Program Analysis and Verification

0368-4479

Noam Rinetzky

Lecture 10: Shape Analysis

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

Abstract Interpretation [Cousot'77]

- Mathematical foundation of static analysis
  - Abstract domains
    - Abstract states
    - Join ($\sqcup$)
  - Transformer functions
    - Abstract steps
  - Chaotic iteration
    - Abstract computation
    - Structured Programs

Lattices
(D, $\sqsubseteq$, $\sqcup$, $\sqcap$, $\bot$, $\top$)

Monotonic functions

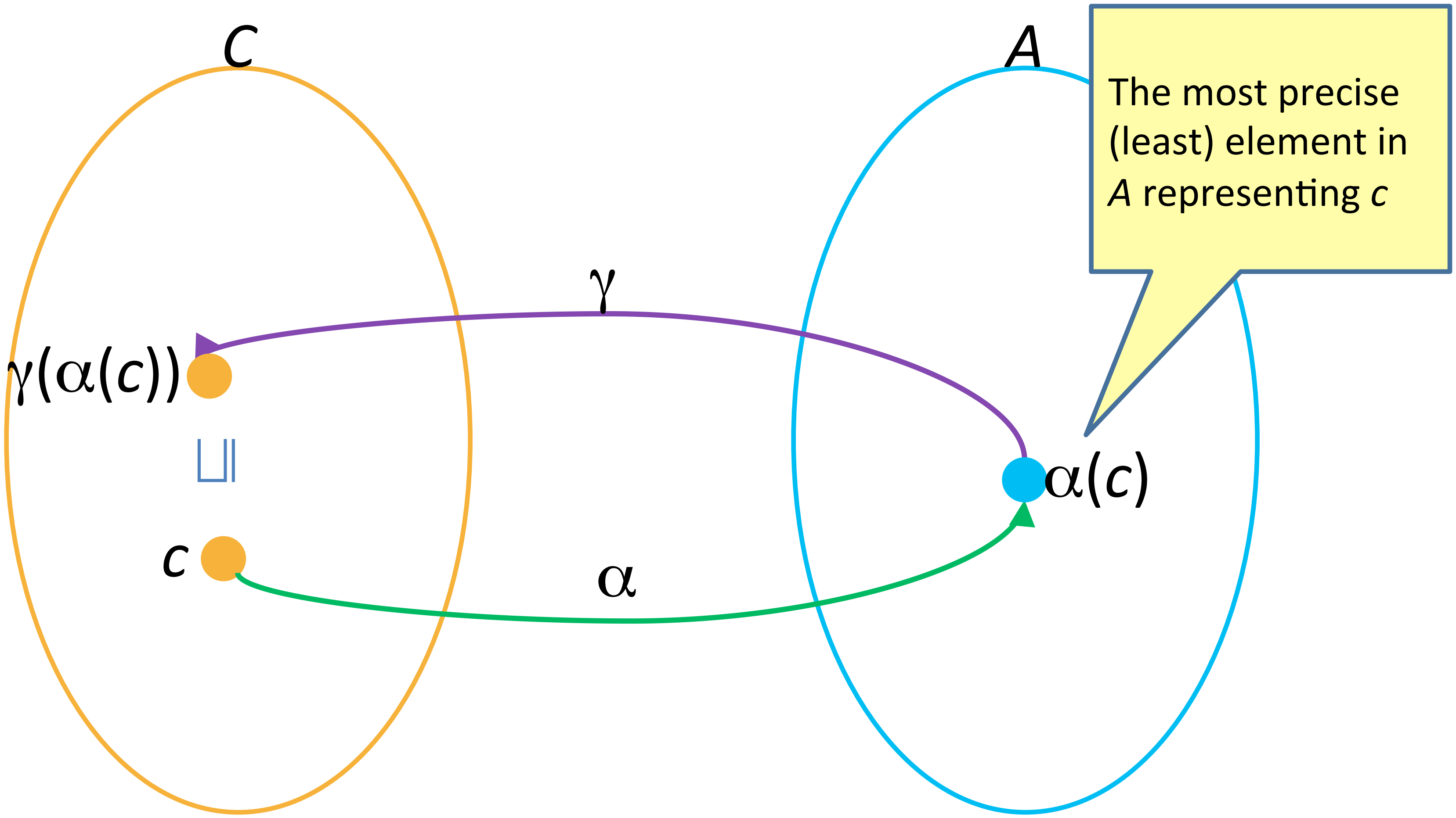
Fixpoints

The collecting lattice

- Lattice for a given control-flow node *v*:
  L_v=($2^{\textbf{State}}$, $\subseteq$, $\cup$, $\cap$, $\emptyset$, **State**)
- Lattice for entire control-flow graph with nodes *V*:

  $$L_{CFG} = \text{Map}(V, L_v)$$
- We will use this lattice as a baseline for static analysis and define abstractions of its elements

Galois Connection

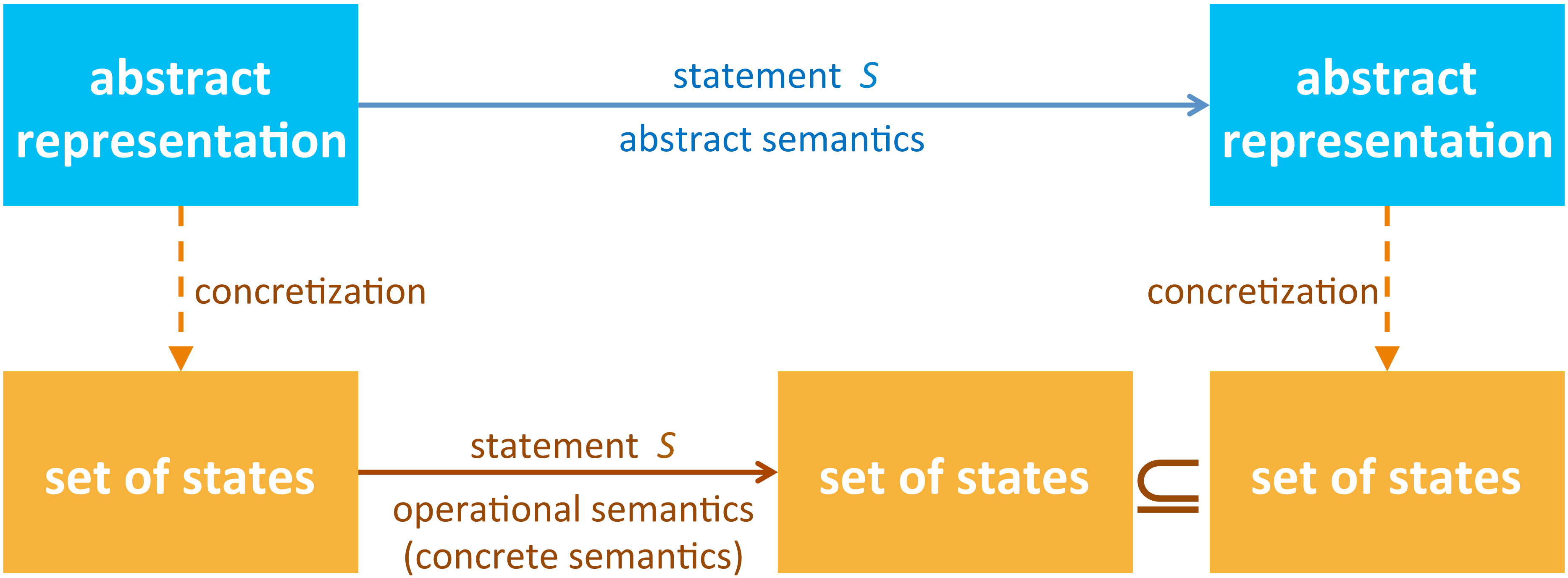


$$c \sqsubseteq \gamma(\alpha(c))$$

Galois Connection

- Given two complete lattices
  $C = (D^C, \sqsubseteq^C, \sqcup^C, \sqcap^C, \bot^C, \top^C)$ – concrete domain
  $A = (D^A, \sqsubseteq^A, \sqcup^A, \sqcap^A, \bot^A, \top^A)$ – abstract domain
- A Galois Connection (GC) is quadruple (C, α, γ, A) that relates C and A via the monotone functions
  – The abstraction function $\alpha : D^C \rightarrow D^A$
  – The concretization function $\gamma : D^A \rightarrow D^C$
- for every concrete element $c \in D^C$ and abstract element $a \in D^A$
  $\alpha(\gamma(a)) \sqsubseteq a$ and $c \sqsubseteq \gamma(\alpha(c))$
- Alternatively $\alpha(c) \sqsubseteq a$ iff $c \sqsubseteq \gamma(a)$

Abstract (conservative) interpretation



Shape Analysis

Automatically verify properties of programs manipulating dynamically allocated storage

Identify all possible shapes (layout) of the heap

Analyzing Singly Linked Lists

Limitations of pointer analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```

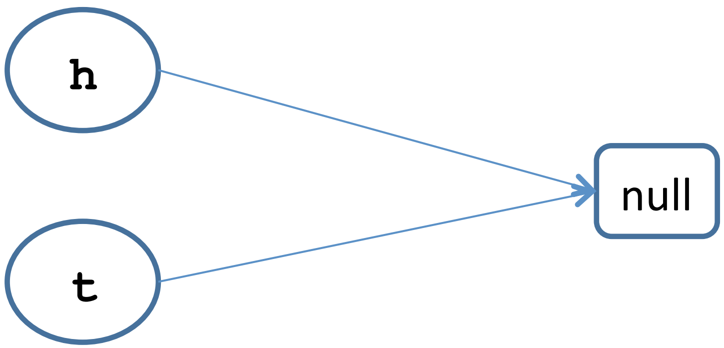
```
// Singly-linked list
// data type.
class SLL {
  int data;
  public SLL n; // next cell

  SLL(Object data) {
    this.data = data;
    this.n = null;
  }
}
```

Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```

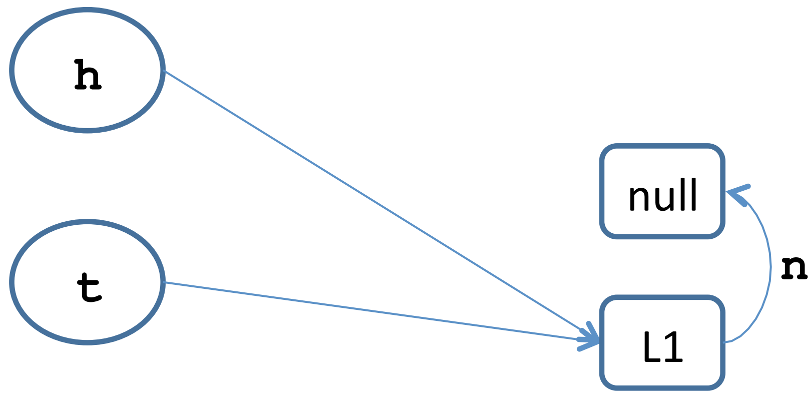
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



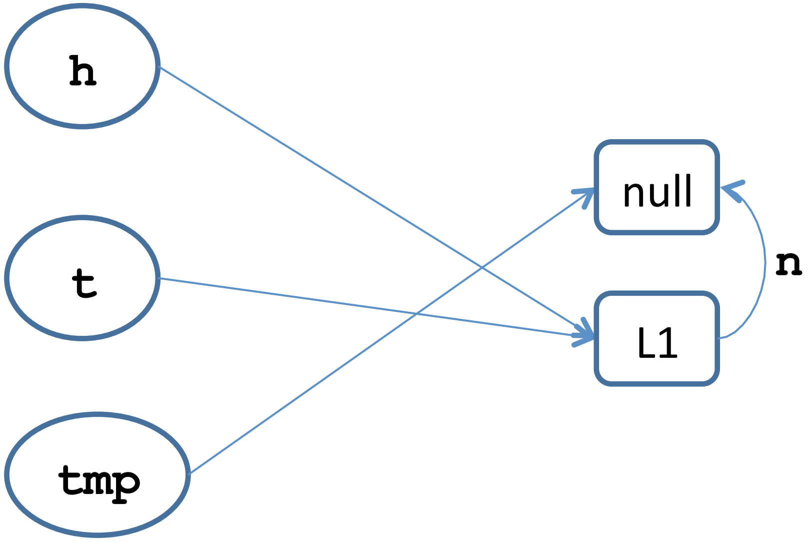
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



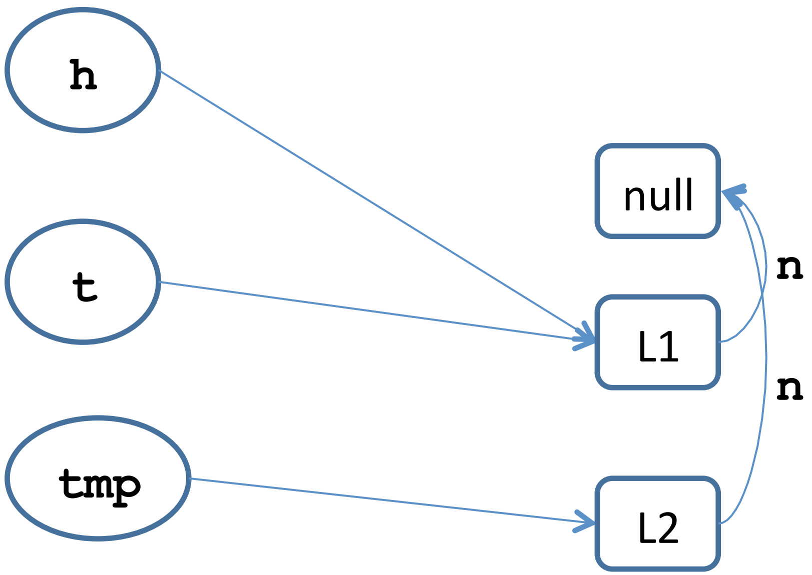
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



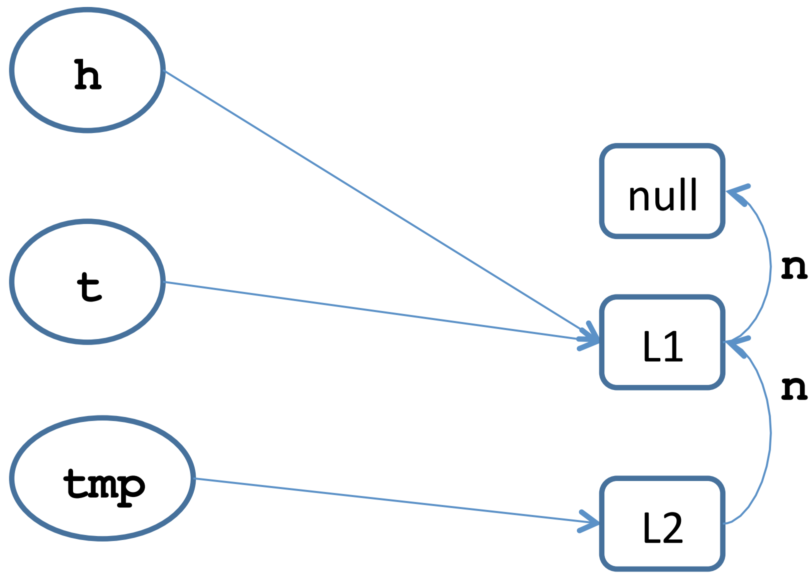
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



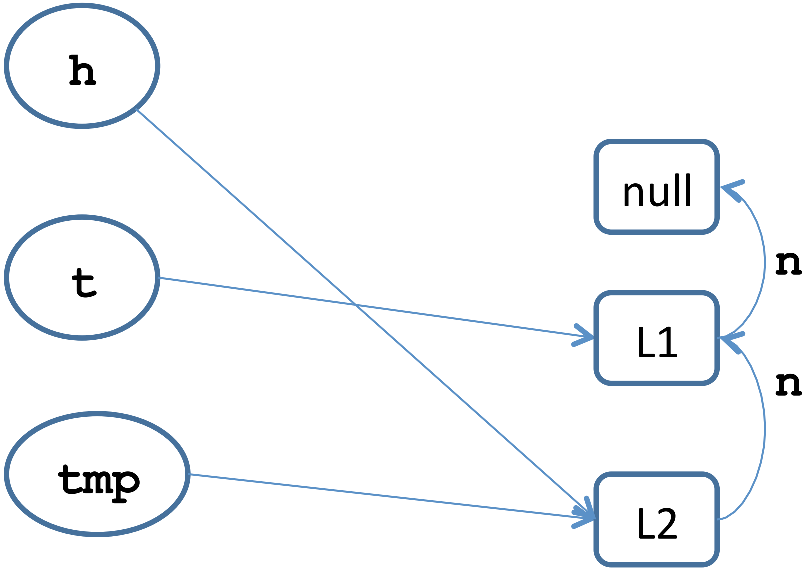
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



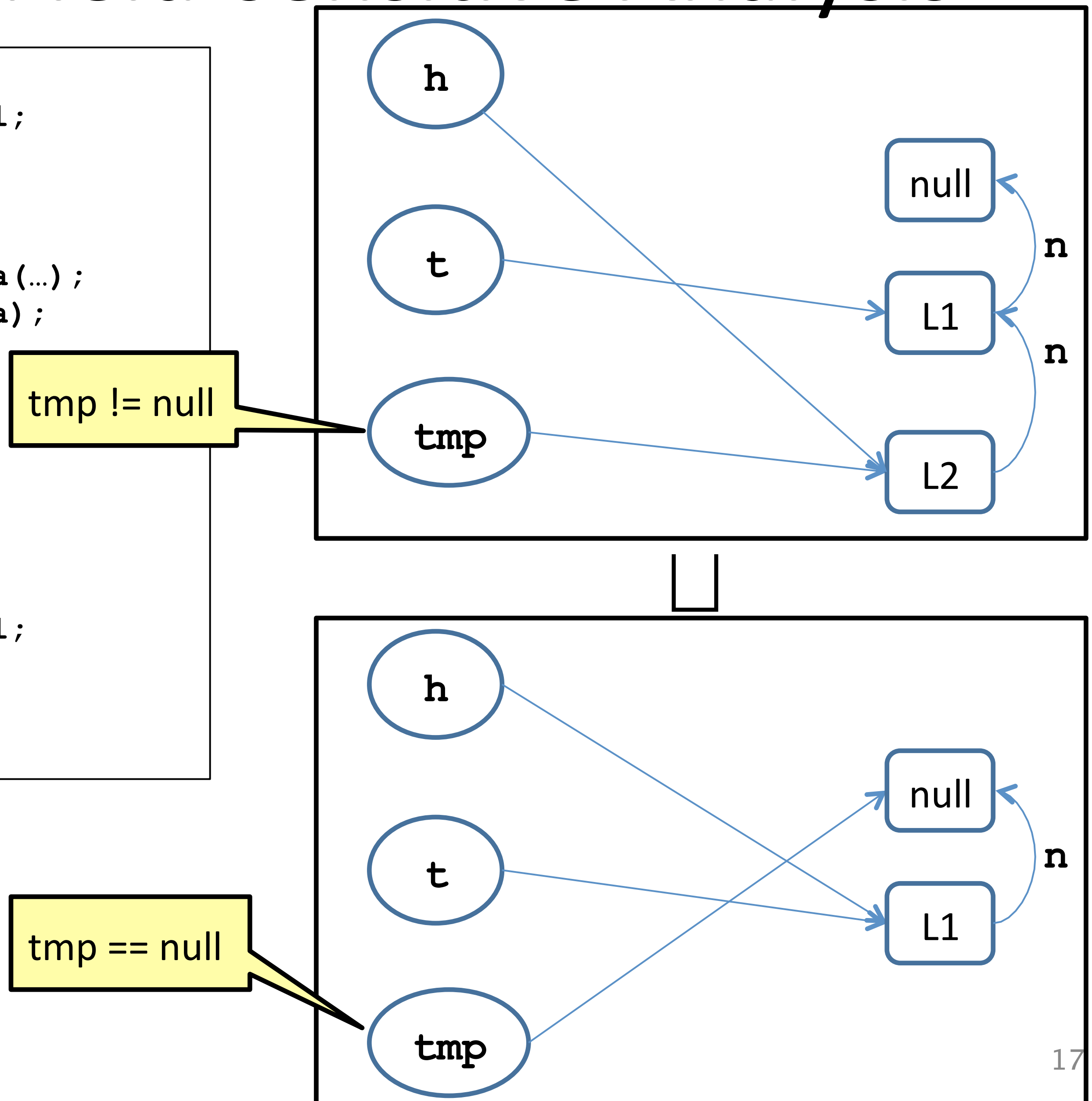
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



