

# Interprocedural Analysis

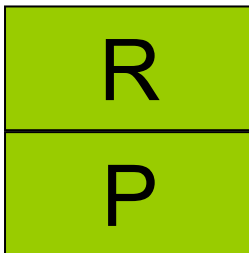
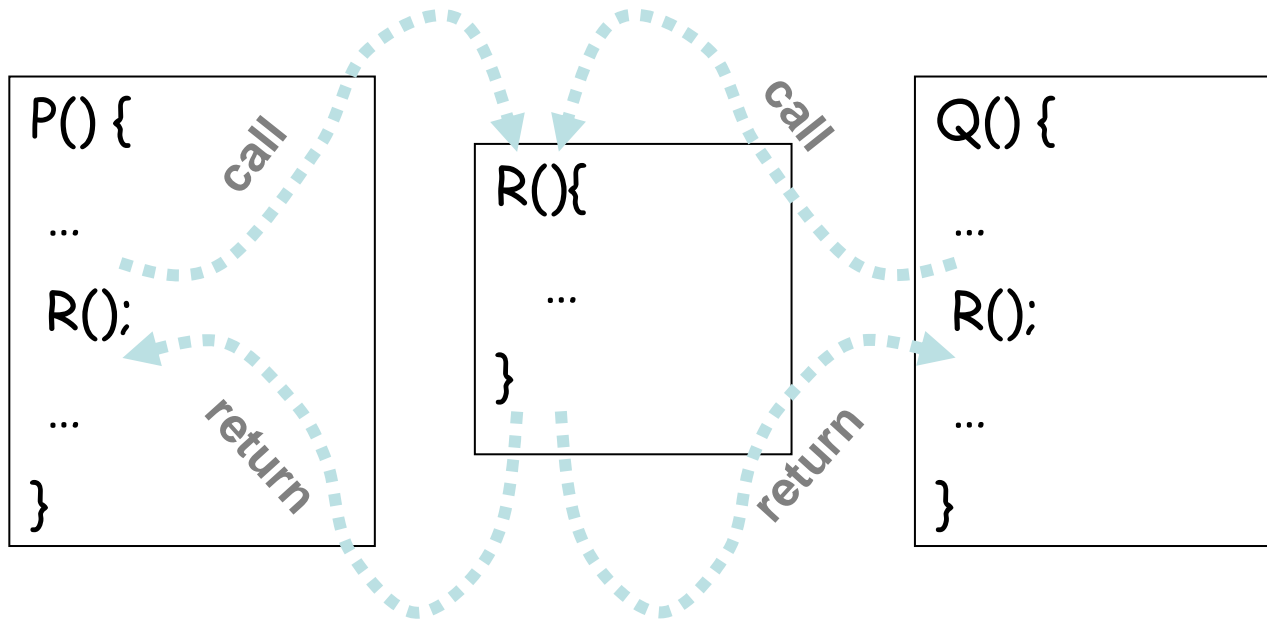
Noam Rinetzky

Mooly Sagiv

# Challenges in Interprocedural Analysis

- Respect call-return mechanism
- Handling recursion
- Local variables
- Parameter passing mechanisms
- The called procedure is not always known
- The source code of the called procedure is not always available
- Concurrency

# Stack regime



# Guiding light

- Exploit stack regime
  - Precision
  - Efficiency



# Simplifying Assumptions

- ✗ Parameter passed by value
- ✗ No OO
- ✗ No nesting
- ✗ No concurrency
- ✓ Recursion is supported

# Topics Covered

- The trivial approach
- Procedure Inlining
- The naive approach
- The call string approach
- Functional Approach
- Valid paths
- Interprocedural Analysis via Graph Reachability
- Beyond reachability
- [Demand analysis]
- [Advanced Topics]
  - Karr's analysis
  - Backward
  - MOPED

# Constant Propagation

- Find variables with constant value at a given program location

```
int p(int x) {
    return x * x;
}
void main() {
    {
    int z;
    if (...)
        z = p(5) + 5;
        else z = p(4) + 4;
    printf (z);
    }
```

# Example Program

$x = 5;$

$y = 7;$

if (...)

$y = x + 2;$

$z = x + y;$



# Undecidability Issues

- It is undecidable if a program point is reachable in some execution
- Some static analysis problems are undecidable even if the program conditions are ignored

# The Constant Propagation Example

```
while (...) {  
    if (...) x_1 = x_1 + 1;  
    if (...) x_2 = x_2 + 1;  
    ...  
    if (...) x_n = x_n + 1;  
}  
y = truncate (1/ (1 + p2(x_1, x_2, ..., x_n))  
/* Is y=0 here? */
```

# Conservative Solution

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

# The extra cost of procedures

- Call return complicates the analysis
- Unbounded stack memory
- Increases asymptotic complexity
- But can be tolerable in practice
- Sometimes even cheaper/more precise than intraprocedural analysis

# A trivial treatment of procedure

- Analyze a single procedure
- After every call continue with conservative information
  - Global variables and local variables which “may be modified by the call” have unknown values
- Can be easily implemented
- Procedures can be written in different languages
- Procedure inline can help

# Disadvantages of the trivial solution

- Modular (object oriented and functional) programming encourages small frequently called procedures
- Almost all information is lost

# Procedure Inlining

- Inline called procedures
- Simple
- Does not handle recursion
- Exponential blow up

```
p1 {  
  call p2  
  ...  
  call p2  
}  
  
p2 {  
  call p3  
  ...  
  call p3  
}  
  
p3{  
  }
```

# A Naive Interprocedural solution

- Treat procedure calls as gotos
- Abstract call/return correlations
- Obtain a conservative solution
- Use chaotic iterations
- Usually fast



# 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 ↦ 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]  
}
```



# The Call String Approach

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

# 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]  
}
```

```

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 small call strings
- For finite domains can be precise even with recursion
- Limited precision
- 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

# 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);  
}
```

```
→ 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 ↦ τ]  
        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 ↦ τ]  
        → a = a + 1;  
        [a ↦ a0, x ↦ τ]  
    }  
    [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 ↦ -2(a0-1) + 5]
```

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

```
        [a ↦ a0, x ↦ -2(a0-1) + 5]
```

```
    }
```



```
    [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) {  
  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;  
    }  
    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(a0,x0) = [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;

```
[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]  
}
```

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

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

# Phase 2

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

```
int p(int a) {  
    [a ↦ τ, x ↦ 0]  
    if (...) {  
        a = a - 1 ;  
        [a ↦ τ, x ↦ 0]  
        p (a);  
        [a ↦ τ, x ↦ τ]  
        a = a + 1;  
        [a ↦ τ, x ↦ τ]  
    }  
}
```

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



