TVLA: A system for inferring Quantified Invariants

Tal Lev-Ami

Tom Reps

Mooly Sagiv

Reinhard Wilhelm

Greta Yorsh

Example: 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)
       )
}
```

$\forall v\colon \text{marked}(v) \Leftrightarrow \exists r\colon \text{root}(r) \wedge \text{reach}(w, v)$

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}
    }
  }
}
```

$\exists r: root(r) \wedge (pending(r) \vee marked(r)) \wedge$

$\forall v: ((marked(v) \vee pending(v)) \rightarrow reach(r, v))$

$\wedge$

$\neg(pending(v) \wedge marked(v))$

$\wedge$

$\forall v, w: ((marked(v) \wedge \neg marked(w) \wedge \neg pending(w)) \rightarrow \neg successor(v, w))$

$\forall v: marked(v) \Leftrightarrow \exists r: root(r) \wedge reach(r, v)$

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))
}
```

Run Demo

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))
}
```

$\exists$r: root(r) $\wedge$ reach(r, e) $\wedge$ $\neg$ pending(r) $\wedge$ marked(r)

$\exists$e: reach(r, e) $\wedge$ $\neg$marked(e)) $\wedge$ $\neg$root(e) $\wedge$ $\neg$ pending(v)

$\forall$r, e:

((reach(r, e) $\wedge$ $\neg$marked(e)) $\wedge$ $\neg$root(e) $\wedge$ $\neg$ pending

$\wedge$

(root(r) $\wedge$ reach(r, e) $\wedge$ $\neg$ pending(r) $\wedge$ marked(r))

$\rightarrow$ $\neg$left(e, r))

A Singleton Buffer

```
Boolean empty = true;
Object b = null;
```

```
produce() {
1: Object p = new();
2: await (empty) then {
        b = p;
        empty = false;
   }
3:
}
```

```
consume() {
Object c;
4: await (!empty) then {
        c = b;
        empty = true;
   }
5: use(c);
6: dispose(c);
7:
}
```

Safe Dereference

No Double free

$$\forall t, e, v: t \neq e \wedge c(t, v) \wedge c(e, w) \rightarrow v \neq w$$

```
Boolean empty = true;
Object b = null;
```

```
produce() {
1: Object p = new();
2: await (empty) then {
       b = p;
       empty = false;
   }
3:
}
```

```
consume() {
Object c;
4: await (!empty) then {
       c = b;
       empty = true;
   }
5: use(c);
6: dispose(c);
7:
}
```

Quantified Invariants are hard

- Corner cases
- Sizes
- Nested loops
- Code updates
- First order reasoning is hard

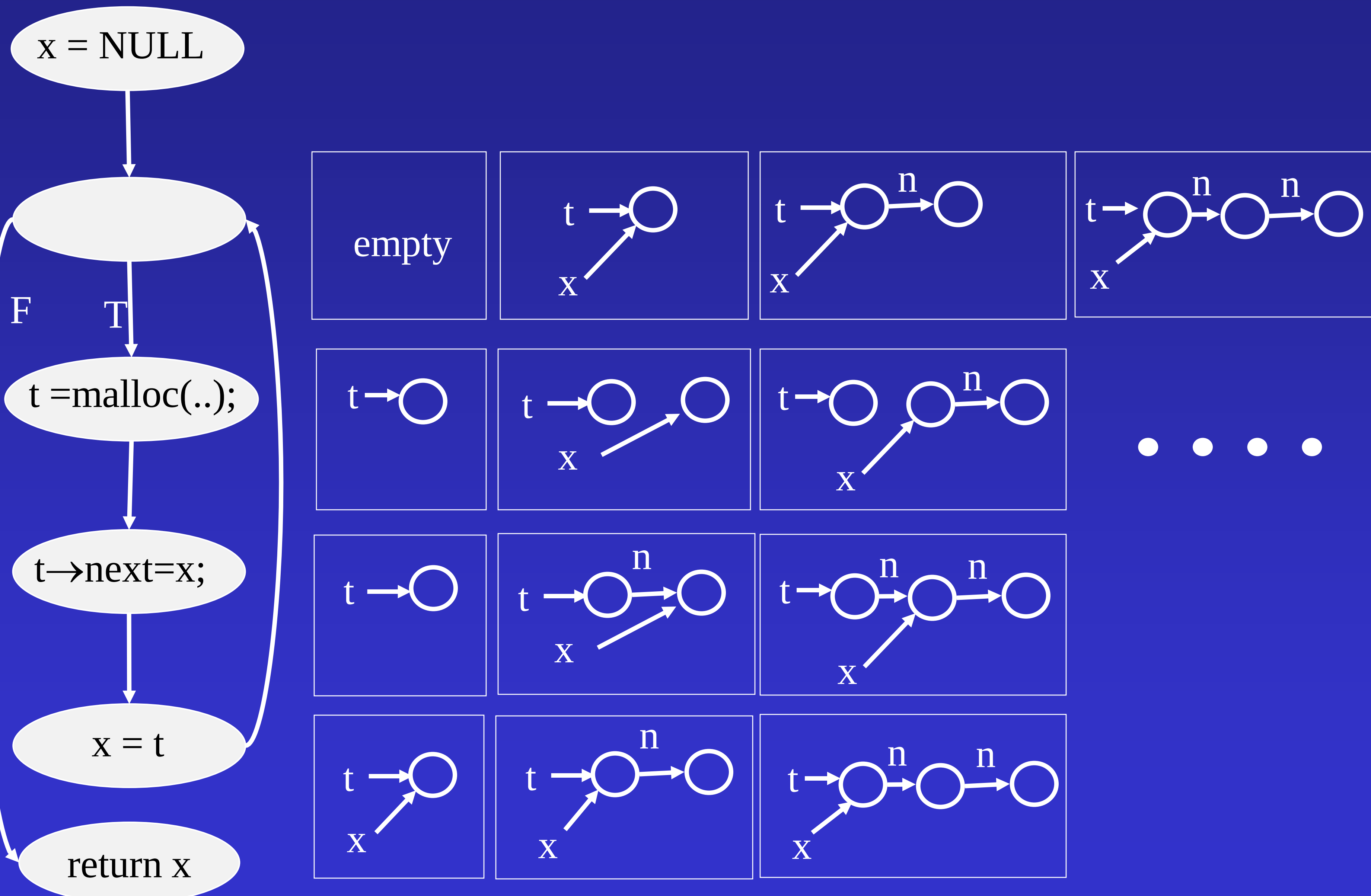
Our approach

- Automatically infer sound invariants
- Library (PL) designers can define families of interesting invariants
- Limited form of invariants
- Sound but incomplete reasoning
  - Abstract interpretation over quantified invariants
  - Parameterized by the

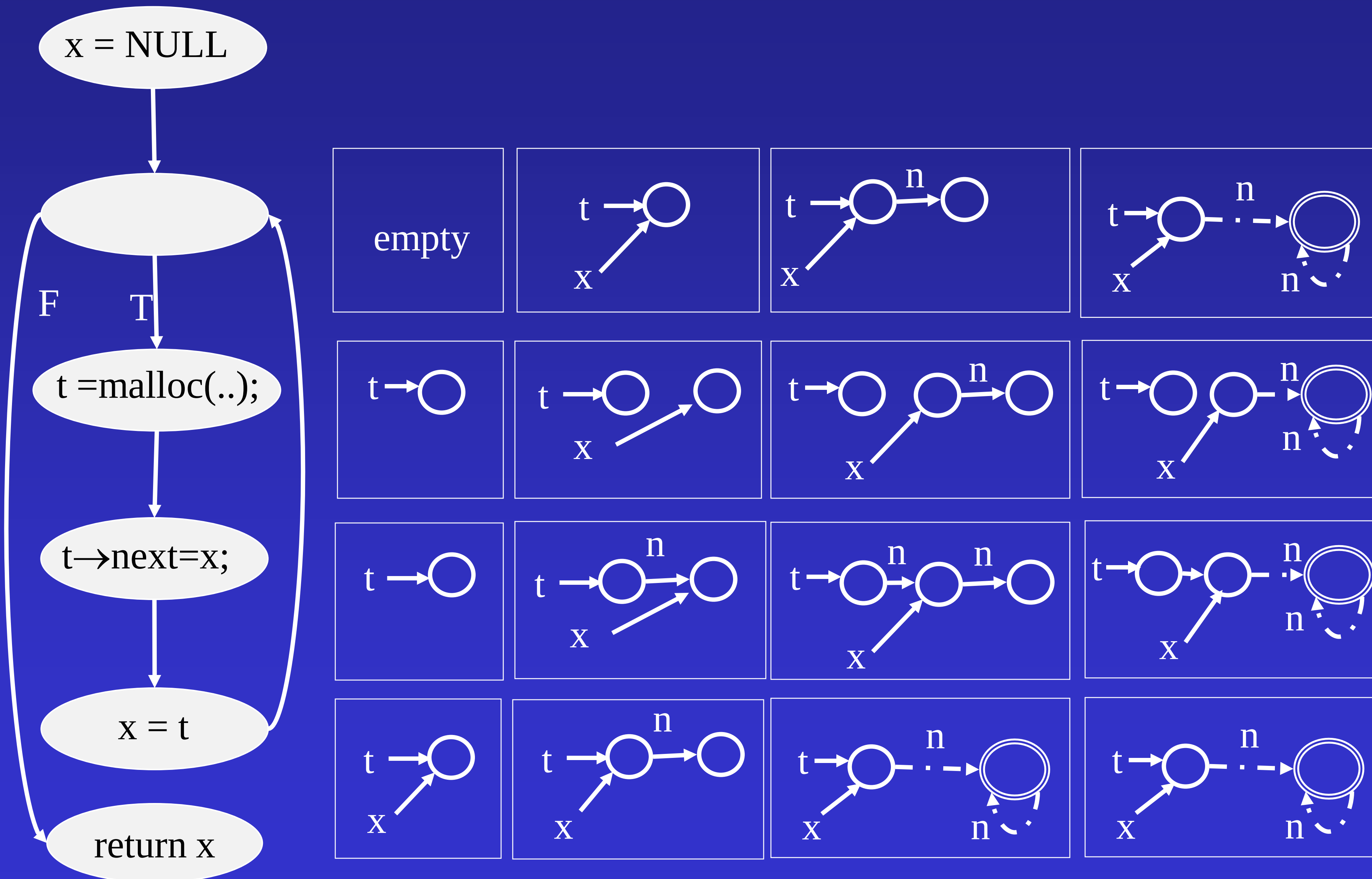
Applications

- Memory safety & preservation of data structure invariants [Dor, SAS'00, Loginov, ISSTA'08]
- Compile-time garbage collection [Shaham, SAS'03]
- Correct API usage [Ramalingam, PLDI'02, Yahav. PLDI'04]
- Typestate verification [Yahav, ISSTA'06]
- Partial & total correctness
  - Sorting implementations [Lev-Ami, ISTTA'00, Rinetzky, SAS'05]
  - Deutsch-Shorr-Waite [Loginov, SAS'06]
- Thread modular shape analysis [Gotsman, PLDI'07]
- Linearizability [Amit, CAV'07, Berdine, CAV'08]

Example: Concrete Interpretation



Shape Analysis



Outline

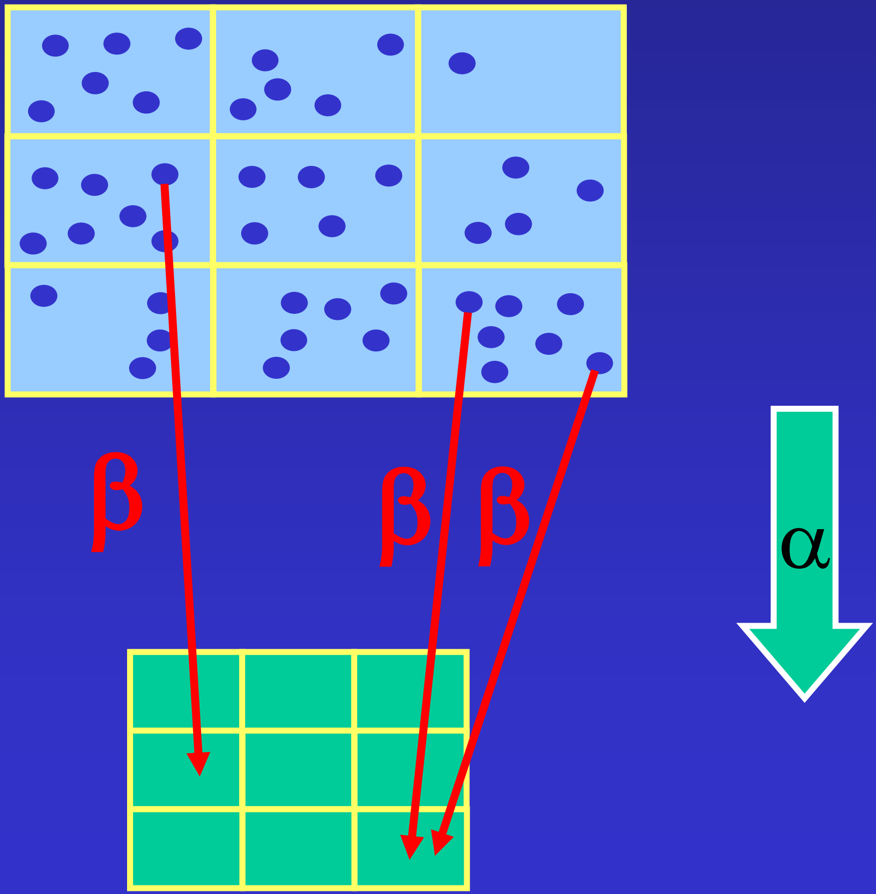
- Canonical Abstraction [TOPLAS’02]
- Quantified Invariants
- Operating on Abstractions

Abstract Interpretation [Cousot & Cousot]

- Checking interesting program properties is undecidable
- Use abstractions
- Every verified property holds (sound)
- But may fail to prove properties which always hold (incomplete)
  - false alarms
- Minimal false alarms

