Program Analysis and Verification

0368-4479

Noam Rinetzky

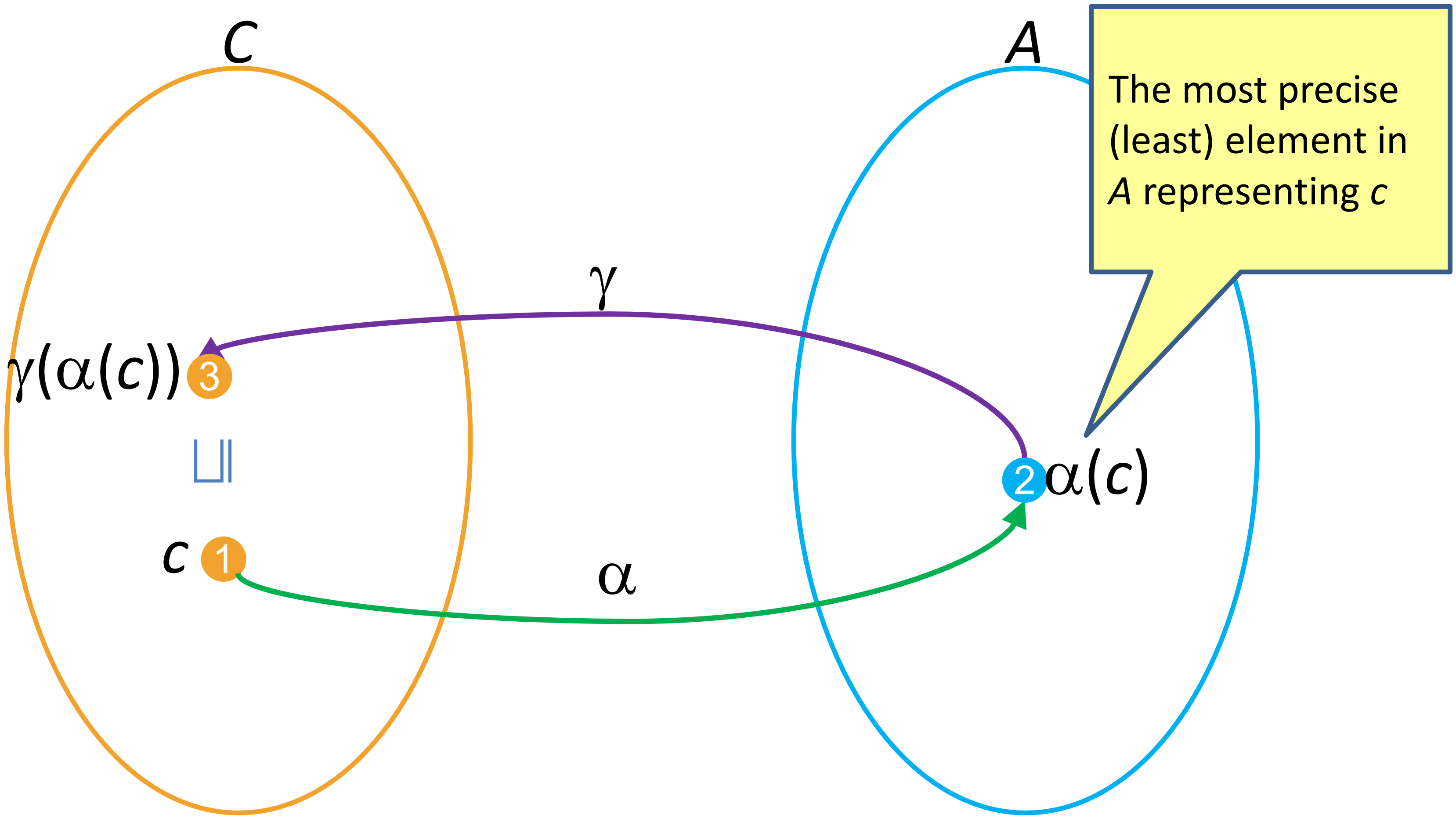
Lecture 10: Shape Analysis + Numerical Analysis

