# Program Analysis and Verification

0368-4479

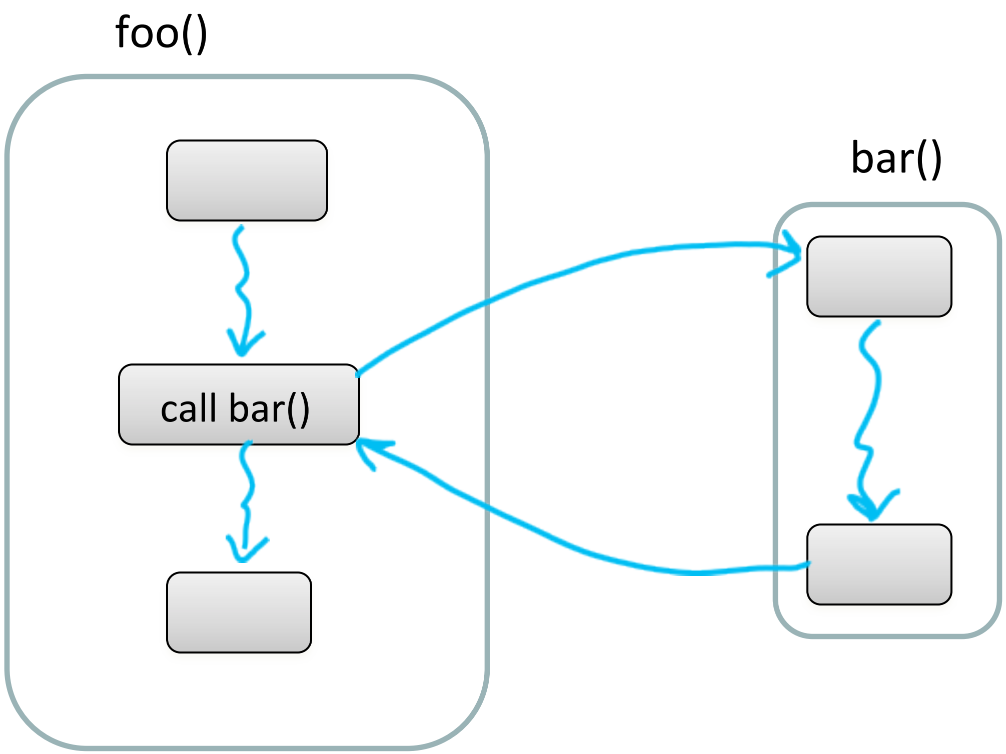
http://www.cs.tau.ac.il/~maon/teaching/2013-2014/paav/paav1314b.html

Noam Rinetzky

## Lecture 13: Interprocedural Shape Analysis

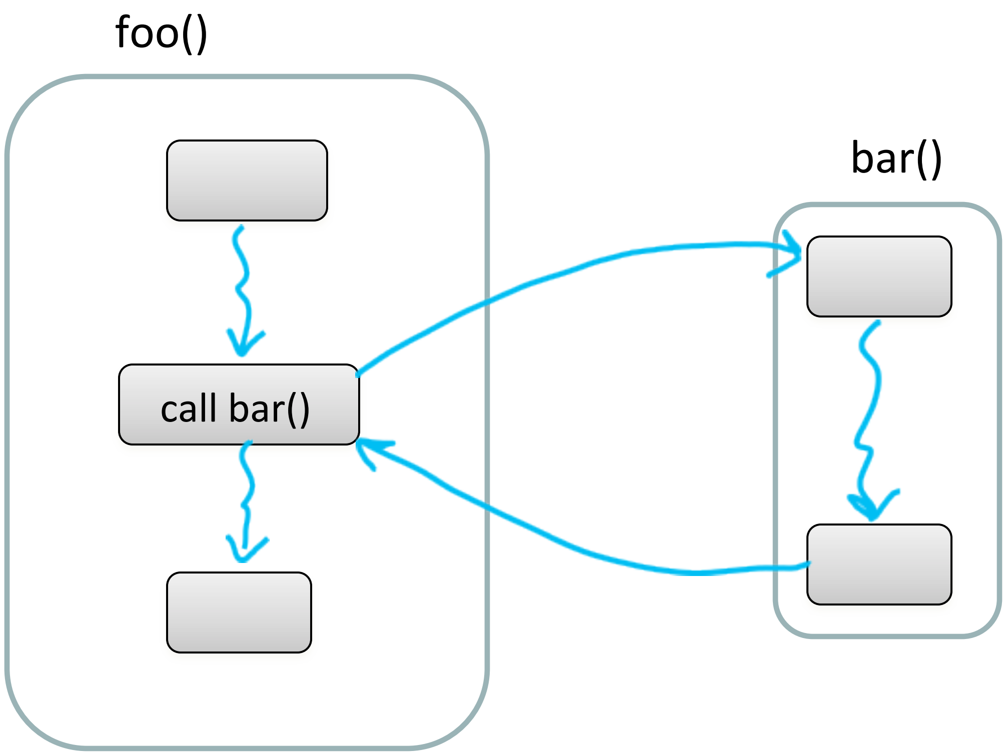
Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

# 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

# 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

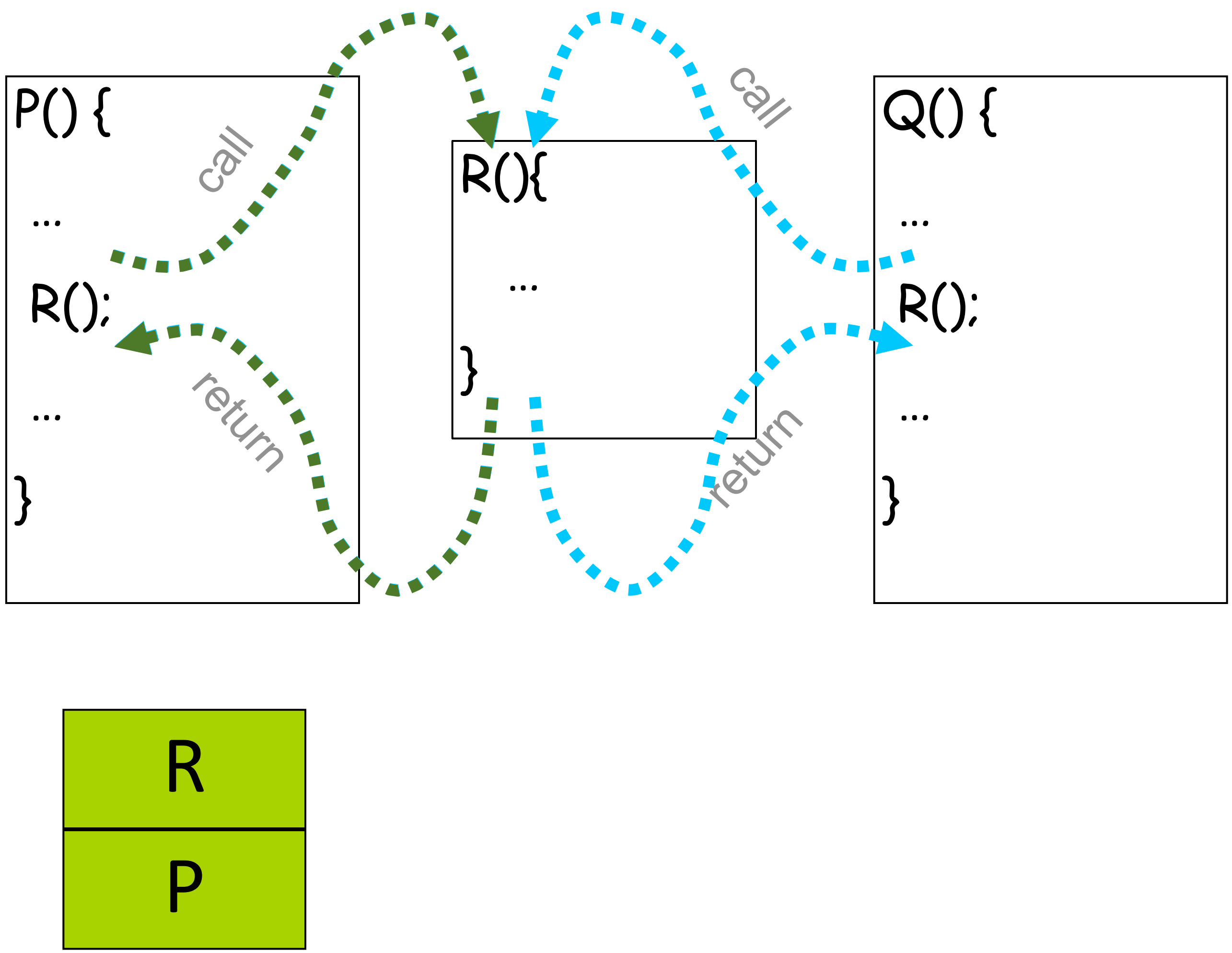
# Reduction to intraprocedural analysis

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

# Interprocedural analysis: Guiding light

- Exploit stack regime
  - ➔Precision
  - ➔Efficiency

# Stack regime



# Simplifying Assumptions

- Parameter passed by value
- No procedure nesting
- No concurrency

✓ Recursion is supported

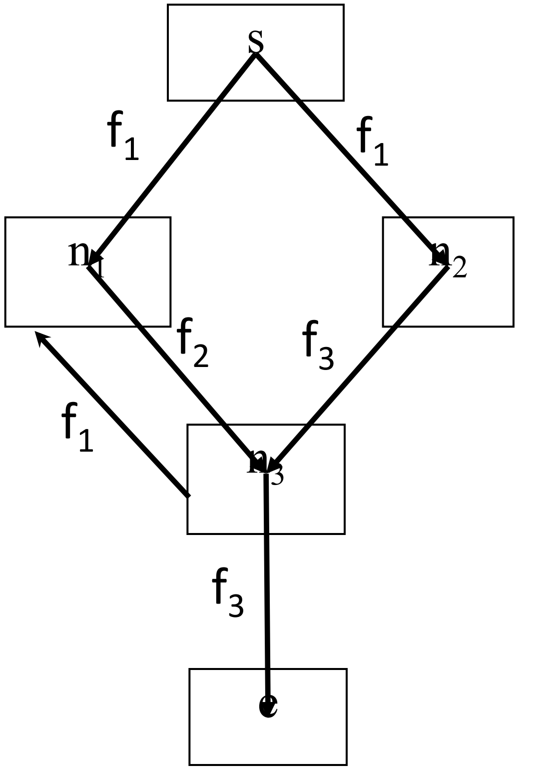
# The collecting lattice

- Lattice for a given control-flow node *v*:
  $L_v=(2^{\mathbf{State}}, \subseteq, \cup, \cap, \emptyset, \mathbf{State})$
- Lattice for entire control-flow graph with nodes *V*:

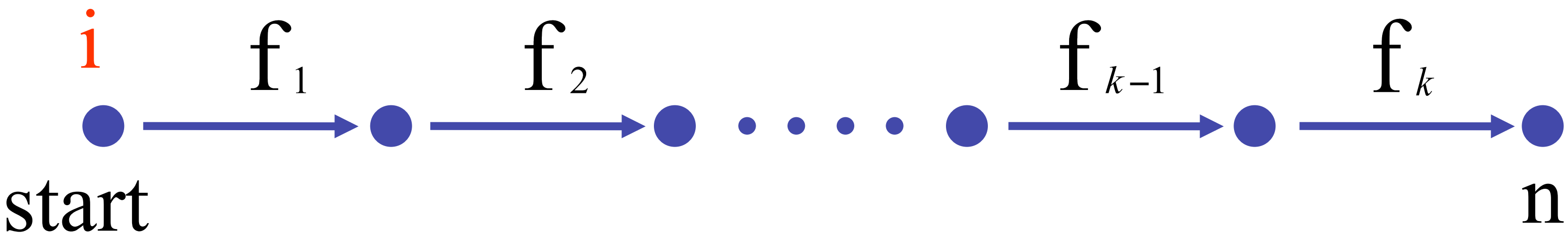
  $$L_{CFG} = Map(V, L_v)$$
- We will use this lattice as a baseline for static analysis and define abstractions of its elements

# Paths

- paths(n) the set of paths from s to n
  - ( $(s,n_1)$, $(n_1,n_3)$, $(n_3,n_1)$ )



# Join Over All Paths (JOP)



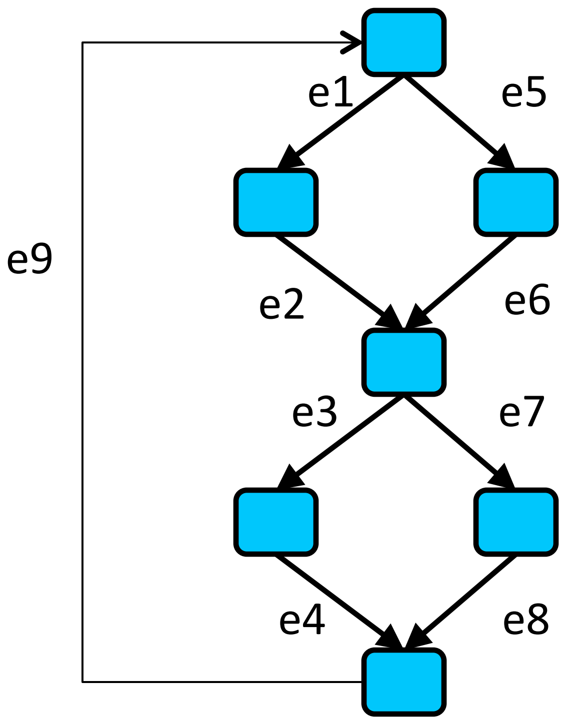

$$[\![ f_k \circ \ldots \circ f_1 ]\!] \in L \rightarrow L$$

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

# 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) $\sqcup$
f[e1,e2,e7,e8](initial) $\sqcup$
f[e5,e6,e7,e8](initial) $\sqcup$
f[e5,e6,e3,e4](initial) $\sqcup$ …

Number of program paths is unbounded due to loops

# The lfp computation approximates JOP

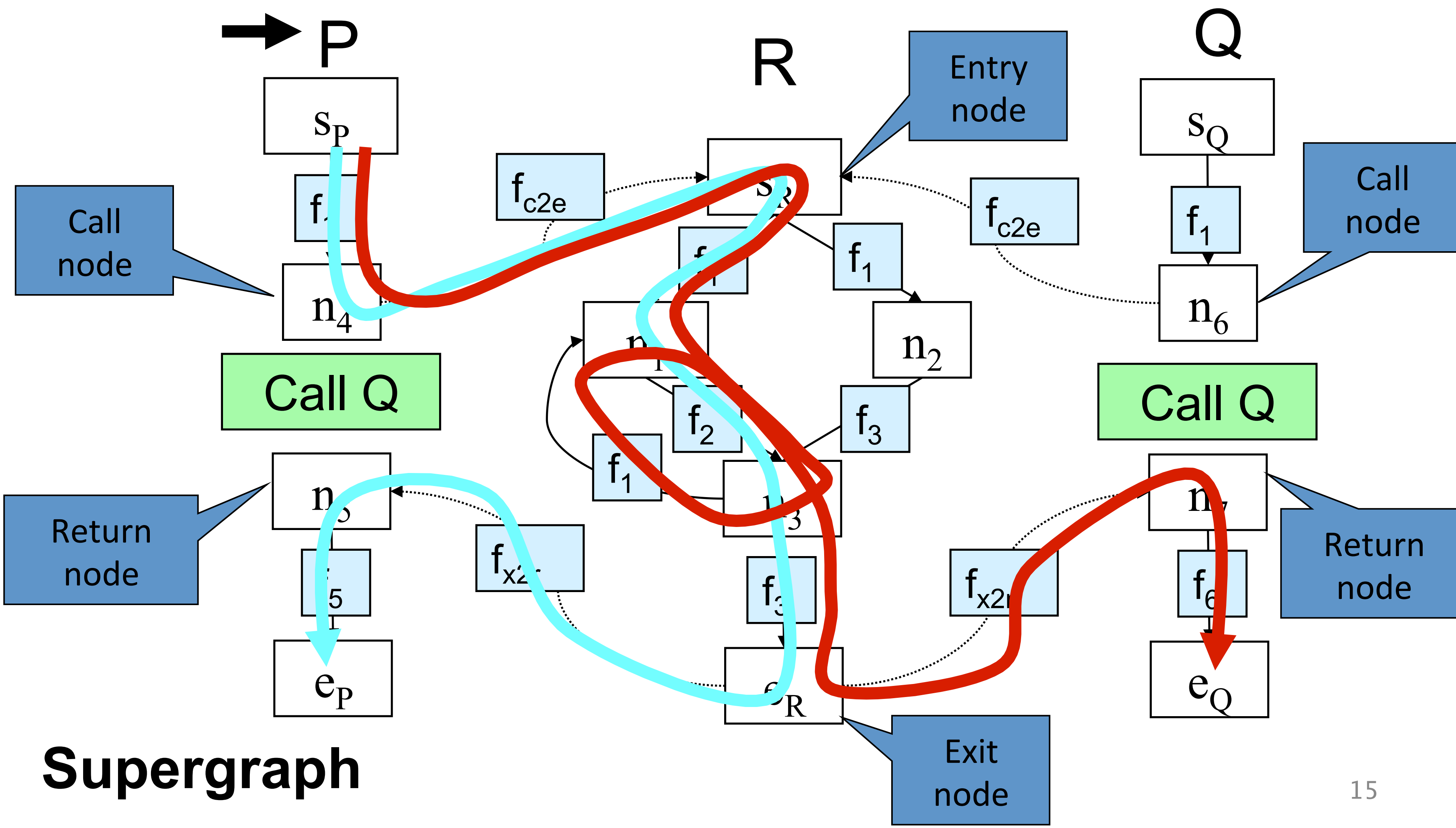
- JOP[v] $= \sqcup\{f\,[e_1, e_2, \ldots, e_n](\iota) \mid [e_1, e_2, \ldots, e_n] \in \text{paths}(v)\}$
- LFP[v] $= \sqcup\{f\,[e](\text{LFP}[v']) \mid e = (v', v)\}$

  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
  - $f(x \sqcup y) = f(x) \sqcup f(y)$

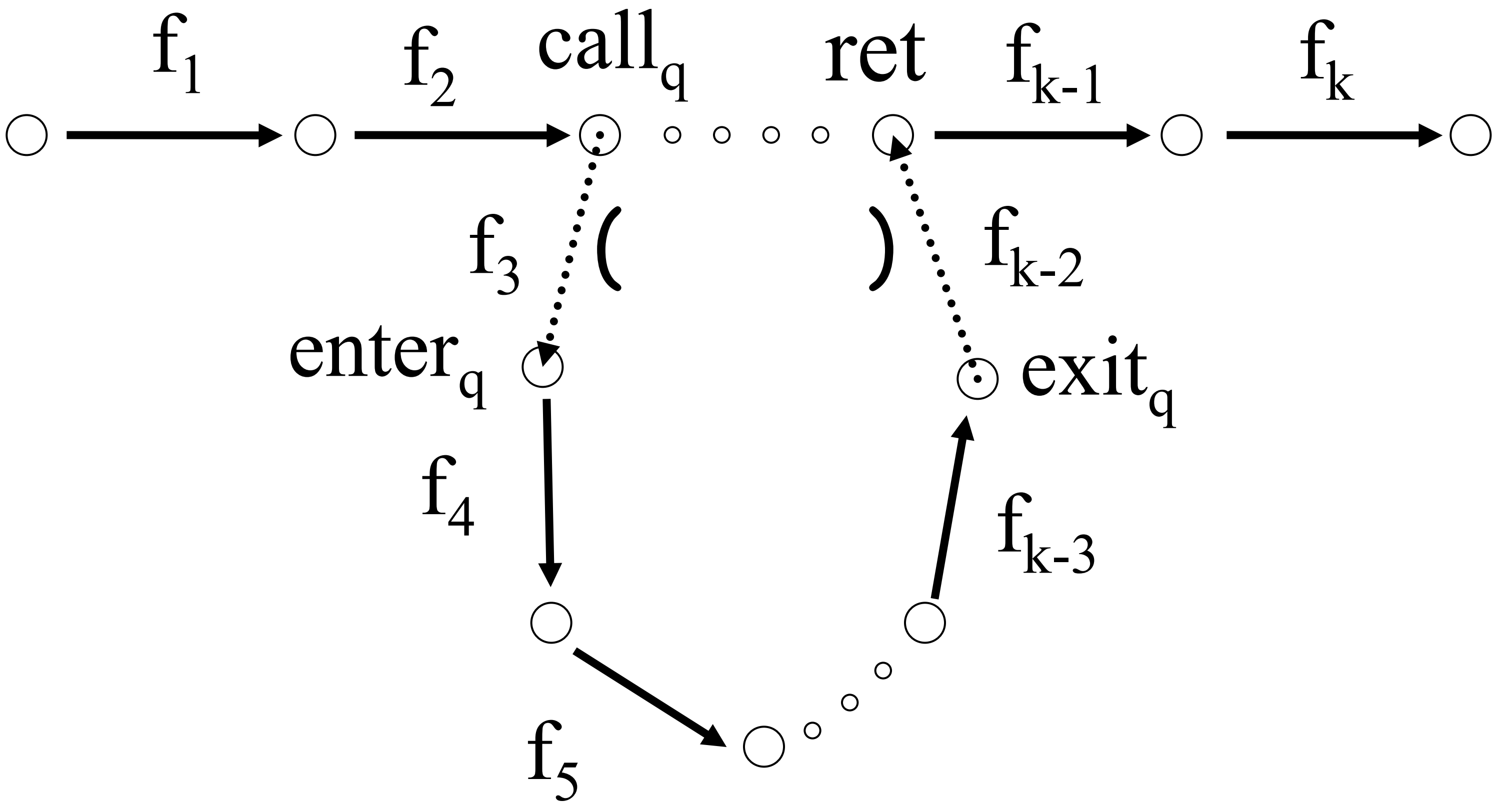
JOP may not be precise enough for interprocedural analysis!

# Interprocedural analysis




**Supergraph**

# 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

# 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

# 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);
```

$[a \mapsto a_0\text{-}1, x \mapsto \text{-}2a_0+7]$

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

$[a \mapsto a_0, x \mapsto \text{-}2a_0+7]$

```
 }
```

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

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

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

```
}
```

# 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 ↦⊤]
}
```

# 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: $S[n',x] = S[n',x] \sqcup [\![n]\!](y))$
  - n intra-node: ➔ n' : $(x, [\![n]\!](y))$
  - 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', $S[n_c,u]=x$

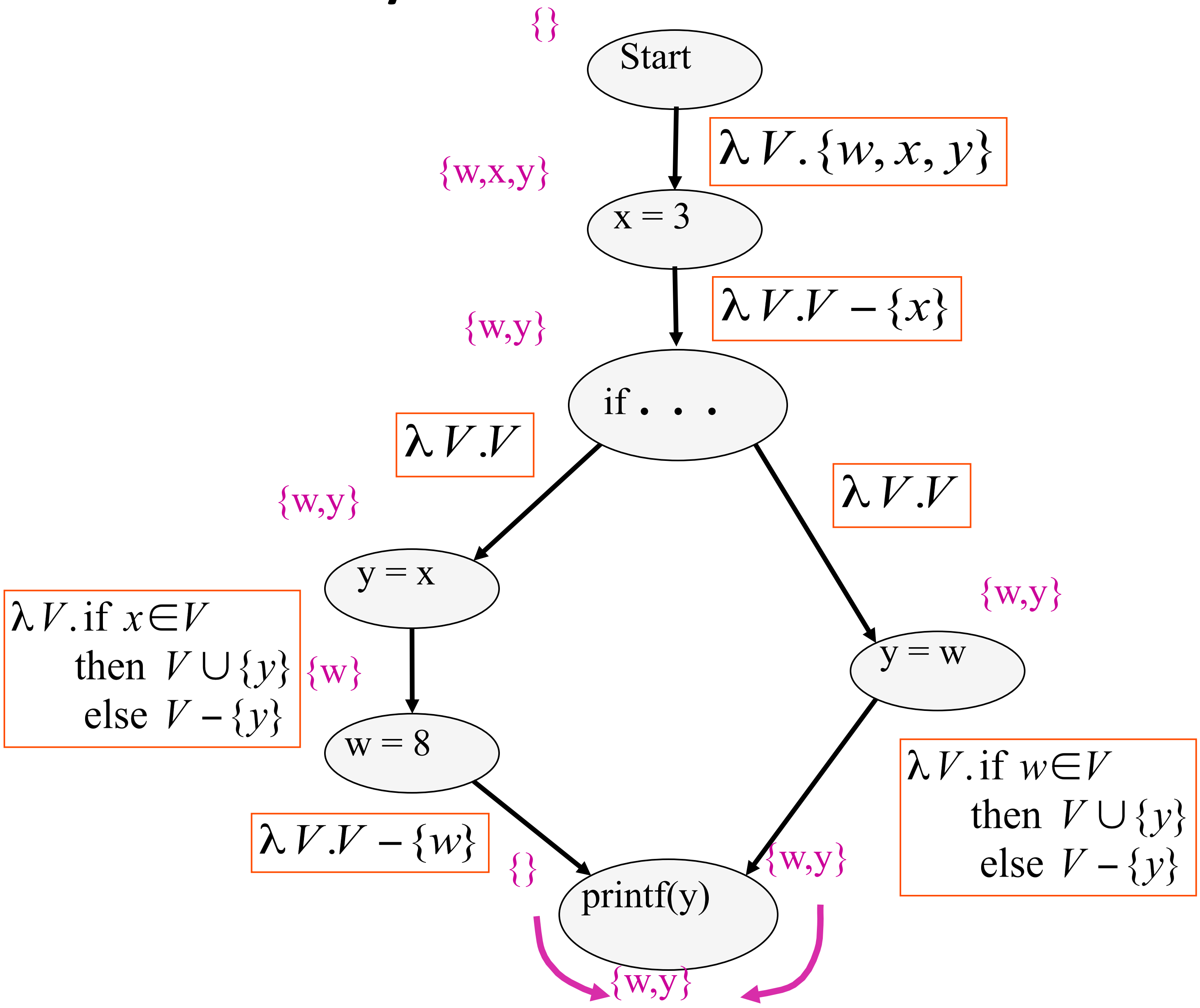
# CFL-Graph reachability

- 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



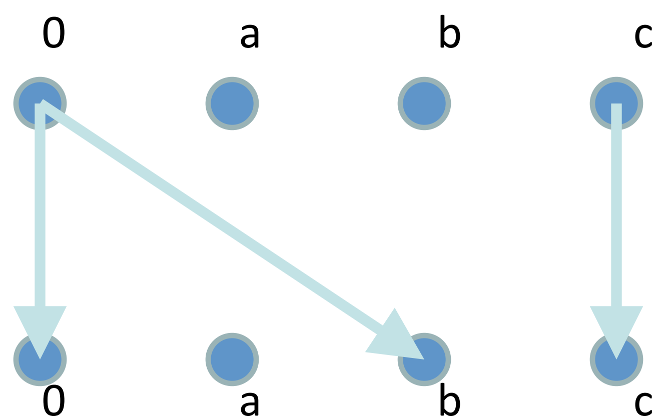
# IFDS Problems

- 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 $\Lambda$)
- Example: D = { a, b, c }
  f(S) = (S – {a}) U {b}




