Program Analysis and Verification

0368-4479

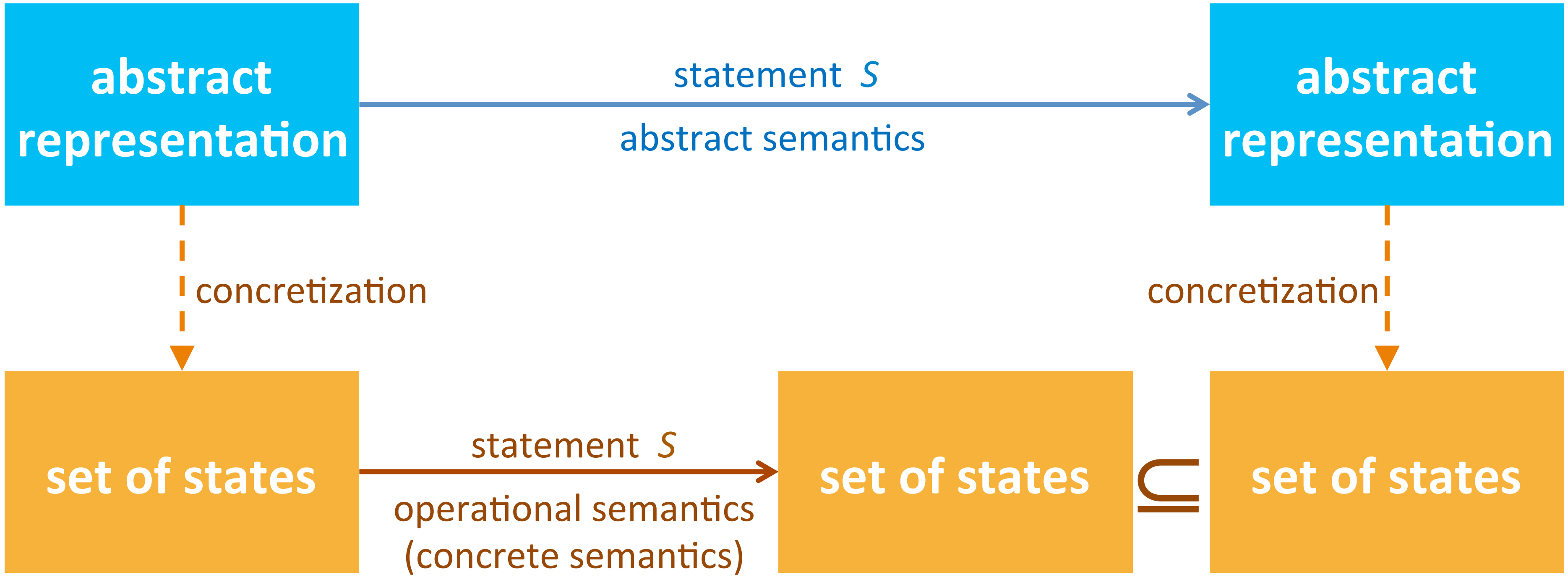
http://www.cs.tau.ac.il/~maon/teaching/2013-2014/paav/paav1314b.html

Noam Rinetzky

Lecture 14: Shape Domains & Interprocedural Analysis

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

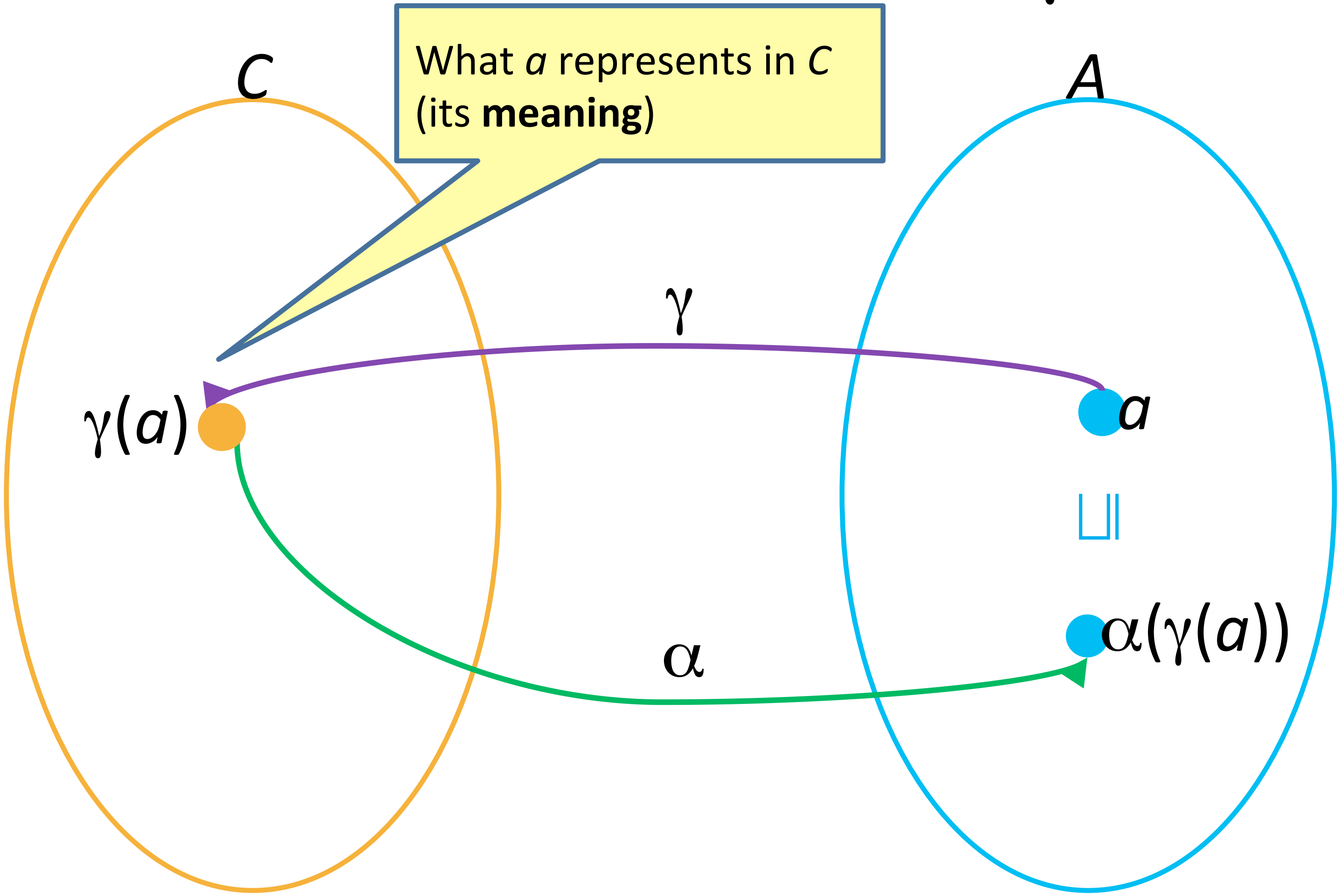
Abstract (conservative) interpretation



The collecting lattice

- Lattice for a given control-flow node *v*:
  $L_v = (2^{\textbf{State}}, \subseteq, \cup, \cap, \varnothing, \textbf{State})$
- Lattice for entire control-flow graph with nodes *V*:
  $$L_{CFG} = \text{Map}(V, L_v)$$
- We will use this lattice as a baseline for static analysis and define abstractions of its elements

Galois Connection: $\alpha(\gamma(a)) \sqsubseteq a$



Resulting algorithm

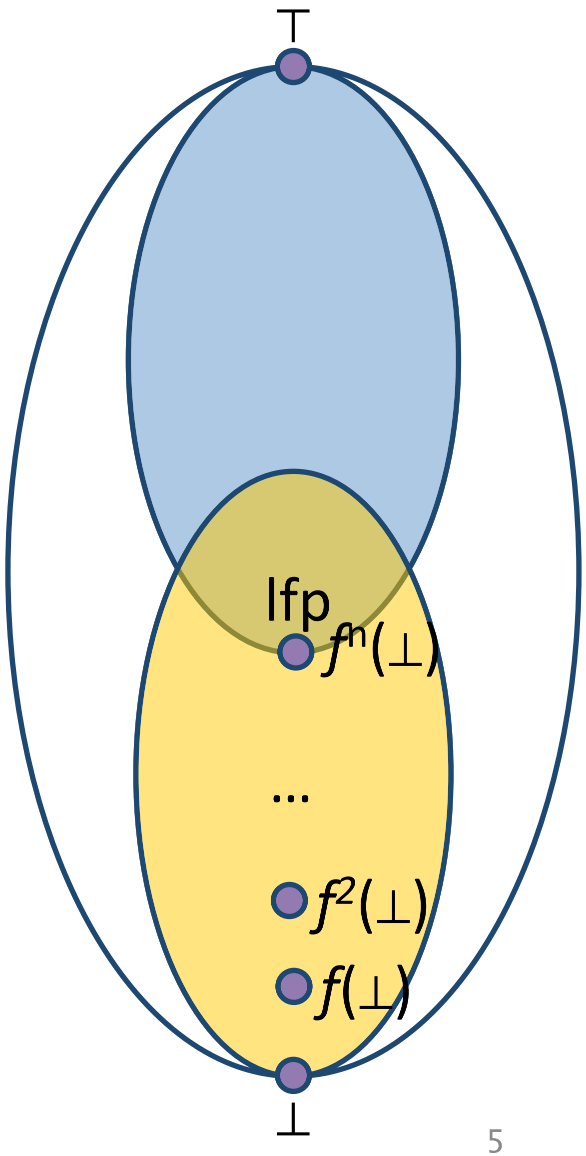
- Kleene's fixed point theorem gives a constructive method for computing the lfp

Mathematical definition

$$\text{lfp}(f) = \bigsqcup_{n \in N} f^n(\bot)$$

Algorithm

$d := \bot$
while $f(d) \neq d$ **do**
    $d := d \sqcup f(d)$
return d



Shape Analysis

Automatically verify properties of programs manipulating dynamically allocated storage

Identify all possible shapes (layout) of the heap

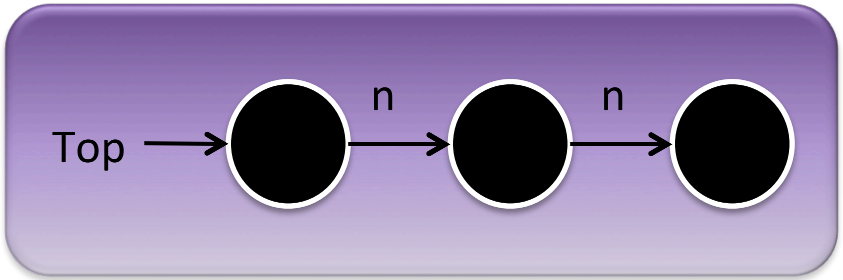
Shape Analysis via 3-valued Logic

1) Abstraction
- 3-valued logical structure
- canonical abstraction

2) Transformers
- via logical formulae
- soundness by construction
  - embedding theorem, [SRW02]

Concrete State

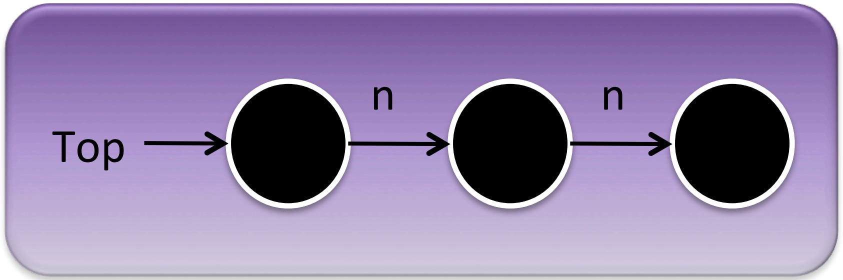
- represent a concrete state as a two-valued logical structure
  - Individuals = heap allocated objects
  - Unary predicates = object properties
  - Binary predicates = relations
- parametric vocabulary



(storeless, no heap addresses)

Concrete State

- S = <U, ι > over a vocabulary P
- U – universe
- ι - interpretation, mapping each predicate from p to its truth value in S



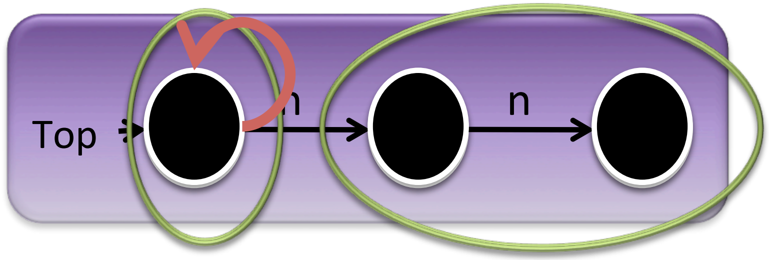
- U = { u1, u2, u3}
- P = { Top, n }
- ι(n)(u1,u2) = 1, ι(n)(u1,u3)=0, ι(n)(u2,u1)=0,…
- ι(Top)(u1)=1, ι(Top)(u2)=0, ι(Top)(u3)=0

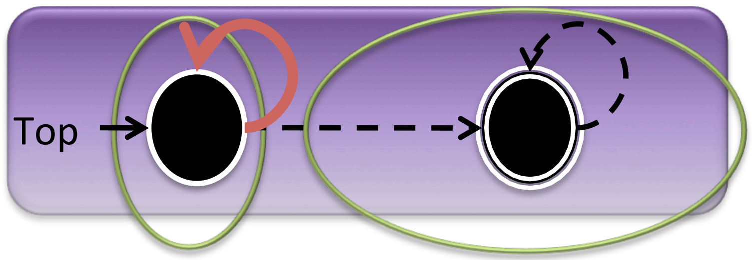
Collecting Semantics

- At every program point – a potentially infinite set of two-valued logical structures
- Representing (at least) all possible heaps that can arise at the program point

- Challenge:
  find a bounded abstract representation

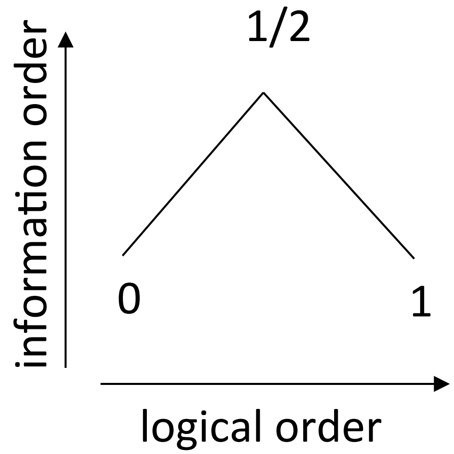
Canonical Abstraction: “Abstraction by Partinioning”





3-Valued Logic

- 1 = true
- 0 = false
- 1/2 = unknown



- A join semi-lattice, $0 \sqcup 1 = 1/2$

3-Valued Logical Structures

- A set of individuals (nodes) *U*
- Relation meaning
  - Interpretation of relations symbols in *P*

    $p^0() \rightarrow \{0,1, 1/2\}$

    $p^1(v) \rightarrow \{0,1, 1/2\}$

    $p^2(u,v) \rightarrow \{0,1, 1/2\}$
- A join semi-lattice: $0 \sqcup 1 = 1/2$

Canonical Abstraction (β)

