Principles of Shape Analysis

Mooly Sagiv

Thomas Reps

Reinhard Wilhelm

. . . and also

- Universität des Saarlandes
  - J. Bauer, Kreiker,
  - R. Biber
  - S. Parduhn
  - R. Seidel
  - J. Reineke
- University of Wisconsin
  - F. DiMaio
  - D. Gopan
  - A. Loginov
- IBM Research
  - J. Field
  - H. Kolodner
  - M. Rodeh
- Inria
  - B. Jeannet
- Berkeley
  - B. McCloskey
  -
- Tel-Aviv University
  - D. Amit
  - I. Bogudlov
  - G. Arnold
  - G. Erez
  - N. Dor
  - T. Lev-Ami
  - R. Manevich
  - R. Shaham
  - A. Rabinovich
  - N. Rinetzky
  - E. Yahav
  - G. Yorsh
  - A. Warshavsky
- Microsoft Research
  - J. Berdine
  - B. Cook
  - G. Ramalingam

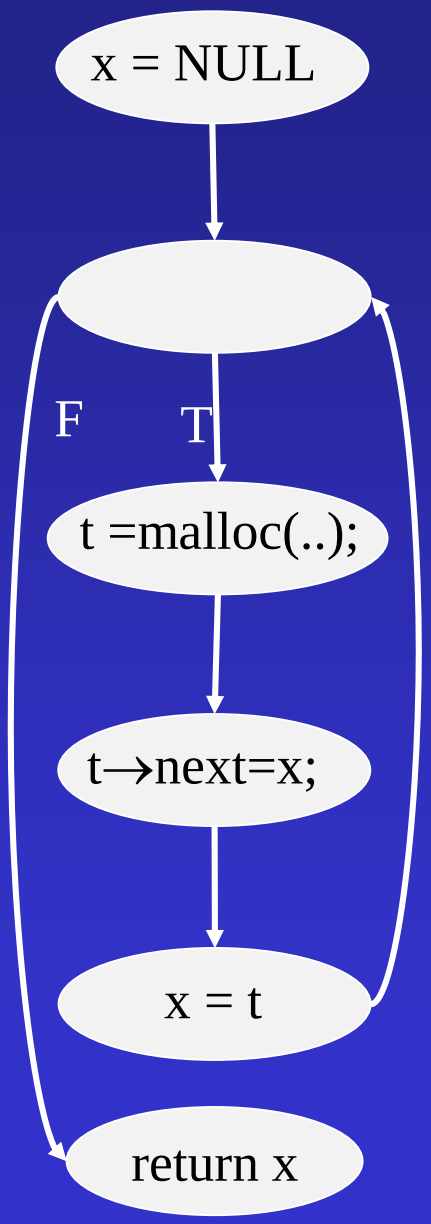
Original Problem: Shape Analysis (Jones and Muchnick 1981)

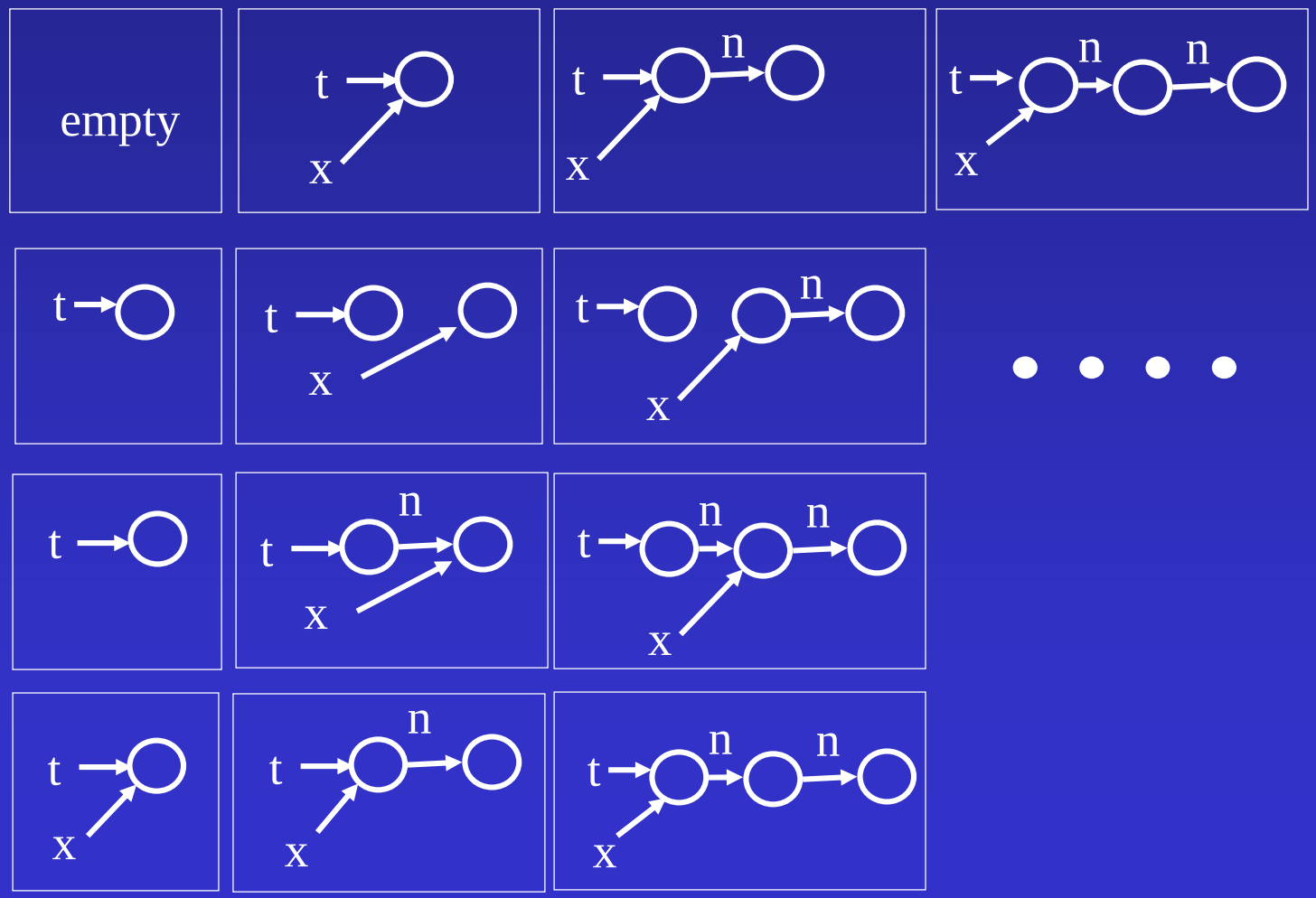
- Characterize dynamically allocated data
  - *x* points to an acyclic list, cyclic list, tree, dag, etc.
  - show that data-structure invariants hold
- Identify may-alias relationships
- Establish "disjointedness" properties
  - *x* and *y* point to structures that do not share cells
- Memory Safety
  - No null and dangling de-references
  - No memory leaks
- In OO programming
  - Everything is in the heap ➔ requires shape analysis

Why Bother?

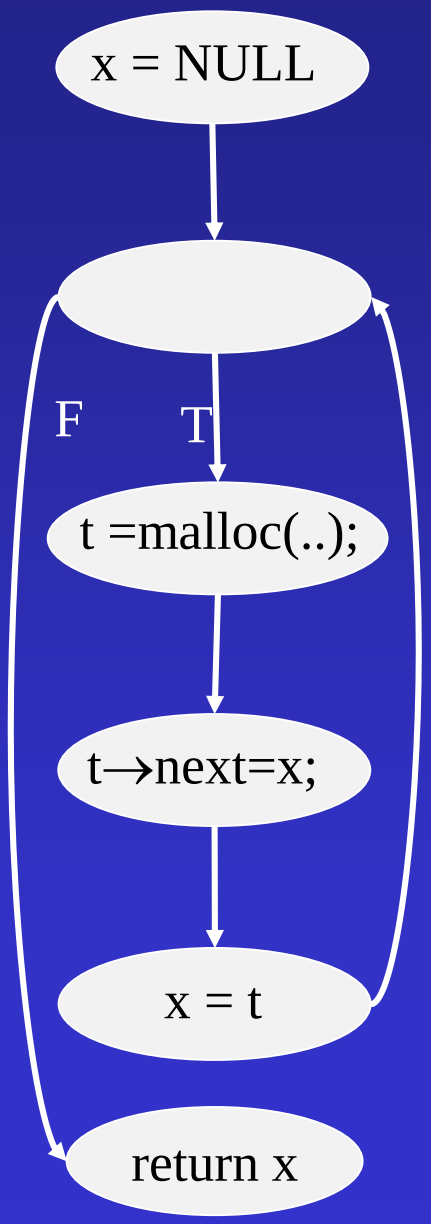
```
    int *p,  *q;
    q = (int *) malloc();
    p = q;
l1: *p = 5;
     p = (int *) malloc();
l2: printf(*q);  /* printf(5) */
```

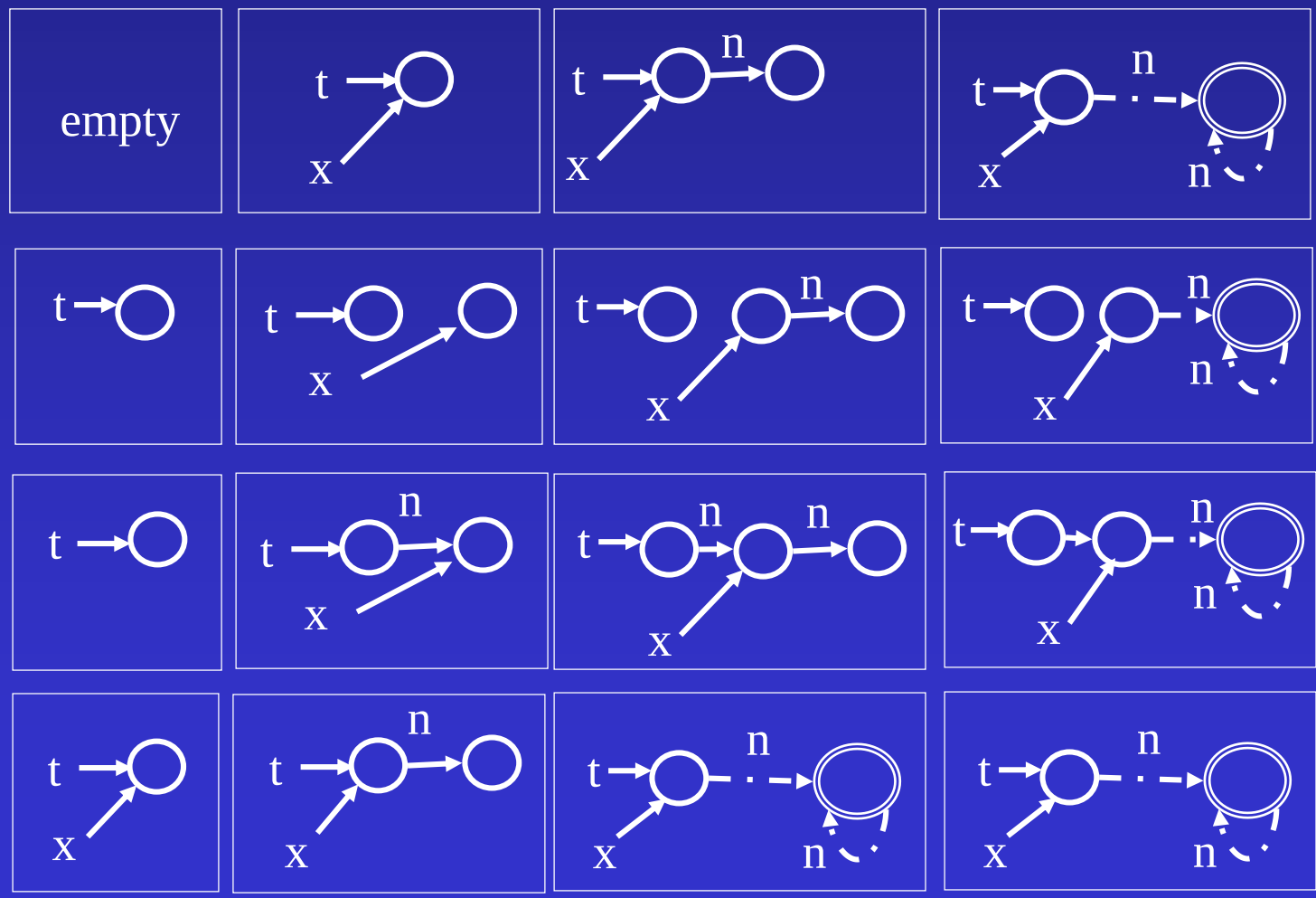
Example: Concrete Interpretation





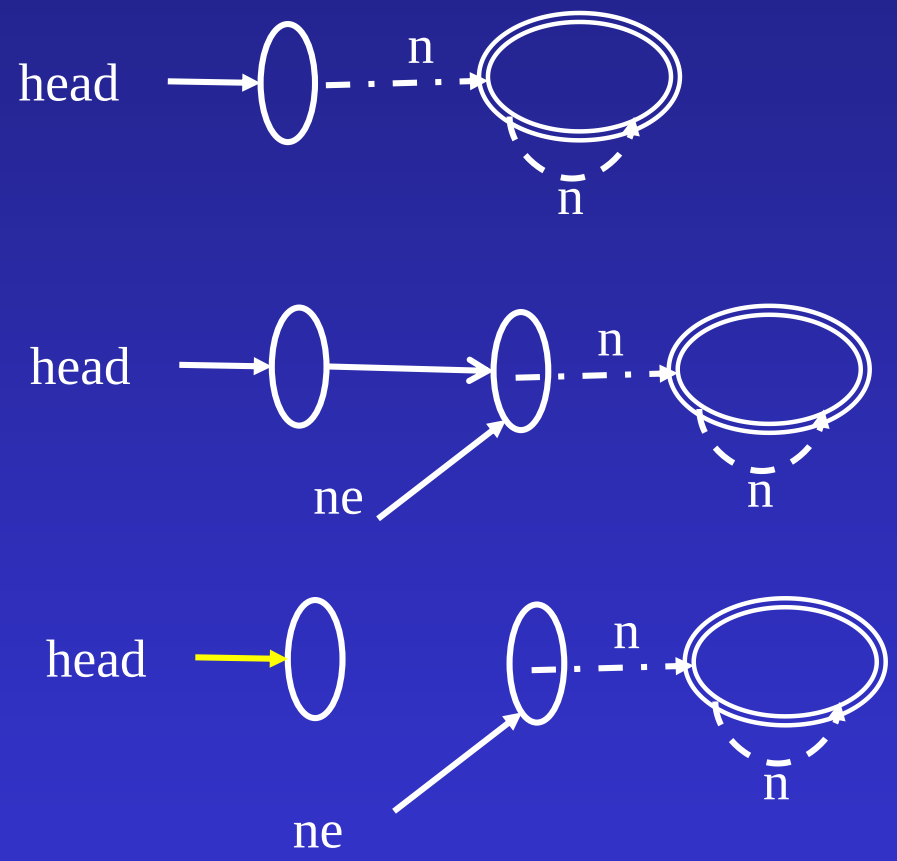
Example: Abstract Interpretation





Memory Leakage

```
List reverse(Element *head)
{
    List rev, ne;
    rev = NULL;
    while (head != NULL) {
        ne = head →next;
        head → next = rev;
        head = ne;
        rev = head;
    }
    return rev;
}
```



leakage of address pointed to by head

Memory Leakage

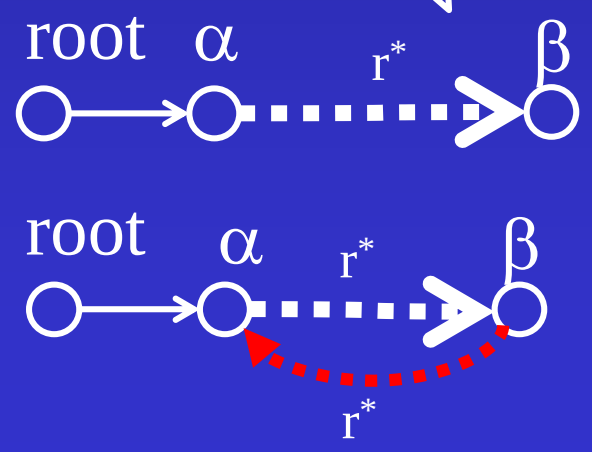
```
Element* reverse(Element *head)
{
   Element *rev, *ne;
    rev = NULL;
    while (head != NULL) {
          ne = head → next;
          head → next = rev;
          rev = head;
          head = ne;
    }
    return rev;
}
```

