Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 11: Shape Analysis + Interprocedural Analysis

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

3-Value logic based shape analysis

Sequential Stack

```
void push (int v) {
 Node *x = malloc(sizeof(Node));
 x->d = v;
 x->n = Top;
 Top = x;
}

 int pop() {
 if (Top == NULL) return EMPTY;
 Node *s = Top->n;
 int r = Top->d;
 Top = s;
 return r;
}
```

Want to Verify

No Null Dereference

Underlying list remains acyclic after each operation

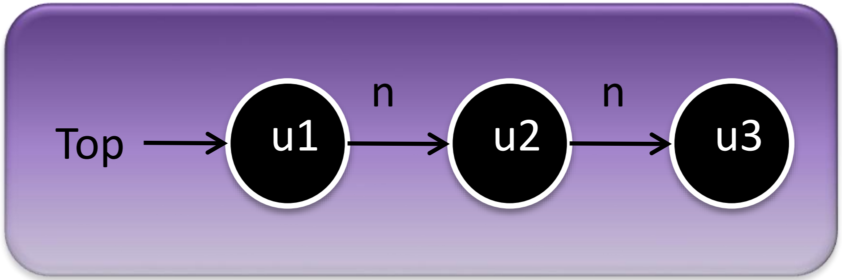
Shape Analysis via 3-valued Logic

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

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

Concrete State

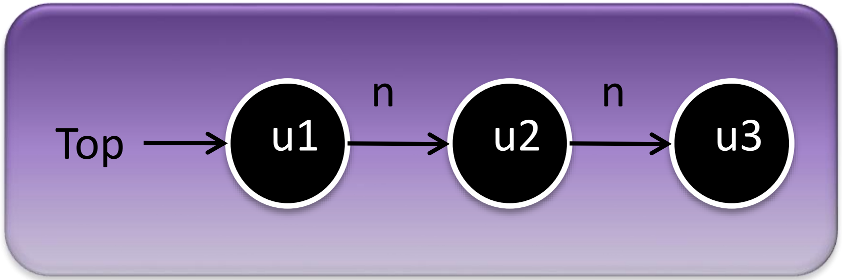
- represent a concrete state as a two-valued logical structure
  - Individuals = heap allocated objects
  - Unary predicates = object properties
  - Binary predicates = relations
- parametric vocabulary



(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

Formulae for Observing Properties

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

$\exists w: x(w)$

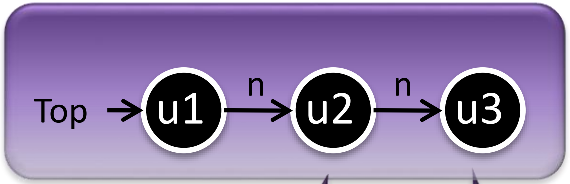
```
 x→d = v;
```

$\exists w: x(w)$

```
 x→n = Top;


 Top = x;

}
```

$\neg\exists v1,v2: n(v1, v2) \wedge n^*(v2, v1)$

$\neg\exists v1,v2: n(v1, v2) \wedge Top(v2)$



Top != null

$\exists w:Top(w)$ 1

No node precedes Top

$\neg\exists v1,v2: n(v1, v2) \wedge Top(v2)$ 1

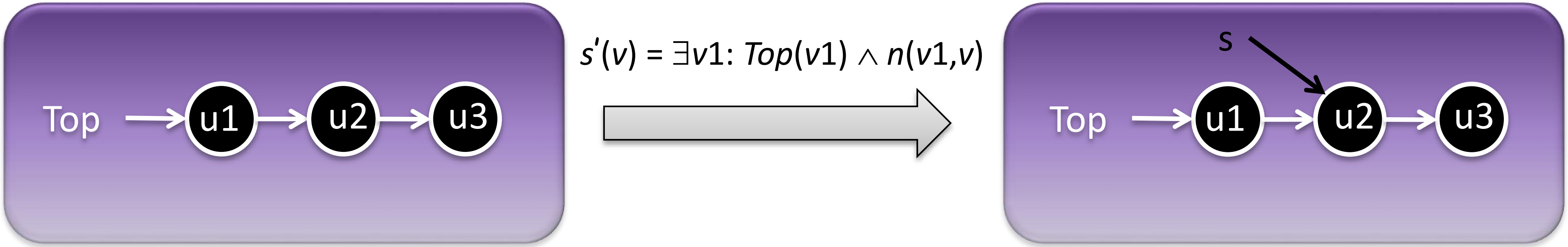
No Cycles

$\neg\exists v1,v2: n(v1, v2) \wedge n^*(v2, v1)$ 1

Concrete Interpretation Rules

Statement	Update formula
x =NULL	$x'(v)= 0$
x= malloc()	$x'(v) = \text{IsNew}(v)$
x=y	$x'(v)= y(v)$
x=y →next	$x'(v)= \exists w: y(w) \land n(w, v)$
x →next=y	$n'(v, w) = (\neg x(v) \land n(v, w)) \lor (x(v) \land y(w))$

Example: $s = Top \rightarrow n$



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

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

Collecting Semantics

$$
\mathrm{CSS}\,[v] = \begin{cases}
\{ <\emptyset,\emptyset> \} & \text{if } v = \text{entry} \\[2ex]
\bigcup\limits_{\substack{(w,v) \in E(G),\\ w \in \mathrm{Assignments}(G)}} \{ \llbracket \mathrm{st}(w) \rrbracket(S) \mid S \in \mathrm{CSS}[w] \} \cup & \\[2ex]
\bigcup\limits_{\substack{(w,v) \in E(G),\\ w \in \mathrm{Skip}(G)}} \{ S \mid S \in \mathrm{CSS}[w] \} \cup & \\[2ex]
\bigcup\limits_{(w,v) \in \text{True-Branches}(G)} \{ S \mid S \in \mathrm{CSS}[w] \ \text{ and } S \vDash \mathrm{cond}(w) \} \cup & \text{othrewise} \\[2ex]
\bigcup\limits_{(w,v) \in \text{False-Branches}(G)} \{ S \mid S \in \mathrm{CSS}[w] \ \text{ and } S \vDash \neg \mathrm{cond}(w) \} &
\end{cases}
$$

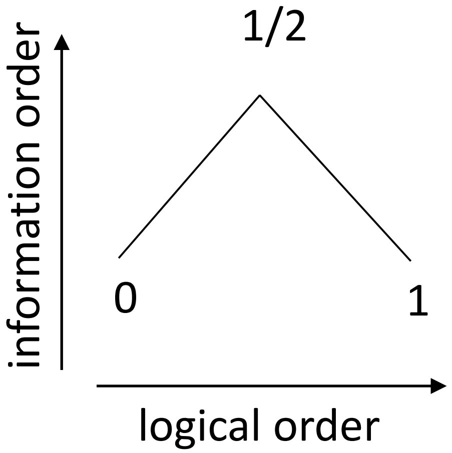
Collecting Semantics

- 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

- Next step:
  find a bounded abstract representation

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 relation symbols in P

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

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

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

Boolean Connectives [Kleene]

$\wedge$	0	1/2	1
0	0	0	0
1/2	0	1/2	1/2
1	0	1/2	1

$\vee$	0	1/2	1
0	0	1/2	1
1/2	1/2	1/2	1
1	1	1	1

Property Space

- 3-struct[P] = the set of 3-valued logical structures over a vocabulary (set of predicates) P

- Abstract domain
  - $\wp$(3-Struct[P])
  - $\sqsubseteq$ is $\subseteq$

Embedding Order

- Given two structures S = <U,ι >, S' = <U', ι'> and an onto function f : U $\rightarrow$ U' mapping individuals in U to individuals in U'
- We say that f embeds S in S' (denoted by S $\sqsubseteq$ S') if
  - for every predicate symbol p $\in$ P of arity k: u1, …, uk $\in$ U, ι(p)(u1, ..., uk) $\sqsubseteq$ ι'(p)(f(u1), ..., f (uk))
  - and for all u' $\in$ U'
    (| { u | f (u) = u' } | > 1) $\sqsubseteq$ ι'(sm)(u')

- We say that S can be embedded in S' (denoted by S $\sqsubseteq$ S') if there exists a function f such that S $\sqsubseteq^f$ S'

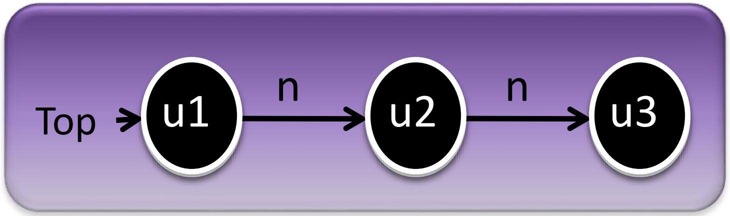
Tight Embedding

- S' = <U', ι'> is a tight embedding of S=< U, ι > with respect to a function f if:
  - S' does not lose unnecessary information

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

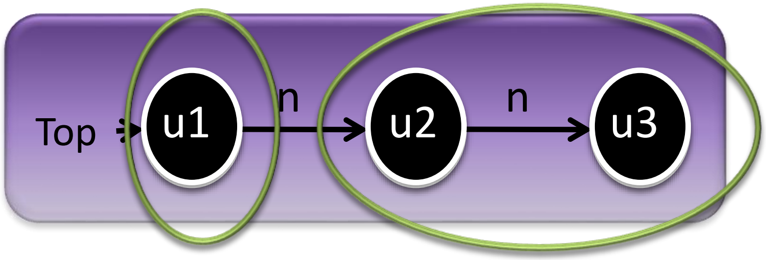
- One way to get tight embedding is canonical abstraction

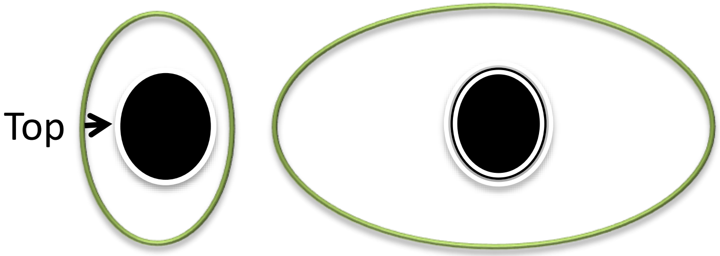
Canonical Abstraction



[Sagiv, Reps, Wilhelm, TOPLAS02]

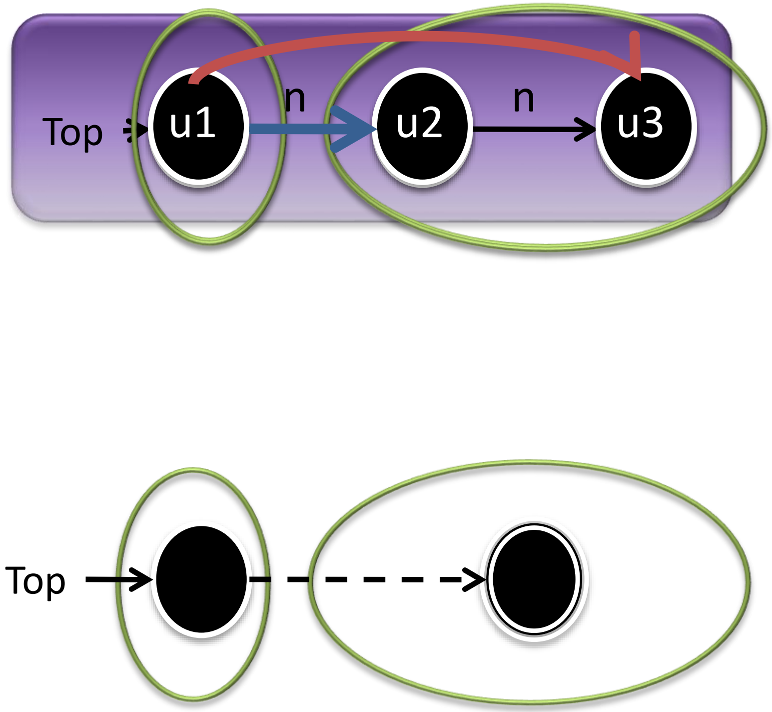
Canonical Abstraction



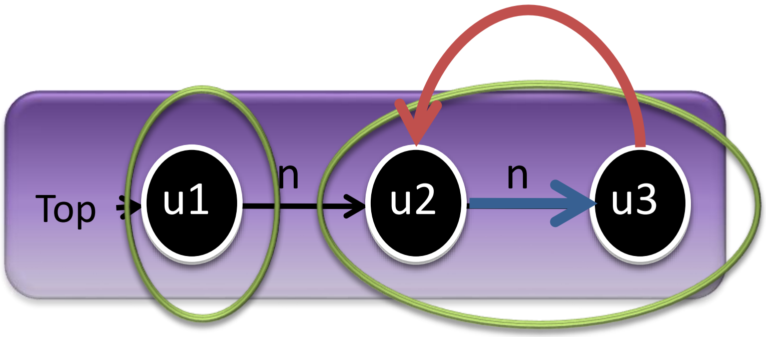


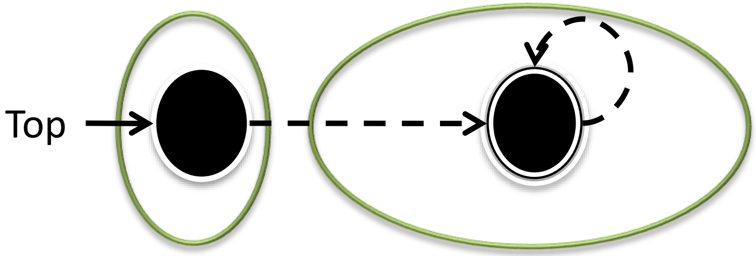
[Sagiv, Reps, Wilhelm, TOPLAS02]

Canonical Abstraction

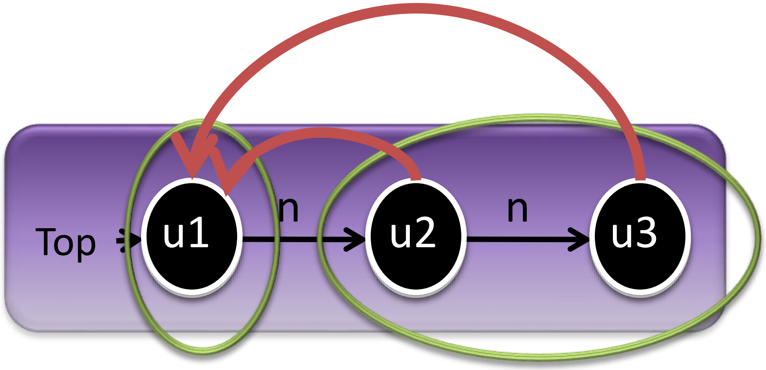


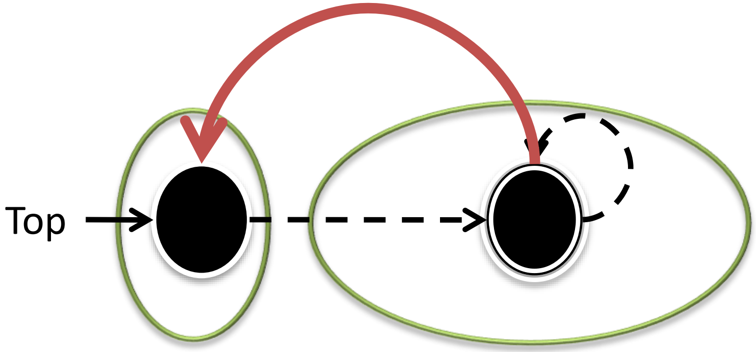
Canonical Abstraction



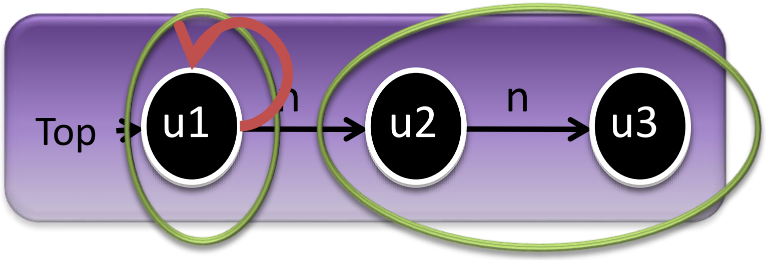


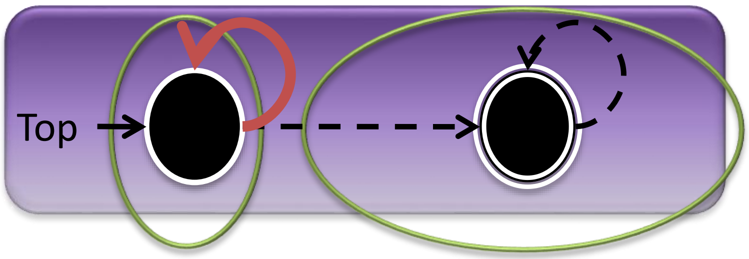
Canonical Abstraction





Canonical Abstraction





Canonical Abstraction (β)

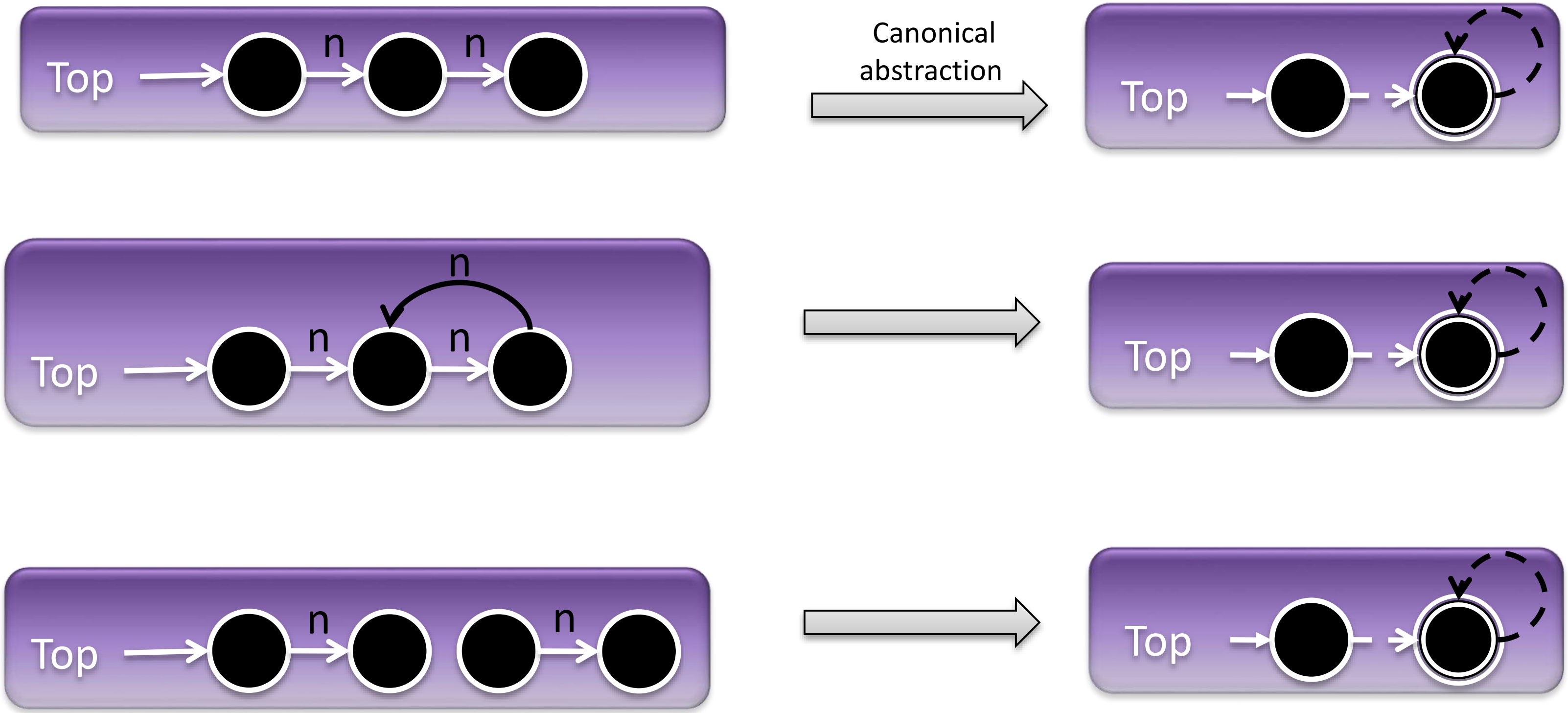
- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values

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

- 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

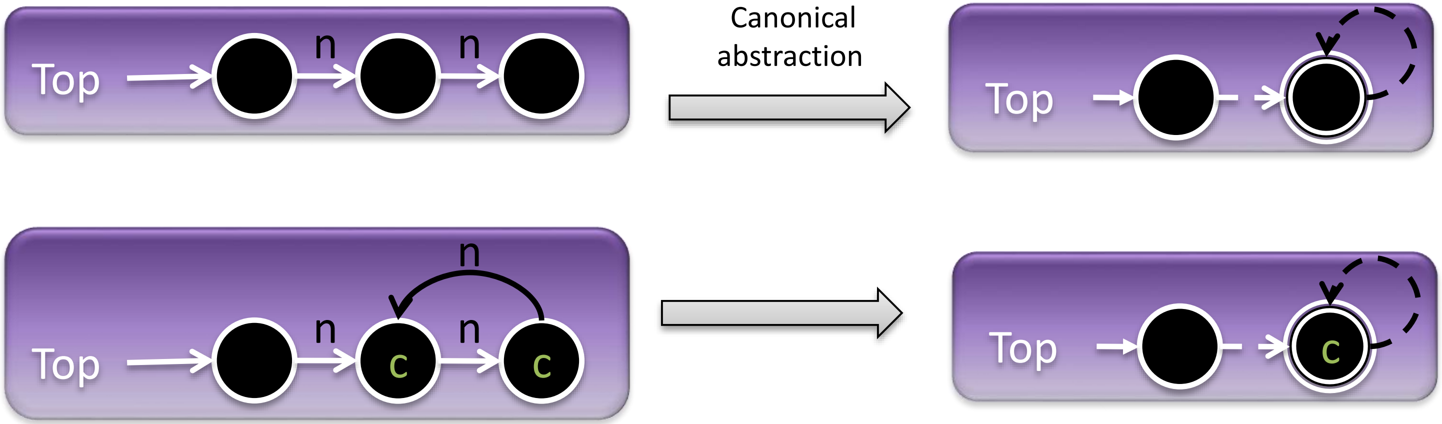


Instrumentation Predicates

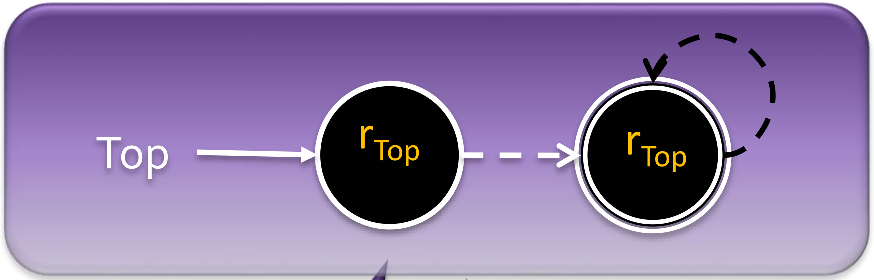
- Record additional derived information via predicates

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

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



Embedding Theorem:
Conservatively Observing Properties



No Cycles

$\neg\exists v1,v2: n(v1, v2) \wedge n^*(v2, v1)$ 1/2

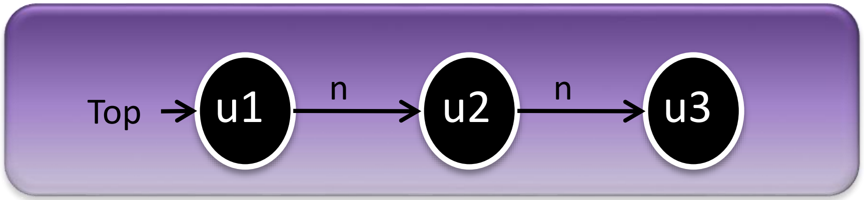
No cycles (derived)

$\forall v: \neg c(v)$ 1

Operational Semantics

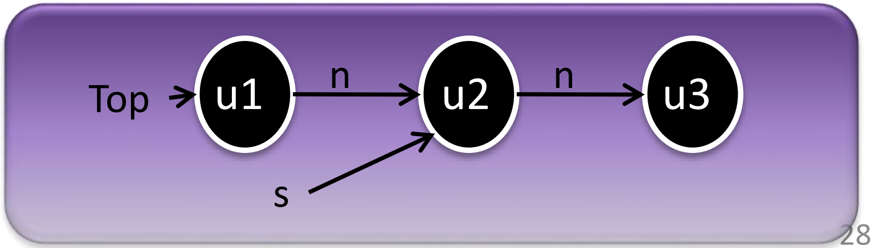
```
void push (int v) {
 Node *x = malloc(sizeof(Node));
 x->d = v;
 x->n = Top;
 Top = x;
}
```

```
 int pop() {
 if (Top == NULL) return EMPTY;
 Node *s = Top->n;
 int r = Top->d;
 Top = s;
 return r;
}
```

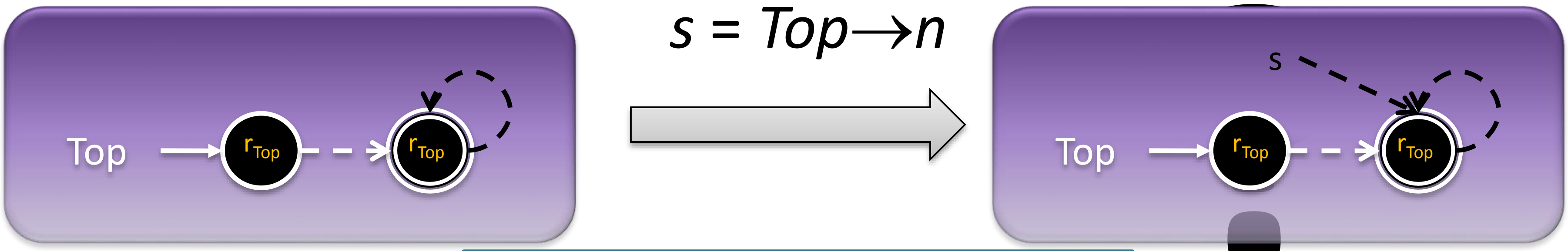


$[\![$`s = Top->n`$]\!]$

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



Abstract Semantics

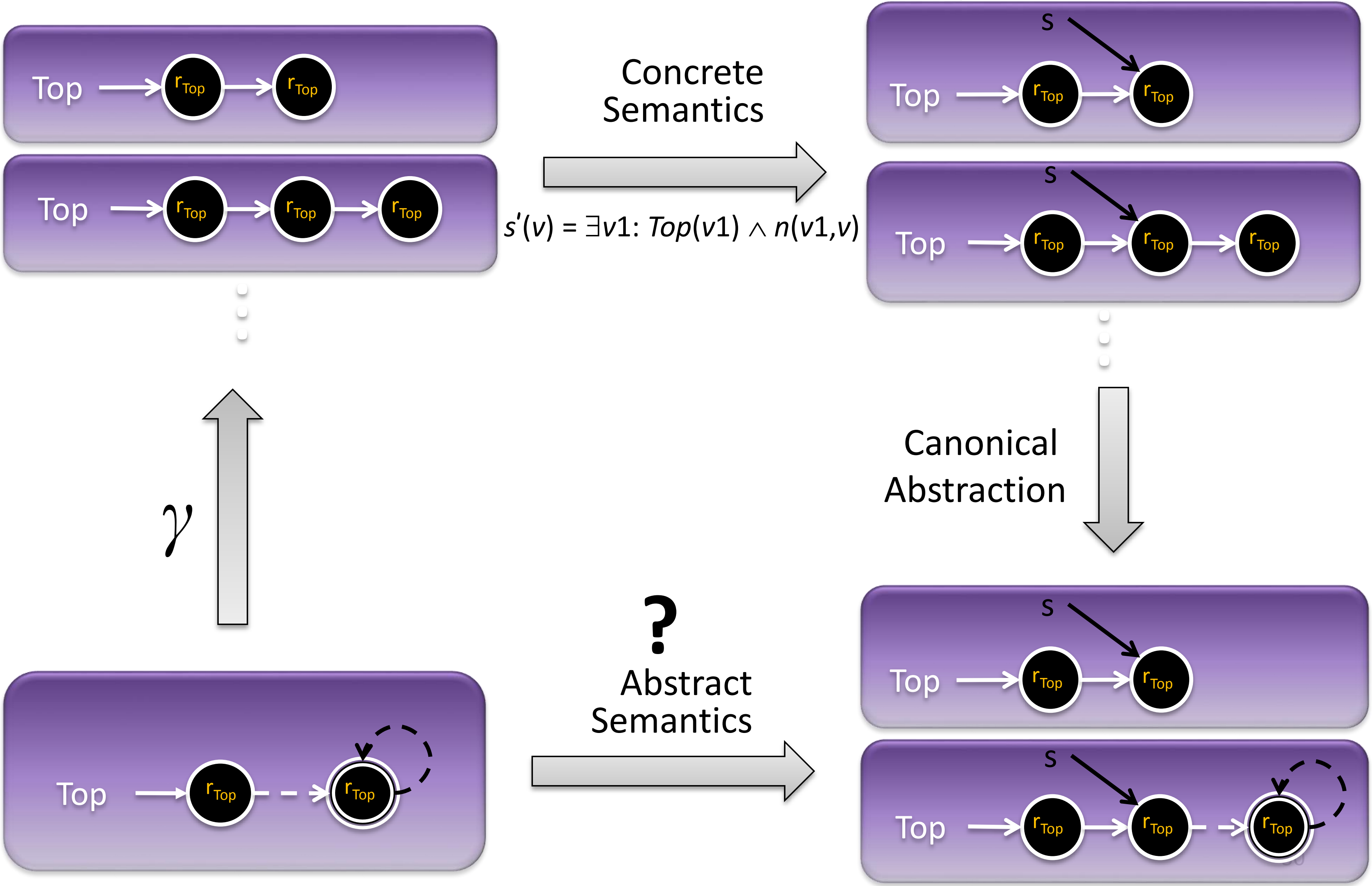


$s = Top \rightarrow n$

$[\![$`s = Top->n`$]\!]$

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

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



Concrete Semantics

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

Canonical Abstraction

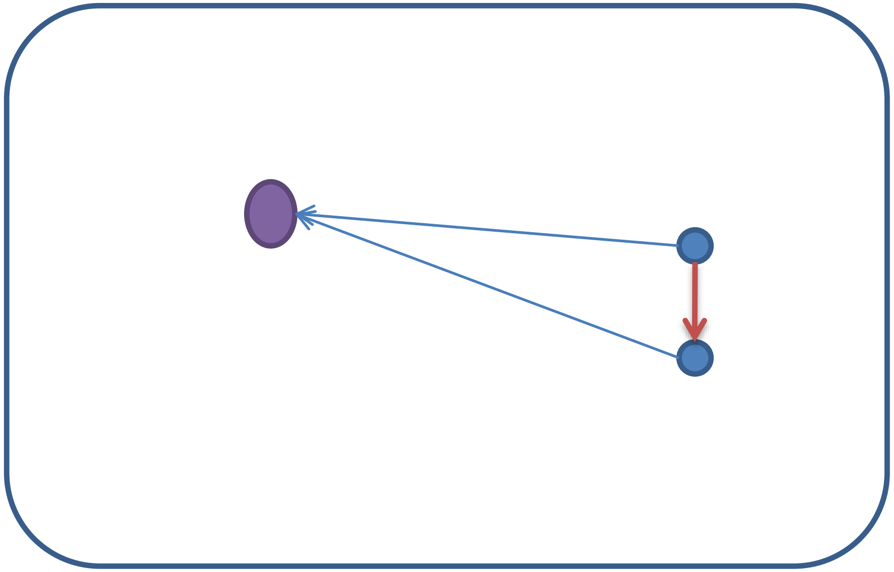
γ

?

