Interprocedural Analysis

Noam Rinetzky

Mooly Sagiv

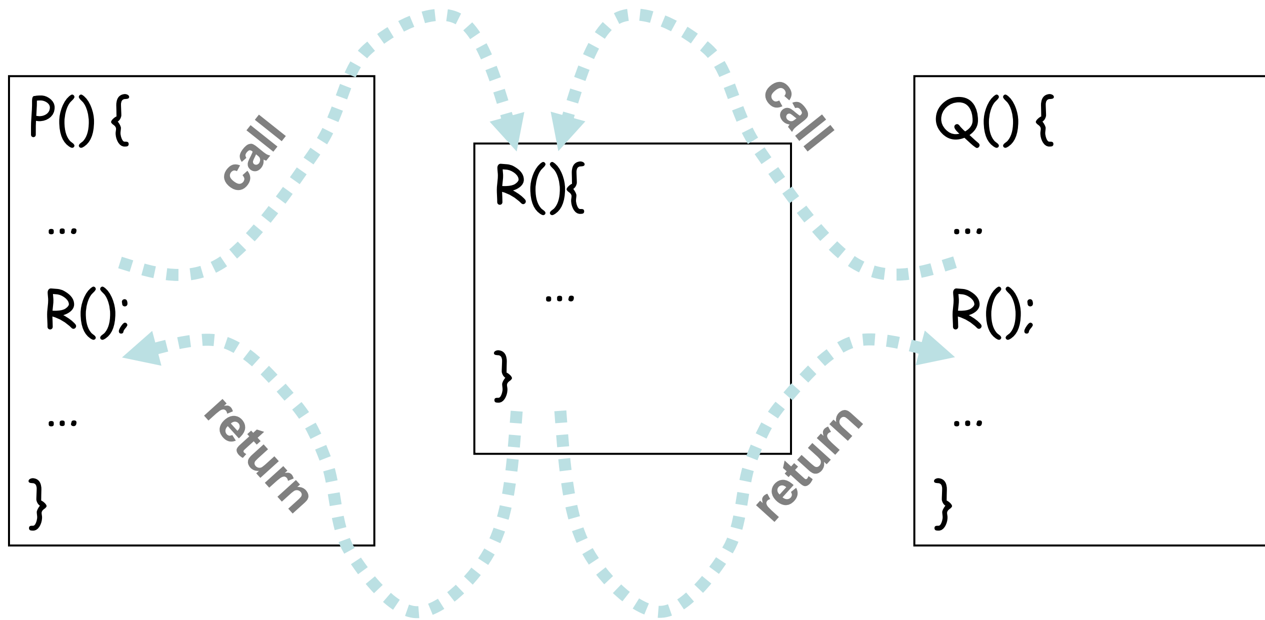
Bibliography

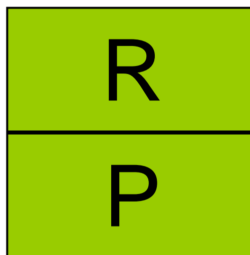
- **Textbook 2.5**
- Patrick Cousot & Radhia Cousot. Static determination of dynamic properties of recursive procedures In *IFIP Conference on Formal Description of Programming Concepts*, E.J. Neuhold, (Ed.), pages 237-277, St-Andrews, N.B., Canada, 1977. North-Holland Publishing Company (1978).
- **Two Approaches to interprocedural analysis by Micha Sharir and Amir Pnueli**
- 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*

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

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 + p^2(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 =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 ↦⊤]
    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 ↦⊤]
    x = p(9) ;  ⇦
     [x ↦⊤]