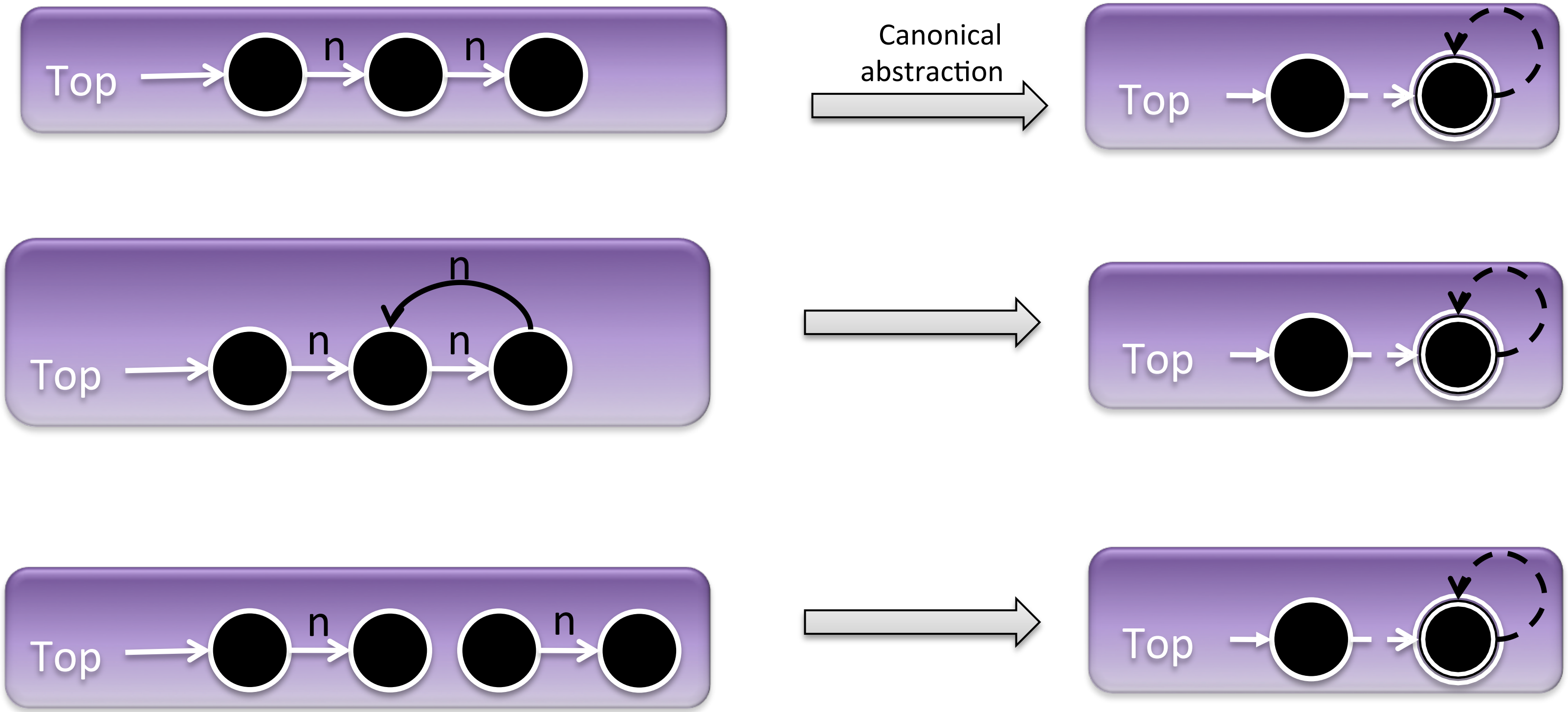
- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values

$$\iota'(u'_1, \ldots, u'_k) = \sqcup \{\iota(u_1, \ldots, u_k) \mid f(u_1)=u'_1, \ldots, f(u_k)=u'_k\}$$

- Converts a state of arbitrary size into a 3-valued abstract state of bounded size

- $\alpha(C) = \sqcup \{ \beta(c) \mid c \in C \}$

Information Loss



Shape Analysis via 3-valued Logic

1) Abstraction

- 3-valued logical structure
- canonical abstraction

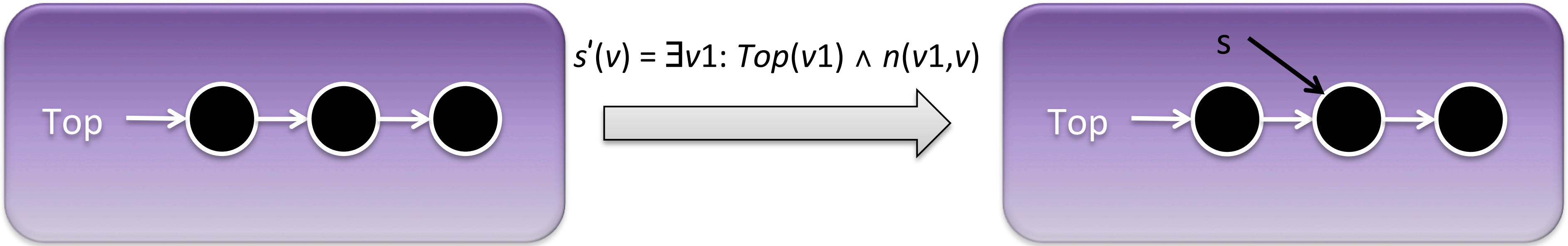
2) Transformers

- via logical formulae
- soundness by construction
  - embedding theorem, [SRW02]

Concrete Interpretation Rules

Statement	Update formula
x =NULL	$x'(v) = 0$
x= malloc()	$x'(v) = IsNew(v)$
x=y	$x'(v) = y(v)$
x=y →next	$x'(v) = \exists w: y(w) \wedge n(w, v)$
x →next=y	$n'(v, w) = (\neg x(v) \wedge n(v, w)) \vee (x(v) \wedge y(w))$

Example: $s = Top \rightarrow n$



Top	
u1	1
u2	0
u3	0

n	u1	u2	U3
u1	0	1	0
u2	0	0	1
u3	0	0	0

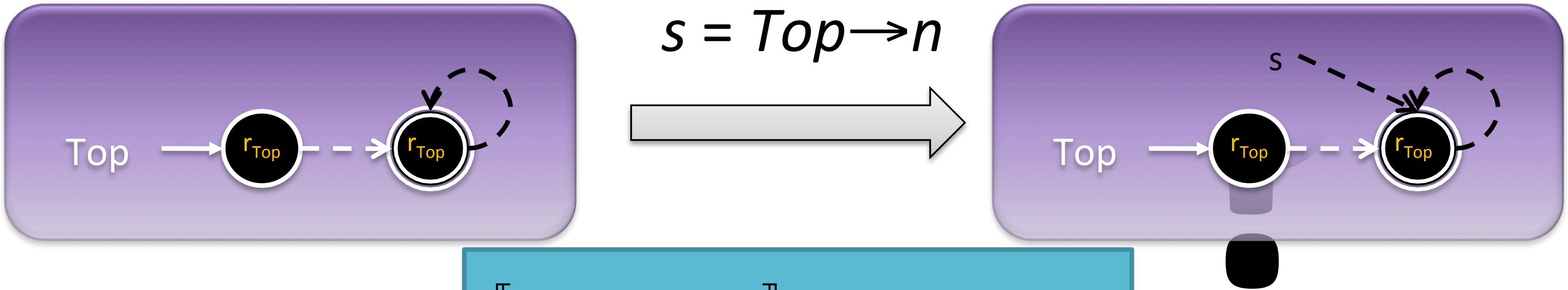
s	
u1	0
u2	0
u3	0

Top	
u1	1
u2	0
u3	0

n	u1	u2	U3
u1	0	1	0
u2	0	0	1
u3	0	0	0

s	
u1	0
u2	1
u3	0

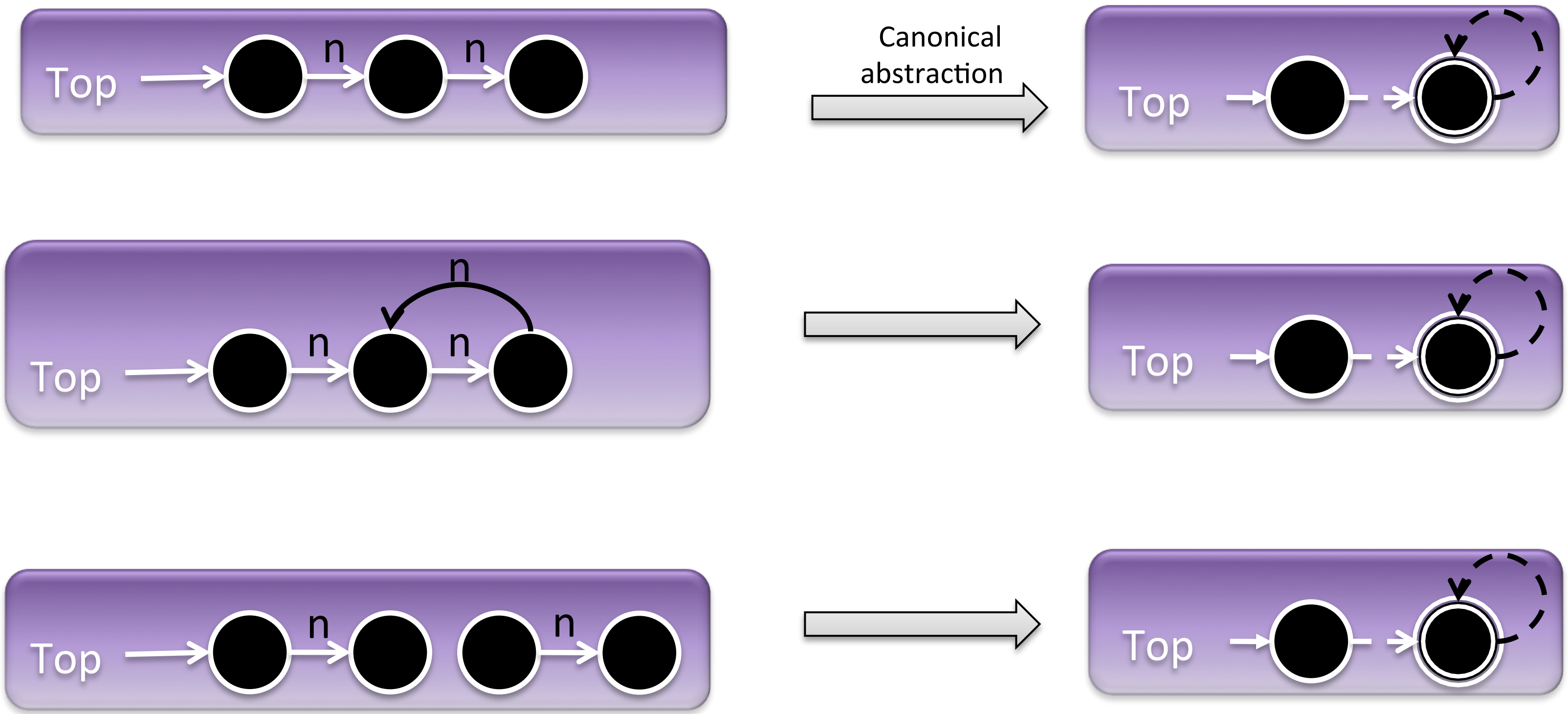
Abstract Semantics



⟦s = Top->n⟧

$$s'(v) = \exists v1 : Top(v1) \wedge n(v1,v)$$

Problem: Information Loss

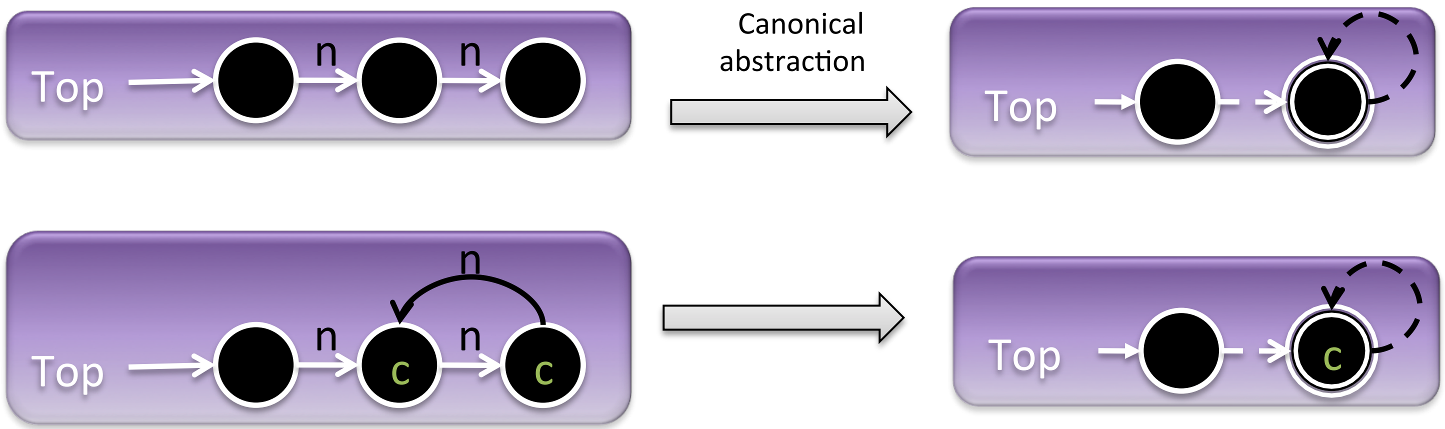


Instrumentation Predicates

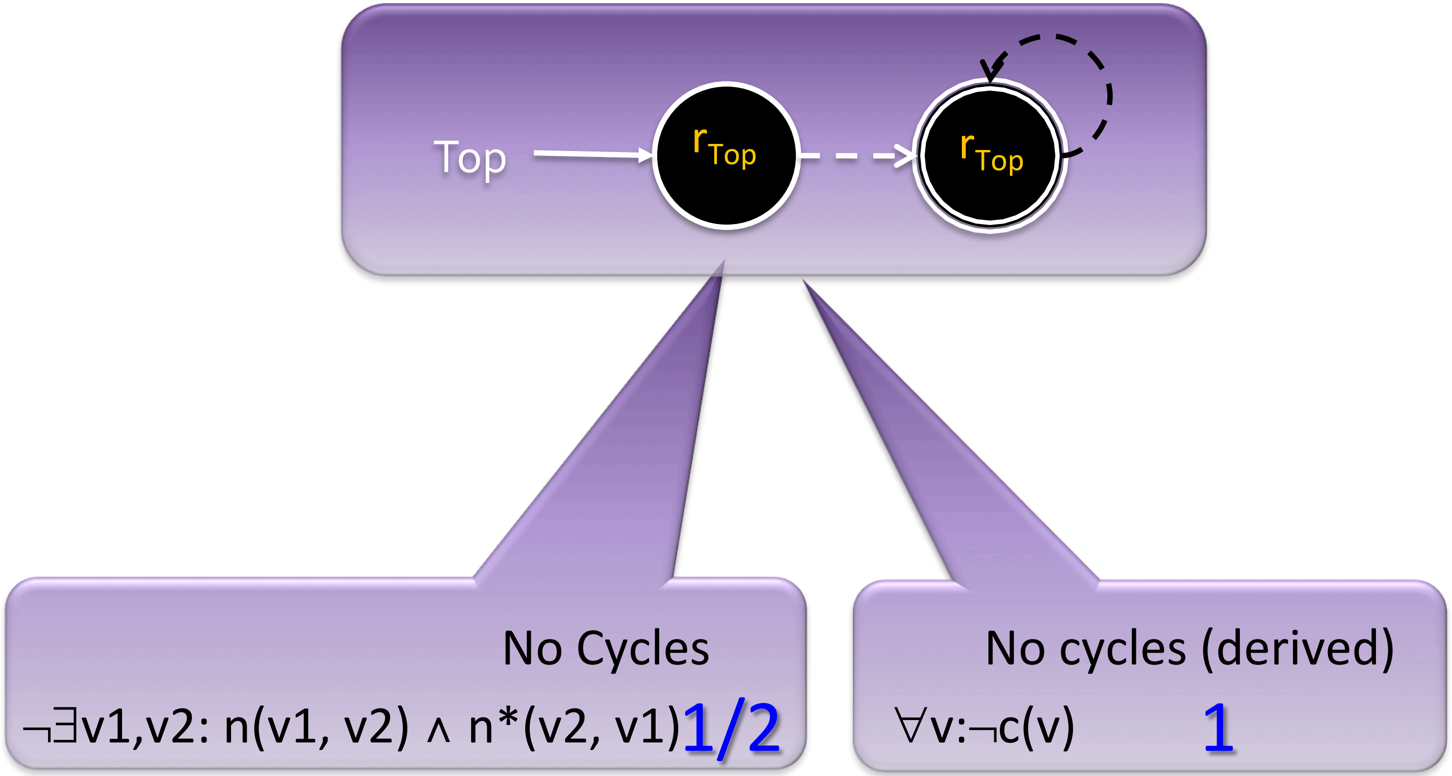
- Record additional derived information via predicates