Slides credit: Roman Manevich, Mooly Sagiv, Eran Yahav

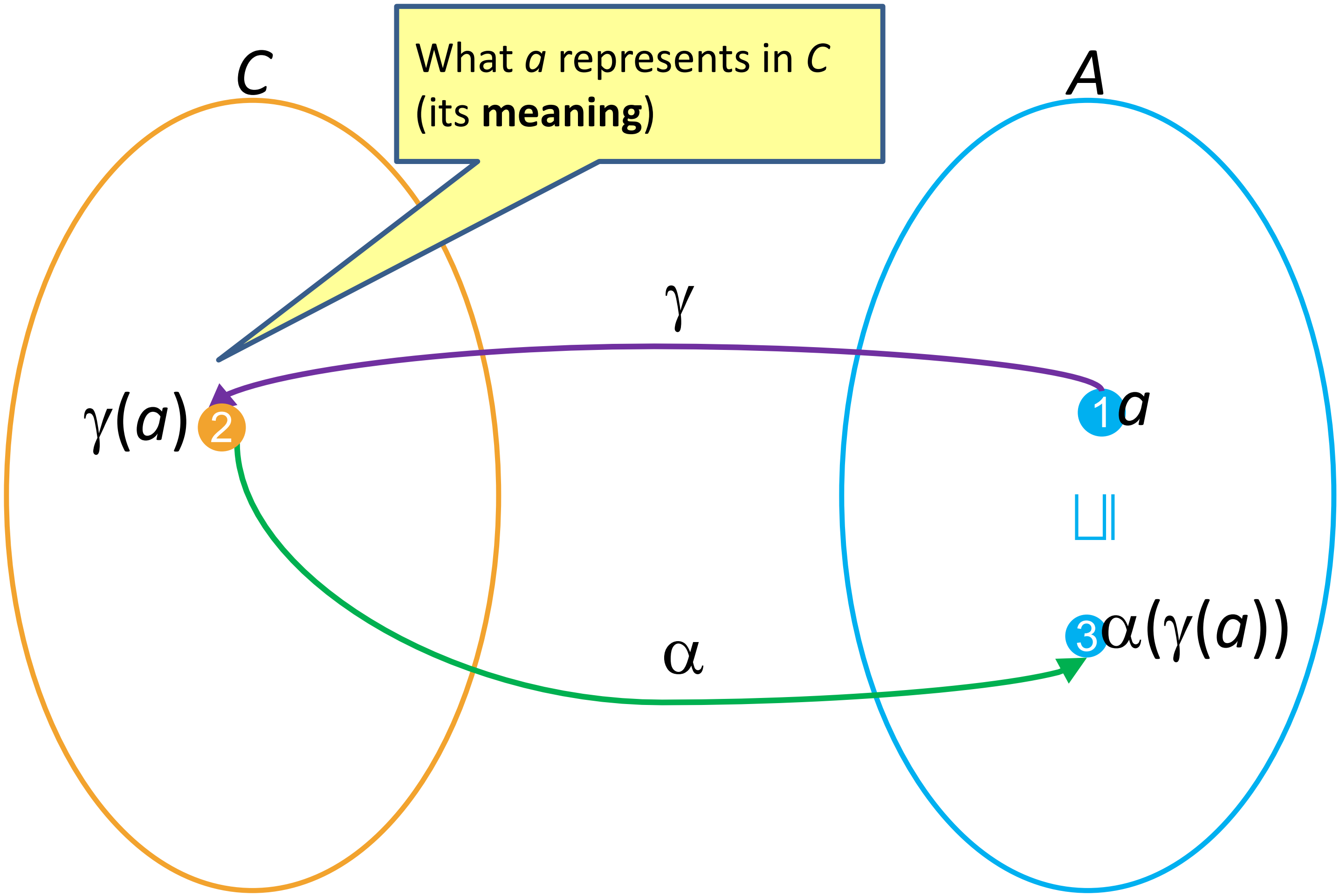
Abstract Interpretation [Cousot'77]

