Lecture 06 – Inter-procedural Analysis

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

Previously

- Verifying absence of buffer overruns
- (requires) Heap abstractions
- (requires) Combining heap and numerical information

Today

- LFP computation and join-over-all-paths
- Inter-procedural analysis

- Acks
  - Some slides adapted from
    Noam Rinetzky and Mooly Sagiv
  - DNA of some slides traces back to Tom Reps

Join over all paths (JOP)

- Propagate analysis information along paths
- A path is a sequence of edges in the CFG $[e_1, e_2, \ldots, e_n]$
- Can define the transfer function for a path via transformer composition
  - let f_i denote $f(e_i)$
  - $f[e_1, e_2, \ldots, e_n] = f_n \circ \ldots \circ f_2 \circ f_1$

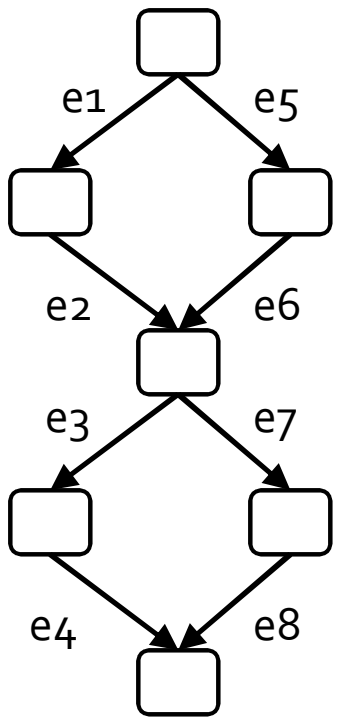
- Consider the (potentially infinite) set of paths reaching a label in the program from entry. Denote by Paths-to-in(entry,l) the set of paths from entry to label l, similarly for paths-to-out

- The result at program point l can be defined as
  JOPin (l) = $\sqcup$ { f[p](initial) | p $\in$ paths-to-in(entry,l) }
  JOPout(l) = $\sqcup$ { f[p](initial) | p $\in$ paths-to-out(entry,l) }

The lfp computation approximates JOP

- JOPout(l) =
  $\sqcup$ { f[p](initial) | p $\in$ paths-to-out(entry,l) }
- Set of paths potentially infinite (non computable)
- The lfp computation take joins "earlier"
  - Merging sub-paths
- For a monotone function
  - $f(x \sqcup y) \sqsupseteq f(x) \sqcup f(y)$

lfp computation and JOP



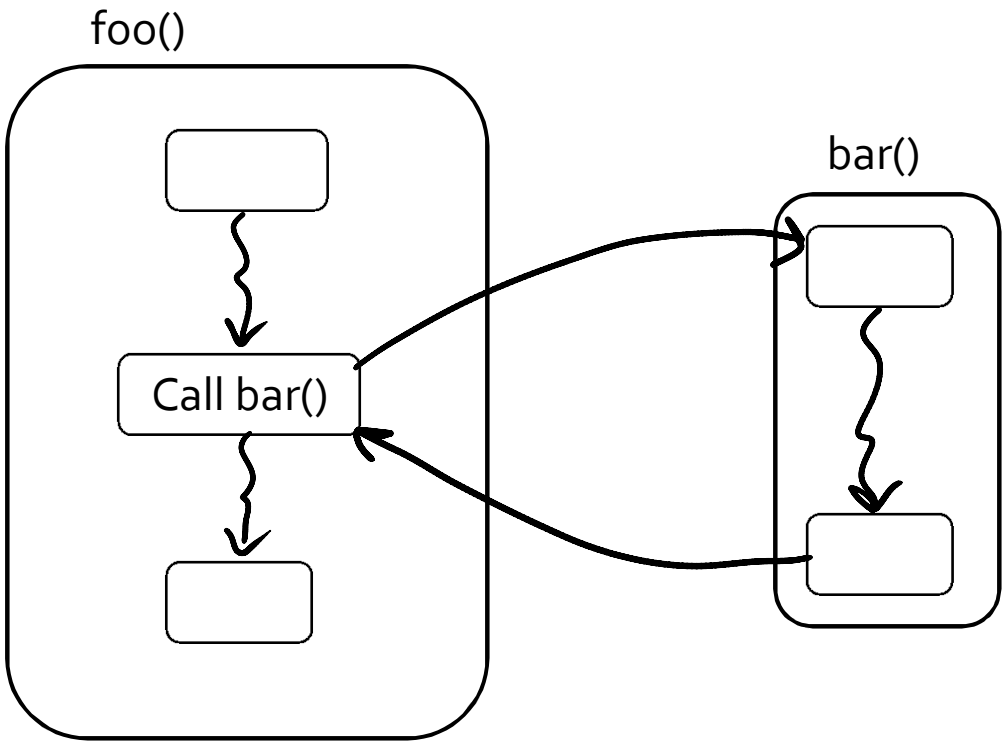
Paths transformers:
f[e1,e2,e3,e4]
f[e1,e2,e7,e8]
f[e5,e6,e7,e8]
f[e5,e6,e3,e4]

JOP:
f[e1,e2,e3,e4](initial) ⊔
f[e1,e2,e7,e8](initial) ⊔
f[e5,e6,e7,e8](initial) ⊔
f[e5,e6,e3,e4](initial)

"lfp" computation:
j1 =f[e1,e2](initial) ⊔ f[e5,e6](initial)
j2 = f[e3,e4](j1) ⊔ f[e7,e8](j1)

(wrote "lfp" just to emphasize that no iteration is required here, no loops)

Interprocedural Analysis



- The effect of calling a procedure is the effect of executing its body

Interprocedural Analysis

- Stack can grow without a bound
- Matching of call/return

Solution Attempt #1

- Inline callees into callers
  - End up with one big procedure
  - CFGs of individual procedures = duplicated many times
- Good: it is precise
  - distinguishes different calls to the same function
- Bad
  - exponential blow-up, not efficient
  - doesn't work with recursion

Simple Example

```
void main() {
    int x ;
    x = p(7);
    x = p(9) ;
}
```

```
int p(int a) {
  return a + 1;
  }
```

Simple Example

```
void main() {
    int x ;
    x = 7; x = x + 1;
    [x ↦8]
    x = 9; x = x + 1;
    [x ↦10]
}
```

Inlining: Exponential Blowup

```
main() {
  f1();
  f1();
}
f1() {
  f2();
  f2();
}
...
fn() { ... }
```

Solution Attempt #2

- Build a "supergraph" = inter-procedural CFG
- Replace each call from P to Q with
  - An edge from point before the call (call point) to Q's entry point
  - An edge from Q's exit point to the point after the call (return pt)
  - Add assignments of actuals to formals, and assignment of return value
- Good: efficient
  - Graph of each function included exactly once in the supergraph
  - Works for recursive functions (although local variables need additional treatment)
- Bad: imprecise, "context-insensitive"
  - The "unrealizable paths problem": dataflow facts can propagate along infeasible control paths

Simple Example

```
void main() {
    int x ;
⇨   x = p(7);
    x = p(9) ;
}
```

```
int p(int a) {
  return a + 1;
  }
```

Simple Example

```
void main() {
    int x ;
    x = p(7);
    x = p(9) ;
}
```

⇨

```
int p(int a) {
    [a ↦7]
    return a + 1;
}
```

Simple Example

```
void main() {
    int x ;
    x = p(7);
    x = p(9) ;
}
```

```
int p(int a) {
    [a ↦7]
⇨   return a + 1;
    [a ↦7, $$ ↦8]

}
```

Simple Example

```
void main() {
    int x ;
    x = p(7); ⇦
    [x ↦8]
    x = p(9) ; ⇦
    [x ↦8]
}
```

```
int p(int a) {
   [a ↦7]
    return a + 1;
  [a ↦7, $$ ↦8]

}
```

Simple Example

```
void main() {
    int x ;
    x =l p(7);
     [x ↦8]
⇨   x = p(9) ;
    [x ↦8]
}
```

```
int p(int a) {
  [a ↦7]
    return a + 1;
  [a ↦7, $$ ↦8]

}
```

Simple Example

```
void main() {
    int x ;
    x = p(7);
    [x ↦8]
    x = p(9) ;
    [x ↦8]
}
```

⇨

```
int p(int a) {
    [a ↦τ]
      return a + 1;
    [a ↦7, $$ ↦8]

}
```

Simple Example

```
void main() {
    int x ;
    x = p(7);
    [x ↦8]
    x = p(9);
    [x ↦8]

}
```

```
int p(int a) {
    [a ↦τ]
⇨   return a + 1;
    [a ↦τ, $$ ↦τ]

}
```

Simple Example

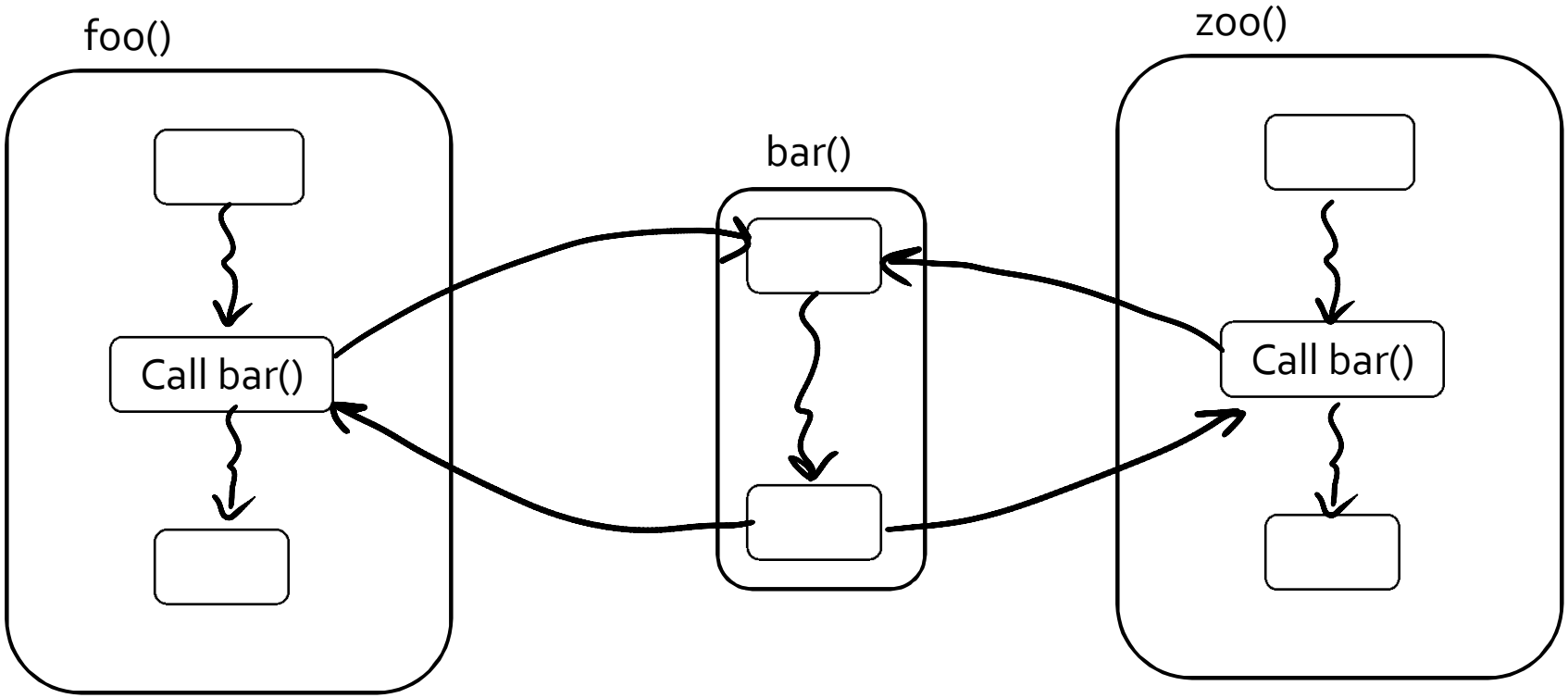
```
void main() {
    int x ;
    x = p(7) ; ⇦
     [x ↦τ]
    x = p(9) ; ⇦
     [x ↦τ]

}
```

