Lecture 07 – Shape Analysis

PROGRAM ANALYSIS & SYNTHESIS

Eran Yahav

Previously

- LFP computation and join-over-all-paths
- Inter-procedural analysis
  - call-string approach
  - functional approach

Today

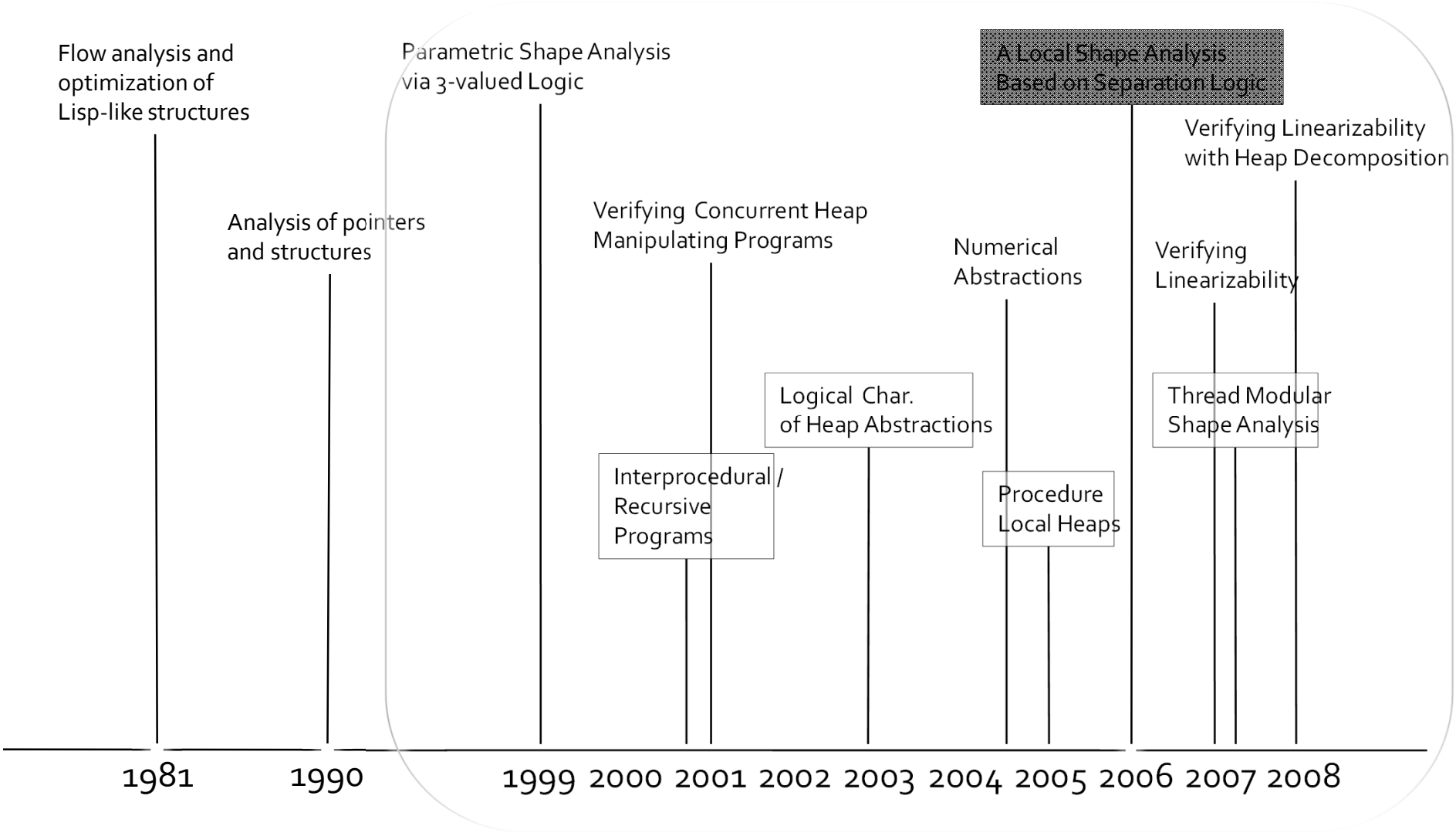
- Shape Analysis
- Typestate Verification

Shape Analysis

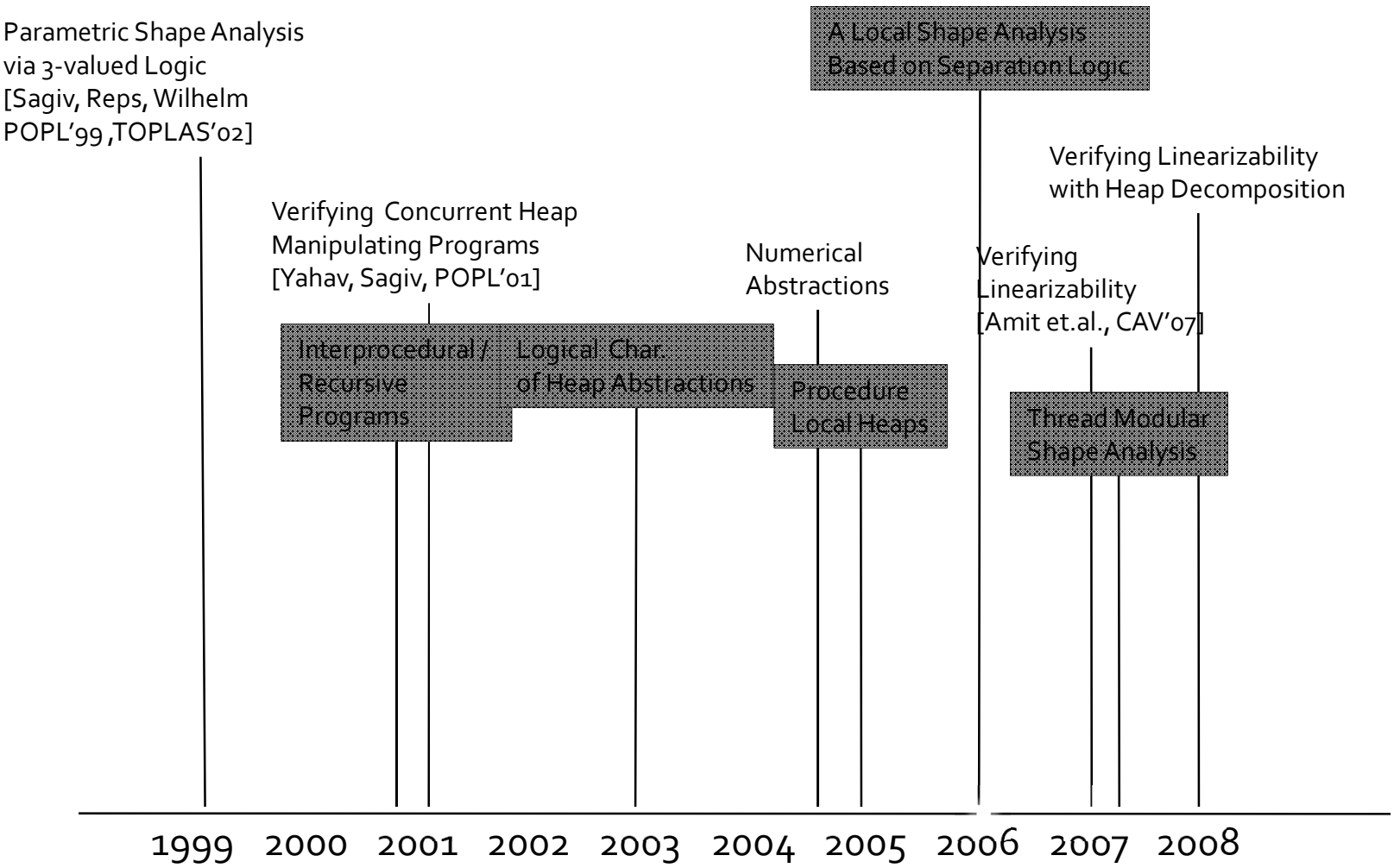
Automatically verify properties of programs manipulating dynamically allocated storage

Identify all possible shapes (layout) of the heap

Timeline: Shape Analysis



Timeline: Shape Analysis via 3-Valued Logic



Sequential Stack

```
void push (int v) {
 Node *x = malloc(sizeof(Node));
 x->d = v;
 x->n = Top;
 Top = x;
}
int pop() {
 if (Top == NULL) return EMPTY;
 Node *s = Top->n;
 int r = Top->d;
 Top = s;
 return r;
}
```

Want to Verify
No Null Dereference
Underlying list remains acyclic after each operation

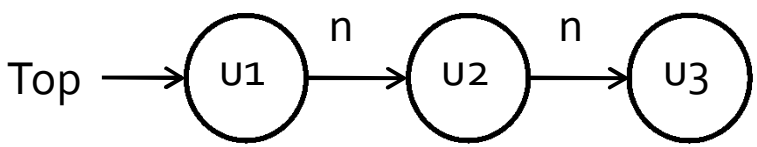
Shape Analysis via 3-valued Logic

1) Abstraction
   - 3-valued logical structure
   - canonical abstraction

2) Transformers
   - via logical formulae
   - soundness by construction
     - embedding theorem, [SRW02]

Concrete State

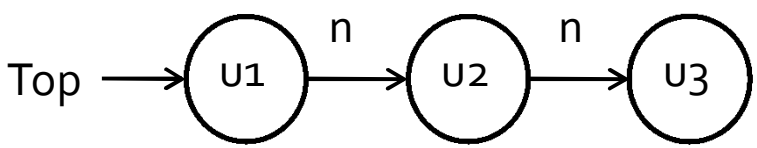
- represent a concrete state as a two-valued logical structure
  - Individuals = heap allocated objects
  - Unary predicates = object properties
  - Binary predicates = relations
- parametric vocabulary



(storeless, no heap addresses)

