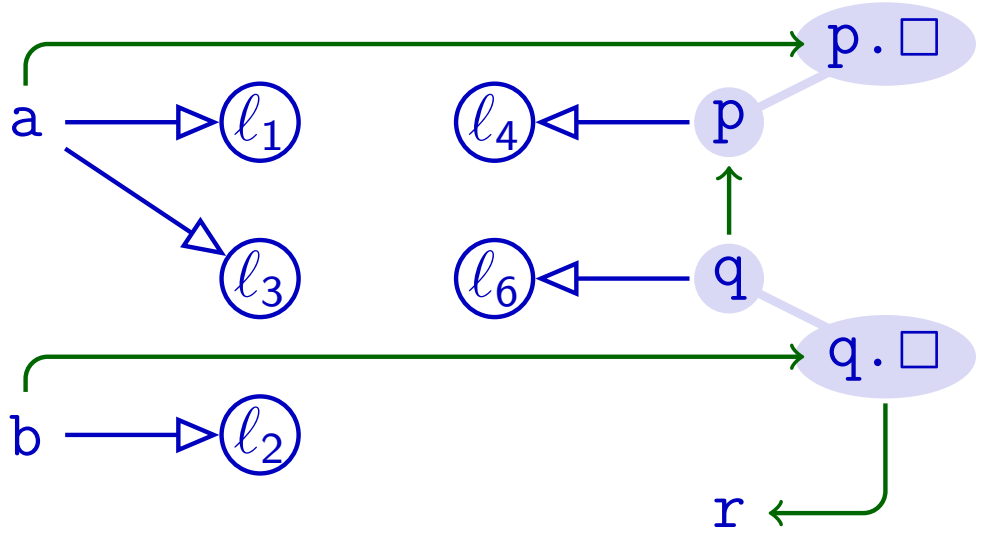
`x := new`$_{\ell_z}$	$x \rightarrowtriangle \ell_z$
`x := y`	$x \leftarrow y$
`x := y.`$\square$	$x \leftarrow y.\square$
`x.`$\square$ `:= y`	$x.\square \leftarrow y$

$\ell \leftarrowtriangle y$ and $x \leftarrow y \implies \ell \leftarrowtriangle x$
$v \leftarrowtriangle y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrowtriangle x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

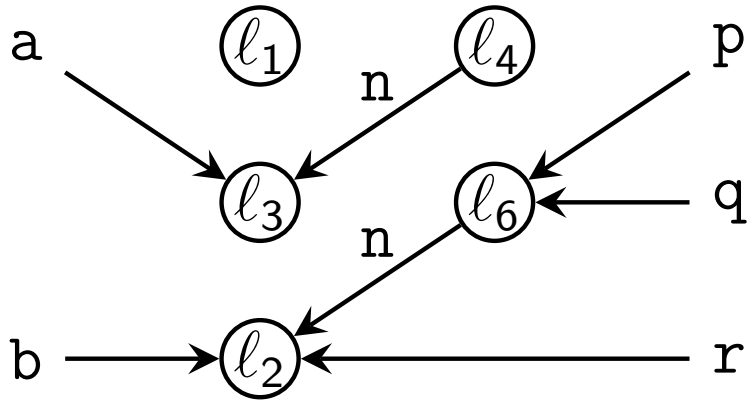
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

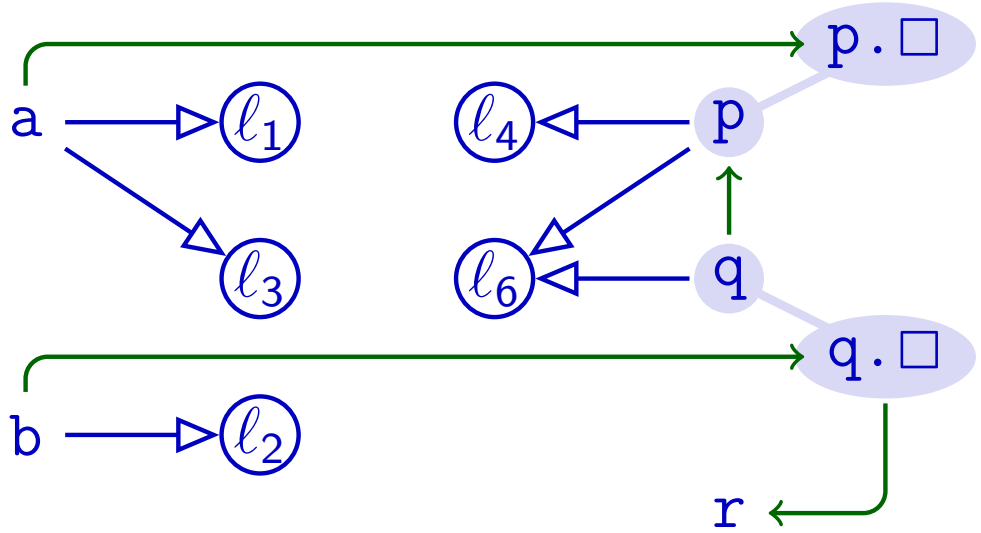
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$v \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrow x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

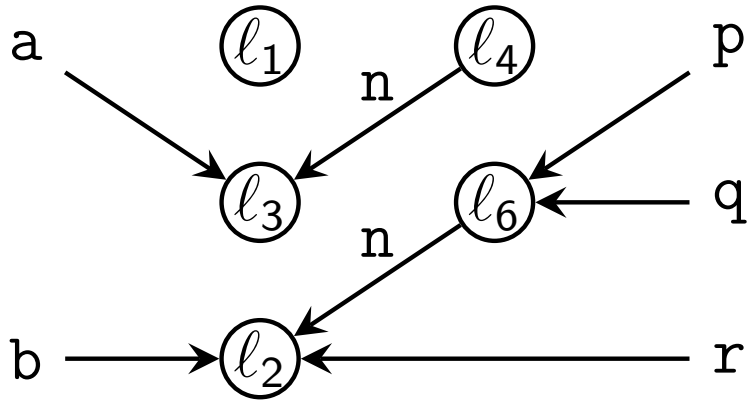
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

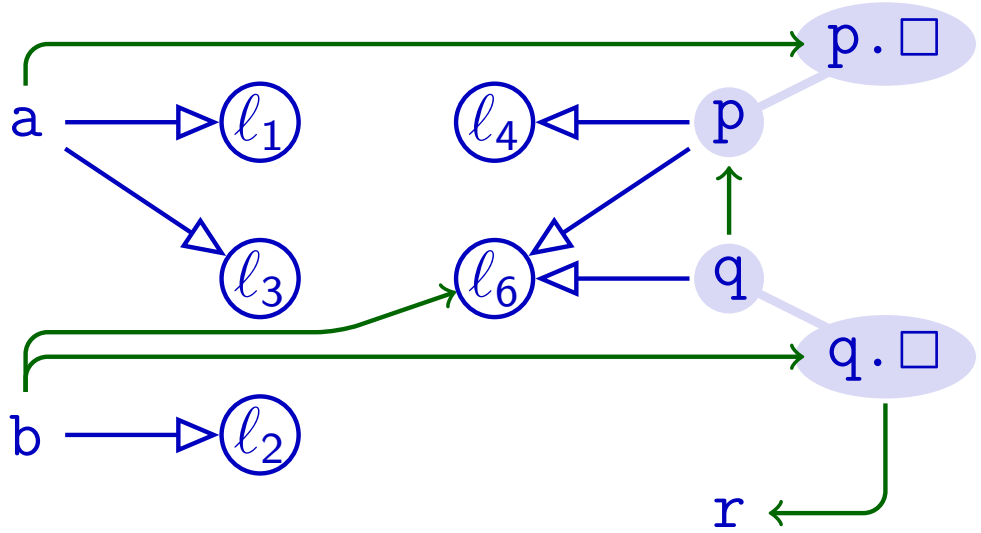
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$v \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrow x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