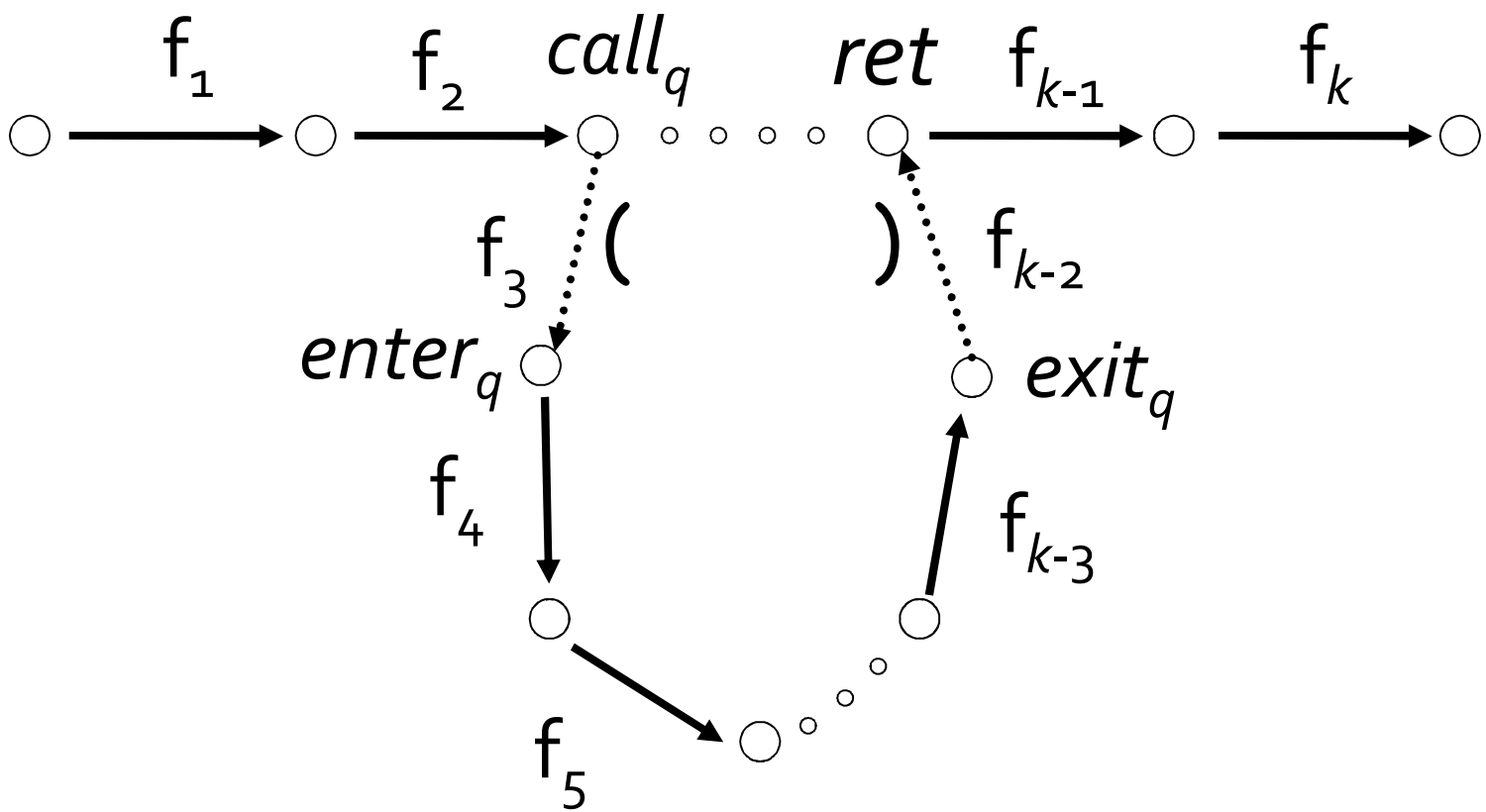
```
int p(int a) {
   [a ↦τ]
    return a + 1;
   [a ↦τ, $$ ↦τ]

}
```

Unrealizable Paths

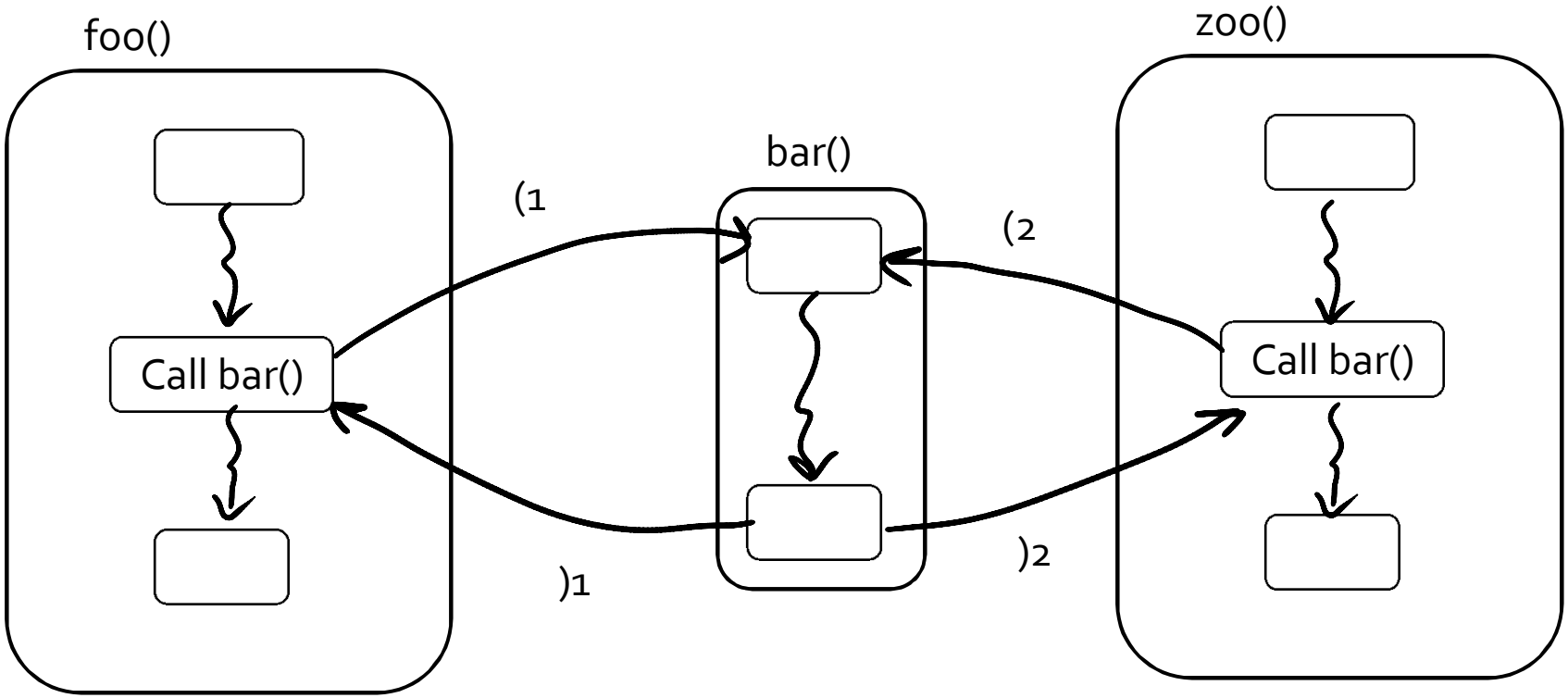


IVP: Interprocedural Valid Paths



- IVP: all paths with matching calls and returns
  - And prefixes

Valid Paths



Interprocedural Valid Paths

- **IVP** set of paths
  - Start at program entry
- Only considers matching calls and returns
  - aka, valid

- Can be defined via context free grammar
  - matched ::= matched $(_i$ matched $)_i$ | ε
  - valid ::= valid $(_i$ matched | matched
    - *paths* can be defined by a regular expression

The Join-Over-Valid-Paths (JVP)

- vpaths(n) all valid paths from program start to n
- JVP[n] $= \sqcup\{[\![e_1, e_2, \ldots, e]\!]$ (initial)
  $(e_1, e_2, \ldots, e) \in \text{vpaths}(n)\}$
- JVP $\sqsubseteq$ JFP
  - In some cases the JVP can be computed
  - (Distributive problem)

Sharir and Pnueli ‘82

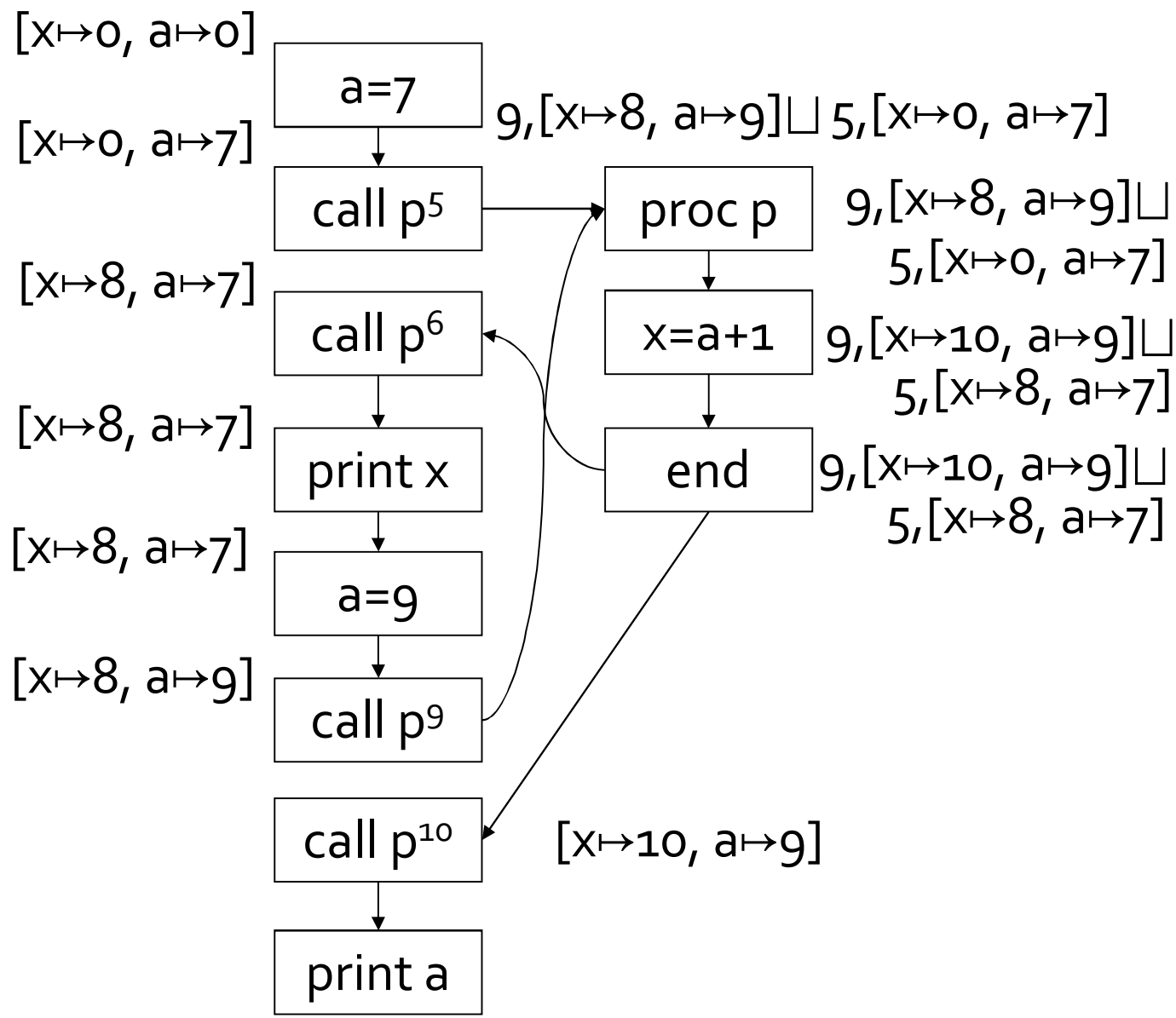
- Call String approach
  - Blend interprocedural flow with intra procedural flow
  - Tag every dataflow fact with call history
- Functional approach
  - Determine the effect of a procedure
    - E.g., in/out map
  - Treat procedure invocations as “super ops”