```
[a ↦ τ, x ↦ τ]  
x = -2*a + 5;  
[a ↦ 7, x ↦ -9]  
}
```

# Phase 2

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

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

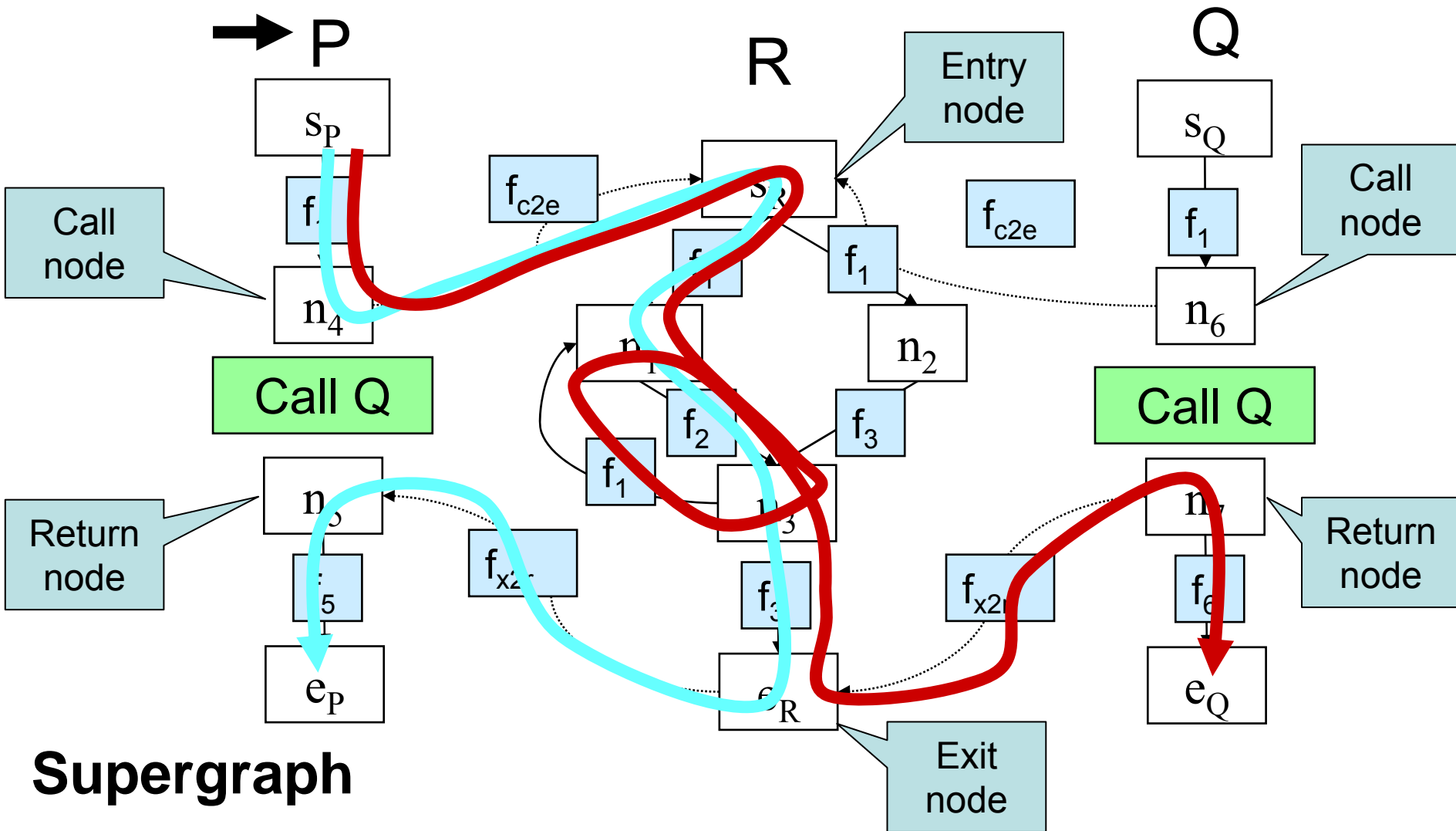
```
int p(int a) {  
    [a ↦ τ, x ↦ 0]  
    if (...) {  
        a = a - 1 ;  
        [a ↦ τ, x ↦ 0]  
        p (a);  
        [a ↦ τ, x ↦ τ]  
        a = a + 1;  
        [a ↦ τ, x ↦ τ]  
    }  
    [a ↦ τ, x ↦ τ]  
    x = -2*a + 5;  
    [a ↦ τ, x ↦ τ]  
}
```



# Summary Functional approach

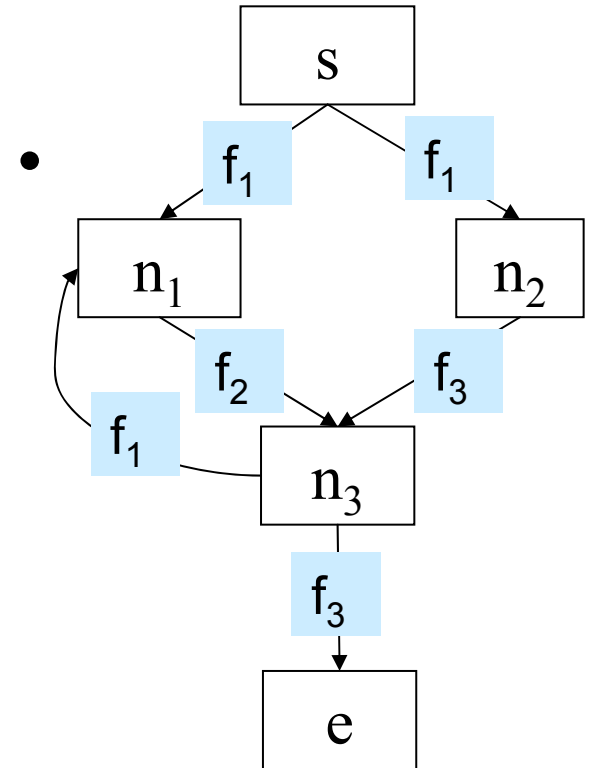
- Computes procedure abstraction
- Sharing between different contexts
- Rather precise
- Recursive procedures may be more precise/efficient than loops
- But requires more from the implementation
  - Representing relations
  - Composing relations

# Interprocedural analysis



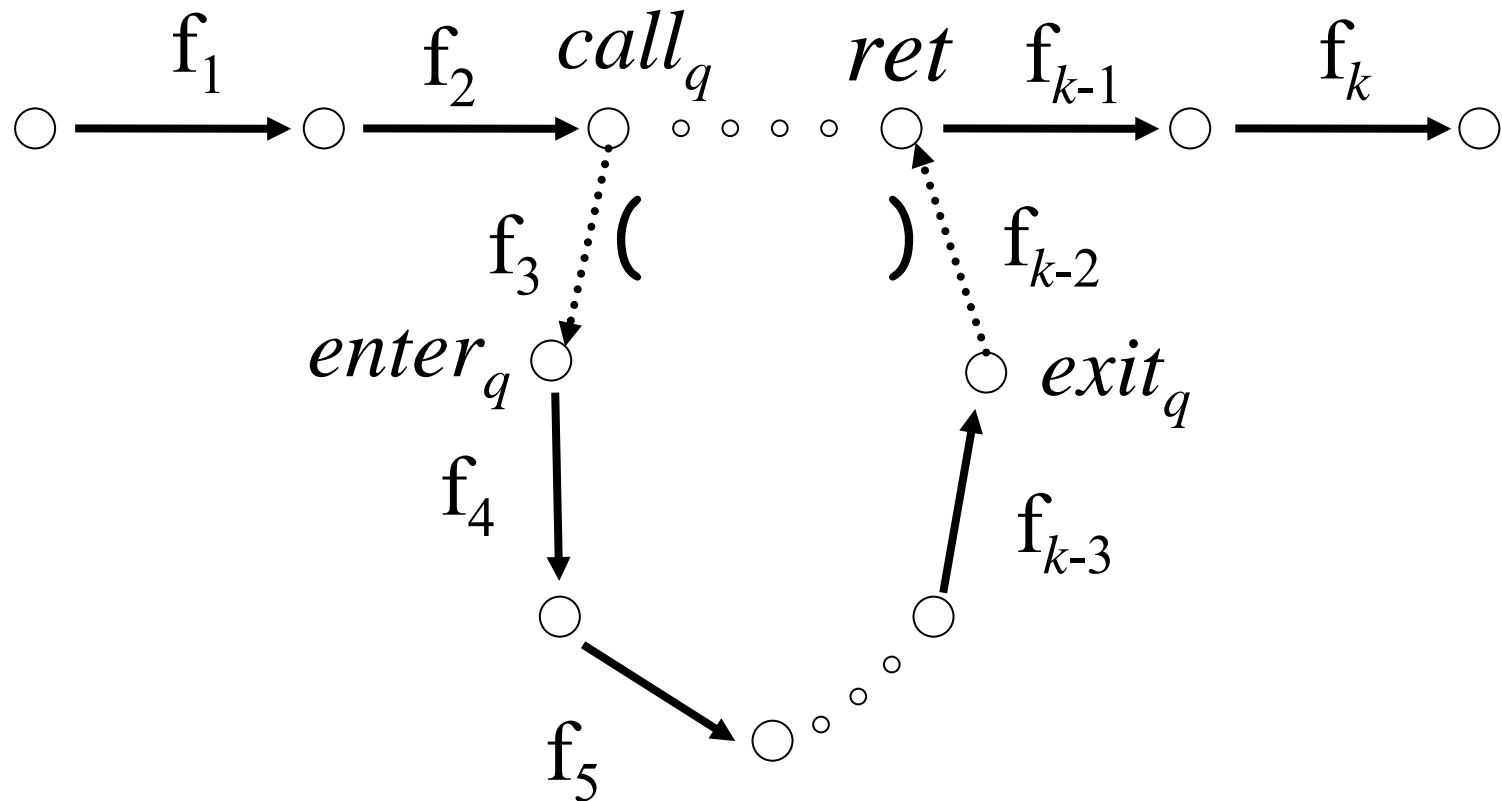
# Paths

**paths(n)** the set of paths from s to n  
( (s,n<sub>1</sub>), (n<sub>1</sub>,n<sub>3</sub>), (n<sub>3</sub>,n<sub>1</sub>) ) –





# Interprocedural Valid Paths

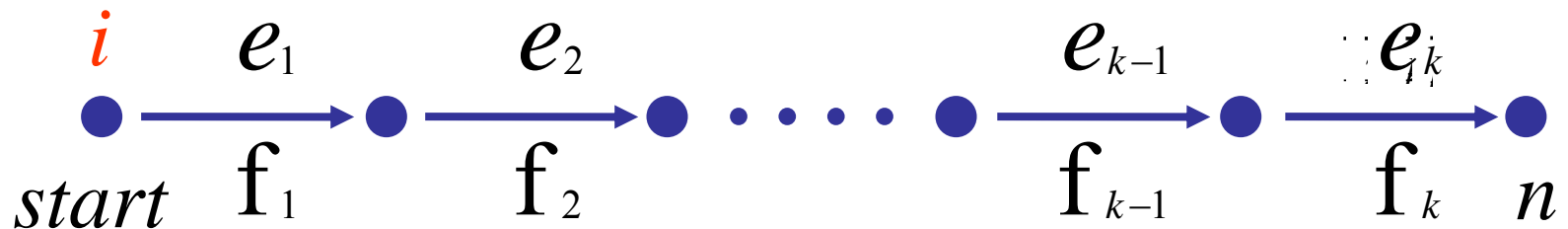


- IVP: all paths with matching calls and returns