Abstract Semantics

Semantic Reduction

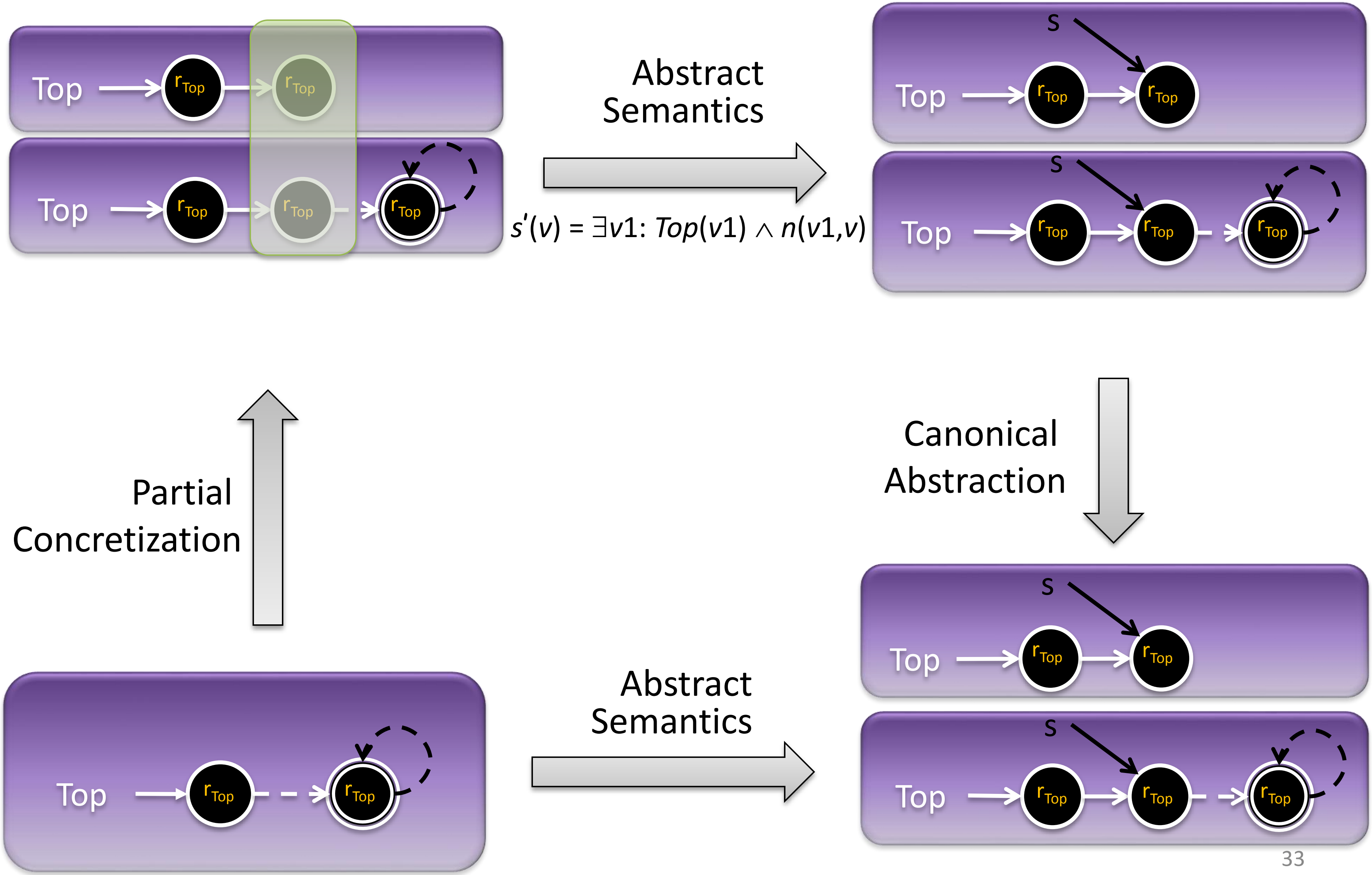
- Improve the precision of the analysis by recovering properties of the program semantics
- A Galois connection (C, α, γ, A)
- An operation op:A$\rightarrow$A is a semantic reduction when
  - $\forall l \in L_2$ op(l)$\sqsubseteq$l and
  - $\gamma(op(l)) = \gamma(l)$



The Focus Operation

- Focus: Formula$\rightarrow$($\wp$(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 ($s = Top \rightarrow n$)



Abstract Semantics

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

Canonical Abstraction

Partial Concretization

Abstract Semantics

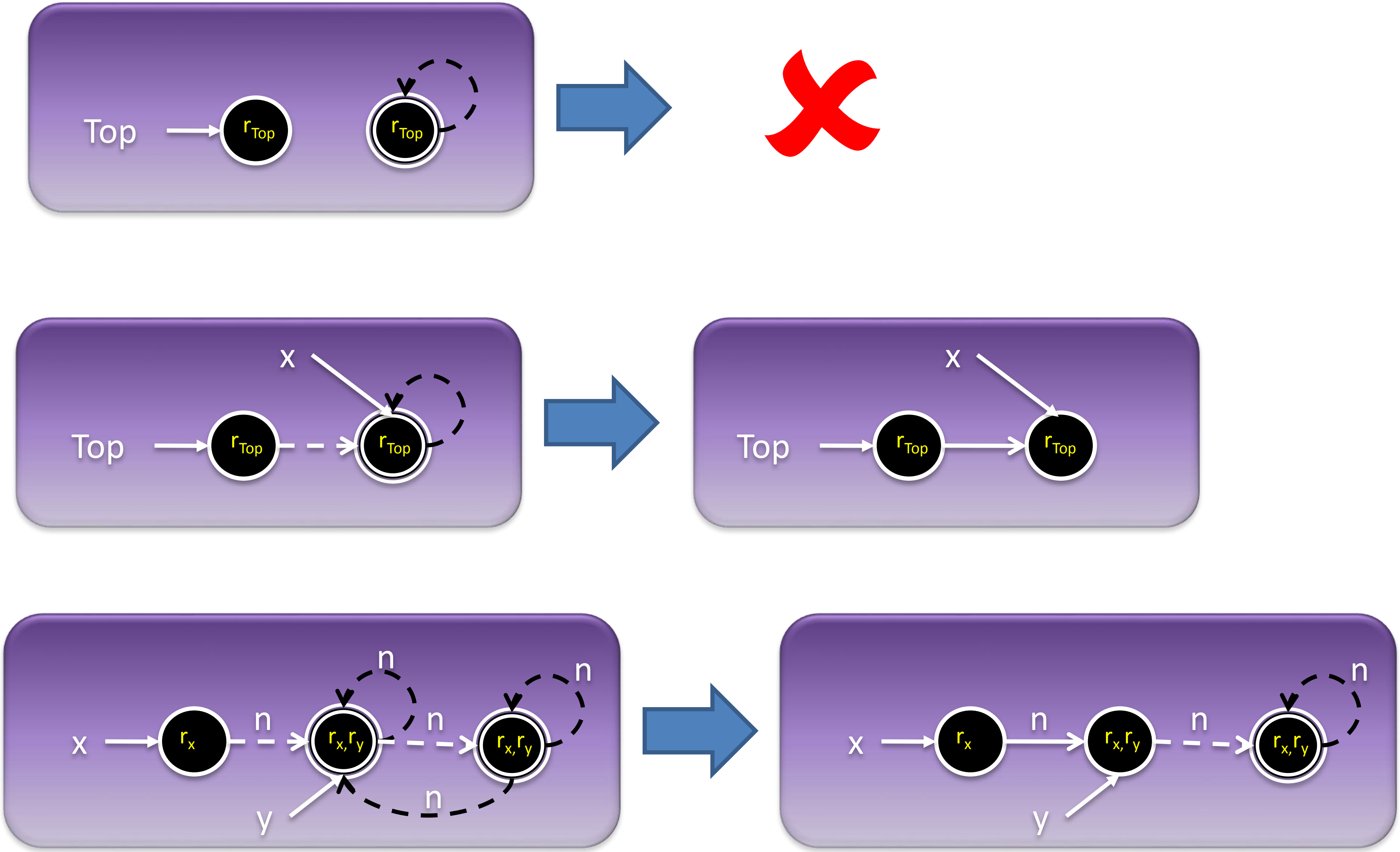
Partial Concretization

- 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

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

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

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

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

Abstract Transformers: Summary

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

Abstract Semantics

$$
SS\,[v] = \begin{cases}
\{ <\emptyset,\emptyset> \} & \text{if } v = \text{entry} \\
\bigcup_{\substack{(w,v) \in E(G), \\ w \in \text{Assignments}(G)}} \{ \text{t\_embed}(\text{coerce}(\llbracket st(w) \rrbracket_3(\text{focus}_{F(w)}(SS[w])))) \cup & \\
\bigcup_{\substack{(w,v) \in E(G), \\ w \in \text{Skip}(G)}} \{ S \mid S \in SS[w] \} \cup & \text{othrewise} \\
\bigcup_{(w,v) \in \text{True-Branches}(G)} \{ \text{t\_embed}(S) \mid S \in \text{coerce}(\llbracket st(w) \rrbracket_3(\text{focus}_{F(w)}(SS[w]))) \text{ and } S \models 3 \text{ cond}(w)\} \cup & \\
\bigcup_{(w,v) \in \text{False-Branches}(G)} \{ \text{t\_embed}(S) \mid S \in \text{coerce}(\llbracket st(w) \rrbracket_3(\text{focus}_{F(w)}(SS[w]))) \text{ and } S \models_3 \neg\text{cond}(w)\} \cup &
\end{cases}
$$

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));
```

$\exists v: x(v)$

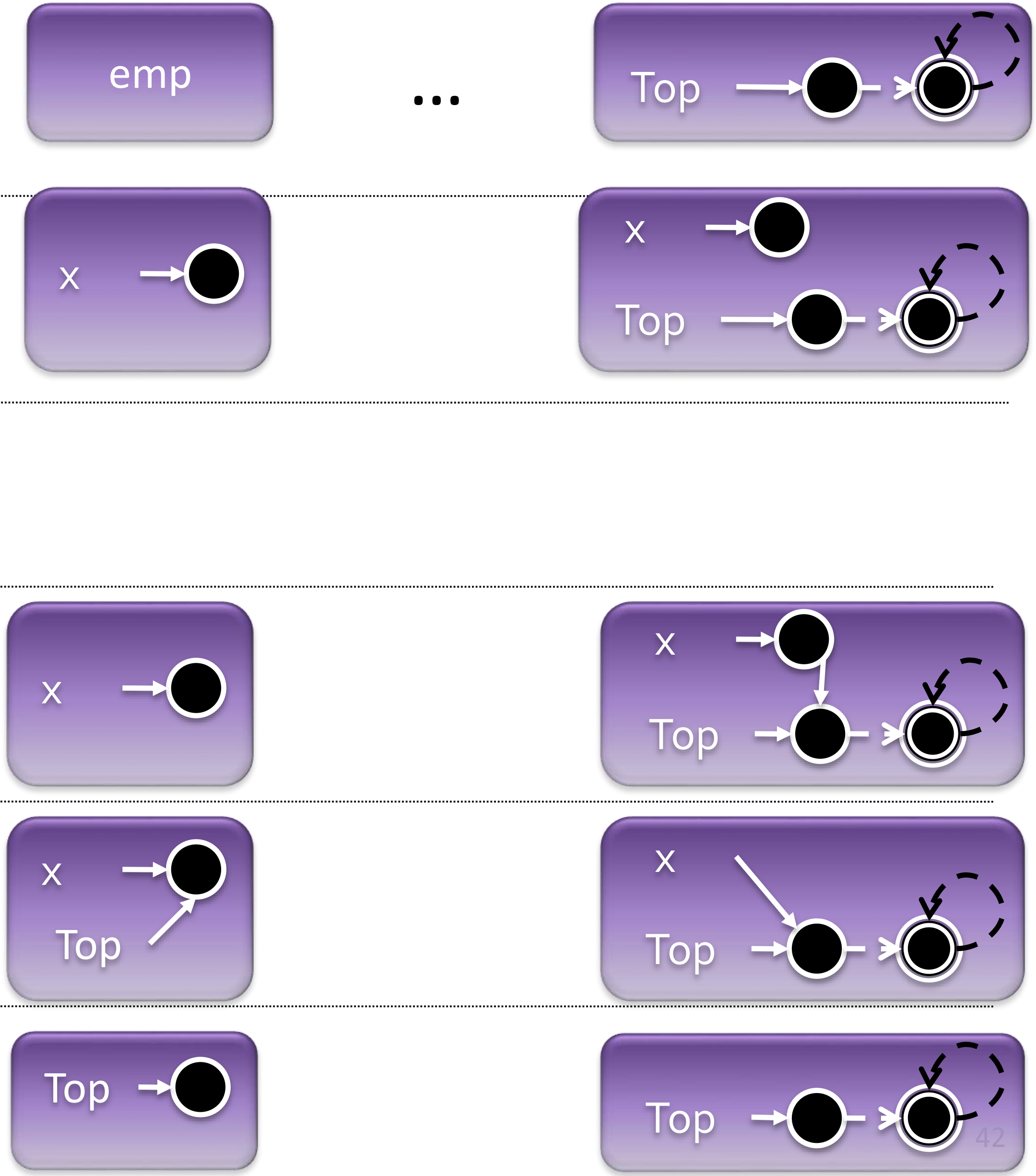
```
 x→d = v;
```

$\exists v: x(v)$

```
 x→n = Top;

 Top = x;
}
```

$\neg\exists v1,v2: n(v1, v2) \wedge Top(v2)$

$\forall v:\neg c(v)$



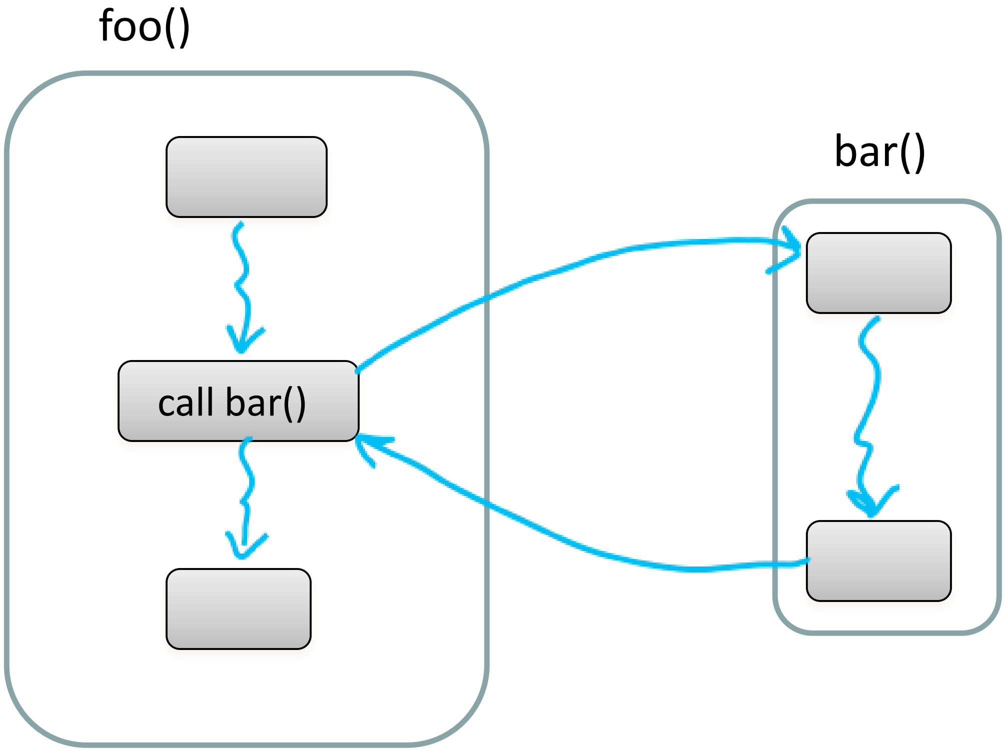
What about procedures?

Procedural program

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

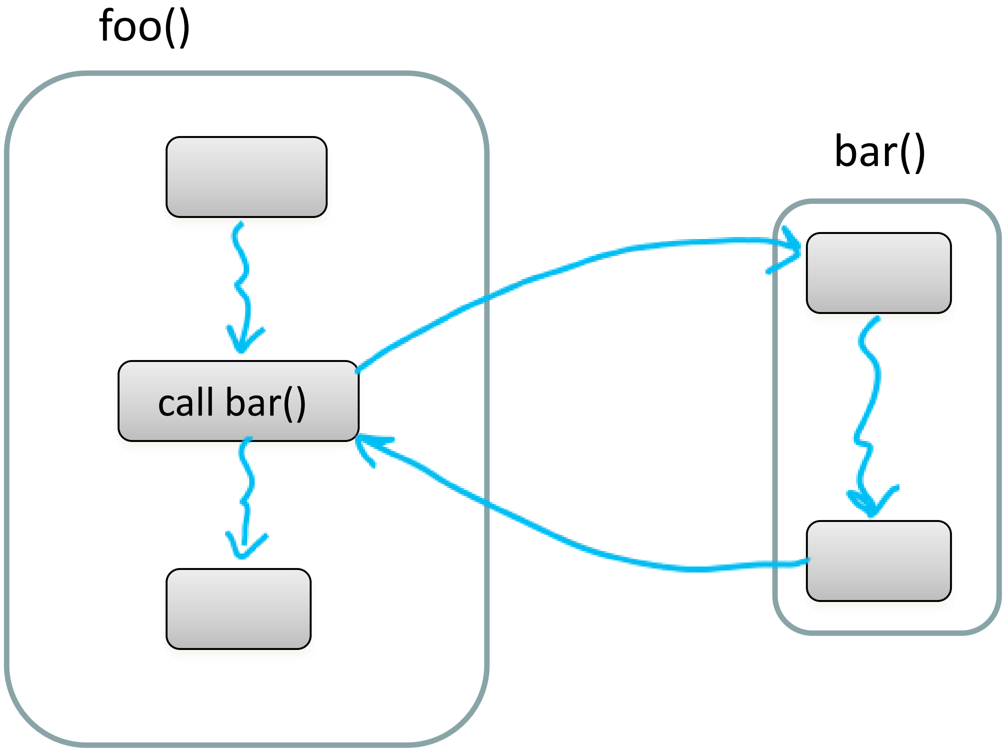
```
int p(int a) {
  return a + 1;
}
```

Effect of procedures



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

Interprocedural Analysis

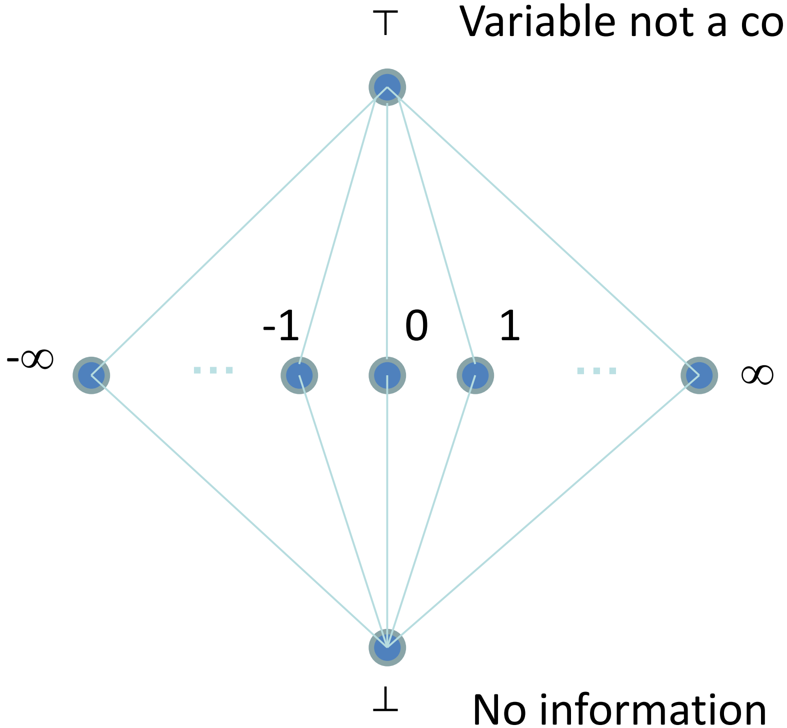


goal: compute the abstract effect of calling a procedure

Reduction to intraprocedural analysis

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

Reminder: Constant Propagation



Variable not a constant

No information

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

Reminder: Constant Propagation

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

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

Reminder: Constant Propagation

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

Procedure Inlining

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

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

Procedure Inlining

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

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

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

Procedure Inlining

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

```
p1 {                p2 {                p3{
  call p2             call p3           }
   …                   …
  call p2             call p3
}                   }
```

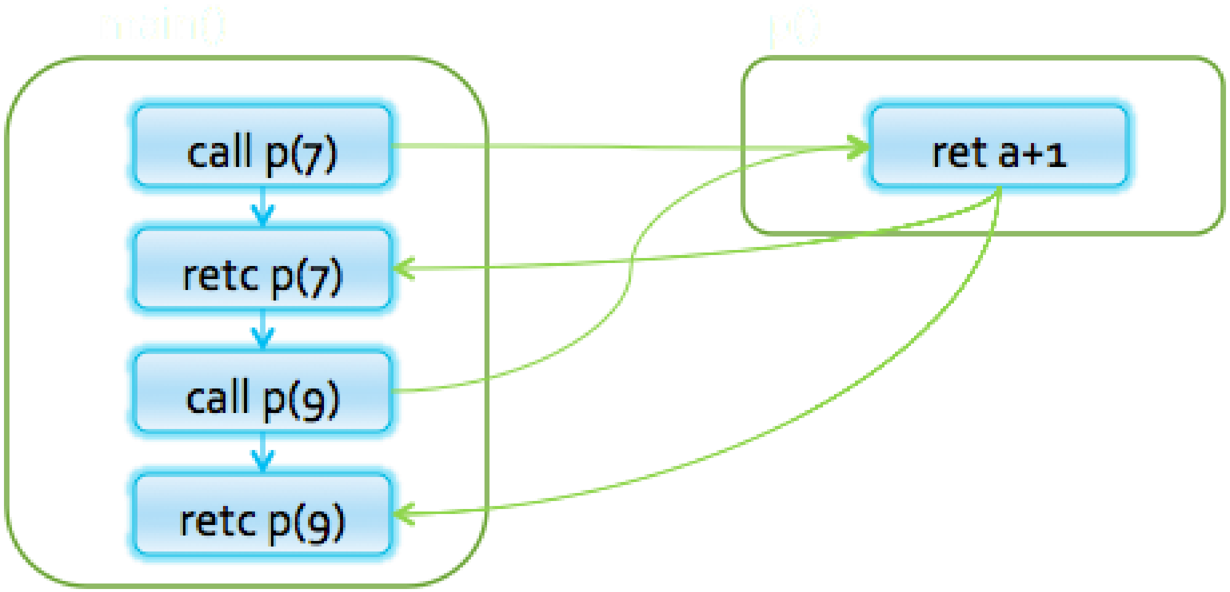
A Naive Interprocedural solution

- Treat procedure calls as gotos

Simple Example

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

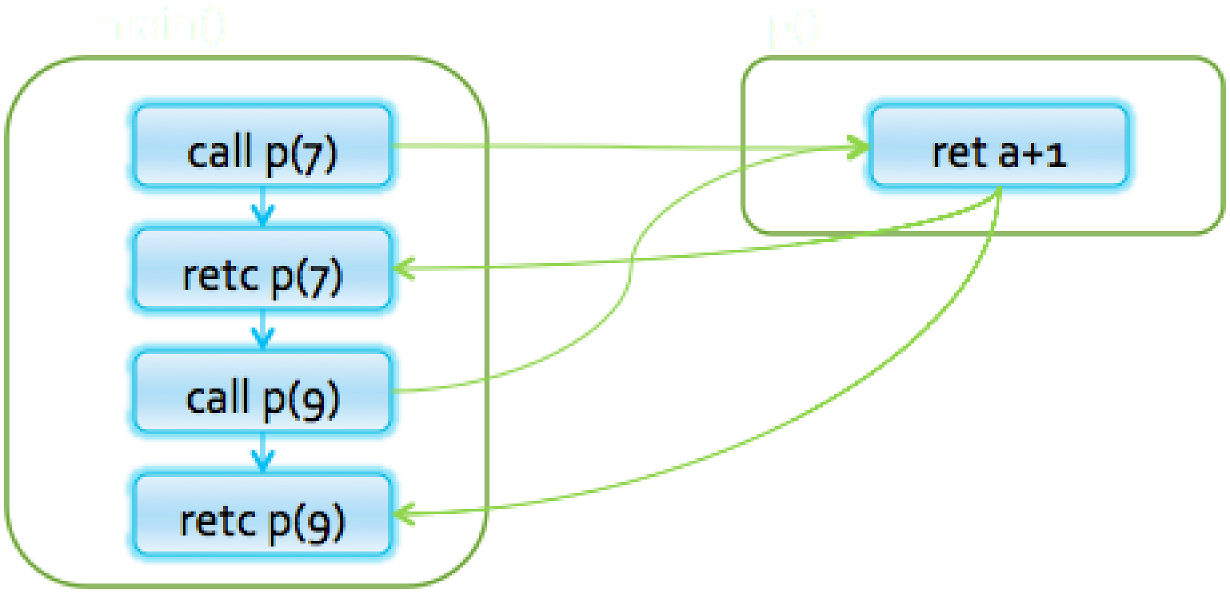
```
int p(int a) {
  return a + 1;
  }
```



Simple Example

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

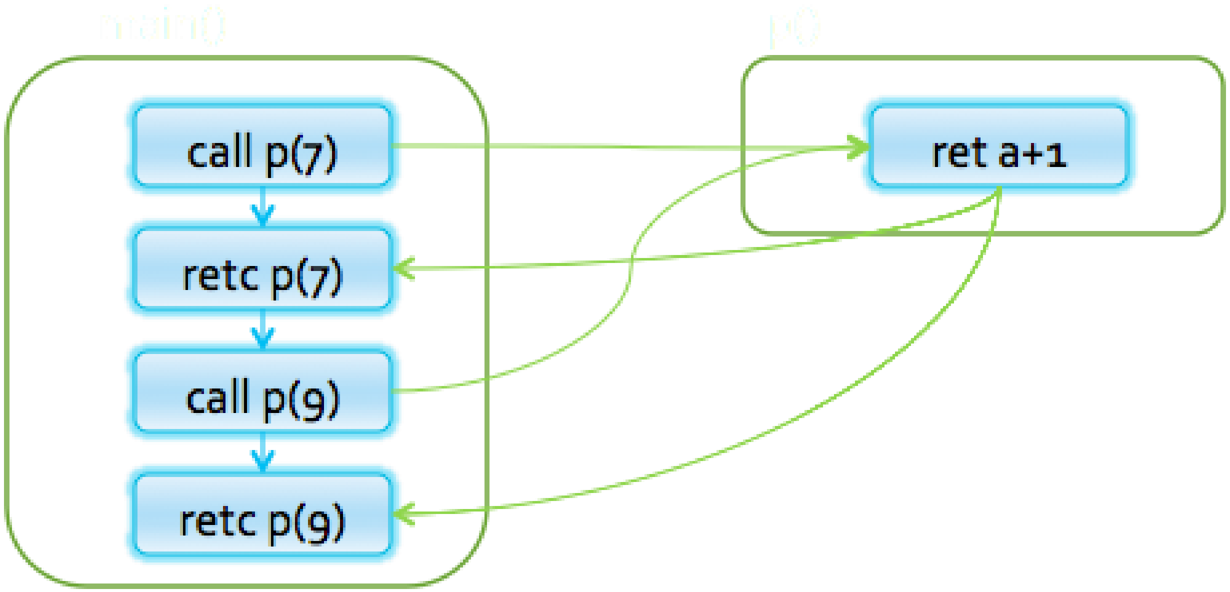
```
➡ int p(int a) {
     [a ↦7]
     return a + 1;
  }
```



Simple Example

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

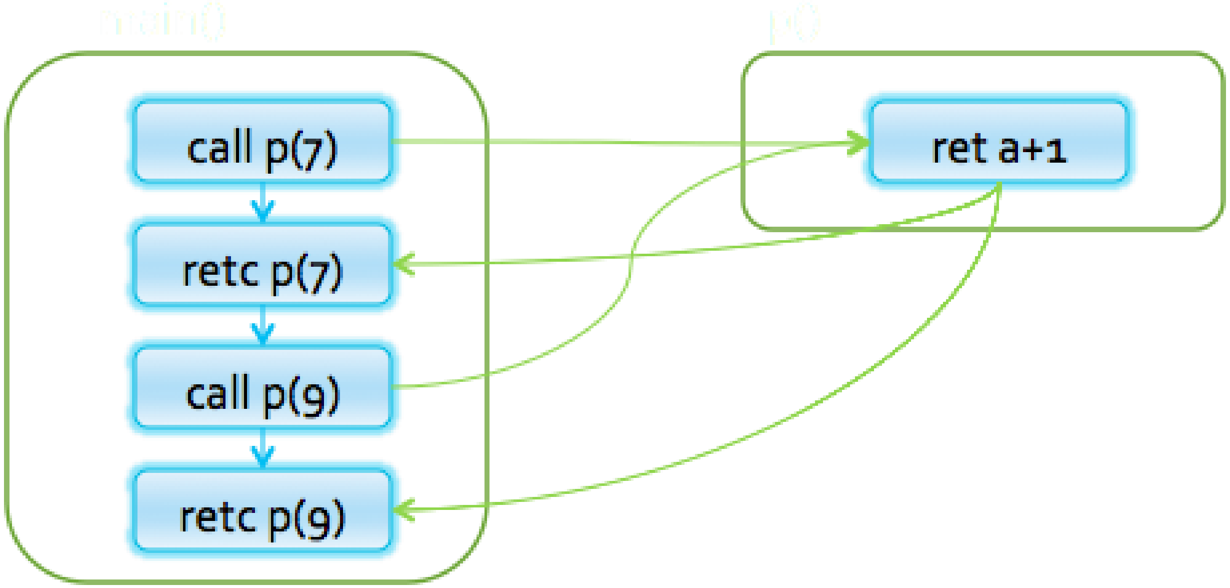
```
int p(int a) {
  [a ↦7]
➡return a + 1;
 [a ↦7, $$ ↦8]

}
```



Simple Example

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

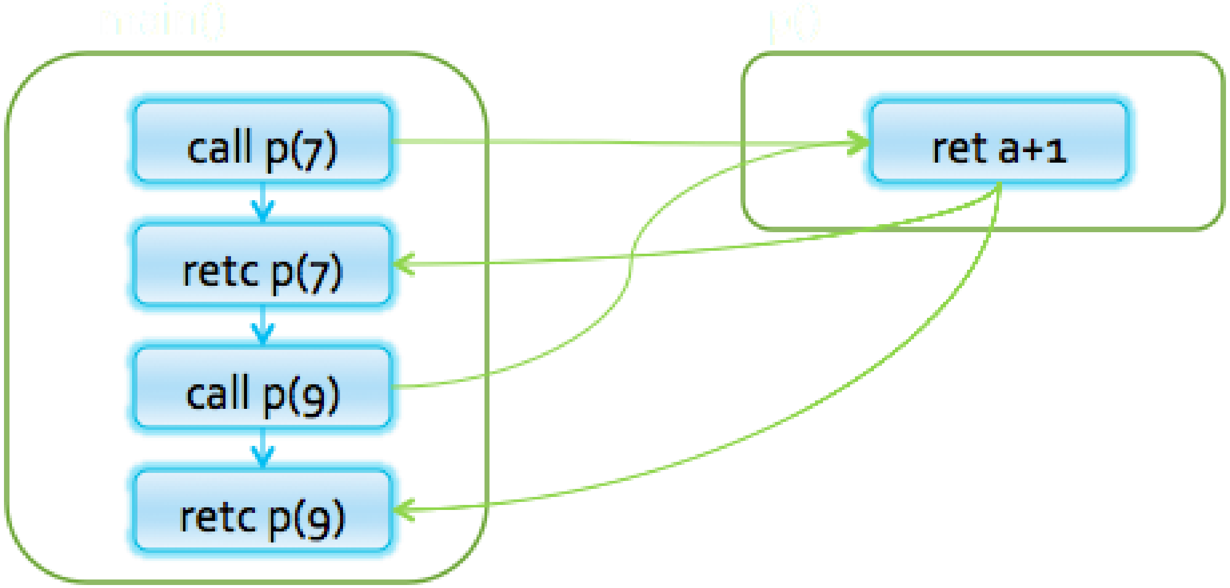
```
int p(int a) {
   [a ↦7]
    return a + 1;
  [a ↦7, $$ ↦8]

}
```



Simple Example

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

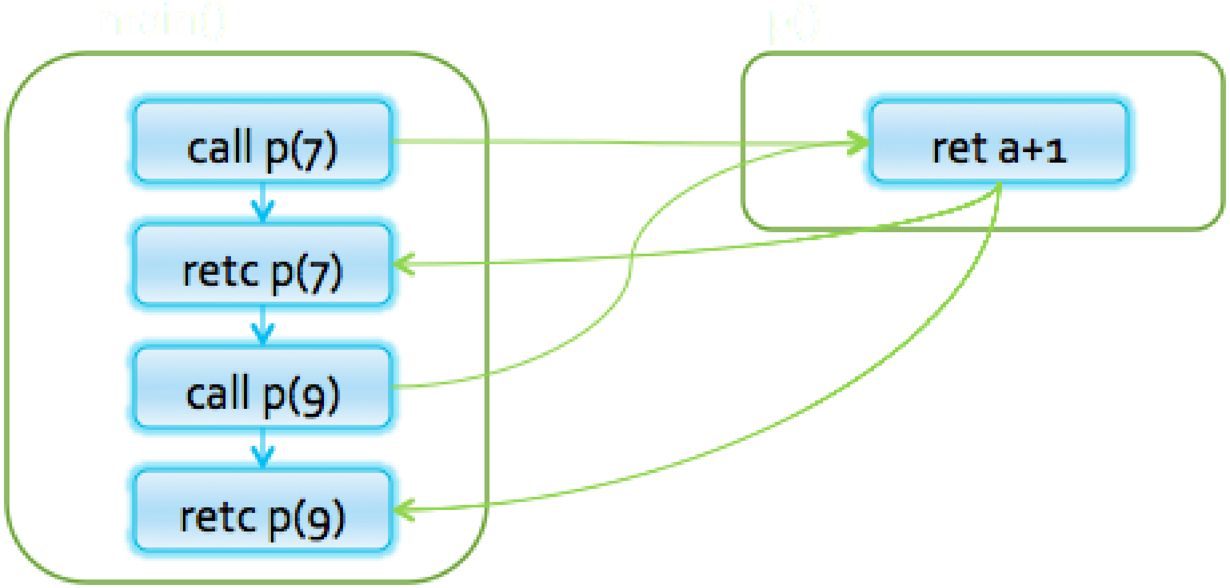
```
int p(int a) {
  [a ↦7]
   return a + 1;
  [a ↦7, $$ ↦8]


}
```



Simple Example

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

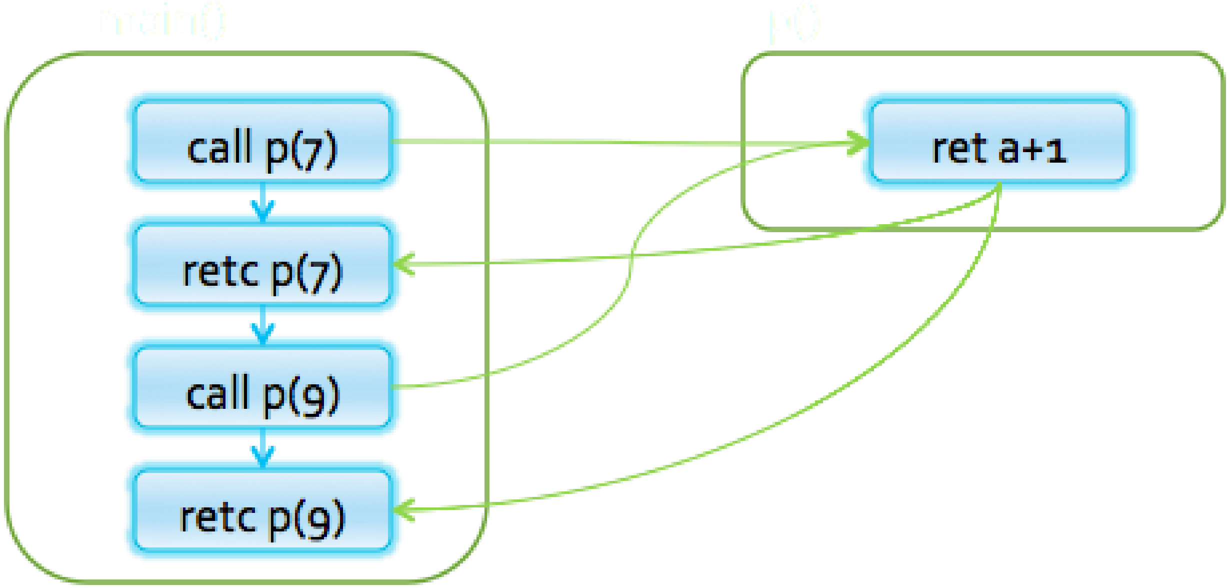
```
int p(int a) {
   [a ↦7]  [a ↦9]
    return a + 1;
   [a ↦7, $$ ↦8]