}
```

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


}
```

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, $$ ↦⊤]
    c2:[a ↦9, $$ ↦⊤]
}
```

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

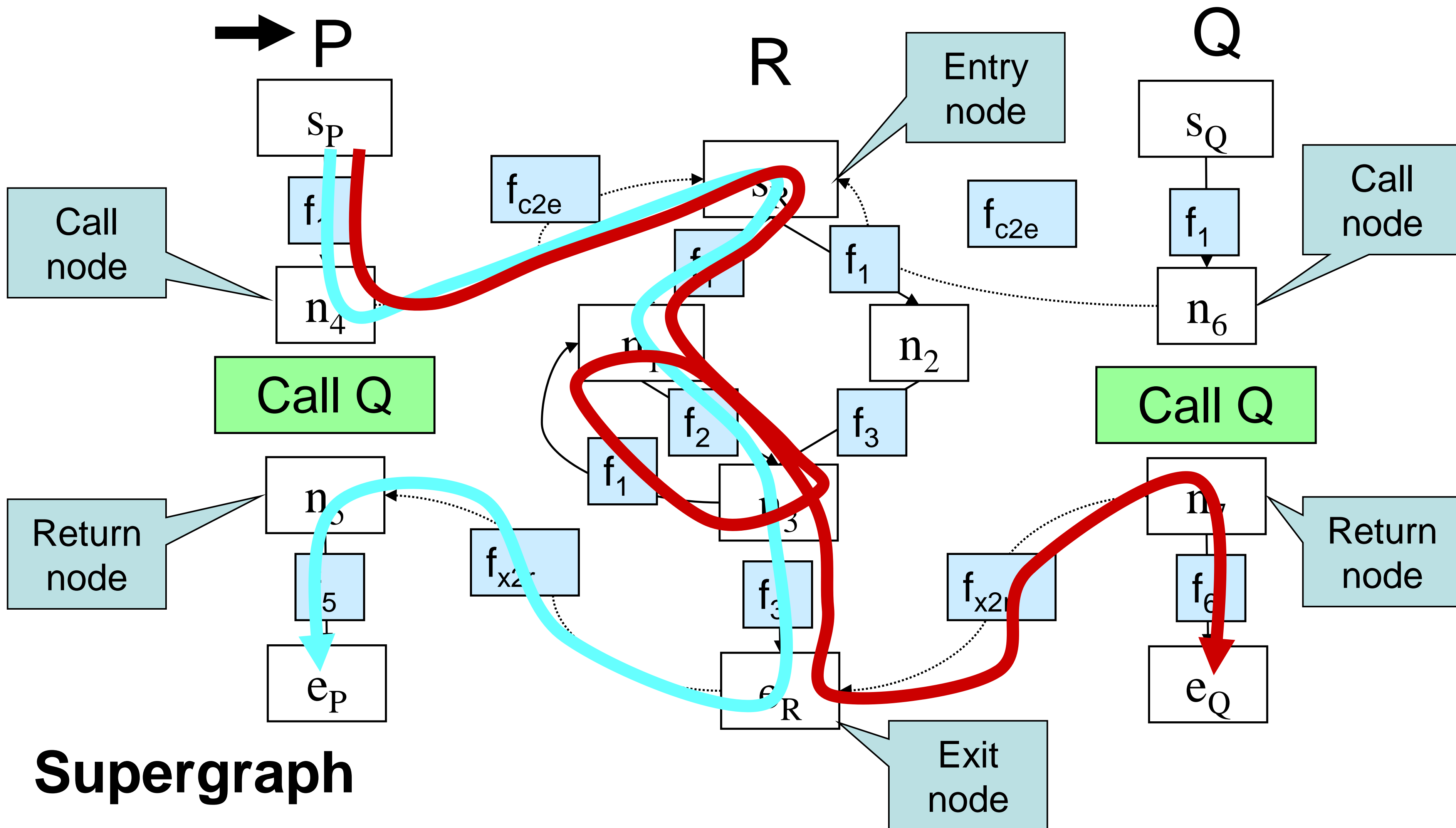
```
void main() {
    c1: p(7);
ε: [x ↦⊤]
}
```

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

Summary Call String

- Easy to implement
- Efficient for very small call strings
- Too expensive even for call strings of length 3
- For finite domains can be precise even with recursion
- Limited precision
- Order of calls can be abstracted
- Related method: procedure cloning

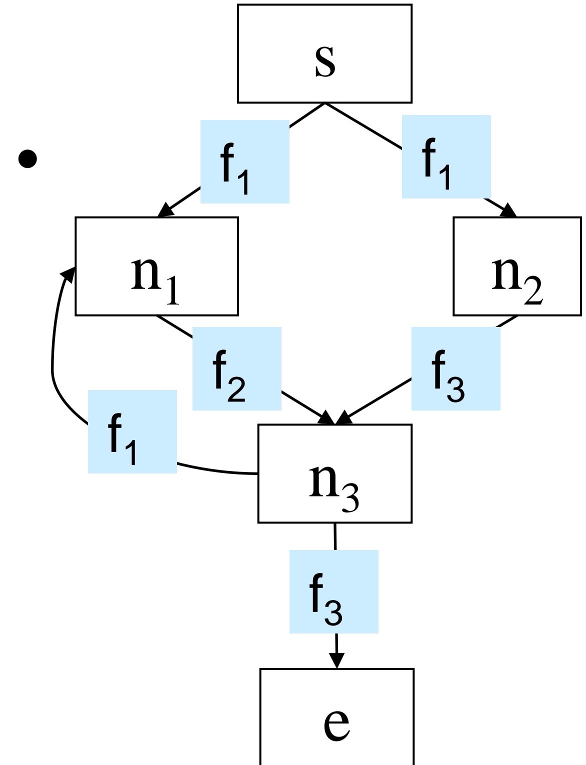
Interprocedural analysis



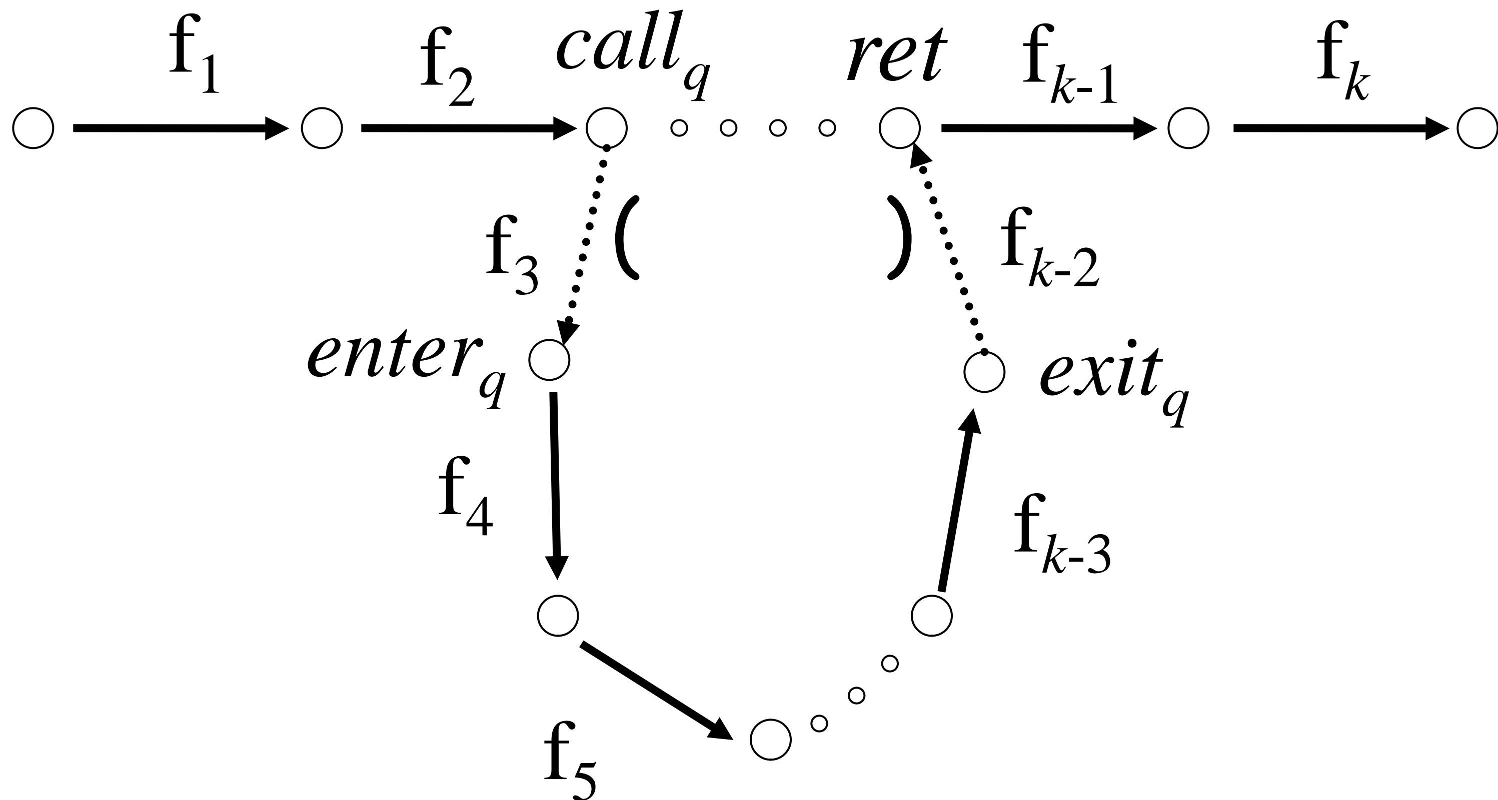
Supergraph

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

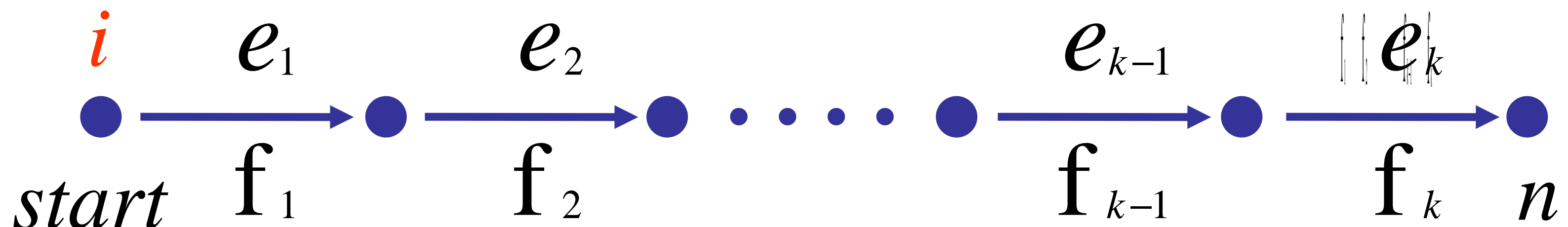


- IVP: all paths with matching calls and returns
  - And prefixes

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)



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

- JOP[v] $= \sqcup\{[[e_1, e_2, \ldots, e_n]](\iota) \mid (e_1, \ldots, e_n) \in \text{paths}(v)\}$
- 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] $= \sqcup\{[[e_1, e_2, \ldots, e]](\iota)$
  $(e_1, e_2, \ldots, e) \in \text{vpaths}(n)\}$
- JVP $\sqsubseteq$ JFP
  - In some cases the JVP can be computed
  - (Distributive problem)

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);
}
```

```
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]
}
```

