Shape Analysis via 3-Valued Logic

Mooly Sagiv
Tel Aviv University

http://www.cs.tau.ac.il/~msagiv/toplas02.ps

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

Credits

- Tel-Aviv University
  - G. Arnold
  - T. Lev-Ami
  - R. Manevich
  - A. Rabinovich
  - N. Rinetzky
  - G. Yorsh
- University of Wisconsin
  - D. Gopan
  - A. Lal
  - A. Loginov
  - **T. Reps**
- Universität des Saarlandes
  - Jörg Bauer
  - Ronald Biber
  - Sascha Parduhn
  - Jan Reineke
  - **R. Wilhelm**
  - Bjoern Wachter
- University of Massachusetts
  - N. Immerman
- Inria
  - B. Jeannet
- IBM Research
  - N. Dor
  - J. Field
  - H. Kolodner
  - G. Ramalingam
  - M. Rodeh
  - R. Shaham
  - E. Yahav

Topics

- A new abstract domain for static analysis
- Abstract dynamically allocated memory
- TVLA: A system for generating abstract interpreters
- Applications

Motivation

- Dynamically allocated storage and pointers are essential programming tools
  - Object oriented
  - Modularity
  - Data structure
- But
  - Error prone
  - Inefficient
- Static analysis can be very useful here

A Pathological C Program

```
a =  malloc(…) ;

b = a;

free (a);

c = malloc (…);

if  (b == c)  printf("unexpected equality");
```

Dereference of NULL pointers

```
typedef struct element {
   int value;
   struct element *next;
} Elements
```

```
bool search(int value, Elements *c) {
    Elements *elem;
    for (elem = c;
          c != NULL;
          elem = elem->next;)
               if (elem->val == value)
                    return TRUE;
    return FALSE
```

Dereference of NULL pointers

```
typedef struct element {
   int value;
   struct element *next;
} Elements
```

```
bool search(int value, Elements *c) {
   Elements *elem;
   for (elem = c;
         c != NULL;
         elem = elem->next;)
               if (elem->val == value)
                     return TRUE;
   return FALSE
```

potential null
de-reference

Memory leakage

```
typedef struct element {
   int value;
   struct element *next;
} Elements
```

```
Elements* reverse(Elements *c)
   {
   Elements *h,*g;
   h = NULL;
   while (c!= NULL) {
        g = c->next;
        h = c;
        c->next = h;
        c = g;
              }
   return h;
```

Memory leakage

```
typedef struct element {
   int value;
   struct element *next;
} Elements
```

leakage of address pointed-by h

```
Elements* reverse(Elements *c)
   {
   Elements *h,*g;
   h = NULL;
   while (c!= NULL) {
         g = c->next;
         h = c;
         c->next = h;
         c = g;
               }
   return h;
```

Memory leakage

```
typedef struct element {
   int value;
   struct element *next;
} Elements
```

```
Elements* reverse(Elements *c)
    {
    Elements *h,*g;
    h = NULL;
    while (c!= NULL) {
          g = c->next;
          h = c;
          c->next = h;
          c = g;
                    }
    return h;
```