207

# Efficiently Representing Functions

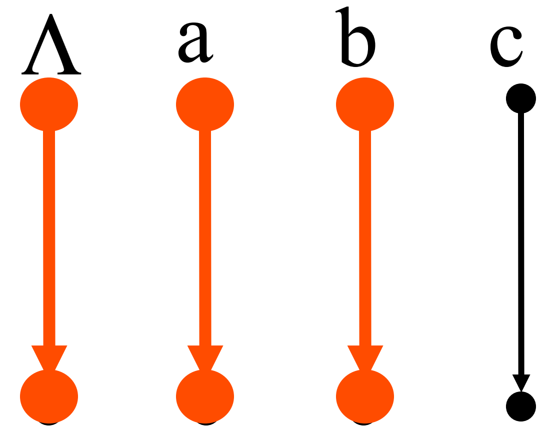
- Let $f: 2^D \rightarrow 2^D$ be a distributive function
- Then:
  - $f(X) = f(\emptyset) \cup (\cup \{ f(\{z\}) \mid z \in X \})$
  - $f(X) = f(\emptyset) \cup (\cup \{ f(\{z\}) \setminus f(\emptyset) \mid z \in X \})$

# Representing Dataflow Functions

Identity Function

$f = \lambda V.V$

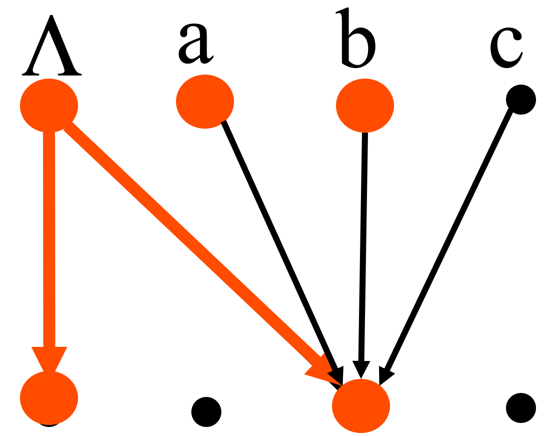
$f(\{a,b\}) = \{a,b\}$




Constant Function

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

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

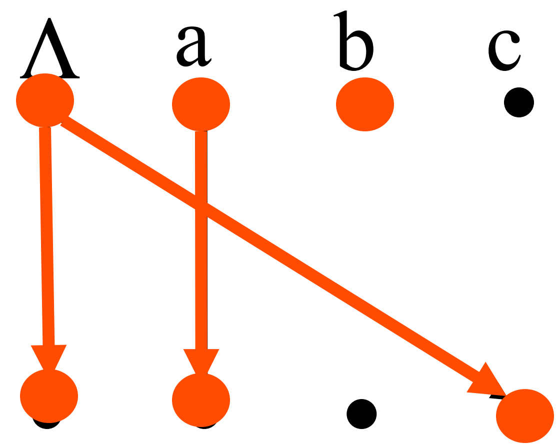


# Representing Dataflow Functions

"Gen/Kill" Function

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

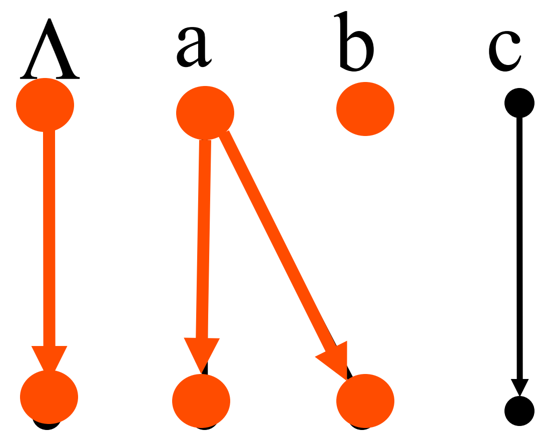
$$f(\{a,b\}) = \{a,c\}$$

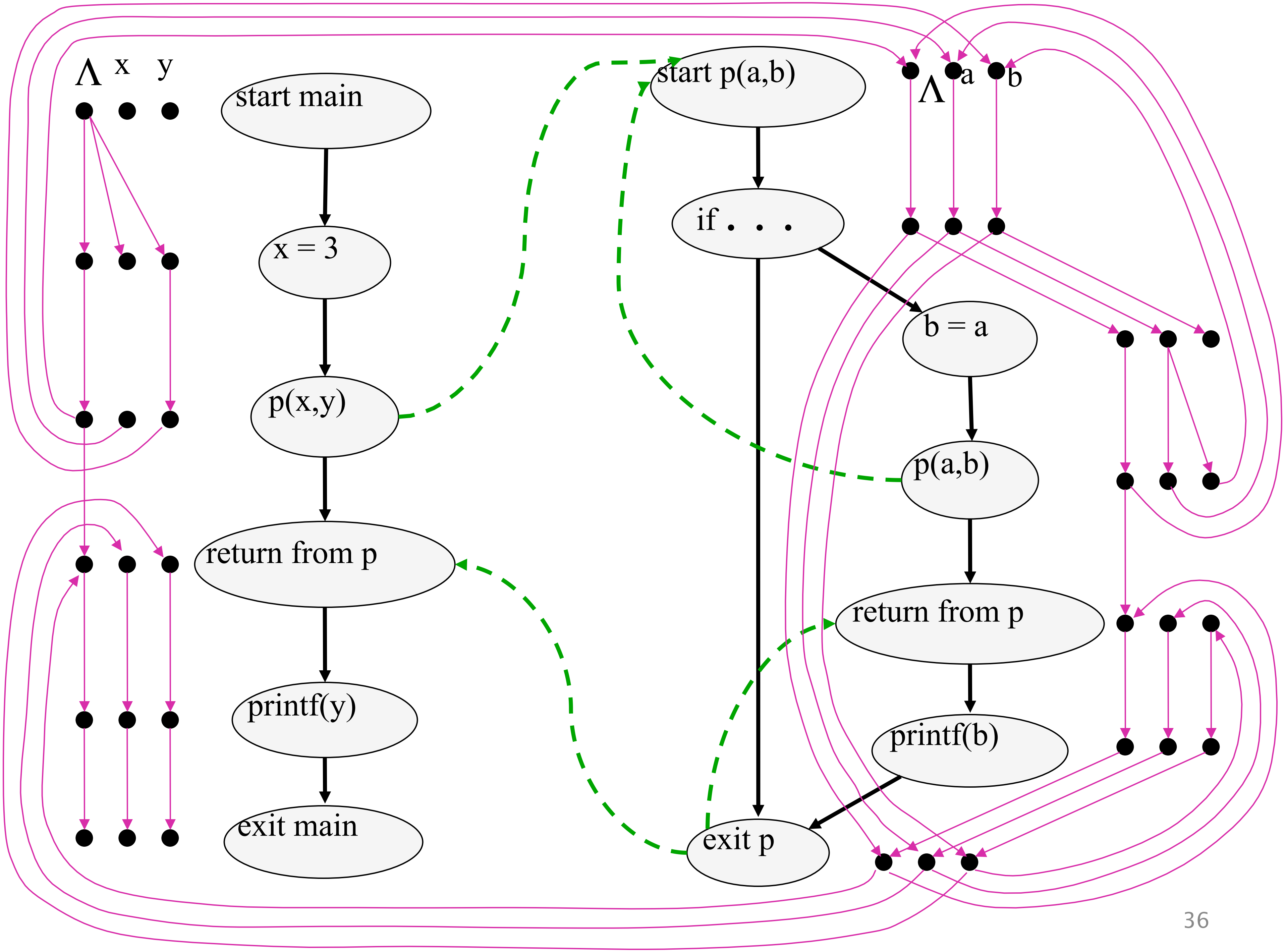



Non-"Gen/Kill" Function

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

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

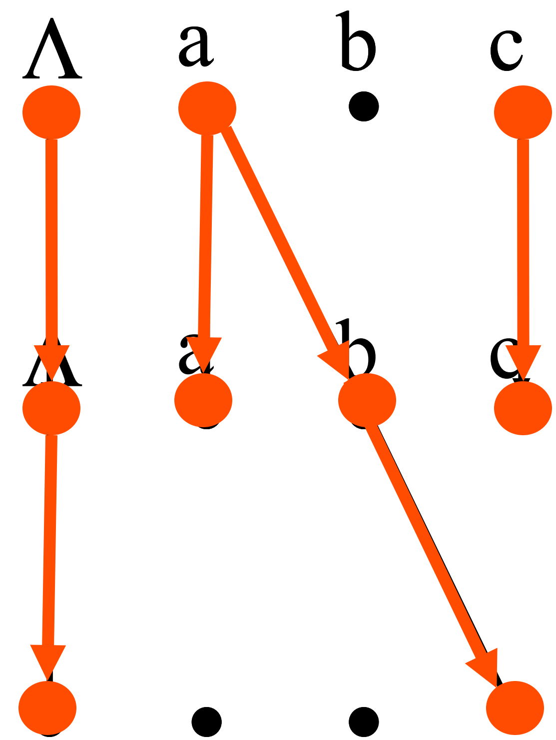



Λ
x
y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
start p(a,b)
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
Λ
a
b

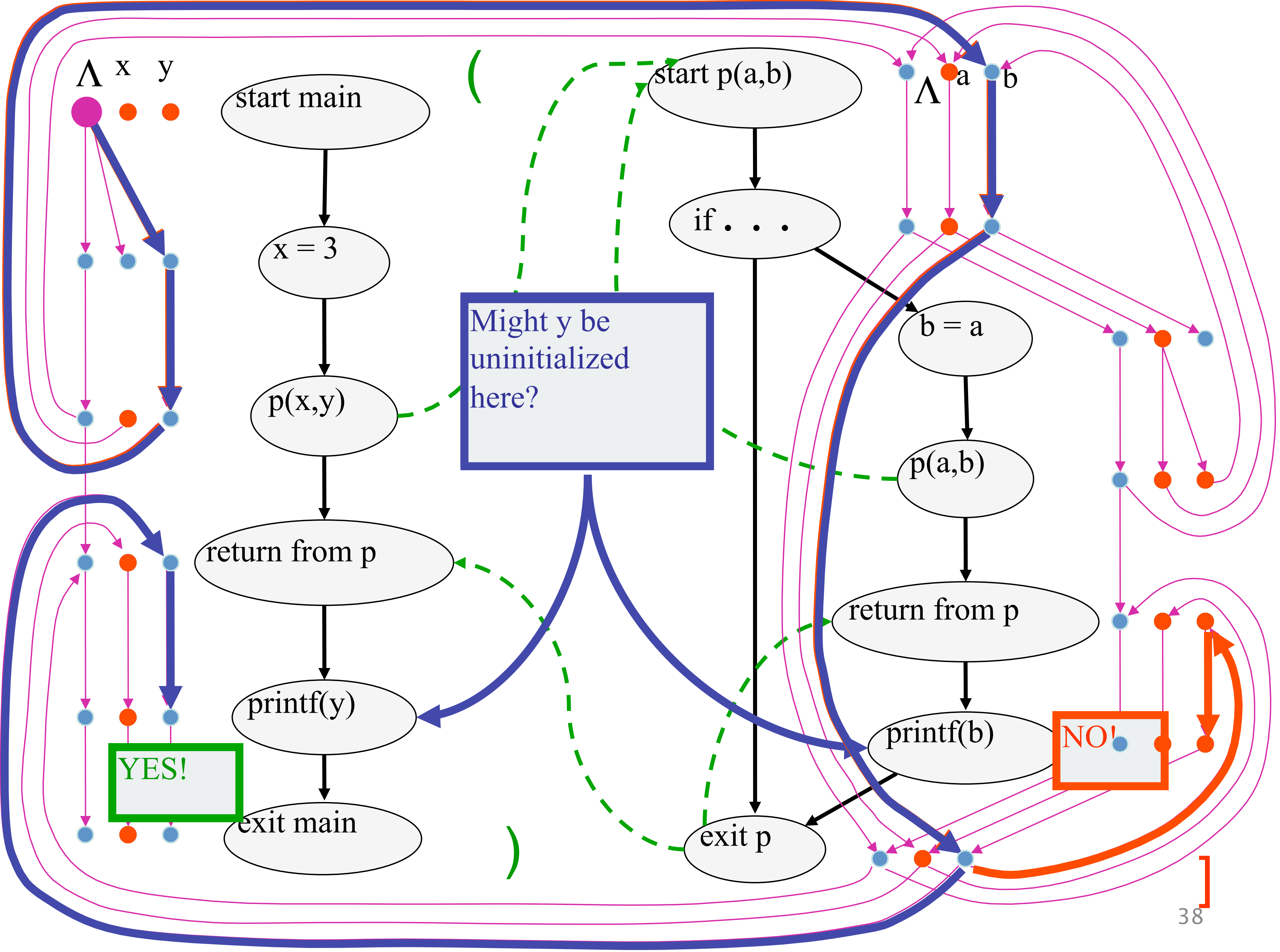
# Composing Dataflow Functions

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

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




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



Λ x y
start main
x = 3
p(x,y)
return from p
printf(y)
exit main
YES!
Might y be uninitialized here?
start p(a,b)
Λ a b
if . . .
b = a
p(a,b)
return from p
printf(b)
exit p
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

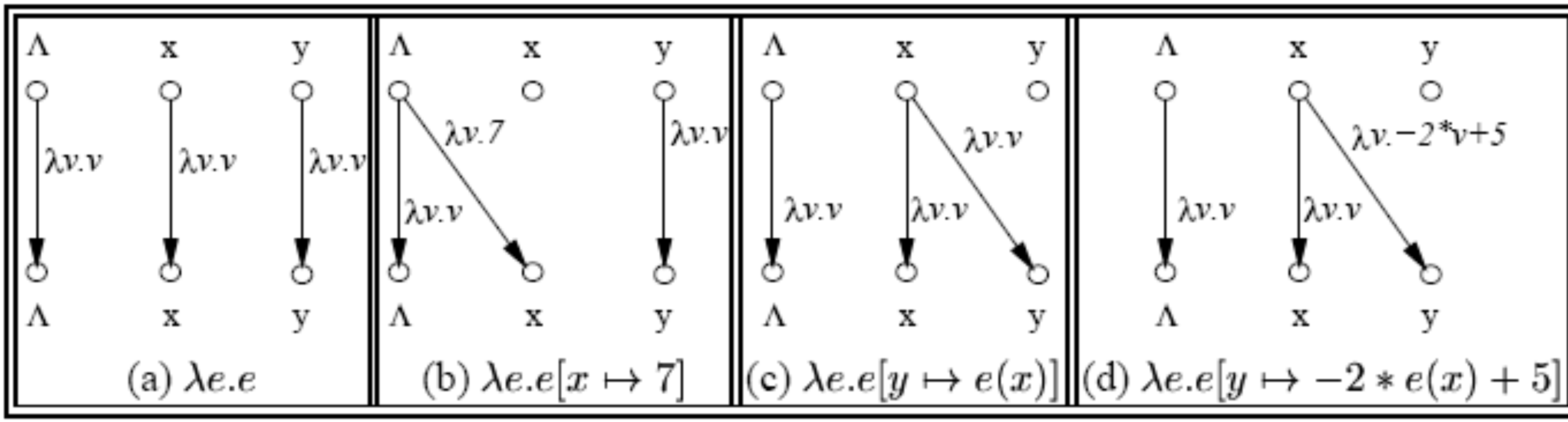
# Example Linear Constant Propagation

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

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

  $a_x, c \in Z \cup \{\bot, \top\} \quad b_x \in Z$
- Supports efficient composition and "functional" join
  - [z := a * y + b]
  - What about [z:=x+y]?

# Linear constant propagation

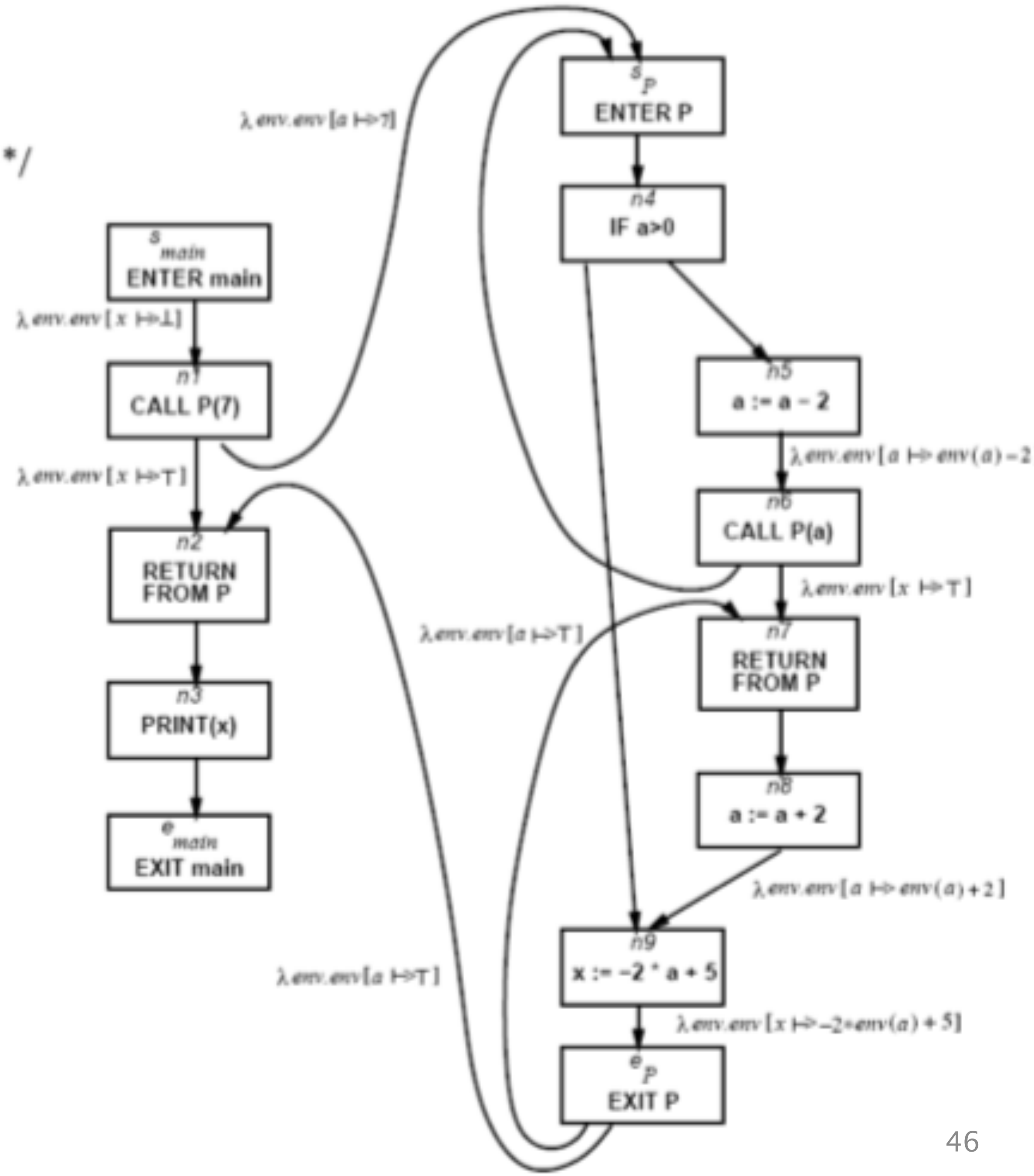


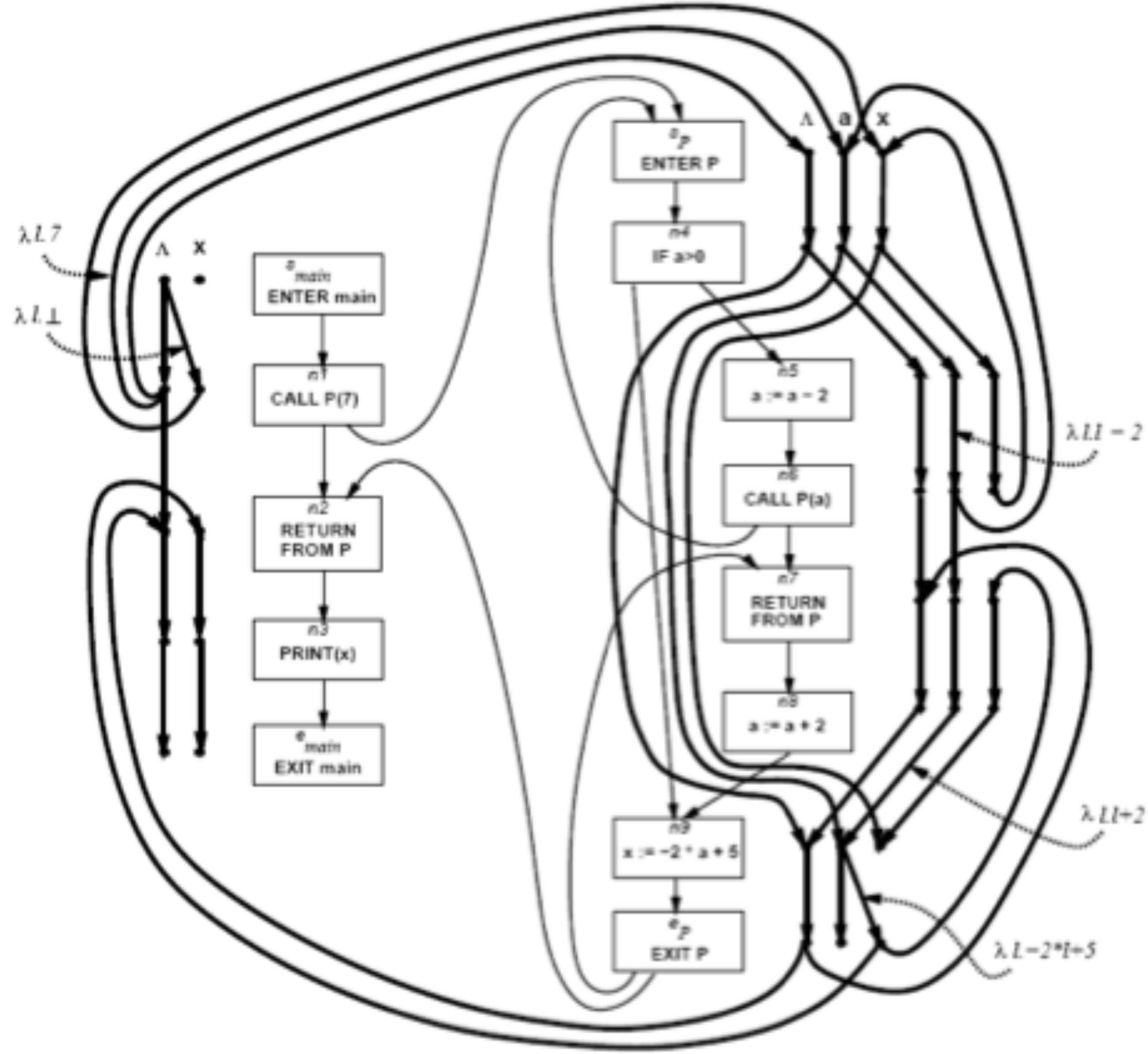

Point-wise representation of environment transformers

# 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
$e_{main}$
ENTER main
n1
CALL P(7)
n2
RETURN FROM P
n3
PRINT(x)
$e_{main}$
EXIT main
$e_P$
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