No memory leaks

Directed Reachability

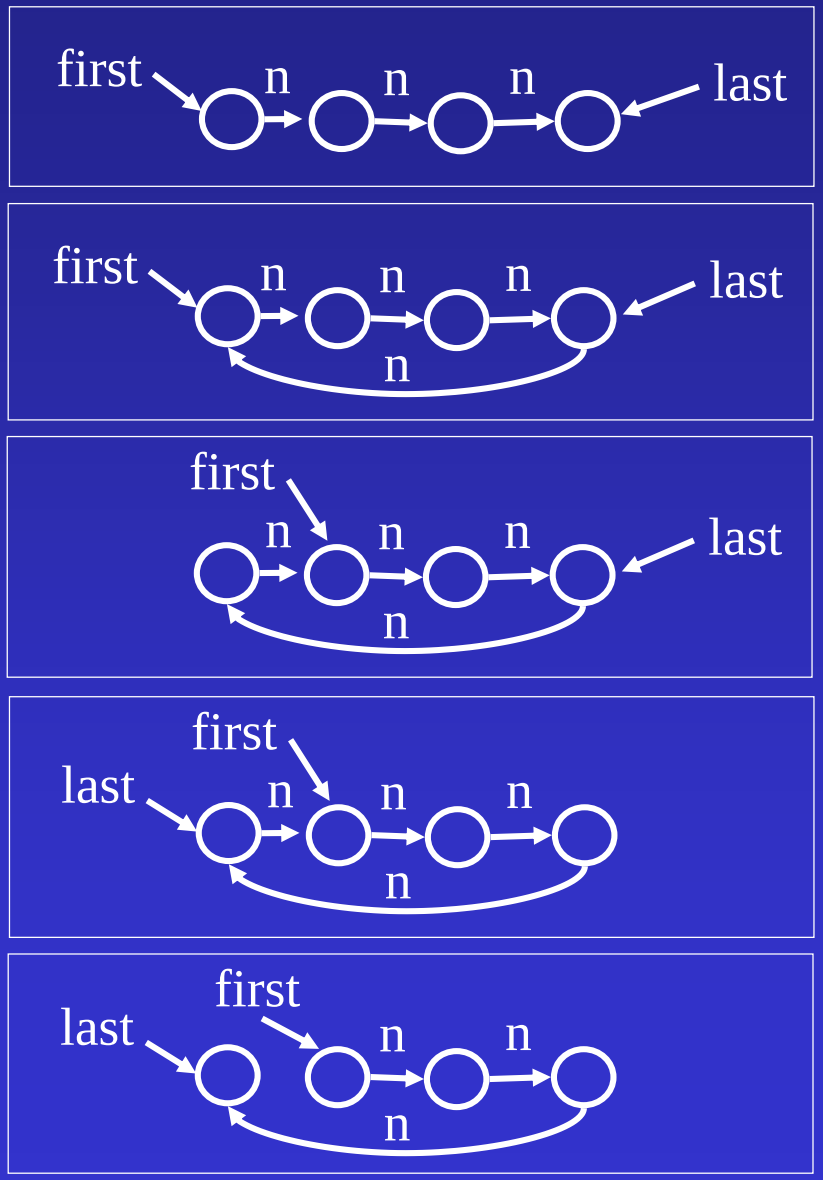
- Directed reachabiliy suffice to describe many properties of data structures
  - Absence of garbage
    - $\forall x$: r*(root, x)
  - Acyclicity
    - $\forall x$: $x \neq$ root $\Rightarrow \neg$r*(root, x)
  - Data Structure Invariants
    - $\forall x$: f*(root, x) $\Leftrightarrow$ b*(root, x)

r*(x, y) denotes a finite directed path of relation of r from x to y



Reachability in Dynamically Evolving Graphs

```
rotate(List first, List last) {
    if ( first != NULL) {
        last → next = first;
        first = first → next;
        last = last → next;
        last → next = NULL;
    }
 assert acyclic first;
}
```



Logical Structures (Labeled Graphs)

Fixed

- Nullary relation symbols
- Unary relation symbols
- Binary relation symbols
- FO^{TC} over TC, $\forall \exists \neg \wedge \vee$ express logical structure properties
- Logical Structures provide meaning for relations
  - A set of individuals (nodes) U
  - Interpretation of relation symbols in P

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

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

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

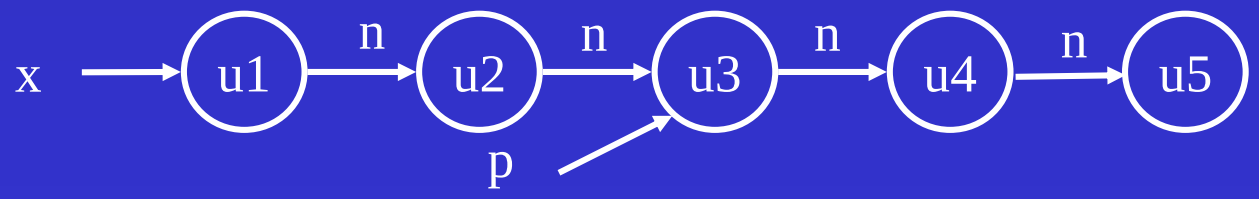
Representing Stores as Logical Structures

- Locations ≈ Individuals
- Program variables ≈ Unary relations
- Fields ≈ Binary relations
- Example
  - U = {u1, u2, u3, u4, u5}
  - x = {u1}, p = {u3} n = {<u1, u2>, <u2, u3>, <u3, u4>, <u4, u5>}

	x
u1	1
u2	0
u3	0
u4	0
u5	0

	p
u1	0
u2	0
u3	1
u4	0
u5	0

n	u1	u2	u3	u4	u5
u1	0	1	0	0	0
u2	0	0	1	0	0
u3	0	0	0	1	0
u4	0	0	0	0	1
u5	0	0	0	0	0



Interesting Properties

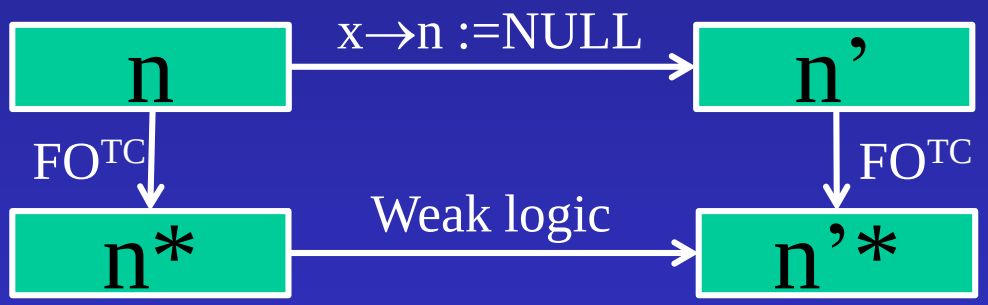
```
rotate(List first, List last) {
    if ( first != NULL) {
        last → next = first;
        first = first → next;
        last = last → next;
        last → next = NULL;
    }
}
```

- ✓No null-de references
- ✓No memory leaks
- ✓Returns an acyclic linked list
- ✓Partially correct

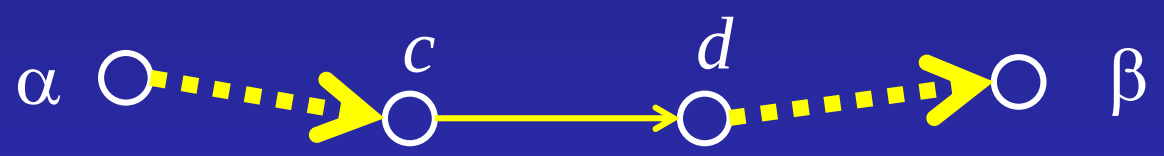
Reasoning about Directed Reachability is hard

- Not first order expressible
- Not recursively enumerable
- Not clear if can be updated in first order logic
- What about modularity
  - Do we need to reason about the calling context?

Incremental Reachabilty Small (local) updates

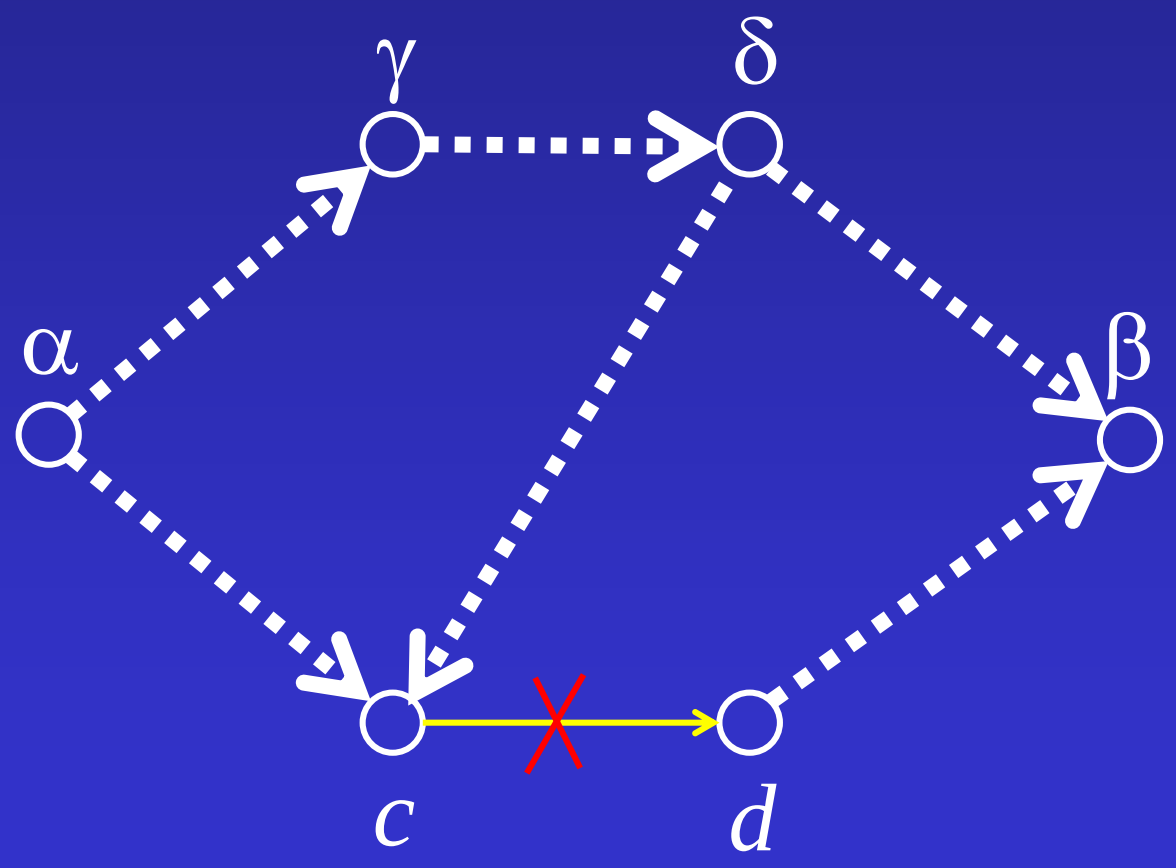


Adding an edge
$c \rightarrow n = d$



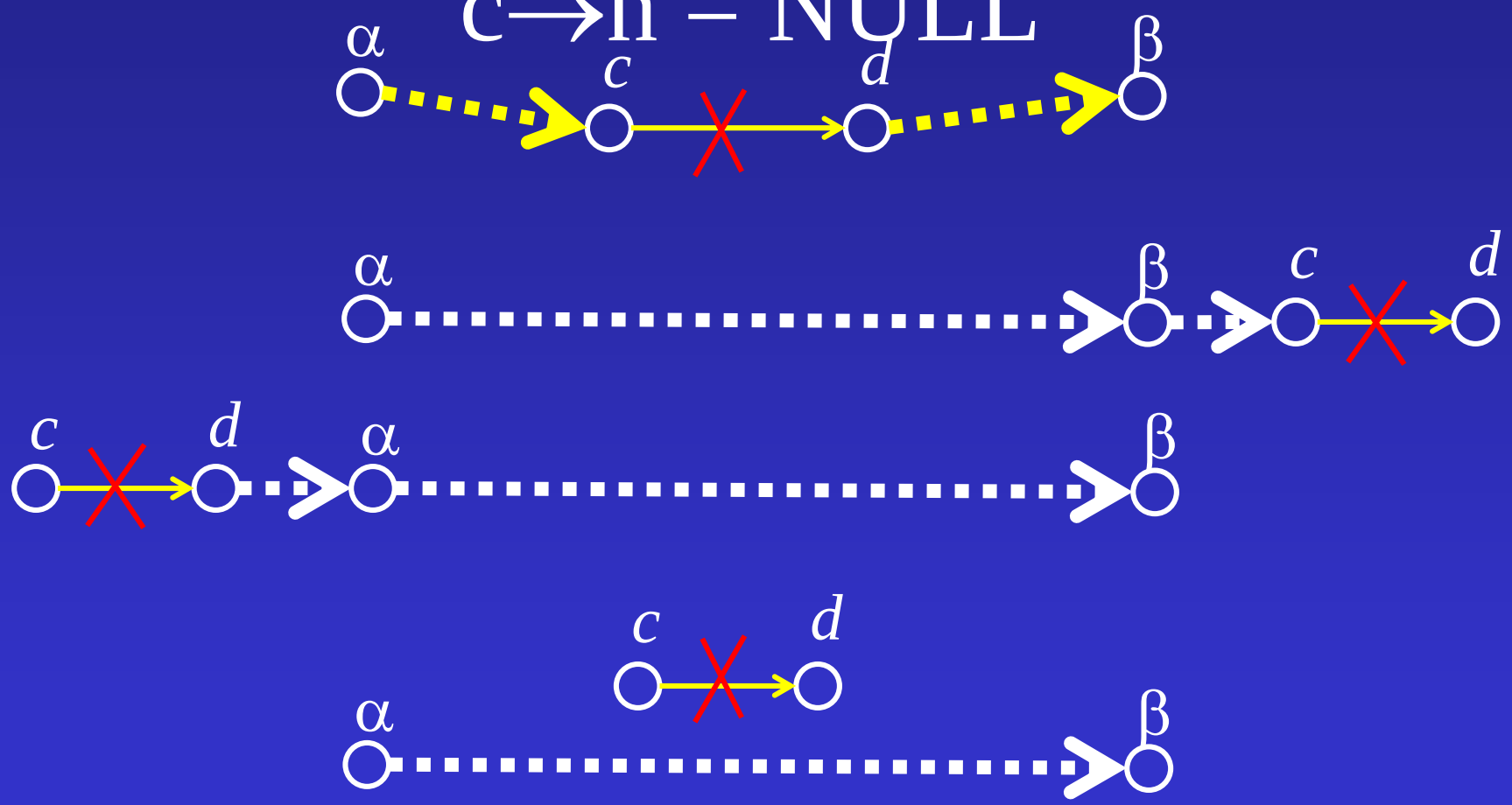
$$n^*(\alpha,\beta) \leftrightarrow \underline{n}^*(\alpha,\beta) \vee (\underline{n}^*(\alpha, c) \wedge \underline{n}^*(d,\beta)$$

Updating Directed Reachabilty in General Graph is Hard



Removing an edge (destructive update)

$c \rightarrow n = NULL$



$$n^*(\alpha,\beta) \leftrightarrow \underline{n}^*(\alpha,\beta) \wedge \neg(\underline{n}^*(\alpha,c) \wedge \underline{n}^+(c,\beta))$$