The Call String Approach

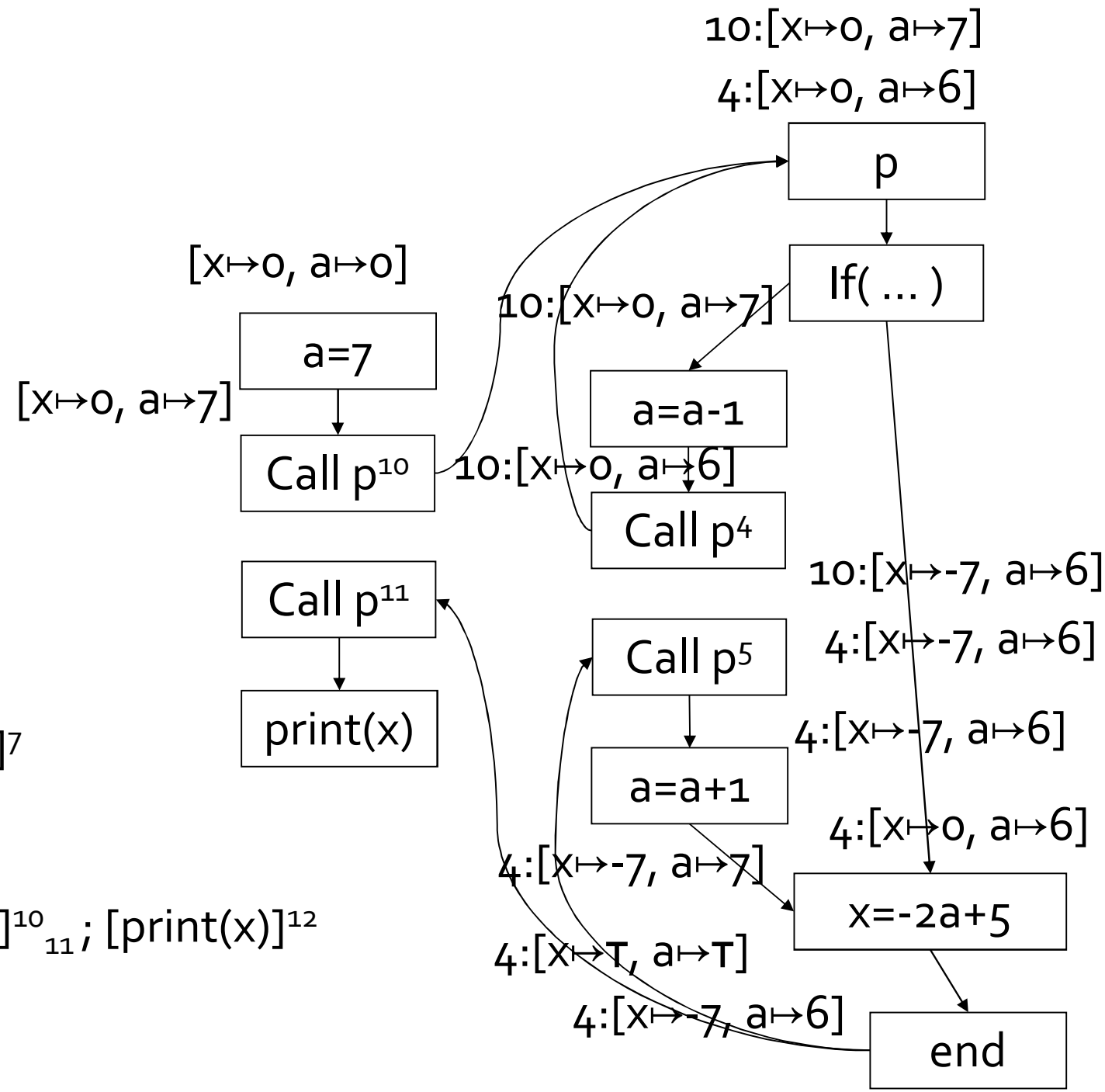
- Record at every node a pair (l, c) where $l \in L$ is the dataflow information and c is a suffix of unmatched calls

- Use Chaotic iterations
- To guarantee termination limit the size of c (typically 1 or 2)
- Emulates inline (but no code growth)
- Exponential in size of c

```
begin
proc p() is¹
   [x := a + 1]²
end³
[a=7]⁴
[call p()]⁵₆
[print x]⁷
[a=9]⁸
[call p()]⁹₁₀
[print a]¹¹
end
```



```
begin⁰
 proc p() is¹
   if [b]²  then (
    [a := a -1]³
    [call p()]⁴₅
    [a := a + 1]⁶
   )
  [x := -2* a + 5]⁷
end⁸
[a=7]⁹ ; [call p()]¹⁰₁₁ ; [print(x)]¹²
end¹³
```



Simple Example

```
void main() {
    int x ;
⇨   c1: x =  p(7);
    c2: x =  p(9) ;

}
```

```
int p(int a) {
    return a + 1;
    }
```

Simple Example

```
void main() {
    int x ;
    c1: x =  p(7);
    c2: x =  p(9) ;

}
```

⇨

```
int p(int a) {
    c1: [a ↦7]
    return a + 1;
}
```

Simple Example

```
void main() {
    int x ;
    c1: x =  p(7);
    c2: x =  p(9) ;

}
```

⇨

```
int p(int a) {
    c1: [a ↦7]
    return a + 1;
    c1:[a ↦7, $$ ↦8]
}
```

Simple Example

```
void main() {
    int x ;
    c1: x =  p(7); ⇦
     ε: x ↦ 8
    c2: x =  p(9) ;

}
```

```
int p(int a) {
    c1: [a ↦7]
    return a + 1;
    c1:[a ↦7, $$ ↦8]
}
```

Simple Example

```
void main() {
    int x ;
    c1: x = p(7);
    ε: [x ↦ 8]
⇨   c2: x = p(9) ;
}
```

```
int p(int a) {
   c1:[a ↦7]
    return a + 1;
   c1:[a ↦7, $$ ↦8]
}
```

Simple Example

```
void main() {
    int x ;
    c1: x = p(7);
    ε: [x ↦ 8]
    c2: x = p(9) ;
}
```

⇨

```
int p(int a) {
    c1:[a ↦7]
    c2:[a ↦9]
    return a + 1;
    c1:[a ↦7, $$ ↦8]
}
```

Simple Example

```
void main() {
    int x ;
    c1: x = p(7);
    ε: [x ↦ 8]
    c2: x = p(9) ;
}
```

⇨

```
int p(int a) {
    c1:[a ↦7]
    c2:[a ↦9]
    return a + 1;
    c1:[a ↦7, $$ ↦8]
    c2:[a ↦9, $$↦10]
}
```

Simple Example

```
void main() {
    int x ;
    c1: x = p(7);
    ε: [x ↦ 8]
    c2: x = p(9) ; ⟸
    ε: [x ↦ 10]
}
```

```
int p(int a) {
    c1:[a ↦7]
    c2:[a ↦9]
    return a + 1;
    c1:[a ↦7, $$ ↦8]
    c2:[a ↦9, $$ ↦10]
}
```

Another Example

```
void main() {
    int x ;
    c1: x = p(7);
    ε: [x ↦ 8]
    c2: x = p(9) ;
    ε: [x ↦ 10]
}
```

```
int p(int a) {
    c1:[a ↦7]
    c2:[a ↦9]
    return c3: p1(a + 1);
    c1:[a ↦7, $$ ↦τ]
    c2:[a ↦9, $$ ↦τ]
}
```

```
int p1(int b) {
    (c1|c2)c3:[b ↦τ]
    return 2 * b;
(c1|c2)c3:[b ↦τ, $$↦τ]
}
```

Recursion

```
void main() {
    c1: p(7);
ε: [x ↦τ]
}
```

```
int p(int a) {
    c1: [a ↦ 7]    c1.c2+: [a ↦ τ]
    if (…)  {
      c1: [a ↦ 7]  c1.c2+: [a ↦ τ]
      a = a -1 ;
      c1: [a ↦ 6]   c1.c2+: [a ↦ τ]
      c2: p (a);
      c1.c2*: [a ↦ τ]
      a = a + 1;
      c1.c2*: [a ↦ τ]
}
c1.c2*: [a ↦ τ]
x = -2*a + 5;
c1.c2*: [a ↦ τ, x↦τ]
}
```

Summary: Call String

- Simple
- Only feasible for very short call strings
  - exponential in call-string length
- Often loses precision under recursion
  - although can still be precise in some cases

The Functional Approach

- The meaning of a procedure is mapping from states into states
- The abstract meaning of a procedure is function from an abstract state to abstract states

Functional Approach: Main Idea

- Iterate on the abstract domain of functions from L to L
- Two phase algorithm
  - Compute the dataflow solution at the exit of a procedure as a function of the initial values at the procedure entry (functional values)
  - Compute the dataflow values at every point using the functional values
- Computes JVP for distributive problems

Phase 1

```
void main() {
    p(7);
}
```

⇨

```
int p(int a) {
    [a ↦a0, x ↦x0]
    if (…)  {
        a = a -1 ;
        p (a);
        a = a + 1;
    }
    x = -2*a + 5;
    [a ↦a0, x ↦-2a0 + 5]
}
```

Phase 1

```
void main() {
    p(7);
}
```

```
int p(int a) {
  [a ↦a0, x ↦x0]
 if (…)  {
      a = a -1 ;
      p (a);
      a = a + 1;
    }
```

⇨

```
[a ↦a0, x ↦x0]
x = -2*a + 5;
}
```

Phase 1

```
void main() {
    p(7);
}
```

```
int p(int a) {
  [a ↦a0, x ↦x0]
 if (…)  {
      a = a -1 ;
      p (a);
      a = a + 1;
     }
[a ↦a0, x ↦x0]
⇨ x = -2*a + 5;
[a ↦a0, x ↦-2a0 + 5]
}
```