# 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

# Disadvantages of the trivial solution

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

# 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

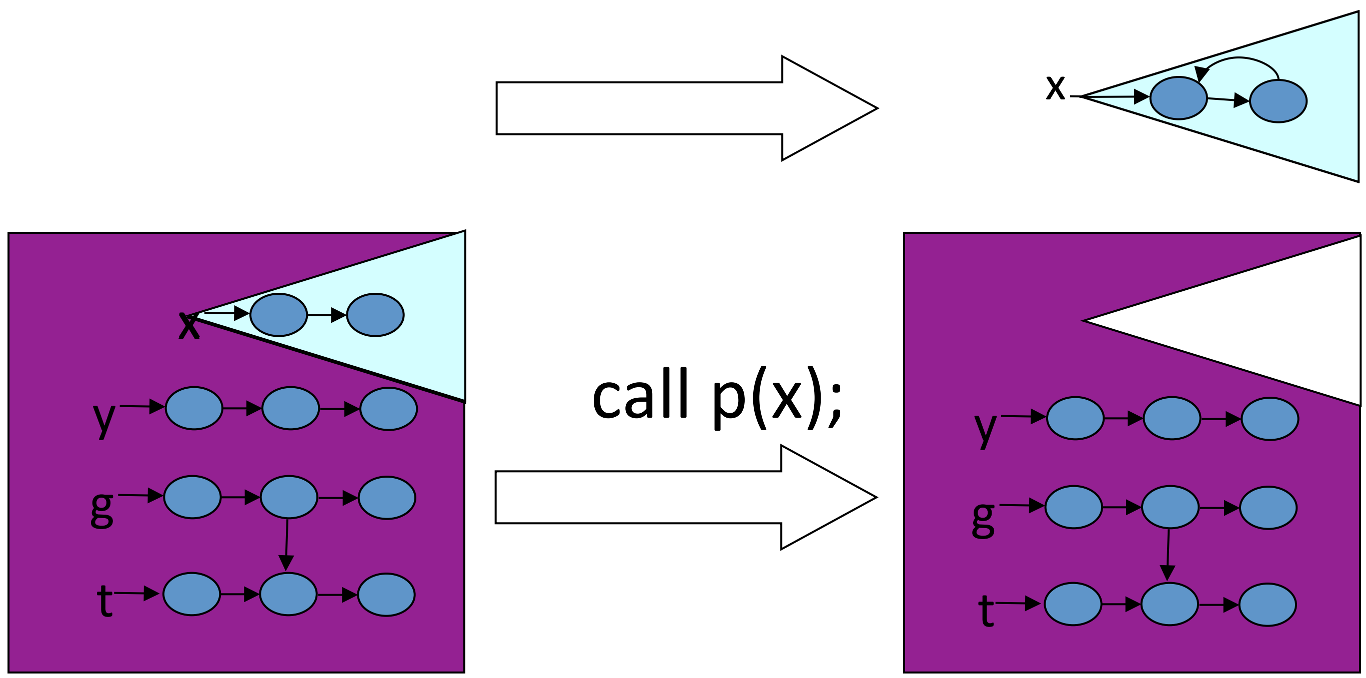
Reinhard Wilhelm Universität des Saarlandes

# 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

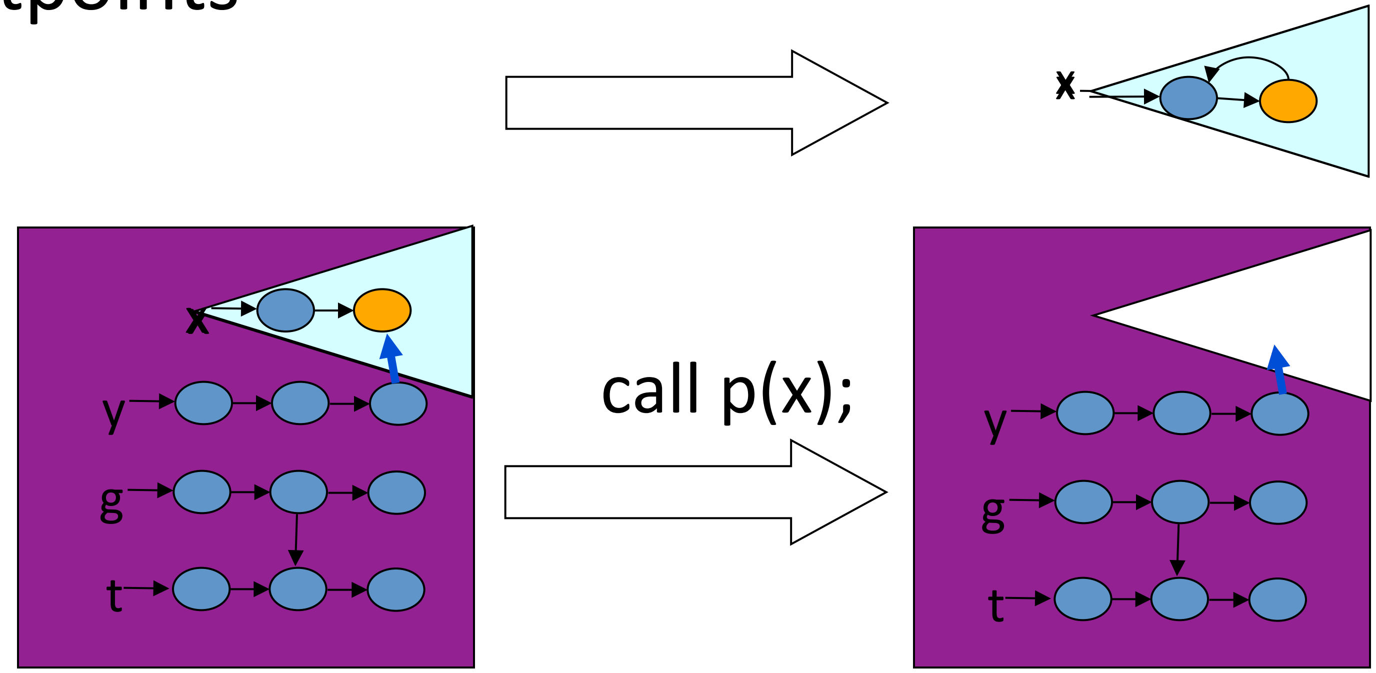
# Main idea

- Local heaps



# Main idea

- Local heaps
- Cutpoints



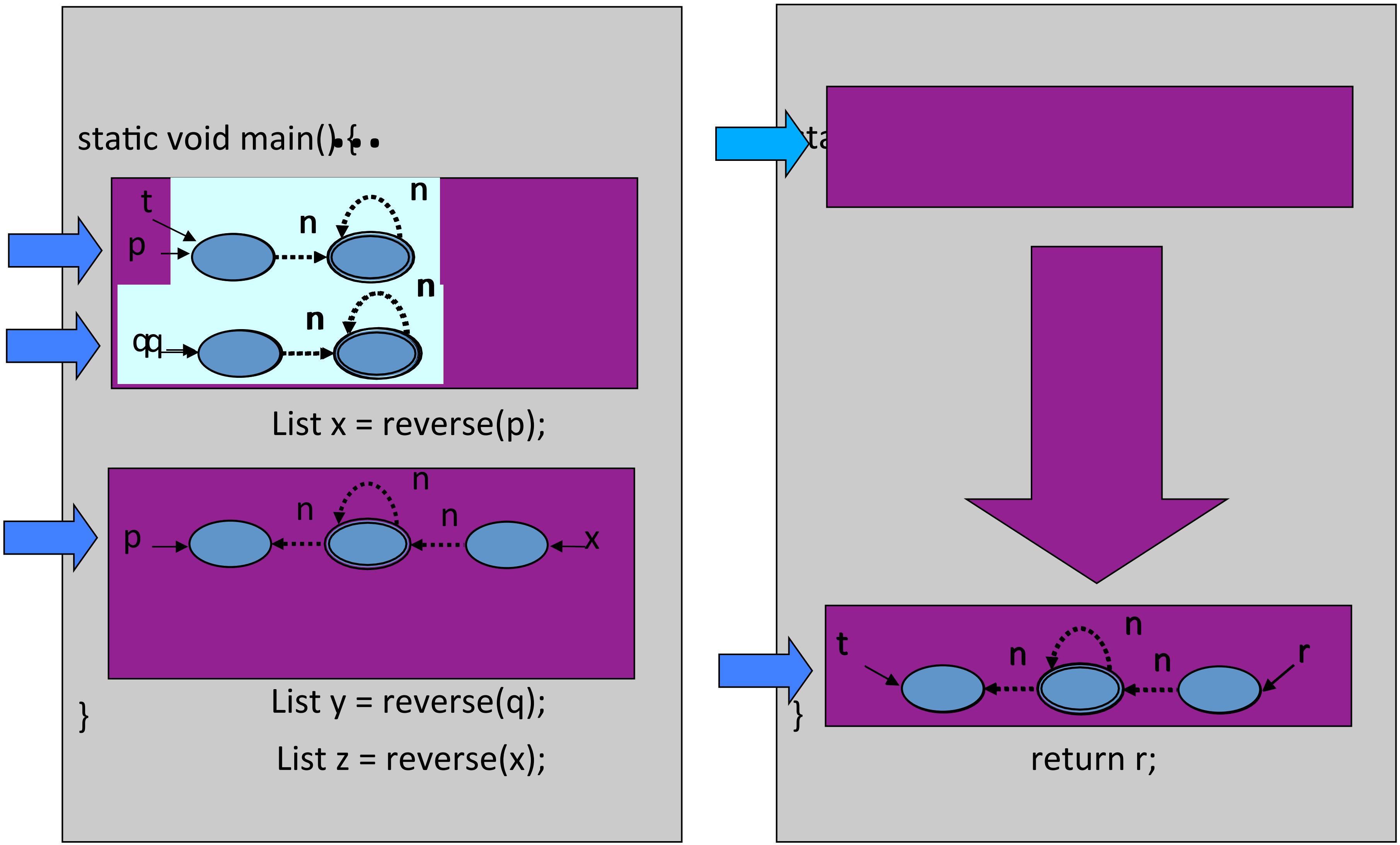
# Main Results

- 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]
  - …

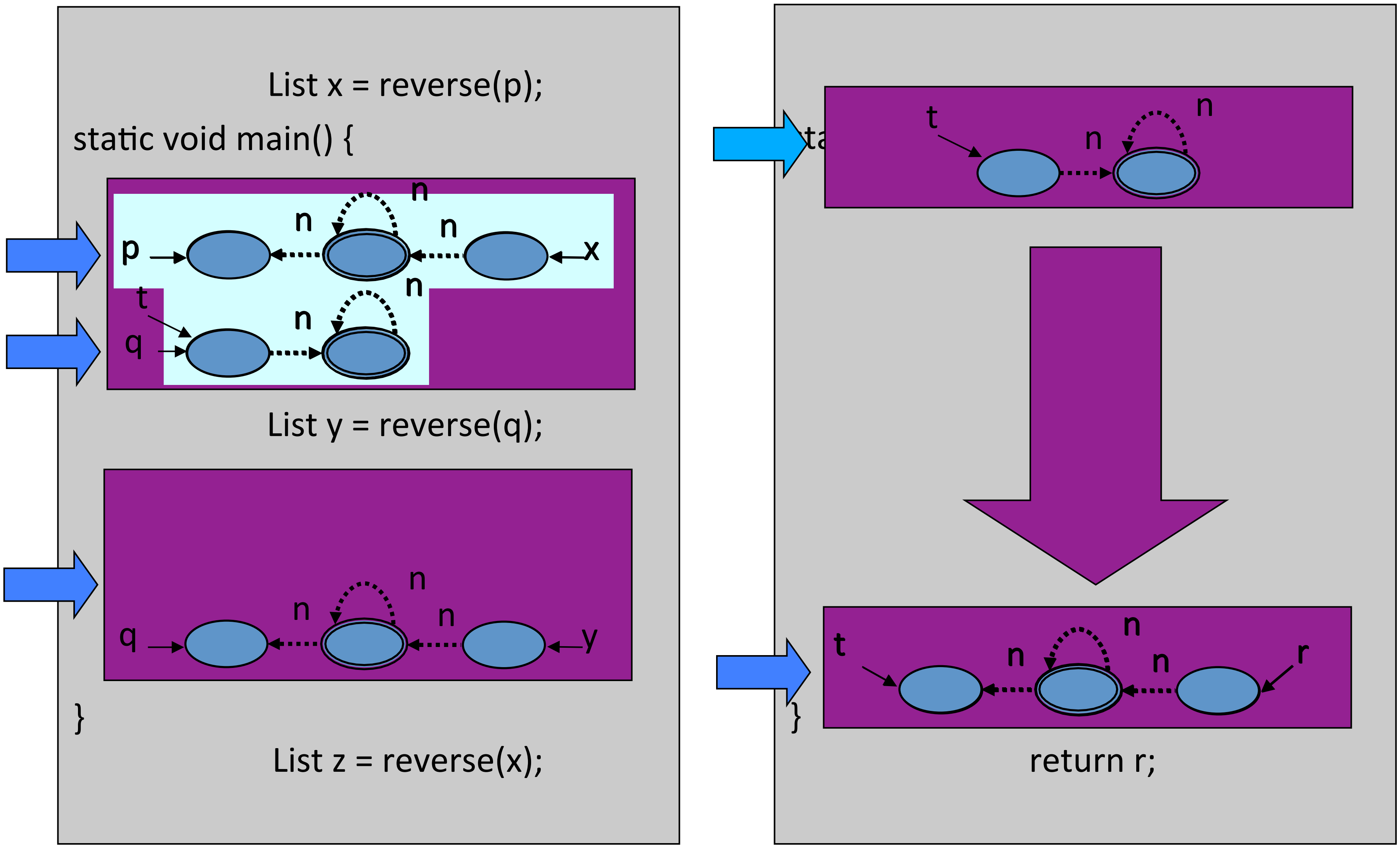
# Outline

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

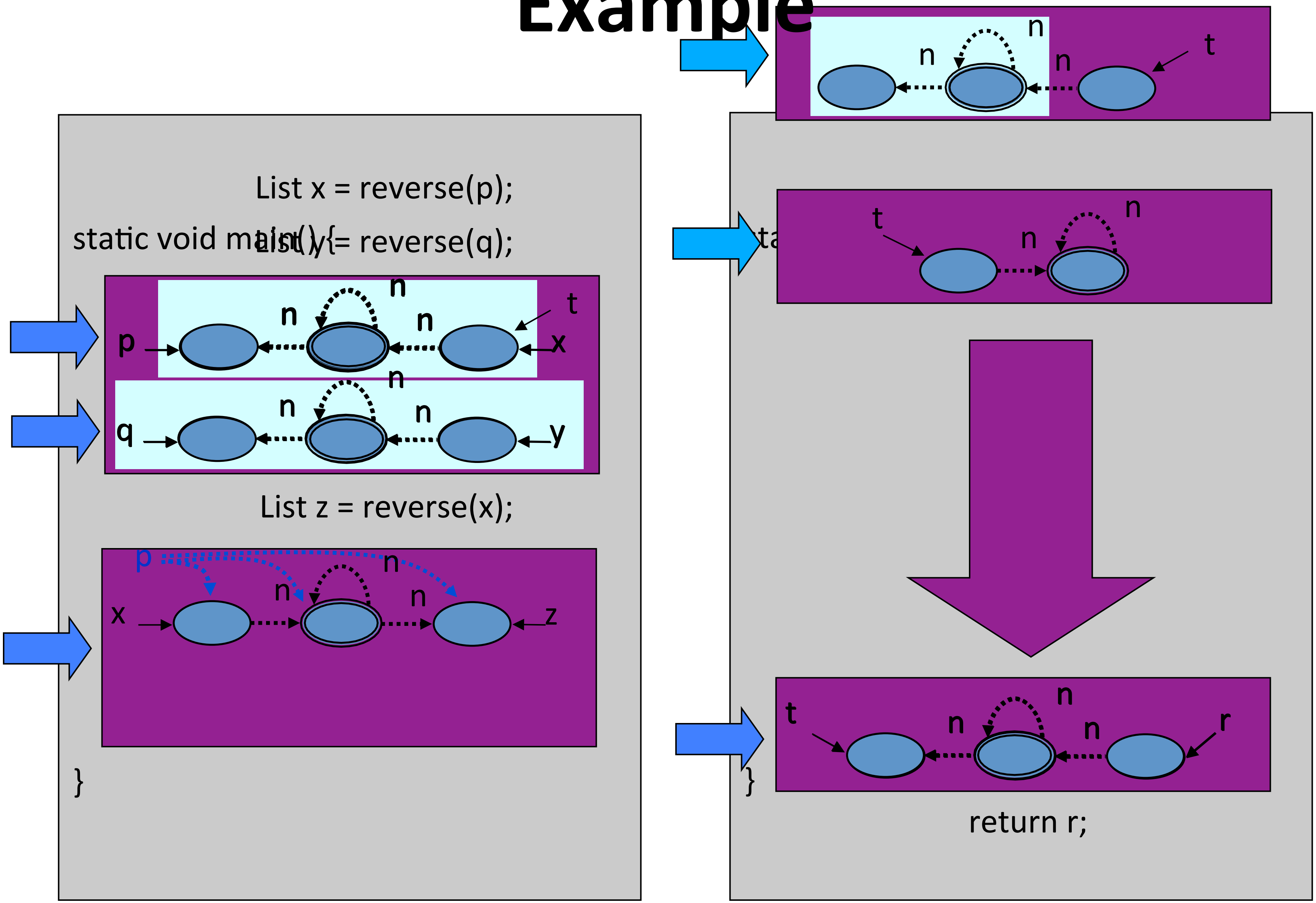
# Example


static void main() {
t
p
n
q
List x = reverse(p);
p
n
x
List y = reverse(q);
List z = reverse(x);
}
t
n
r
}
return r;

# Example


List x = reverse(p);
static void main() {
p
n
n
n
x
t
q
n
n
List y = reverse(q);
q
n
n
n
y
}
List z = reverse(x);
t
n
n
ta
t
n
n
n
r
}
return r;

# Example


List x = reverse(p);
static void main() {
List y = reverse(q);
p
q
n
t
x
y
List z = reverse(x);
z
}
return r;
r

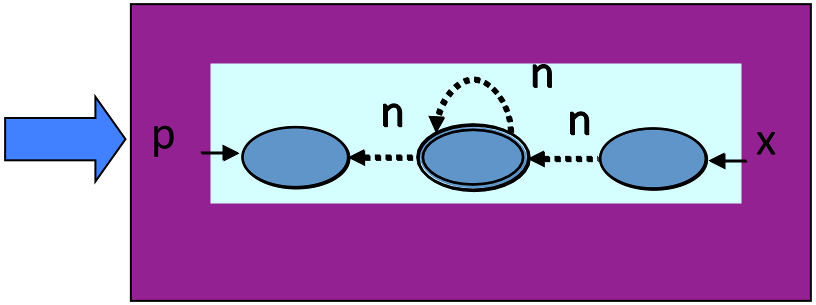
# Cutpoints

- Separating objects
  - Not pointed-to by a parameter

# Cutpoints

- Separating objects
  - Not pointed-to by a parameter

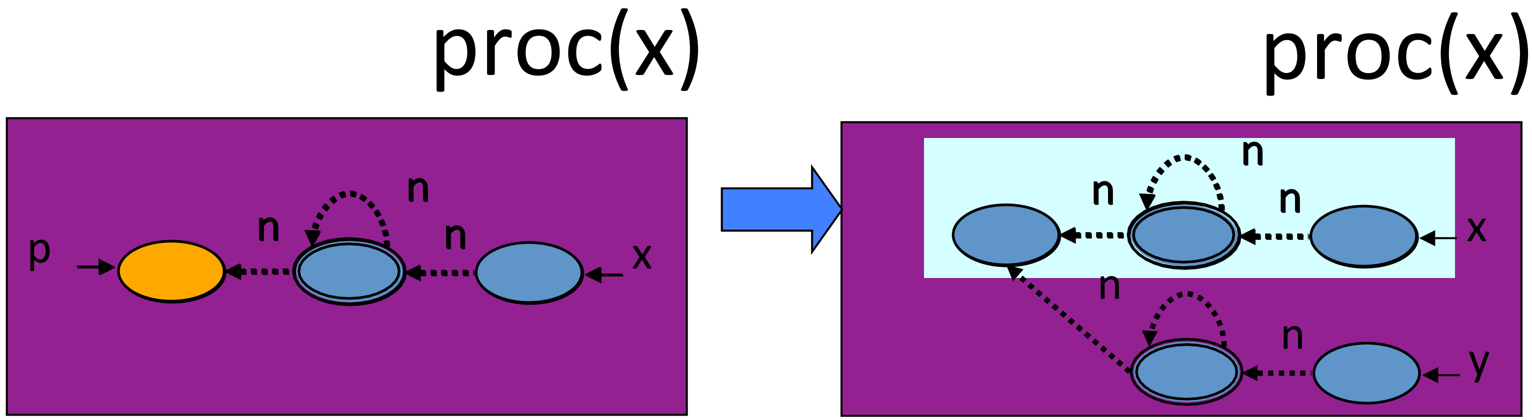
proc(x)




