

Compile-Time Verification of Properties of Heap Intensive Programs

Mooly Sagiv

Thomas Reps

Reinhard Wilhelm

<http://www.cs.tau.ac.il/~TVLA>

<http://www.cs.tau.ac.il/~msagiv/toplas02.pdf>

. . . and also

- University of Wisconsin
 - F. DiMaio
 - D. Gopan
 - A. Loginov
- IBM Research
 - J. Field
 - H. Kolodner
 - M. Rodeh
- Microsoft Research
 - G. Ramalingam
- University of Massachusetts
 - N. Immerman
 - B. Hesse
- The Technical University of Denmark
 - H.R. Nielson
 - F. Nielson
- Weizmann Institute/NYU
 - A. Pnueli
- Inria
 - B. Jeannet
- Tel-Aviv University
 - G. Arnold
 - I. Bogudlov
 - G. Erez
 - N. Dor
 - T. Lev-Ami
 - R. Manevich
 - R. Shaham
 - A. Rabinovich
 - N. Rinetzky
 - E. Yahav
 - G. Yorsh
 - A. Warshavsky
- Universität des Saarlandes
 - Jörg Bauer
 - Ronald Biber

Shape Analysis

- Determine the possible shapes of a dynamically allocated data structure at given program point
- Relevant questions:
 - Does `x.next` point to a shared element?
 - Does a variable point `p` to an allocated element every time `p` is dereferenced
 - Does a variable point to an acyclic list?
 - Does a variable point to a doubly-linked list?
 - ?
 - Can a procedure create a memory-leak

Problem

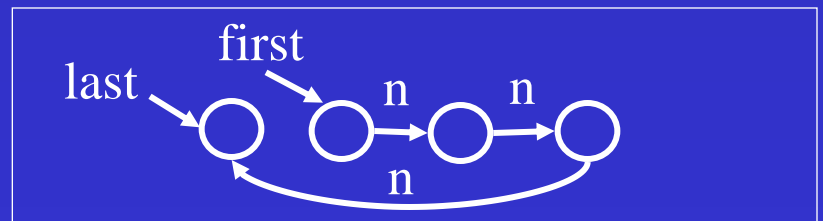
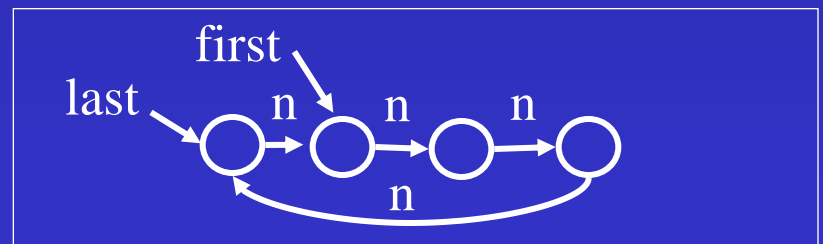
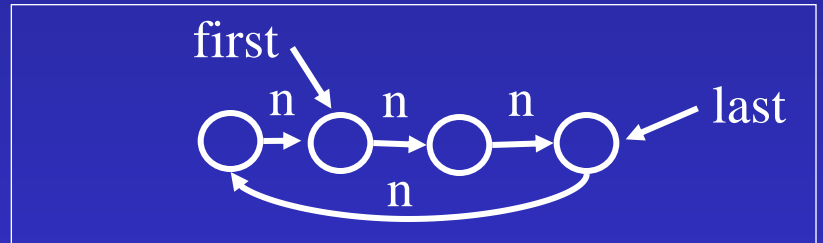
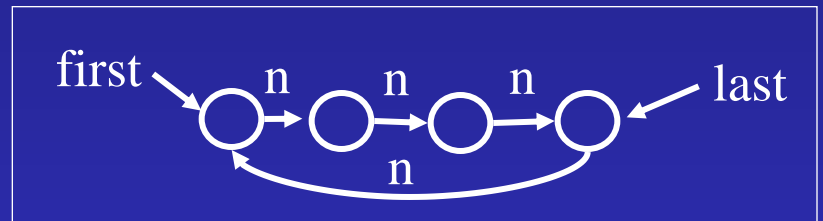
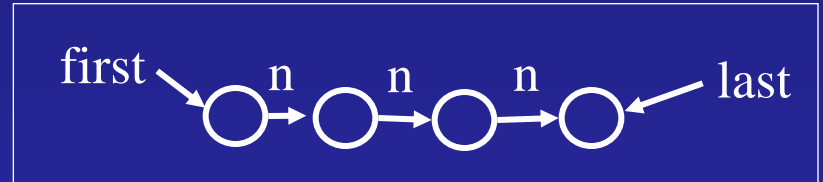
- Programs with pointers and dynamically allocated data structures are error prone
- Automatically prove correctness
- Identify subtle bugs at compile time

Interesting Properties of Heap Manipulating Programs

- No null dereference
- No memory leaks
- Preservation of data structure invariant
- Correct API usage
- Partial correctness
- Total correctness

Example

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



Interesting Properties

```
rotate(List first, List last) {  ✓No null-de references
  if ( first != NULL) {
    last → next = first;
    first = first → next;
    last = last → next;
    last → next = NULL;
  }
}
```

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

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

Partial Correctness

```
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;  
}
```

Partial Correctness

```
List quickSort(List p, List q) {  
    if(p==q || q == NULL)  
        return p;  
    List h = partition(p,q);  
    List x = p→n;  
    p →n = NULL;  
    List low = quickSort(h, p);  
    List high = quickSort(x, NULL);  
    p→n = high;  
    return low;  
}
```

Challenges

- Specification
 - Desired properties
 - Program Semantics
- Automatic Verification
 - Program Semantics \Rightarrow Desired properties
 - Undecidable even for simple programs and prooperties

Plan

- Concrete Interpretation of Heap
- Canonical Heap Abstraction
- Abstract Interpretation using Canonical Abstraction
- The TVLA system
- Applications
- Techniques for scaling

Logical Structures (Labeled Graphs)

Fixed

- Nullary relation symbols
- Unary relation symbols
- Binary relation symbols
- FO^{TC} over $\text{TC}, \forall \exists \neg \wedge \vee$ express logical structure properties
- Logical Structures provide meaning for relations
 - A set of individuals (nodes) U
 - Interpretation of relation symbols in P
 - $p^0() \rightarrow \{0,1\}$
 - $p^1(v) \rightarrow \{0,1\}$
 - $p^2(u,v) \rightarrow \{0,1\}$

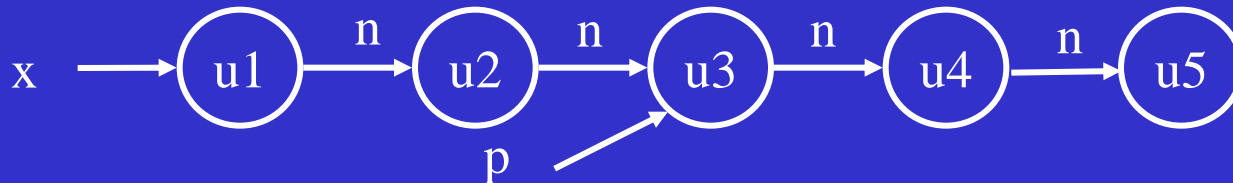
Representing Stores as Logical Structures

- Locations \approx Individuals
- Program variables \approx Unary relations
- Fields \approx Binary relations
- Example
 - $U = \{u1, u2, u3, u4, u5\}$
 - $x = \{u1\}$, $p = \{u3\}$ $n = \{\langle u1, u2 \rangle, \langle u2, u3 \rangle, \langle u3, u4 \rangle, \langle u4, u5 \rangle\}$

	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



Example: List Creation

```
typedef struct node {  
    int val;  
    struct node *next;  
} *List;
```

```
List create (...)
```

```
{
```

```
List x, t;
```

```
x = NULL;
```

```
while (...) do {
```

```
    t = malloc();
```

```
    t →next=x;
```

```
    x = t ;}
```

```
return x;
```

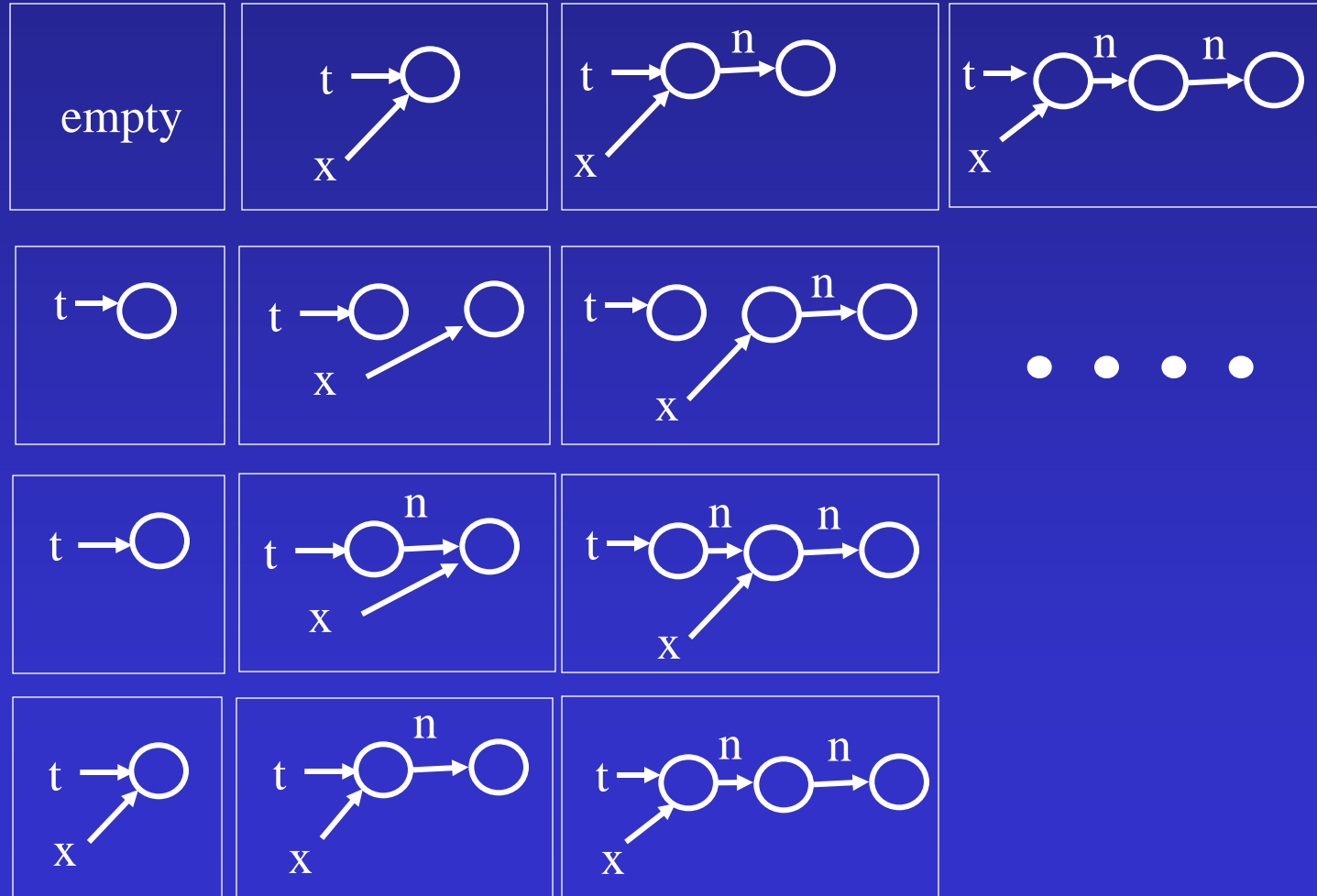
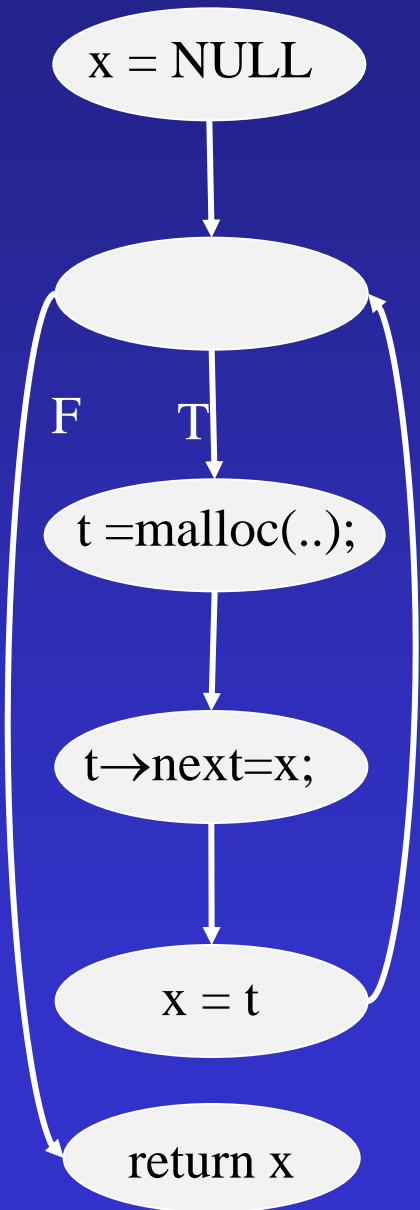
```
}
```

✓ No null dereferences

✓ No memory leaks

✓ Returns acyclic list

Example: Concrete Interpretation



Concrete Interpretation Rules

Statement	Update formula
$x = \text{NULL}$	$x'(v) = 0$
$x = \text{malloc}()$	$x'(v) = \text{IsNew}(v)$
$x = y$	$x'(v) = y(v)$
$x = y \rightarrow \text{next}$	$x'(v) = \exists w: y(w) \wedge n(w, v)$
$x \rightarrow \text{next} = y$	$n'(v, w) = (x(v) ? y(w) : n(v, w))$