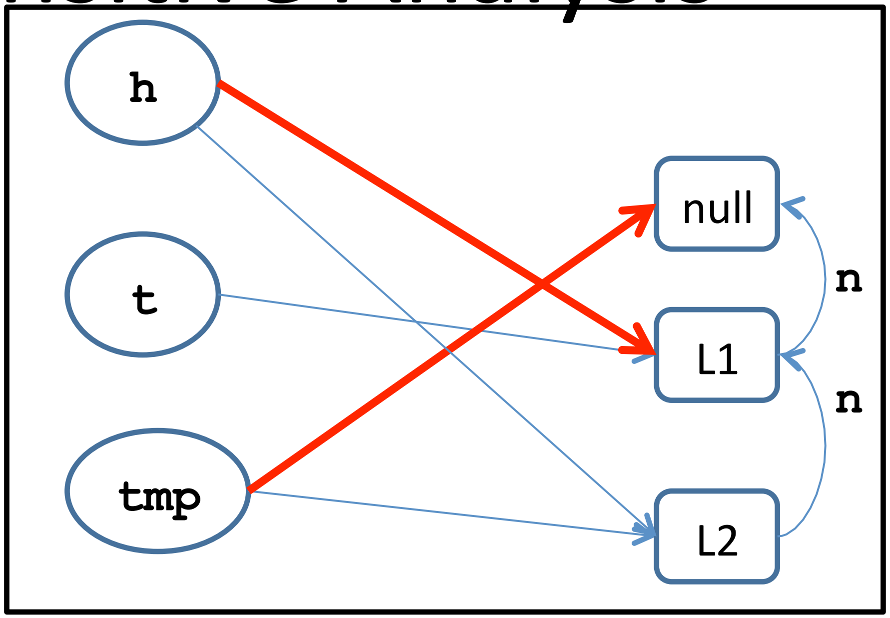
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```

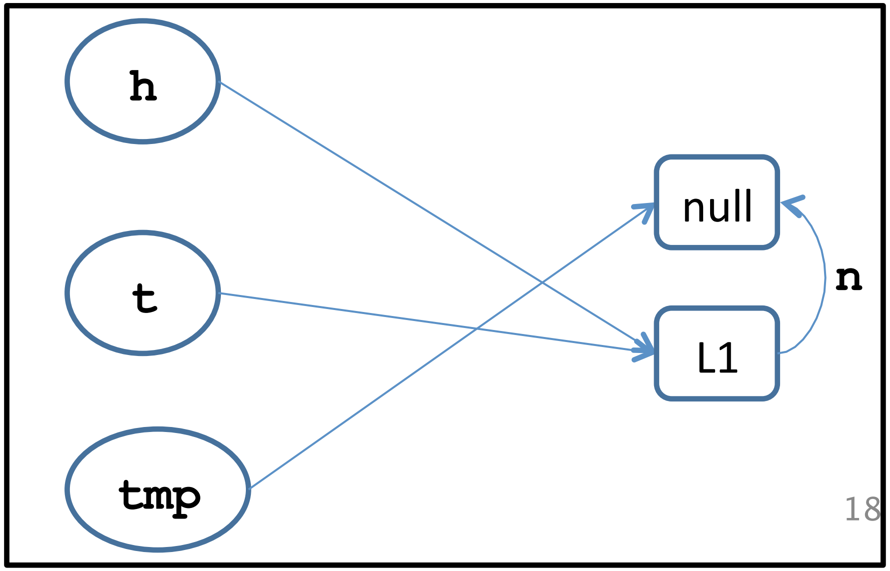


Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```

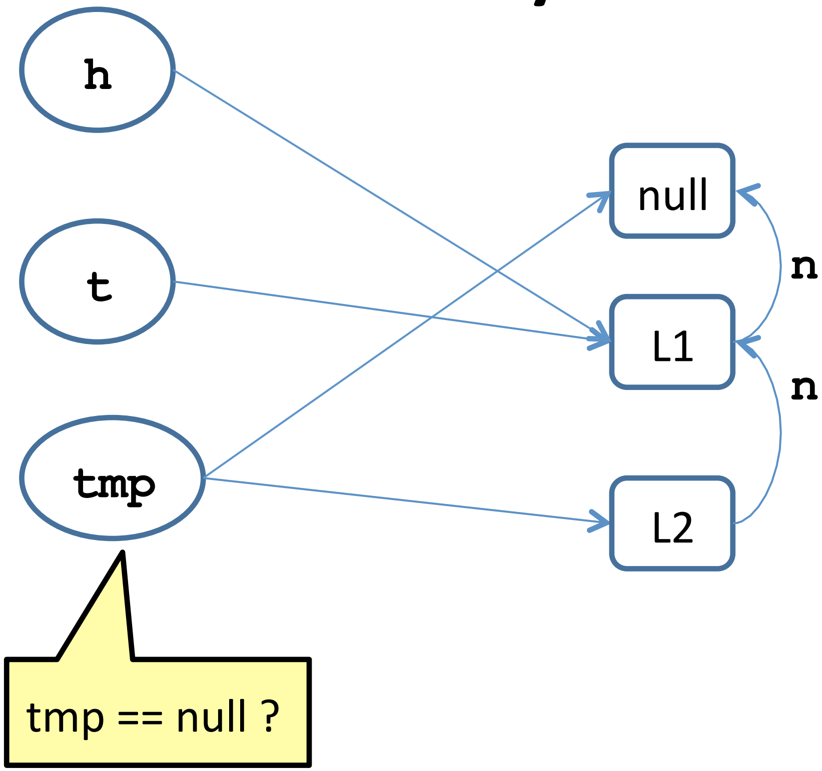


⊔



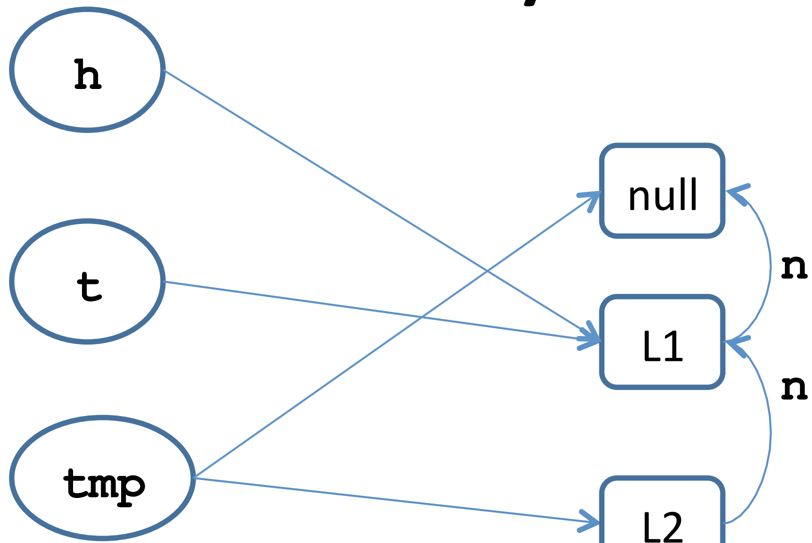
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



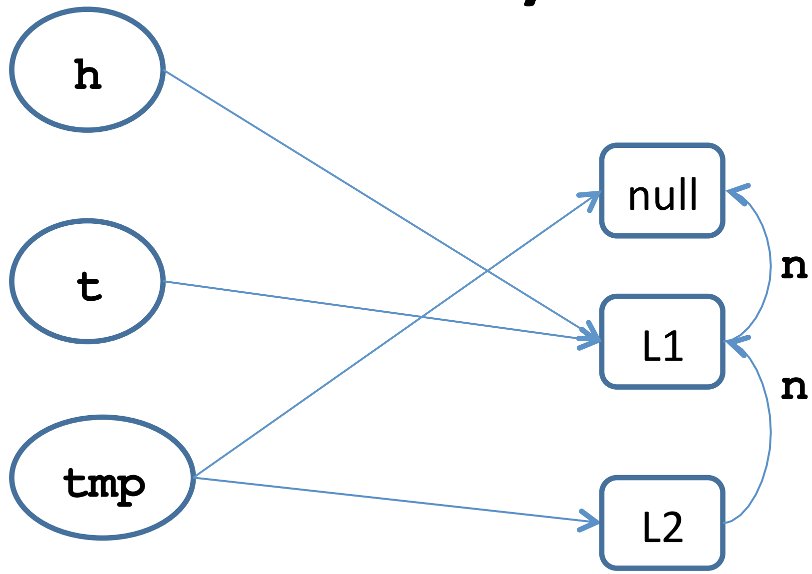
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



Flow&Field-sensitive Analysis

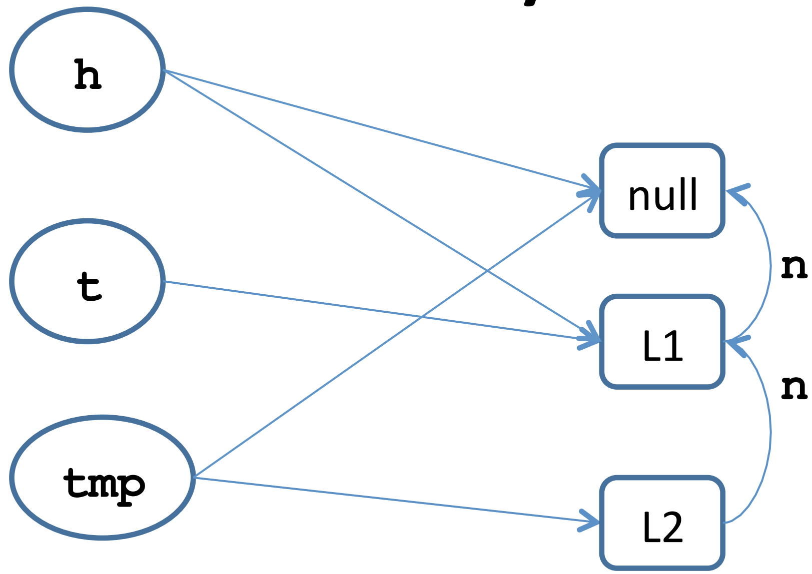
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



Possible null dereference!

Flow&Field-sensitive Analysis

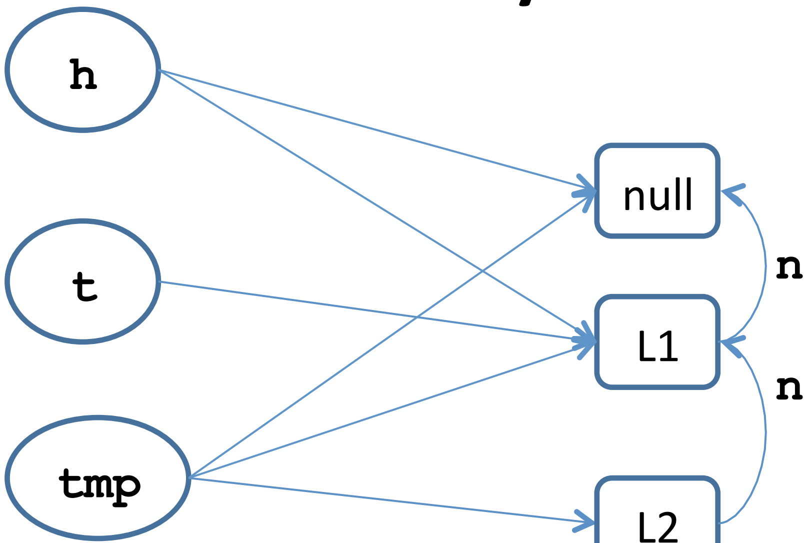
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



Fixed-point for first loop

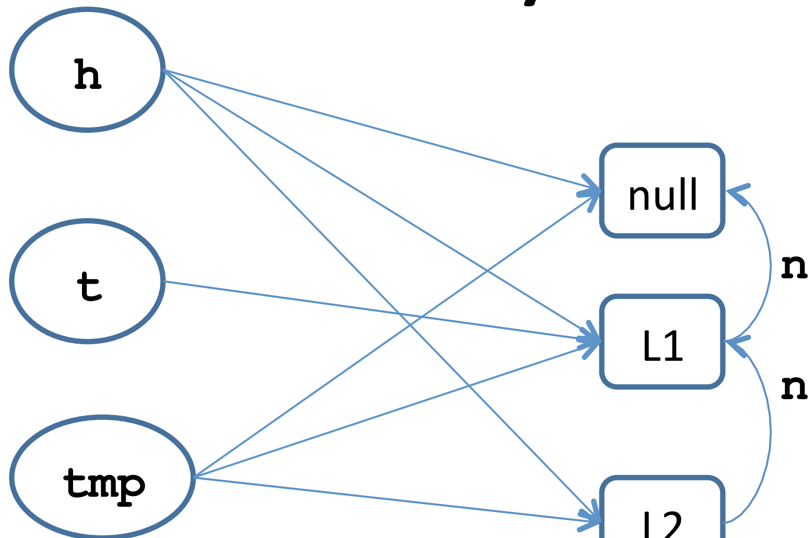
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



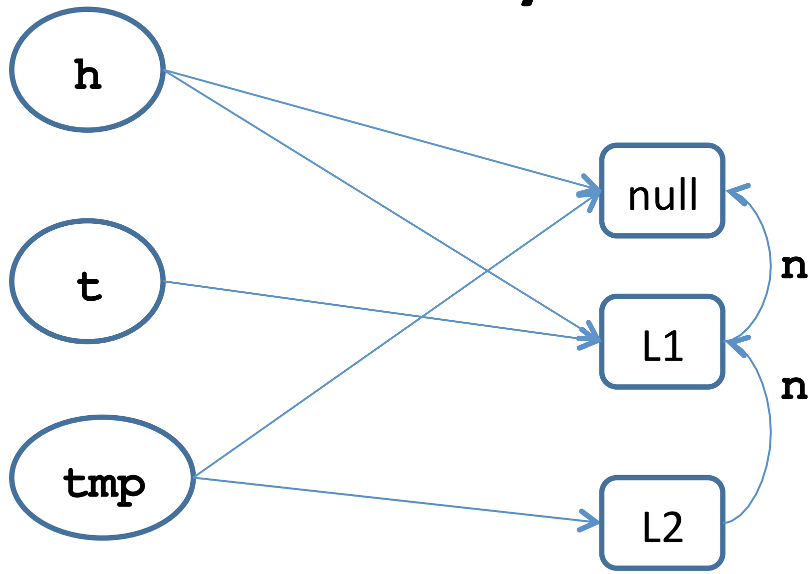
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



Possible null dereference!

What was the problem?

- Pointer analysis abstract all objects allocated at same program location into one summary object. However, objects allocated at same memory location may behave very differently
  - E.g., object is first/last one in the list

- Number of objects represented by summary object $\geq$1 – does not allow strong updates

- Join operator very coarse – abstracts away important distinctions (tmp=null/tmp!=null)

Improved solution

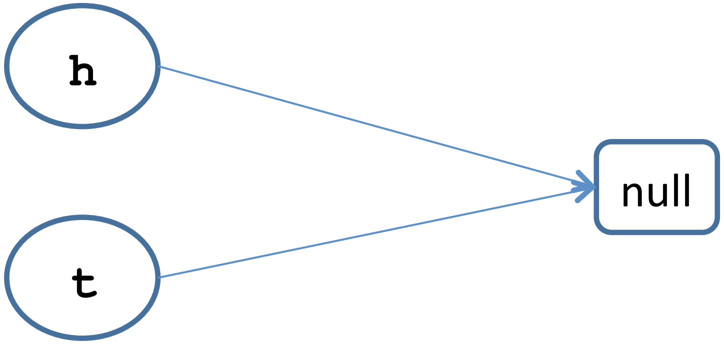
- Pointer analysis abstract all objects allocated at same program location into one summary object. However, objects allocated at same memory location may behave very differently
  - E.g., object is first/last one in the list
  - Add extra instrumentation predicates to distinguish between objects with different roles
- Number of objects represented by summary object ≥ 1 – does not allow strong updates
  - Distinguish between concrete objects (#=1) and abstract objects (#$\geq$1)
- Join operator very coarse – abstracts away important distinctions (tmp=null/tmp!=null)
  - Apply disjunctive completion

Adding properties to objects

- Let's first drop allocation site information and instead…
- Define a unary predicate x(v) for each pointer variable x meaning x points to x
- Predicate holds for at most one node
- Merge together nodes with same sets of predicates

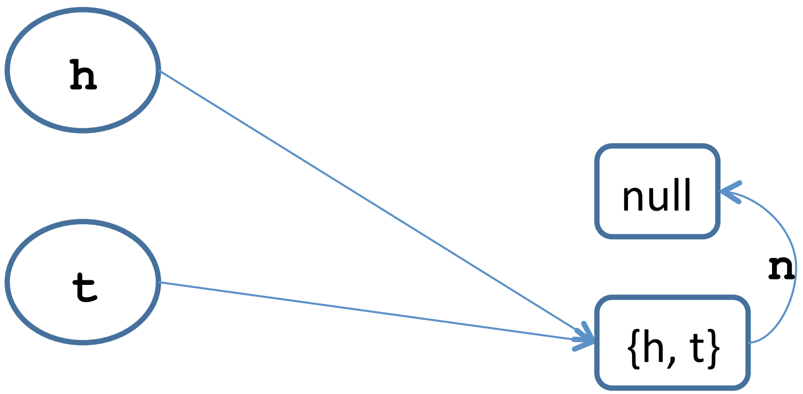
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



