Interprocedural Analysis

Optimizations

For languages like C and C++ there are three granularities of optimizations

Complexity

1. Local optimizations
   - Apply to a basic block in isolation
2. Global optimizations
   - Apply to a control-flow graph (method body) in isolation
3. Inter-procedural optimizations
   - Apply across method boundaries

powerful

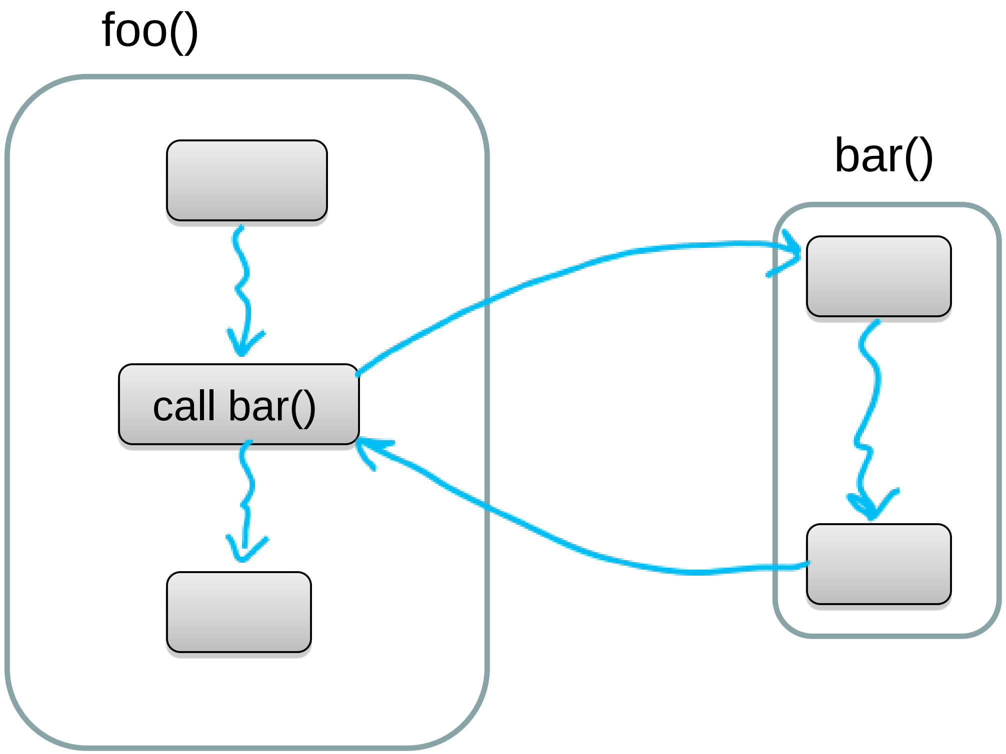
Most compilers do (1), many do (2), few do (3)

Procedural program

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

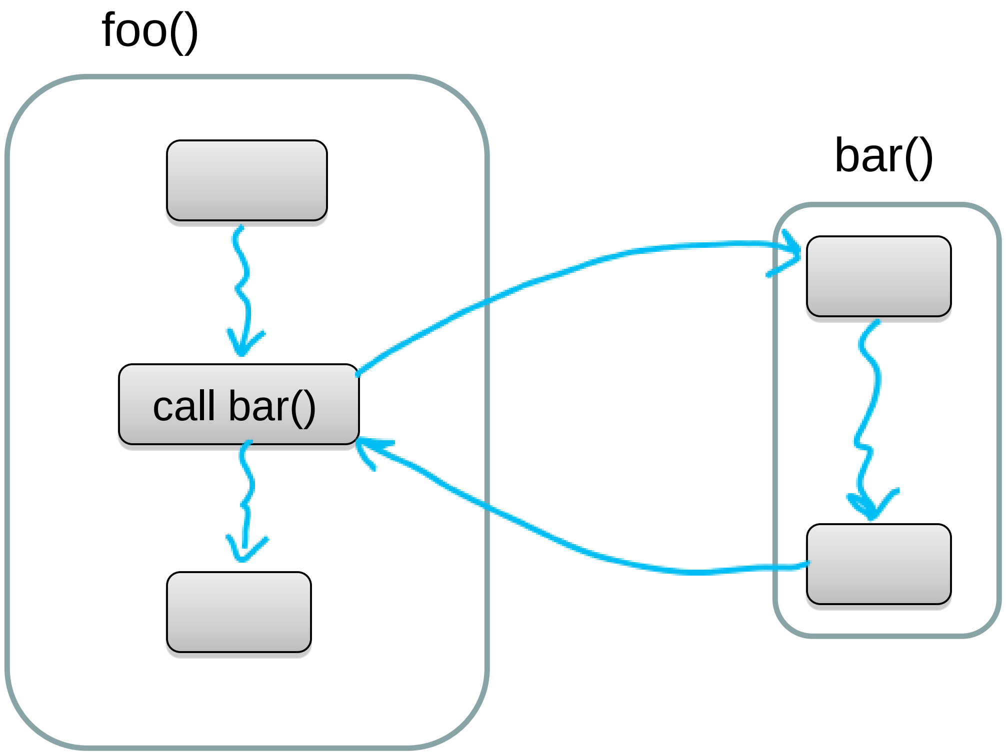
```
int p(int a) {
  return a + 1;
}
```

Effect of procedures



The effect of calling a procedure is the effect of executing its body（ parameter passing+return）

Interprocedural Analysis



goal: compute the abstract effect of calling a procedure

- How to do interprocedural analysis?
- Can we extend intraprocedural analysis?

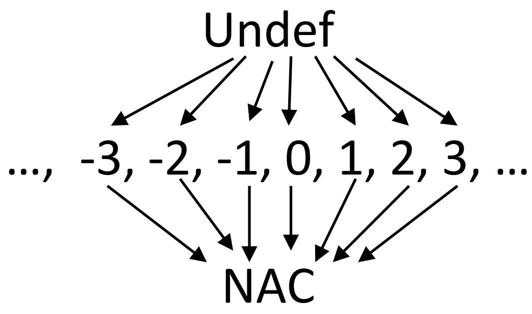
Reduction to intraprocedural analysis

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

Reminder: Constant Propagation

- To make the problem precise, we associate one of the following values with X at every program point

Value	description
Undef	means that analysis hasn't determined if control reaches that point
c	constant c
NAC	is definitely not a constant 1. Assigned by an input value 2. Not a constant 3. Assigned different values via paths



- The set of values: product of semi-lattices, one component for each variable
- Represented by a map m: Var-> V

Reminder: Constant Propagation

v_1	v_2	$v_1 \wedge v_2$
Undef	Undef	Undef
	c_2	c_2
	NAC	NAC
c_1	Undef	c_1
	c_2	NAC if $c_1 != c_2$ c_1 otherwise
	NAC	NAC
NAC	Undef	NAC
	c_2	NAC
	NAC	NAC

- Meet: $m_1 \wedge m_2 = m_3$, such that $m_1(x) \wedge m_2(x) = m_3(x)$
- i.e., $m_1 \leq m_2$ iff $m_1(x) \leq m_2(x)$ for all x in Var

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 ->Undef, x -> Undef, ret -> Undef]
    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{
}
```

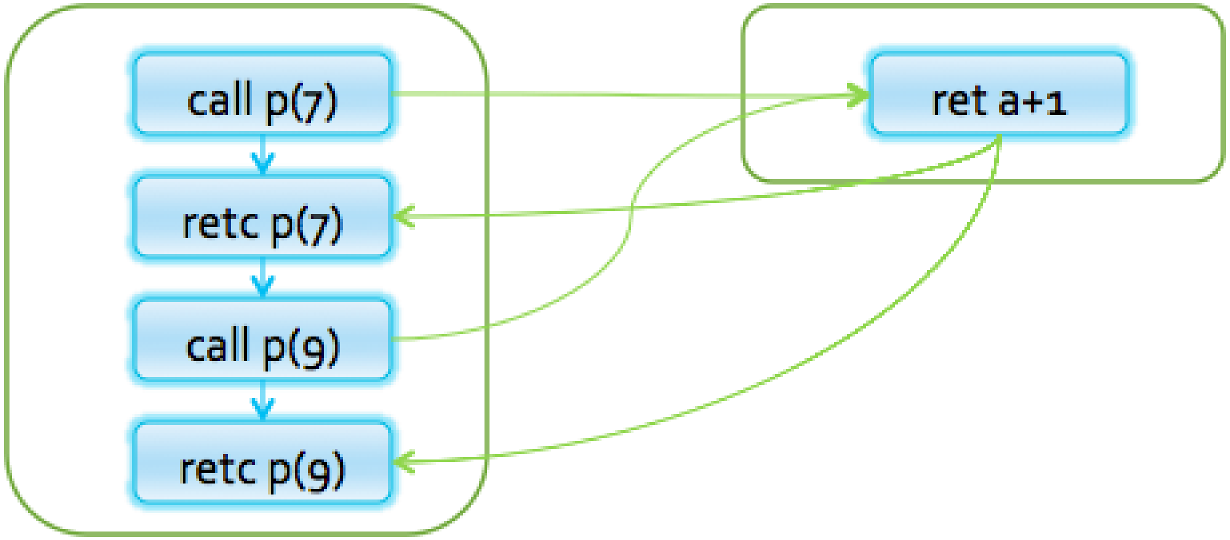
A Naive Interprocedural solution

- Treat procedure calls as gotos

Simple Example

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

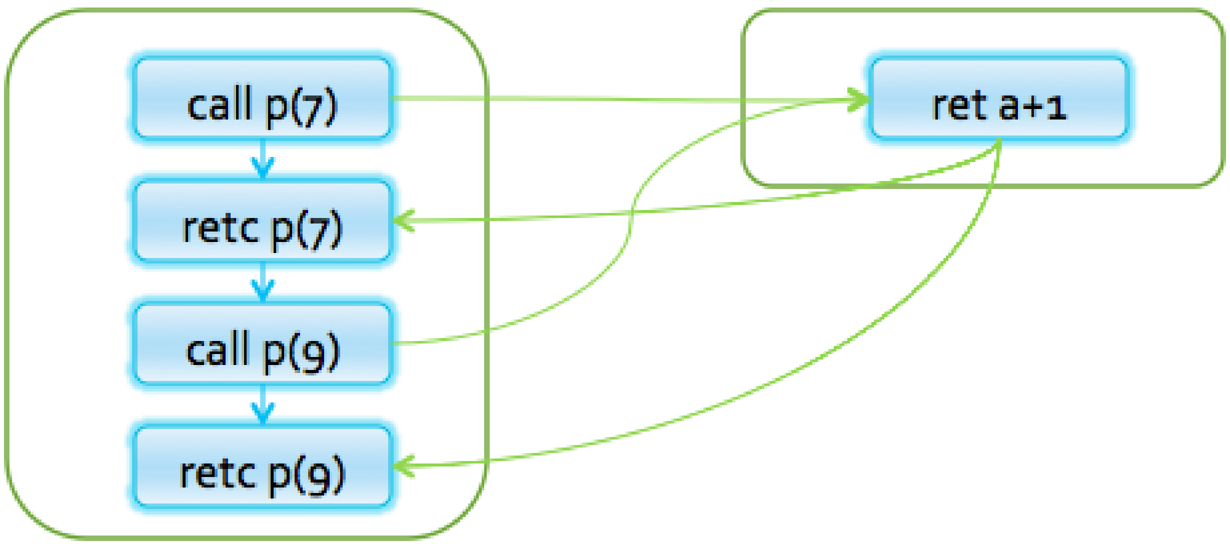
```
int p(int a) {
  return a + 1;
}
```



Simple Example

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

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

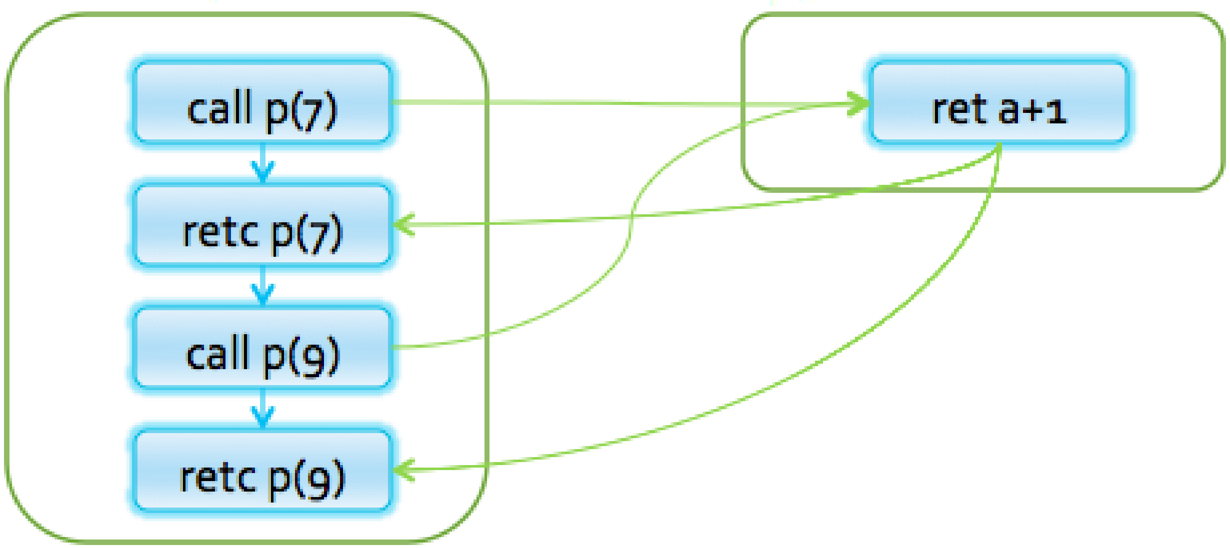


Simple Example

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

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

}
```