? No memory leaks

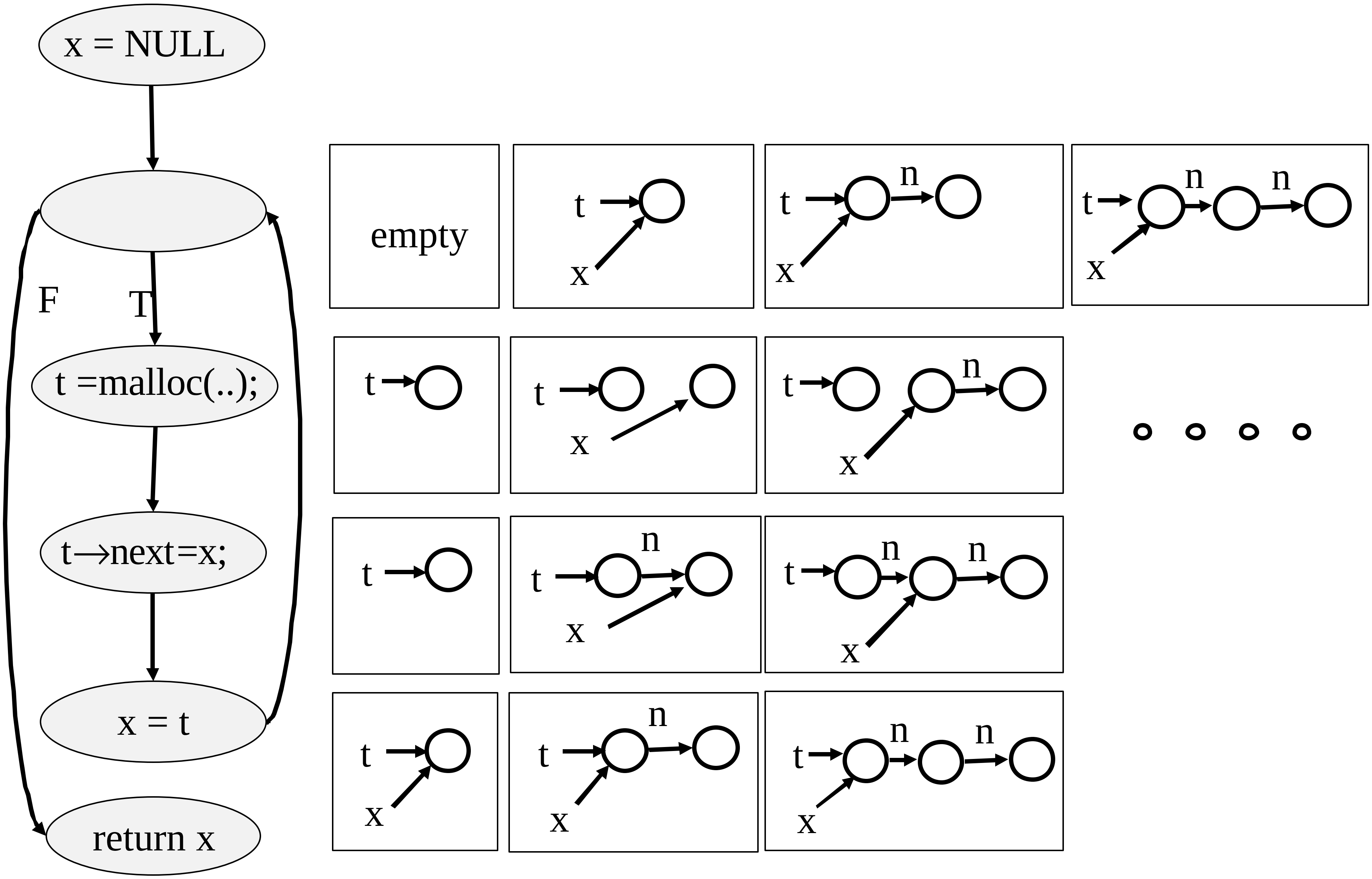
Example: List Creation

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

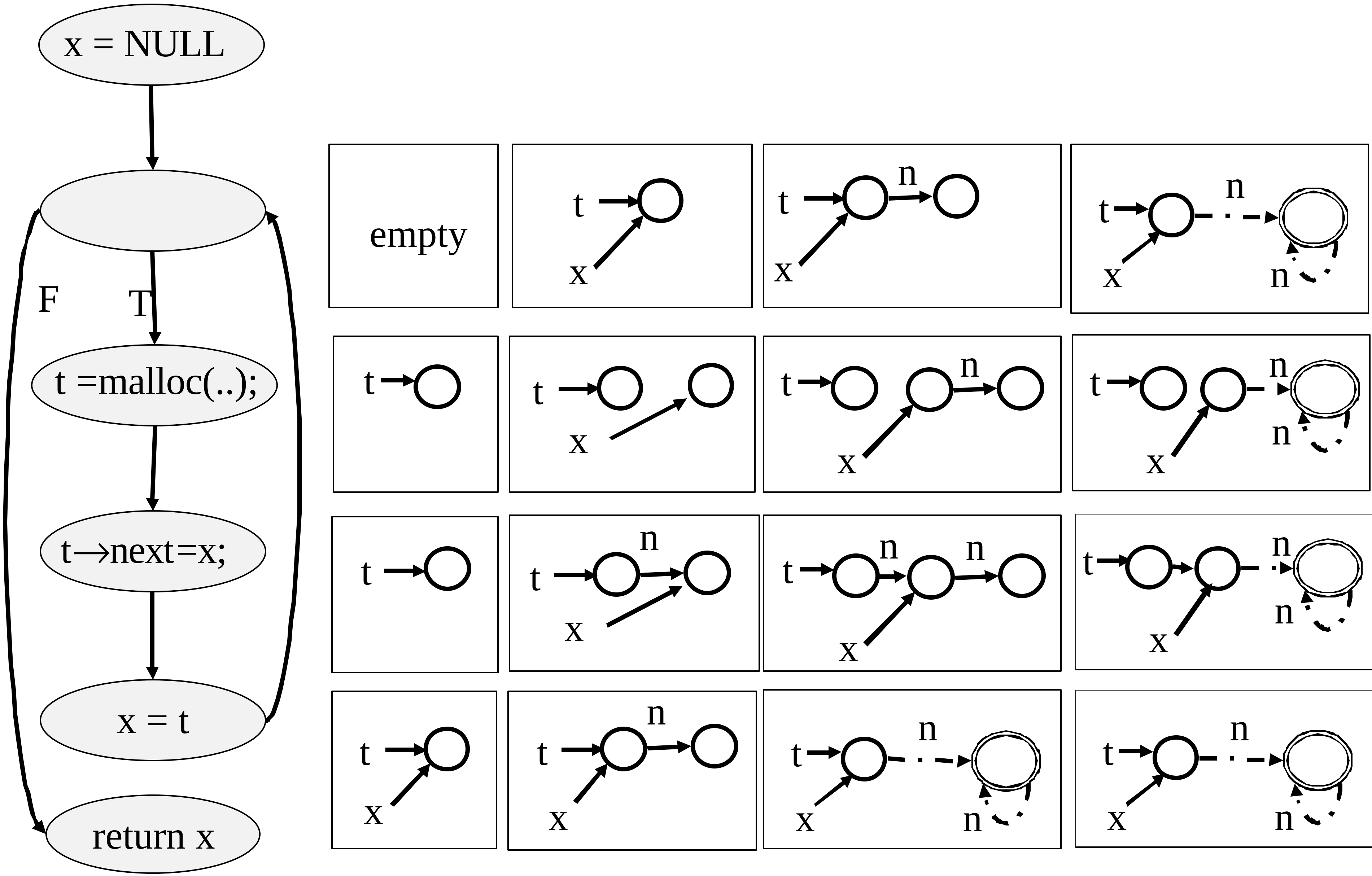
```
List create (…)
{
List x, t;
x = NULL;
while (…) do {
      t = malloc();
      t →next=x;
     x = t ;}
return x;
}
```

- ? No null dereferences
- ? No memory leaks
- ? Returns acyclic list

Example: Collecting Interpretation



Example: Abstract Interpretation



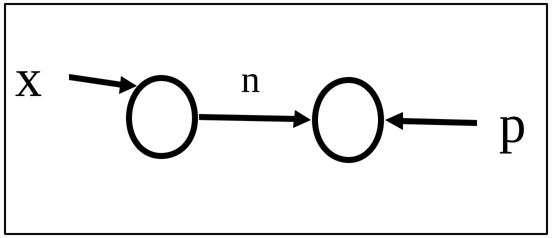
Challenge 1 - Memory Allocation

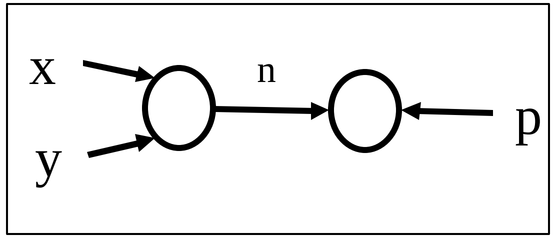
- The number of allocated objects/threads is not known
- Concrete state space is infinite
- How to guarantee termination?

