Interprocedural Analysis

Noam Rinetzky

Mooly Sagiv

Why do runtime errors occur?

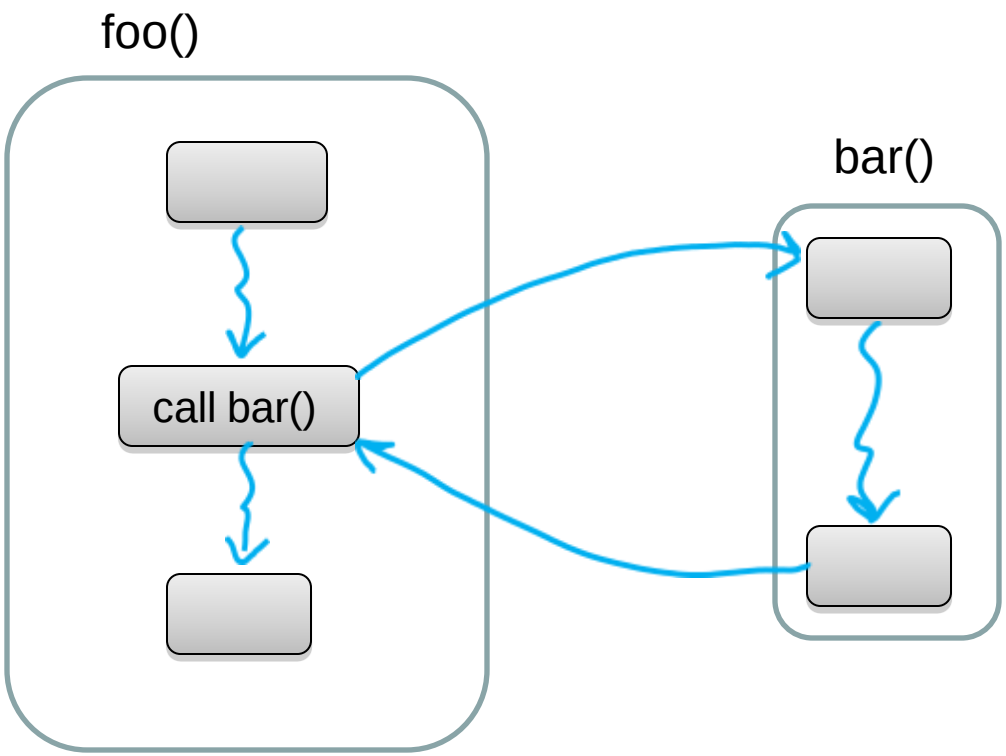
- Logical errors
- Incorrect initializations
- Resource limitations
- Aliasing problems
- API mismatch

Procedural program

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

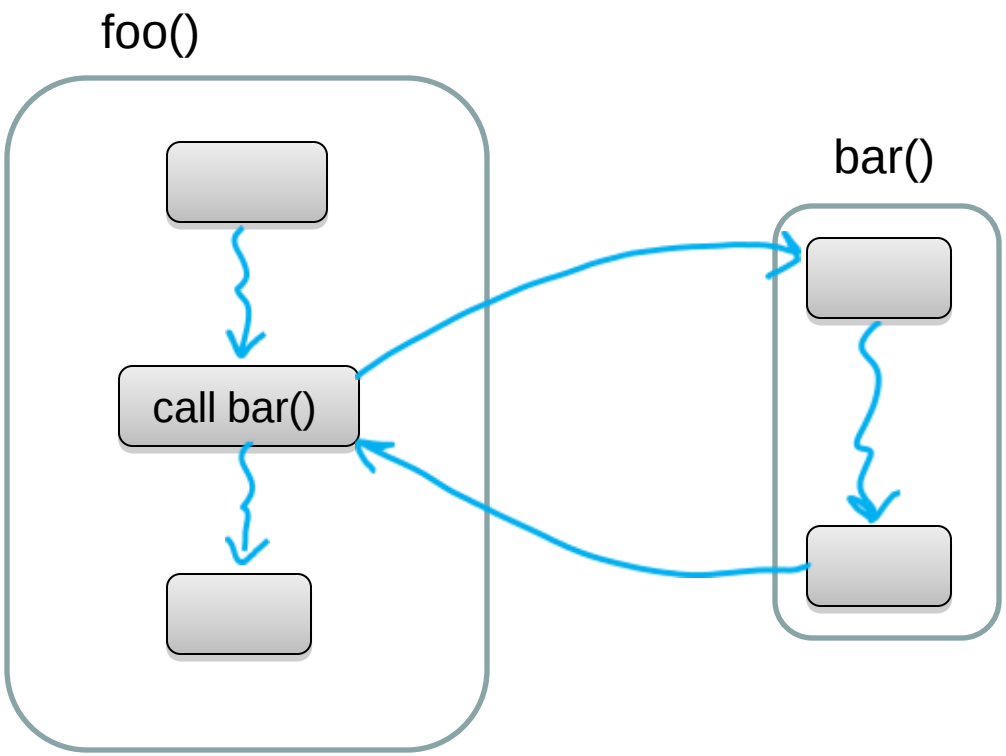
```
int p(int a) {
  return a + 1;
}
```

Effect of procedures



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

Interprocedural Analysis

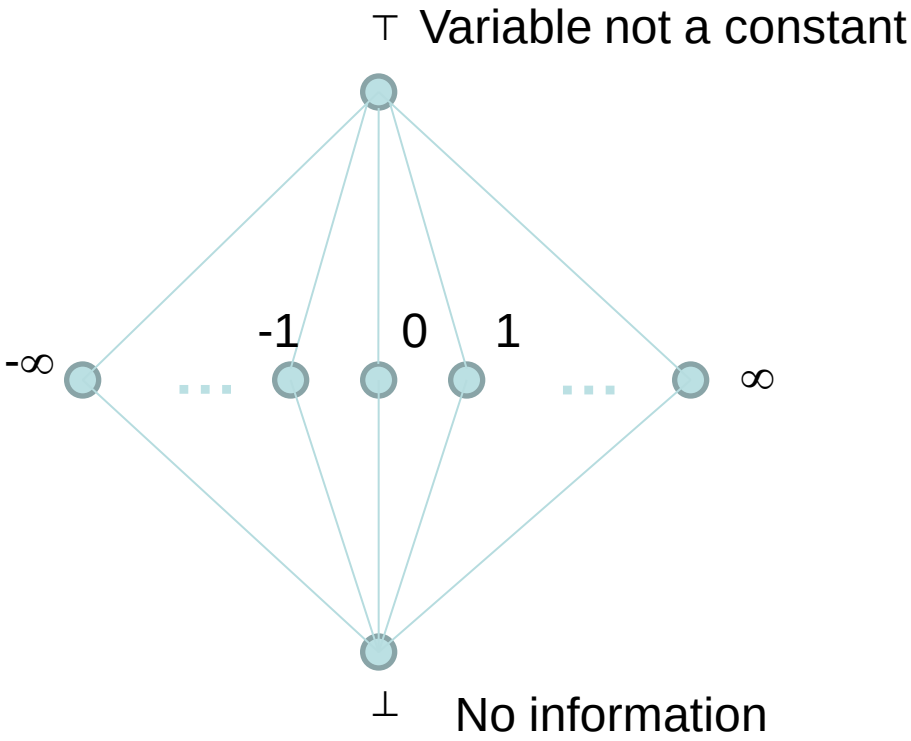


goal: compute the abstract effect of calling a procedure

Reduction to intraprocedural analysis

- Procedure inlining
- Naive solution: call-as-goto

Reminder: Constant Propagation



$Z_{\perp}^{\top}$

Reminder: Constant Propagation

- $L = (Var \rightarrow Z_{\perp}^{\top}, \sqsubseteq)$
- $\sigma_1 \sqsubseteq \sigma_2$ iff $\forall x$: $\sigma_1(x) \sqsubseteq' \sigma_2(x)$
  - $\sqsubseteq'$ ordering in the $Z_{\perp}^{\top}$ lattice

- Examples:
  - $[x \mapsto \perp, y \mapsto 42, z \mapsto \perp] \sqsubseteq [x \mapsto \perp, y \mapsto 42, z \mapsto 73]$
  - $[x \mapsto \perp, y \mapsto 42, z \mapsto 73] \sqsubseteq [x \mapsto \perp, y \mapsto 42, z \mapsto \top]$

Reminder: Constant Propagation

- Conservative Solution
  - Every detected constant is indeed constant
    - But may fail to identify some constants
  - Every potential impact is identified
    - Superfluous impacts

Procedure Inlining

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

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

Procedure Inlining

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

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

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

Procedure Inlining

- Pros
  - Simple
- Cons
  - Does not handle recursion
  - Exponential blow up
  - Reanalyzing the body of procedures

```
p1 {
  call p2
  …
  call p2
}
```

```
p2 {
  call p3
  …
  call p3
}
```

```
p3{
}
```