Stack sharing

# Cutpoints

- Separating objects
  - Not pointed-to by a parameter

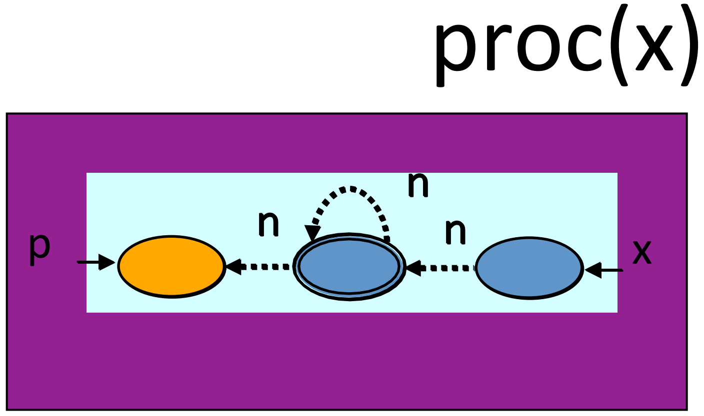



Stack sharing

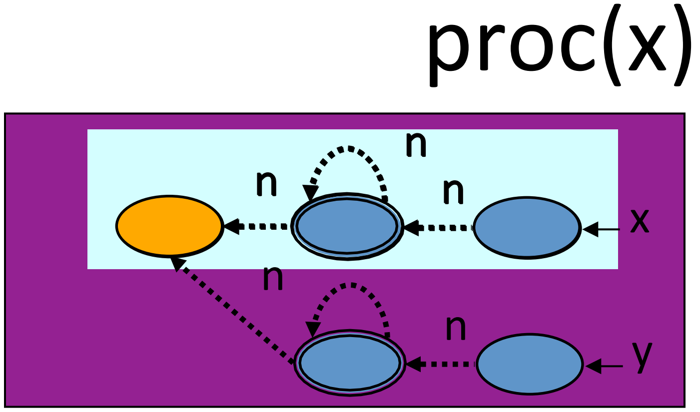
Heap sharing

# Cutpoints

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

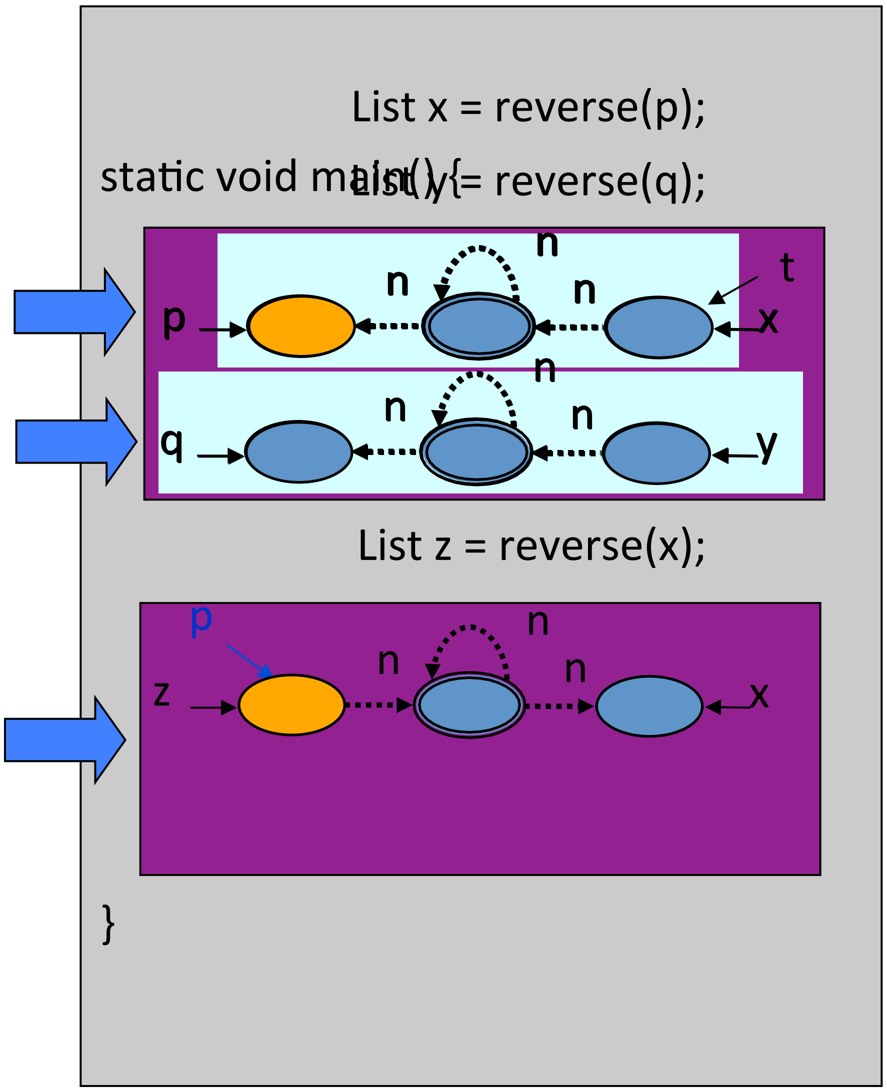


Stack sharing

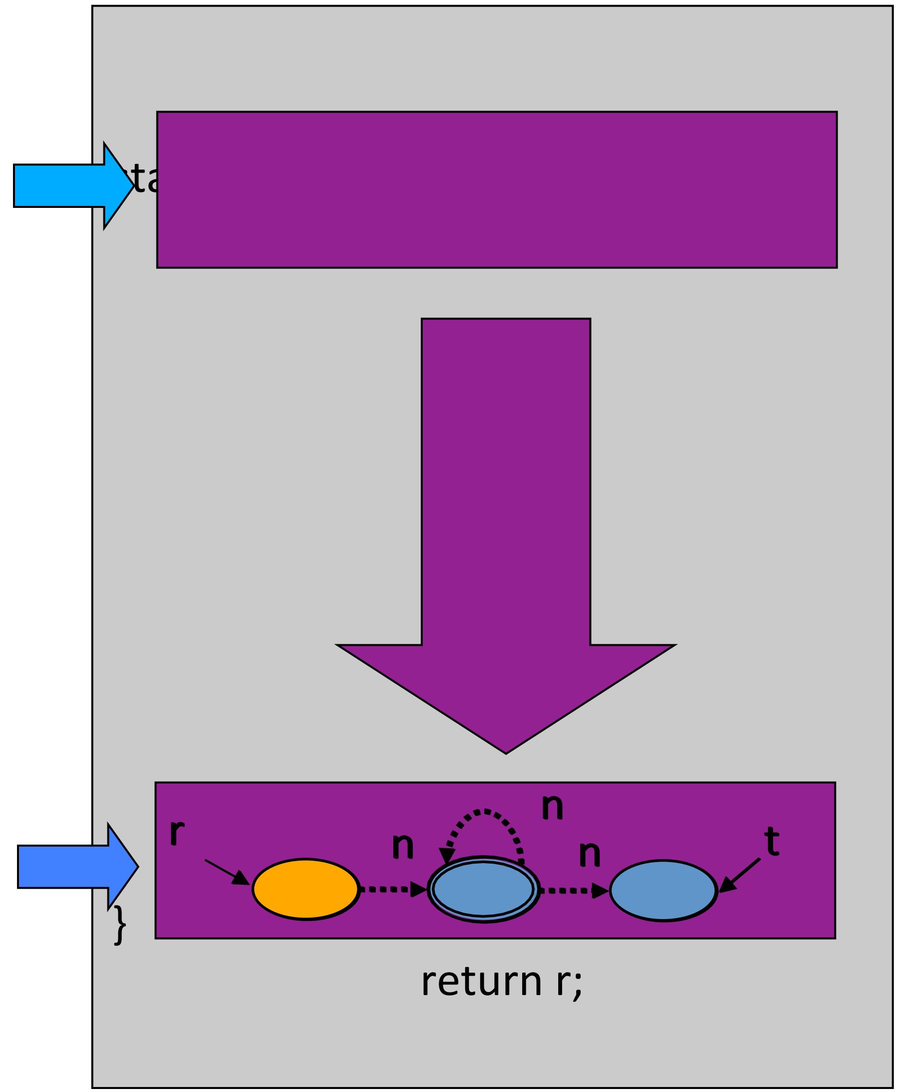


Heap sharing

# Example



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

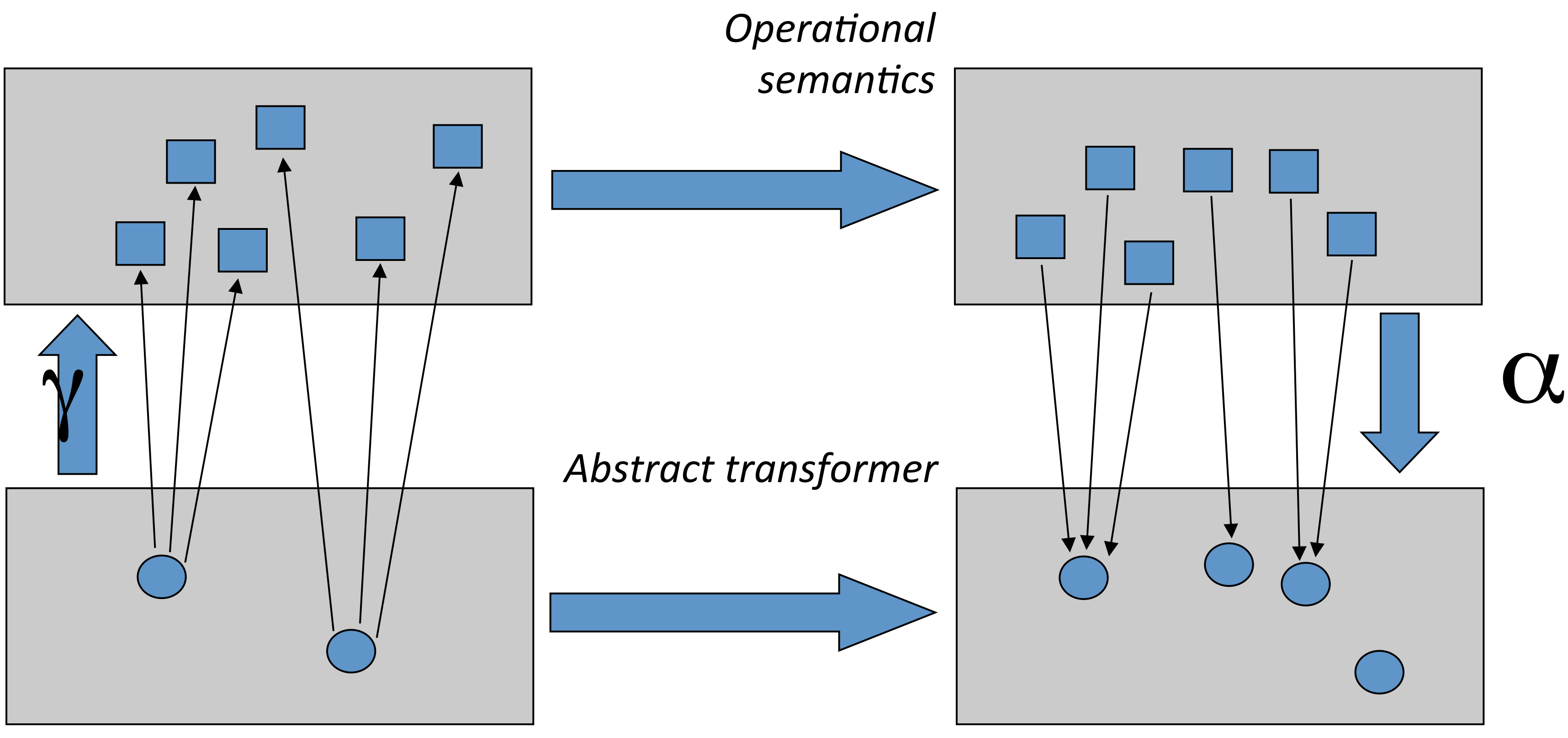



r
n
n
n
t
}
return r;

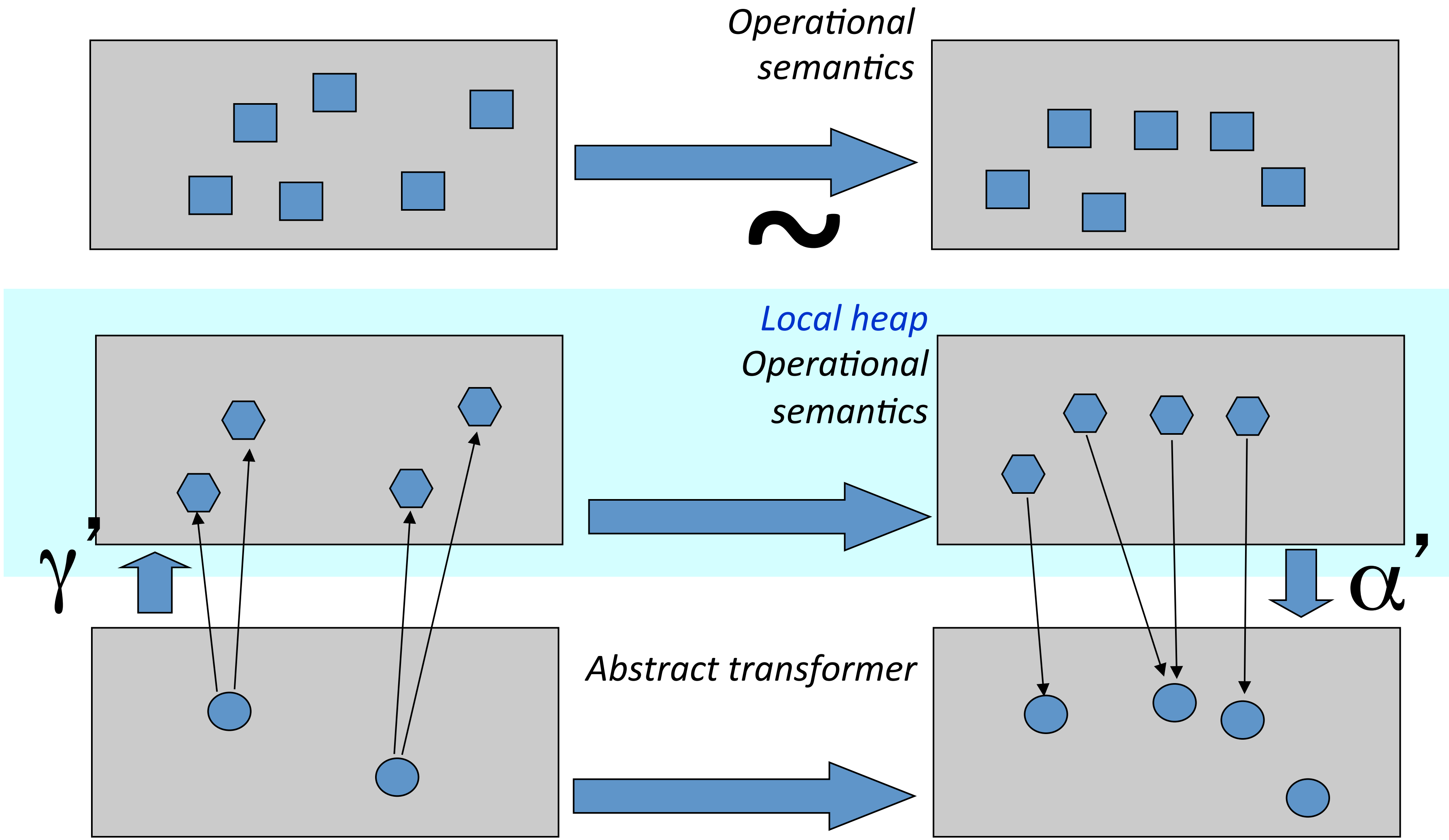
# Outline

✓ 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

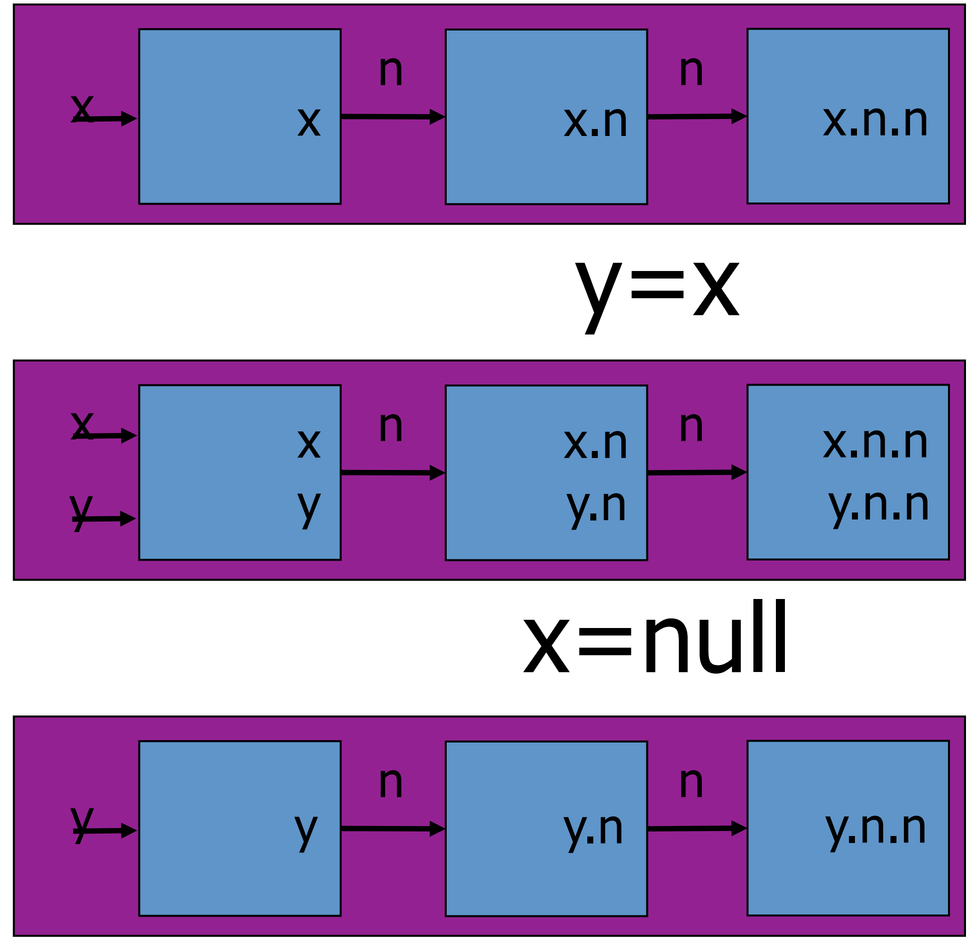
- Single threaded
- Procedures
  - ✓ Value parameters
  - ✓ Recursion
- Heap
  - ✓ Recursive data structures
  - ✓ Destructive update
  - ✘ No explicit addressing (&)
  - ✘ No pointer arithmetic

# Simplifying assumptions

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

# Storeless semantics

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

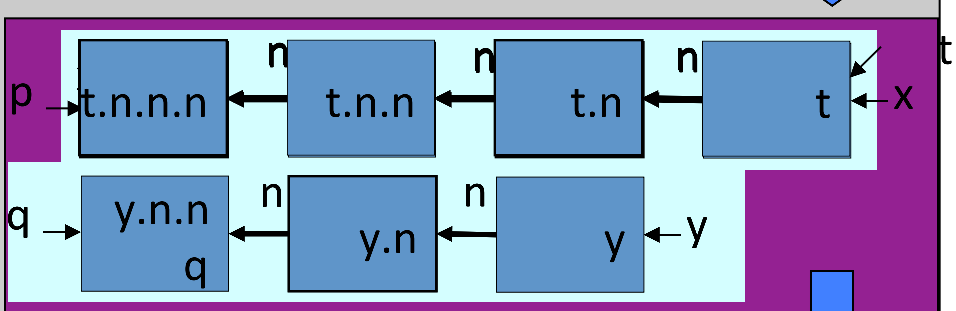


# Example

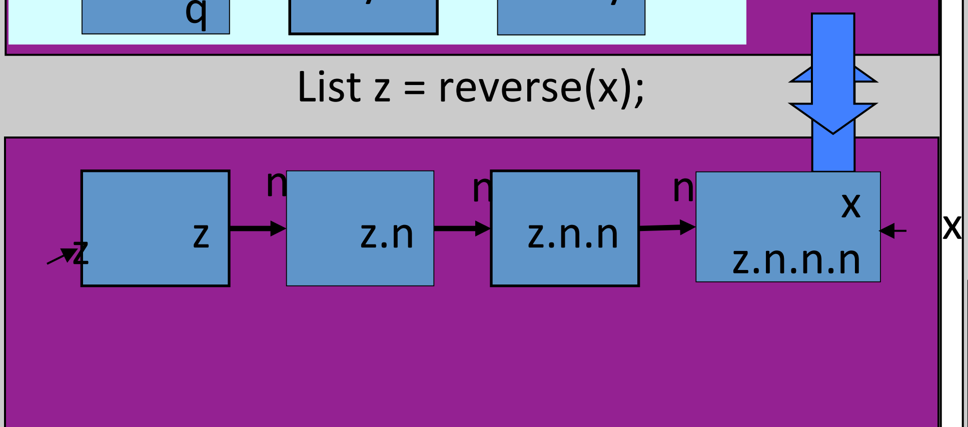
static void main() {

List x = reverse(p);

List y = reverse(q);

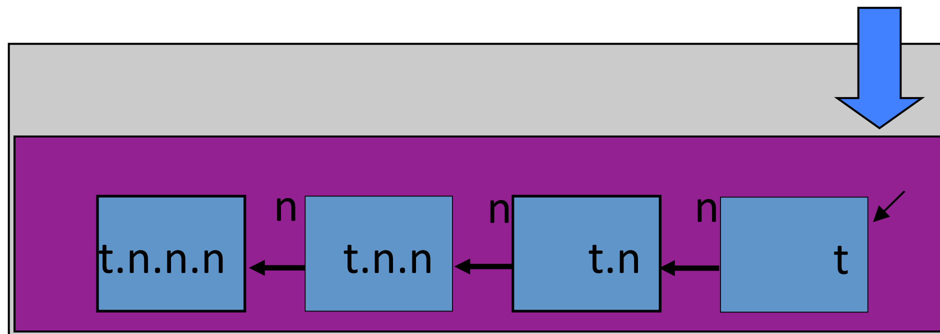


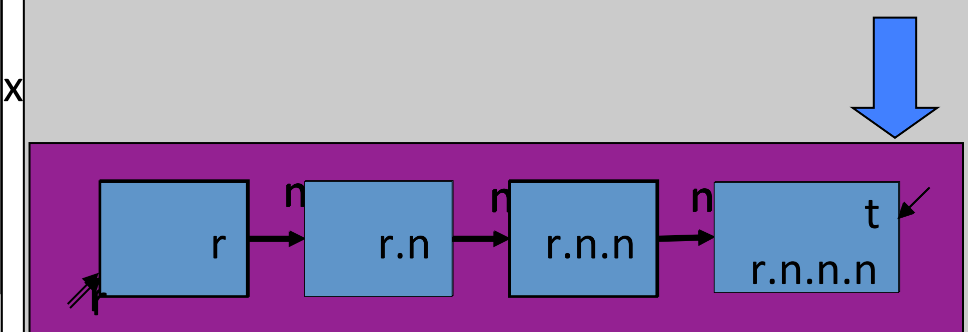

List z = reverse(x);




}

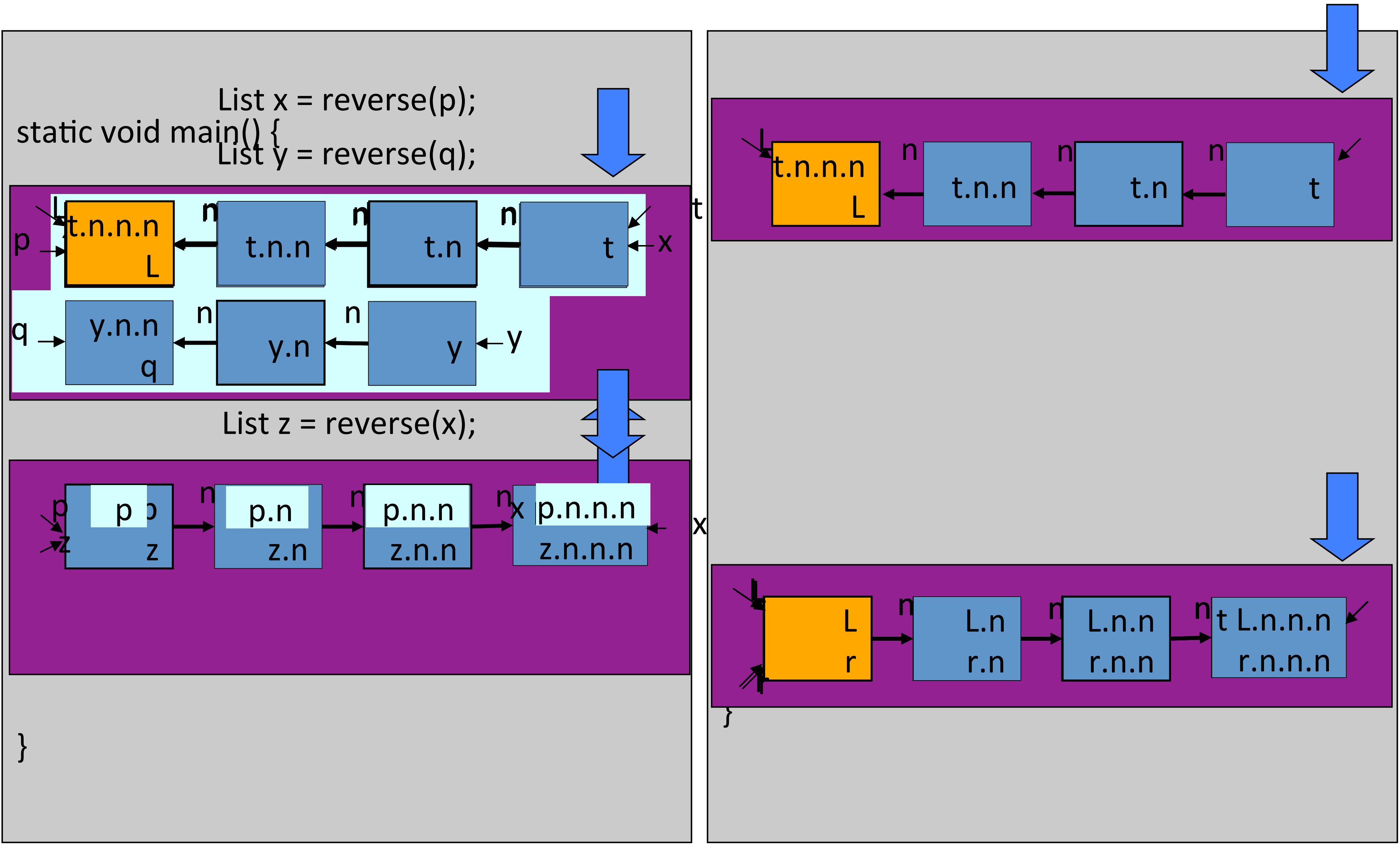
p?




}

# Example



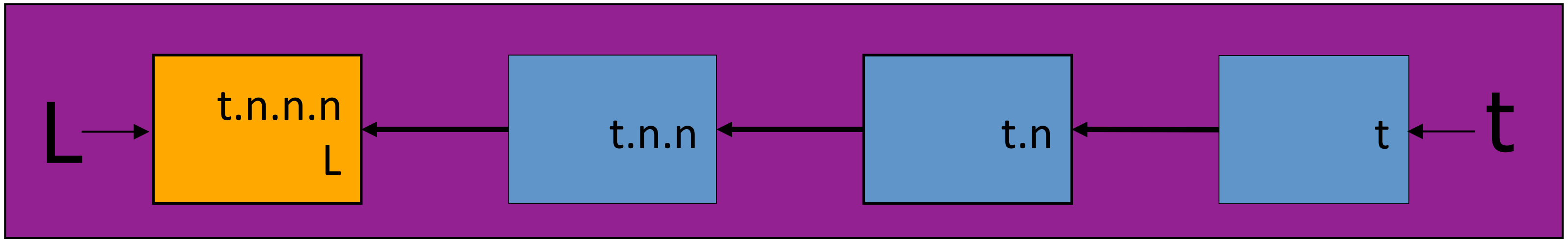
List x = reverse(p);
static void main() {
List y = reverse(q);
t.n.n.n
L
t.n.n
t.n
t
n
p
x
t
y.n.n
q
y.n
y
q
y
List z = reverse(x);
p
z
p.n
z.n
p.n.n
z.n.n
p.n.n.n
z.n.n.n
x
}
t.n.n.n
L
t.n.n
t.n
t
L
r
L.n
r.n
L.n.n
r.n.n
t L.n.n.n
r.n.n.n
}

# Cutpoint labels

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

# Cutpoint labels

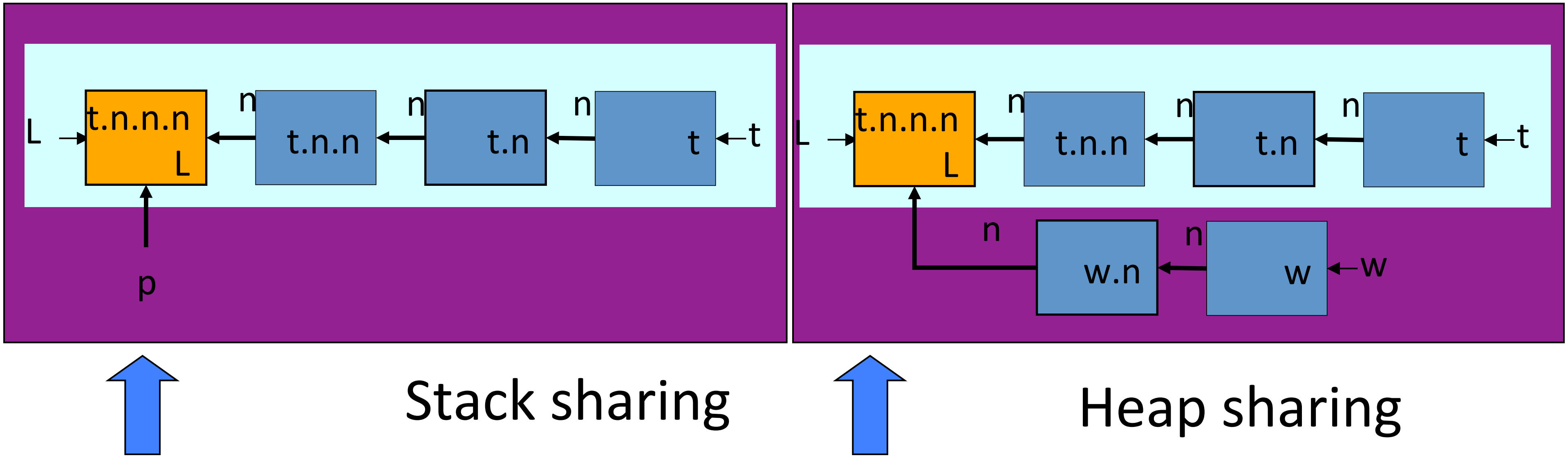
- **Cutpoint label**: the set of access paths that point to a cutpoint
  - when the invoked procedure starts




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

# Sharing patterns

- Cutpoint labels encode sharing patterns




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

# Observational equivalence

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

$\sigma_L$ and $\sigma_G$ 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)
- $\sigma_L$ and $\sigma_G$ observationally equivalent

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

$\sigma'_L$ and $\sigma'_G$ 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