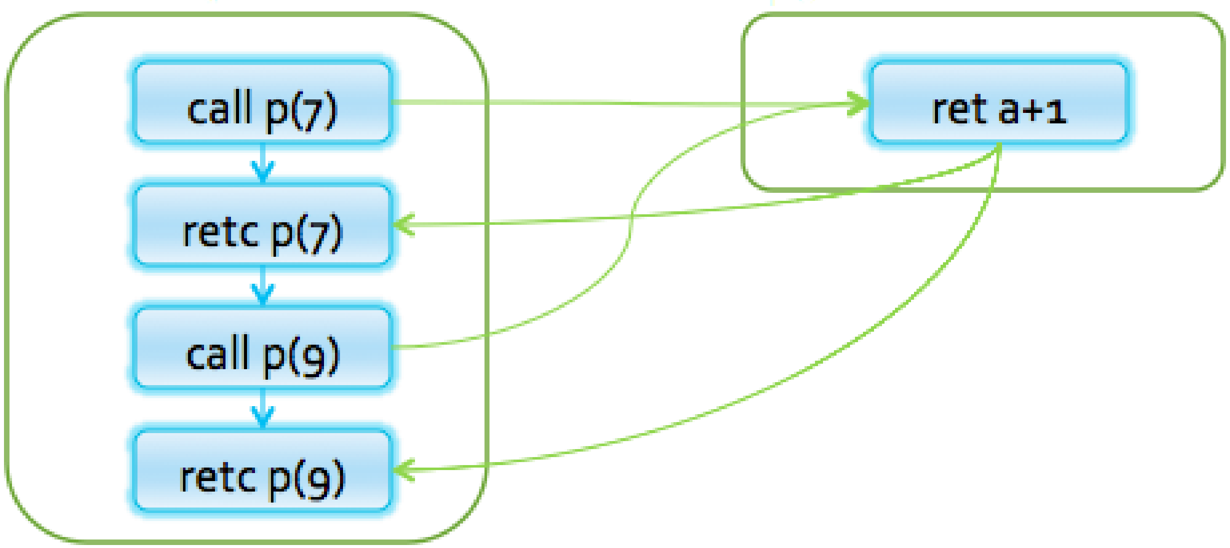


Simple Example

```
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]

}
```

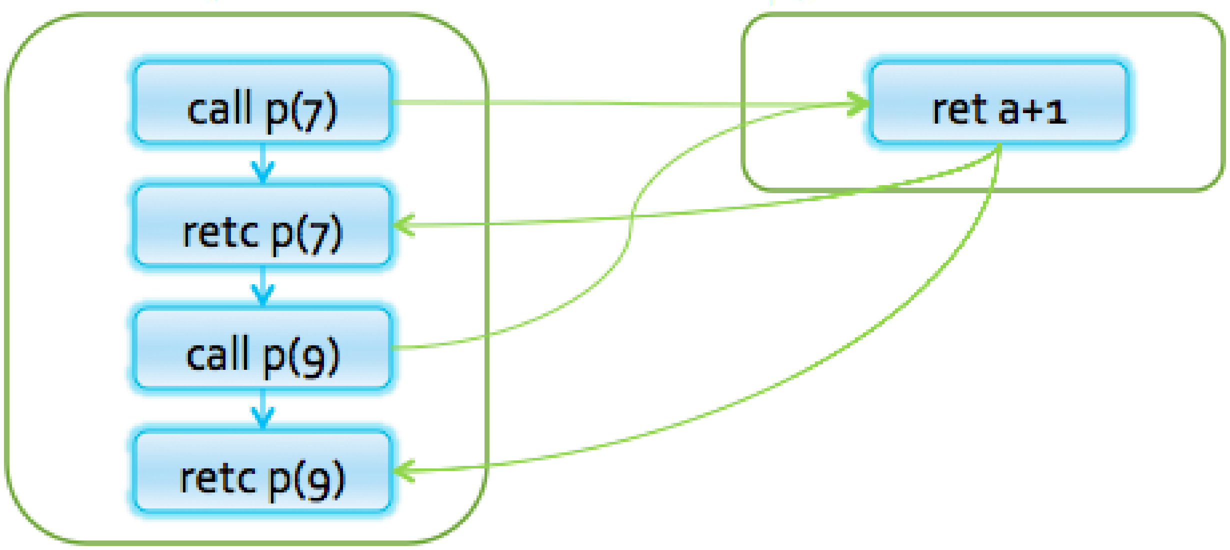


Simple Example

```
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]

}
```

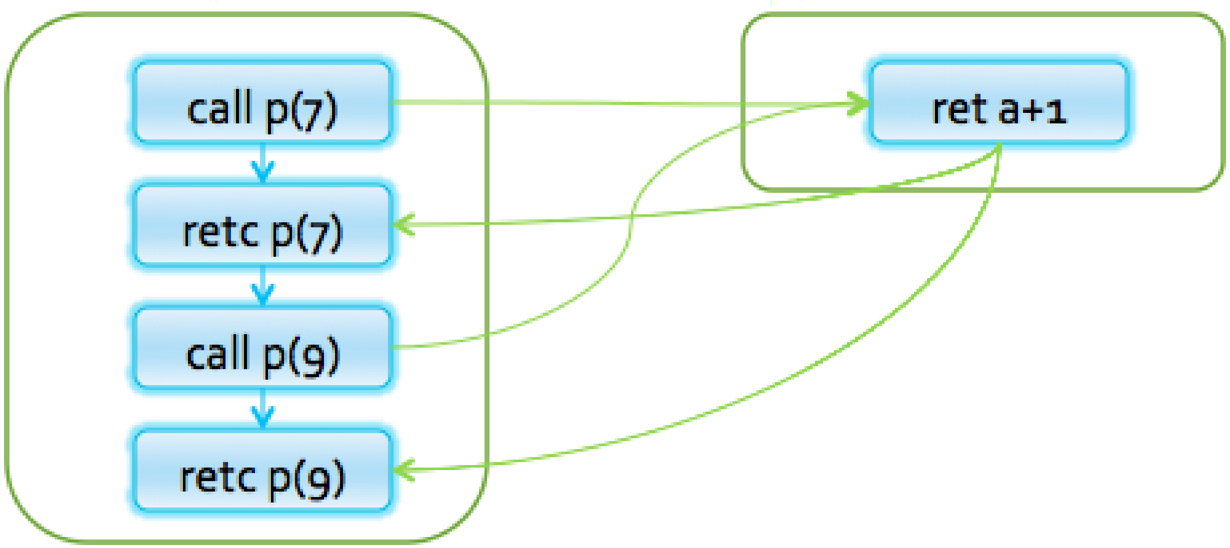


Simple Example

```
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]

}
```