Phase 1

```
void main() {
    p(7);
}
```

```
int p(int a) {
  [a ↦a0, x ↦x0]
 if (…)  {
⇨ a = a -1 ;
     [a ↦a0-1, x ↦x0]
      p (a);
      a = a + 1;
     }
[a ↦a0, x ↦x0]
x = -2*a + 5;
[a ↦a0, x ↦-2a0 + 5]
}
```

Phase 1

```
void main() {
    p(7);
}
```

```
int p(int a) {
  [a ↦a0, x ↦x0]
 if (…)  {
      a = a -1 ;
     [a ↦a0-1, x ↦x0]
⇨ p (a);
     [a ↦a0-1, x ↦-2a0+7]
      a = a + 1;
     }
[a ↦a0, x ↦x0]
x = -2*a + 5;
[a ↦a0, x ↦-2a0 + 5]
}
```

Phase 1

```
void main() {
    p(7);
}
```

```
int p(int a) {
  [a ↦a0, x ↦x0]
 if (…)  {
      a = a -1 ;
     [a ↦a0-1, x ↦x0]
      p (a);
     [a ↦a0-1, x ↦-2a0+7]
⇨ a = a + 1;
      [a ↦a0, x ↦-2a0+7]
     }
[a ↦a0, x ↦x0]
x = -2*a + 5;
[a ↦a0, x ↦-2a0 + 5]
}
```

Phase 1

```
void main() {
    p(7);
}
```

```
int p(int a) {
  [a ↦a0, x ↦x0]
 if (…)  {
      a = a -1 ;
     [a ↦a0-1, x ↦x0]
      p (a);
     [a ↦a0-1, x ↦-2a0+7]
      a = a + 1;
      [a ↦a0, x ↦-2a0+7]
⇨ }
 [a ↦a0, x ↦τ]
 x = -2*a + 5;
 [a ↦a0, x ↦-2a0 + 5]
 }
```

Phase 1

```
void main() {
    p(7);
}
```

```
int p(int a) {
  [a ↦a0, x ↦x0]
 if (…)  {
      a = a -1 ;
     [a ↦a0-1, x ↦x0]
      p (a);
     [a ↦a0-1, x ↦-2a0+7]
      a = a + 1;
      [a ↦a0, x ↦-2a0+7]
     }
[a ↦a0, x ↦τ]
⇨ x = -2*a + 5;
[a ↦a0, x ↦-2a0 + 5]
}
```

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

⟹

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
      a = a -1 ;
     p (a);
     a = a + 1;
      }
x = -2*a + 5;
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
      a = a -1 ;
     p (a);
     a = a + 1;
⇨ }
[a ↦7, x ↦0]
x = -2*a + 5;
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
      a = a -1 ;
     p (a);
     a = a + 1;
      }
[a ↦7, x ↦0]
⇨ x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
⇨ a = a -1 ;
      [a ↦6, x ↦0]
      p (a);
       a = a + 1;
       }
[a ↦7, x ↦0]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦6, x ↦0]
⇨ p (a);
     [a ↦6, x ↦-9]
     a = a + 1;
      }
[a ↦7, x ↦0]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦6, x ↦0]
     p (a);
     [a ↦6, x ↦-9]
⇨    a = a + 1;
     [a ↦7, x ↦-9]
      }
[a ↦7, x ↦0]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

$p(a_0,x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]$

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦6, x ↦0]
     p (a);
     [a ↦6, x ↦-9]
     a = a + 1;
     [a ↦7, x ↦-9]
⇨  }
[a ↦7, x ↦τ]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

$p(a_0,x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]$

```
int p(int a) {
[a ↦7, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦6, x ↦0]
⇨    p (a);
     [a ↦6, x ↦-9]
     a = a + 1;
     [a ↦7, x ↦-9]
      }
[a ↦7, x ↦⊤]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

⇨

```
int p(int a) {
[a ↦7, x ↦0] [a ↦6, x ↦0]
 if (…)  {
     a = a -1 ;
    [a ↦6, x ↦0]
    p (a);
    [a ↦6, x ↦-9]
    a = a + 1;
    [a ↦7, x ↦-9]
     }
[a ↦7, x ↦⊤]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

⇨

```
int p(int a) {
[a ↦⊤, x ↦0]
 if (…)  {
     a = a -1 ;
    [a ↦6, x ↦0]
    p (a);
    [a ↦6, x ↦-9]
    a = a + 1;
    [a ↦7, x ↦-9]
     }
[a ↦7, x ↦⊤]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

$p(a_0,x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]$

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

$p(a_0,x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]$

⇨

```
int p(int a) {
[a ↦⊤, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦⊤, x ↦0]
     p (a);
     [a ↦6, x ↦-9]
     a = a + 1;
     [a ↦7, x ↦-9]
      }
[a ↦7, x ↦⊤]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

$p(a_0,x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]$

```
int p(int a) {
[a ↦⊤, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦⊤, x ↦0]
     p (a);
⇨    [a ↦⊤, x ↦⊤]
     a = a + 1;
     [a ↦7, x ↦-9]
      }
[a ↦7, x ↦⊤]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

$p(a_0,x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]$

```
int p(int a) {
[a ↦⊤, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦⊤, x ↦0]
     p (a);
     [a ↦⊤, x ↦⊤]
     a = a + 1;
     [a ↦⊤, x ↦⊤]
 }
[a ↦7, x ↦⊤]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

$p(a_0,x_0) = [a \mapsto a_0, x \mapsto -2a_0 + 5]$

```
int p(int a) {
[a ↦⊤, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦⊤, x ↦0]
     p (a);
     [a ↦⊤, x ↦⊤]
     a = a + 1;
     [a ↦⊤, x ↦⊤]
 }
[a ↦⊤, x ↦⊤]
x = -2*a + 5;
[a ↦7, x ↦-9]
}
```

Phase 2

```
void main() {
    p(7);
[x ↦ -9]
}
```

$p(a_0,x_0) = [a \mapsto a_0,\ x \mapsto -2a_0 + 5]$

```
int p(int a) {
[a ↦⊤, x ↦0]
 if (…)  {
      a = a -1 ;
     [a ↦⊤, x ↦0]
     p (a);
     [a ↦⊤, x ↦⊤]
     a = a + 1;
     [a ↦⊤, x ↦⊤]
 }
[a ↦⊤, x ↦⊤]
x = -2*a + 5;
[a ↦⊤, x ↦⊤]
}
```

Issues in Functional Approach

- How to guarantee finite height for functional lattice?
  - Possible that L has finite height and yet the lattice of monotonic function from L to L do not
- Efficiently represent functions
  - Functional join
  - Functional composition
  - Testing equality

Tabulation

- Special case: L is finite
- Data facts: $d \in L \times L$
- Initialization:
  - $f_{start,start} = (\top,\top)$; otherwise $(\bot,\bot)$
  - $S[start, \top] = \top$

- Propagation of (x,y) over edge e = (n,n′)
  - Maintain summary: $S[n',x] = S[n',x] \sqcup [\![n]\!](y))$
  - n intra-node: ➔ n′: $(x, [\![n]\!](y))$
  - n call-node: ➔ n′: (y,y) *if* $S[n',y] = \bot$ and n′= entry node
    ➔ n′: (y,z) *if* $S[n',y] = z$ and n′= rest-site-of n
  - n return-node: ➔ n′: (u,y) ; n_c = call-site-of n′, $S[n_c,u]=x$

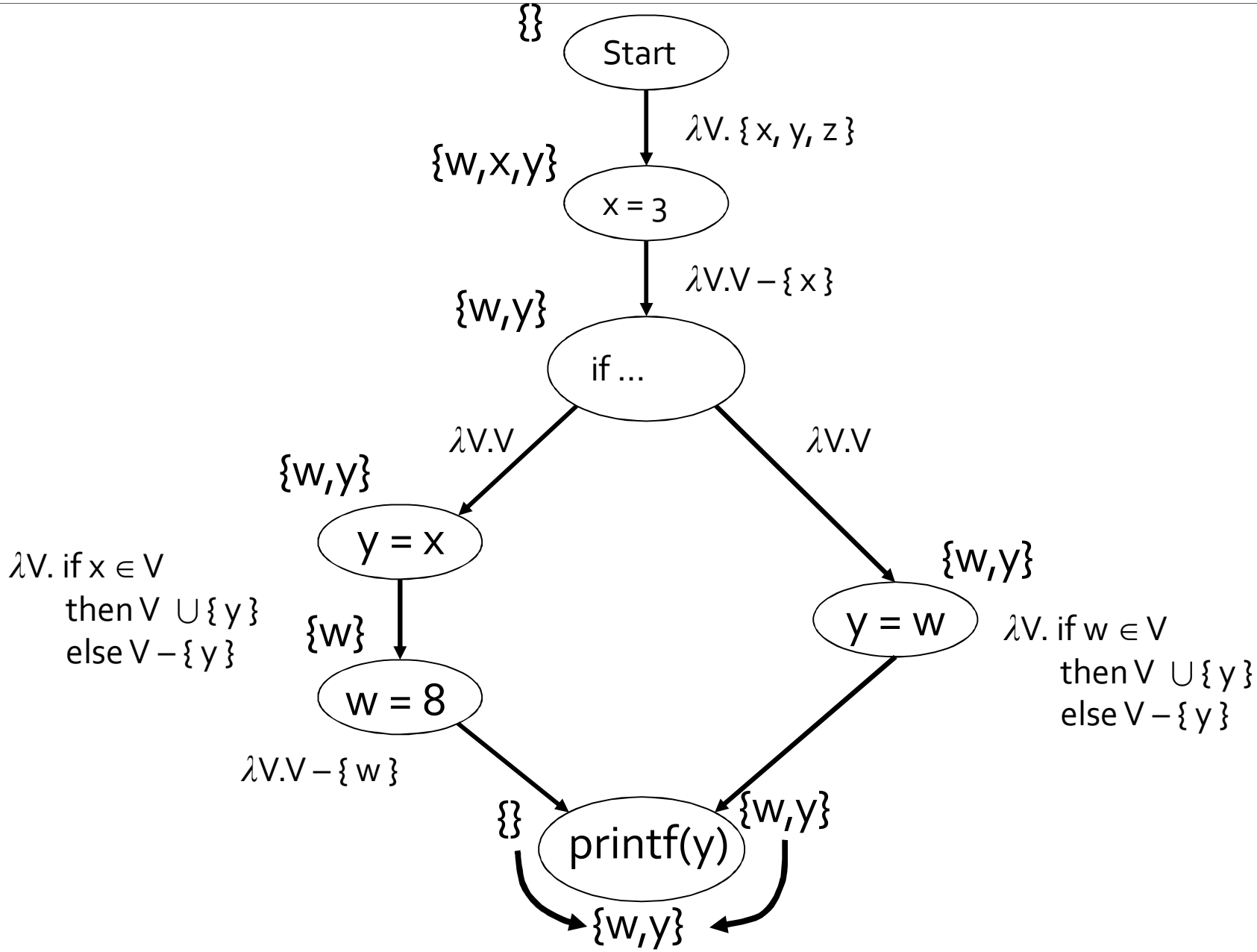
IFDS

- **IFDS** Interprocedural Distributive Finite Subset **Precise interprocedural dataflow analysis via graph reachability.** ***Reps, Horowitz, and Sagiv, POPL'95***

IFDS Problems

- Finite subset distributive
  - Lattice L = $\wp$(D)
  - $\sqsubseteq$ is $\subseteq$
  - $\sqcup$ is $\cup$
  - Transfer functions are distributive
- Efficient solution through formulation as CFL reachability