Invariants

- No garbage

$$\forall v: \bigvee_{\{x \in PVar\}} \exists w: x(w) \wedge n^*(w, v)$$

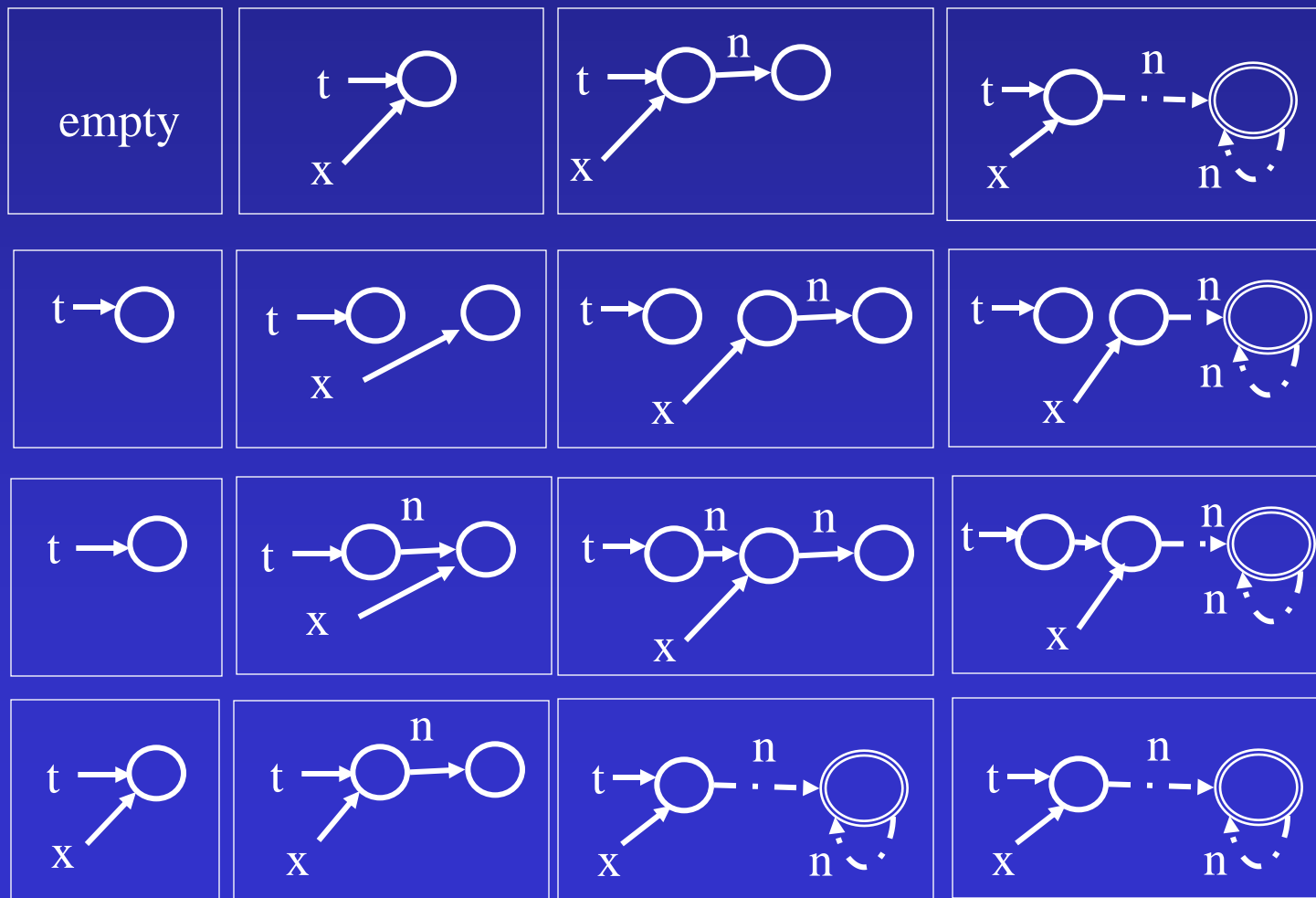
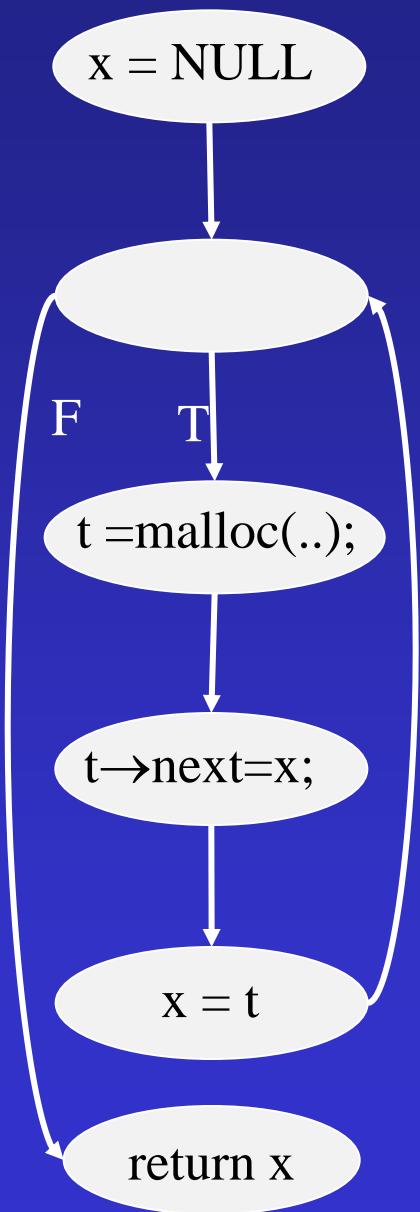
- Acyclic list(x)

$$\forall v, w: x(v) \wedge n^*(v, w) \rightarrow \neg n^+(w, w)$$

- Reverse (x)

$$\forall v, w, r: x(v) \wedge n^*(v, w) \rightarrow \\ n(w, r) \leftrightarrow n'(r, w)$$

Example: Abstract Interpretation



3-Valued Logical Structures

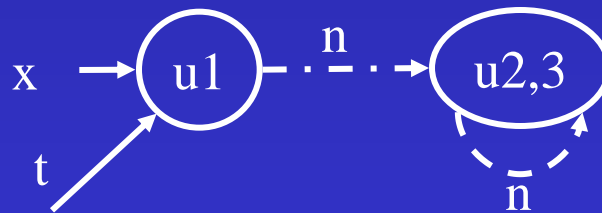
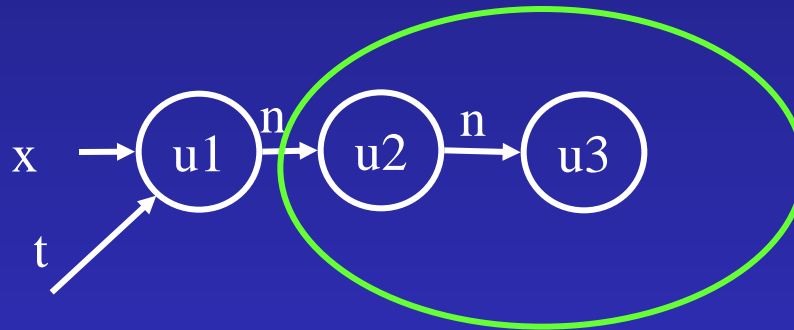
- A set of individuals (nodes) U
- Relation meaning
 - Interpretation of relation symbols in P
 - $p^0() \rightarrow \{0, 1, 1/2\}$
 - $p^1(v) \rightarrow \{0, 1, 1/2\}$
 - $p^2(u, v) \rightarrow \{0, 1, 1/2\}$
- A join semi-lattice: $0 \sqcup 1 = 1/2$

Canonical Abstraction (β)

- Partition the individuals into **equivalence classes** based on the values of their unary relations
 - Every individual is mapped into its equivalence class
- Collapse relations via \sqcup
 - $p^S(u'_1, \dots, u'_k) = \sqcup \{p^B(u_1, \dots, u_k) \mid f(u_1)=u'_1, \dots, f(u_k)=u'_k\}$
- At most 2^A abstract individuals

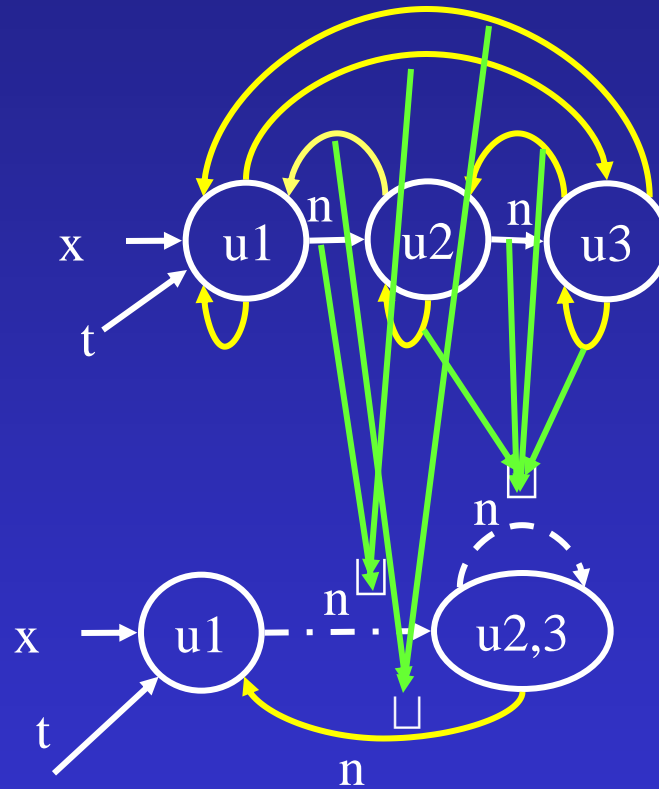
Canonical Abstraction

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



Canonical Abstraction

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

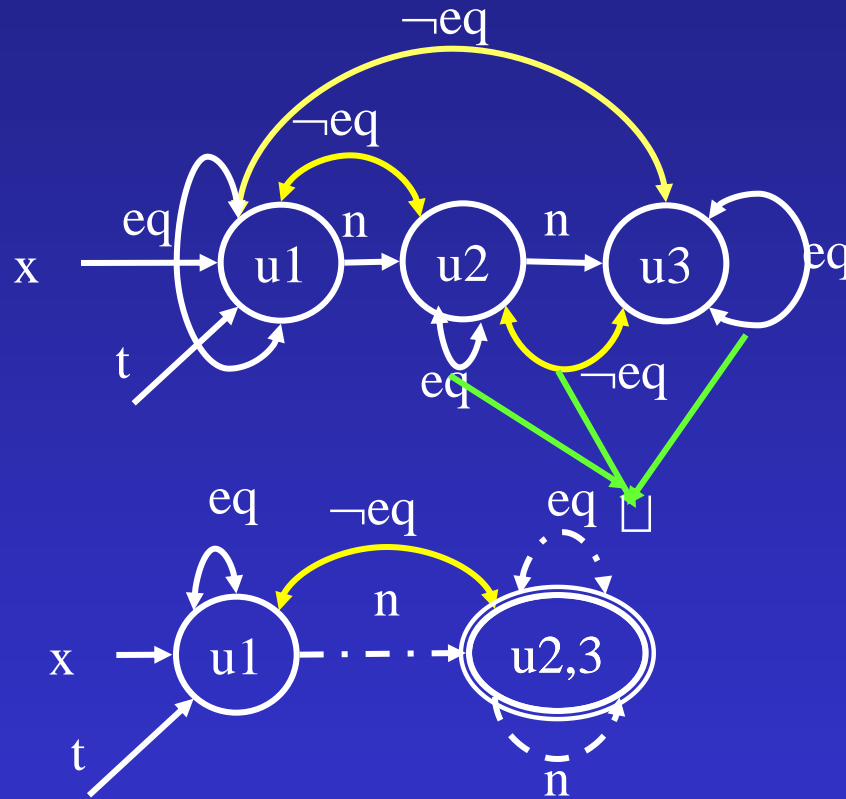


Canonical Abstraction and Equality

- **Summary nodes** may represent more than one element
- (In)equality need not be preserved under abstraction
- Explicitly record equality
- Summary nodes are nodes with $eq(u, u)=1/2$

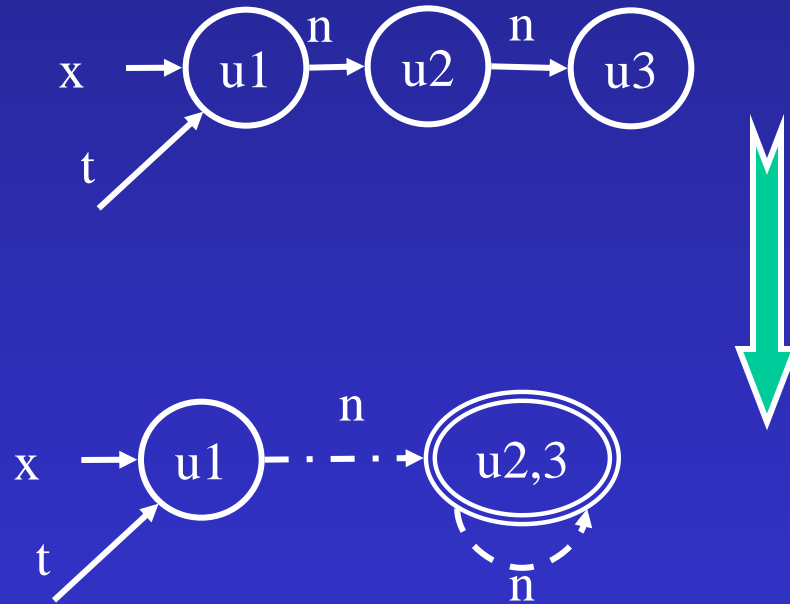
Canonical Abstraction and Equality

```
x = NULL;  
while (...) do {  
    t = malloc();  
    t->next=x;  
    x = t  
}
```



Canonical Abstraction

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

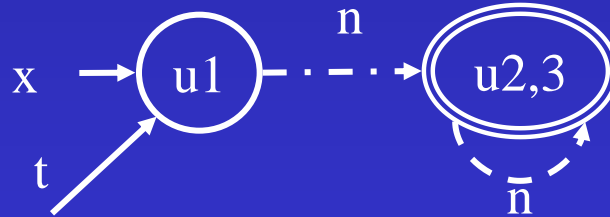
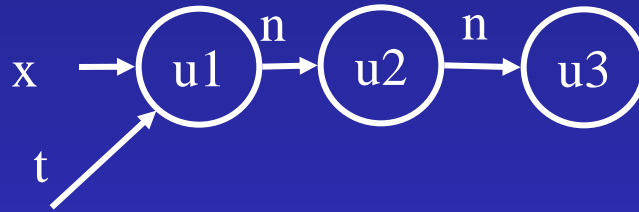


Canonical Abstraction

- Partition the individuals into **equivalence classes** based on the values of their unary relations
 - Every individual is mapped into its equivalence class
- Collapse relations via \sqcup
 - $p^S(u'_1, \dots, u'_k) = \sqcup \{p^B(u_1, \dots, u_k) \mid f(u_1)=u'_1, \dots, f(u_k)=u'_k\}$
- At most 2^A abstract individuals

Canonical Abstraction

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



Limitations

- Information on summary nodes is lost

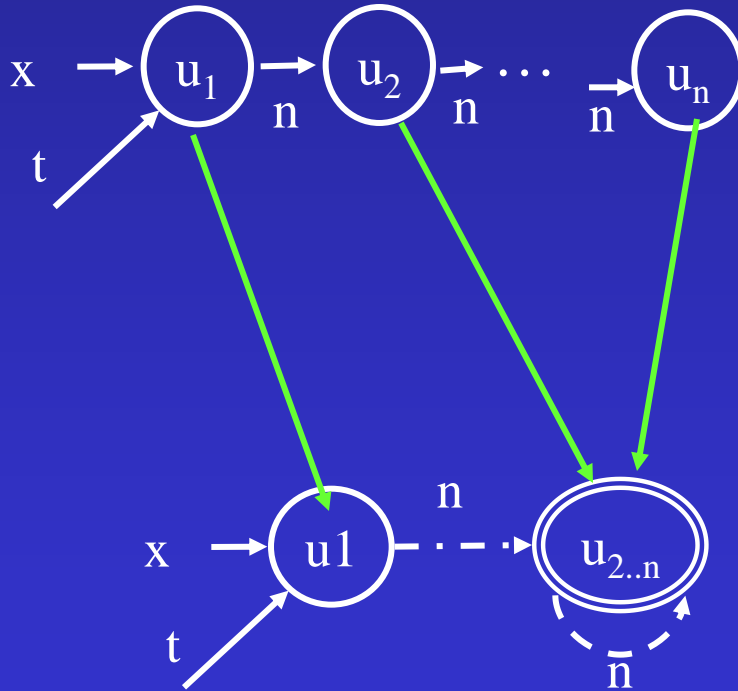
Increasing Precision

- Global invariants
 - User-supplied, or consequence of the semantics of the programming language
 - Naturally expressed in FO^{TC}
- Record extra information in the concrete interpretation
 - Tunes the abstraction
 - Refines the concretization