}
```



Simple Example

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

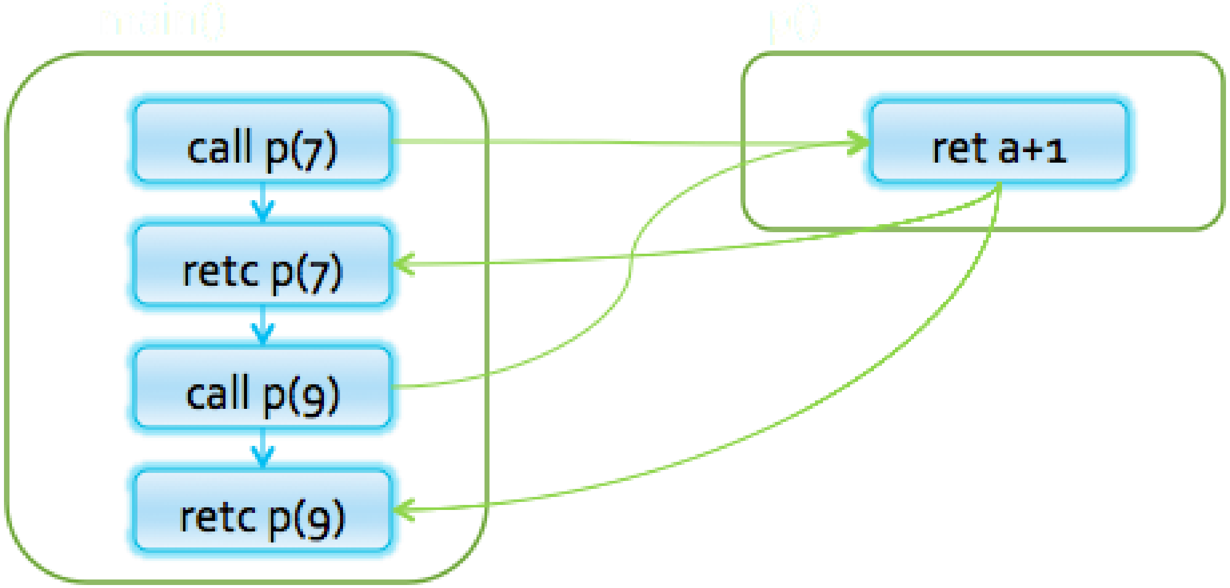
```
➡int p(int a) {
   [a ↦⊤]
    return a + 1;
   [a ↦7, $$ ↦8]


}
```



Simple Example

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

}
```

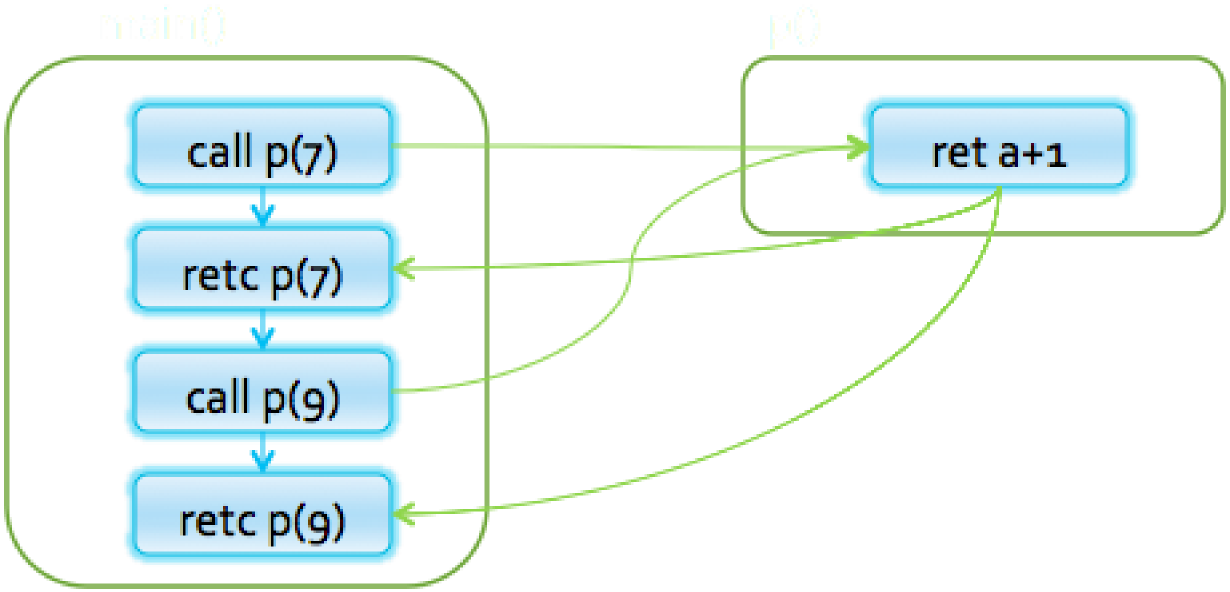
```
int p(int a) {
  [a ↦⊤]
➡return a + 1;
  [a ↦⊤, $$ ↦⊤]

}
```



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 ↦⊤]
}
```

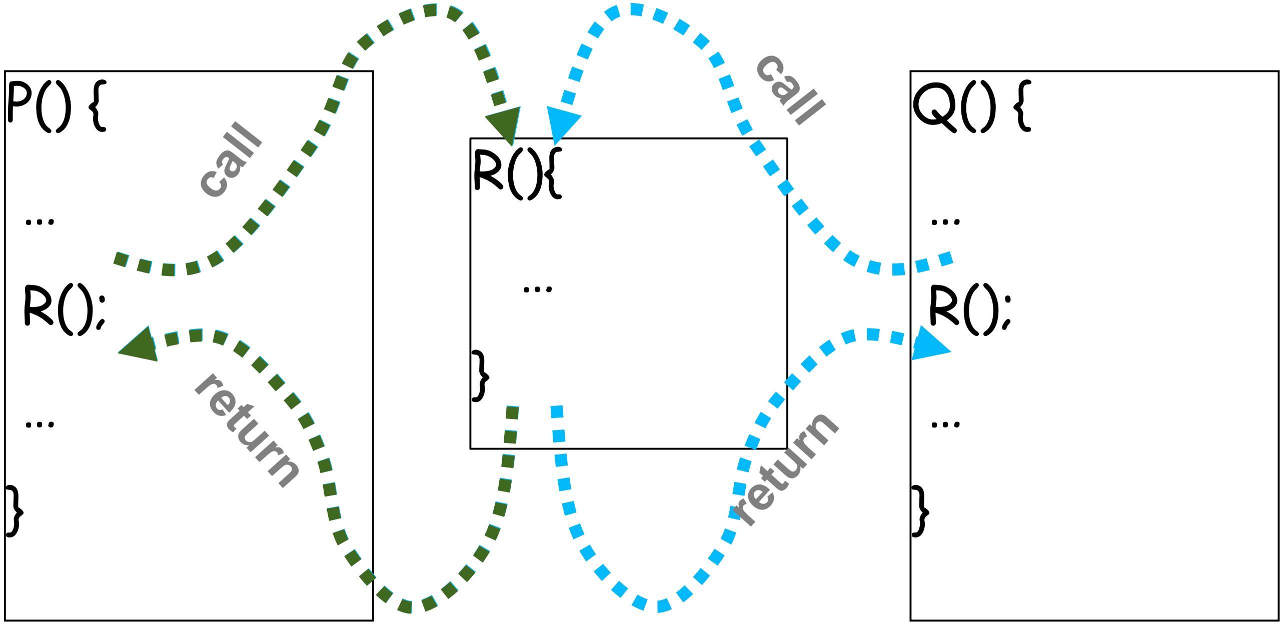
```
int p(int a) {
  [a ↦⊤]
   return a + 1;
  [a ↦⊤, $$ ↦⊤]
}
```

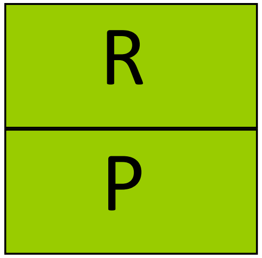
Procedure inlining

```
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

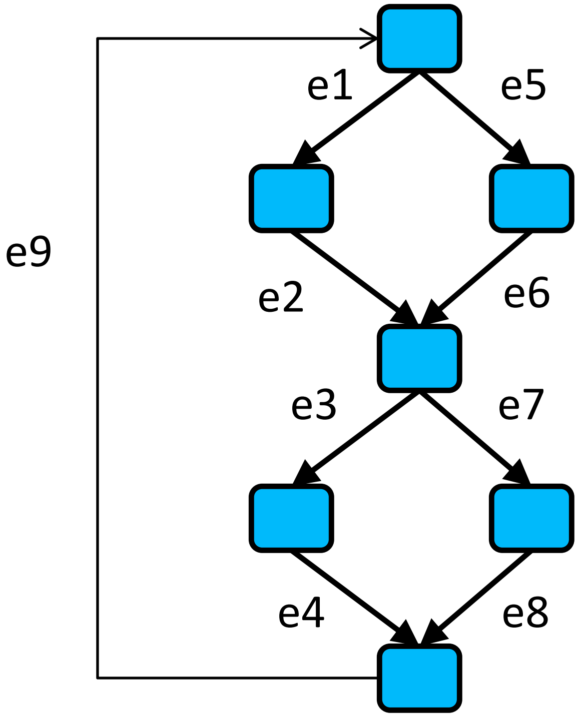
- ✓ Procedure Inlining
- ✓ The naive approach
- Valid paths
- The callstring approach
- The Functional Approach
- IFDS: Interprocedural Analysis via Graph Reachability
- IDE: Beyond graph reachability

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

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

The lfp computation approximates JOP

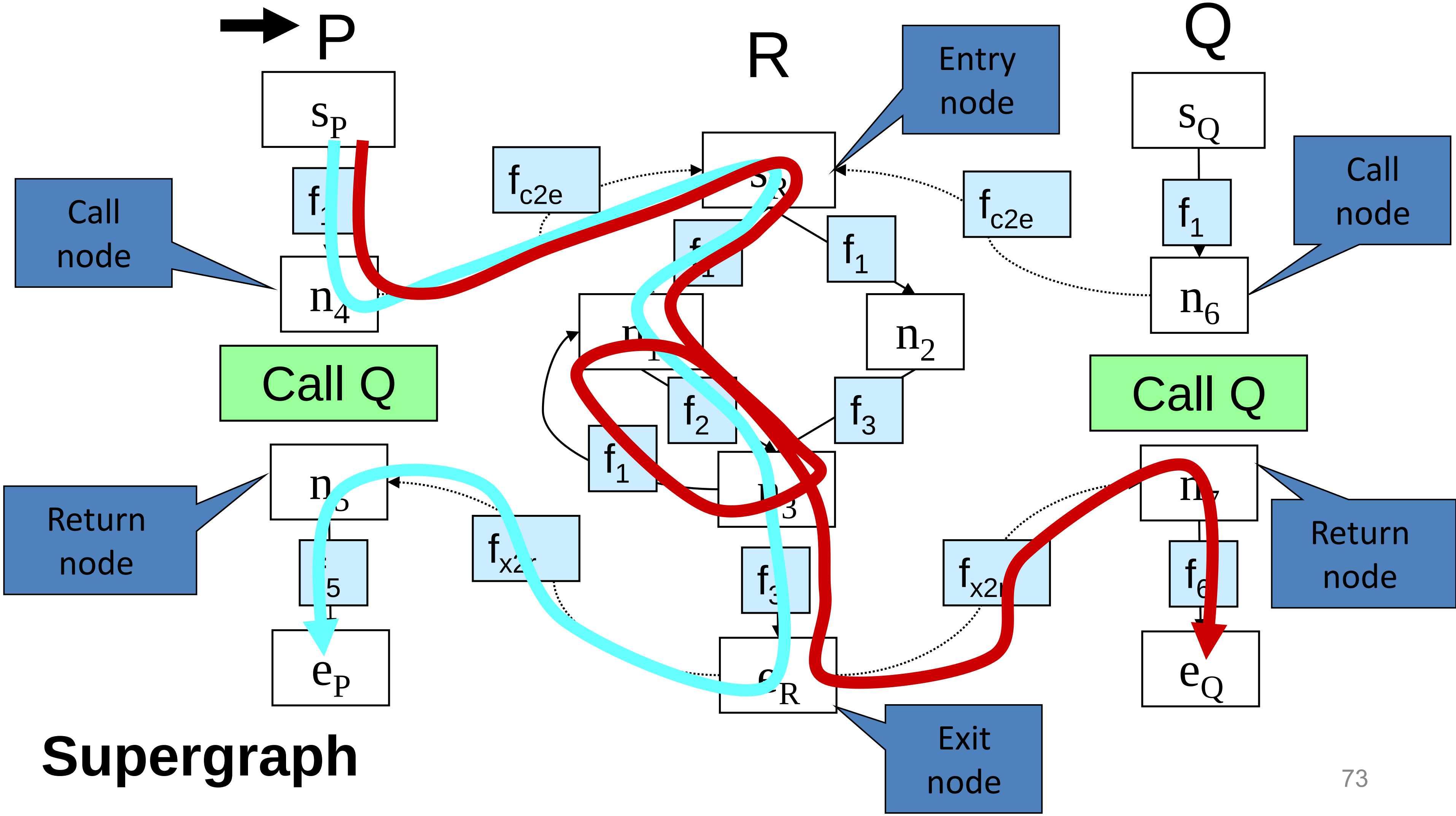
- JOP[v] $= \sqcup\{f\,[e_1, e_2, \ldots, e_n](\iota) \mid$
  $[e_1, e_2, \ldots, e_n] \in \text{paths}(v)\}$
- LFP[v] $= \sqcup\{f\,[e](\text{LFP}[v']) \mid e = (v', v)\}$

  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!

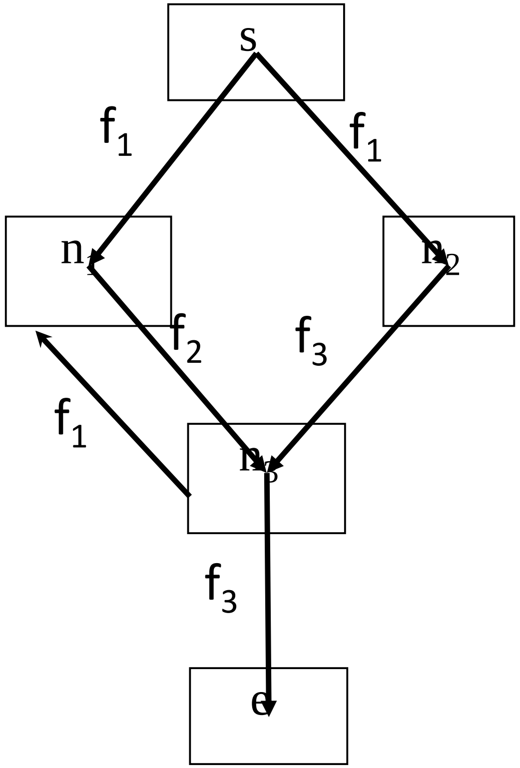
Interprocedural analysis



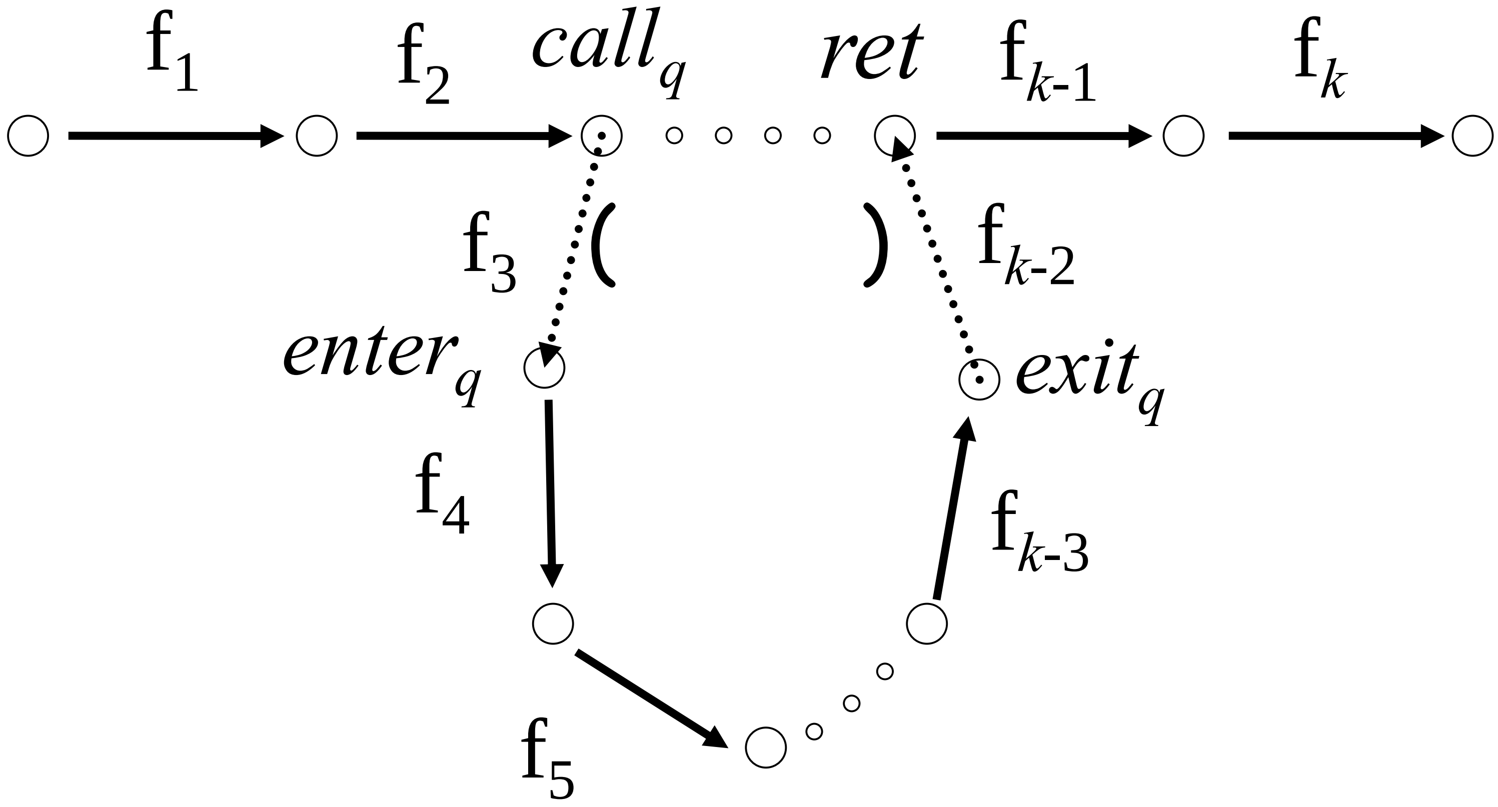
Supergraph

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

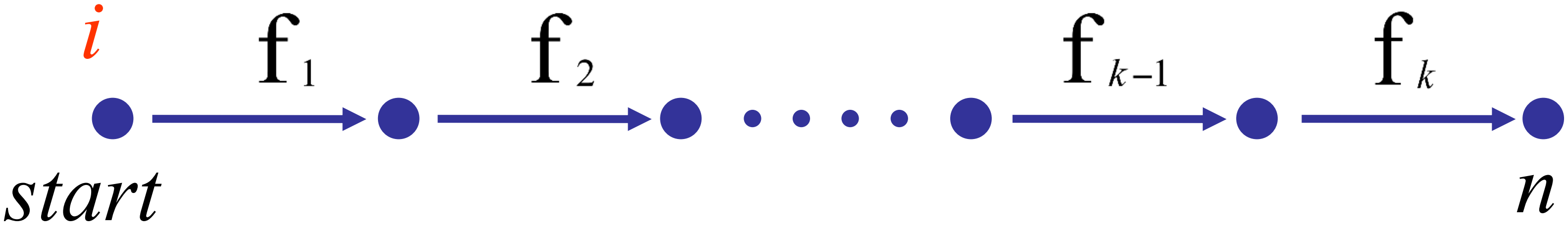


- IVP: all paths with matching calls and returns
  - And prefixes

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)



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

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

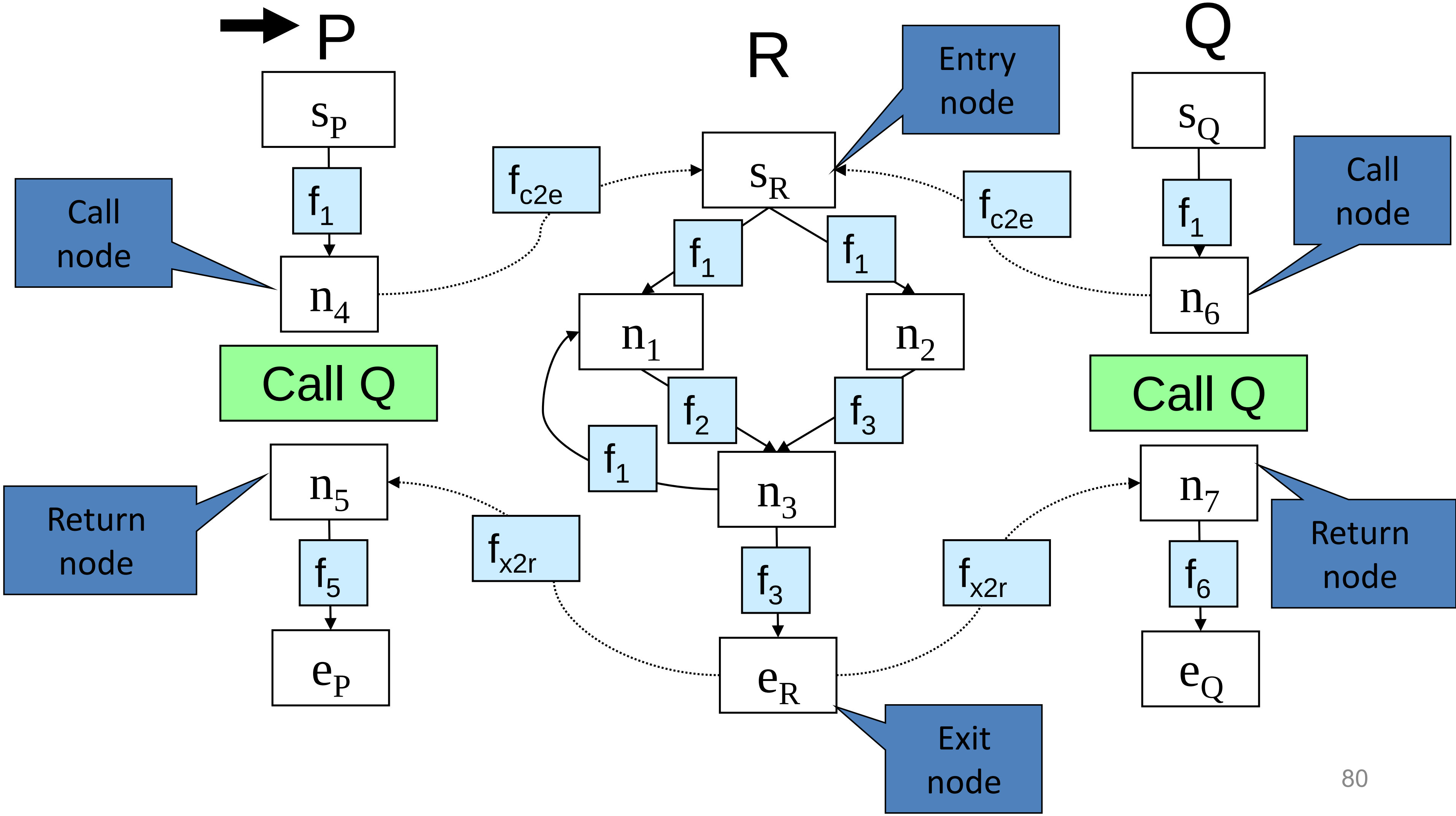
- JVP[n] $= \sqcup\{[[e_1, e_2, \ldots, e]](\iota) \mid (e_1, e_2, \ldots, e) \in \text{vpaths}(n)\}$

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

Simple Example

```
void main() {
   int x ;
   c1: x =  p(7);
   c2: x =  p(9) ;


}
```

```
int p(int a) {


   return a + 1;


}
```

Simple Example

```
void main() {
   int x ;
   c1: x =  p(7);
   c2: x =  p(9) ;

}
```

```
➡ int p(int a) {
   c1: [a ↦7]
   return a + 1;

}
```

Simple Example

```
void main() {
   int x ;
   c1: x =  p(7);
   c2: x =  p(9) ;

}
```

```
int p(int a) {
   c1: [a ↦7]
➡  return a + 1;
   c1:[a ↦7, $$ ↦8]
}
```

Simple Example

```
void main() {
   int x ;
   c1: x =  p(7);      ⬅
   ∑: 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 ↦⊤]

}
```

```
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
- 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 ;
   $[a \mapsto a_0\text{-}1,\ x \mapsto x_0]$
    p (a);
   $[a \mapsto a_0\text{-}1,\ x \mapsto -2a_0+7]$
    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 ↦ -9]

}
```

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

```
int p(int a) {

 [a ↦7, x ↦0]          [a ↦⊤, x ↦0]
 if (…)  {

   [a ↦7, x ↦0]        [a ↦⊤, x ↦0]

   a = a -1 ;

   [a ↦6, x ↦0]        [a ↦⊤, x ↦0]

   p (a);
    [a ↦6, x ↦-7]      [a ↦⊤, x ↦⊤]

   a = a + 1;
   [a ↦7, x ↦-7]     [a ↦⊤, x ↦⊤]

 }
 [a ↦7, x ↦0]      [a ↦⊤, x ↦⊤]

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

}
```

[a ↦6, x ↦0]

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:
    - $f_{start,start} = (\top,\top)$; otherwise $(\bot,\bot)$
    - $S[start, \top] = \top$

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

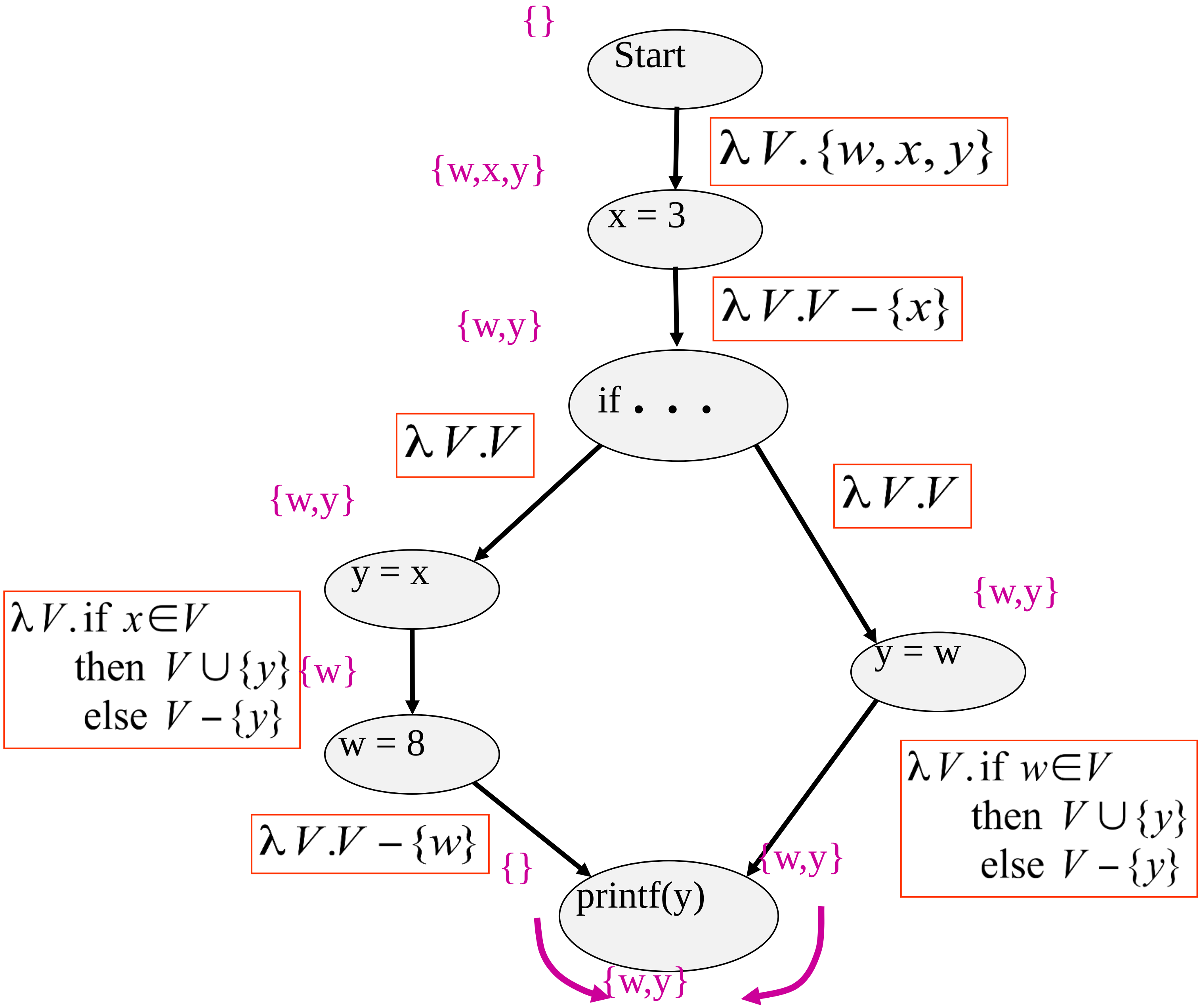
CFL-Graph reachability

- 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



{}

Start

$\lambda V.\{w, x, y\}$

{w,x,y}

x = 3

$\lambda V.V - \{x\}$

{w,y}

if . . .