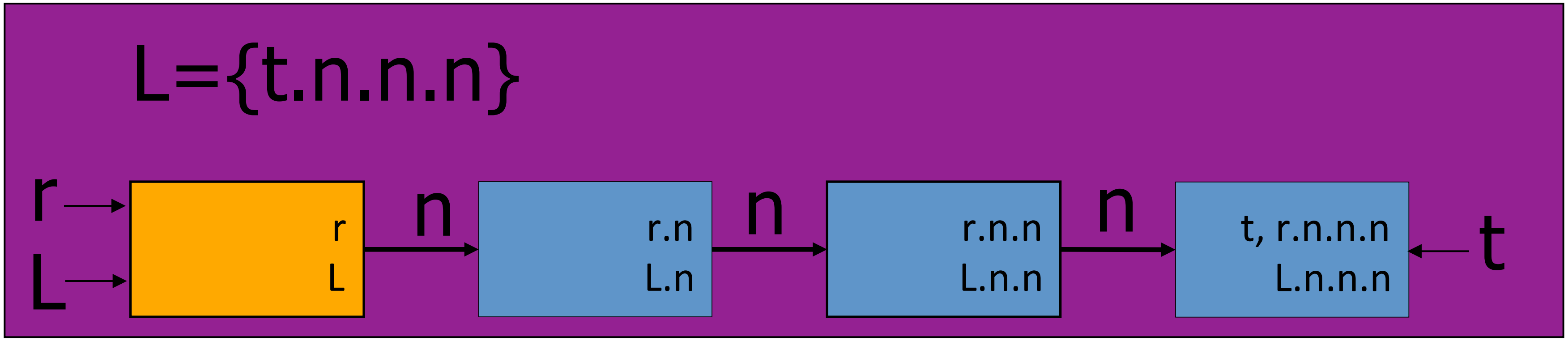
# Related work

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

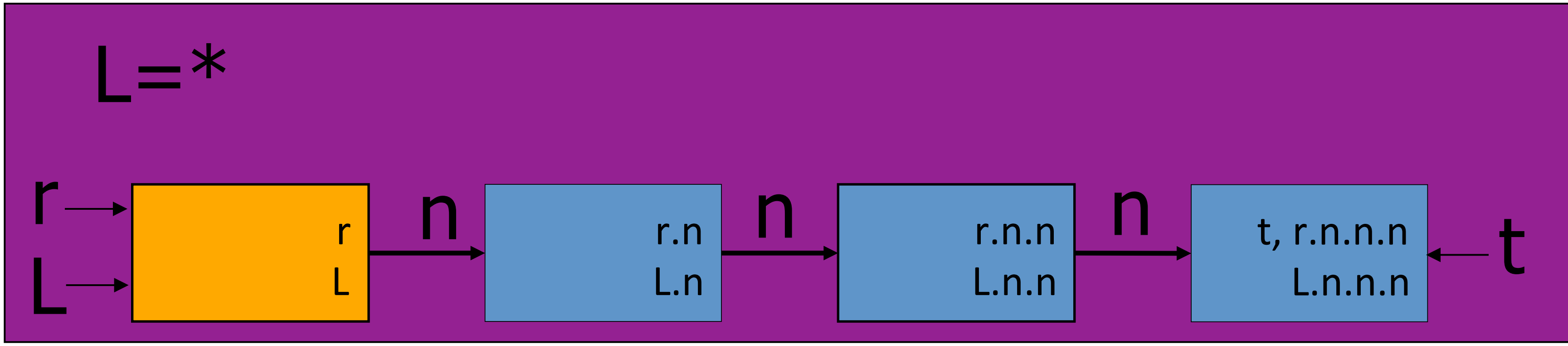
# Shape abstraction

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

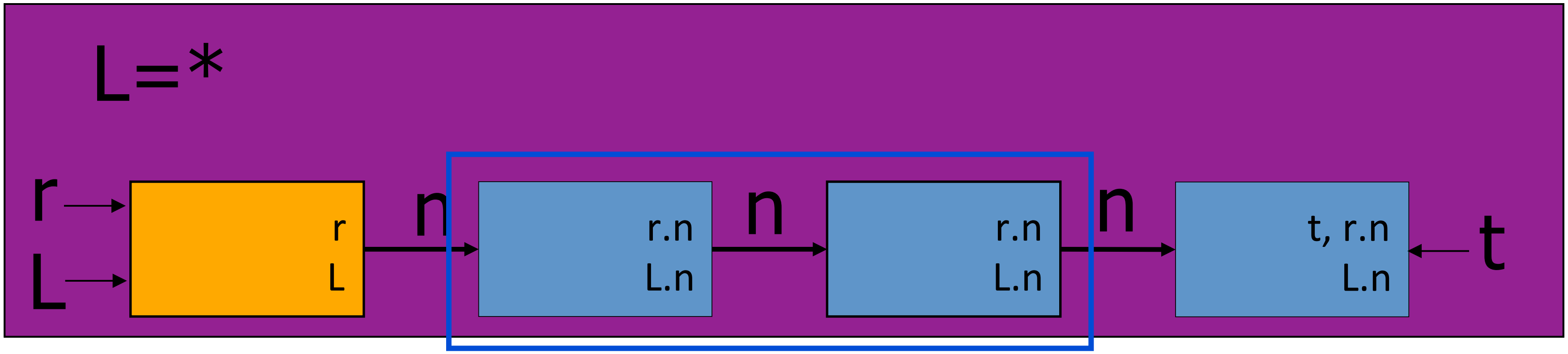
# A Shape abstraction



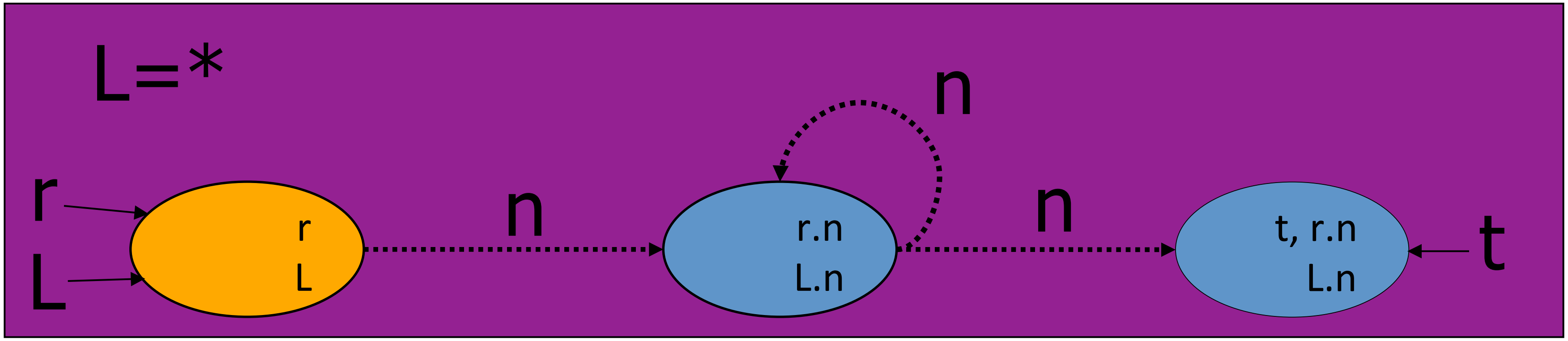
# A Shape abstraction



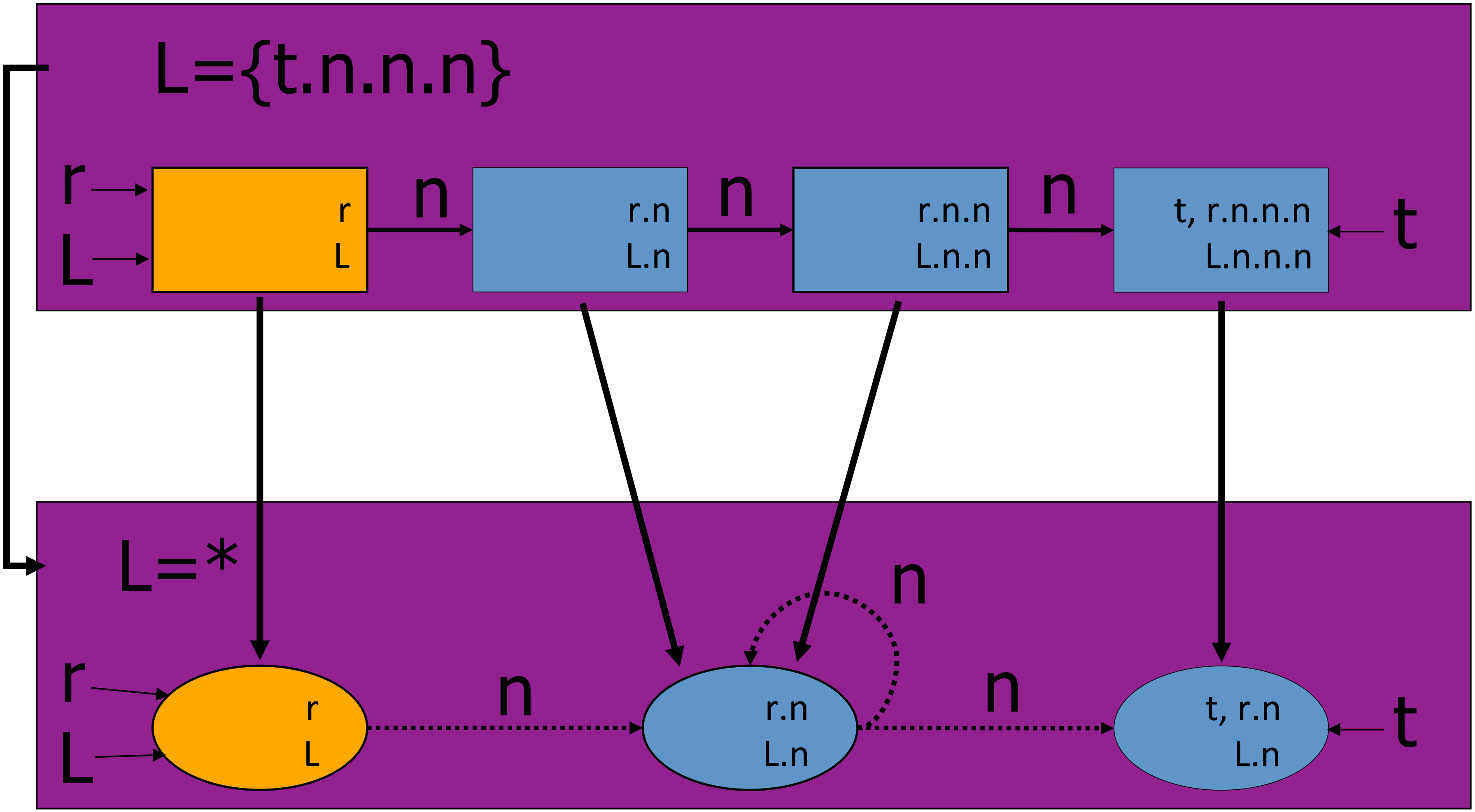
# A Shape abstraction



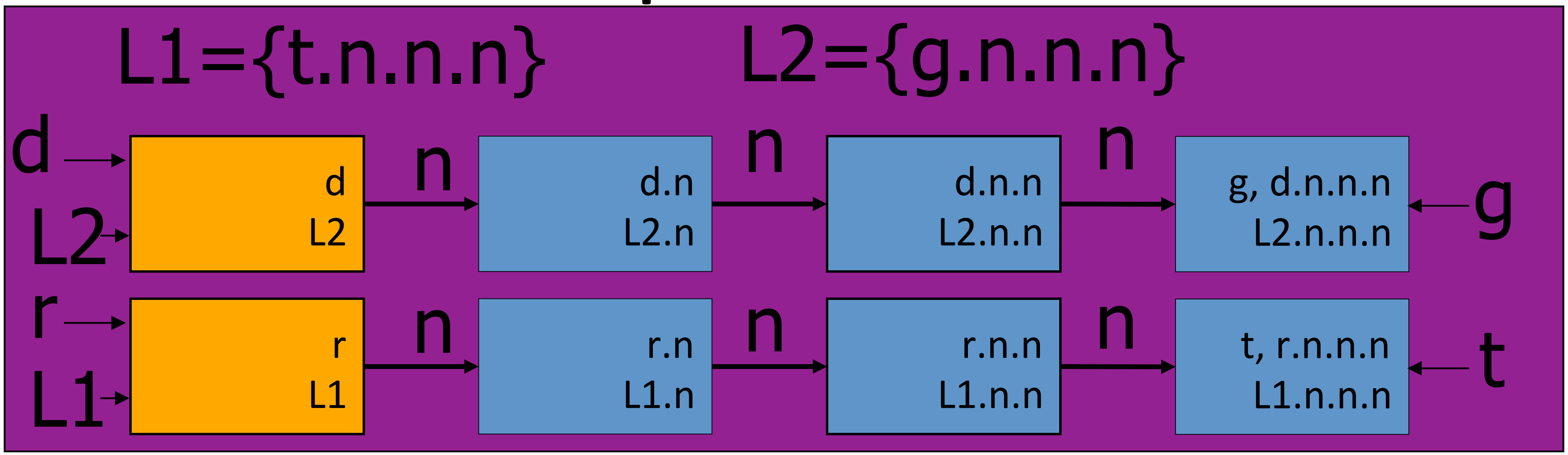
# A Shape abstraction

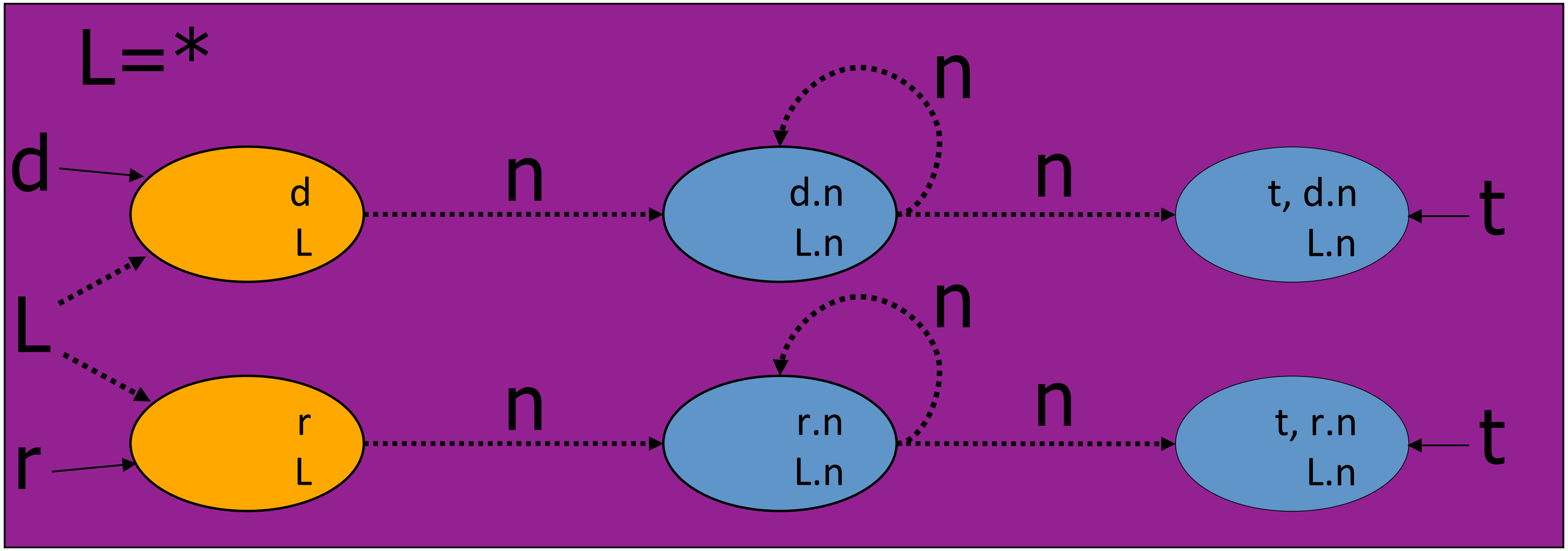


# A Shape abstraction



# A Shape abstraction

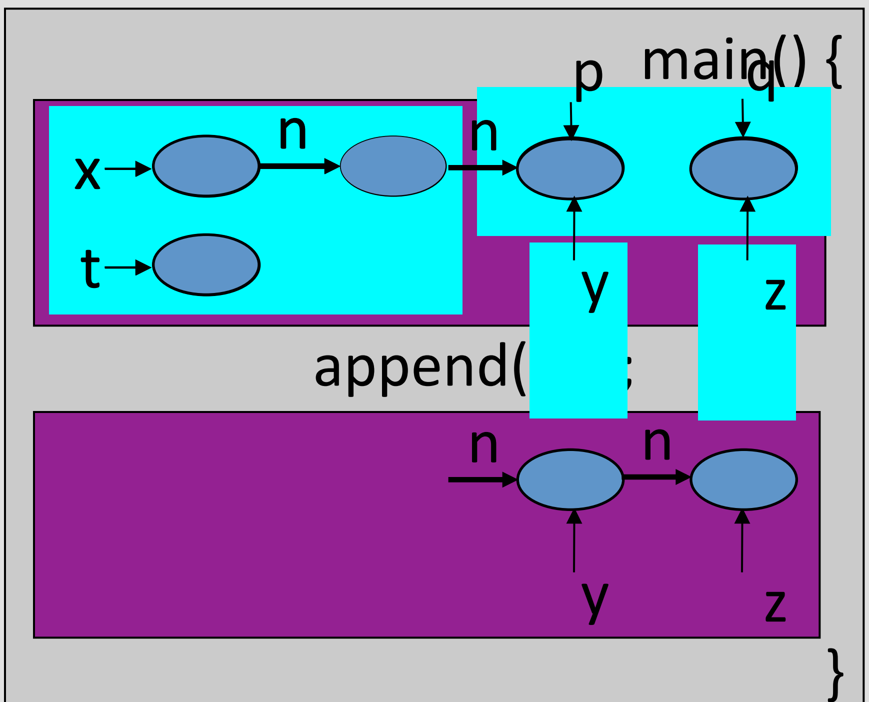


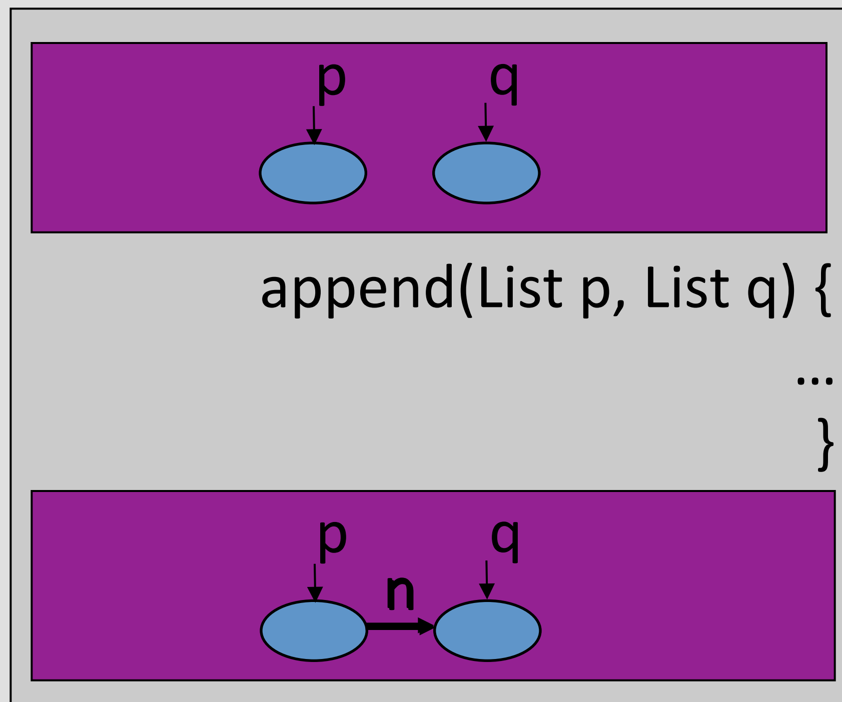



# Cutpoint-Freedom

# How to tabulate procedures?

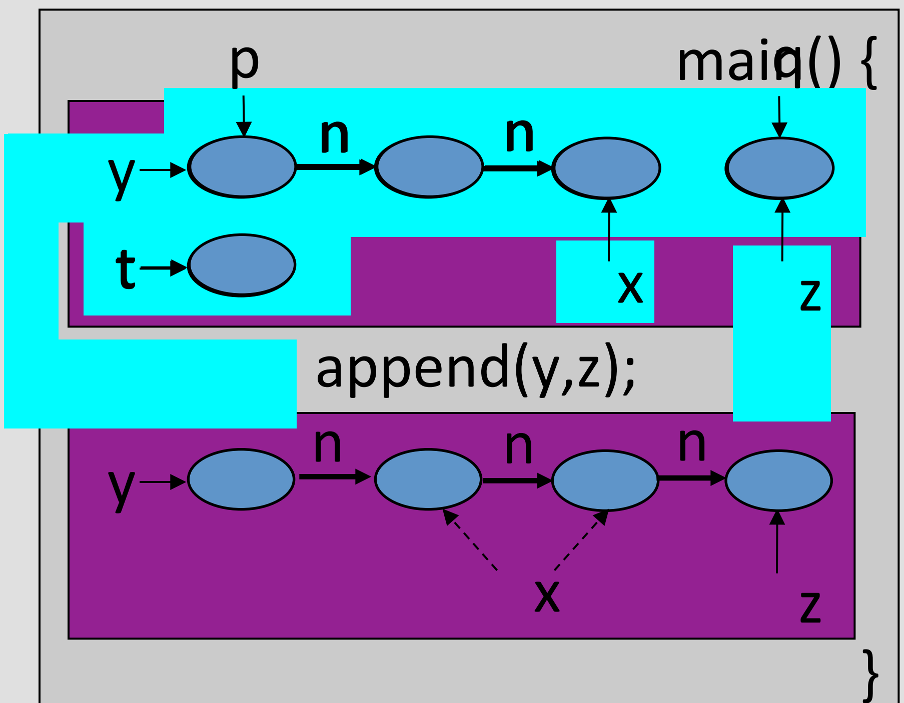
- Procedure ≡ input/output relation
  - Not reachable ➔ Not effected
  - proc: local (≡reachable) heap ➔ local heap

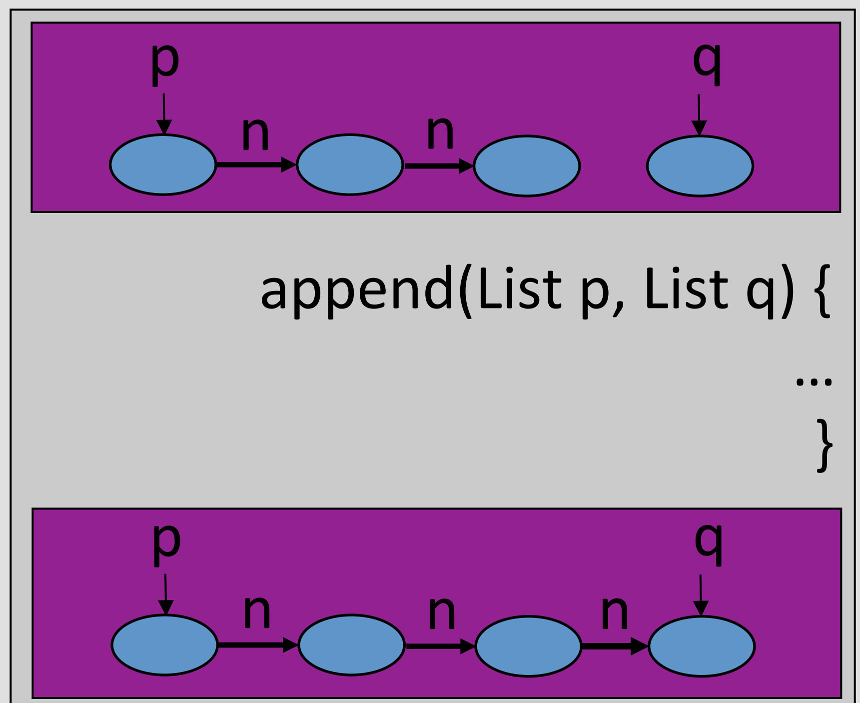


# How to handle sharing?

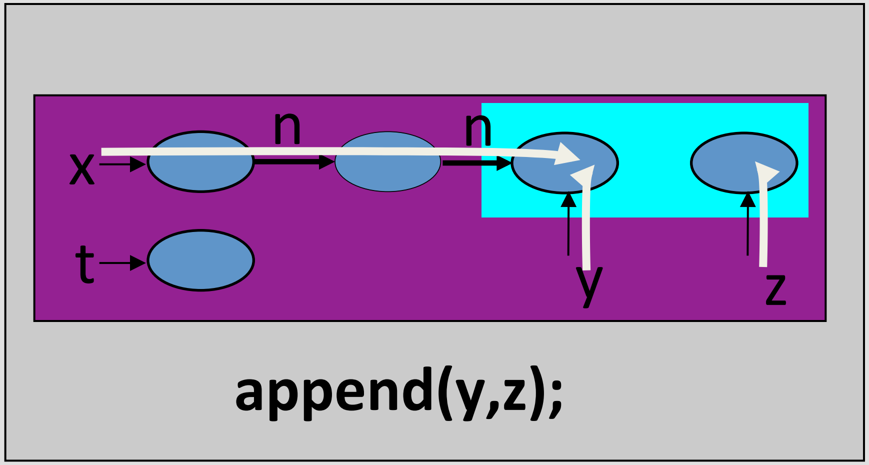
- External sharing may break the functional view