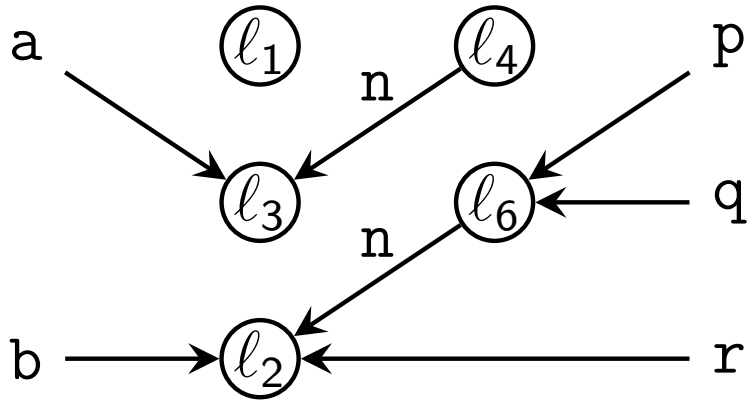
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

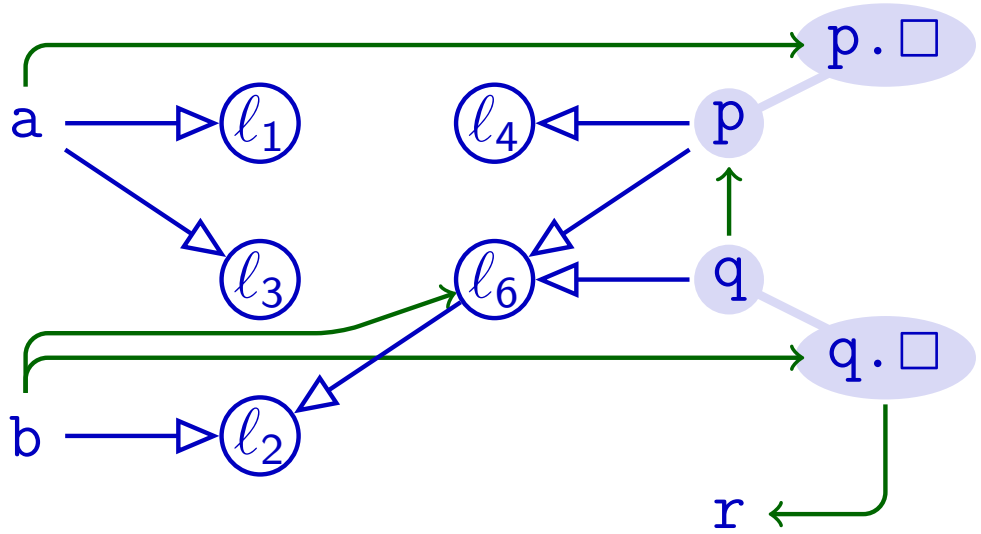
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$v \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrow x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

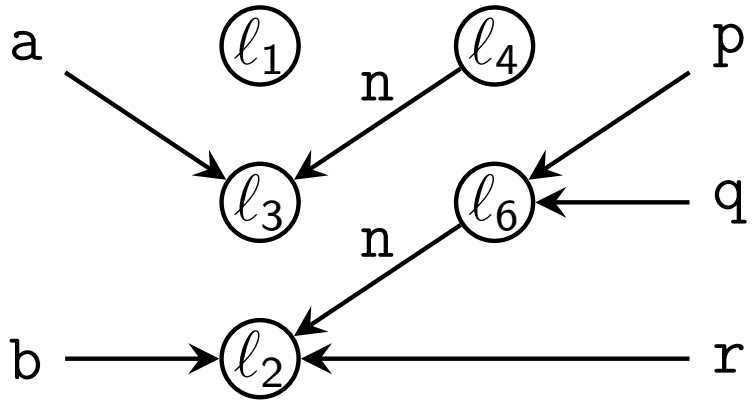
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

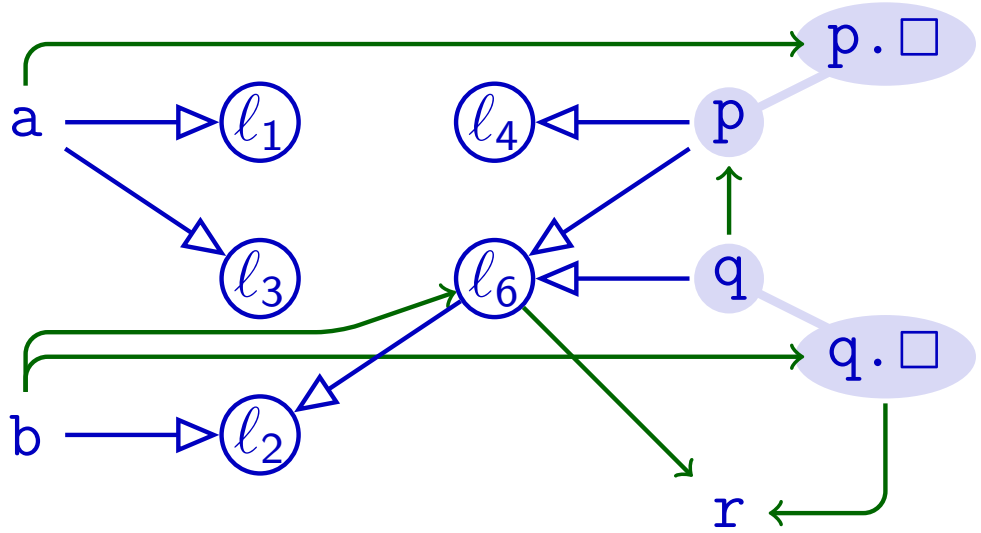
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$v \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrow x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

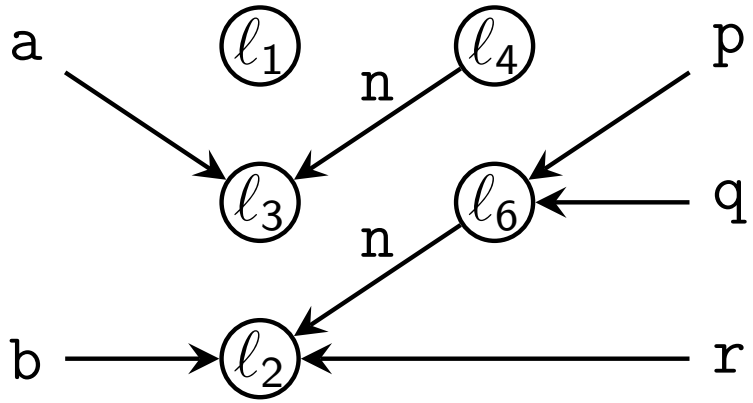
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