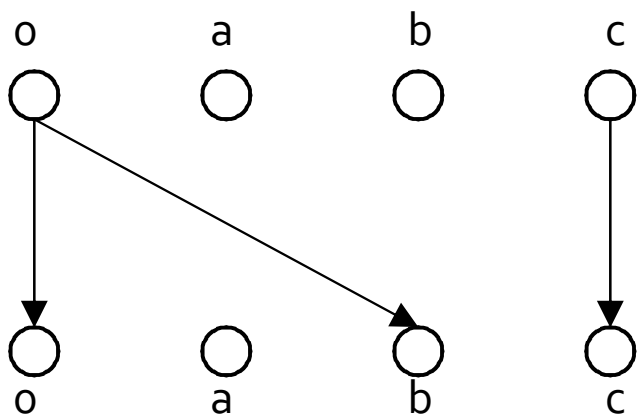
Possibly Uninitialized Variables



Encoding Transfer Functions

- Enumerate all input space and output space
- Represent functions as graphs with 2(D+1) nodes
- Special symbol "o" denotes empty sets (sometimes denoted Λ)
- Example: D = { a, b, c }

  f(S) = (S – {a}) U {b}

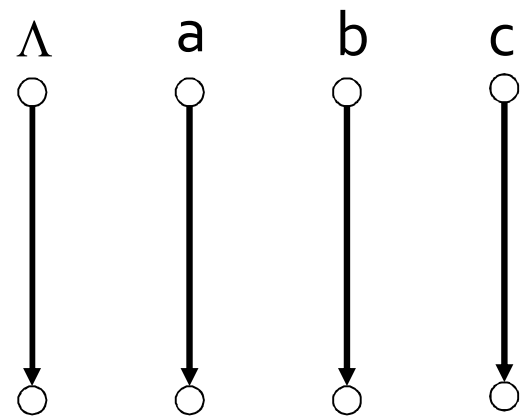


Representing Dataflow Functions

Identity Function

$$\mathrm{f} = \lambda V.V$$

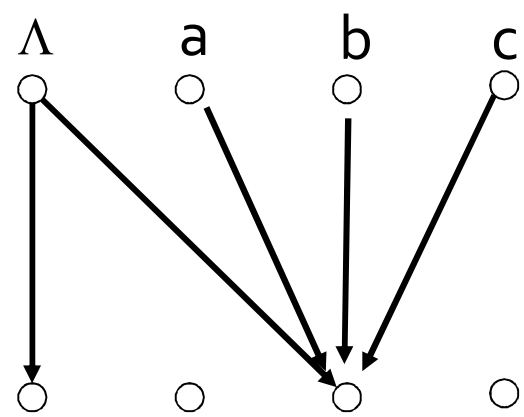
$$\boxed{\mathrm{f}(\{a,b\}) = \{a,b\}}$$



Constant Function

$$\mathrm{f} = \lambda V.\{b\}$$

$$\boxed{\mathrm{f}(\{a,b\}) = \{b\}}$$

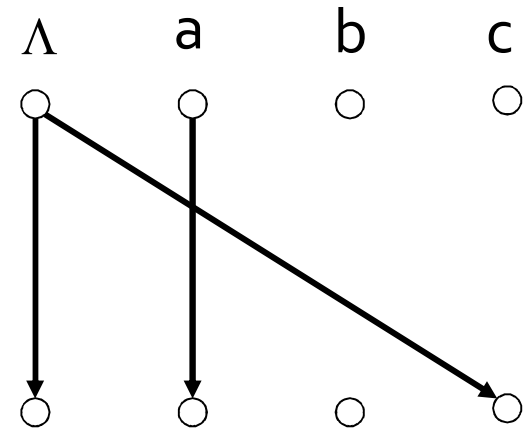


Representing Dataflow Functions

"Gen/Kill" Function

$$\mathrm{f} = \lambda V.(V - \{b\}) \cup \{c\}$$

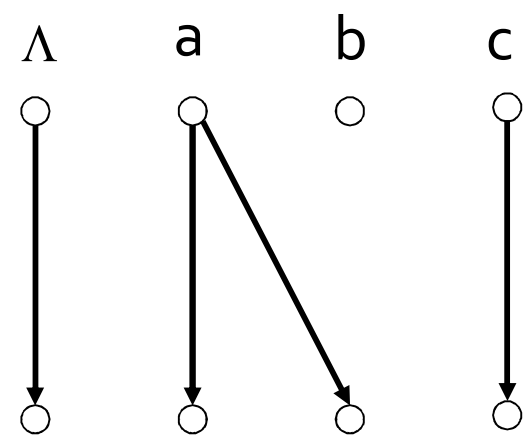
$$\boxed{\mathrm{f}(\{a,b\}) = \{a,c\}}$$



Non-"Gen/Kill" Function

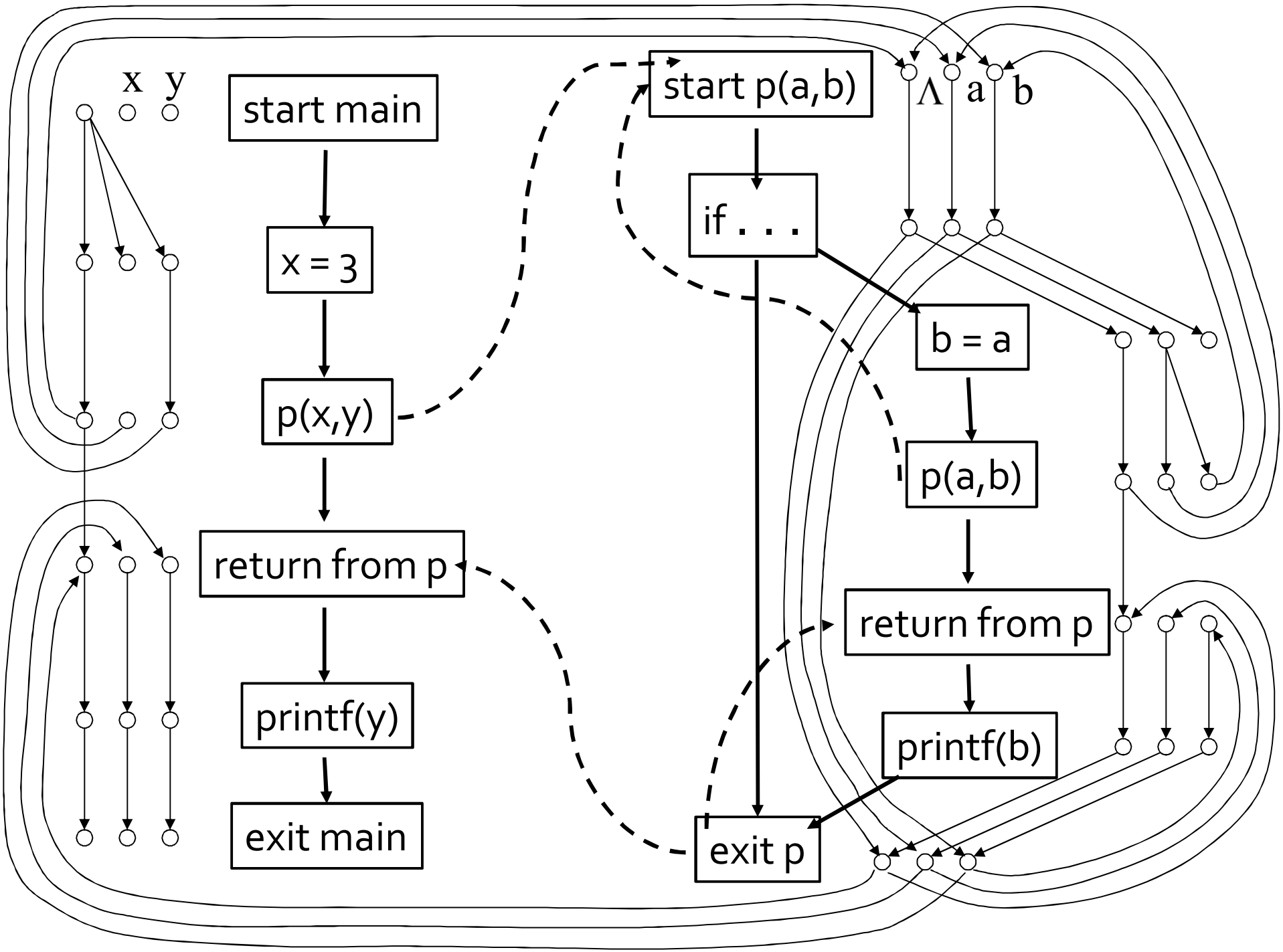
$$\mathrm{f} = \lambda V.\text{if } a \in V$$
$$\text{then } V \cup \{b\}$$
$$\text{else } V - \{b\}$$

$$\boxed{\mathrm{f}(\{a,b\}) = \{a,b\}}$$



Exploded Supergraph

- Exploded supergraph
  - Start with supergraph
  - Replace each node by its graph representation
  - Add edges between corresponding elements in D at consecutive program points
- CFL reachability
  - Finding MOVP solution is equivalent to computing CFL reachability over the exploded supergraph using the valid parentheses grammar.


x
y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
Λ
a
b

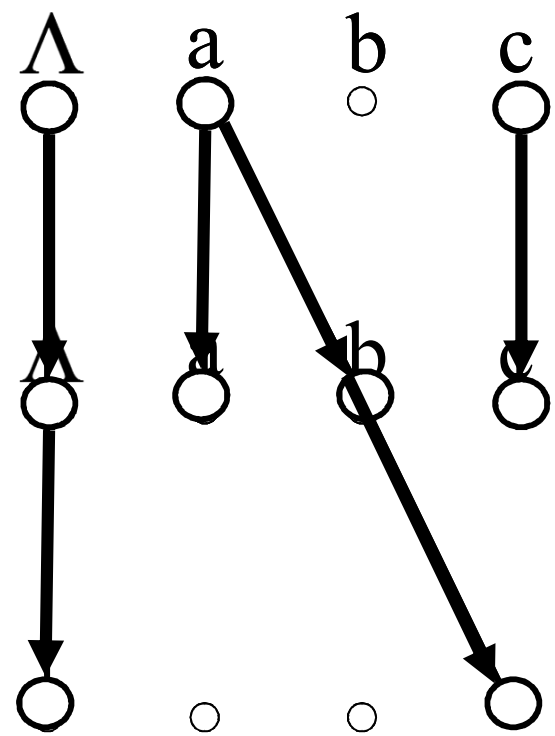
The Tabulation Algorithm

- Worklist algorithm, start from entry of "main"
- Keep track of
  - Path edges: matched paren paths from procedure entry
  - Summary edges: matched paren call-return paths
- At each instruction
  - Propagate facts using transfer functions; extend path edges
- At each call
  - Propagate to procedure entry, start with an empty path
  - If a summary for that entry exits, use it
- At each exit
  - Store paths from corresponding call points as summary paths
  - When a new summary is added, propagate to the return node

