Shape Analysis via 3-Valued Logic

Mooly Sagiv
Tel Aviv University

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

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

Clarifications

- More precise = Represents fewer states = $\sqsubseteq$
- Monotone = $x \sqsubseteq y \rightarrow f(x) \sqsubseteq f(y)$
- Deflationary = $f(x) \sqsubseteq x$

Galois Connections

- α: C $\rightarrow$A and γ: A $\rightarrow$C
- The pair of functions (α, γ) form Galois connection if
  - α and γ are monotone
  - $\forall$ a $\in$ A: $\alpha(\gamma(a)) \sqsubseteq a$
  - $\forall$ c $\in$ C: $c \sqsubseteq \gamma(\alpha(C))$
- Alternatively if:
  $\forall$ c $\in$ C, $\forall$ a $\in$ A

  $\alpha(c) \sqsubseteq a$ iff $c \sqsubseteq \gamma(a)$

Homework

- Define a Galois connection for constant propagation
- Show that every Galois connection abstraction and concretization determine each other
- Read the orange booklet on TVLA
- Download the TVLA system from www.cs.tau.ac.il/~tvla 2α (5pm)

Schedule

- ✓ Lecture 1: Abstract Interpretation in the nutshell
- Lecture 2: Operational Semantics & Naive Abstraction of Heap Allocated Data Structures
- Lecture 3: Abstract interpretation of Heap Allocated Data Structures
- Lecture 4: Demo and Applications

Topics

- A new abstract domain for static analysis
- Abstract dynamically allocated memory