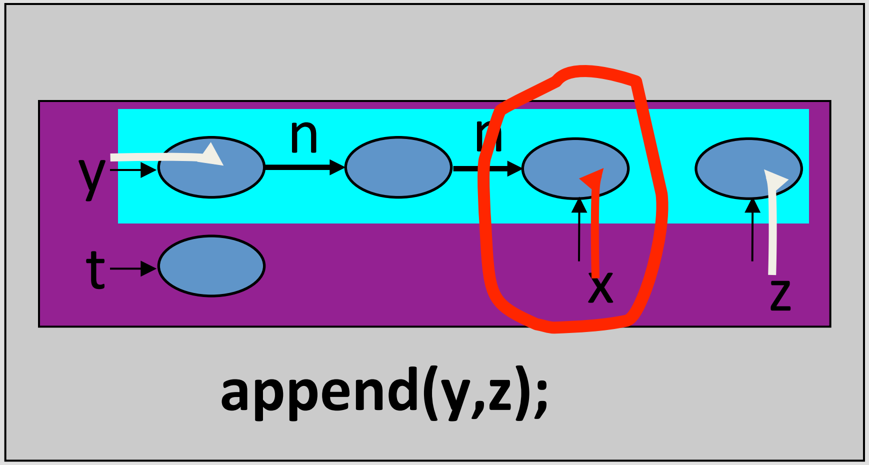



# What's the difference?

**1st Example**



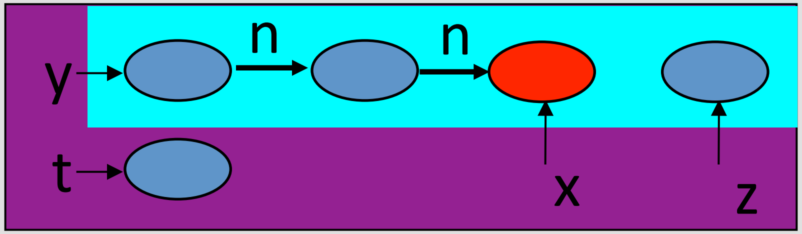

**2nd Example**



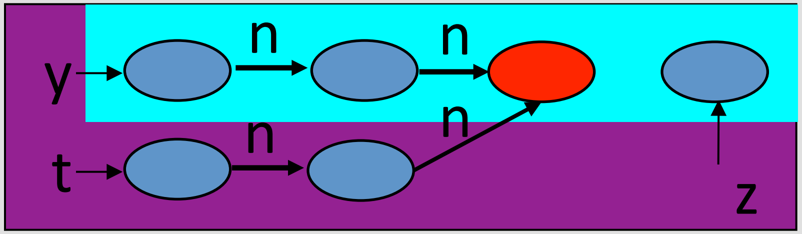
# Cutpoints

- 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)



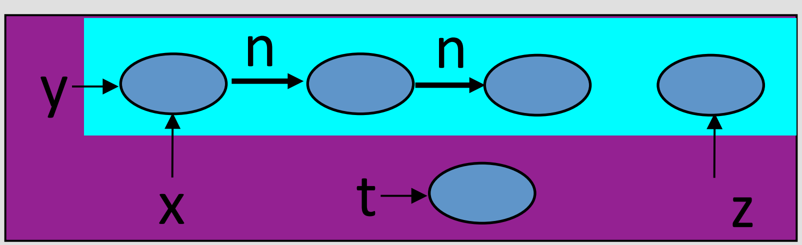

append(y,z)



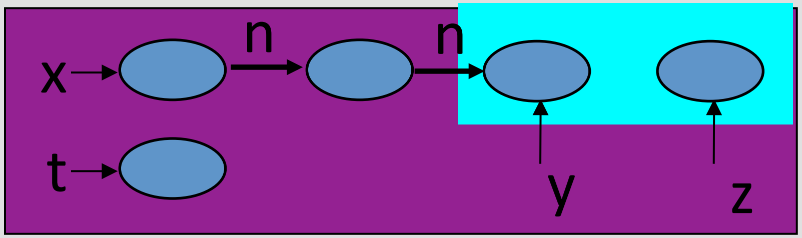
# Cutpoint freedom

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

append(y,z)




append(y,z)

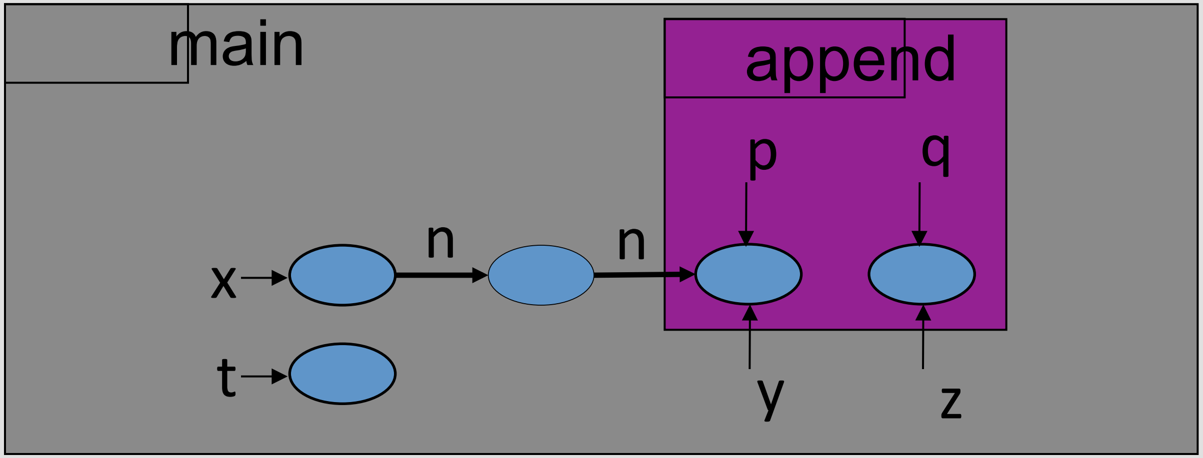


# Interprocedural shape analysis for cutpoint-free programs

# 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

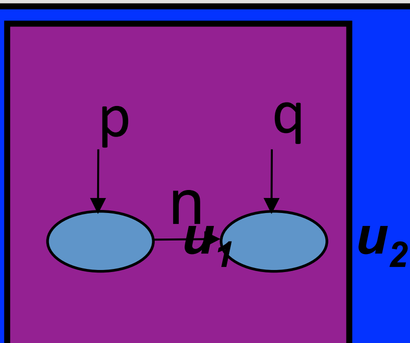
- Represented by first-order logical structures

| 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 |
|---|---|
| **x**(*v*) | Variable **x** points to *v* |
| **n**($v_1$,$v_2$) | Field **n** of object $v_1$ points to $v_2$ |




| | **p** |
|---|---|
| $u_1$ | 1 |
| $u_2$ | 0 |

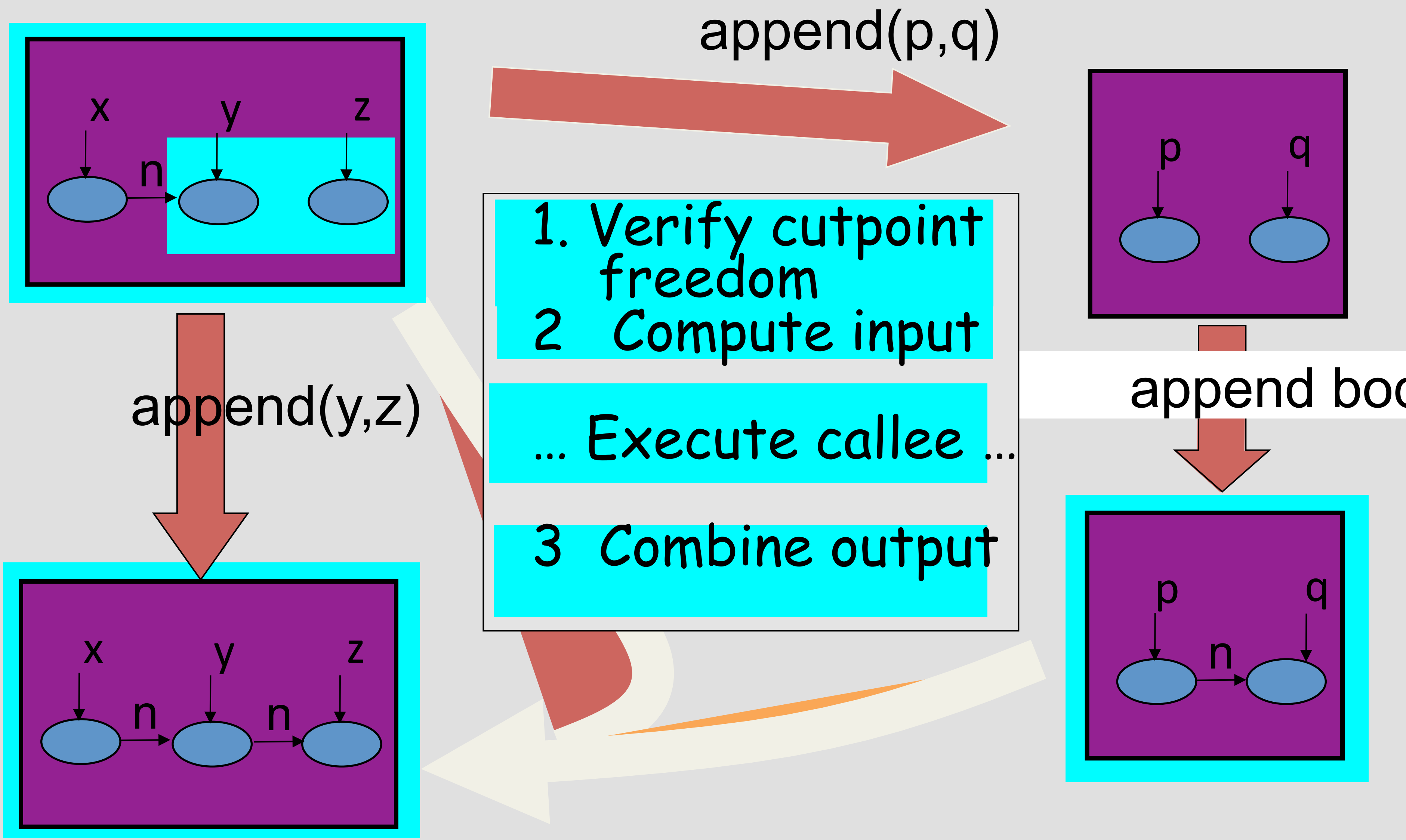
| | **q** |
|---|---|
| $u_1$ | 0 |
| $u_2$ | 1 |

| **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

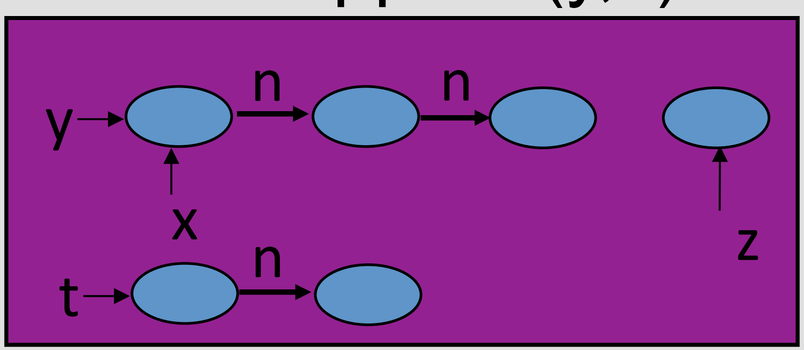
# Procedure calls



# Procedure call:
# 1. Verifying cutpoint-freedom

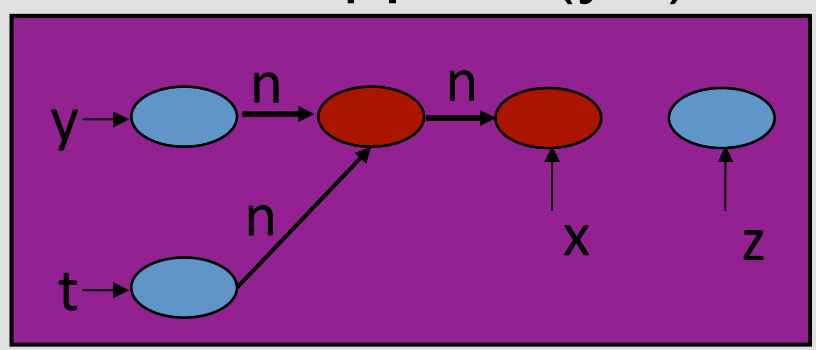
- An object is a **cutpoint** for an invocation
  - Reachable from actual parameters
  - Not pointed to by an actual parameter
  - Reachable without going through a parameter

append(y,z)




**Cutpoint free**

append(y,z)

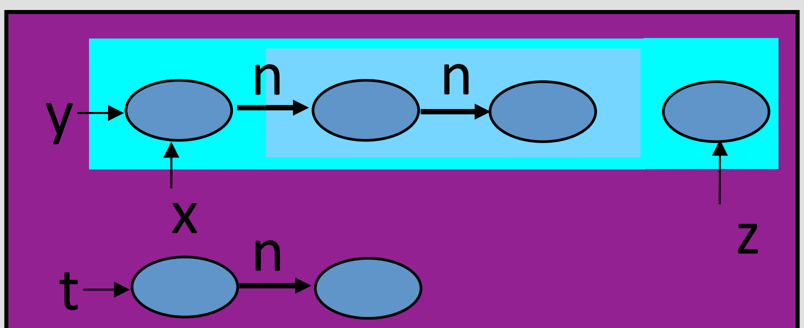



**Not Cutpoint free**

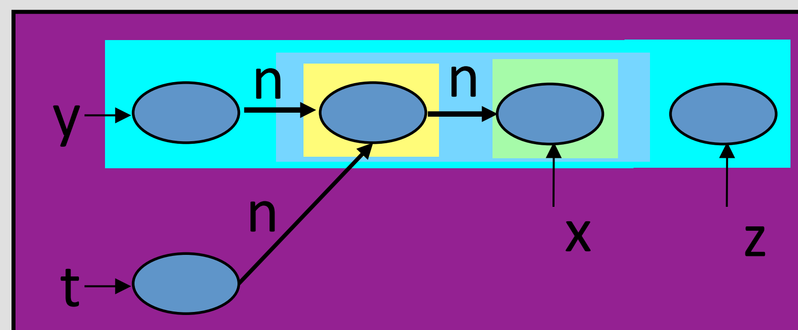
# Procedure call:
# 1. Verifying cutpoint-freedom

- 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)



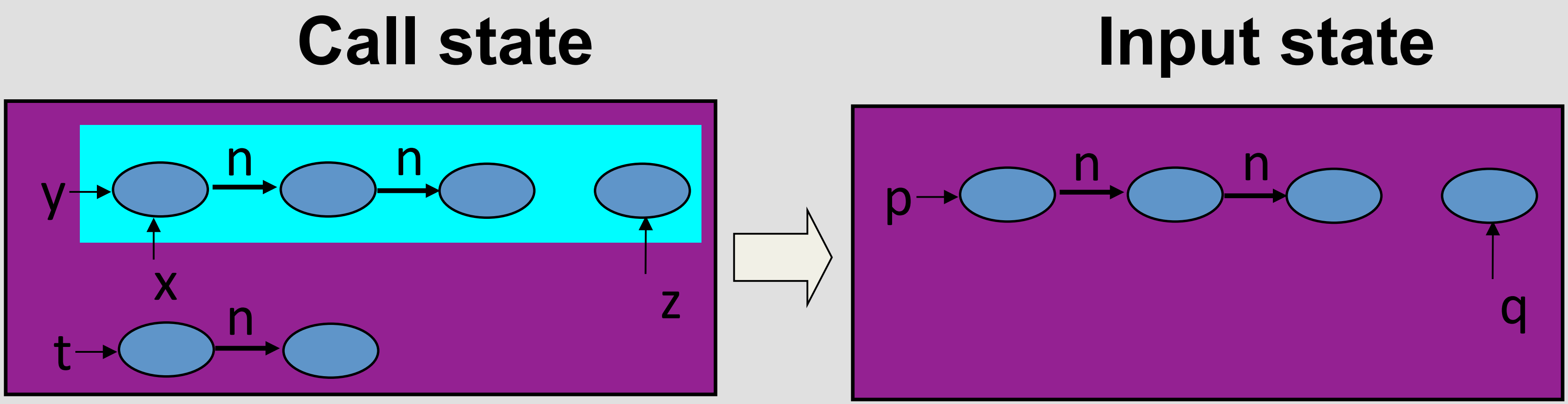

**Cutpoint free**



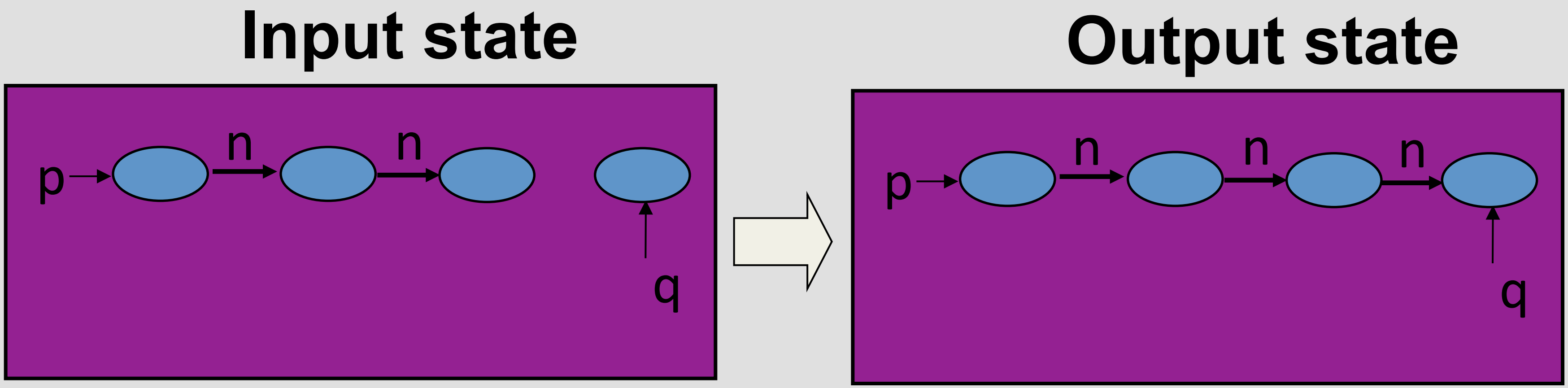

**Not Cutpoint free**

# Procedure call:
# 2. Computing the input local heap

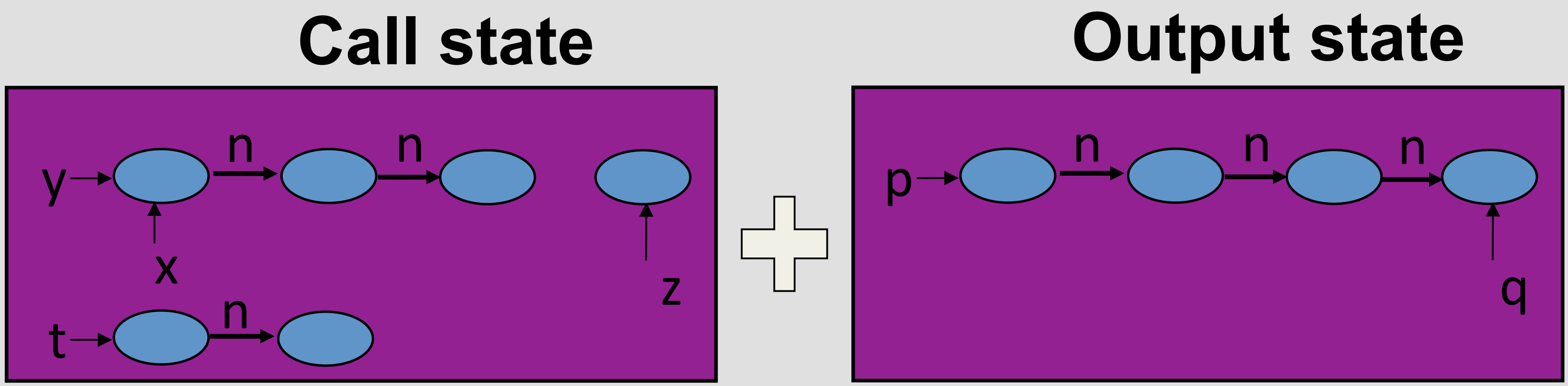
- Retain only reachable objects
- Bind formal parameters



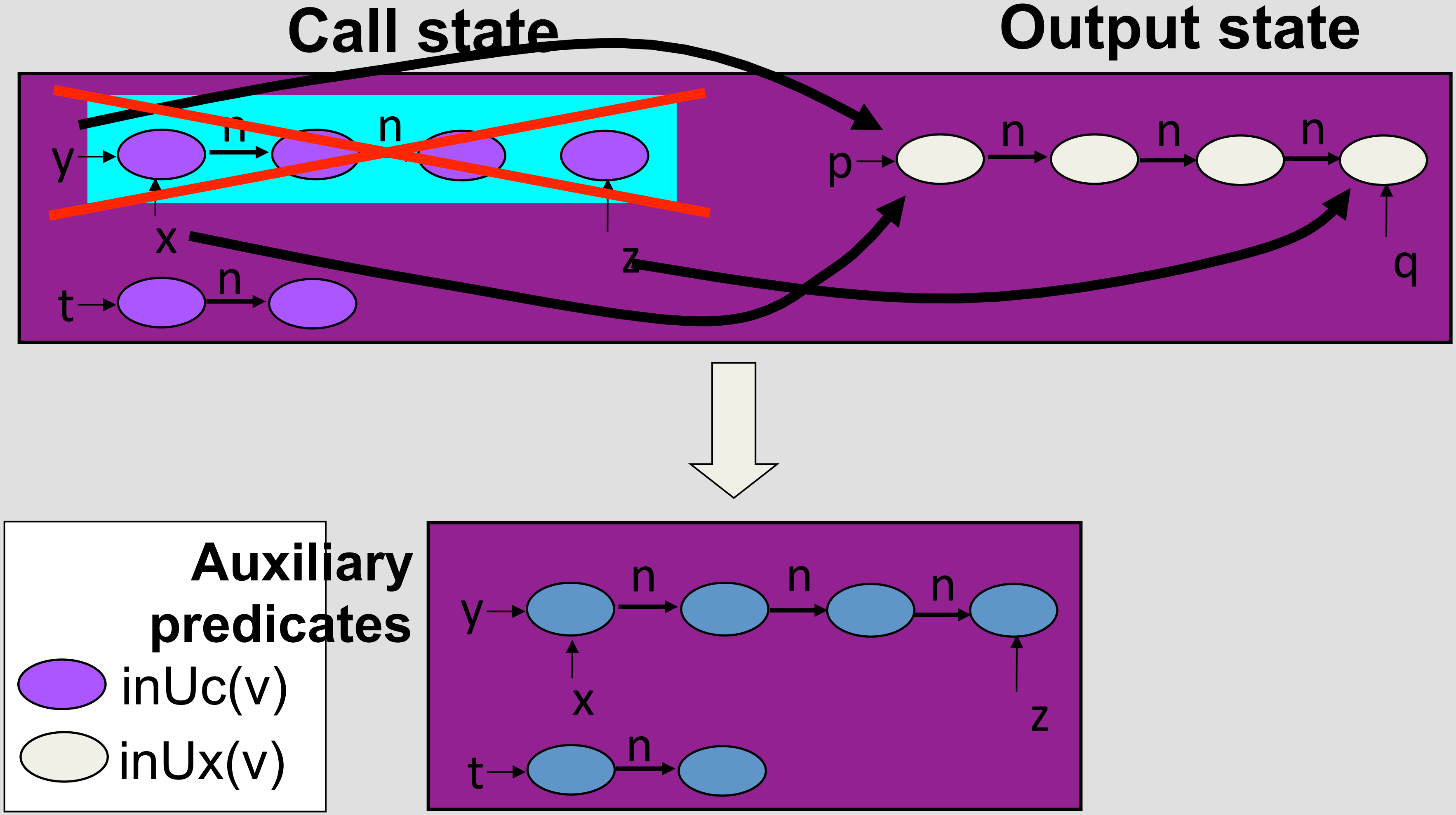
# Procedure body: append(p,q)