$\lambda V.V$

{w,y}

$\lambda V.V$

y = x

{w,y}

$\lambda V.\text{if } x \in V$
$\quad \text{then } V \cup \{y\}$
$\quad \text{else } V - \{y\}$

{w}

y = w

w = 8

$\lambda V.\text{if } w \in V$
$\quad \text{then } V \cup \{y\}$
$\quad \text{else } V - \{y\}$

$\lambda V.V - \{w\}$

{}

{w,y}

printf(y)

{w,y}

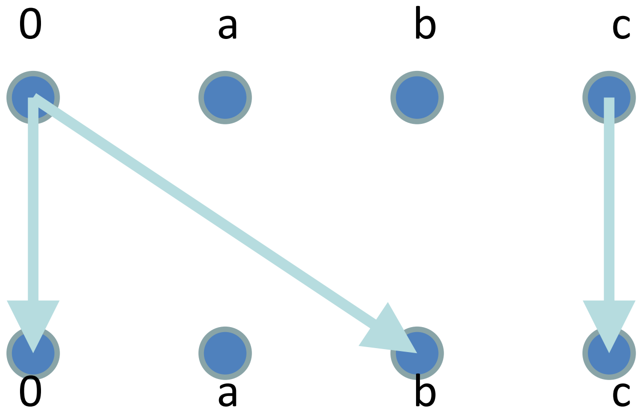
IFDS Problems

- 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}



Efficiently Representing Functions

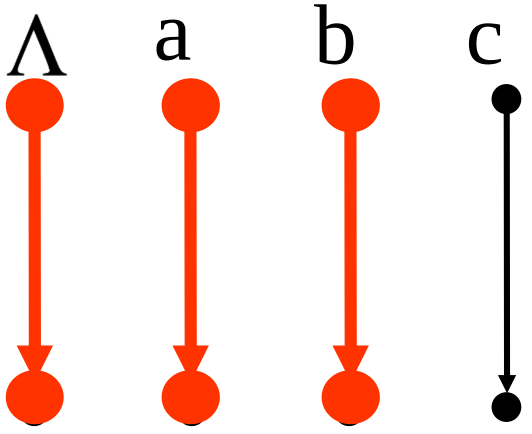
- Let $f:2^D \rightarrow 2^D$ be a distributive function
- Then:
  - $f(X) = f(\emptyset) \cup (\bigcup \{ f(\{z\}) \mid z \in X \})$
  - $f(X) = f(\emptyset) \cup (\bigcup \{ f(\{z\}) \setminus f(\emptyset) \mid z \in X \})$

Representing Dataflow Functions

Identity Function

$$f = \lambda V.V$$

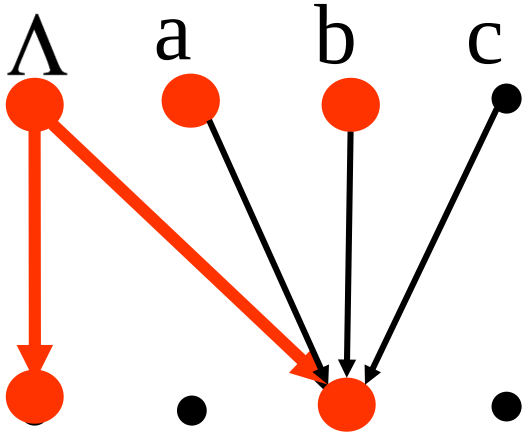
$$\boxed{f(\{a,b\}) = \{a,b\}}$$



Constant Function

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

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

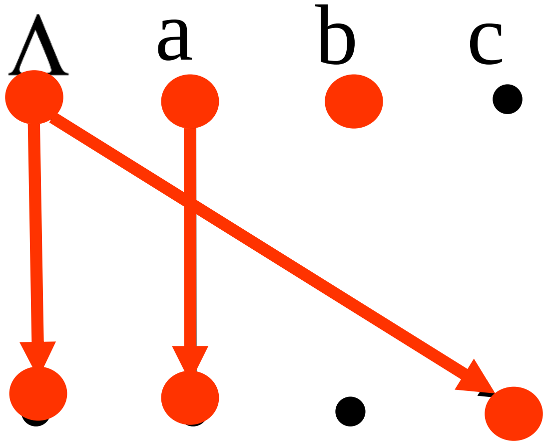


Representing Dataflow Functions

"Gen/Kill" Function

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

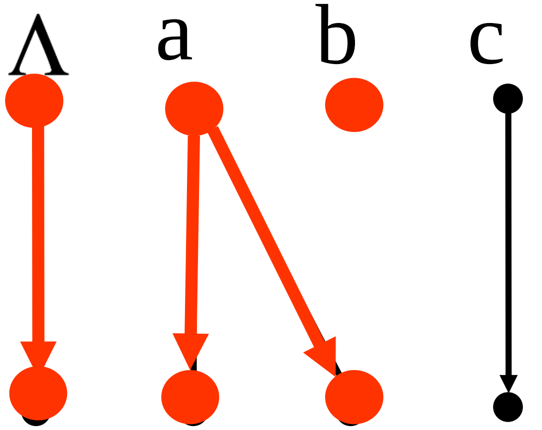
$$\boxed{f(\{a,b\}) = \{a,c\}}$$

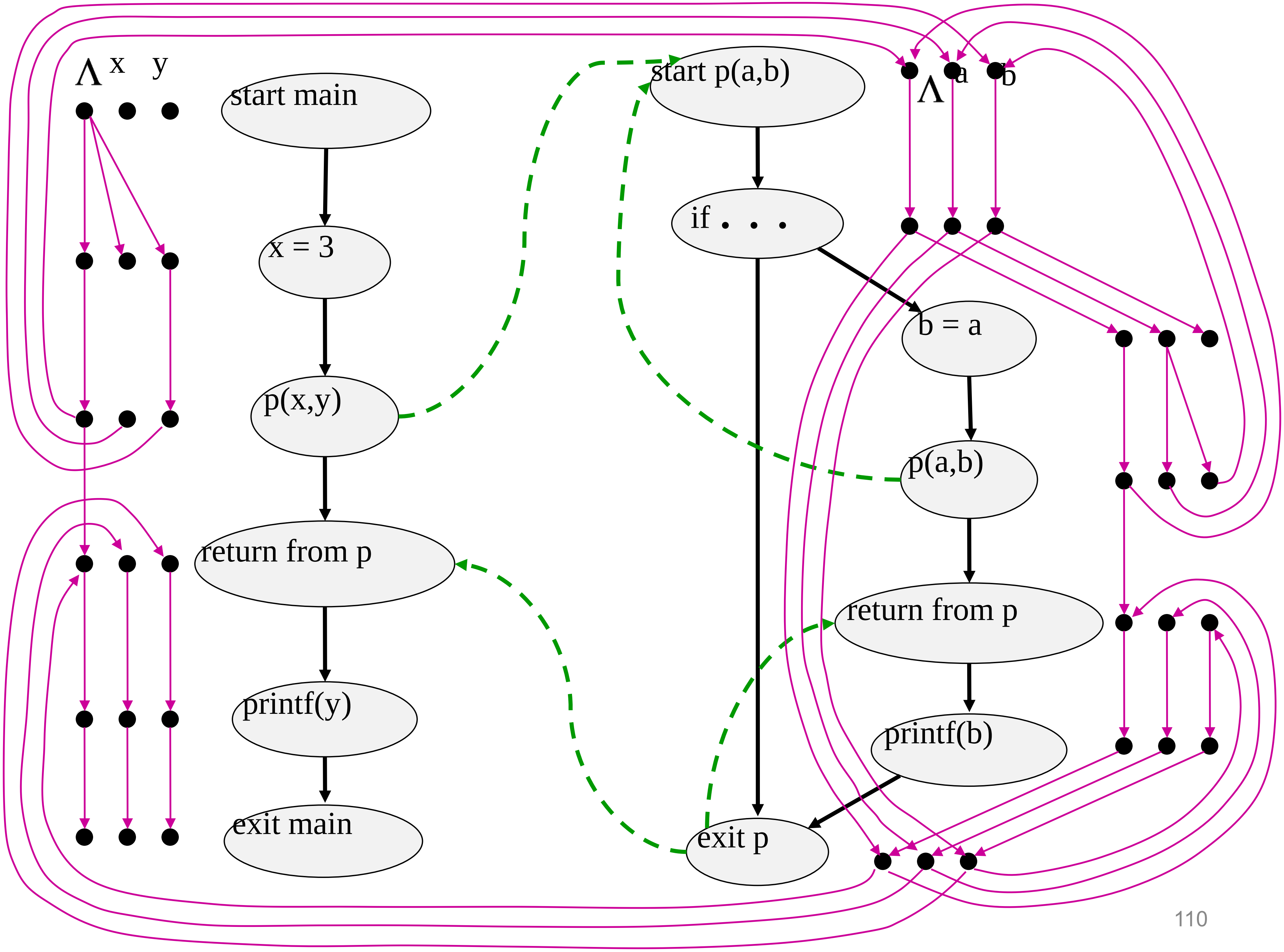


Non-"Gen/Kill" Function

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

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

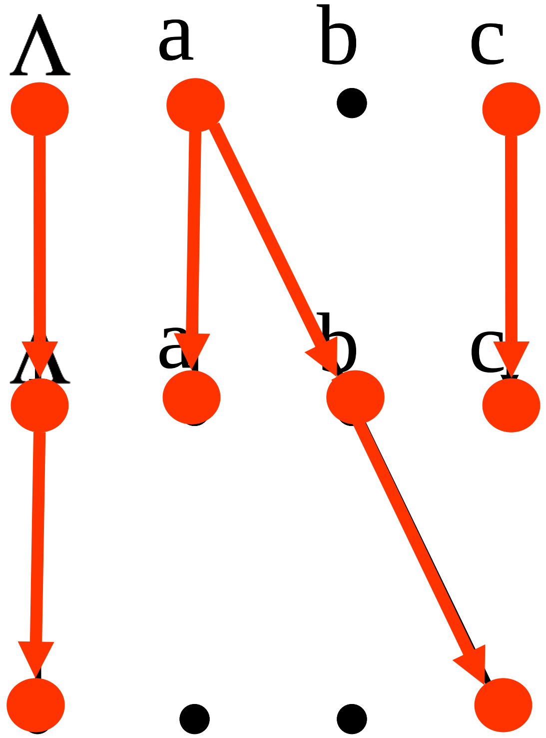



Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
Λ a b
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p

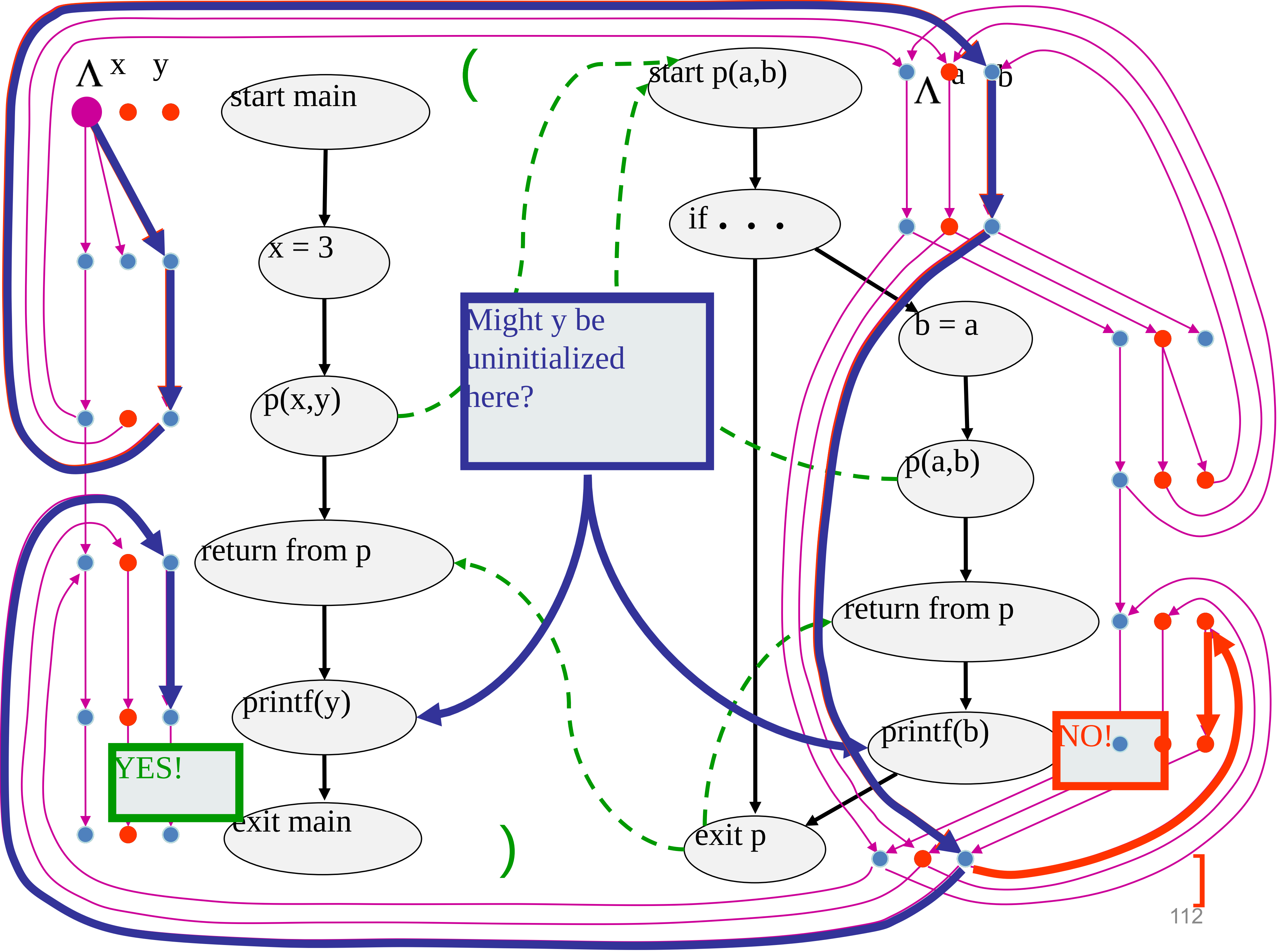
Composing Dataflow Functions

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

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



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



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

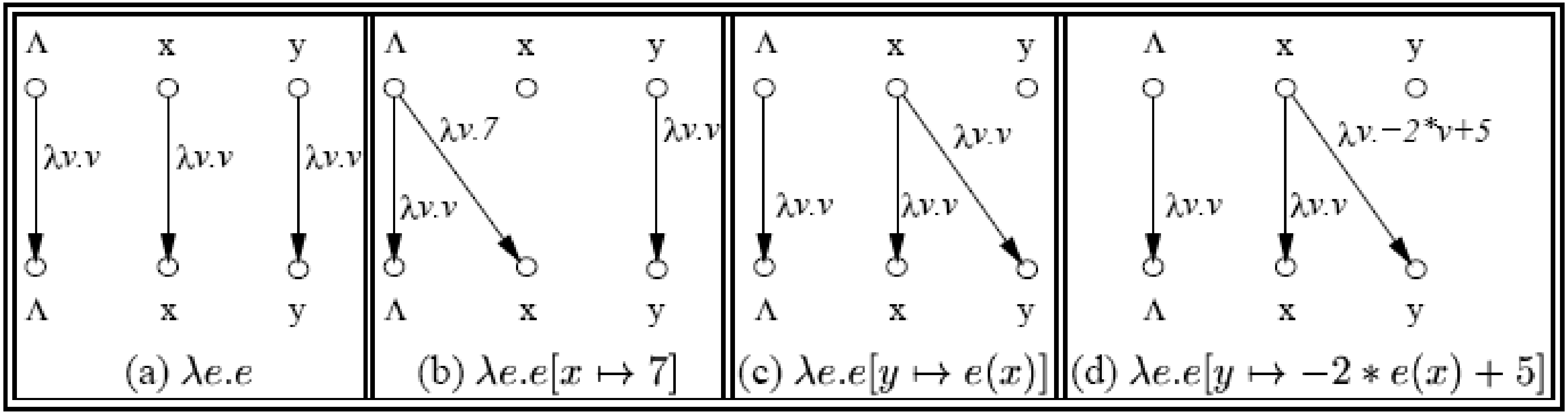
- Consider the constant propagation lattice
- The value of every variable y at the program exit can be represented by:

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

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

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

Linear constant propagation

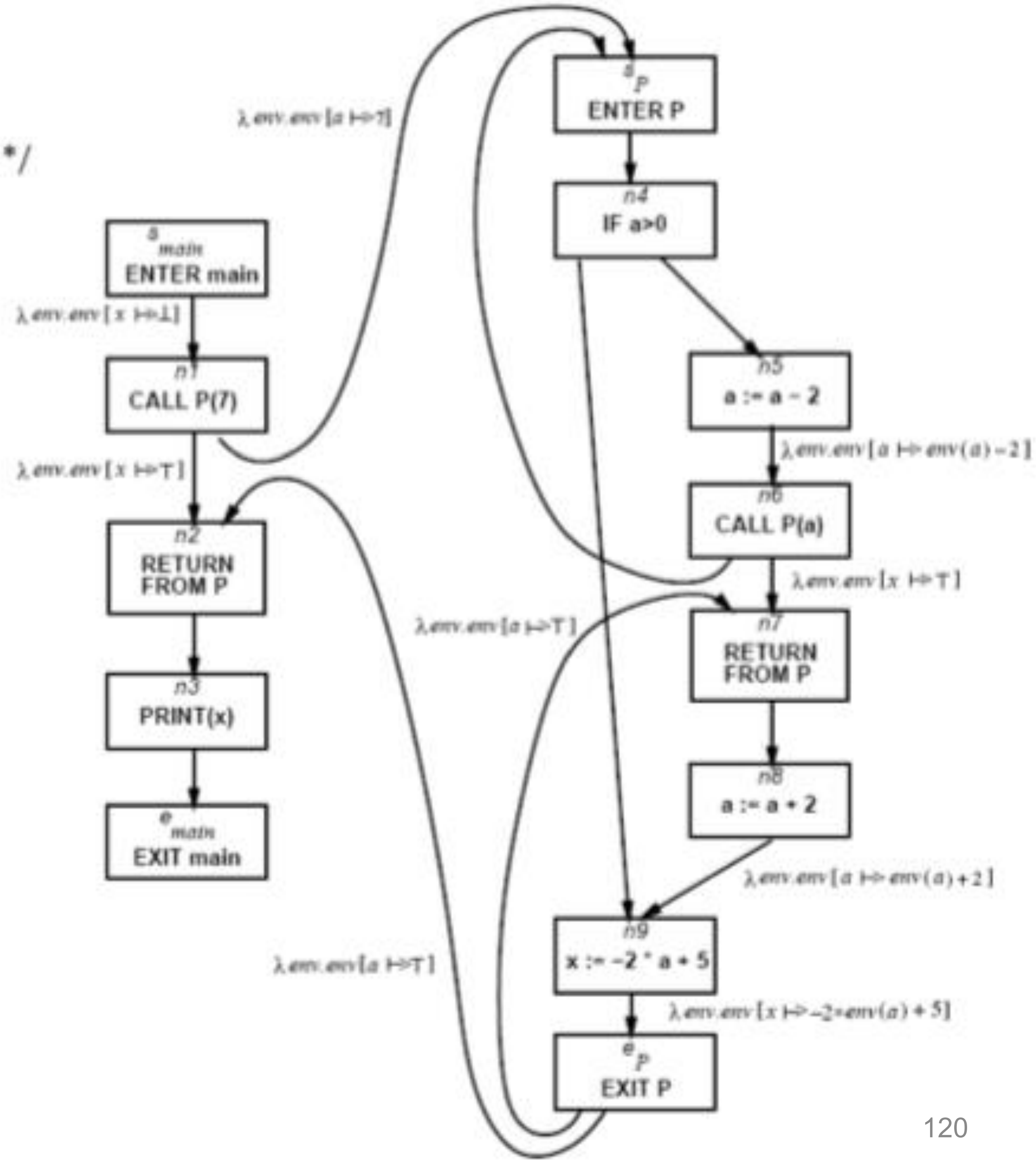


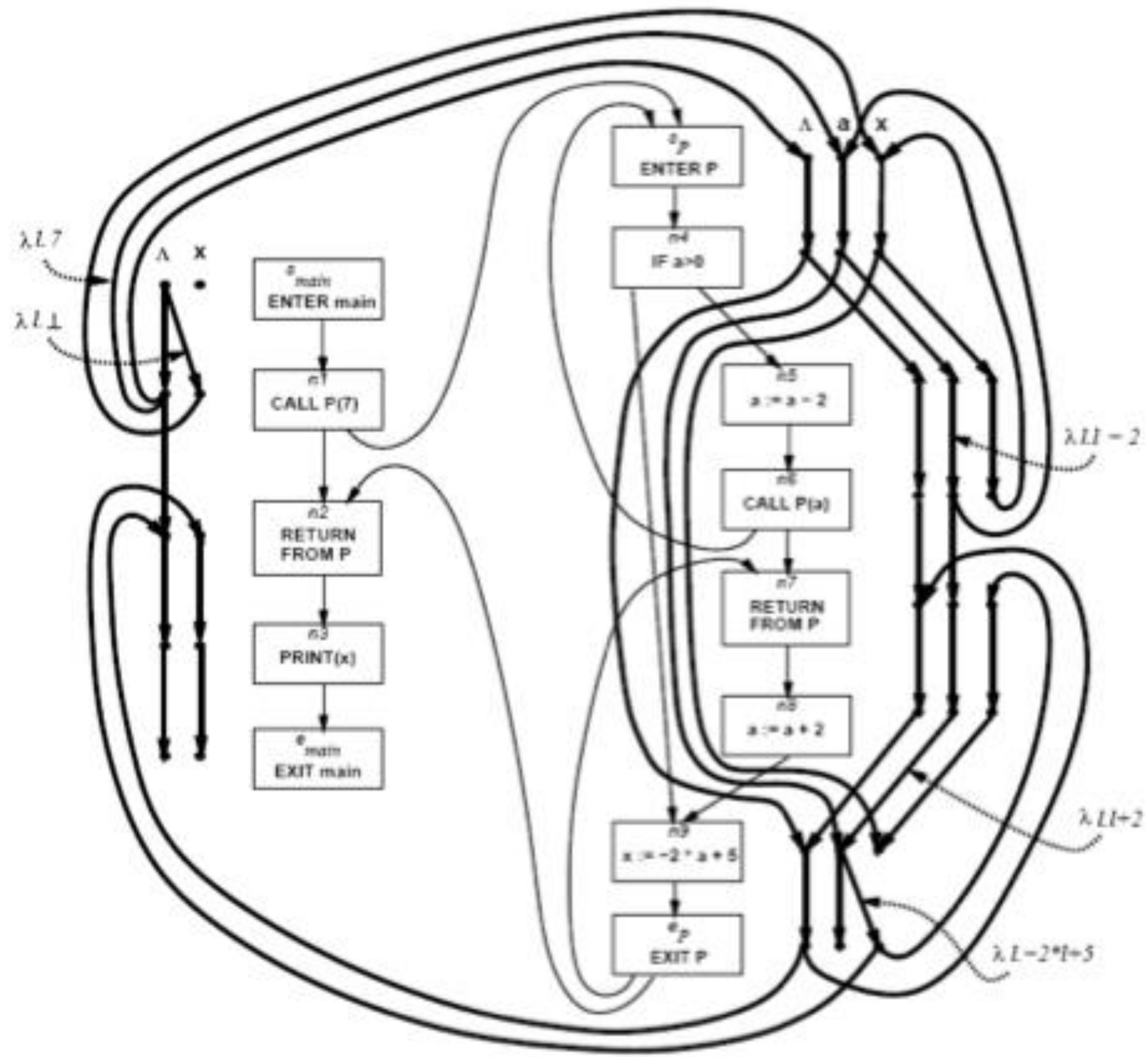
Point-wise representation of environment transformers

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
```




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*

A Semantics for Procedure Local Heaps and its Abstractions

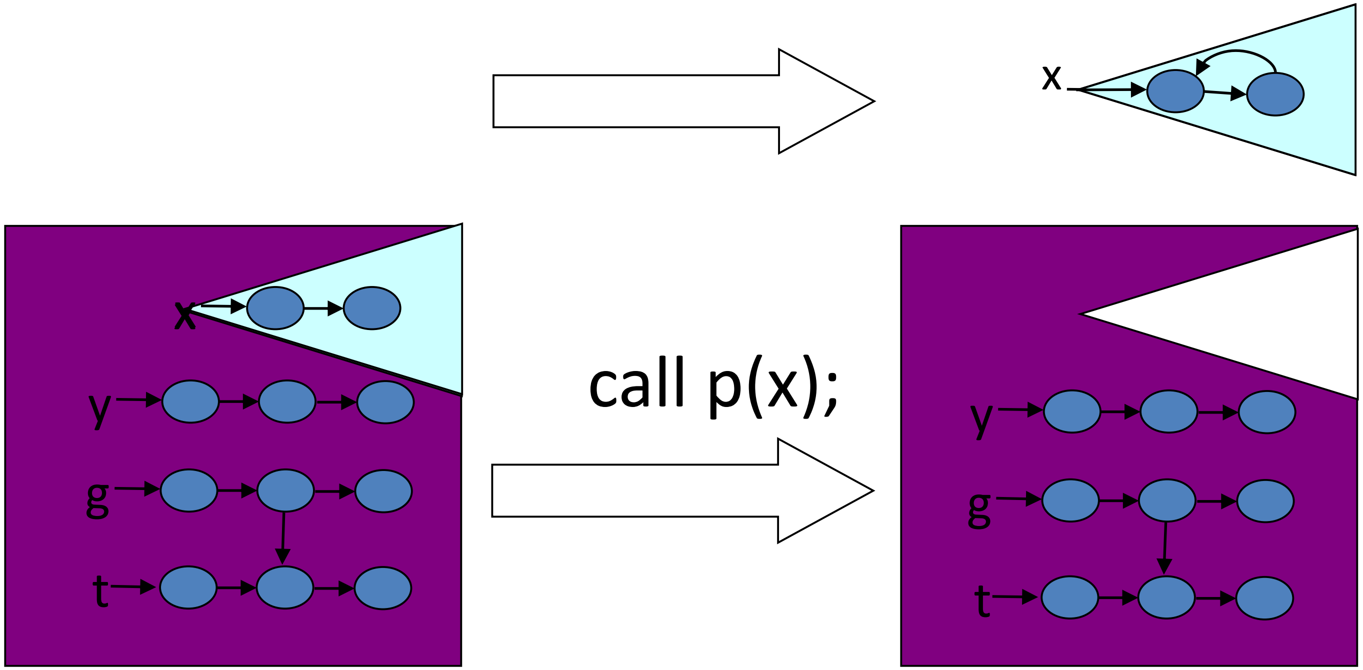
Noam Rinetzky Tel Aviv University
Jörg Bauer Universität des Saarlandes
Thomas Reps University of Wisconsin
Mooly Sagiv Tel Aviv University
Reinhard Wilhelm Universität des Saarlandes

Motivation

- Interprocedural shape analysis
  - Conservative static pointer analysis
  - Heap intensive programs
    - Imperative programs with procedures
    - Recursive data structures
- Challenge
  - Destructive update
  - Localized effect of procedures

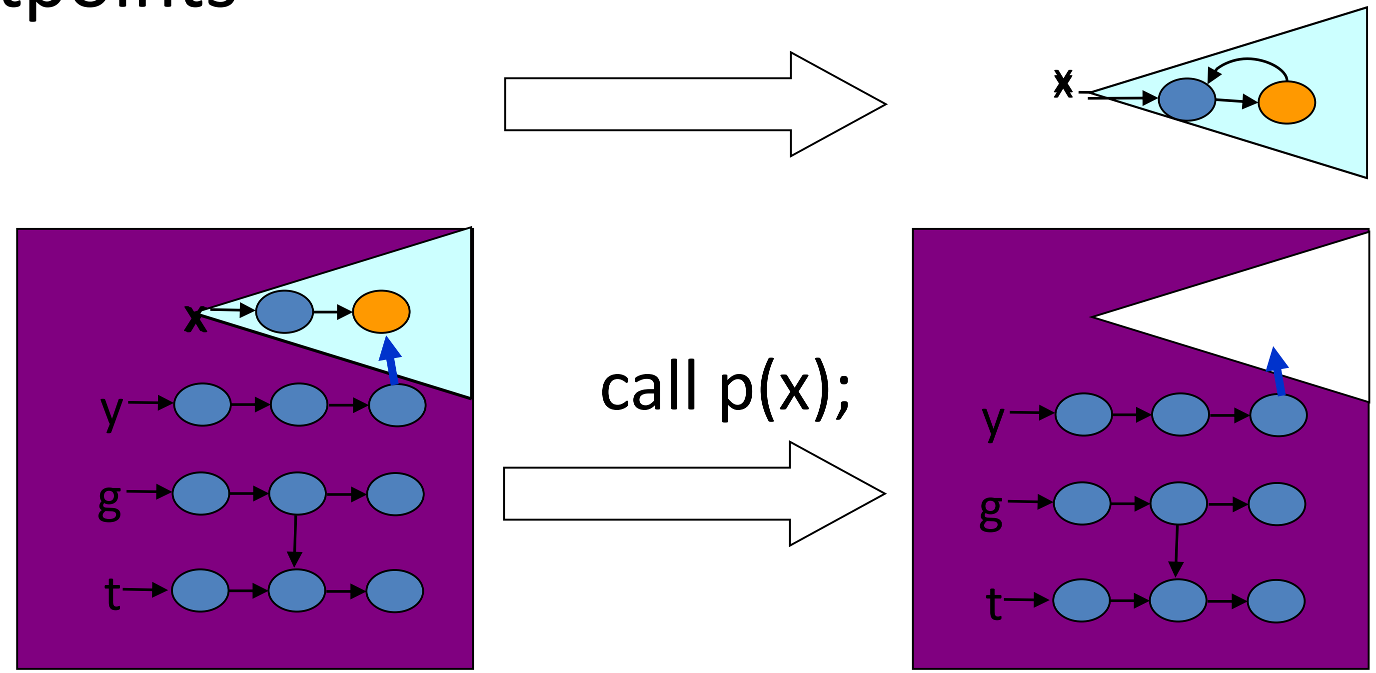
Main idea

- Local heaps



Main idea

- Local heaps
- Cutpoints



call p(x);

Main Results

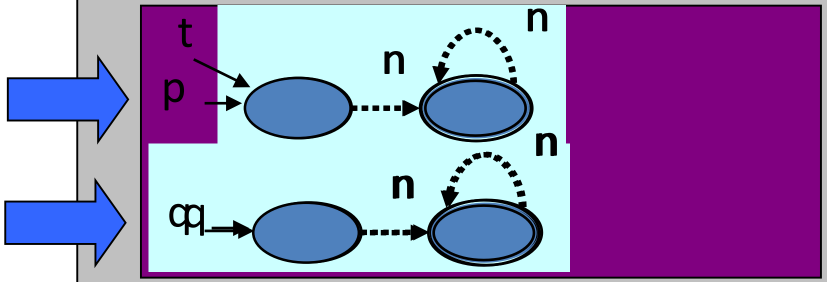
- Concrete operational semantics
  - Large step
    - Functional analysis
  - Storeless
    - Shape abstractions
  - Local heap
  - Observationally equivalent to “standard” semantics
    - Java and “clean” C
- Abstractions
  - Shape analysis [Sagiv, Reps, Wilhelm, TOPLAS ‘02]
  - May-alias [Deutsch, PLDI ‘94]
  - …

Outline

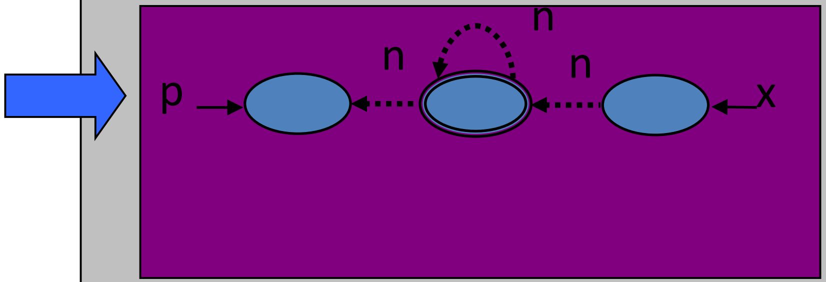
- Motivating example
  - Local heaps
  - Cutpoints
- Why semantics
- Local heap storeless semantics
- Shape abstraction

Example

```
static void main() {

...
```

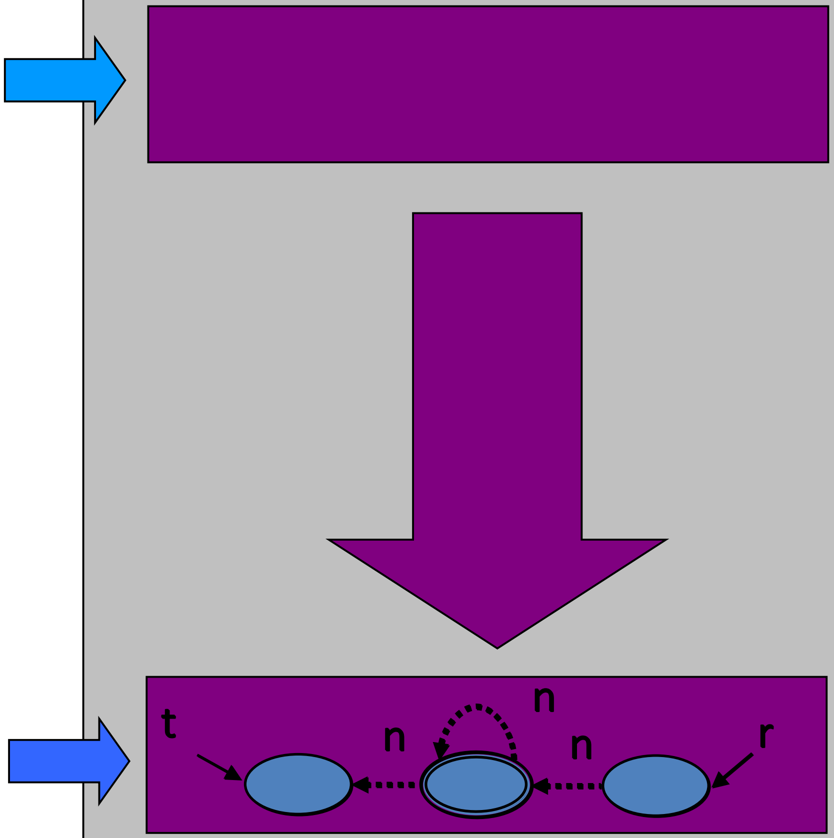


```
List x = reverse(p);
```



```
List y = reverse(q);

List z = reverse(x);

}
```

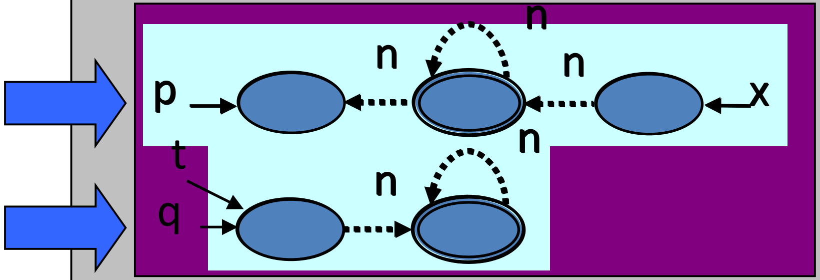
```
static List reverse(List t) {
```



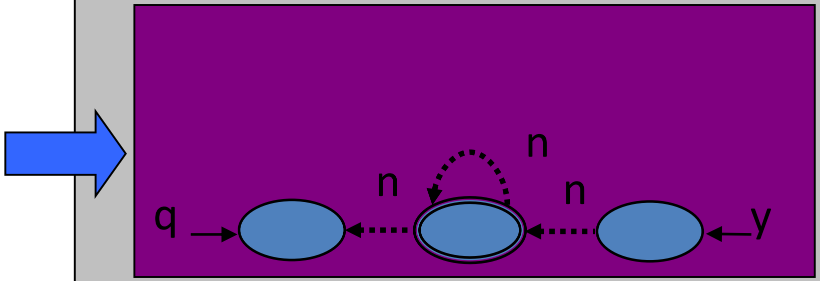
```
return r;

}
```

Example

```
static void main() {
        List x = reverse(p);
```

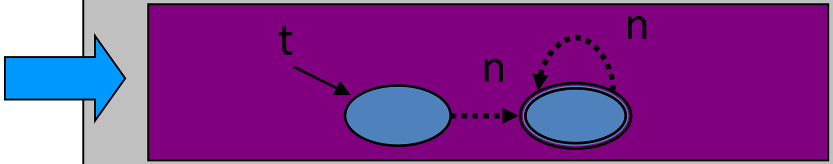


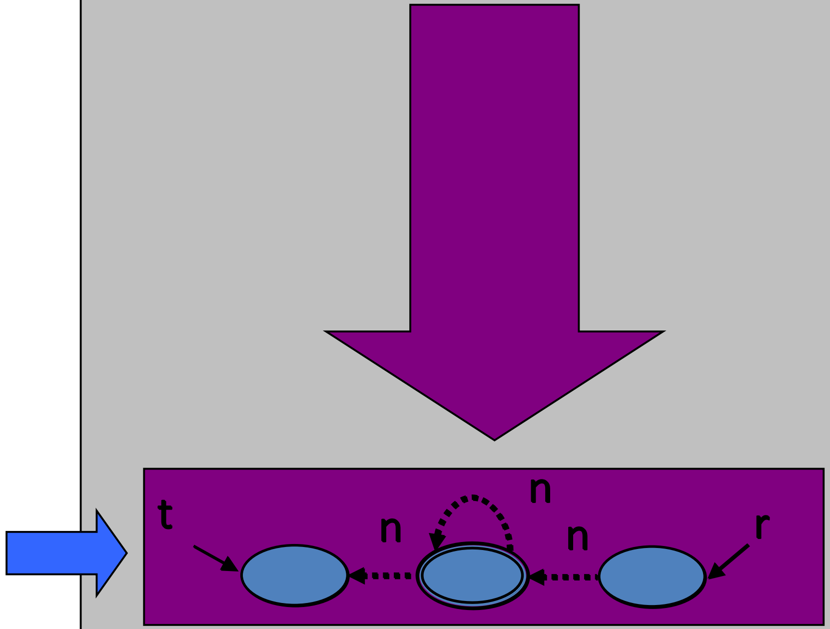
```
        List y = reverse(q);
```