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
```

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

leakage of address pointed-by 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

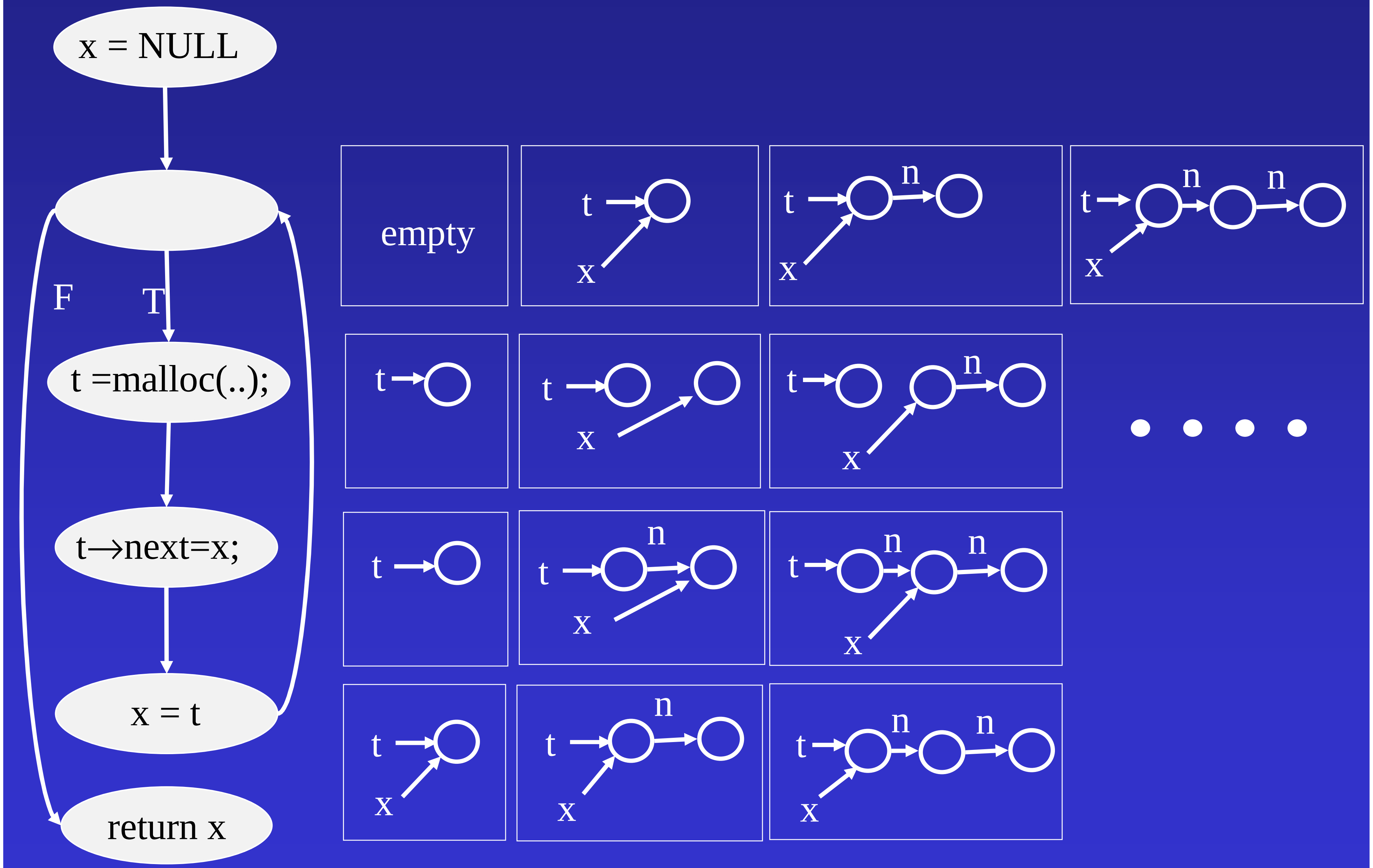
Example: List Creation

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

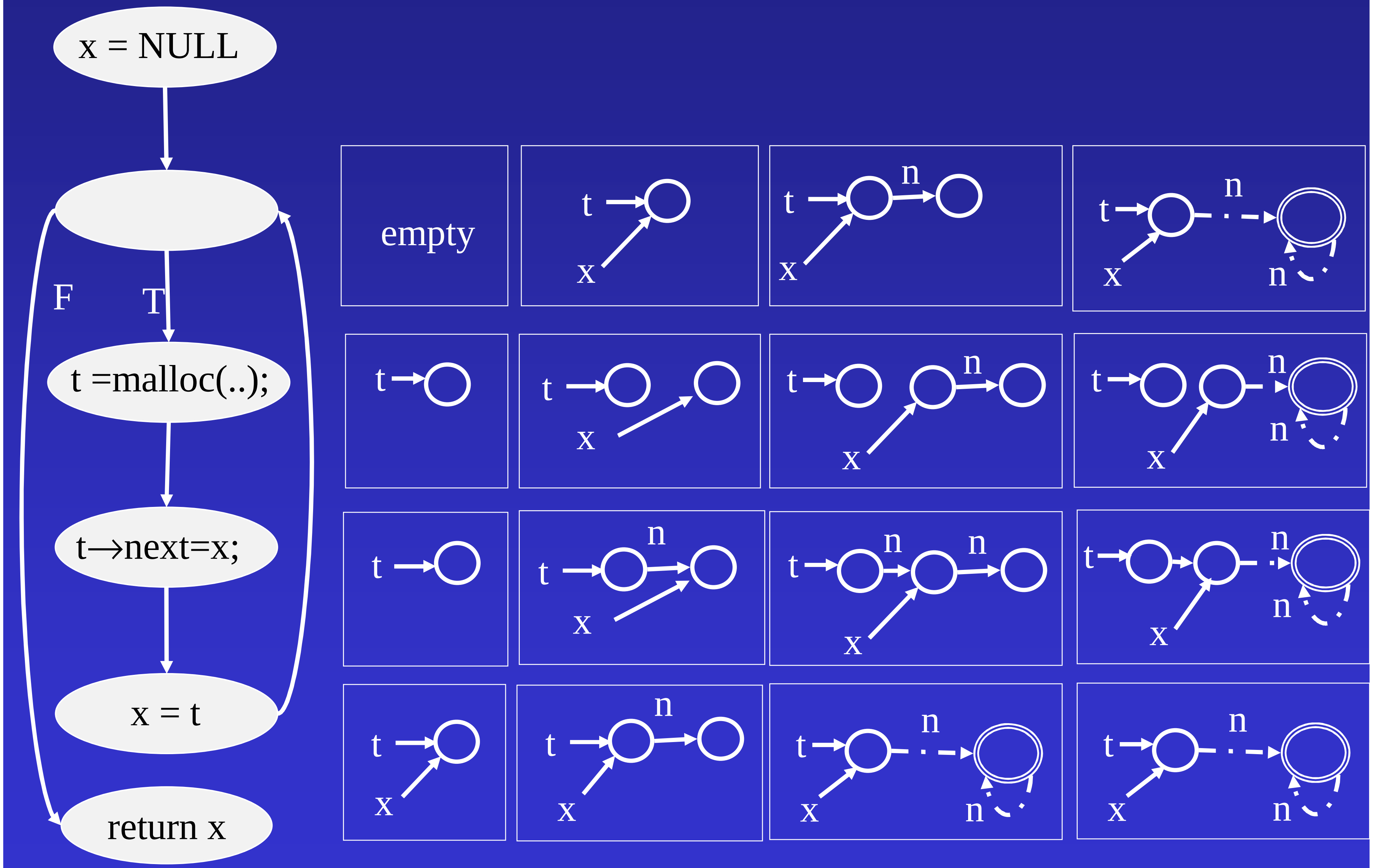
```
List create (…)
{
List x, t;