$$r_x(v) = \exists v1: x(v1) \wedge n^*(v1,v)$$

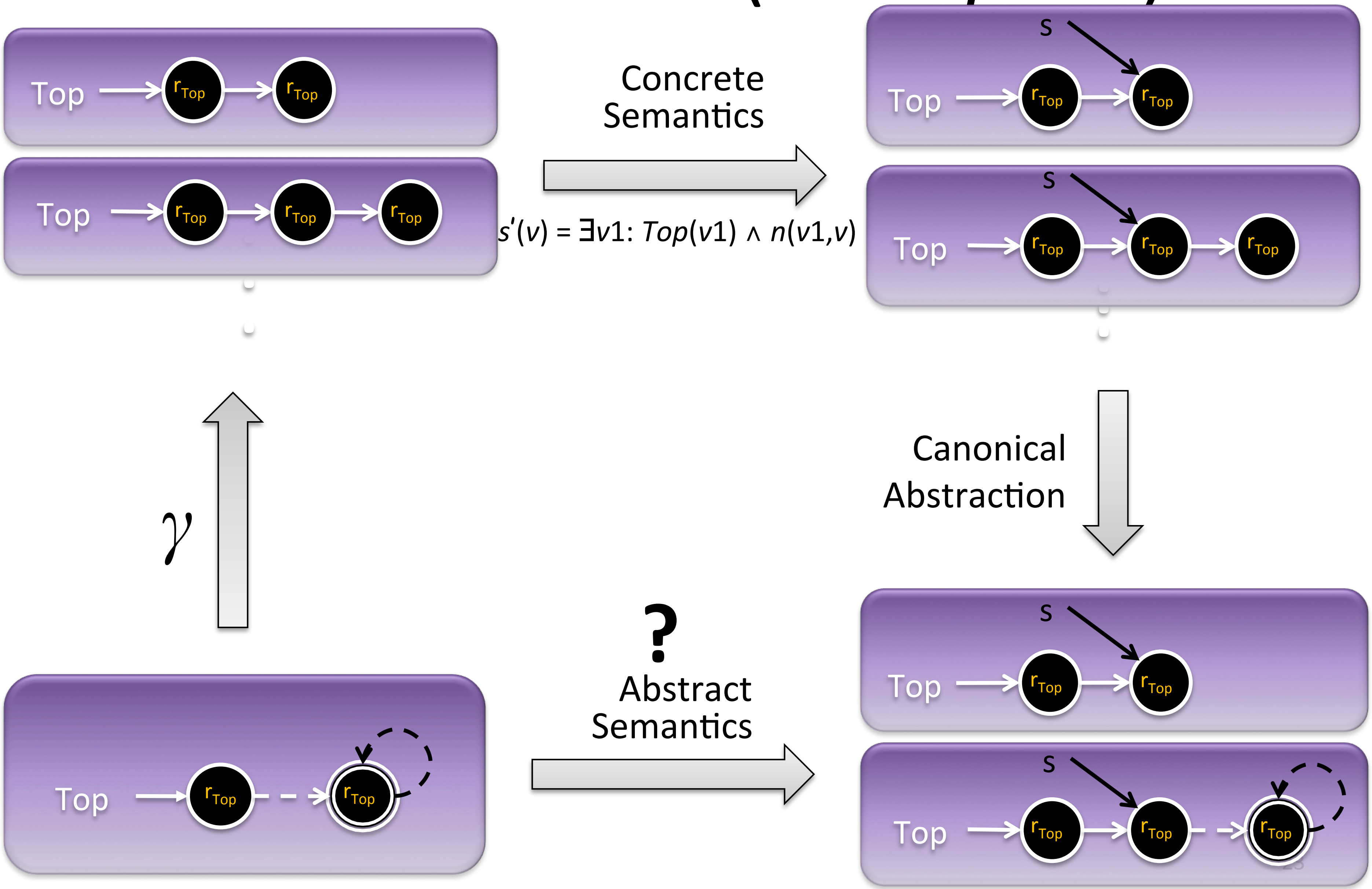
$$c(v) = \exists v1: n(v1, v) \wedge n^*(v, v1)$$



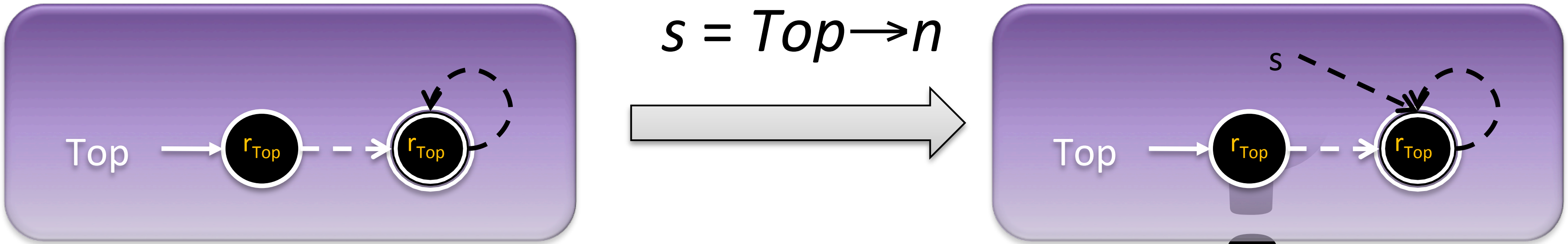
Embedding Theorem:
Conservatively Observing Properties



Best Transformer ($s = Top \rightarrow n$)



Abstract Semantics



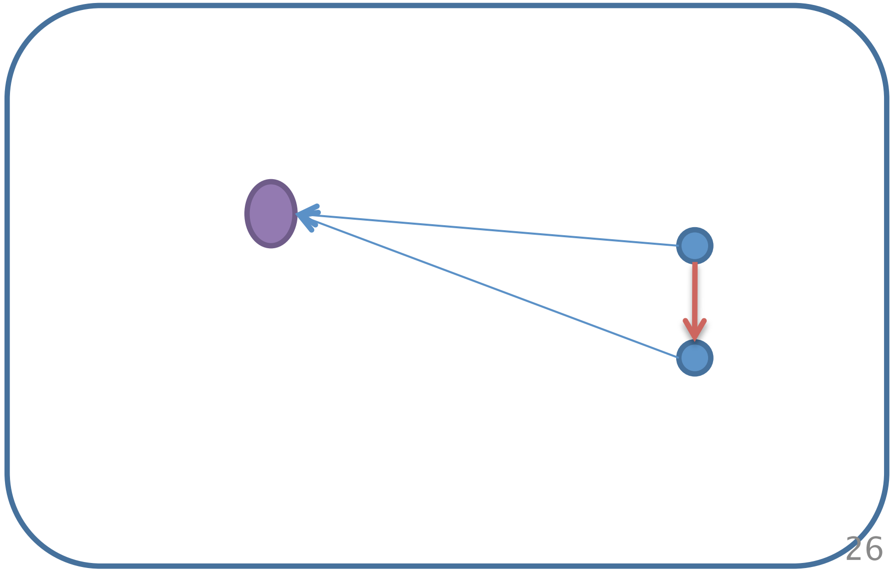
⟦s = Top->n⟧

$$s'(v) = \exists v1\text{: } Top(v1) \wedge n(v1,v)$$

Problem: Imprecise Transformers

Semantic Reduction

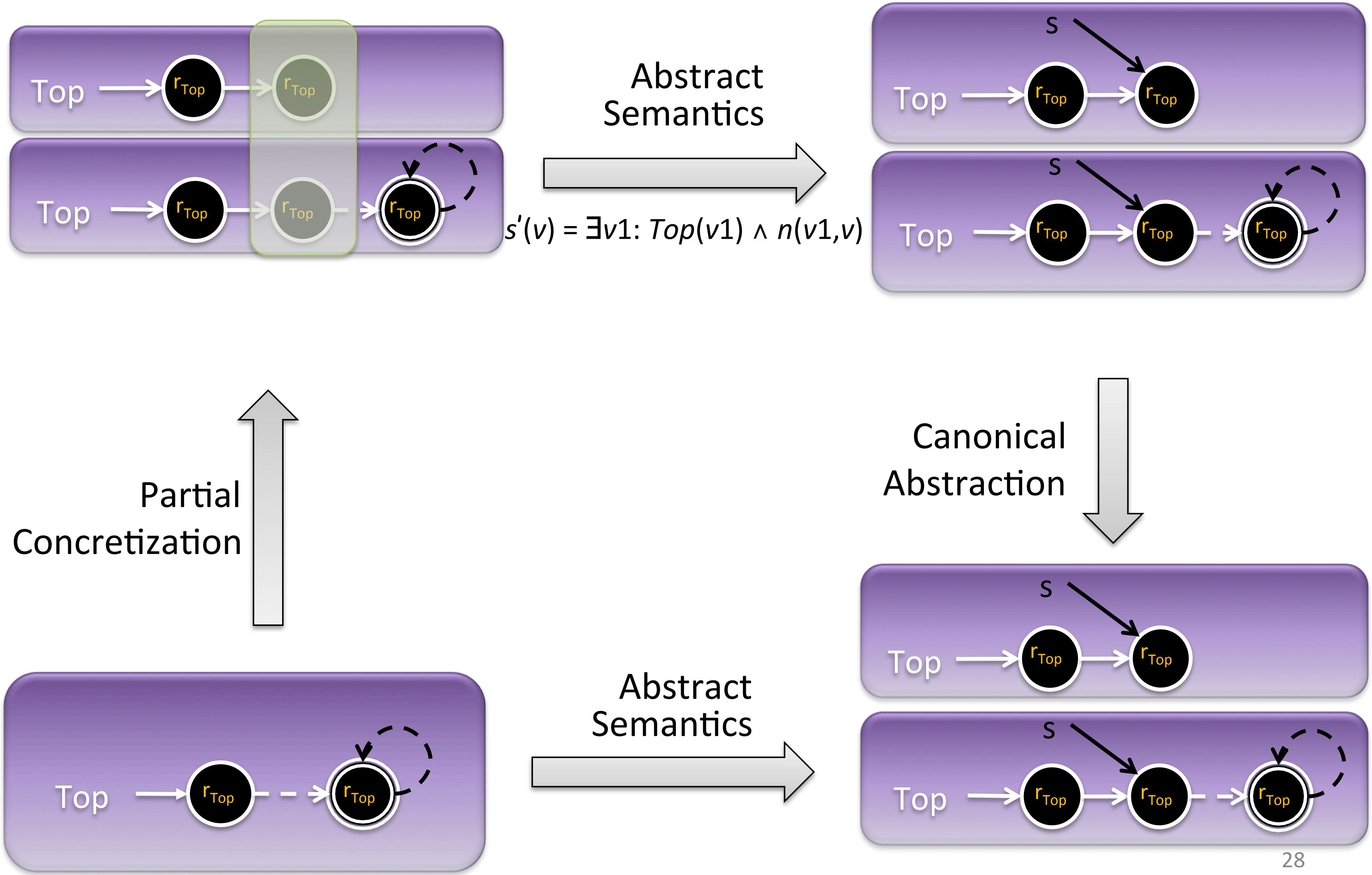
- Improve the precision of the analysis by recovering properties of the program semantics
- A Galois connection (C, α, γ, A)
- An operation $op: A \rightarrow A$ is a semantic reduction when
  - $\forall l \in L_2 \; op(l) \sqsubseteq l$ and
  - $\gamma(op(l)) = \gamma(l)$



The Focus Operation

- Focus: Formula→($\wp$(3-Struct) $\hookrightarrow$ $\wp$(3-Struct))
- Generalizes materialization
- For every formula φ
  - Focus(φ)(X) yields structure in which φ evaluates to a definite values in all assignments
  - Only maximal in terms of embedding
  - Focus(φ) is a semantic reduction
  - But Focus(φ)(X) may be undefined for some X

Partial Concretization Based on Transformer ($s = Top \rightarrow n$)



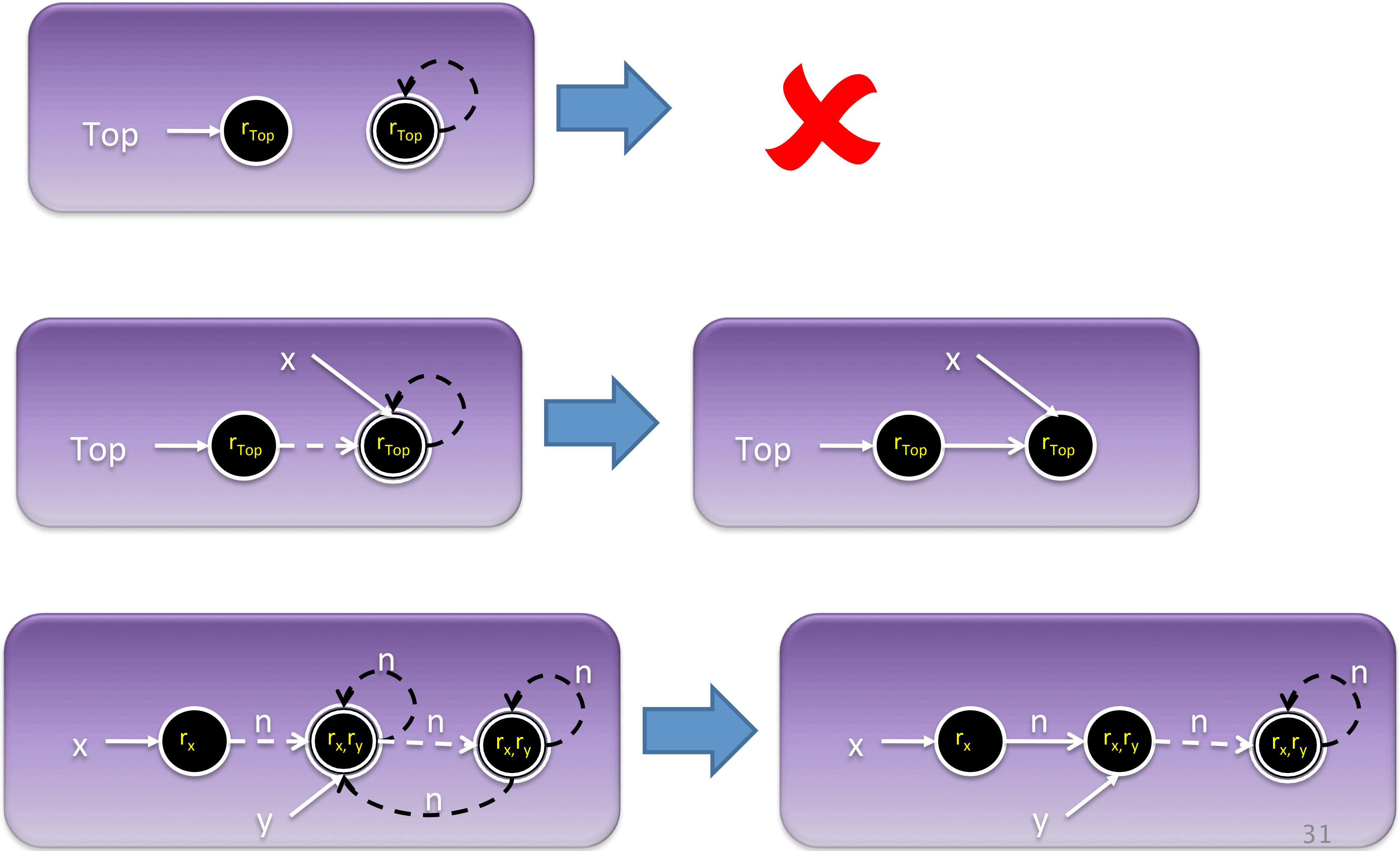
Partial Concretization

- Locally refine the abstract domain per statement
- Soundness is immediate
- Employed in other shape analysis algorithms
  [Distefano et.al., TACAS'06, Evan et.al., SAS'07, POPL'08]

The Coercion Principle

- Another Semantic Reduction
- Can be applied after Focus or after Update or both
- Increase precision by exploiting some structural properties possessed by all stores (Global invariants)
- Structural properties captured by constraints
- Apply a constraint solver

Apply Constraint Solver



Sources of Constraints

- Properties of the operational semantics
- Domain specific knowledge
  - Instrumentation predicates
- User supplied

Example Constraints

$x(v1) \wedge x(v2) \rightarrow eq(v1, v2)$

$n(v, v1) \wedge n(v,v2) \rightarrow eq(v1, v2)$

$n(v1, v) \wedge n(v2,v) \wedge \neg eq(v1, v2) \leftrightarrow is(v)$

$n^*(v3, v4) \leftrightarrow t[n](v1, v2)$

Abstract Transformers: Summary

- Kleene evaluation yields sound solution
- Focus is a statement-specific partial concretization
- Coerce applies global constraints

Abstract Semantics

$$
SS\ [v] = \begin{cases}
\{ \langle \emptyset,\emptyset \rangle \} & \text{if } v = \text{entry} \\[2ex]
\bigcup\limits_{\substack{(w,v) \in E(G), \\ w \in Assignments(G)}} \{ \text{t\_embed}(\text{coerce}([\![st(w)]\!]_3(\text{focus}_{F(w)}(SS[w]\ )))) \cup & \\[2ex]
\bigcup\limits_{\substack{(w,v) \in E(G), \\ w \in Skip(G)}} \{ S \mid S \in SS[w]\ \} \cup & \text{othrewise} \\[2ex]
\bigcup\limits_{(w,v) \in \text{True-Branches}(G)} \{ \text{t\_embed}(S) \mid S \in \text{coerce}([\![st(w)]\!]_3(\text{focus}_{F(w)}(SS[w]\ ))) \text{ and } S \vDash 3 \text{ cond}(w)\} \cup & \\[2ex]
\bigcup\limits_{(w,v) \in \text{False-Branches}(G)} \{ \text{t\_embed}(S) \mid S \in \text{coerce}([\![st(w)]\!]_3(\text{focus}_{F(w)}(SS[w]\ ))) \text{ and } S \vDash_3 \neg\text{cond}(w)\} \cup &
\end{cases}
$$

Recap

- Abstraction
  - canonical abstraction
  - recording derived information
- Transformers
  - partial concretization (focus)
  - constraint solver (coerce)
  - sound information extraction