  - And prefixes

# 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
  - $\text{matched} ::= \text{matched } ( ; \text{matched } ) ; | \varepsilon$
  - $\text{valid} ::= \text{valid } ( ; \text{matched } | \text{matched}$ 
    - *paths* can be defined by a regular expression

# Join Over All Paths (JOP)



$$[[ (e_1, \dots, e_k) ]] = f_k \circ f_{k-1} \cdots \circ f_2 \circ f_1 \in L \rightarrow L$$

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

# The Join-Over-Valid-Paths (JVP)

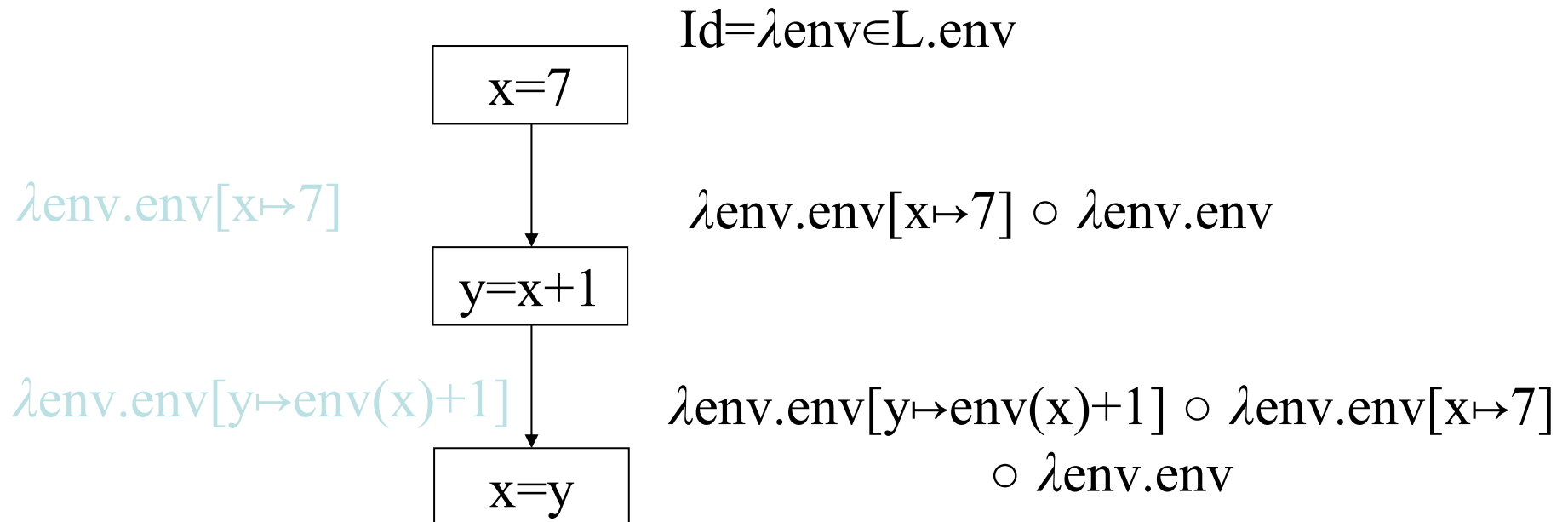
- $vpaths(n)$  all valid paths from program start to  $n$
- $JVP[n] = \sqcup \{ [[e_1, e_2, \dots, e]](1) \mid (e_1, e_2, \dots, e) \in vpaths(n) \}$
- $JVP \sqsubseteq JFP$ 
  - In some cases the JVP can be computed
  - (Distributive problem)

# Main idea functional approach

- 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

# Example: Constant propagation

- $L = \text{Var} \rightarrow \mathbb{N} \cup \{\tau, \perp\}$
- Domain:  $F: L \rightarrow L$ 
  - $(f_1 \sqcup f_2)(x) = f_1(x) \sqcup f_2(x)$



# Example: Constant propagation

- $L = \text{Var} \rightarrow \mathbb{N} \cup \{\tau, \perp\}$