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

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

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 ↦⊤]
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 ↦⊤]
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 ↦a_0, x ↦$-2a_0 + 5$]

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

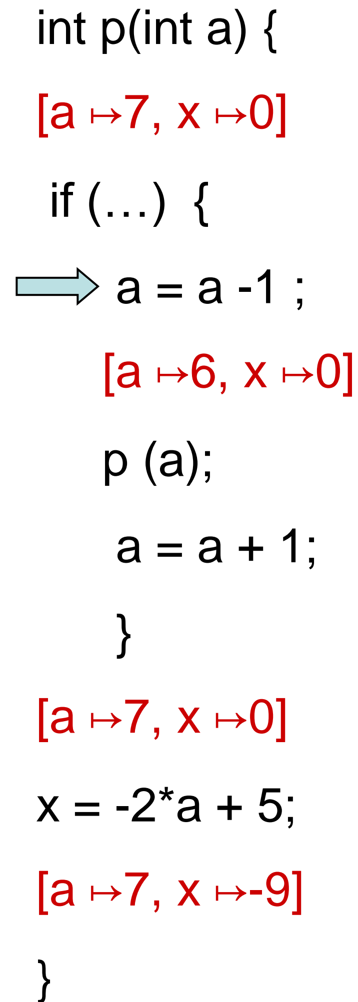
```
void main() {
    p(7);
[x ↦ -9]
}
```

```
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]
}
```



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

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

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]
 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]
}
```

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

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

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 ↦⊤, 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]
}
```

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

```
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 ↦ a_0, x ↦ $-2a_0 + 5$]

```
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 ↦ a_0, x ↦ $-2a_0 + 5$]

```
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 ↦ a_0, x ↦ $-2a_0 + 5$]

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

```
[a ↦T, x ↦T]
}
```

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

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

A Complicated Example