```
        List z = reverse(x);
}
```

```
static List reverse(List t) {
```





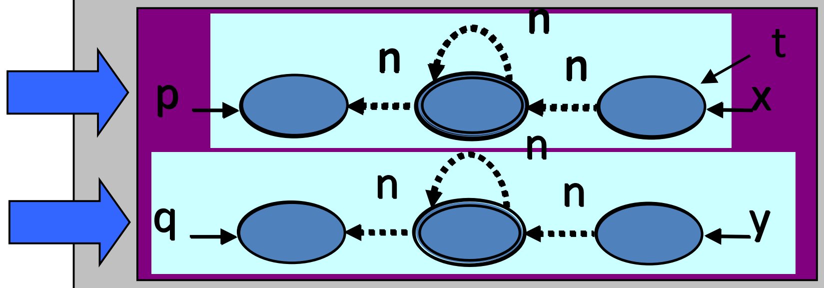
```
        return r;
}
```

Example

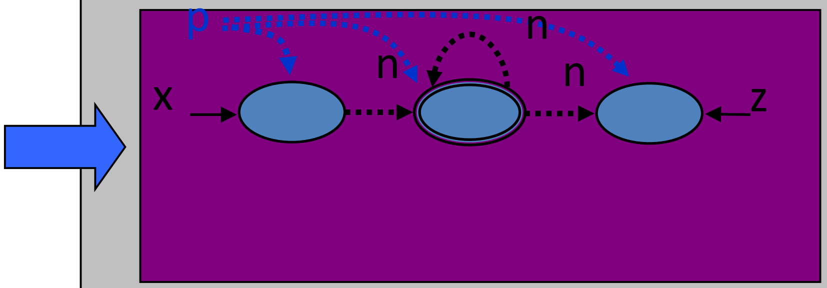
static void main() {

List x = reverse(p);

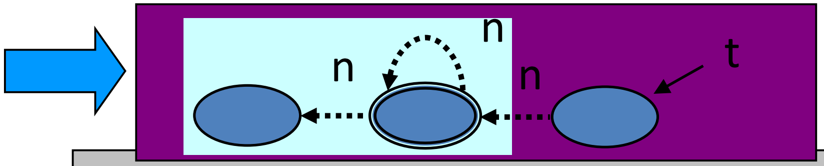
List y = reverse(q);



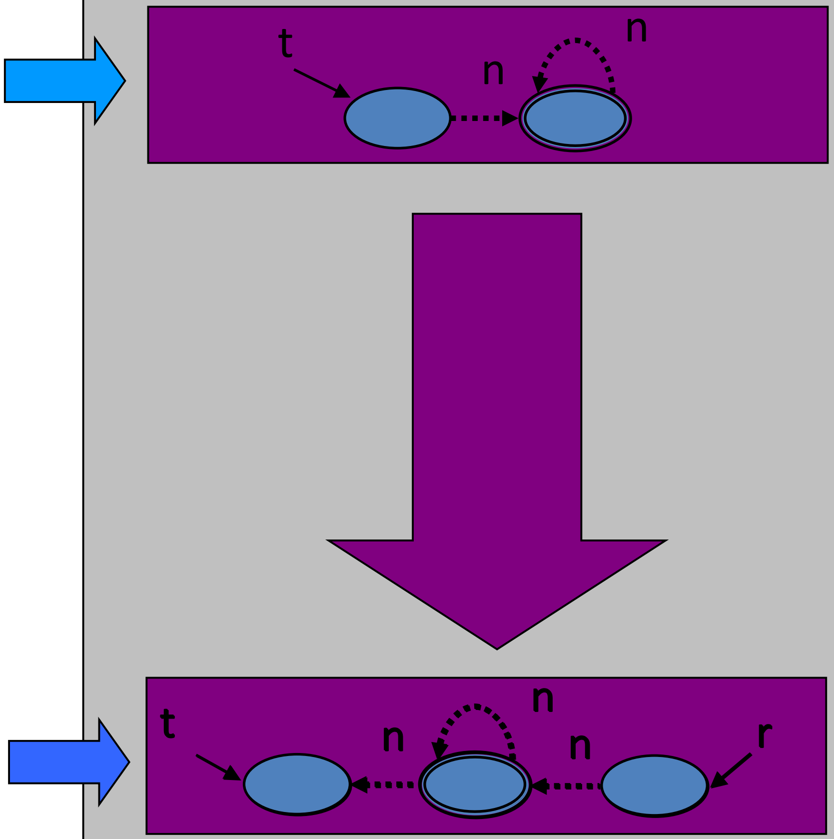
List z = reverse(x);



}



static List reverse(List t) {



return r;

}

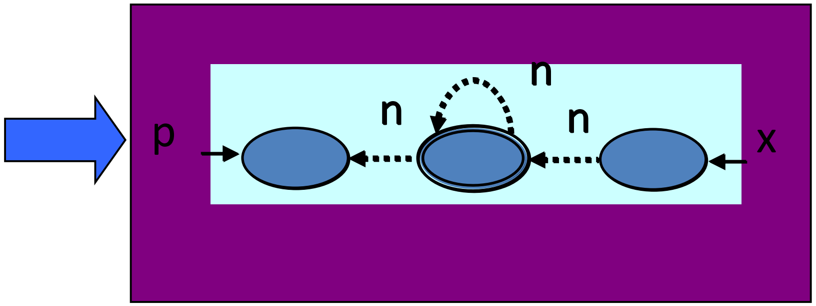
Cutpoints

- Separating objects
  - Not pointed-to by a parameter

Cutpoints

- Separating objects
  - Not pointed-to by a parameter

proc(x)



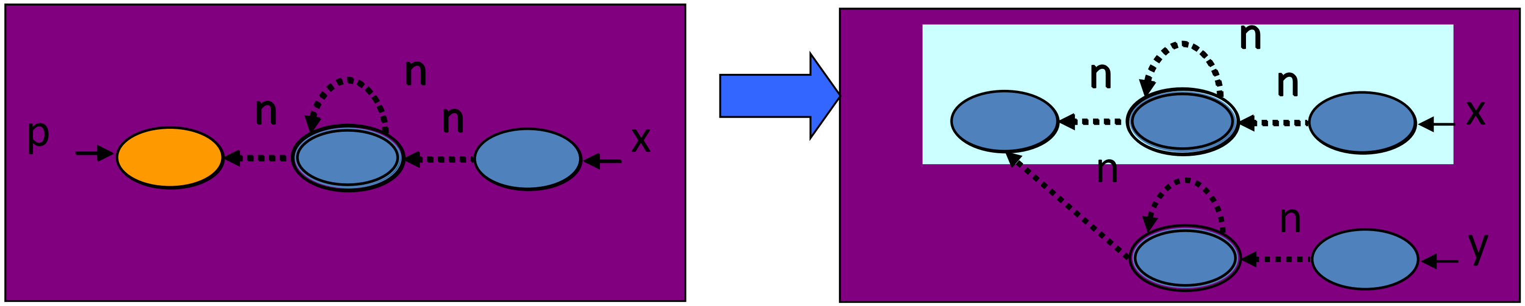
Stack sharing

Cutpoints

- Separating objects
  - Not pointed-to by a parameter

proc(x)

proc(x)

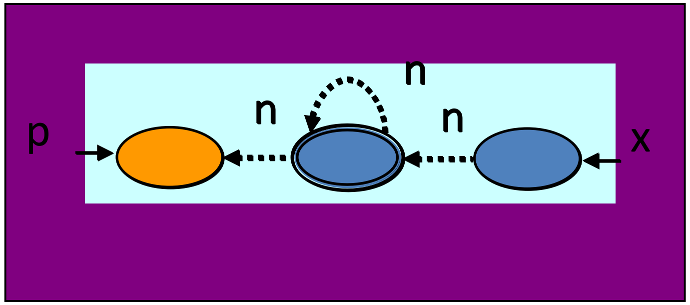


Stack sharing

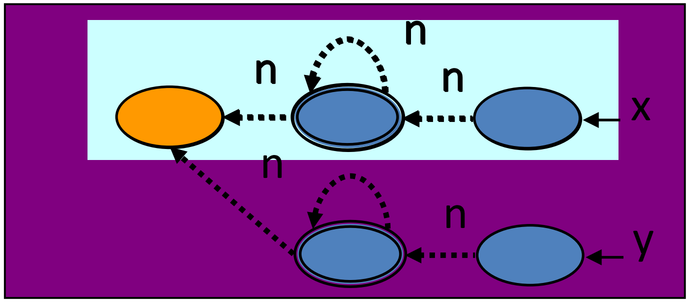
Heap sharing

Cutpoints

- Separating objects
  - Not pointed-to by a parameter
- Capture external sharing patterns


proc(x)

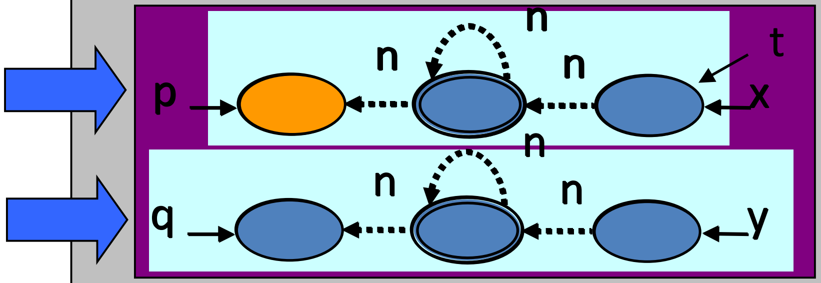
Stack sharing


proc(x)

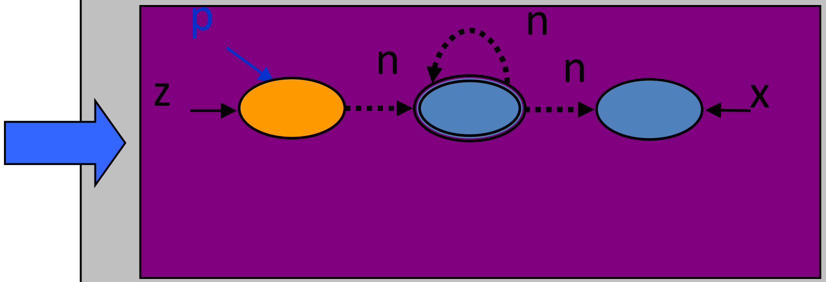
Heap sharing

Example

```
static void main() {
        List x = reverse(p);
        List y = reverse(q);
```

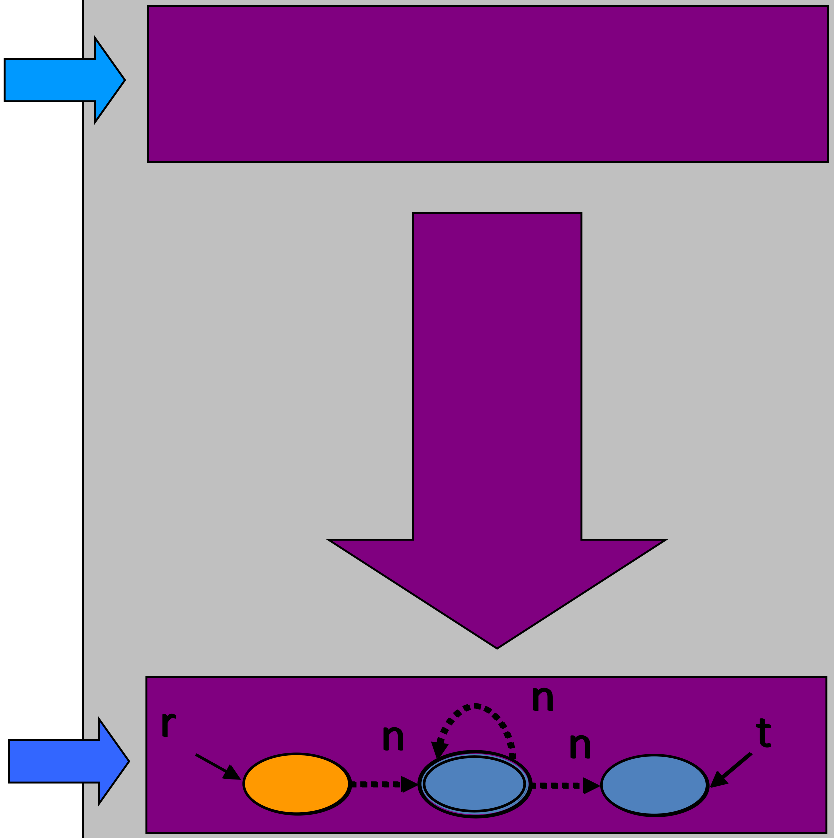


```
        List z = reverse(x);
```



```
}
```

```
static List reverse(List t) {
```



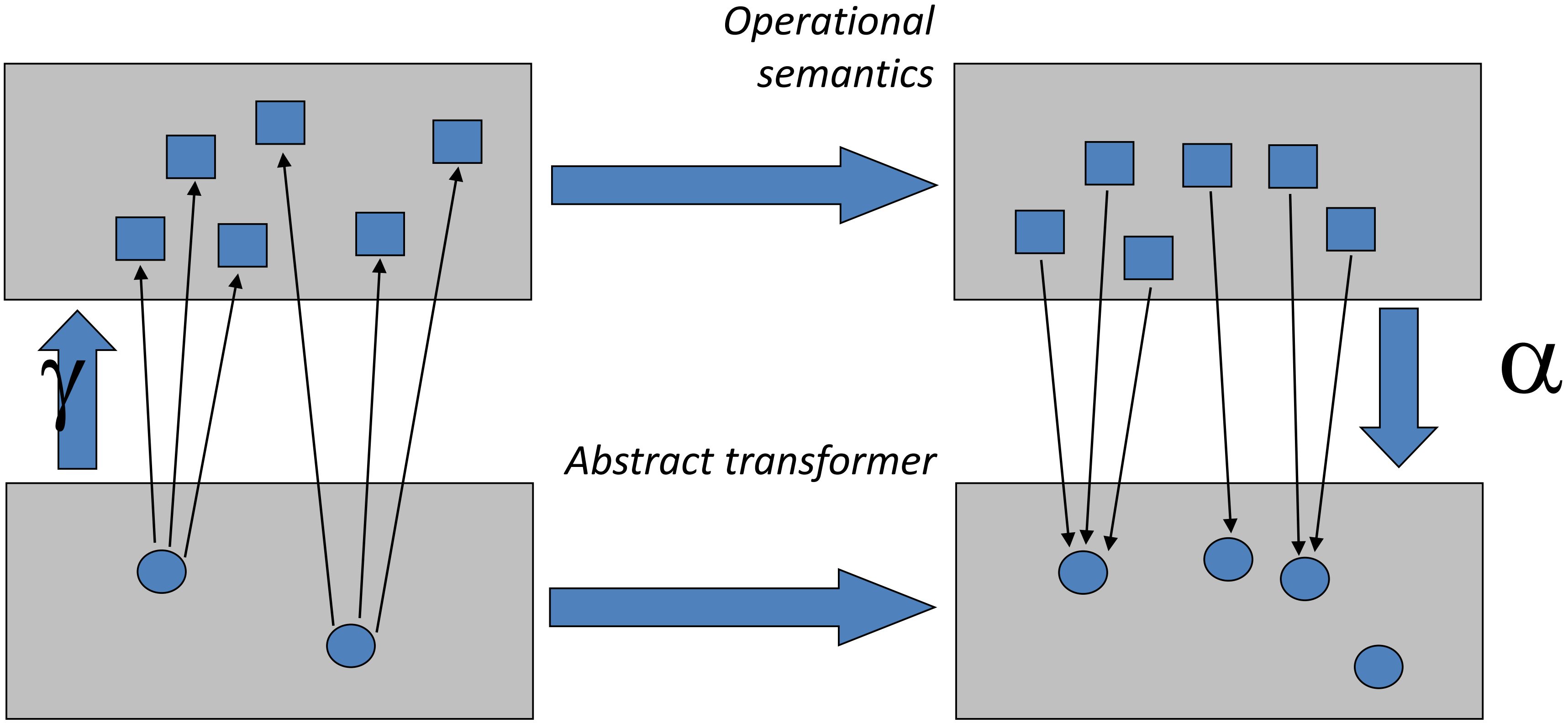
```
        return r;
}
```

Outline

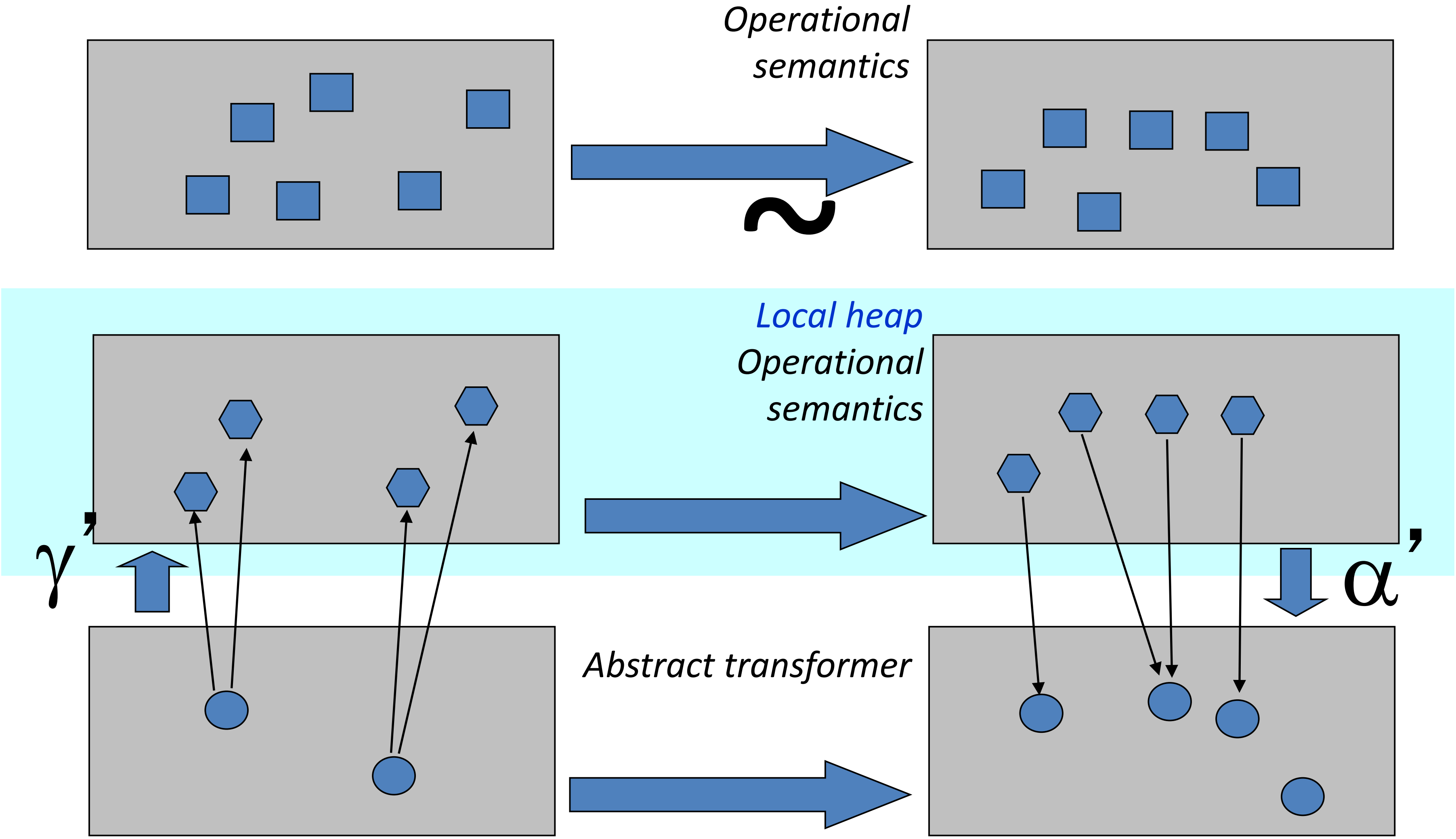
✓ Motivating example
- Why semantics
- Local heap storeless semantics
- Shape abstraction

Abstract Interpretation

[Cousot and Cousot, POPL ' 77]



Introducing local heap semantics



Outline

- ✓ Motivating example
- ✓ Why semantics
- Local heap storeless semantics
- Shape abstraction

Programming model

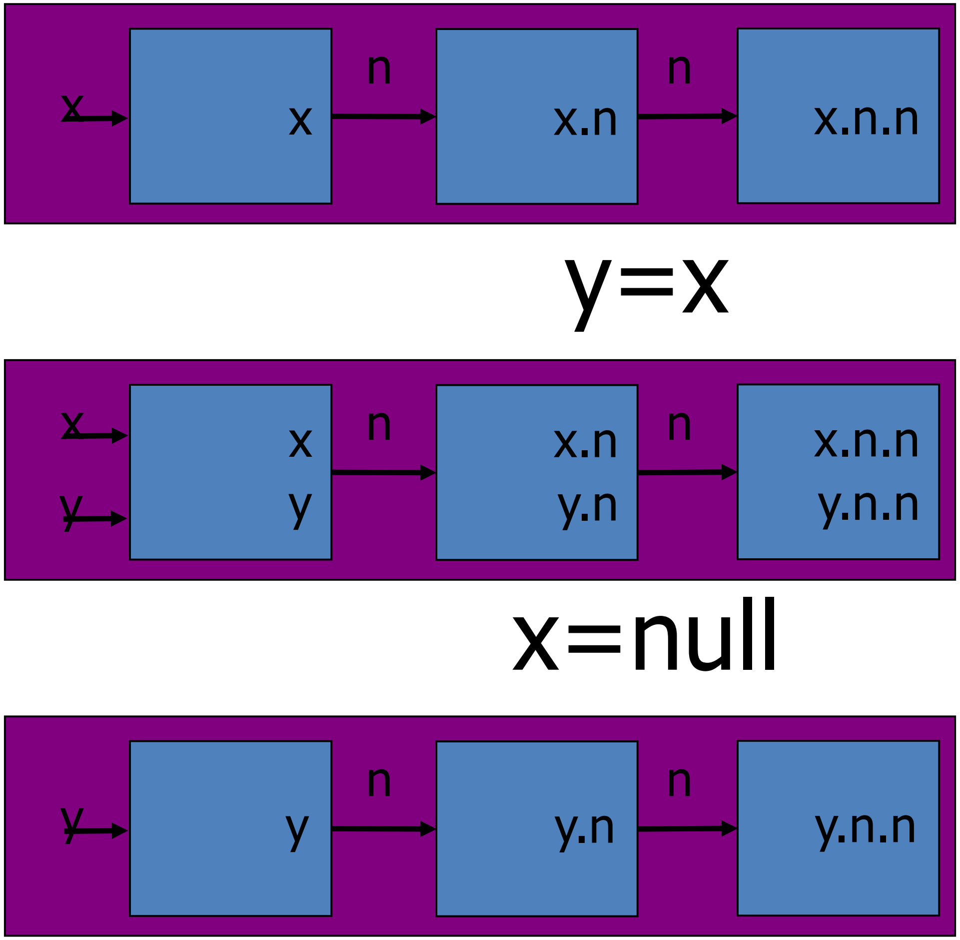
- Single threaded
- Procedures
  - ✓ Value parameters
  - ✓ Recursion
- Heap
  - ✓ Recursive data structures
  - ✓ Destructive update
  - ✗ No explicit addressing (&)
  - ✗ No pointer arithmetic

Simplifying assumptions

- No primitive values (only references)
- No globals
- Formals not modified

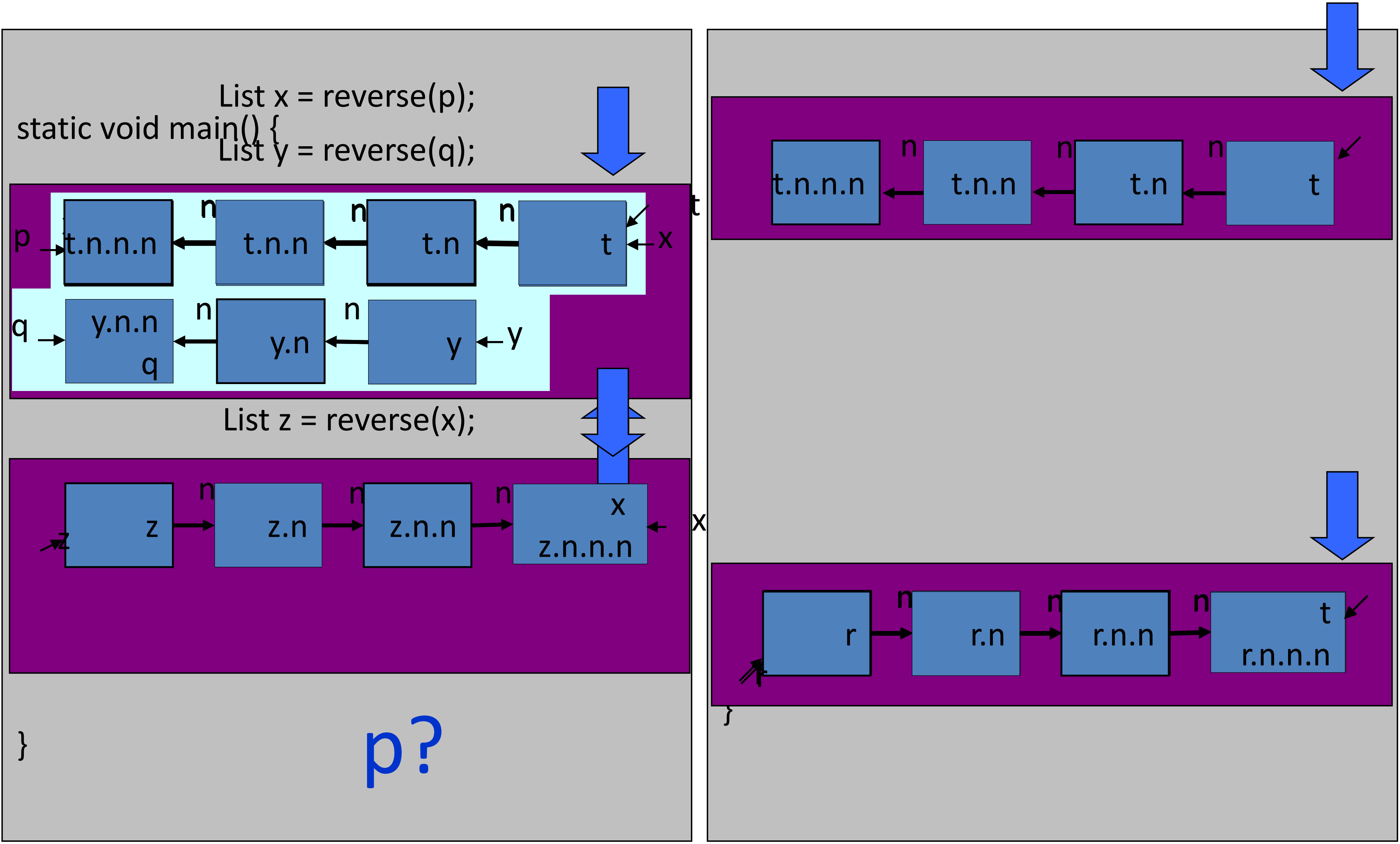
Storeless semantics

- No addresses
- Memory state:
  - Object: $2^{\text{Access paths}}$
  - Heap: 2^{Object}
- Alias analysis



Example

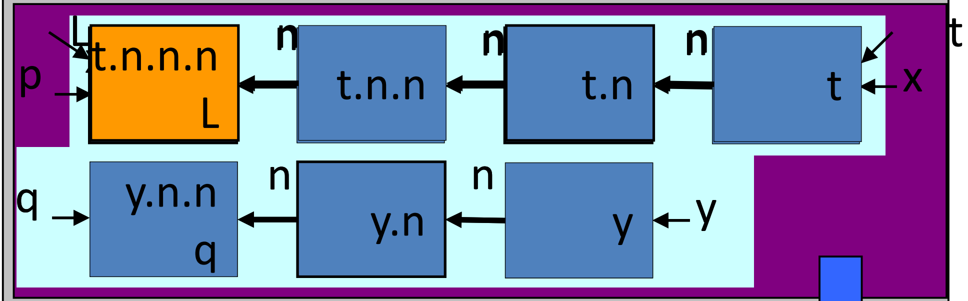
```
static void main() {
    List x = reverse(p);
    List y = reverse(q);
    List z = reverse(x);
}
```



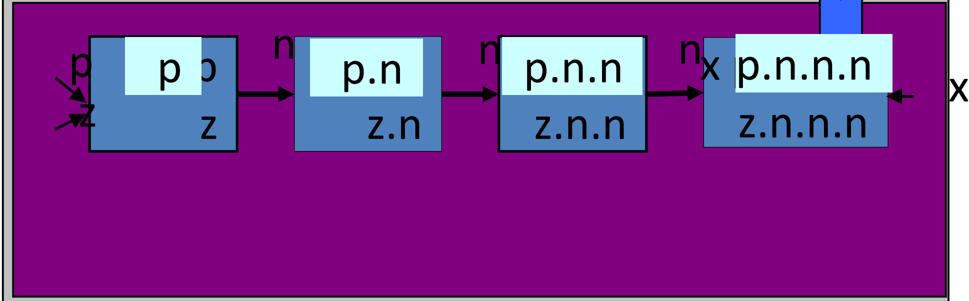
p?

Example

```
static void main() {
        List x = reverse(p);
        List y = reverse(q);
```

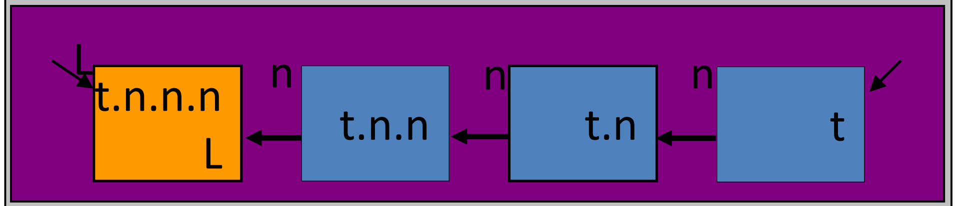


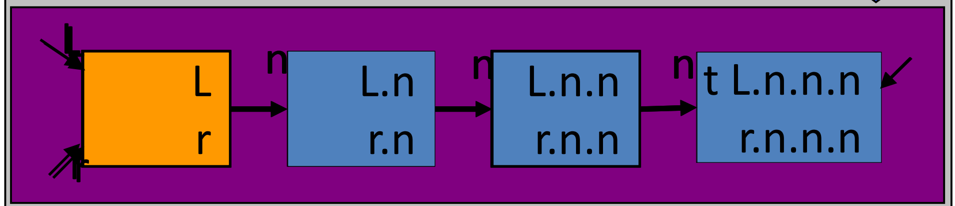
```
        List z = reverse(x);
```