- Domain:  $F: L \rightarrow L$ 
  - $(f_1 \sqcup f_2)(x) = f_1(x) \sqcup f_2(x)$

$\text{Id} = \lambda \text{env}. \text{env}$

$x=7$

$\lambda \text{env}. \text{env}[x \mapsto 7]$

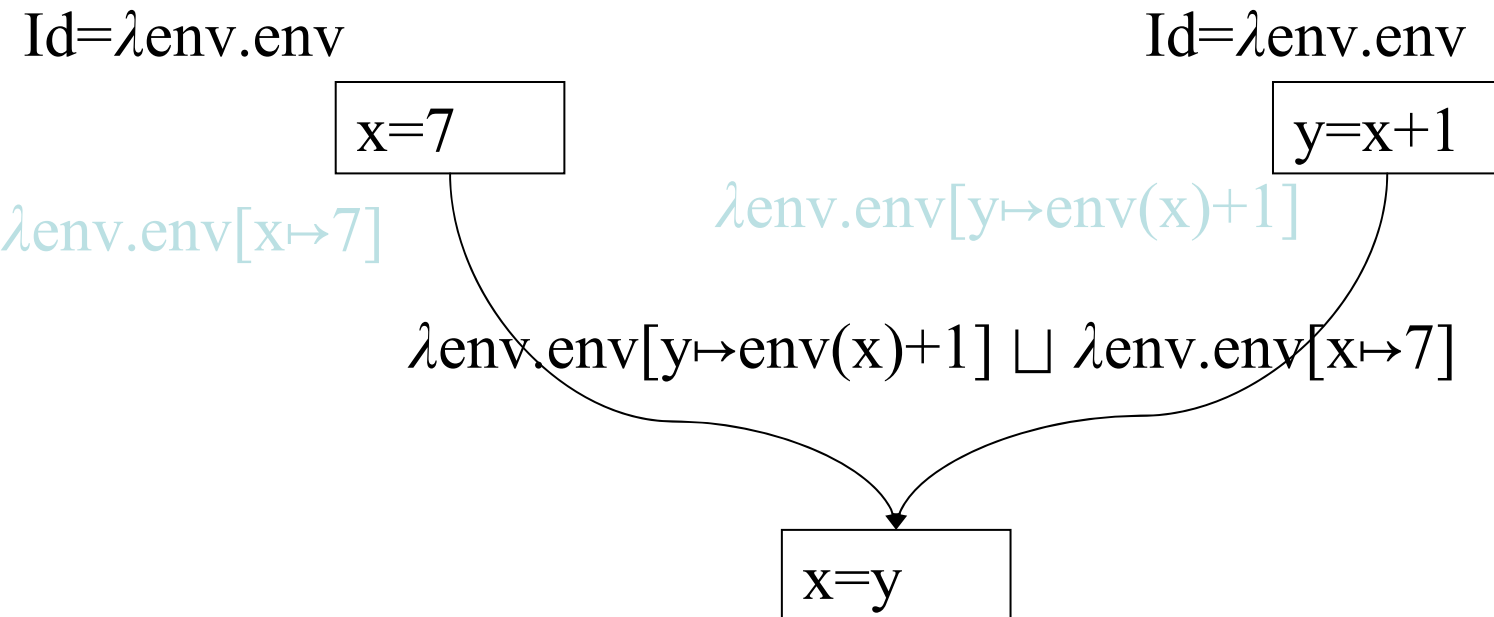
$\text{Id} = \lambda \text{env}. \text{env}$

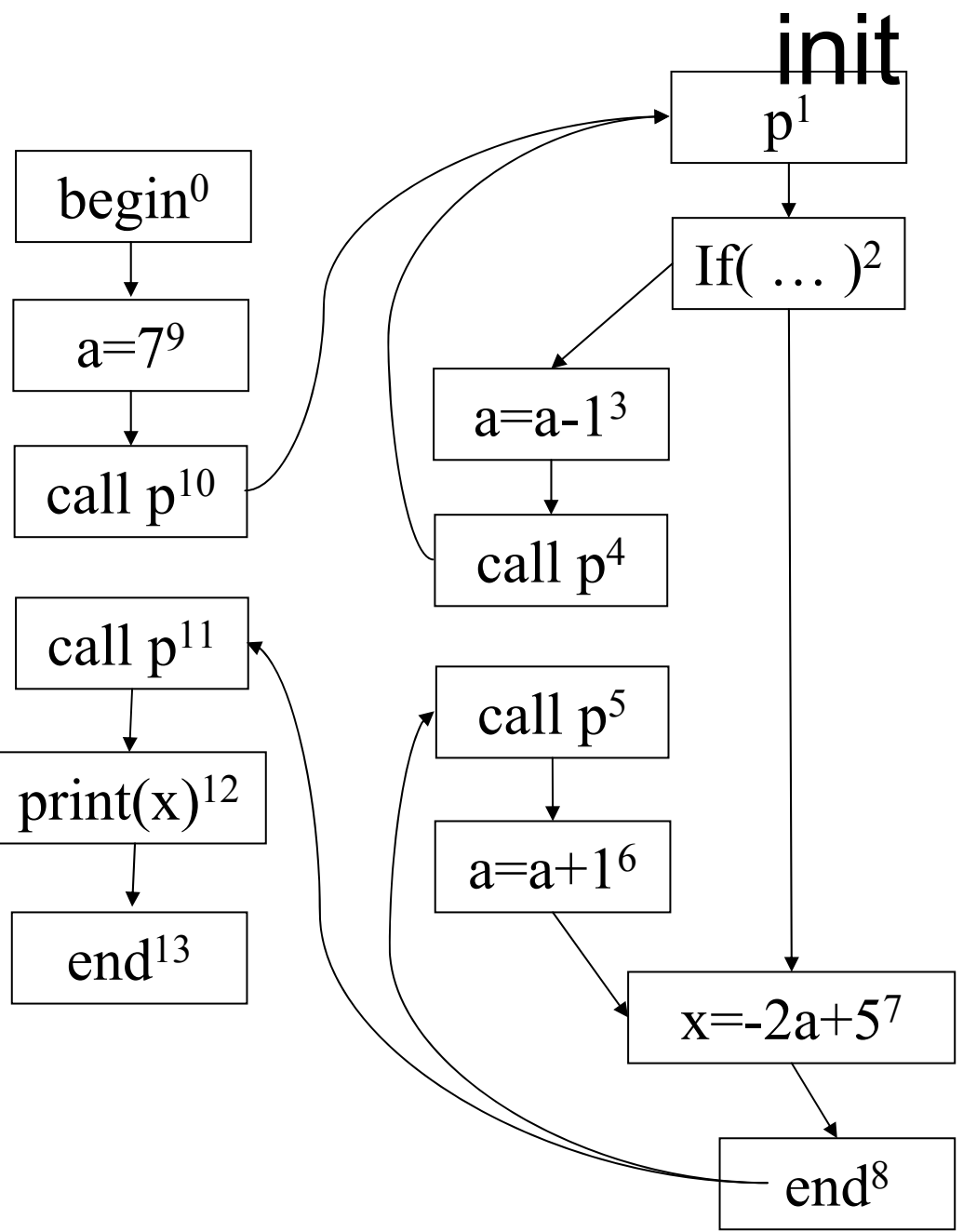
$y=x+1$

$\lambda \text{env}. \text{env}[y \mapsto \text{env}(x)+1]$

$\lambda \text{env}. \text{env}[y \mapsto \text{env}(x)+1] \sqcup \lambda \text{env}. \text{env}[x \mapsto 7]$

$x=y$

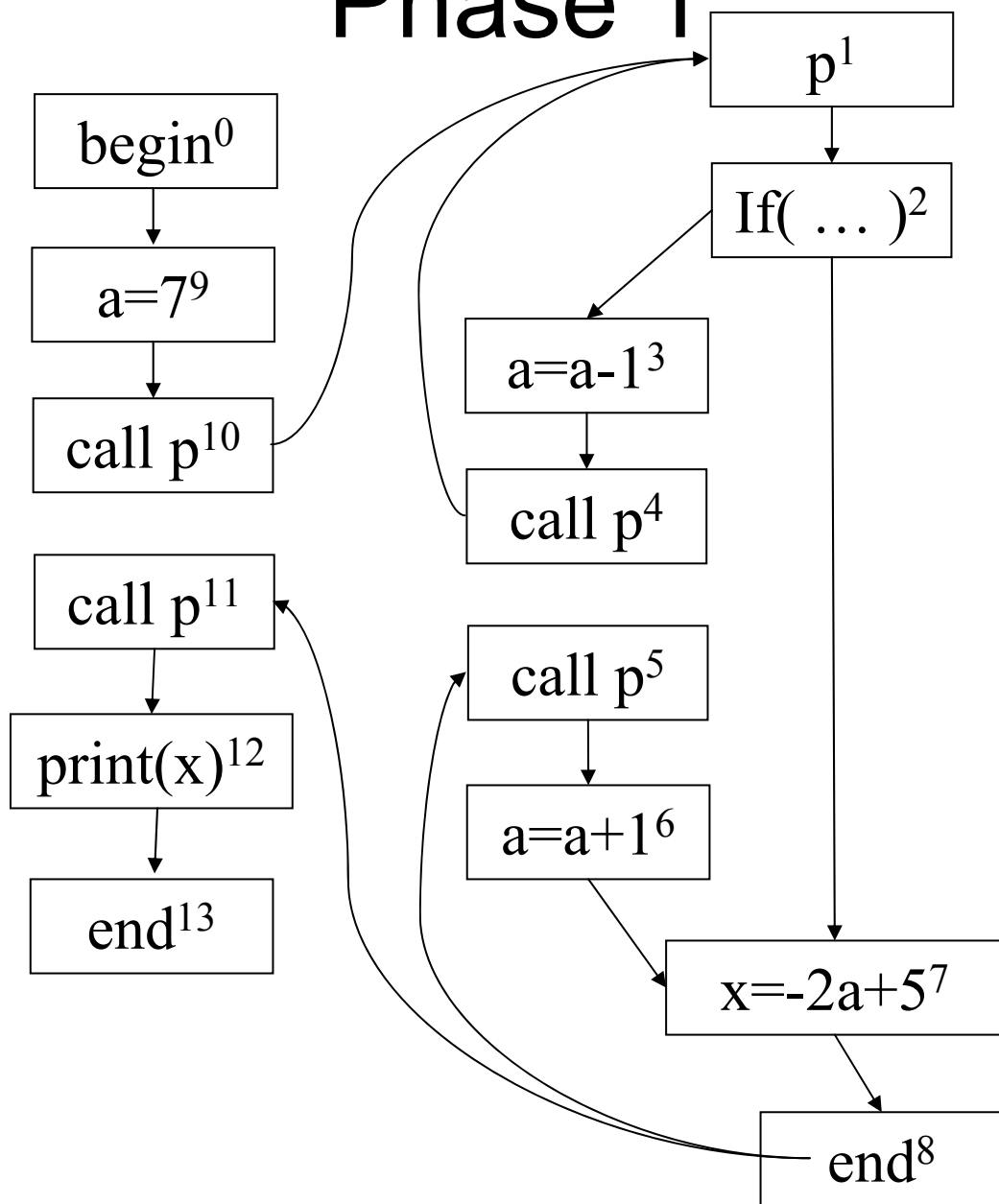




N	Function
0	$\lambda e.[x \mapsto e(x), a \mapsto e(a)] = id$
1	$\lambda e.[x \mapsto e(x), a \mapsto e(a)] = id$
3-13	$\lambda e.\perp$



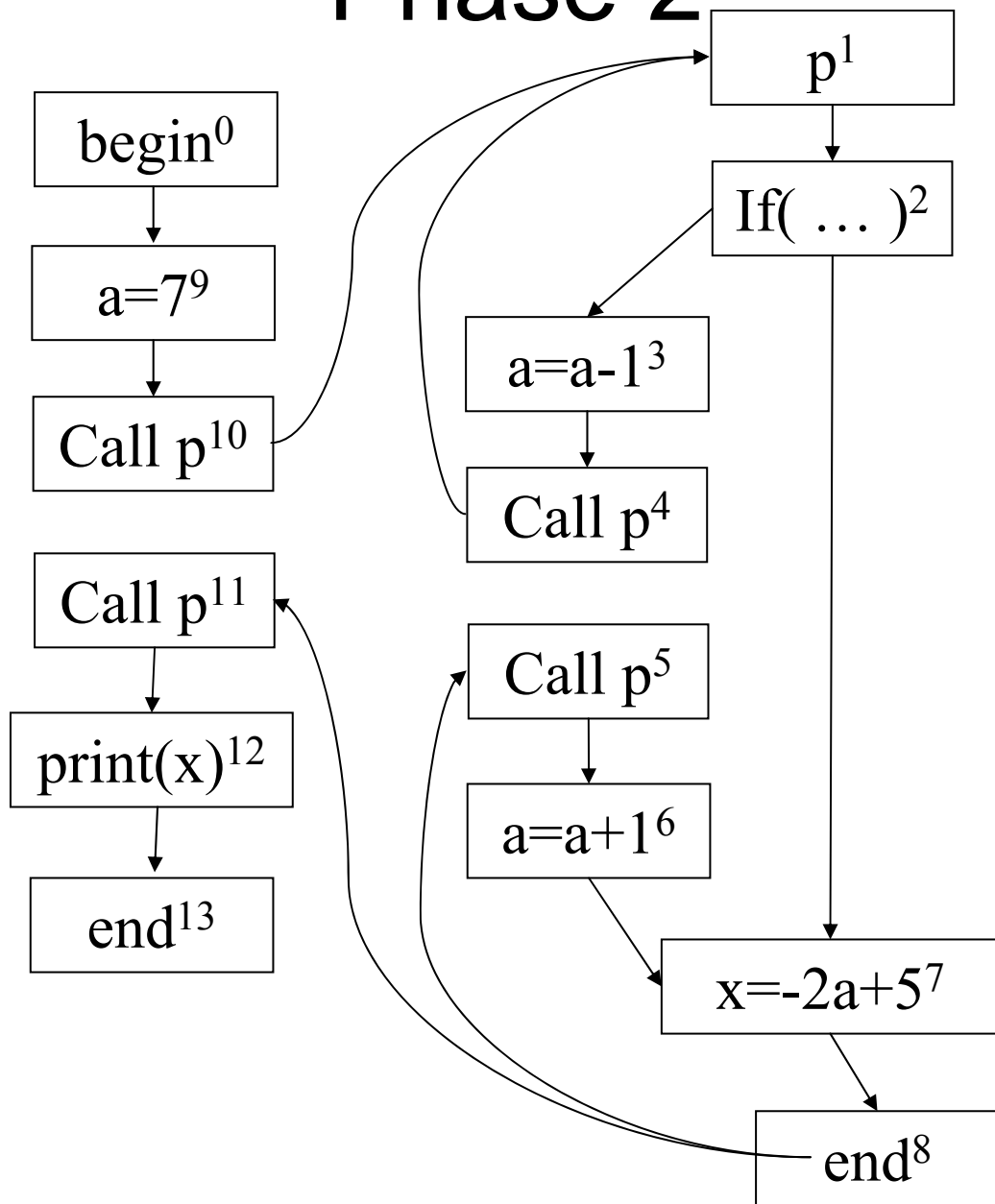
# Phase 1



N	Function
1	$\lambda e. [x \mapsto e(x), a \mapsto e(a)] = \text{id}$
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	$\lambda e. [x \mapsto e(x), a \mapsto e(a)] = \text{id}$
10	$\lambda e. [x \mapsto e(x), a \mapsto 7]$
11	$\lambda a, x. [x \mapsto -2e(a)+5, a \mapsto e(a)] \circ f_{10}$

# Phase 2



N	Function
1	$[x \mapsto 0, a \mapsto 7]$
2	$[x \mapsto 0, a \mapsto 7]$