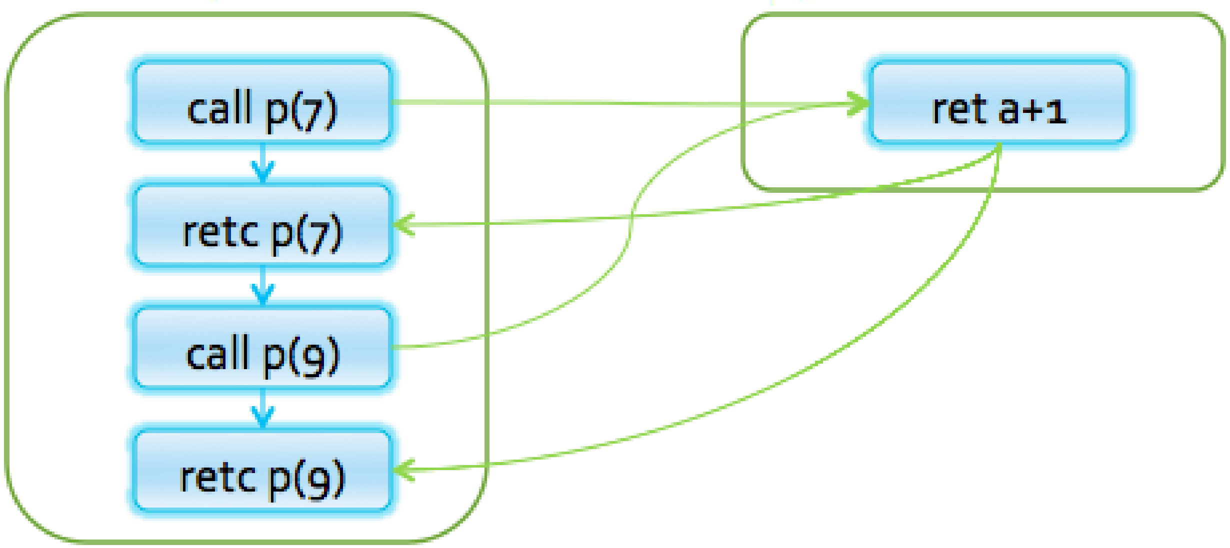


Simple Example

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

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

}
```



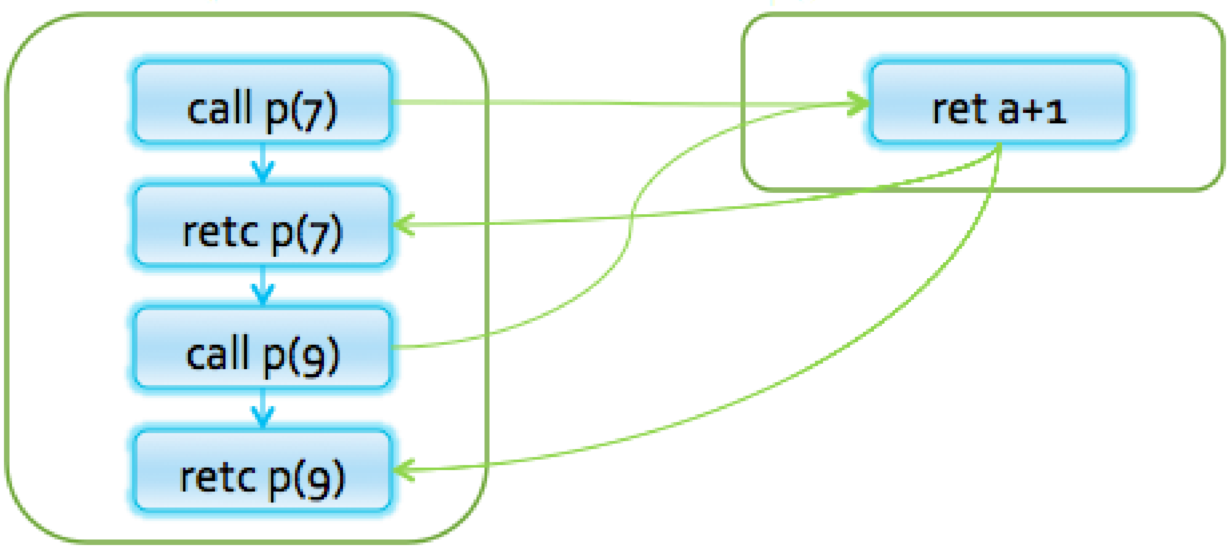
Simple Example

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

}
```

```
int p(int a) {
    [a ->NAC]
    return a + 1;
    [a ->NAC, $$ ->NAC]


}
```



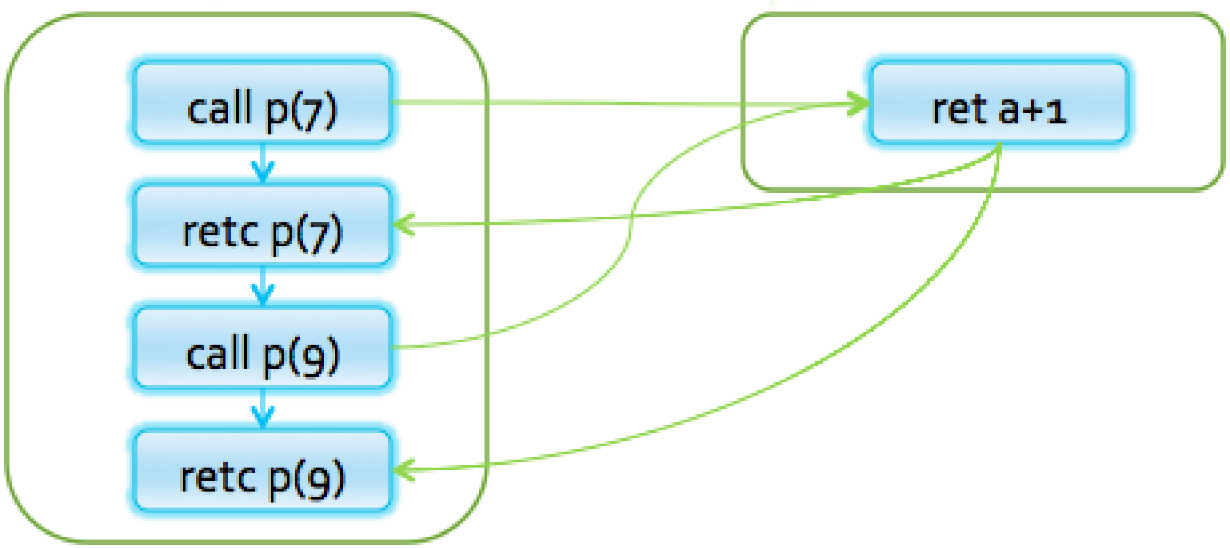
Simple Example

```
void main() {
    int x ;
    x = p(7) ;
     [x ->NAC]
    x = p(9) ;
     [x ->NAC]

}
```

```
int p(int a) {
   [a ->NAC]
    return a + 1;
   [a ->NAC, $$ ->NAC]

}
```