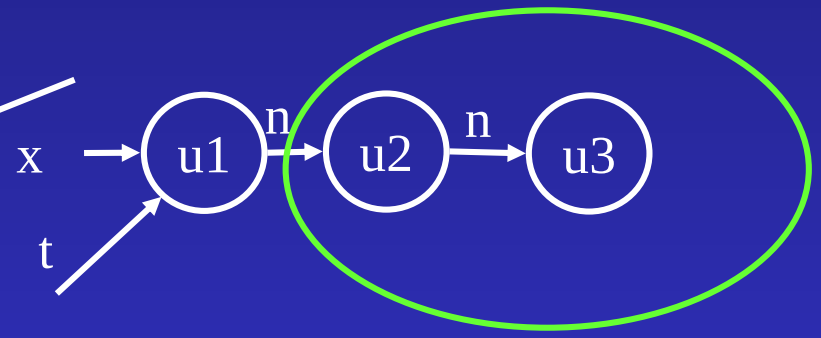
Canonical Abstraction

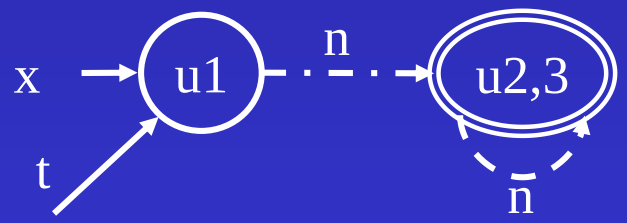
- Model the heap as a set of relations evolve over time
- Not just binary
- Every relation has three values $0 \sqcup 1 = ½$ (don't know)
- Partition the individuals into equivalence classes based on the values in the unary relations
  - Every individual is mapped to its equivalence class
- Combine relations via $\sqcup$
  - $p^S(u'_1, ..., u'_k) = \sqcup \{p^B(u_1, ..., u_k) \mid f(u_1)=u'_1, ..., f(u_k)=u'_k\}$
- At most 2^A abstract individuals
- The basis of Three-Valued-Analysis (TVLA)

[TOPLAS'02] S. Sagiv, T.W. Reps, R. Wilhelm: Parametric Shape Analysis via 3-Valued Logic
[SAS'00] T. Lev-Ami, S. Sagiv: TVLA: A System for Implementing Static Analyses

Canonical Abstraction

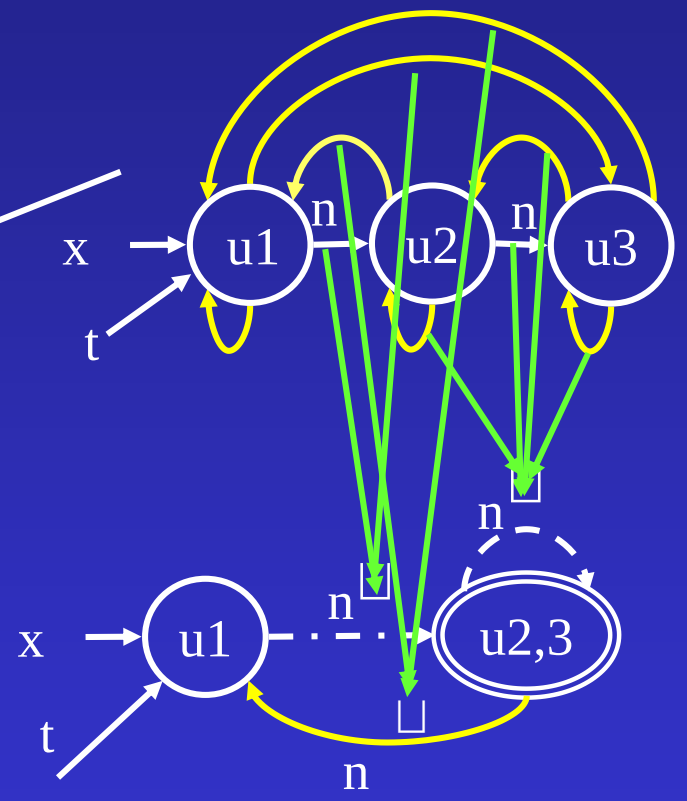
```
x = NULL;
while (…) do {
    t = malloc();
    t →next=x;
    x = t
}
```





Canonical Abstraction

```
x = NULL;
while (…) do {
    t = malloc();
    t →next=x;
    x = t
}
```



$\forall V: (x = V \wedge t = V) \vee (x \neq V \wedge t \neq V)$

$\forall V, W: (x \neq V \wedge t \neq V) \wedge (x = W \wedge t = W) \rightarrow \neg n(V, W)$

Don’t go generic! Domain Specialization

- Many programs manipulate specialized data structures
  - singly, doubly-linked (circular) lists, trees
- Design specialized abstract domains
- Similar to theories in decision procedures

[LPC’93] J. Plevyak, A. Chien, V. Karamcheti: Analysis of Dynamic Structures for Efficient Parallel Execution

[POPL’96,TOPLAS’98] Shmuel Sagiv, Thomas W. Reps, Reinhard Wilhelm: Solving Shape-Analysis Problems in Languages with Destructive Updating

[SAS’02] T. Yavuz-Kahveci, T. Bultan: Automated Verification of Concurrent Linked Lists with Counters.

[VMCAI’05] R. Manevich, E. Yahav, G. Ramalingam, S. Sagiv: Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists

[TACAS’06] D. Distefano, P.W. O'Hearn, H. Yang: A Local Shape Analysis Based on Separation Logic

[CAV’06] T.Lev-Ami, N.Immerman, S. Sagiv: Abstraction for Shape Analysis with Fast and Precise Transformers.

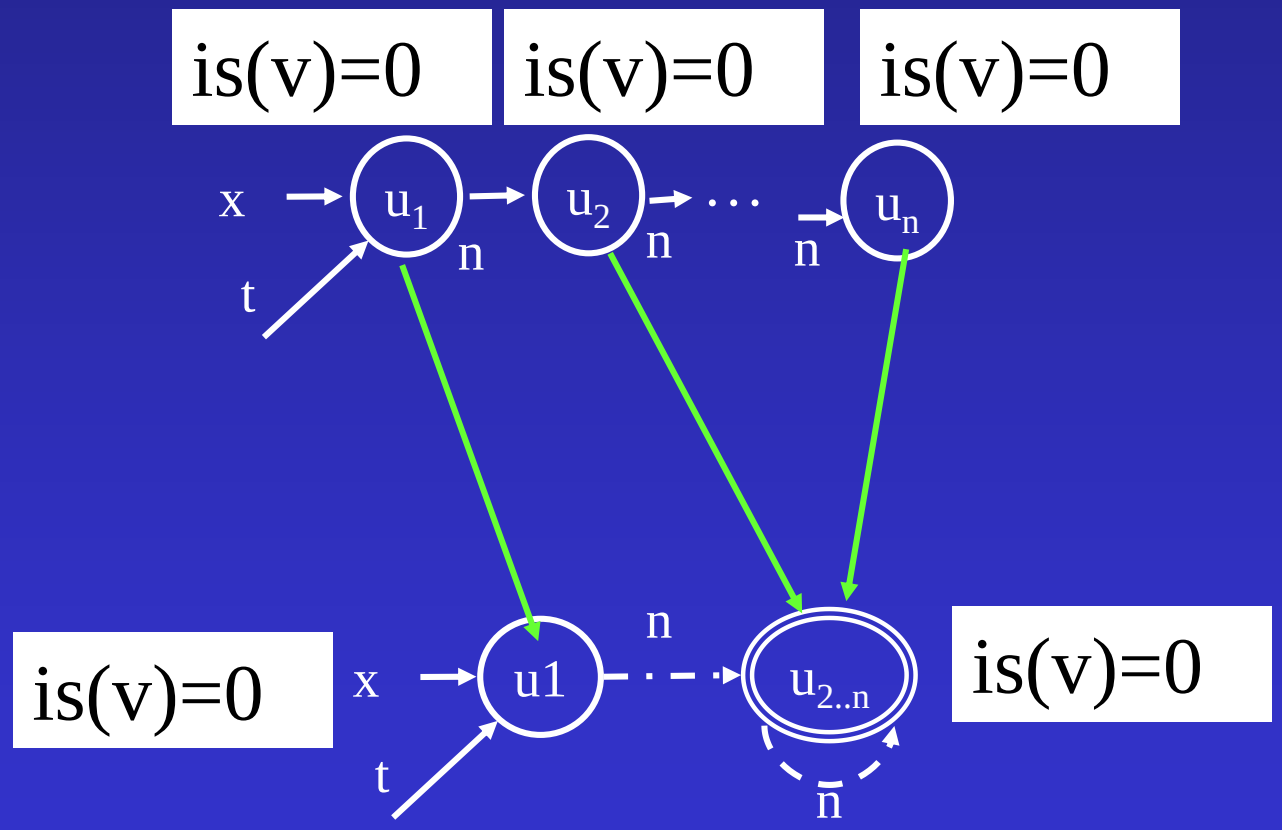
The Instrumentation Principle

- Users define extra derived relations
- Jargon for expressing inductive invariants
- Refines the abstraction
- Refines concretization
- TVLA generates update-formulas

[TOPLAS'10] T.W. Reps, M. Sagiv, A. Loginov:
Finite differencing of logical formulas for static analysis

Heap-Sharing Relation

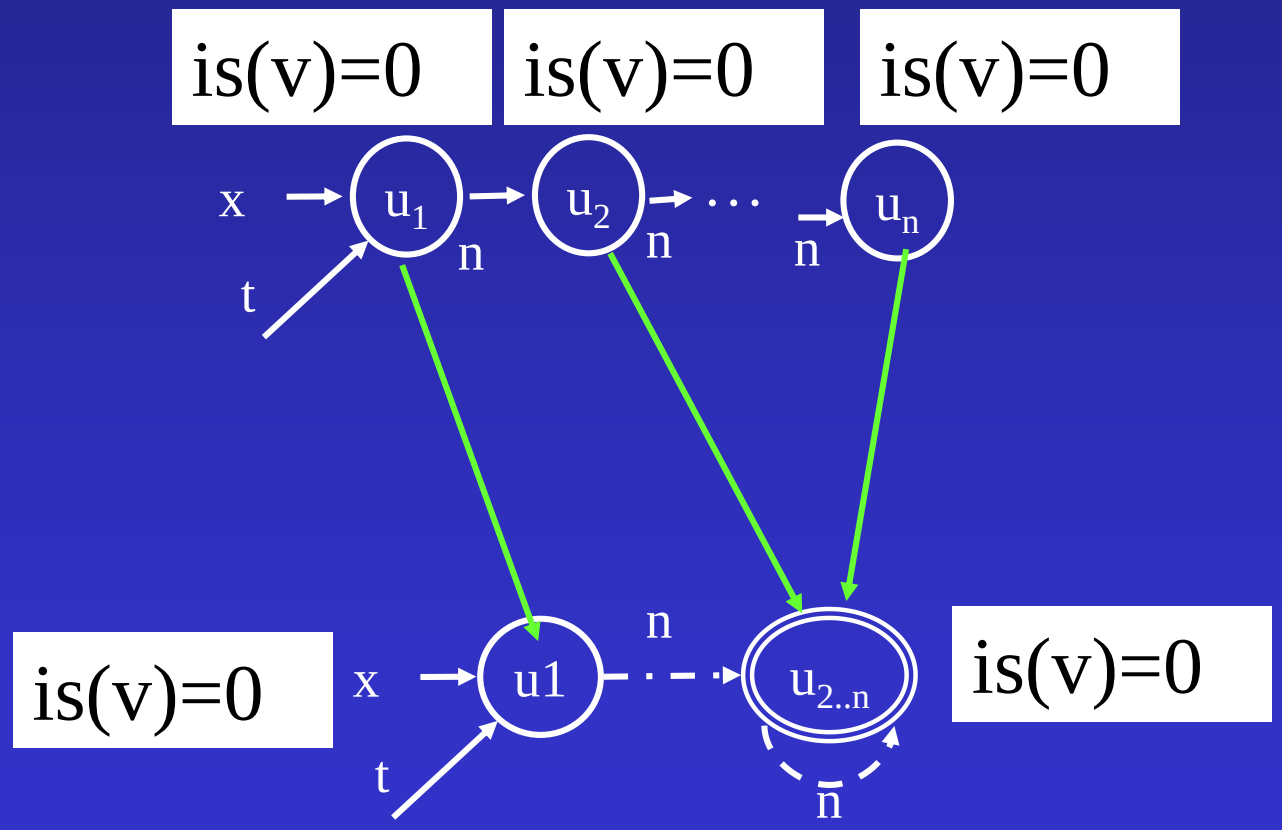
$$is(V) = \exists V_1, V_2 : n(V_1, V) \wedge n(V_2, V) \wedge V_1 \neq V_2$$



$$\forall V: (x = V \wedge t = V \wedge \neg is(V)) \vee (x \neq V \wedge y \neq V \wedge \neg is(V))$$

Heap-Sharing Relation

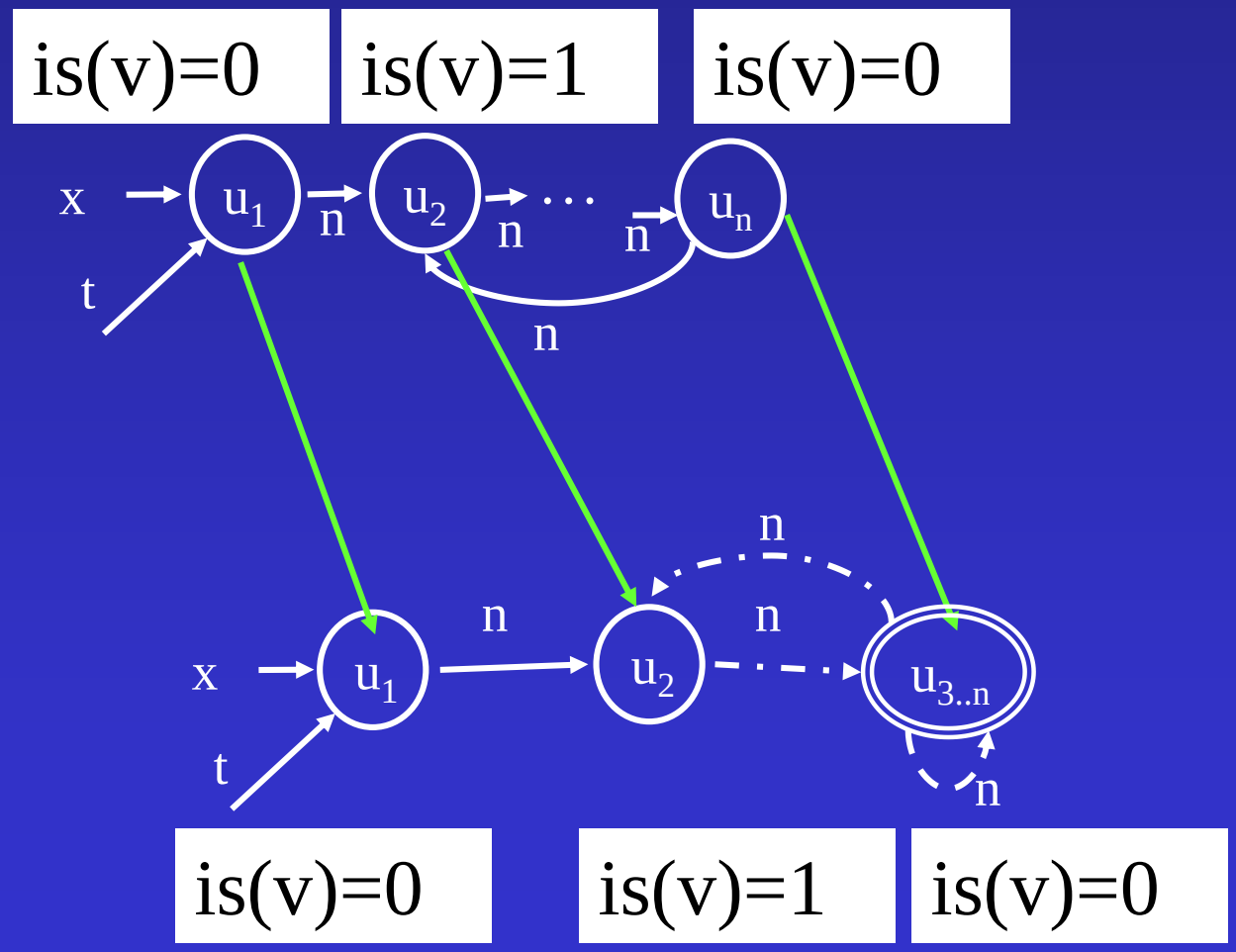
$$is(V) = \exists V_1, V_2 : n(V_1, V) \wedge n(V_2, V) \wedge V_1 \neq V_2$$



$$\forall V: (x = V \wedge t = V \wedge \neg is(V)) \vee (x \neq V \wedge y \neq V \wedge \neg is(V))$$