x = NULL;
while (…) do {
      t = malloc();
      t →next=x;
     x = t ;}
return x;
}
```

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

Example: Collecting Interpretation



Example: Abstract Interpretation



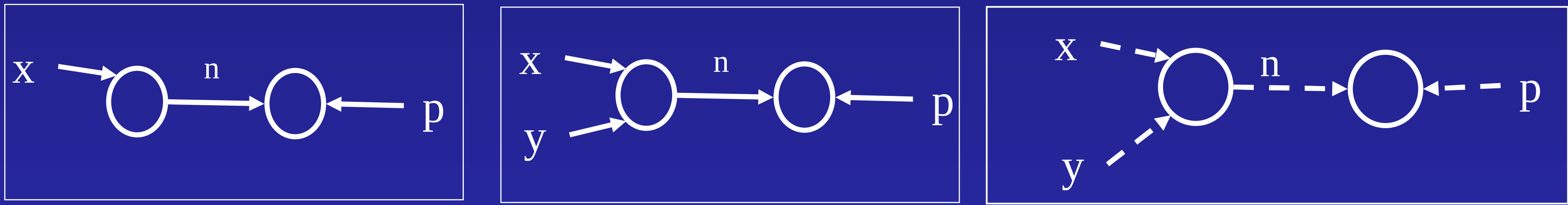
Challenge 1 - Memory Allocation

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

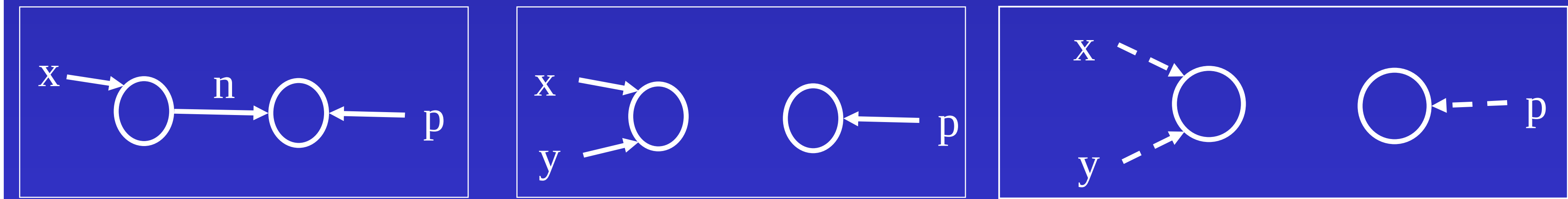