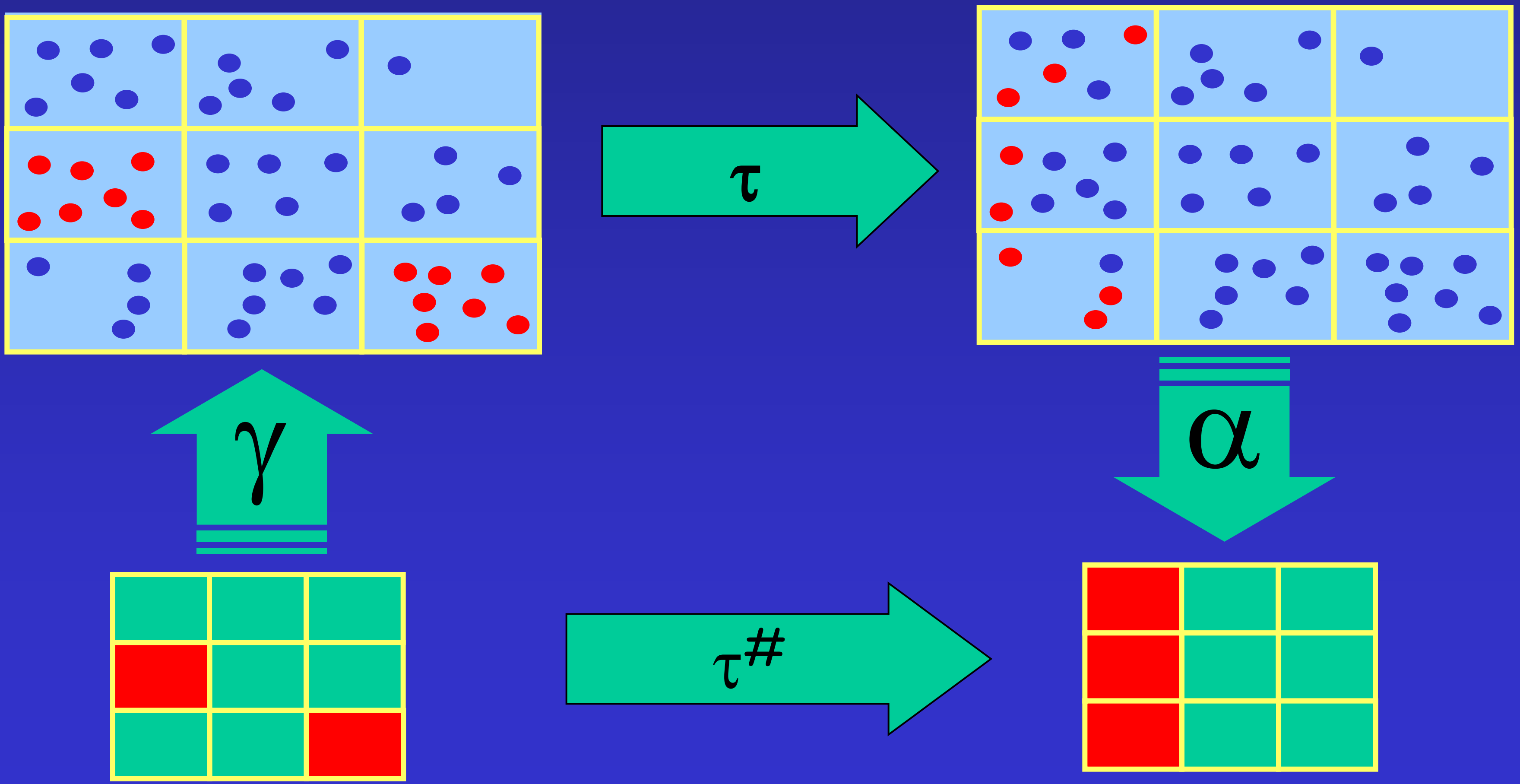
Simplified Abstract Interpretation

Concrete domain
(unbounded)



Abstract domain
(bounded)

Most Precise Abstract Transformer [Cousot, Cousot POPL 1979]

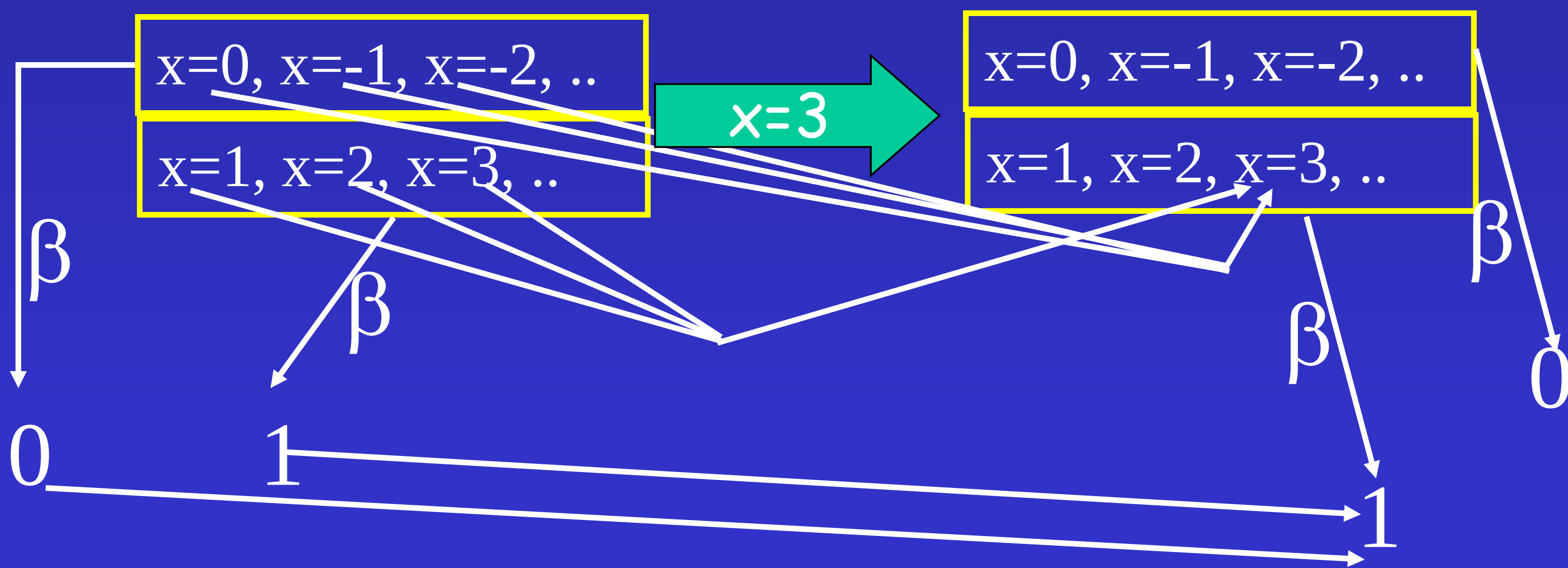


An example predicate abstraction

```
x = 3
while (true) {
   x = x +1 ;
 }
```

predicates

$\{x>0\}$

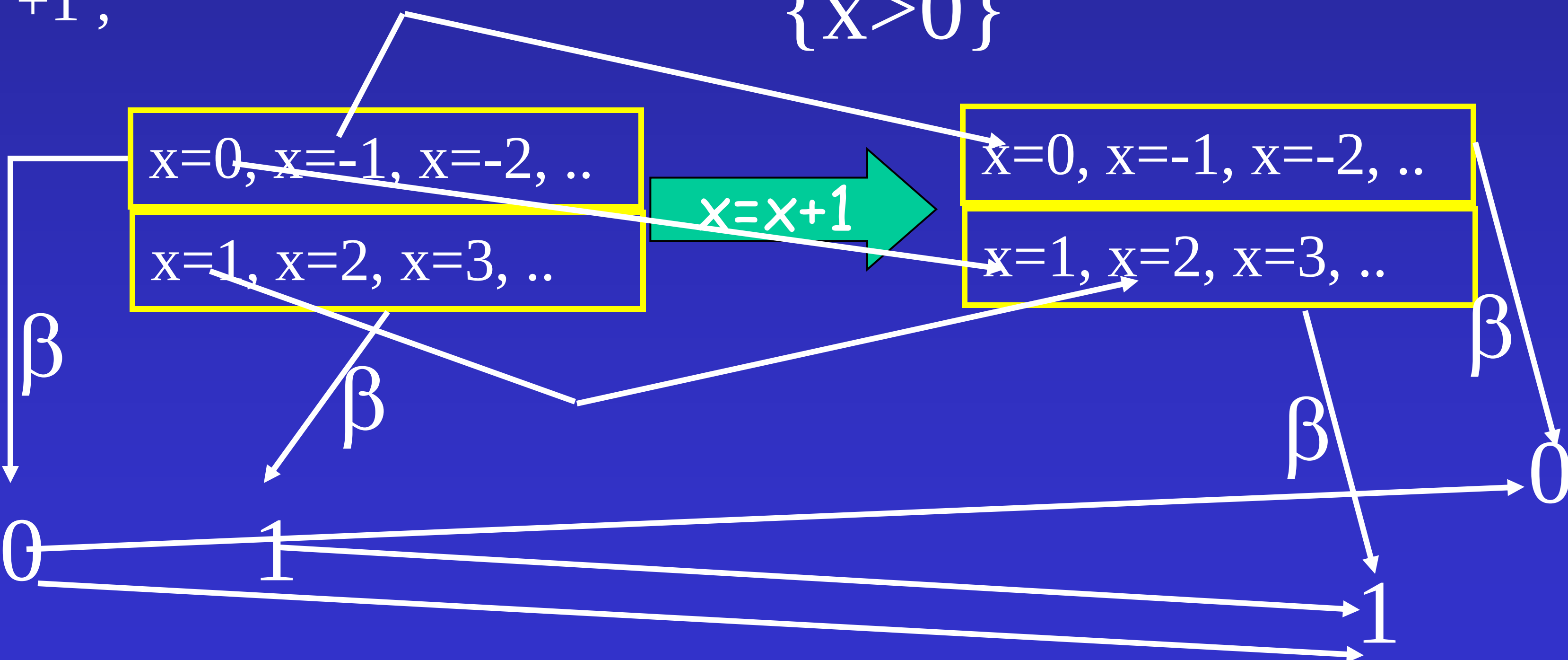


An example predicate abstraction

```
x = 3
while (true) {
   x = x +1 ;
 }
```

predicates

{x>0}



An example predicate abstraction

predicates

{x>0}

{0, 1}

x = 3

{1}

while (true) { {1}

{1} x = x +1 ; {1}

}

Shape Analysis as Abstract Interpretation

- Represent concrete stores as labeled directed graphs
  - 2-valued structures {0, 1}
  - Abstract away
    - Concrete locations
    - Primitive values
  - But unbounded
- Represent abstract stores as labeled directed graphs
  - 3-vaued structures {0, 1, ½}
  - Several concrete nodes are represented by a summary node
  - Abstract away field correlations

Representing Concrete Stores by Logical Structures

- Parametric vocabulary
- Heap
  - Locations ≈ Individuals
  - Program variables ≈ Unary relations
  - Fields ≈ Binary relations

Representing Stores as Logical Structures (Example)

x

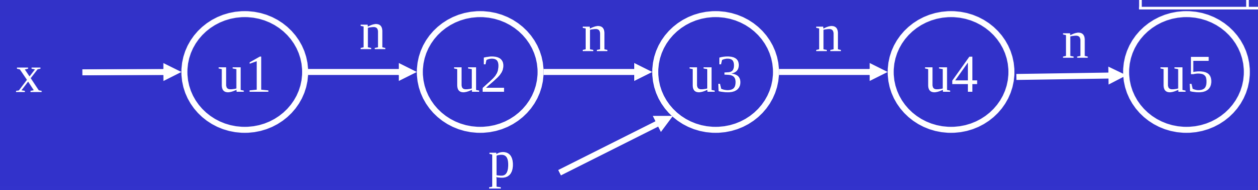
u_1	1
u_2	0
u_3	0
u_4	0
u_5	0

p

u_1	0
u_2	0
u_3	1
u_4	0
u_5	0

n

	u_1	u_2	u_3	u_4	u_5
u_1	0	1	0	0	0
u_2	0	0	1	0	0
u_3	0	0	0	1	0
u_4	0	0	0	0	1
u_5	0	0	0	0	0



Representing Abstract Stores by 3-Valued Logical Structures

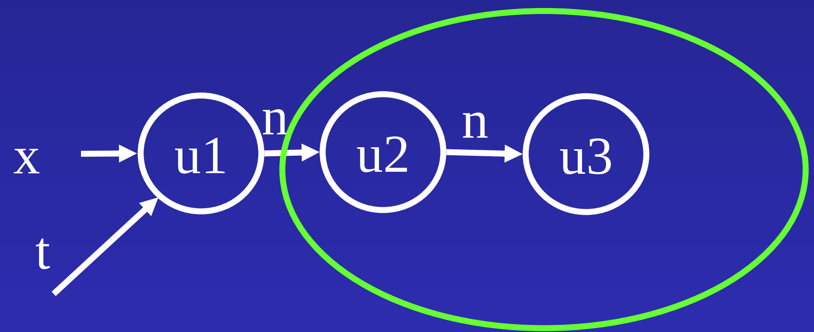
- A join semi-lattice: $0 \sqcup 1 = 1/2$
- $\{0, 1, \frac{1}{2}\}$ values for relations

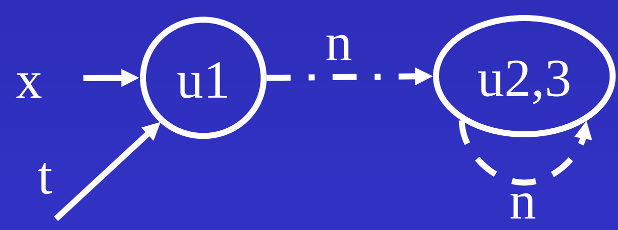
Canonical Abstraction (β)