Cyclicity relation

$$c[x]() = \exists v_1, v_2: x(v_1) \wedge n^*(v_1, v_2) \wedge n^+(v_2, v_2)$$

$$c[x]()=0$$

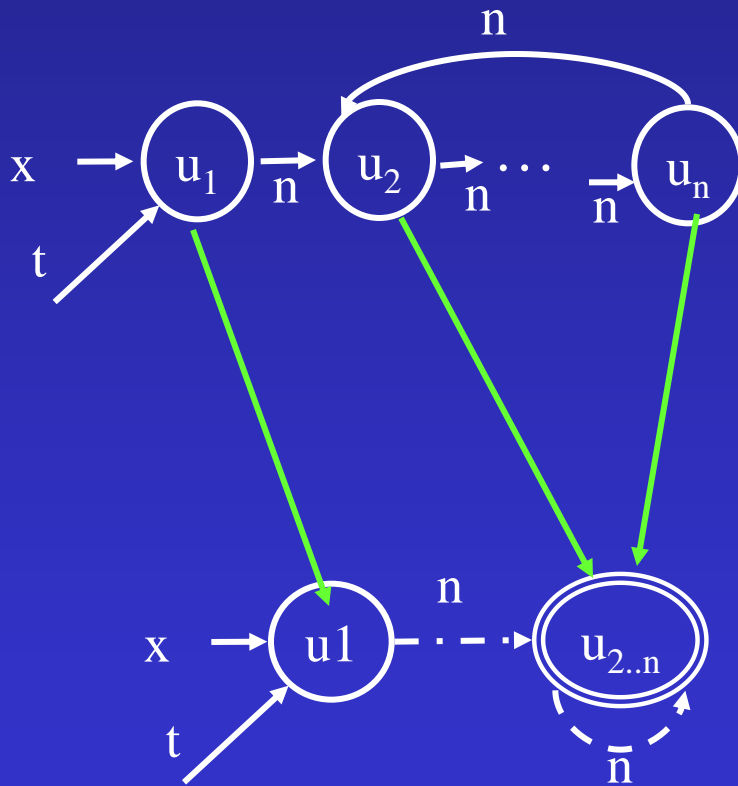


$$c[x]()=0$$

Cyclicity relation

$$c[x]() = \exists v_1, v_2: x(v_1) \wedge n^*(v_1, v_2) \wedge n^+(v_2, v_2)$$

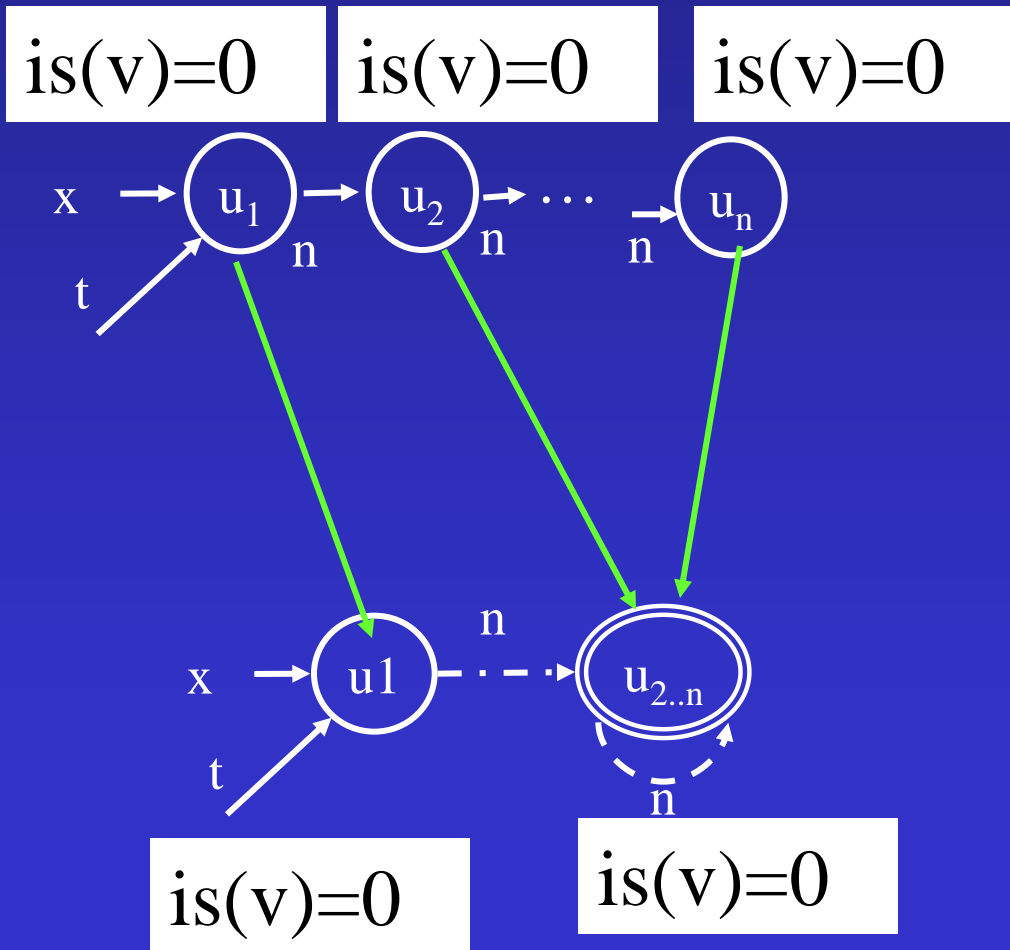
$$c[x]()=1$$



$$c[x]()=1$$

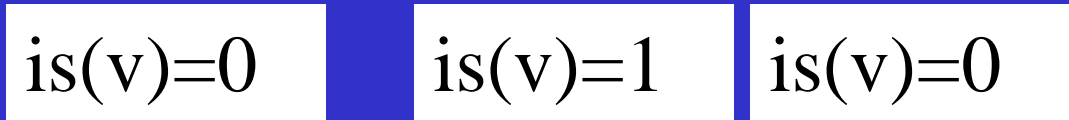
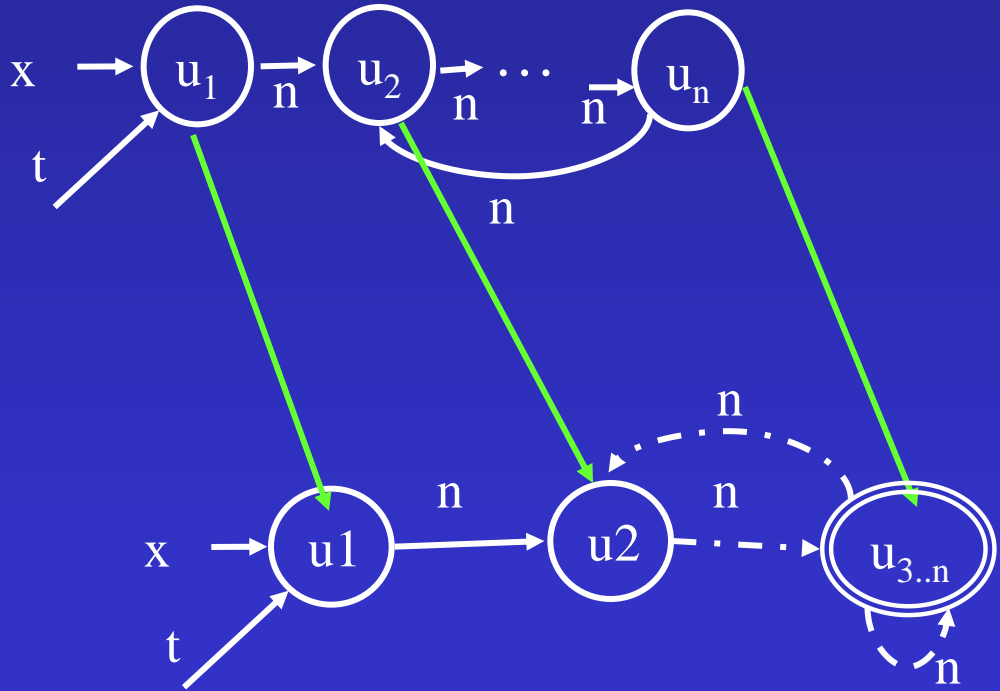
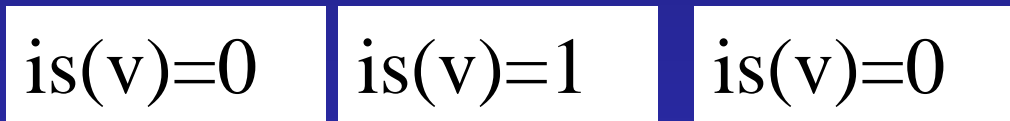
Heap Sharing relation

$$is(v) = \exists v_1, v_2: n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$$



Heap Sharing relation

$$is(v) = \exists v_1, v_2: n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$$

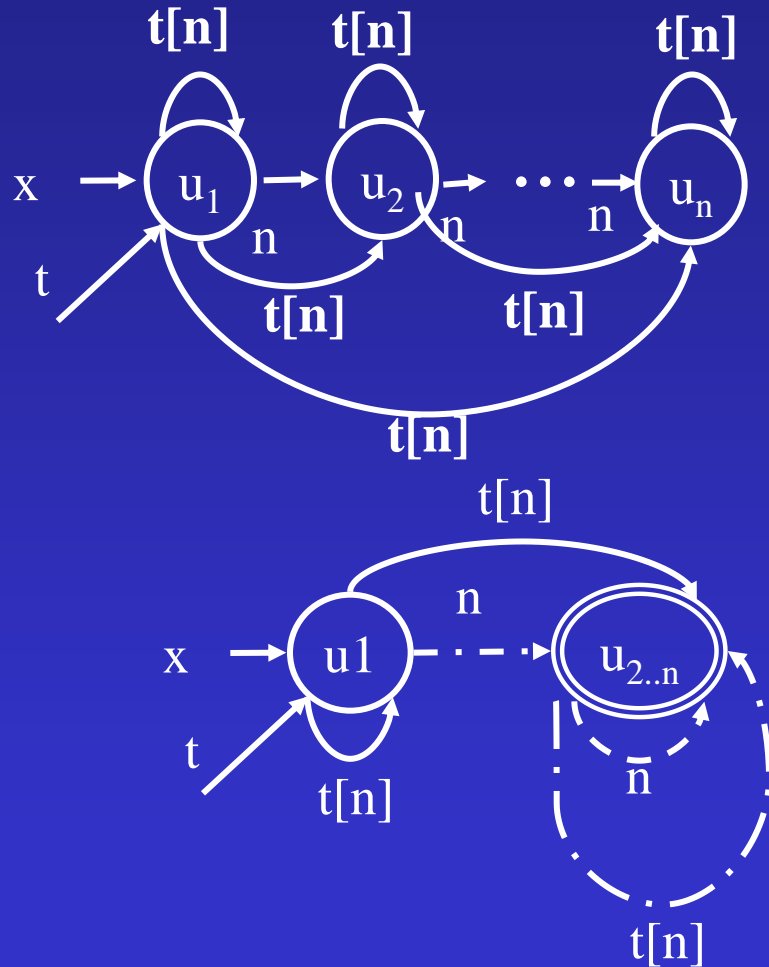


Concrete Interpretation Rules

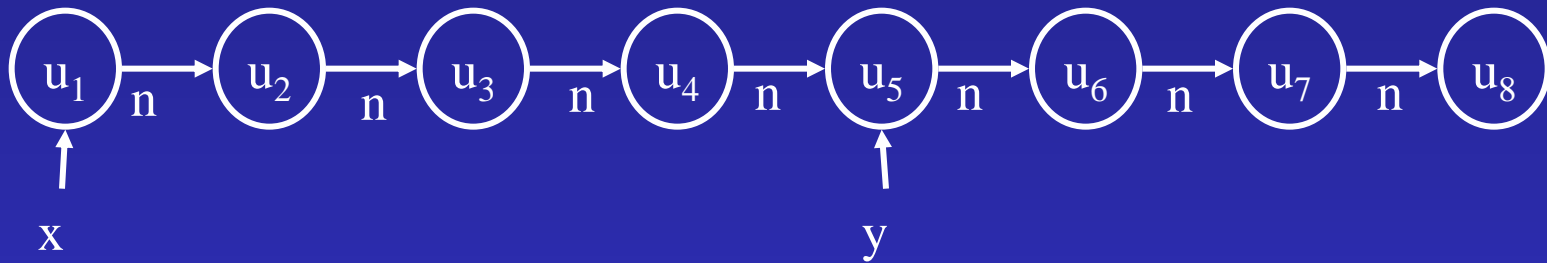
Statement	Update formula
$x = \text{NULL}$	$x'(v) = 0$
$x = \text{malloc}()$	$x'(v) = \text{IsNew}(v)$ $\text{is}'(v) = \text{is}(v) \wedge \neg \text{IsNew}(v)$
$x = y$	$x'(v) = y(v)$
$x = y \rightarrow \text{next}$	$x'(v) = \exists w: y(w) \wedge n(w, v)$
$x \rightarrow \text{next} = \text{NULL}$	$n'(v, w) = \neg x(v) \wedge n(v, w)$ $\text{is}'(v) = \text{is}(v) \wedge$ $\exists v1, v2: n(v1, v) \wedge \neg x(v1) \wedge$ $n(v2, v) \wedge \neg x(v2) \wedge$ $\neg \text{eq}(v1, v2)$

Reachability relation

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

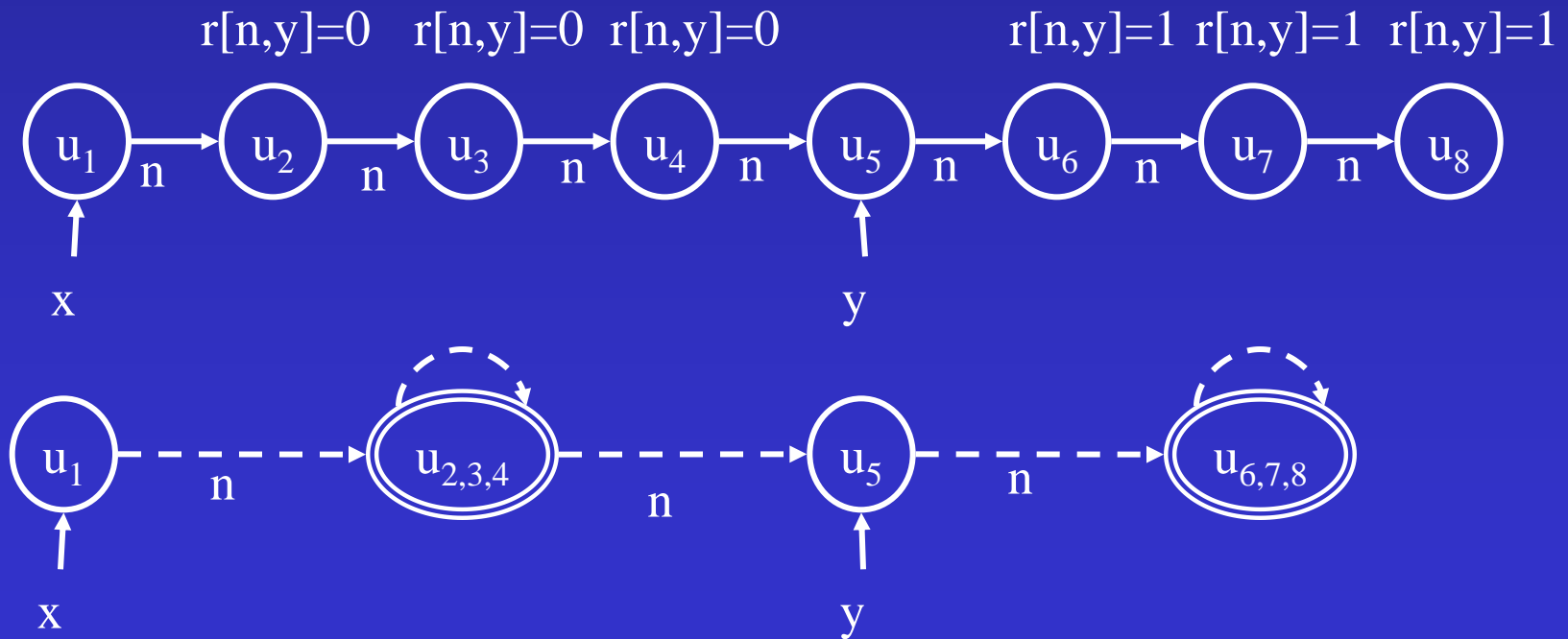


List Segments



Reachability from a variable

- $r[n,y](v) = \exists w: y(w) \wedge n^*(w, v)$



Additional Instrumentation relations

- $\text{inOrder}(v) = \forall w: n(v, w) \rightarrow \text{data}(v) \leq \text{data}(w)$
- $c_{fb}(v) = \forall w: f(v, w) \rightarrow b(w, v)$
- $\text{tree}(v)$
- $\text{dag}(v)$
- Weakest Precondition
[Ramalingam, PLDI'02]
- Learned via Inductive Logic Programming
[Loginov, CAV'05]
- Counterexample guided refinement

Instrumentation (Summary)

- Refines the abstraction

$$\text{is}(v) = \exists v_1, v_2: n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$$

- Adds global invariants

$$\text{is}(v) \leftrightarrow \exists v_1, v_2: n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$$