7	$[x \mapsto 0, a \mapsto 7]$
8	$[x \mapsto -9, a \mapsto 7]$
3	$[x \mapsto 0, a \mapsto 7]$
4	$[x \mapsto 0, a \mapsto 6]$
1	$[x \mapsto -7, a \mapsto 6]$
6	$[x \mapsto -7, a \mapsto 7]$
7	$[x \mapsto \tau, a \mapsto 7]$
8	$[x \mapsto -9, a \mapsto 7]$
1	$[x \mapsto \tau, a \mapsto \tau]$

0	$[x \mapsto 0, a \mapsto 0]$
10	$[x \mapsto 0, a \mapsto 7]$
11	$[x \mapsto -9, a \mapsto 7]$

# 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

# CFL-Graph reachability

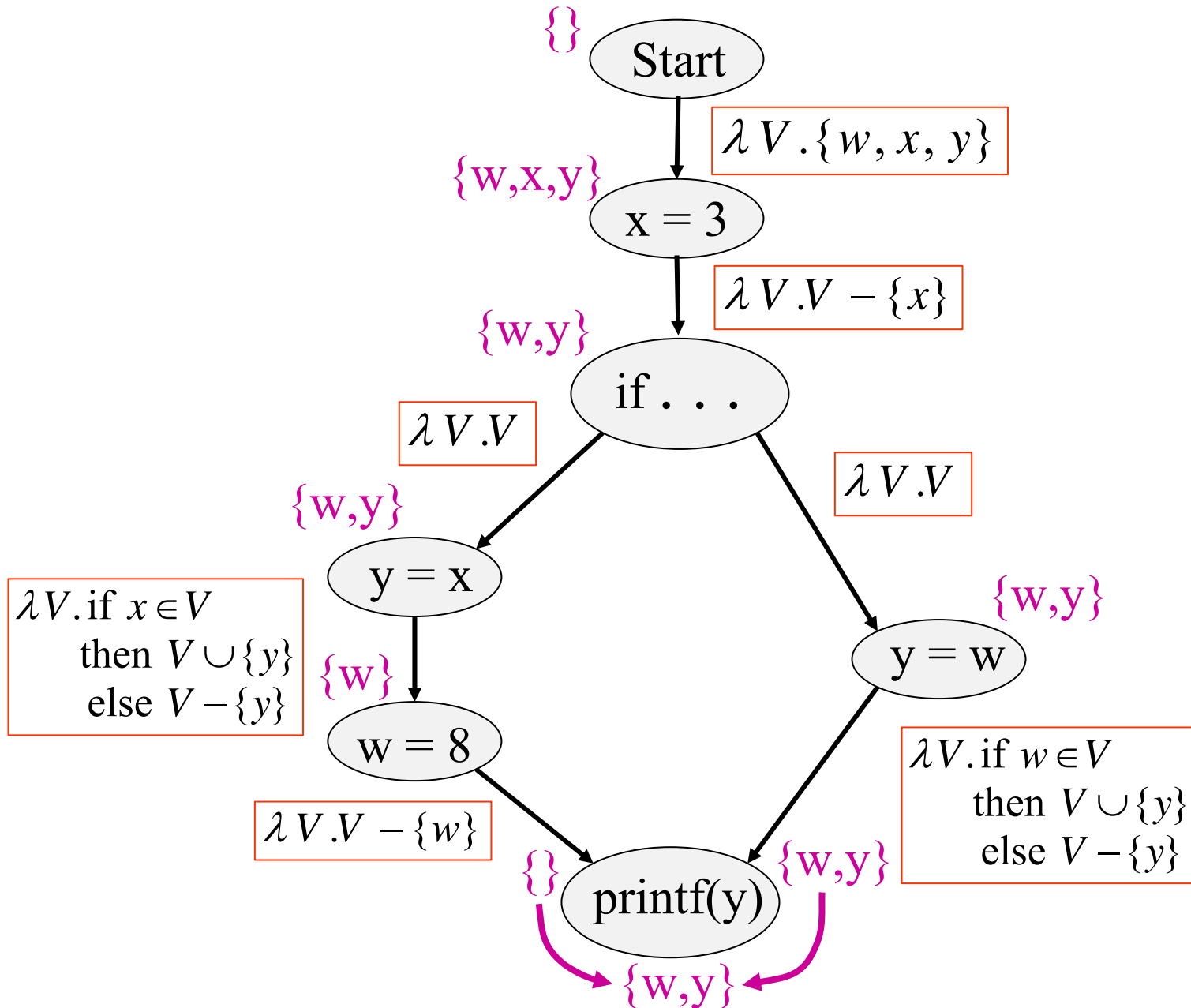
- Special cases of functional analysis
- Finite distributive lattices
- Provides more efficient analysis algorithms
- Reduce the interprocedural analysis problem to finding context free reachability



# IDFS / IDE

- **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.** *Reps, Horowitz, and Sagiv, FASE'95, TCS'96*
  - *More general solutions exist*

# Possibly Uninitialized Variables



# Efficiently Representing Functions

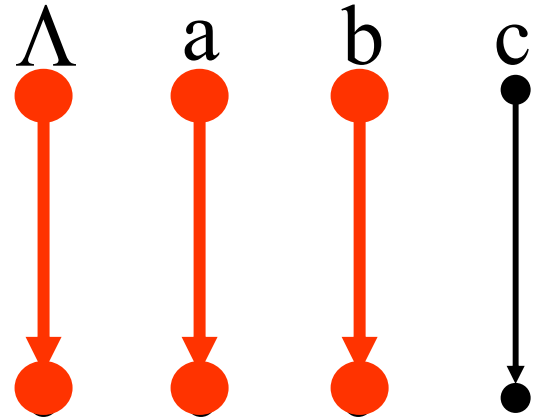
- Let  $f:2^D \rightarrow 2^D$  be a distributive function
- Then:  $f(X) = f(\emptyset) \cup \cup\{z \in X: f(\{z\})\}$

# Representing Dataflow Functions

Identity Function