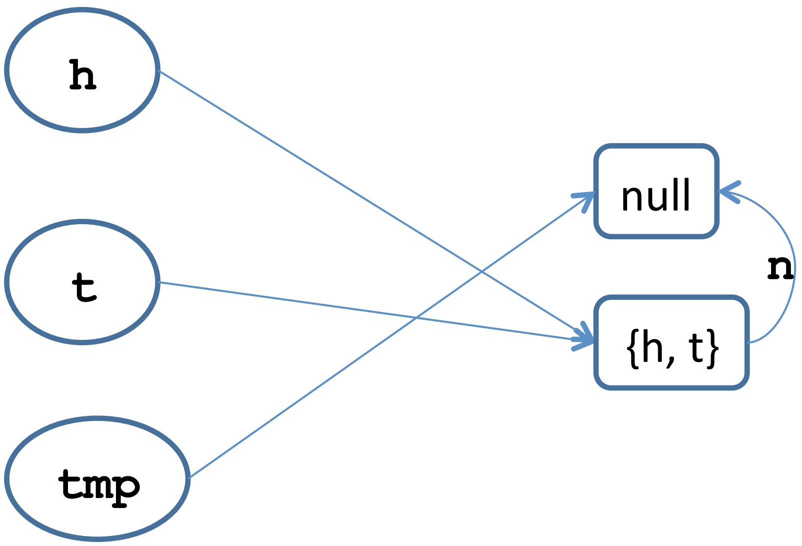
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



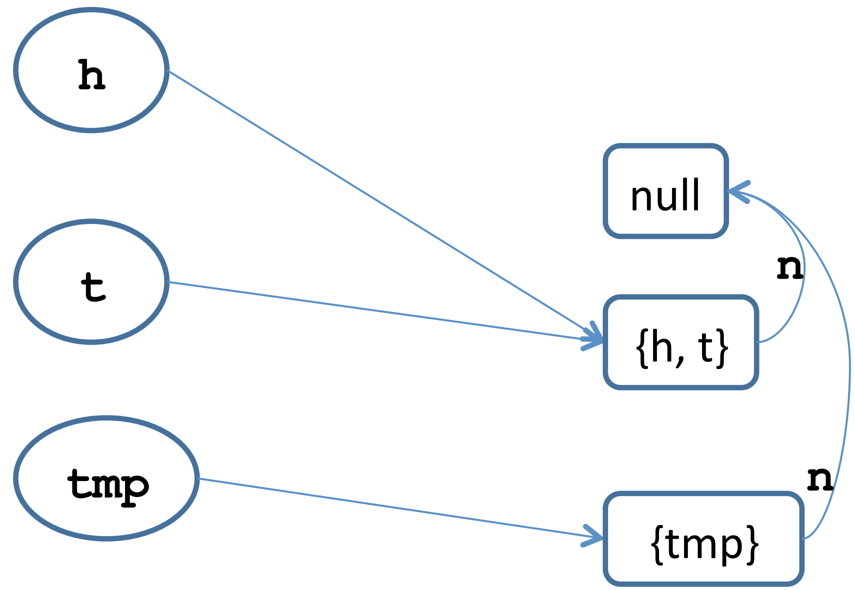
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



Flow&Field-sensitive Analysis

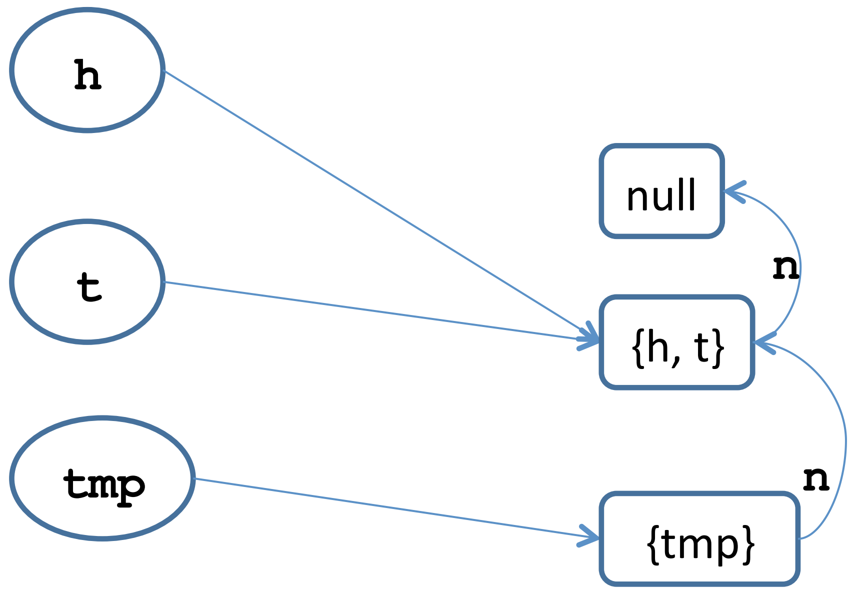
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



No null dereference

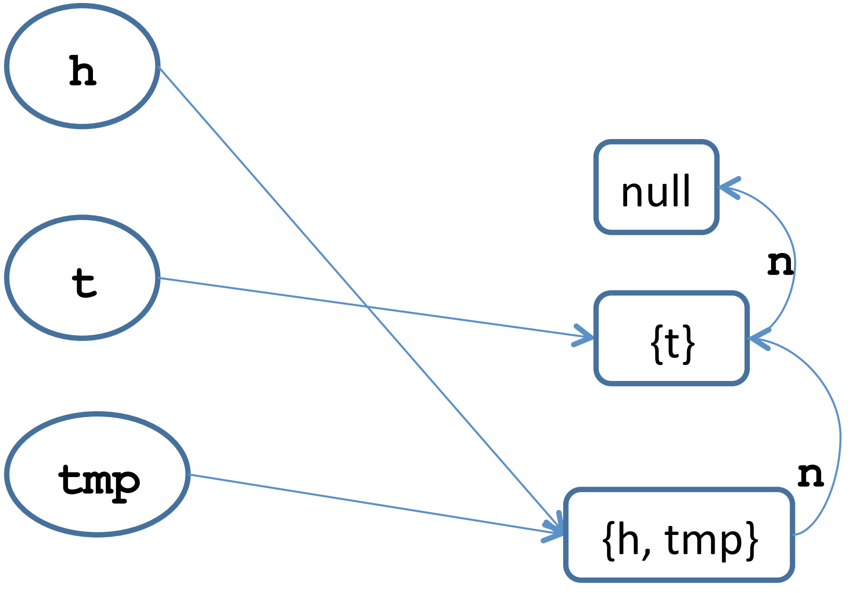
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



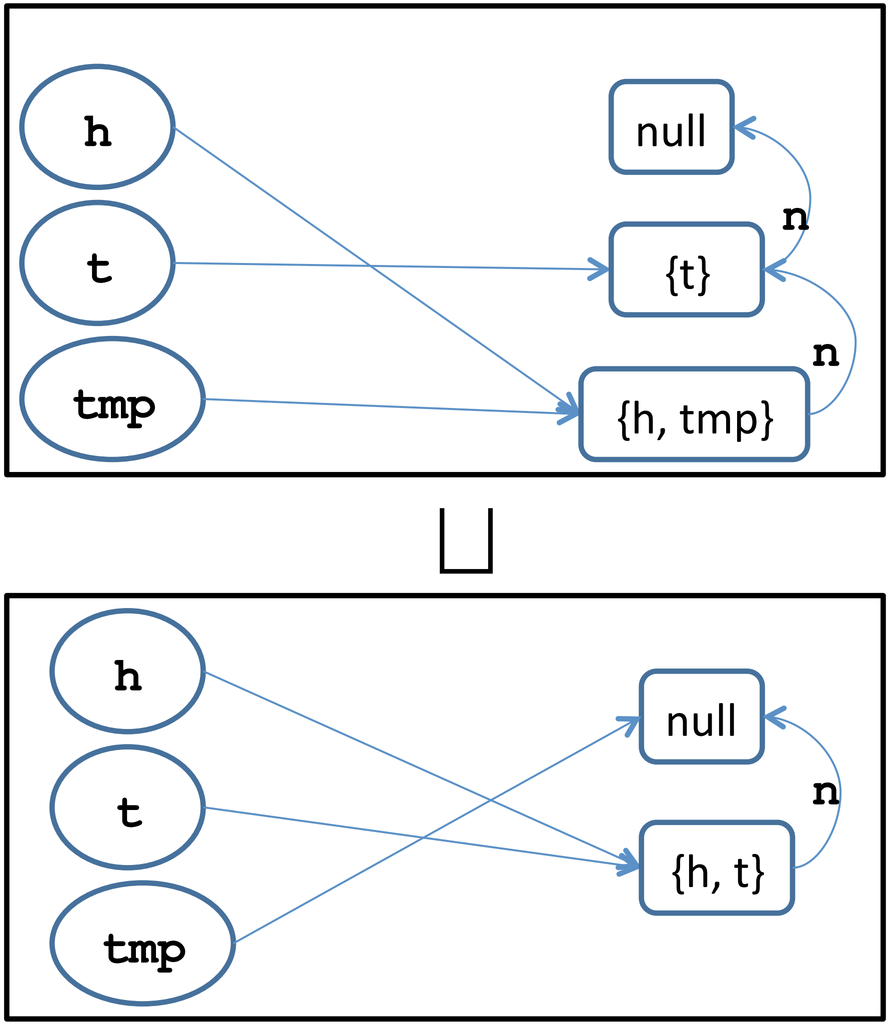
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



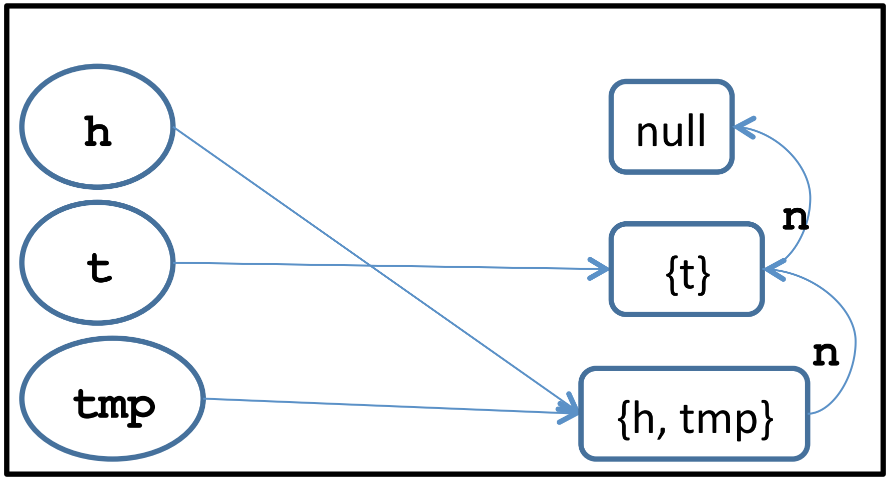
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```

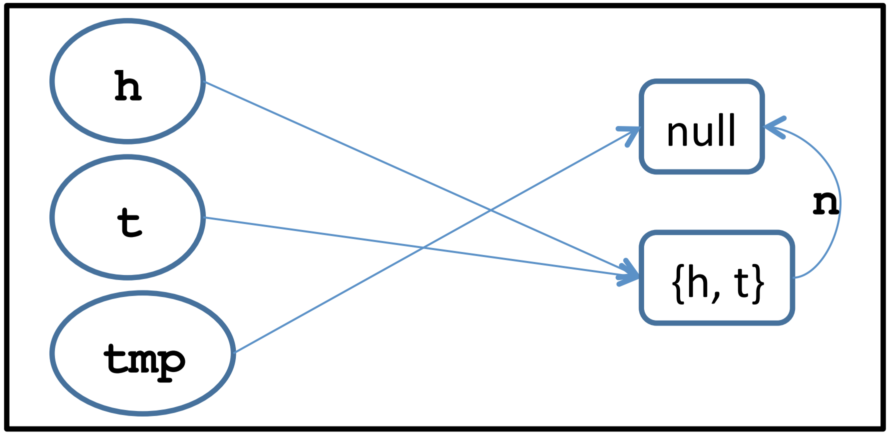


Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



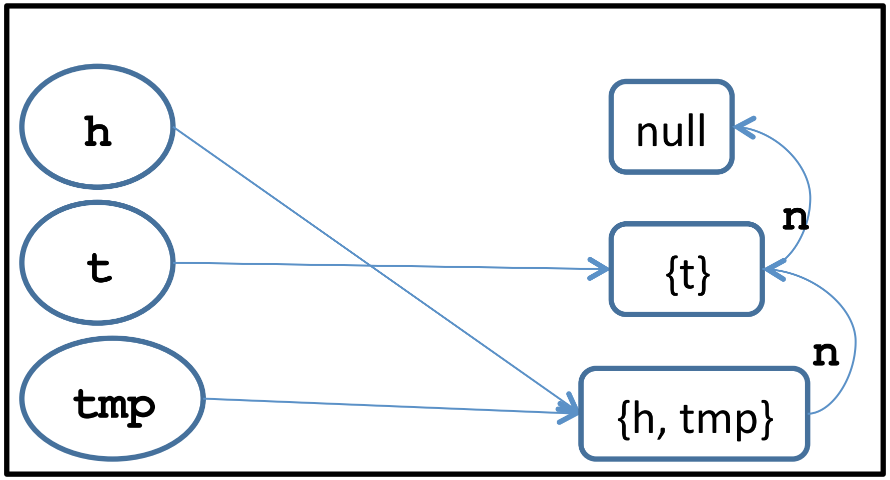
⋁



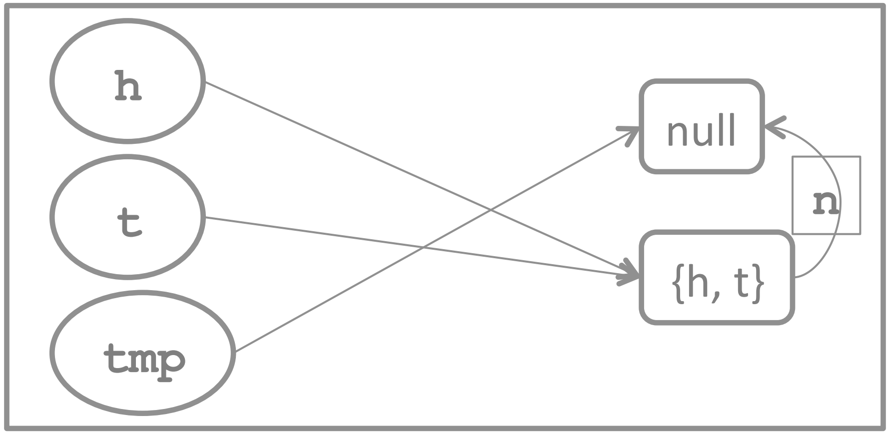
Do we need to analyze this shape graph again?

Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```

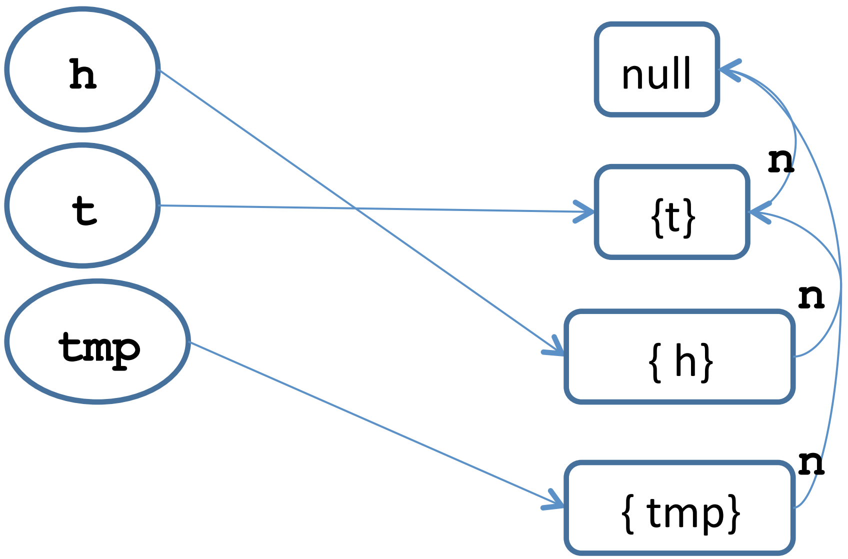


⋁



Flow&Field-sensitive Analysis

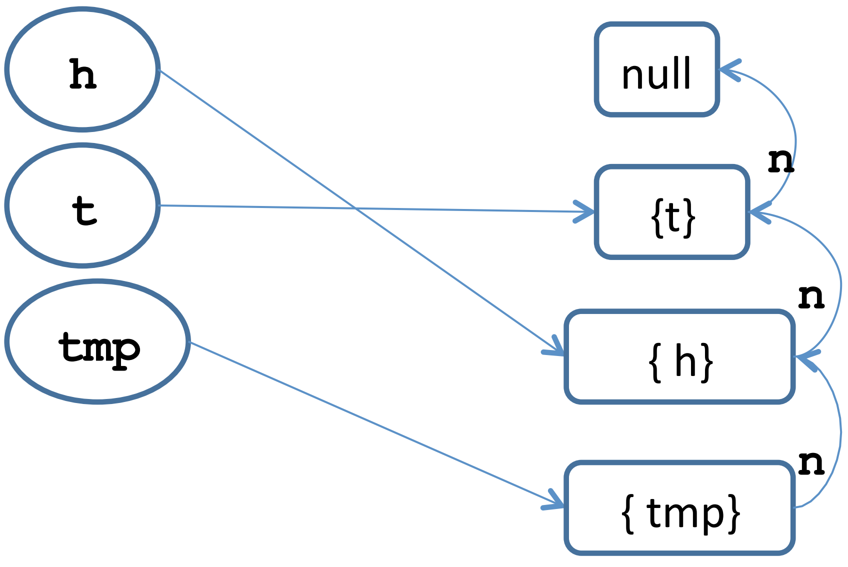
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



No null dereference

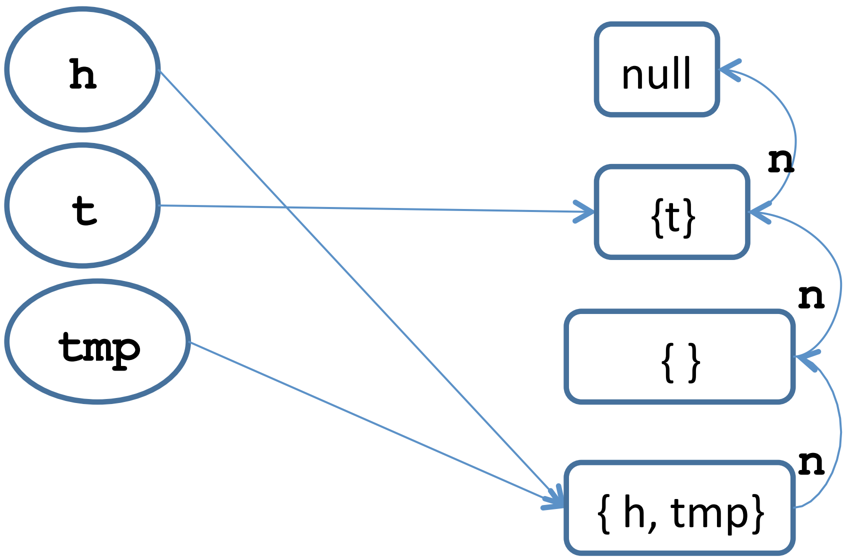
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