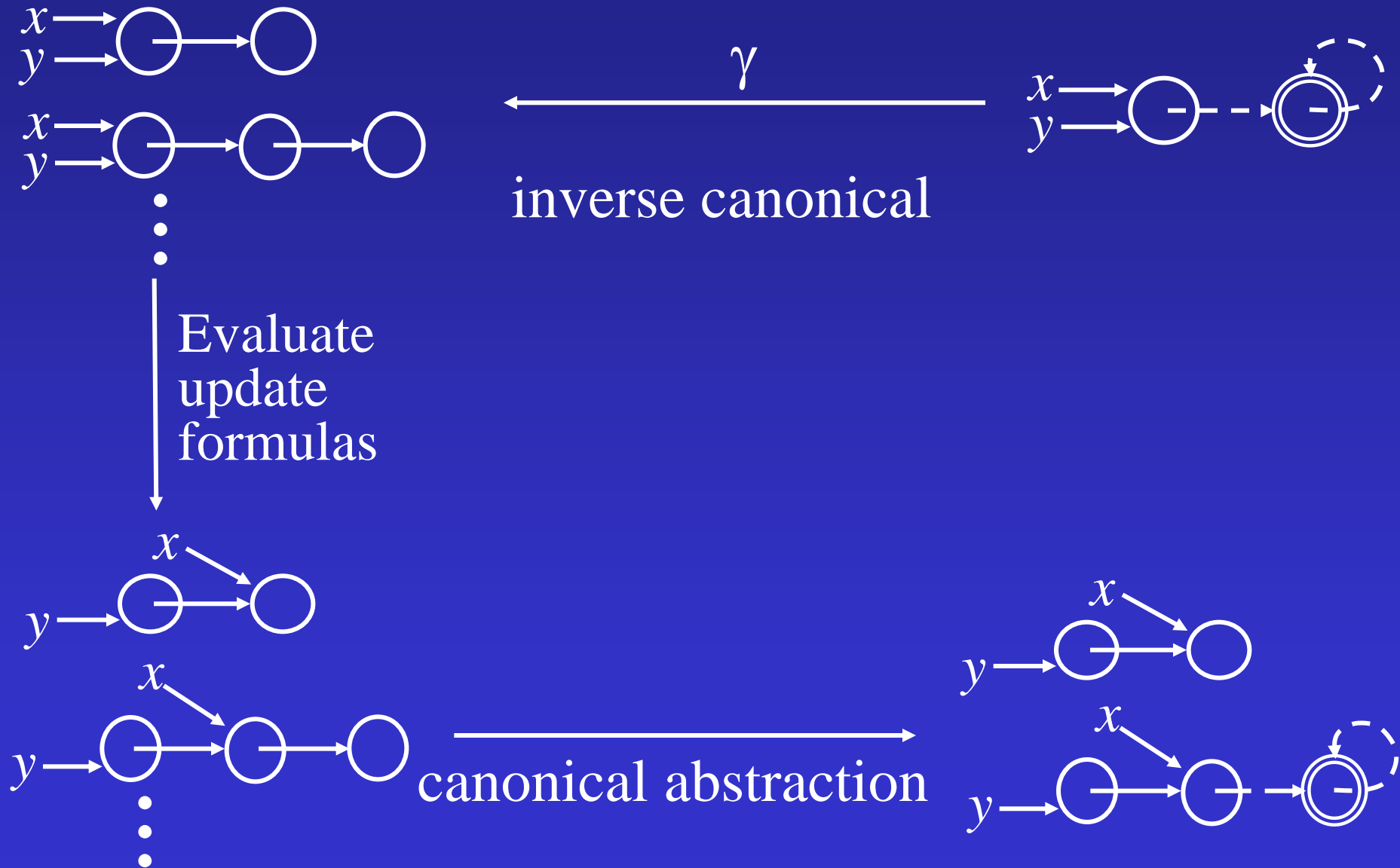
$$\gamma(S^\#) = \{S : S \models \Sigma, \beta(S) = S^\#\}$$

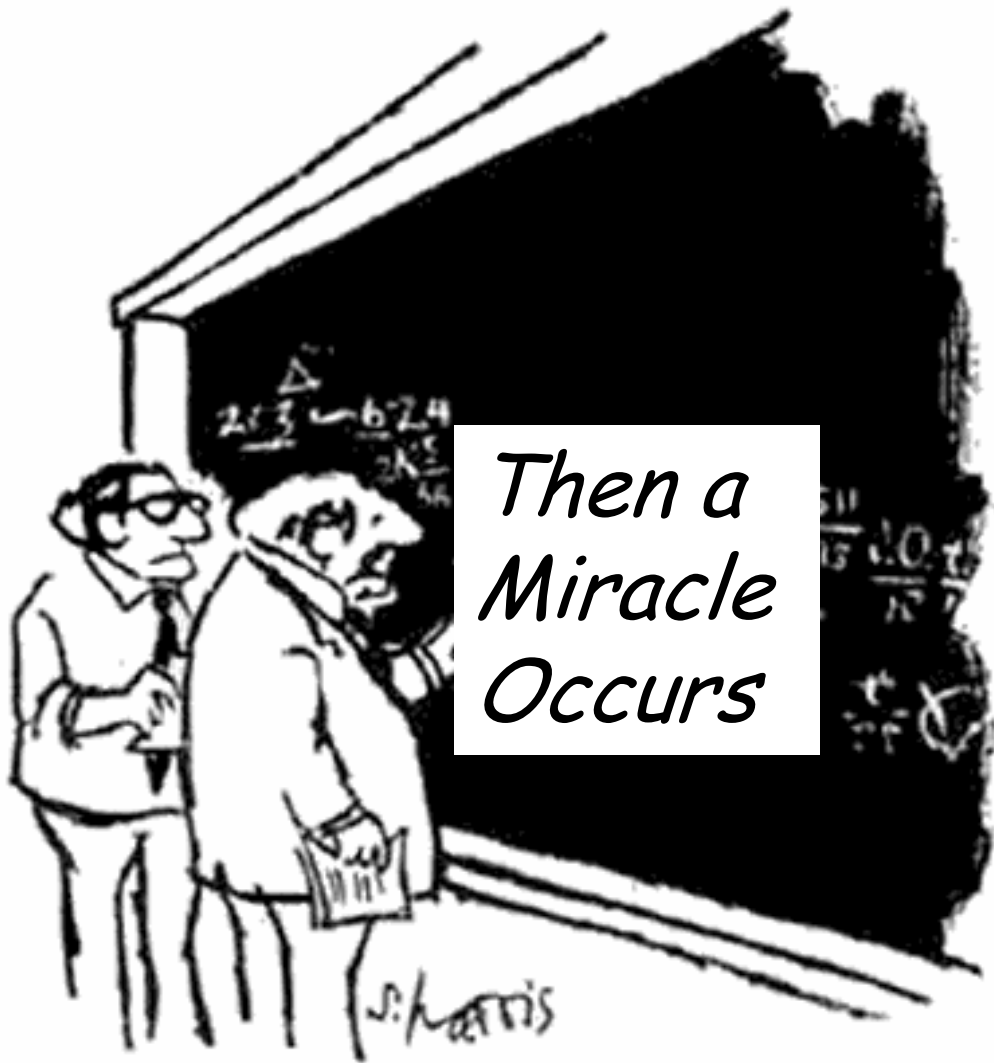
- But requires update-formulas (generated automatically in TVLA2)

Abstract Interpretation

- Best Transformers
- Kleene Evaluation
- Kleene Evaluation + semantic reduction
 - Focus Based Transformers

Best Transformer Transformer ($x = x \rightarrow n$)





*Then a
Miracle
Occurs*

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

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

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

Embedding

- A logical structure **B** can be embedded into a structure **S** via an onto function f ($\mathbf{B} \sqsubseteq^f \mathbf{S}$) if the basic relations are preserved, i.e.,

$$p^{\mathbf{B}}(u_1, \dots, u_k) \sqsubseteq p^{\mathbf{S}}(f(u_1), \dots, f(u_k))$$

- **S** is a **tight embedding** of **B** with respect to f if:
 - **S** does not lose unnecessary information, i.e.,
 - $p^{\mathbf{S}}(u^{\#}_1, \dots, u^{\#}_k) = \sqcup \{p^{\mathbf{B}}(u_1, \dots, u_k) \mid f(u_1)=u^{\#}_1, \dots, f(u_k)=u^{\#}_k\}$
- Canonical Abstraction is a tight embedding

Embedding and Concretization

- Two natural choices
 - $B \in \gamma_1(S)$ if B can be embedded into S via an onto function f ($B \sqsubseteq^f S$)
 - $B \in \gamma_2(S)$ if S is a tight embedding of B

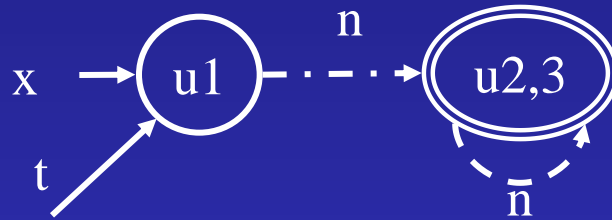
Embedding Theorem

- Assume $\mathbf{B} \sqsubseteq^f \mathbf{S}$,

$$p^{\mathbf{B}}(u_1, \dots, u_k) \sqsubseteq p^{\mathbf{S}}(f(u_1), \dots, f(u_k))$$

- Then every formula φ is preserved:
 - If $\llbracket \varphi \rrbracket = 1$ in \mathbf{S} , then $\llbracket \varphi \rrbracket = 1$ in \mathbf{B}
 - If $\llbracket \varphi \rrbracket = 0$ in \mathbf{S} , then $\llbracket \varphi \rrbracket = 0$ in \mathbf{B}
 - If $\llbracket \varphi \rrbracket = 1/2$ in \mathbf{S} , then $\llbracket \varphi \rrbracket$ could be 0 or 1 in \mathbf{B}

Embedding Theorem



$$\exists v: x(v)$$

1=Yes

$$\exists v: x(v) \wedge t(v)$$

1=Yes

$$\exists v: x(v) \wedge y(v)$$

0=No

$$\exists v_1, v_2: x(v_1) \wedge n(v_1, v_2)$$

1/2=Maybe

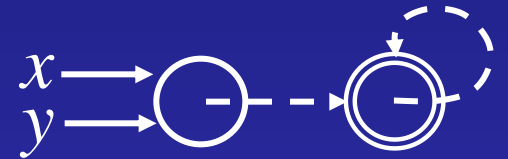
$$\exists v_1, v_2: x(v_1) \wedge n(v_1, v_2) \wedge n^*(v_2, v_1)$$

0=No

$$\exists v_1, v_2: x(v_1) \wedge n^*(v_1, v_2) \wedge n+(v_2, v_2)$$

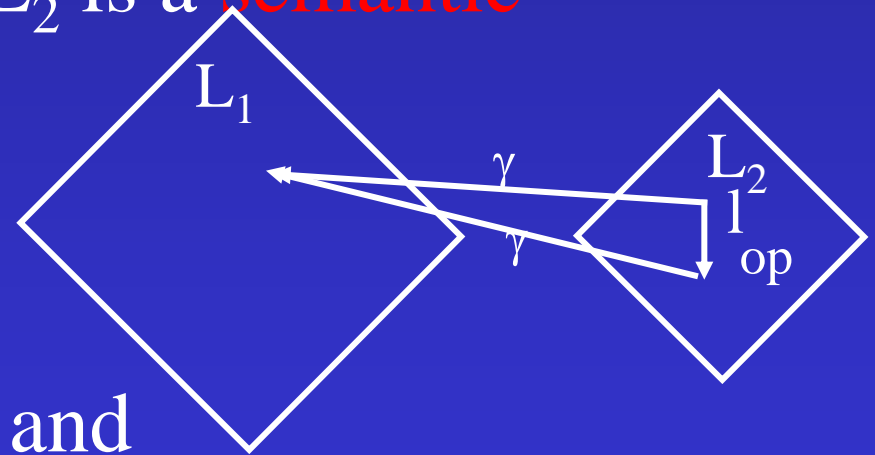
1/2=Maybe

Kleene Transformer ($x = x \rightarrow n$)

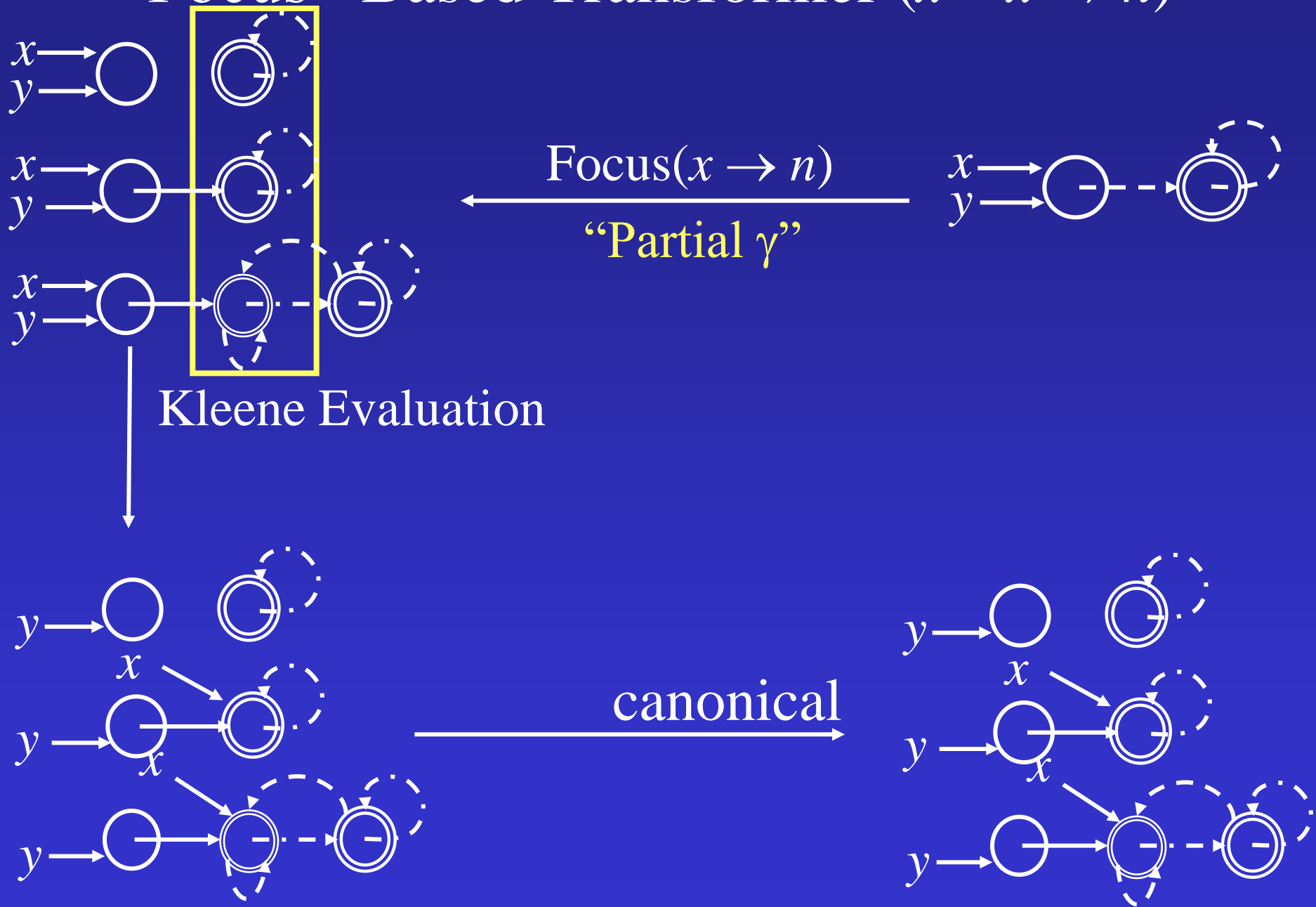


Semantic Reduction

- 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**
 - $\forall l \in L_2 \quad op(l) \sqsubseteq l$
 - $\gamma(op(l)) = \gamma(l)$
- Can be applied before and after basic operations