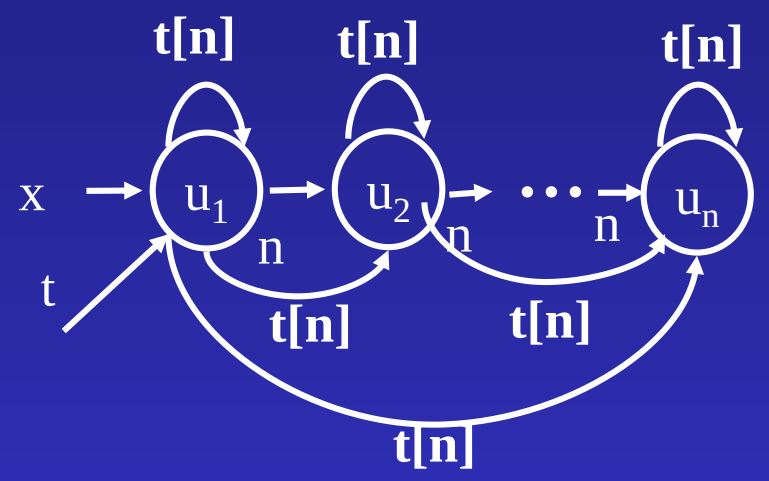
Heap-Sharing Relation

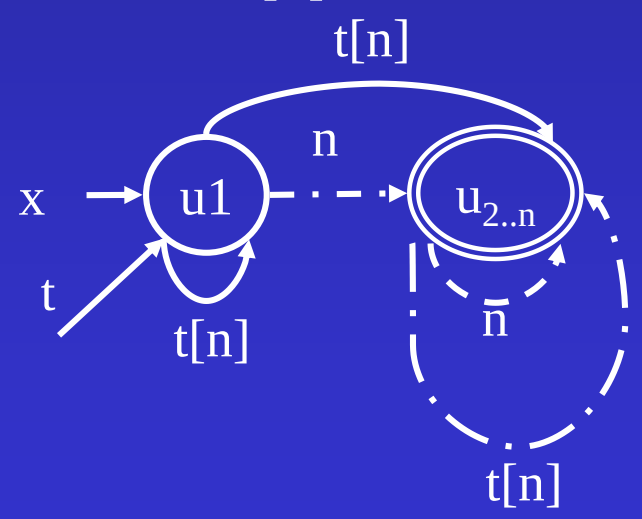
$$is(v) = \exists V_1, V_2 : n(V_1, V) \wedge n(V_2, V) \wedge V_1 \neq V_2$$



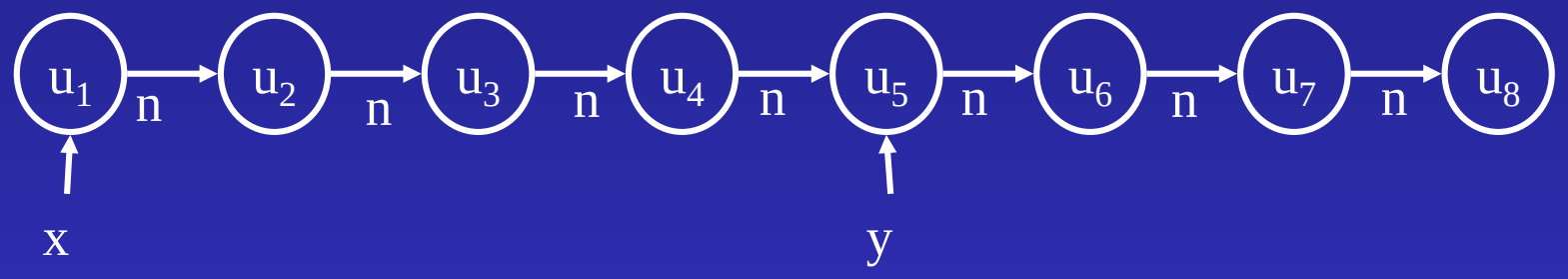
Reachability relation

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



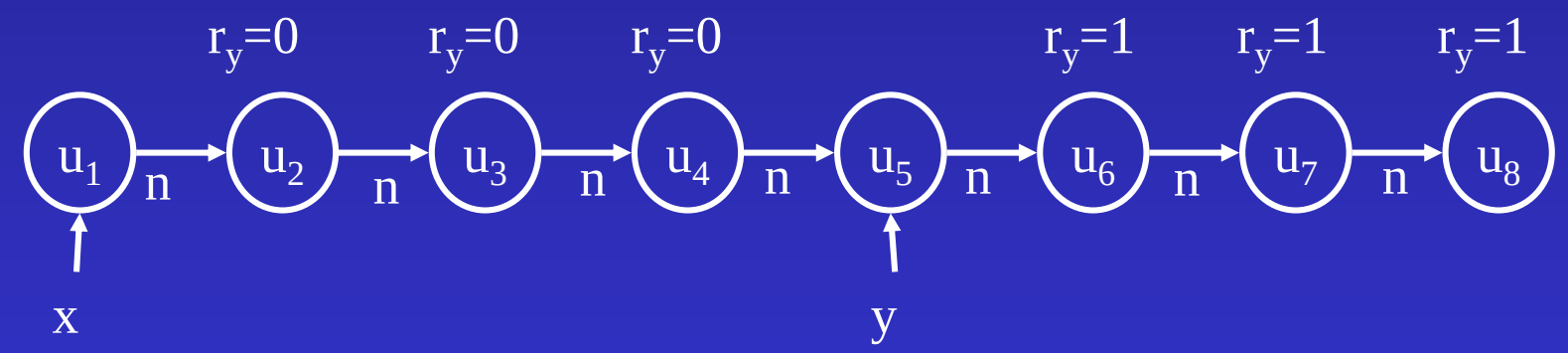


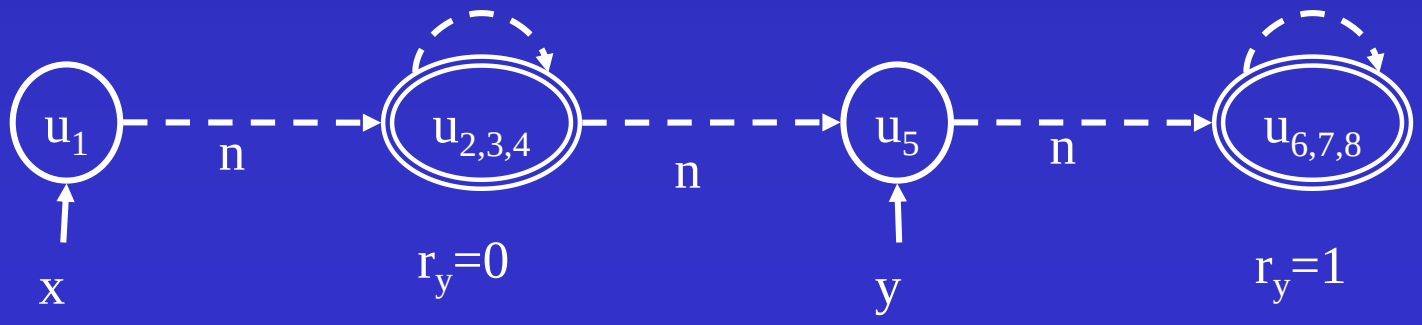
List Segments



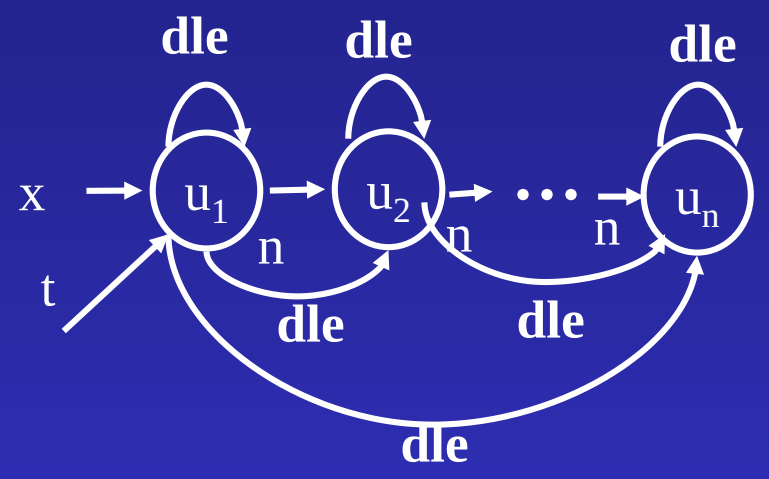
Reachability from a variable

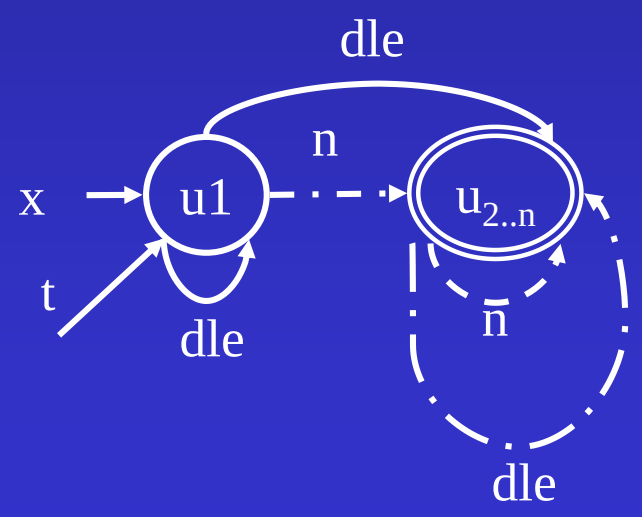
- $r_y(v) = n^*(y, v)$





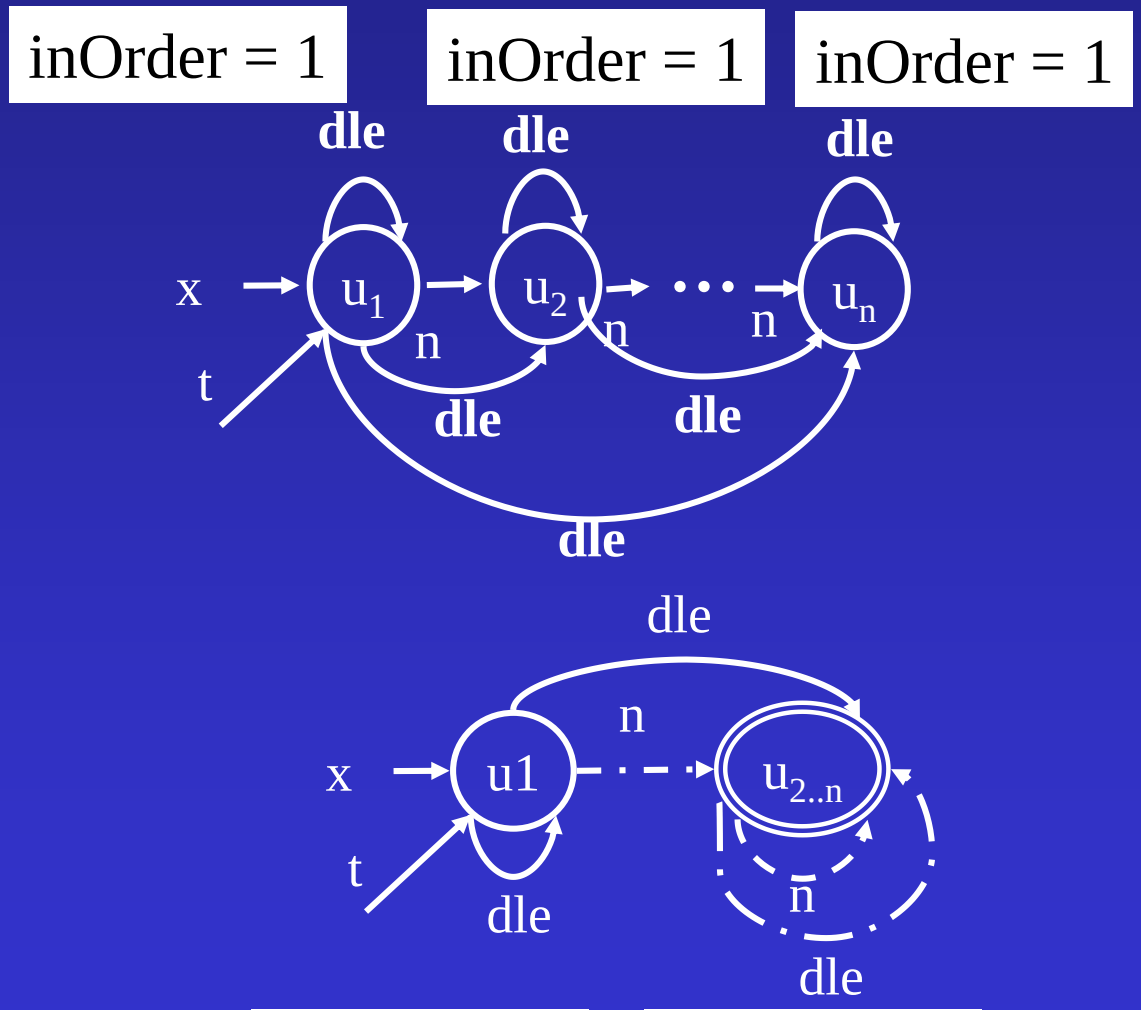
Sortedness





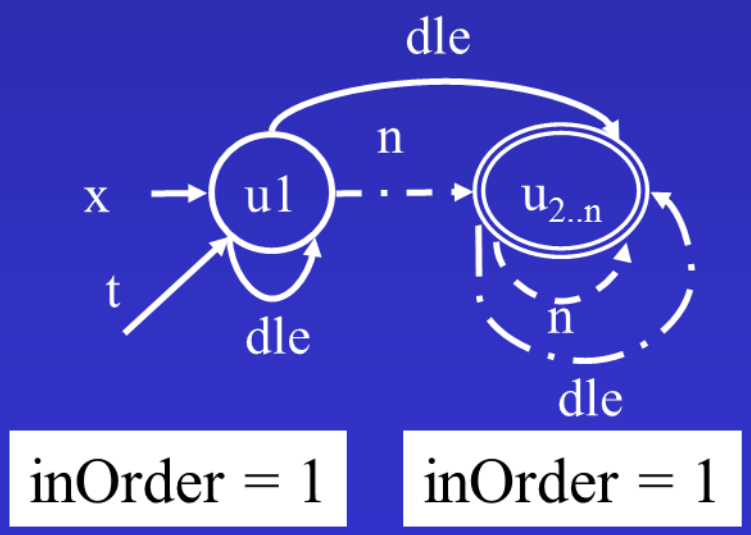
Example: Sortedness

$$\text{inOrder}(v) = \forall v1: n(v,v1) \rightarrow \text{dle}(v, v1)$$



Example: InsertSort

```
typedef struct list_cell {
    int data;
    struct list_cell *n;
} *List;
```

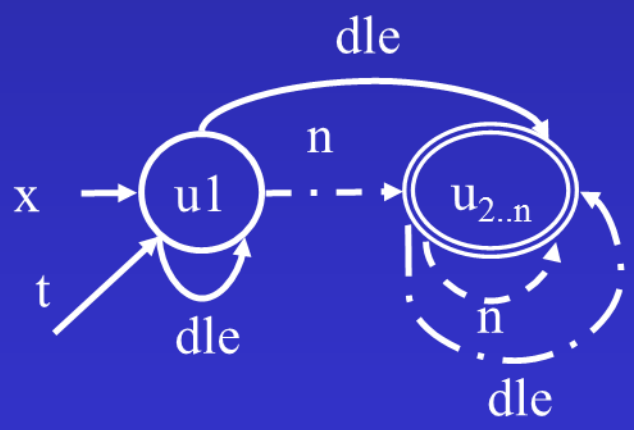



```
List InsertSort(List x) {
List r, pr, rn, l, pl; r = x; pr = NULL;
  while  (r != NULL) {
    l = x; rn = r → n; pl = NULL;
    while  (l != r) {
      if  (l → data > r → data) {
        pr → n = rn; r → n = l;
        if  (pl == NULL) x = r;
        else pl → n = r;
        r = pr;
        break;
      }
      pl = l; l = l → n;
    }
    pr = r; r = rn;
  }
  return x;
}
```