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 ->NAC]
    x = p(9) ;
     [x ->NAC]
}
```

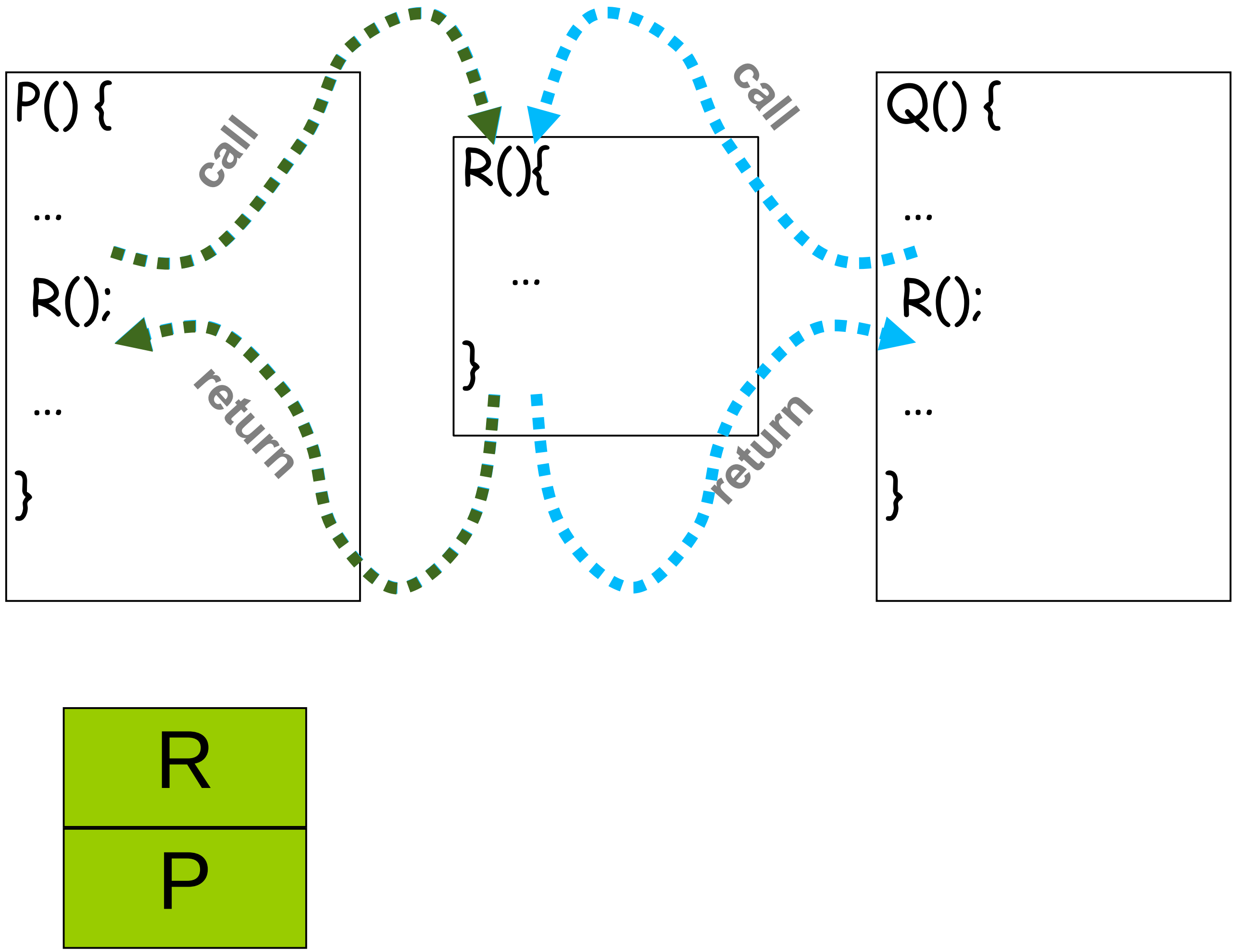
```
int p(int a) {
   [a ->NAC]
    return a + 1;
   [a ->NAC, $$ ->NAC]
}
```

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

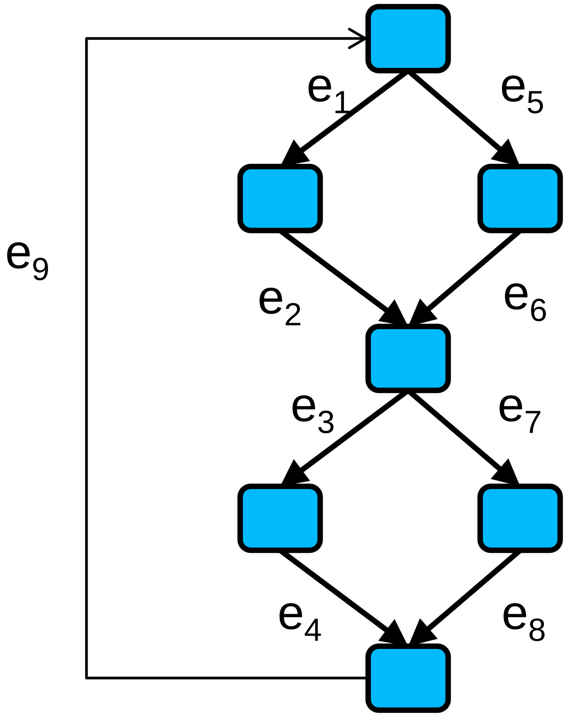
- ✓ Procedure Inlining
- ✓ The naive approach
- Valid paths
- The callstring approach
- The Functional Approach
- IFDS: Interprocedural Analysis via Graph Reachability
- IDE: Beyond graph reachability

- The trivial modular approach

Meet-Over-All-Paths (MOP)

- 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
  $f[e_1, e_2, \ldots, e_n](l) = f(e_n)(\ldots(f(e_2)(f(e_1)(l))\ldots)$
- $MOP[v] = \sqcap \{f[e_1, e_2, \ldots, e_n](l) \mid [e_1, e_2, \ldots, e_n] \in \text{paths}(v)\}$

Meet-Over-All-Paths (MOP)



Paths transformers:

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

MOP:

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

Number of program paths is unbounded due to loops

MFP approximates MOP

- MOP[v] $= \sqcap\{f\,[e_1, e_2, \ldots, e_n](I) \mid [e_1, e_2, \ldots, e_n] \in \text{paths}(v)\}$
- MFP[v] $= \sqcap\ \{f\,[e](\text{MFP}[v']) \mid e = (v', v)\}$
  MFP[v_0] = initial

- MOP $\leq$ MFP - for a monotone function
  - $f(x \sqcap y) \leq f(x) \sqcap f(y)$
- MOP = MFP - for a distributive function
  - $f(x \sqcap y) = f(x) \sqcap f(y)$

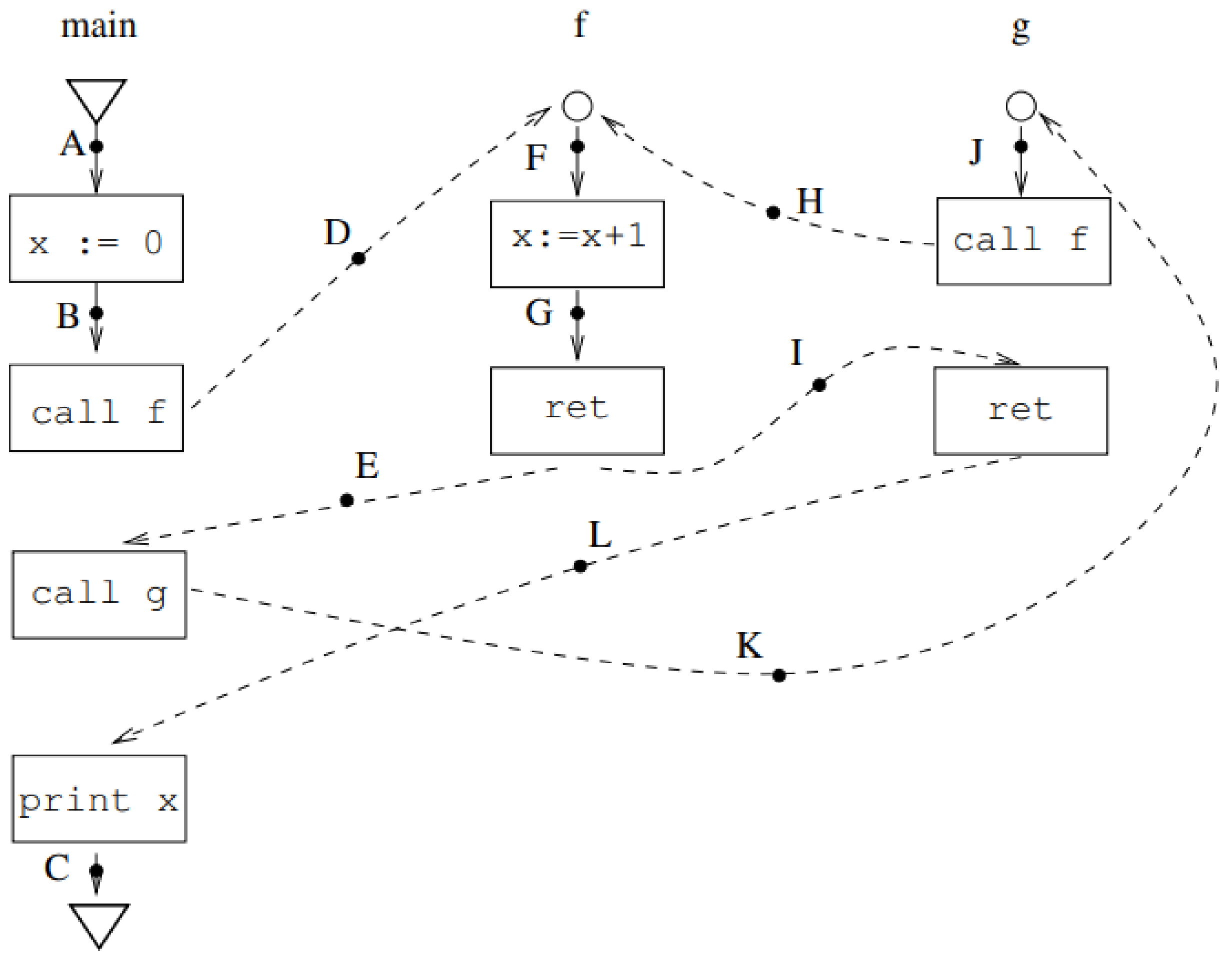
MOP may not be precise enough for interprocedural analysis!

Ex. 1. Actual collecting state at C?

{x → 2}.

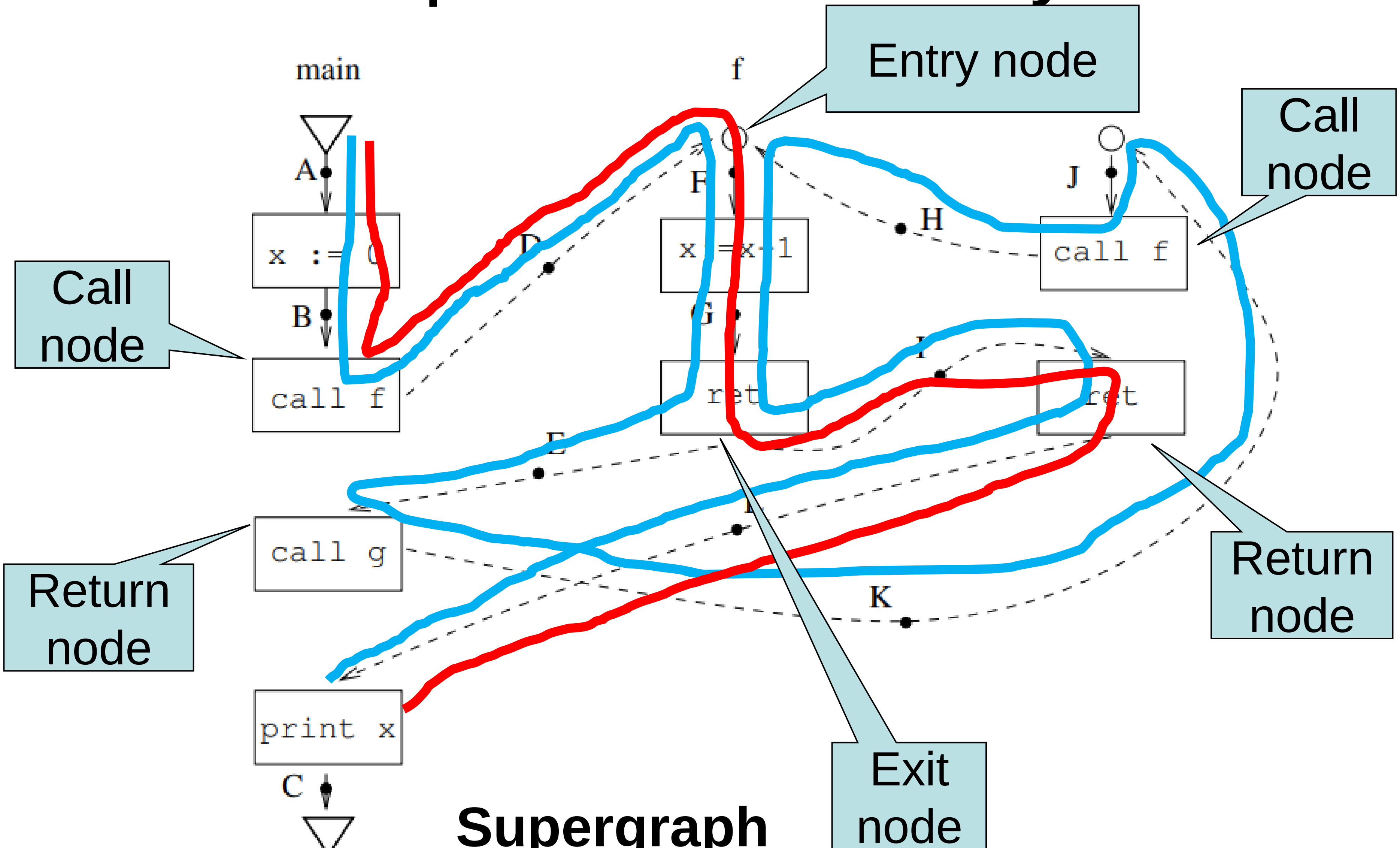
Ex. 2. MOP at C using collecting analysis?

{x → 1, x → 2, x → 3, . . .}.



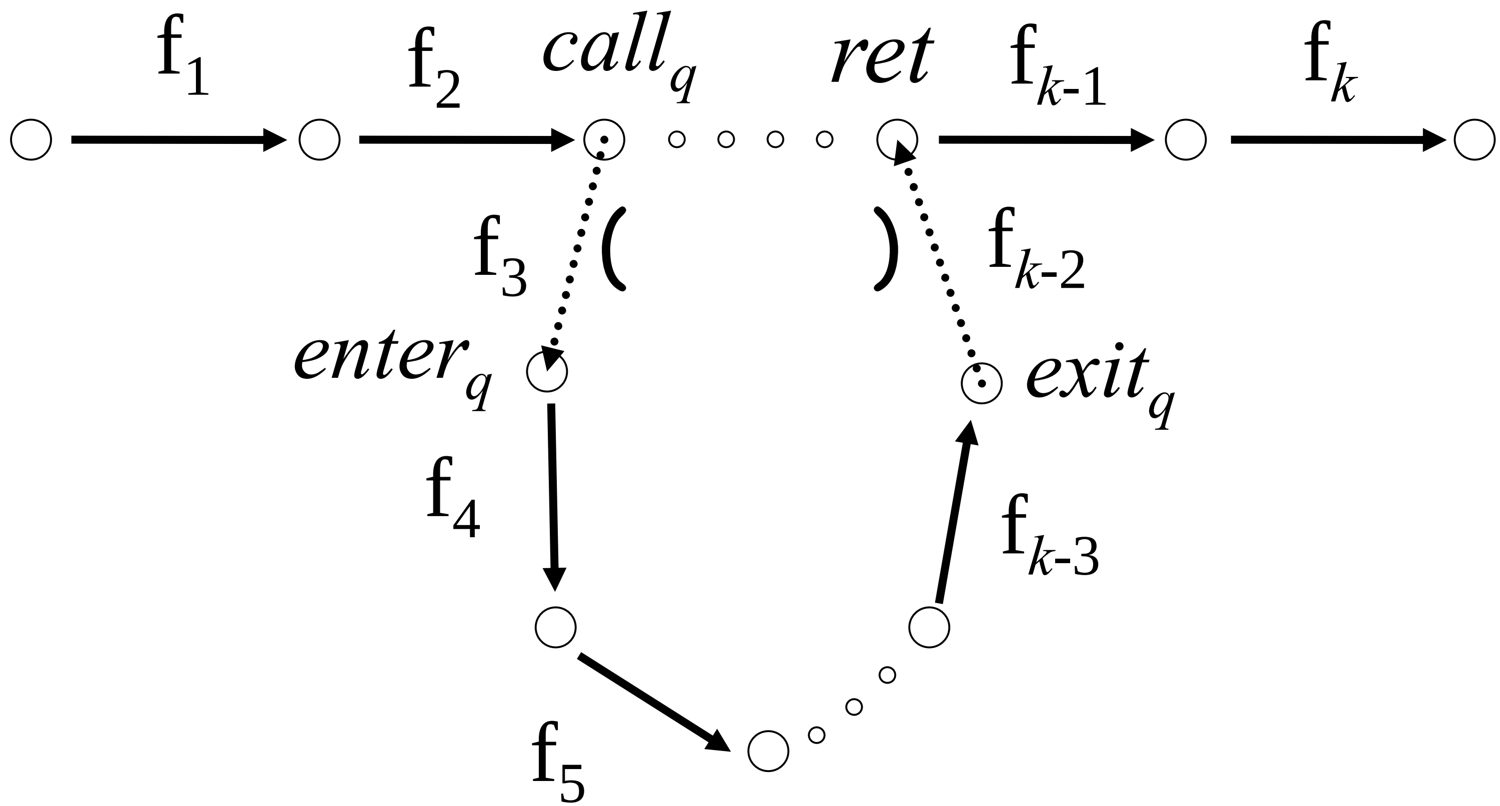
- MOP is sound but very imprecise.
- Reason: Some paths don't correspond to executions of the program: Eg. ABDFGILC.

Interprocedural analysis



Supergraph

Interprocedural Valid Paths



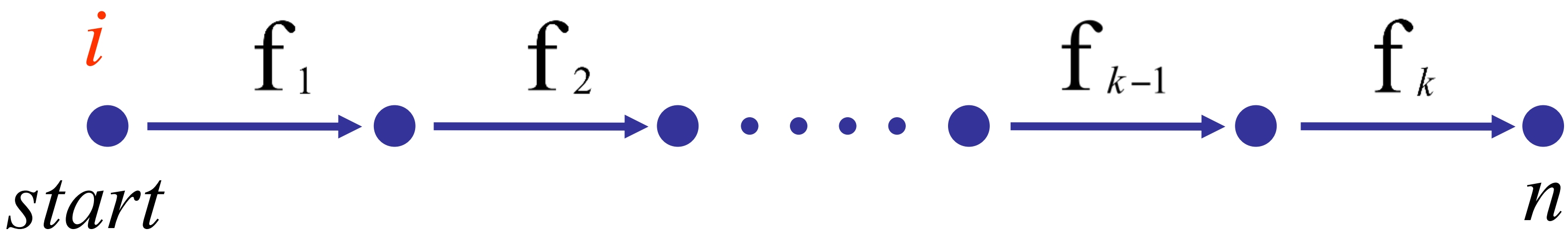
- IVP: all paths with matching calls and returns
  – And prefixes

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

Meet Over All Paths (MOP)



$$[\![f_k \ \mathrm{o} \ \ldots \ \mathrm{o} \ f_1]\!] : L \to L$$