Stack Push

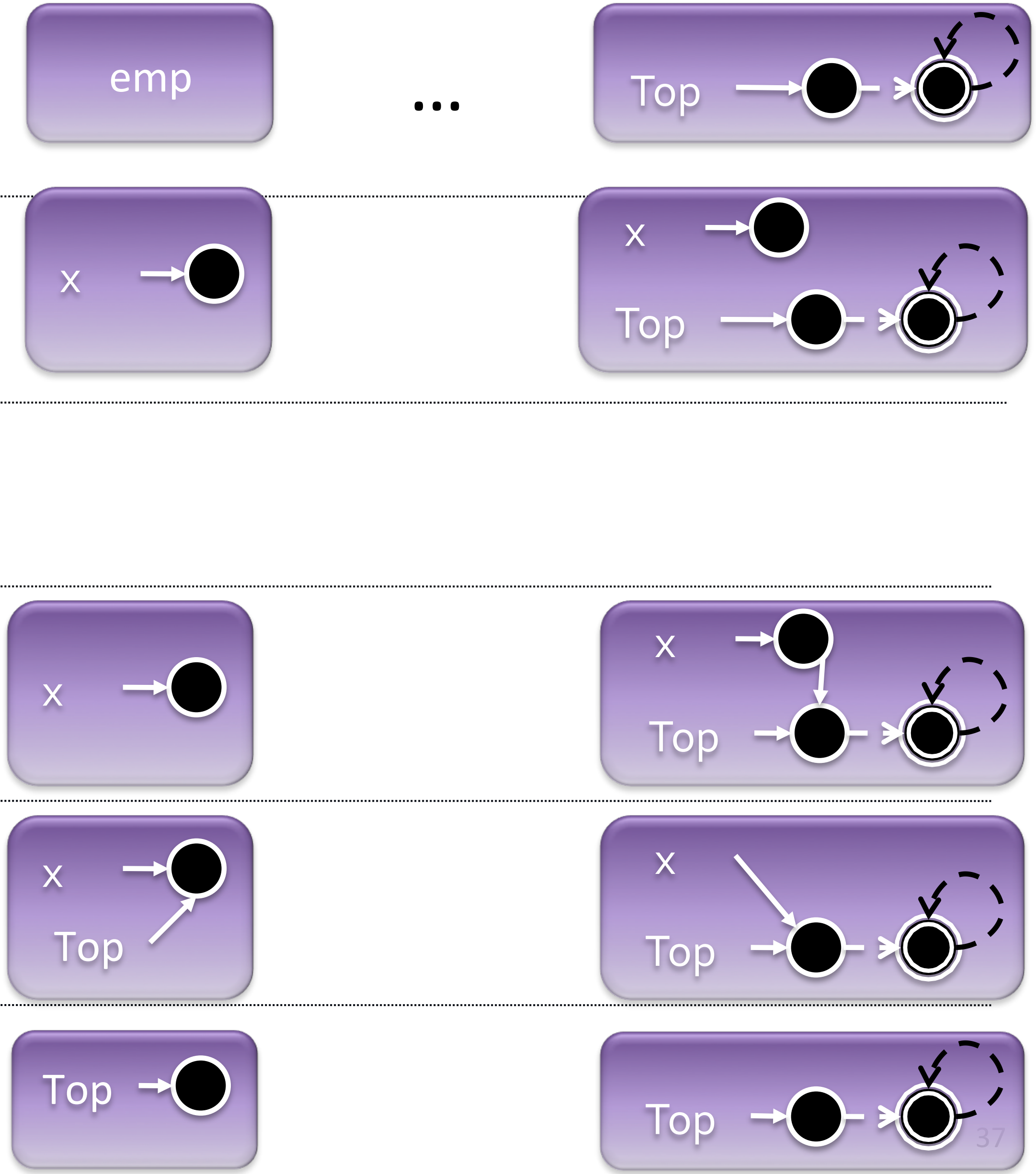
```
void push (int v) {
  Node *x =
    alloc(sizeof(Node));

  x→

  x→

  Top = x;
```

∃v: x(v)

∃v: x(v)

¬∃v1,v2: n(v1, v2) ∧Top(v2)

∀v:¬c(v)



Interprocedural Analysis

Today

- Analyzing programs with procedures
  - Precision (consider calling contexts)
  - Performance

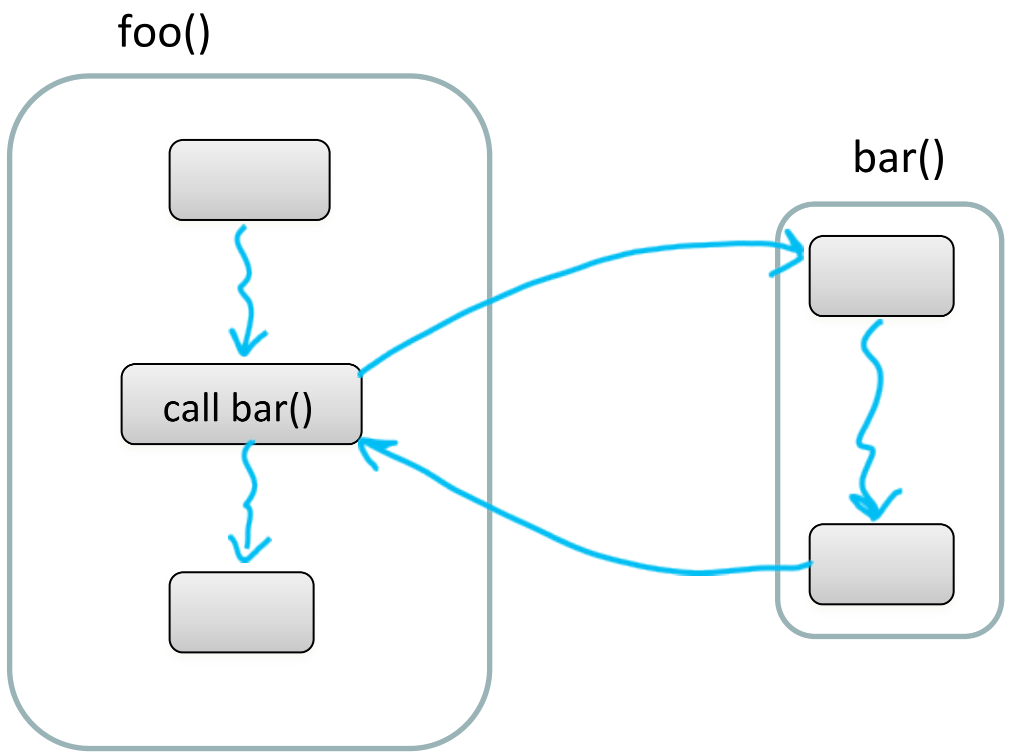
2

Procedural program

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

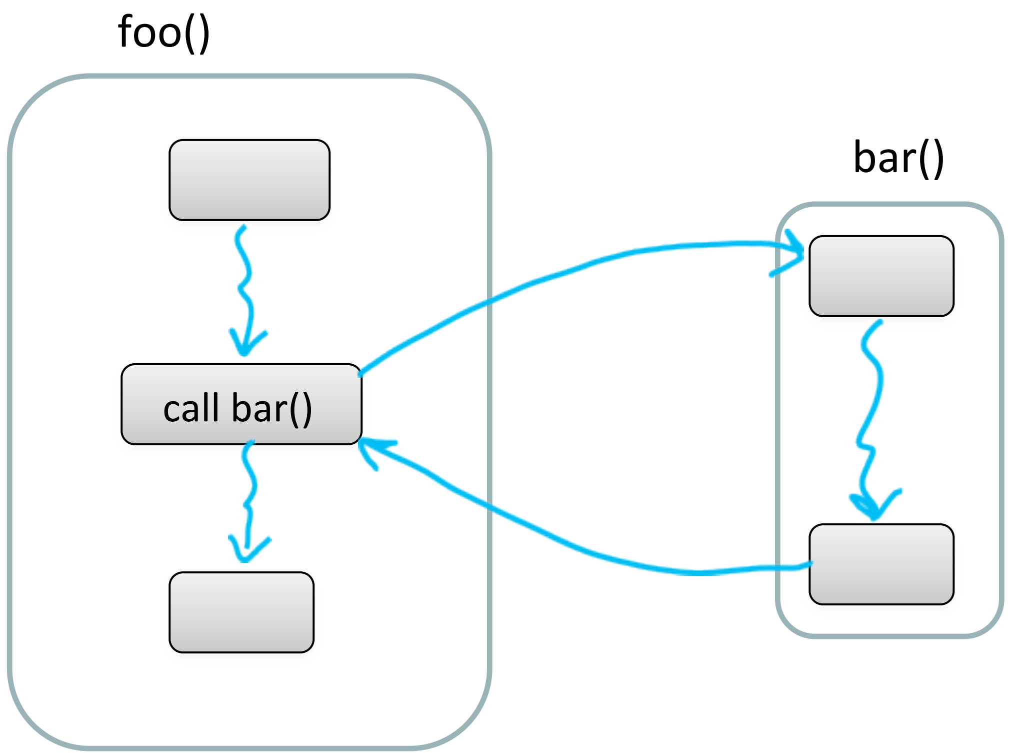
```
int p(int a) {
  return a + 1;
}
```

Effect of procedures



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

Interprocedural Analysis

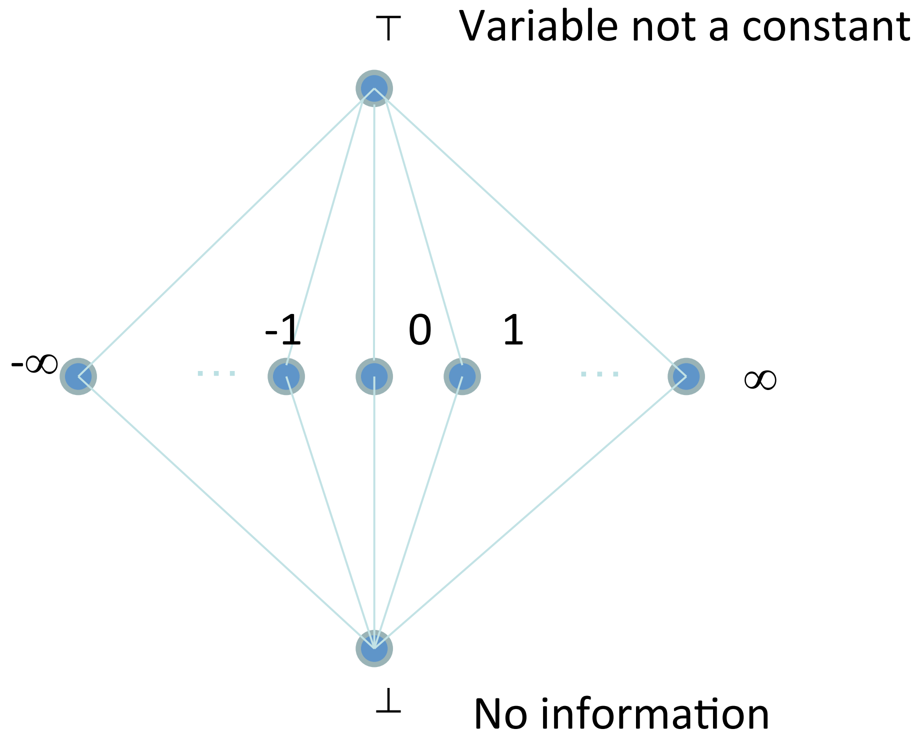


goal: compute the abstract effect of calling a procedure

Reduction to intraprocedural analysis

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

Reminder: Constant Propagation



$Z^{\top}_{\bot}$

Reminder: Constant Propagation

- L = (Var → $Z_{\bot}^{\top}$, $\sqsubseteq$)
- $\sigma_1 \sqsubseteq \sigma_2$ iff $\forall$x: σ_1(x) $\sqsubseteq'$ σ_2(x)
  - $\sqsubseteq'$ ordering in the $Z_{\bot}^{\top}$ lattice

- Examples:
  - [x ↦ ⊥, y ↦ 42, z ↦ ⊥] $\sqsubseteq$ [x ↦ ⊥, y ↦ 42, z ↦ 73]
  - [x ↦ ⊥, y ↦ 42, z ↦ 73] $\sqsubseteq$ [x ↦ ⊥, y ↦ 42, z ↦ ⊤]