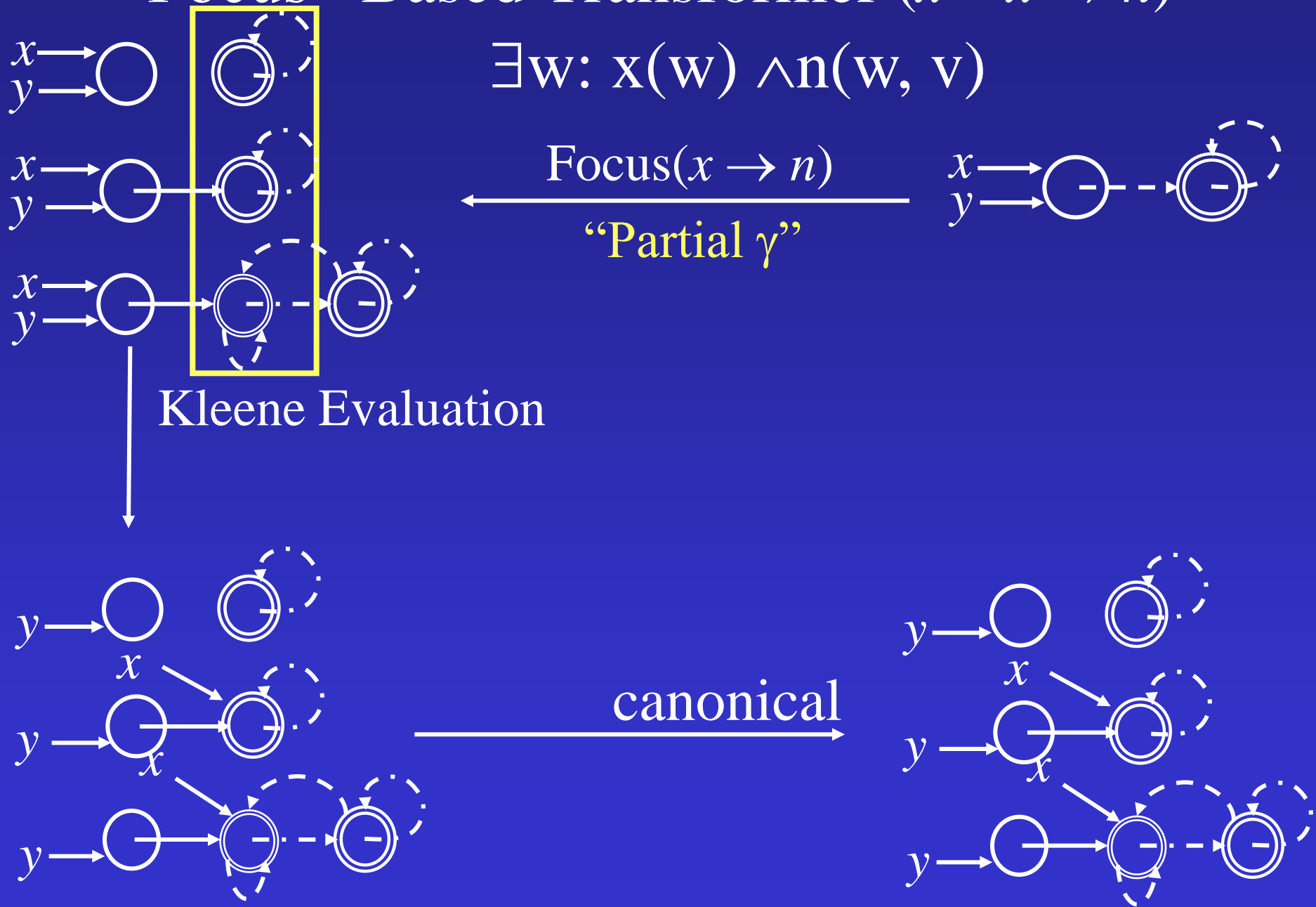
“Focus”-Based Transformer ($x = x \rightarrow n$)



The Focus Operation

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

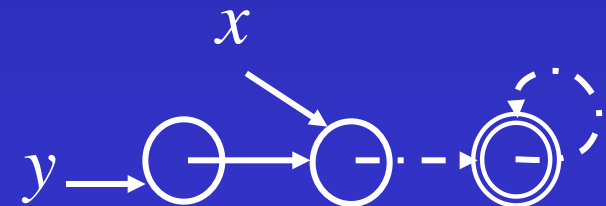
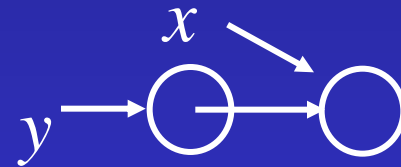
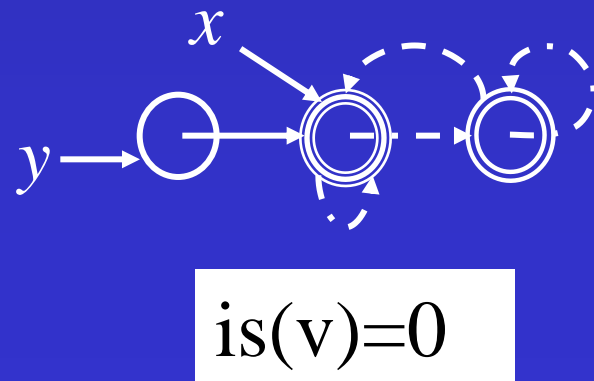
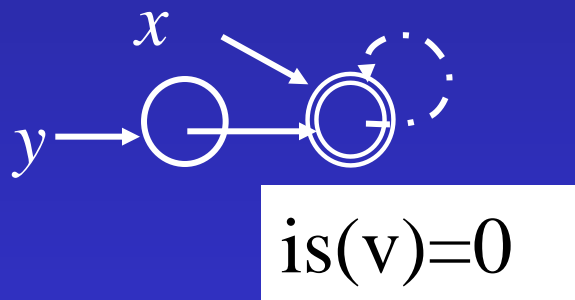
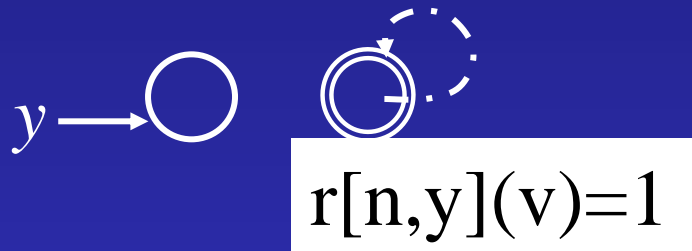
“Focus”-Based Transformer ($x = x \rightarrow n$)



The Coercion Principle

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

Apply Constraint Solver



Sources of Constraints

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

Example Constraints

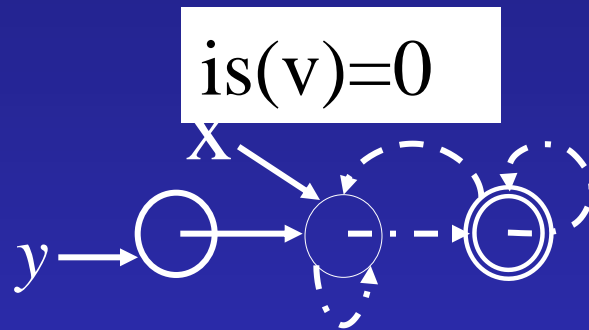
$$x(v1) \wedge x(v2) \rightarrow eq(v1, v2)$$

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

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

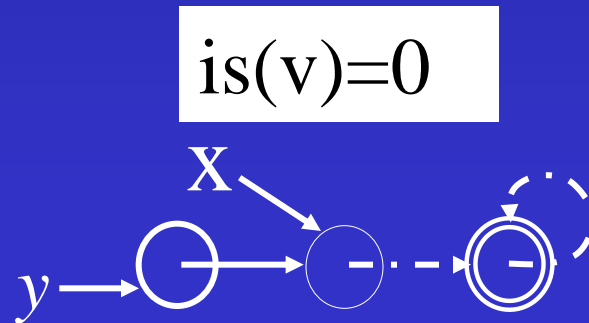
$$n^*(v3, v4) \leftrightarrow t[n](v1, v2)$$

Apply Constraint Solver



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

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



Summary Transformers

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

Three Valued Logic Analysis (TVLA)

T. Lev-Ami & R. Manevich

- Input (FO^{TC})
 - Concrete interpretation rules
 - Definition of instrumentation relations
 - Definition of safety properties
 - First Order Transition System (TVP)
- Output
 - Warnings (text)
 - The 3-valued structure at every node (invariants)

TVLA inputs

- TVP - Three Valued Program
 - Predicate declaration
 - Action definitions SOS
 - Statements
 - Conditions
 - Control flow graph
 - TVS - Three Valued Structure
- Program independent

Memory Leakage

```
List reverse(Element *head)
```

```
{
```

```
    List rev, ne;
```

```
    rev = NULL;
```

```
    while (head != NULL) {
```

```
        ne = head → next;
```

```
        head → next = rev;
```

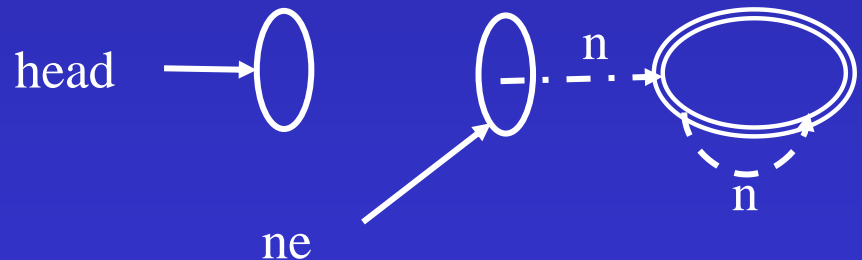
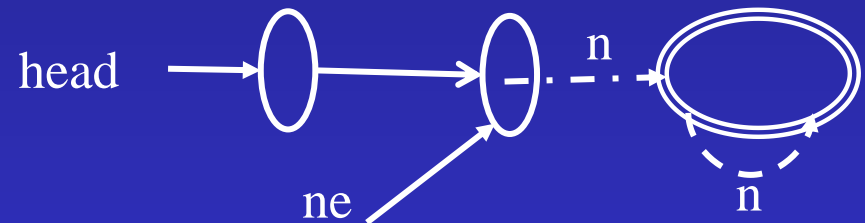
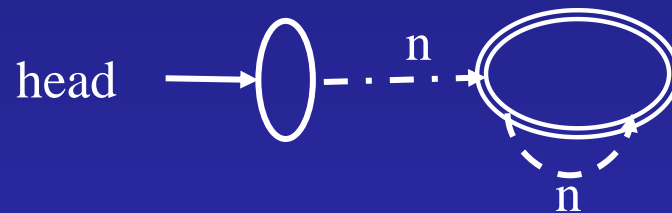
```
        head = ne;
```

```
        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;
```

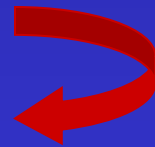
```
        head → next = rev;
```

```
        rev = head;
```

```
        head = ne;
```

```
    }
```

```
    return rev;
```



✓ No memory
leaks

Mark and Sweep

```
void Mark(Node root) {  
  if (root != NULL) {  
    pending =  $\emptyset$   
    pending = pending  $\cup$  {root}  
    marked =  $\emptyset$   
    while (pending  $\neq$   $\emptyset$ ) {  
      x = SelectAndRemove(pending)  
      marked = marked  $\cup$  {x}  
      t = x  $\rightarrow$  left  
      if (t  $\neq$  NULL)  
        if (t  $\notin$  marked)  
          pending = pending  $\cup$  {t}  
      t = x  $\rightarrow$  right  
      if (t  $\neq$  NULL)  
        if (t  $\notin$  marked)  
          pending = pending  $\cup$  {t}  
    }  
  }  
  assert(marked == Reachset(root))  
}
```

```
void Sweep() {  
  unexplored = Universe  
  collected =  $\emptyset$   
  while (unexplored  $\neq$   $\emptyset$ ) {  
    x = SelectAndRemove(unexplored)  
    if (x  $\notin$  marked)  
      collected = collected  $\cup$  {x}  
  }  
  assert(collected ==  
          Universe - Reachset(root))  
}
```

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

Mark

```
void Mark(Node root) {
  if (root != NULL) {
    pending =  $\emptyset$ 
    pending = pending  $\cup$  {root}
    marked =  $\emptyset$ 
    while (pending  $\neq$   $\emptyset$ ) {
      x = SelectAndRemove(pending)
      marked = marked  $\cup$  {x}
      t = x  $\rightarrow$  left
      if (t  $\neq$  NULL)
        if (t  $\notin$  marked)
          pending = pending  $\cup$  {t}
      t = x  $\rightarrow$  right
      if (t  $\neq$  NULL)
        if (t  $\notin$  marked)
          pending = pending  $\cup$  {t}
    }
  }
}
```

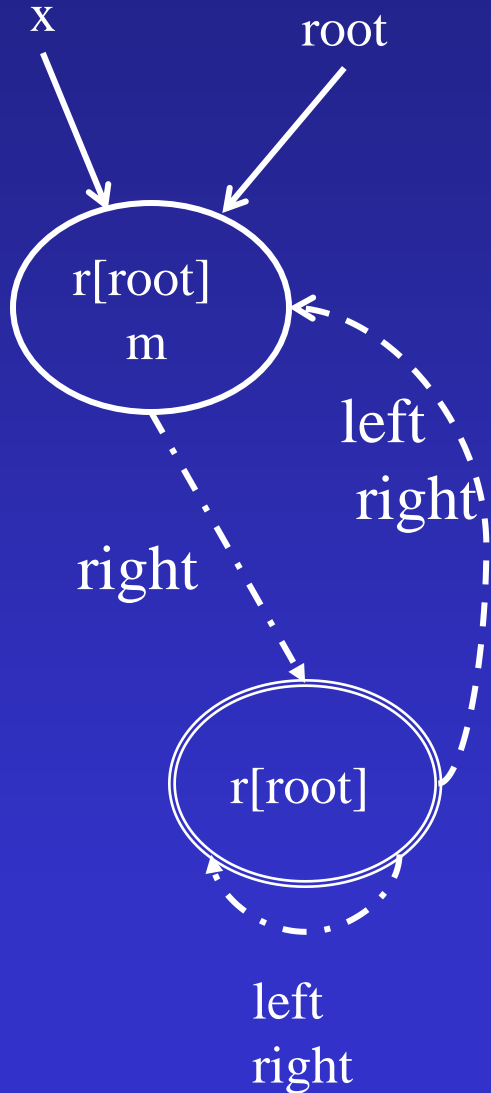
$\exists r: \text{root}(r) \wedge (\text{p}(r) \vee \text{m}(r)) \wedge$
 $\forall v: ((\text{m}(v) \vee \text{p}(v)) \rightarrow \text{reach}[\text{root}](v)) \wedge$
 $\neg(\text{p}(v) \wedge \text{m}(v))$
 $\wedge \forall v, w: ((\text{m}(v) \wedge \neg \text{m}(w) \wedge \neg \text{p}(w)) \rightarrow$
 $\neg \text{successor}(v, w))$

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

Example: Mark

```
void Mark(Node root) {
    if (root != NULL) {
        pending =  $\emptyset$ 
        pending = pending  $\cup$  {root}
        marked =  $\emptyset$ 
        while (pending  $\neq$   $\emptyset$ ) {
            x = SelectAndRemove(pending)
            marked = marked  $\cup$  {x}
            t = x  $\rightarrow$  left
            if (t  $\neq$  NULL)
                if (t  $\notin$  marked)
                    pending = pending  $\cup$  {t}
/*      t = x  $\rightarrow$  right
*      if (t  $\neq$  NULL)
*          if (t  $\notin$  marked)
*              pending = pending  $\cup$  {t}
*/ }
        }
    }
    assert(marked == Reachset(root))
}
```

Bug Found



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

$$\begin{aligned}
 & \exists r: \text{root}(r) \wedge r[\text{root}](r) \wedge \neg p(r) \wedge m(r) \wedge \\
 & \exists e: r[\text{root}](e) \wedge \neg m(e) \wedge \neg \text{root}(e) \wedge \neg p(e) \\
 & \forall r, e: (\text{root}(r) \wedge r[\text{root}](r) \wedge \neg p(r) \wedge m(r) \wedge \\
 & \quad r[\text{root}](e) \wedge \neg m(e)) \wedge \neg \text{root}(e) \wedge \neg p(e) \\
 & \quad \rightarrow \neg \text{left}(r, e)
 \end{aligned}$$

Scaling for Larger Programs

- Staged Analyses
- Represent 3-valued structures with BDDs [Manevich SAS'02]
- Coercer Abstractions [Manevich SAS'04]
- Reduce static costs
- Handling procedures
- Assume/Guarantee Reasoning
 - Use procedure specifications [Yorsh, TACAS'04]
 - Decision procedures for linked data structures [Immerman, CAV'04, Lev-Ami, CADE'05, Yorsh FOSSACS06]