- Partition the individuals into equivalence classes based on the values of their unary relations
  - Every individual is mapped into its equivalence class
- Collapse binary relations via ⊔
  - $p^{S}(u'_1, u'_2) = \sqcup \{p^{B}(u_1, u_2) \mid f(u_1)=\mathrm{u}'_1,\ f(u_2)=u'_2)\}$
- At most 2^{A} abstract individuals

Canonical Abstraction

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





Canonical Abstraction

x = NULL;

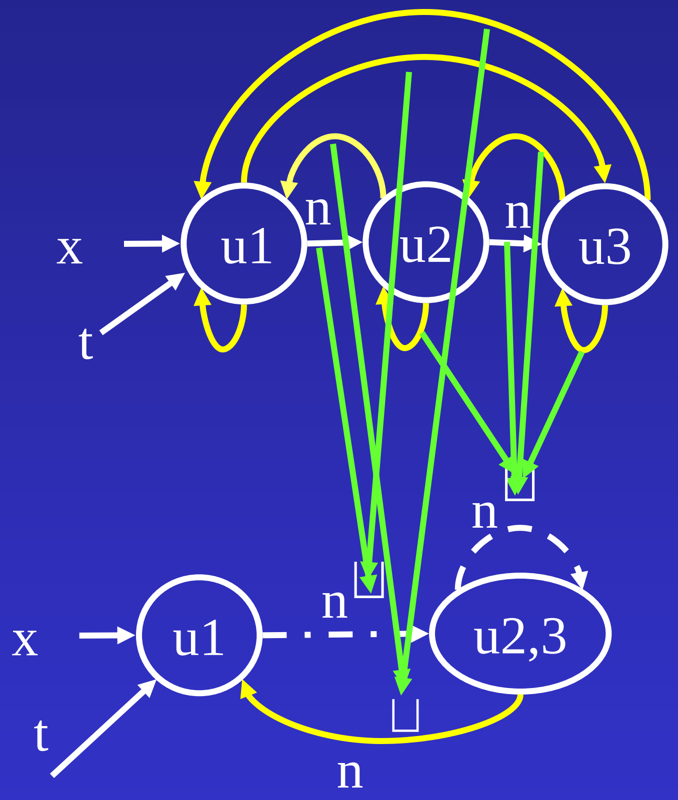
while (…) do {

t = malloc();

t →next=x;

x = t

}

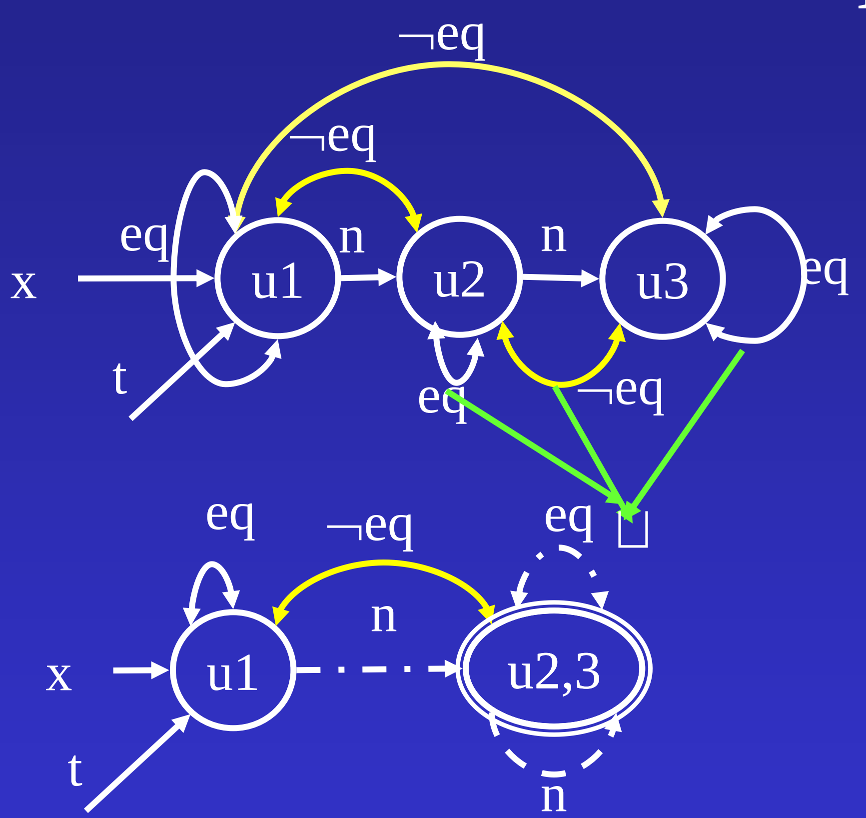


Canonical Abstraction and Equality

- 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

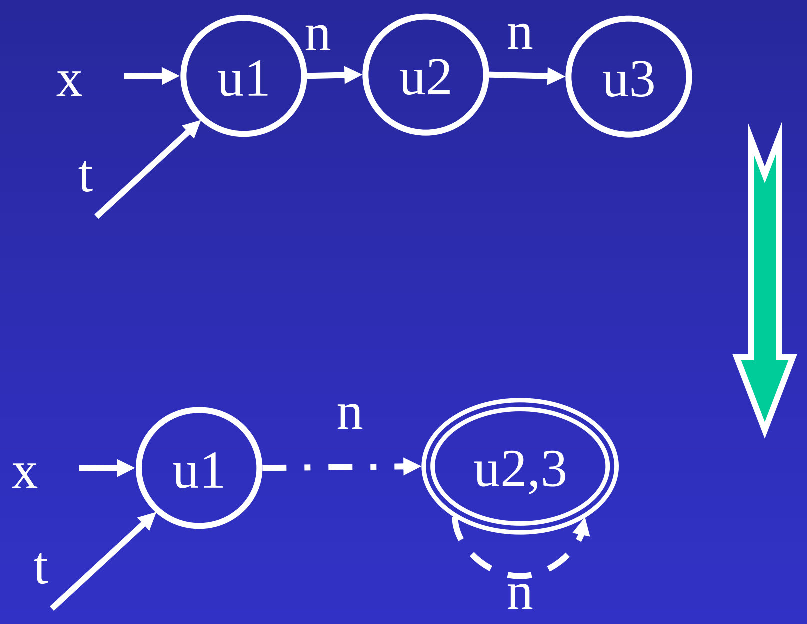
Canonical Abstraction and Equality

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



Canonical Abstraction

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

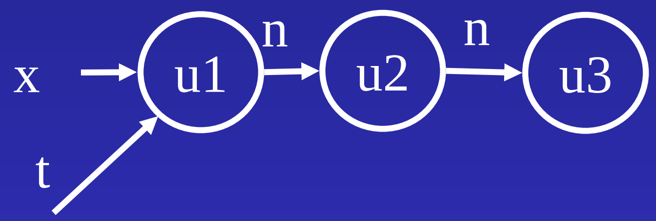


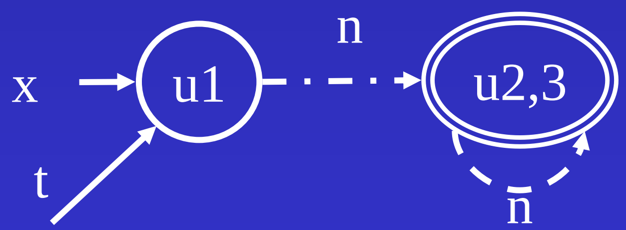
Canonical Abstraction

- 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 $\sqcup$
  - $p^{S}(u'_1, ..., u'_k) = \sqcup \{p^{B}(u_1, ..., u_k) \mid f(u_1)=u'_1, ..., f(u'_k)=u_k\}$
- At most 2^{A} abstract individuals