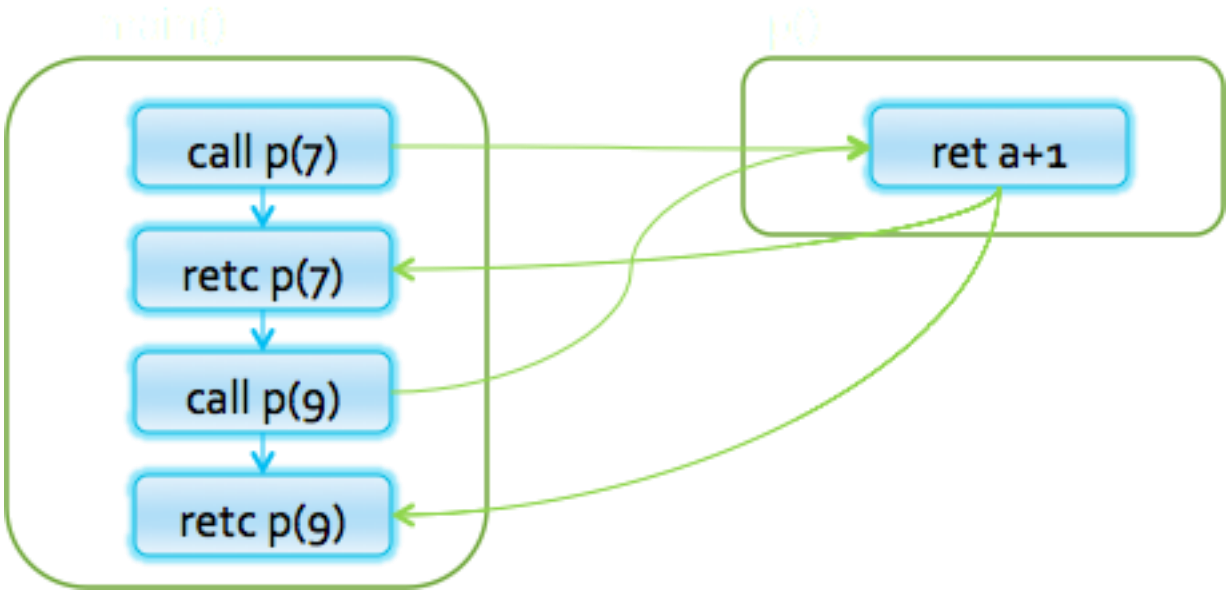
A Naive Interprocedural solution

- Treat procedure calls as gotos

Simple Example

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

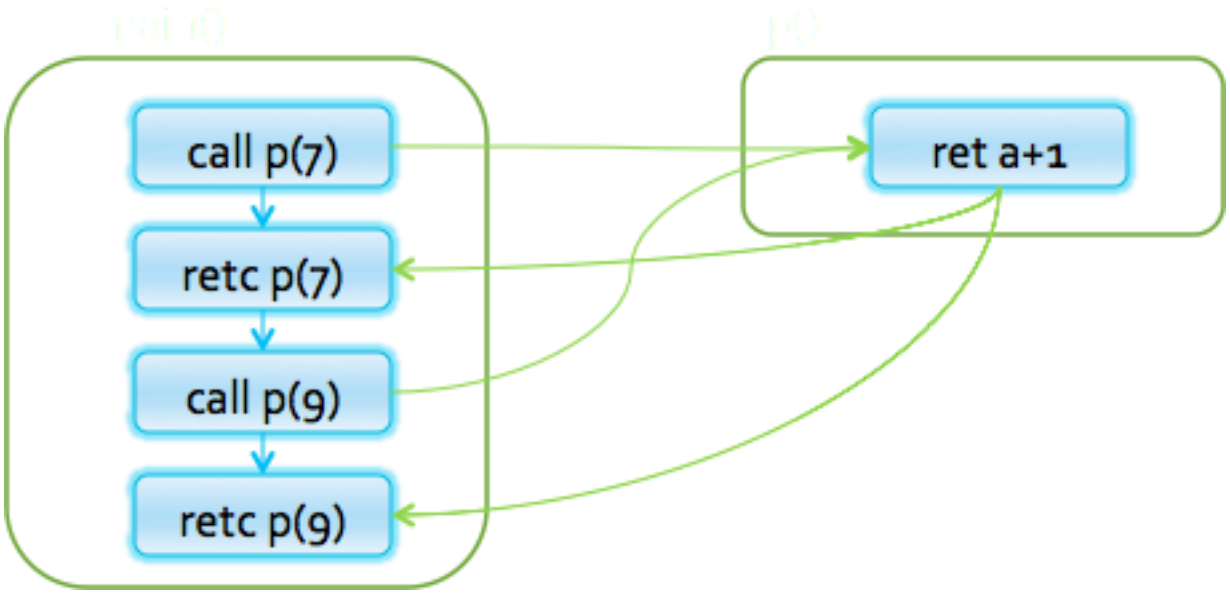
```
int p(int a) {
  return a + 1;
  }
```



Simple Example

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

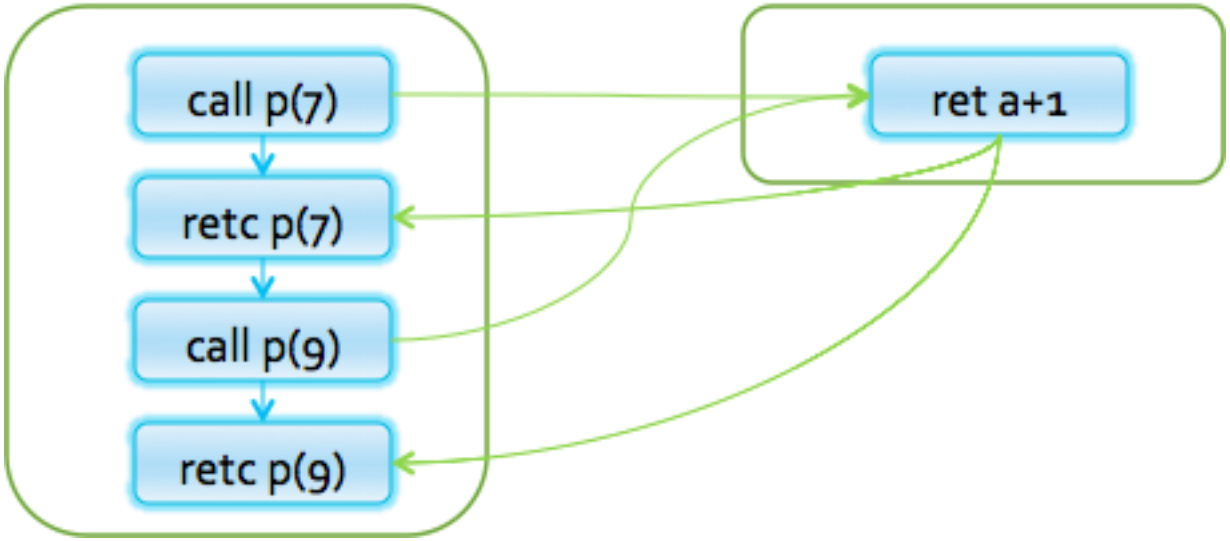
```
int p(int a) {
    [a ↦7]
    return a + 1;
}
```



Simple Example

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

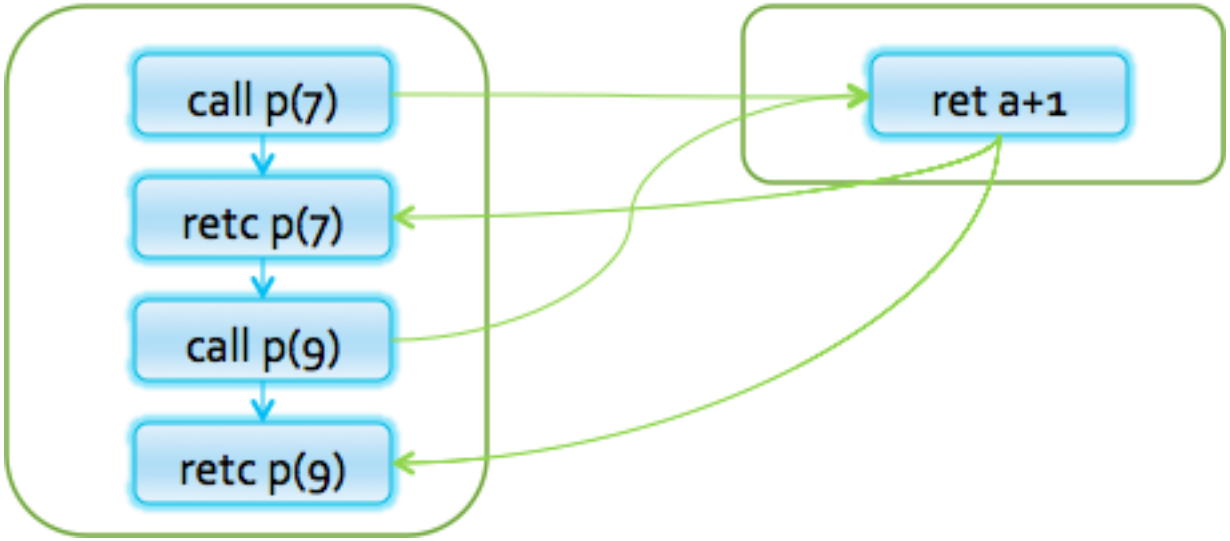
```
int p(int a) {
    [a ↦7]
    return a + 1;
    [a ↦7, $$ ↦8]
}
```



Simple Example

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

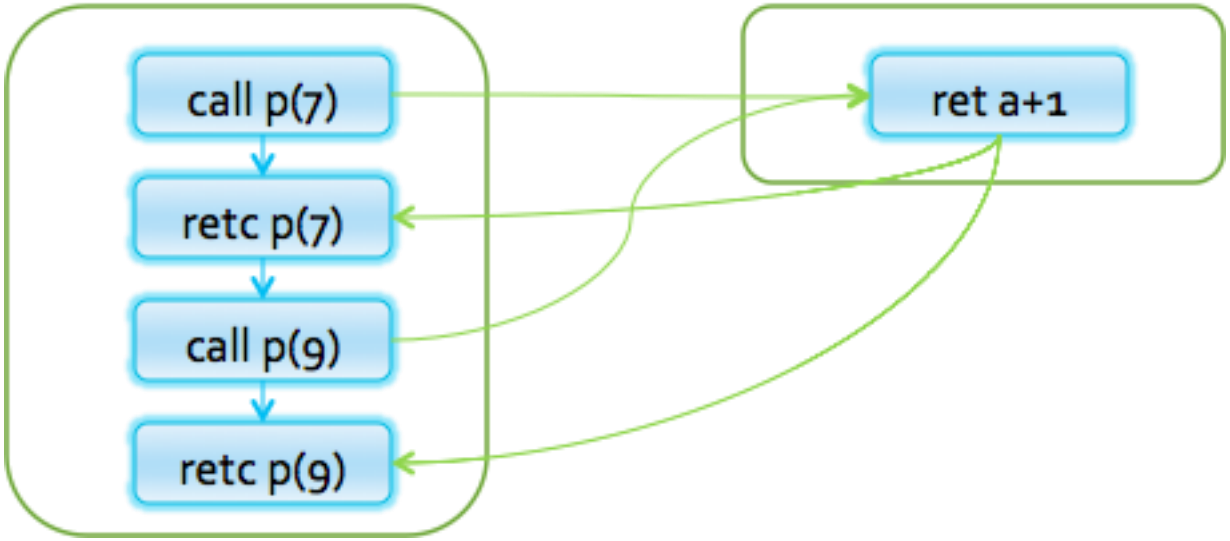
```
int p(int a) {
    [a ↦7]
    return a + 1;
    [a ↦7, $$ ↦8]
}
```



Simple Example

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

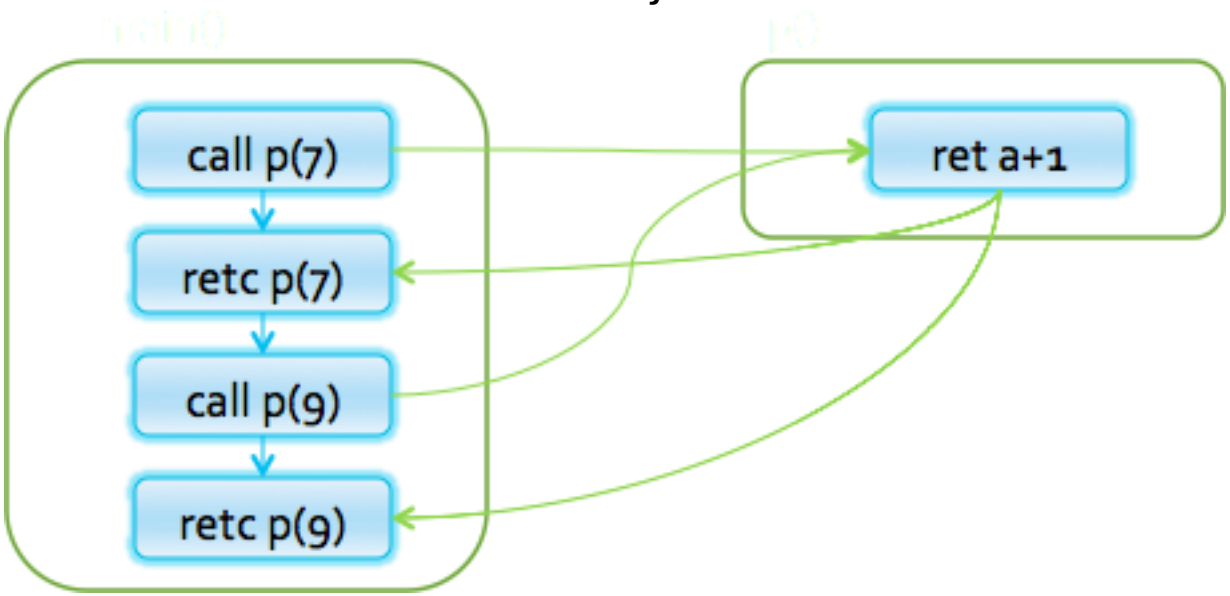
```
int p(int a) {
  [a ↦7]
    return a + 1;
  [a ↦7, $$ ↦8]

}
```