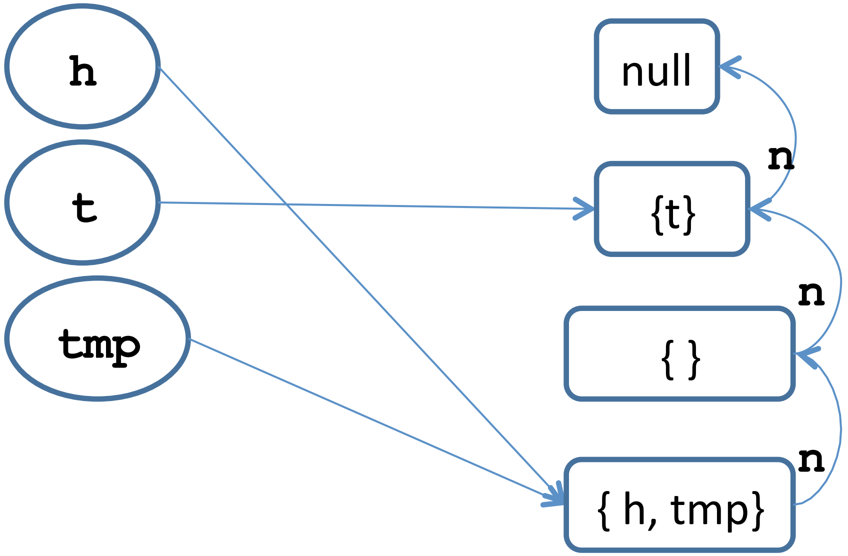
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



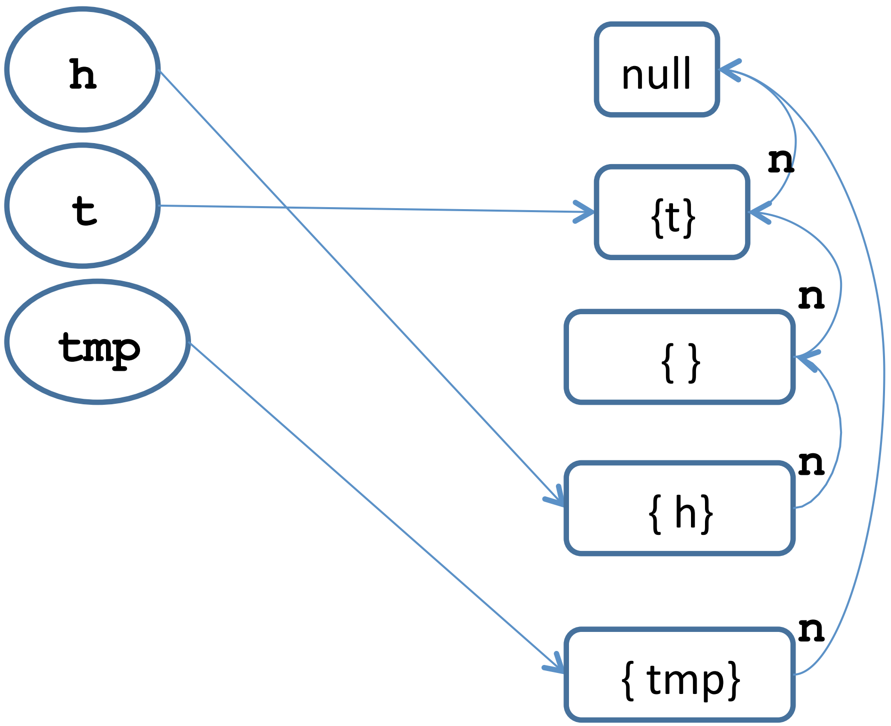
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



Flow&Field-sensitive Analysis

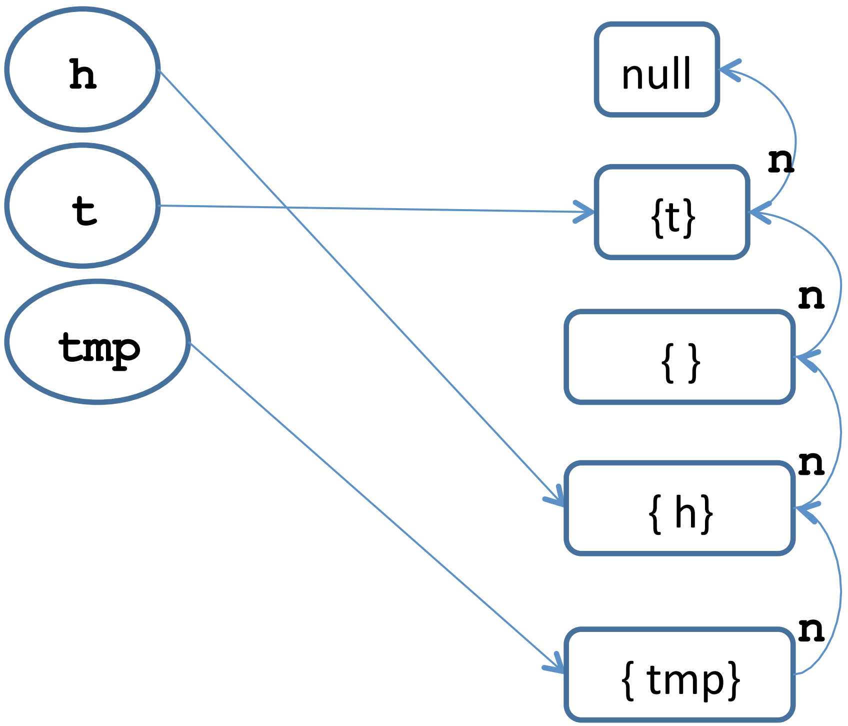
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



No null dereference

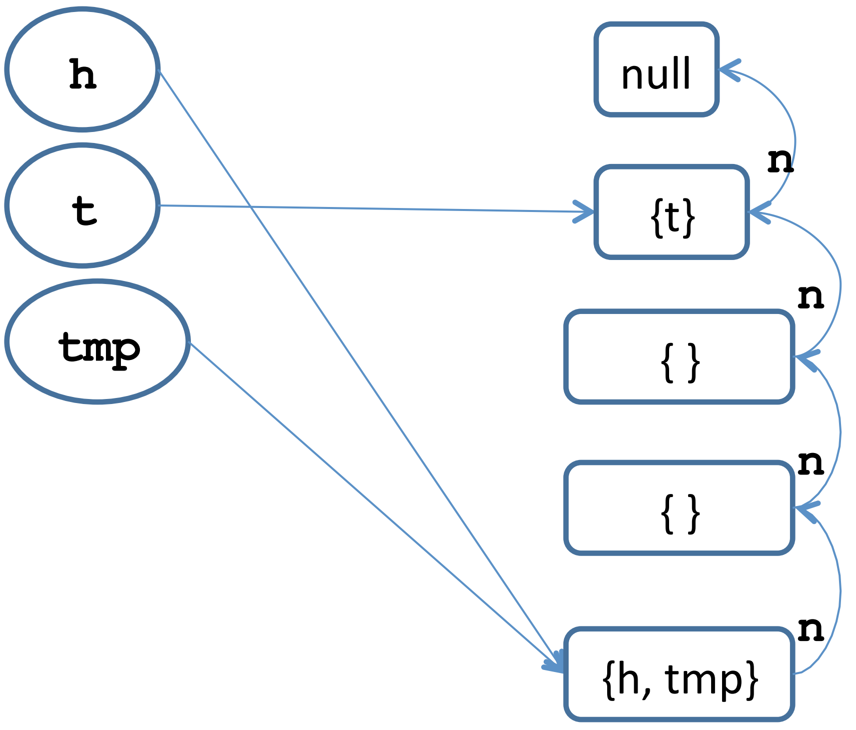
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



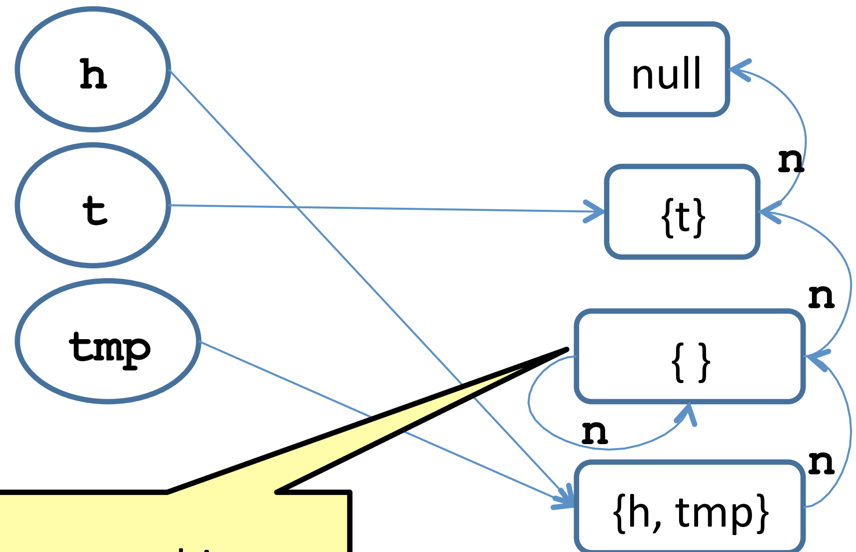
Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```



Flow&Field-sensitive Analysis

```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

    // Process elements
    tmp = h;
    while (tmp != t) {
      assert tmp != null;
      tmp.data += 1;
      tmp = tmp.n;
    }
```

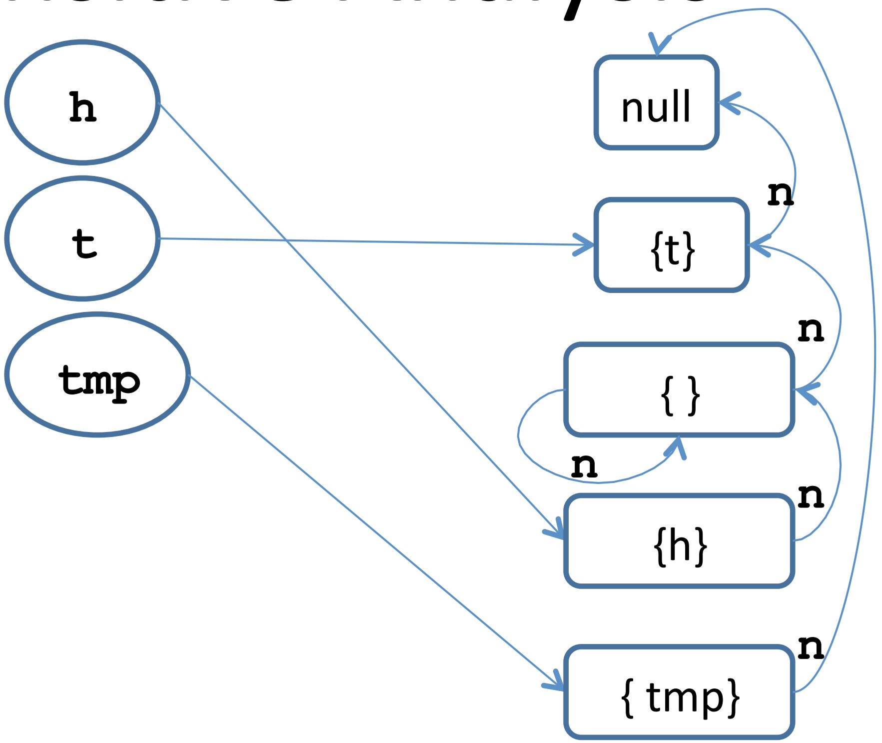


How many objects does it represent?

Summarization

Flow&Field-sensitive Analysis

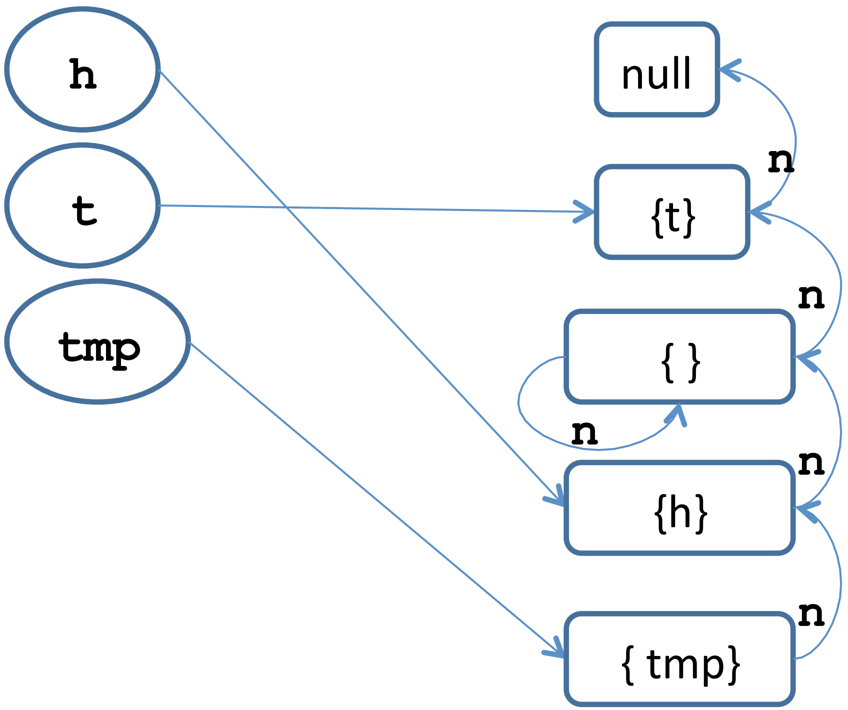
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



No null dereference

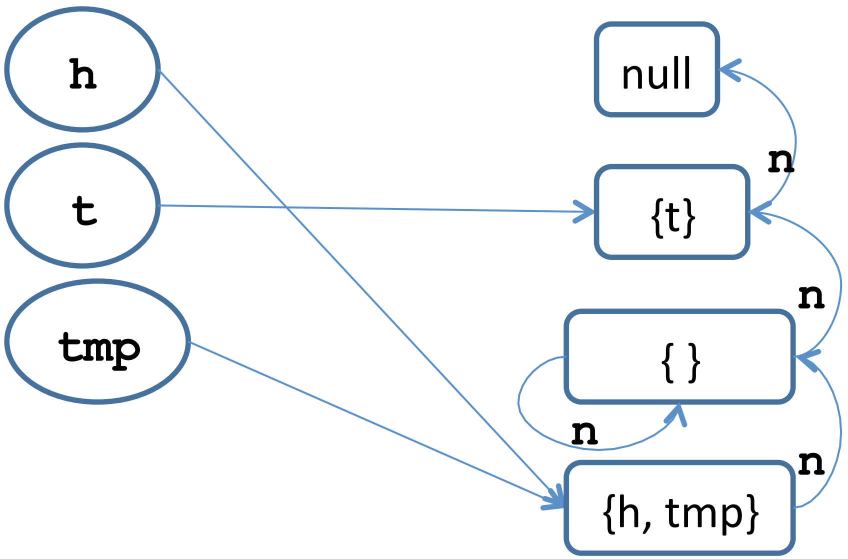
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



Flow&Field-sensitive Analysis

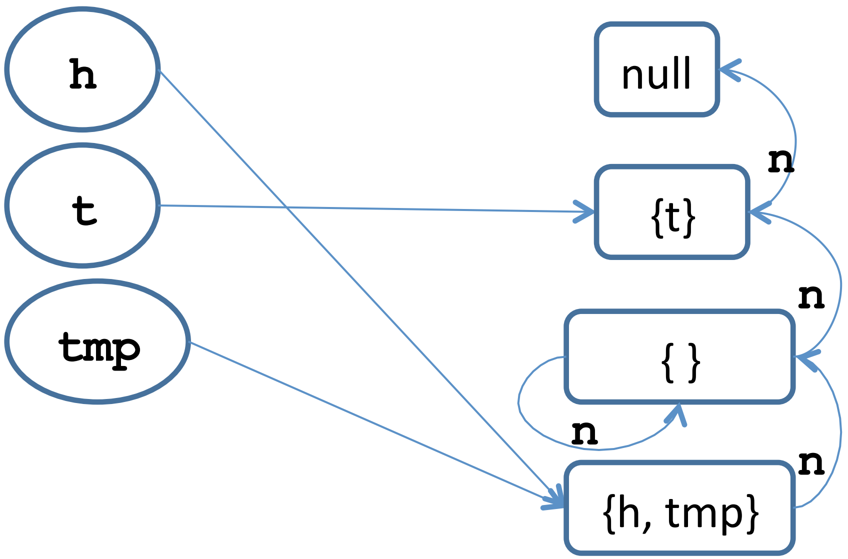
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



Fixed-point for first loop

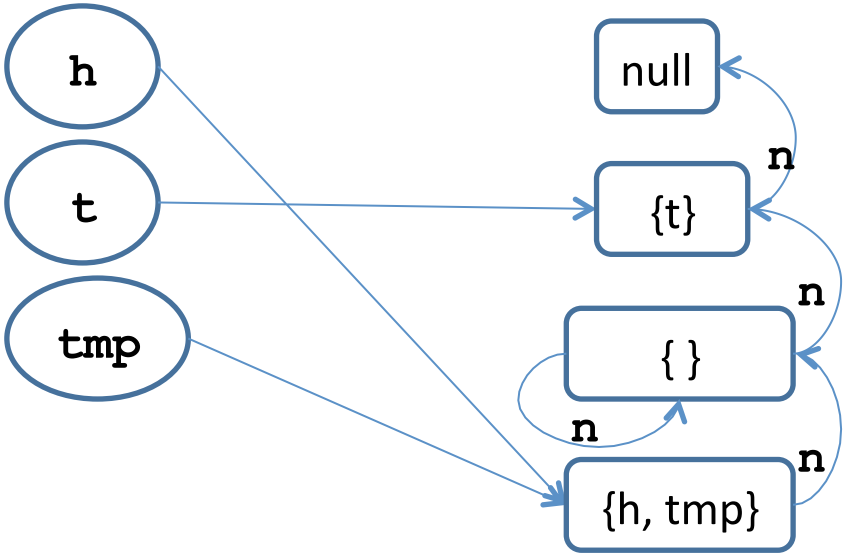
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