Canonical Abstraction

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





Limitations

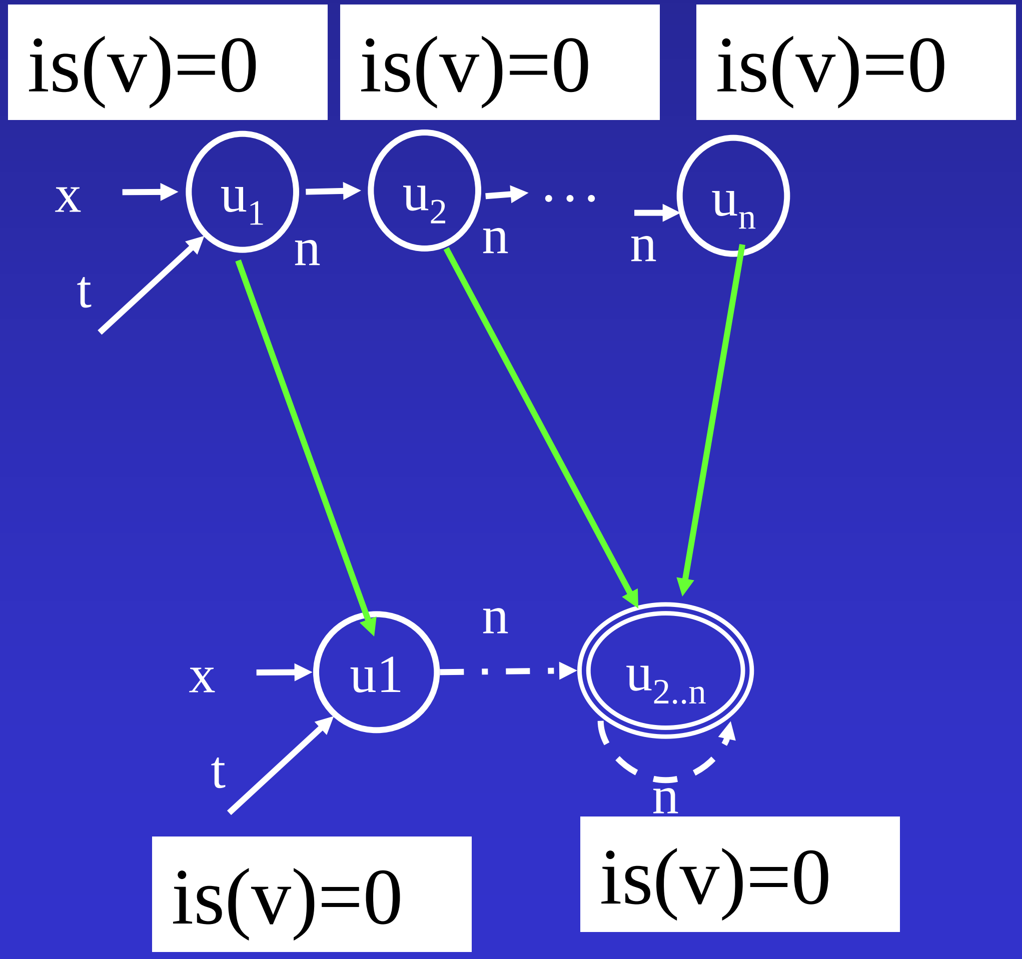
- Information on summary nodes is lost

Increasing Precision

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

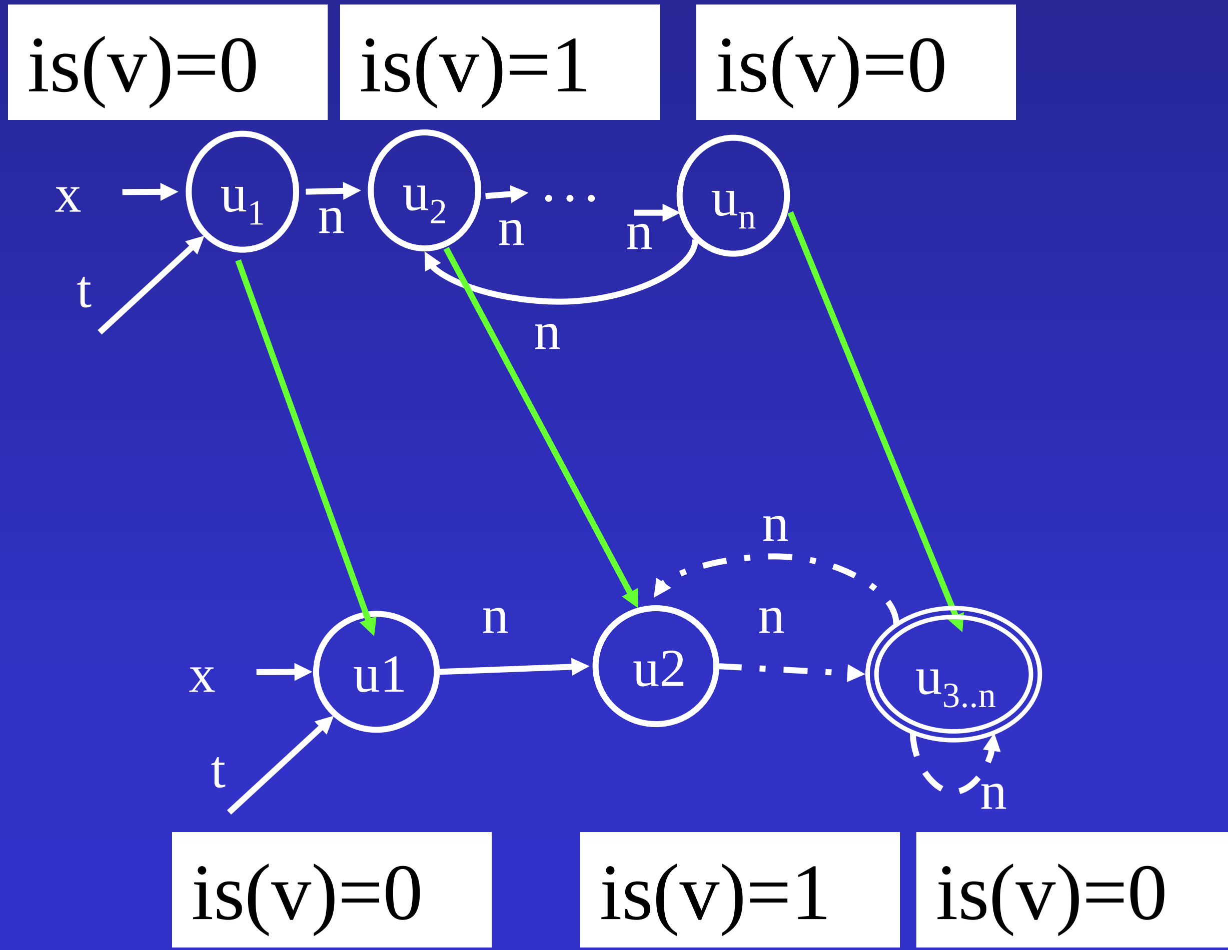
Heap Sharing relation

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



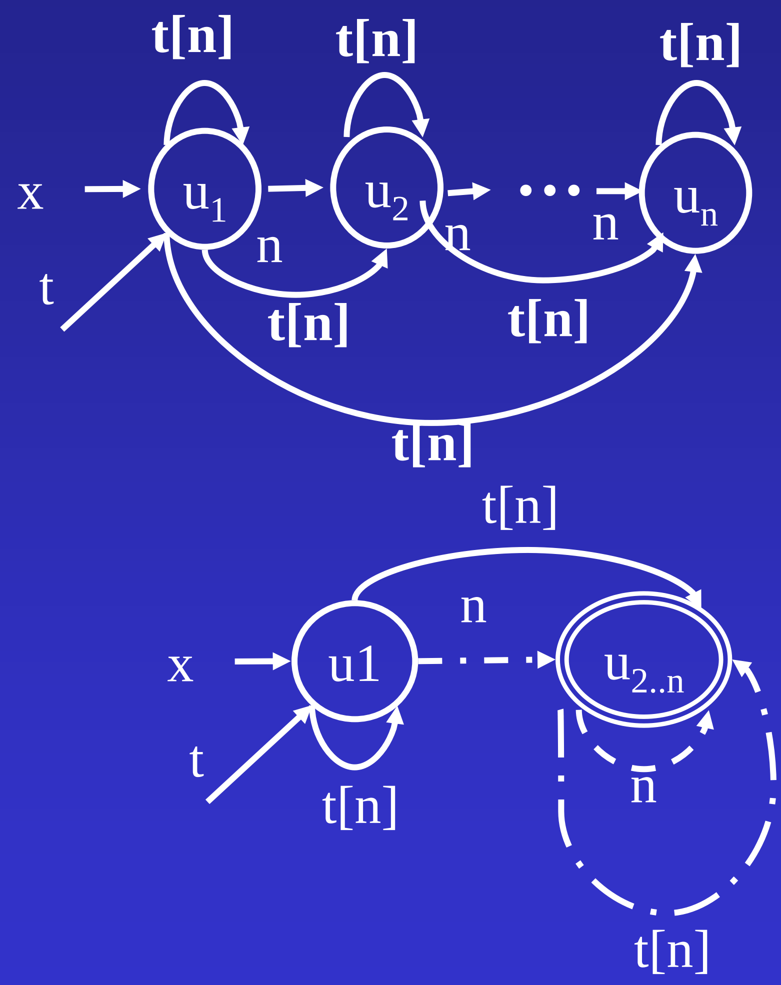
Heap Sharing relation

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

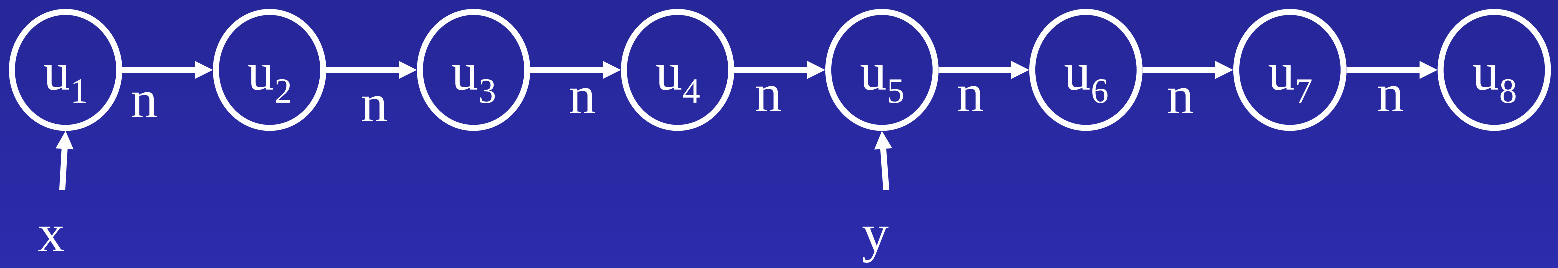


Reachability relation

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

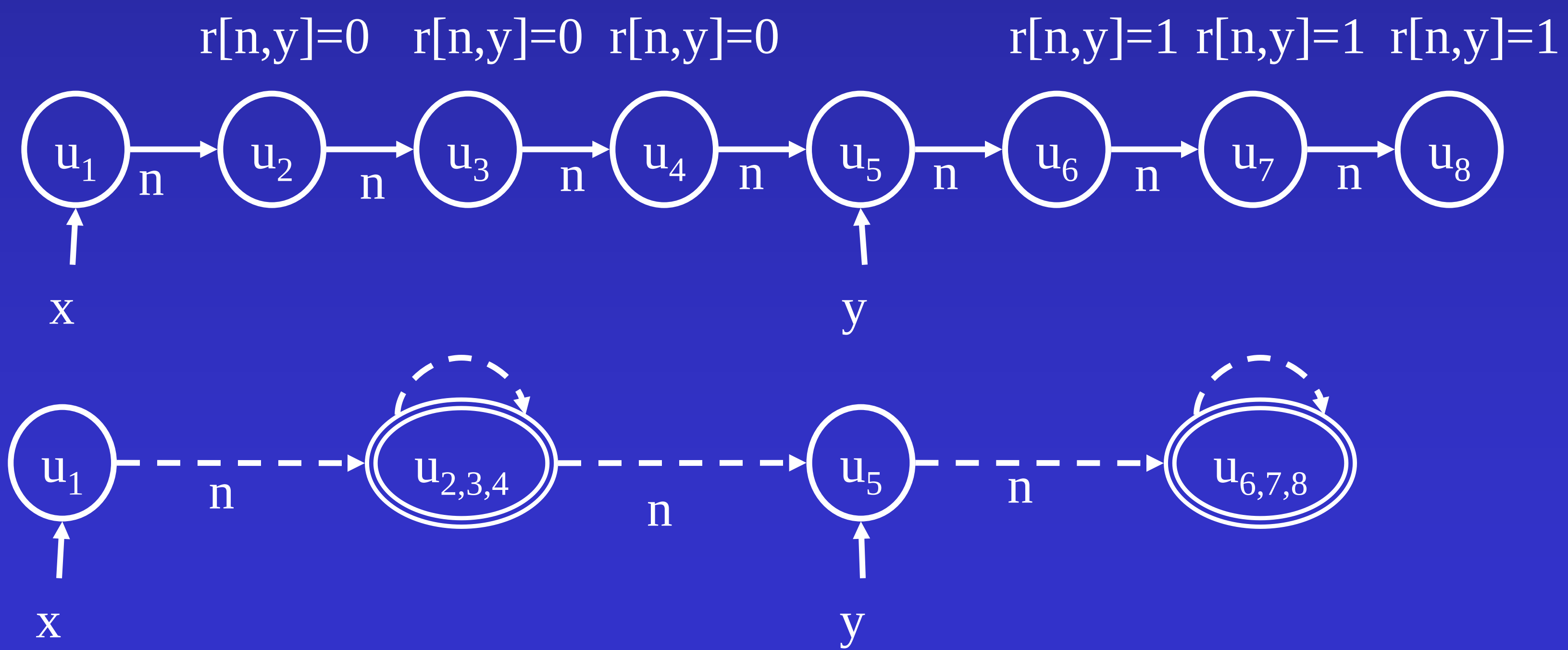


List Segments



Reachability from a variable

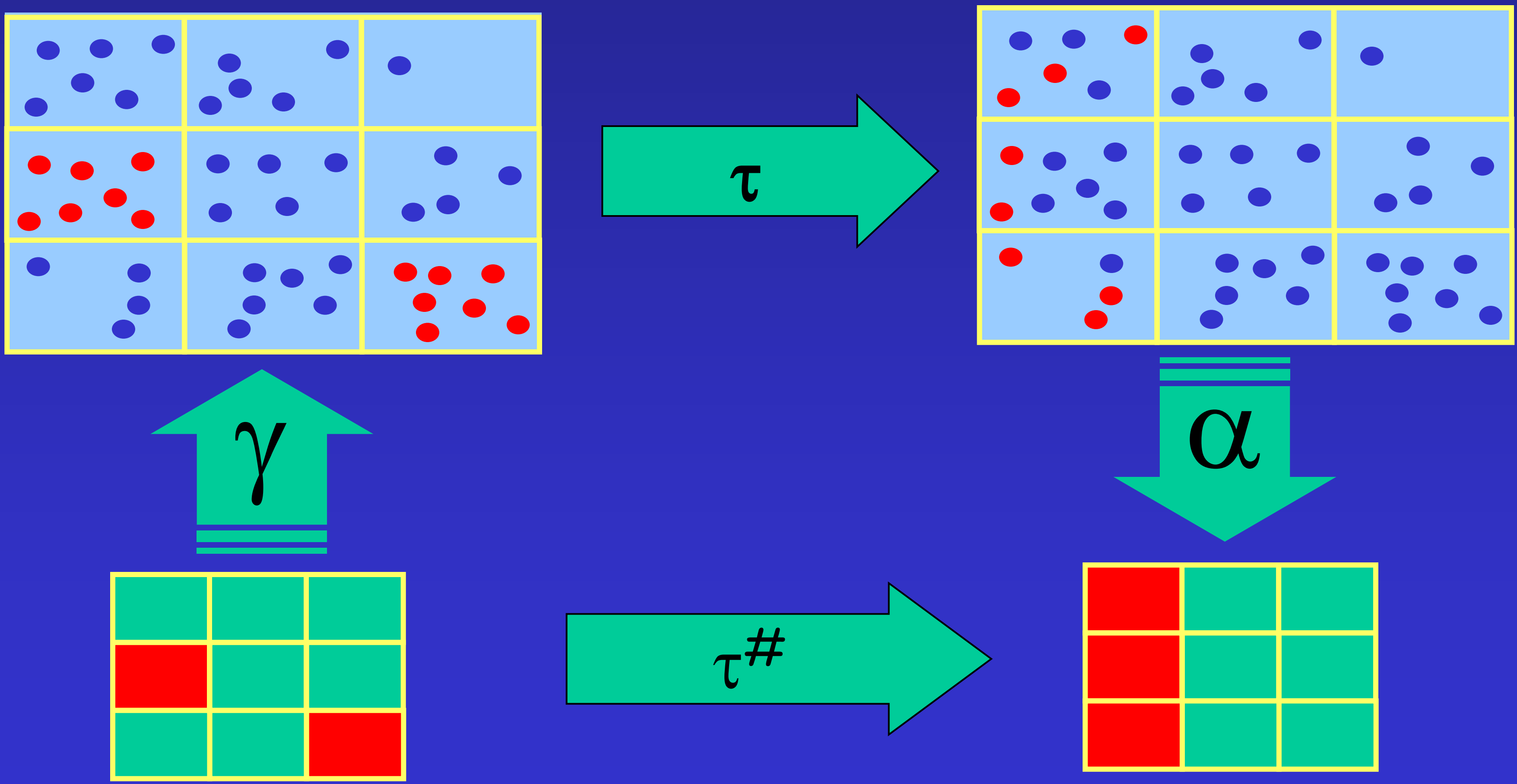
- $r[n,y](v) = \exists w\colon y(w) \wedge n^*(w, v)$