Concrete State

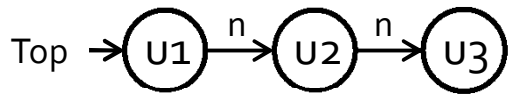
- S = <U, ι > over a vocabulary P
- U – universe
- ι - interpretation, mapping each predicate from p to its truth value in S



- U = { u1, u2, u3}
- P = { Top, n }
- ι(n)(u1,u2) = 1, ι(n)(u1,u3)=0, ι(n)(u2,u1)=0,…
- ι(Top)(u1)=1, ι(Top)(u2)=0, ι(Top)(u3)=0

Formulae for Observing Properties

```
void push (int v) {
 Node *x =
   malloc(sizeof(Node));
```



Top != null

$\exists w: Top(w)$ **1**

$\boxed{\exists w: x(w)}$

```
 x→d = v;
```

$\boxed{\exists w: x(w)}$

```
 x→n = Top;
```

No node precedes Top

$\neg\exists v1,v2: n(v1, v2) \wedge Top(v2)$ **1**

```
 Top = x;
}
```

No Cycles

$\neg\exists v1,v2: n(v1, v2) \wedge n^*(v2, v1)$ **1**

$\boxed{\neg\exists v1,v2: n(v1, v2) \wedge n^*(v2, v1)}$

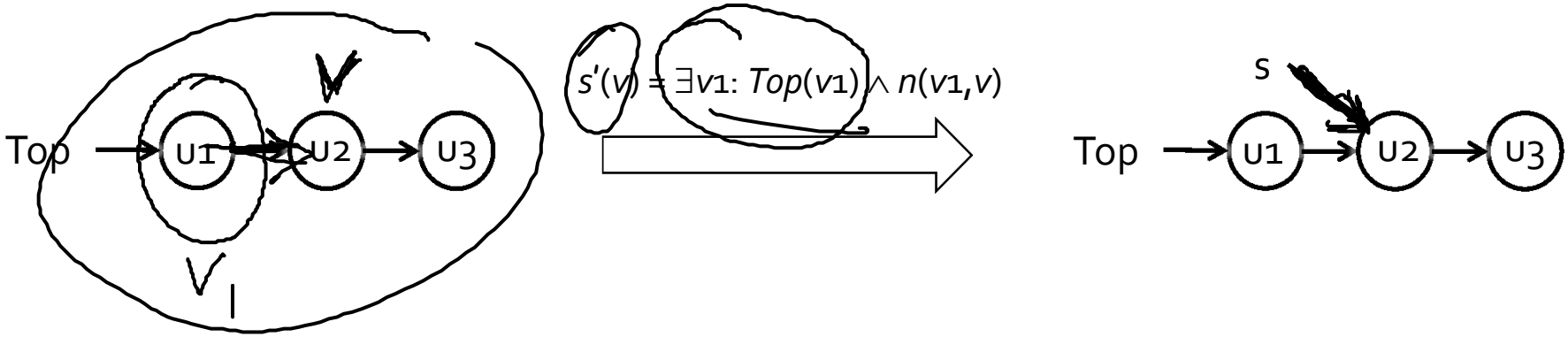
$\boxed{\neg\exists v1,v2: n(v1, v2) \wedge Top(v2)}$

Concrete Interpretation Rules

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

n	v_1	v_2	v_3
v_1			
v_2			
v_3			

Example: $s = Top \rightarrow n$



Top	
u1	1
u2	0
u3	0

n	u1	u2	u3
u1	0	1	0
u2	0	0	1
u3	0	0	0

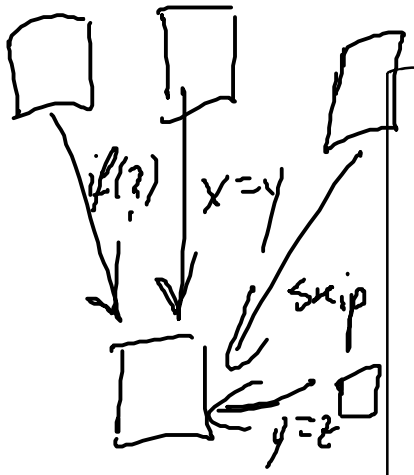
Top	
u1	1
u2	0
u3	0

n	u1	u2	u3
u1	0	1	0
u2	0	0	1
u3	0	0	0

s	
u1	0
u2	0
u3	0

s	
u1	0
u2	1
u3	0

Collecting Semantics



$$
\text{CSS}[v] = \begin{cases}
\{ \langle \emptyset, \emptyset \rangle \} & \text{if } v = \text{entry} \\
\bigcup \{ [\![st(w)]\!](S) \mid S \in \text{CSS}[w] \} \cup & \\
\quad (w,v) \in E(G), & \\
\quad w \in \text{Assignments}(G) & \\
\bigcup \{ S \mid S \in \text{CSS}[w] \} \cup & \\
\quad (w,v) \in E(G), & \\
\quad w \in \text{Skip}(G) & \\
\bigcup \{ S \mid S \in \text{CSS}[w] \text{ and } S \vDash \text{cond}(w) \} \cup & \text{othrewise} \\
\quad (w,v) \in \text{True-Branches}(G) & \\
\bigcup \{ S \mid S \in \text{CSS}[w] \text{ and } S \vDash \neg\text{cond}(w) \} & \\
\quad (w,v) \in \text{False-Branches}(G) &
\end{cases}
$$

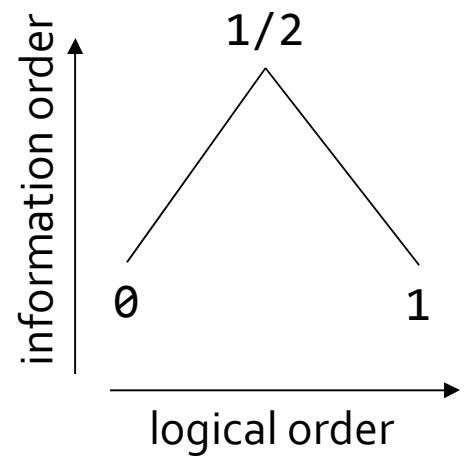
Collecting Semantics

- At every program point – a potentially infinite set of two-valued logical structures
- Representing (at least) all possible heaps that can arise at the program point

- Next step:
  find a bounded abstract representation

3-Valued Logic

- 1 = true
- 0 = false
- 1/2 = unknown



- A join semi-lattice, $0 \sqcup 1 = 1/2$

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: $0 \sqcup 1 = 1/2$

Boolean Connectives [Kleene]

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

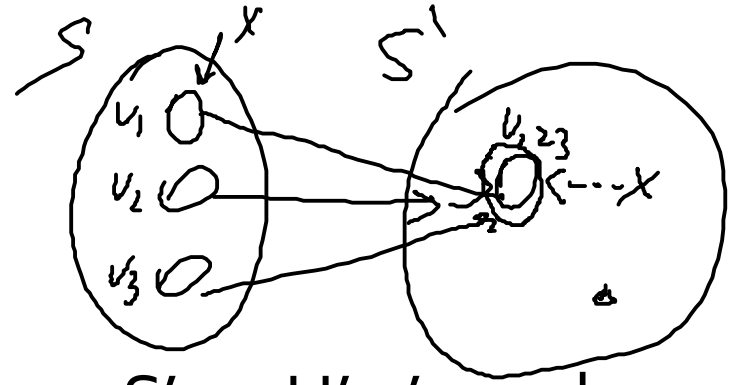
$\vee$	0	1/2	1
0	0	1/2	1
1/2	1/2	1/2	1
1	1	1	1

Property Space

- 3-struct[P] = the set of 3-valued logical structures over a vocabulary (set of predicates) P

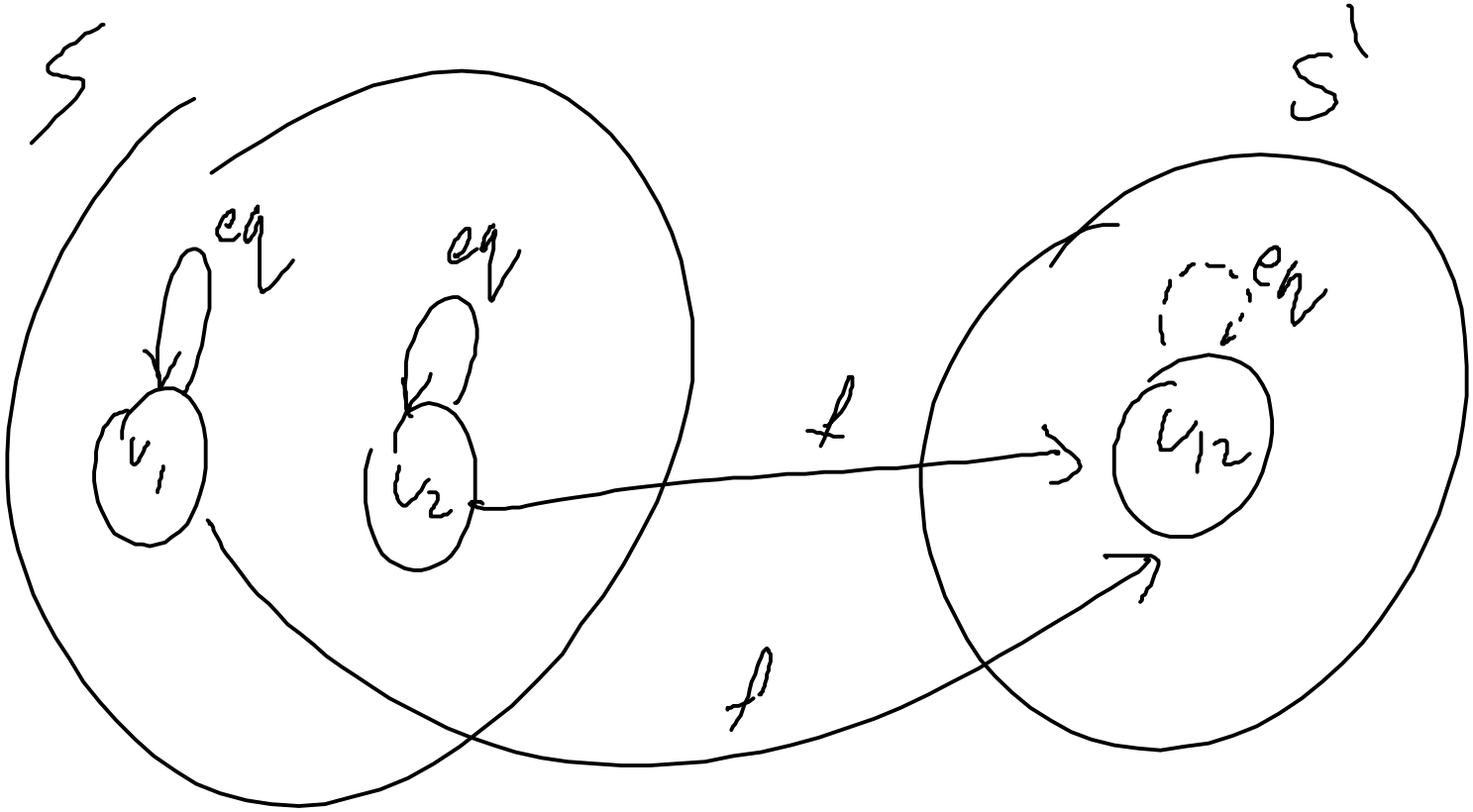
- Abstract domain
  - $\wp$(3-Struct[P])
  - $\sqsubseteq$ is $\subseteq$
    - We will see alternatives later (maybe)