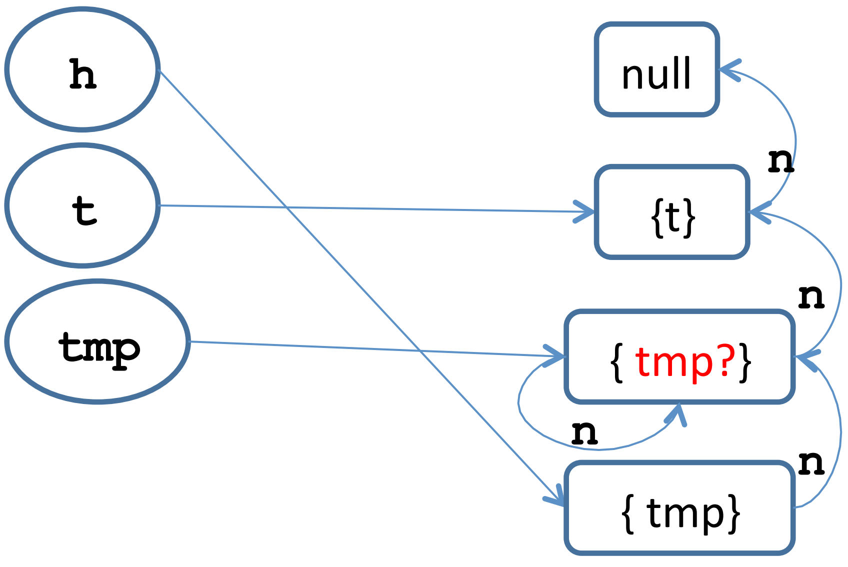
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



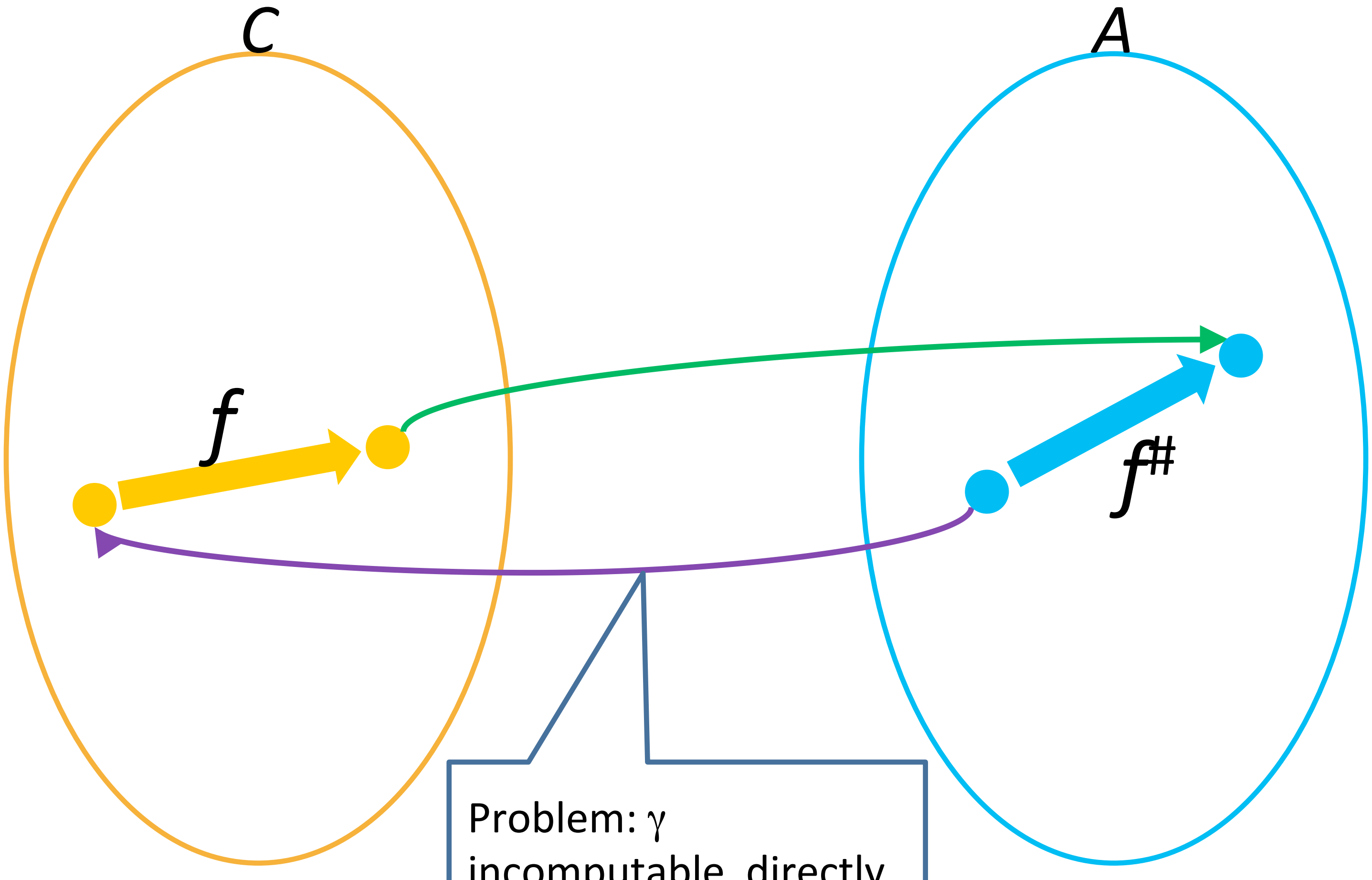
Flow&Field-sensitive Analysis

```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```



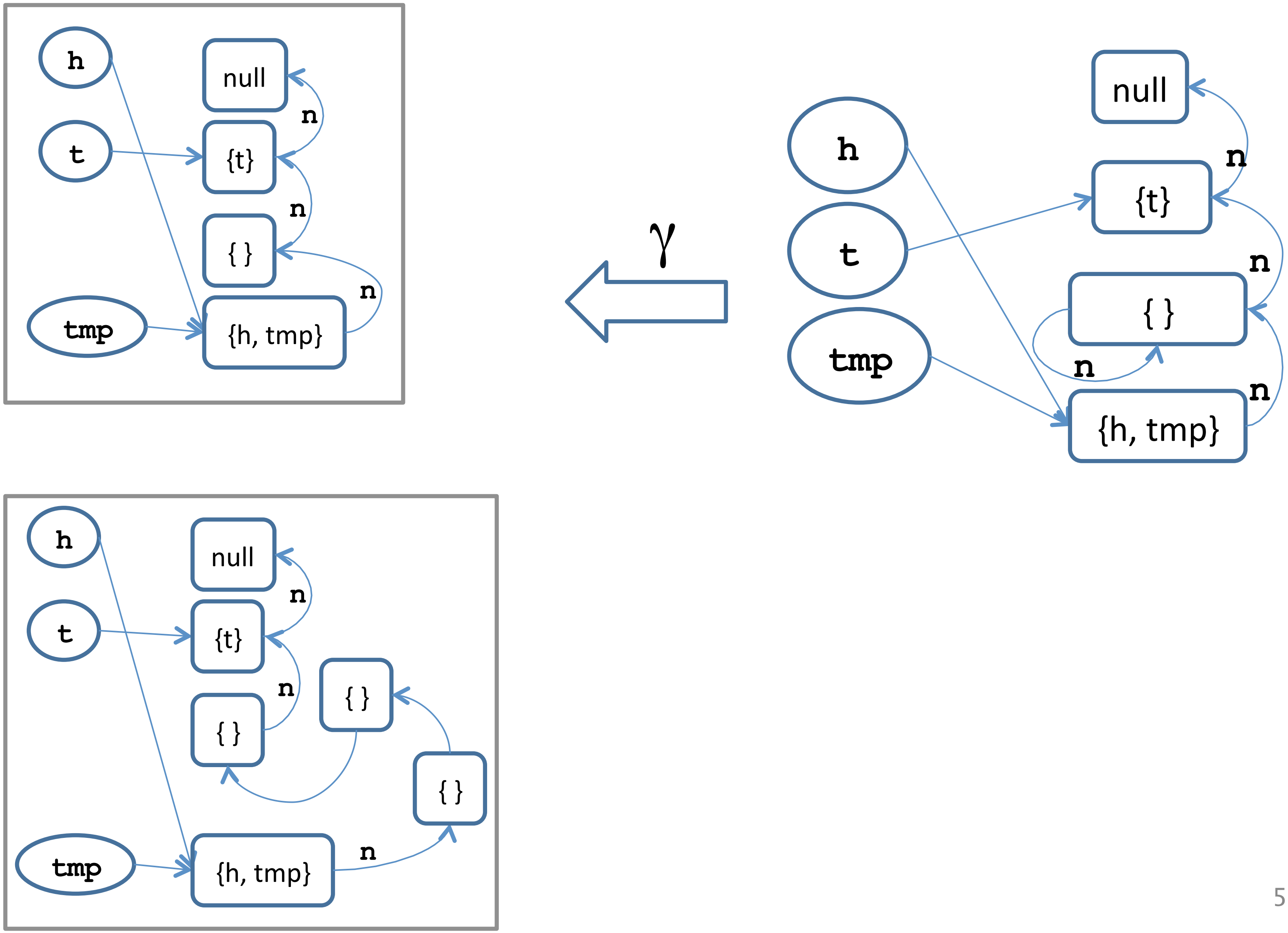
Best (induced) transformer

$$f^{\#}(a) = \alpha(f(\gamma(a)))$$

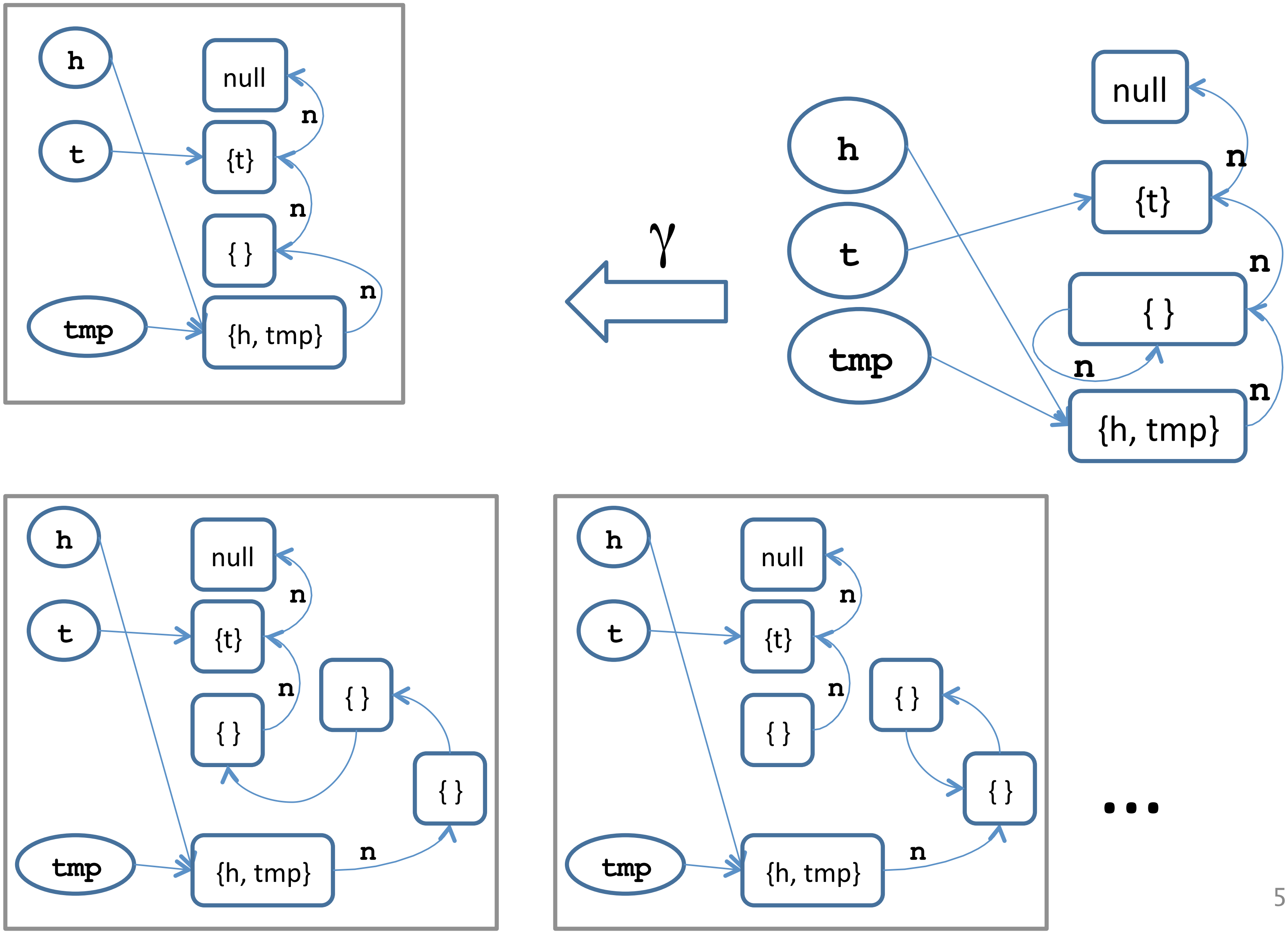


Problem: γ incomputable directly

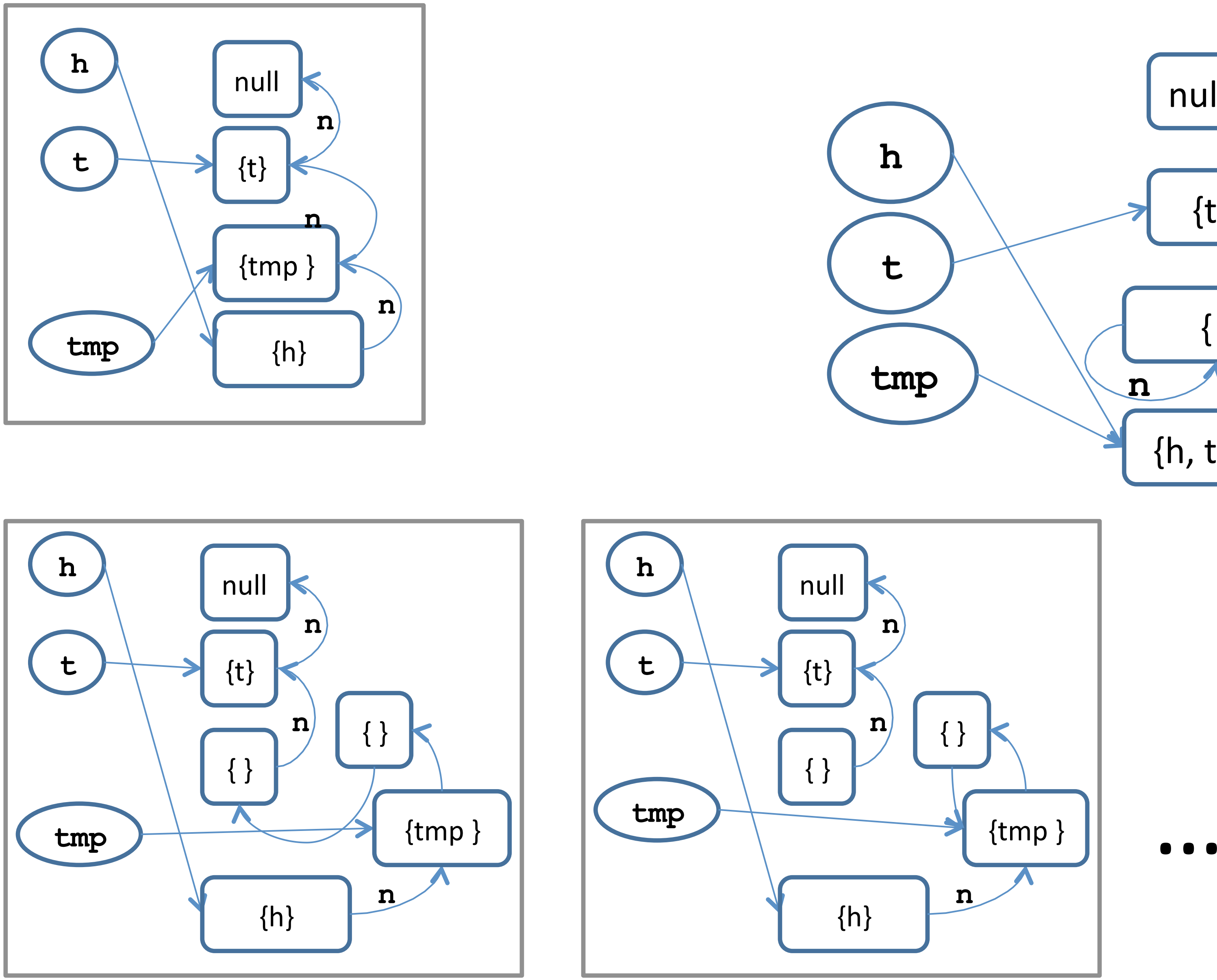
Best transformer for tmp=tmp.n



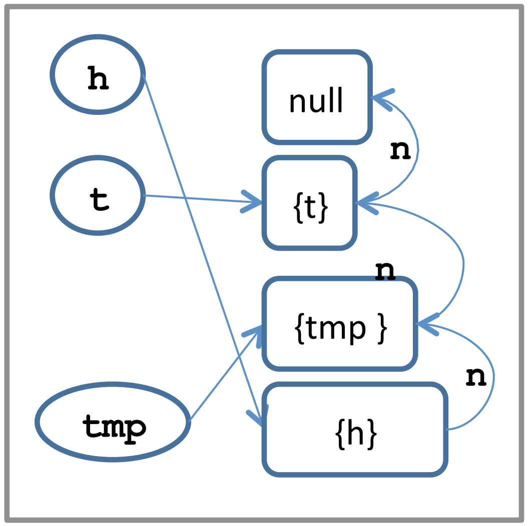
Best transformer for tmp=tmp.n: γ

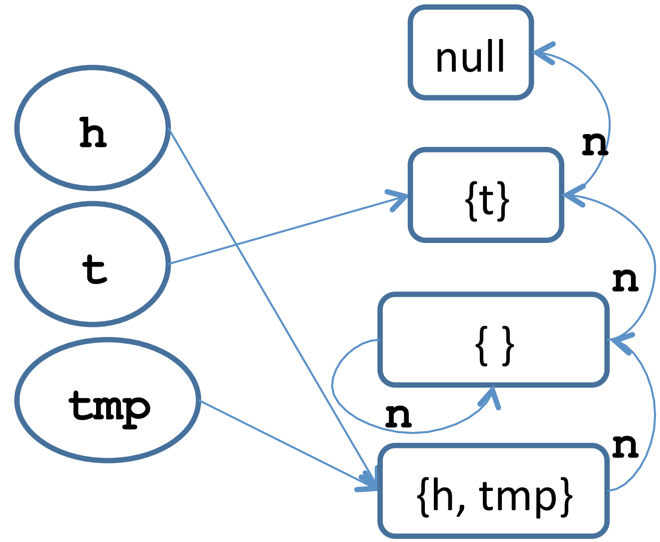


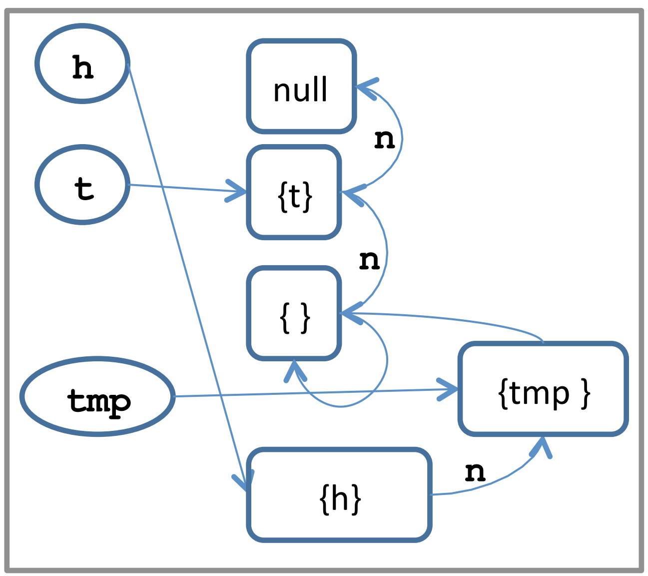
Best transformer for ⟦tmp=tmp.n⟧

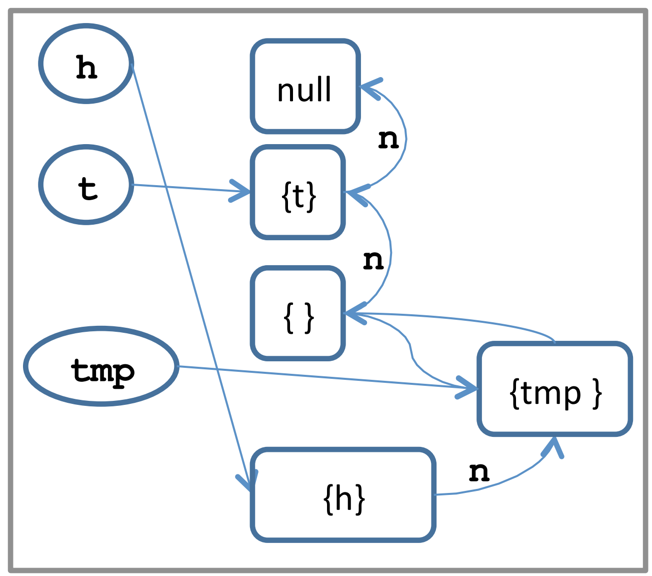


Best transformer for tmp=tmp.n: α







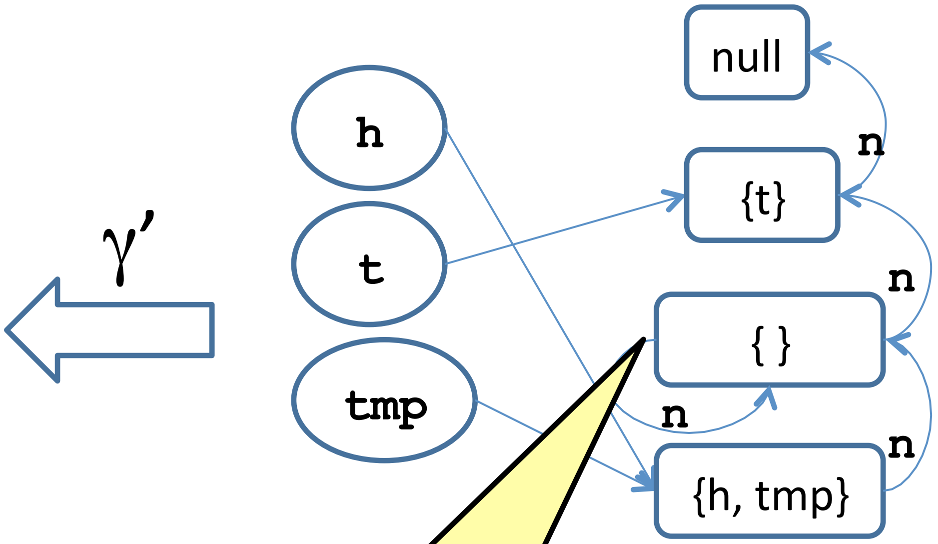


…

Handling updates on summary nodes

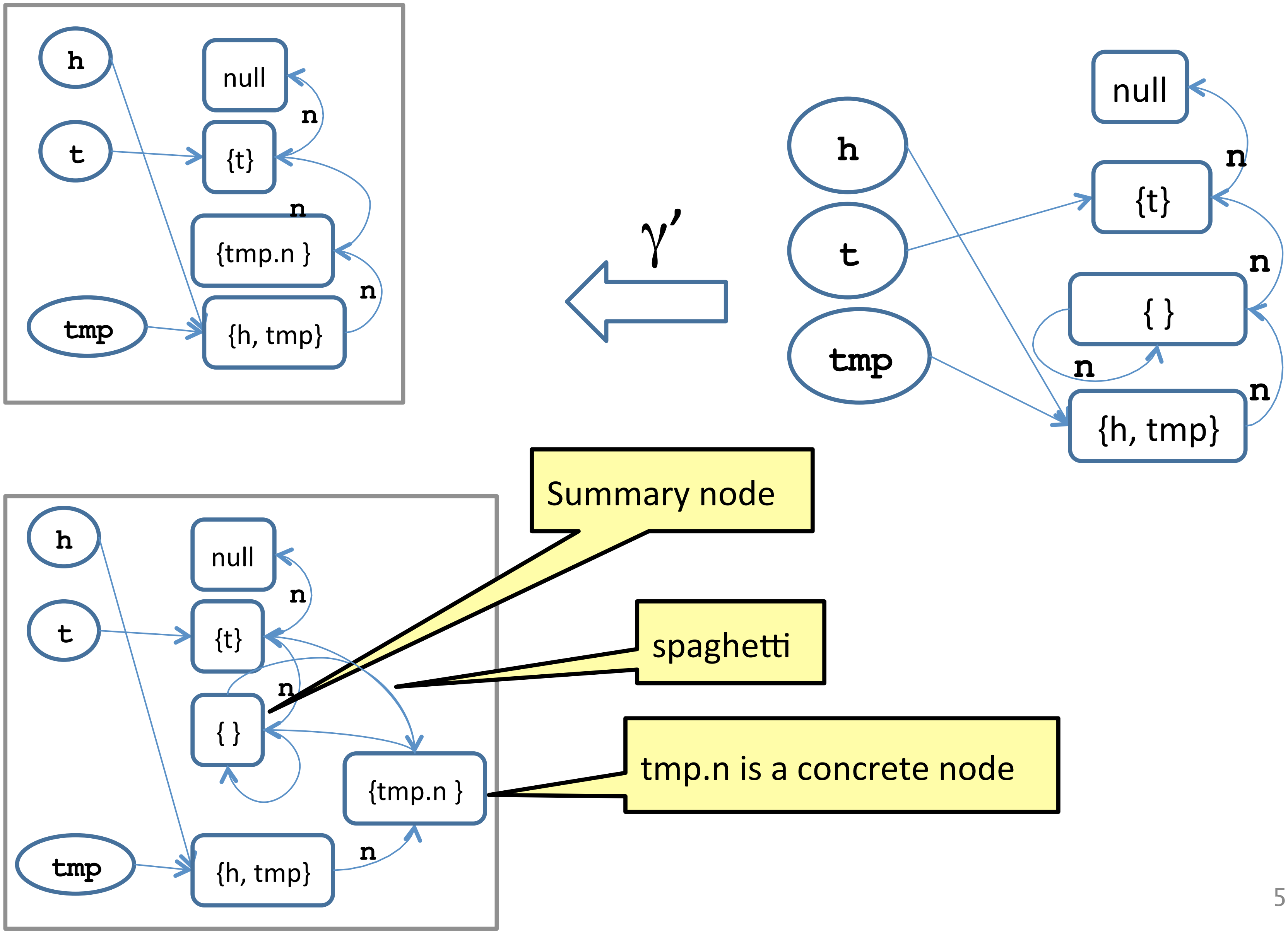
- Transformers accessing only concrete nodes are easy
- Transformers accessing summary nodes are complicated
- Can't concretize summary nodes – represents potentially unbounded number of concrete nodes
- We need to split into cases by "materializing" concrete nodes from summary node
  - Introduce a new temporary predicate tmp.n
  - Partial concretization

Transformer for tmp=tmp.n: γ'

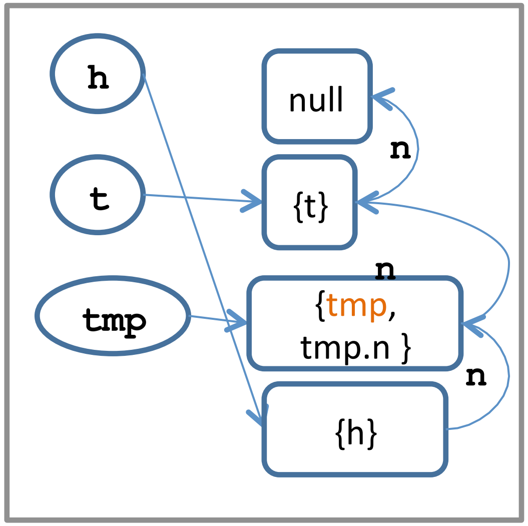


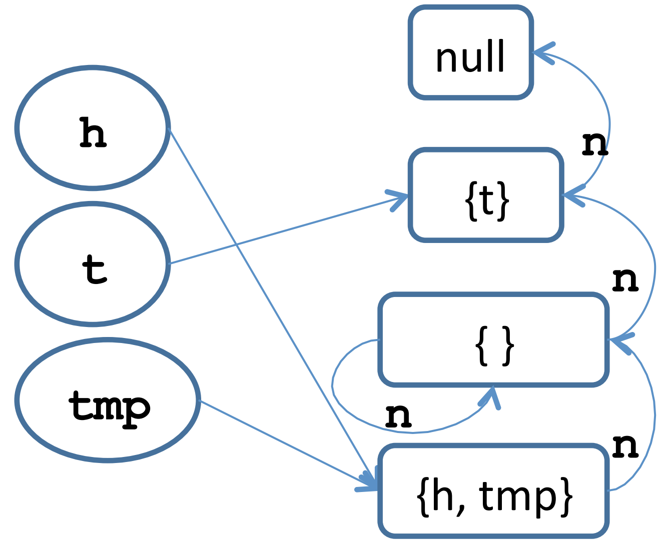
Case 1: Exactly 1 object.
Case 2: >1 objects.

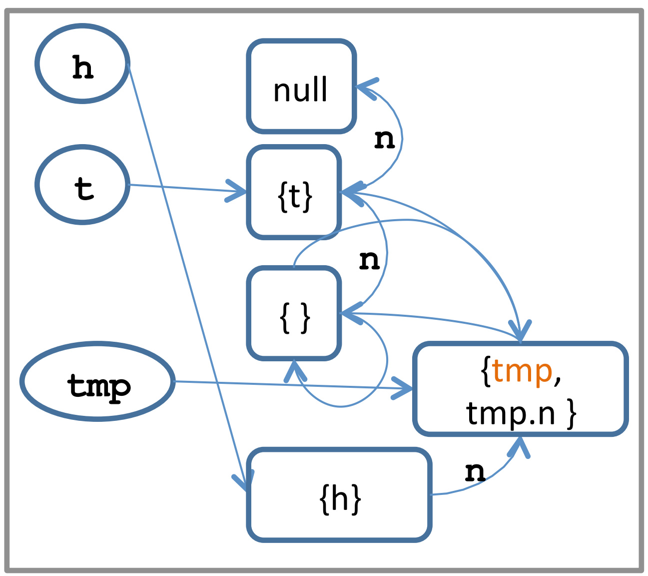
Transformer for tmp=tmp.n: γ'



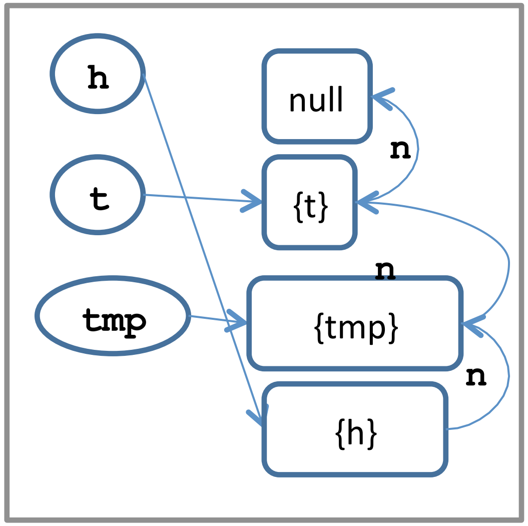
Transformer ⟦tmp=tmp.n⟧

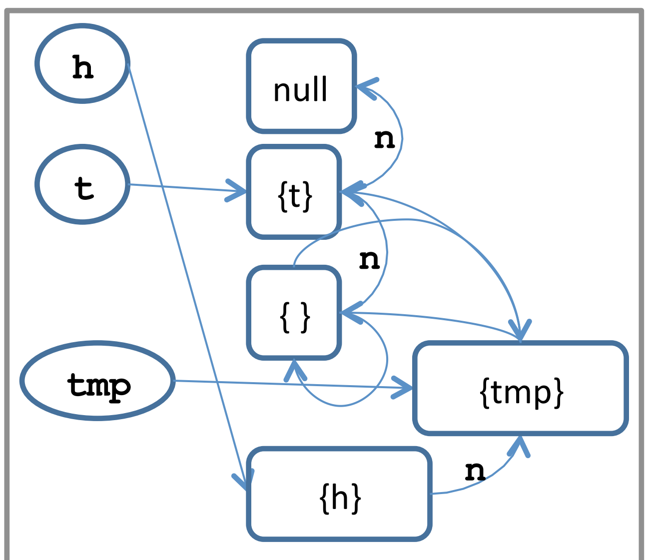






Transformer for tmp=tmp.n: α

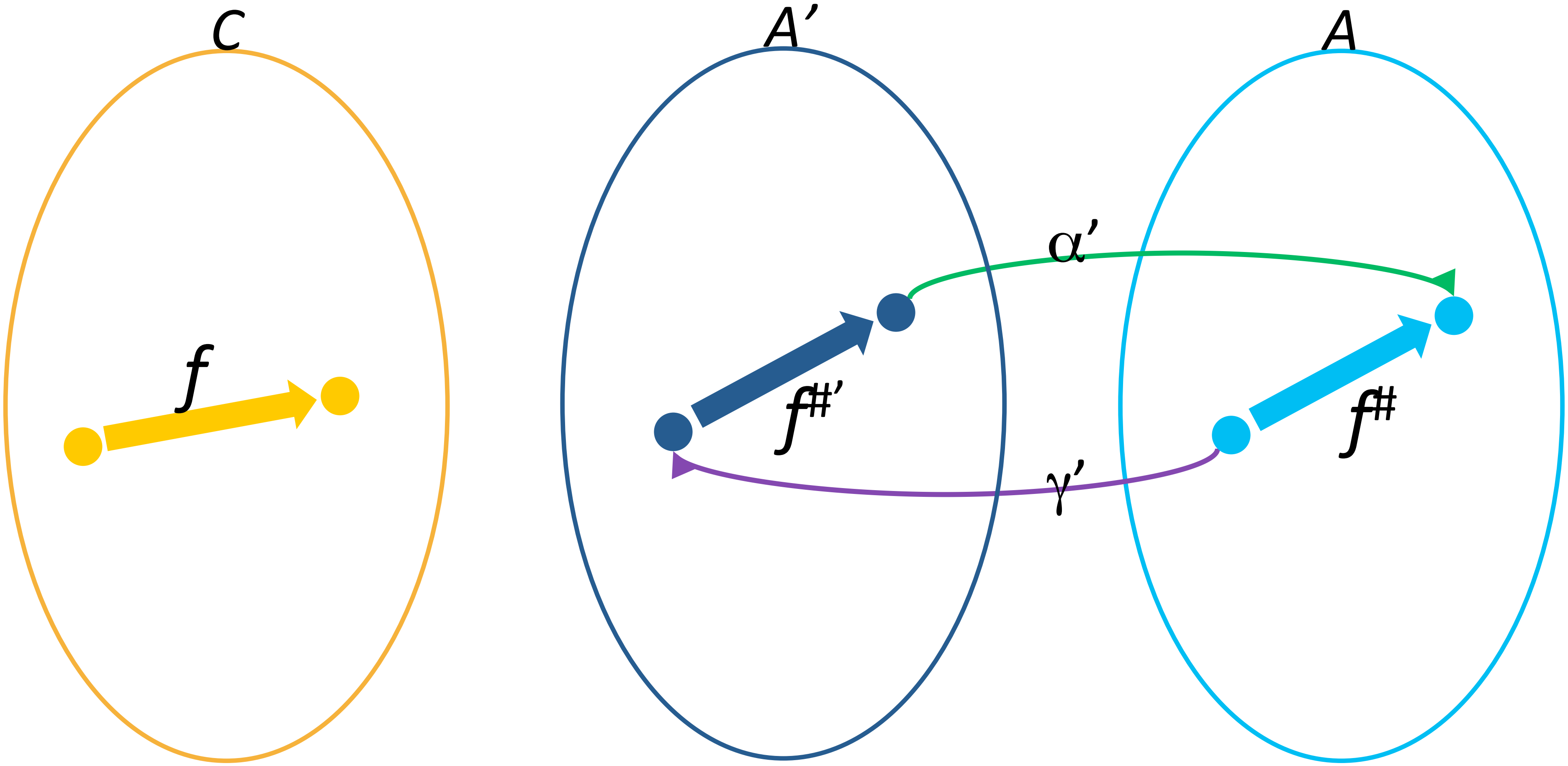