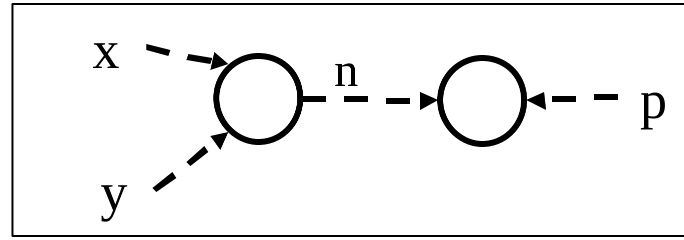
Challenge 2 - Destructive Updates

- The program manipulates states using destructive updates
  - $e \rightarrow next = t$
- Hard to define concrete interpretation
- Harder to define abstract interpretation

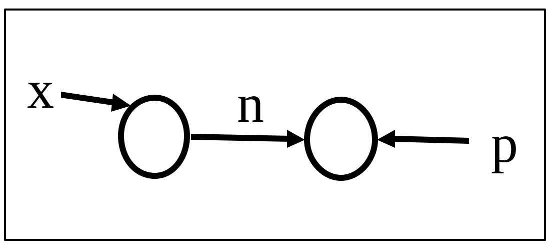
Challenge 2 - Destructive Update

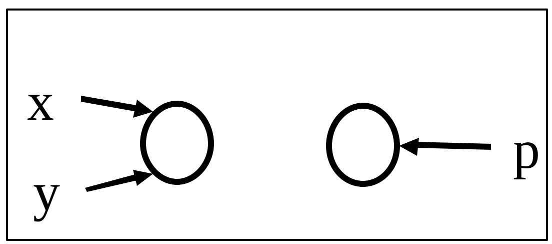


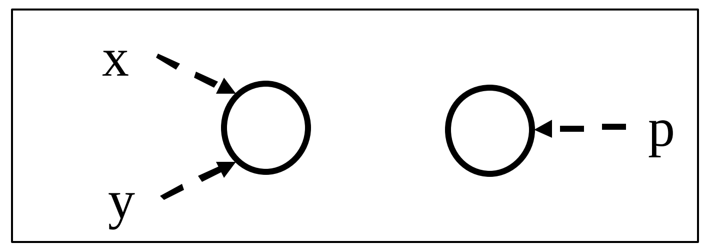




y→next = NULL

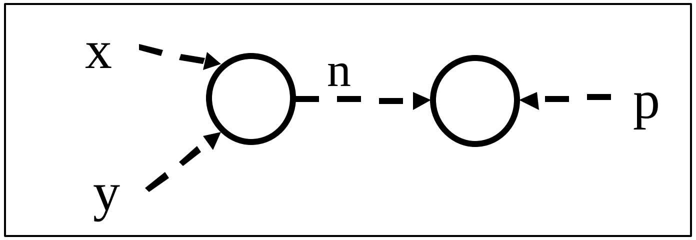




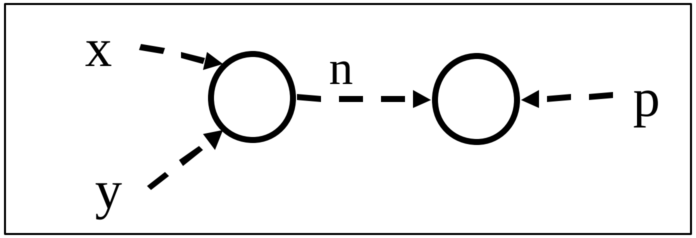


Unsound ☹

Challenge 2 - Destructive Update



y→next = NULL



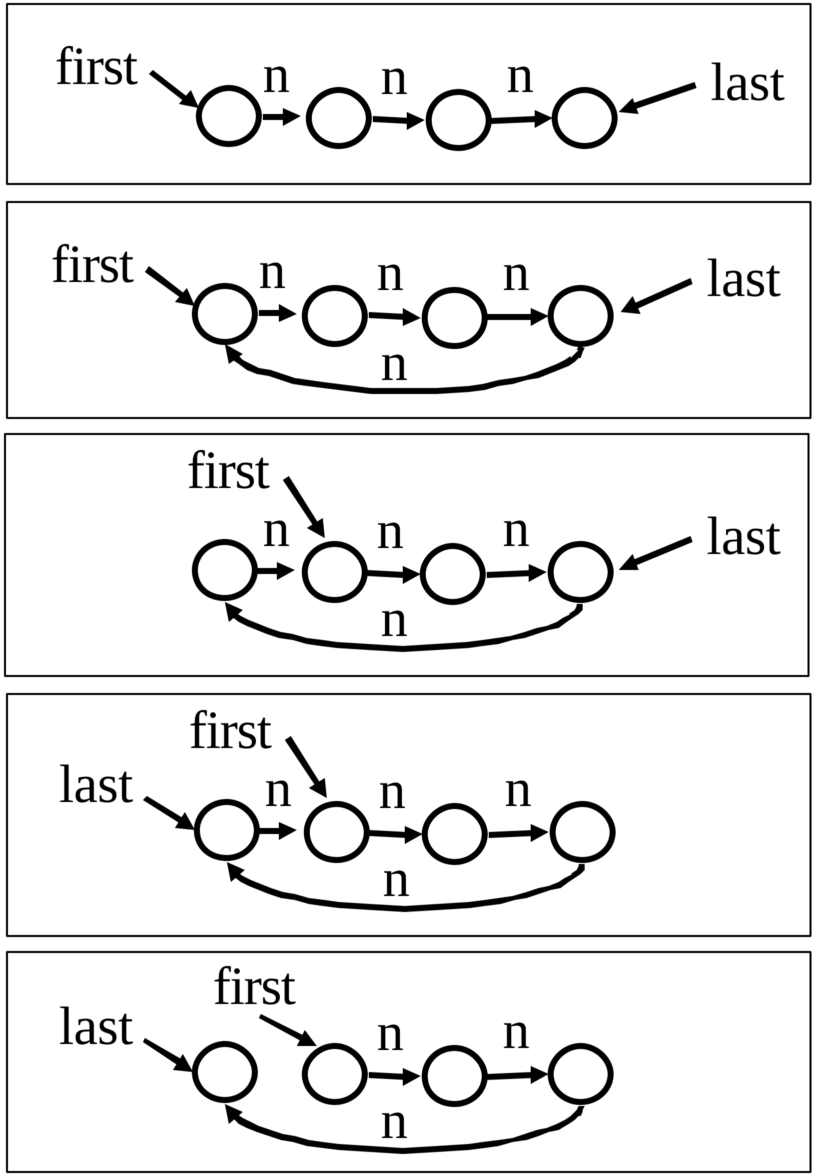
Imprecise ☹

Challenge 3 – Re-establishing Data Structure Invariants

- Data-structure invariants typically only hold at the beginning and end of ADT operations
- Need to verify that data-structure invariants are re-established