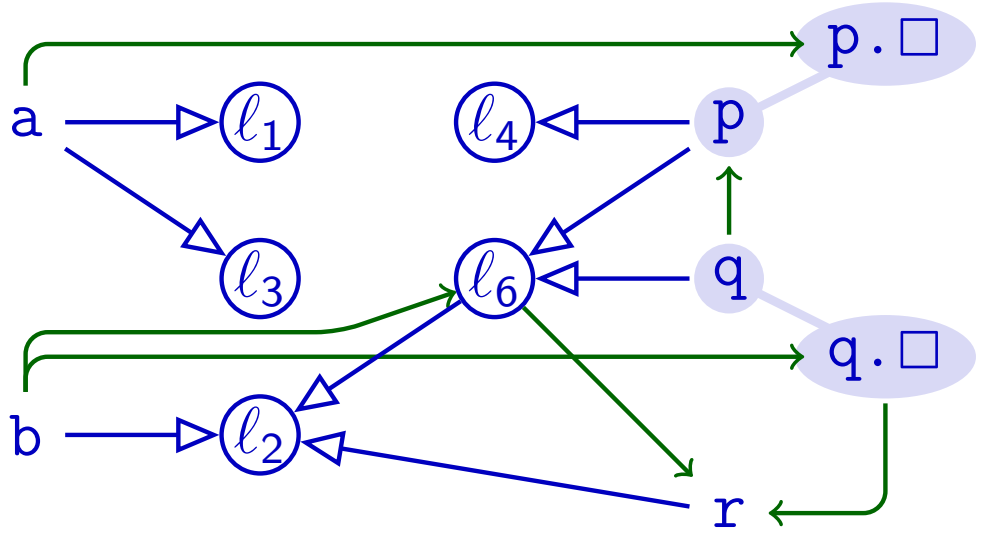
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$v \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrow x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

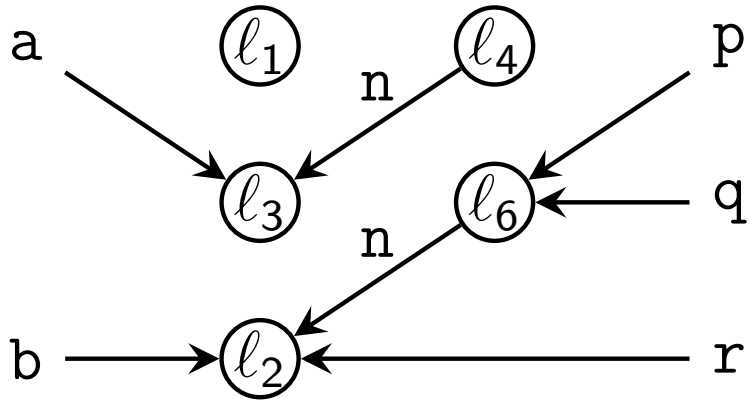
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

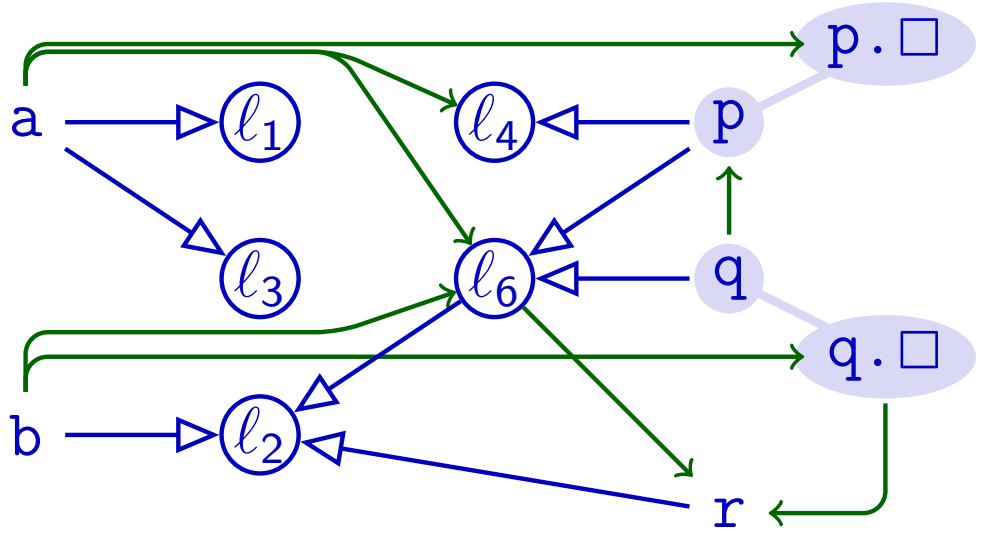
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$v \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrow x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

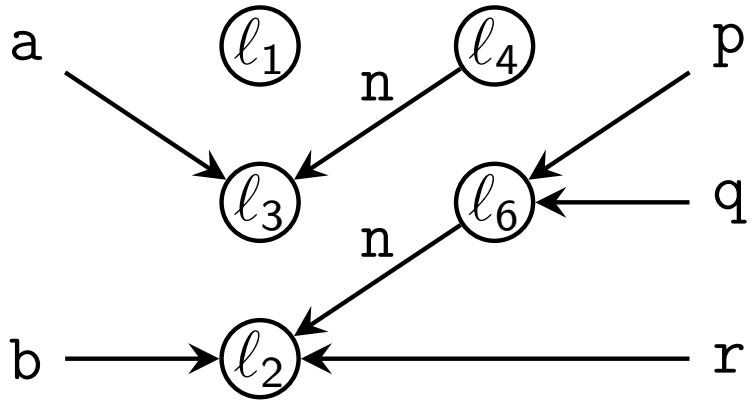
```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

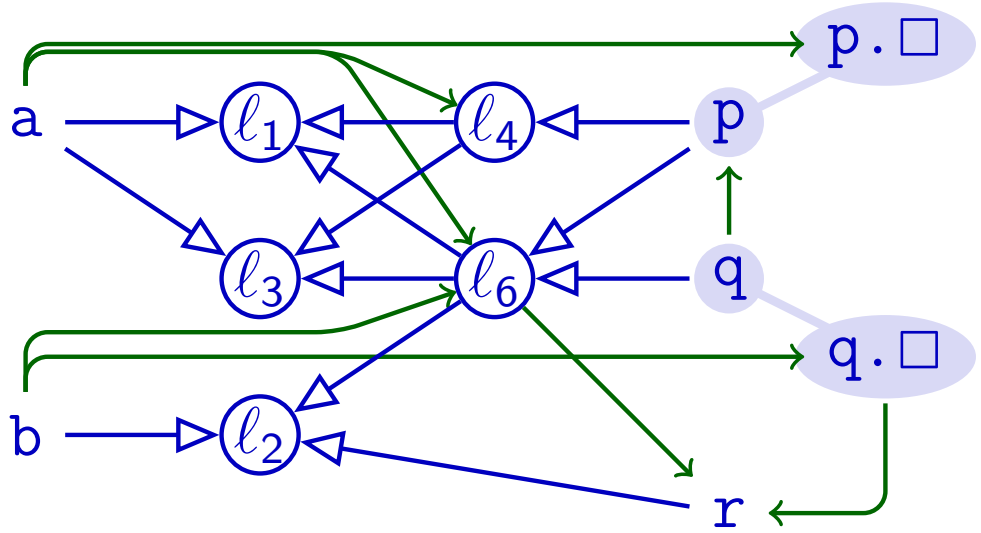
Example

$\ell \leftarrow y$ and $x \leftarrow y \implies \ell \leftarrow x$
$v \leftarrow y$ and $x \leftarrow y.\square \implies x \leftarrow v$
$v \leftarrow x$ and $x.\square \leftarrow y \implies v \leftarrow y$

- **Actual**:



- **Andersen**:



Teal