```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

     // Process elements
     tmp = h;
     while (tmp != t) {
       assert tmp != null;
       tmp.data += 1;
       tmp = tmp.n;
     }
```

Transformer via partial-concretization

$$f^{\#}(a) = \alpha'(f^{\#'}(\gamma'(a)))$$



Recap

- Adding more properties to nodes refines abstraction
- Can add temporary properties for partial concretization
  - Materialize concrete nodes from summary nodes
  - Allows turning weak updates into strong ones
  - Focus operation in shape-analysis lingo
  - Not trivial in general and requires more semantic reduction to clean up impossible edges
  - General algorithms available via 3-valued logic and implemented in TVLA system

3-Value logic based shape analysis

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

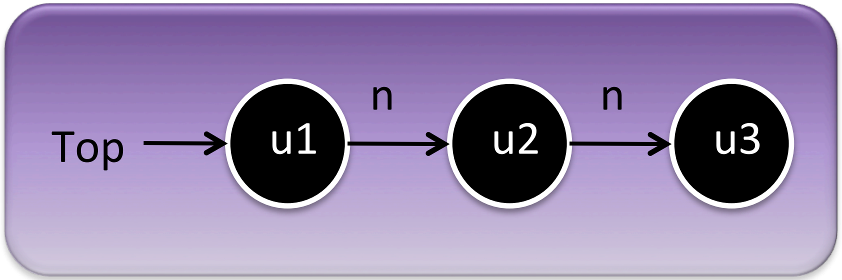
Shape Analysis via 3-valued Logic

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

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

Concrete State

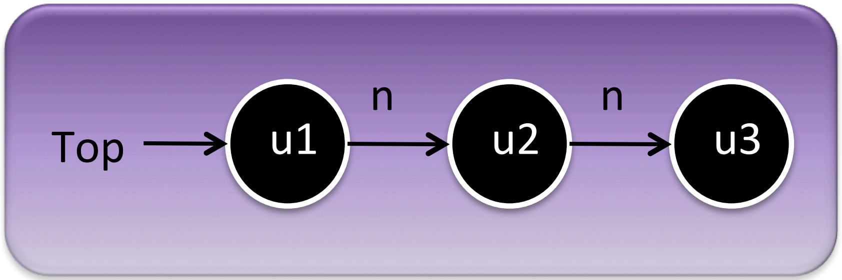
- represent a concrete state as a two-valued logical structure
  - Individuals = heap allocated objects
  - Unary predicates = object properties
  - Binary predicates = relations
- parametric vocabulary



(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));
```