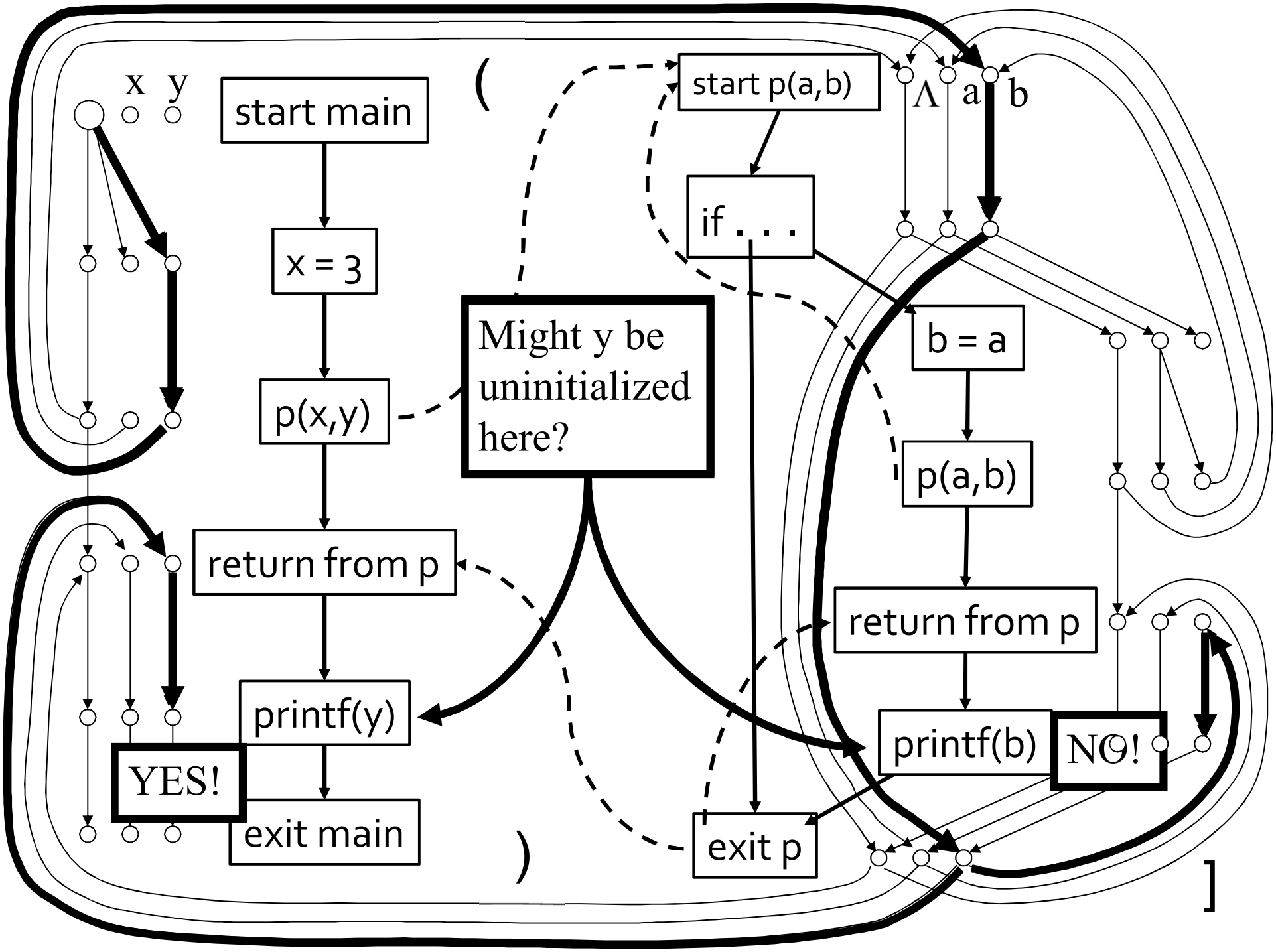
Composing Dataflow Functions

$$\mathrm{f}_1 = \lambda V.\text{if } a \in V \text{ then } V \cup \{b\} \text{ else } V - \{b\}$$

$$\mathrm{f}_2 = \lambda V.\text{if } b \in V \text{ then } \{c\} \text{ else } \phi$$



$$\mathrm{f}_2 \circ \mathrm{f}_1(\{a,c\}) = \boxed{\{c\}}$$


x
y
start main
x = 3
p(x,y)
return from p
printf(y)
YES!
exit main
(
)
Might y be uninitialized here?
start p(a,b)
if . . .
exit p
Λ
a
b
b = a
p(a,b)
return from p
printf(b)
NO!
]

Interprocedural Dataflow Analysis via CFL-Reachability

- Graph: Exploded control-flow graph
- *L*: *L*(*unbalLeft*)
  - unbalLeft = valid
- Fact *d* holds at *n* iff there is an *L*(*unbalLeft*)-path from <Start main,Λ> to <n,d>

Asymptotic Running Time

- CFL-reachability
  - Exploded control-flow graph: *ND* nodes
  - Running time: $O(N^3D^3)$
- Exploded control-flow graph $\Rightarrow$ Special structure

Running time: $O(ED^3)$

Typically: $E \approx N$, hence $O(ED^3) \approx O(ND^3)$

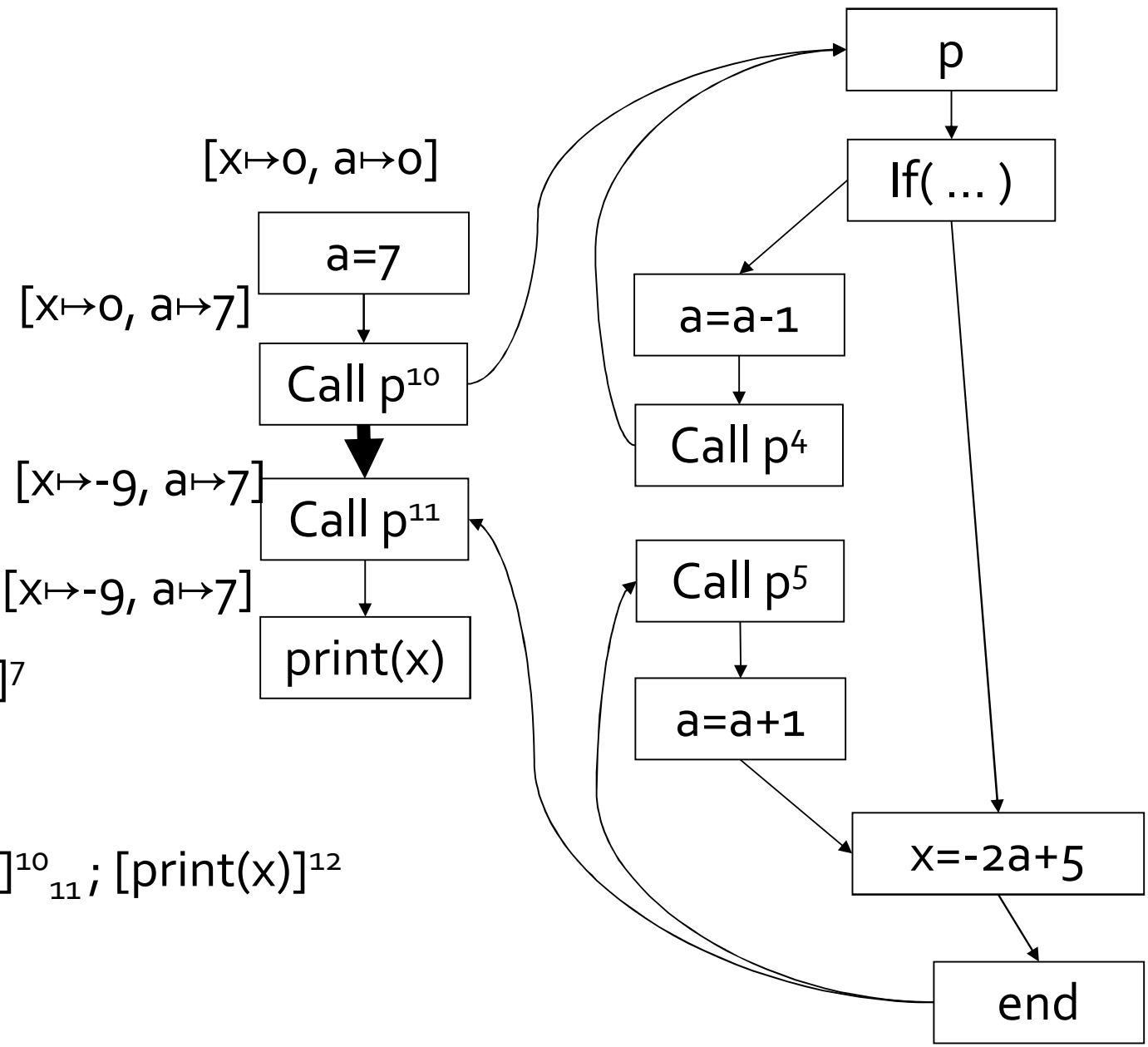
"Gen/kill" problems: $O(ED)$

Recap

- inter-procedural analysis
- inlining
- call-string approach
- functional approach

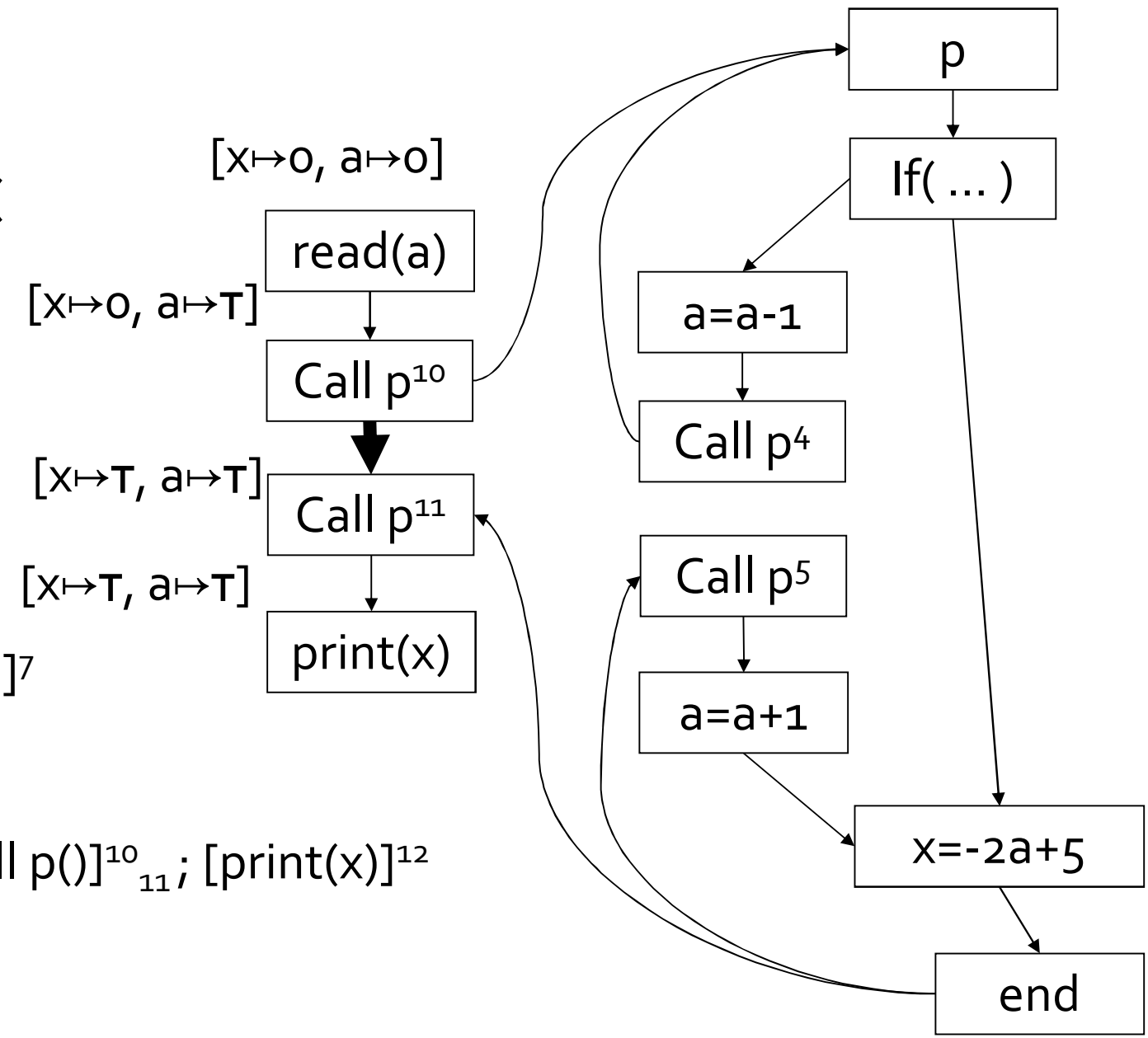
$\lambda e.[x \mapsto -2e(a)+5,\ a \mapsto e(a)]$

```
begin
 proc p() is¹
   if [b]²  then (
    [a := a -1]³
    [call p()]⁴₅
    [a := a + 1]⁶
   )
  [x := -2* a + 5]⁷
end⁸
[a=7]⁹ ; [call p()]¹⁰₁₁ ; [print(x)]¹²
end
```