```
var a := new_ℓ1();
var b := new_ℓ2();
a := new_ℓ3();
var p := new_ℓ4();
p.n := a;
var q := new_ℓ6();
q.n := b;
p := q;
var r := q.n;
```

Andersen's algorithm must propagate along inclusion graph

Implementation

- Graph structure
- Two types of edges
- Connection between x and $x.\square$
- Worklist:
  - Track all *new* edges (at start: *all* extracted edges)
  - Process one edge at a time:
    - Remove from worklist, add to "completed edges"
    - Check our thee rules: does current edge + completed edges allow producing new edge that is neither in worklist nor completed?
    - If so: add all such edges to worklist (may be several!)

$$\ell \leftarrow y \text{ and } x \leftarrow y \implies \ell \leftarrow x$$
$$v \leftarrow y \text{ and } x \leftarrow y.\square \implies x \leftarrow v$$
$$v \leftarrow x \text{ and } x.\square \leftarrow y \implies v \leftarrow y$$

Complexity

- Complexity of graph closure: $O(n^3)$
- Traditional assumption about Andersen's analysis
- Close to $O(n^2)$ if:

1. Few statements dereference each variable
2. Control flow graphs not too complex

   Both conditions are common in practical programs

Manu Sridharan, Stephen J. Fink, "The Complexity of Andersen's Analysis in Practice", in SAS 2009

Summary

- Andersen's analysis:
  - Subset-based
  - Builds inclusion graph for propagating memory locations along subset constraints
  - $O(n^3)$ worst-case behaviour
  - Closer to $O(n^2)$ in practice
  - More precise than Steensgaard's analysis
  - Less scalable than Steensgaard's analysis

Challenges Towards OO Support

- (+) Flow-sensitivity
- (+) Points-to information
- Dynamic Dispatch
- Advanced features:
  - Pointer arithmetic
  - Dynamic Class Loading
  - "Native Calls" (into C/assembly/Syscalls)
  - Reflection

The Call Graph

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

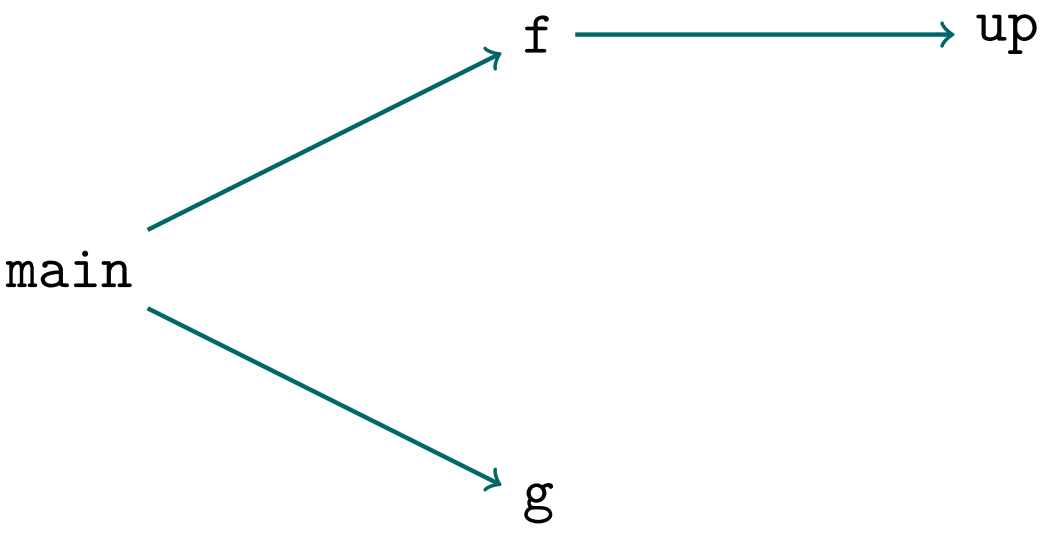
```
void f(char *s) {
    for (char *p = s; *p; p++) {
        *p = up(*p);
    }
    puts(s);
}
```

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

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

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

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

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

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

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

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

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

Finding Calls and Targets

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

a.f

a.g

a2.g

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

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

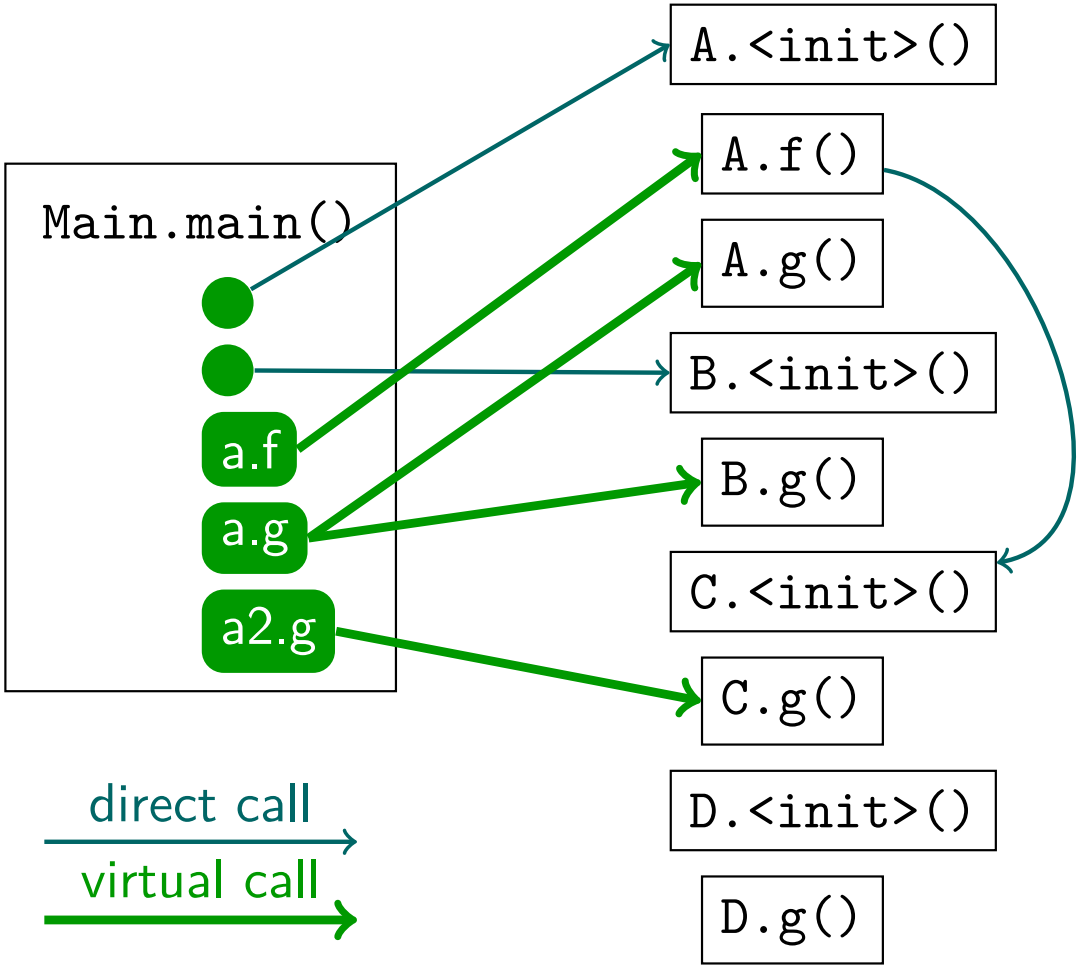
```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