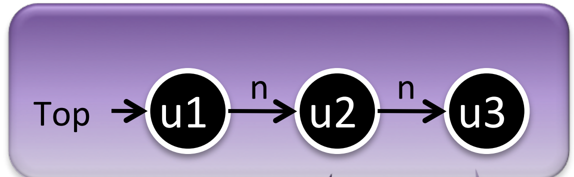
∃w: x(w)

∃w: x(w)

```
 Top = x;

}
```

¬∃v1,v2: n(v1, v2) ∧ n*(v2, v1)

¬∃v1,v2: n(v1, v2) ∧Top(v2)



Top != null

∃w:Top(w) 1

No node precedes Top

¬∃v1,v2: n(v1, v2) ∧Top(v2) 1

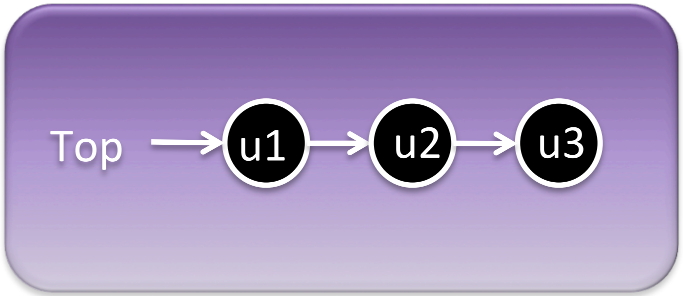
No Cycles

¬∃v1,v2: n(v1, v2) ∧ n*(v2, v1) 1

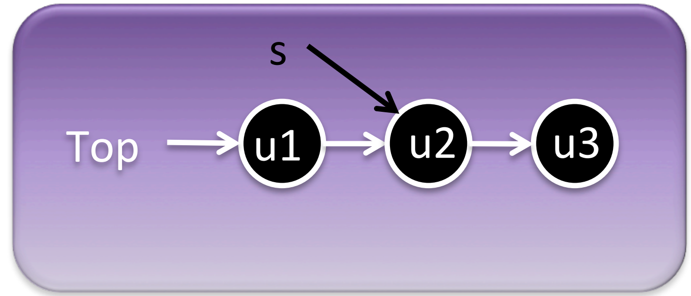
Concrete Interpretation Rules

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

Example: $s = Top \rightarrow n$



$s'(v) = \exists v1{:}\ Top(v1) \land n(v1,v)$



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

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	1
u3	0

Collecting Semantics

CSS [v] =

- $\{ <\varnothing,\varnothing> \}$ — if v = entry
- otherwise:

$$\bigcup_{\substack{(w,v) \in E(G),\\ w \in Assignments(G)}} \{ [\![st(w)]\!](S) \mid S \in CSS[w] \} \cup$$

$$\bigcup_{\substack{(w,v) \in E(G),\\ w \in Skip(G)}} \{ S \mid S \in CSS[w] \} \cup$$

$$\bigcup_{(w,v) \in True\text{-}Branches(G)} \{ S \mid S \in CSS[w] \text{ and } S \vDash cond(w)\} \cup$$

$$\bigcup_{(w,v) \in False\text{-}Branches(G)} \{ S \mid S \in CSS[w] \text{ and } S \vDash \neg cond(w)\}$$

othrewise

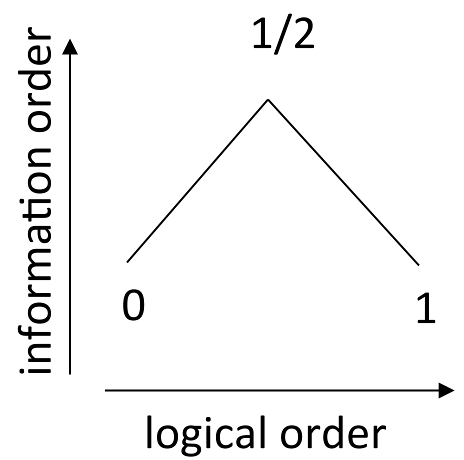
Collecting Semantics

- 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 $\subseteq$

Embedding Order

- Given two structures $S = \langle U, \iota \rangle$, $S' = \langle U', \iota' \rangle$ and an onto function $f : U \rightarrow U'$ mapping individuals in U to individuals in U'
- We say that f embeds S in S' (denoted by $S \sqsubseteq S'$) if
  - for every predicate symbol $p \in P$ of arity k: $u1, \ldots, uk \in U$, $\iota(p)(u1, \ldots, uk) \sqsubseteq \iota'(p)(f(u1), \ldots, f(uk))$
  - and for all $u' \in U'$ $(| \{ u \mid f(u) = u' \} | > 1) \sqsubseteq \iota'(sm)(u')$
- We say that S can be embedded in S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$

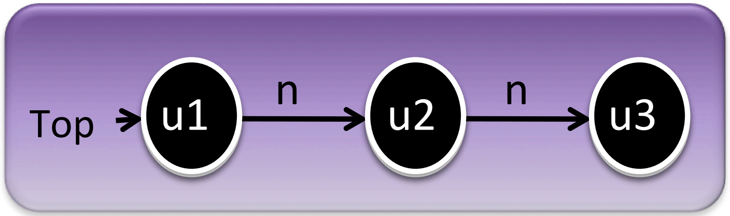
Tight Embedding

- S' = <U', ι'> is a tight embedding of S=< U, ι > with respect to a function f if:
  - S' does not lose unnecessary information

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

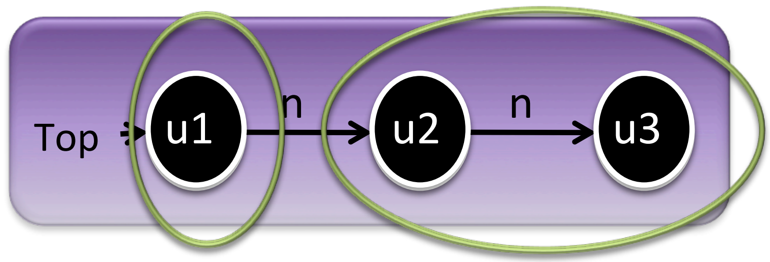
- One way to get tight embedding is canonical abstraction

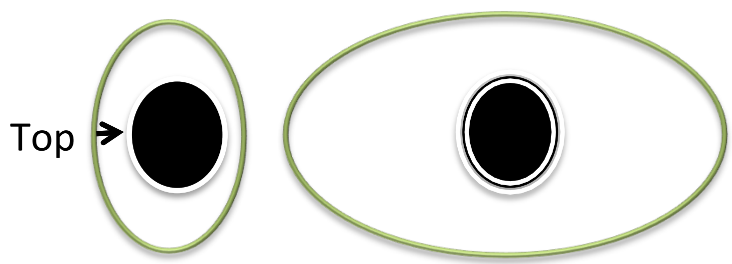
Canonical Abstraction



[Sagiv, Reps, Wilhelm, TOPLAS02]

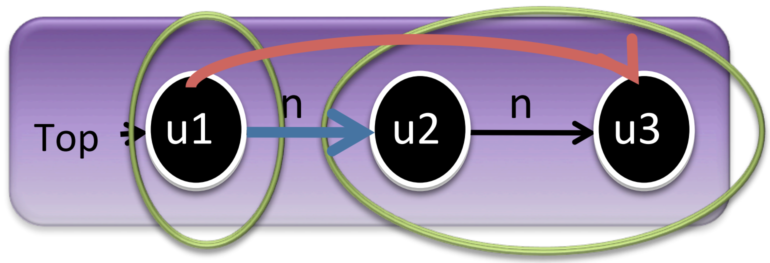
Canonical Abstraction

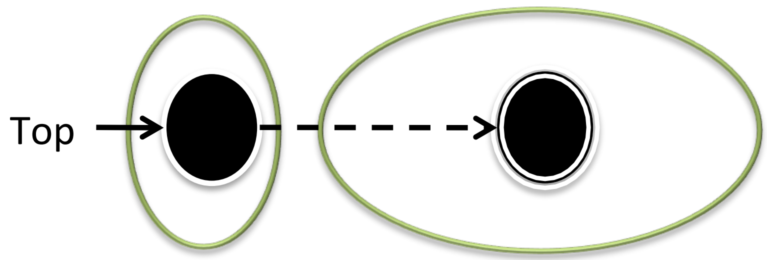




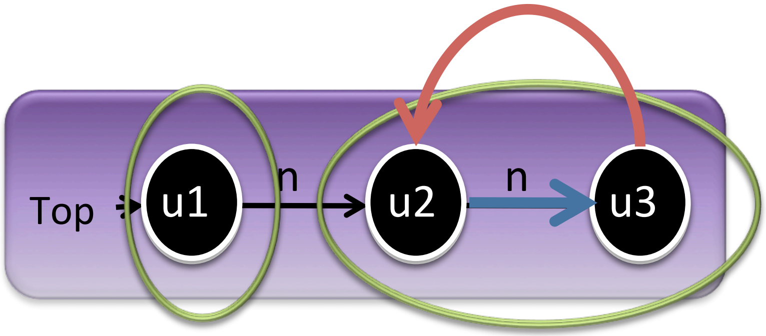
[Sagiv, Reps, Wilhelm, TOPLAS02]

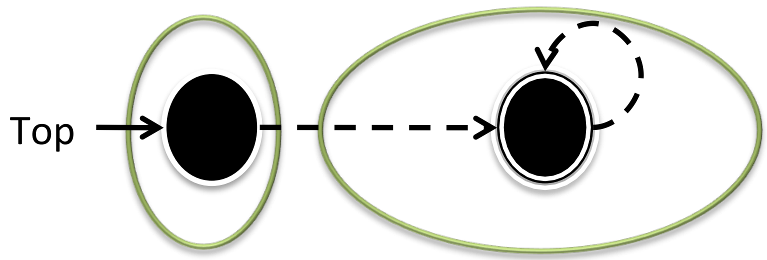
Canonical Abstraction



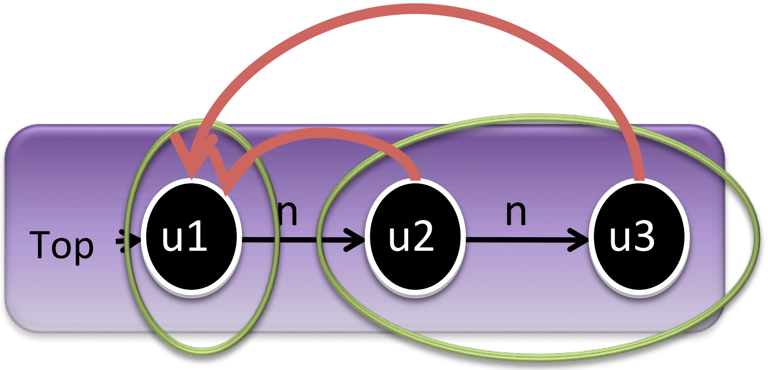


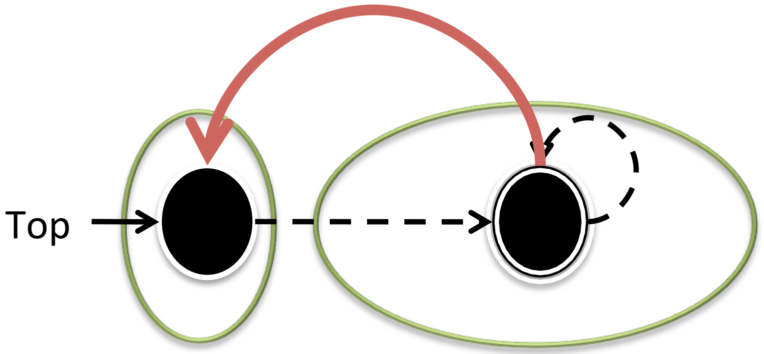
Canonical Abstraction



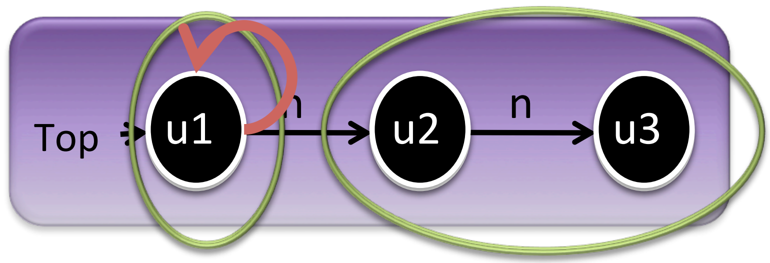


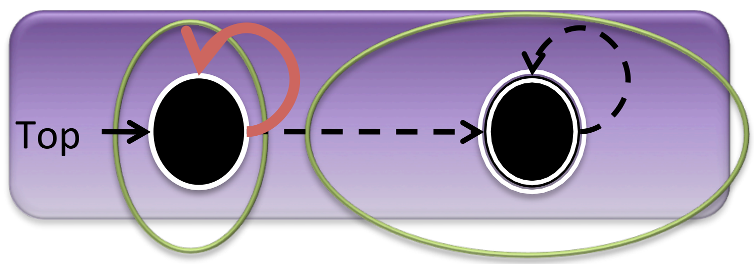
Canonical Abstraction





Canonical Abstraction





Canonical Abstraction (β)

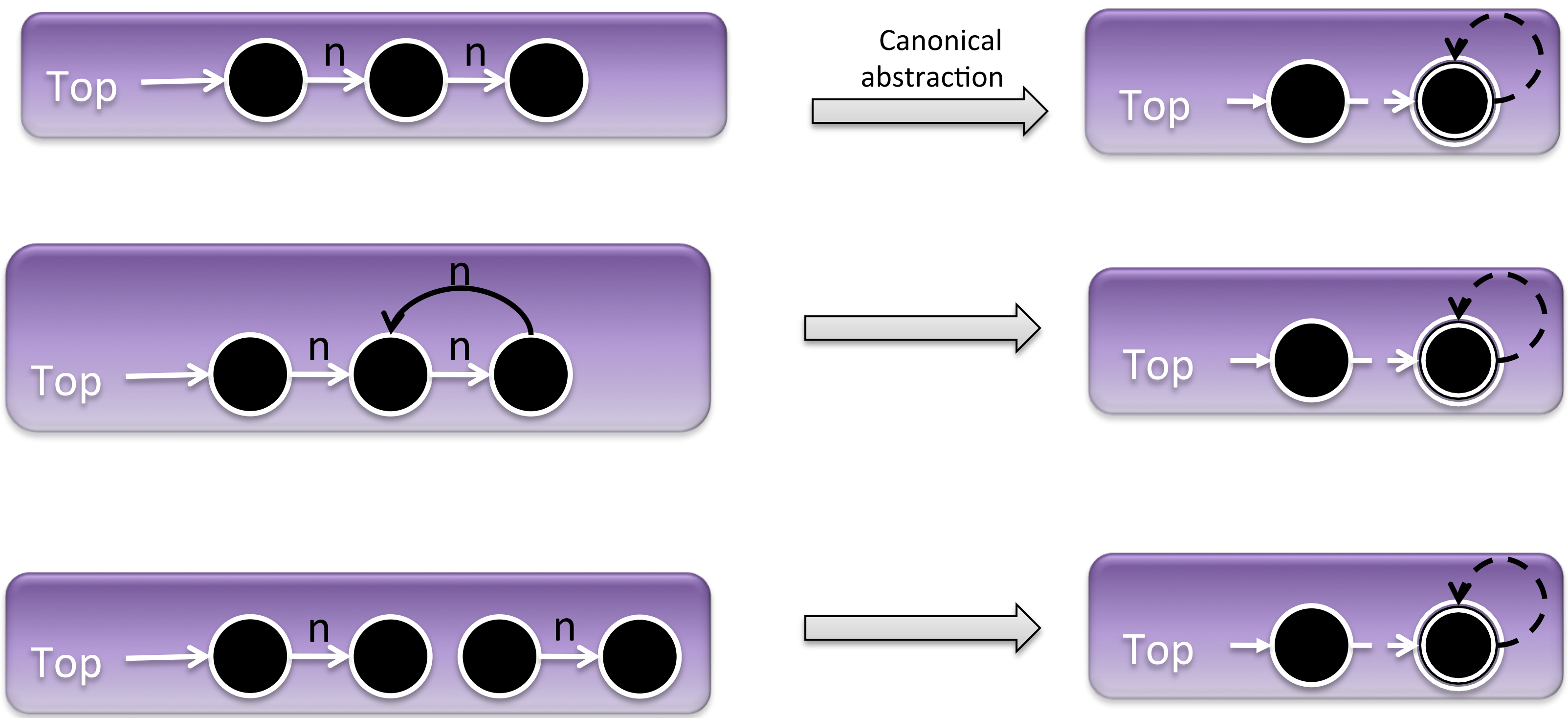
- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values

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

- 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

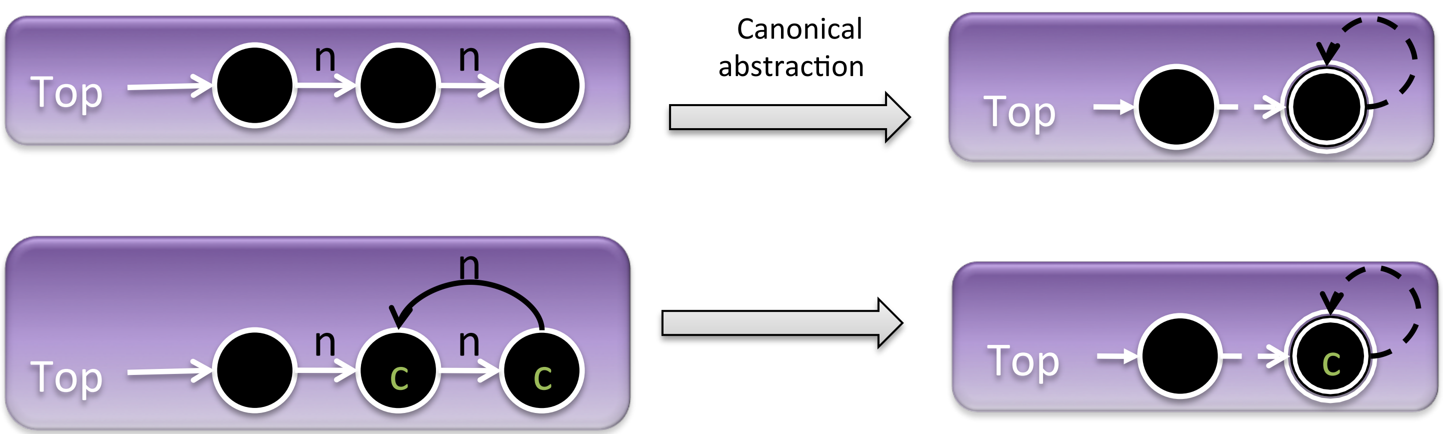


Instrumentation Predicates

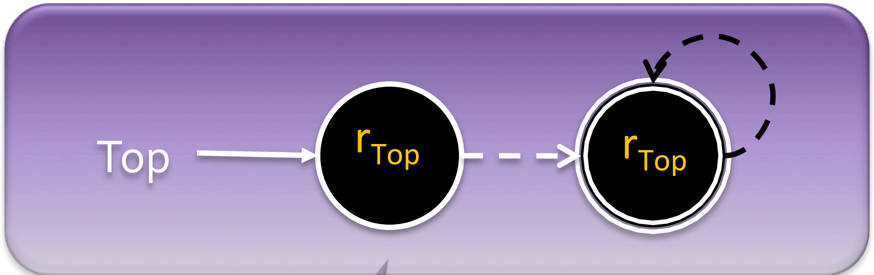
- Record additional derived information via predicates