Challenge 3 – Re-establishing Data Structure Invariants

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

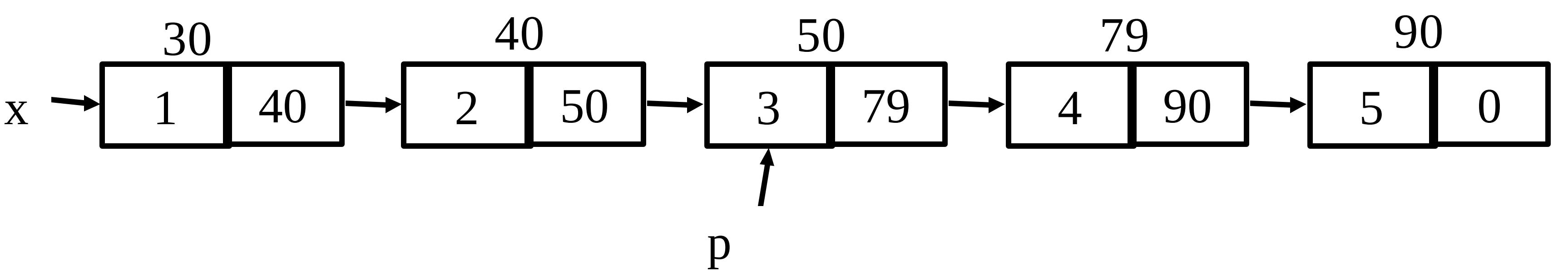


Plan

- Concrete interpretation
- Canonical abstraction
- Abstract interpretation using canonical abstraction (next lesson)

Traditional Heap Interpretation

- States = Two level stores
  - Env: Var ⇀ *Values*
  - fields: Loc ⇀ *Values*
  - *Values*=Loc ∪Atoms
- Example
  - Env = [x ↦ 30, p ↦ 79]
  - next = [30 ↦40, 40 ↦ 50, 50 ↦79, 79 ↦ 90]
  - val = [30 ↦1, 40 ↦ 2, 50 ↦3, 79 ↦ 4, 90 ↦5]

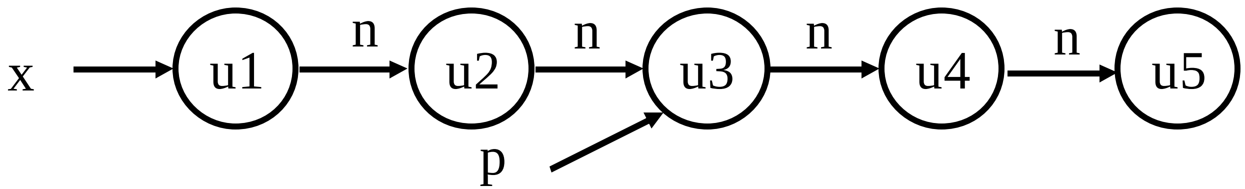


Predicate Logic

- Vocabulary
  - A finite set of predicate symbols *P* each with a fixed arity
- Logical Structures S provide meaning for predicates
  - A set of individuals (nodes) *U*
  - $p^S: (U^S)^k$ ® $\{0, 1\}$
- FO^{TC} over TC,∀∃¬∧∨ express logical structure properties

Representing Stores as Logical Structures

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



Formal Semantics of First Order Formulae

- For a structure S=<U^S, p^S>
- Formulae φ with LVar free variables
- Assignment z: LVar→U^S
- $[\![\varphi]\!]^S(z)$: {0, 1}

$$[\![1]\!]^S(z)=1$$

$$[\![0]\!]^S(z)=0$$

$$[\![p\,(v_1, v_2, \ldots, v_k)]\!]^S(z)=p^S\,(z(v_1), z(v_2), \ldots, z(v_k))$$

Formal Semantics of First Order Formulae

- For a structure $S=\langle U^S, p^S \rangle$
- Formulae φ with LVar free variables
- Assignment $z: LVar \rightarrow U^S$
- $[\![\varphi]\!]^S(z): \{0, 1\}$

$$[\![\varphi_1 \vee \varphi_2]\!]^S(z) = \max([\![\varphi_1]\!]^S(z), [\![\varphi_2]\!]^S(z))$$

$$[\![\varphi_1 \wedge \varphi_2]\!]^S(z) = \min([\![\varphi_1]\!]^S(z), [\![\varphi_2]\!]^S(z))$$

$$[\![\neg \varphi_1]\!]^S(z) = 1 - [\![\varphi_1]\!]^S(z)$$

$$[\![\exists v: \varphi_1]\!]^S(z) = \max\{[\![\varphi_1]\!]^S(z[v \mapsto u]) : u \in U^S\}$$

Formal Semantics of Transitive Closure

- For a structure $S=<U^S, p^S>$
- Formulae φ with LVar free variables
- Assignment $z: LVar \rightarrow U^S$
- $[\![\varphi]\!]^S(z)$: $\{0, 1\}$

$$[\![p^*(v_1, v_2)]\!]^S(z) =$$
$$\max \{u_1, ..., u_k \in U, Z(v_1)=u_1, Z(v_2)=u_k\}$$
$$\min\{1 \leq i < k\} \;\; p^S(u_i, u_{i+1})$$

Concrete Interpretation Rules

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

Invariants

- No memory leaks

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

- Acyclic list(x)

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

- Reverse (x)

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

  $n(w, r) \leftrightarrow n'(r, w)$

Why use logical structures?