Simple Example

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

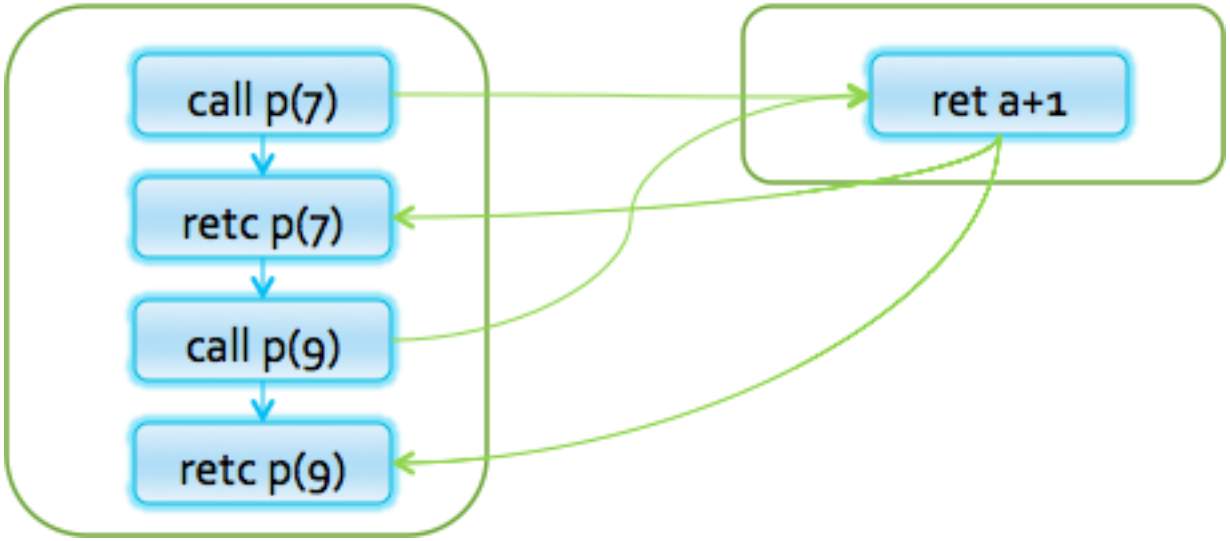
```
int p(int a) {
    [a ↦7]  [a ↦9]
    return a + 1;
    [a ↦7, $$ ↦8]
}
```



Simple Example

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

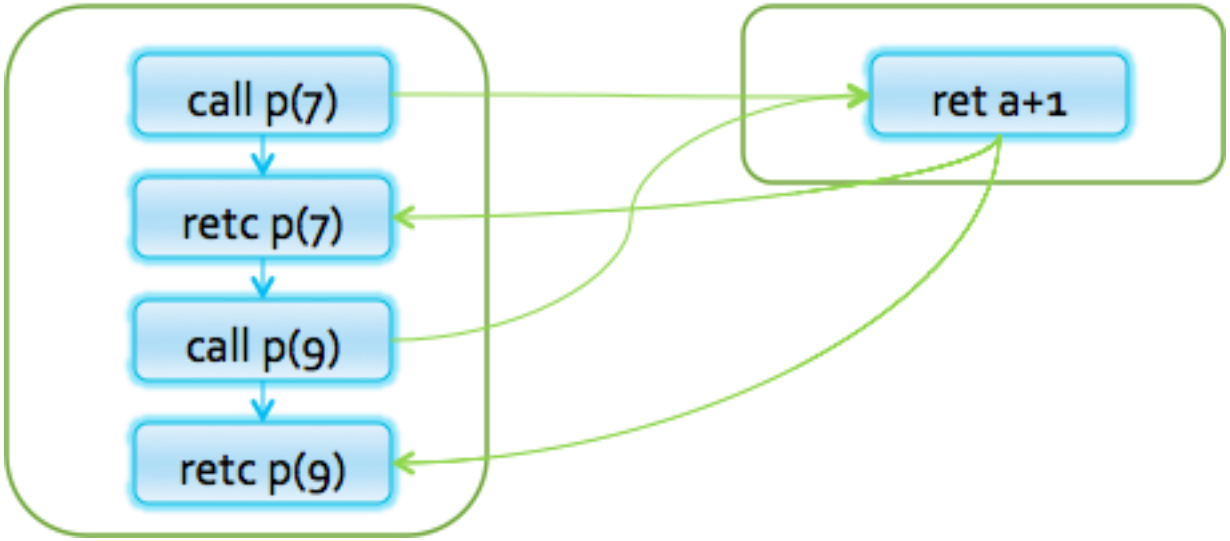
```
int p(int a) {
    [a ↦T]
    return a + 1;
    [a ↦7, $$ ↦8]
}
```



Simple Example

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

}
```

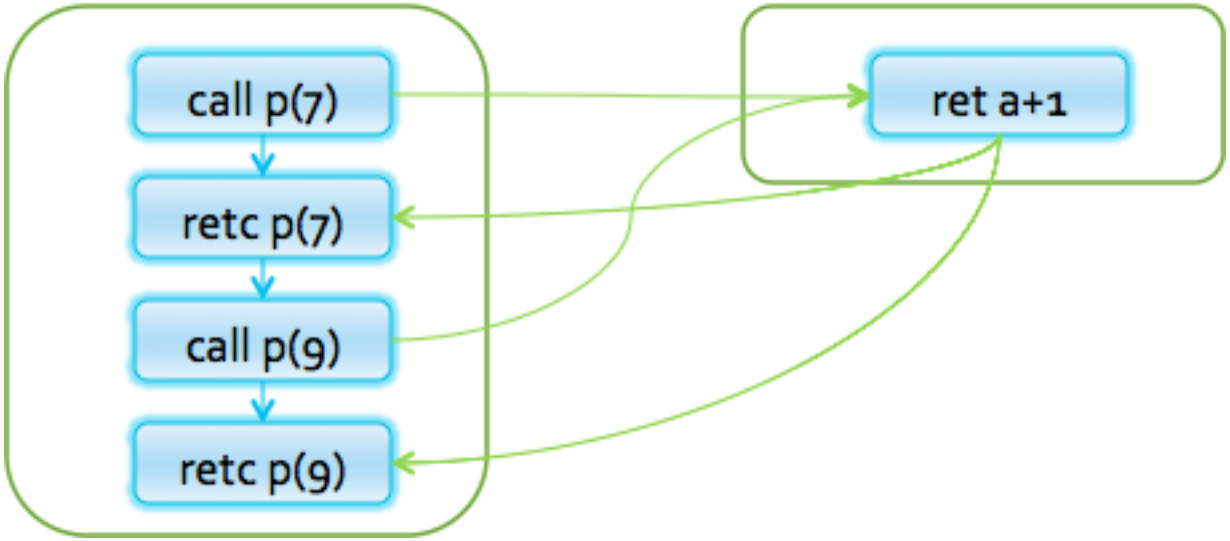
```
int p(int a) {
    [a ↦⊤]
    return a + 1;
    [a ↦⊤, $$ ↦⊤]

}
```



Simple Example

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

}
```