- MOP[v] $= \Pi\{\ [[e_1, e_2, \ldots, e_n]](l) \ \mid (e_1, \ldots, e_n) \in \text{paths}(v)\}$
- MOP is over-approximated by MFP
  - Sometimes MOP = MFP
    - precise up to "**symbolic execution**"
    - Distributive problem

The Meet-Over-Valid-Paths (MOVP)

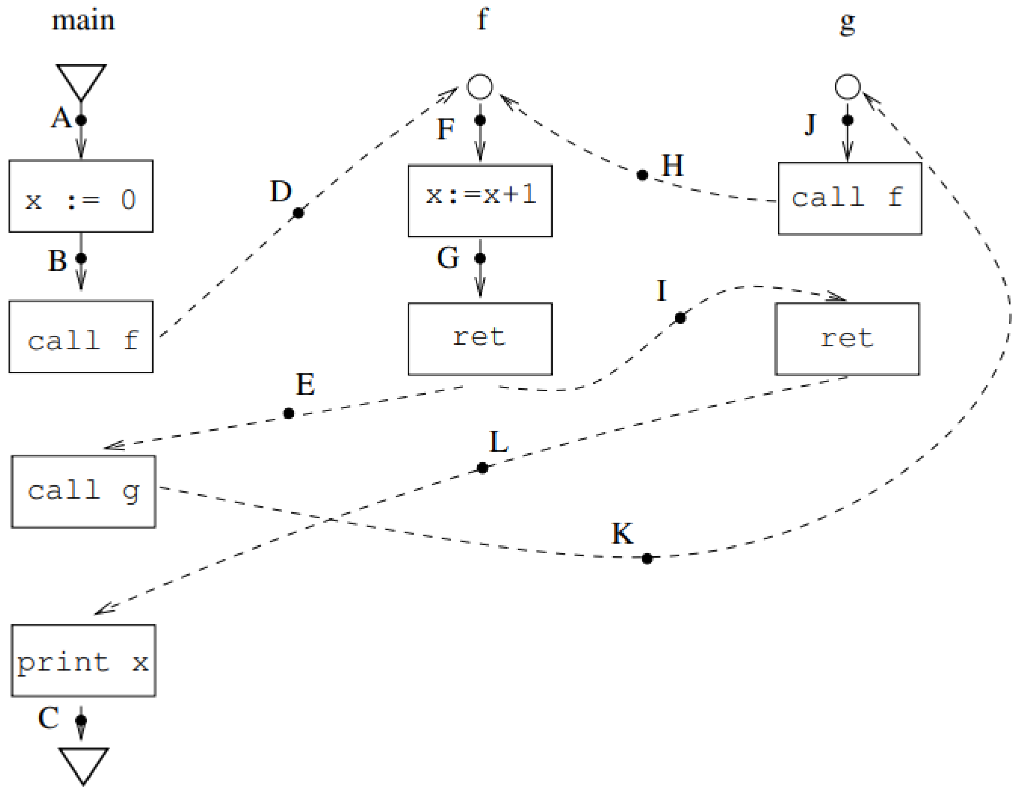
- vpaths(n) all valid paths from program start to n
- MOVP[n] $= \Pi\{ [[e_1, e_2, \ldots, e]] (l)$
  $(e_1, e_2, \ldots, e) \in$ vpaths(n)}

The Call-String Approach

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

Micha Sharir and Amir Pnueli: Two approaches to interprocedural data flow analysis, in *Program Flow Analysis: Theory and Applications* (Eds. Muchnick and Jones) (1981).

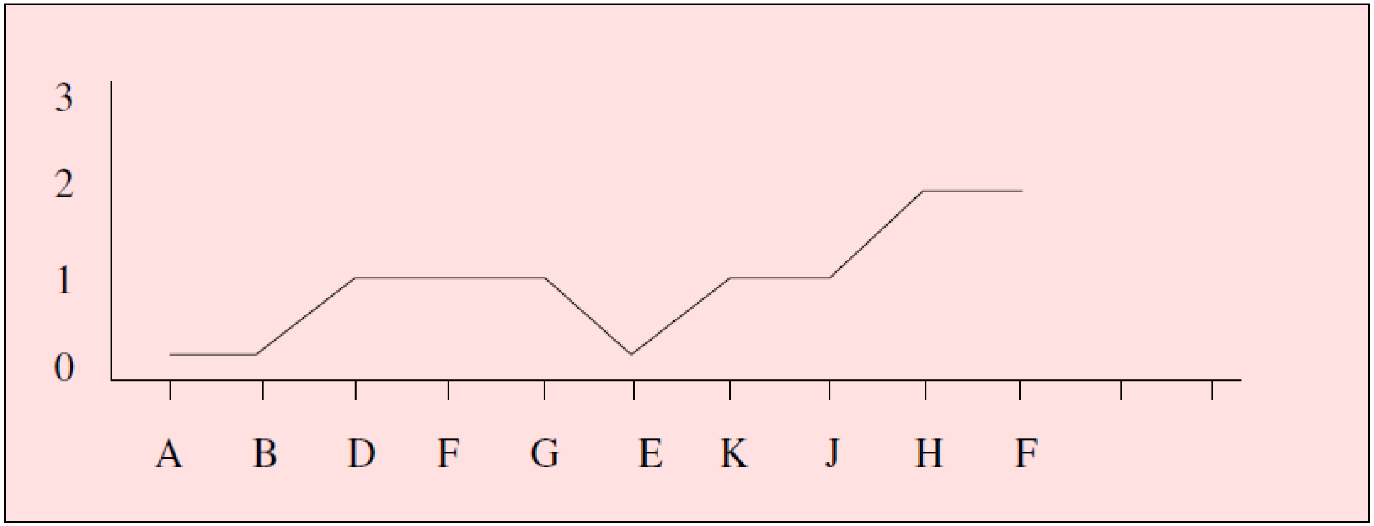
Interprocedurally valid paths and their call-strings



- The call-string of a valid path is a subsequence of call edges which have not been returned" as yet.
- For example, cs(ABDFGEKJHF) is KH".

Interprocedurally valid paths and their call-strings

- A path = ABDFGEKJHF in IVP for example program:



- Associated call-string cs(ABDFGEKJHF) is KH
- For path ρ=ABDFGEK, cs(ρ) = K
- For path ρ=ABDFGE, cs(ρ) = ε.

Interprocedurally valid paths and their call-strings

More formally: Let ρ be a path in G. We define when ρ is interprocedurally valid (and we say $\rho \in IVP(G)$) and its call-string $cs(\rho)$, by induction on the length of ρ.

- If $\rho = \varepsilon$ then $\rho \in IVP(G)$. In this case $cs(\rho) = \varepsilon$.
- If $\rho = \rho'\, n$ then $\rho \in IVP(G)$ iff $\rho \in IVP(G)$, and one of the following holds:
  - n is neither a call nor a ret edge. In this case $cs(\rho) = cs(\rho')$
  - n is a call edge. In this case $cs(\rho) = cs(\rho')\, N$.
  - n is ret edge. Suppose $cs(\rho') = \pi\, C$, and n corresponds to the call edge C. In this case, $cs(\rho) = \pi$

The set of (potential) call-strings in G is C*, where C is the set of call edges in G

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, (MFP)
- 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 -> NAC]
    c2: x = p(9) ;
    ε:  [x -> NAC]
}
```