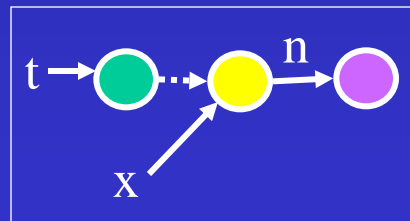
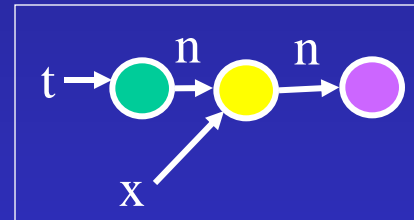
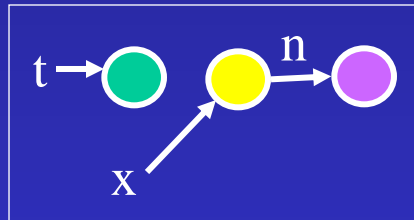
Scaling

- Staged analysis
- Reduce static costs
- Controlled complexity
 - **More coarse abstractions** [Manevich SAS'04]
 - Counter example based refinement
- Exploit “good” program properties
 - **Encapsulation** & Data abstraction
- Handle **procedures efficiently**

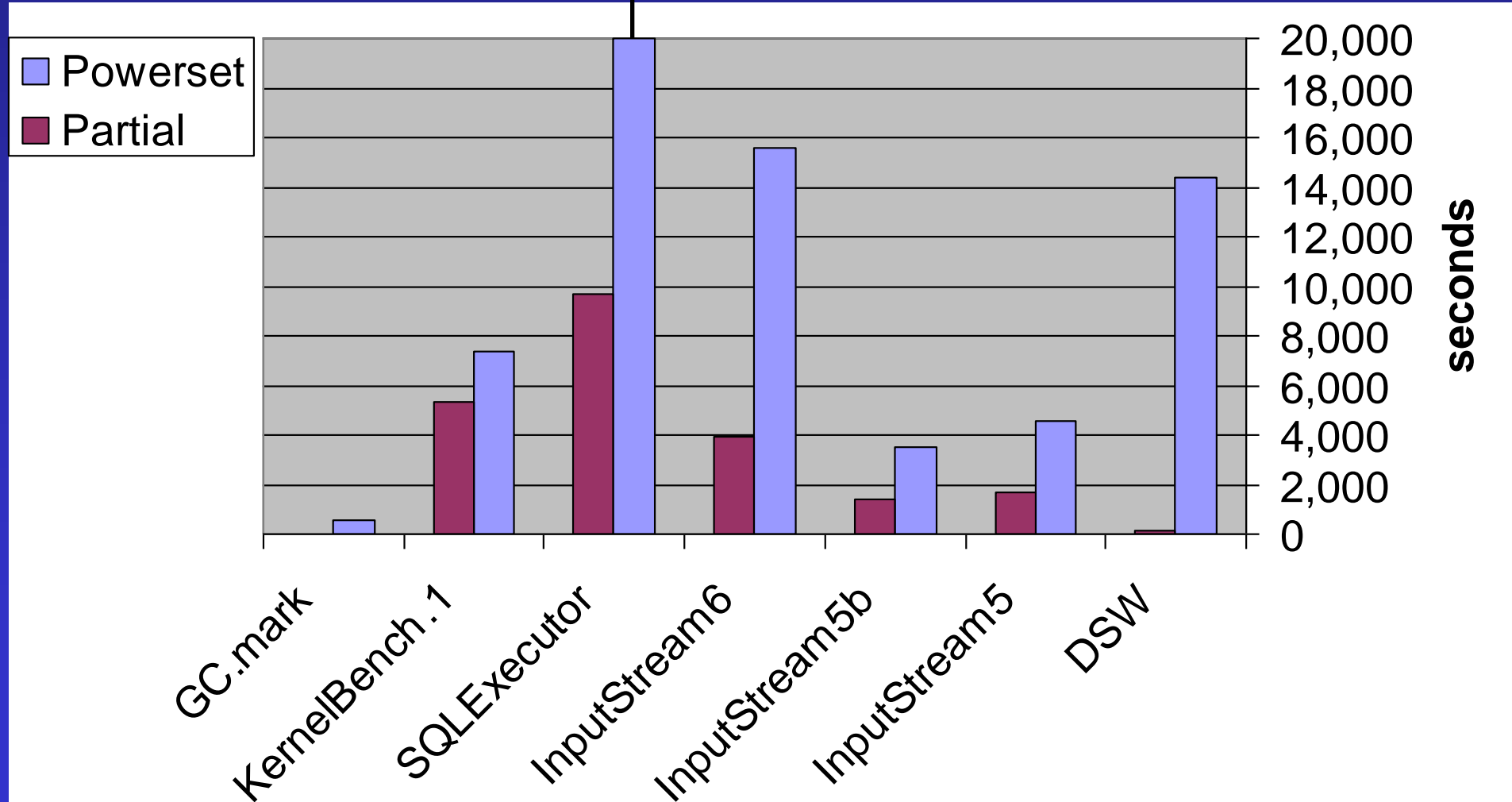
Partially Disjunctive Heap Abstraction (Manevich, SAS'04)

- 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



Interprocedural Analysis

Noam Rinetzky

www.cs.tau.ac.il/~maon

How to handle procedures?

- Pure functions
 - Procedure \equiv input/output relation
 - No side-effects

```
main() {  
  int w=0,x=0,y=0,z=0;  
  w = inc(y);  
  x = inc(z);  
  assert: w+x is even  
}
```

p	ret
0	1
1	2
2	3
..	...

```
int inc(int p) {  
  return 2 + p - 1;  
}
```

How to handle procedures?

- Pure functions
 - Procedure \equiv input/output relation
 - No side-effects

```
main() {  
  int w=0,x=0,y=0,z=0;  
  w = inc(y);  
  x = inc(z);  
  assert: w+x is even  
}
```

w	x	y	z
E	E	E	E
O	E	E	E
O	O	E	E

p	ret
Even	Odd
Odd	Even

```
int inc(int p) {  
  return 2 + p - 1;  
}
```

What about global variables?

- Procedures have side-effects
- Easy fix

p	g	ret	g'
0	0	1	0
...

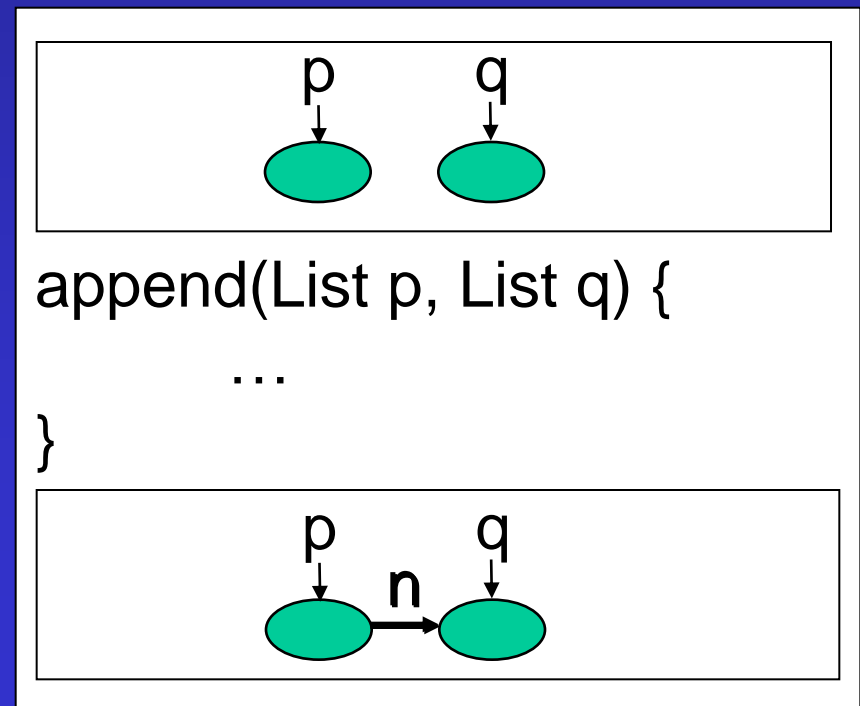
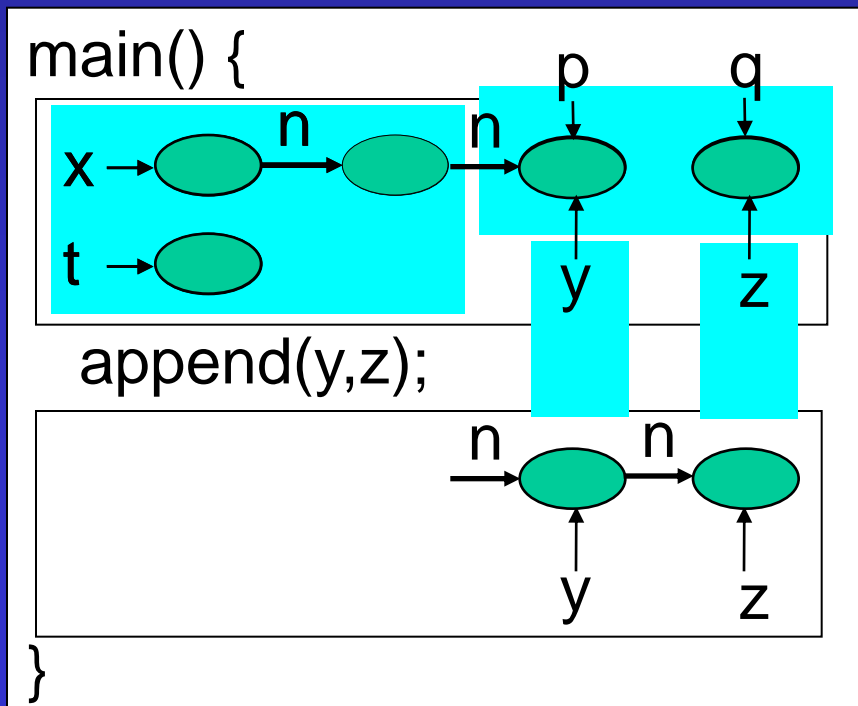
p	g	ret	g'
Even	E/O	Odd	Even
Odd	E/O	Even	Odd

```
int g = 0;
main() {
  int w=0,x=0,y=0,z=0;
  w = inc(y);
  x = inc(z);
  assert: w+x+g is even
}
```

```
int inc(int p) {
  g = p;
  return 2 + p - 1;
}
```

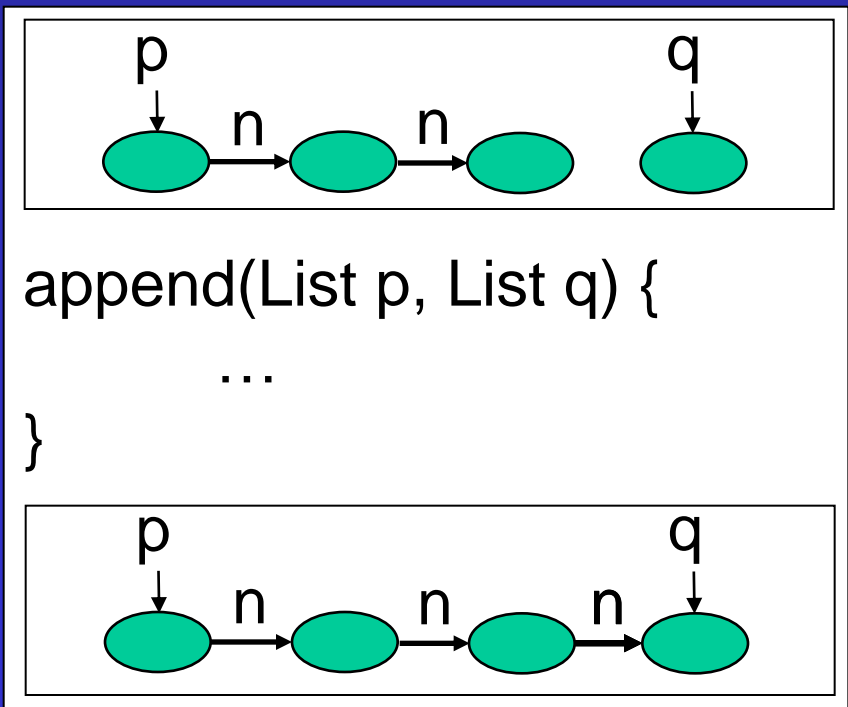
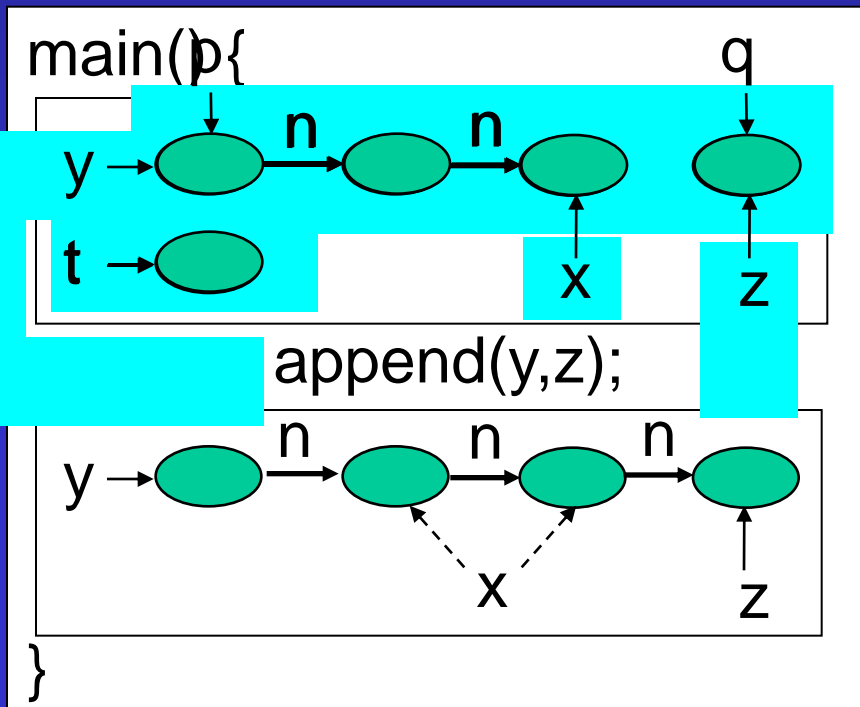

How to tabulate procedures?

- Procedure \equiv input/output relation
 - Not reachable \rightarrow Not effected
 - proc: local (\equiv reachable) heap \rightarrow local heap



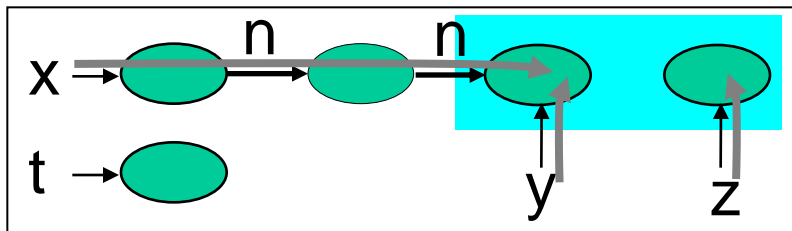
How to handle sharing?

- External sharing may break the functional view



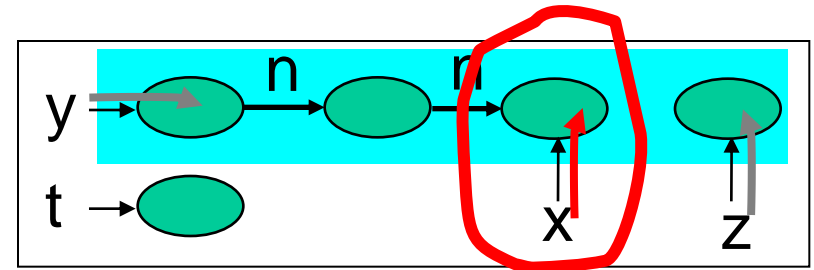
What's the difference?