```
}
```

```
static List reverse(List t) {
```





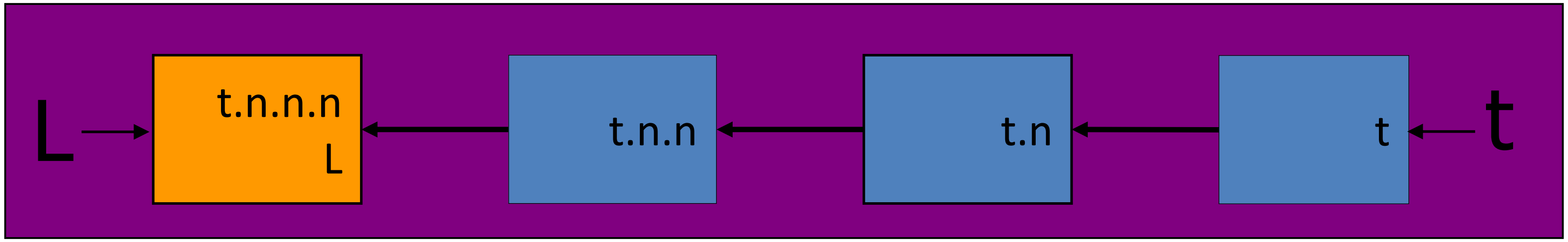
```
    return r;
}
```

Cutpoint labels

- Relate pre-state with post-state
- Additional roots
- Mark cutpoints at and throughout an invocation

Cutpoint labels

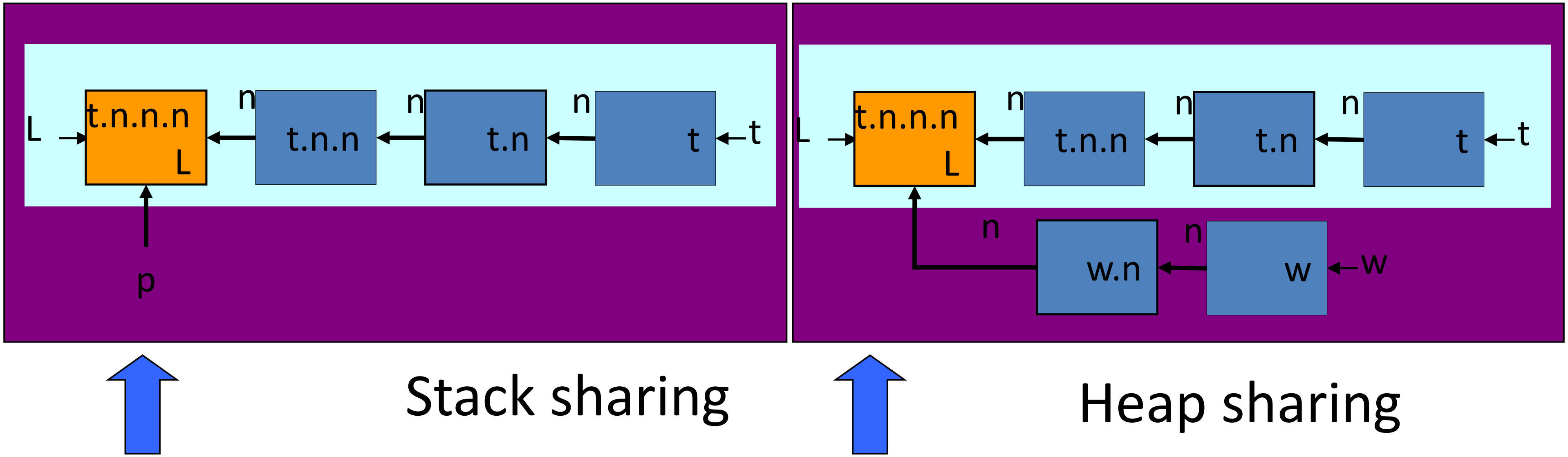
- **Cutpoint label**: the set of access paths that point to a cutpoint
  - when the invoked procedure starts



$$L \equiv \{t.n.n.n\}$$

Sharing patterns

- Cutpoint labels encode sharing patterns



Stack sharing

Heap sharing

$$L \equiv \{t.n.n.n\}$$

Observational equivalence

- $\sigma_L \in \Sigma_L$ (Local-heap Storeless Semantics)
- $\sigma_G \in \Sigma_G$ (Global-heap Store-based Semantics)

σ_L and σ_G observationally equivalent when for every access paths AP_1, AP_2

$$[\![AP_1 = AP_2]\!](\sigma_L) \Leftrightarrow [\![AP_1 = AP_2]\!](\sigma_G)$$

Main theorem: semantic equivalence

- $\sigma_L \in \Sigma_L$ (Local-heap Storeless Semantics)
- $\sigma_G \in \Sigma_G$ (Global-heap Store-based Semantics)
- σ_L and σ_G observationally equivalent

$$\langle st, \sigma_L \rangle \overset{LSL}{\rightsquigarrow} \sigma'_L \Leftrightarrow \langle st, \sigma_G \rangle \overset{GSB}{\rightsquigarrow} \sigma'_G$$

σ'_L and σ'_G are observationally equivalent

Corollaries

- Preservation of invariants
  - Assertions: $AP_1 = AP_2$
- Detection of memory leaks

Applications

- Develop new static analyses
  - Shape analysis
- Justify soundness of existing analyses

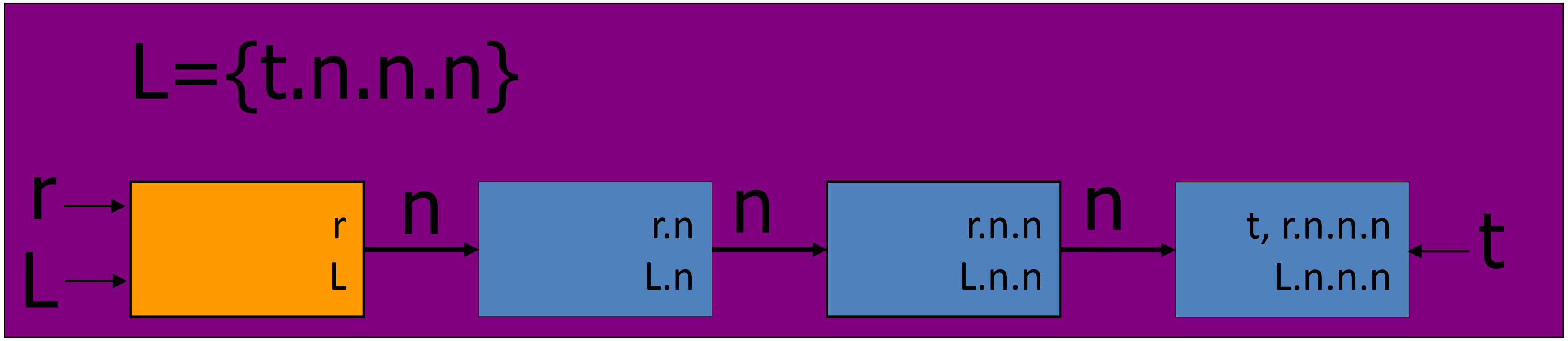
Related work

- **Storeless semantics**
  - Jonkers, Algorithmic Languages '81
  - Deutsch, ICCL '92

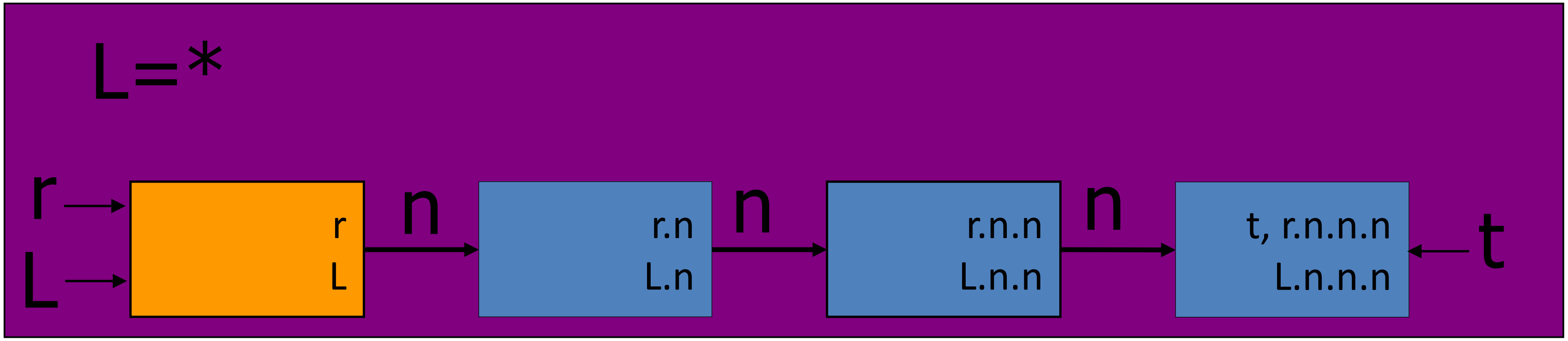
Shape abstraction

- Shape descriptors represent unbounded memory states
  - Conservatively
  - In a bounded way
- Two dimensions
  - Local heap (objects)
  - Sharing pattern (cutpoint labels)

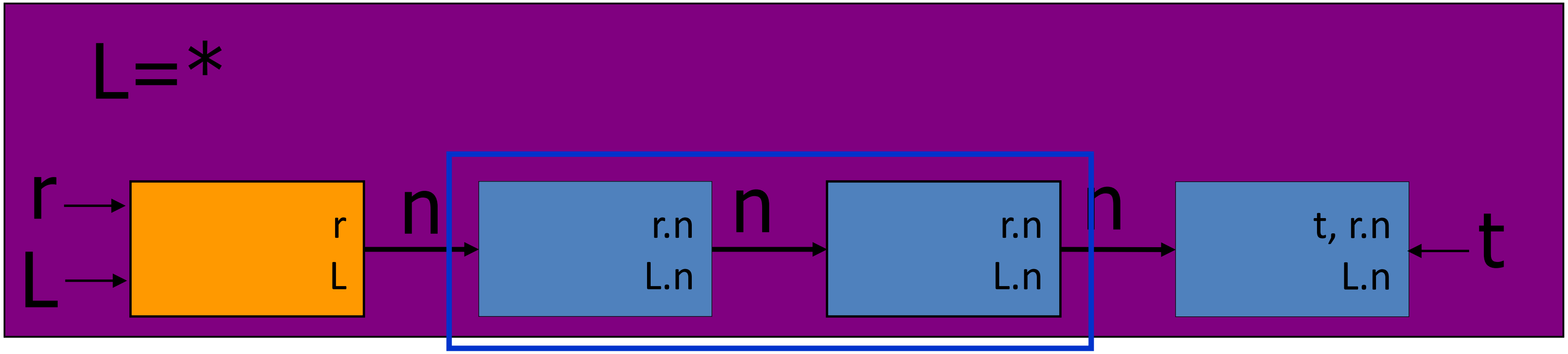
A Shape abstraction



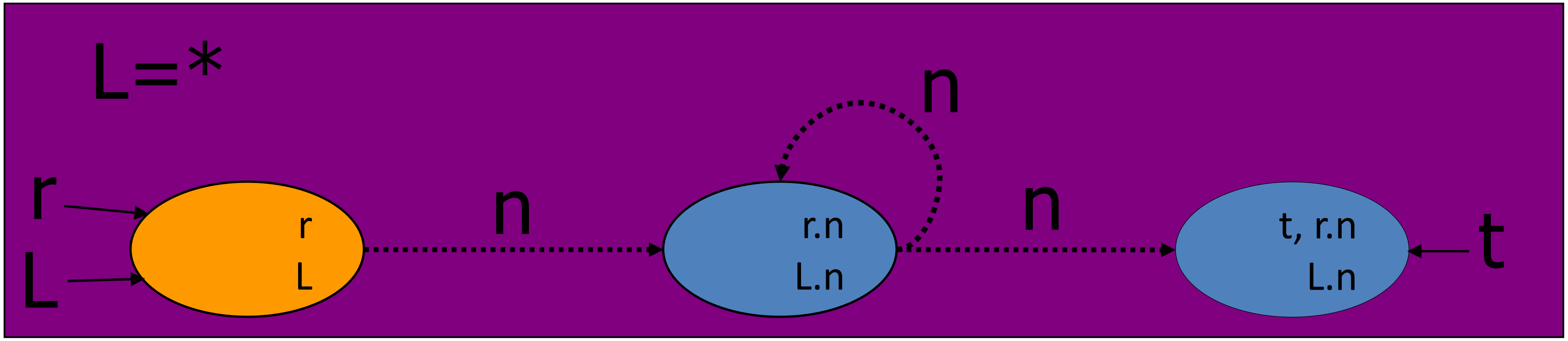
A Shape abstraction



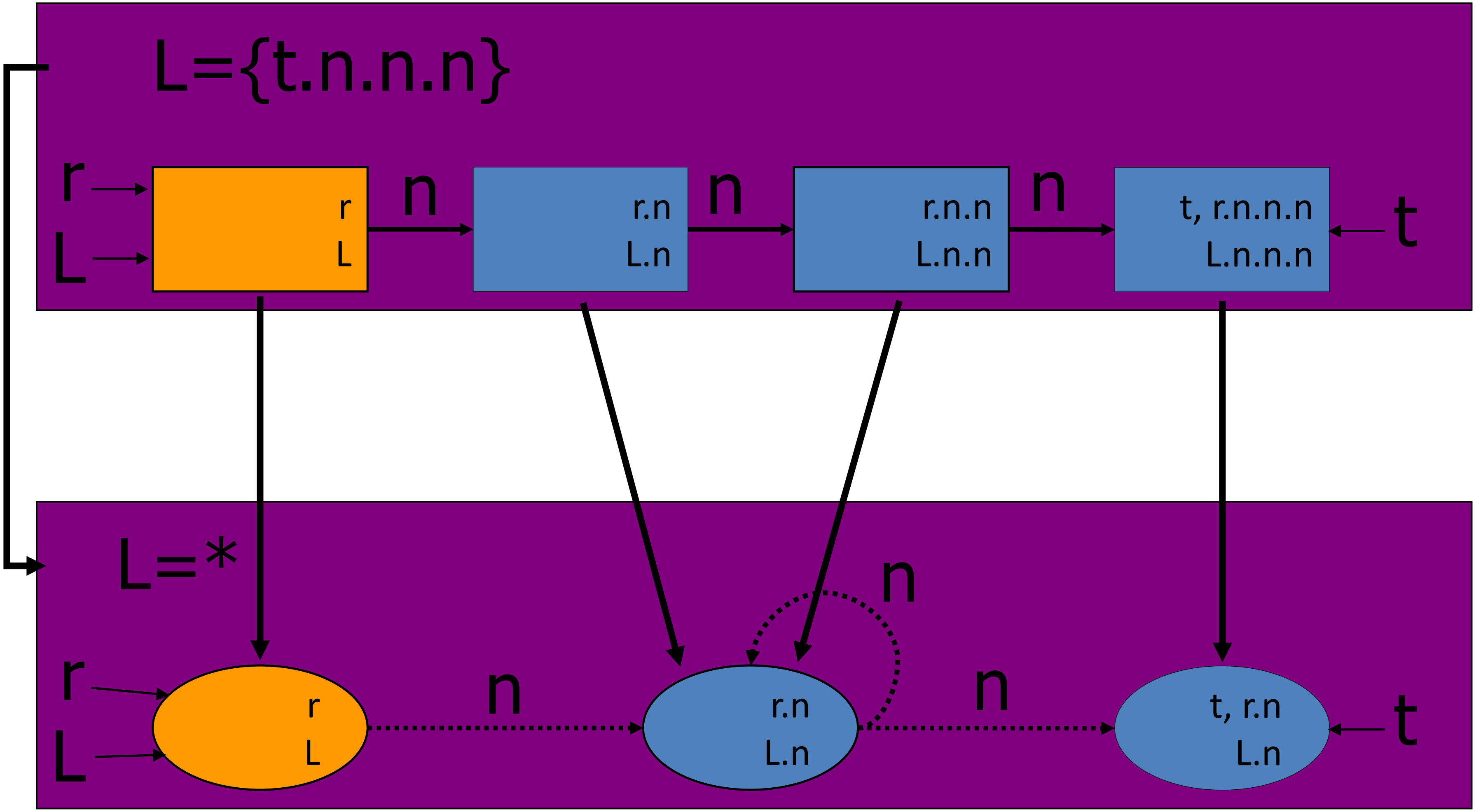
A Shape abstraction



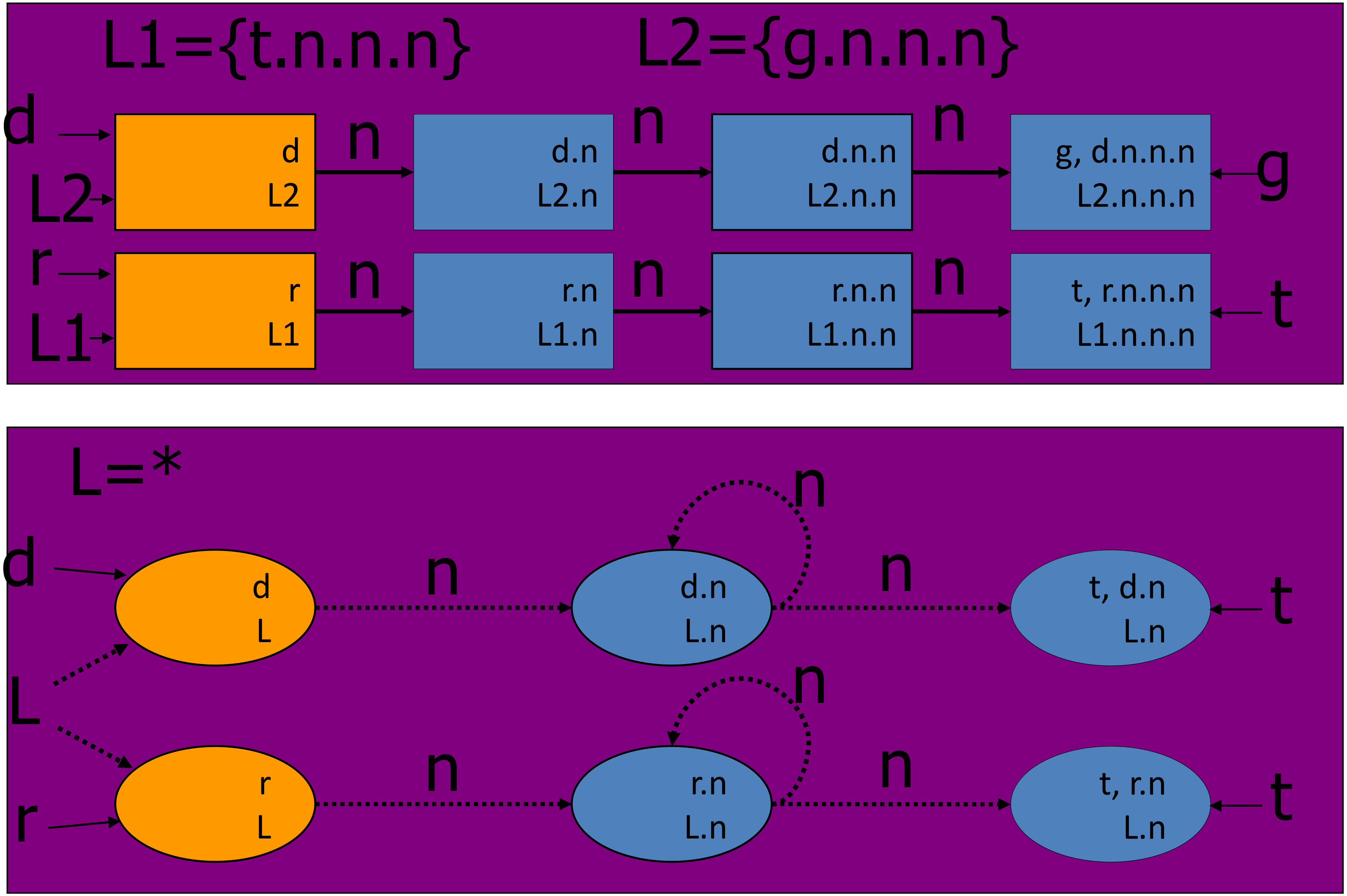
A Shape abstraction



A Shape abstraction



A Shape abstraction



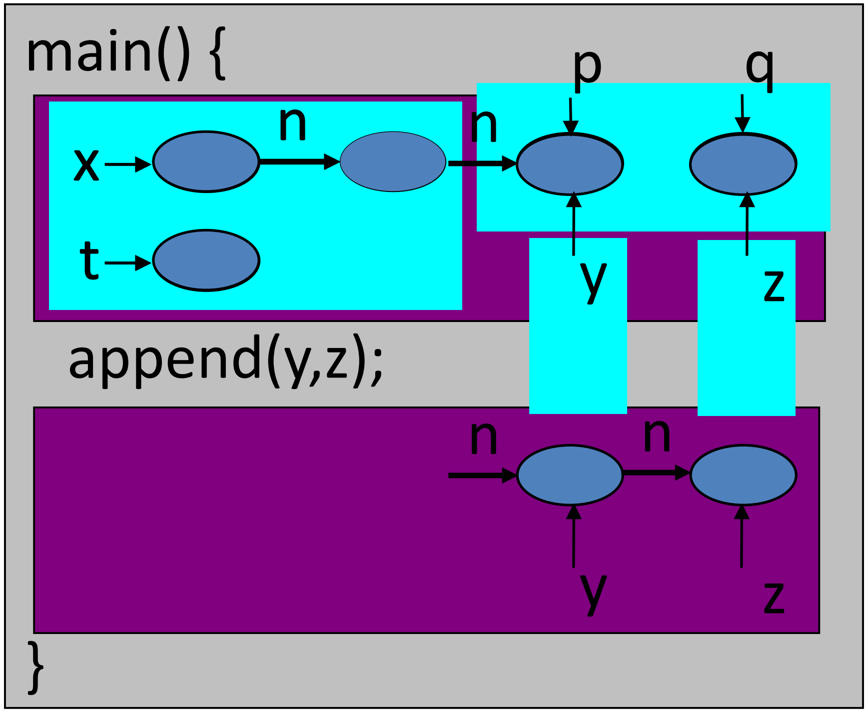
L1={t.n.n.n} L2={g.n.n.n}

L=*

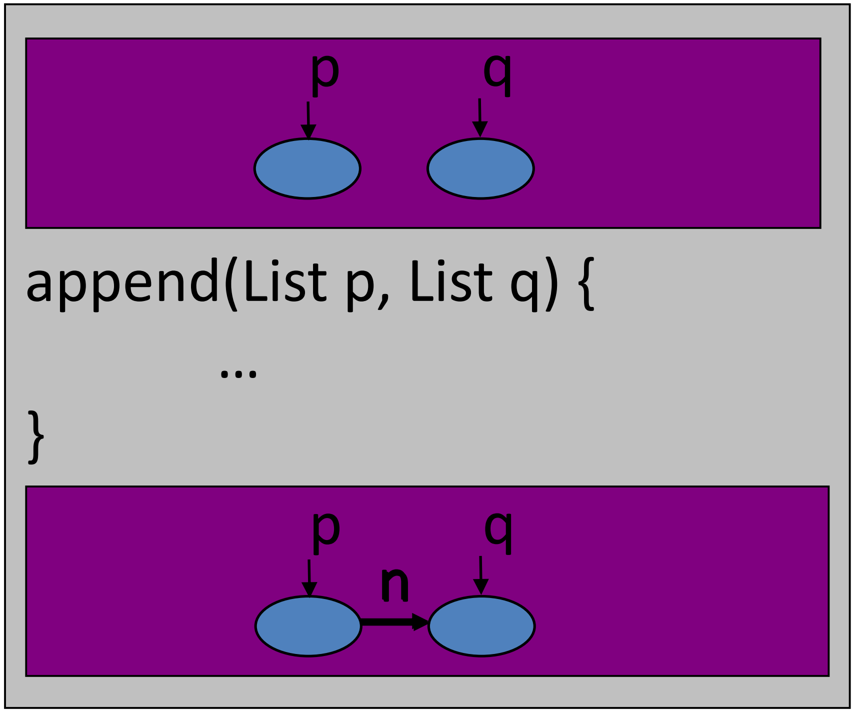
Cutpoint-Freedom

How to tabulate procedures?

- Procedure ≡ input/output relation
  - Not reachable ➔ Not effected
  - proc: local (≡reachable) heap ➔ local heap



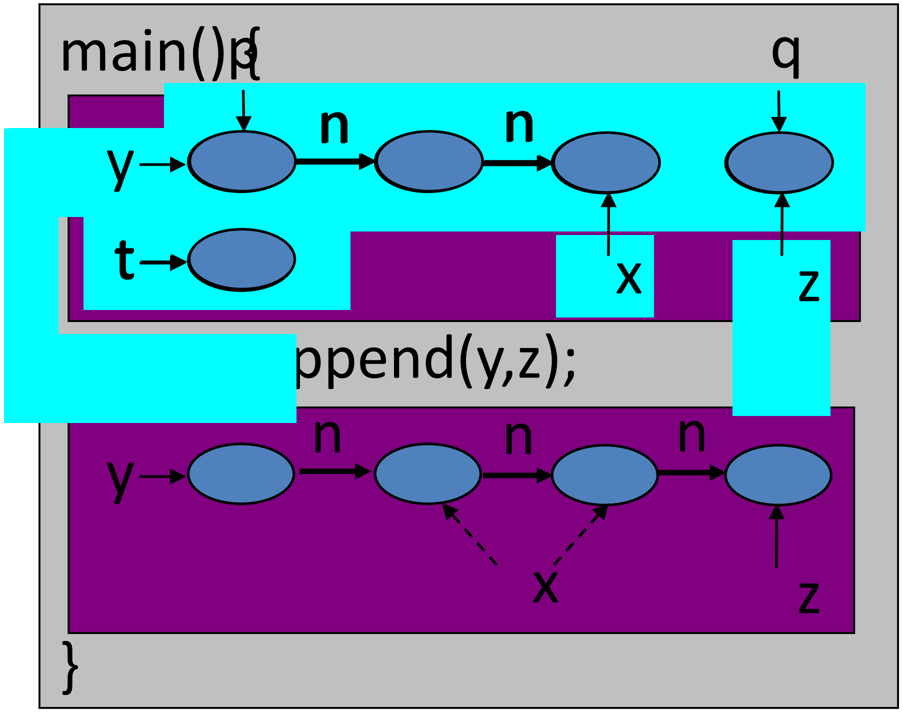
```
main() {
  append(y,z);
}
```



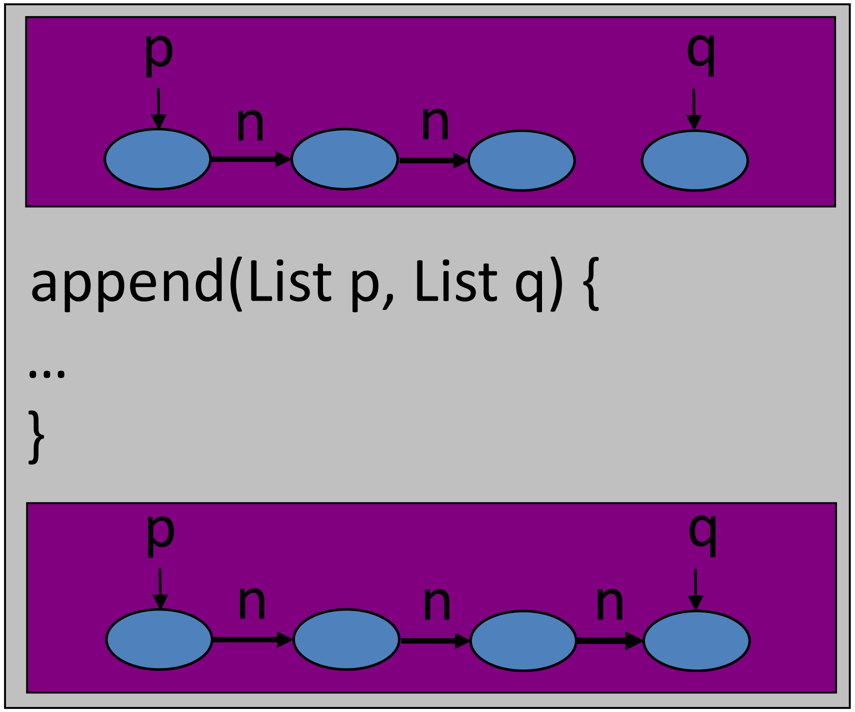
```
append(List p, List q) {
        …
}
```

How to handle sharing?

- External sharing may break the functional view