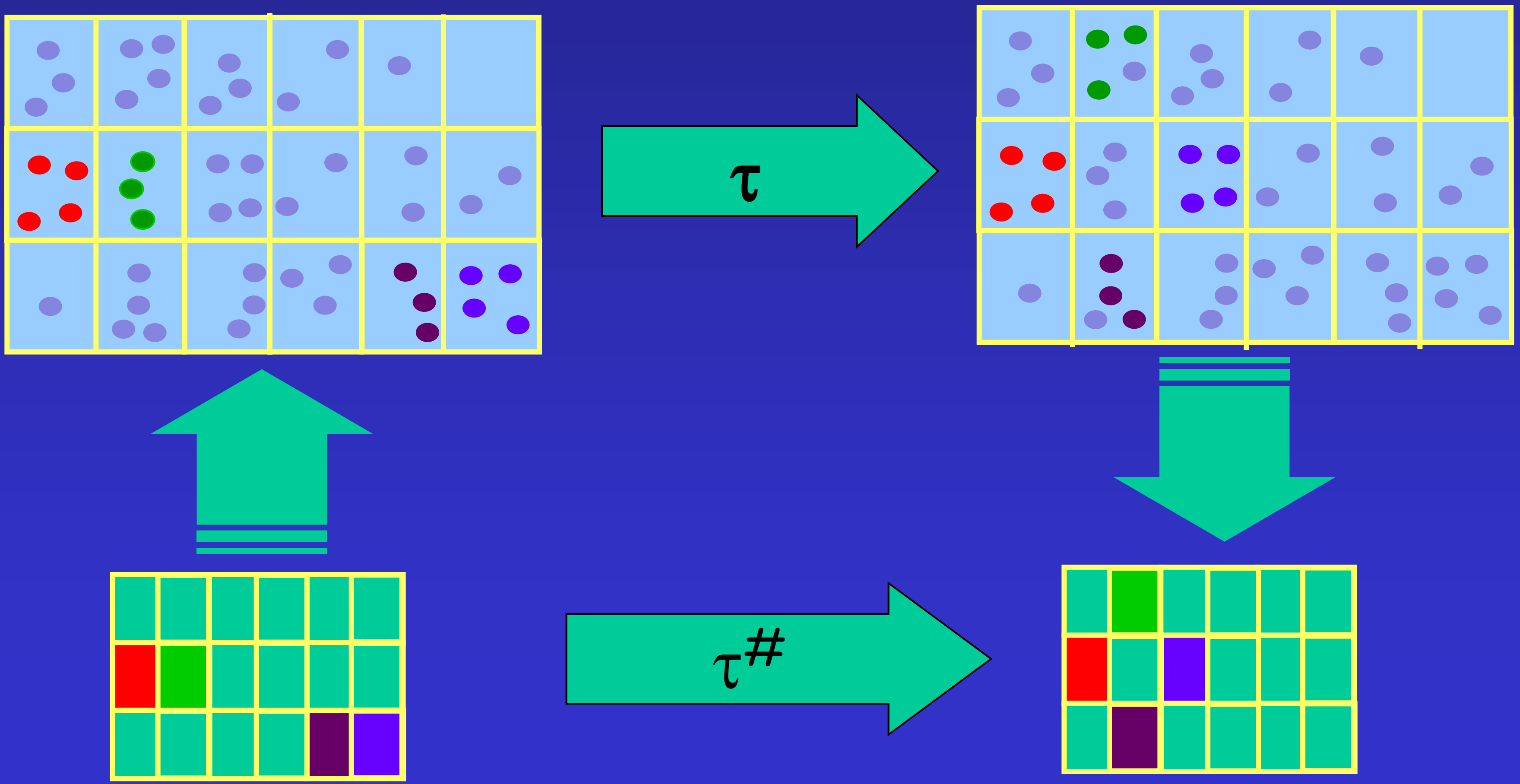
Additional Instrumentation relations

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

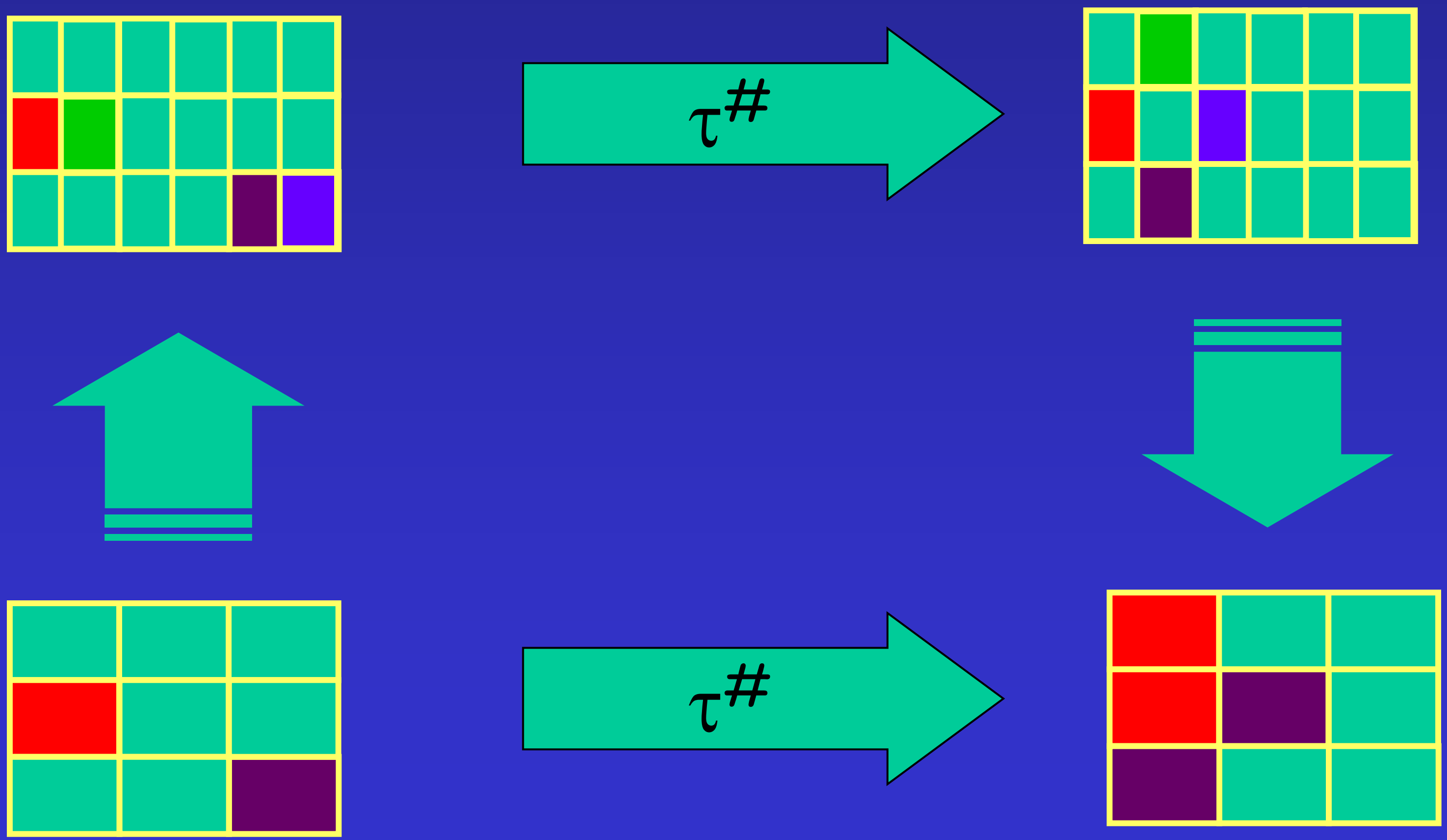
Most Precise Abstract Transformer [Cousot, Cousot POPL 1979]



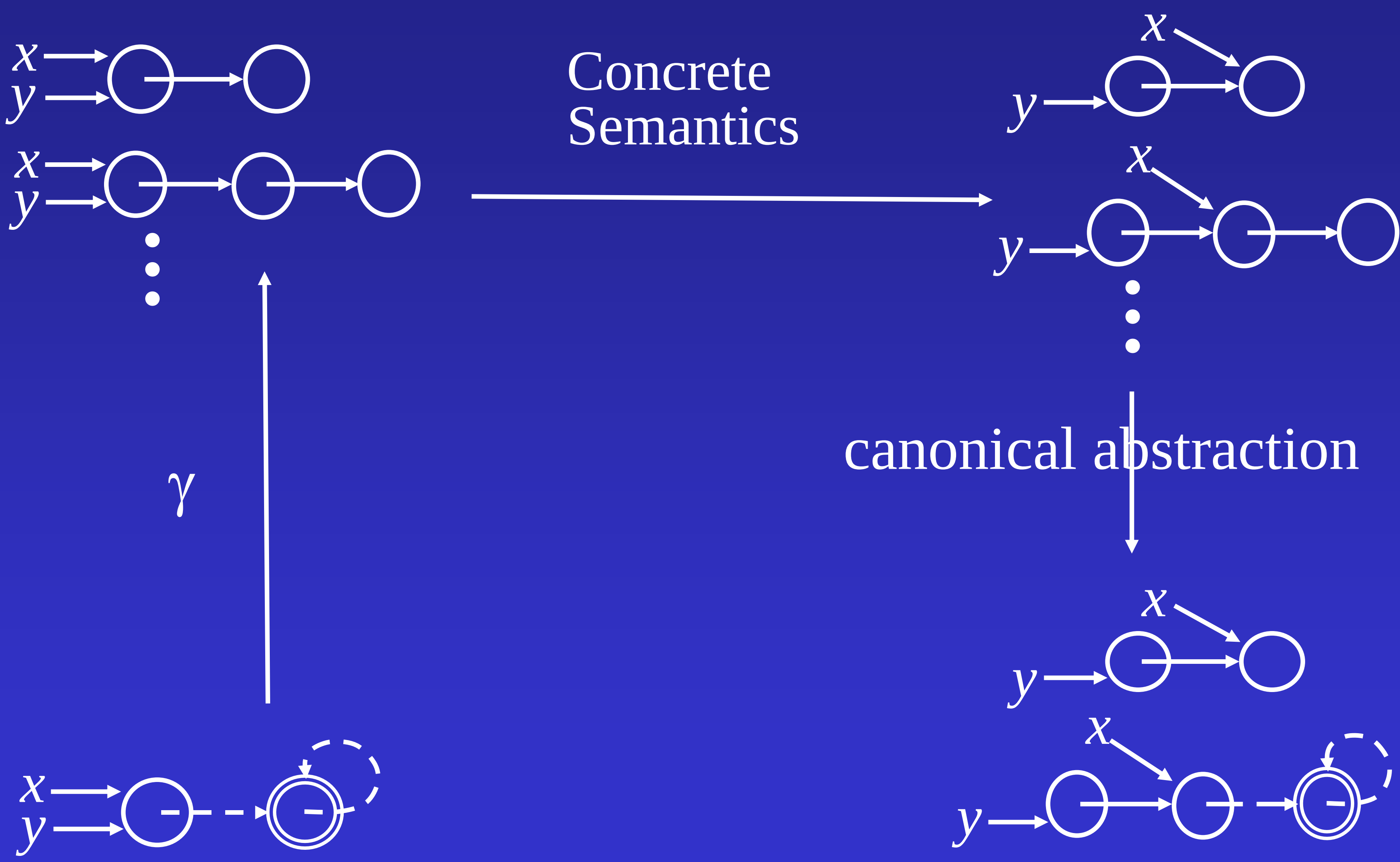
Partial Concretization



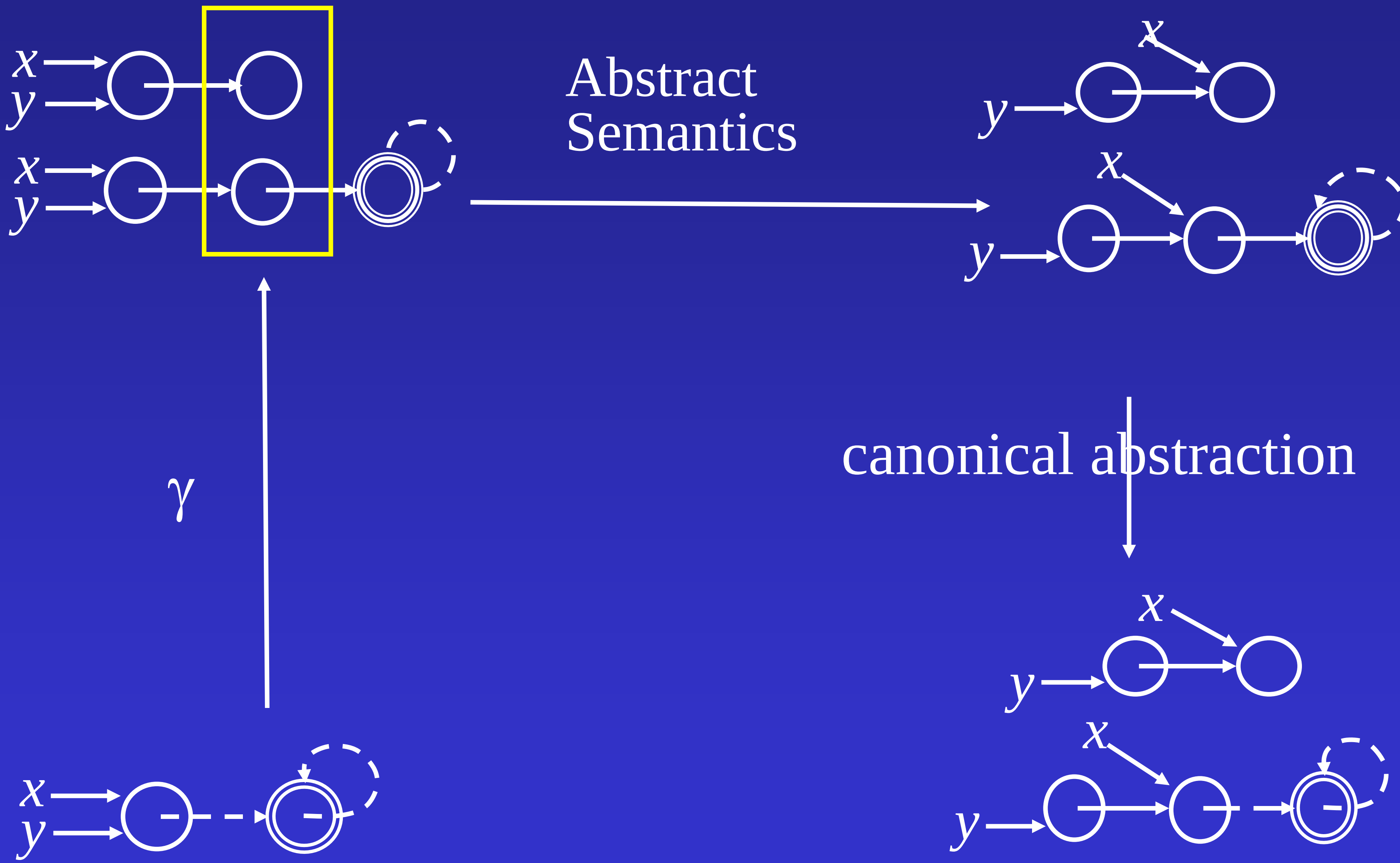
Partial Concretization



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



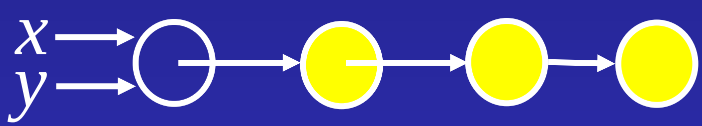
Partial Concretization based Transformer ($x = x \rightarrow n$)