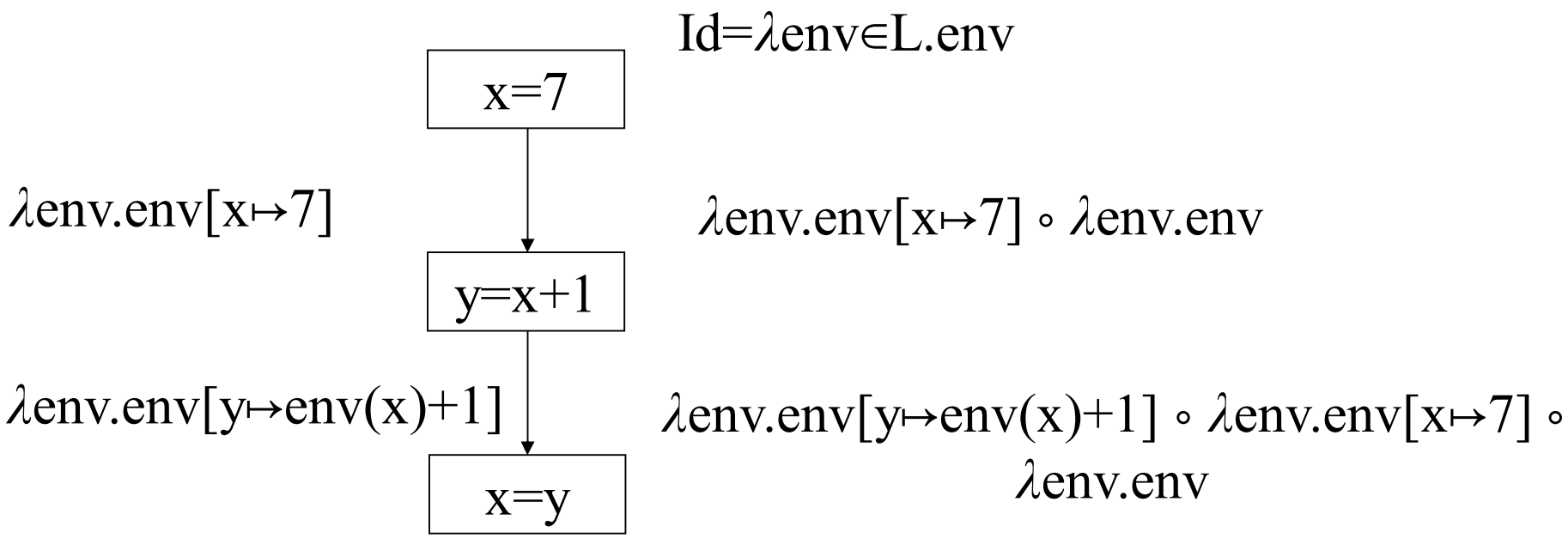
$\lambda e.[x \mapsto -2e(a)+5,\ a \mapsto e(a)]$

```
begin
 proc p() is¹
   if [b]²  then (
    [a := a -1]³
    [call p()]⁴₅
    [a := a + 1]⁶
   )
  [x := -2* a + 5]⁷
end⁸
[read(a)]⁹ ; [call p()]¹⁰₁₁ ; [print(x)]¹²
end
```



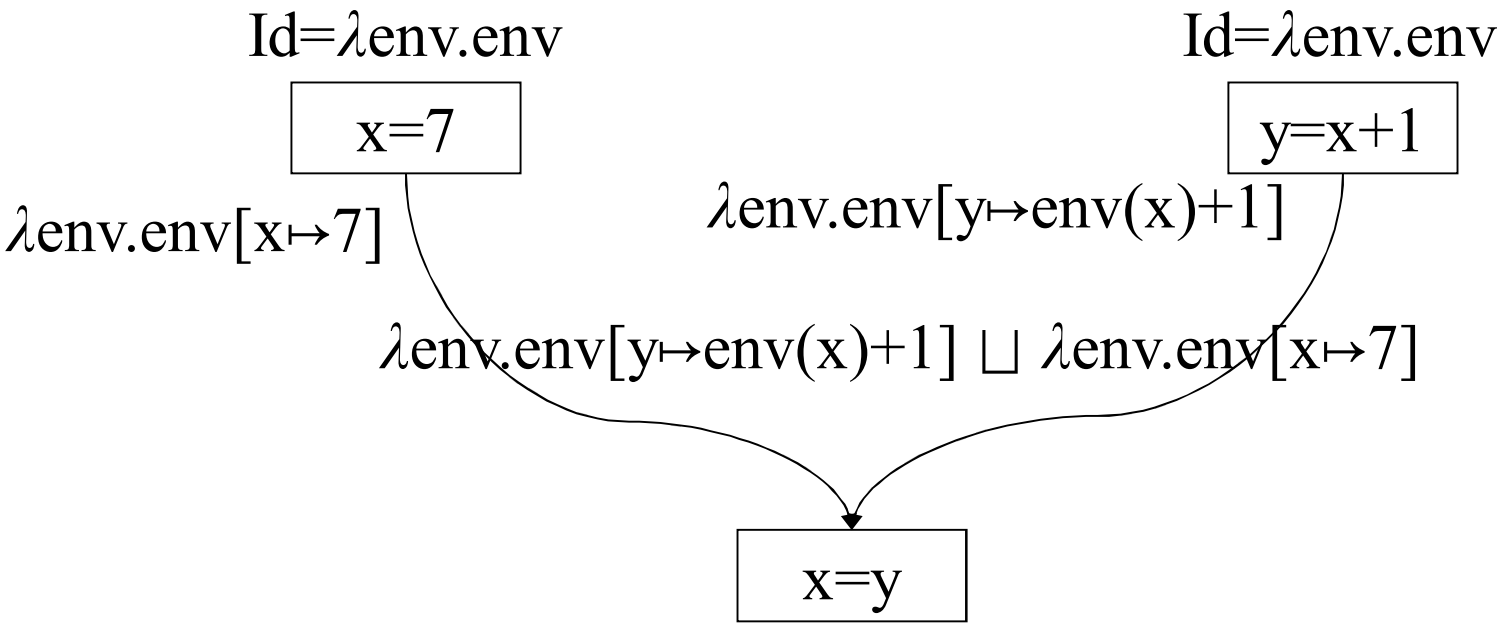
Example: Constant propagation

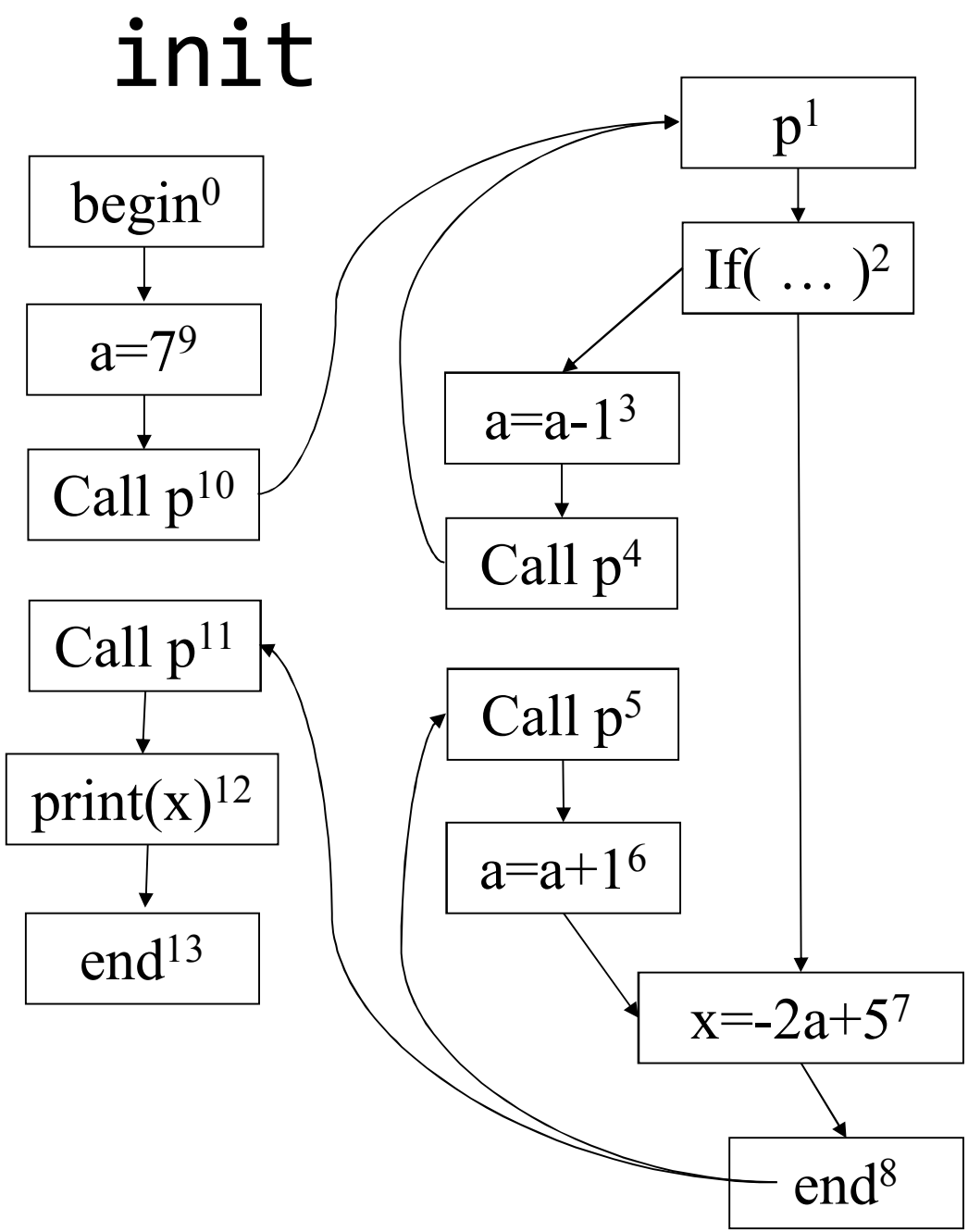
- $L = Var \rightarrow N \cup \{\top, \bot\}$
- Domain: $F: L \rightarrow L$
  - $(f_1 \sqcup f_2)(x) = f_1(x) \sqcup f_2(x)$