Embedding Order



- Given two structures S = <U, ι >, S′ = <U′, ι'> and an onto function f : U ⟶ U′ mapping individuals in U to individuals in U′
- We say that f embeds S in S′ (denoted by S $\sqsubseteq^f$ S′) if
  - for every predicate symbol p ∈ P of arity k: u1, …, uk ∈ U, ι(p)(u1, …, uk) ⊑ ι'(p)(f(u1), …, f (uk))
  - and for all u′ ∈ U′
    (| { u | f (u) = u′ } | > 1) ⊑ ι'(sm)(u′)

- We say that S can be embedded in S′ (denoted by S ⊑ S′) if there exists a function f such that S $\sqsubseteq^f$ S′


S
S'
eq
eq
eq
v_1
v_2
v_{12}
f
f

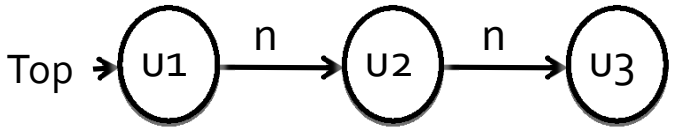
Tight Embedding

- $S' = \langle U', \iota' \rangle$ is a tight embedding of $S = \langle U, \iota \rangle$ with respect to a function f if:
  - S' does not lose unnecessary information

$$\iota'(u'_1, \ldots, u'_k) = \sqcup\{\iota(u_1 \ldots, u_k) \mid f(u_1)=u'_1, \ldots, f(u_k)=u'_k\}$$

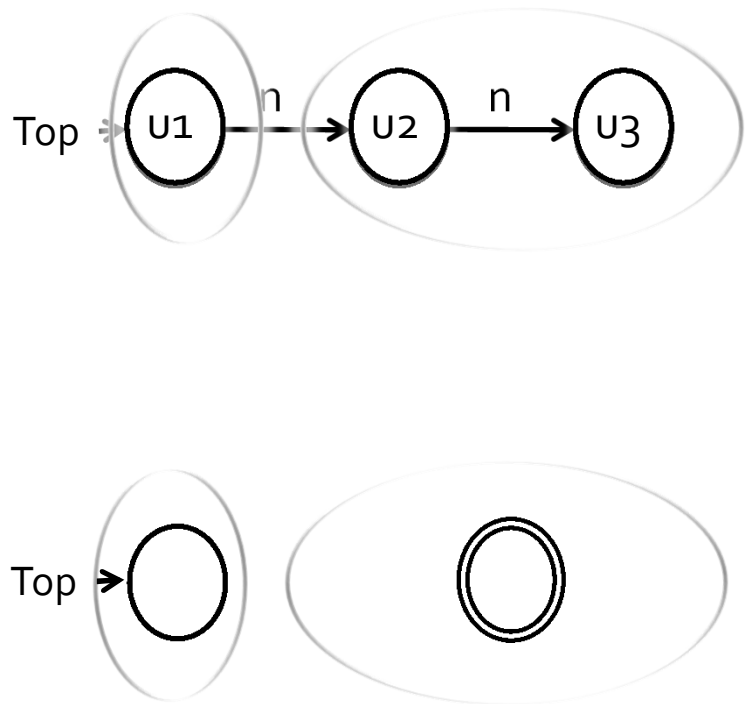
- One way to get tight embedding is canonical abstraction

Canonical Abstraction



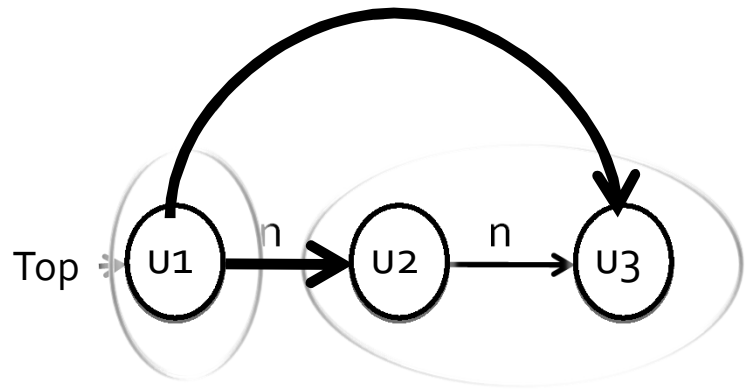
[Sagiv, Reps, Wilhelm, TOPLAS02]

Canonical Abstraction



[Sagiv, Reps, Wilhelm, TOPLAS'02]