# Procedure call:
# 3. Combine output



# Procedure call:
# 3. Combine output



# Observational equivalence

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

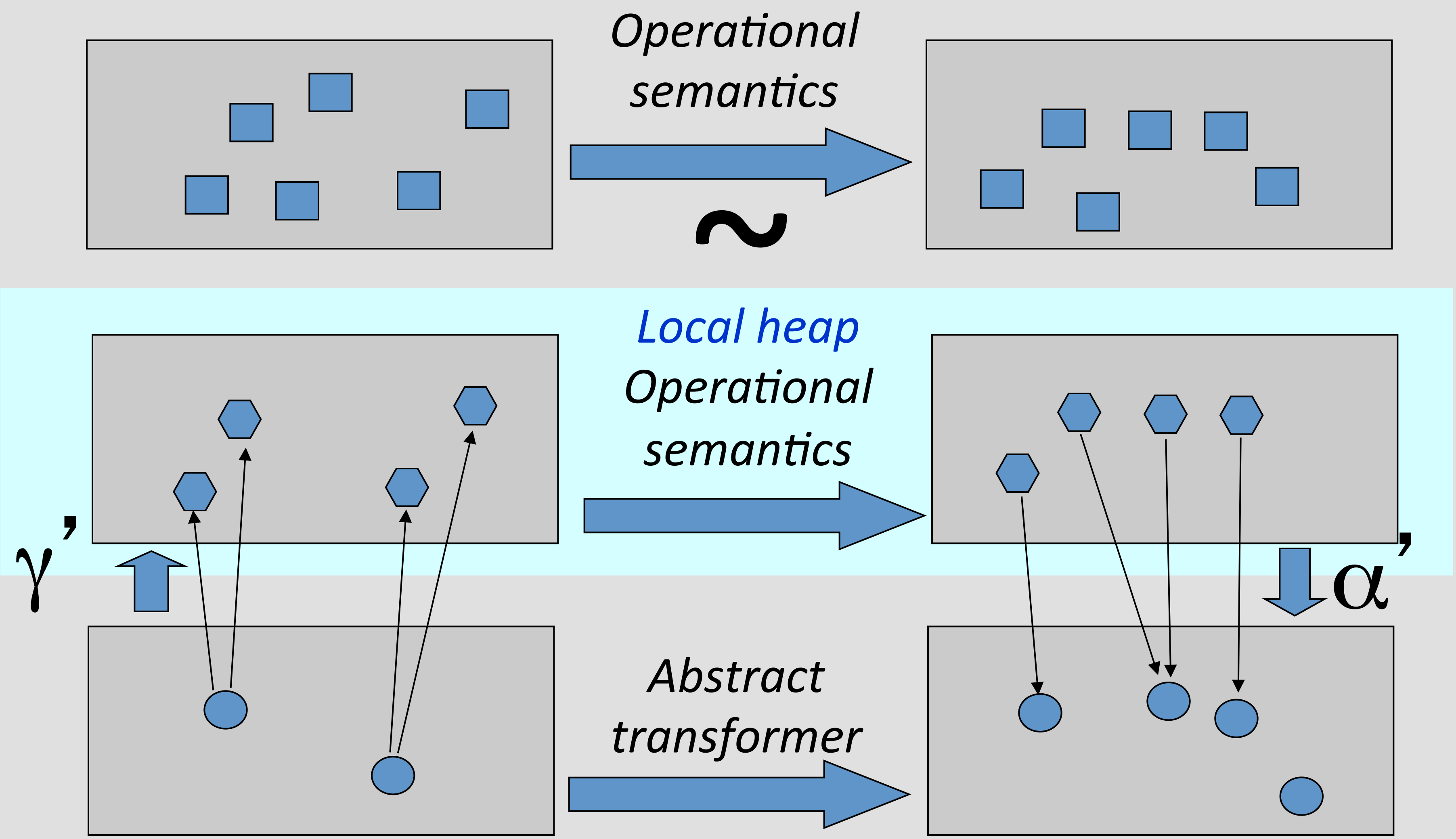
$\sigma_{CPF}$ and $\sigma_{GSB}$ observationally equivalent when for every access paths $AP_1$, $AP_2$

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

# Observational equivalence

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

# Introducing local heap semantics



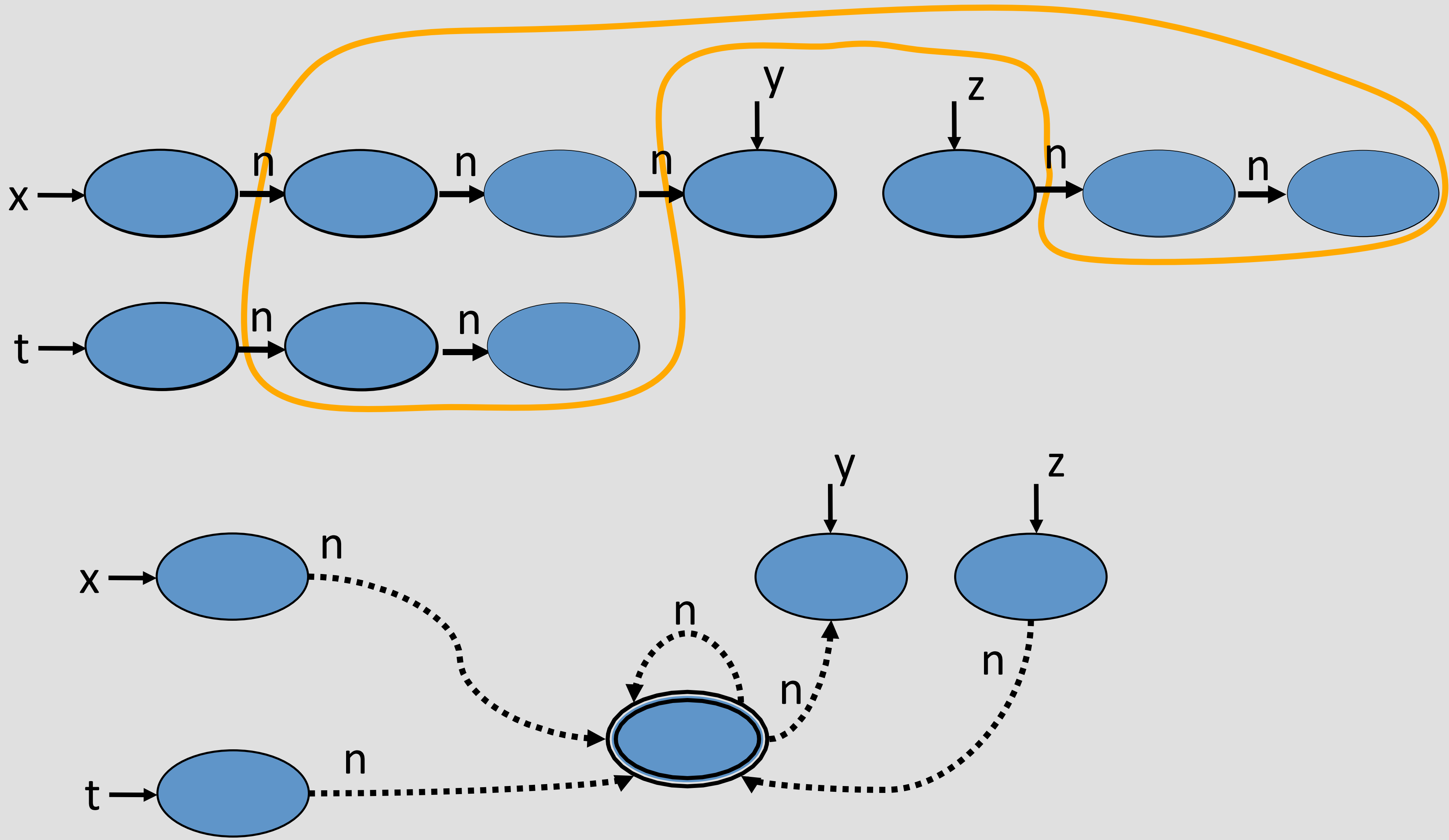
# Shape abstraction

- 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

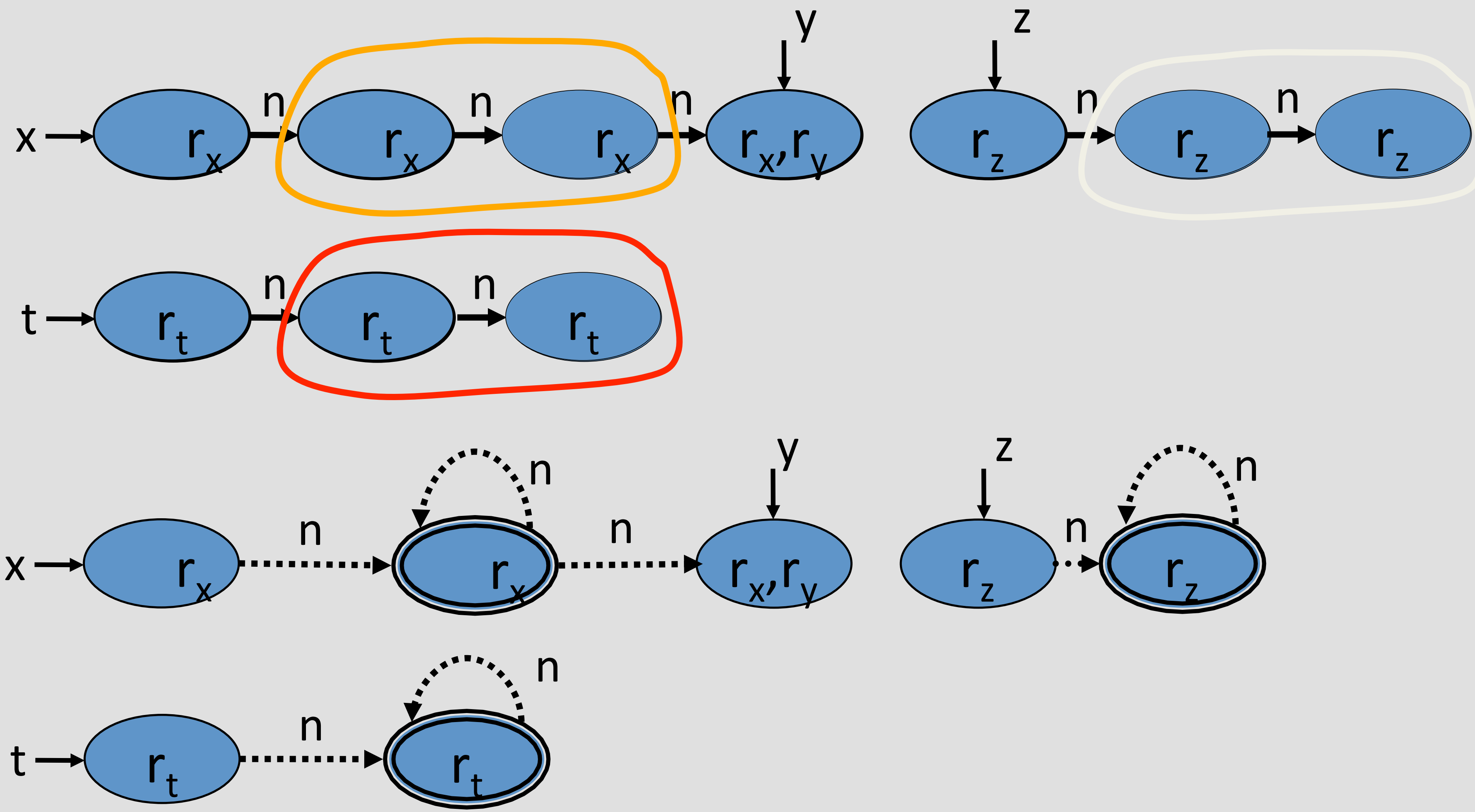


# Instrumentation predicates

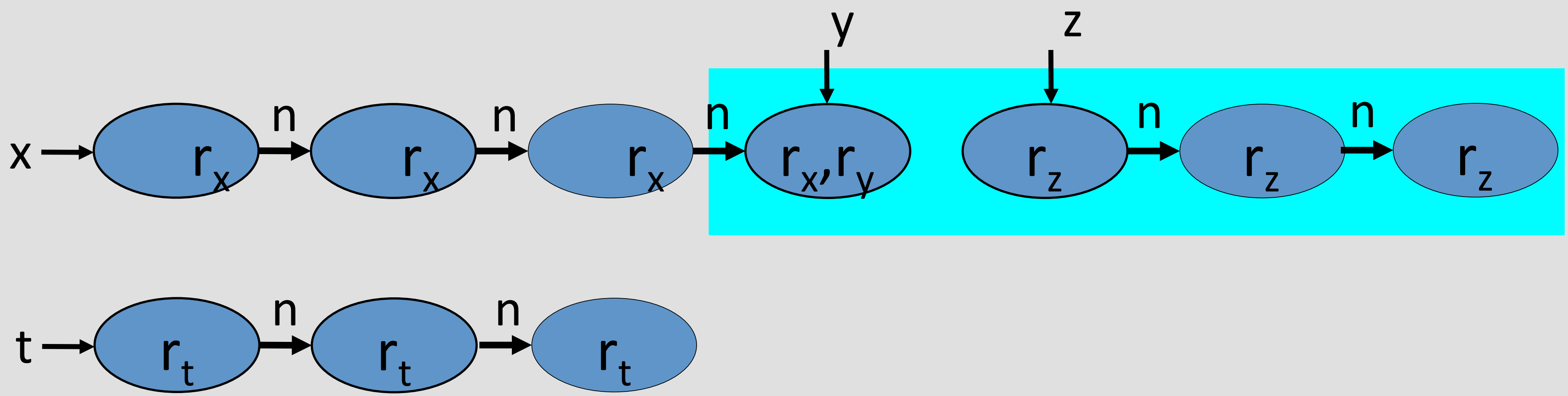
- Record derived properties
- Refine the abstraction
  - Instrumentation principle [SRW, TOPLAS'02]
- Reachability is central!

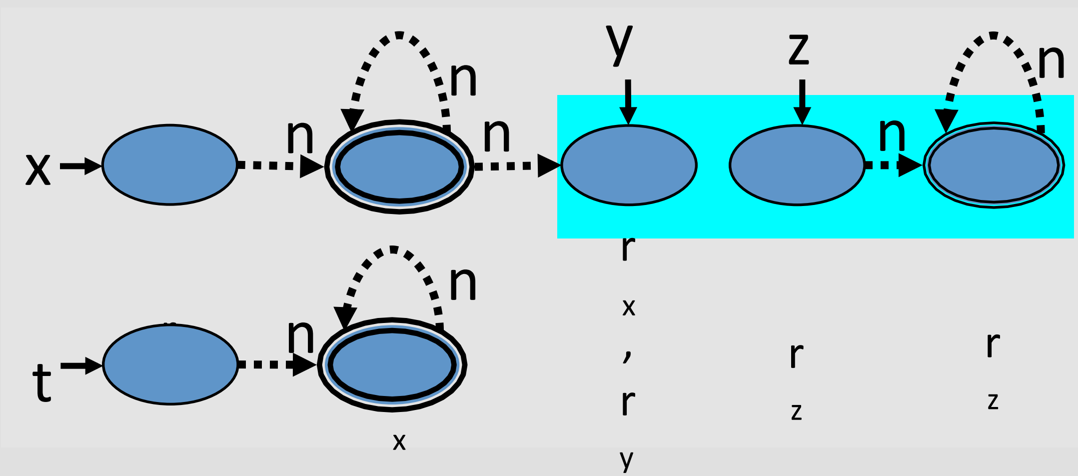
| Predicate | Meaning |
|---|---|
| $r_x(v)$ | $v$ is reachable from variable x |
| $r_{obj}(v_1,v_2)$ | $v_2$ is reachable from $v_1$ |
| ils($v$) | $v$ is heap-shared |
| c($v$) | $v$ resides on a cycle |

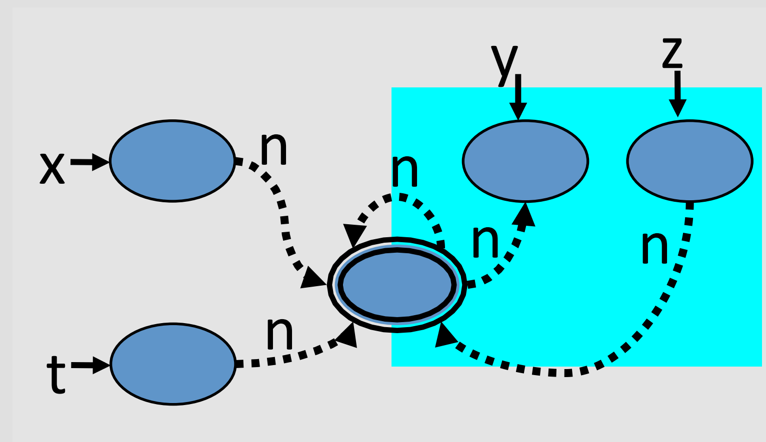
# Abstract memory states (with reachability)