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

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

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
  - Not straightforward

Callgraphs with Points-to Data

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

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

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

```
A a = new A();
a = a.f();
a = a.f();
```

- Precision of call graph affects quality of all interprocedural analyses
  - IFDS, IDE
  - Points-to analyses
- Idea: Use points-to analysis to determine *dynamic* type of objects
  - More precise virtual call resolution!
- **Problem**: Mutual dependency between call-graph and points-to analysis!

Finding Calls and Targets

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

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

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

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

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

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

Finding Calls and Targets

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

a.f

a.g

a2.g

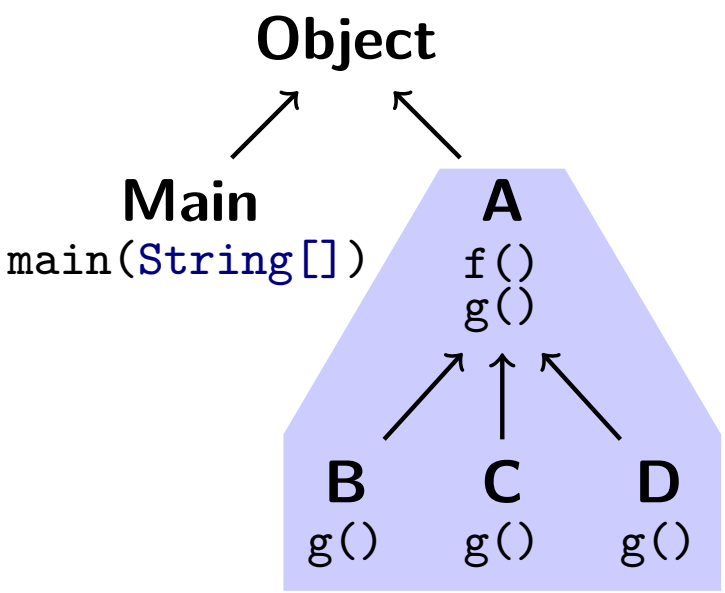
```
class A {
  public A
  f() { return new C(); }

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

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

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

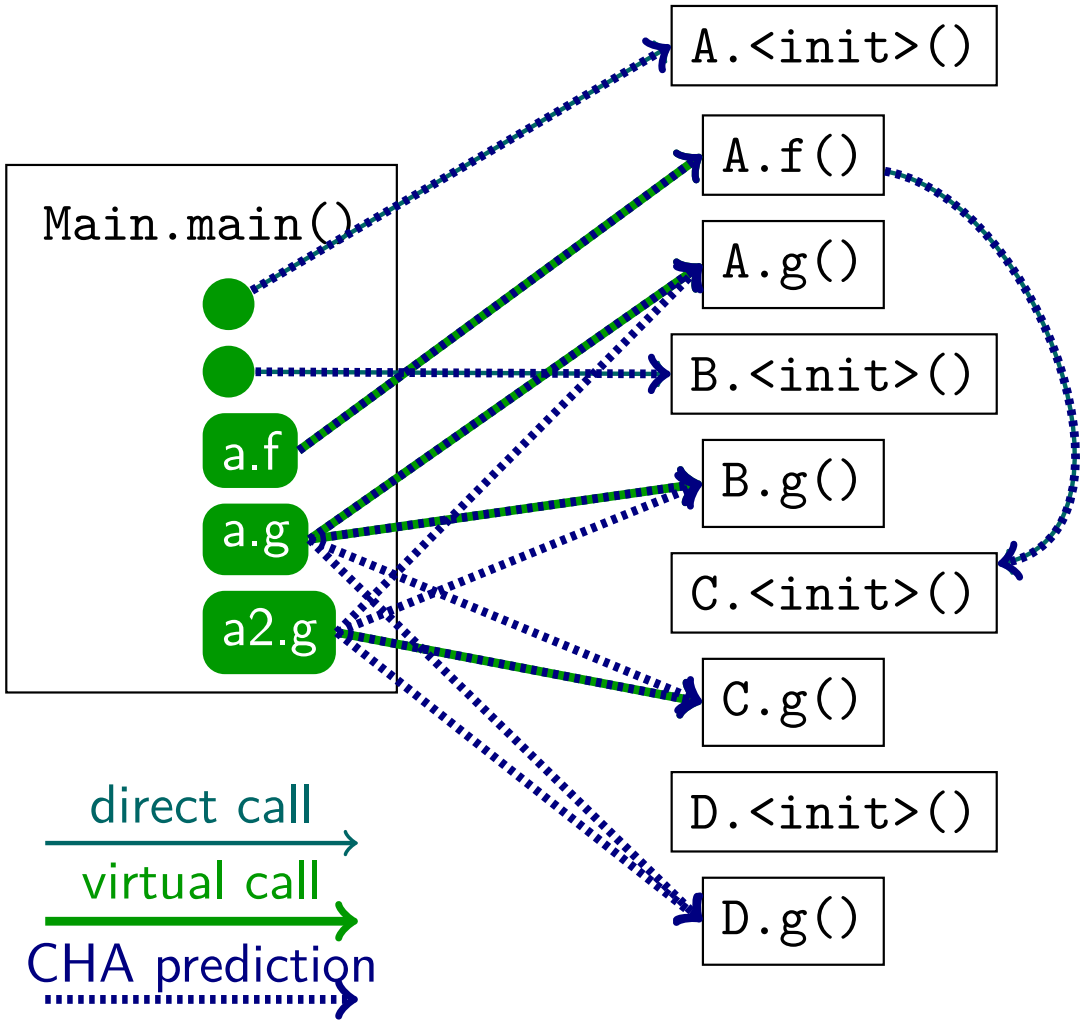
```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Class Hierarchy Analysis