Canonical Abstraction

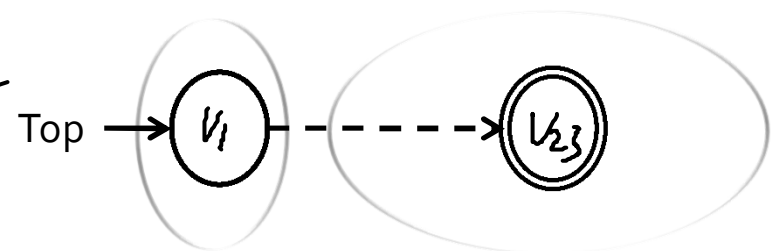


$n(v_1, v_{23}) = ?$

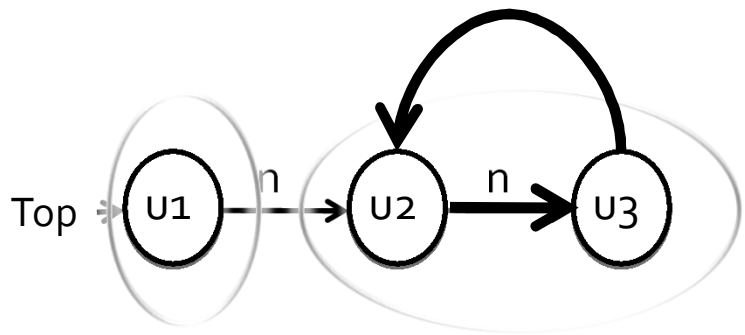
$\sqcup(n(u_1, u_2) = 1$

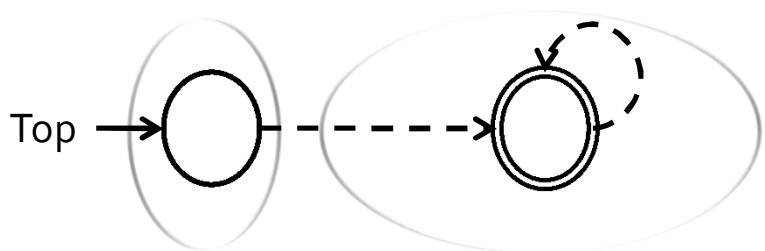
$n(v_1, v_3))$

$= 1/2$

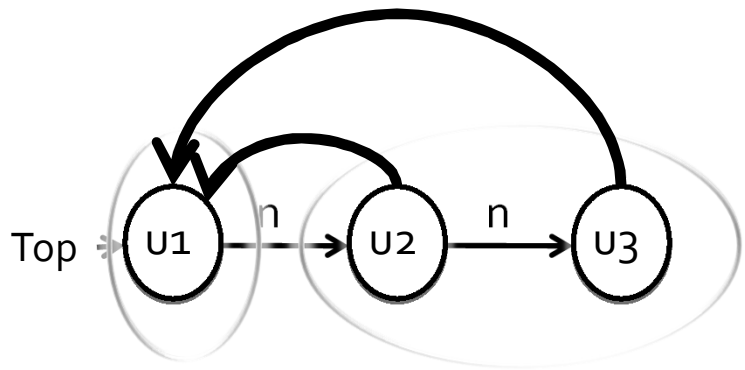


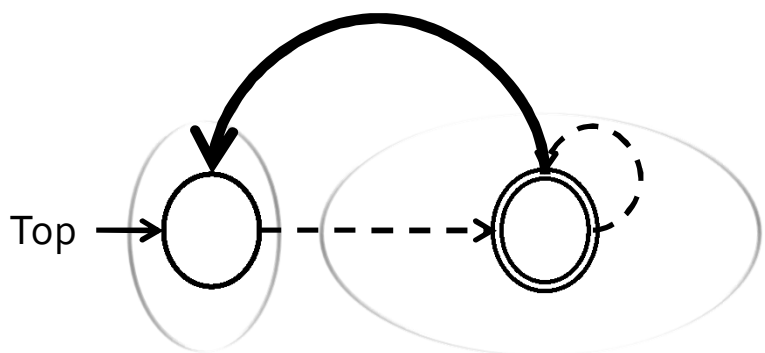
Canonical Abstraction



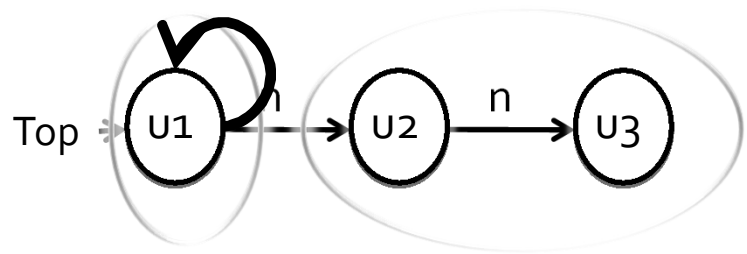


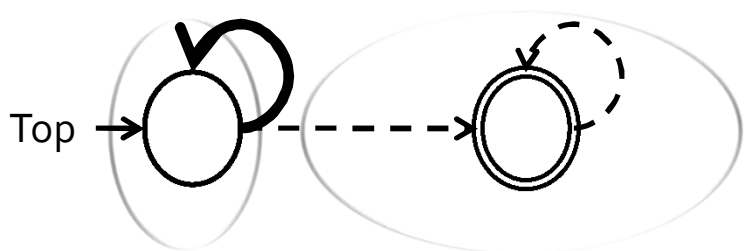
Canonical Abstraction





Canonical Abstraction





Canonical Abstraction (β)

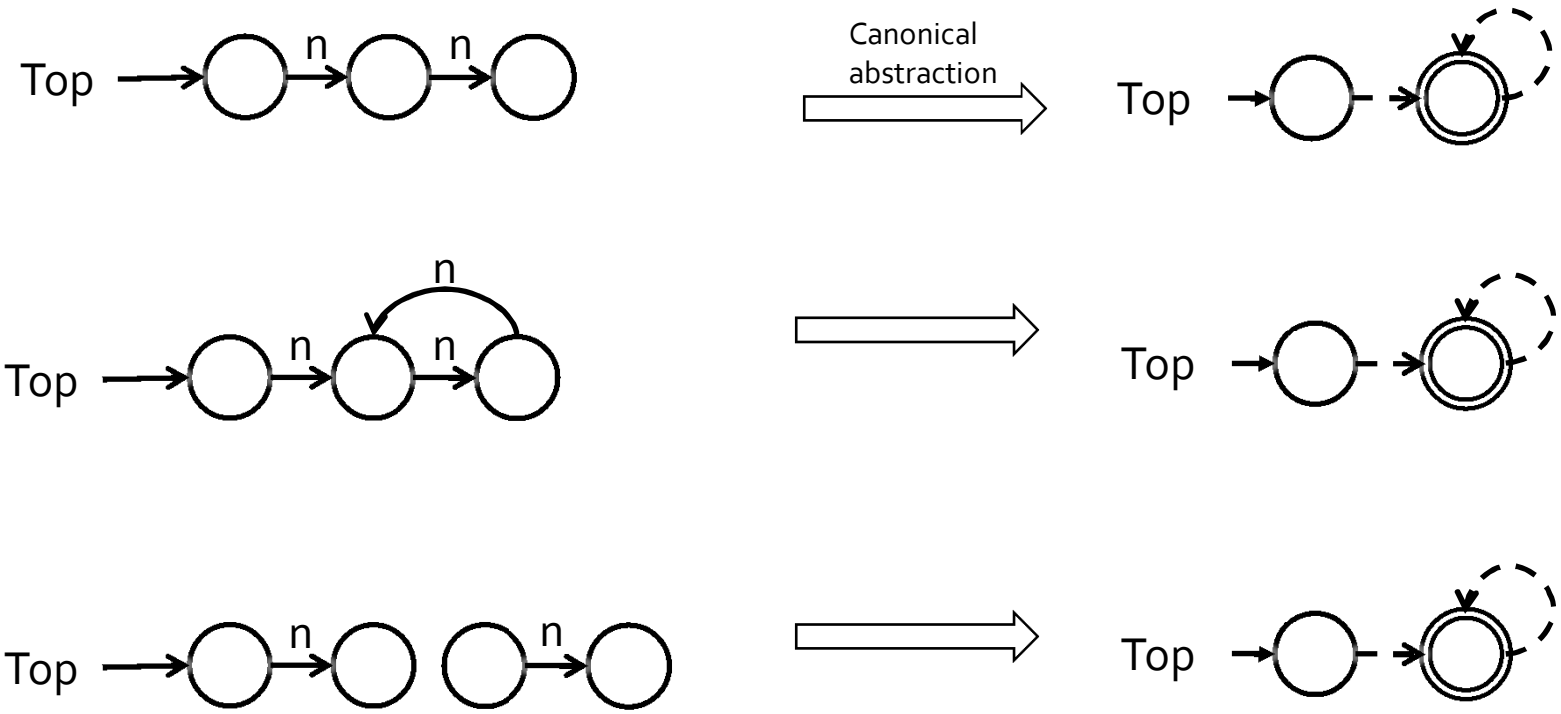
- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values

$$\iota'(u'_1,\ldots,u'_k) = \sqcup \{\iota(u_1,\ldots,u_k) \mid f(u_1)=u'_1,\ldots,f(u_k)=u'_k\}$$

- Converts a state of arbitrary size into a 3-valued abstract state of bounded size

- $\alpha(C) = \sqcup \{ \beta(c) \mid c \in C \}$

Information Loss

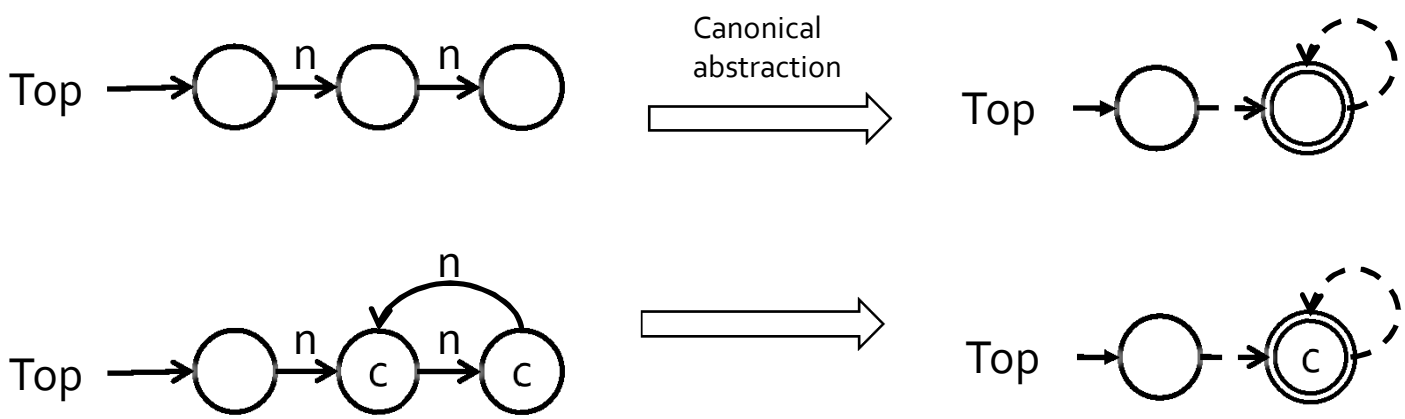


Instrumentation Predicates

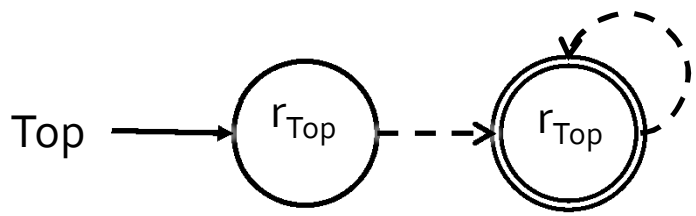
- Record additional derived information via predicates

$$r_x(v) = \exists v1: x(v1) \wedge n^*(v1,v)$$

$$c(v) = \exists v1: n(v1, v) \wedge n^*(v, v1)$$



Embedding Theorem:
Conservatively Observing Properties



No Cycles

$\neg\exists v1,v2: n(v1, v2) \wedge n^*(v2, v1)$ **1/2**

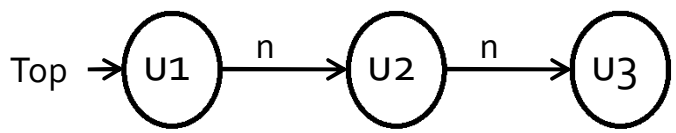
No cycles (derived)

$\forall v: \neg c(v)$ **1**

Operational Semantics

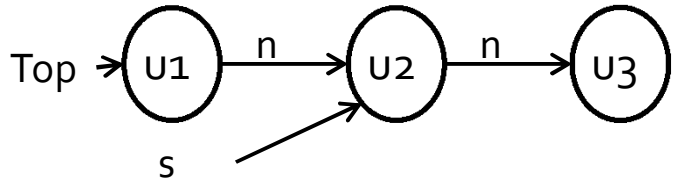
```
void push (int v) {
 Node *x = malloc(sizeof(Node));
 x->d = v;
 x->n = Top;
 Top = x;
}

int pop() {
 if (Top == NULL) return EMPTY;
 Node *s = Top->n;
 int r = Top->d;
 Top = s;
 return r;
}
```