Challenge 2 - Destructive Updates

- 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

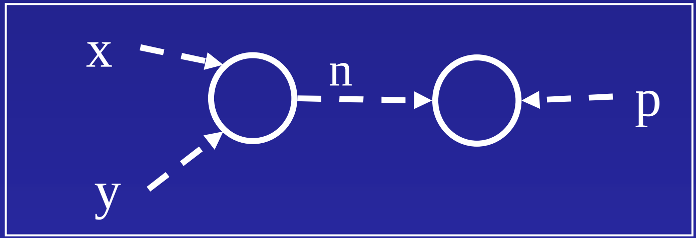


y→next = NULL

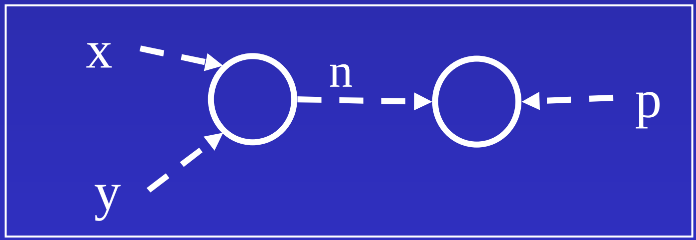


Unsound ☹

Challenge 2 - Destructive Update



y→next = NULL



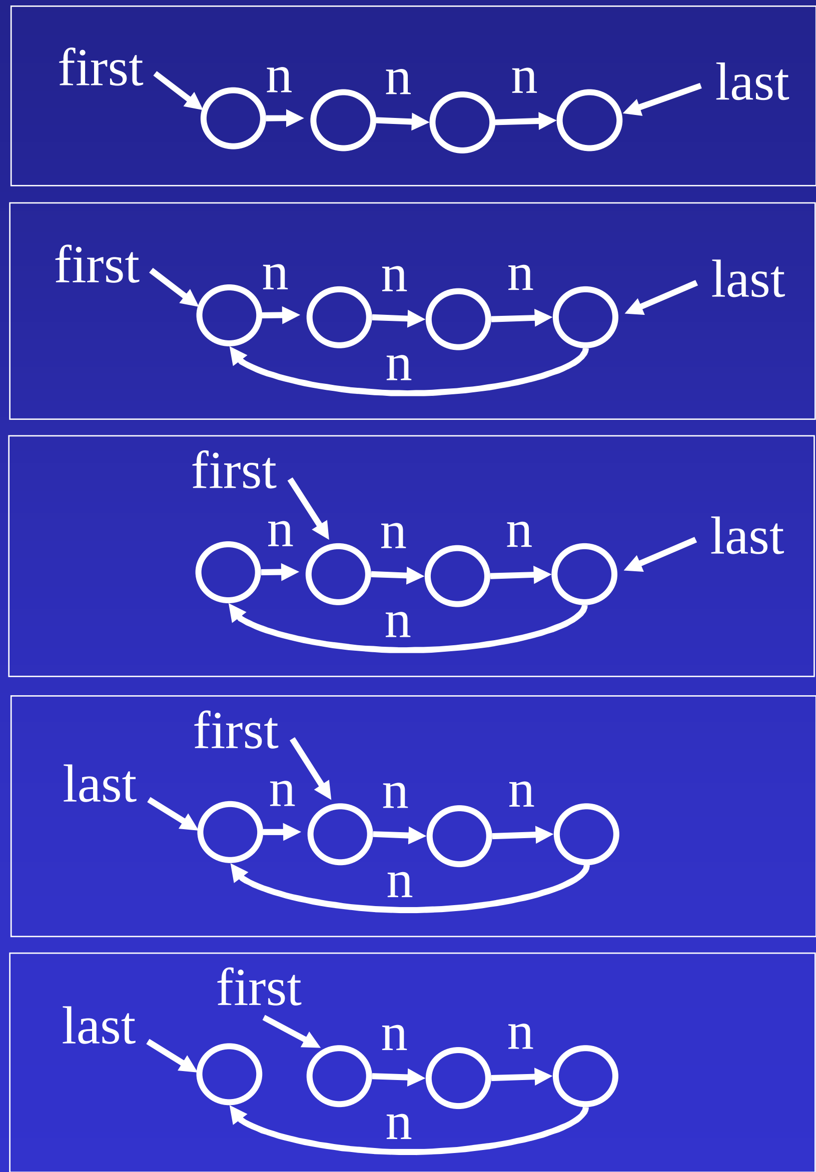
Imprecise ☹

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;
    }
}
```