Example: InsertSort

```
typedef struct list_cell {
    int data;
    struct list_cell *n;
} *List;
```

```
List InsertSort(List x) {
 if (x == NULL)    return NULL
 pr = x;  r = x->n;
 while (r != NULL) {
            pl = x; rn = r->n; l = x->n;
            while (l != r) {
                        pr->n = rn ;
                        r->n = l;
                        pl->n = r;
                        r = pr;
                        break;
                        }
                  pl = l;
                  l = l->n;
                  }
            pr = r;
            r = rn;
            }
```



inOrder = 1/2 | inOrder = 1

Mark and Sweep

```
void Mark(Node root) {
  if (root != NULL) {
    pending = ∅
    pending = pending ∪ {root}
    marked = ∅
    while (pending ≠ ∅) {
      x = SelectAndRemove(pending)
      marked = marked ∪ {x}
      t = x → left
      if (t ≠ NULL)
        if (t ∉ marked)
          pending = pending ∪ {t}
      t = x → right
      if (t ≠ NULL)
        if (t ∉ marked)
          pending = pending ∪ {t}
    }
  }
  assert(marked == Reachset(root))
}
```

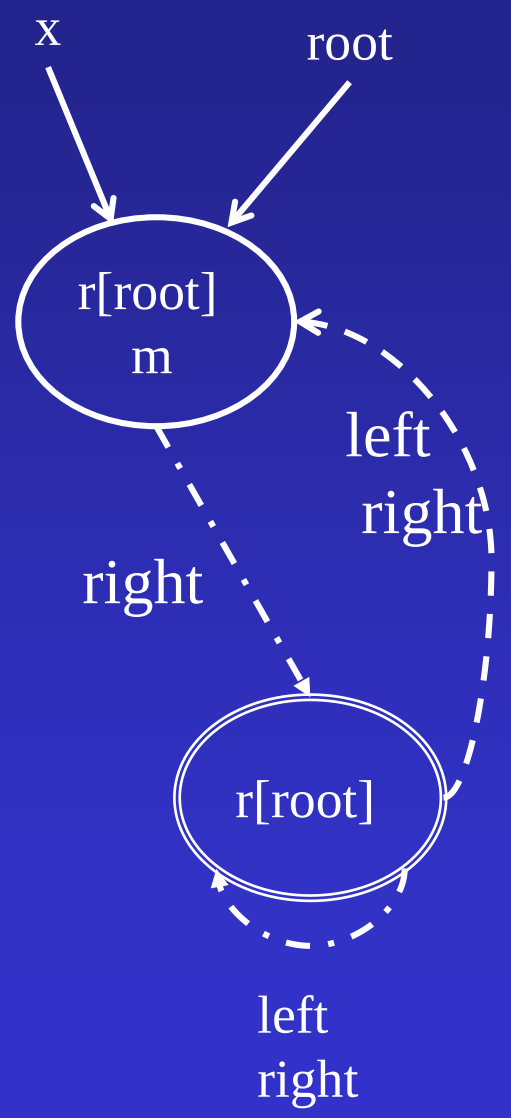
```
void Sweep() {
  unexplored = Universe
  collected = ∅
  while (unexplored ≠ ∅) {
    x = SelectAndRemove(unexplored)
    if (x ∉ marked)
      collected = collected ∪ {x}
  }
  assert(collected ==
          Universe – Reachset(root)
        )
}
```

$\forall v$: marked(v) $\Leftrightarrow$ reach[root](v)

Example: Mark

```
void Mark(Node root) {
  if (root != NULL) {
    pending = ∅
    pending = pending ∪ {root}
    marked = ∅
    while (pending ≠ ∅) {
      x = SelectAndRemove(pending)
      marked = marked ∪ {x}
      t = x → left
      if (t ≠ NULL)
        if (t ∉ marked)
          pending = pending ∪ {t}
/*    t = x → right
 *    if (t ≠ NULL)
 *      if (t ∉ marked)
 *        pending = pending ∪ {t}
 */  }
  }
  assert(marked = = Reachset(root))
}
```

Bug Found



- There may exist an individual that is reachable from the root, but not marked

$$\exists r: root(r) \wedge r[root](r) \wedge \neg p(r) \wedge m(r) \wedge$$
$$\exists e: r[root](e) \wedge \neg m(e) \wedge \neg root(e) \wedge \neg p(e)$$
$$\forall r, e: (root(r) \wedge r[root](r) \wedge \neg p(r) \wedge m(r) \wedge$$
$$r[root](e) \wedge \neg m(e)) \wedge \neg root(e) \wedge \neg p(e))$$
$$\rightarrow \neg left(r,e)$$

Materialization

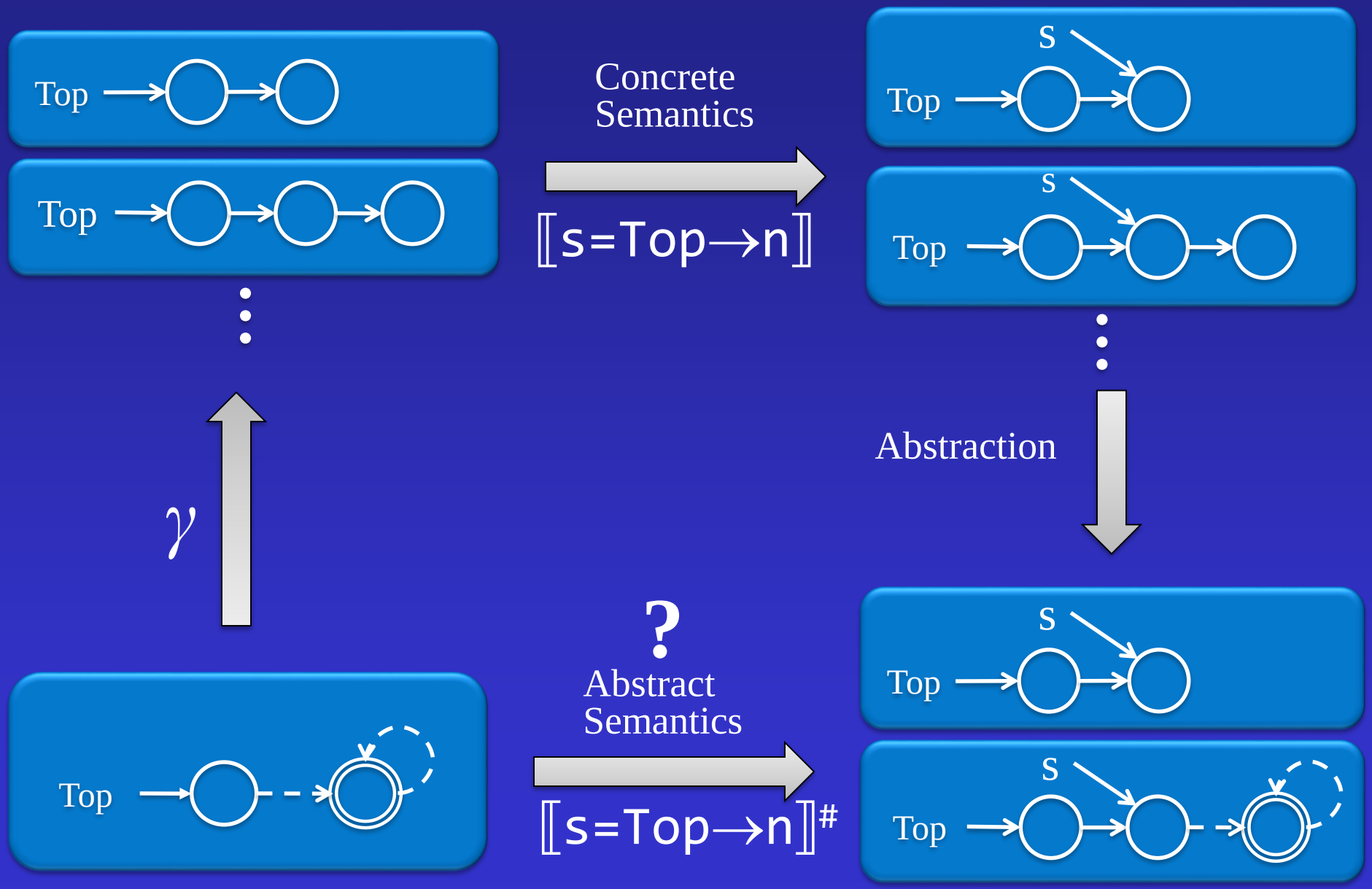
- Materialize elements from summary nodes
- Exploit locality of the program
  - Small changes in the invariant

[PLDI90] D.R. Chase, M.N. Wegman, F. K. Zadeck: Analysis of Pointers and Structures.
[LISP93] E. Wang, P.N. Hilfinger: Analysis of Recursive Types in Lisp-Like Languages

[TOPLAS'98] S. Sagiv, T.W. Reps, R. Wilhelm: *Solving Shape-Analysis Problems in Languages with Destructive Updating*

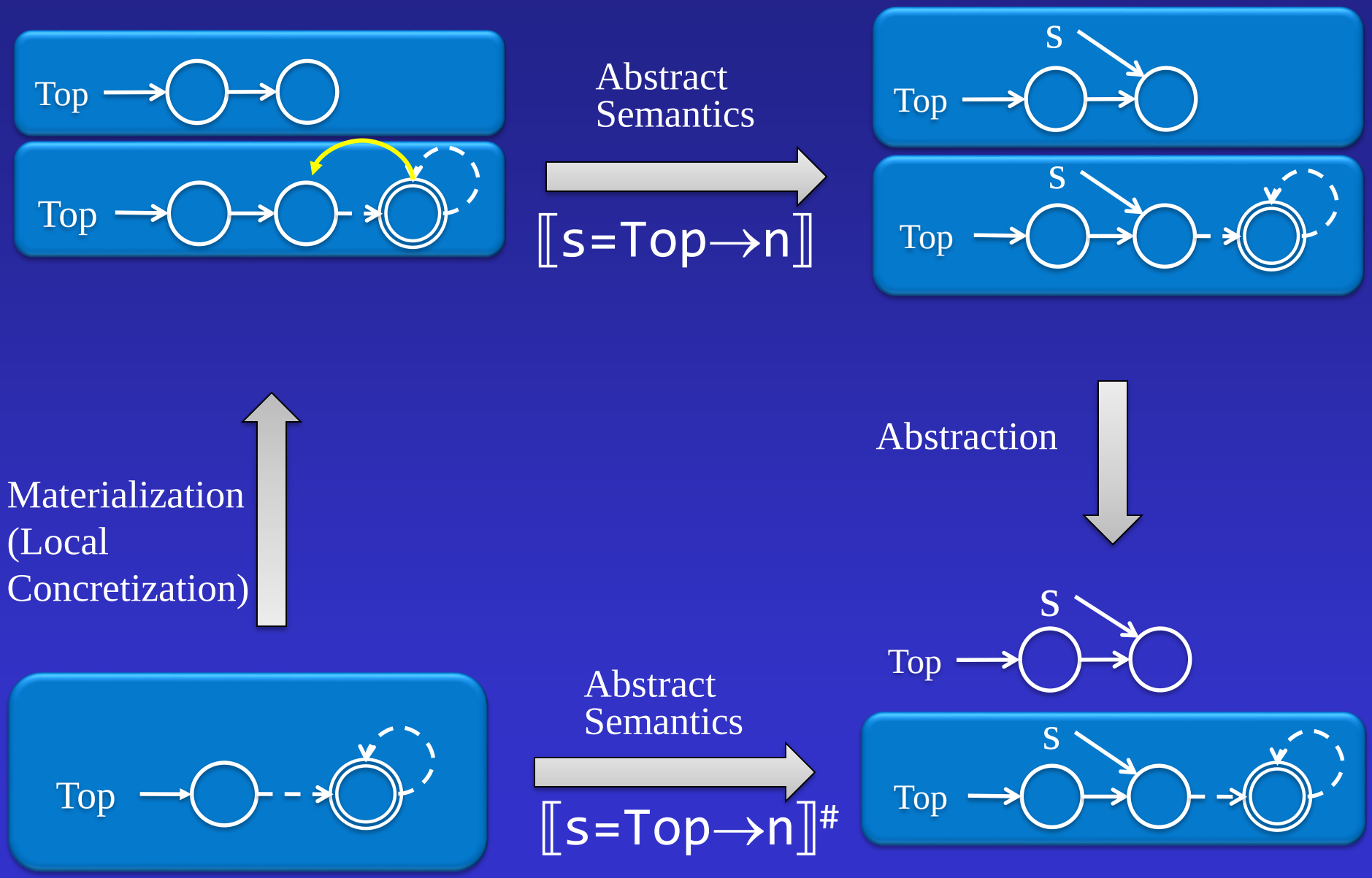
[TACAS'06] D. Distefano, P.W. O'Hearn, H. Yang: A Local Shape Analysis Based on Separation Logic.
[SAS'07] B. Evan Chang, X. Rival, G.C. Necula: Shape Analysis with Structural Invariant Checkers.
[POPL'08] B. Evan Chang, X. Rival: Relational inductive shape analysis.
[CAV'11] J. Berdine, B. Cook, S. Ishtiaq: SLAyer: Memory Safety for Systems-Level Code

Best Transformer [CC79]



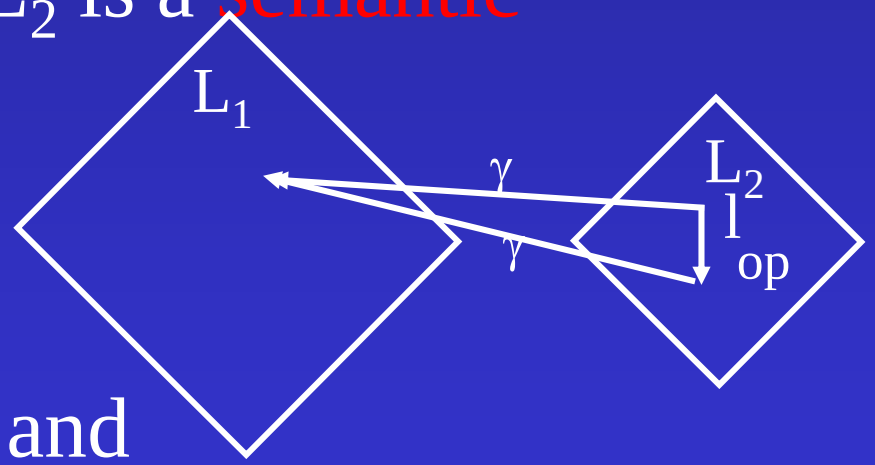
"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