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

}
```



A Naive Interprocedural solution

- Treat procedure calls as gotos
- Pros:
  - Simple
  - Usually fast
- Cons:
  - Abstract call/return correlations
  - Obtain a conservative solution

Analysis by reduction

Call-as-goto

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

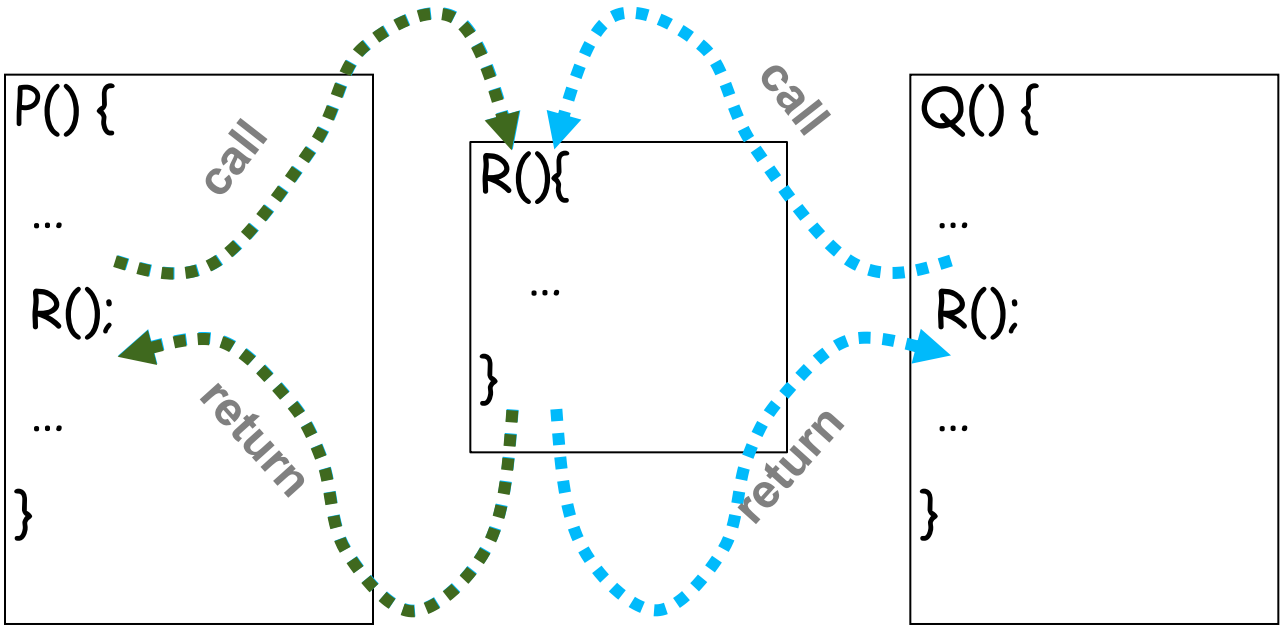
```
int p(int a) {
    [a ↦⊤]
    return a + 1;
    [a ↦⊤, $$ ↦⊤]
}
```

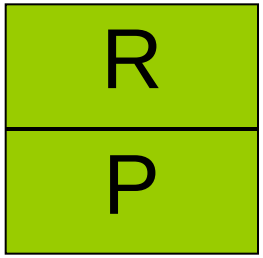
Procedure inlining

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

why was the naive solution less precise?

Stack regime





Guiding light

- Exploit stack regime
  - ➔Precision
  - ➔Efficiency

Simplifying Assumptions

- Parameter passed by value
- No procedure nesting
- No concurrency

✓ Recursion is supported

Topics Covered

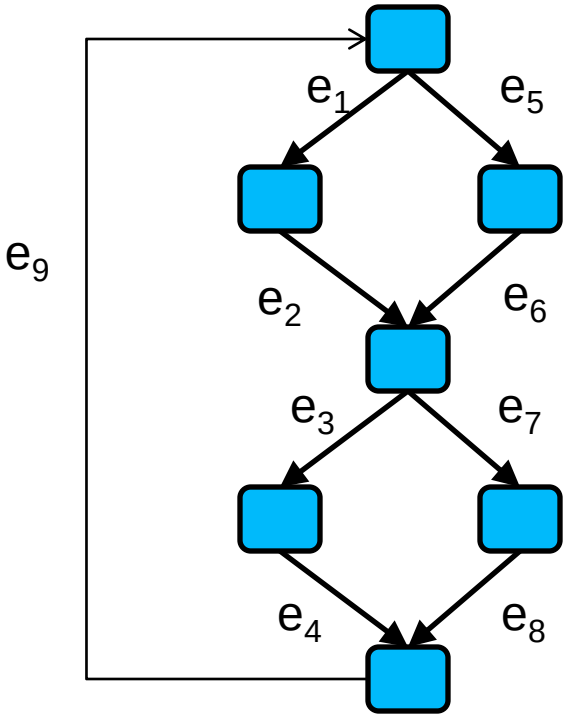
✓ Procedure Inlining
✓ The naive approach
- Valid paths
- The callstring approach
- The Functional Approach
- IFDS: Interprocedural Analysis via Graph Reachability
- IDE: Beyond graph reachability

- The trivial modular approach

Join-Over-All-Paths (JOP)

- Let paths(v) denote the potentially infinite set paths from start to v (written as sequences of edges)
- For a sequence of edges $[e_1, e_2, \ldots, e_n]$ define $f[e_1, e_2, \ldots, e_n]: L \rightarrow L$ by composing the effects of basic blocks
  $f[e_1, e_2, \ldots, e_n](l) = f(e_n)(\ldots(f(e_2)(f(e_1)(l))\ldots)$
- $JOP[v] = \sqcup\{f[e_1, e_2, \ldots, e_n](l) \mid [e_1, e_2, \ldots, e_n] \in paths(v)\}$

Join-Over-All-Paths (JOP)



Paths transformers:

$f[e_1,e_2,e_3,e_4]$
$f[e_1,e_2,e_7,e_8]$
$f[e_5,e_6,e_7,e_8]$
$f[e_5,e_6,e_3,e_4]$
$f[e_1,e_2,e_3,e_4,e_9,\ e_1,e_2,e_3,e_4]$
$f[e_1,e_2,e_7,e_8,e_9,\ e_1,e_2,e_3,e_4,e_9,\ldots]$
…

JOP:

$f[e_1,e_2,e_3,e_4](\text{initial}) \sqcup$
$f[e_1,e_2,e_7,e_8](\text{initial}) \sqcup$
$f[e_5,e_6,e_7,e_8](\text{initial}) \sqcup$
$f[e_5,e_6,e_3,e_4](\text{initial}) \sqcup \ldots$

Number of program paths is unbounded due to loops

LFP approximates JOP

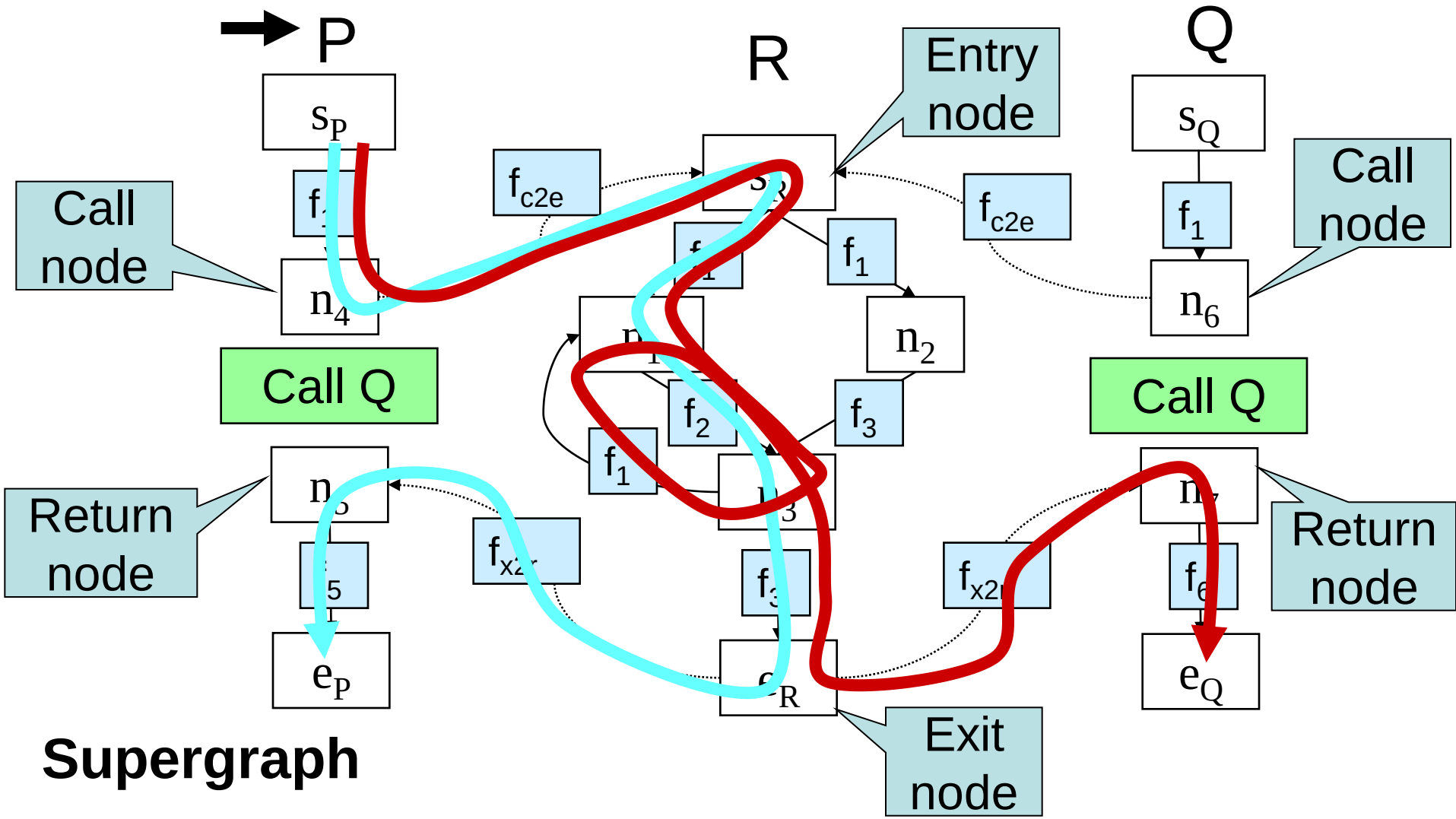
- JOP[v] $= \sqcup\{f\,[e_1, e_2, \ldots, e_n](I) \mid [e_1, e_2, \ldots, e_n] \in \text{paths}(v)\}$
- LFP[v] $= \sqcup\{f\,[e](\text{LFP}[v']) \mid e = (v', v)\}$

  LFP[v_0] $= \iota$

- JOP $\sqsubseteq$ LFP - for a monotone function
  - $f(x \sqcup y) \sqsupseteq f(x) \sqcup f(y)$
- JOP = LFP - for a distributive function
  - $f(x \sqcup y) = f(x) \sqcup f(y)$

JOP may not be precise enough for interprocedural analysis!

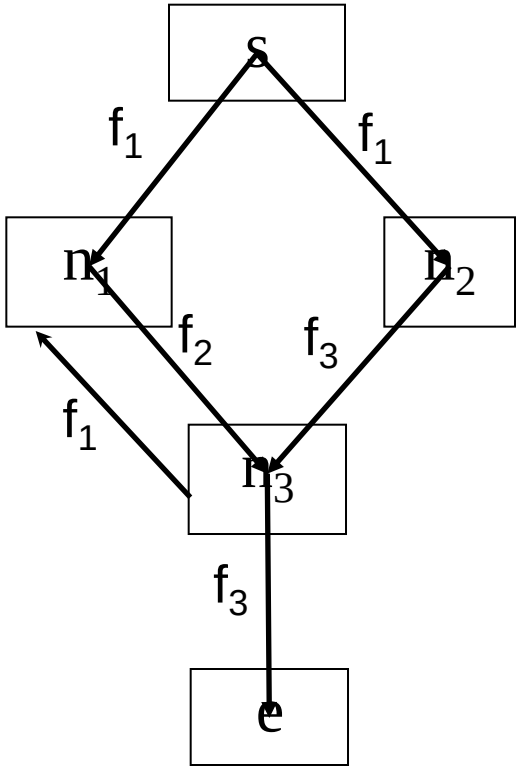
Interprocedural analysis



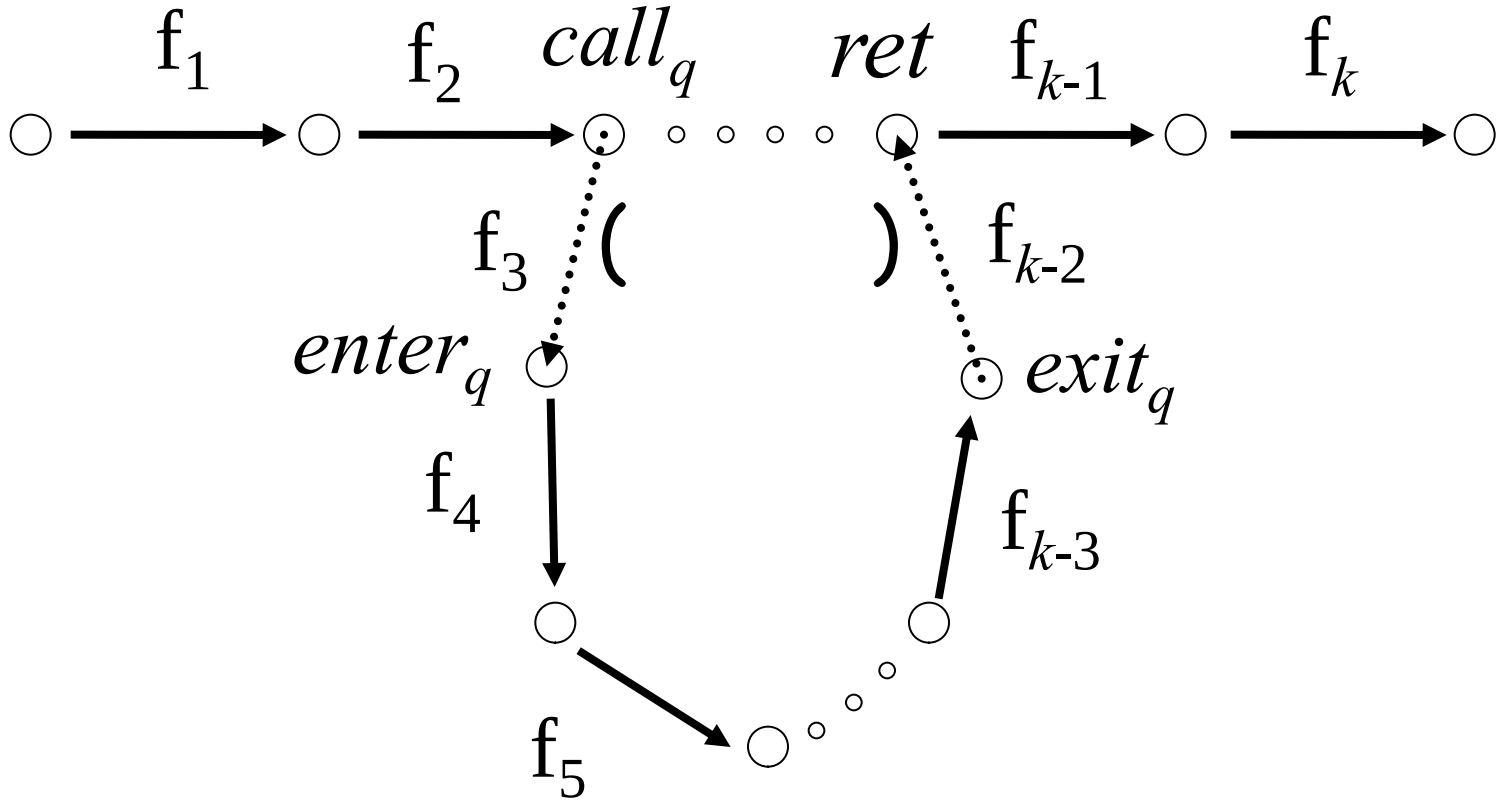
Supergraph

Paths

- **paths(n)** the set of paths from s to n
  - ((s,n_1), (n_1,n_3), (n_3,n_1))



Interprocedural Valid Paths

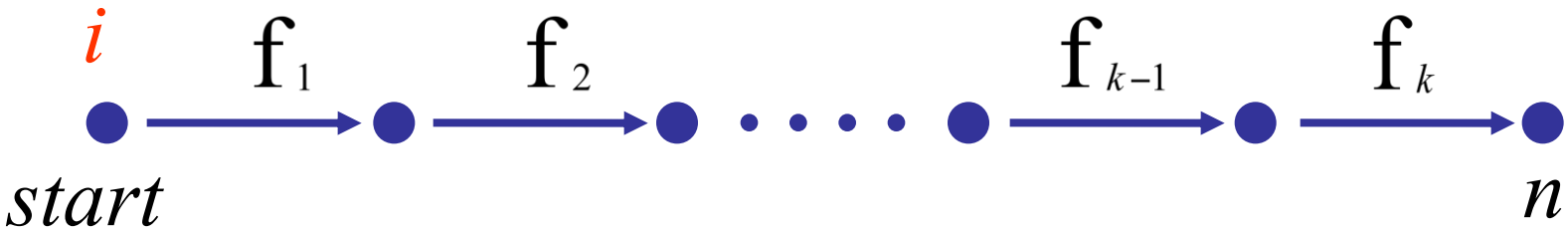


- IVP: all paths with matching calls and returns
  - And prefixes

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

Join Over All Paths (JOP)



$$[\![f_k \circ \ldots \circ f_1]\!] \in L \rightarrow L$$

- JOP[v] $= \sqcup\{[[e_1, e_2, \ldots, e_n]](\bot) \mid (e_1, \ldots, e_n) \in \text{paths}(v)\}$
- JOP $\sqsubseteq$ LFP
  - Sometimes JOP = LFP
    - precise up to "**symbolic execution**"
    - Distributive problem

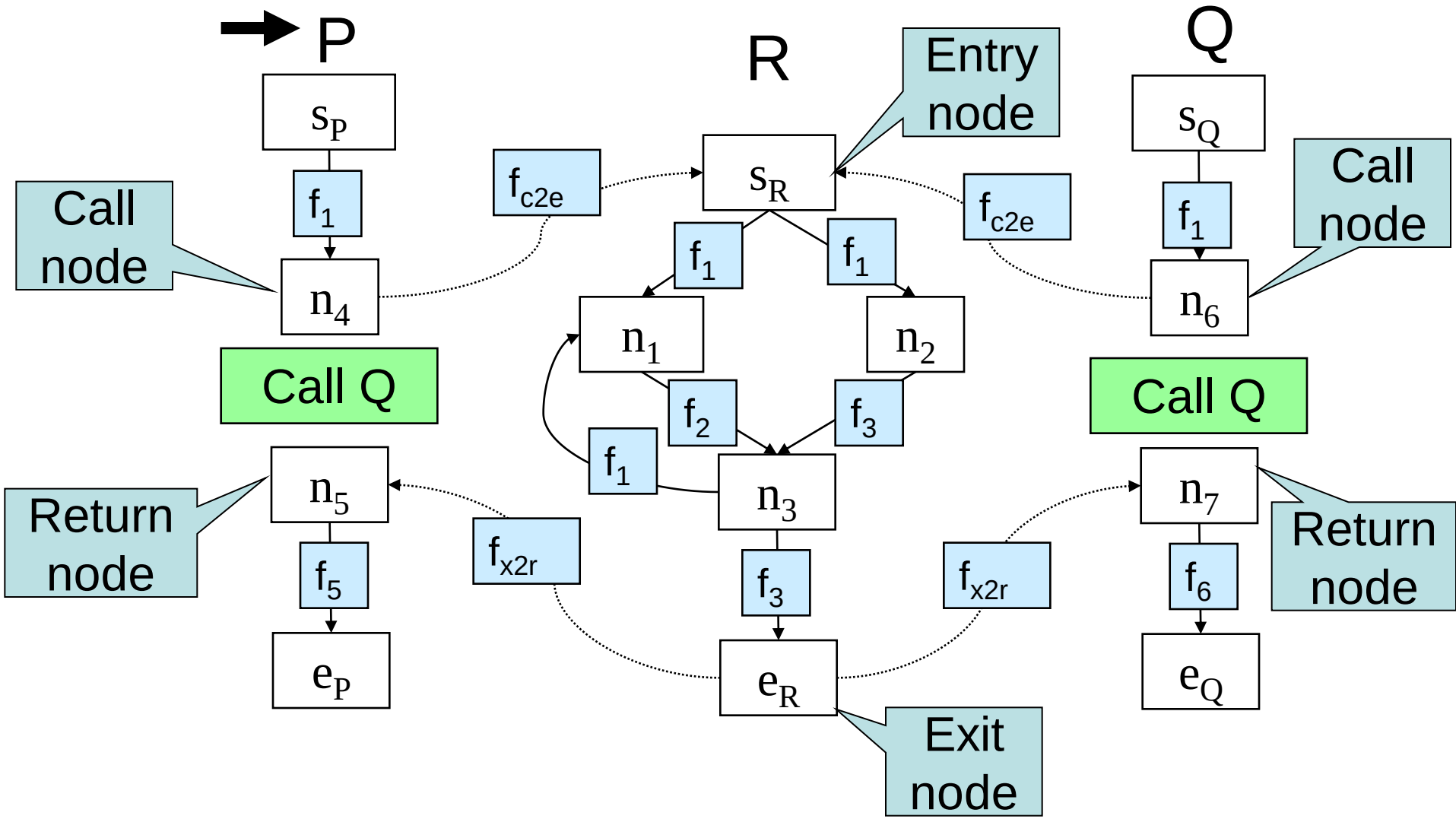
The Join-Over-Valid-Paths (JVP)

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

The Call-String Approach

- The data flow value is associated with sequences of calls (call string)
- Use Chaotic iterations over the supergraph

Supergraph


P
R
Q
Entry node
Call node
Call node
Call Q
Call Q
Return node
Return node
Exit node
s_P
f_1
n_4
n_5
f_5
e_P
f_{c2e}
f_{x2r}
s_R
f_1
f_1
n_1
n_2
f_2
f_3
f_1
n_3
f_3
e_R
f_{c2e}
f_{x2r}
s_Q
f_1
n_6
n_7
f_6
e_Q

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]
}
```