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{
}
```

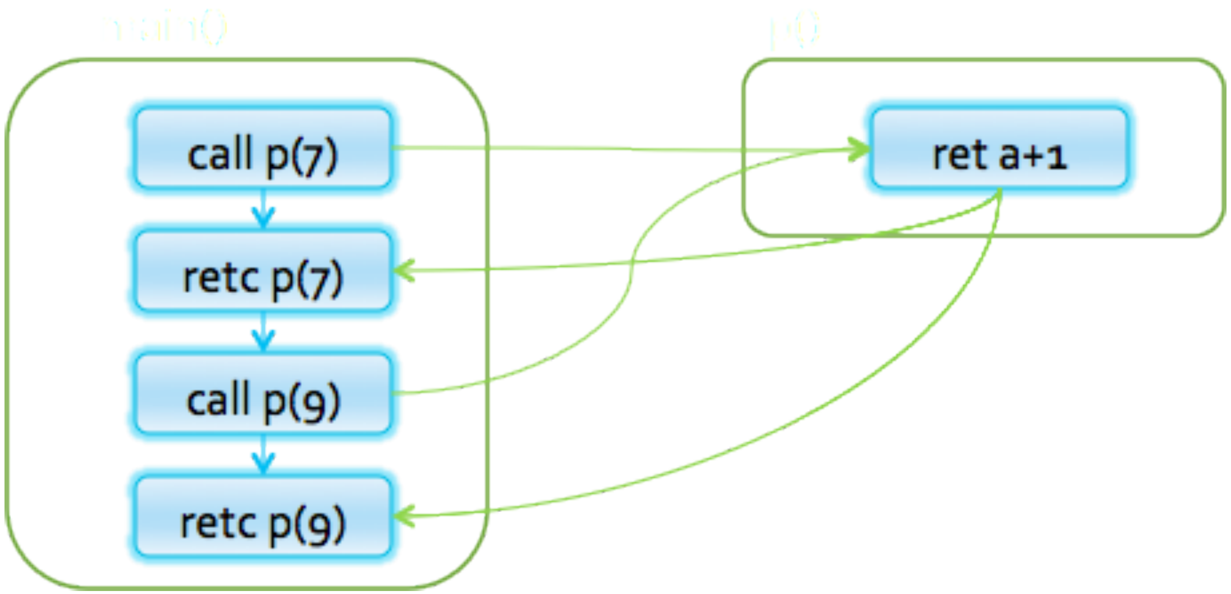
A Naive Interprocedural solution

- Treat procedure calls as gotos

Simple Example

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

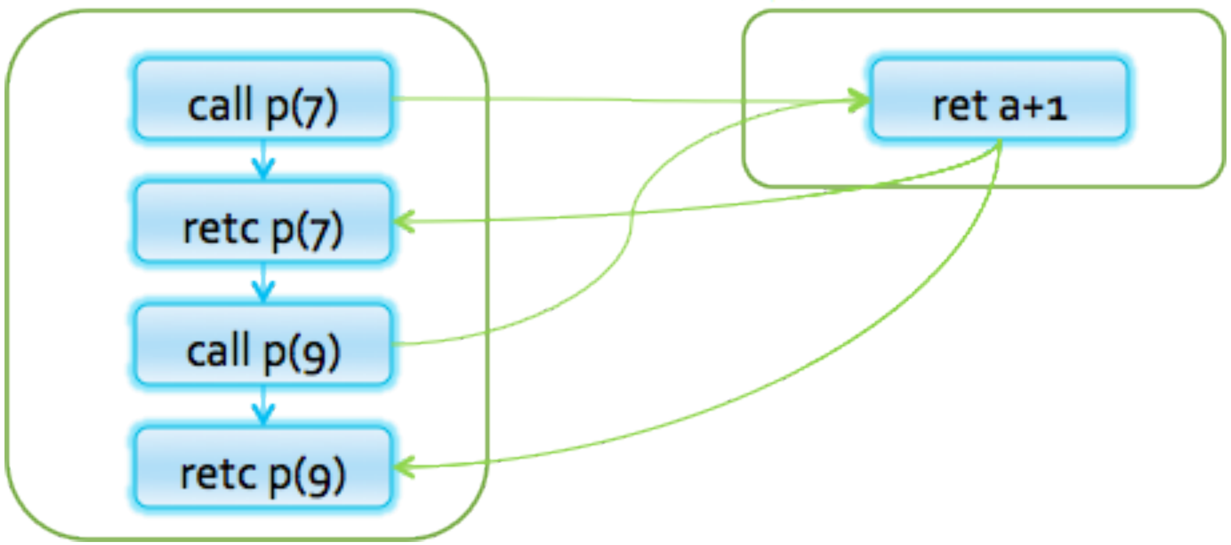
```
int p(int a) {
  return a + 1;
  }
```



Simple Example

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

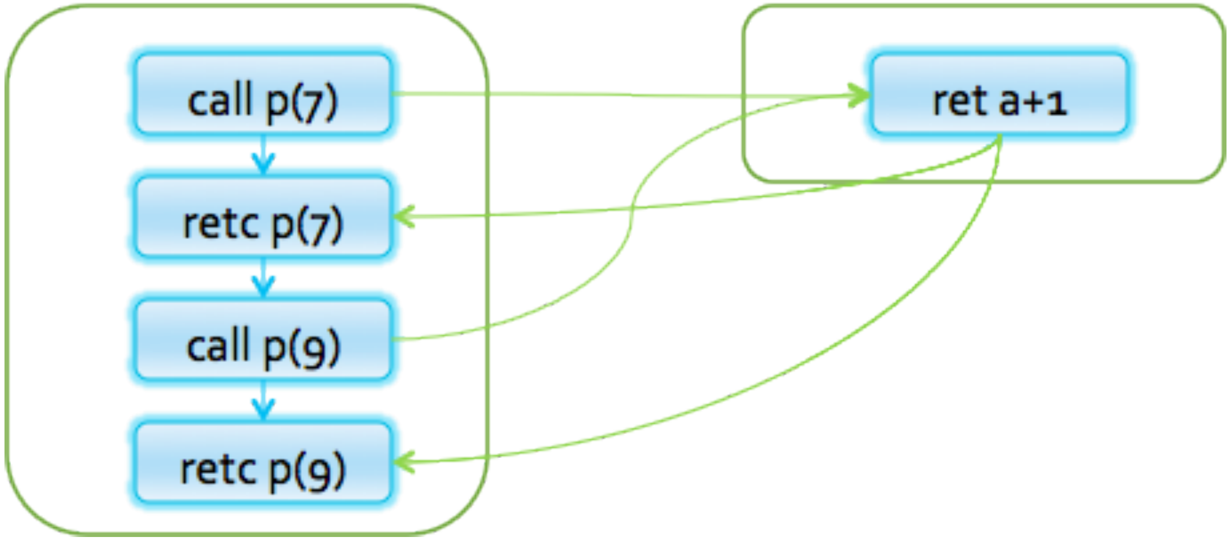
```
int p(int a) {
    [a ↦7]
    return a + 1;
}
```



Simple Example

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

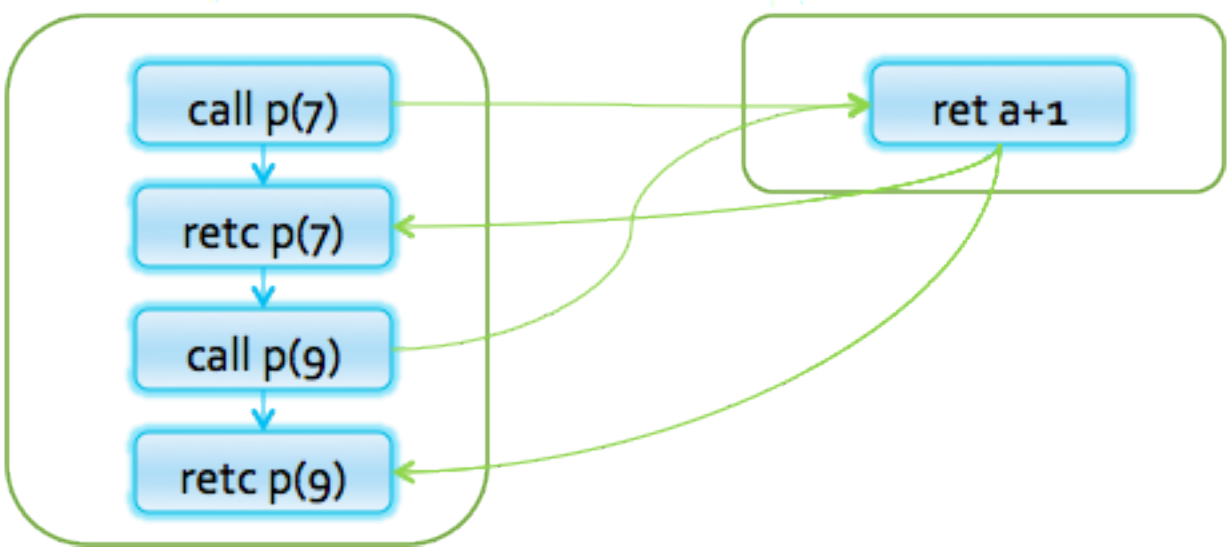
```
int p(int a) {
    [a ↦7]
    return a + 1;
    [a ↦7, $$ ↦8]

}
```



Simple Example

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

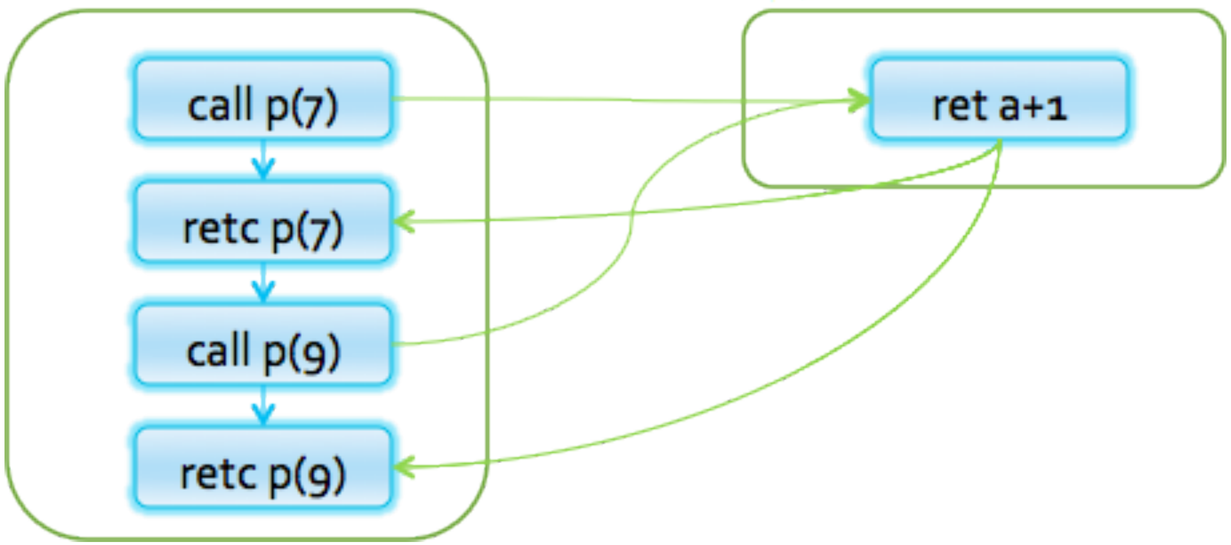
```
int p(int a) {
  [a ↦7]
    return a + 1;
  [a ↦7, $$ ↦8]

}
```



Simple Example

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

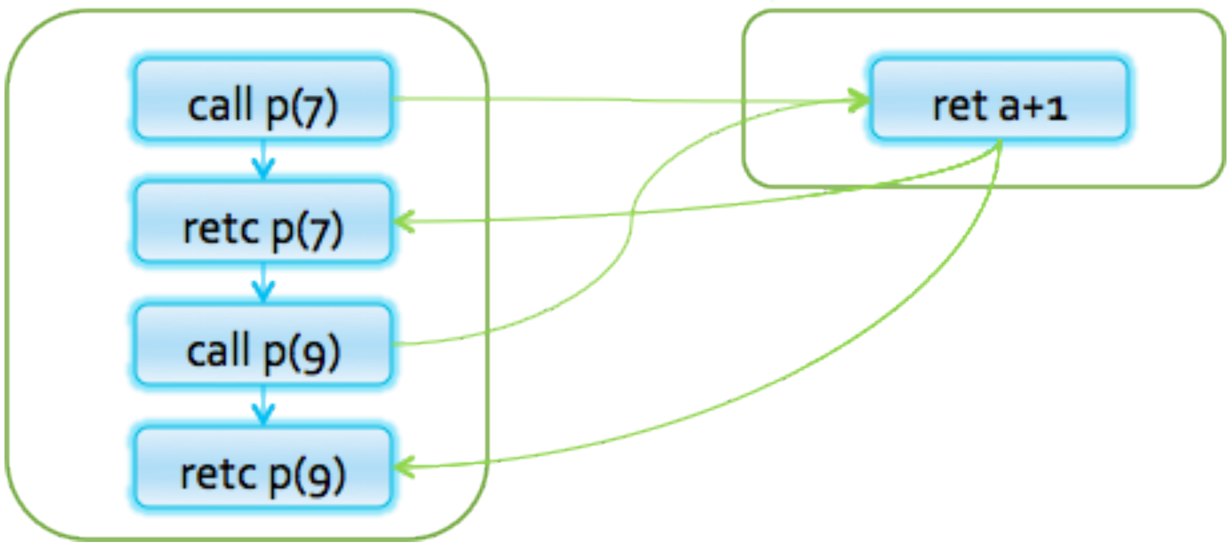
```
int p(int a) {
  [a ↦7]
    return a + 1;
  [a ↦7, $$ ↦8]

}
```



Simple Example

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

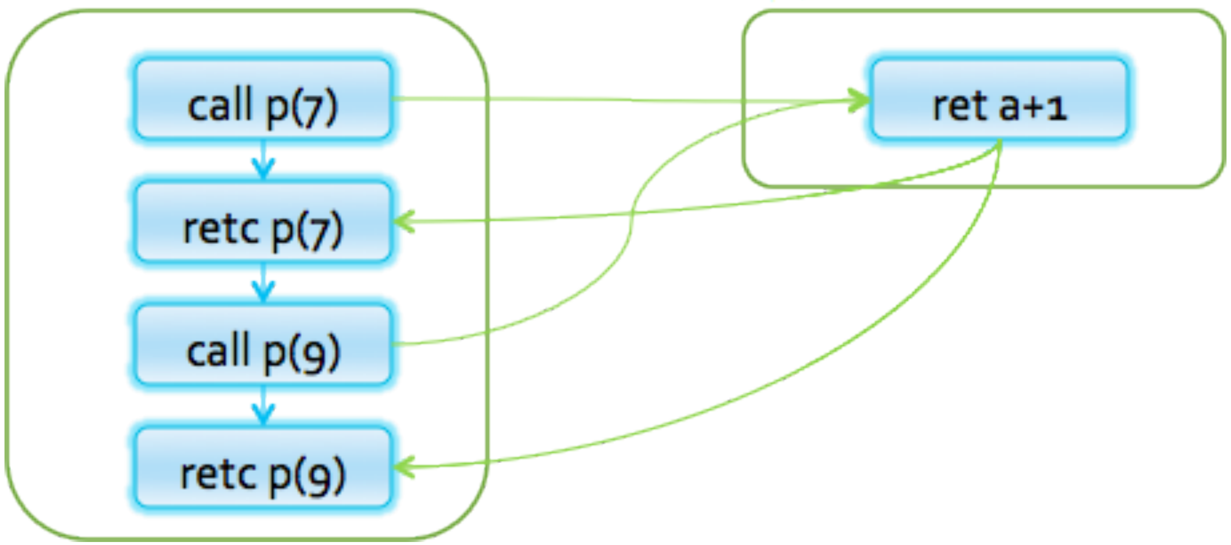
```
int p(int a) {
  [a ↦7]  [a ↦9]
    return a + 1;
  [a ↦7, $$ ↦8]

}
```



Simple Example

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

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

}
```



Simple Example

void main() {

    int x ;

    x = p(7);

    [x ↦8]

    x = p(9);

    [x ↦8]

}

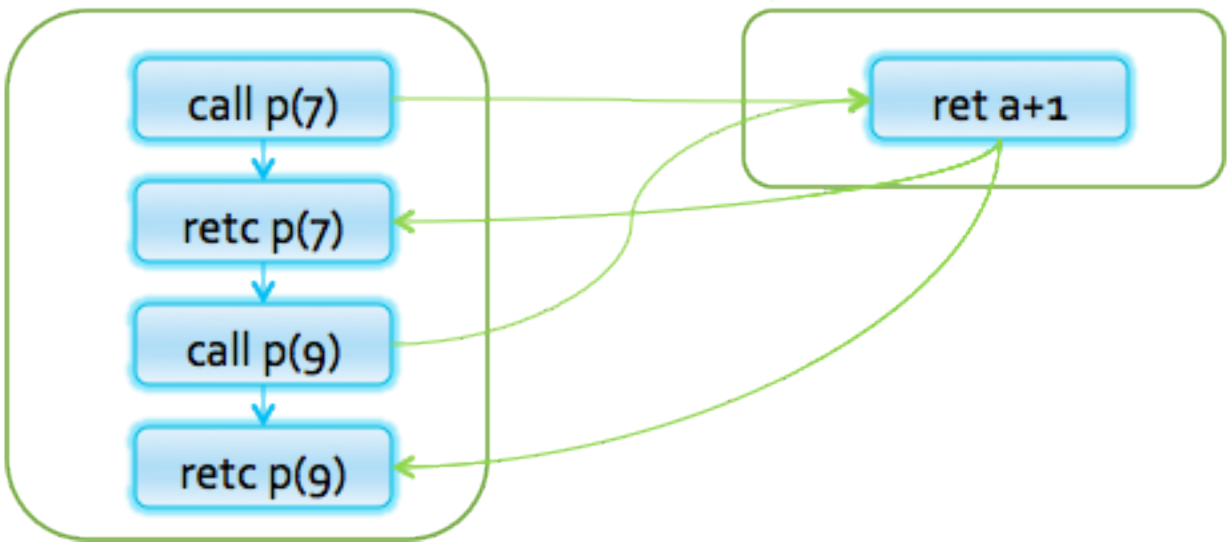
int p(int a) {

    [a ↦⊤]

    return a + 1;

    [a ↦⊤, $$ ↦⊤]