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

Class Hierarchy Analysis: Example



Summary

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

Rapid Type Analysis

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

Finding Calls and Targets

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

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

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

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

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

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

Finding Calls and Targets

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

a.f

a.g

a2.g

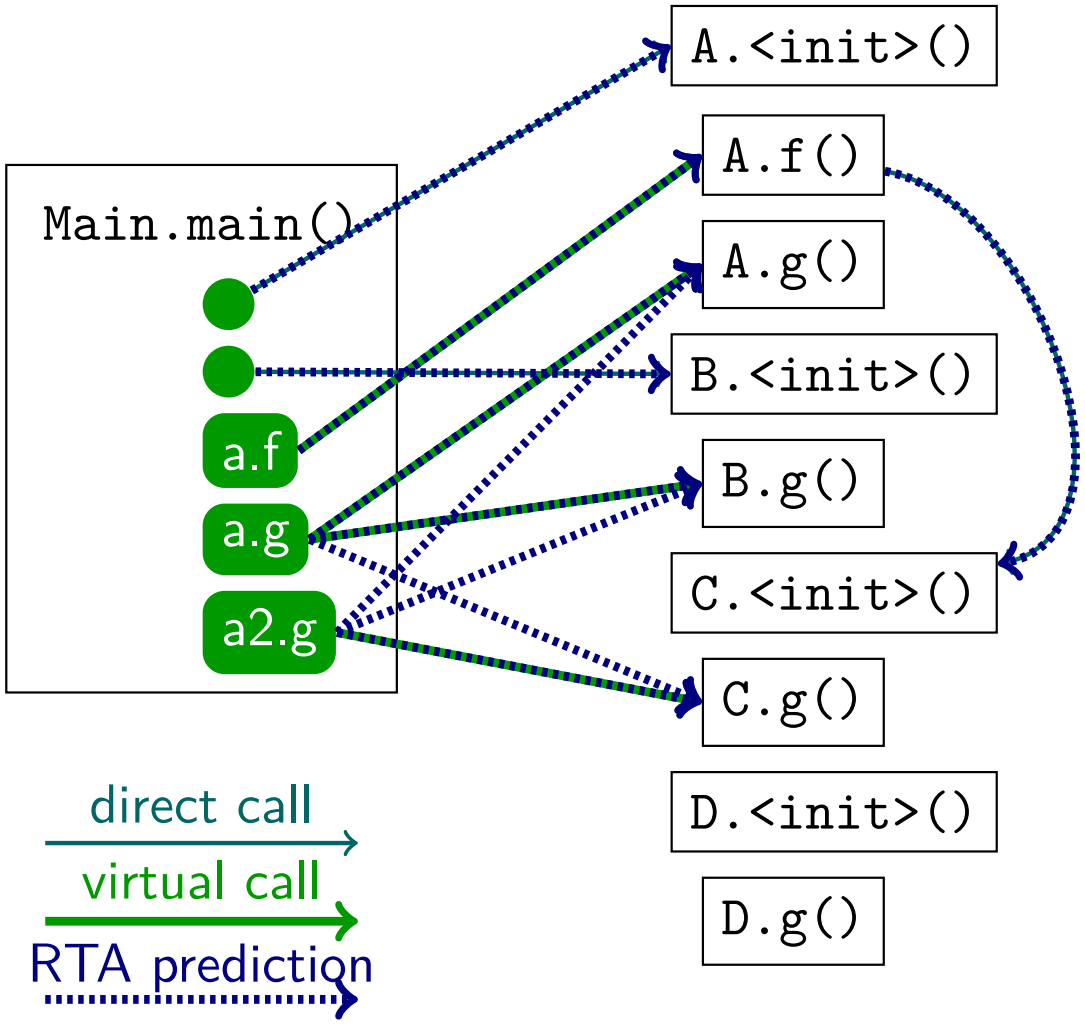
```
class A {
  public A
  f() { return new C(); }

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

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

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

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

Rapid Type Analysis: Example



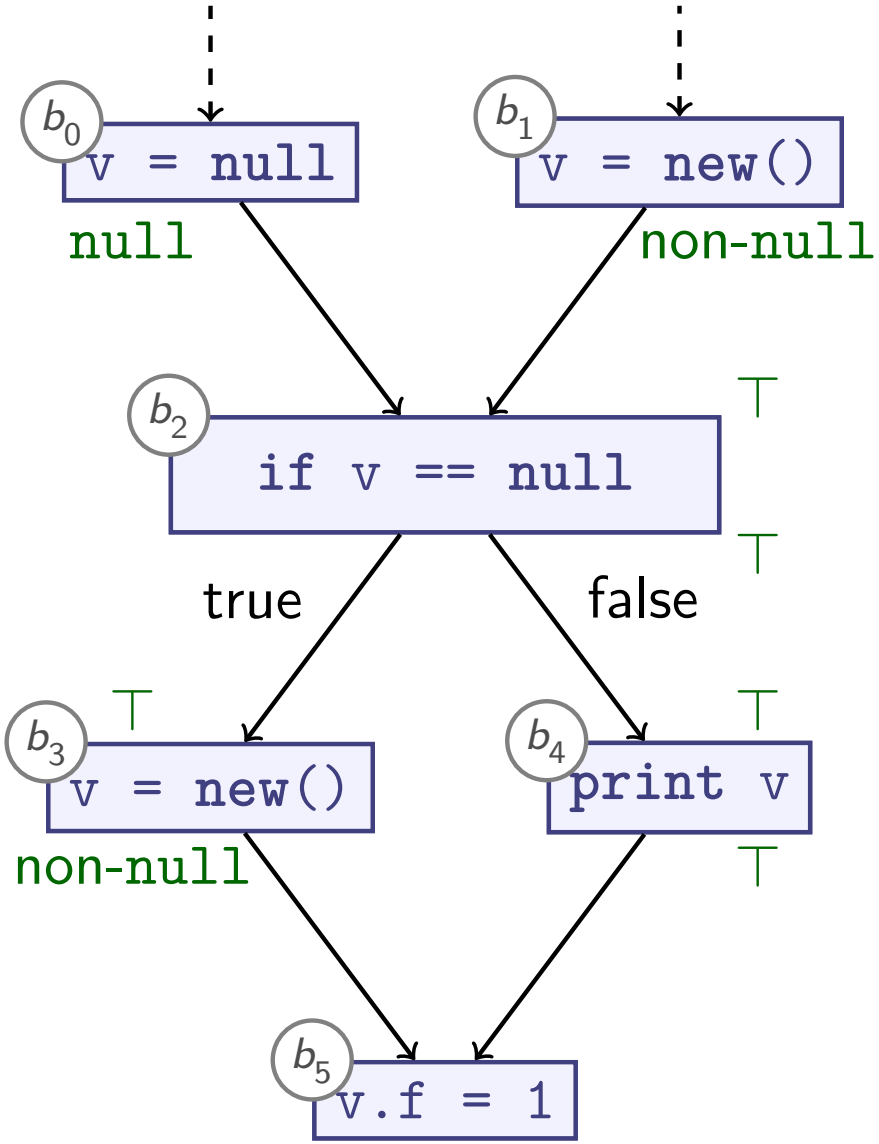
Rapid Type Analysis Algorithm Sketch

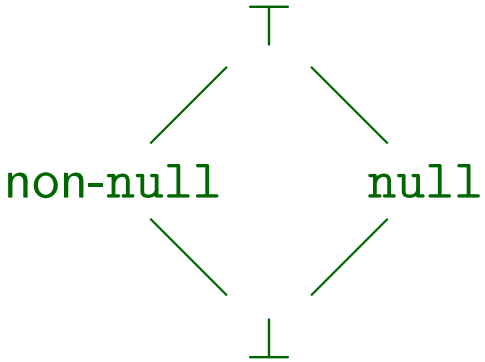
```
Procedure RTA(mainproc, <:):
begin
  WORKLIST := {mainproc}
  VIRTUALCALLS := ∅
  LIVECLASSES := ∅
  while s ∈ mainproc do
    foreach call c ∈ s do
      if c is direct call to p then
        addToWorklist(p)
        registerCallEdge(c → p)
      else if c = v.m() and v : T then begin
        VIRTUALCALLS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