```
main() {
ppend(y,z);
}
```

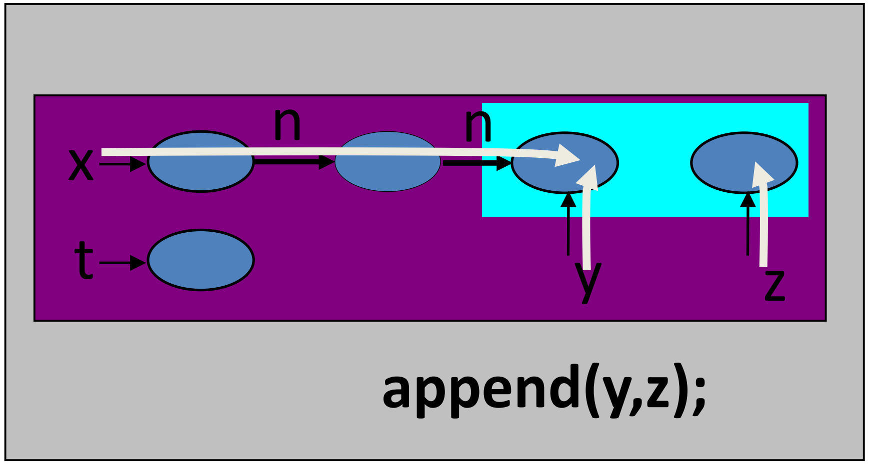


```
append(List p, List q) {
…
}
```

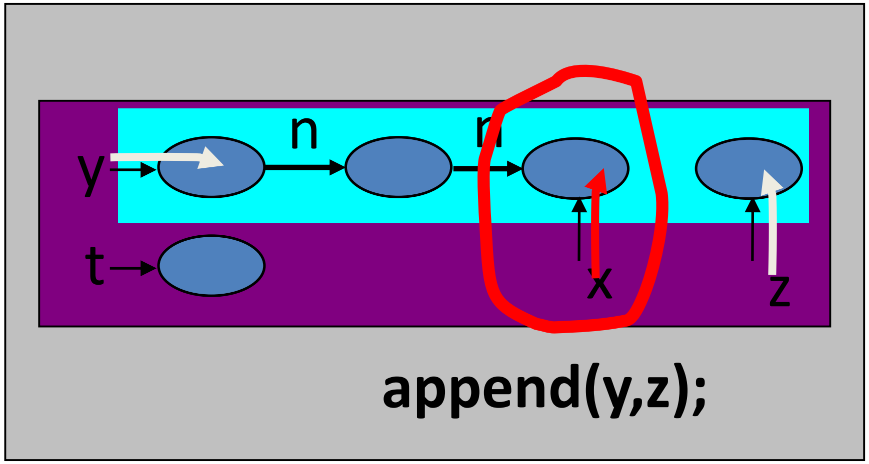
What's the difference?

1st Example

2nd Example



append(y,z);

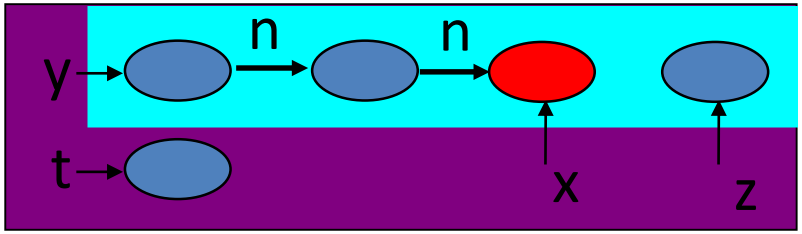


append(y,z);

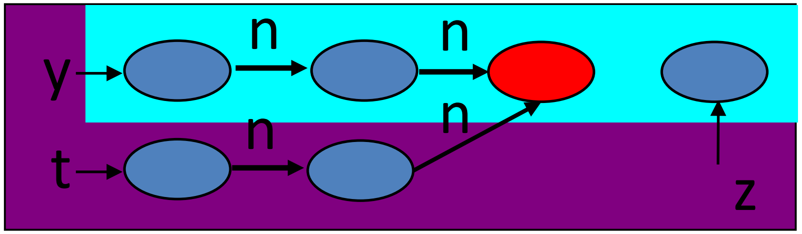
Cutpoints

- An object is a **cutpoint** for an invocation
  - Reachable from actual parameters
  - Not pointed to by an actual parameter
  - Reachable without going through a parameter

append(y,z)



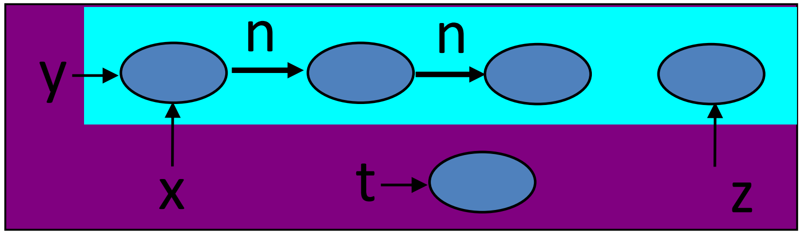
append(y,z)



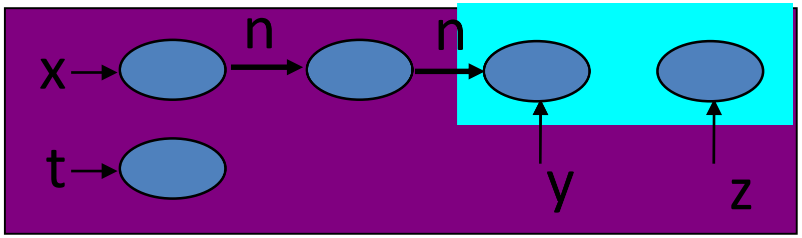
Cutpoint freedom

- **Cutpoint-free**
  - Invocation: has no cutpoints
  - Execution: every invocation is cutpoint-free
  - Program: every execution is cutpoint-free

append(y,z)



append(y,z)

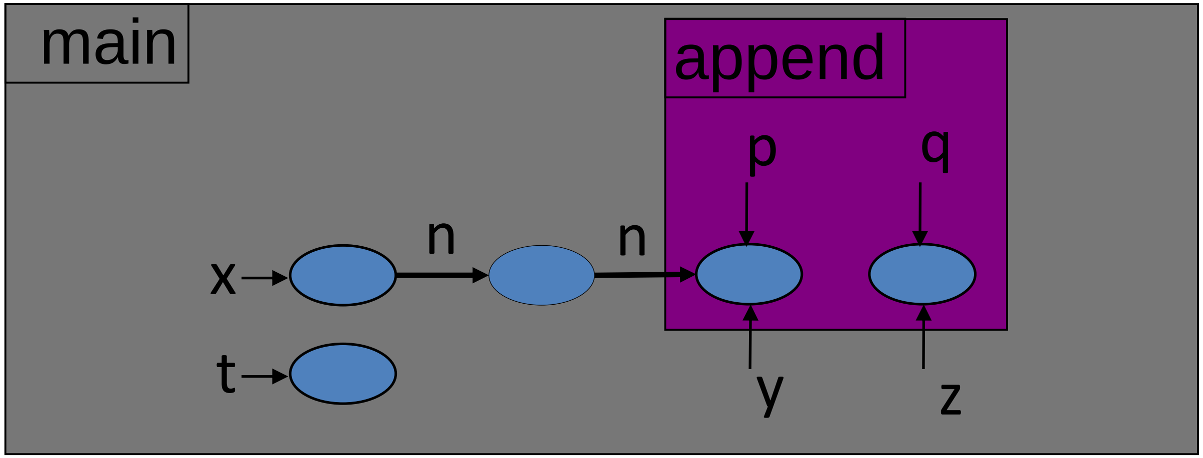


Interprocedural shape analysis for cutpoint-free programs

using 3-Valued Shape Analysis

Memory states: 2-Valued Logical Structure

- A memory state encodes a **local heap**
  - Local variables of the **current procedure invocation**
  - Relevant part of the heap
    - Relevant ≡ Reachable



Memory states

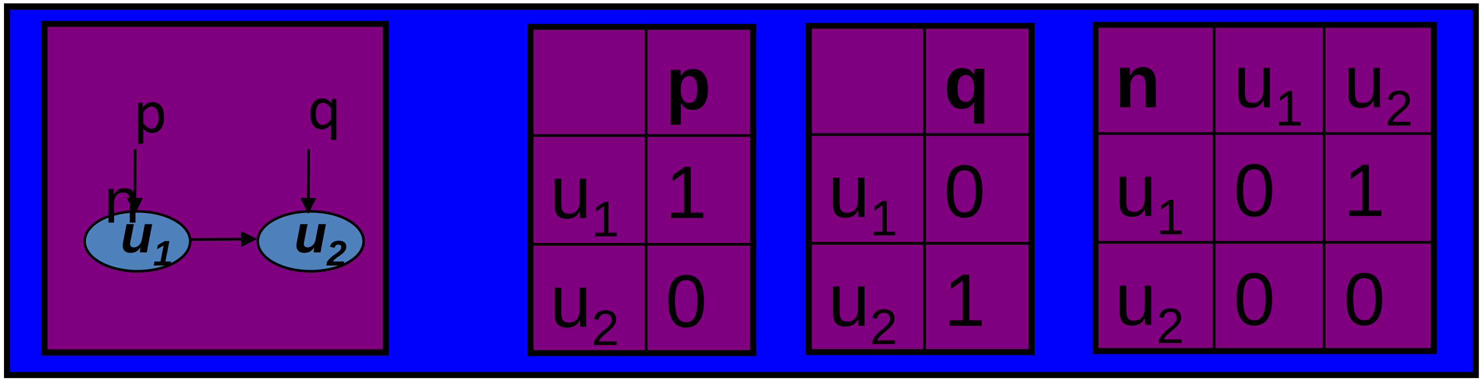
- Represented by first-order logical structures

Predicate	Meaning
x(v)	Variable **x** points to v
n(v_1,v_2)	Field **n** of object v_1 points to v_2

Memory states

- Represented by first-order logical structures

Predicate	Meaning
$\mathbf{x}(v)$	Variable **x** points to v
$\mathbf{n}(v_1,v_2)$	Field **n** of object v_1 points to v_2



	p
u_1	1
u_2	0

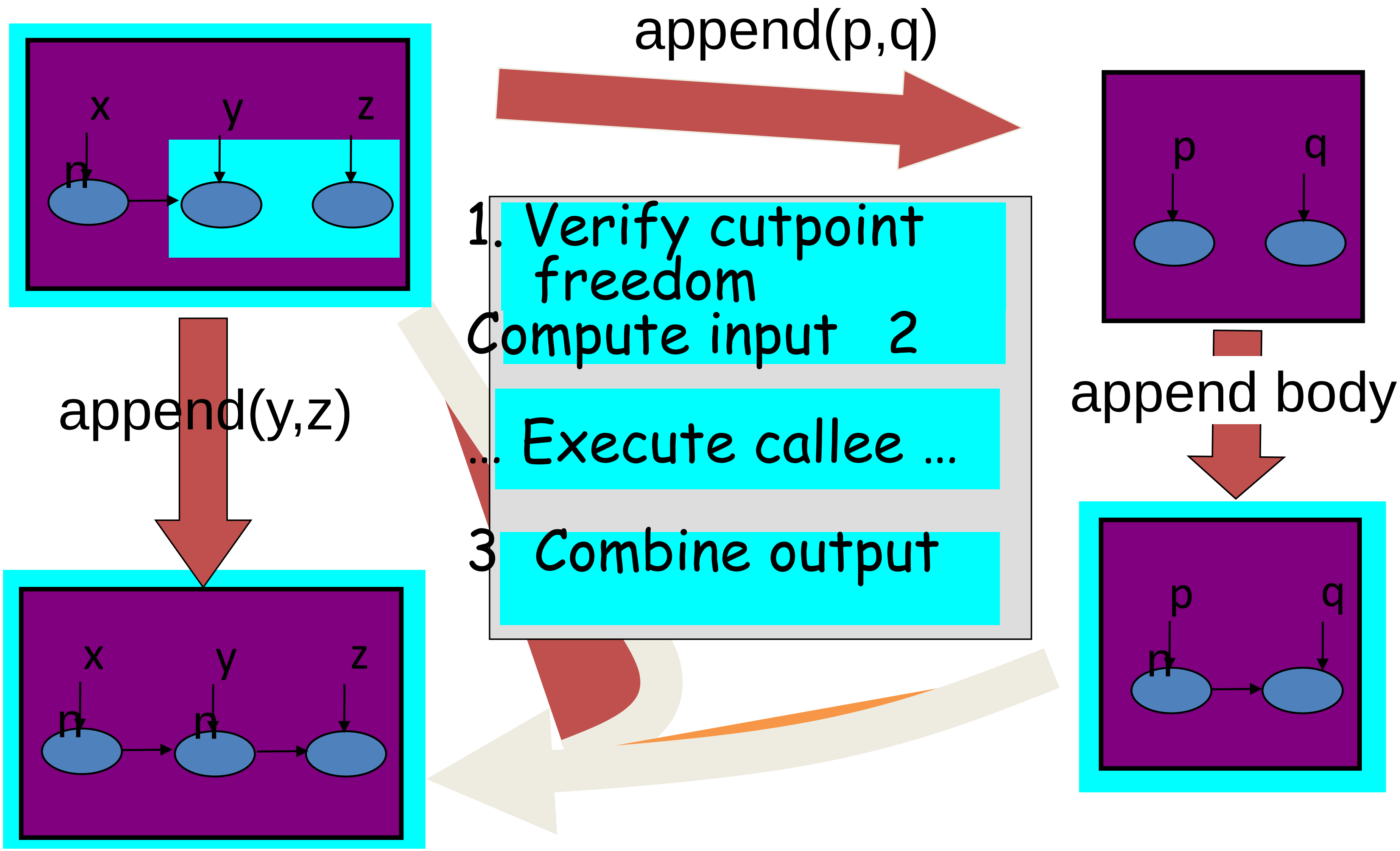
	q
u_1	0
u_2	1

n	u_1	u_2
u_1	0	1
u_2	0	0

Operational semantics

- Statements modify values of predicates
- Specified by predicate-update formulae
  - Formulae in FO-TC

Procedure calls



1. Verify cutpoint freedom
Compute input 2

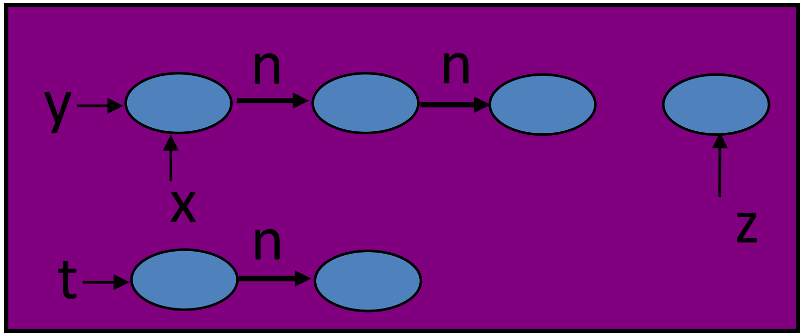
… Execute callee …

3 Combine output

Procedure call:
1. Verifying cutpoint-freedom

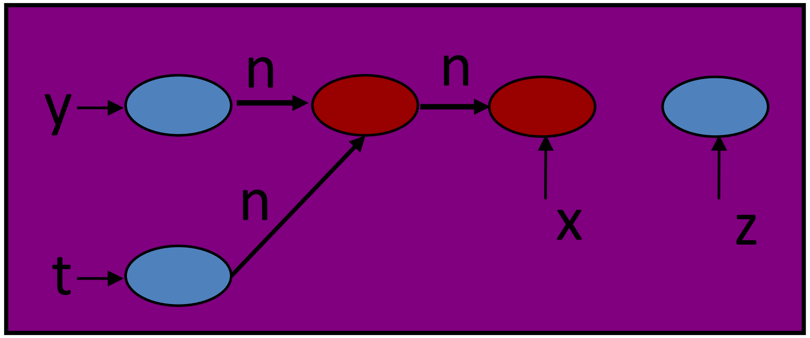
- An object is a **cutpoint** for an invocation
  - Reachable from actual parameters
  - Not pointed to by an actual parameter
  - Reachable without going through a parameter

append(y,z)



Cutpoint free

append(y,z)



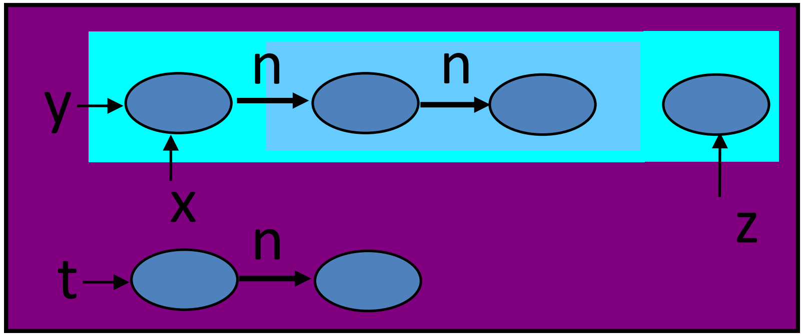
Not Cutpoint free

Procedure call:
1. Verifying cutpoint-freedom

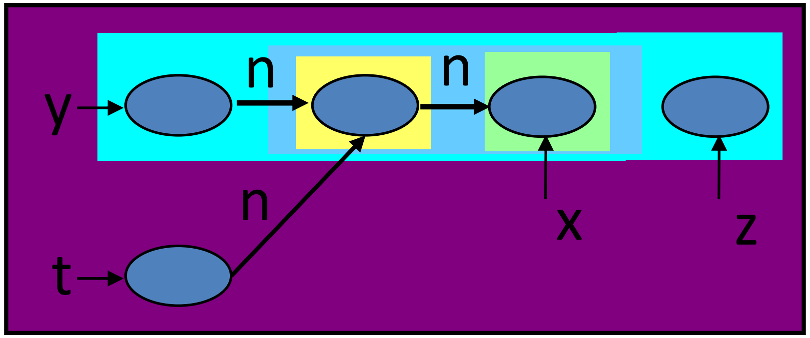
- Invoking append(y,z) in main
  - $R_{\{y,z\}}(v)=\exists v_1{:}y(v_1)\wedge n^*(v_1,v) \vee \exists v_1{:}z(v_1)\wedge n^*(v_1,v)$
  - $isCP_{main,\{y,z\}}(v)= R_{\{y,z\}}(v) \wedge (\neg y(v)\wedge\neg z(v_1)) \wedge$

    $(x(v) \vee t(v) \vee \exists v_1{:} \neg R_{\{y,z\}}(v_1)\wedge n(v_1,v))$

(main's locals: x,y,z,t)



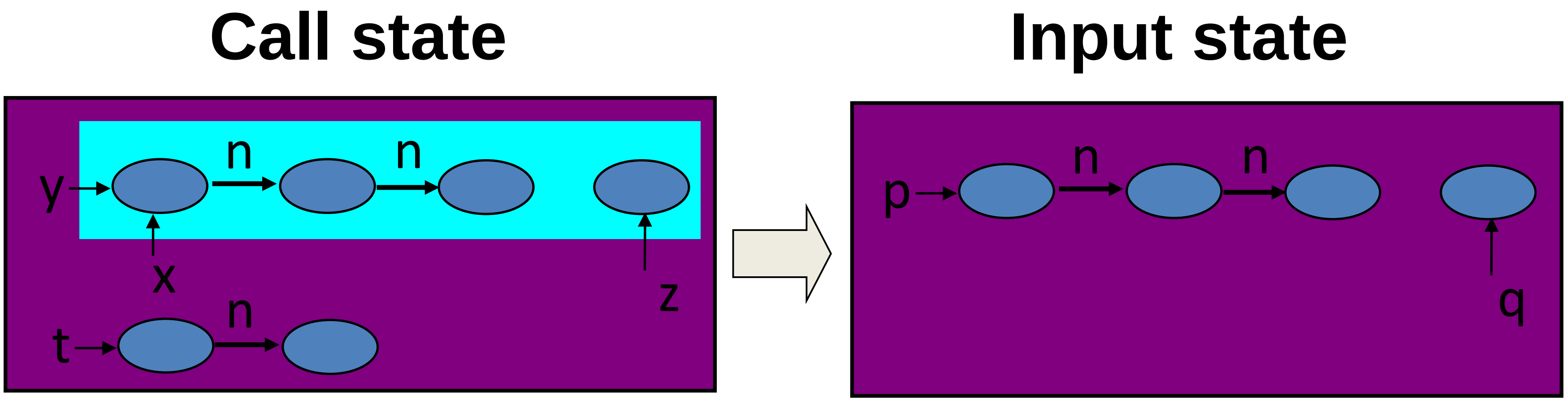
Cutpoint free



Not Cutpoint free

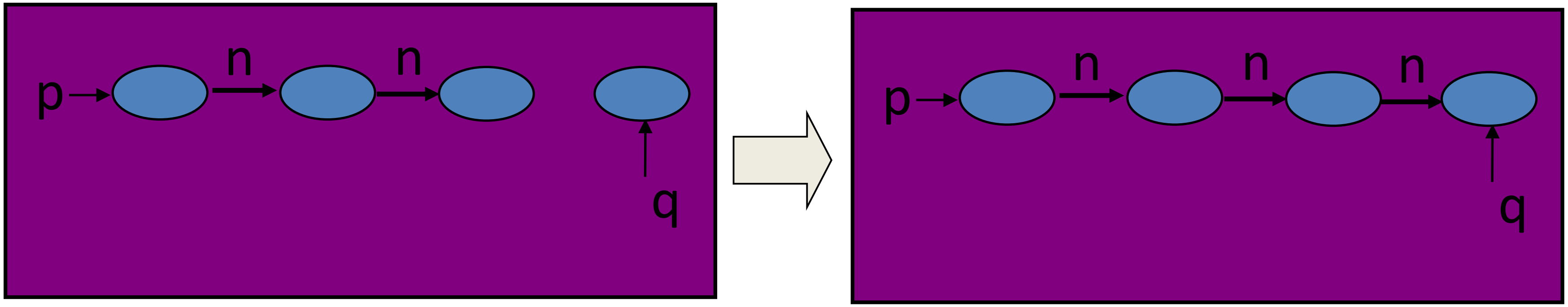
Procedure call:
2. Computing the input local heap

- Retain only reachable objects
- Bind formal parameters


Call state

Input state

Procedure body: append(p,q)



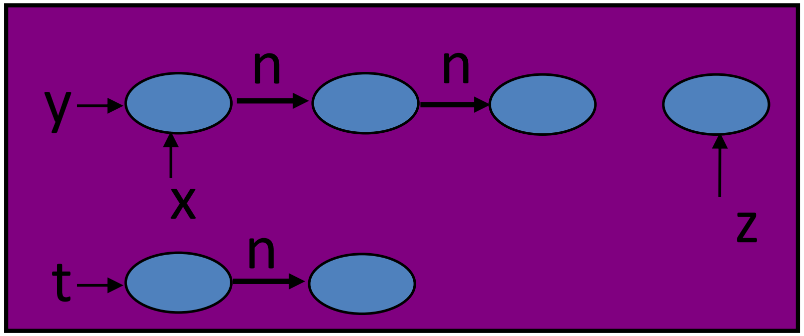
Input state

Output state

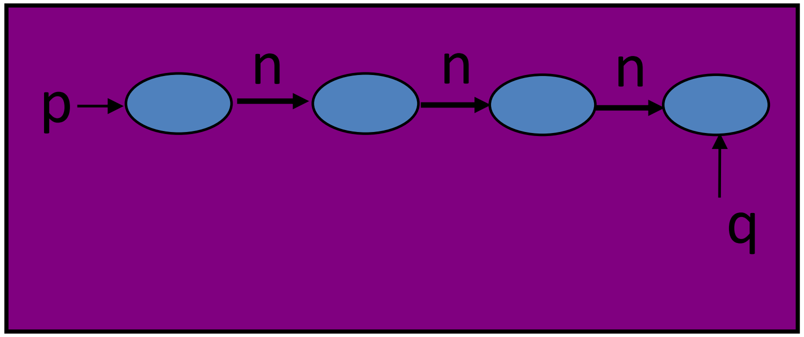
Procedure call:

3. Combine output

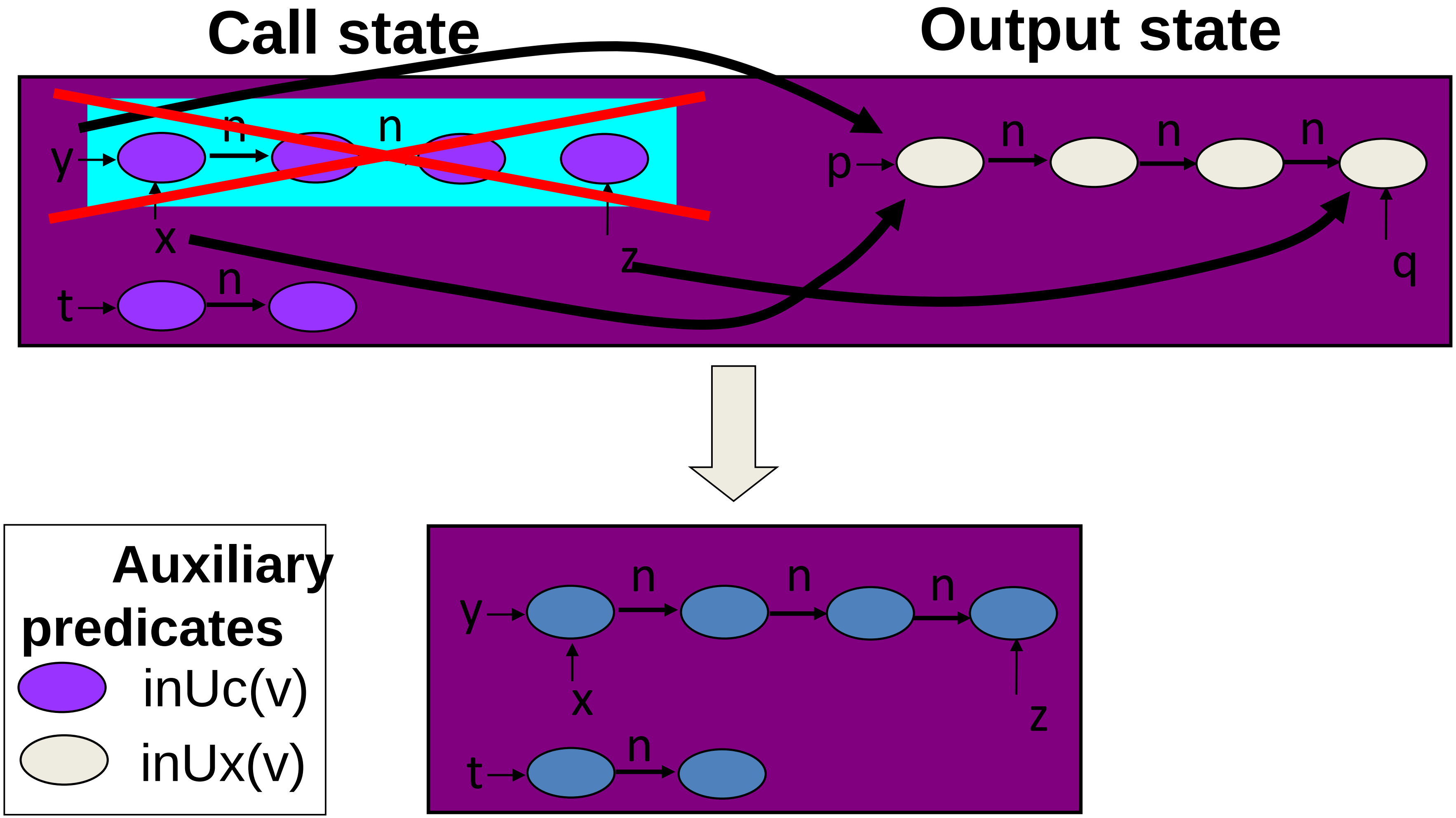
Call state



Output state



Procedure call:
3. Combine output



Auxiliary predicates

● inUc(v)

○ inUx(v)

Observational equivalence

- $\sigma_{CPF} \in \Sigma_{CPF}$ (Cutpoint free semantics)
- $\sigma_{GSB} \in \Sigma_{GSB}$ (Standard semantics)

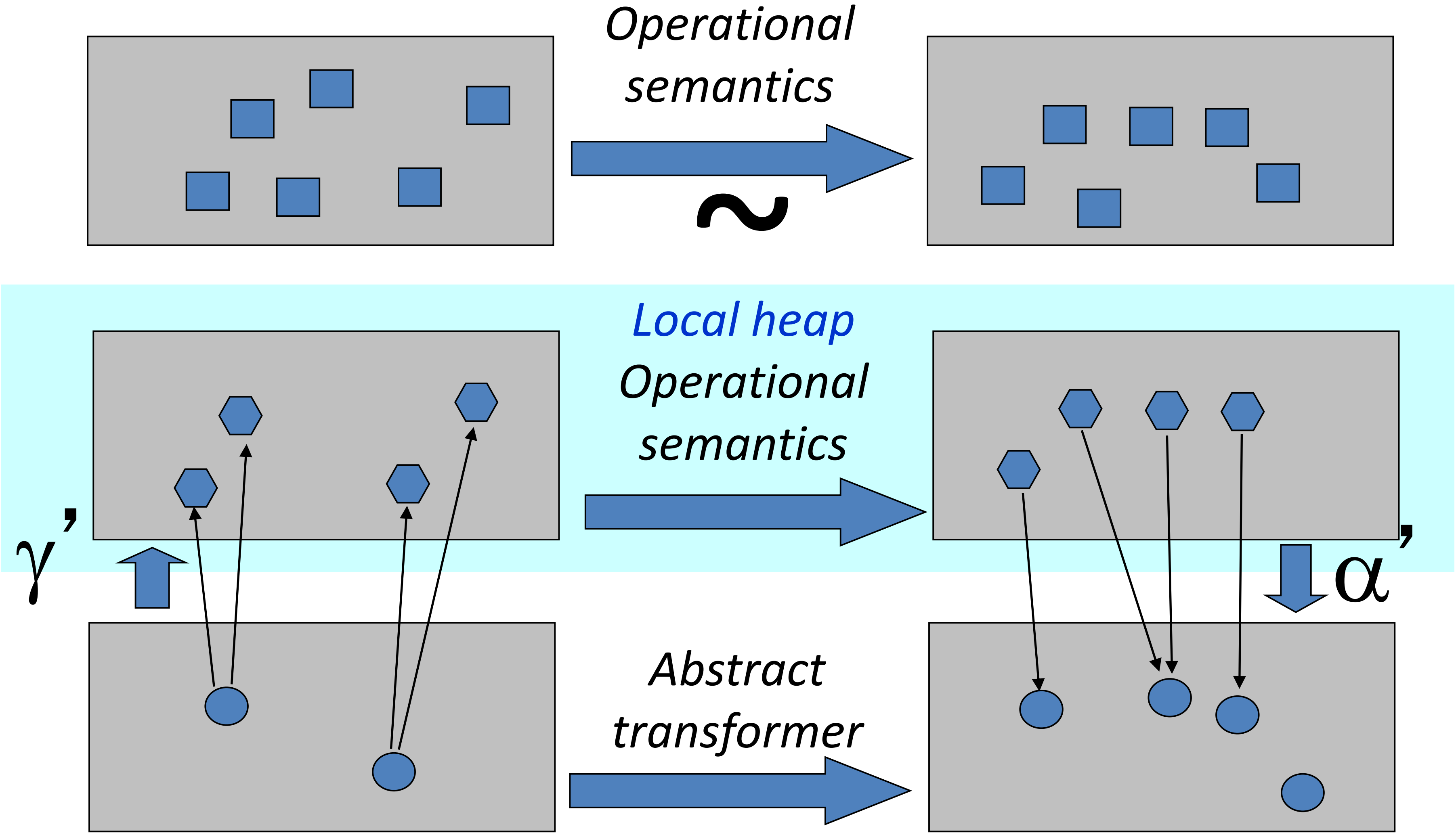
σ_{CPF} and σ_{GSB} observationally equivalent when for every access paths AP_1, AP_2

$$[\![AP_1 = AP_2]\!](\sigma_{CPF}) \Leftrightarrow [\![AP_1 = AP_2]\!](\sigma_{GSB})$$

Observational equivalence

- For cutpoint free programs:
  - $\sigma_{CPF} \in \Sigma_{CPF}$ (Cutpoint free semantics)
  - $\sigma_{GSB} \in \Sigma_{GSB}$ (Standard semantics)
  - σ_{CPF} and σ_{GSB} observationally equivalent
- It holds that
  - $\langle st, \sigma_{CPF} \rangle \rightsquigarrow \sigma'_{CPF} \Leftrightarrow \langle st, \sigma_{GSB} \rangle \rightsquigarrow \sigma'_{GSB}$
  - σ'_{CPF} and σ'_{GSB} are observationally equivalent

Introducing local heap semantics



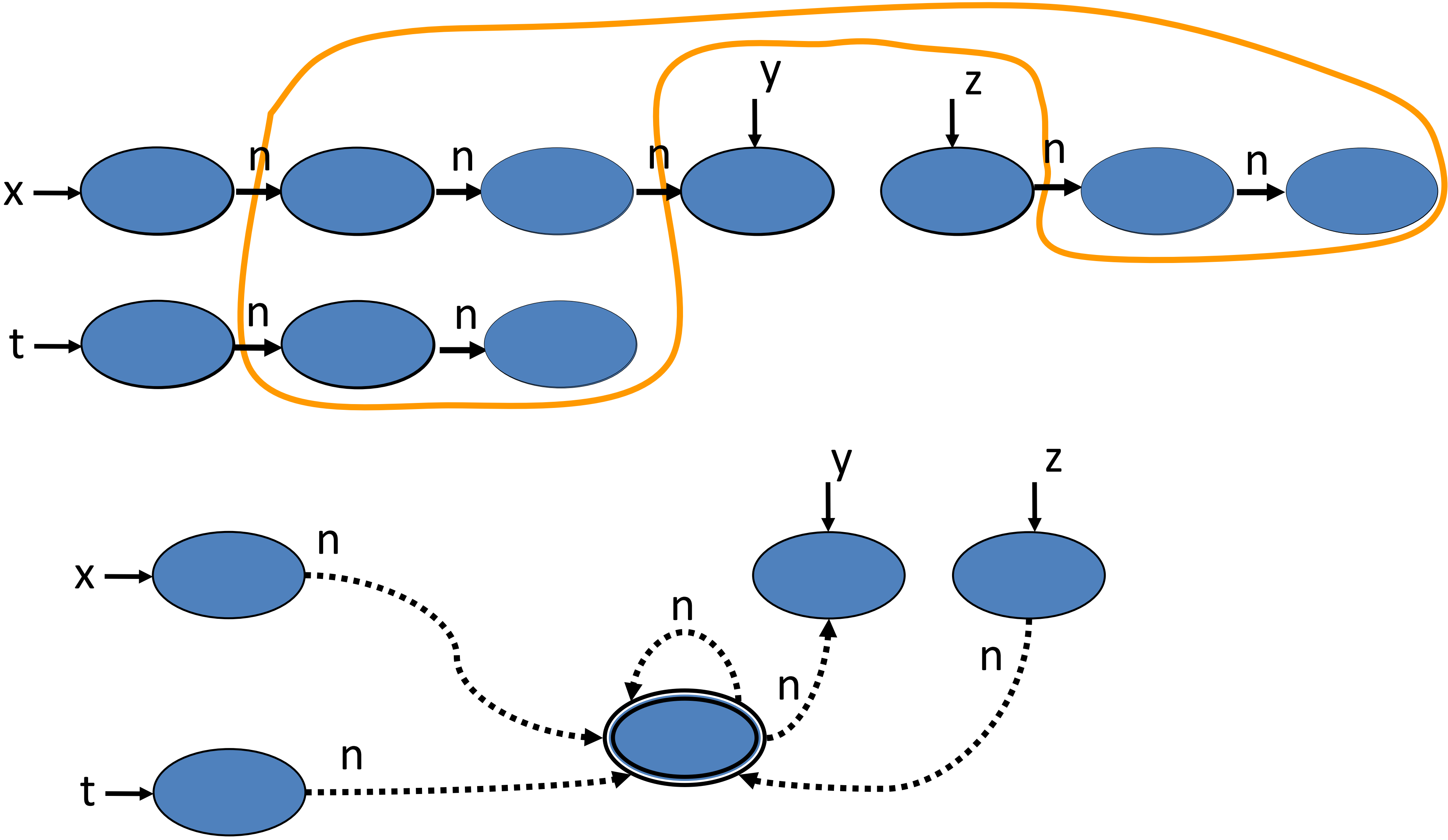
Shape abstraction

- Abstract memory states represent unbounded concrete memory states
  - Conservatively
  - In a bounded way
  - Using 3-valued logical structures

3-Valued logic

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

Canonical abstraction

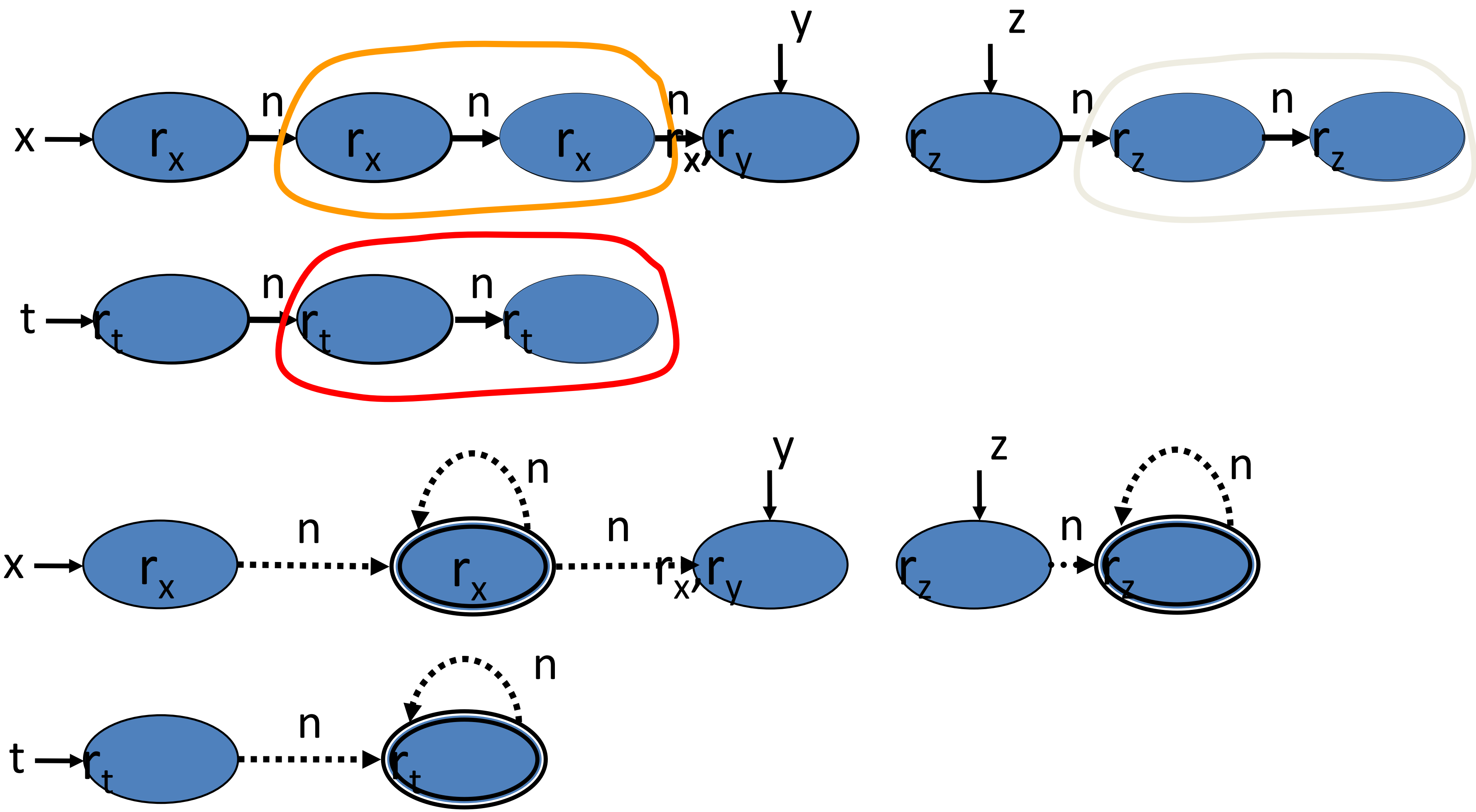


Instrumentation predicates

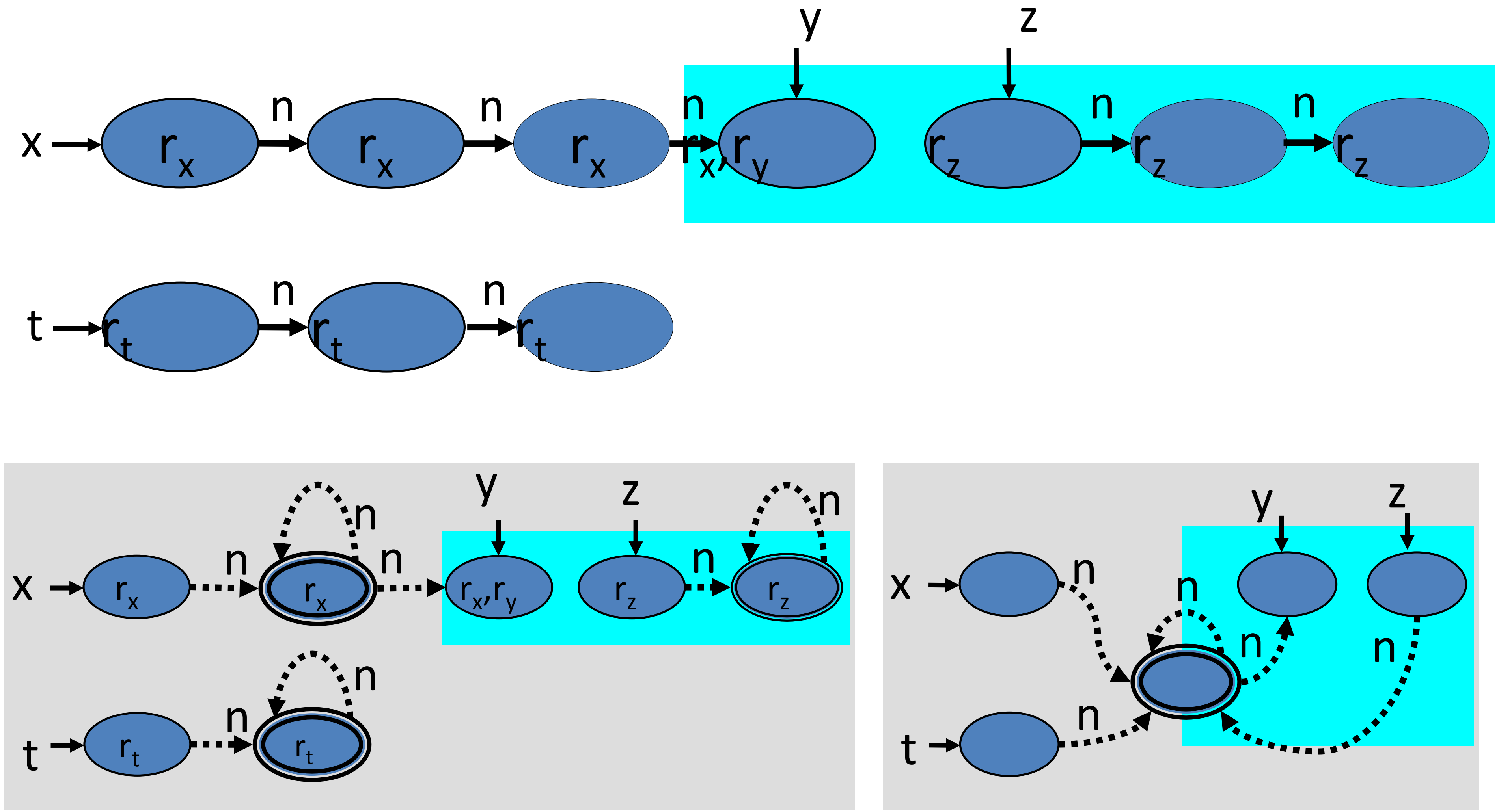
- Record derived properties
- Refine the abstraction
  - Instrumentation principle [SRW, TOPLAS’02]
- Reachability is central!

Predicate	Meaning
$r_x(v)$	v is reachable from variable x
$r_{obj}(v_1,v_2)$	v_2 is reachable from v_1
ils(v)	v is heap-shared
c(v)	v resides on a cycle

Abstract memory states (with reachability)



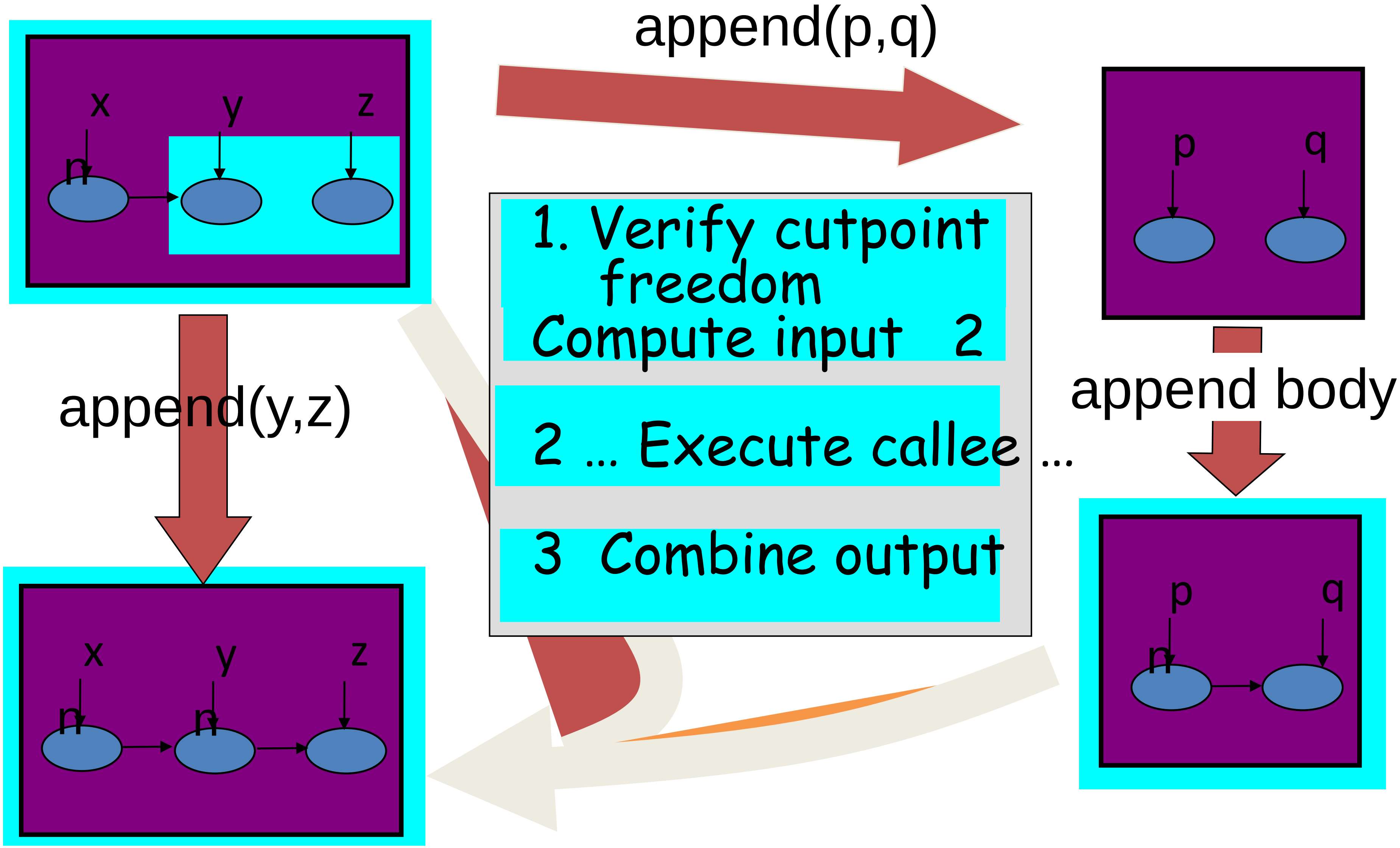
The importance of reachability:
Call append(y,z)



Abstract semantics

- Conservatively apply statements on abstract memory states
  - Same formulae as in concrete semantics
  - Soundness guaranteed [SRW, TOPLAS'02]

Procedure calls



1. Verify cutpoint freedom

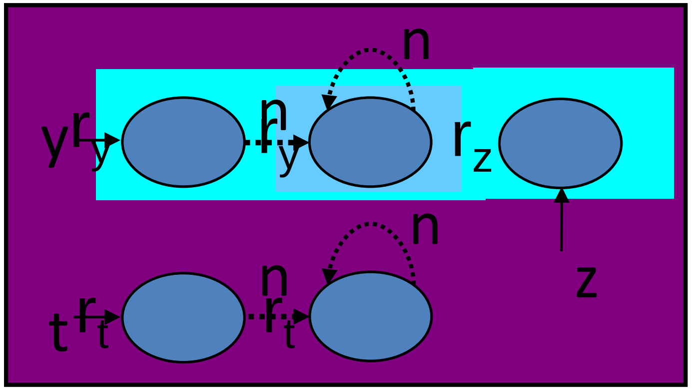
Compute input 2

2 … Execute callee …

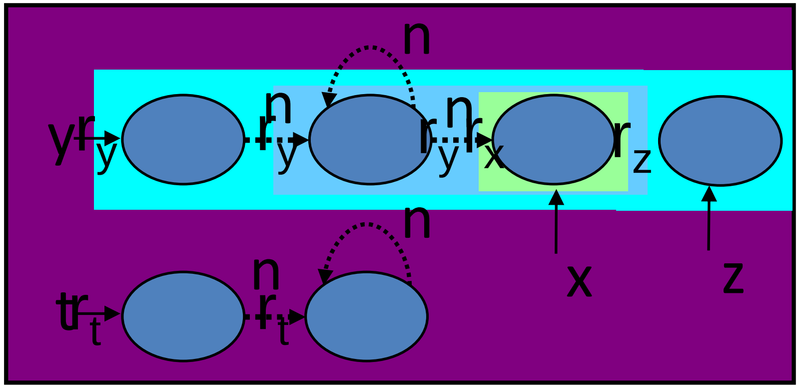
3 Combine output

Conservative verification of cutpoint-freedom

- Invoking append(y,z) in main
  - $R_{\{y,z\}}(v)=\exists v_1:y(v_1)\wedge n^*(v_1,v) \vee \exists v_1:z(v_1)\wedge n^*(v_1,v)$
  - $isCP_{main,\{y,z\}}(v)= R_{\{y,z\}}(v) \wedge (\neg y(v)\wedge\neg z(v_1)) \wedge (x(v) \vee t(v) \vee \exists v_1: \neg R_{\{y,z\}}(v_1)\wedge n(v_1,v))$

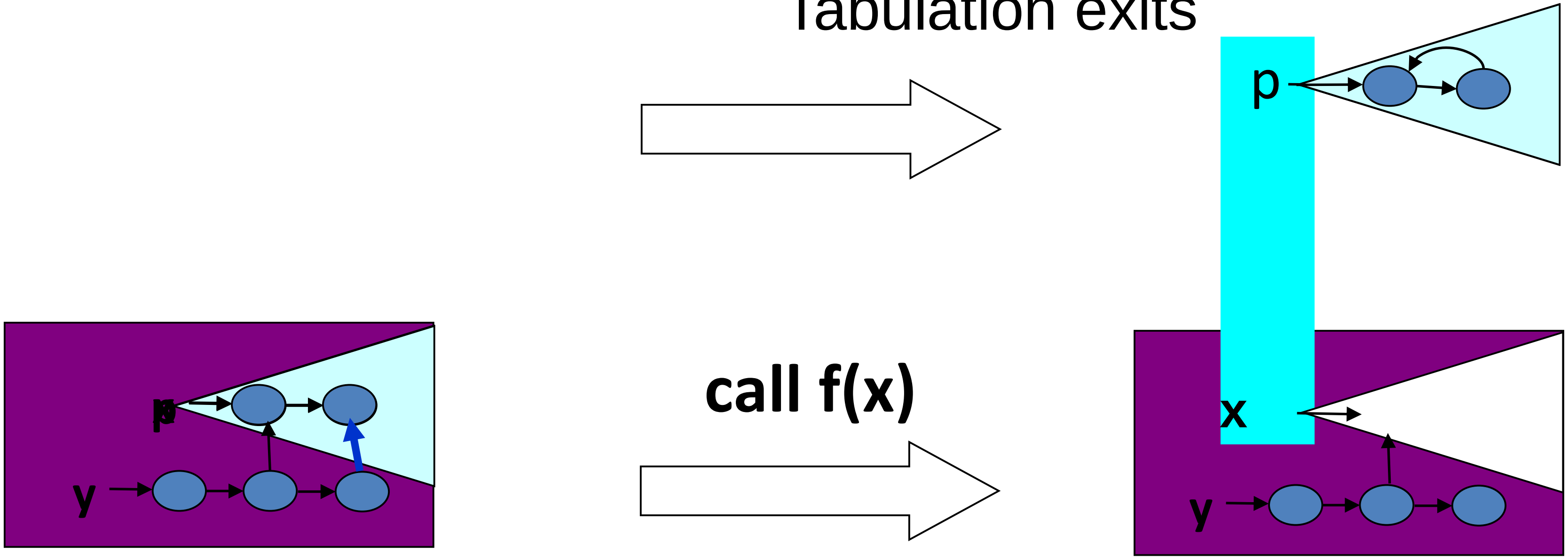


Cutpoint free



Not Cutpoint free

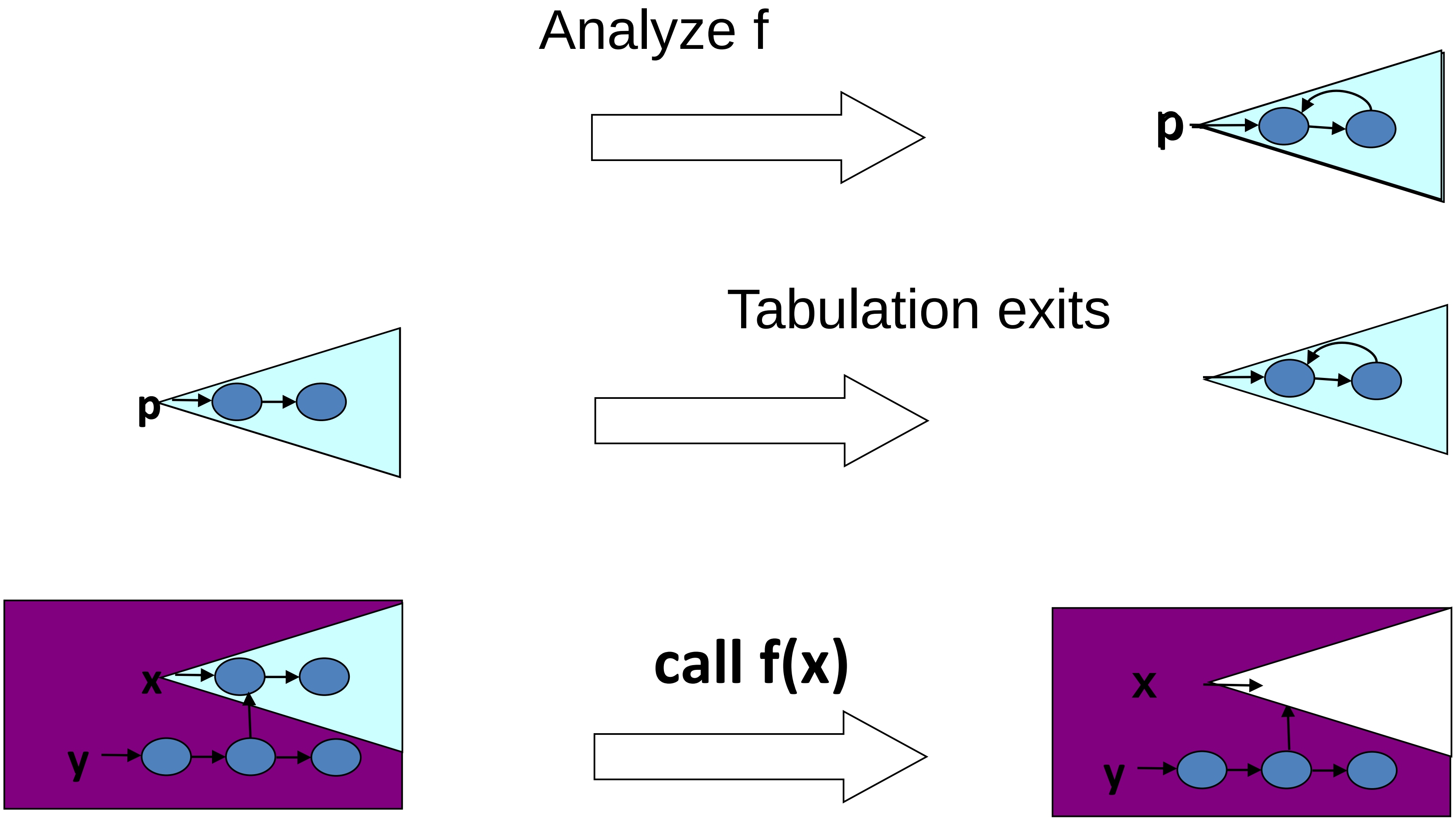
Interprocedural shape analysis



Tabulation exits

call f(x)

Interprocedural shape analysis



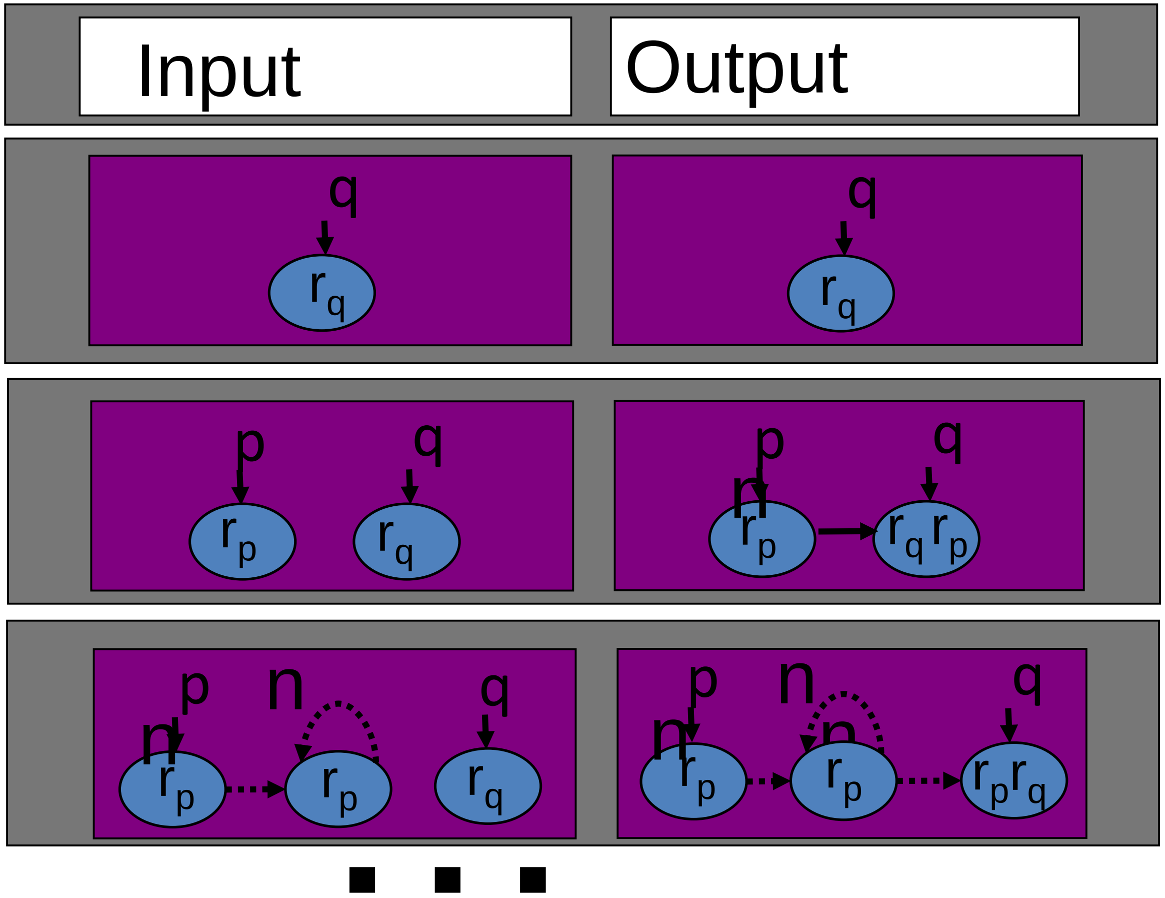
Analyze f

Tabulation exits

call f(x)

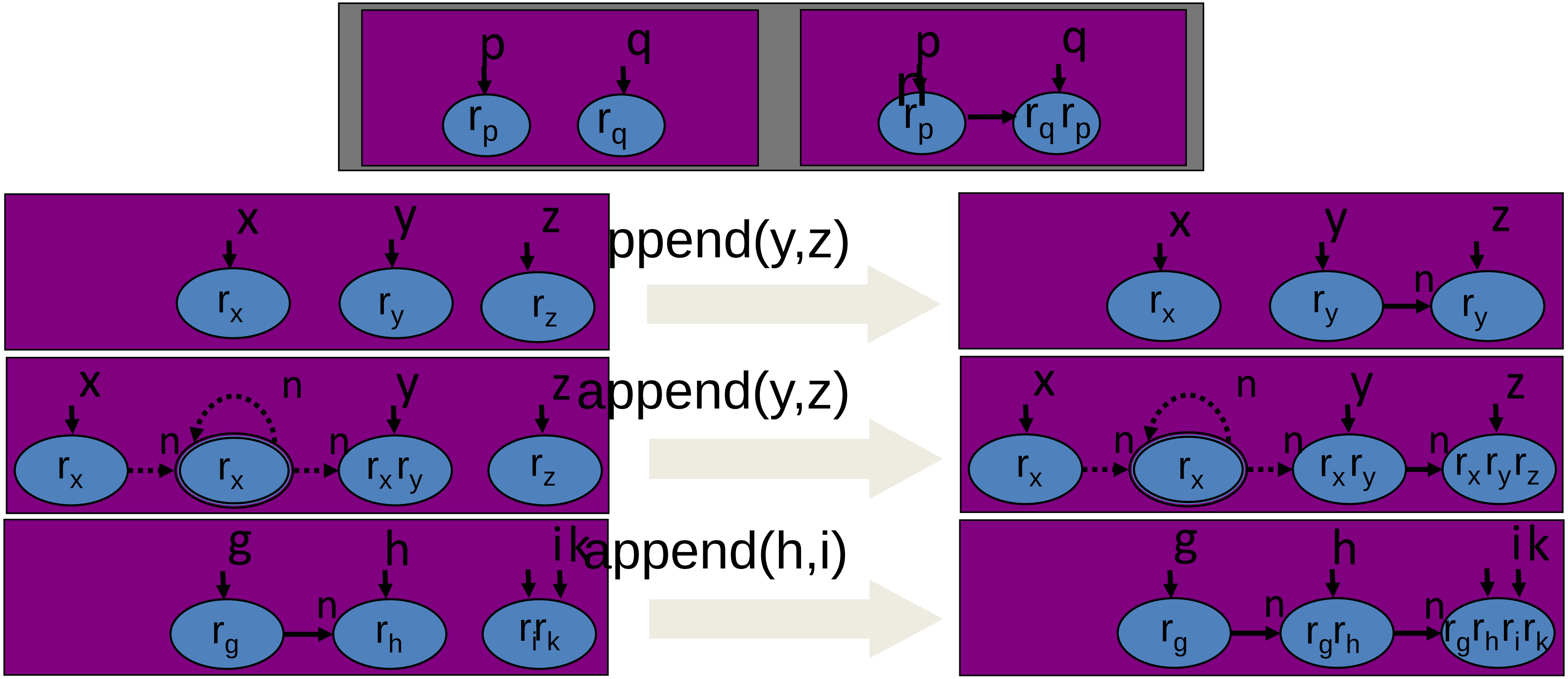
Interprocedural shape analysis

- Procedure ≡ input/output relation



Interprocedural shape analysis

- Reusable procedure summaries
  - Heap modularity



Plan

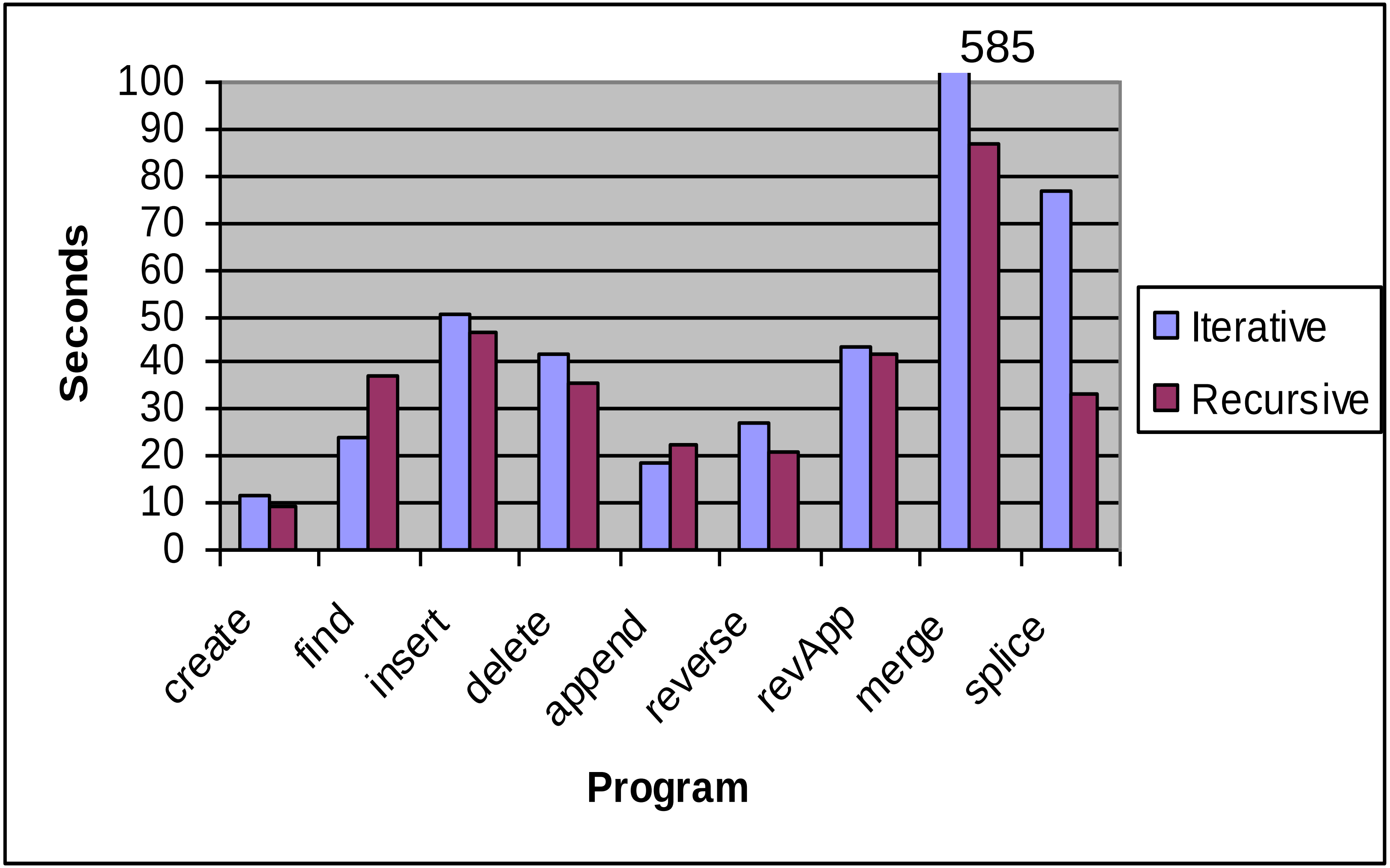
- ✓ Cutpoint freedom
- ✓ Non-standard concrete semantics
- ✓ Interprocedural shape analysis
- Prototype implementation

Prototype implementation

- TVLA based analyzer
- Soot-based Java front-end
- Parametric abstraction

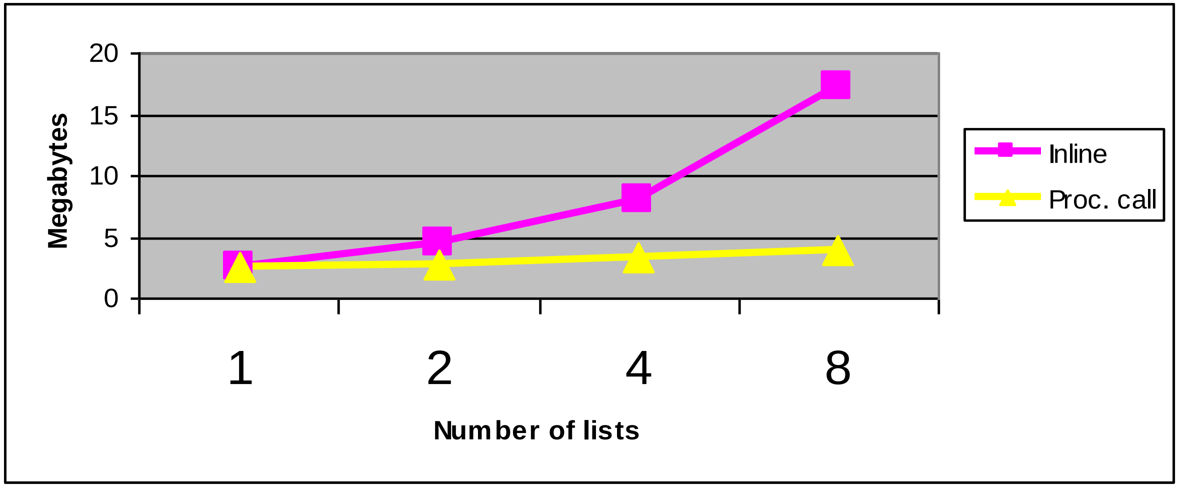
Data structure	**Verified properties**
Singly linked list	Cleanness, acyclicity
Sorting (of SLL)	+ Sortedness
Unshared binary trees	Cleaness, tree-ness

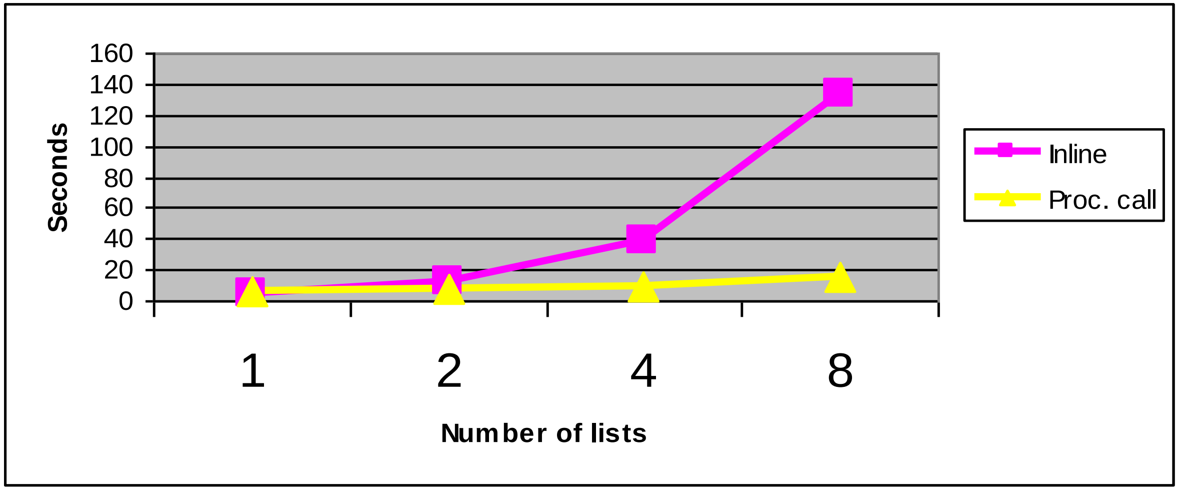
Iterative vs. Recursive (SLL)



Inline vs. Procedural abstraction

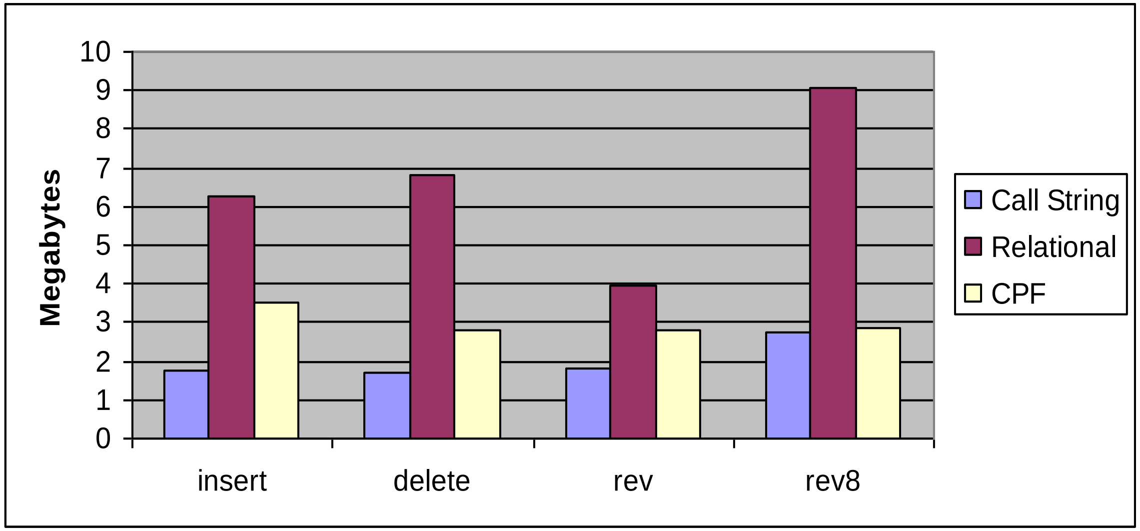
```
// Allocates a list of
        // length 3
    List create3(){
            …
            }