$$f = \lambda V.V$$

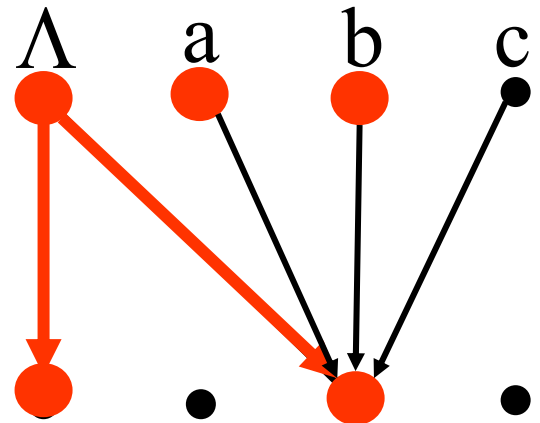
$$f(\{a, b\}) = \{a, b\}$$



Constant Function

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

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



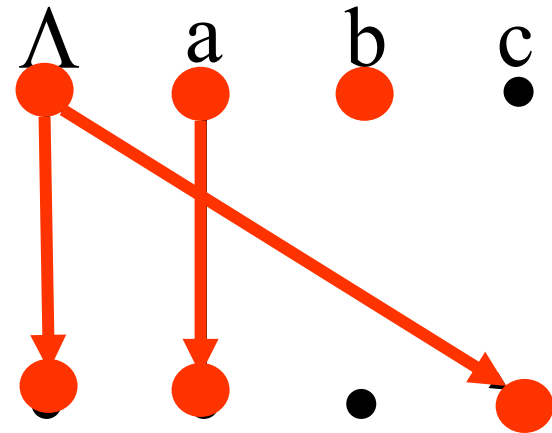


# Representing Dataflow Functions

“Gen/Kill” Function

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

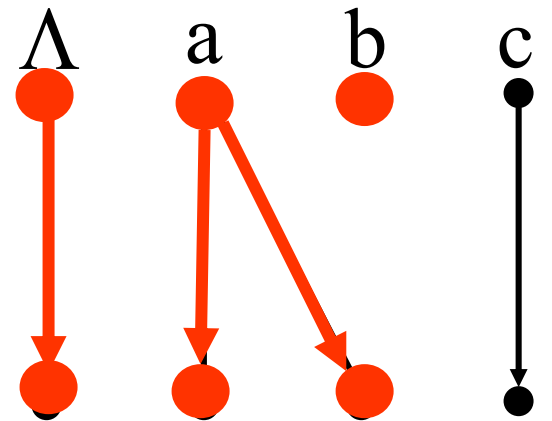
$$f(\{a, b\}) = \{a, c\}$$

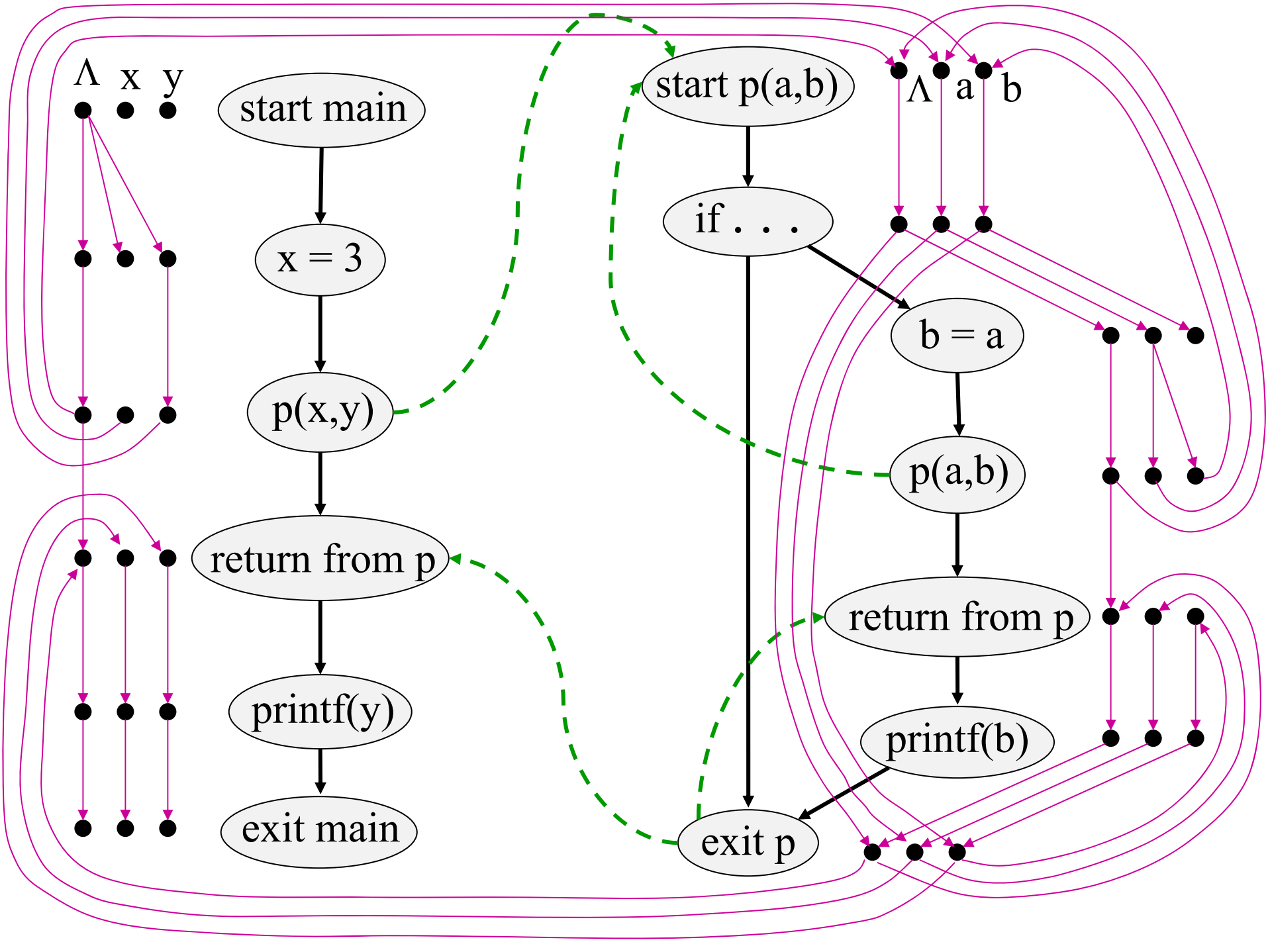


Non-“Gen/Kill” Function

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

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

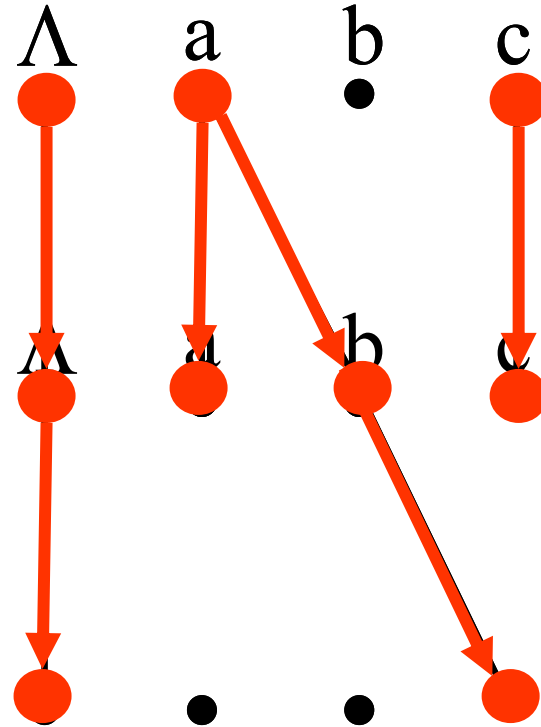




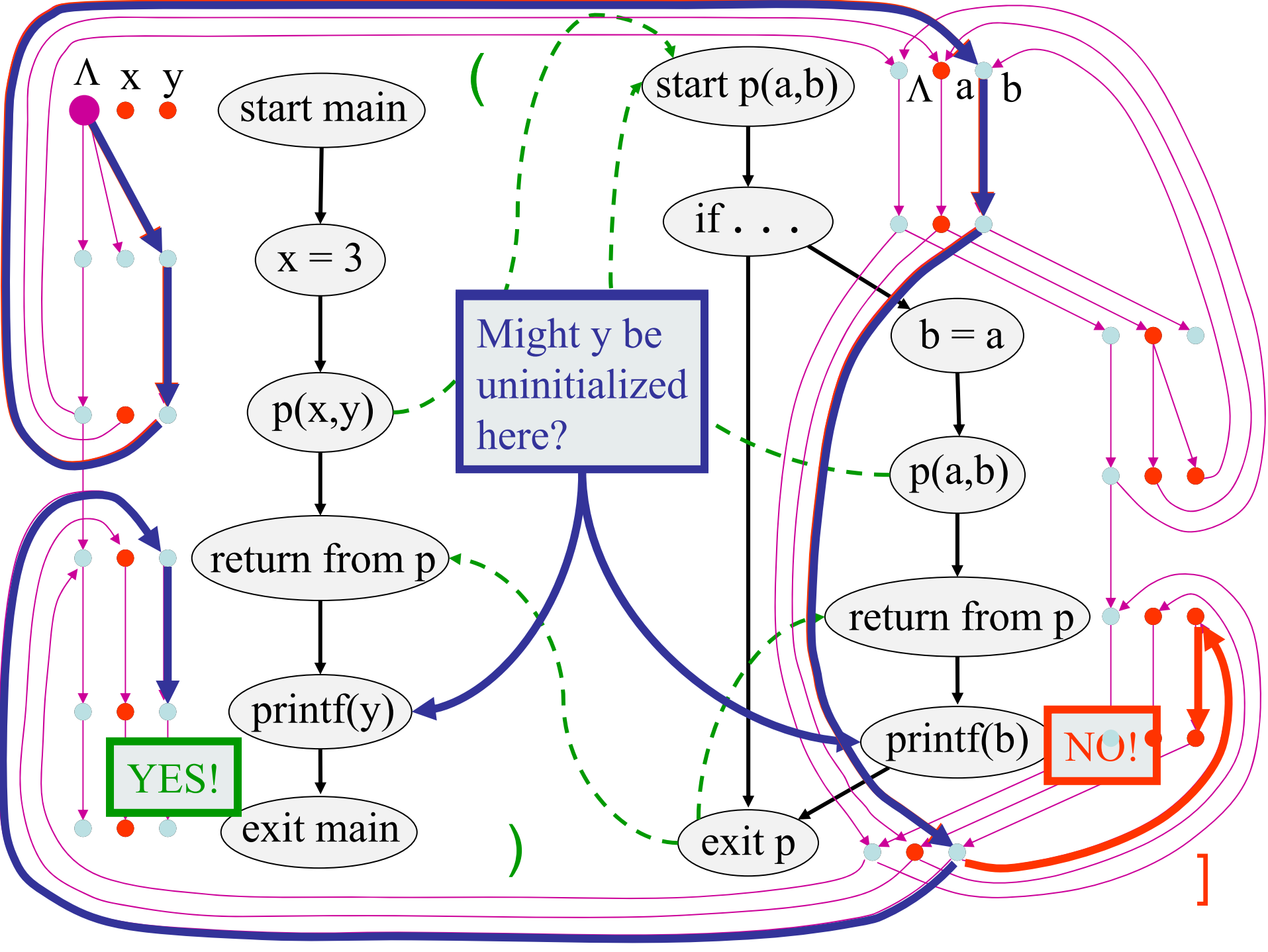
# Composing Dataflow Functions

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

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




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



# Interprocedural Dataflow Analysis via CFL-Reachability

- Graph: Exploded control-flow graph
- $L$ :  $L(\text{unbalLeft})$ 
  - $\text{unbalLeft} = \text{valid}$