Example: Constant propagation

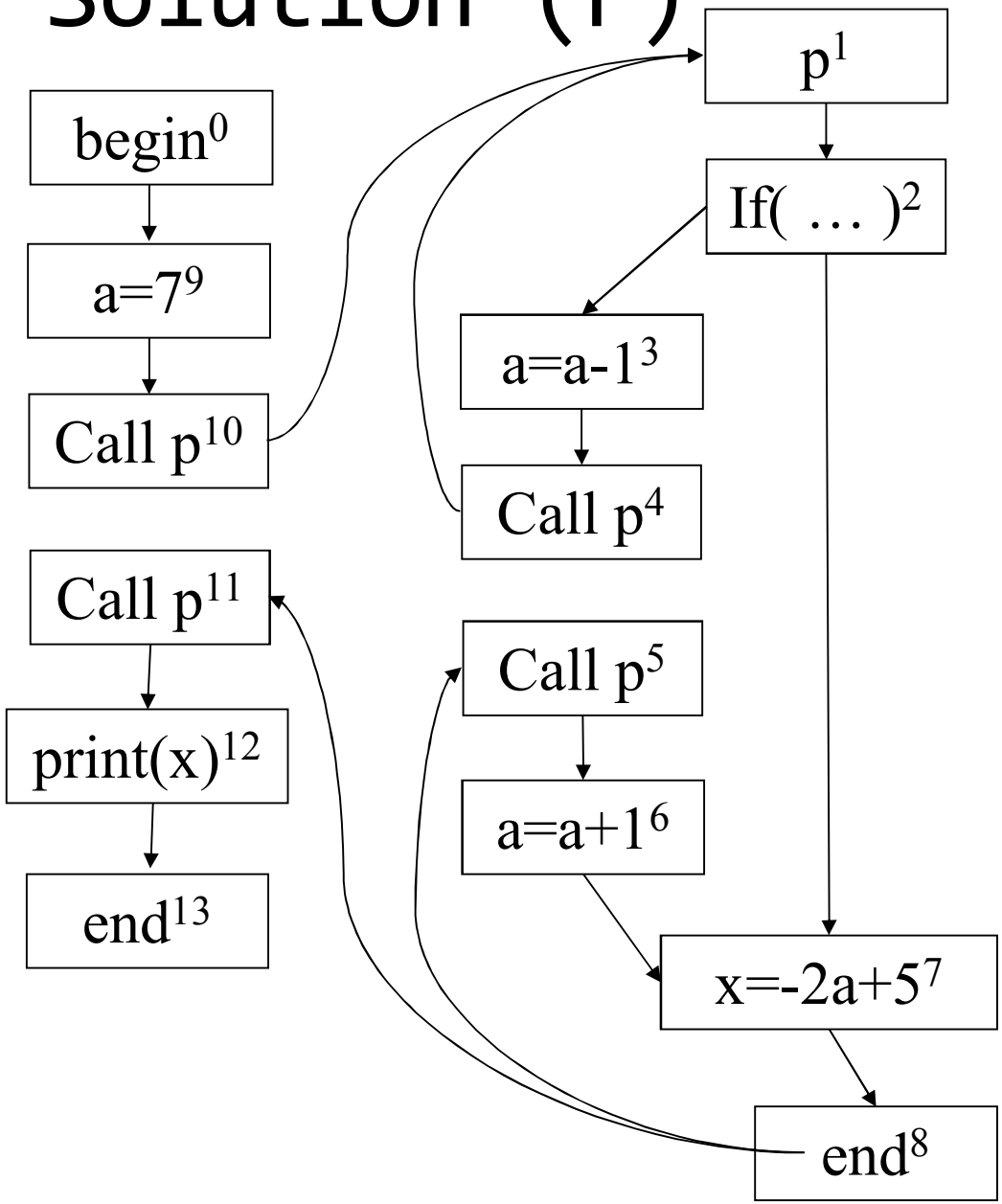
- $L = Var \rightarrow N \cup \{\top, \bot\}$
- Domain: $F: L \rightarrow L$
  - $(f_1 \sqcup f_2)(x) = f_1(x) \sqcup f_2(x)$





N	Function
0	$\lambda e.[x \mapsto e(x), a \mapsto e(a)]=id$
1	$\lambda e.[x \mapsto e(x), a \mapsto e(a)]=id$
3-13	$\lambda e.\perp$

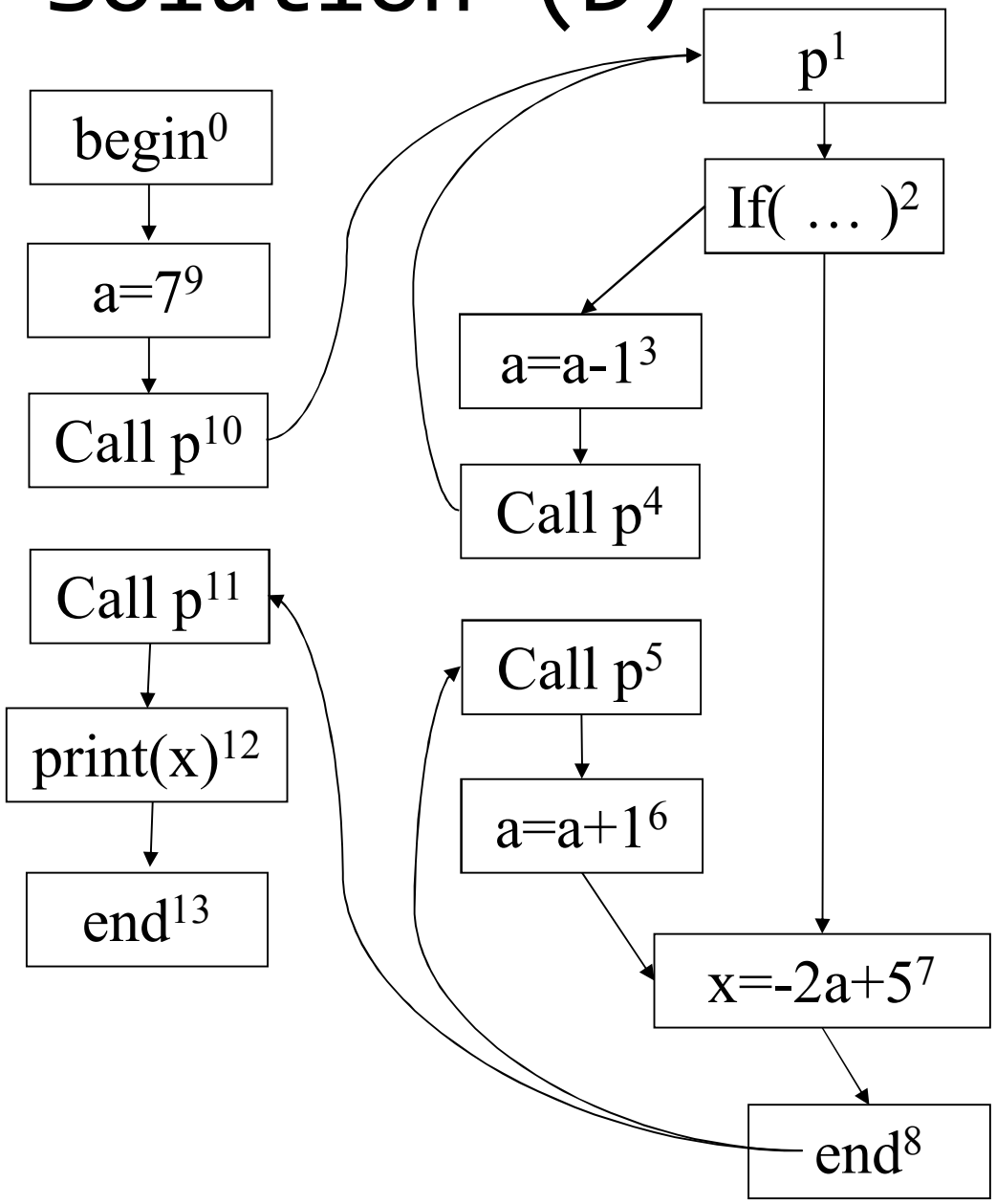
Solution (F)



N	Function
1	$\lambda e.\ [x \mapsto e(x),\ a \mapsto e(a)] = id$
2	id
7	id
8	$\lambda e.[x \mapsto -2e(a)+5,\ a \mapsto e(a)]$
3	id
4	$\lambda e.[x \mapsto e(x),\ a \mapsto e(a)-1]$
5	$f_8 \circ \lambda e.[x \mapsto e(x),\ a \mapsto e(a)-1] =$ $\lambda e.[x \mapsto -2(e(a)-1)+5,\ a \mapsto e(a)-1]$
6	$\lambda e.[x \mapsto -2(e(a)-1)+5,\ a \mapsto e(a)-1]$
7	$\lambda e.[x \mapsto -2(e(a)-1)+5,\ a \mapsto e(a)] \sqcup$ $\lambda e.[x \mapsto e(x),\ a \mapsto e(a)]$
8	$\lambda a, x.[x \mapsto -2e(a)+5,\ a \mapsto e(a)]$

0	$\lambda e.[x \mapsto e(x),\ a \mapsto e(a)] = id$
10	$\lambda e.[x \mapsto e(x),\ a \mapsto 7]$
11	$\lambda a, x.[x \mapsto -2e(a)+5,\ a \mapsto e(a)] \circ f_{10}$

Solution (D)



N	Function
1	[x↦0, a↦7]
2	[x↦0, a↦7]
7	[x↦0, a↦7]
8	[x↦-9, a↦7]
3	[x↦0, a↦7]
4	[x↦0, a↦6]
1	[x↦-7, a↦6]
6	[x↦-7, a↦7]
7	[x↦т, a↦7]
8	[x↦-9, a↦7]
1	[x↦т, a↦т]

0	[x↦0, a↦0]
10	[x↦0, a↦7]
11	[x↦-9, a↦7]

Interprocedural Analysis

begin

proc p() is^{1}

  [x := a + 1]^{2}

end^{3}

[a=7]^{4}

[call p()]$^{5}_{6}$

[print x]^{7}

[a=9]^{8}

[call p()]$^{9}_{10}$

[print a]^{11}

end

- Extend language with begin/end and with [call p()]$^{clab}_{rlab}$
- Call label clab, and return label rlab

Work-list Algorithm

```
Chaotic(G(V, E): CFG, s: Node, L: Lattice, ι: L, f: E →(L →L)){
  for each v in V to n do df_entry[v] := ⊥
  df[v] = ι
  WL = {s}
  while (WL ≠ Ø ) do
     select and remove an element u ∈ WL
     for each v, such that (u, v) ∈E do
          temp = f(e)(df_entry[u])
          new := df_entry(v) ⊔ temp
          if (new ≠ df_entry[v]) then
               df_entry[v] := new;
               WL := WL ∪ {v}
```

Complexity of Chaotic Iterations

- Parameters
  - n the number of CFG nodes
  - k is the maximum outdegree of edges
  - A lattice of height h
  - c is the maximum cost of
    - applying $f_{(e)}$
    - $\sqcup$
    - L comparisons

- Complexity: O(n * h * c * k)

(slide from Mooly Sagiv)