- Naturally model pointers and dynamic allocation
- No a priori bound on number of locations
- Use formulas to express semantics
- Indirect store updates using quantifiers
- Can model other features
  - Concurrency
  - Abstract fields

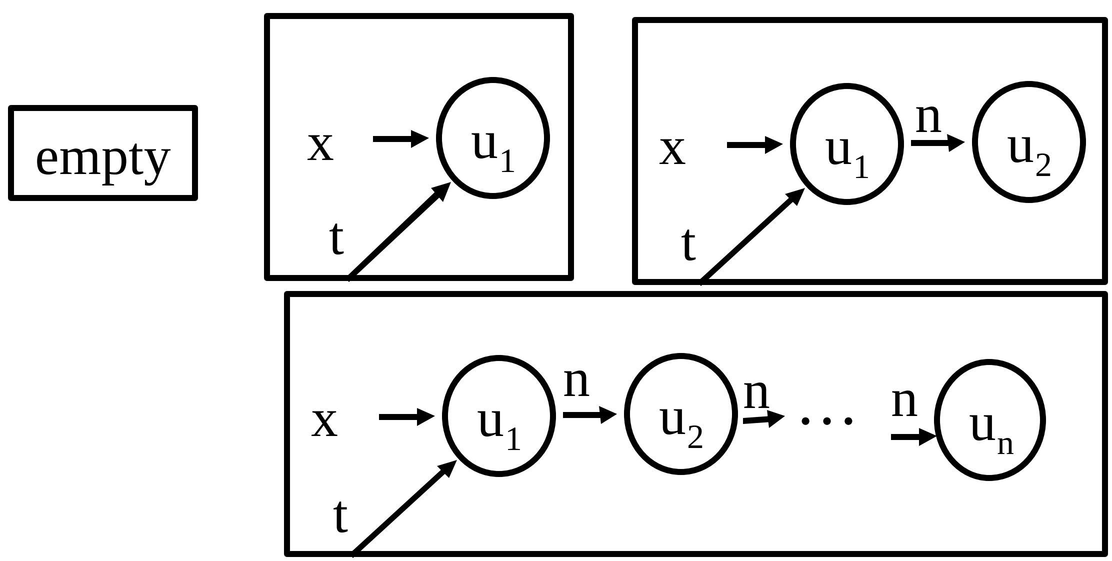
Why use logical structures?

- Behaves well under abstraction
- Enables automatic construction of abstract interpreters from concrete interpretation rules (TVLA)

Collecting Interpretation

- The set of reachable logical structures in every program point
- Statements operate on sets of logical structures
- Cannot be directly computed for programs with unbounded store and loops

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



Plan

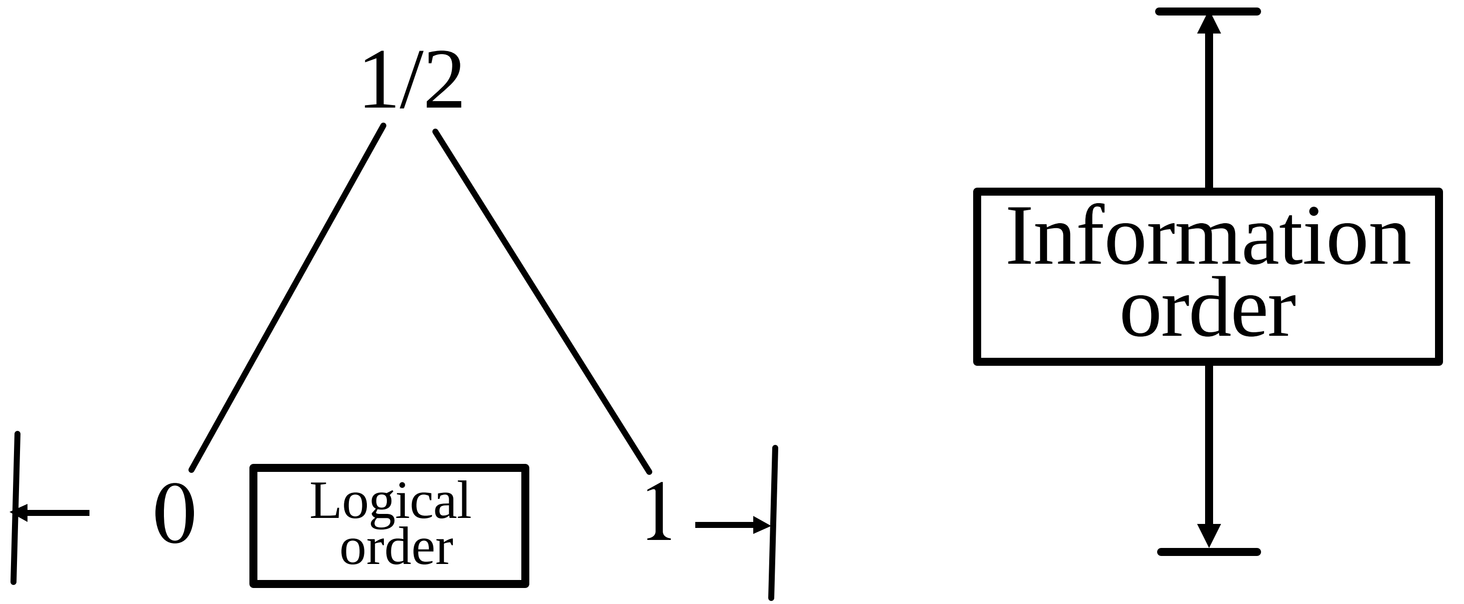
- Concrete interpretation
- Canonical abstraction
- TVLA

Canonical Abstraction

- Convert logical structures of unbounded size into bounded size
- Guarantees that number of logical structures in every program is finite
- Every first-order formula can be conservatively interpreted

Kleene Three-Valued Logic

- 1: True
- 0: False
- 1/2: Unknown
- A join semi-lattice: $0 \sqcup 1 = 1/2$



Boolean Connectives [Kleene]