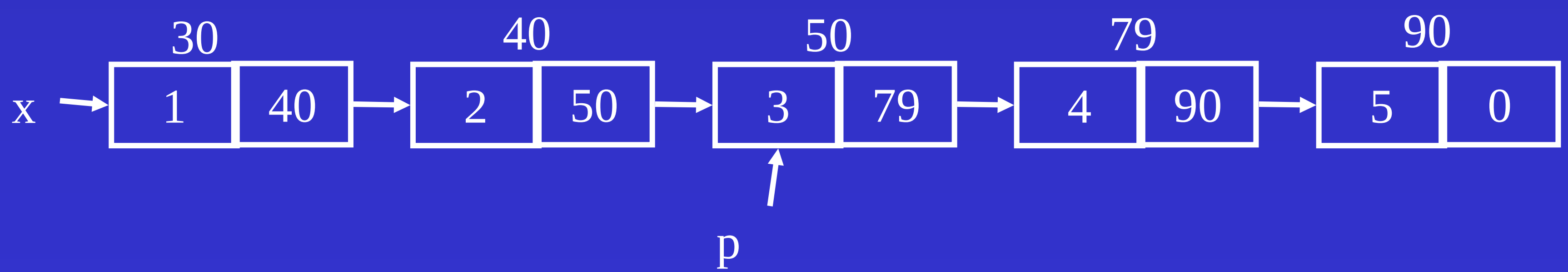


Plan

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

Traditional Heap Interpretation

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

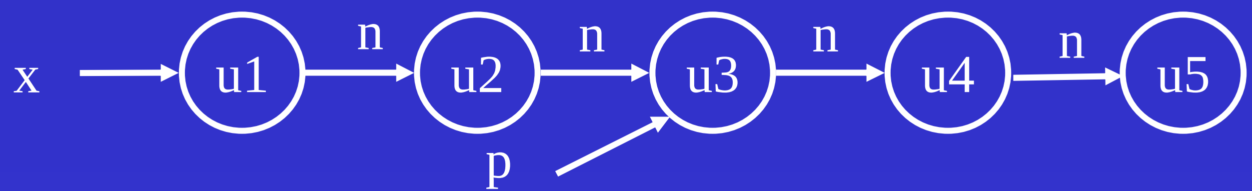


Predicate Logic

- 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
  - $p^S: (U^S)^k \rightarrow \{0, 1\}$
- FO^{TC} over TC,$\forall\exists\neg\wedge\vee$ 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}
  - next = {<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 $S=<U^S, p^S>$
- 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→U^S
- $[\![\varphi]\!]^S(z)$: {0, 1}

$$[\![p^*(v_1, v_2)]\!]^S(z) = \max_{\{u_1, \ldots, u_k \in U,\ Z(v_1)=u_1,\ Z(v_2)=u_2\}} \ \min_{\{1 \le i < k\}} p^S(u_i, u_{i+1})$$

Concrete Interpretation Rules (Pnueli)

Statement	Safety Precondition	Postcondition
x =NULL	1	$x' = \lambda(v).0$
x=malloc()	$\exists v_0: \neg active(v_0)$	$x' = \lambda v.\ eq(v, v_0)$ $active' = \lambda v.\ active(v) \lor eq(v, v_0)$
x=y	1	$x' = \lambda v.y(v)$
x=y →next	$\exists v_0: y(v_0) \land active(v_0)$	$x' = \lambda v.next(v_0, v)$
x →next=y	$\exists v_0: x(v_0) \land active(v_0)$	$next' = \lambda.v_1, v_2.$ $\neg eq(v_1, v_0) \land next(v_1, v_2)$ $\lor eq(v_1, v_0) \land y(v_2)$

Invariants

- No memory leaks

  $\forall v: active(v) \rightarrow \vee_{\{x \in PVar\}} \exists v_1: x(v_1) \wedge next^*(v_1, v)$

- Acyclic list(x)

  $\forall v_1,v_2: x(v_1) \wedge next^*(v_1, v_2) \rightarrow \neg next^+(v_2, v_1)$

- Reverse (x)