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



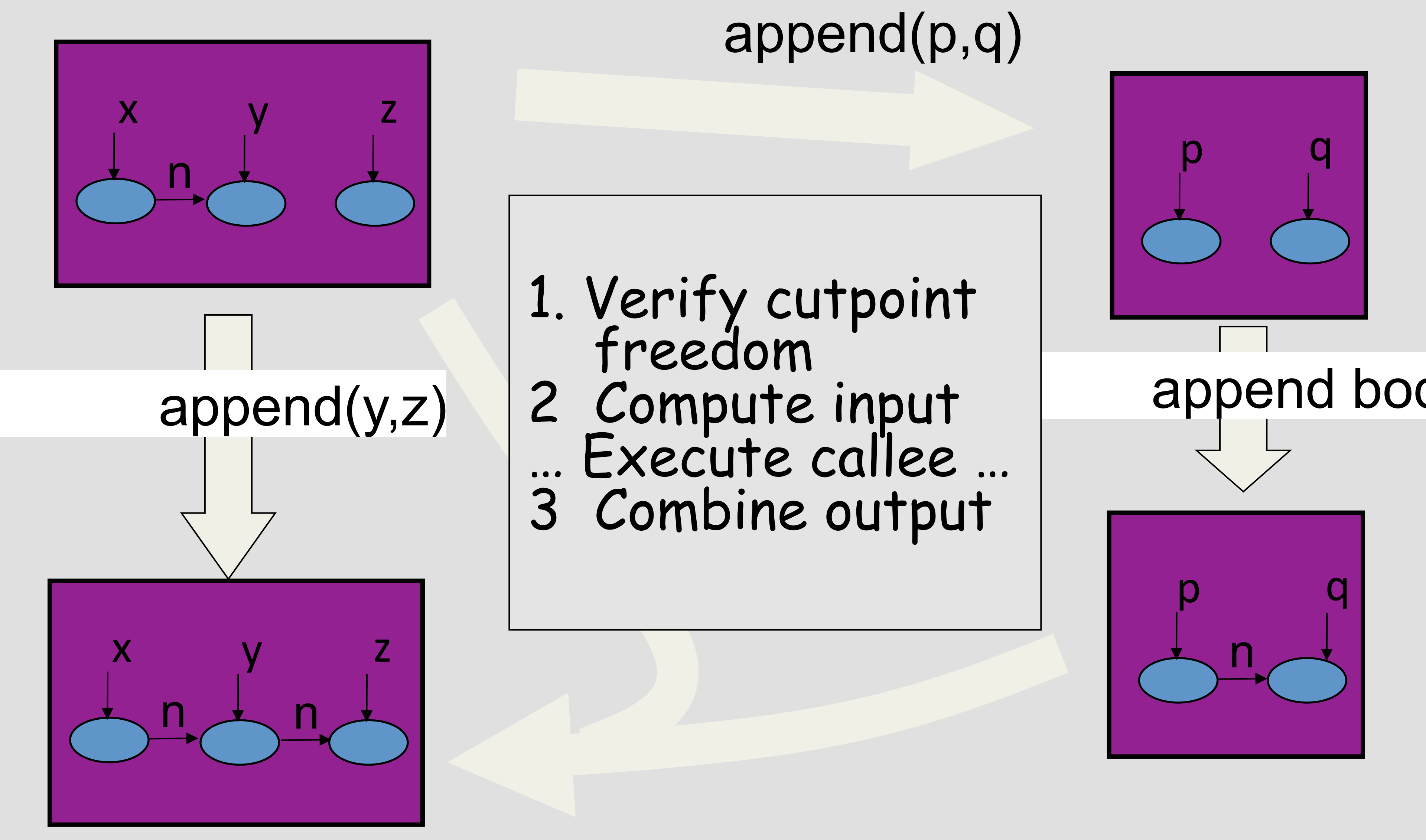



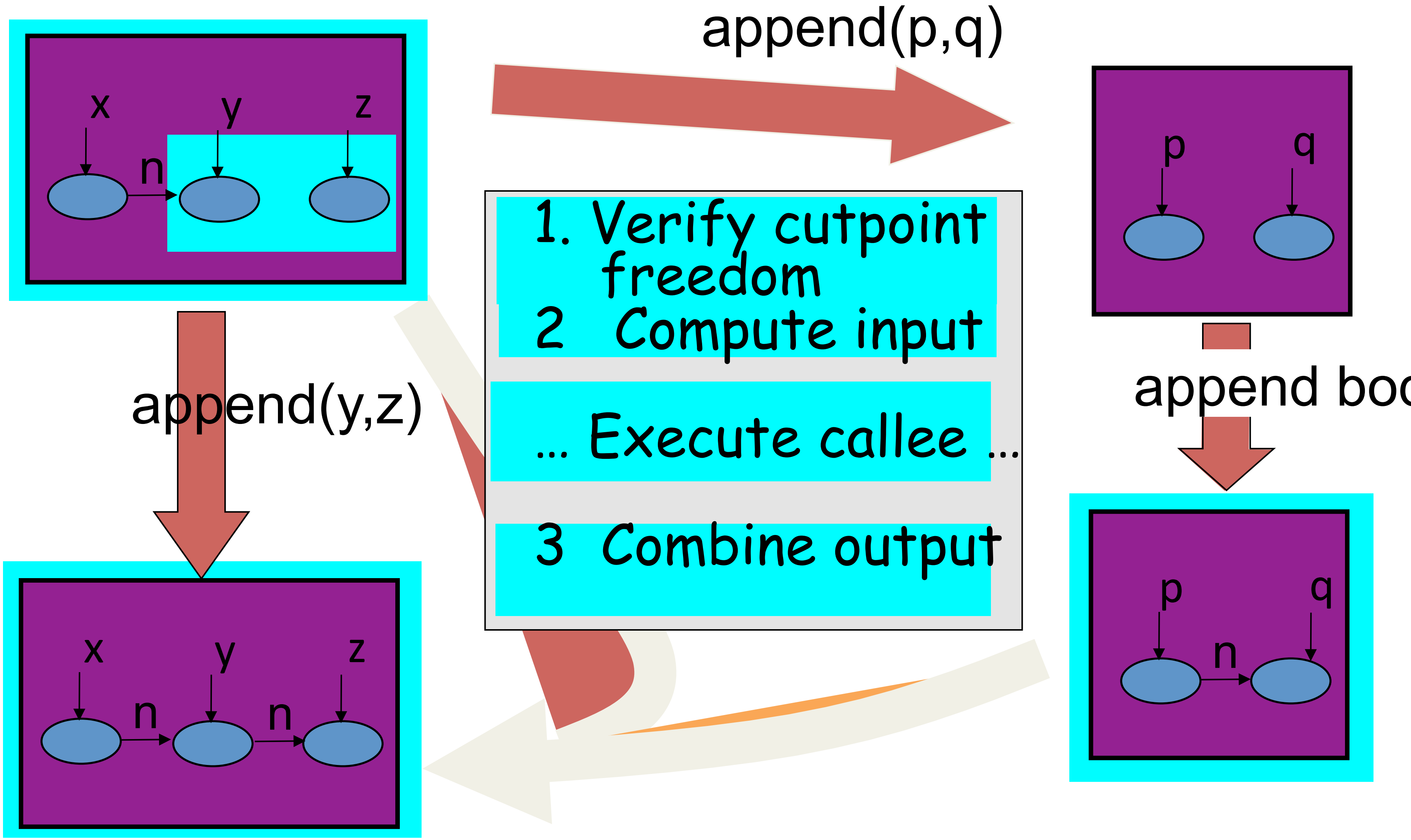
# Abstract semantics

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

# Procedure calls

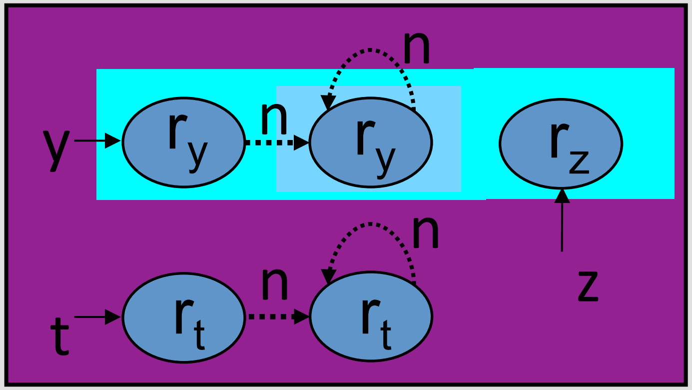


# Procedure calls

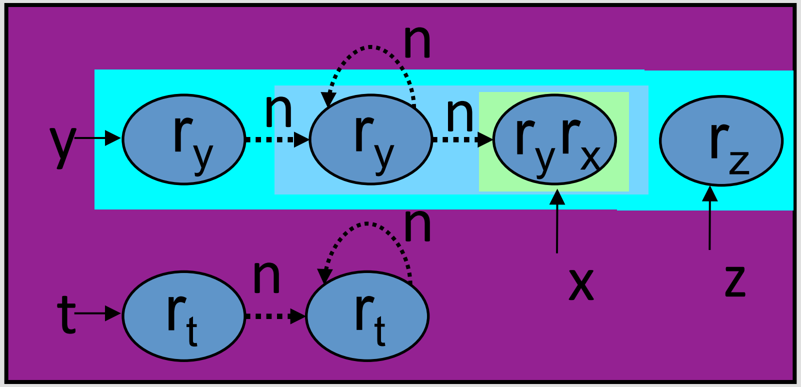


# Conservative verification of cutpoint-freedom

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

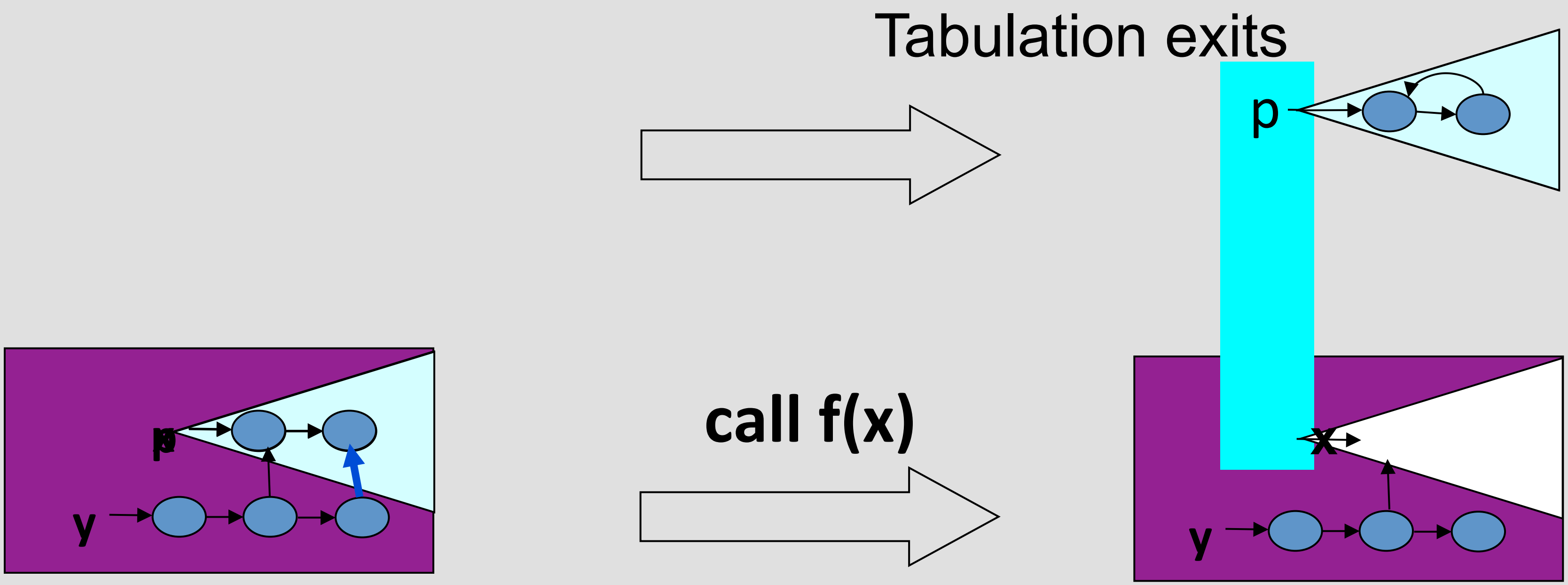



**Cutpoint free**

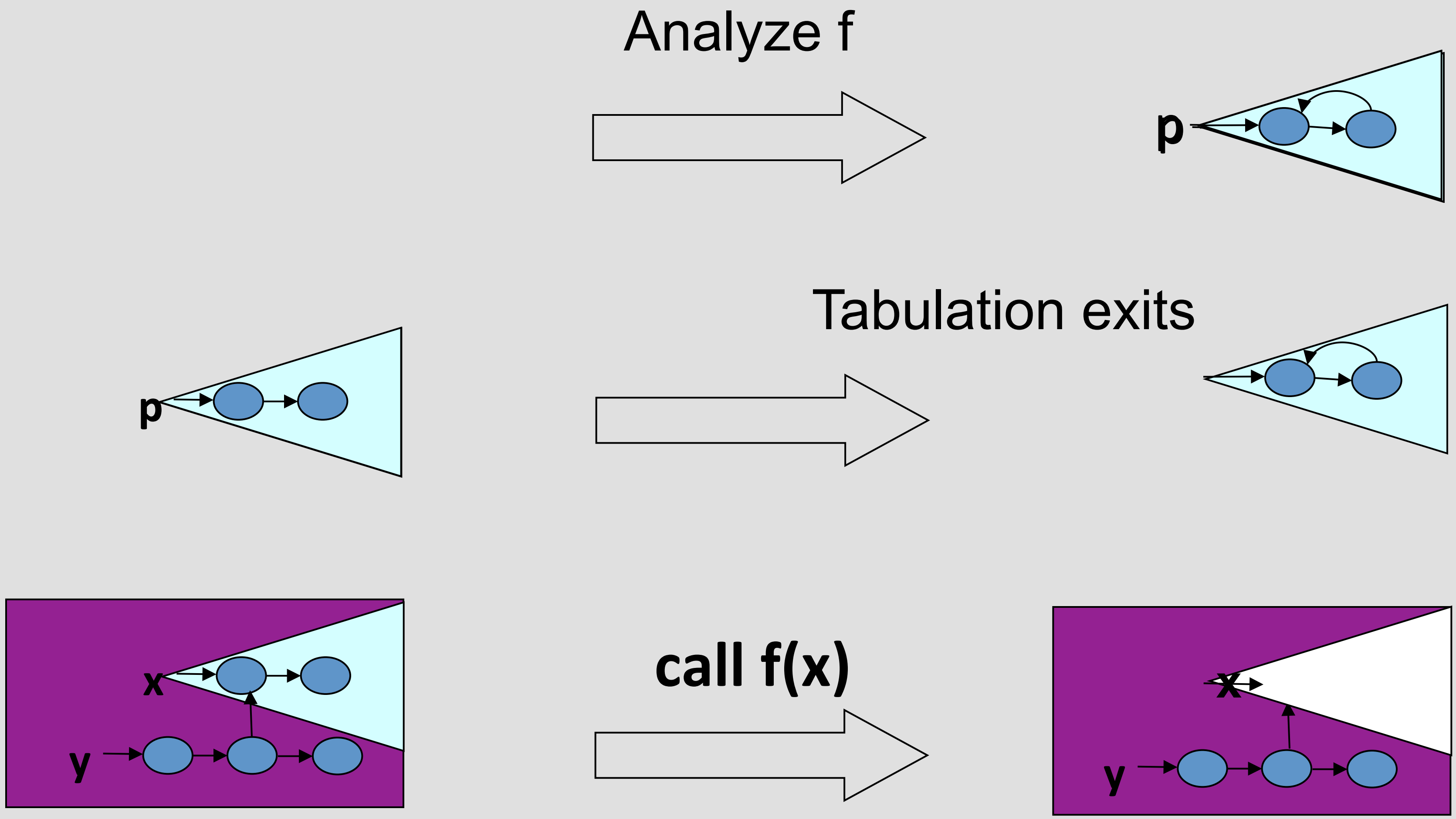



**Not Cutpoint free**

# Interprocedural shape analysis

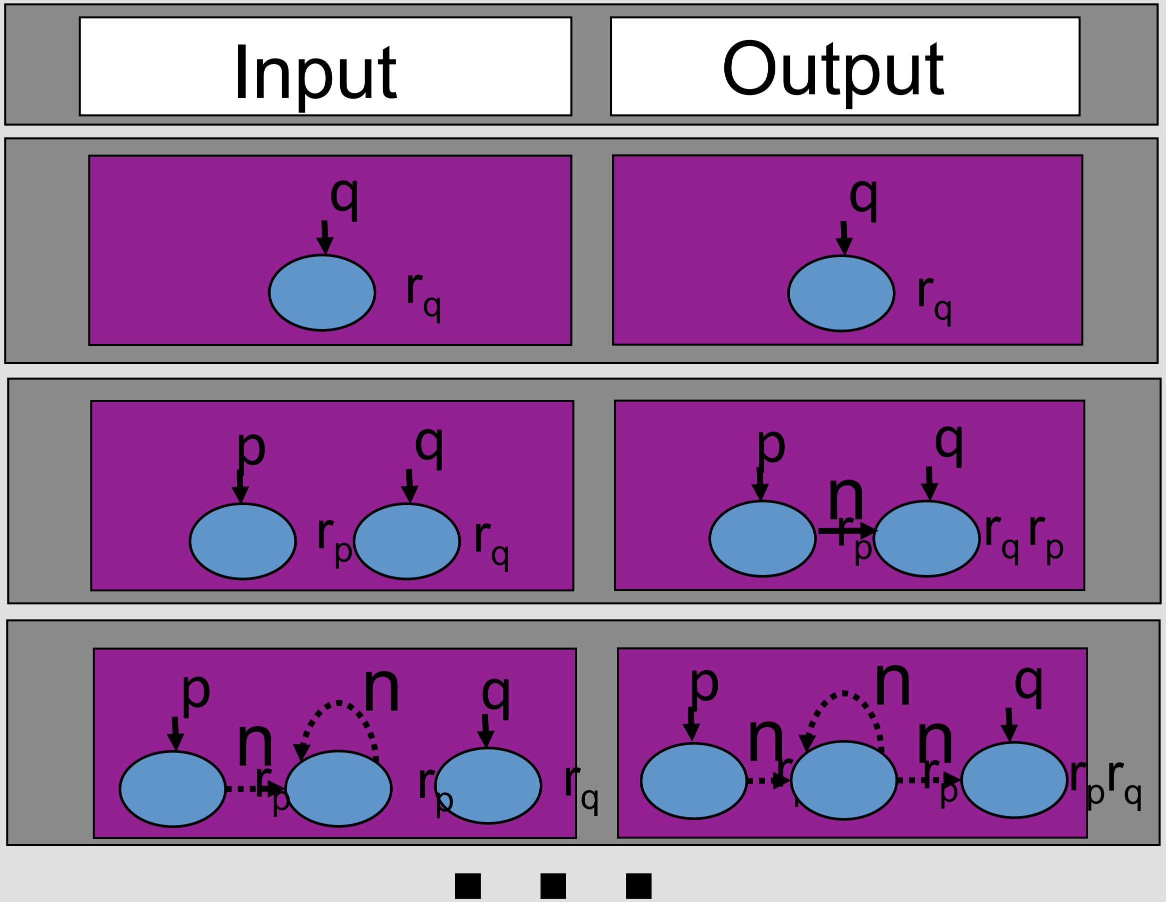


# Interprocedural shape analysis



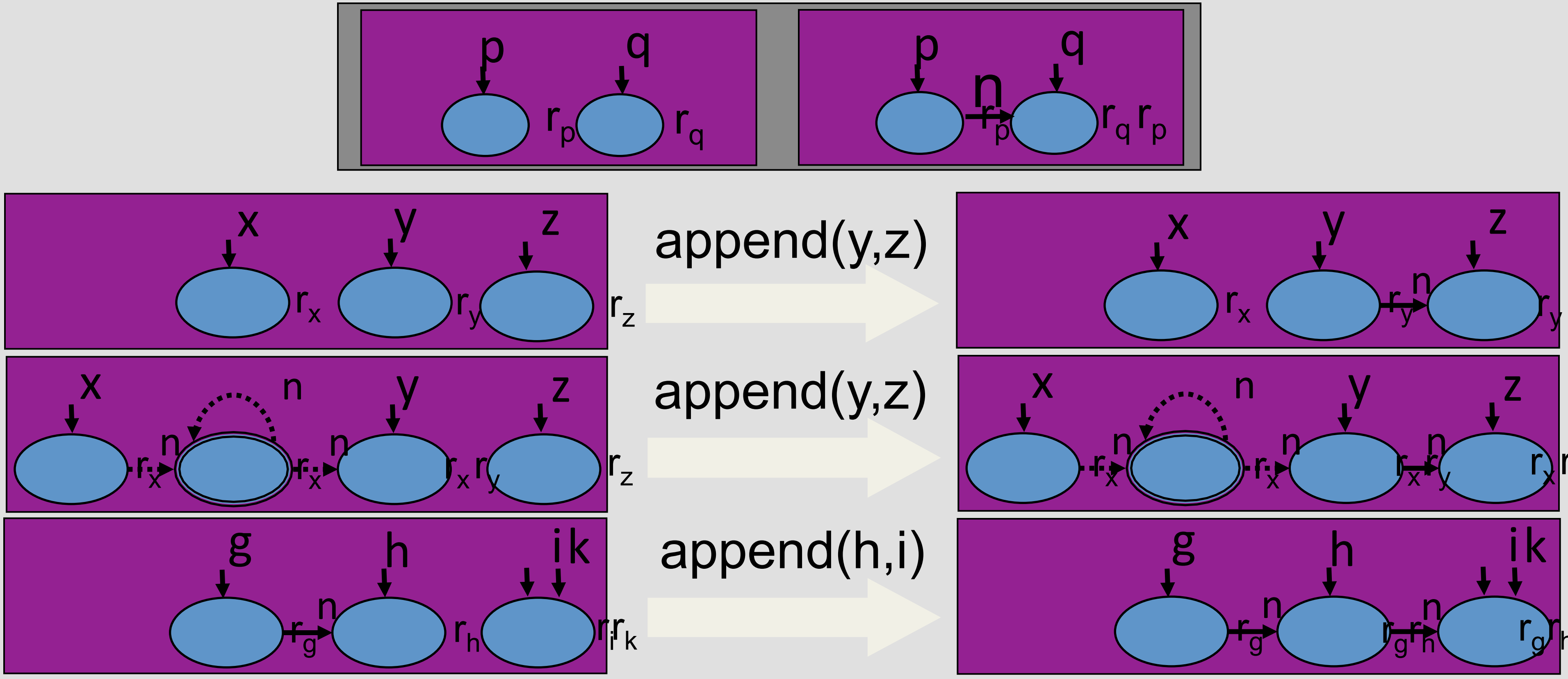
# Interprocedural shape analysis

- Procedure ≡ input/output relation



# Interprocedural shape analysis

- Reusable procedure summaries
  - Heap modularity



# Plan

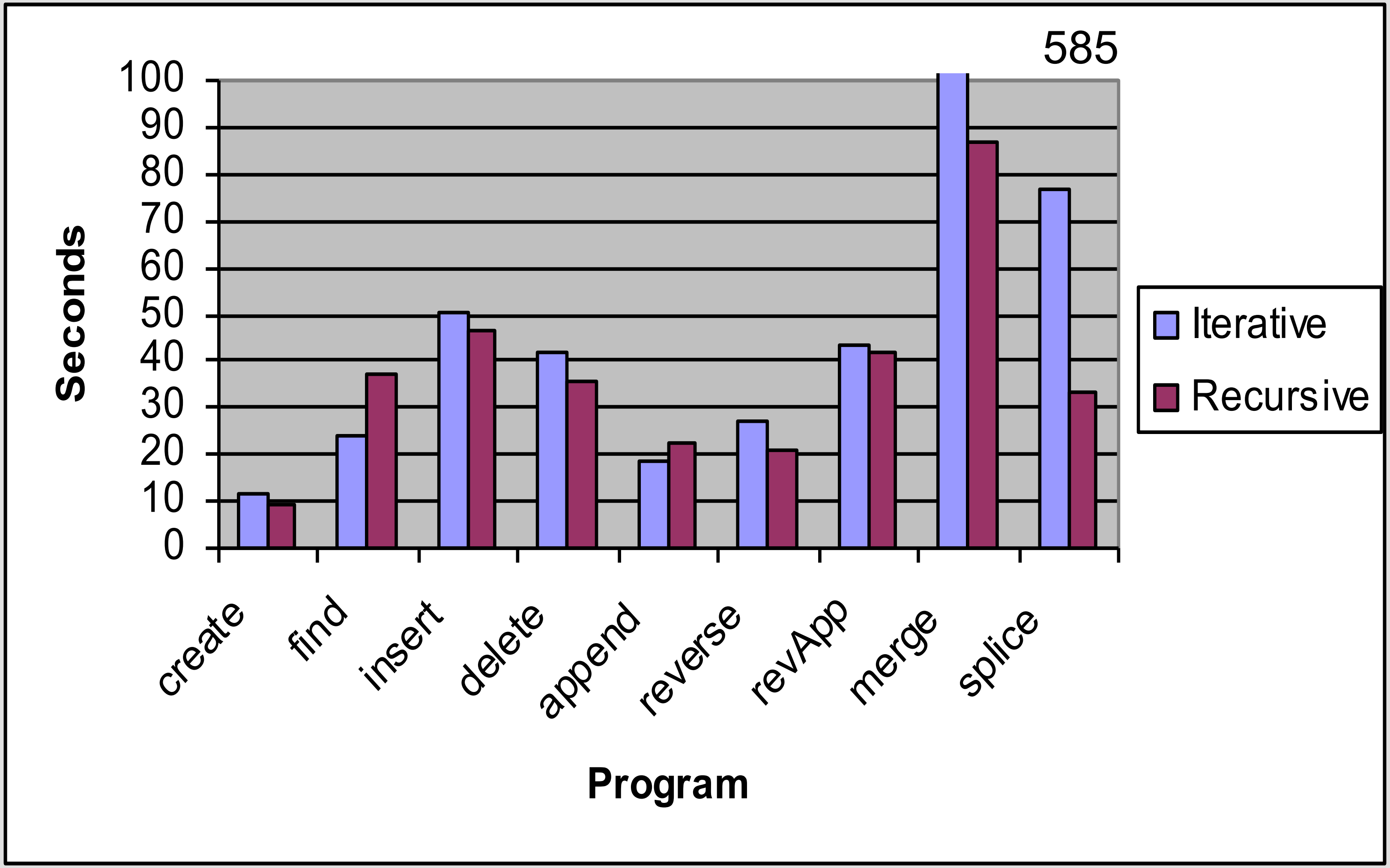
- ✓ Cutpoint freedom
- ✓ Non-standard concrete semantics
- ✓ Interprocedural shape analysis
- Prototype implementation

# Prototype implementation

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

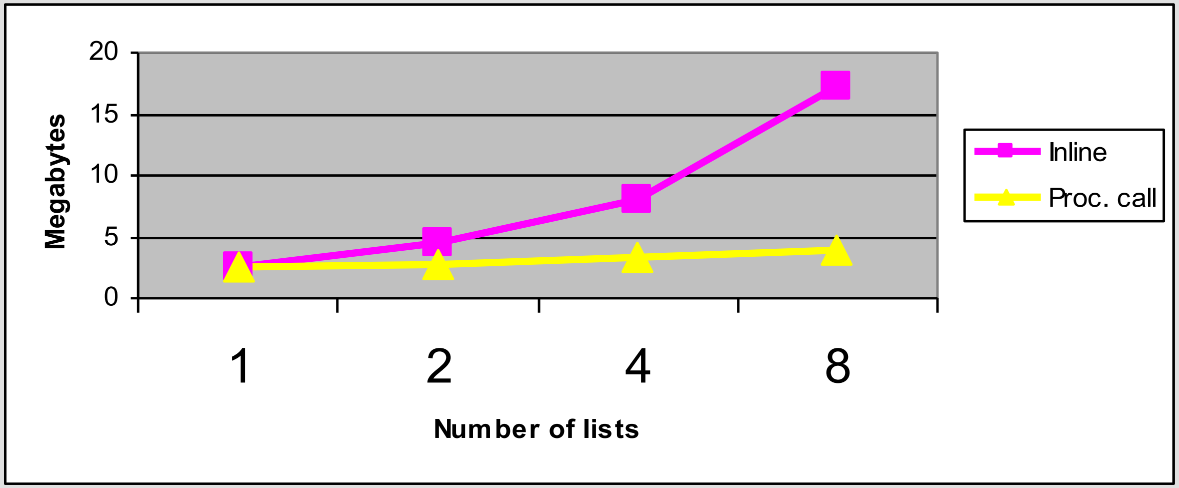
| Data structure | Verified properties |
|---|---|
| Singly linked list | Cleanness, acyclicity |
| Sorting (of SLL) | + Sortedness |
| Unshared binary trees | Cleaness, tree-ness |

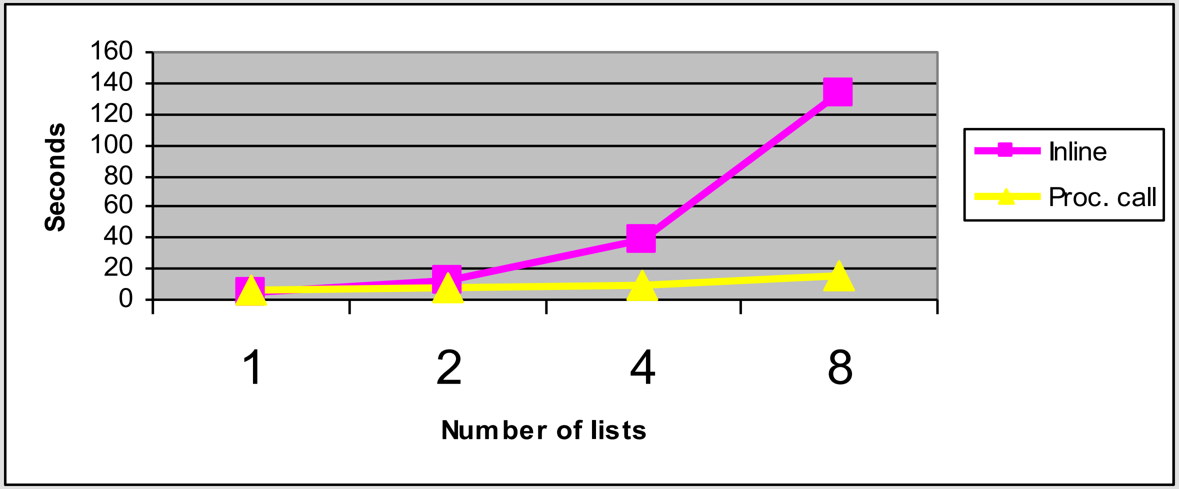
# Iterative vs. Recursive (SLL)



# Inline vs. Procedural abstraction

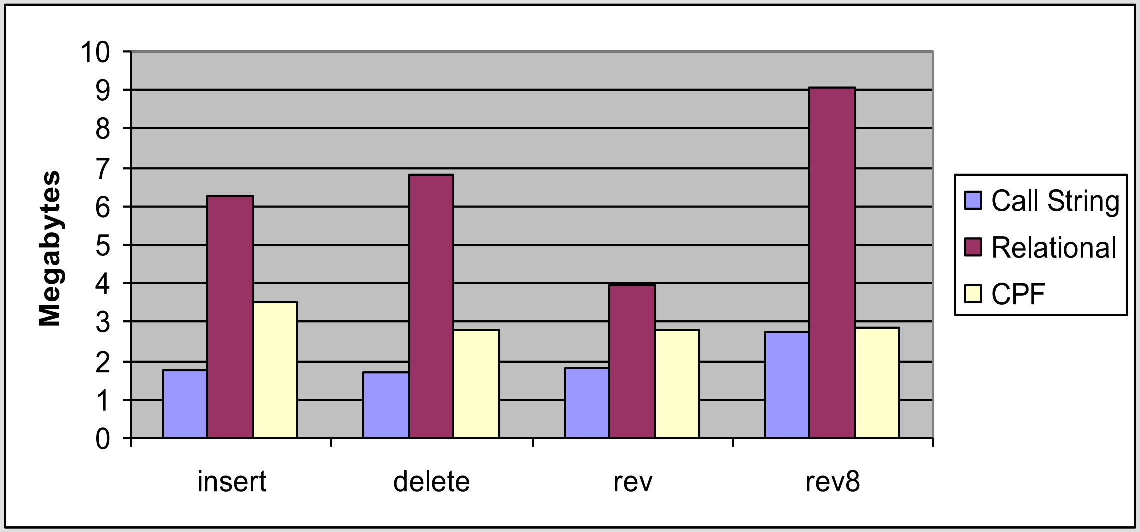
```
// Allocates a list of
// length 3
List create3(){
…
}

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

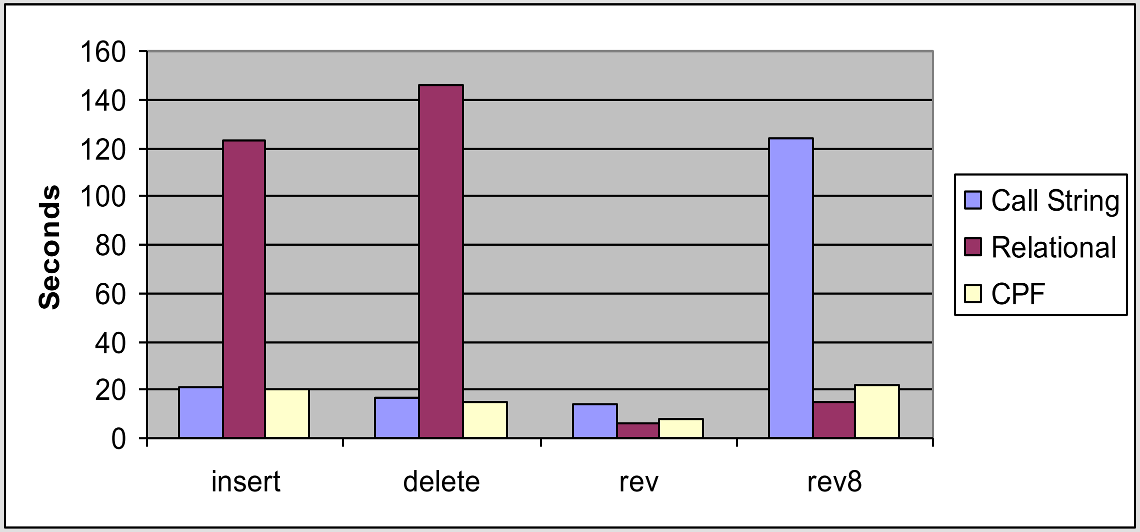


# Call string vs. Relational vs. CPF

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



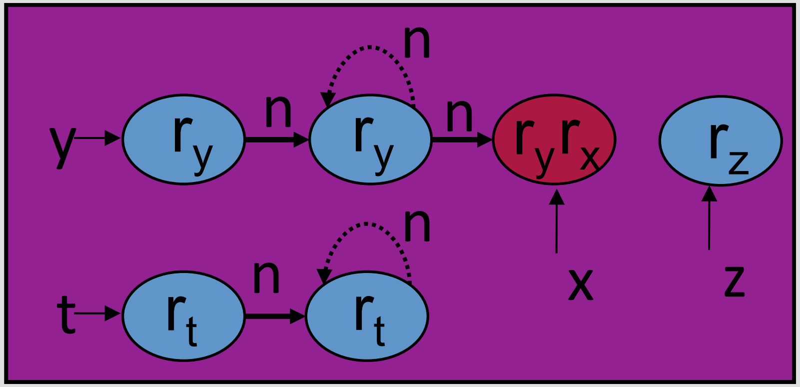
# Related Work

- **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
  - ...

# Future work

- Bounded number of cutpoints
- False cutpoints
  - Liveness analysis

```
append(y,z);
x = null;
```



# 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, Bauer, Reps, Sagiv, Wilhelm POPL'05
    - Cutpoints
  - Rinetzky and Sagiv, CC ' 01
    - Global heap
  - Jeannet et al., SAS ' 04
    - Local heap, relational
  - Chong and Rugina, SAS ' 03
    - Local heap
- **Local reasoning**
  - Ishtiaq and O' Hearn, POPL '01
  - Reynolds, LICS ' 02

# Summary

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

# Project

- 1-2 Students in a group
- 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