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

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

3-Valued Logical Structures

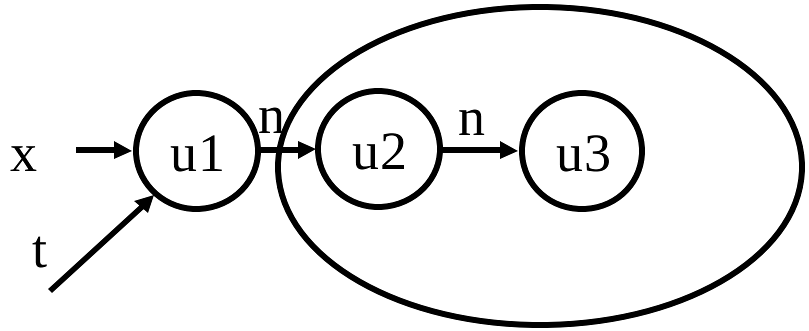
- A set of individuals (nodes) *U*
- Predicate meaning
  - $p^S: (U^S)^k$ ® $\{0, 1, 1/2\}$

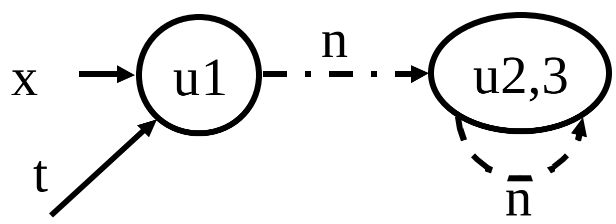
Canonical Abstraction

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

Canonical Abstraction

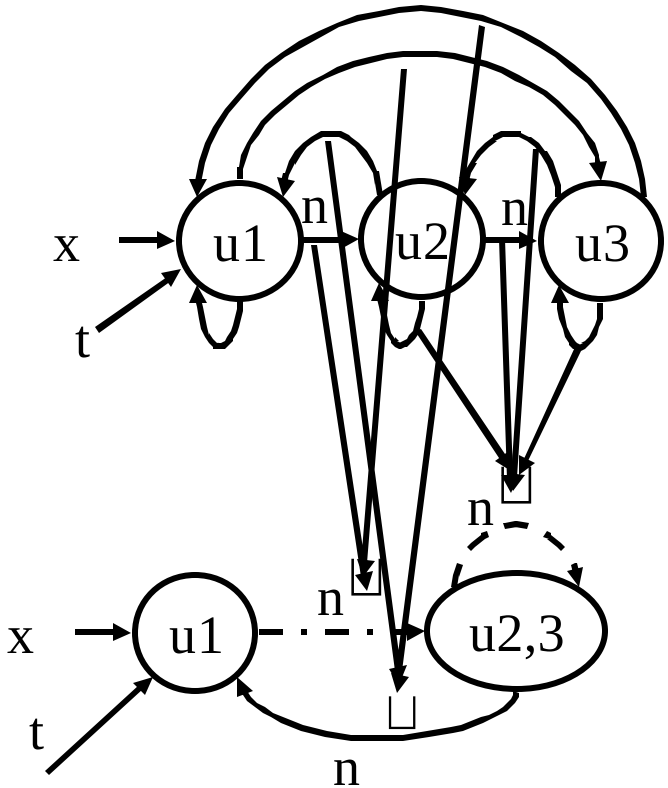
```
x = NULL;
while (…) do {
    t = malloc();
    t →next=x;
    x = t
}
```





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;

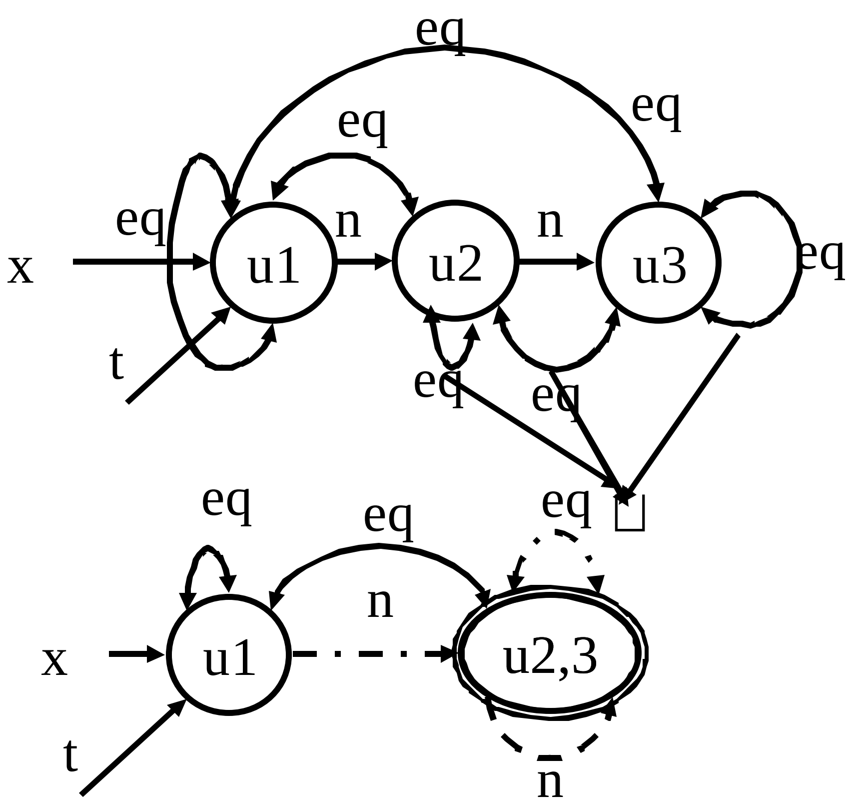
while (…) do {

    t = malloc();

    t →next=x;