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

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



Embedding Theorem:
Conservatively Observing Properties



No Cycles

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

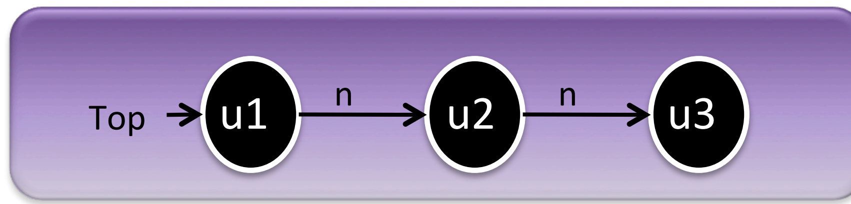
No cycles (derived)

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

Operational Semantics

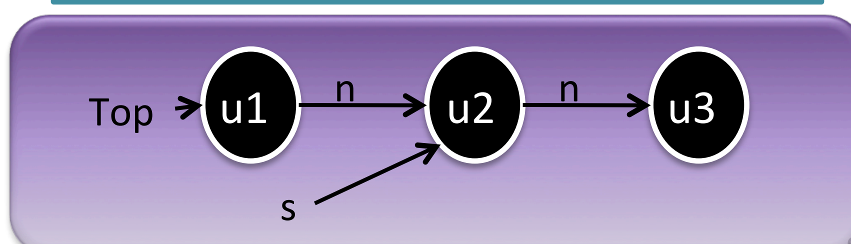
```
void push (int v) {
 Node *x = malloc(sizeof(Node));
 x->d = v;
 x->n = Top;
 Top = x;
}
```

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

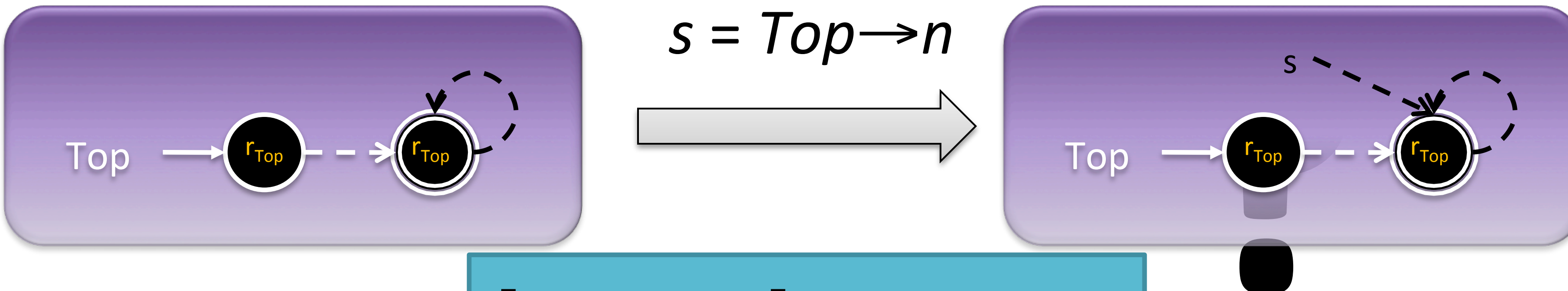


⟦s = Top->n⟧

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



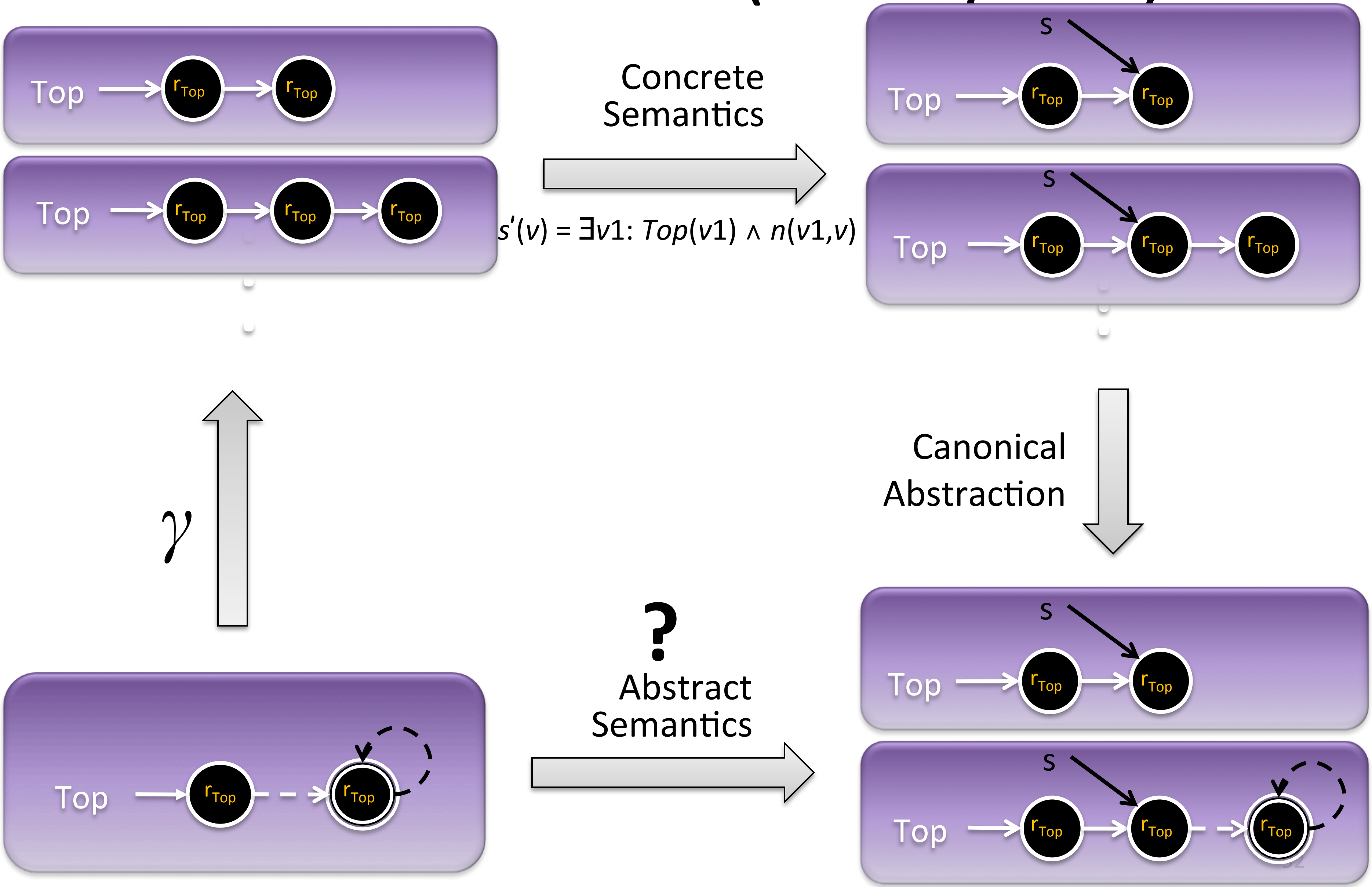
Abstract Semantics



⟦s = Top->n⟧

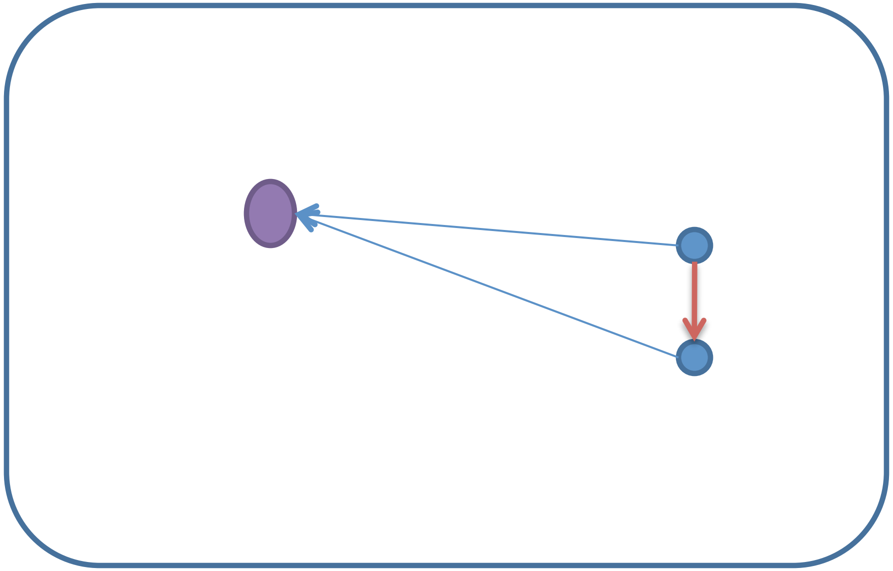
$s'(v) = \exists v1: Top(v1) \land n(v1,v)$

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



Semantic Reduction

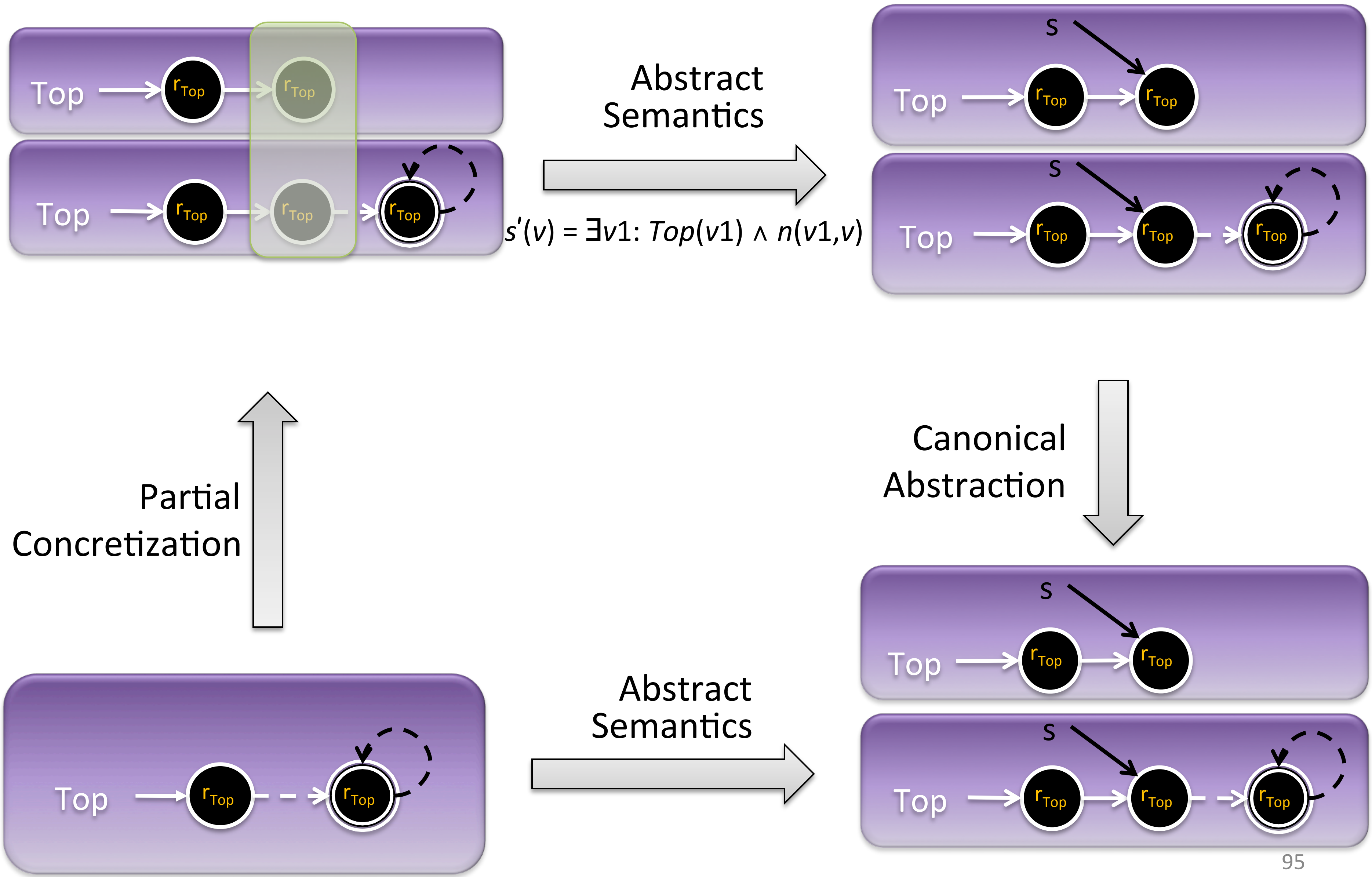
- Improve the precision of the analysis by recovering properties of the program semantics
- A Galois connection (C, α, γ, A)
- An operation op:A→A is a semantic reduction when
  - $\forall l \in L_2$ op(l)$\sqsubseteq$l and
  - γ(op(l)) = γ(l)



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→n)



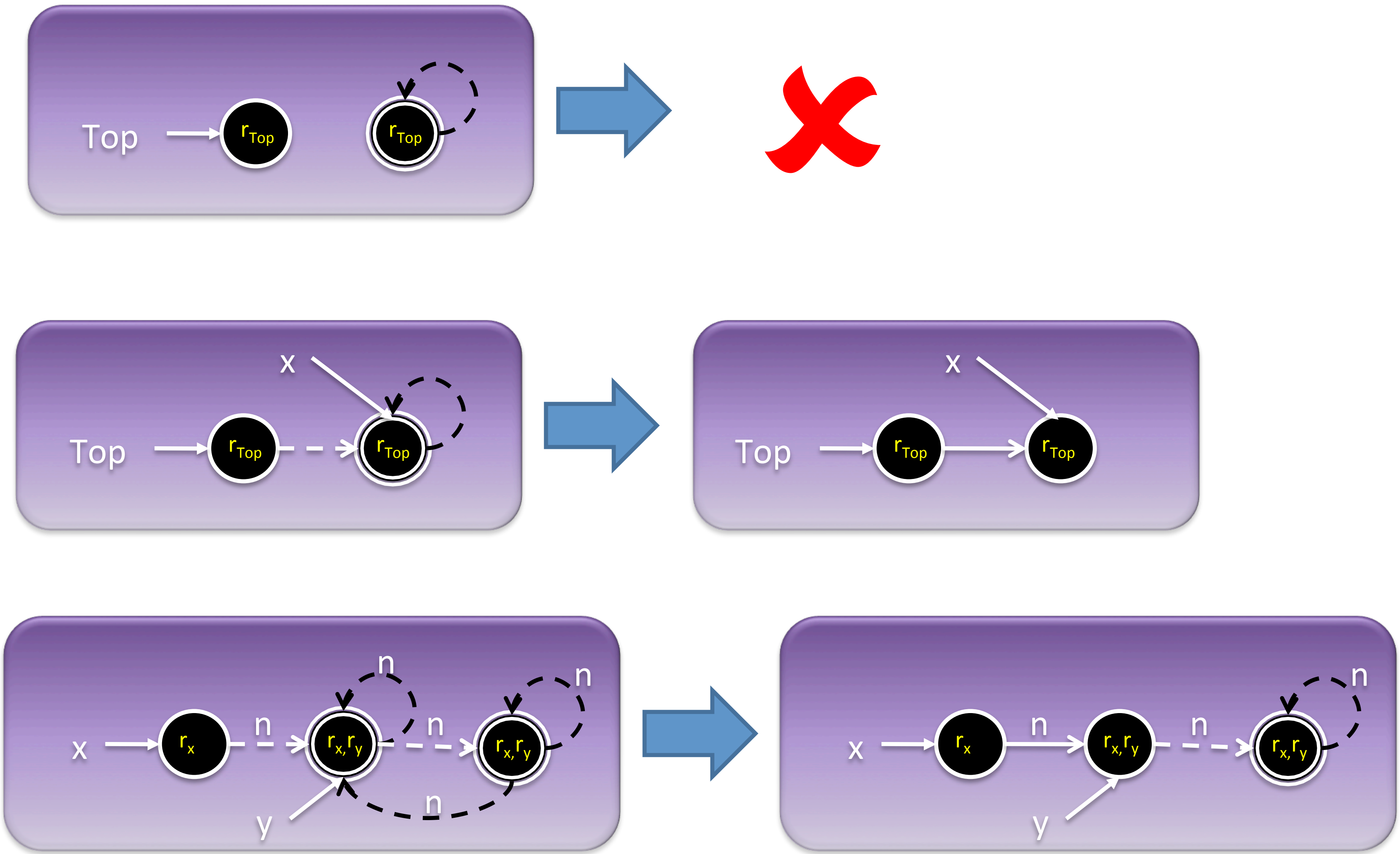
Partial Concretization

- 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]

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

$$
SS[v] = \begin{cases}
\{ <\emptyset,\emptyset> \} & \text{if } v = \text{entry} \\[2ex]
\bigcup_{\substack{(w,v) \in E(G), \\ w \in Assignments(G)}} \{ t\_embed(coerce(\llbracket st(w) \rrbracket_3(focus_{F(w)}(SS[w])))) \cup & \\[2ex]
\bigcup_{\substack{(w,v) \in E(G), \\ w \in Skip(G)}} \{ S \mid S \in SS[w] \} \cup & \text{othrewise} \\[2ex]
\bigcup_{(w,v) \in True\text{-}Branches(G)} \{ t\_embed(S) \mid S \in coerce(\llbracket st(w) \rrbracket_3(focus_{F(w)}(SS[w]))) \text{ and } S \models 3 \; cond(w)\} \cup & \\[2ex]
\bigcup_{(w,v) \in False\text{-}Branches(G)} \{ t\_embed(S) \mid S \in coerce(\llbracket st(w) \rrbracket_3(focus_{F(w)}(SS[w]))) \text{ and } S \models_3 \neg cond(w)\} \cup &
\end{cases}
$$

Recap

- Abstraction
  - canonical abstraction
  - recording derived information
- Transformers
  - partial concretization (focus)
  - constraint solver (coerce)
  - sound information extraction

Stack Push

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

∃v: x(v)

```
x→d     v;
```

∃v: x(v)

```
x→
```

```
Top = x;
}
```

¬∃v1,v2: n(v1, v2) ∧Top(v2)

∀v:¬c(v)