Local Concretization Based Transformer

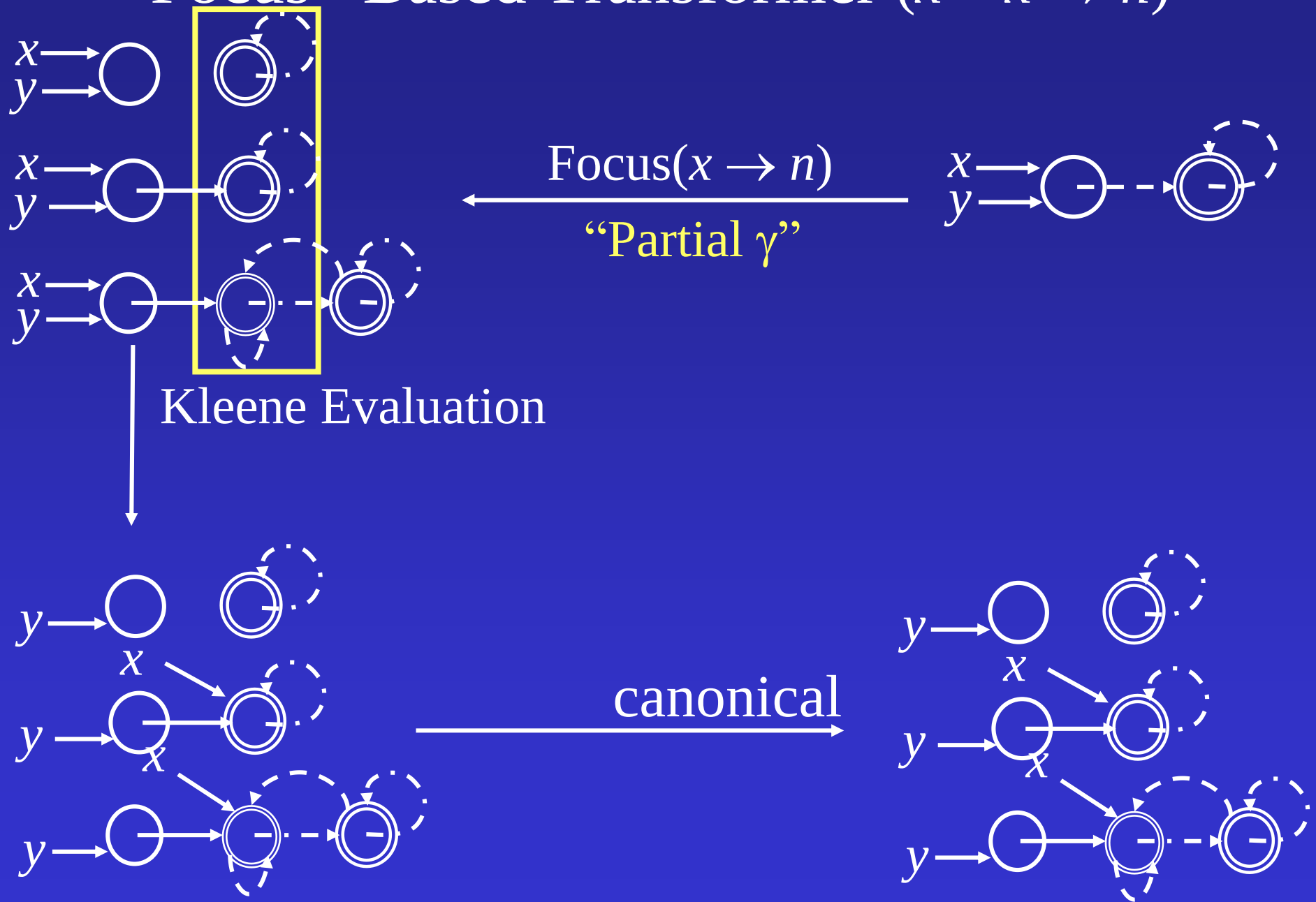


Semantic Reduction

- Improve the precision of the analysis by recovering properties of the program semantics
- A Galois connection $(L_1, \alpha, \gamma, L_2)$
- An operation $op: L_2 \rightarrow L_2$ is a semantic reduction
  - $\forall l \in L_2\ op(l) \sqsubseteq l$
  - $\gamma(op(l)) = \gamma(l)$
- Can be applied before and after basic operations



“Focus”-Based Transformer ($x = x \rightarrow n$)

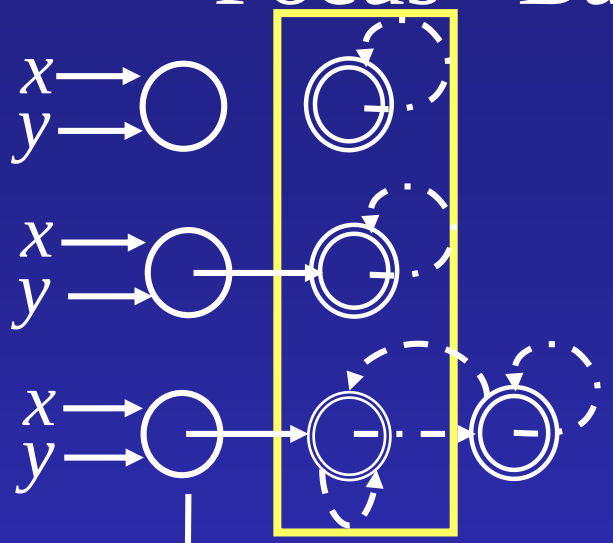


The Focus Operation

- Focus: Formula$\rightarrow$(P(3-Struct) $\hookrightarrow$P(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

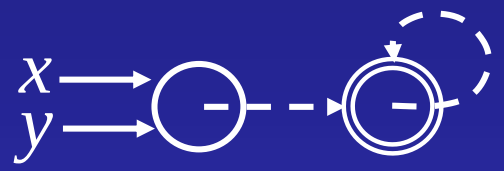
“Focus”-Based Transformer ($x = x \rightarrow n$)

$\exists w: x(w) \wedge n(w, v)$

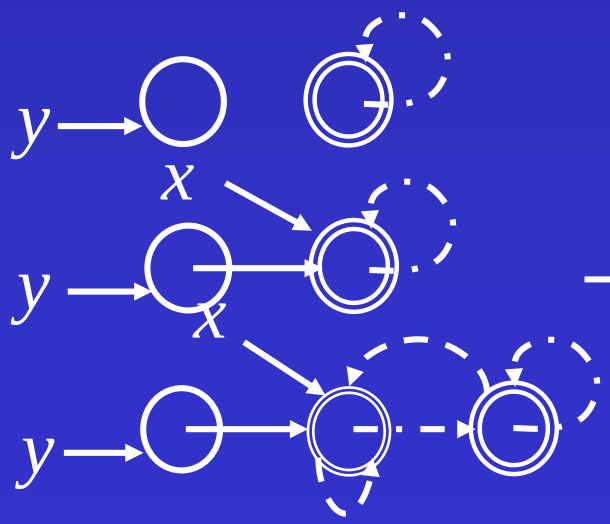


Focus($x \rightarrow n$)

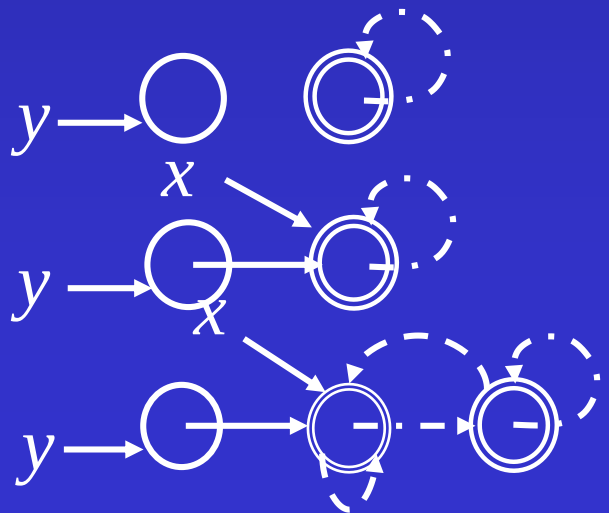
“Partial γ”



Kleene Evaluation



canonical

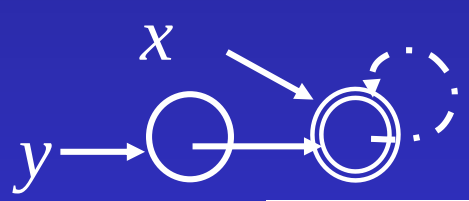


The Coercion Principle

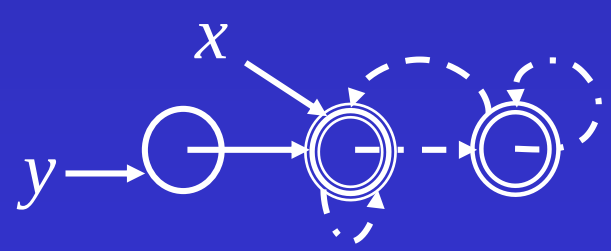
- Another Semantic Reduction
- Can be applied after Focus or after Update or both
- Increase precision by exploiting some structural properties possessed by all stores (Global invariants)
- Structural properties captured by constraints
- Apply a constraint solver

Apply Constraint Solver

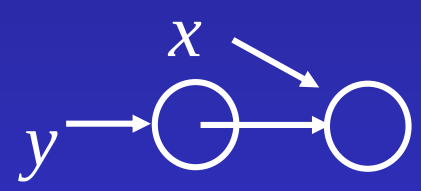
r[n,y](v)=1

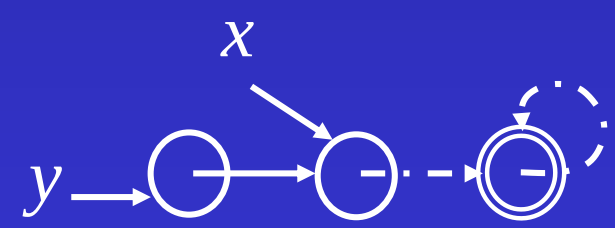


is(v)=0



is(v)=0





Sources of Constraints

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

Example Constraints

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

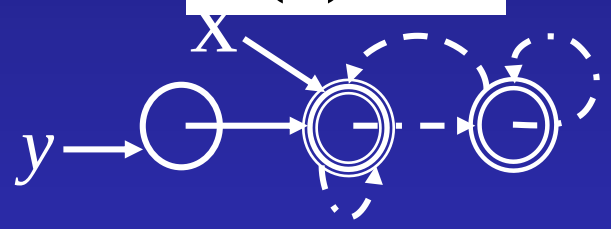
$$n(v, v1) \wedge n(v,v2) \rightarrow eq(v1, v2)$$

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

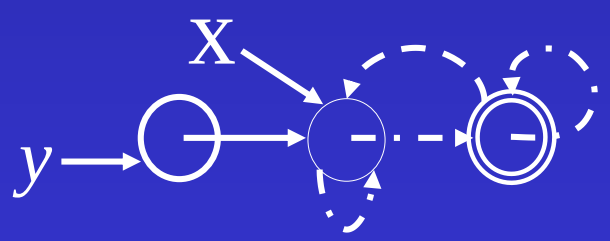
$$n^*(v3, v4) \leftrightarrow t[n](v1, v2)$$

Apply Constraint Solver

is(v)=0



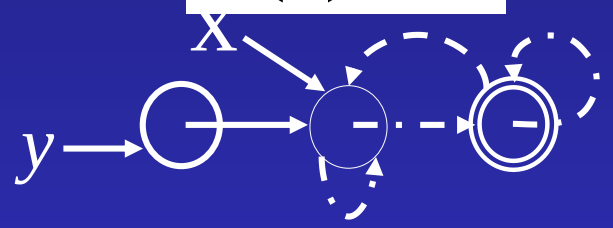
$x(v1) \wedge x(v2) \rightarrow eq(v1, v2)$



is(v)=0

Apply Constraint Solver

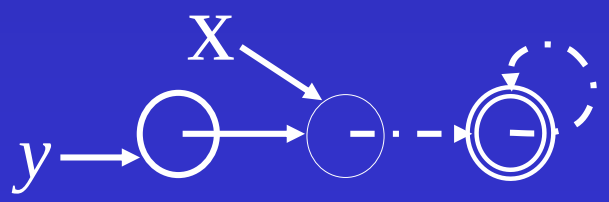
is(v)=0



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

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

is(v)=0

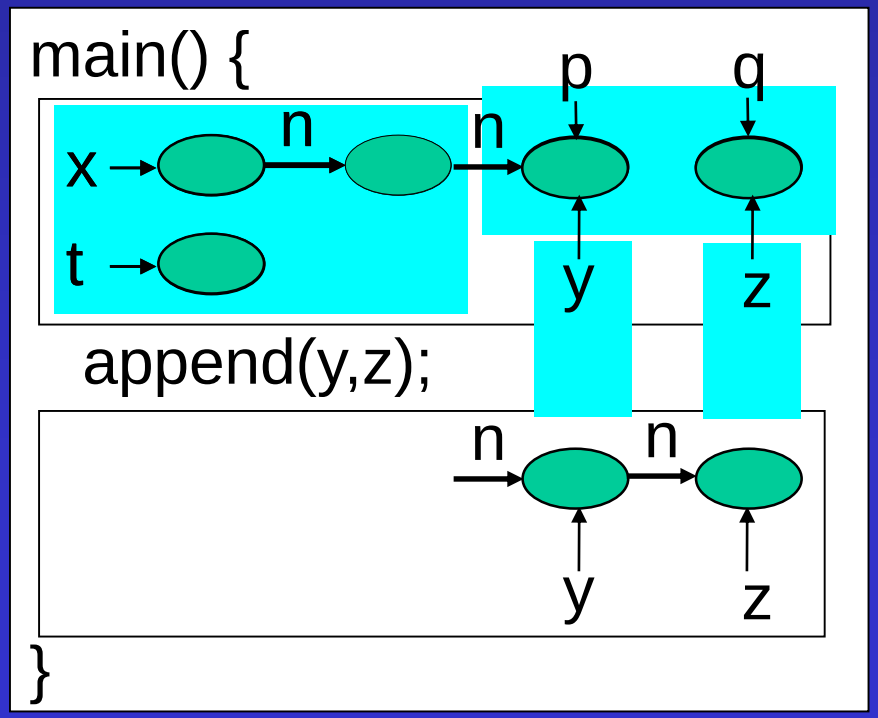


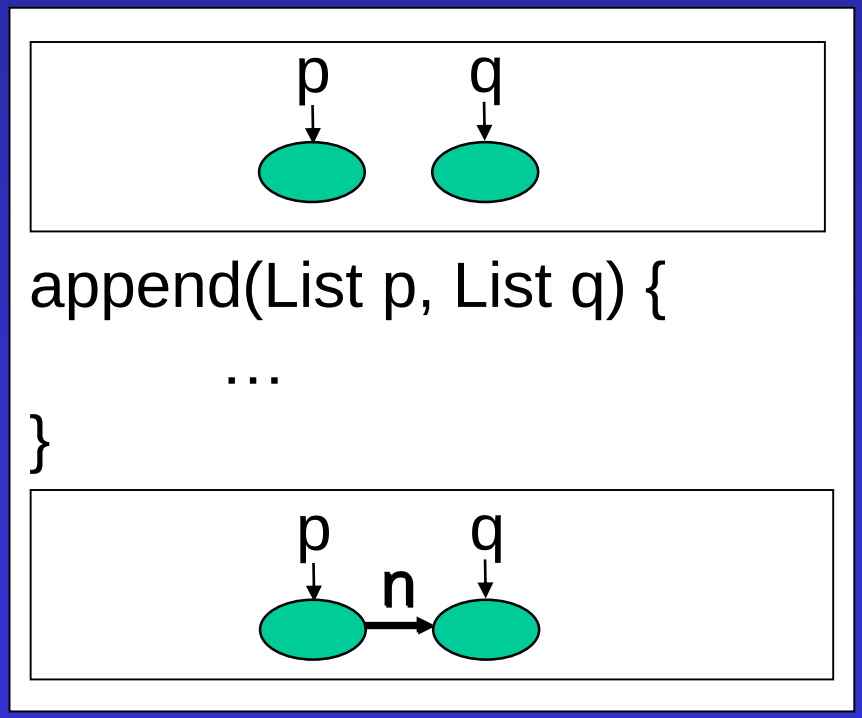
Summary Transformers

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

How to tabulate procedures? N. Rinetzky

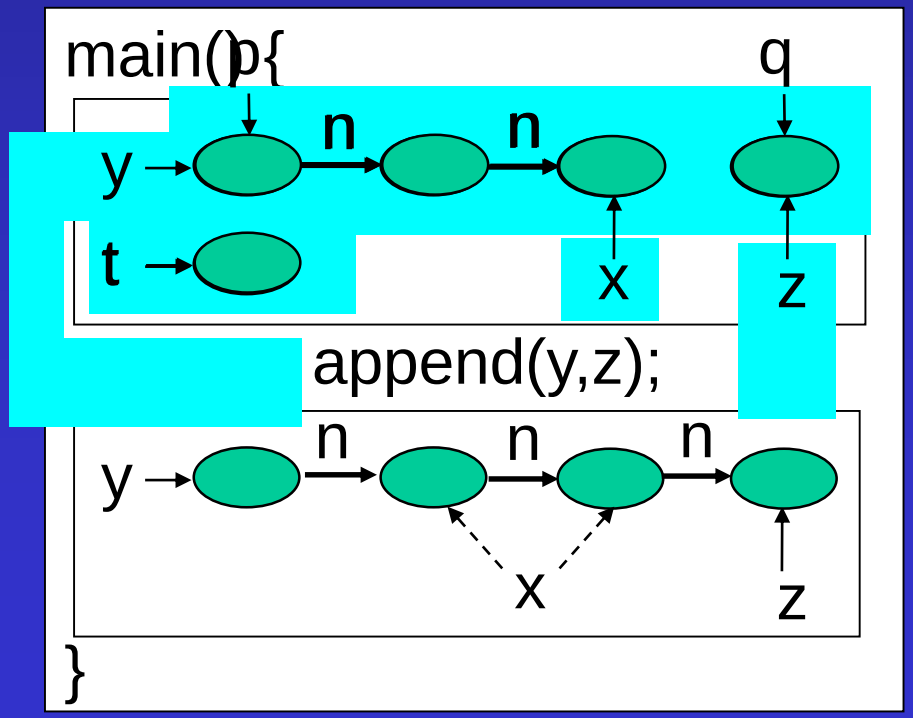
- Procedure ≡ input/output relation
  - Not reachable ➔ Not affected
  - proc: local (≡reachable) heap ➔ local heap