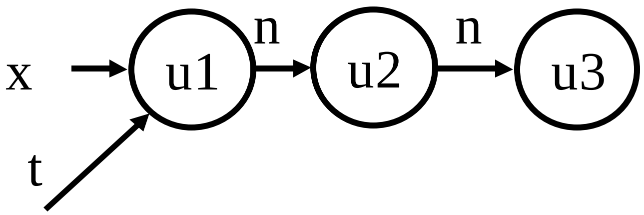
    x = t

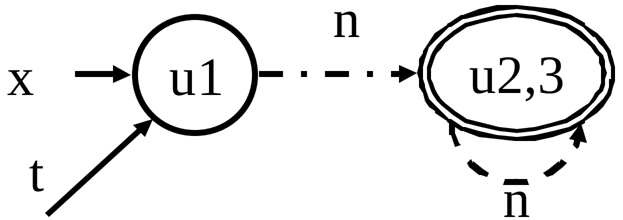
}



Canonical Abstraction

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



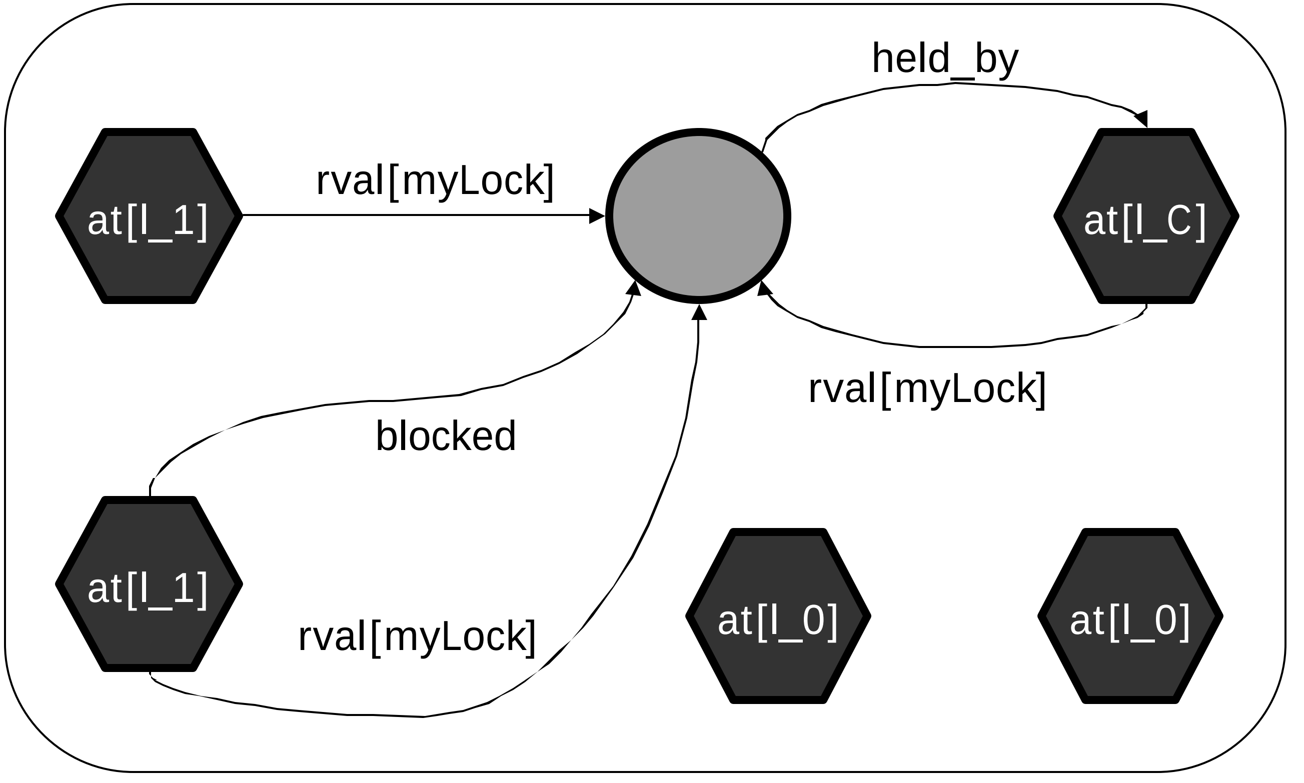


Challenges: Heap & Concurrency [Yahav POPL'01]

- Concurrency with the heap is evil…
- Java threads are just heap allocated objects
- Data and control are strongly related
  - Thread-scheduling info may require understanding of heap structure (e.g., scheduling queue)
  - Heap analysis requires information about thread scheduling

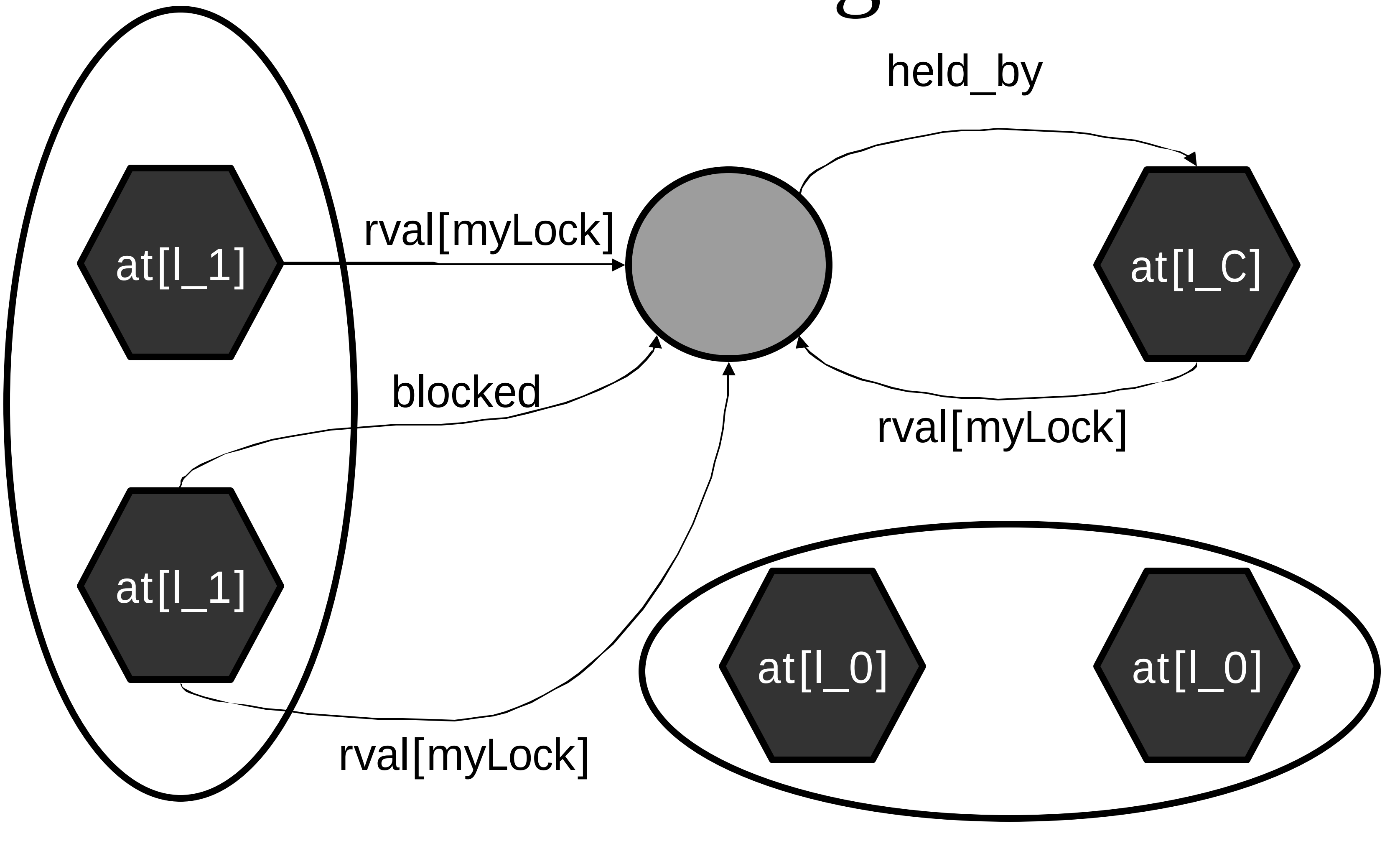
```
Thread t1 = new Thread();
Thread t2 = new Thread();
…
t = t1;
…
t.start();
```