```
main() {
   a = 0;
   p();
   print a;
}
```

```
p() {
 if (…) {
       a= a + 2 + sgn(a -100);
       p();
       a – 1 ;
              }
}
```

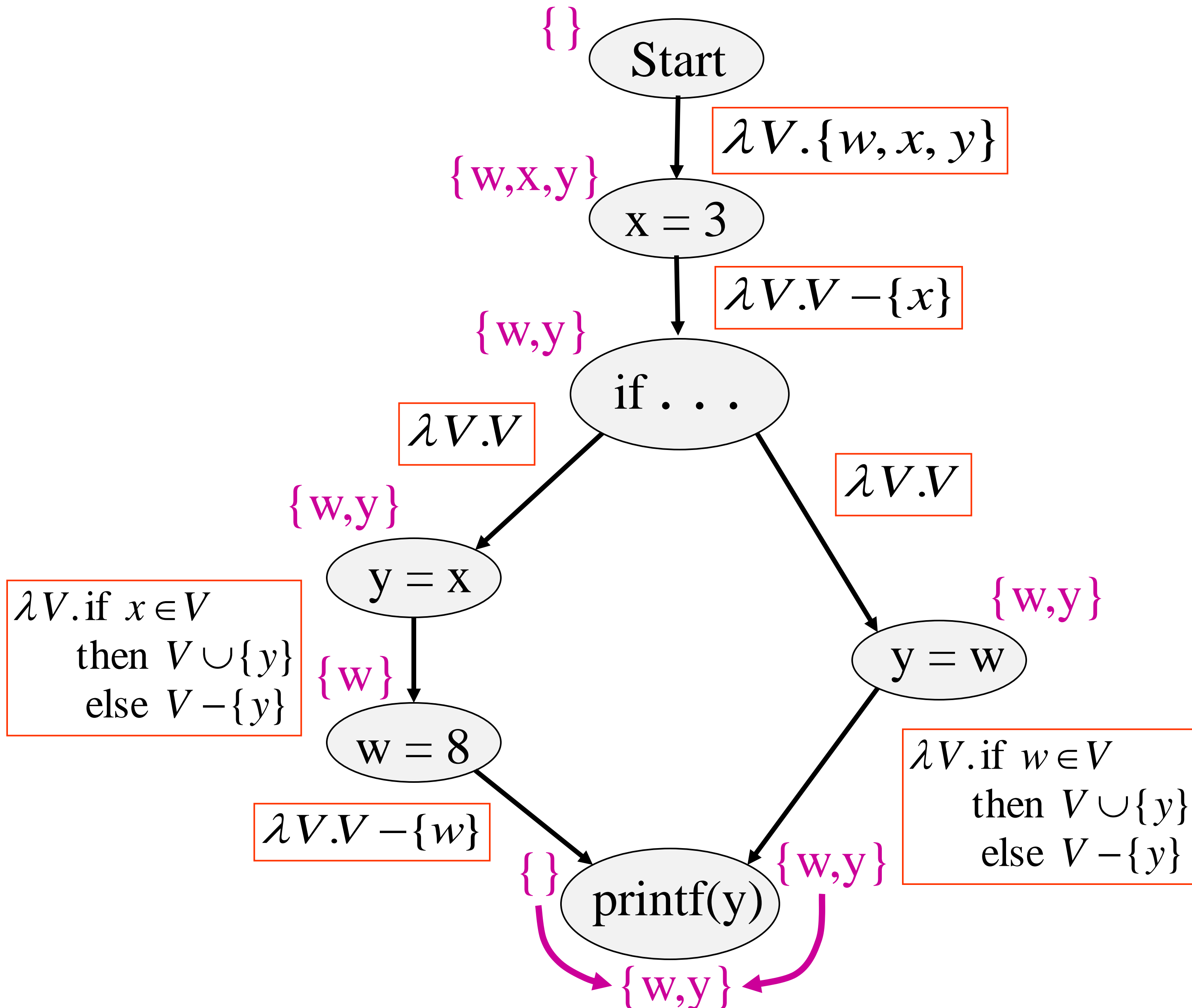
CFL-Graph reachability

- 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

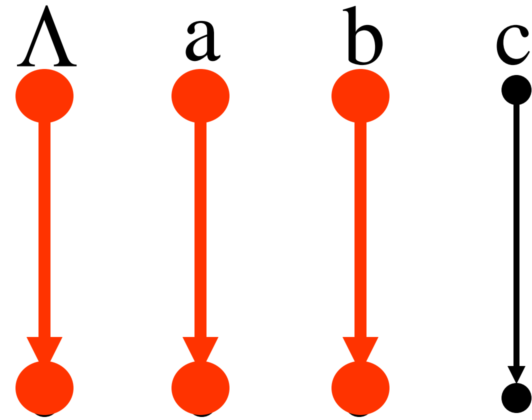
- Let $f: 2^D \rightarrow 2^D$ be a distributive function
- Then: $f(X) = f(\emptyset) \cup \cup\{z \in X: f(\{z\})\}$

Representing Dataflow Functions

Identity Function

$f = \lambda V.V$