The Call-String Approach

- The data flow value is associated with sequences of calls (call string)
- Use Chaotic iterations over the supergraph
- To guarantee termination limit the size of call string (typically 1 or 2)
  - Represents tails of calls

- Abstract inline

Another Example (|cs|=2)

```
void main() {
    int x ;
    c1: x = p(7);
     ε : [x ↦ 16]
    c2: x = p(9) ;
     ε : [x ↦ 20]
}
```

```
int p(int a) {
   c1:[a ↦7]
   c2:[a ↦9]
    return c3: p1(a + 1);
   c1:[a ↦7, $$ ↦16]
   c2:[a ↦9, $$ ↦20]
}
```

```
int p1(int b) {
   c1.c3:[b ↦8]
   c2.c3:[b ↦10]
    return 2 * b;
   c1.c3:[b ↦8,$$↦16]
   c2.c3:[b ↦10,$$↦20]
}
```

Another Example (|cs|=1)

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

```
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]
}
```

Handling 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

- Easy to implement
- Efficient for very small call strings
- Limited precision
  - Often loses precision for recursive programs
  - For finite domains can be precise even with recursion (with a bounded callstring)

- Order of calls can be abstracted
- Related method: procedure cloning

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
- Relation between input and output
- In certain cases can compute JVP

The Functional Approach

- 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

Phase 1

```
void main() {

   p(7);

}
```

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

```
int p(int a) {
```

$[a \mapsto a_0,\ x \mapsto x_0]$

```
if (…)  {
```

$[a \mapsto a_0,\ x \mapsto x_0]$

```
a = a -1 ;
```

$[a \mapsto a_0-1,\ x \mapsto x_0]$

```
p (a);
```

$[a \mapsto a_0-1,\ x \mapsto -2a_0+7]$

```
a = a + 1;
```

$[a \mapsto a_0,\ x \mapsto -2a_0+7]$

```
}
```

$[a \mapsto a_0,\ x \mapsto x_0]\ [a \mapsto a_0,\ x \mapsto \tau]$

```
x = -2*a + 5;
```

$[a \mapsto a_0,\ x \mapsto -2*a_0+5]$

```
}
```

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

Tabulation for finite lattices L

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

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

Issues in Functional Approach

- How to guarantee that finite height for functional lattice?
  - It may happen 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

Summary Functional approach

- Computes procedure abstraction
- Sharing between different contexts
- Rather precise
- Recursive procedures may be more precise/efficient than loops
- But requires more from the implementation
  - Representing (input/output) relations
  - Composing relations

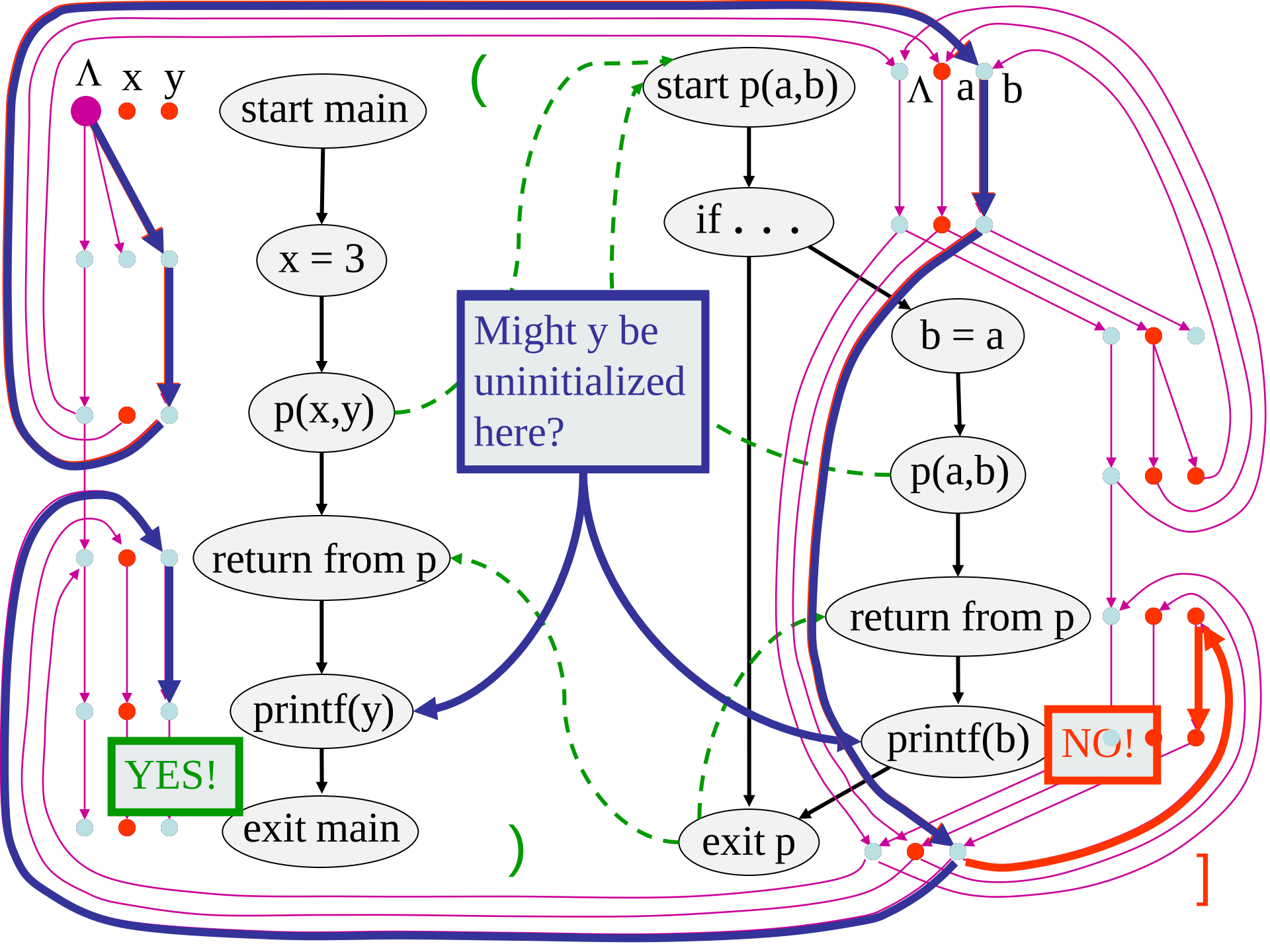
CFL-Graph reachability [RHS’95]