  $\forall v_1,v_2, v_3: x(v1) \wedge next^*(v_1, v_2) \rightarrow$

  $next(v_2, v_3) \leftrightarrow next'(v_3, v_2)$

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

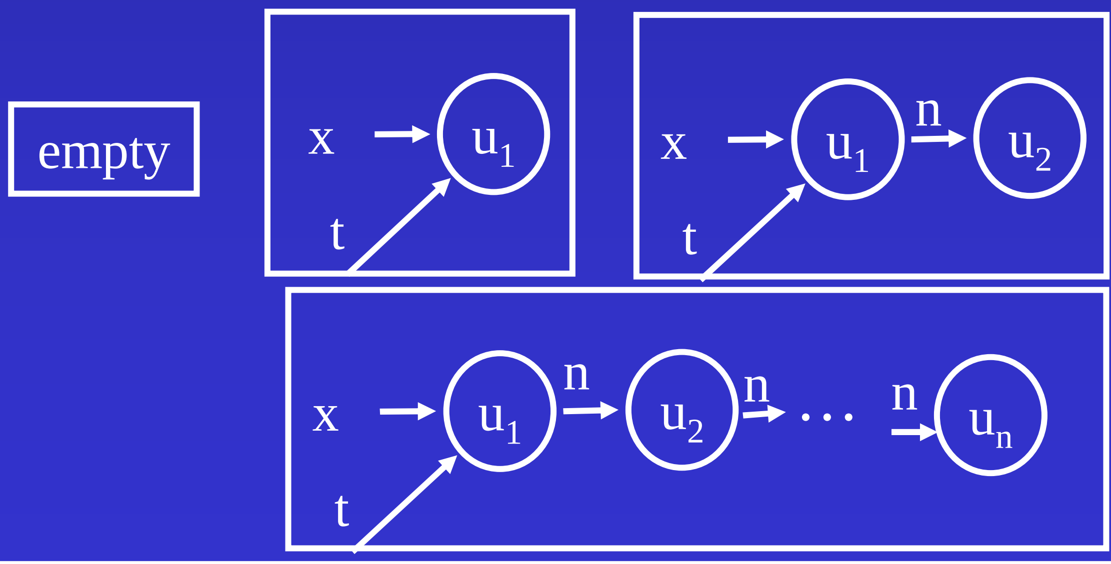
Why use logical structures?

- 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

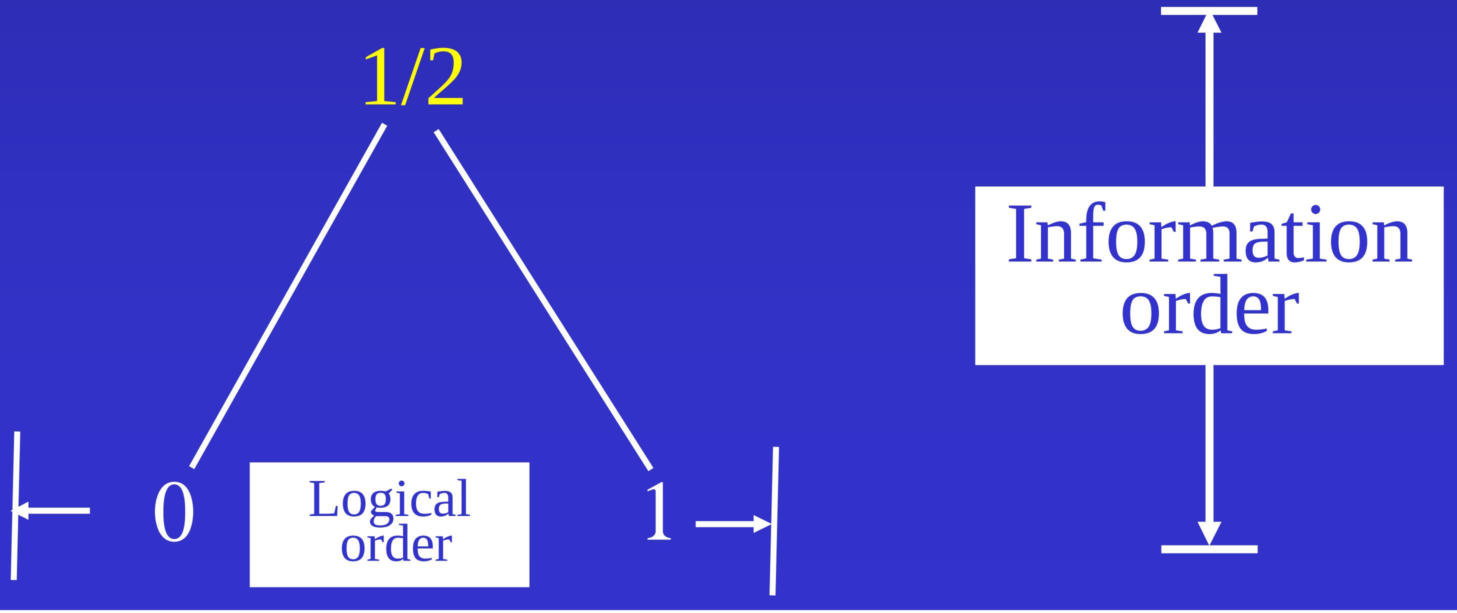
- Concrete interpretation
- Canonical abstraction