Configurations – Example

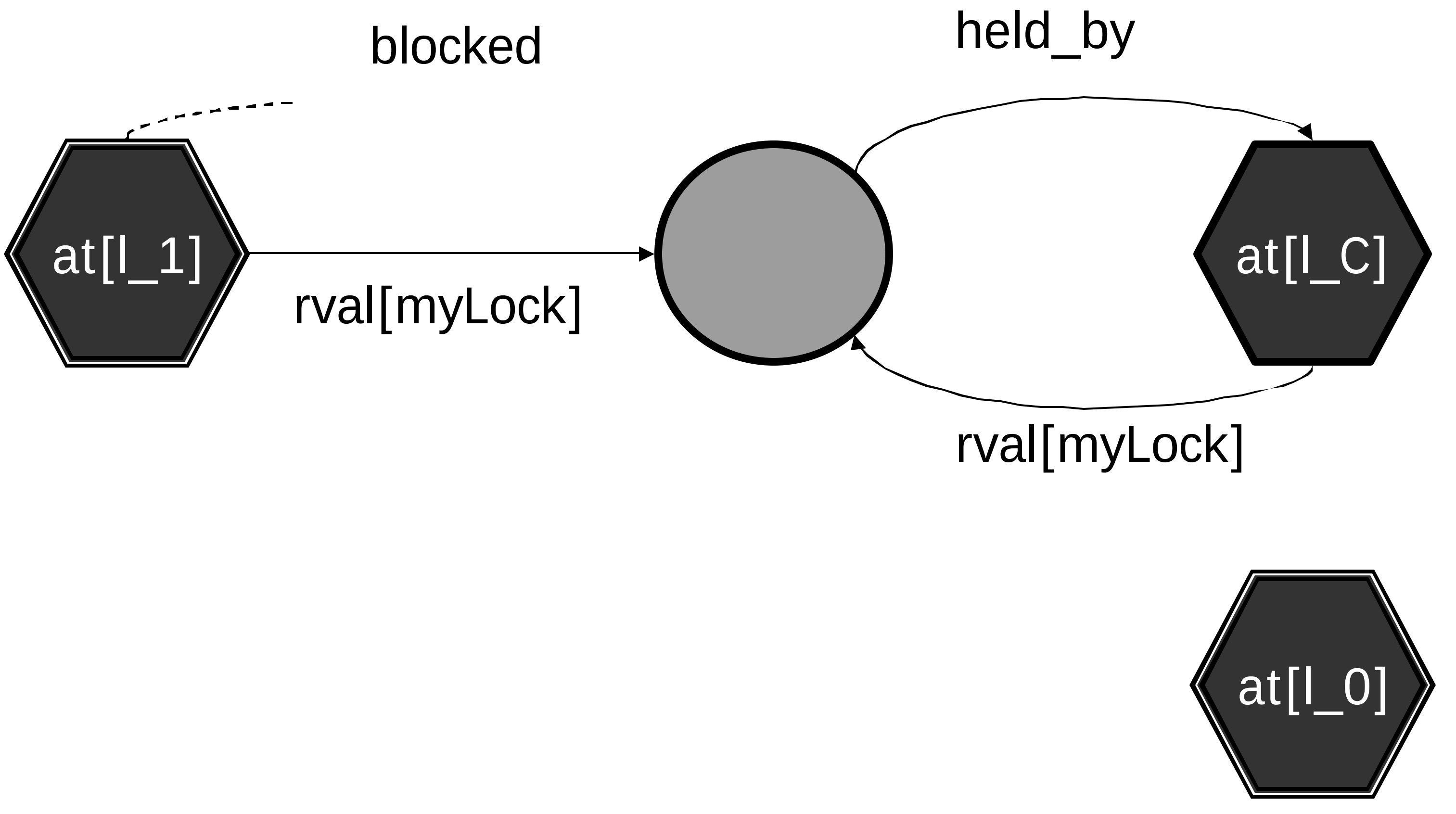



```
l_0: while (true) {
l_1:     synchronized(myLock) {
l_C:          // critical actions
l_2:     }
l_3: }
```

Concrete Configuration



Abstract Configuration



Examples Verified

Program	Property
twoLock Q	No interference No memory leaks Partial correctness
Producer/consumer	No interference No memory leaks
Apprentice Challenge	Counter increasing
Dining philosophers with resource ordering	Absence of deadlock
Mutex	Mutual exclusion
Web Server	No interference

Summary

- Canonical abstraction guarantees finite number of structures
- The concrete location of an object plays no significance
- But what is the significance of 3-valued logic?