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

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

Handling Recursion

```
void main() {
    c1: p(7);
     ε : [x -> NAC]
}
```

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

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 MOVP

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 ->a_0, x ->$-2a_0 + 5$]

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

[a ->a_0, x ->x_0]

```
 if (…) {
```

[a ->a_0, x ->x_0]

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

[a ->a_0-1, x ->x_0]

```
   p (a);
```

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

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

[a ->a_0, x ->$-2a_0$+7]

```
 }
```

[a ->a_0, x ->x_0] [a ->a_0, x ->NAC]

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

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

```
}
```

Phase 2

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

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

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

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 meet
  - 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 mplementation
  - 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

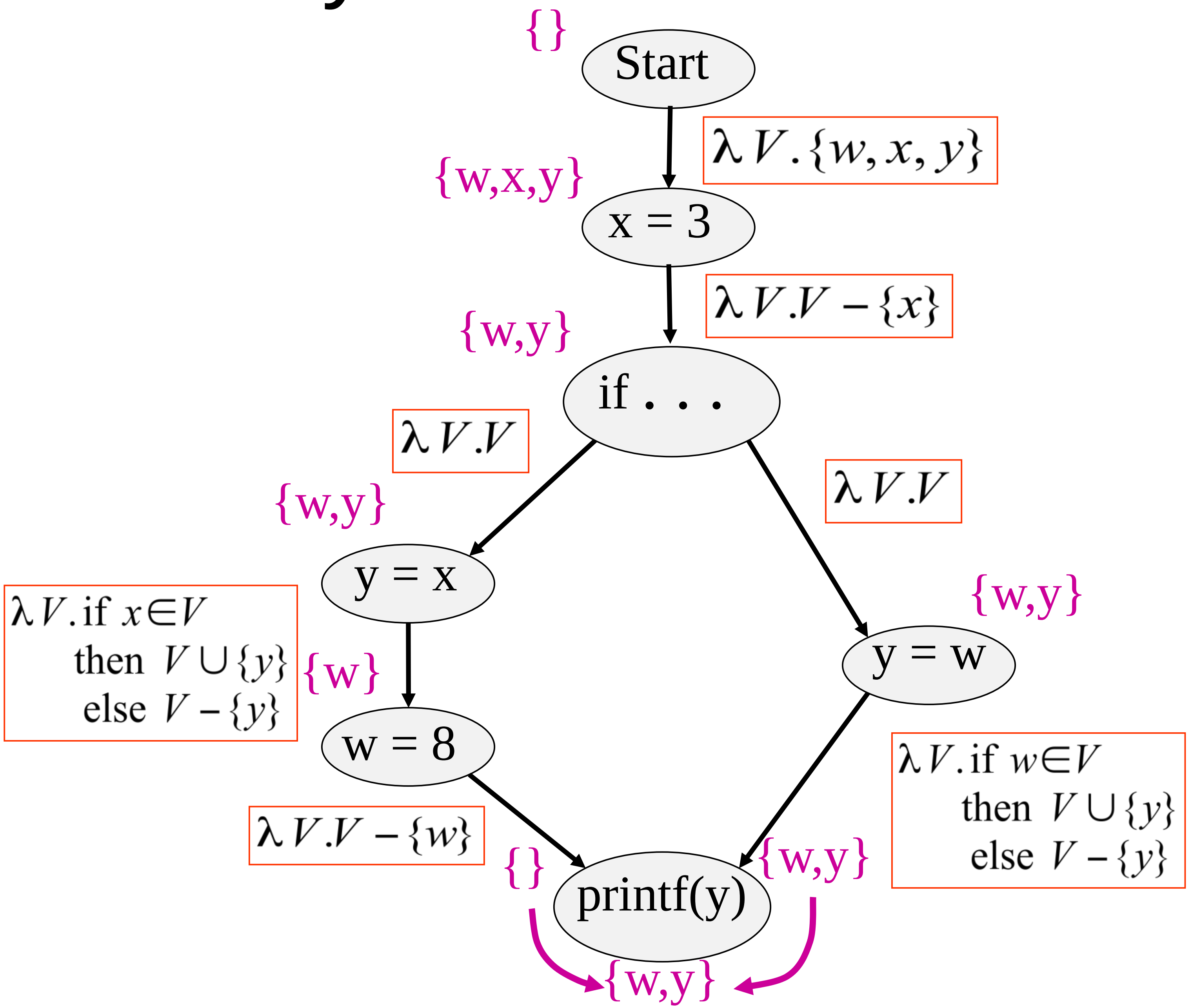
The Context-Free Reachability Problem

- A finite directed graph $G(s, V, E)$
- A finite alphabet Σ
- A labeling function $l: E \rightarrow \Sigma$
- 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

IFDS Problems

- IFDS= interprocedural, finite, distributive, subset
- Finite subset distributive
  - Lattice L = PowerSet(D), D=Data facts
  - Transfer functions L->L are distributive
- Efficient solution through formulation as CFL reachability
- Can be generalized to certain infinite lattices

Possibly Uninitialized Variables



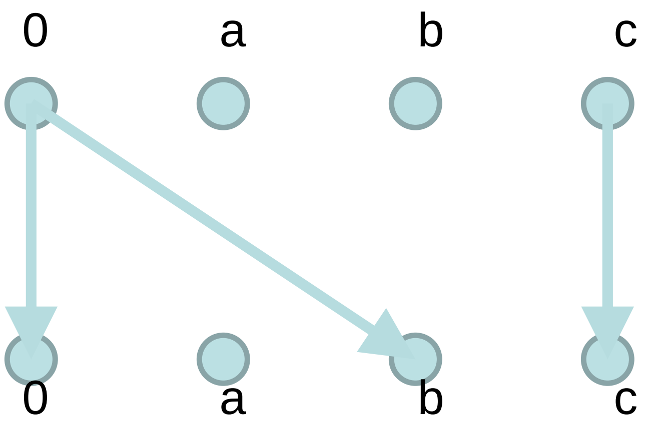
Efficiently Representing Functions

- Let $f: 2^D \rightarrow 2^D$ be a distributive function,
- i.e., $f(x \sqcap y) = f(x) \sqcap f(y)$
- 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}

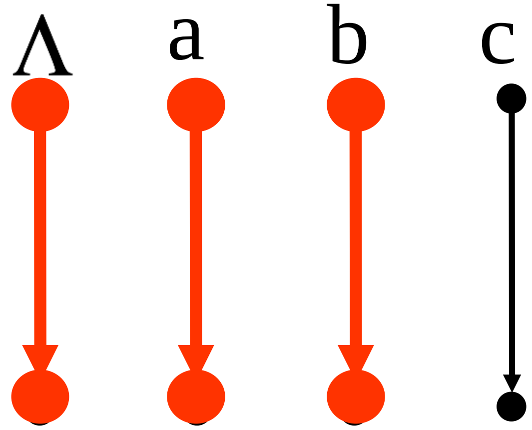


Representing Dataflow Functions

Identity Function

$$f = \lambda V.V$$

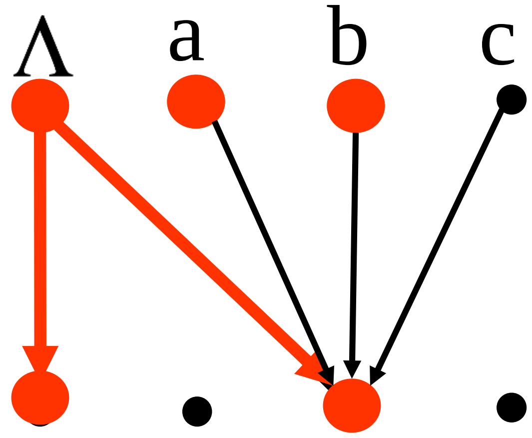
$$\boxed{f(\{a,b\}) = \{a,b\}}$$



Constant Function

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

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

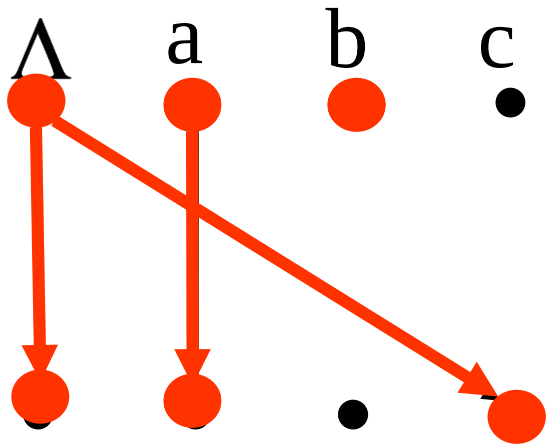


Representing Dataflow Functions

"Gen/Kill" Function

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

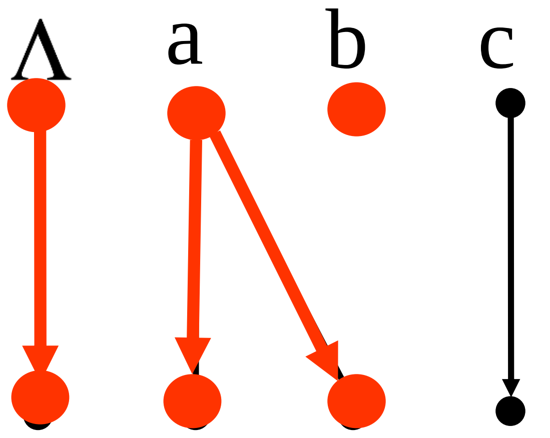
$$\boxed{f(\{a,b\}) = \{a,c\}}$$



Non-"Gen/Kill" Function

$$\begin{aligned} f = \lambda V.\ &\text{if } a \in V \\ &\text{then } V \cup \{b\} \\ &\text{else } V - \{b\} \end{aligned}$$