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

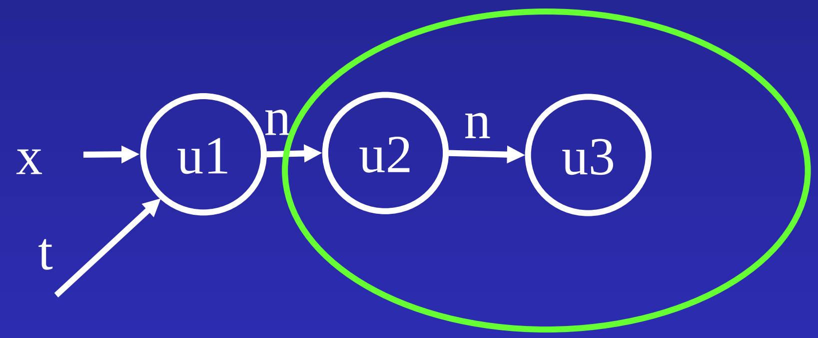
- A set of individuals (nodes) U
- Predicate meaning
  - $p^S: (U^S)^k \rightarrow \{0, 1, 1/2\}$

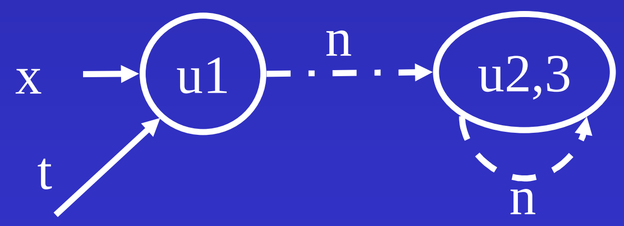
Canonical Abstraction

- 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

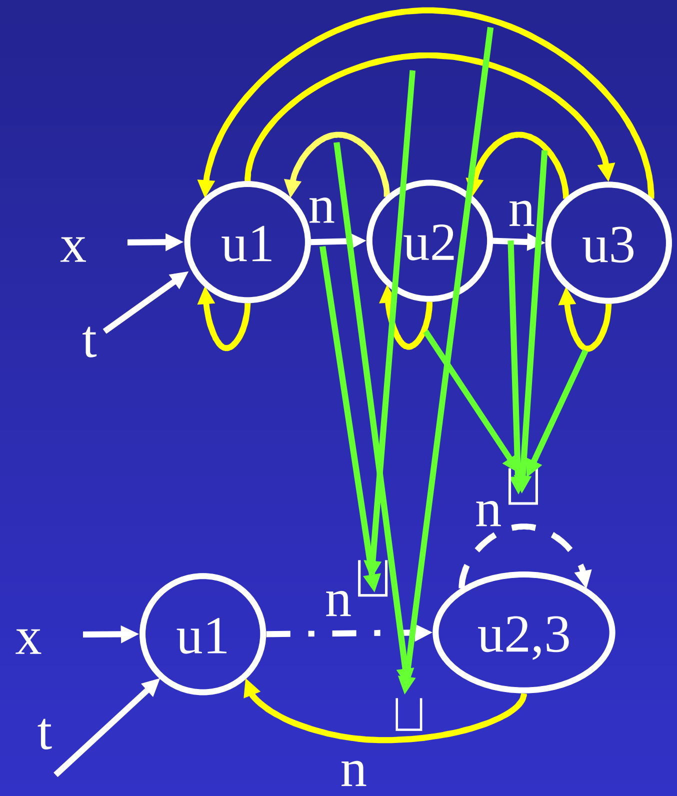
```
x = NULL;
while (…) do {
      t = malloc();
      t →next=x;
     x = t
}
```