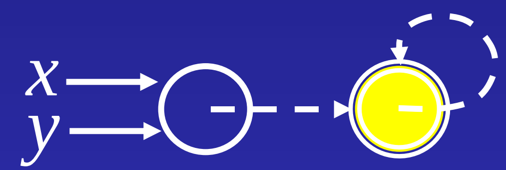


Partial Concretization

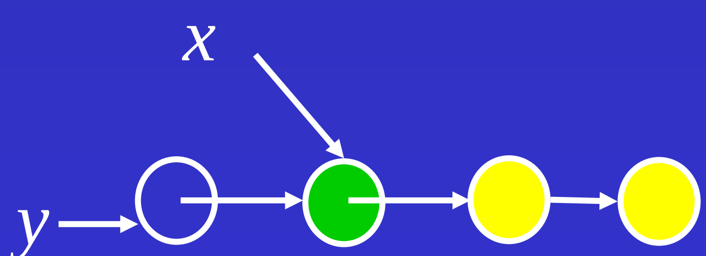
- Employed in other shape analysis algorithms [Distefano, TACAS'06, Evan, SAS'07, POPL'08]
- Soundness is immediate
- Can even guarantee precision under certain conditions [Lev-Ami, VMCAI'07]
- Locally refine the abstract domain per statement

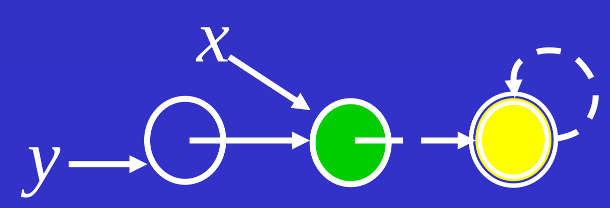
Non-Fixed-Partition



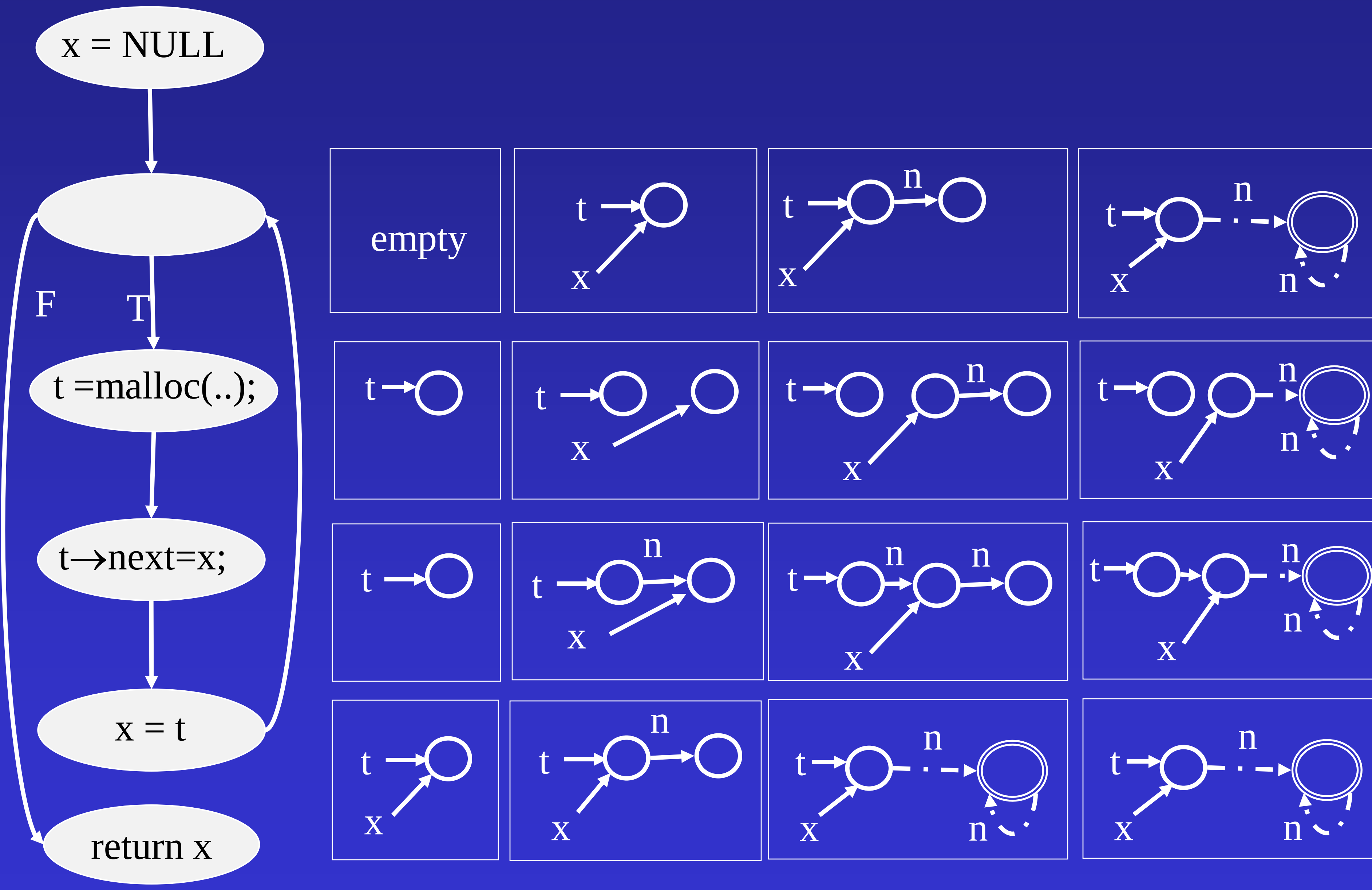


$$x = x \rightarrow n$$





Shape Analysis



[TOPLAS'02, Lev-Ami, SAS'00]

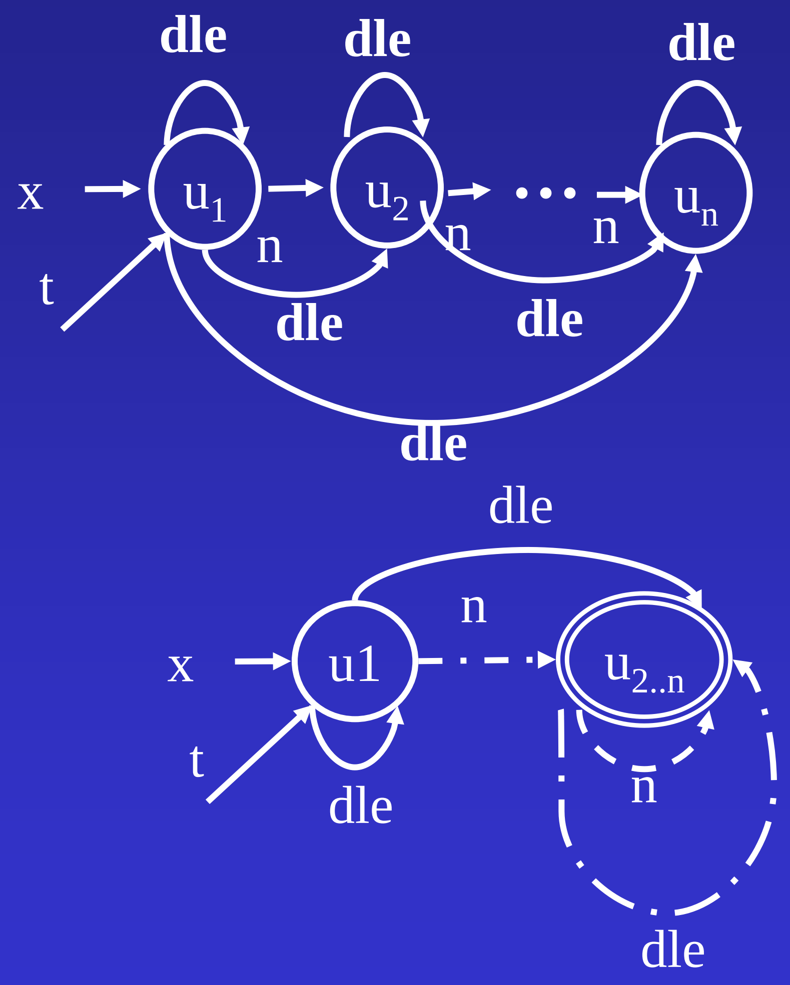
- Concrete transformers using first order formulas
- Effective algorithms for computing transformers
  - Partial concretization
  - 3-valued logic Kleene evaluation
  - Finite differencing & incremental algorithms [Reps, ESOP'03]
- A parametric yacc like system[TVLA]
  - http://www.cs.tau.ac.il/~tvla

Applications

Proving Correctness of Sorting Implementations (Lev-Ami, Reps, S, Wilhelm ISSTA 2000)

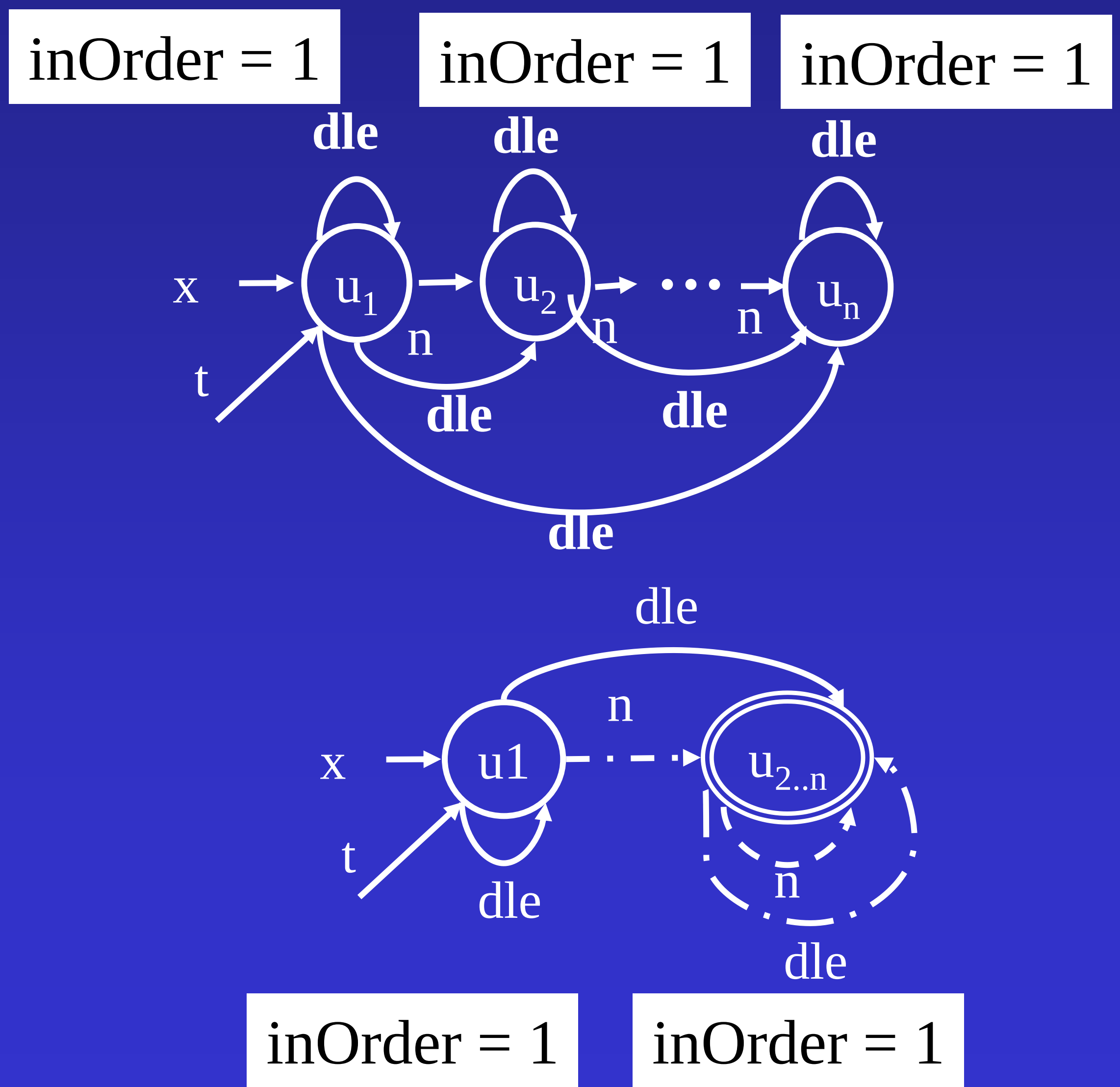
- Partial correctness
  - The elements are sorted
  - The list is a permutation of the original list
- Termination
  - At every loop iterations the set of elements reachable from the head is decreased

Sortedness



Example: Sortedness

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



Example: InsertSort

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

Run Demo

```
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;
             }
```

Run Demo

Verification of Safety Properties (PLDI'02, 04)

The *Canvas* Project (with IBM Watson)

(***C***omponent ***A***nnotation, ***V***erification ***a***nd ***S***tuff)