- Static analysis of programs with proecedures
- Special cases of functional analysis
- Reduce the interprocedural analysis problem to finding context free reachability

[RHS’95] Thomas W. Reps, Susan Horwitz, Shmuel Sagiv: Precise Interprocedural Dataflow Analysis via Graph Reachability. POPL 1995

The Context-Free Reachability Problem

- A finite directed graph G(s, V, E)
- A finite alphabet Σ
- A labeling function l: E $\rightarrow$ Σ
- A context-free grammar C over Σ
- A property holds at n $\in$ N if there exists a path from s to n whose labels are in C


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

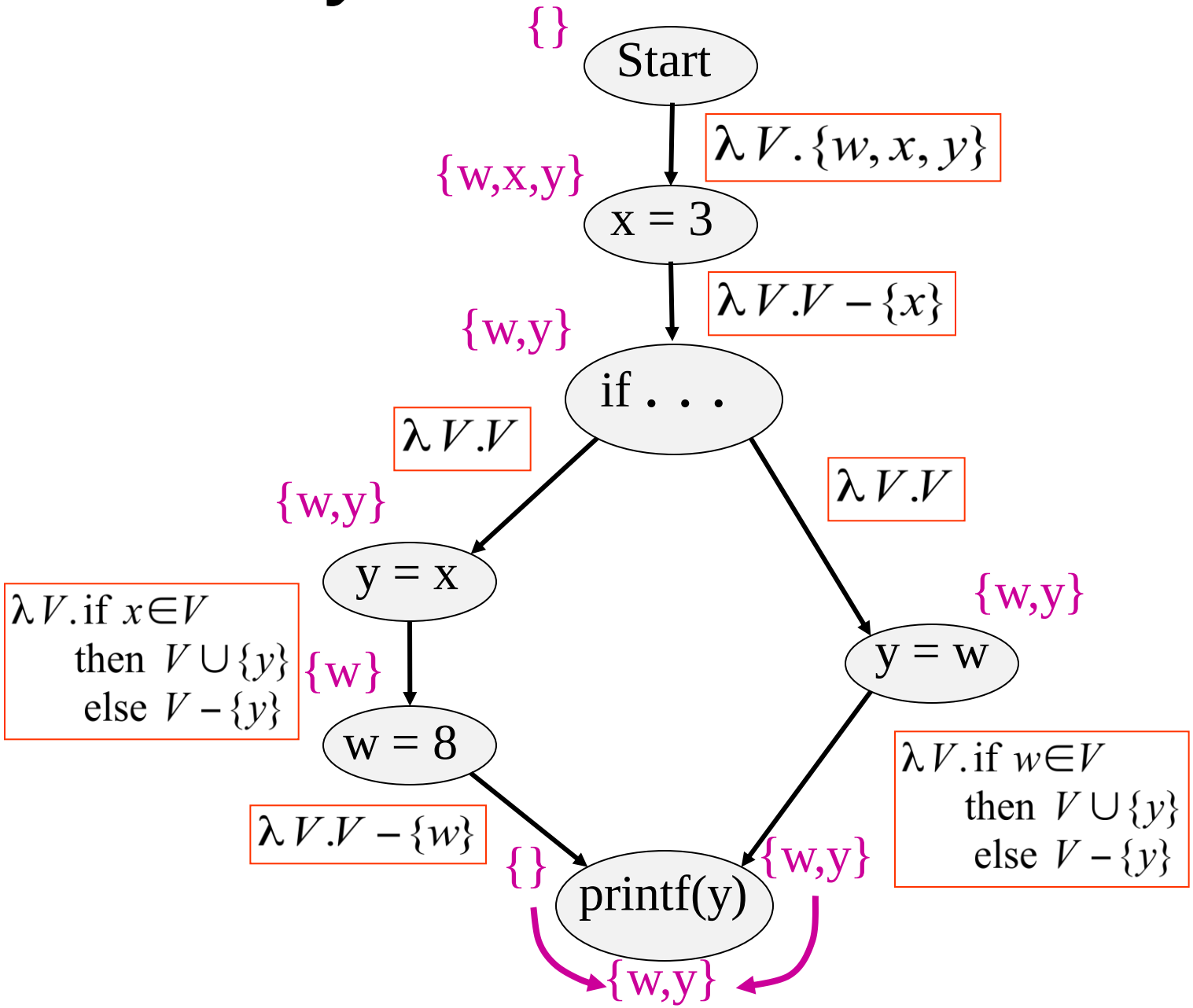
Questions

- When can we reduce a static analysis problem to CFL reachability
- What is the complexity of solving CFL reachability problems
- Interesting generalizations of CFL reachability

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
- Can be generalized to certain infinite lattices

Possibly Uninitialized Variables



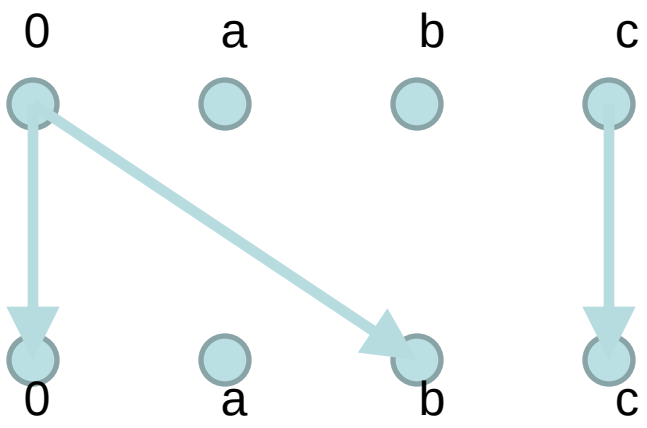
Efficiently Representing Functions

- Let $f:2^D \rightarrow 2^D$ be a distributive function
- Then:
  - $f(X) = \{ f(\{z\}) \mid z \in X \}$
  - $f(X) = f(\emptyset) \cup \{ f(\{z\}) \mid z \in X \}$

Encoding Transfer Functions

- Enumerate all input space and output space
- Represent functions as graphs with 2(D+1) nodes
- Special symbol “0” denotes empty sets (sometimes denoted Λ)
- Example: D = { a, b, c }

  f(S) = (S – {a}) U {b}

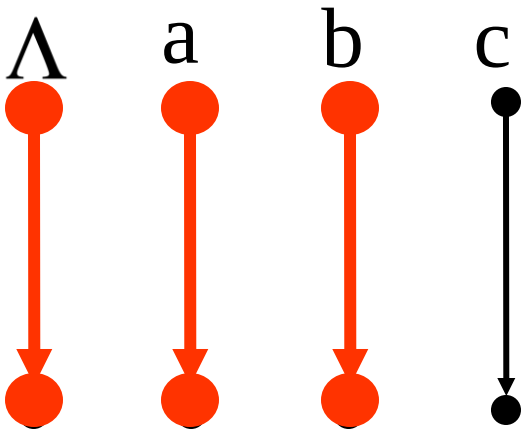


Representing Dataflow Functions

Identity Function

$$f = \lambda V.V$$

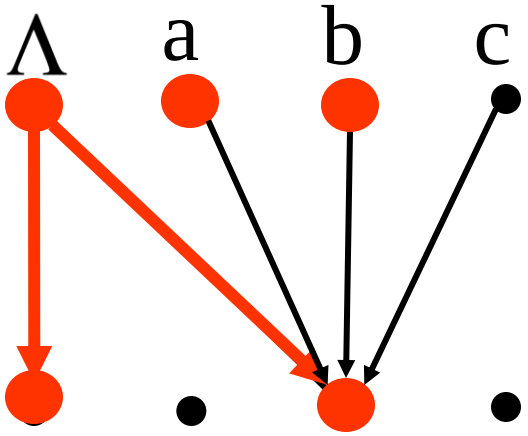
$$\boxed{f(\{a,b\}) = \{a,b\}}$$



Constant Function

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

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

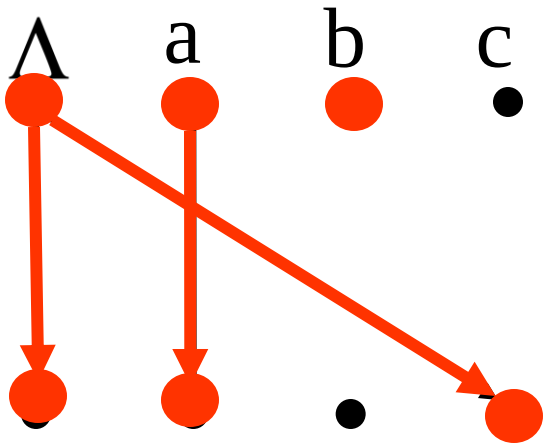


Representing Dataflow Functions

"Gen/Kill" Function

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

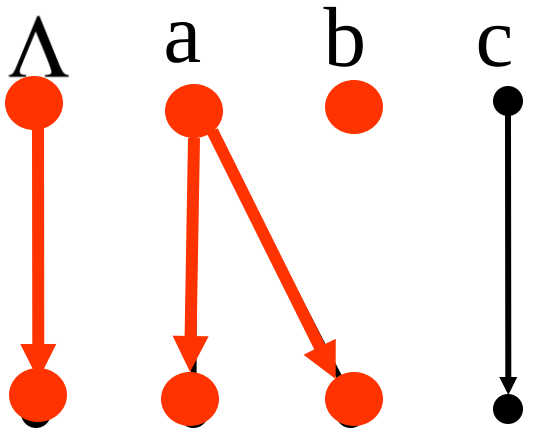
$$\boxed{\mathrm{f}(\{a,b\}) = \{a,c\}}$$