Canonical Abstraction

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

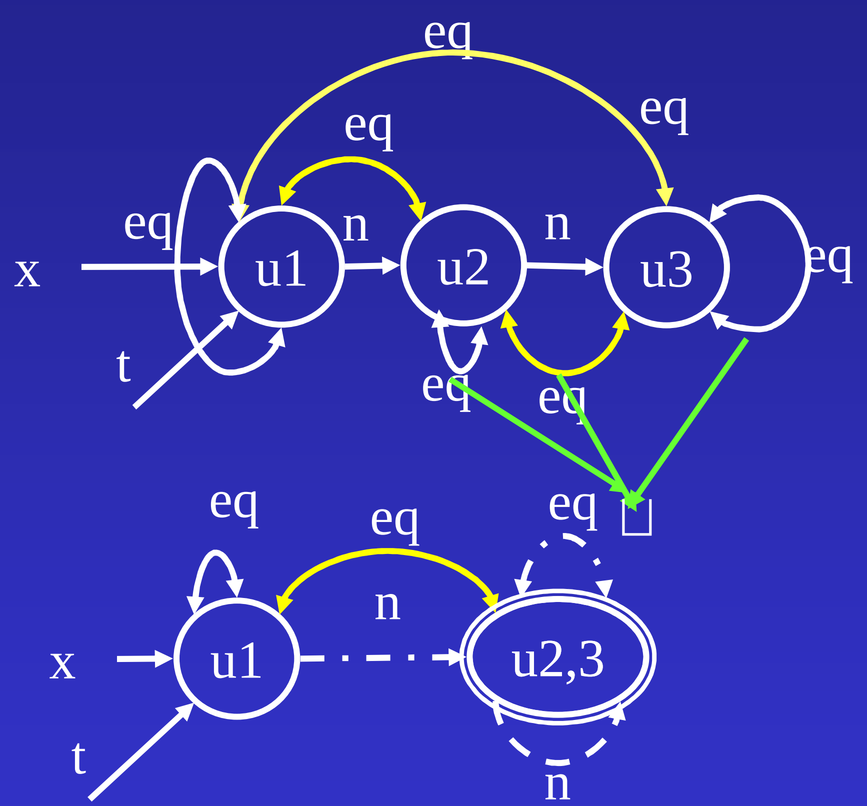


Canonical Abstraction and Equality

- 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

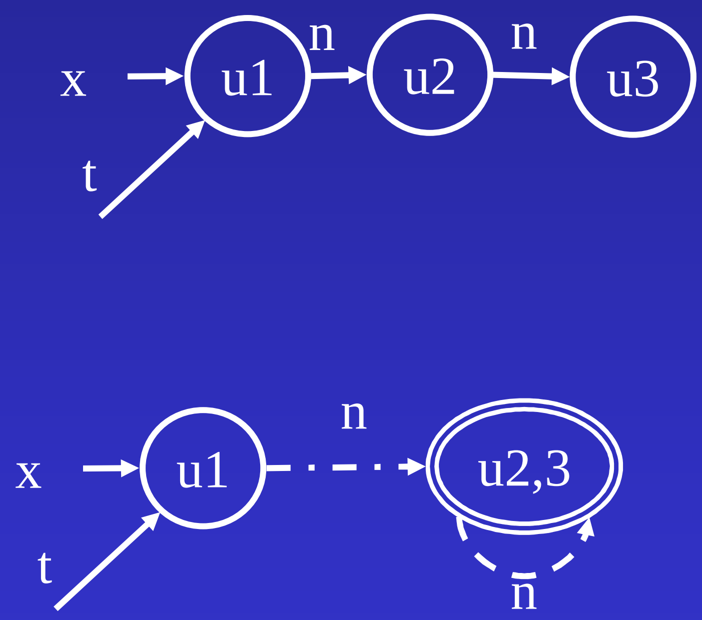
Canonical Abstraction and Equality

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



Canonical Abstraction

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



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?

Summary

- The embedding theorem eliminates the need for proving near commutavity
- Guarantees soundness
- Applied to arbitrary logics
- But can be imprecise