Component
a library with cleanly encapsulated state

Client
a program that uses the library

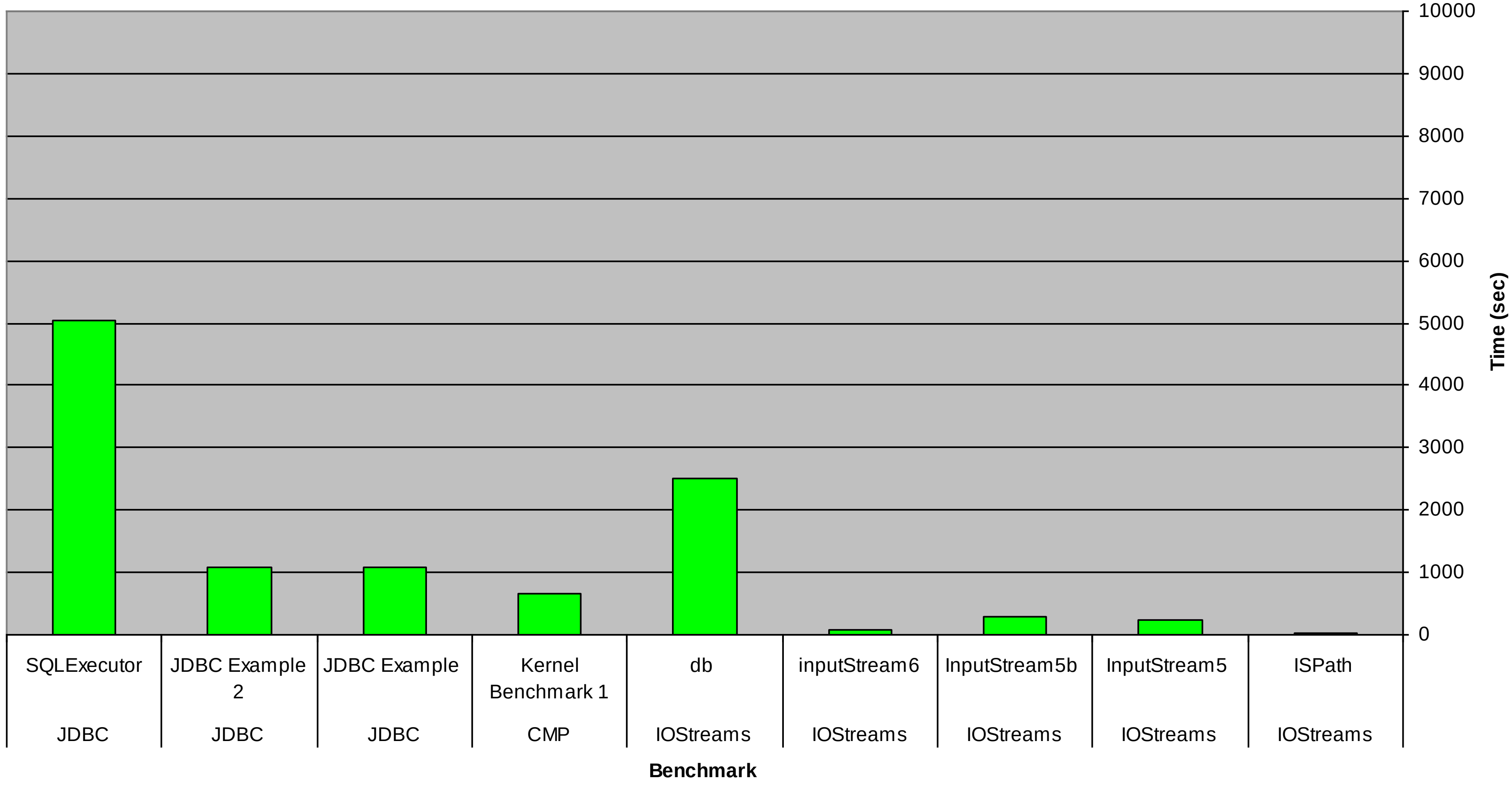
Lightweight Specification
- "correct usage" rules a client must follow
- "call open() before read()"

Certification
does the client program satisfy the lightweight specification?

Prototype Implementation

- Applied to several example programs
  - Up to 5000 lines of Java
- Used to verify
  - Absence of concurrent modification exception
  - JDBC API conformance
  - IOStreams API conformance

Analysis Times



Concurrency

- Models threads as ordinary objects [Yahav, POPL’01]
- Thread-modular shape analysis [Gotsman, PLDI’07]
- Heap decomposition [Manevich, SAS’08]
- Thread quantification [Berdine, CAV’08]
- Enforcing a locking regime [Rinetzkey]

Correctness of concurrent ADT implementations [Amit CAV'07, Berdine, CAV'08]

- Small pointer manipulation programs
- Fine grained concurrency
- Benign data races
- Error prone
- Interesting properties
  - Memory safety
  - Partial correctness
  - Linearizability
  - Liveness

Treiber's Non-blocking Stack

```
void push(Stack *S, data_type v) {
    Node *x = alloc(sizeof(Node));
    x->d = v;
    do {
        Node *t = S->Top;
        x->n = t;
    } while (!CAS(&S->Top,t,x));
}


data_type pop(Stack *S){
    do {
        Node *t = S->Top;
        if (t == NULL)
            return EMPTY;
        Node *s = t->n;
        data_type r = s->d;
        } while (!CAS(&S->Top,t,s));
    return r;
}
```

Experimental results

Verified Programs	#states	time (sec.)
Treiber's stack [1986]	764	7
Two-lock queue [Michael & Scott, PODC'96]	3,415	17
Non-blocking queue [Doherty & Groves, FORTE'04]	10,333	252

- **First automatic verification of linearizability for unbounded number of threads**

Handling Larger Programs

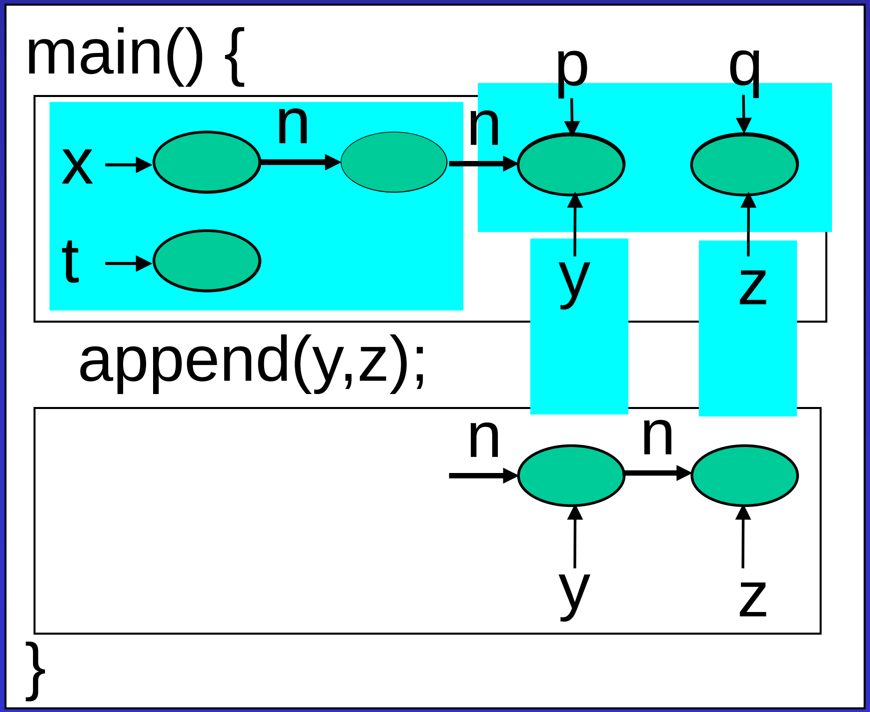
- Staged analysis
- Handling procedures
- Specialized abstractions
  - Counterexample guided refinement [McMillan, POPL’08]
- Coercer abstractions
  - Weaker summary nodes [Arnold, SAS’06]
  - Special join operator [Manevich, SAS’04, TACAS’07, SAS’08, Yang’08]
  - Heterogeneous abstractions [Yahav, PLDI’04]
- Implementation techniques
  - Optimizing transformers [Bogodlov, CAV’07]
  - Optimizing GC
  - Reducing static size
  - Partial evaluation
  - Persistent data structures [Manevich, SAS’04]
  - …

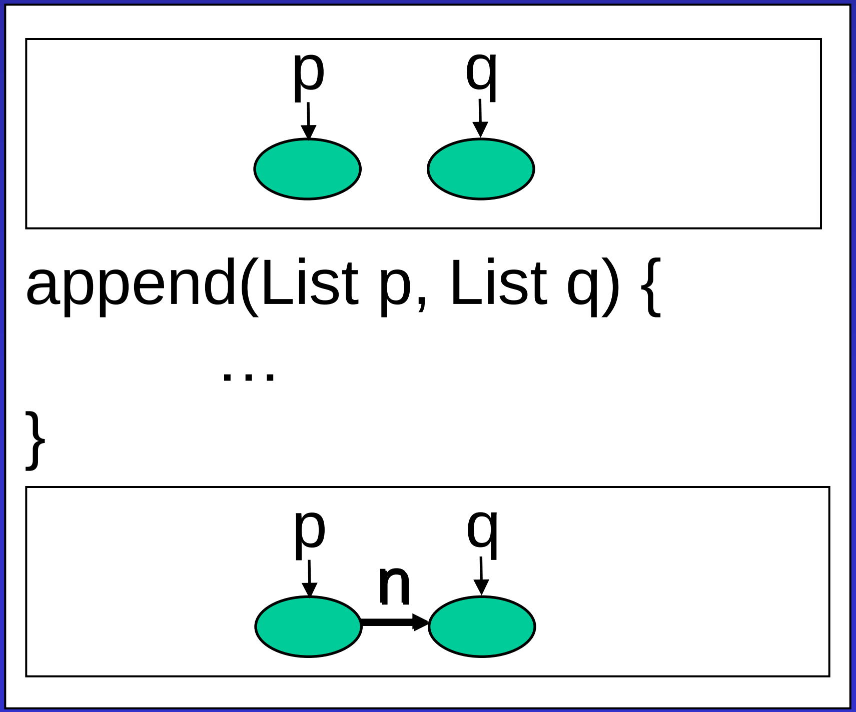
Handling Procedures

- Complicated sharing patterns [Rinetzky, CC'01]
- Relational shape analysis [Jeannet, SAS'04]
- New semantics for procedures (Cutpoints) [Rinetzky, POPL'05]
- Tabulation for cutpoint free programs [Rinetzky, SAS'05]
- Handling cutpoints [Gotsman, SAS'06]
- Modularity [Rinezky, ESOP'07]

How to tabulate procedures?

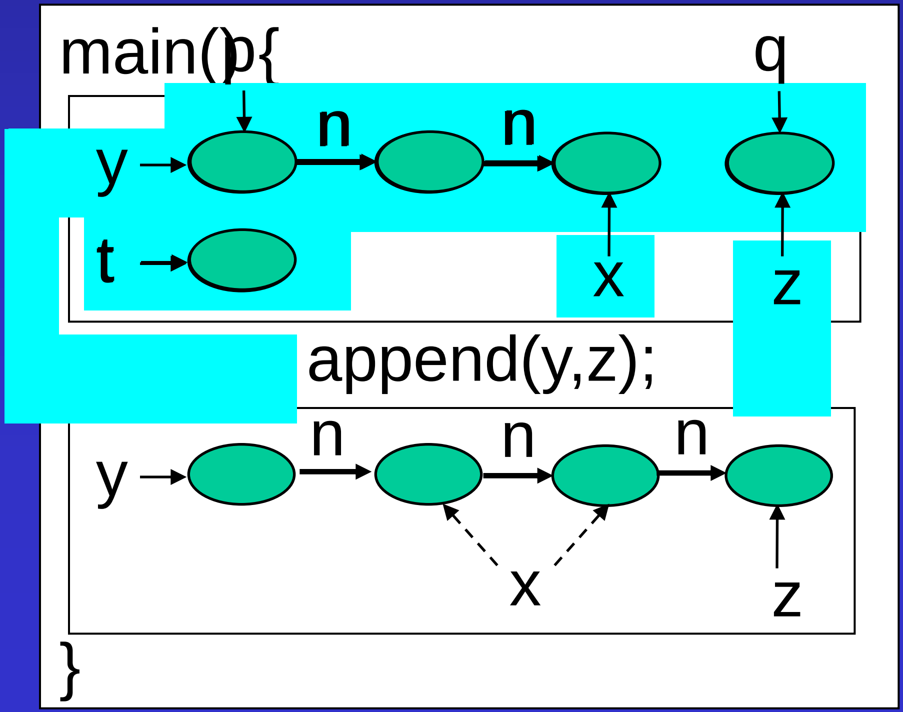
- Procedure ≡ input/output relation
  - Not reachable → Not effected
  - proc: local (≡reachable) heap → local heap

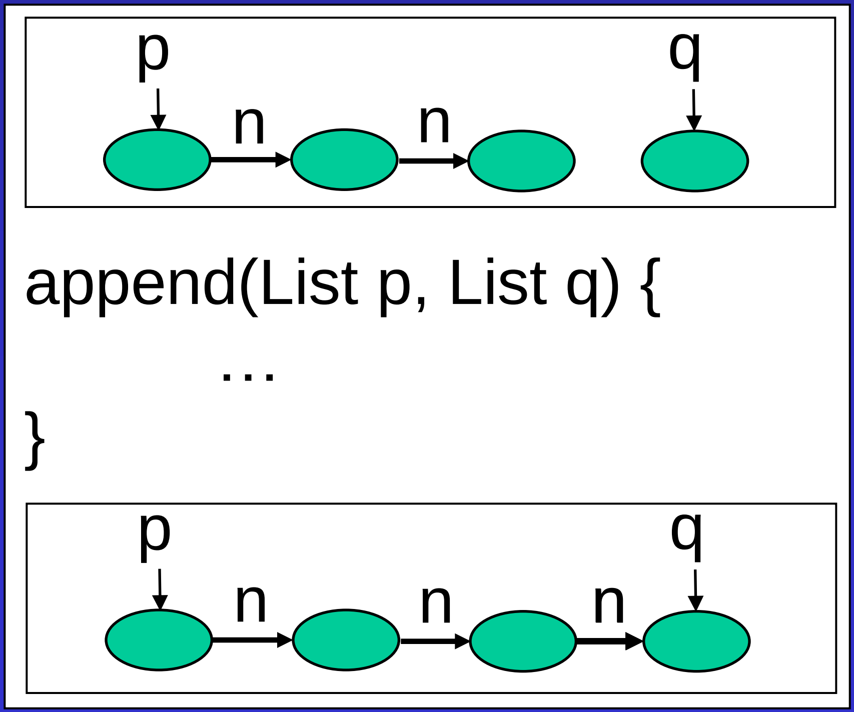




How to handle sharing?