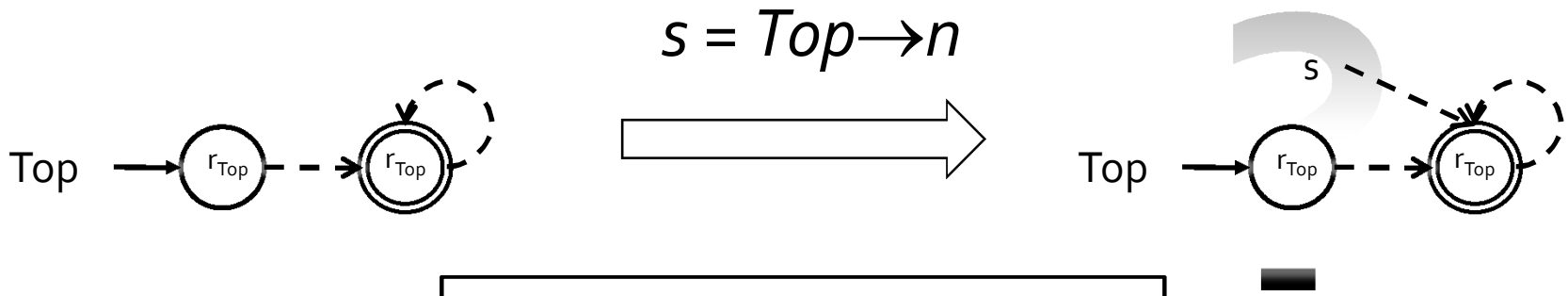


$[\![\texttt{s = Top->n}]\!]$

$s'(v) = \exists v1\colon Top(v1) \wedge n(v1,v)$



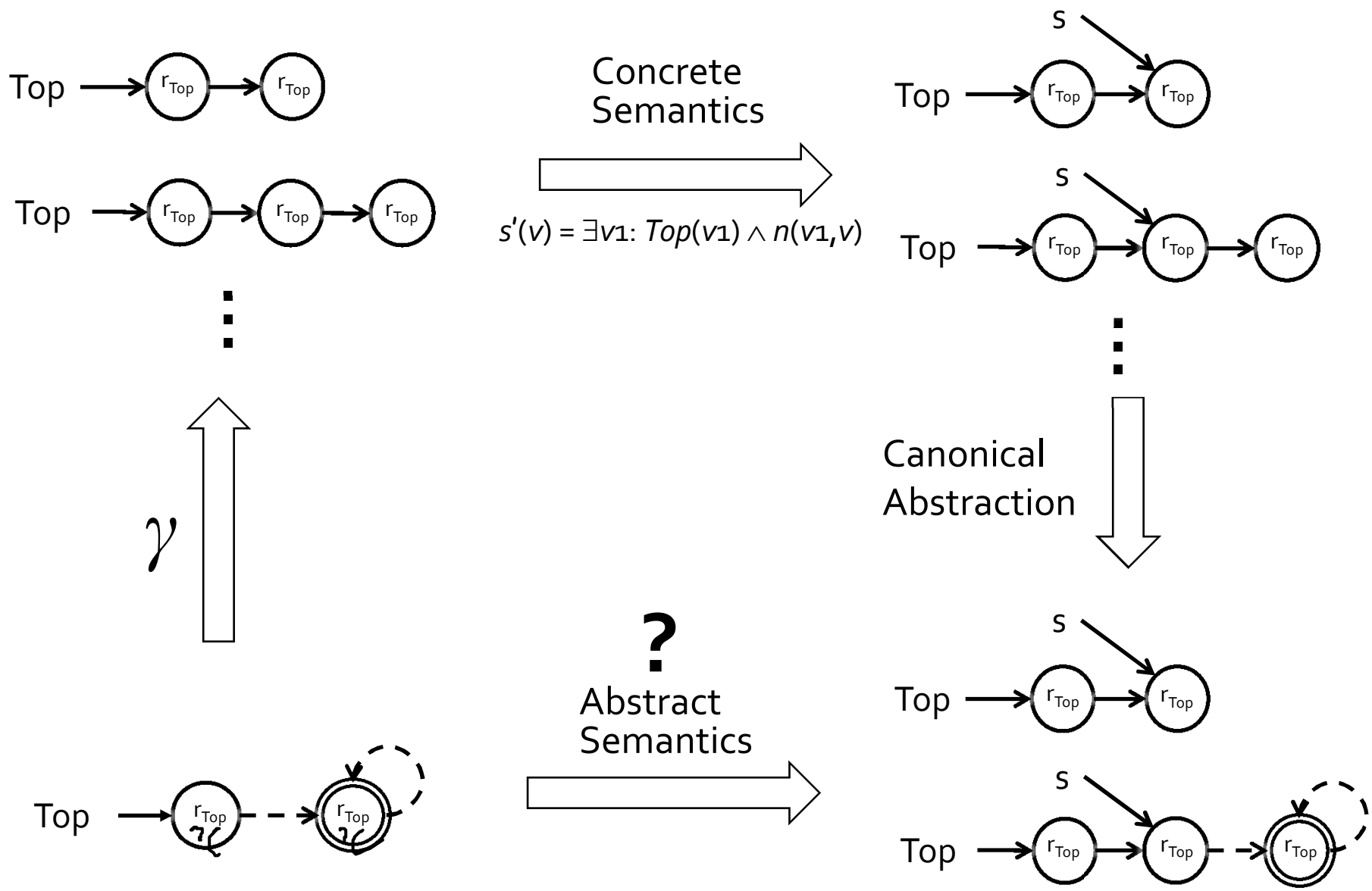
Abstract Semantics

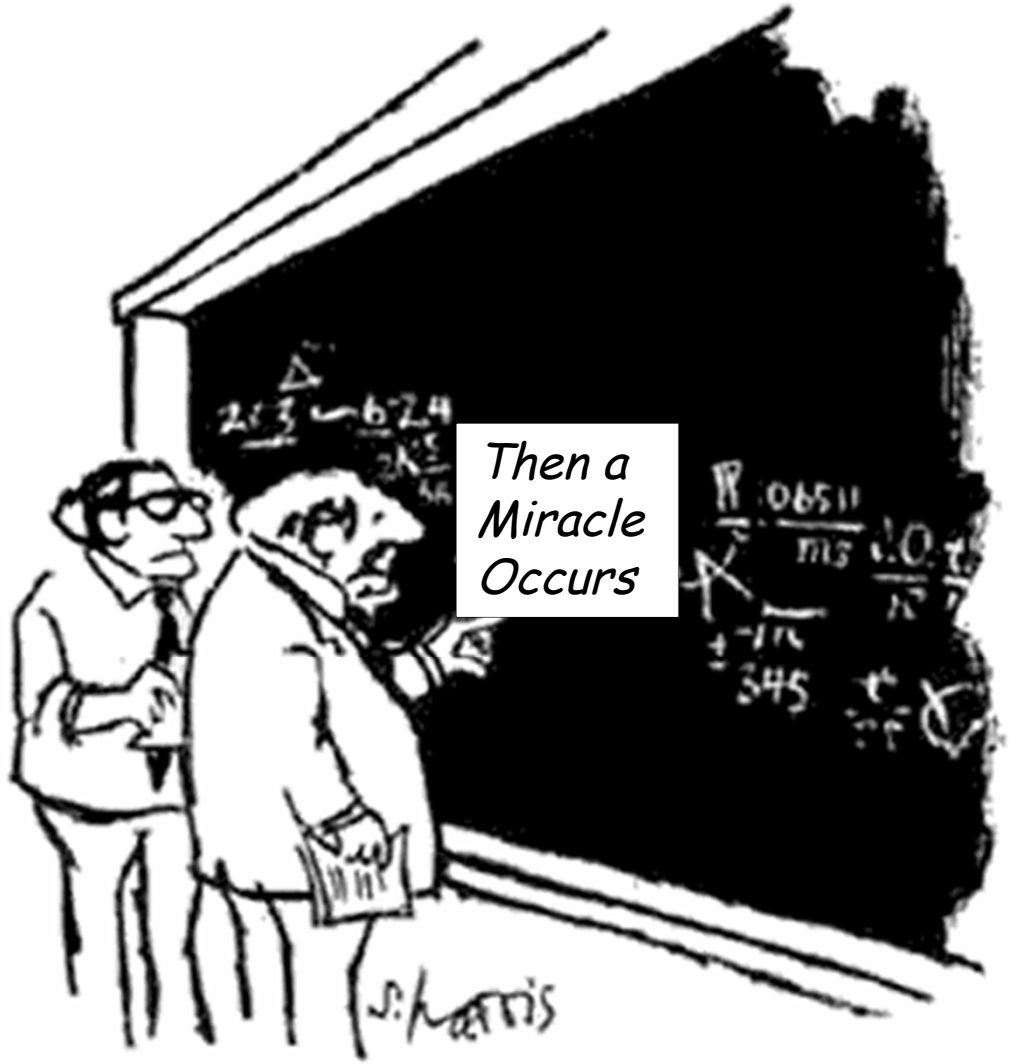


$[\![$`s = Top->n`$]\!]$

$s'(v) = \exists v1\colon Top(v1) \wedge n(v1,v)$

Best Transformer ($s = Top \rightarrow n$)

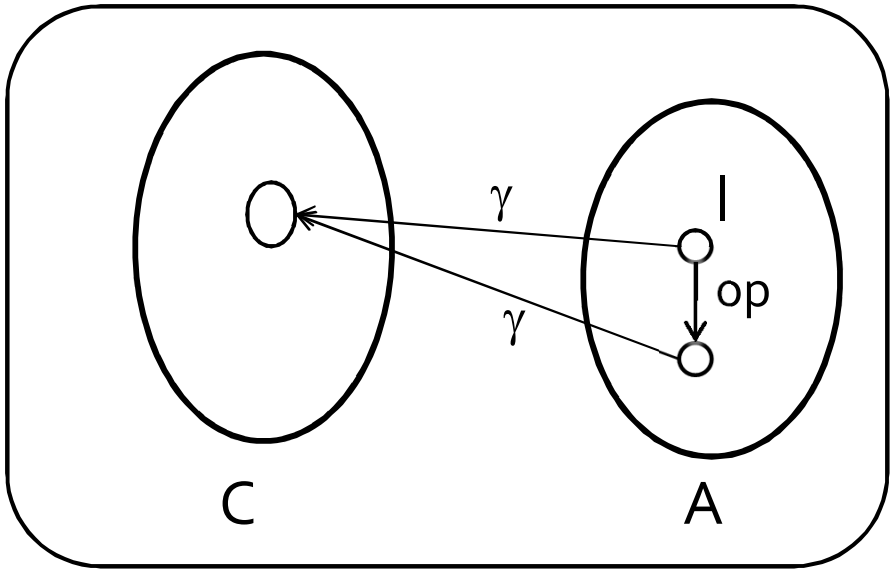




"I THINK YOU SHOULD BE MORE EXPLICIT HERE IN STEP TWO."