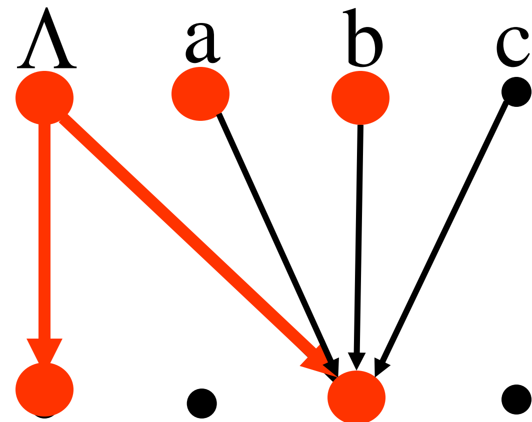
$f(\{a,b\}) = \{a,b\}$



Constant Function

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

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

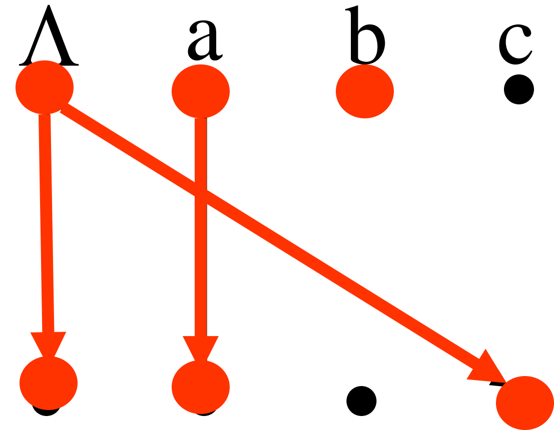


Representing Dataflow Functions

"Gen/Kill" Function

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

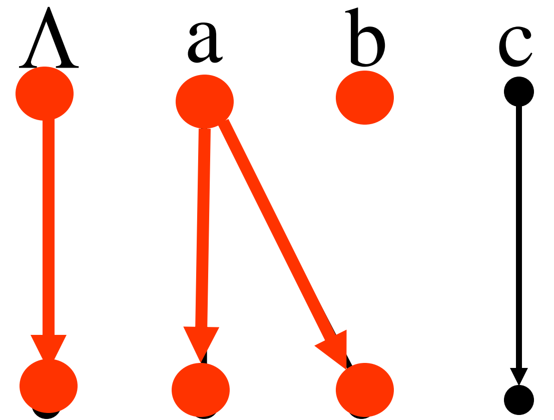
$$f(\{a,b\}) = \{a,c\}$$

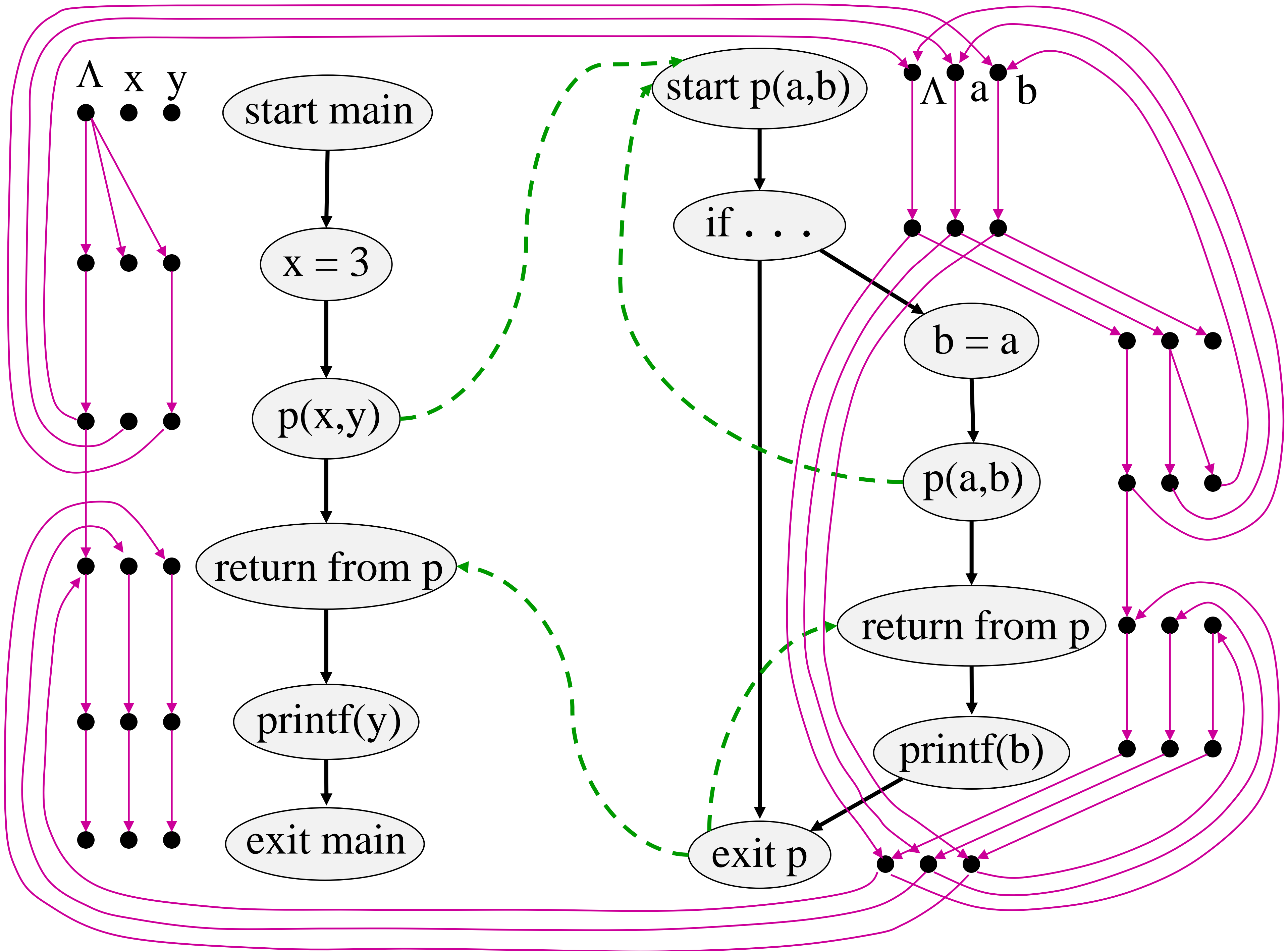


Non-"Gen/Kill" Function

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