1st Example



`append(y,z);`

2nd Example

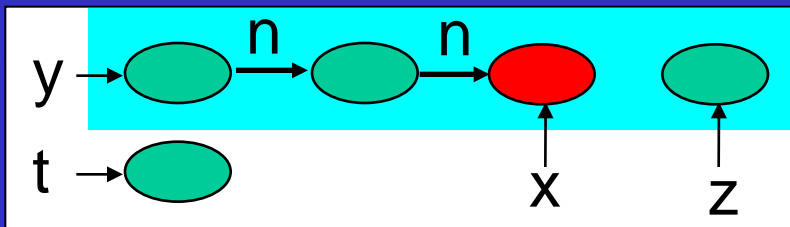


`append(y,z);`

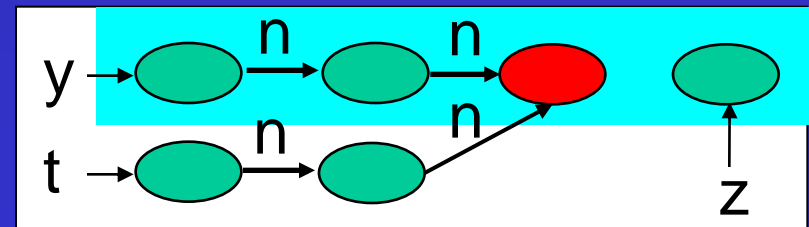
Cutpoints

- 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)



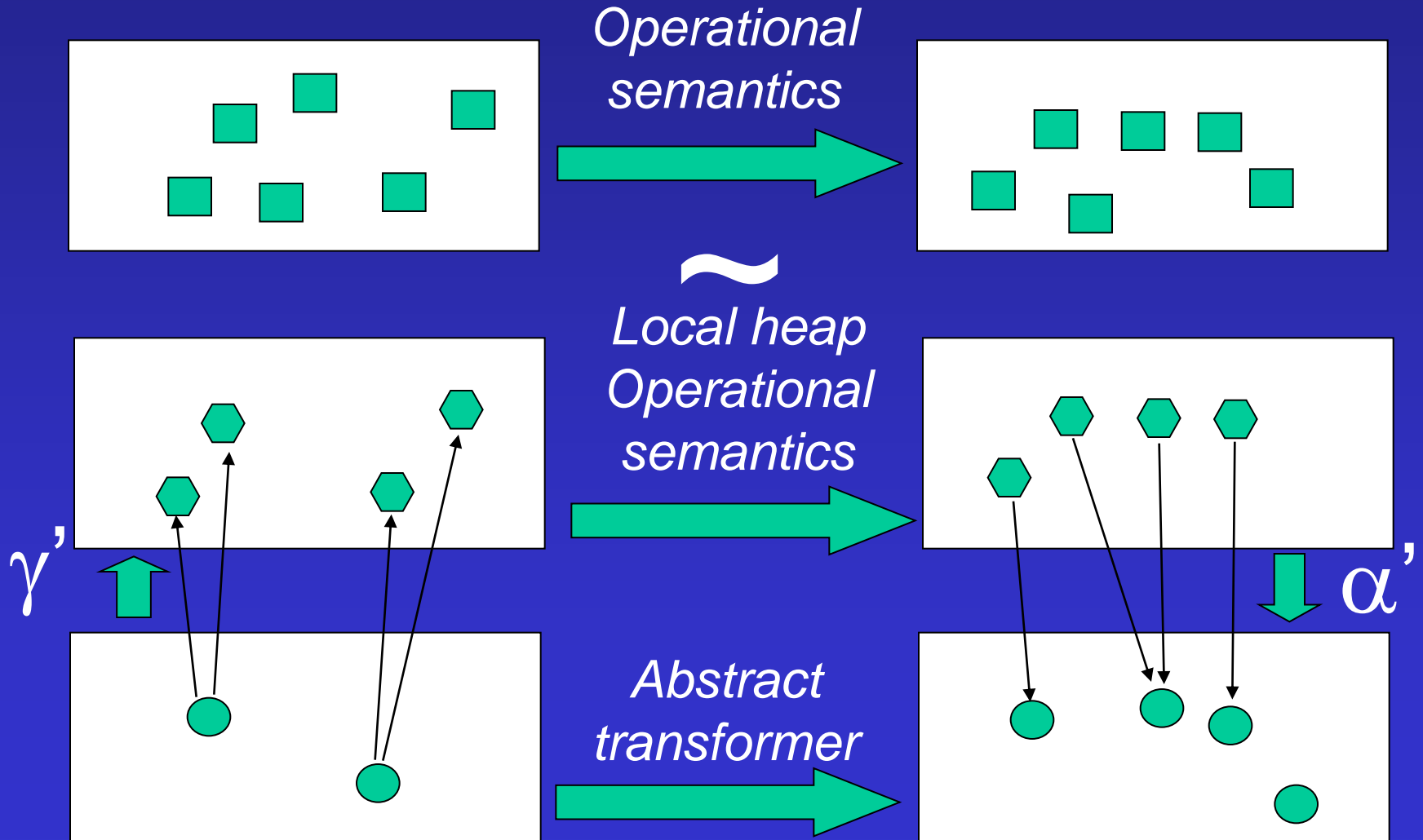
append(y,z)



Main Results(POPL'05)

- Concrete operational semantics
 - Sequential programs
 - Local heap
 - Track cutpoints
 - Storeless
 - good for shape abstractions
 - Observational equivalent with “standard” global store-based heap semantics
 - Java and “clean” C
- Abstractions
 - Shape Analysis of singly-linked lists
 - May-alias [Deutsch, PLDI 04]

Introducing local heap semantics



Main results(SAS'05)

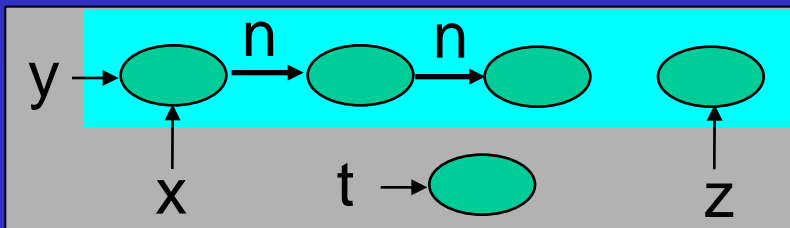
- Cutpoint freedom
- Non-standard concrete semantics
 - Verifies that an execution is cutpoint-free
 - Local heaps
- Interprocedural shape analysis
 - Conservatively verifies
 - program is cutpoint free
 - Desired properties
 - Partial correctness of quicksort
 - Procedure summaries
- Prototype implementation

Cutpoint freedom

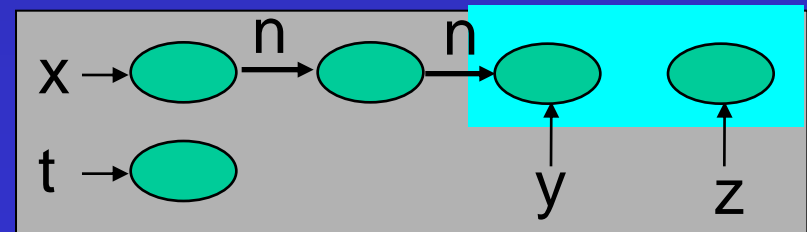
- **Cutpoint-free**

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

append(y,z)



append(y,z)

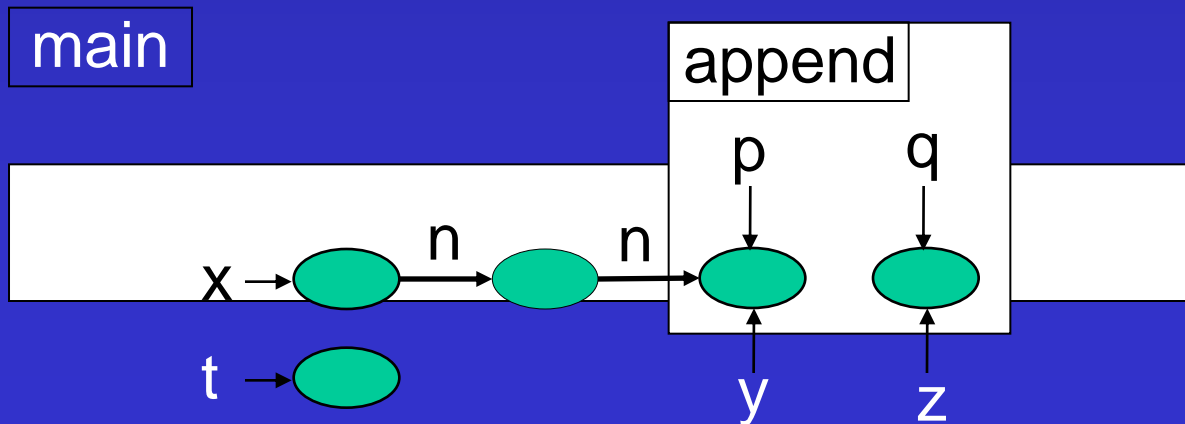


Programming model

- Single threaded
- Procedures
 - ✓ Value parameters
 - Formal parameters not modified
 - ✓ Recursion
- Heap
 - ✓ Recursive data structures
 - ✓ Destructive update
 - ✗ No explicit addressing (&)
 - ✗ No pointer arithmetic

Memory states

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

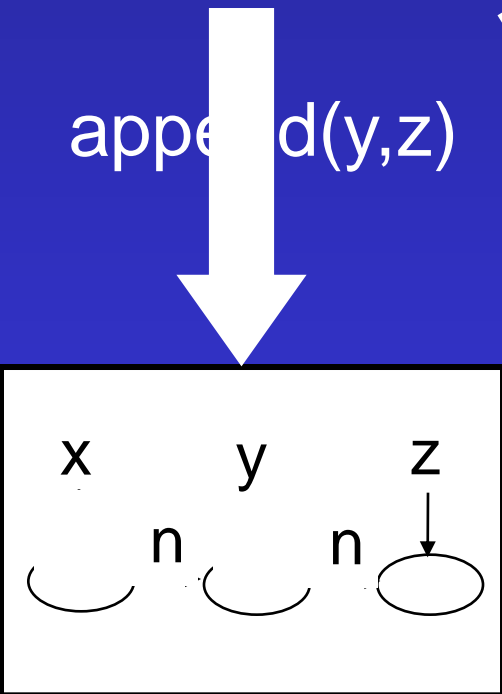
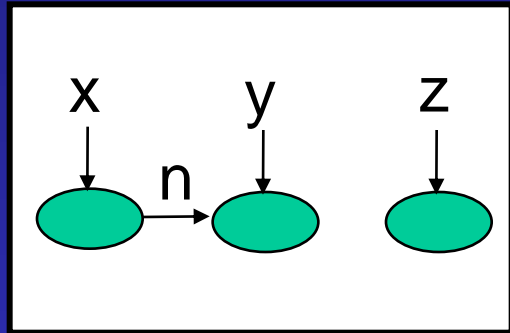


Abstract semantics

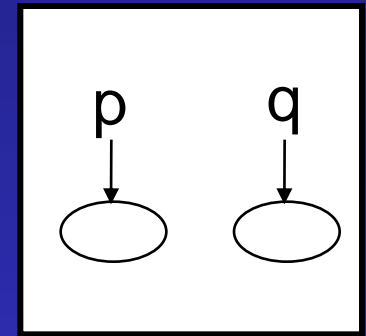
- Conservatively apply statements using 3-valued logic (with the non-standard semantics)
 - Use canonical abstraction
 - Reinterpret FO formulas using Kleene value

Procedure calls

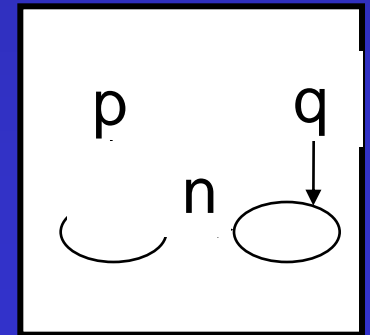
append(p,q)



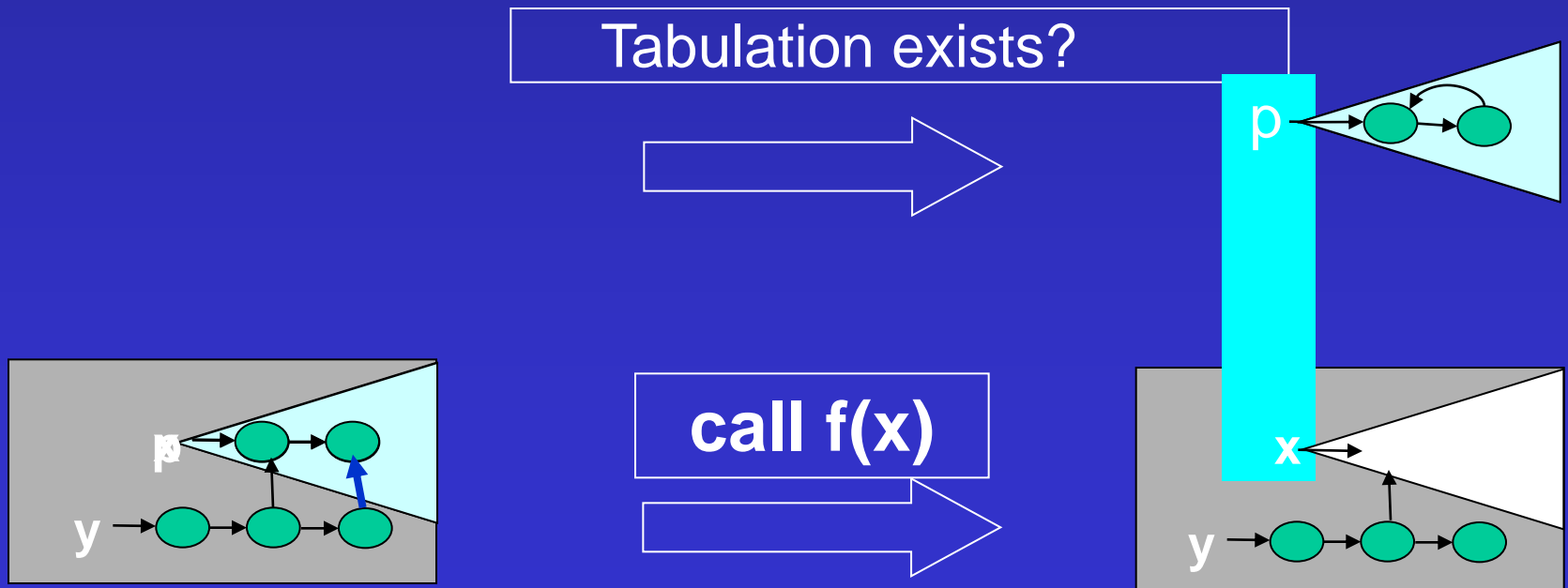
1. Verify cutpoint freedom
2. Compute input
- ... Execute callee ...
3. Combine output



append body

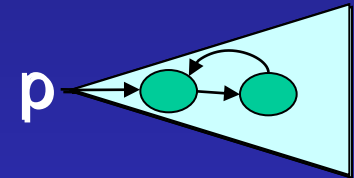


Interprocedural shape analysis

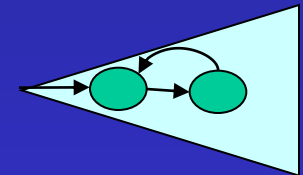
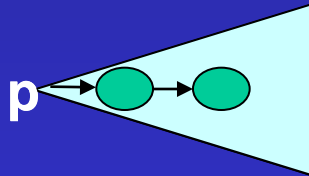


Interprocedural shape analysis

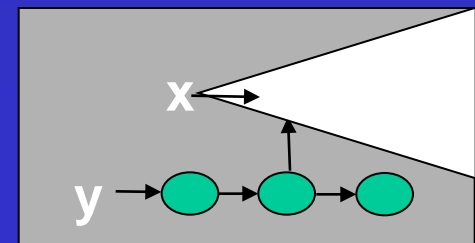
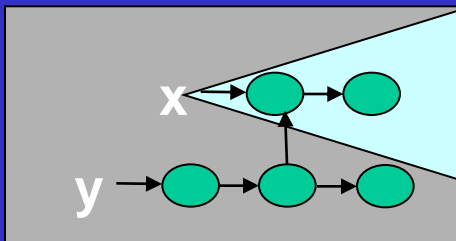
Analyze f



Tabulation exists?



call f(x)