- Fact  $d$  holds at  $n$  iff there is an  $L(\text{unbalLeft})$ -path from  $\langle \text{start}_{\text{main}}, \Lambda \rangle$  to  $\langle n, d \rangle$

# Asymptotic Running Time

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

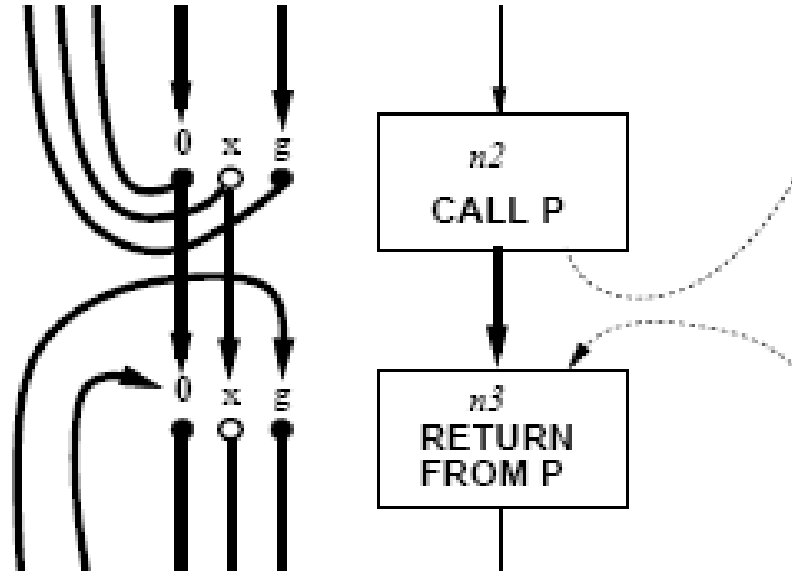
Running time:  $O(ED^3)$

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

“Gen/kill” problems:  $O(ED)$

# Local variables

- Global information:
  - Exit-to-return edges
- Local information:
  - Call-to-return edges
- Can be generalized to handle binary combiner functions
  - Rinetzky, Sagiv, and Yahav, SAS'05



# IDE

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



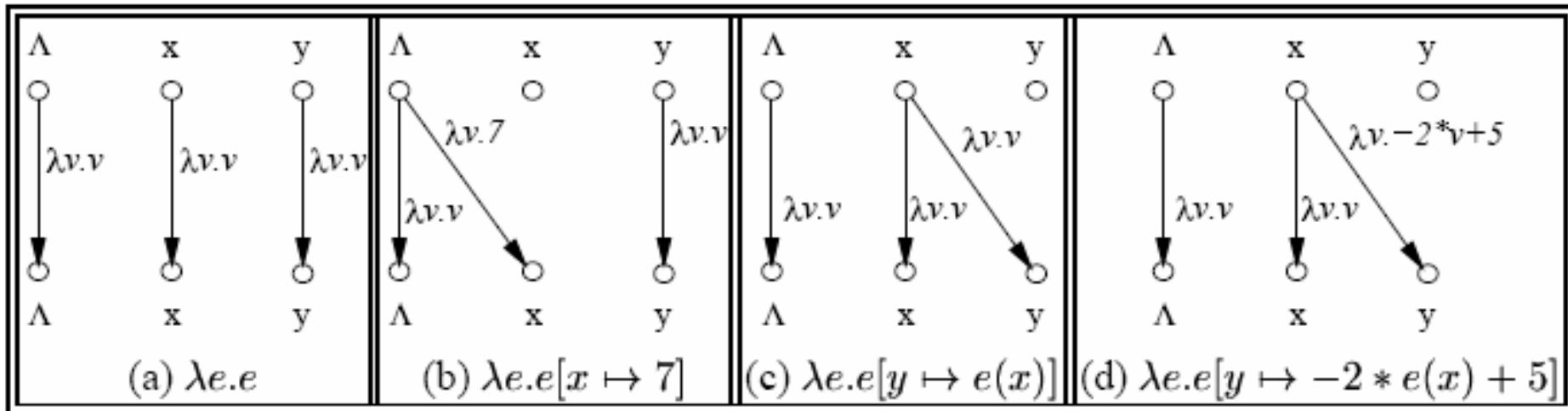
# Example Linear Constant Propagation

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

$$y = \sqcup \{(a_x x + b_x) \mid x \in \text{Var}_*\} \sqcup c$$
$$a_x, c \in \mathbb{Z} \cup \{\perp, \top\} \quad b_x \in \mathbb{Z}$$

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

# Linear constant propagation



Point-wise representation of environment transformers

# IDE Analysis

- Point-wise representation closed under composition
- CFL-Reachability on the exploded graph
- Compose functions