Non-"Gen/Kill" Function

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

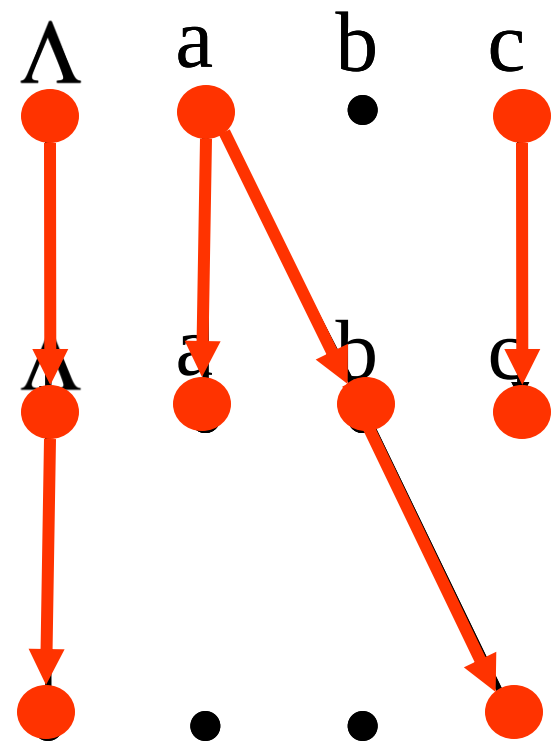
$$\boxed{\mathrm{f}(\{a,b\}) = \{a,b\}}$$



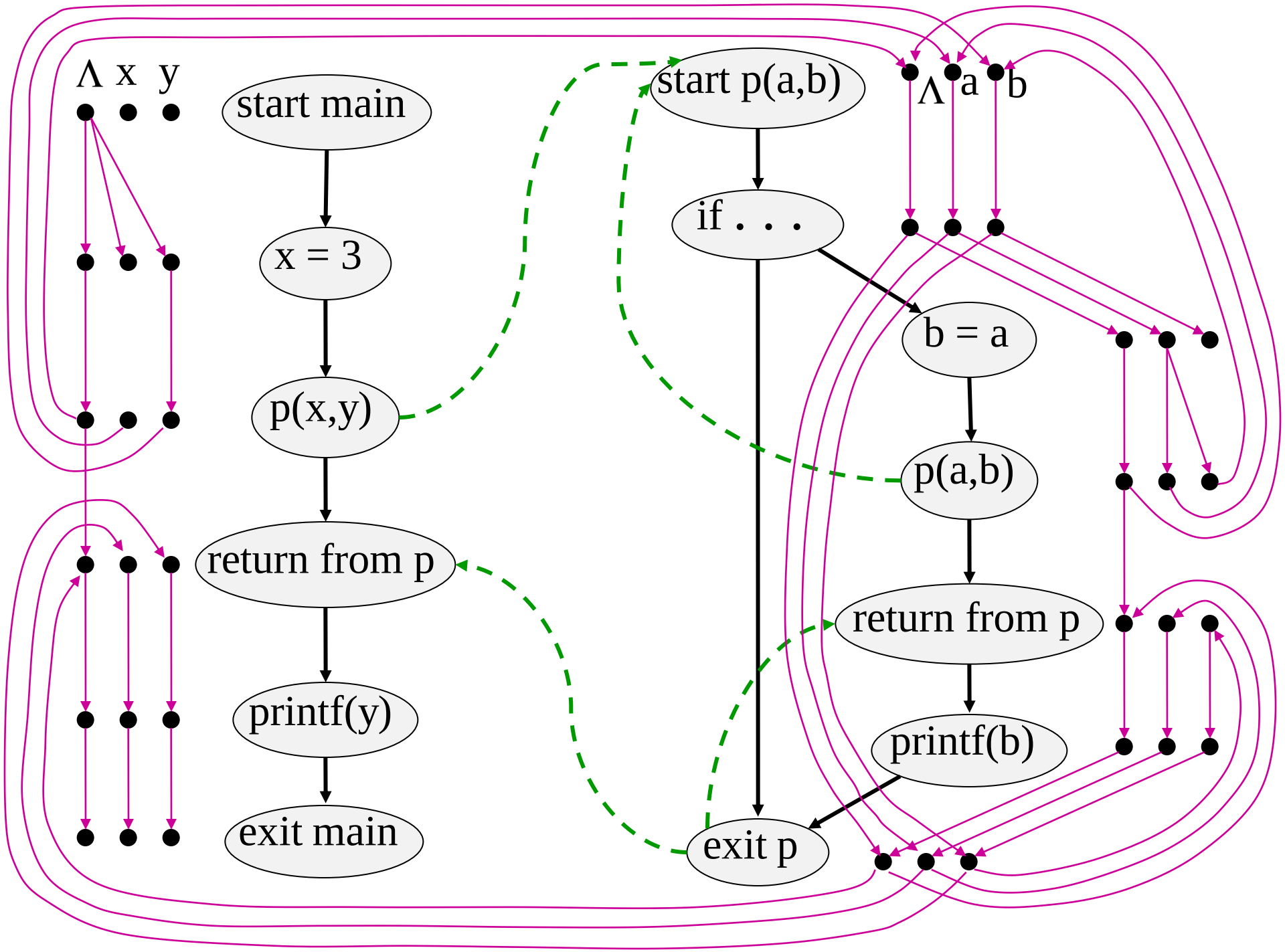
Composing Dataflow Functions

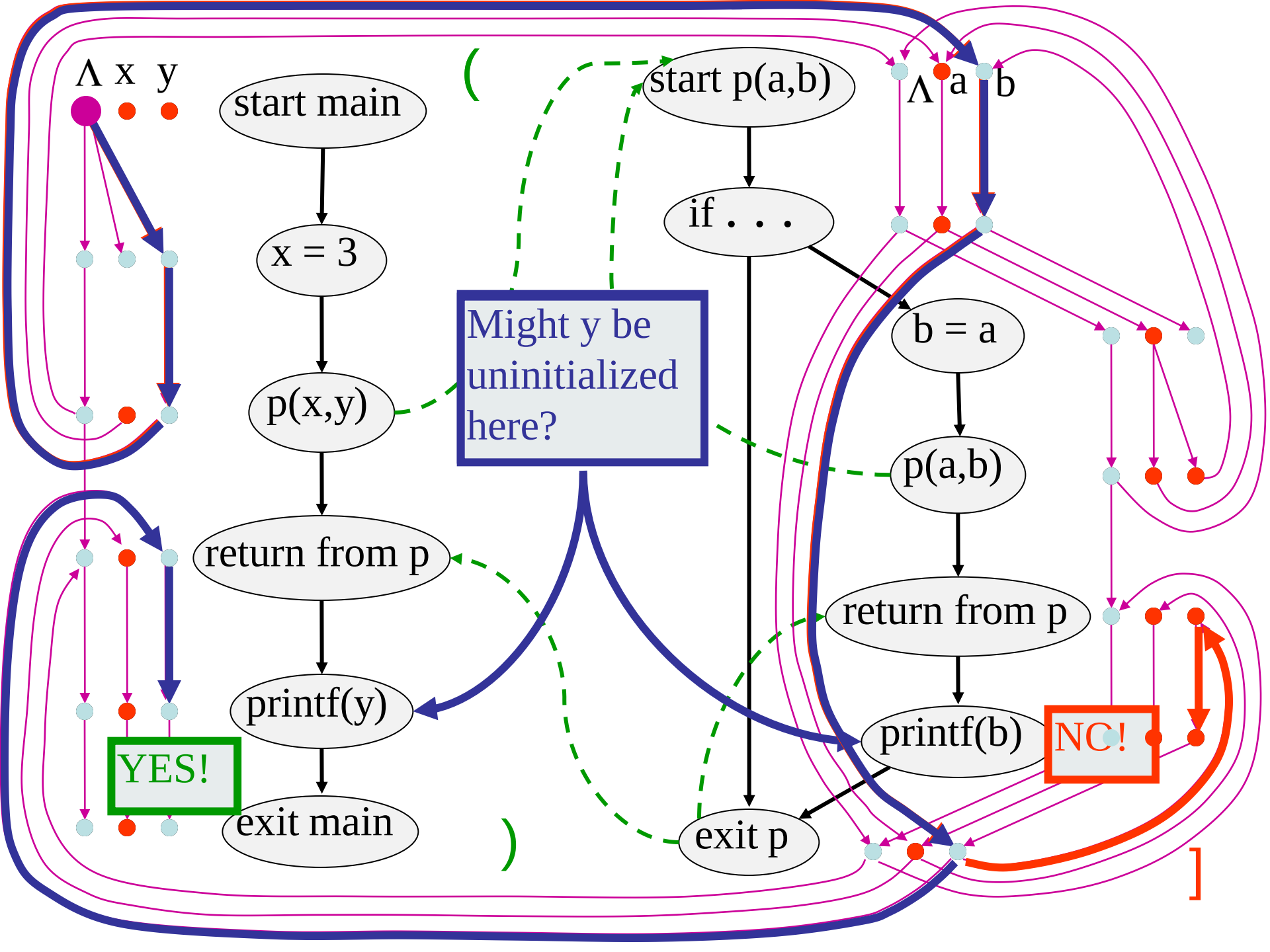
$$f_1 = \lambda V.\text{if } a \in V \text{ then } V \cup \{b\} \text{ else } V - \{b\}$$

$$f_2 = \lambda V.\text{if } b \in V \text{ then } \{c\} \text{ else } \phi$$



$$f_2 \circ f_1(\{a, c\}) = \boxed{\{c\}}$$


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


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

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

Asymptotic Running Time

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

Running time: $O(ED^3)$

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

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

Some Applications

Mayur Naik: Jchord a static analysis for Java
IBM Watson: Wala static analysis tool

Thomas Ball, Vladimir Levin, Sriram K. Rajaman
A decade of software model checking with SLAM.CACM'11

Manu Sridharan, Rastislav Bodík: Refinement-based context-sensitive points-to analysis for Java. PLDI 2006

Nomair A. Naeem, Ondrej Lhoták, Jonathan Rodriguez: Practical Extensions to the IFDS Algorithm. CC 2010: 124-144

Osbert Bastani, Saswat Anand, Alex Aiken: Specification Inference Using Context-Free Language Reachability, POPL'15

K Chatterjee, A Pavlogiannis, Y Velner: Quantitative interprocedural analysis, POPL'15

S Yang, D Yan, H Wu, Y Wang: Static control-flow analysis of user-driven callbacks in Android applications

Asymptotic Running Time

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

Running time: $O(ED^3)$

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

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

IDE

- Goes beyond IFDS problems
  - Can handle unbounded domains
- Requires special form of the domain
- Can be **much** more efficient than IFDS

Example Linear Constant Propagation

- Consider the constant propagation lattice
- The value of every variable y at the program exit can be represented by:

$$y = \sqcup \{(a_x x + b_x) \mid x \in Var_*\} \sqcup c$$

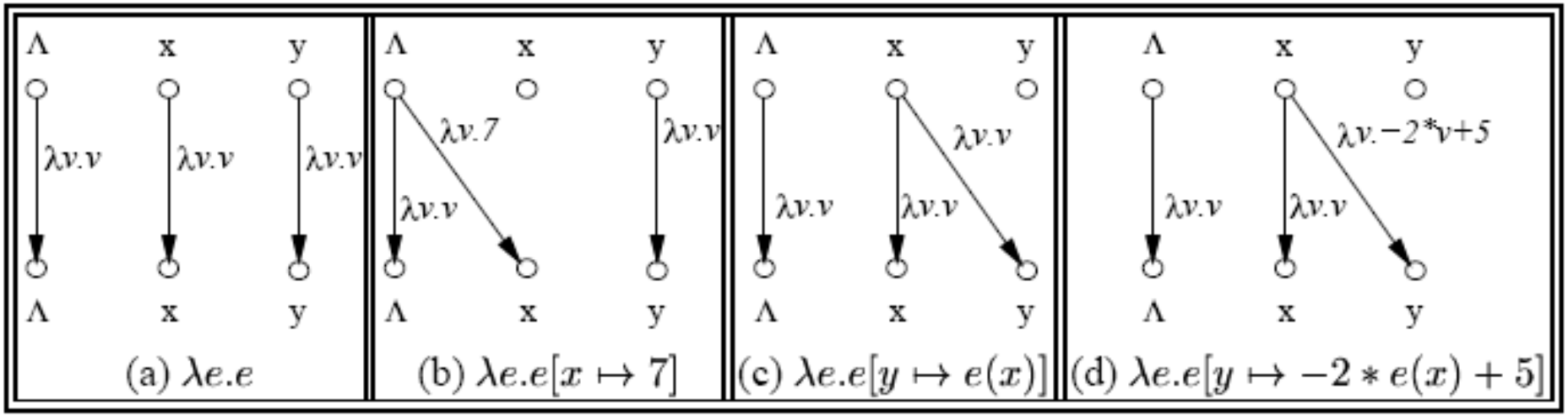
$$a_x, c \in Z * \{\bot, \top\} \qquad b_x \in Z$$

- Supports efficient composition and “functional” join
  - [z := a * y + b]
  - What about [z:=x+y]?