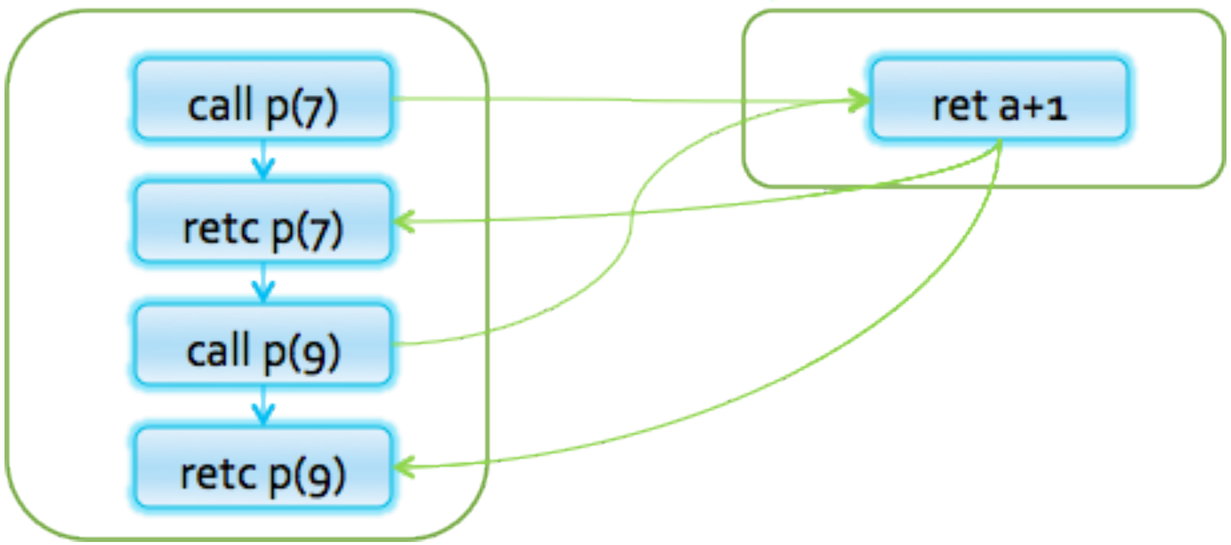
}



Simple Example

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

}
```

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

}
```



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 ↦⊤]
}
```

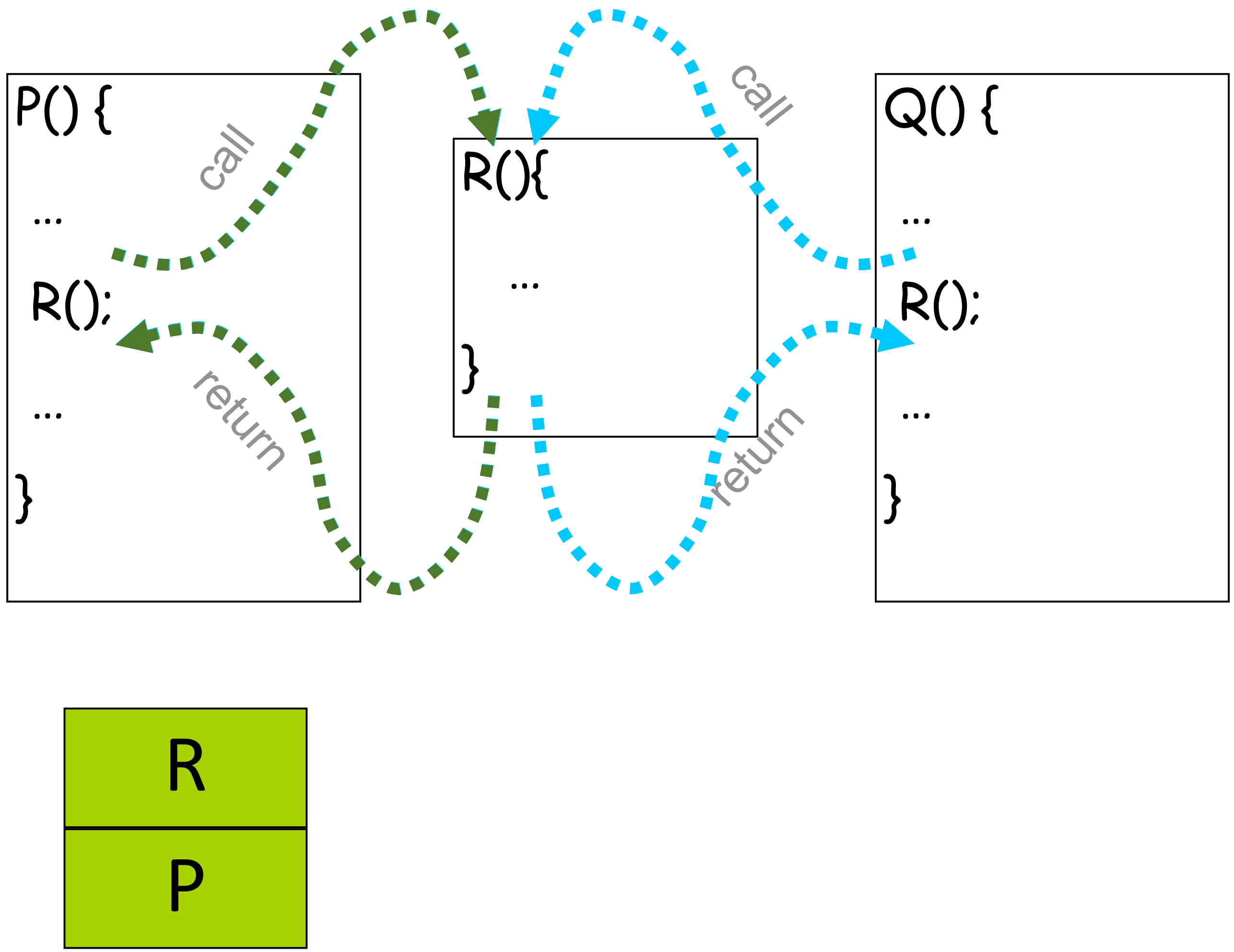
```
int p(int a) {
  [a ↦⊤]
  return a + 1;
  [a ↦⊤, $$ ↦⊤]
}
```

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

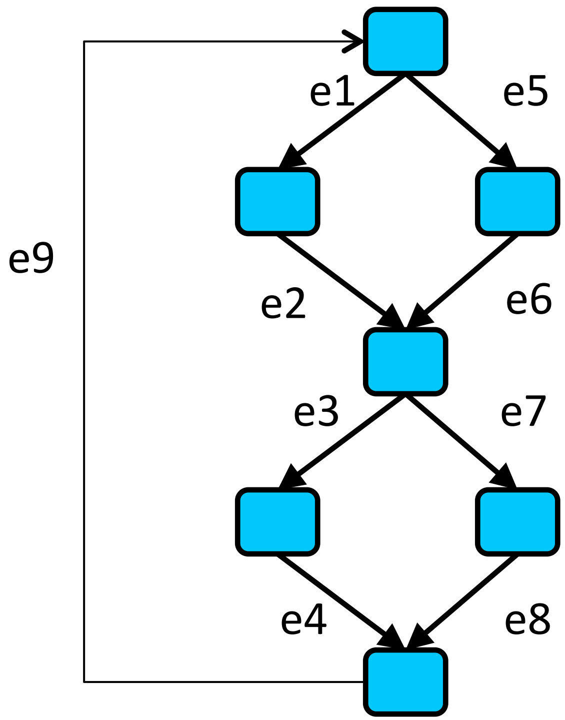
✓ Procedure Inlining
✓ The naive approach
- Valid paths
- The callstring approach
- The Functional Approach
- IFDS: Interprocedural Analysis via Graph Reachability
- IDE: Beyond graph reachability

- 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
  $f[e_1, e_2, \ldots, e_n](l) = f(e_n)(\ldots (f(e_2)(f(e_1)(l)) \ldots)$
- $JOP[v] = \sqcup\{f[e_1, e_2, \ldots, e_n](\iota) \mid [e_1, e_2, \ldots, e_n] \in paths(v)\}$

Join-Over-All-Paths (JOP)



Paths transformers:
f[e1,e2,e3,e4]
f[e1,e2,e7,e8]
f[e5,e6,e7,e8]
f[e5,e6,e3,e4]
f[e1,e2,e3,e4,e9, e1,e2,e3,e4]
f[e1,e2,e7,e8,e9, e1,e2,e3,e4,e9,…]
…

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

Number of program paths is unbounded due to loops

9

The lfp computation approximates JOP

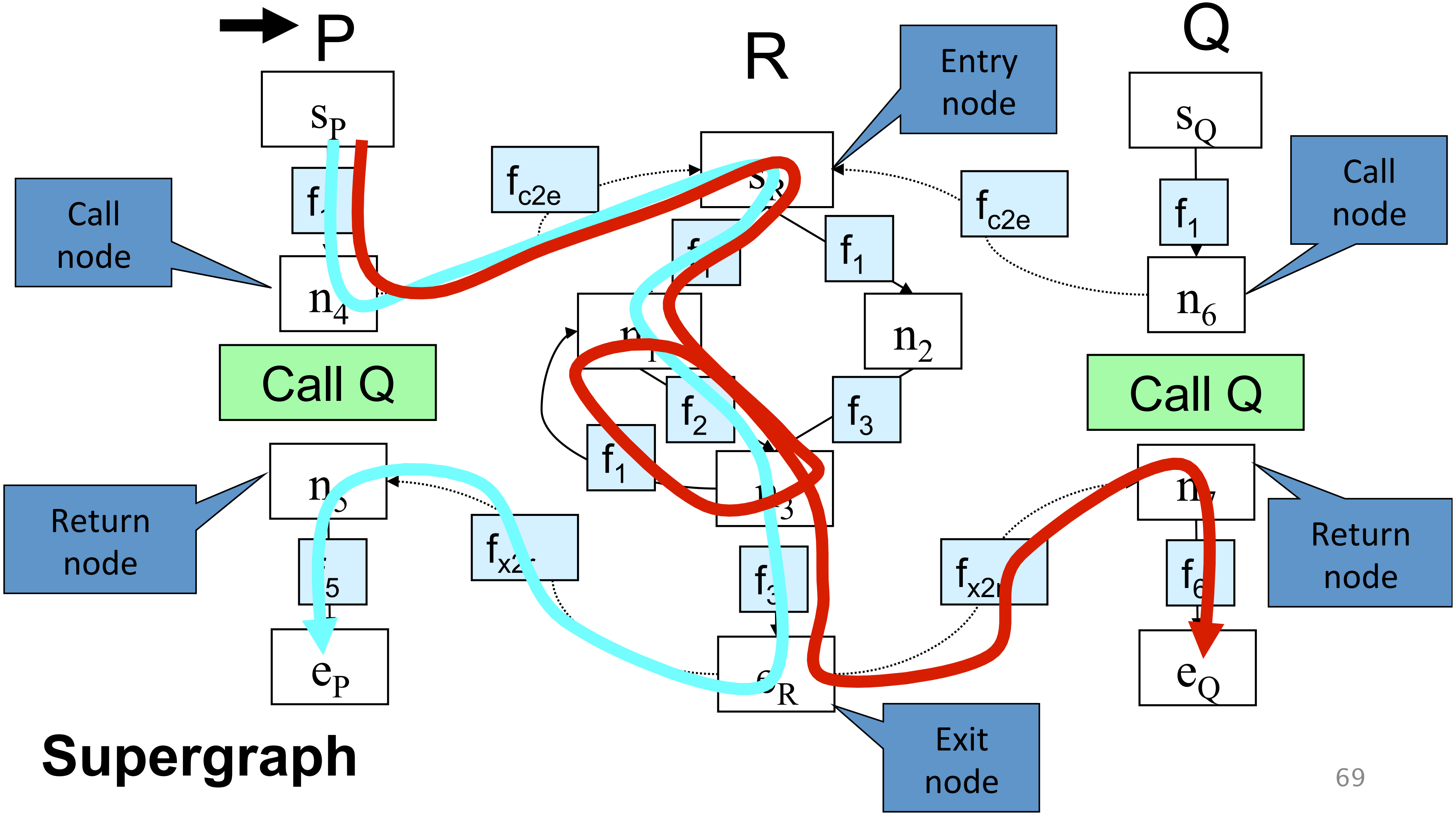
- JOP[v] $= \sqcup\{f\,[e_1, e_2, \ldots, e_n](\iota) \mid [e_1, e_2, \ldots, e_n] \in \text{paths}(v)\}$
- LFP[v] $= \sqcup\{f\,[e](\text{LFP}[v']) \mid e = (v', v)\}$

  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!

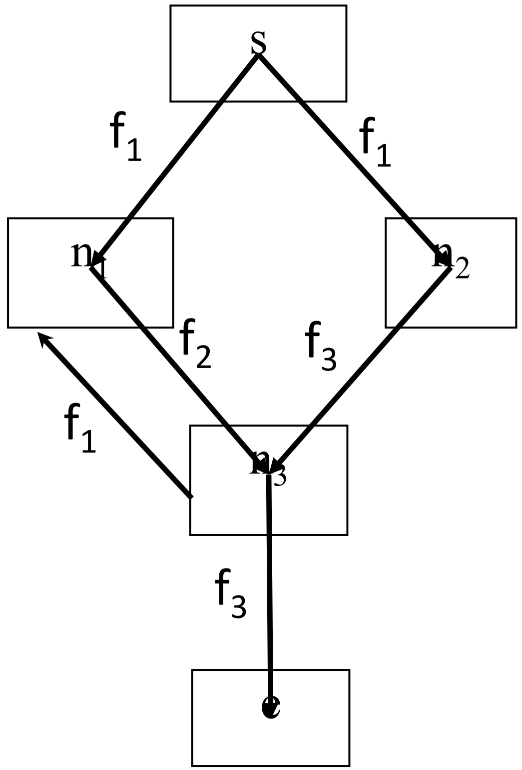
Interprocedural analysis



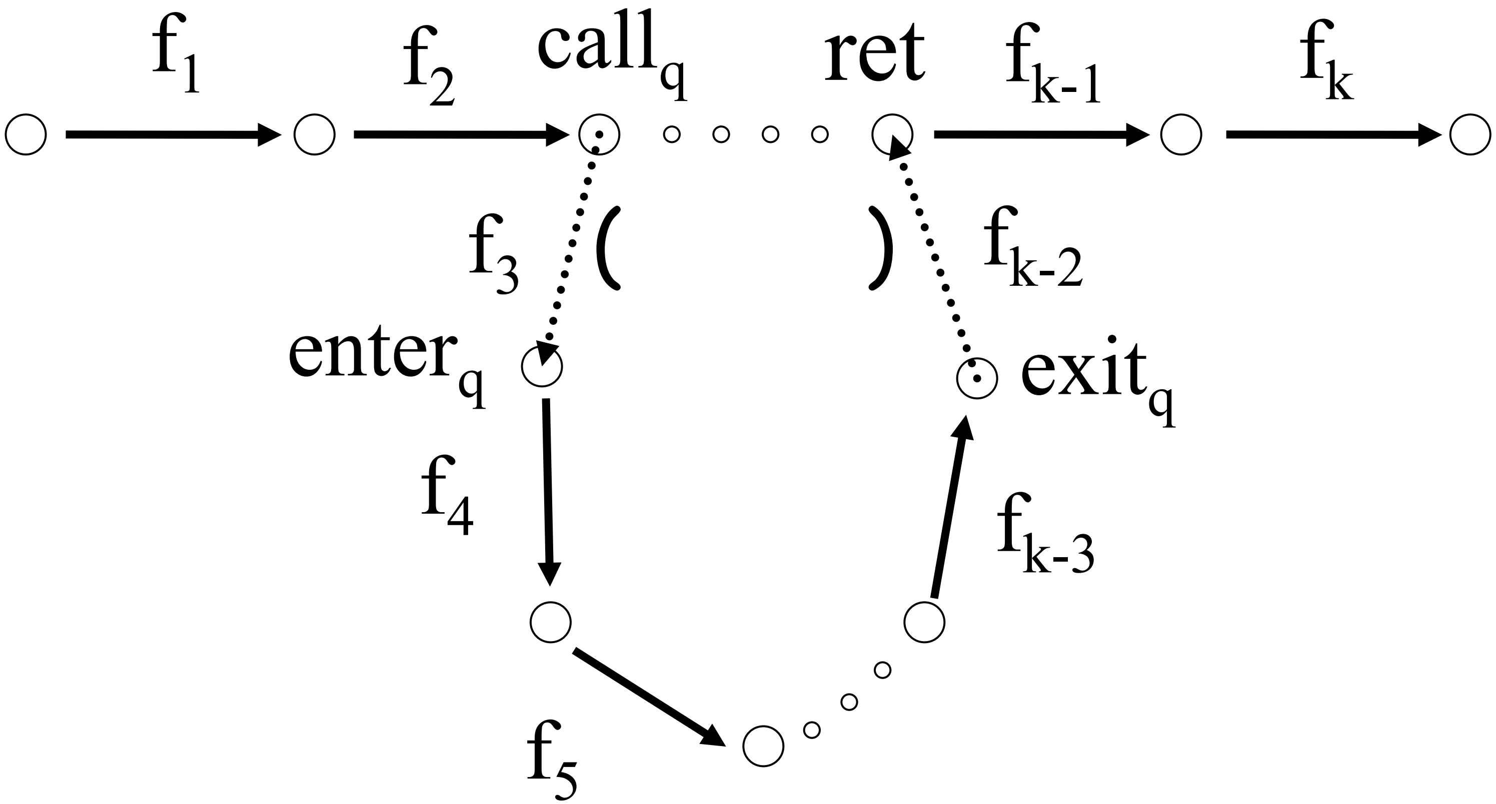
Supergraph

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

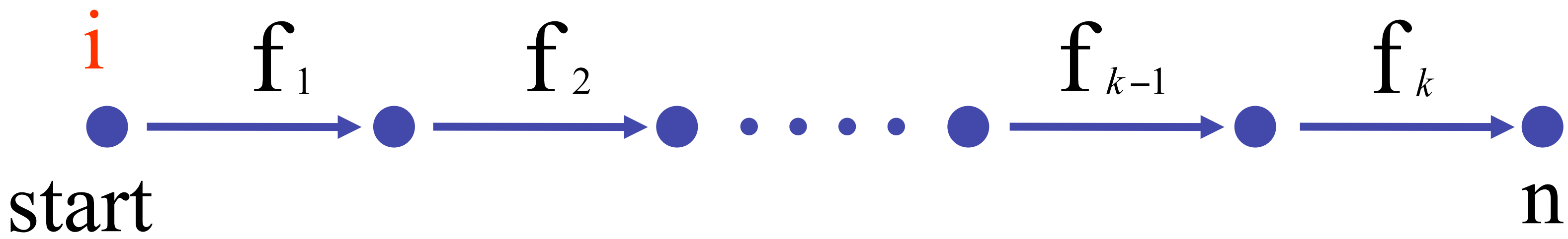


- IVP: all paths with matching calls and returns
  - And prefixes

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)



$$[\![fk \ \circ \ \ldots \circ f1]\!] \in L \rightarrow L$$

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

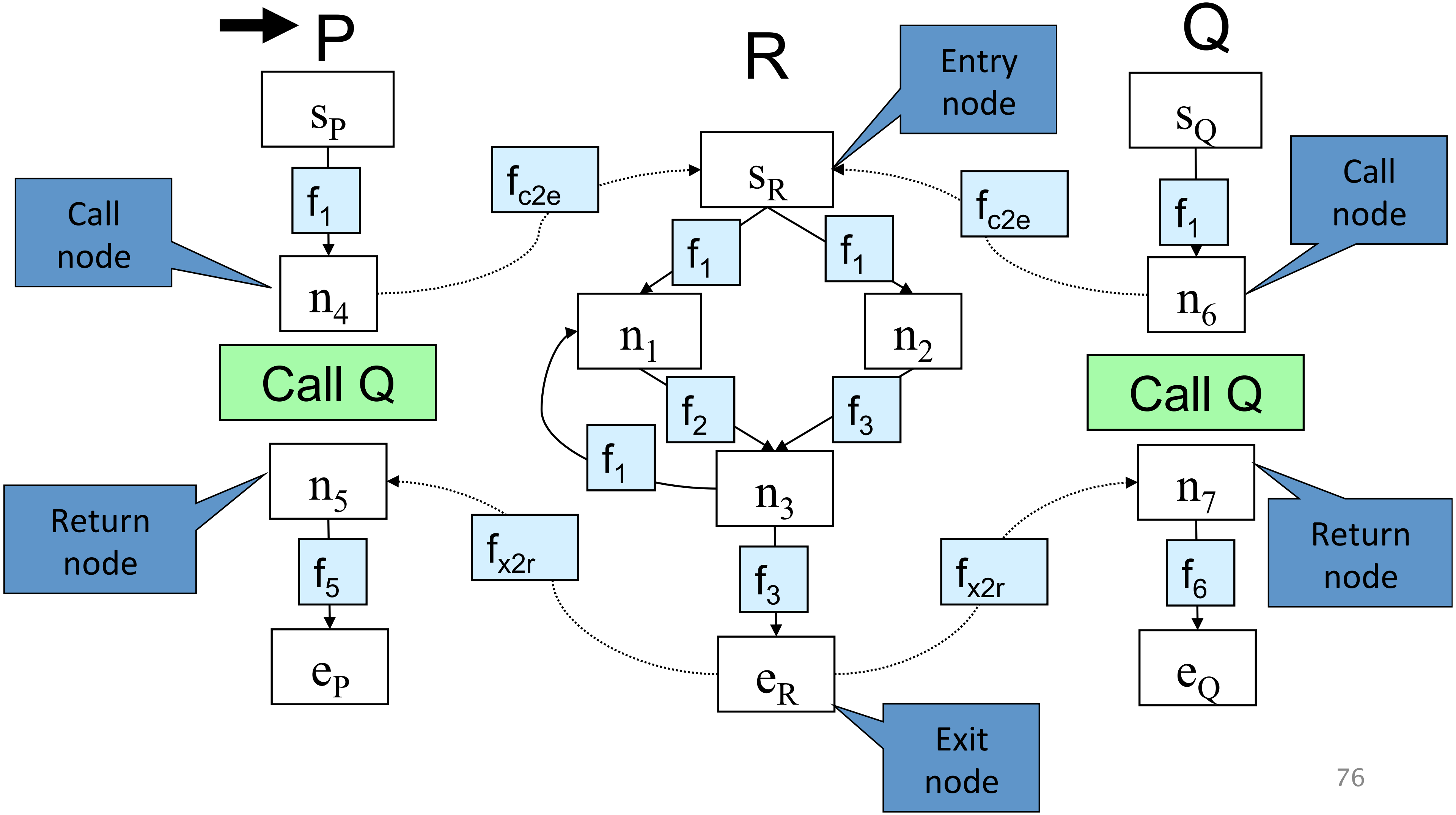
The Join-Over-Valid-Paths (JVP)

- vpaths(n) all valid paths from program start to n
- JVP[n] $= \sqcup\{[[e_1, e_2, \ldots, e]](\iota)$
  $(e_1, e_2, \ldots, e) \in$ vpaths(n)$\}$
- JVP $\sqsubseteq$ JOP
  - 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
s_P
s_R
s_Q
Entry node
Call node
Call node
f_1
f_1
f_1
f_1
f_1
f_{c2e}
f_{c2e}
n_4
n_1
n_2
n_6
Call Q
Call Q
f_2
f_3
n_3
n_5
n_7
Return node
Return node
f_5
f_{x2r}
f_3
f_{x2r}
f_6
e_P
e_R
e_Q
Exit node

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 ↦ ⊤]
    c2: x = p(9) ;
    ε: [x ↦ ⊤]
}
```

```
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 ↦⊤, $$↦⊤]
}
```