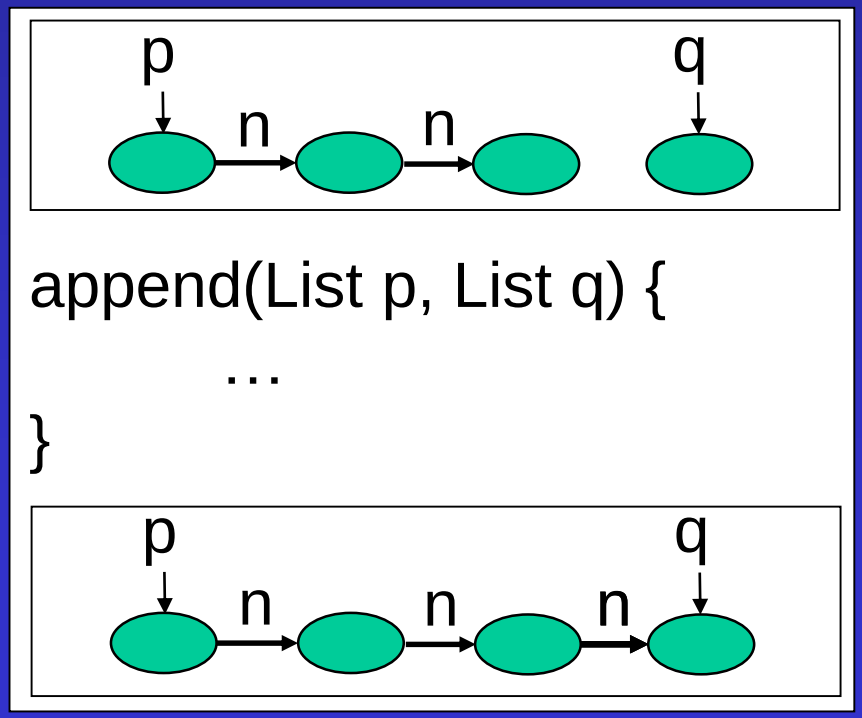




How to handle sharing?

- External sharing may break the functional view

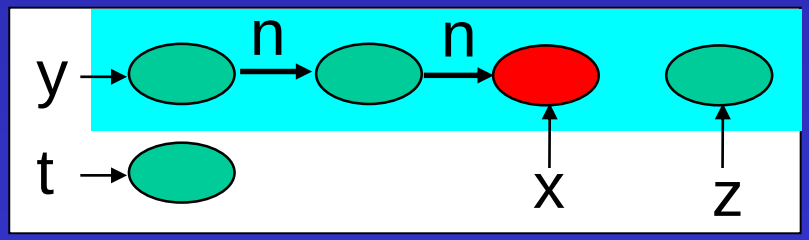




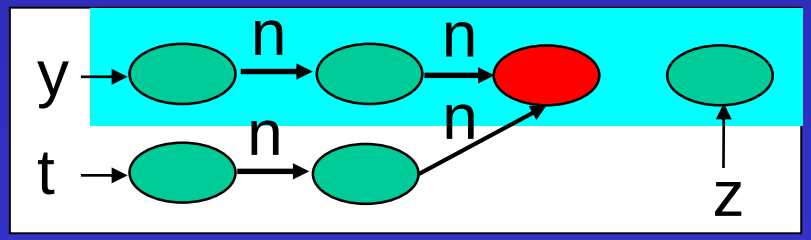
Cutpoints

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

append(y,z)



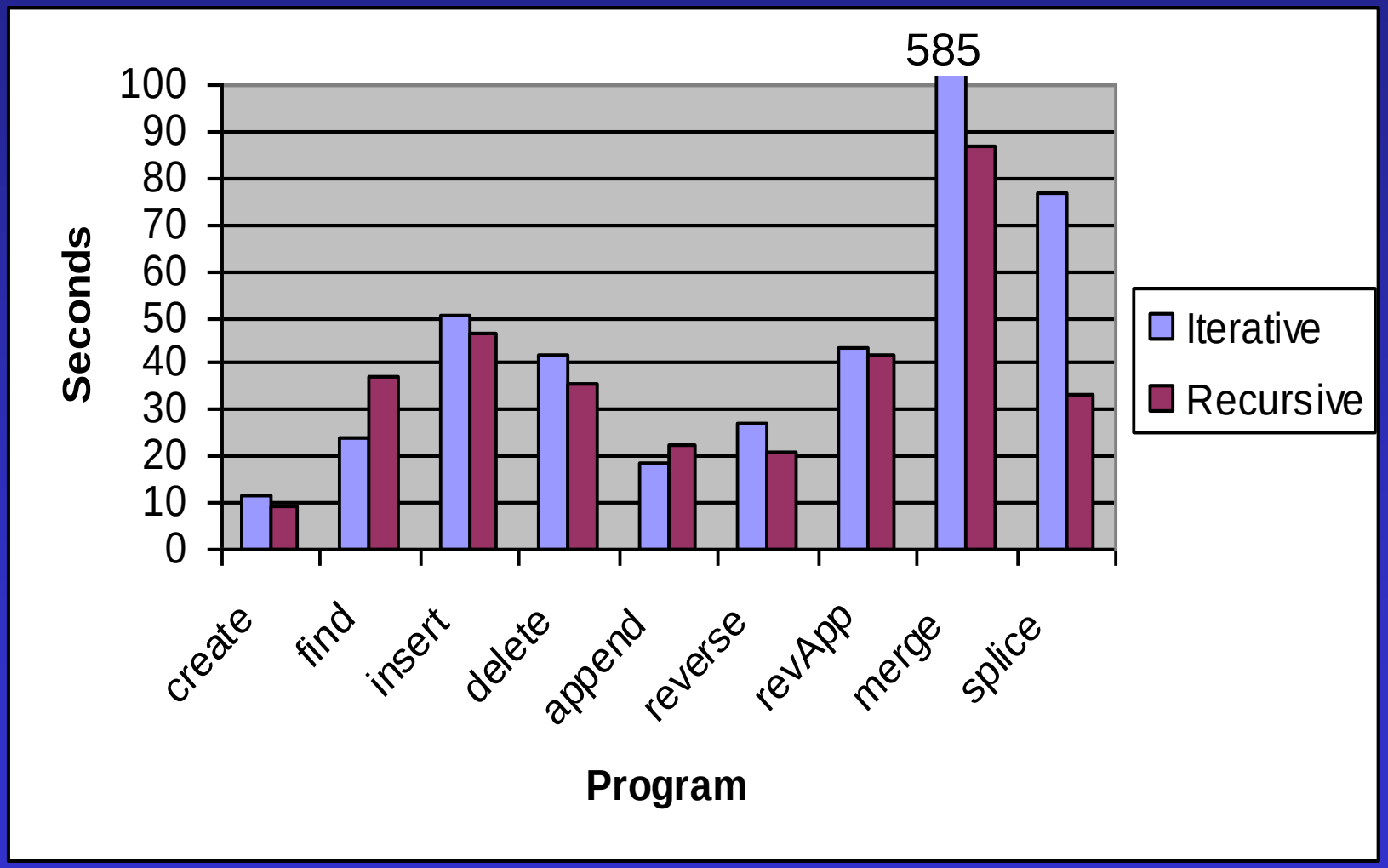
append(y,z)



Cutpoints

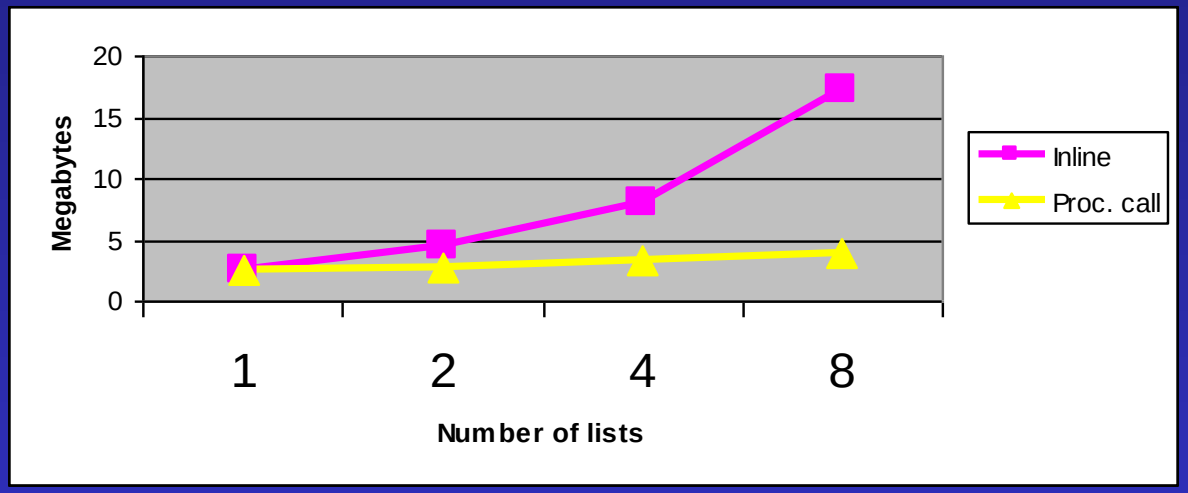
[POPL’05] N. Rinetzky, J. Bauer, T. W. Reps, S.l Sagiv, R. Wilhelm A semantics for procedure local heaps and its abstractions
[SAS’05] N. Rinetzky, M. Sagiv, E. Yahav: Interprocedural Shape Analysis for Cutpoint-Free Programs
[SAS’06] A. Gotsman, J. Berdine, B. Cook: Interprocedural Shape Analysis with Separated Heap Abstractions
[CAV’08] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn: Scalable Shape Analysis for Systems Code.
[CAV’11] J. Berdine, B. Cook, S. Ishtiaq: SLAyer: Memory Safety for Systems-Level Code.
[LMCS11,TACAS09] M. Faouzi Atig, A. Bouajjani, S.z Qadeer: Context-Bounded Analysis For Concurrent Programs With Dynamic Creation of Threads
[ATV’13] P. Aziz Abdulla, L. Holík, B. Jonsson, O. Lengál, C. Quy Trinh, T. Vojnar: Verification of Heap Manipulating Programs with Ordered Data by Extended Forest Automata

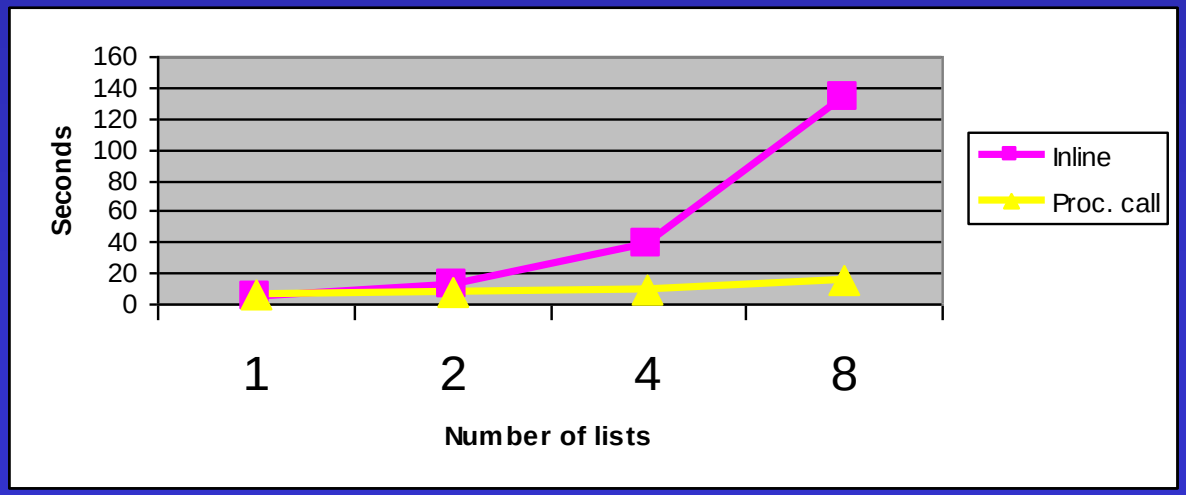
Iterative vs. Recursive (SLL)