IDE Analysis

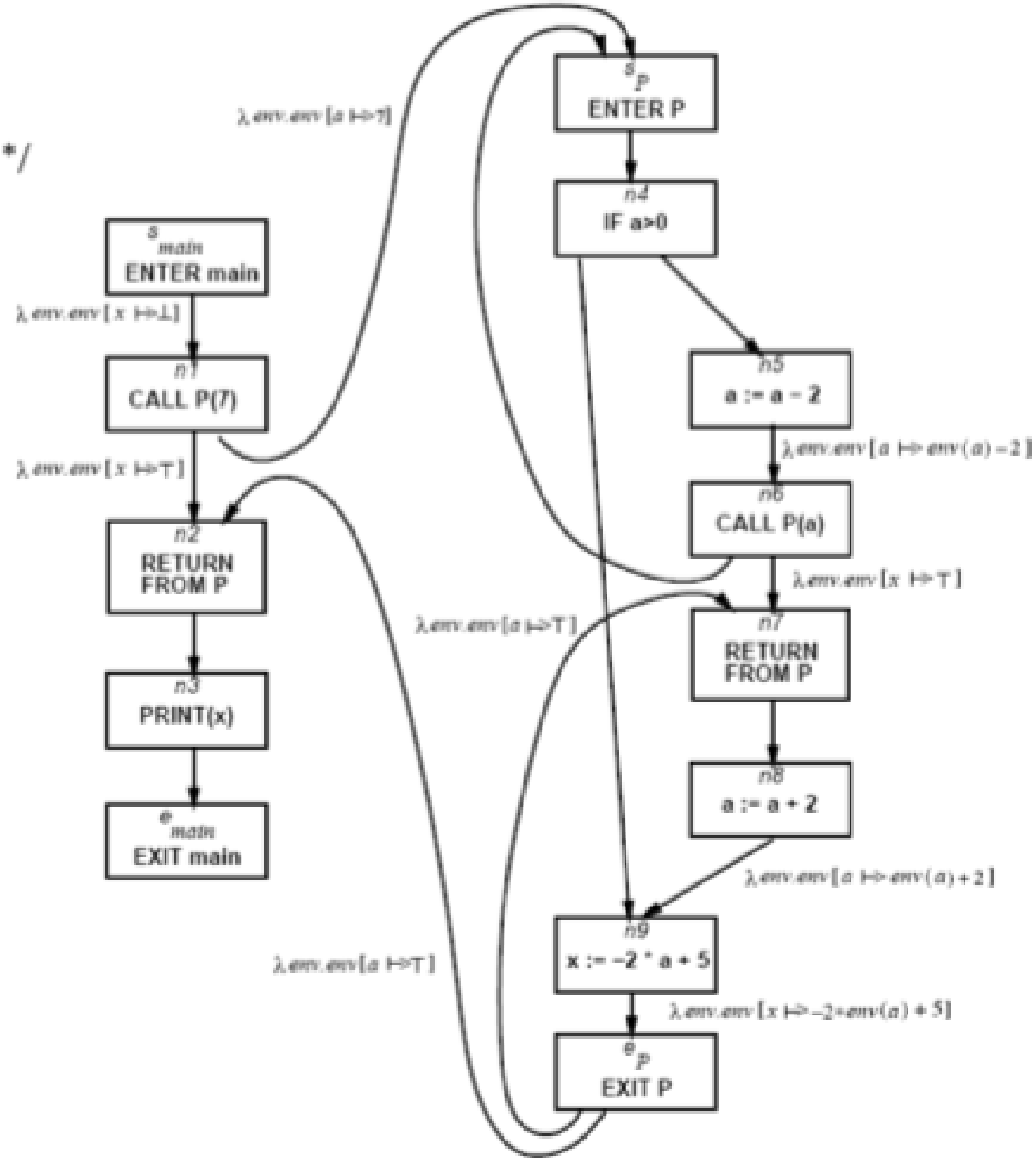
- Point-wise representation closed under composition
- CFL-Reachability on the exploded graph
- Two phase algorithm
  - Compose functions
  - Compute dataflow values

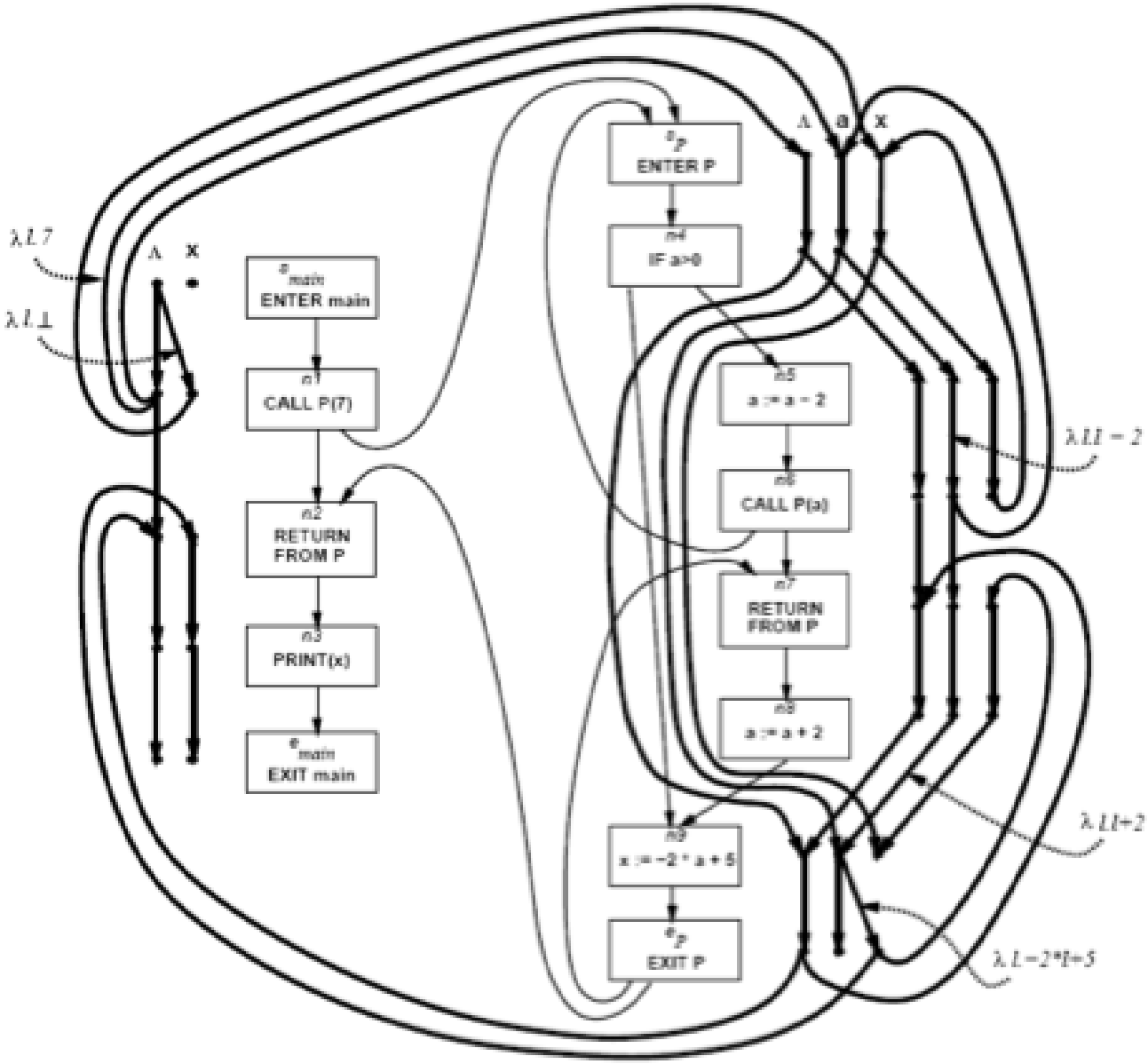
Linear constant propagation



Point-wise representation of environment transformers

```
declare x: integer
program main
begin
        call P(7)
        print (x) /* x is a constant here */
end

procedure P (value a : integer)
begin /* a is not a constant here */
        if a > 0 then
           a := a − 2
           call P (a)
           a := a + 2
        fi
        x := −2 * a + 5
        /* x is not a constant here */
end
```




λ L 7
λ L ⊥
Λ
x
s_main
ENTER main
n1
CALL P(7)
n2
RETURN
FROM P
n3
PRINT(x)
e_main
EXIT main
s_P
ENTER P
n4
IF a>0
n5
a := a - 2
n6
CALL P(a)
n7
RETURN
FROM P
n8
a := a + 2
n9
x := -2 * a + 5
e_P
EXIT P
Λ
a
x
λ L1 = 2
λ L1+2
λ L=-2*L+5

Costs

- $O(ED^3)$
- Class of value transformers $F \subseteq L \rightarrow L$
  - $id \in F$
  - Finite height
- Representation scheme with (efficient)
  - Application
  - Composition
  - Join
  - Equality
  - Storage

Conclusion

- Handling functions is crucial for abstract interpretation
- Virtual functions and exceptions complicate things
- But scalability is an issue
  - Small call strings
  - Small functional domains
  - Demand analysis

Bibliography

- **Textbook 2.5**
- Patrick Cousot & Radhia Cousot. Static determination of dynamic properties of recursive procedures In *IFIP Conference on Formal Description of Programming Concepts*, 1978
- **Two Approaches to interprocedural analysis by Micha Sharir and Amir Pnueli: 1**
- IDFS Interprocedural Distributive Finite Subset Precise interprocedural dataflow analysis via graph reachability. *Reps, Horowitz, and Sagiv, POPL'95*
- **IDE** Interprocedural Distributive Environment Precise interprocedural dataflow analysis with applications to constant propagation. *Sagiv, Reps, Horowitz, and TCS'96*