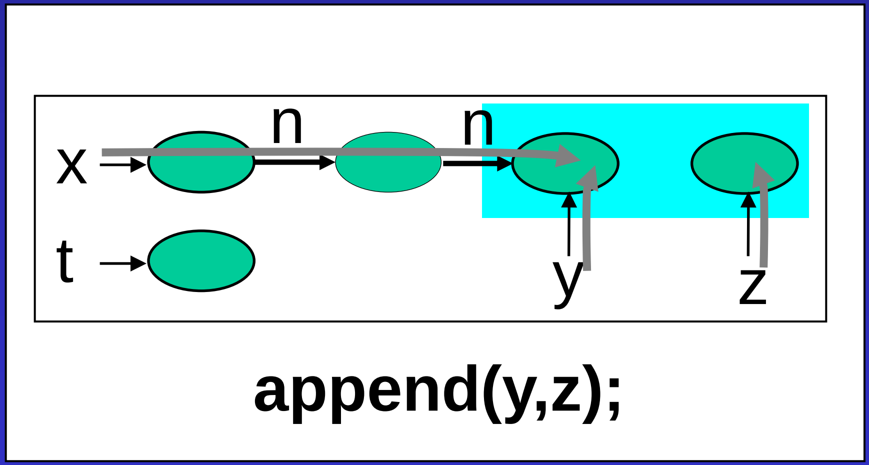
- External sharing may break the functional view



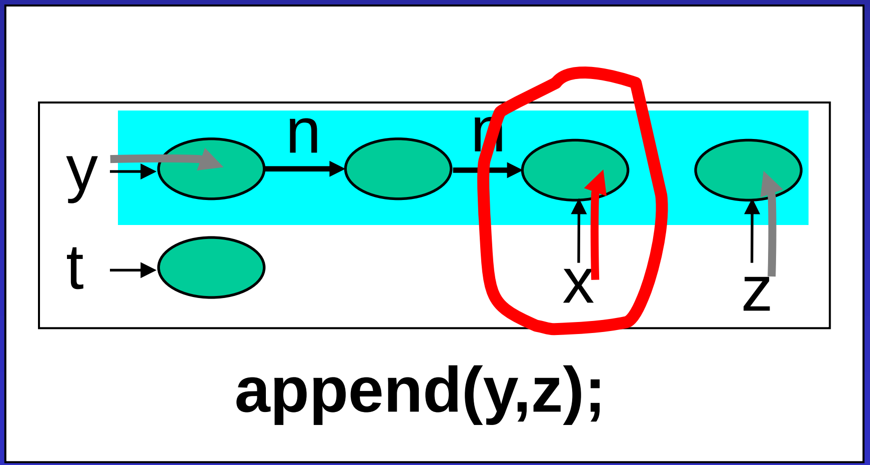


What's the difference?

1st Example



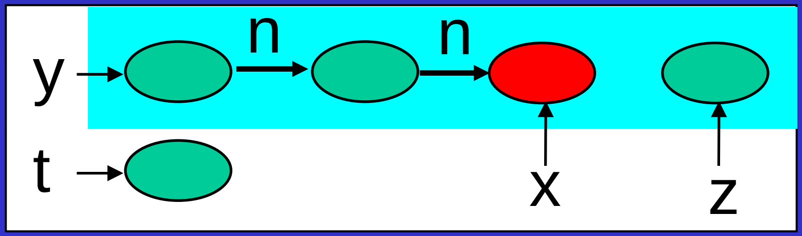
2nd Example



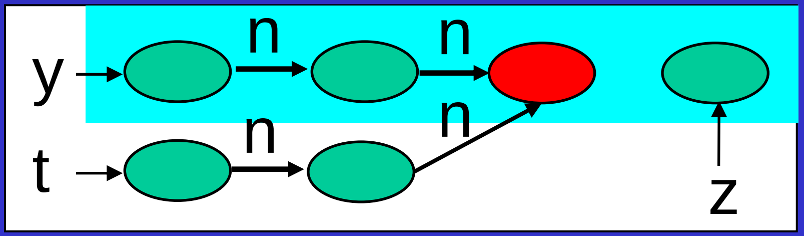
Cutpoints

- 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

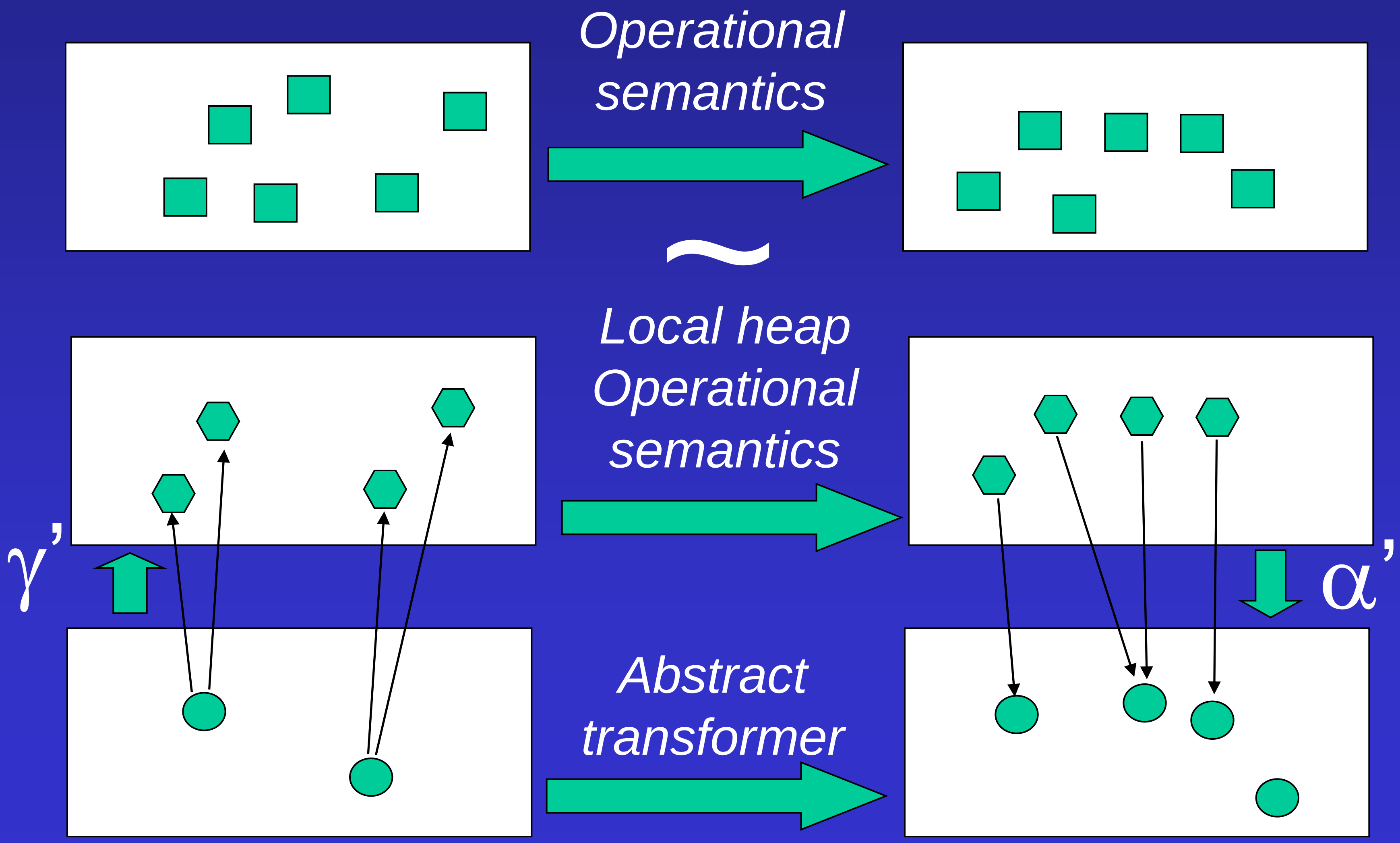
append(y,z)



append(y,z)

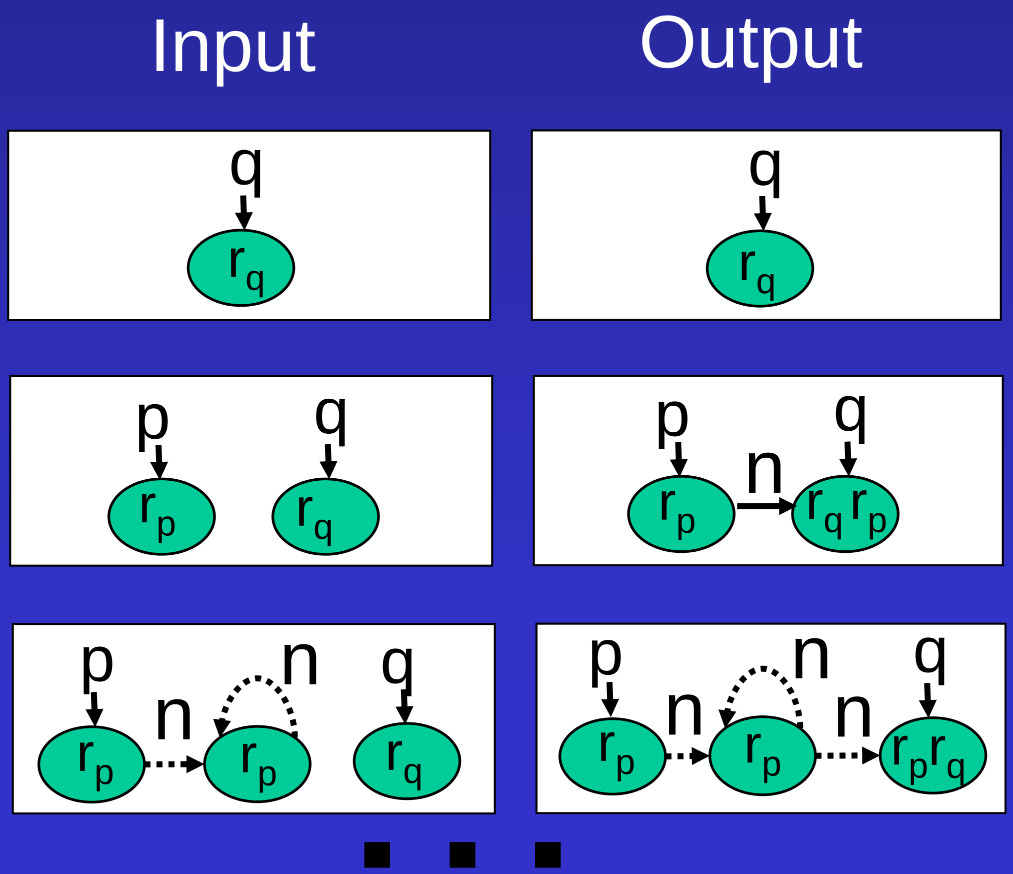


Introducing local heap semantics



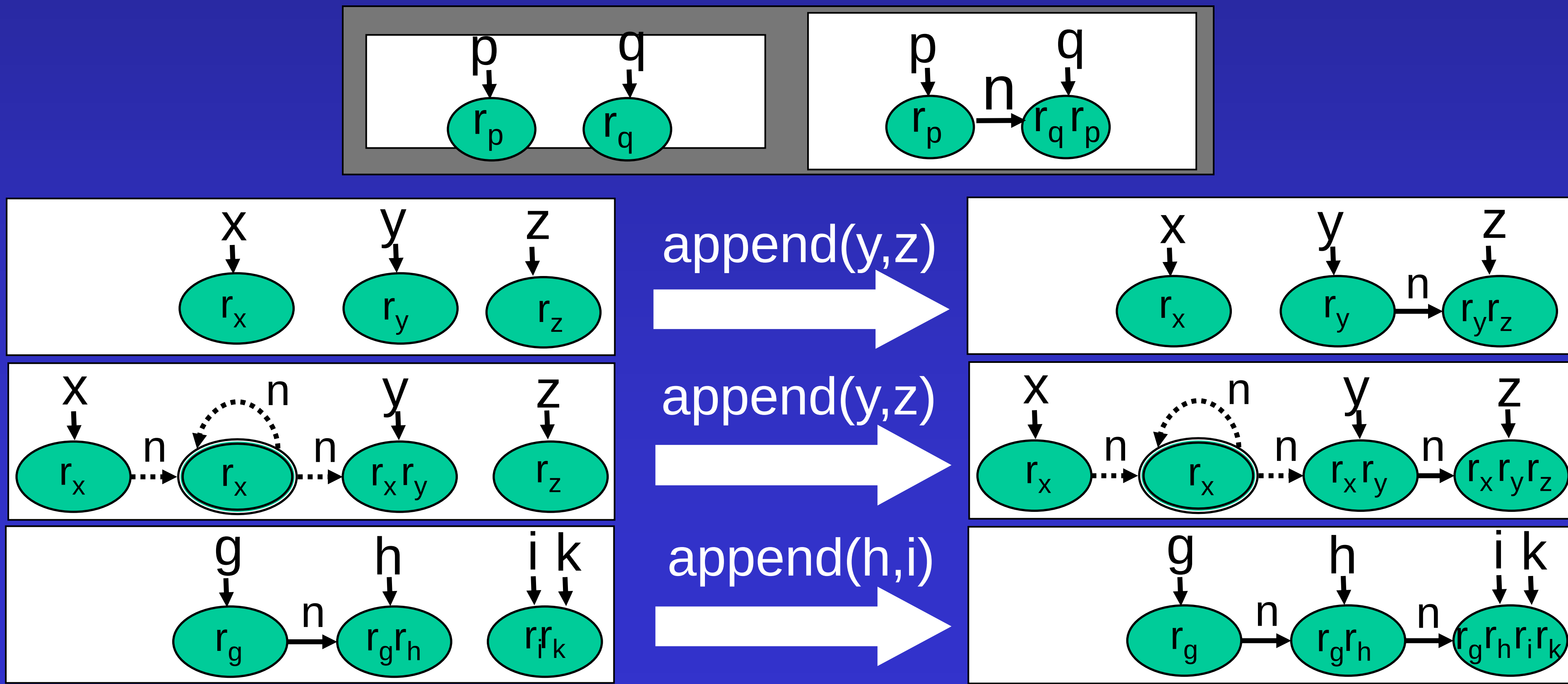
Interprocedural shape analysis

- Procedure ≡ input/output relation



Interprocedural shape analysis

- Reusable procedure summaries
  - Heap modularity



Summary

- Canonical abstraction is powerful
  - Intuitive
  - Adapts to the property of interest
  - More instrumentation may mean more efficient
- Used to verify interesting program properties
  - Very few false alarms
- But scaling is an issue