Semantic Reduction

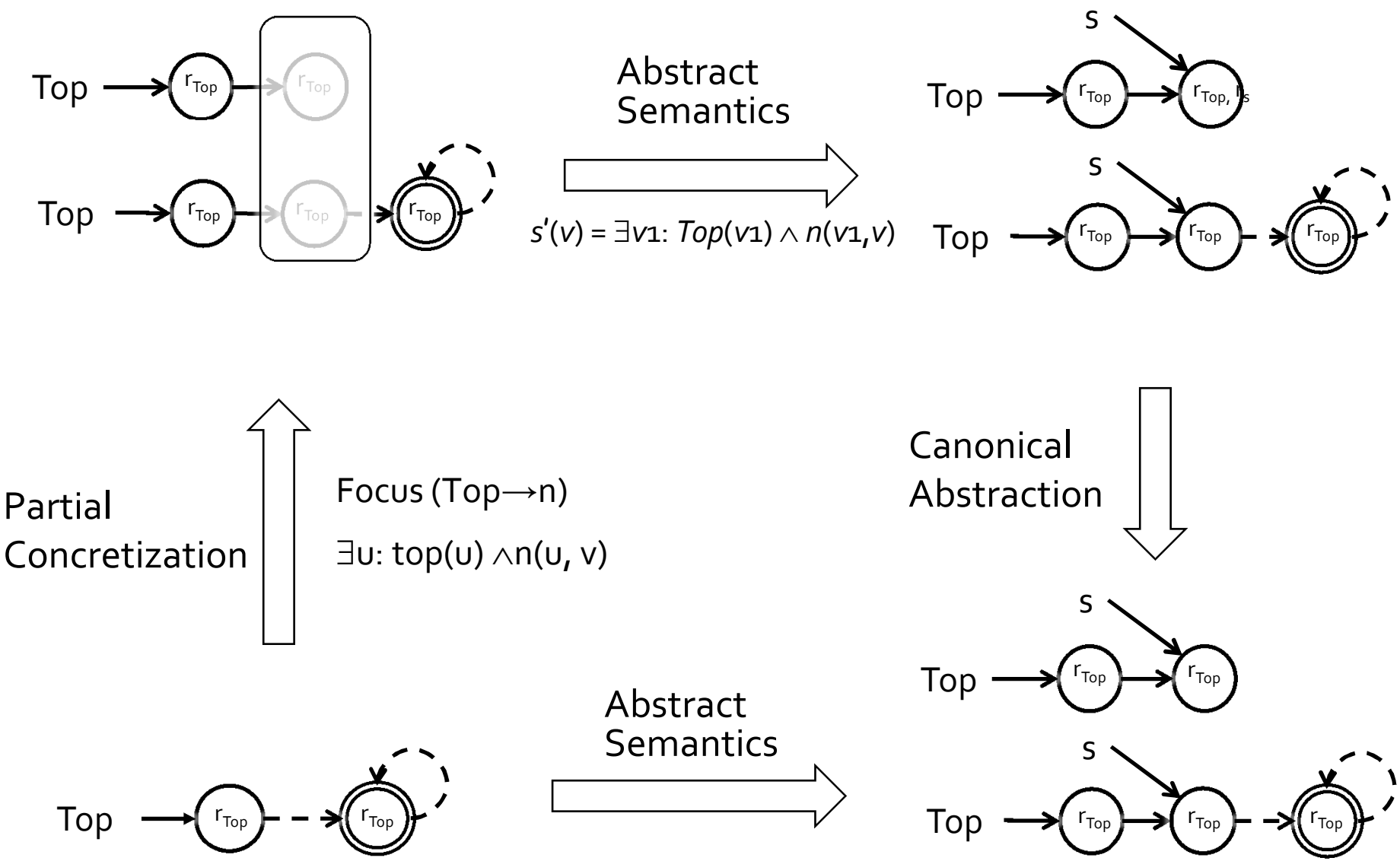
- Improve the precision of the analysis by recovering properties of the program semantics
- A Galois connection (C, α, γ, A)
- An operation op:A$\rightarrow$A is a semantic reduction when
  - $\forall l \in L_2$ op(l)$\sqsubseteq$l and
  - γ(op(l)) = γ(l)



The Focus Operation

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

Partial Concretization Based on Transformer (s=$Top \rightarrow n$)



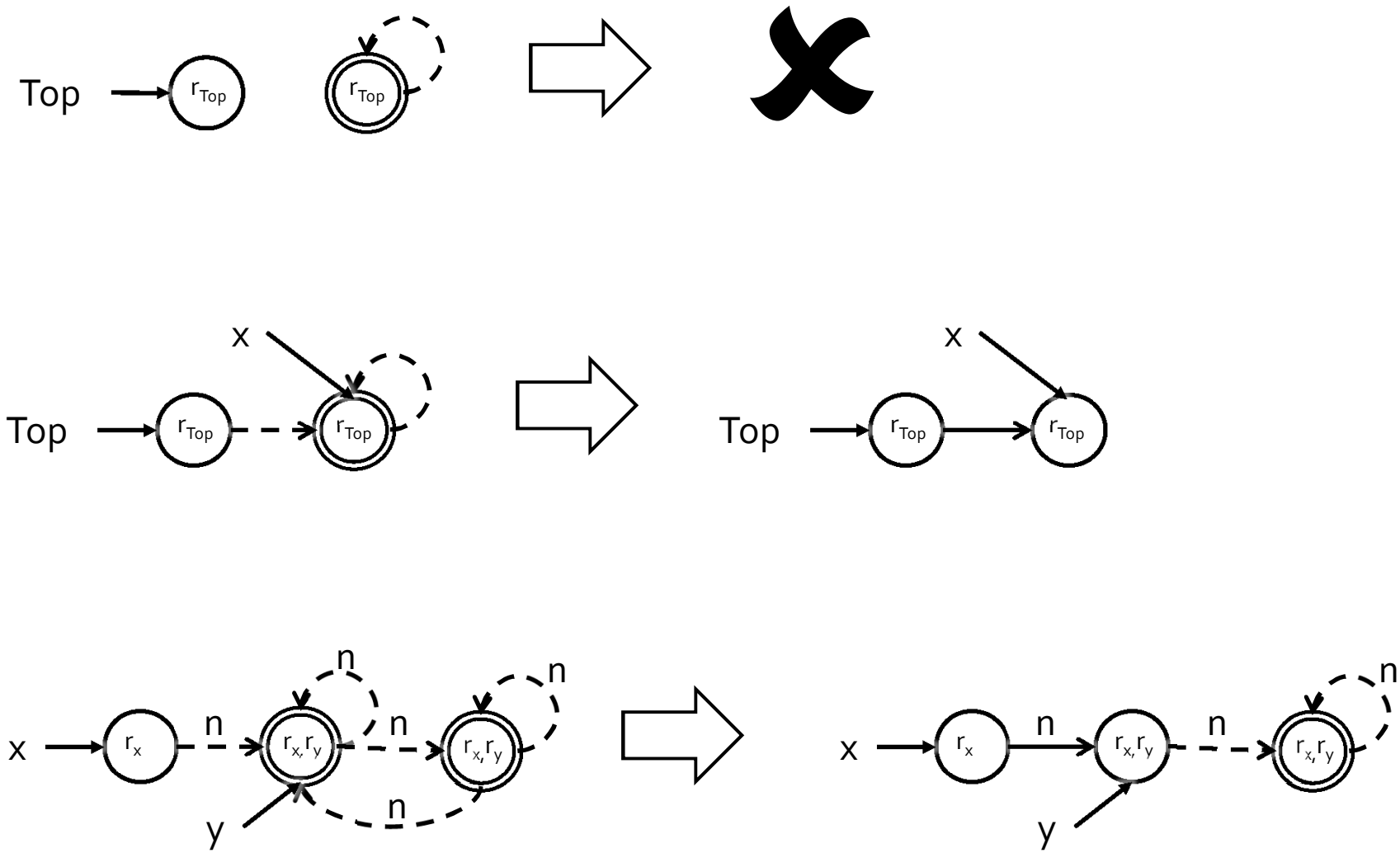
Partial Concretization

- Locally refine the abstract domain per statement
- Soundness is immediate
- Employed in other shape analysis algorithms
  [Distefano et.al., TACAS'06, Evan et.al., SAS'07, POPL'08]
- Emplyed in other analysis algorithms
  [Typestate verification, ISSTA'06]