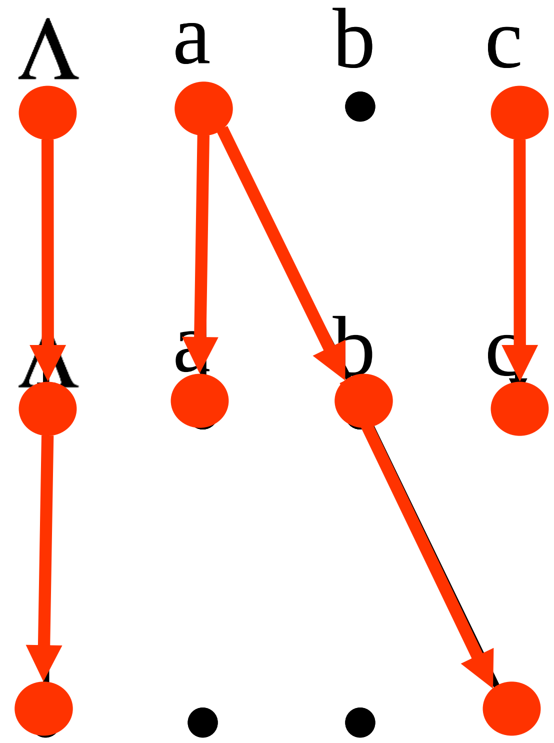
$$\boxed{f(\{a,b\}) = \{a,b\}}$$



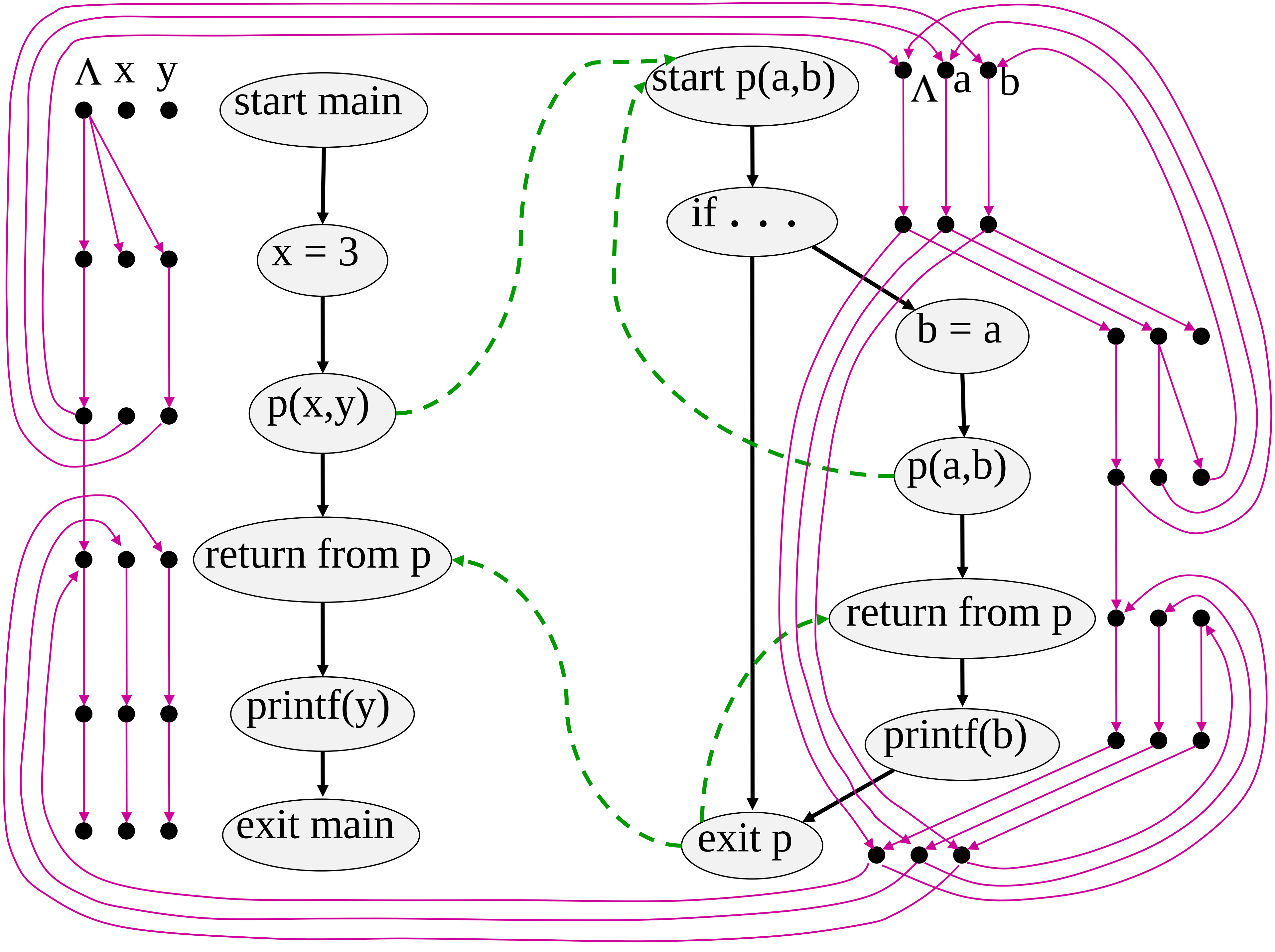
Composing Dataflow Functions

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

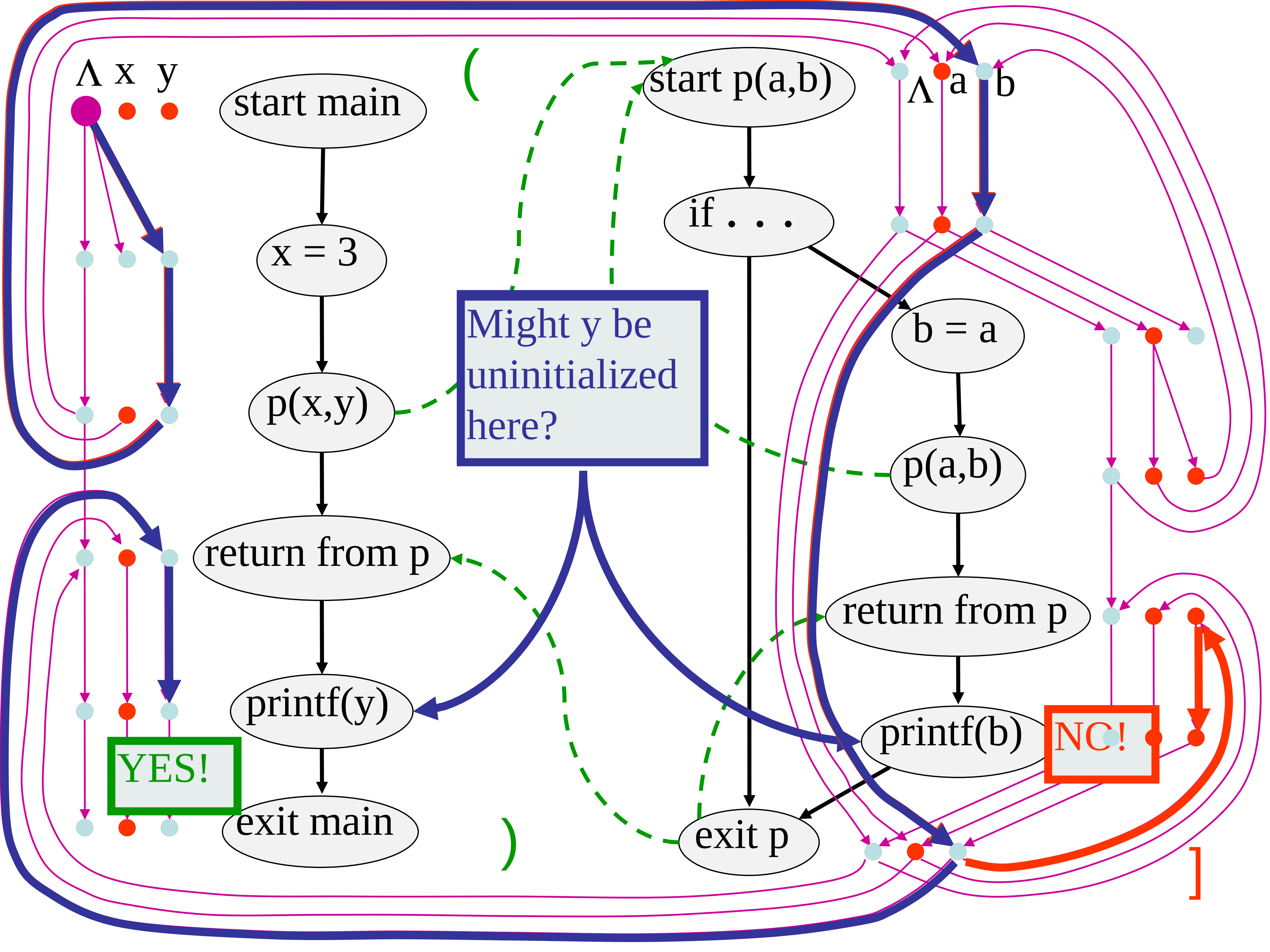
$$f_2 = \lambda V.\,\text{if } b \in V \text{ then } \{c\} \text{ else } \phi$$



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



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

```
declare PathEdge, WorkList, SummaryEdge: global edge set
algorithm Tabulate(G#_IP)
begin
  Let (N#, E#) = G#_IP
  PathEdge := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  WorkList := { ⟨s_main, 0⟩ → ⟨s_main, 0⟩ }
  SummaryEdge := ∅
  ForwardTabulateSLRPs()
  for each n ∈ N* do
    X_n := { d_2 ∈ D | ∃ d_1 ∈ (D ∪ { 0 }) such that ⟨s_procOf(n), d_1⟩ → ⟨n, d_2⟩ ∈ PathEdge }
  od
end
procedure Propagate(e)
begin
  if e ∉ PathEdge then Insert e into PathEdge; Insert e into WorkList fi
end
procedure ForwardTabulateSLRPs()
begin
  while WorkList ≠ ∅ do
    Select and remove an edge ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from WorkList
    switch n

      case n ∈ Call_p :
        for each d_3 such that ⟨n, d_2⟩ → ⟨s_calledProc(n), d_3⟩ ∈ E# do
          Propagate(⟨s_calledProc(n), d_3⟩ → ⟨s_calledProc(n), d_3⟩)
        od
        for each d_3 such that ⟨n, d_2⟩ → ⟨returnSite(n), d_3⟩ ∈ (E# ∪ SummaryEdge) do
          Propagate(⟨s_p, d_1⟩ → ⟨returnSite(n), d_3⟩)
        od
      end case

      case n = e_p :
        for each c ∈ callers(p) do
          for each d_4, d_5 such that ⟨c, d_4⟩ → ⟨s_p, d_1⟩ ∈ E# and ⟨e_p, d_2⟩ → ⟨returnSite(c), d_5⟩ ∈ E# do
            if ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ ∉ SummaryEdge then
              Insert ⟨c, d_4⟩ → ⟨returnSite(c), d_5⟩ into SummaryEdge
              for each d_3 such that ⟨s_procOf(c), d_3⟩ → ⟨c, d_4⟩ ∈ PathEdge do
                Propagate(⟨s_procOf(c), d_3⟩ → ⟨returnSite(c), d_5⟩)
              od
            fi
          od
        od
      end case

      case n ∈ (N_p − Call_p − { e_p }) :
        for each ⟨m, d_3⟩ such that ⟨n, d_2⟩ → ⟨m, d_3⟩ ∈ E# do
          Propagate(⟨s_p, d_1⟩ → ⟨m, d_3⟩)
        od
      end case

    end switch
  od
end
```