        main() {
List x1 = create3();
List x2 = create3();
List x3 = create3();
List x4 = create3();
            …
            }
```

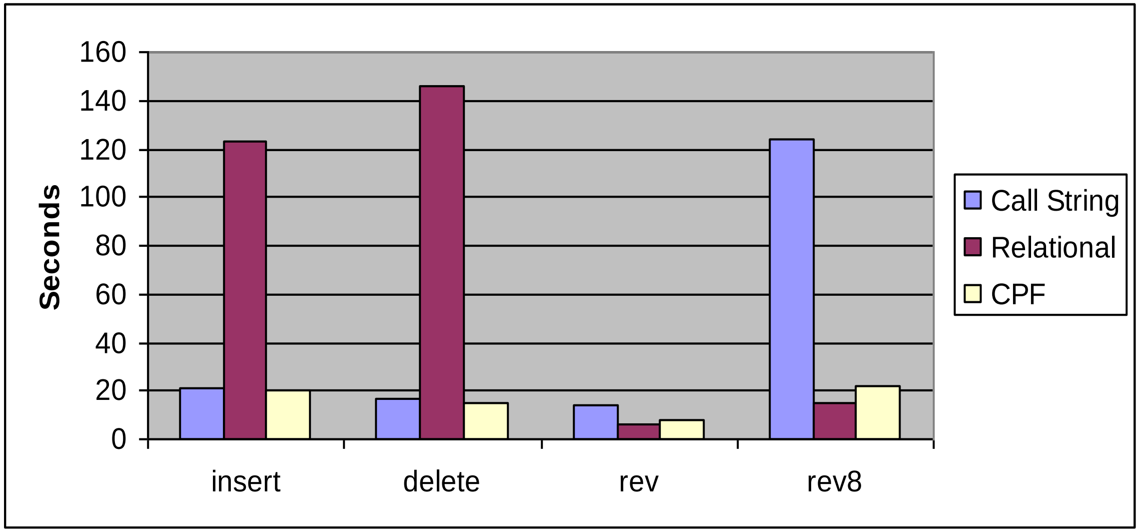




Call string vs. Relational vs. CPF

[Rinetzky and Sagiv, CC' 01] [Jeannet et al., SAS' 04]





Summary

- Cutpoint freedom
- Non-standard operational semantics
- Interprocedural shape analysis
  - Partial correctness of quicksort
- Prototype implementation

Application

- Properties proved
  - Absence of null dereferences
  - Listness preservation
  - API conformance
- Recursive ≈ Iterative
- Procedural abstraction

Related Work

- **Interprocedural shape analysis**
  - Rinetzky and Sagiv, CC ’01
  - Chong and Rugina, SAS ’03
  - Jeannet et al., SAS ’04
  - Hackett and Rugina, POPL ’05
  - Rinetzky et al., POPL ‘05
- **Local Reasoning**
  - Ishtiaq and O’Hearn, POPL ‘01
  - Reynolds, LICS ’02
- **Encapsulation**
  - Noble et al. IWACO ’03
  - …

Summary

- Operational semantics
  - Storeless
  - Local heap
  - Cutpoints
  - Equivalence theorem
- Applications
  - Shape analysis
  - May-alias analysis

Project

- 1-2 Students in a group
  - 3-4: Bigger projects
- Theoretical + Practical
- Your choice of topic
  - Contact me in 2 weeks
- Submission – 15/Sep
  - Code + Examples
  - Document
  - 20 minutes presentation

Past projects

- JavaScript Dominator Analysis
- Attribute Analysis for JavaScript
- Simple Pointer Analysis for C
- Adding program counters to Past Abstraction
- Verification of Asynchronous programs
- Verifying SDNs using TVLA
- Verifying independent accesses to arrays in GO

Past projects

- Detecting index out of bound errors in C programs
- Lattice-Based Semantics for Combinatorial Models Evolution