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

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

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

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

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

Abstract Transformers: Summary

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

Abstract Semantics

$$
SS[v] = \begin{cases}
\{ \langle \emptyset, \emptyset \rangle \} & \text{if } v = \text{entry} \\
\bigcup_{\substack{(w,v) \in E(G), \\ w \in \text{Assignments}(G)}} \{ \text{t\_embed}(\text{coerce}([\![\text{st}(w)]\!]_3(\text{focus}_{F(w)}(SS[w]))))\} \cup & \\
\bigcup_{\substack{(w,v) \in E(G), \\ w \in \text{Skip}(G)}} \{ S \mid S \in SS[w] \} \cup & \text{othrewise} \\
\bigcup_{(w,v) \in \text{True-Branches}(G)} \{ \text{t\_embed}(S) \mid S \in \text{coerce}([\![\text{st}(w)]\!]_3(\text{focus}_{F(w)}(SS[w]))) \text{ and } S \models_3 \text{cond}(w)\} \cup & \\
\bigcup_{(w,v) \in \text{False-Branches}(G)} \{ \text{t\_embed}(S) \mid S \in \text{coerce}([\![\text{st}(w)]\!]_3(\text{focus}_{F(w)}(SS[w]))) \text{ and } S \models_3 \neg\text{cond}(w)\} \cup &
\end{cases}
$$

Recap

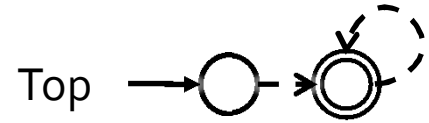
- Abstraction
  - canonical abstraction
  - recording derived information

- Transformers
  - partial concretization (focus)
  - constraint solver (coerce)
  - sound information extraction

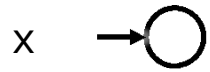
Stack Push

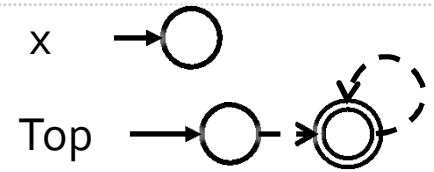
emp

...




```
void push (int v) {
 Node *x =
   alloc(sizeof(Node));
```



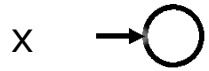


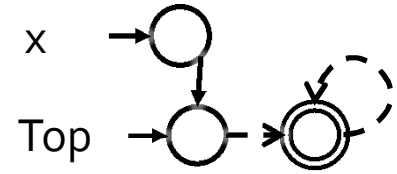
$\exists v: x(v)$

```
 x→d = v;
```

$\exists v: x(v)$

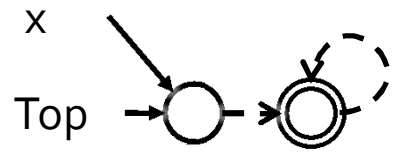
```
 x→n = Top;
```





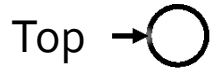

```
 Top = x;
```

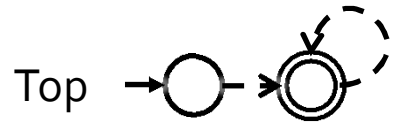
$\neg\exists v1,v2: n(v1, v2) \wedge Top(v2)$




```
}
```

$\forall v: \neg c(v)$





Non-blocking Stack [Treiber 1986]

```
#define EMPTY -1

typedef int data_type;

typedef struct node t {
    data_type d;
    struct node t *n
} Node;

typedef struct stack t {
    struct node t *Top;
} 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 = t->d;
  } while (!CAS(&S->Top,t,s));
  return r;
}
```

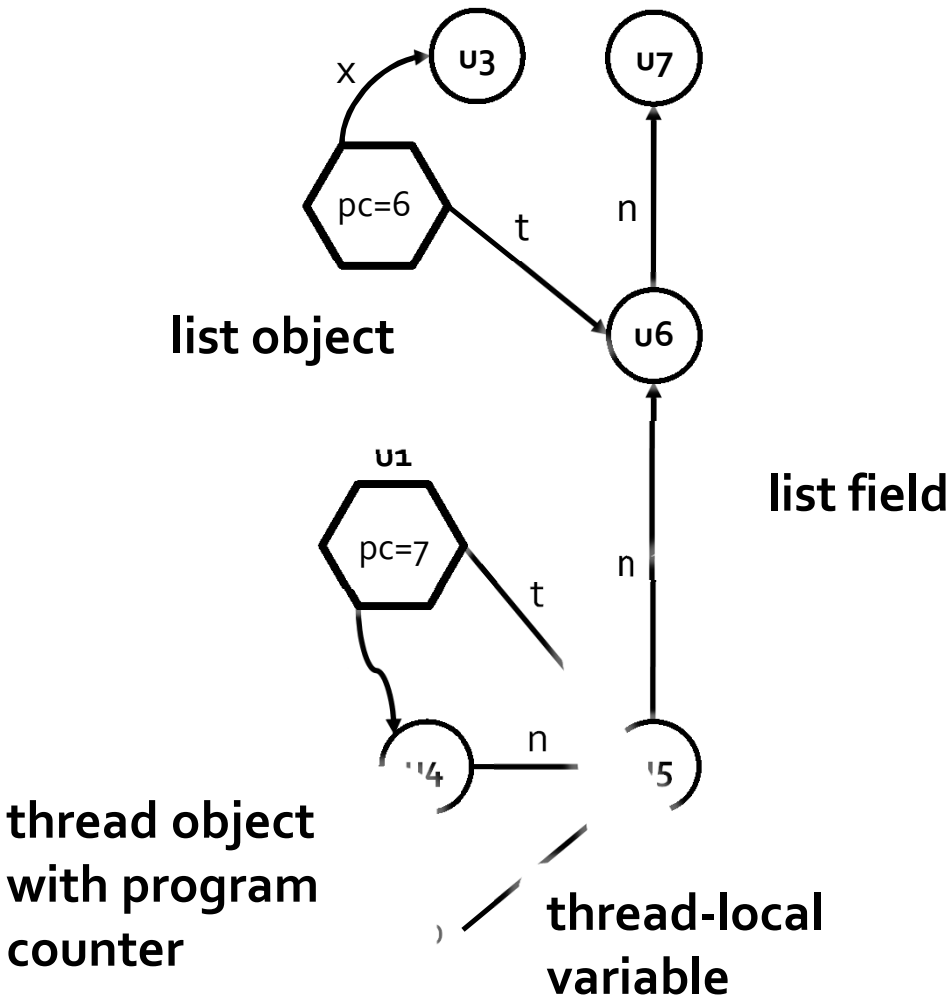
Concurrent Shape Analysis

- a thread is represented as a thread object
- add predicates to vocabulary

- Recipe
  1) abstraction: canonical abstraction
  2) transformers: interleaving + as before

- Bounded threads
  - Static thread names
- Unbounded threads
  - thread objects abstracted via canonical abstraction

Concrete State



U = { u1, u2, u3,…,u7 }

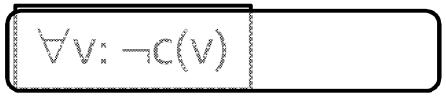
isThread = { u1,u2 }

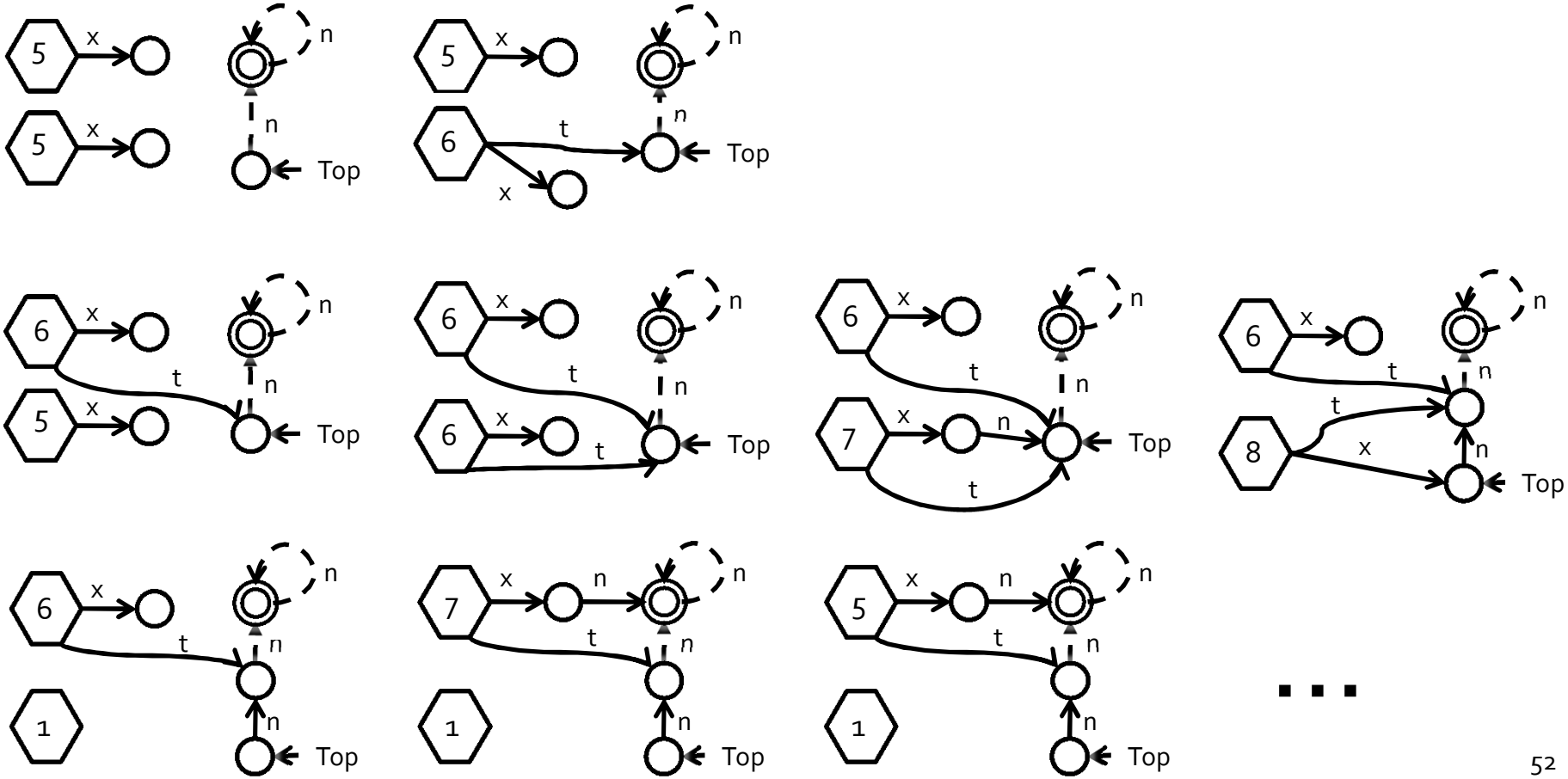
at[pc=1] = { }
…
at[pc=6] = { u3 }
at[pc=7]= { u1 }