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

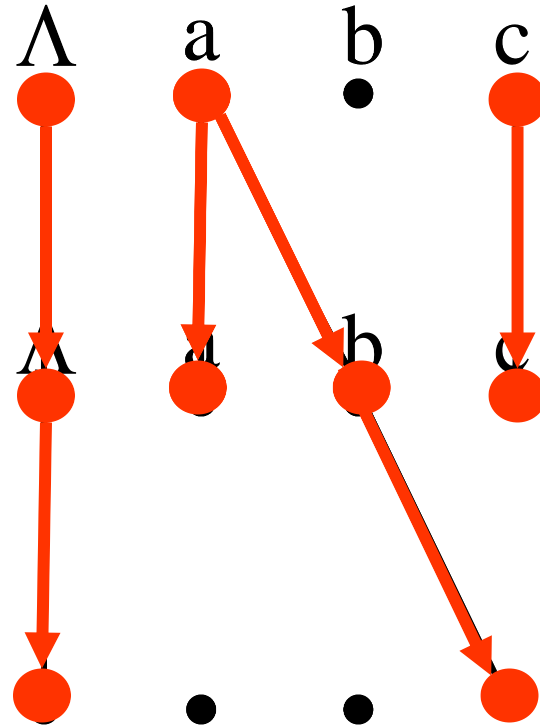



Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
Λ a b
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p

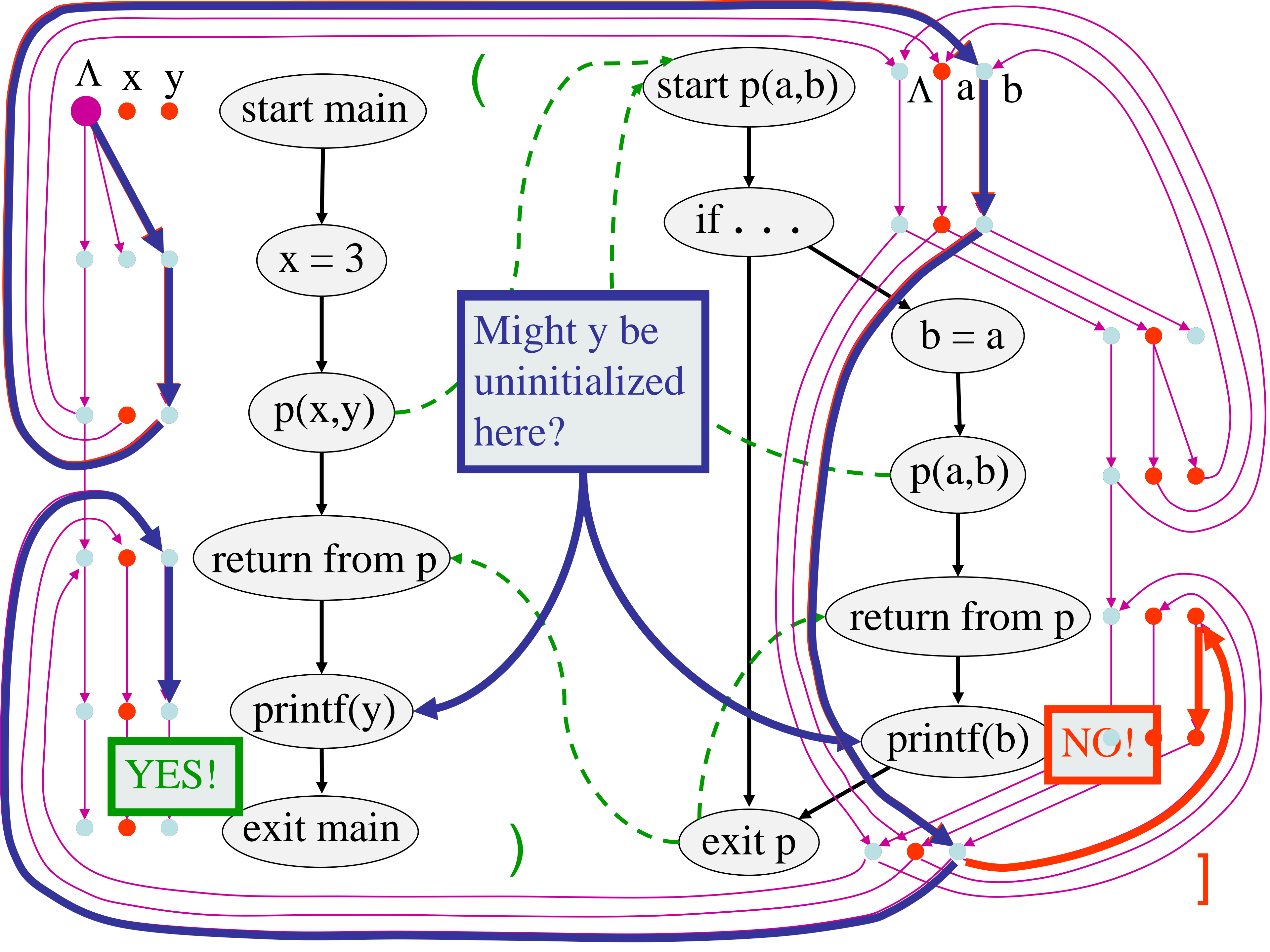
Composing Dataflow Functions

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

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



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

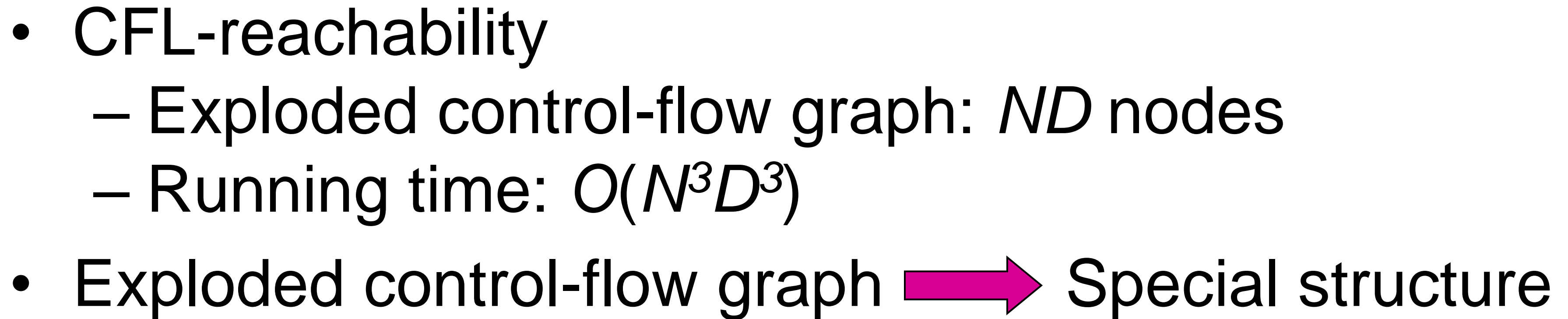


Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