Inline vs. Procedural abstraction

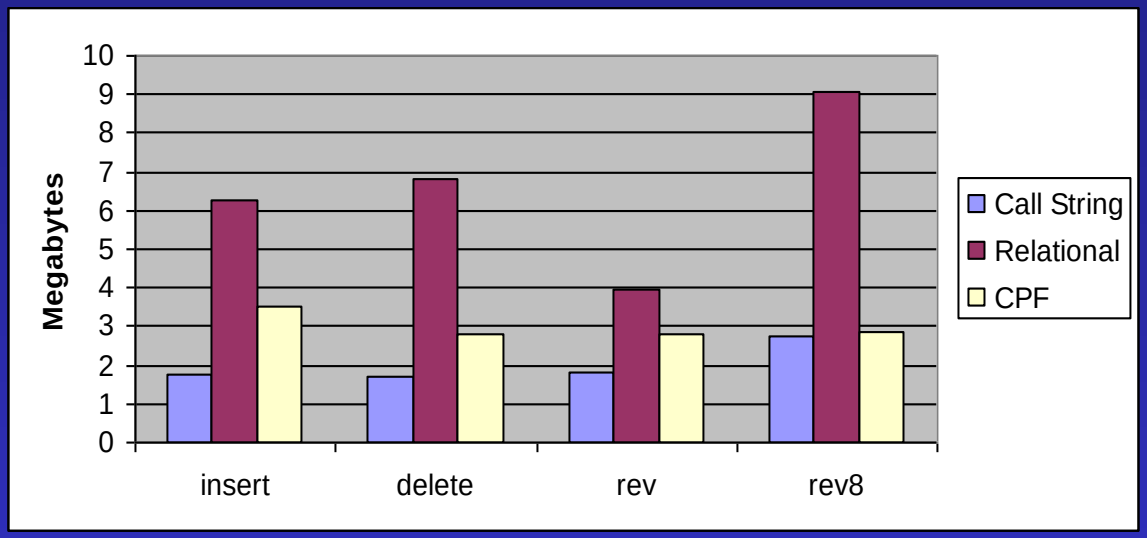
```
// Allocates a list of
// length 3
List create3(){
   …
}

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

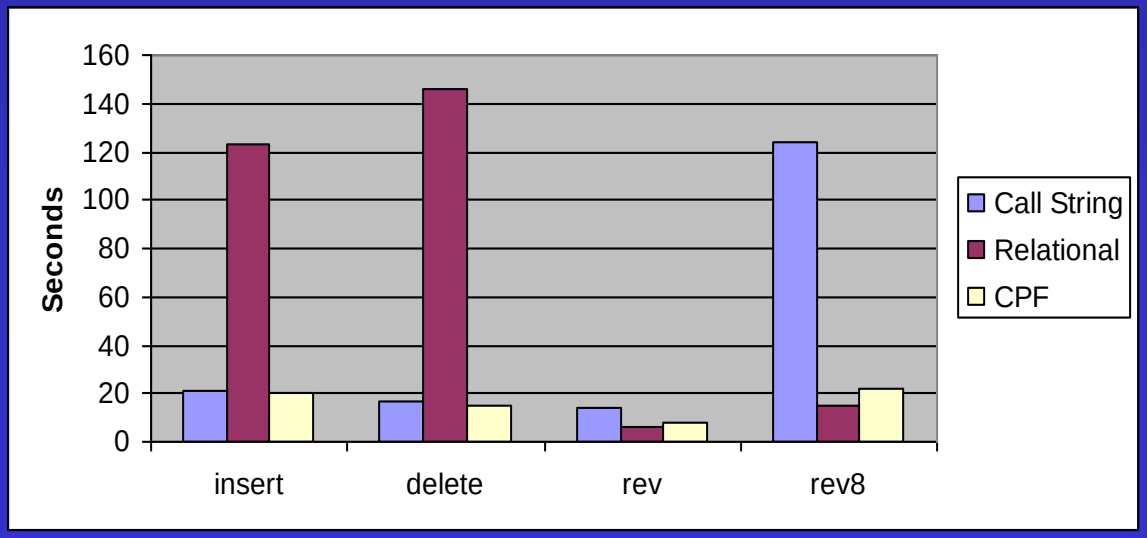




Call string vs. Relational vs. CPF

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

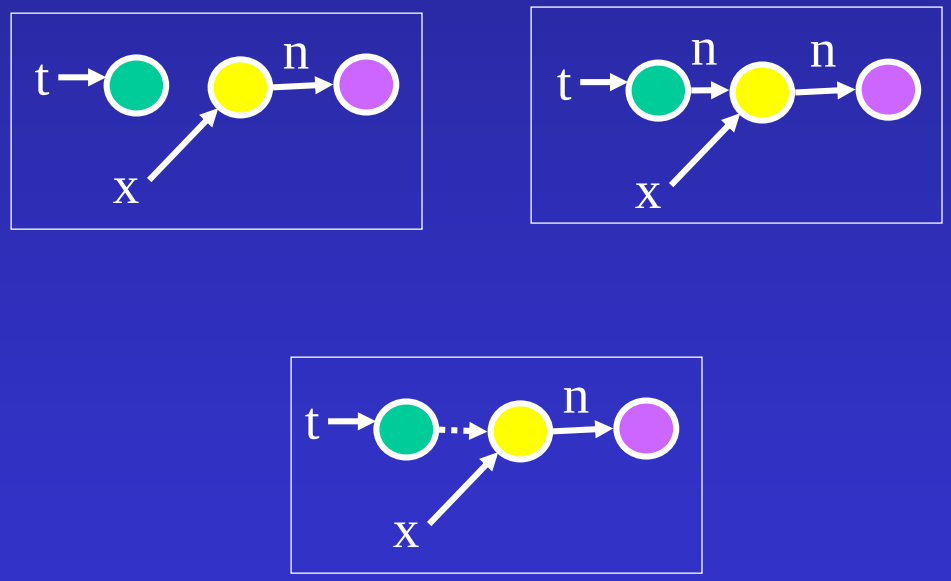




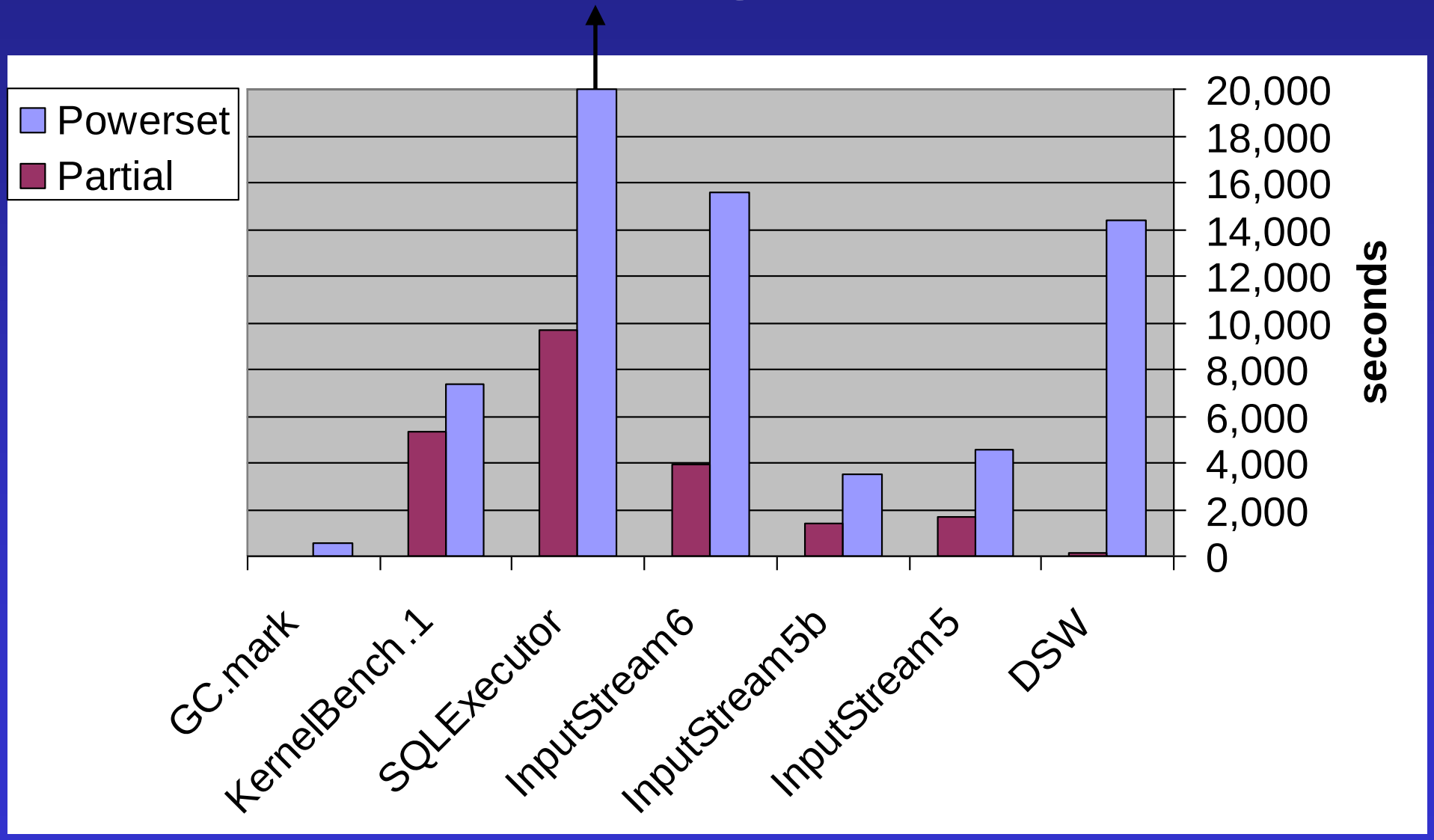
Partially Disjunctive Heap Abstraction (Manevich, SAS'04)

- Use a heap-similarity criterion
  - We defined similarity by universe congruence
- Merge similar heaps
- Avoid merging dissimilar heaps
- The same concrete state can belong to more than one abstract value

Partially Disjunctive Abstraction



Running times



Compositionality

- Apply shape analysis to one procedure at a time
- Calculate the effect of the procedure on the reachable heap
- Propagate the effect into the clients
- Useful in incremental settings
- Adopted by Facebook

[JACM'11] C. Calcagno, Dino Distefano, Peter W. O'Hearn, Hongseok Yang: Compositional Shape Analysis by Means of Bi-Abduction.
[TOPLAS'11] Bhargav S. Gulavani, Supratik Chakraborty, G. Ramalingam, Aditya V. Nori: Bottom-up shape analysis using LISF
[PLDI'11] Isil Dillig, Thomas Dillig, Alex Aiken, Mooly Sagiv: Precise and compact modular procedure summaries for heap manipulating programs.

Shape Analysis Principles

- Reason about directed reachability
- Specialized data structures
- Explore locality of updates
  - Materialization

- Explore locality of procedure and type safety for updating reachability [POPL'05]
- Bottom-up shape analysis [POPL'09, JACM'11]
- Partially disjunctive analysis [SAS'04, CAV'07]

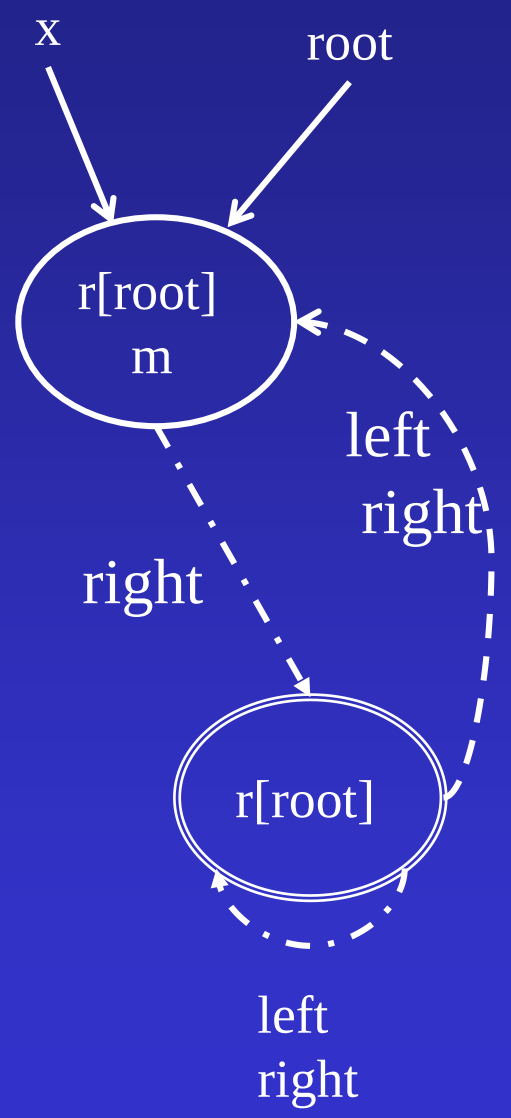
[POPL'05] N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, R. Wilhelm: A semantics for procedure local heaps and its abstractions

[SAS'04] R. Manevich, S. Sagiv, G. Ramalingam, J. Field: Partially Disjunctive Heap Abstraction

[CAV'07] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, H. Yang: Shape Analysis for Composite Data Structures

[POPL'09, JACM'11] C. Calcagno, Dino Distefano, Peter W. O'Hearn, Hongseok Yang: Compositional Shape Analysis by Means of Bi-Abduction

Bug Found



- There may exist an individual that is reachable from the root, but not marked

$$r[root](root) \wedge \neg p(root) \wedge m(root) \wedge$$
$$\exists e: r[root](e) \wedge \neg m(e) \wedge \neg root(e) \wedge \neg p(e)$$
$$\forall r, e: (root(r) \wedge r[root](r) \wedge \neg p(r) \wedge m(r) \wedge$$
$$r[root](e) \wedge \neg m(e)) \wedge \neg root(e) \wedge \neg p(e))$$
$$\rightarrow \neg left(r,e)$$

Properties Proved

Program	Properties	#Graphs	Seconds
LindstromScan	CL, DI	1285	8.2
LindstromScan	CL, DI, IS, TE	183564	2185
SetRemove	CL, DI, SO	13180	106
SetInsert	CL, DI, SO	299	1.75
DeleteSortedTree	CL, DI	2429	6.24
DeleteSortedTree	CL, DI, SO	30754	104
InsertSortedTree	CL, DI	177	0.85
InsertSortedTree	CL, DI, SO	1103	2.5
InsertAVLttree	CL, DI, SO	1855	27.4
RecQuickSot	CL, DI, SO	5585	9.2

CL=memory safety DI=data structure invariant TE=termination SO=sorted

Applying Shape Analysis to Real Code

Lukás Holík, Ondrej Lengál, Adam Rogalewicz, Jirí Simácek, Tomás Vojnar: Fully Automated Shape Analysis Based on Forest Automata. CAV 2013: 740-755

Josh Berdine, Byron Cook, Samin Ishtiaq: SLAyer: Memory Safety for Systems-Level Code. CAV 2011: 178-183

Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn: Scalable Shape Analysis for Systems Code. CAV 2008: 385-398

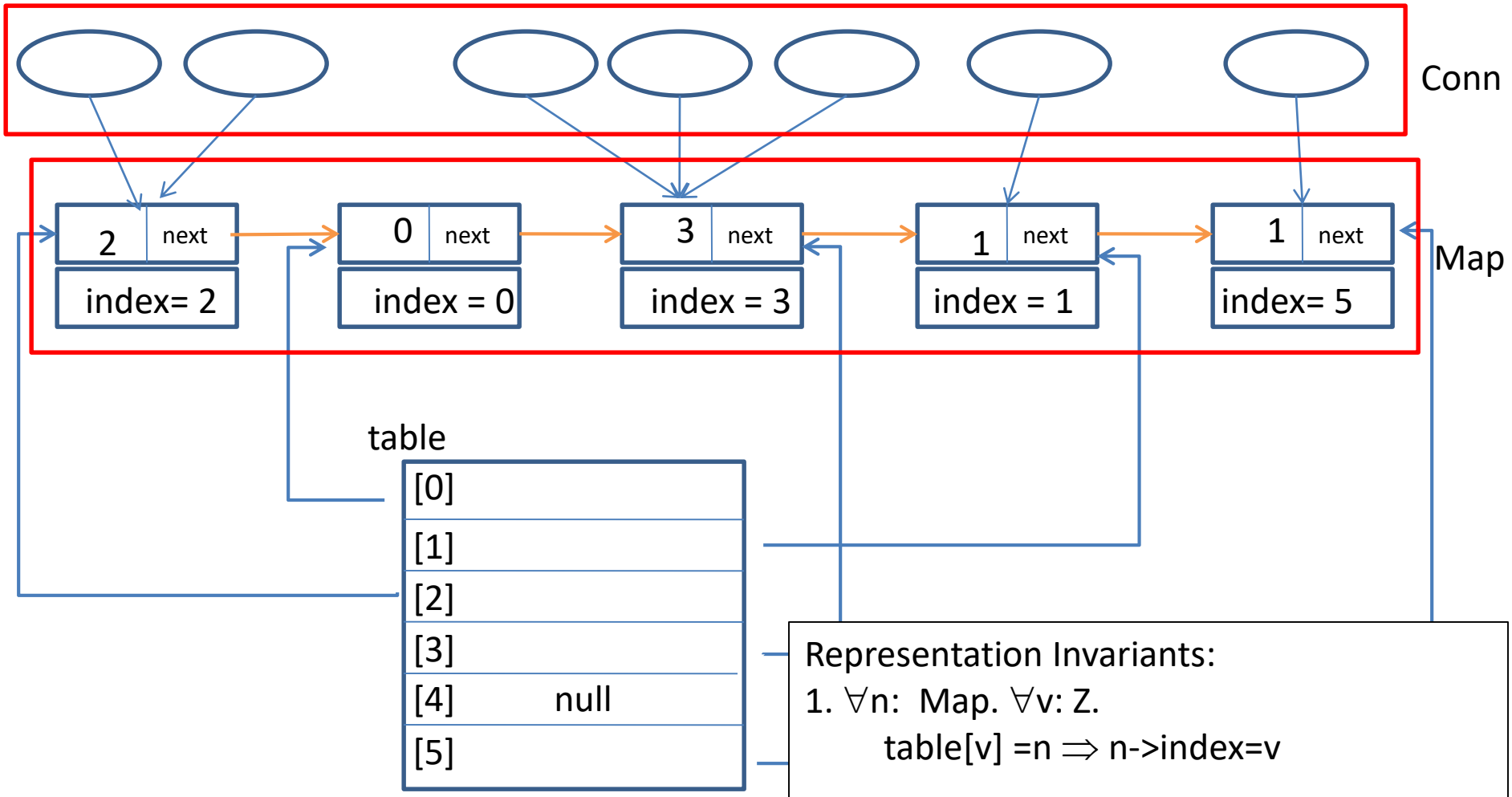
Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, Mangala Gowri Nanda: Verifying dereference safety via expanding-scope analysis. ISSTA 2008: 213-224

Eran Yahav, G. Ramalingam: Verifying safety properties using separation and heterogeneous abstractions. PLDI 2004: 25-3

Limitations of Shape Analysis

- Complex data structures
- Concurrency
- Modularity & libraries
- False alarms

thttpd: Web Server



Representation Invariants:

1. $\forall$n: Map. $\forall$v: Z.

   table[v] =n $\Rightarrow$ n->index=v

2. $\forall$n: Map.

   n->rc =|{n' : Conn . n'->file_data = n}|

thttptd:mmc.c

```
static void add_map(Map *m)
{
    int i = hash(m);
    …
    table[i] = m ;
    …
    m->index= i ;
    …
    m->rc++;

}
```

broken

restored

Representation Invariants:

1. $\forall$n: Map. $\forall$v:Z.
   table[v] =n $\Rightarrow$ index[n]=v

2. $\forall$ n:Map.
   rc[n] =|{n' : Conn . file_data[n’] = n}|