```

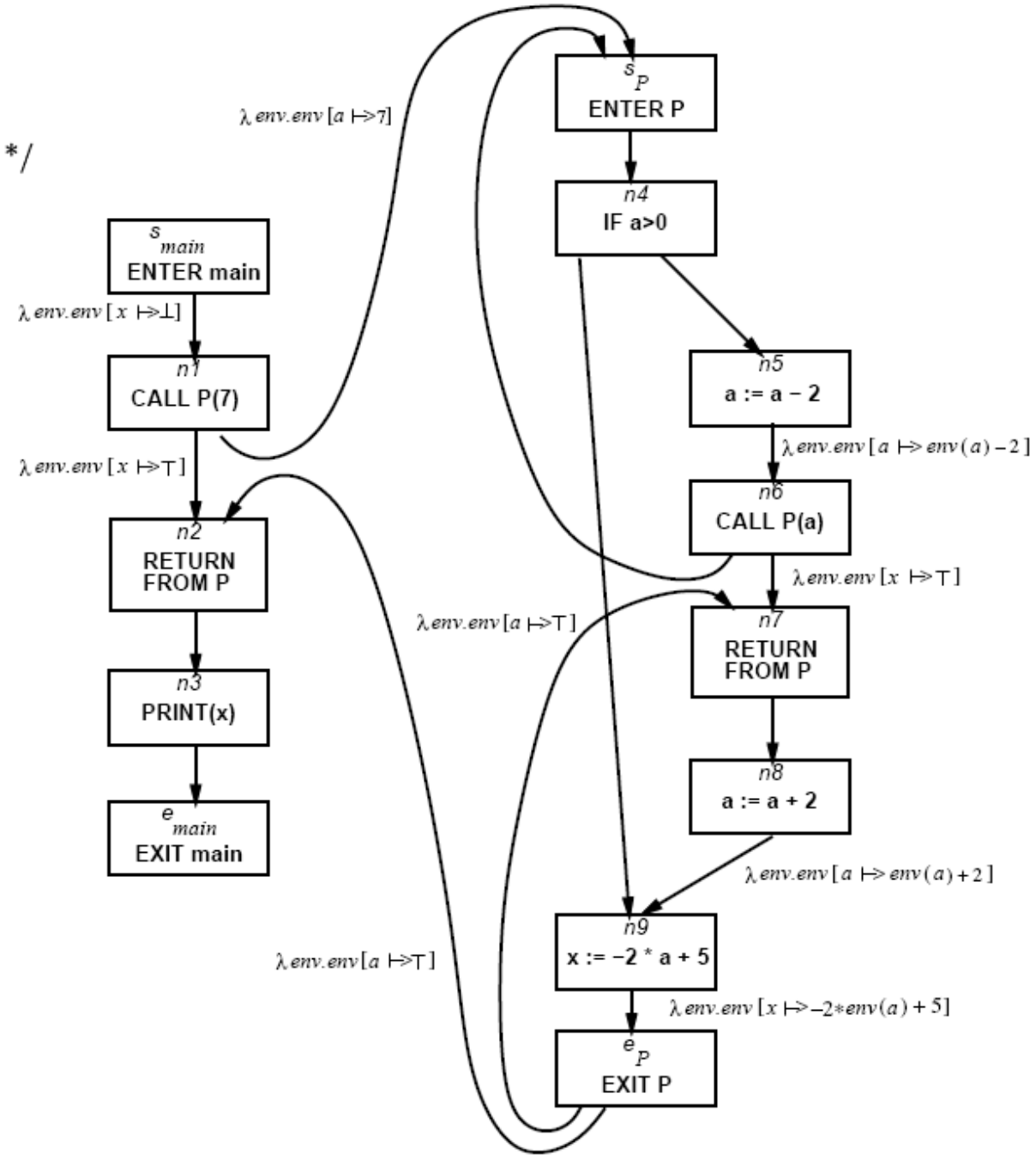
declare x: integer
program main
begin
  call P(7)
  print (x) /* x is a constant here */
end

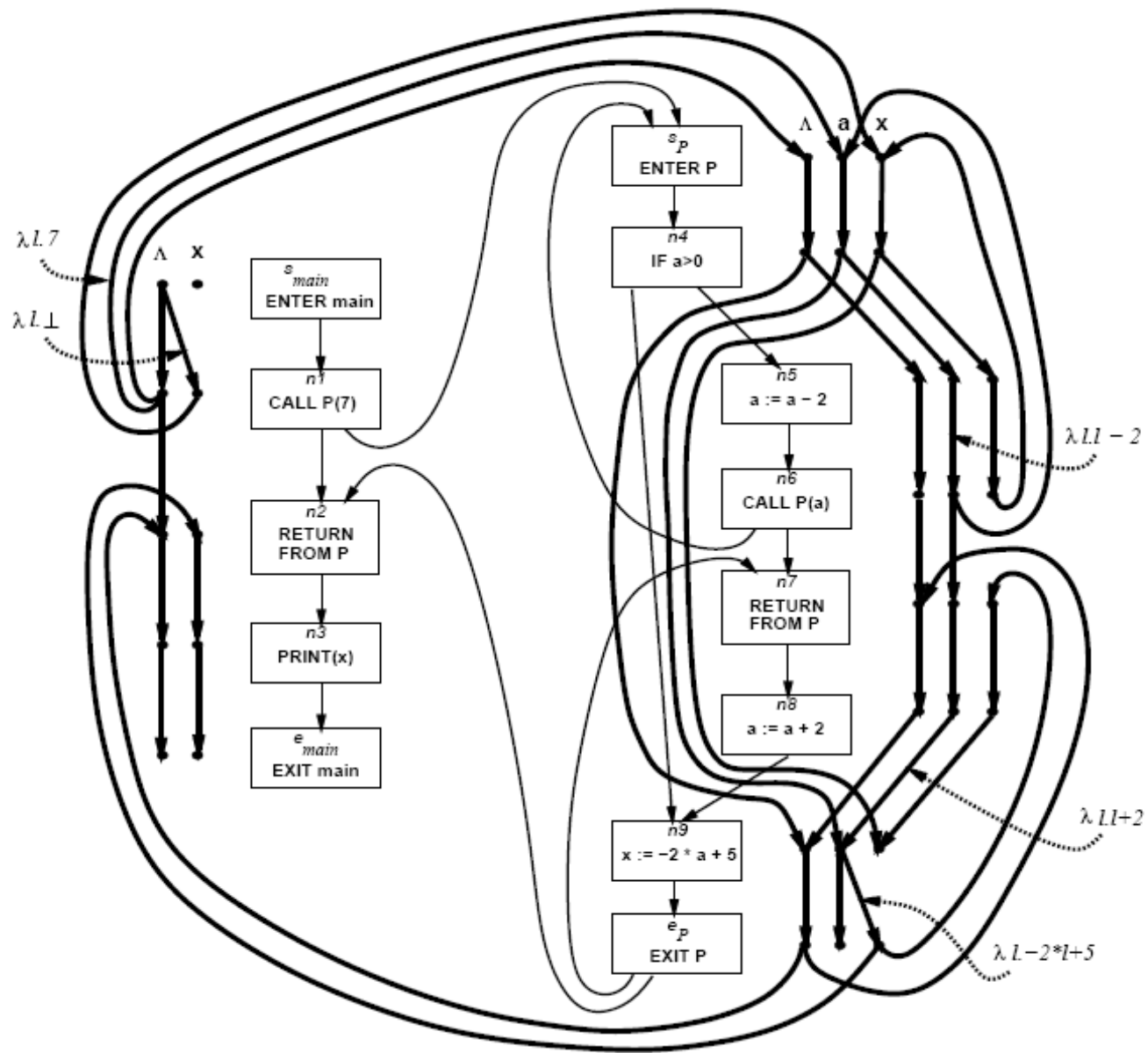
```

```

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

```





# Costs

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

# New Techniques

- Representing affine relations
  - $a_1 * x_1 + a_2 * x_2 + \dots + a_n * x_n$
- Backward summary computations

# 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