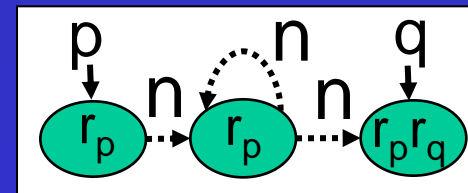
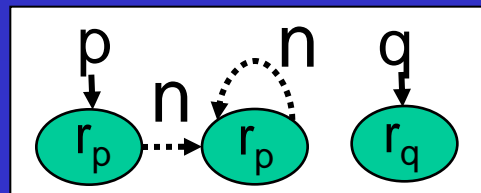
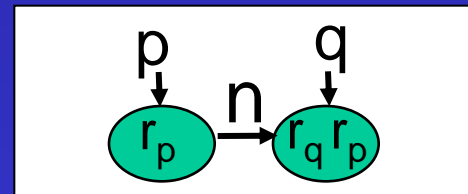
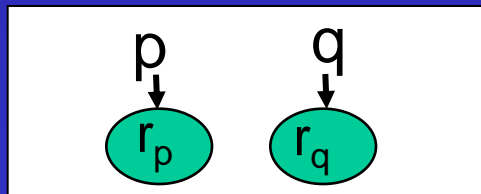
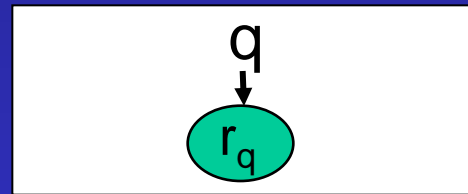
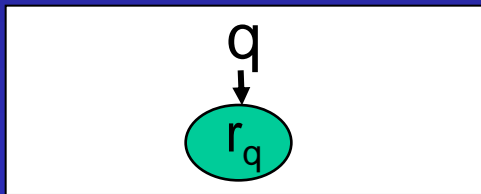


Interprocedural shape analysis

- Procedure \equiv input/output relation

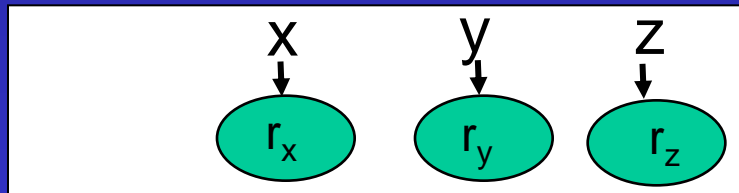
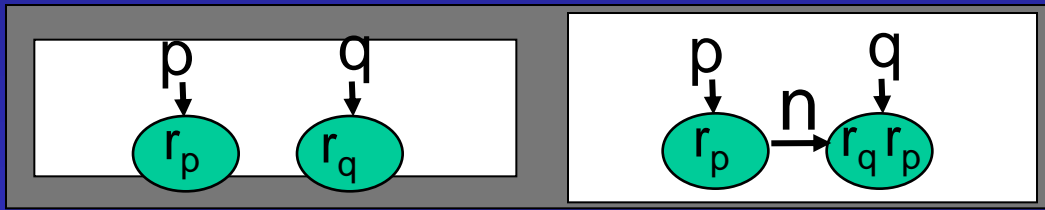
Input

Output

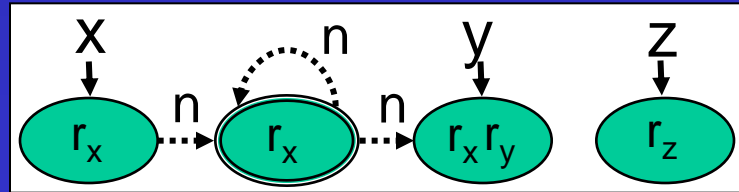
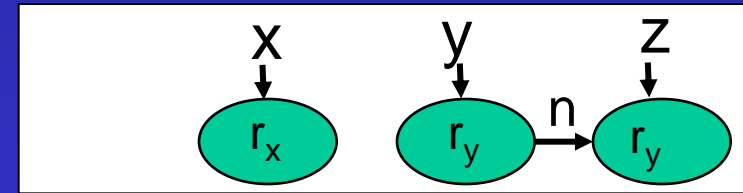


Interprocedural shape analysis

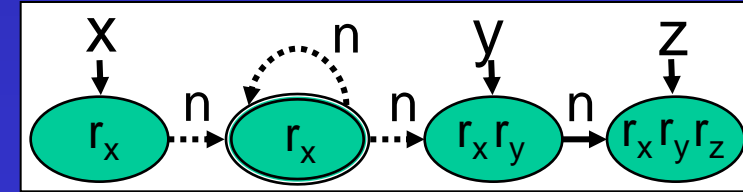
- Reusable procedure summaries
 - Heap modularity



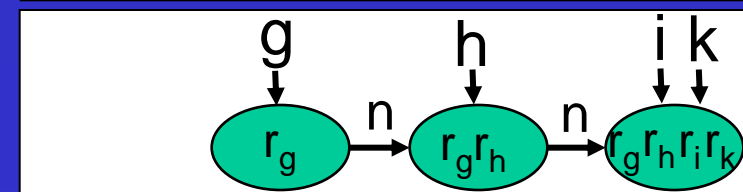
append(y,z)



append(y,z)



append(h,i)

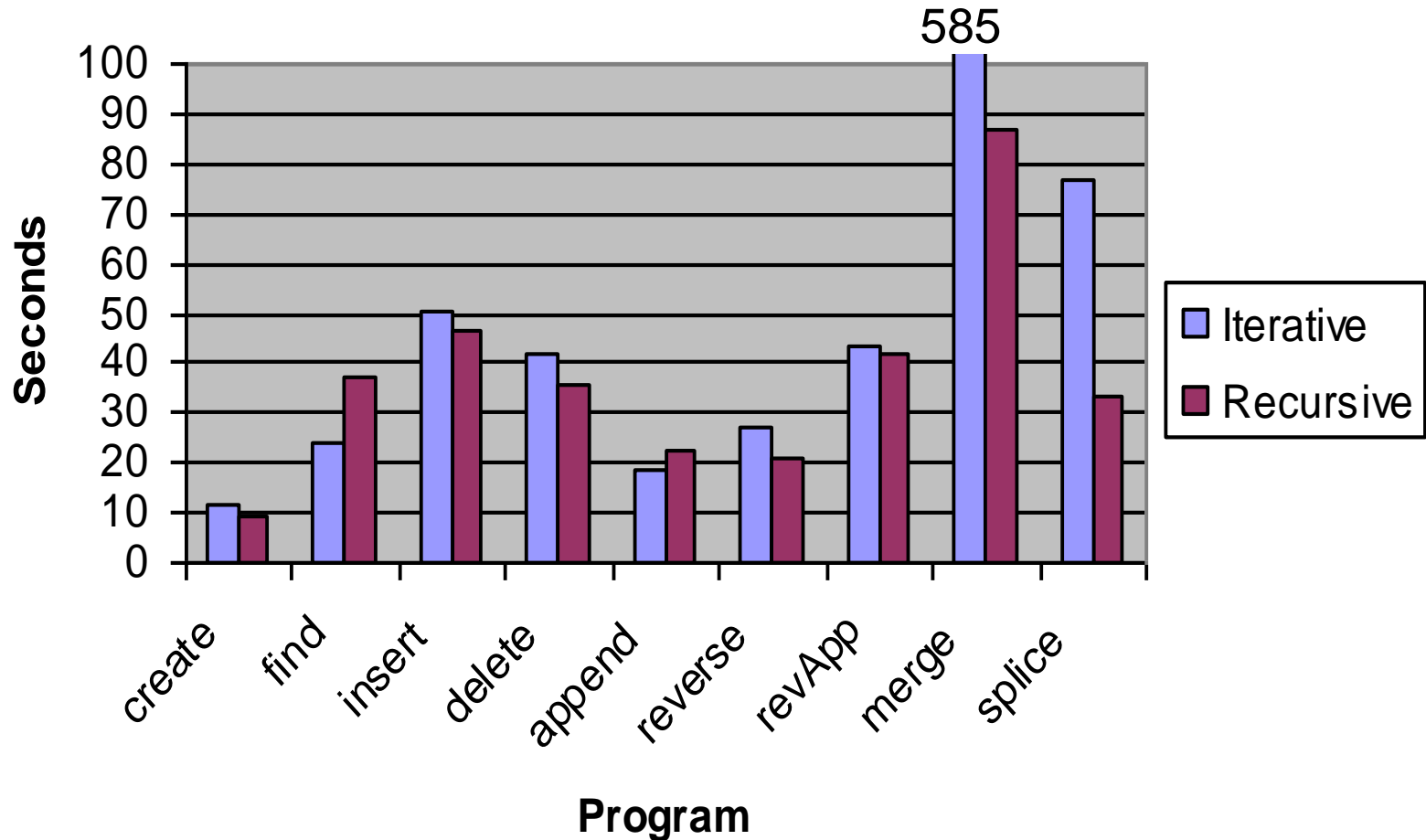


Prototype implementation

- 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	Cleanness, tree-ness

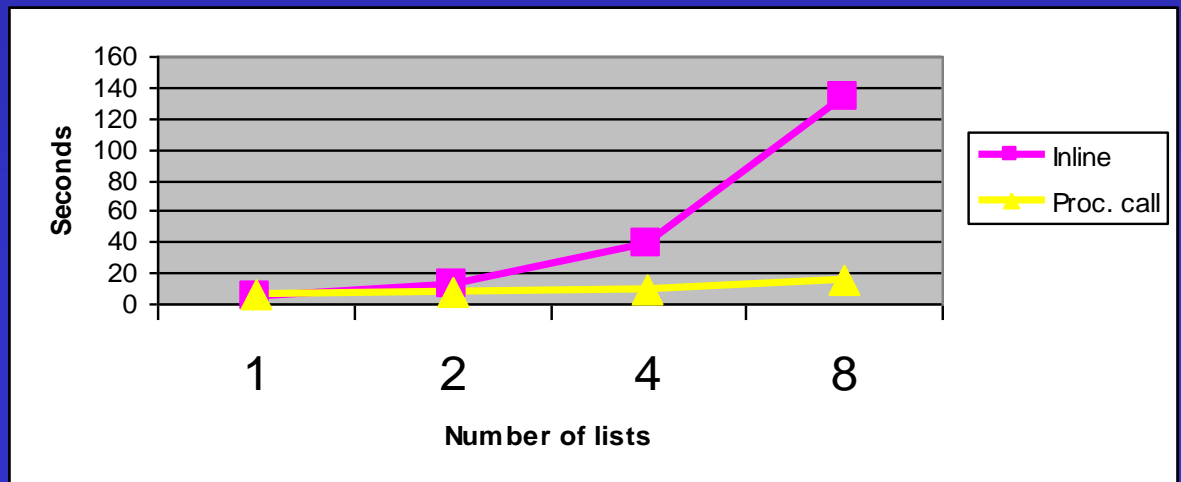
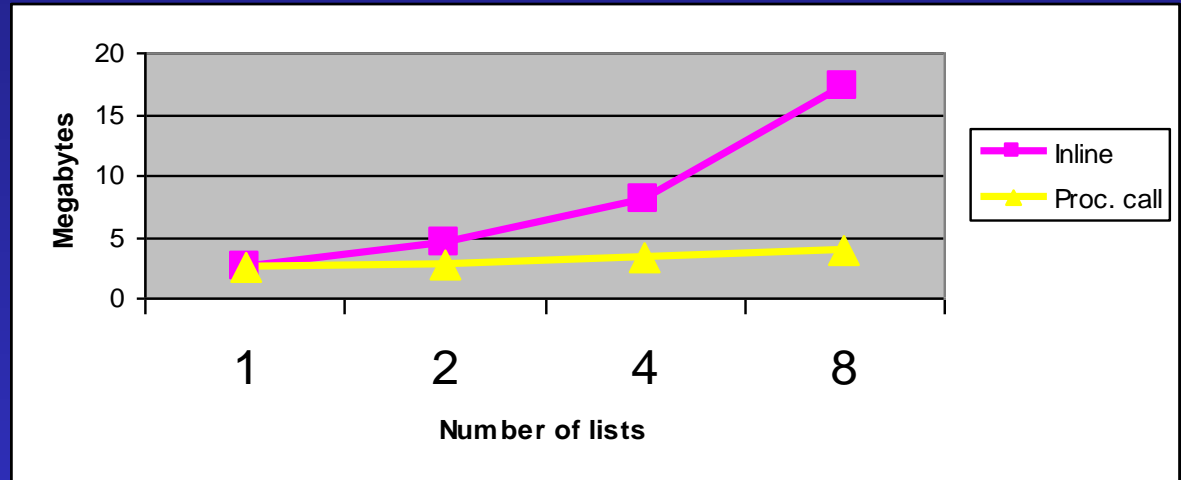
Iterative vs. Recursive (SLL)



Inline vs. Procedural abstraction

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

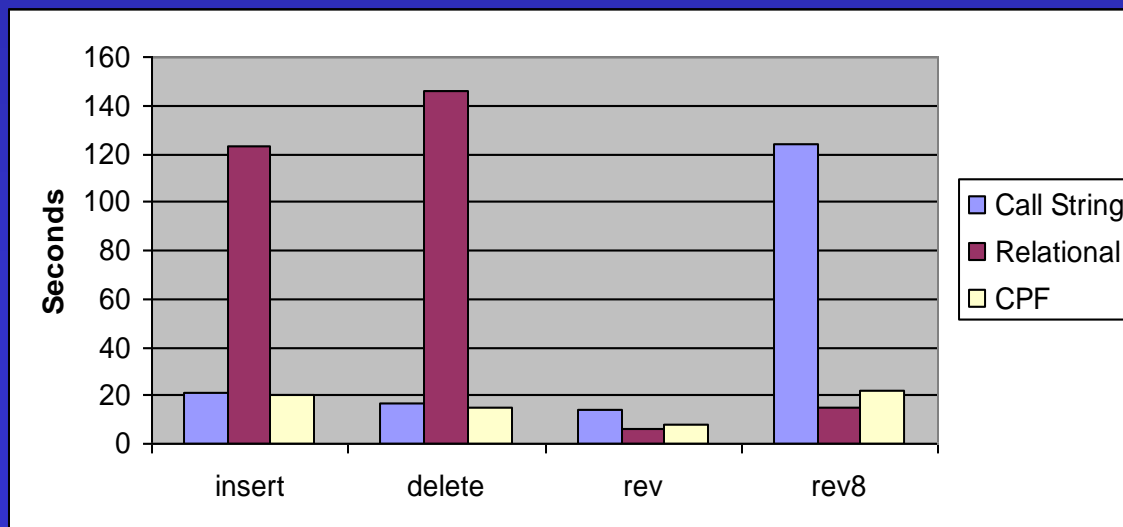
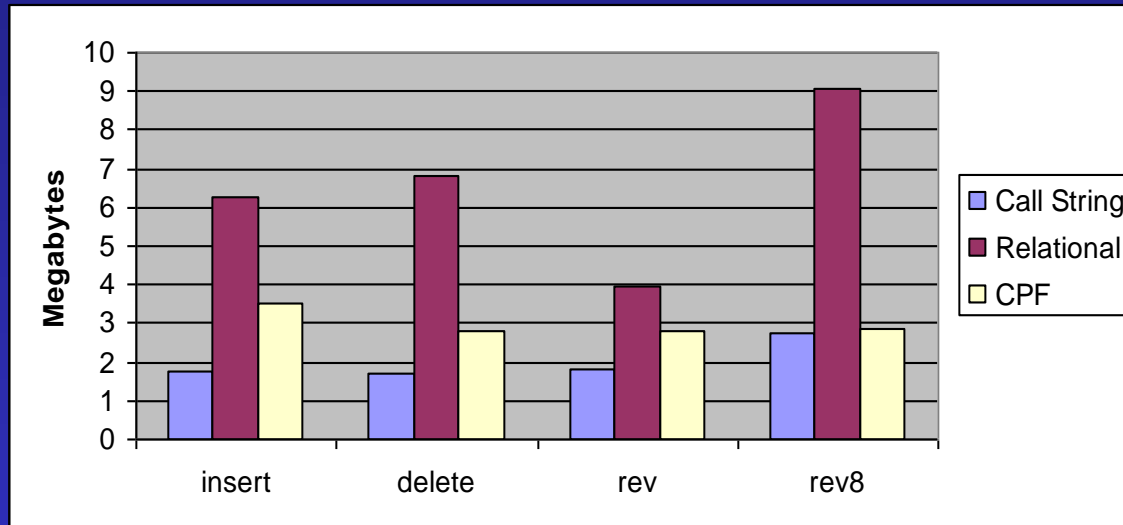
```
main() {  
    List x1 = create3();  
    List x2 = create3();  
    List x3 = create3();  
    List x4 = create3();  
    ...  
}
```



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

Summary

- Reasoning about the heap is challenging
- [Parametric] Abstraction is necessary
- Canonical abstraction is powerful
- Useful for programs with arrays [Gopan POPL'05]
- Information lost by canonical abstraction
 - Correlations between list lengths