Summary

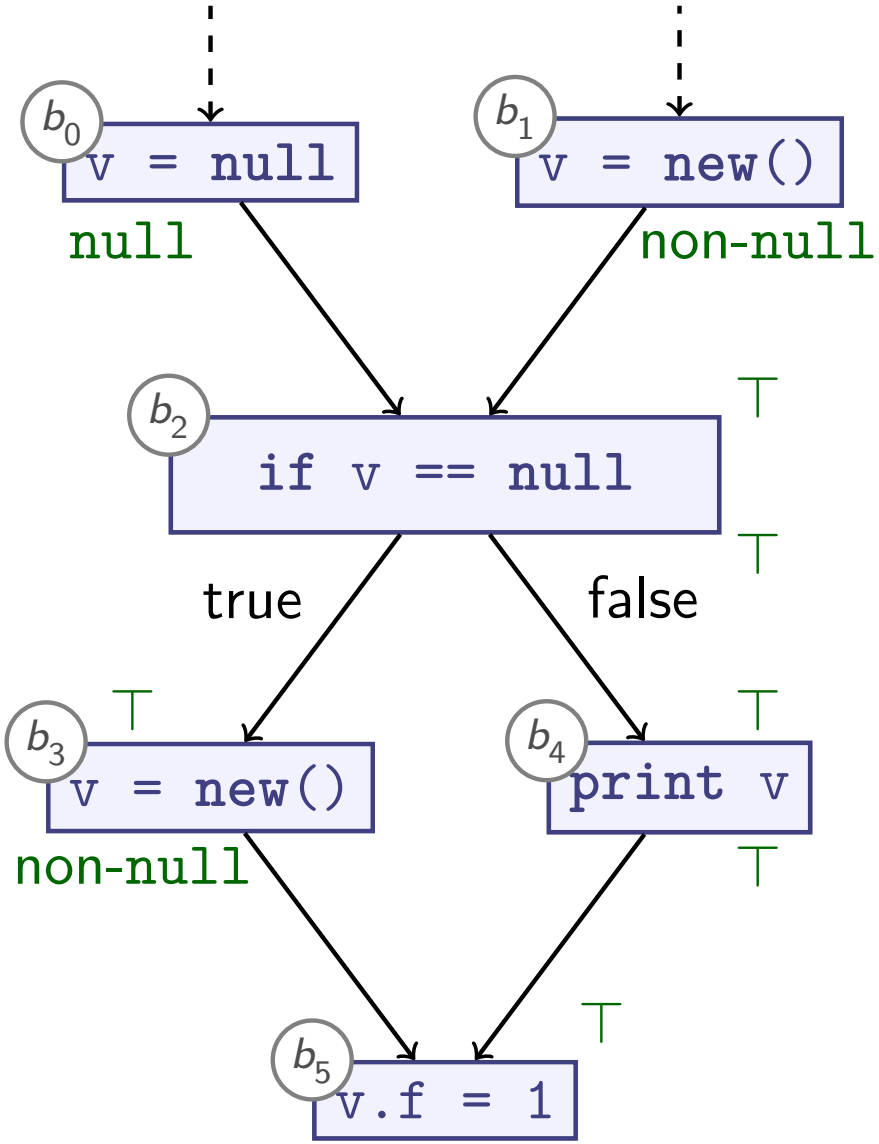
- **Rapid Type Analysis** resolves virtual calls $a.f()$ as follows:
  - Find all classes that can be instantiated in reachable code
  - Expand reachable code:
    - For direct calls to p, add p as reachable
    - For all virtual calls to $v.m()$ with $v : T$:
      $\Rightarrow$ Add $S.m()$ as reachable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