Handling Recursion

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

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

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 $\mapsto \top$]

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

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

```
}
```

Phase 2

void main() {

p(7);

$[x \mapsto -9]$

}

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

int p(int a) {

$[a \mapsto 7, x \mapsto 0]$ $[a \mapsto \top, x \mapsto 0]$

if (…) {

$[a \mapsto 7, x \mapsto 0]$ $[a \mapsto \top, x \mapsto 0]$

a = a -1 ;

$[a \mapsto 6, x \mapsto 0]$ $[a \mapsto \top, x \mapsto 0]$

p (a);

$[a \mapsto 6, x \mapsto -7]$ $[a \mapsto \top, x \mapsto \top]$

a = a + 1;

$[a \mapsto 7, x \mapsto -7]$ $[a \mapsto \top, x \mapsto \top]$

}

$[a \mapsto 7, x \mapsto 0]$ $[a \mapsto \top, x \mapsto \top]$

x = -2*a + 5;

$[a \mapsto 7, x \mapsto -9]$ $[a \mapsto \top, x \mapsto \top]$

}

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

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

Tabulation

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

- Propagation of (x,y) over edge e = (n,n')
  - Maintain summary: S[n' ,x] = S[n' ,x] ⊔ ⟦n⟧ (y))
  - n intra-node: ➔ n' : (x, ⟦n⟧ (y))
  - n call-node:
    - ➔ n' : (y,y) if S[n' ,y] = ⊥ 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

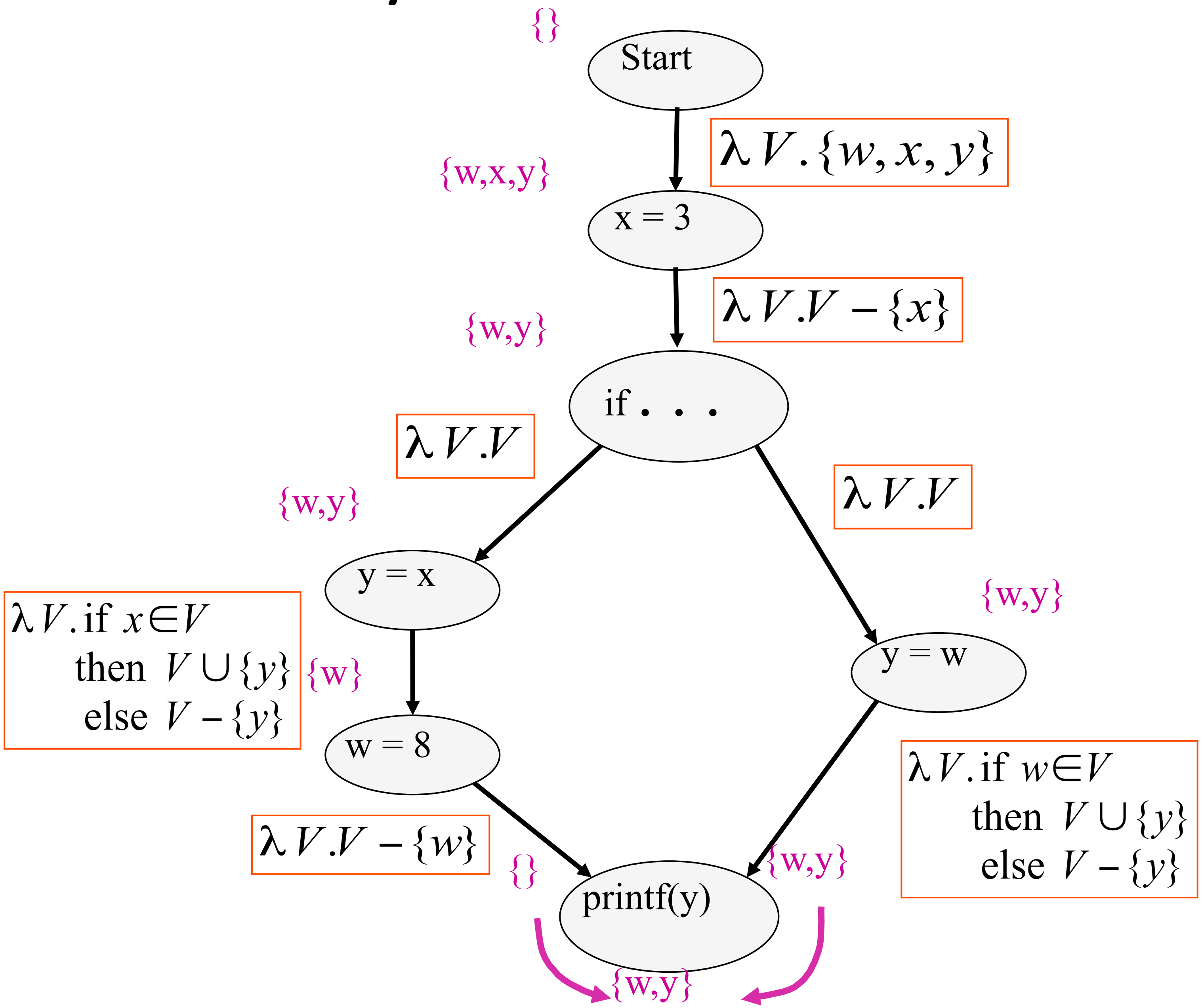
CFL-Graph reachability

- 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



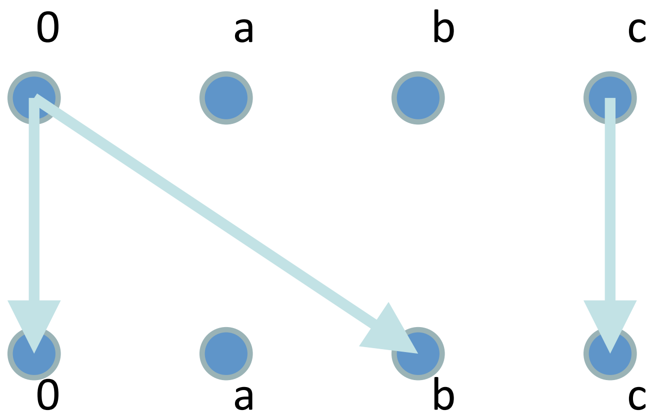
IFDS Problems

- Finite subset distributive
  - Lattice L = $\wp(D)$
  - $\sqsubseteq$ is $\subseteq$
  - $\sqcup$ is $\cup$
  - Transfer functions are distributive

- Efficient solution through formulation as CFL reachability

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}



207

Efficiently Representing Functions

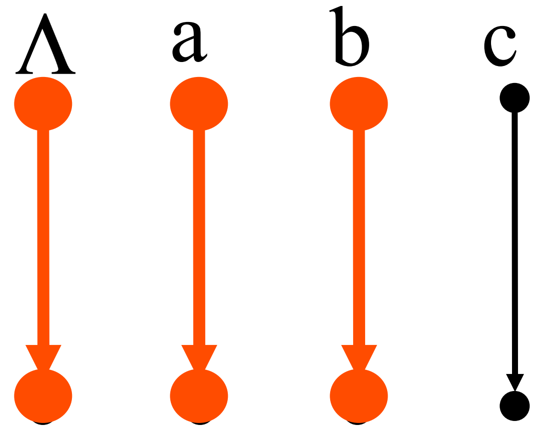
- Let $f:2^D \rightarrow 2^D$ be a distributive function
- Then:
  - $f(X) = f(\emptyset) \cup (\cup \{ f(\{z\}) \mid z \in X \})$
  - $f(X) = f(\emptyset) \cup (\cup \{ f(\{z\}) \setminus f(\emptyset) \mid z \in X \})$