Top = { u5 }
…
x = { (u1,u4), (u2,u3)}
t={ (u1,u5),(u2,u6)}
n = { (u5,u6), (u6,u7) }
t 1= { u1 } t2 = { u2 }

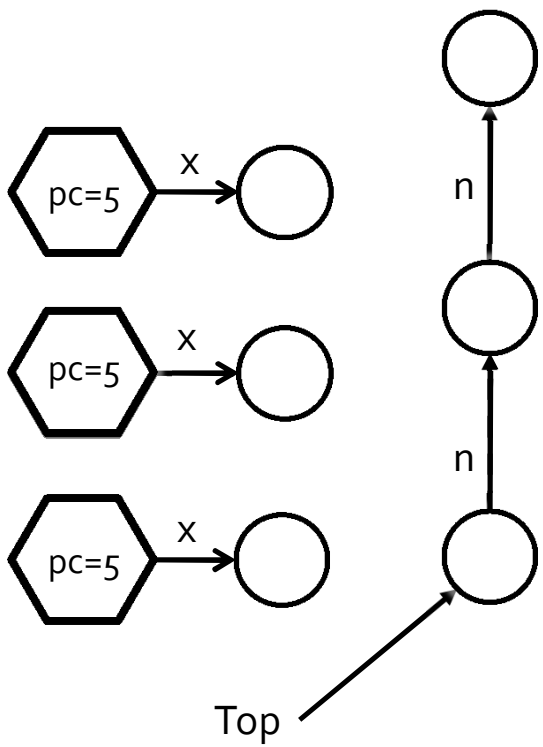
Exploration

```
[1] void push(Stack *S, data_type v) {
[2]   Node *x = alloc(sizeof(Node));
[3]   x->d = v;
[4]   do {
[5]     Node *t = S->Top;
[6]     x->n = t;
[7]   } while (!CAS(&S->Top,t,x));
[8] }
```



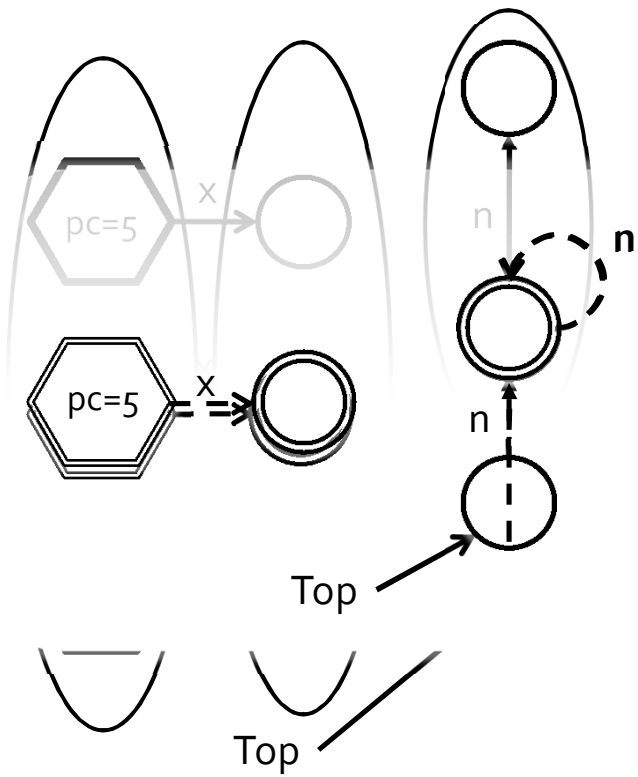


Representing an Unbounded Number of Threads



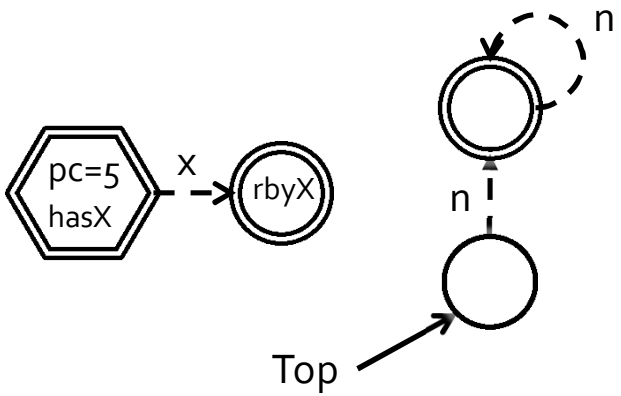

```
[1] void push(Stack *S, data_type v) {
[2]   Node *x = alloc(sizeof(Node));
[3]   x->d = v;
[4]   do {
[5]     Node *t = S->Top;
[6]     x->n = t;
[7]   } while (!CAS(&S->Top,t,x));
[8] }
```

Representing an Unbounded Number of Threads

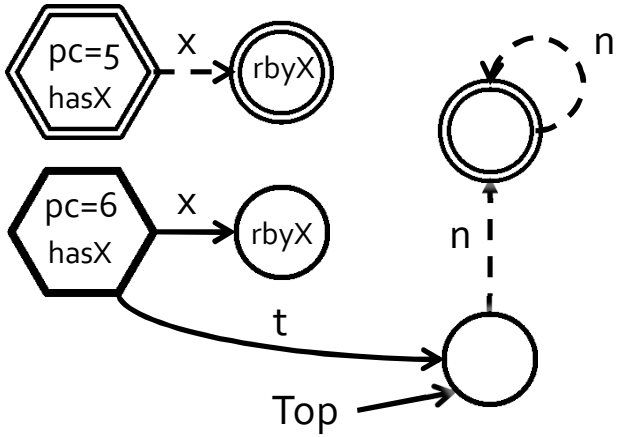



```
[1] void push(Stack *S, data_type v) {
[2]   Node *x = alloc(sizeof(Node));
[3]   x->d = v;
[4]   do {
[5]     Node *t = S->Top;
[6]     x->n = t;
[7]   } while (!CAS(&S->Top,t,x));
[8] }
```

Abstract Semantics



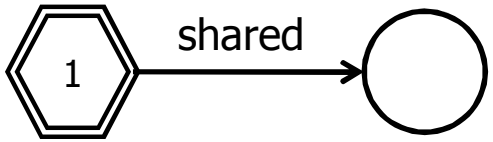

```
[1] void push(Stack *S, data_type v) {
[2]   Node *x = alloc(sizeof(Node));
[3]   x->d = v;
[4]   do {
[5]     Node *t = S->Top;
[6]     x->n = t;
[7]   } while (!CAS(&S->Top,t,x));
[8] }
```



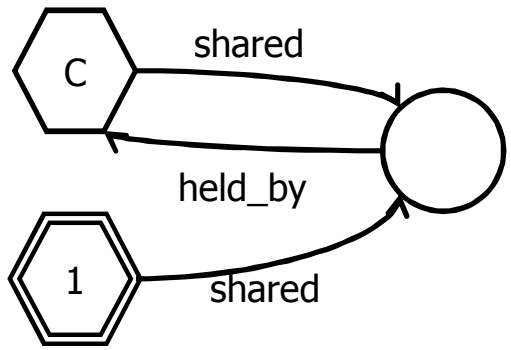
Example - Mutual Exclusion

```
[1] while (true) {
[2]  lock(shared)
[C]  // critical actions
[3]  unlock(shared)
[4] }
```

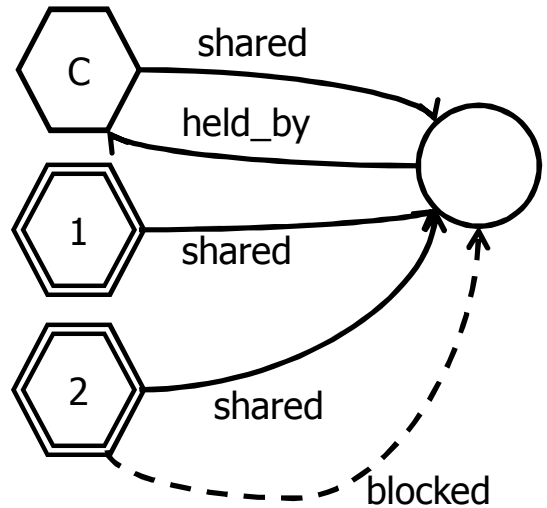
$$\forall t_1, t_2 : (t_1 \neq t_2) \rightarrow \neg(at[pc=c](t_1) \wedge at[pc=c](t_2))$$



Initial configuration



A thread enters the critical section



Other threads may be blocked or just beginning execution

Recap

```
#define EMPTY -1

typedef int data_type;

typ

typ

} N

typ

} Stack;
```

```
[1] void push(Stack *S, data_type v) {
[2]     Node *x = alloc(sizeof(Node));
[3]     x->d = v;
[4]     do {
```

```
[9]  data_type pop(Stack *S){
[10]     do {
[11]         Node *t = S->Top;
[12]         if (t == NULL)
[13]             return EMPTY;
```

- ✓ No null dereferences
- ✓ Structural shape invariants
- **Linearizability**
- ✓ Dynamic Allocation
- ✓ Destructive Updates
- ✓ Concurrency