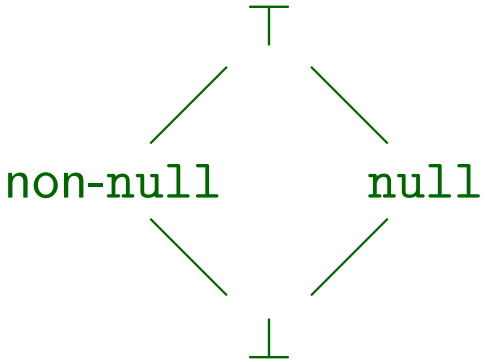
Control Sensitivity





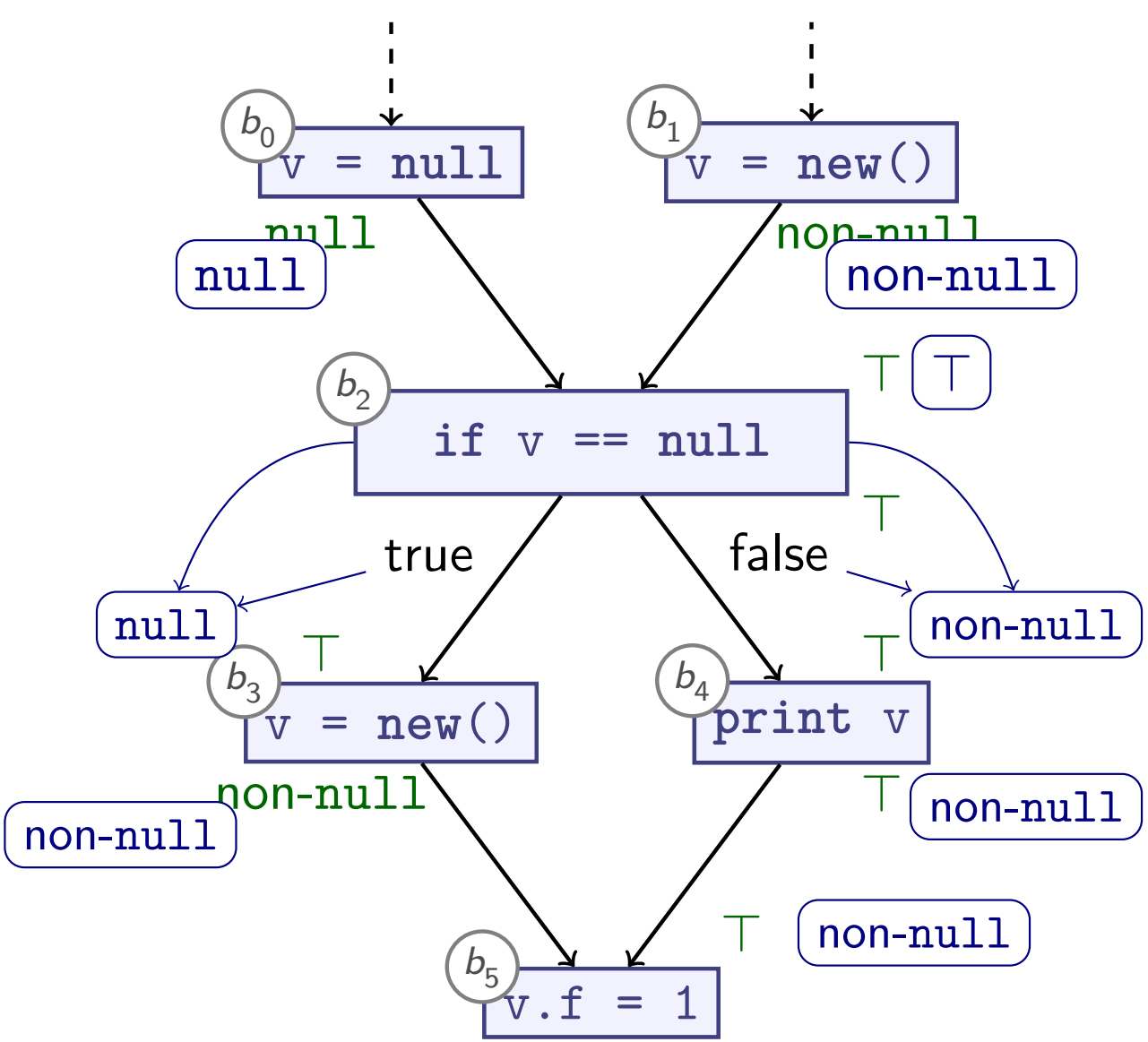
Control Sensitivity

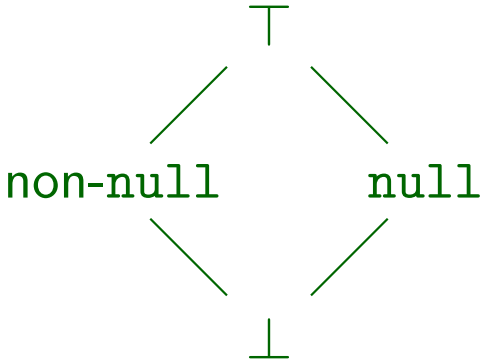




control insensitive

Control Sensitivity

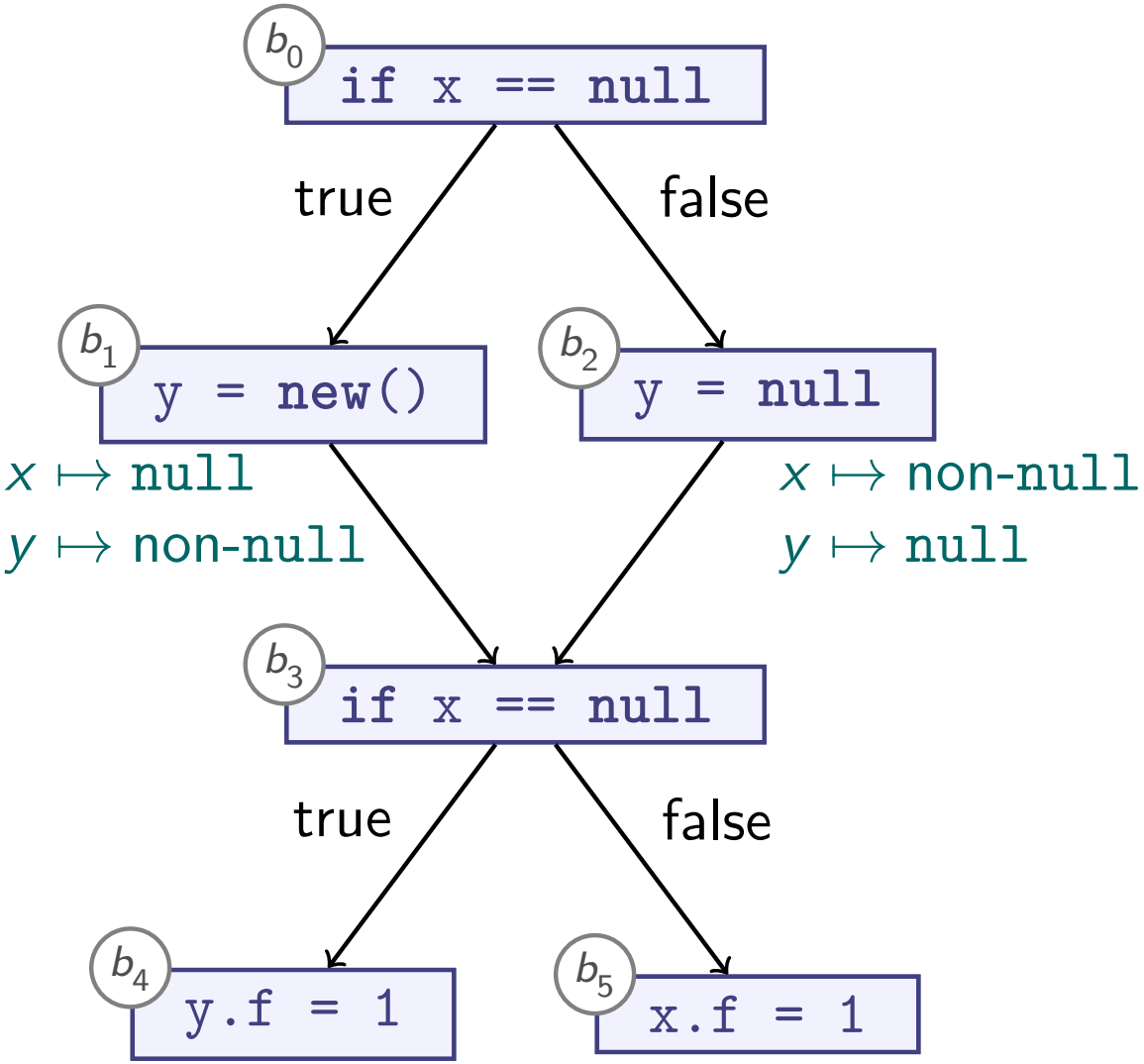




control insensitive

control sensitive

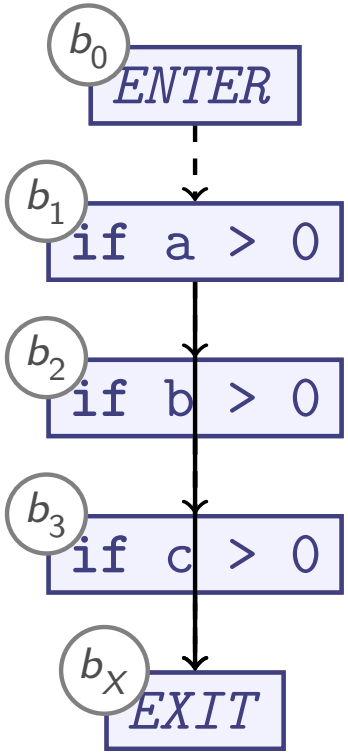
Multiple Conditionals



Should we carry path information across merge points?

Path Sensitivity

proc f(a, b, c)



2 paths

4 paths

8 paths

Number of paths grows exponentially

Summary

- **Control sensitive** analysis considers conditionals:
  - May propagate different information along different edges:
    - **if** P:
    - Special transfer function for '**assert** P' on 'true' edge
    - Special transfer function for '**assert** `not` P' on 'false' edge
- **Path sensitive** analysis considers one sequence of CFG edges (execution path) at a time:
  - May propagate different information along different paths
  - High precision possible, but must cover *all* paths
  - Number of paths $O(\#$ of conditionals$)$
  - Avoid exponential blow-up by merging (as before)
  - Path-sensitive procedure summaries might require exponential number of cases
  - *Usually* not practical