Representing Dataflow Functions

Identity Function

$f = \lambda V.V$

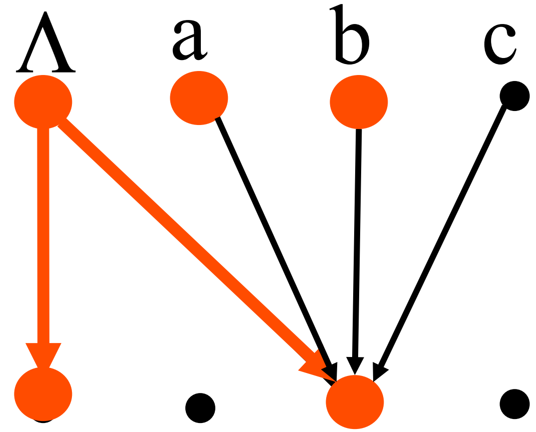
$f(\{a,b\}) = \{a,b\}$



Constant Function

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

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

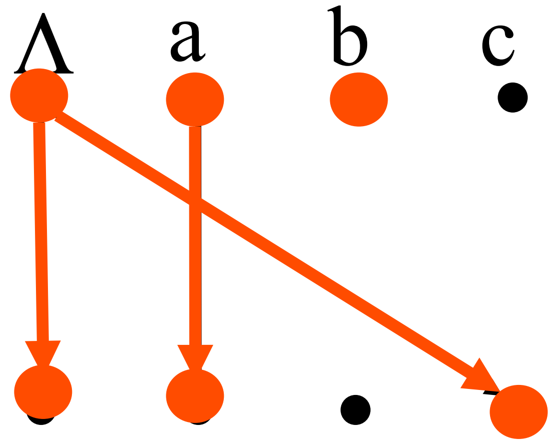


Representing Dataflow Functions

"Gen/Kill" Function

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

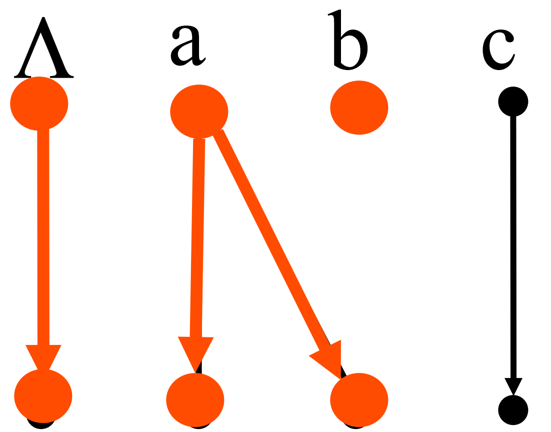
$$f(\{a,b\}) = \{a,c\}$$

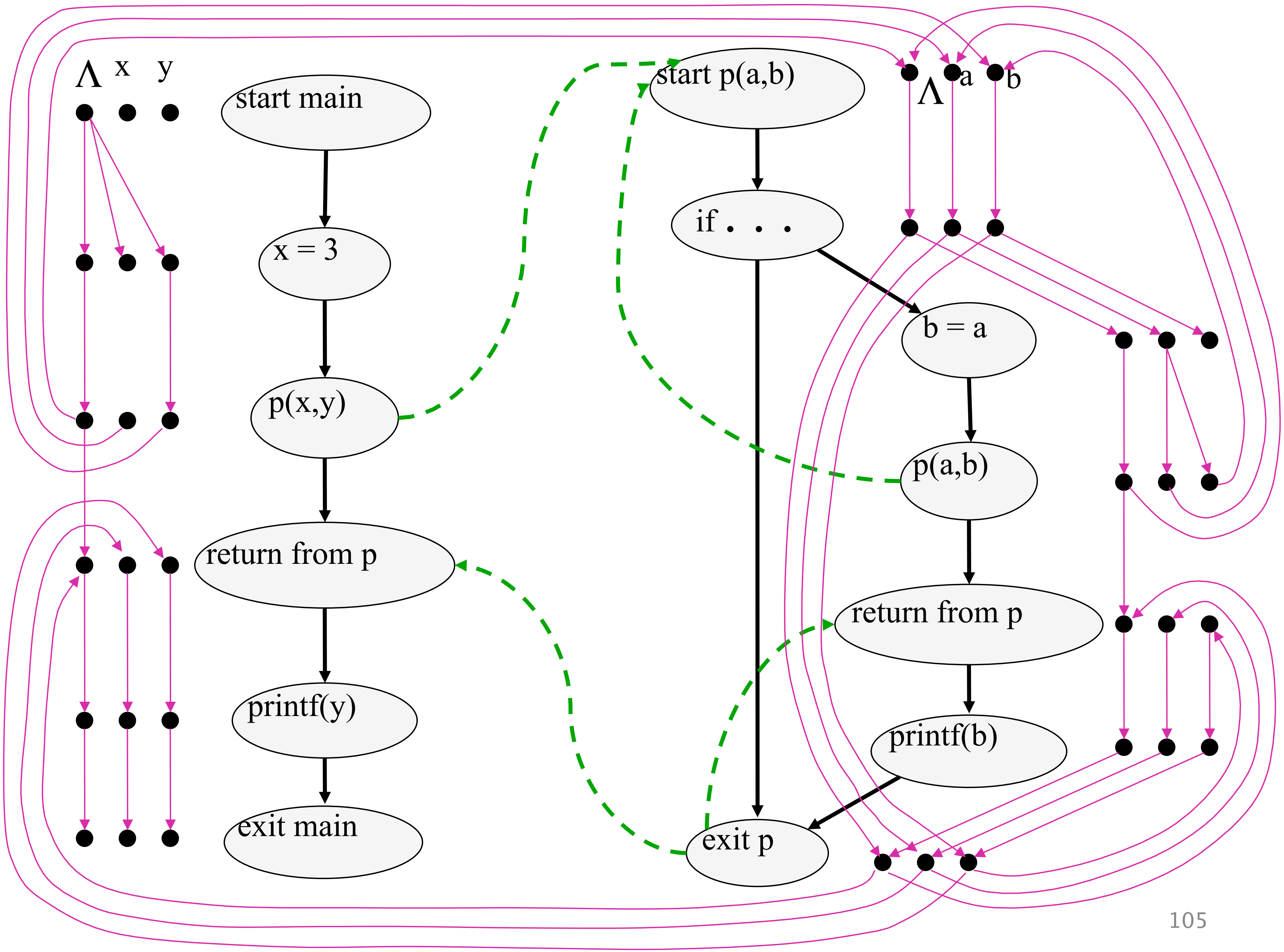


Non-"Gen/Kill" Function

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

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



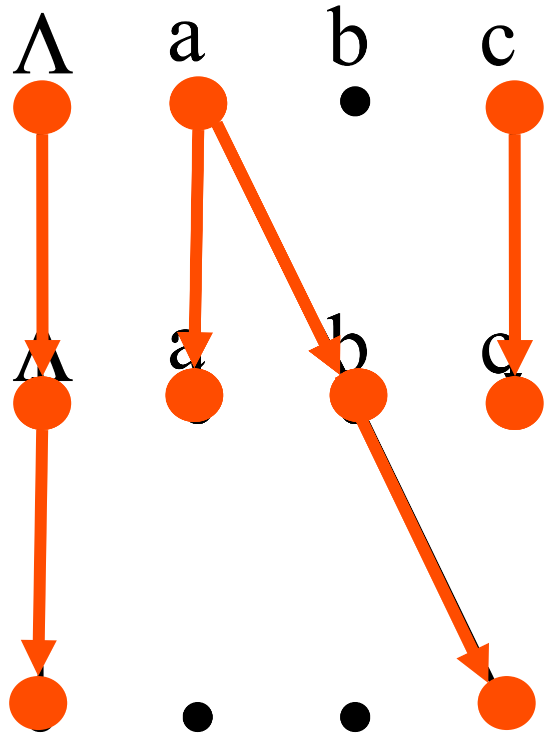


Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
Λ a b
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p

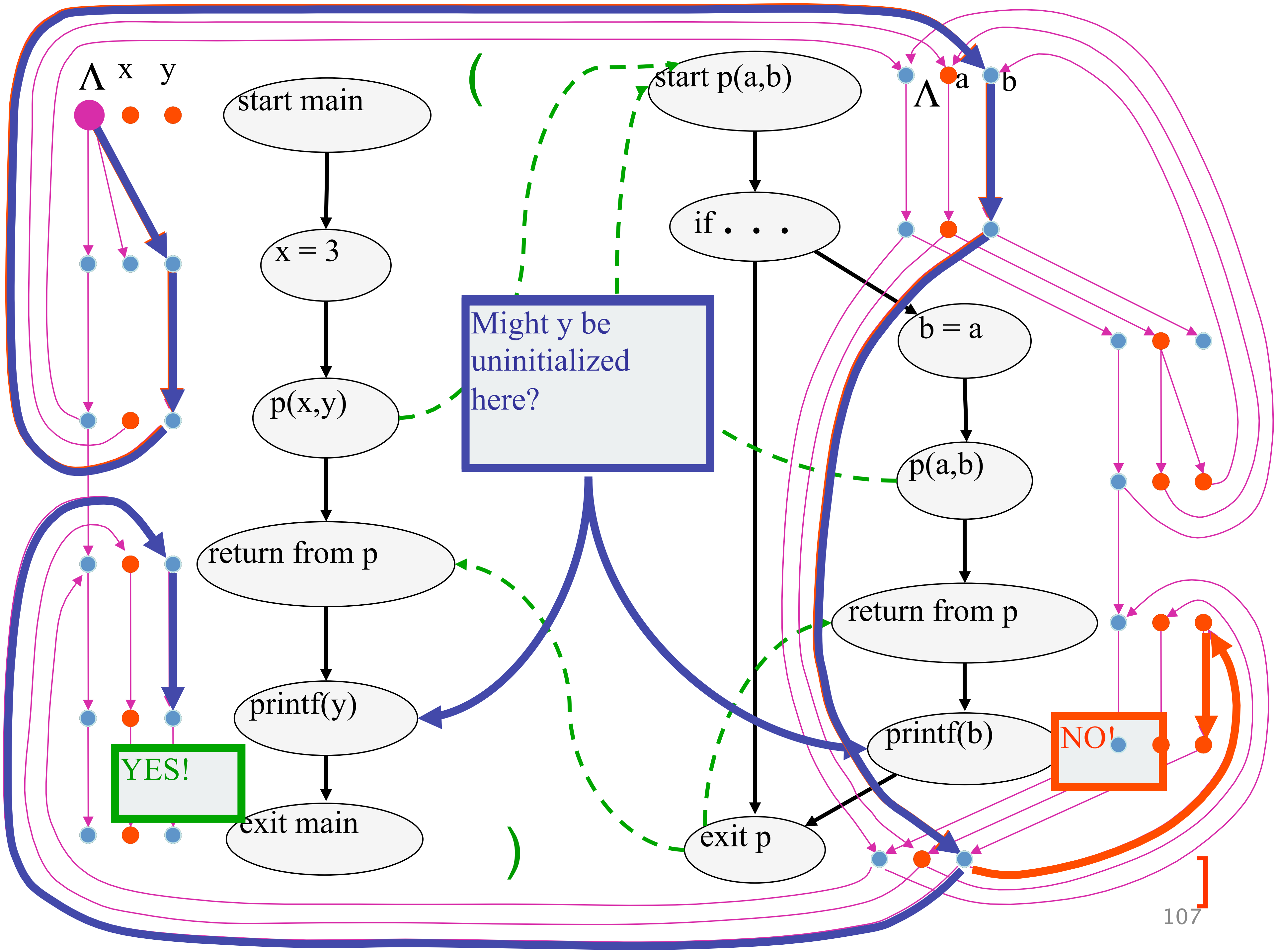
Composing Dataflow Functions

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

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



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



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!
]

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

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

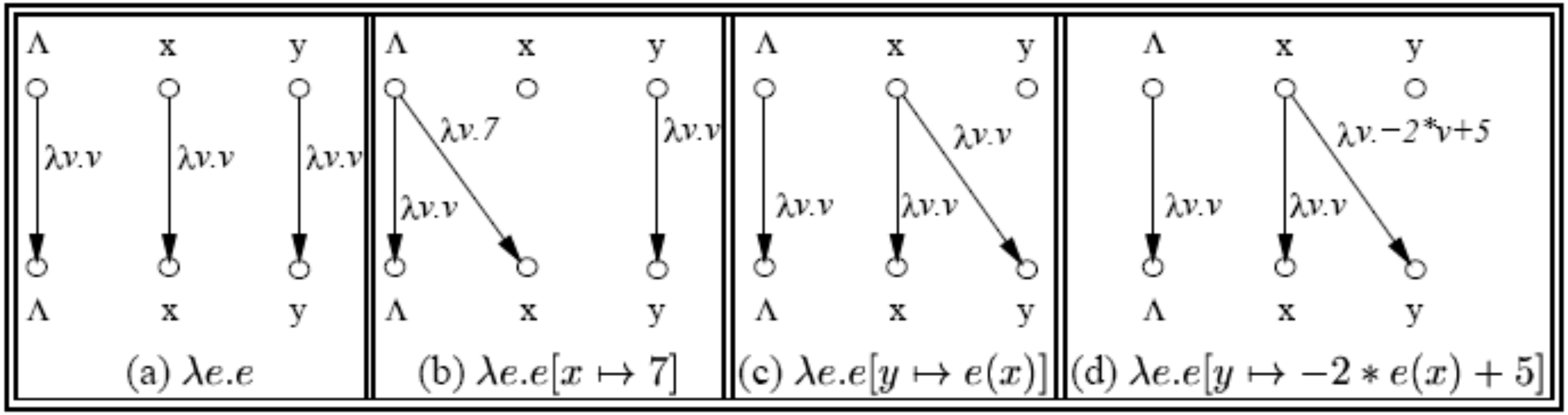
- Consider the constant propagation lattice
- The value of every variable y at the program exit can be represented by:

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

  $a_x, c \in Z \cup \{\perp, \top\} \quad b_x \in Z$

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

Linear constant propagation

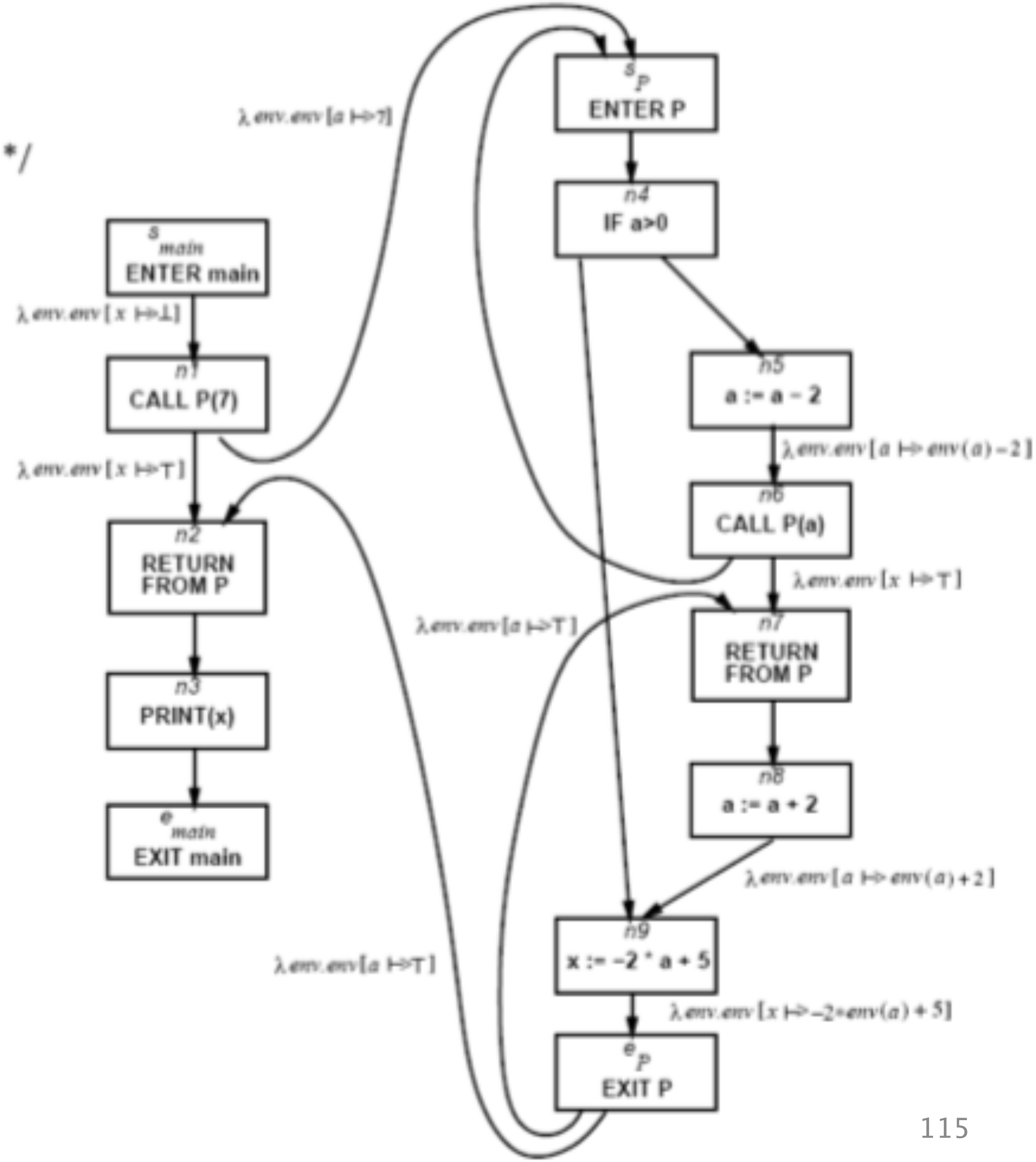


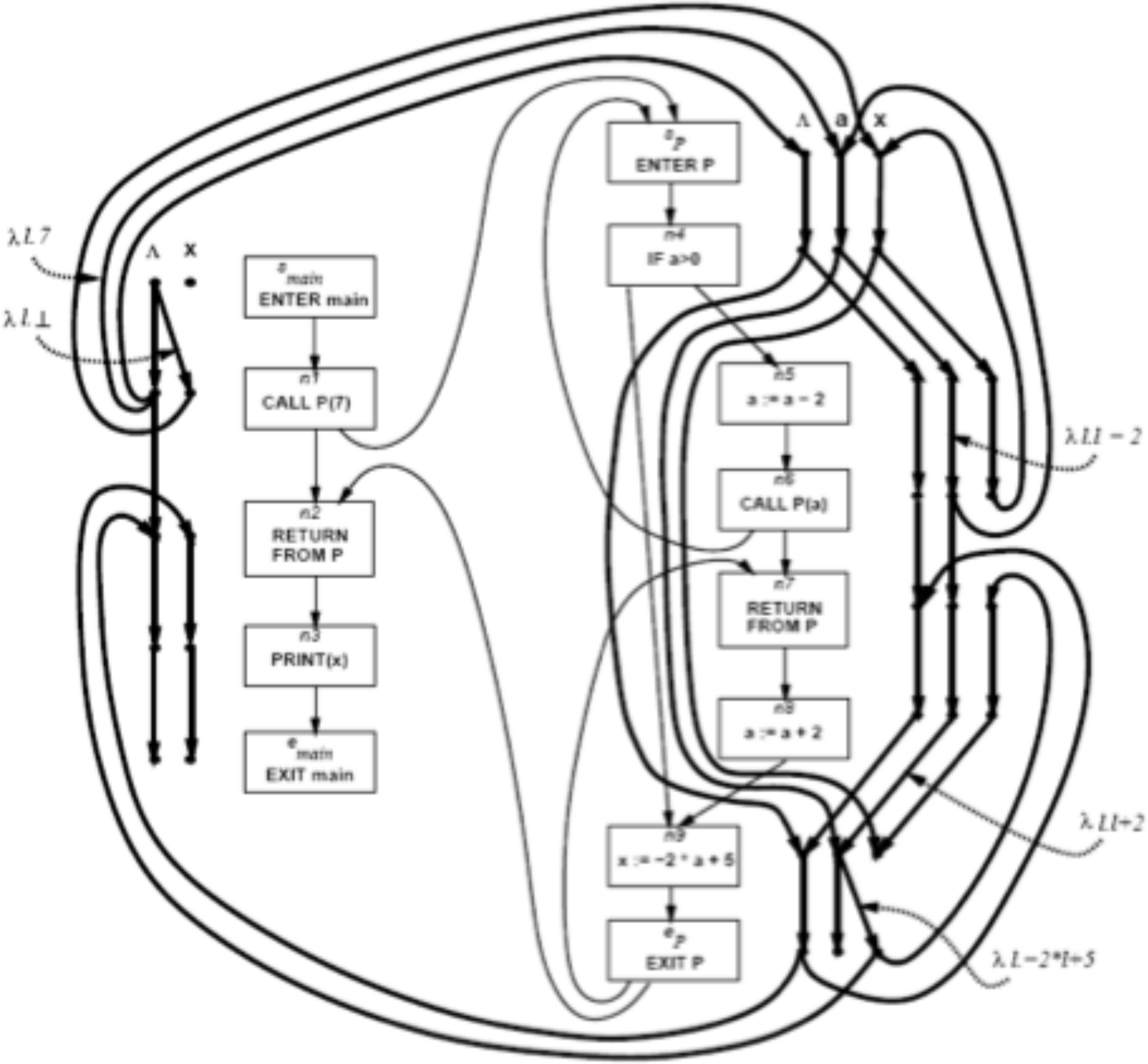
Point-wise representation of environment transformers

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
```




λl.7
λl.⊥
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
λl.l − 2
λl.l+2
λl.−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

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

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

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