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

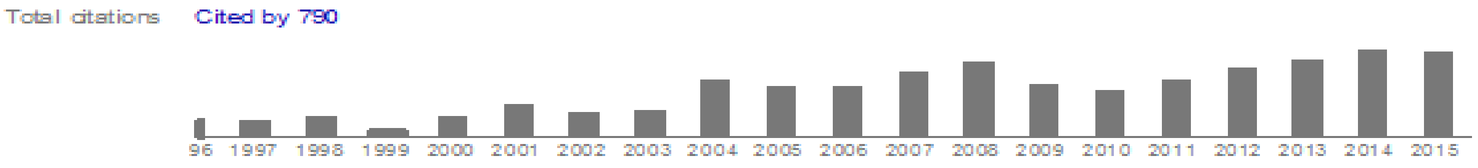
Manu Sridharan, Rastislav Bodík: Refinement-based context-sensitive points-to analysis for Java. PLDI 2006

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



IDE

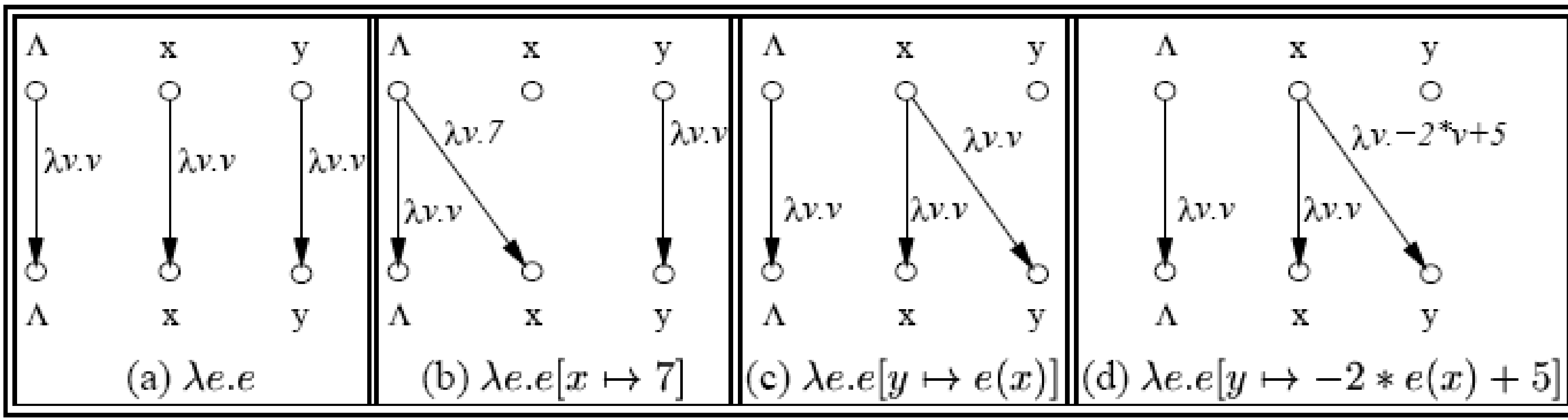
- IDE=Interprocedural Distributive Environment
- Goes beyond IFDS problems
  - Can handle unbounded domains
- Env: finite symbols->infinite domain
- Requires special form of the domain
- Can be **much** more efficient than IFDS

Precise interprocedural dataflow analysis with applications to constant propagation. Mooly Sagiv, Thomas Reps, Susan Horwitz. Theoretical Computer Science, 1996

IDE Analysis

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

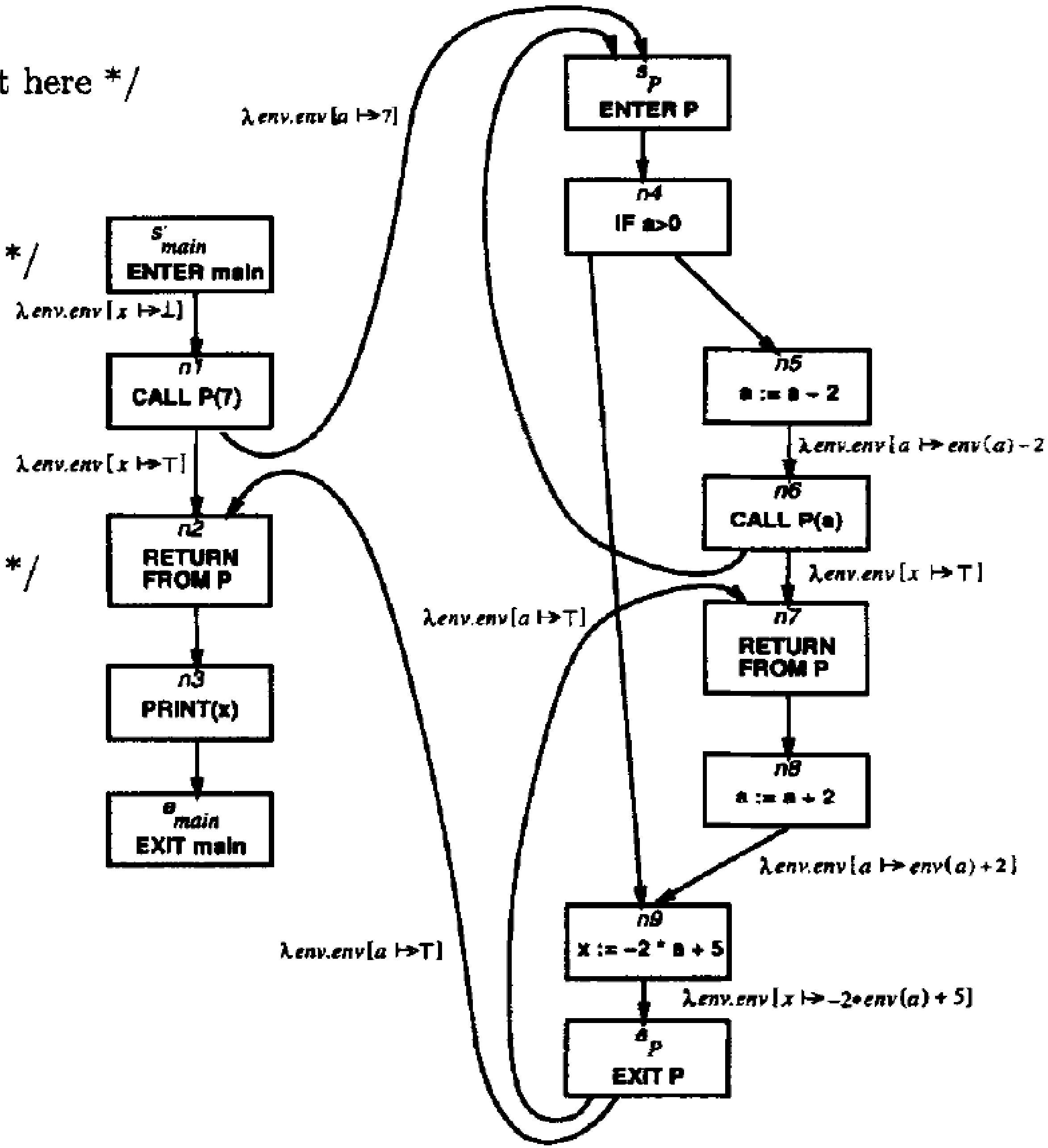
Linear constant propagation

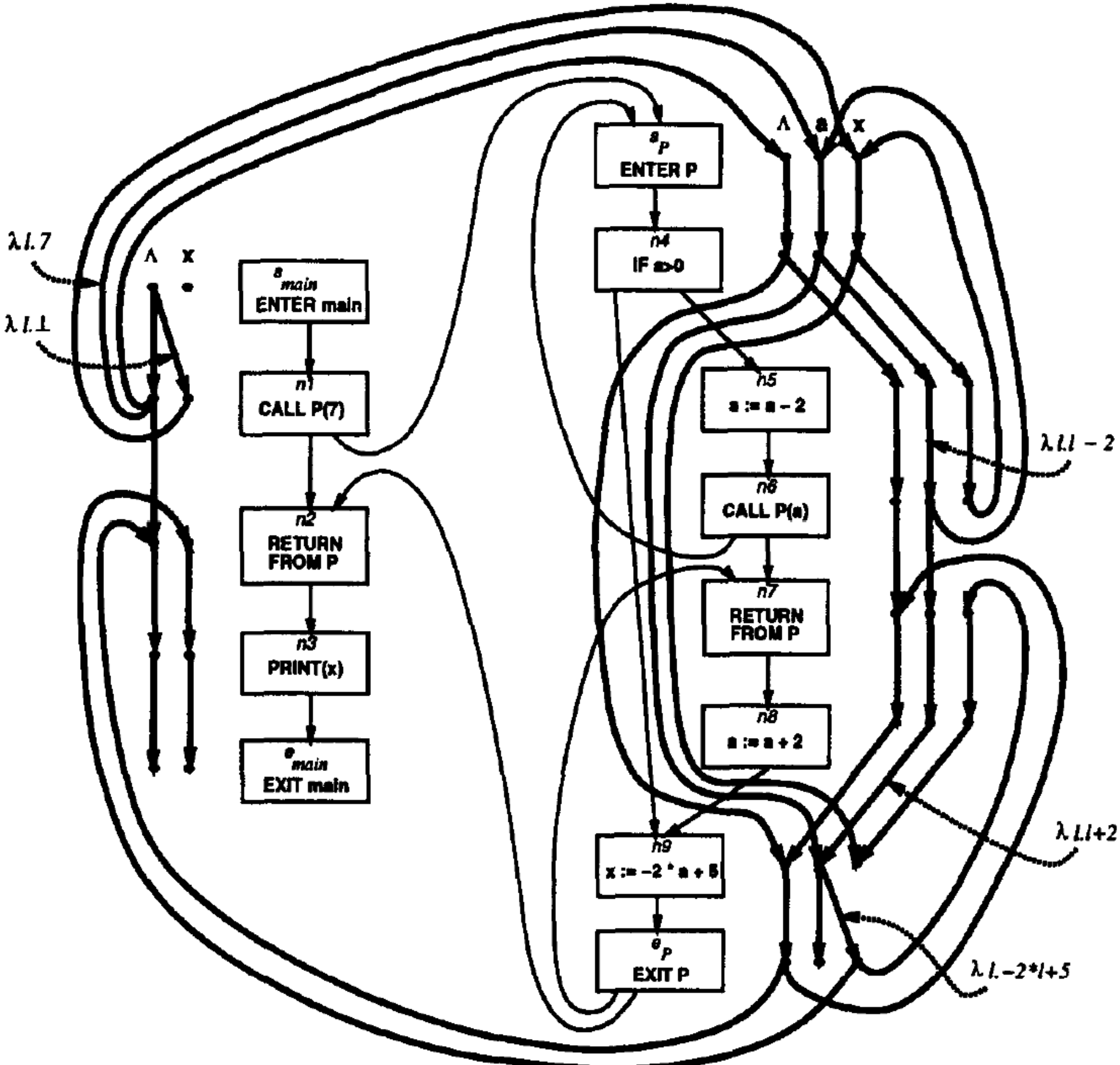


Point-wise representation of environment transformers

```
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 l. l − 2
λ l. l+2
λ l. −2*l+5

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*