YES!
(
)
Might y be uninitialized here?
start p(a,b)
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
Λ a b
NO!
]

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 $\langle start_{main}, \Lambda \rangle$ 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: $O(ED)$

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

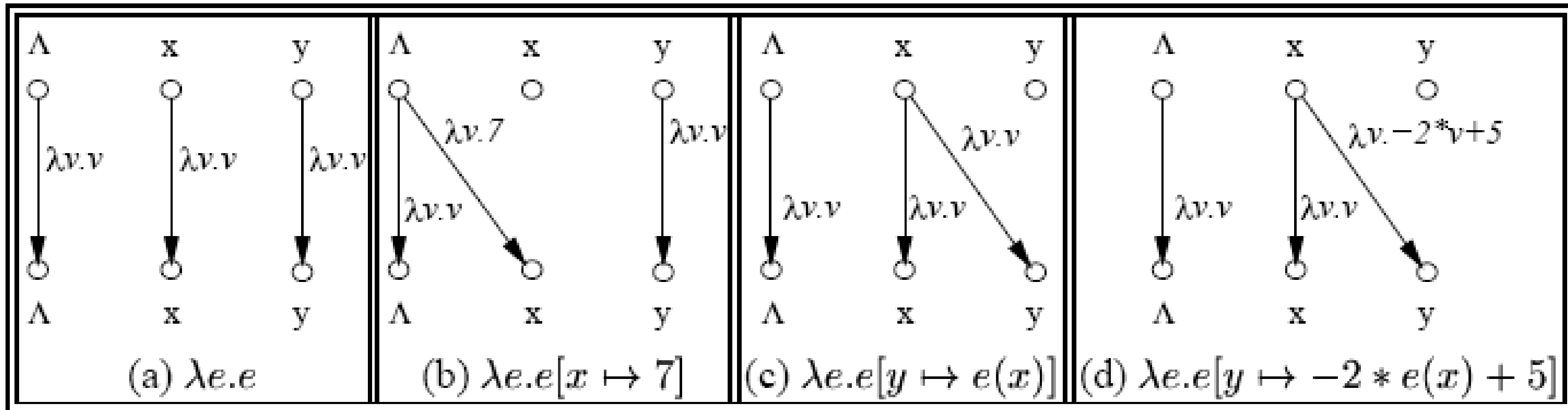
- Consider the constant propagation lattice
- The value of every variable y at the program exit can be represented by:

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

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

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

Linear constant propagation



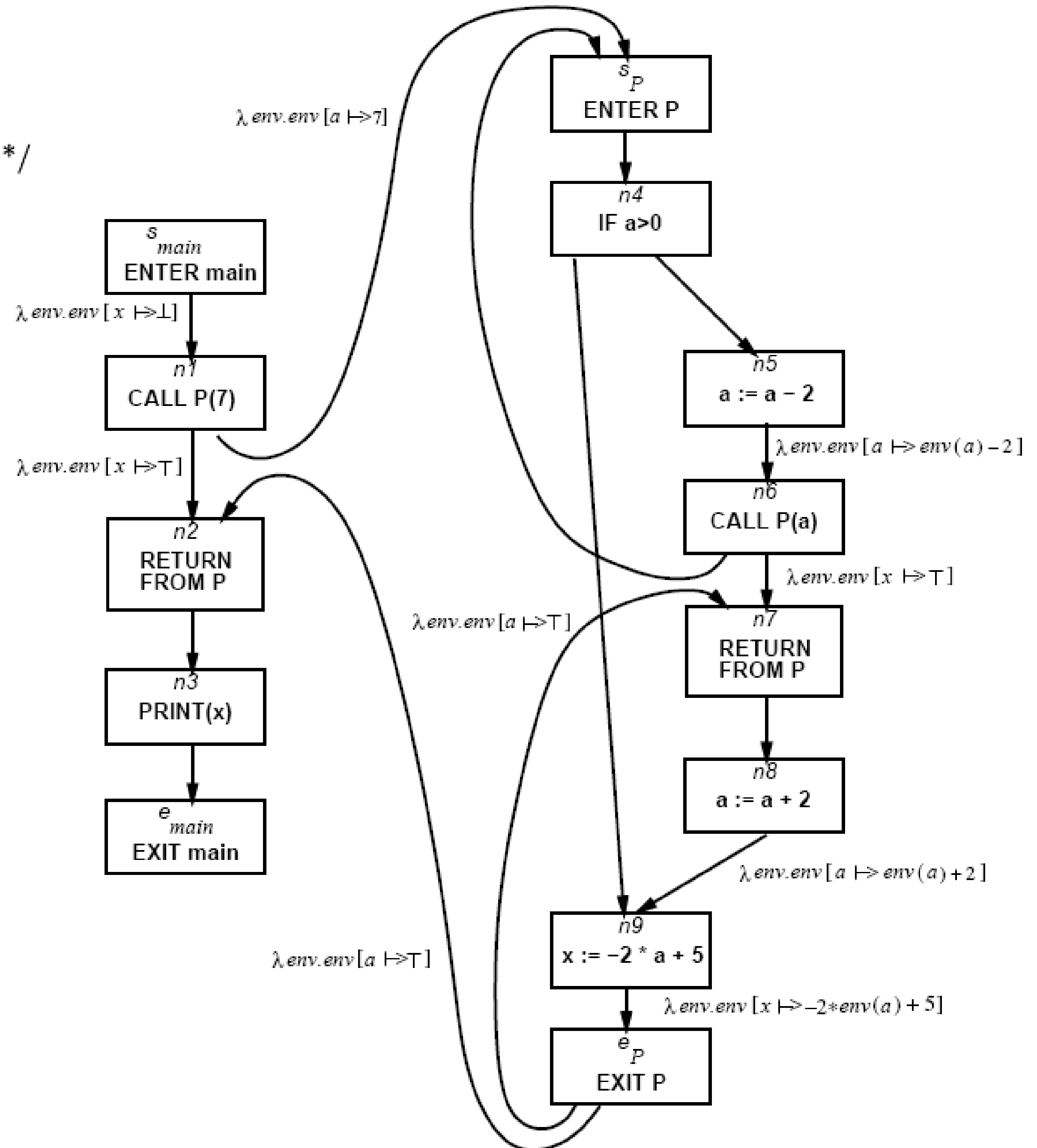
Point-wise representation of environment transformers

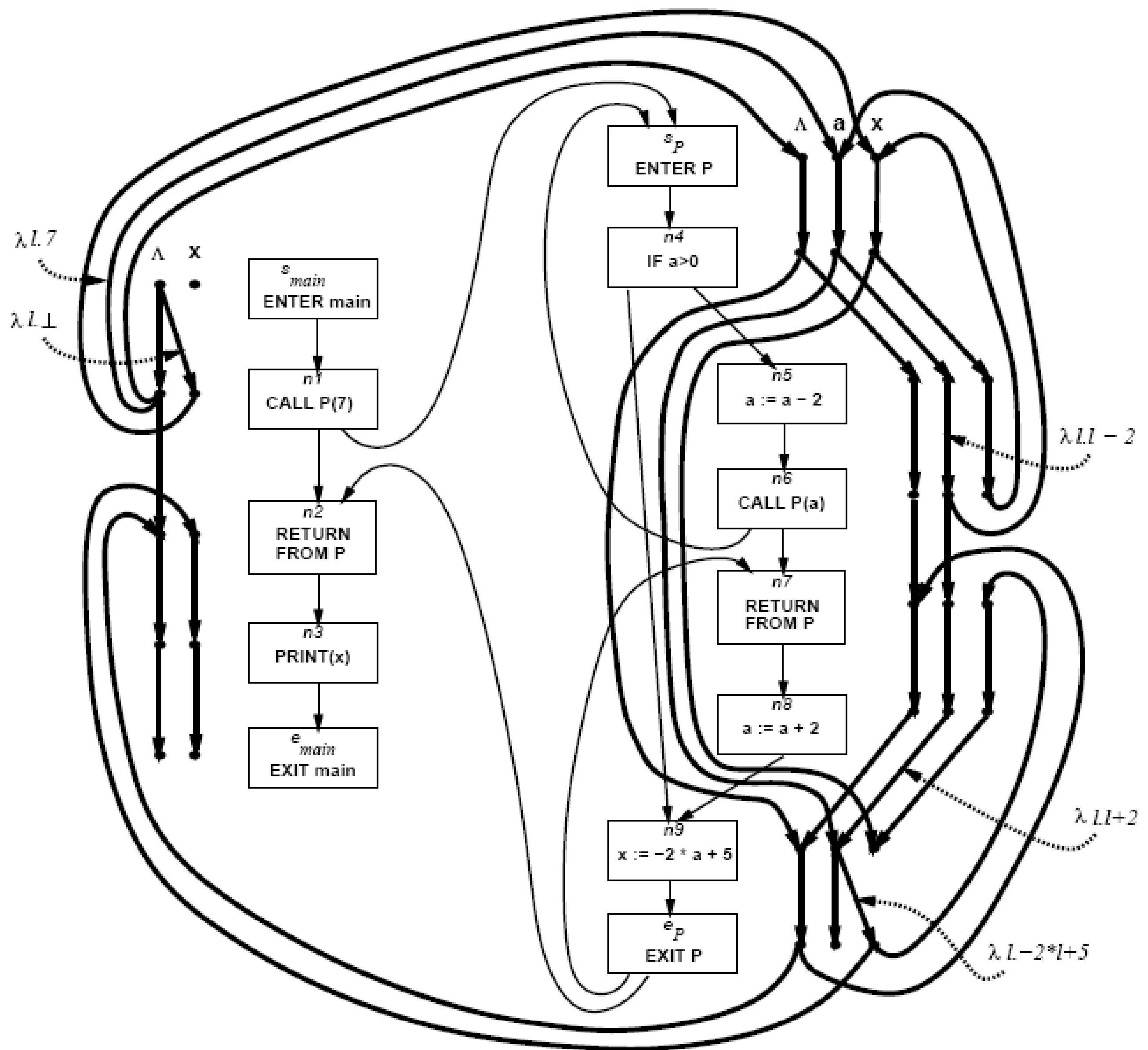
IDE Analysis

- 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
```





Λ
x
λ l. 7
λ l. ⊥
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

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