- Mathematical foundation of static analysis

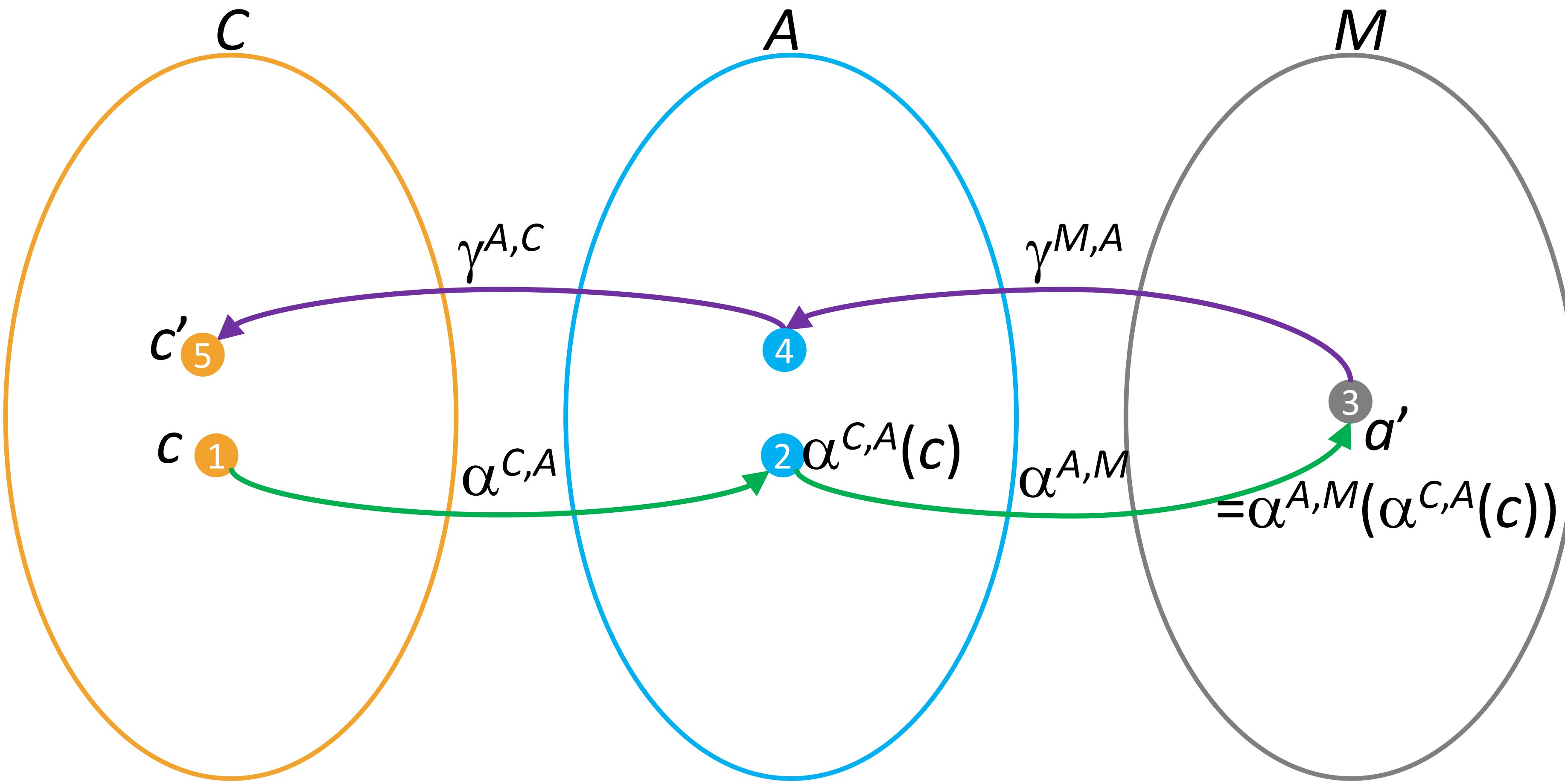
Galois Connection: $c \sqsubseteq \gamma(\alpha(c))$



Galois Connection: $\alpha(\gamma(a)) \sqsubseteq a$

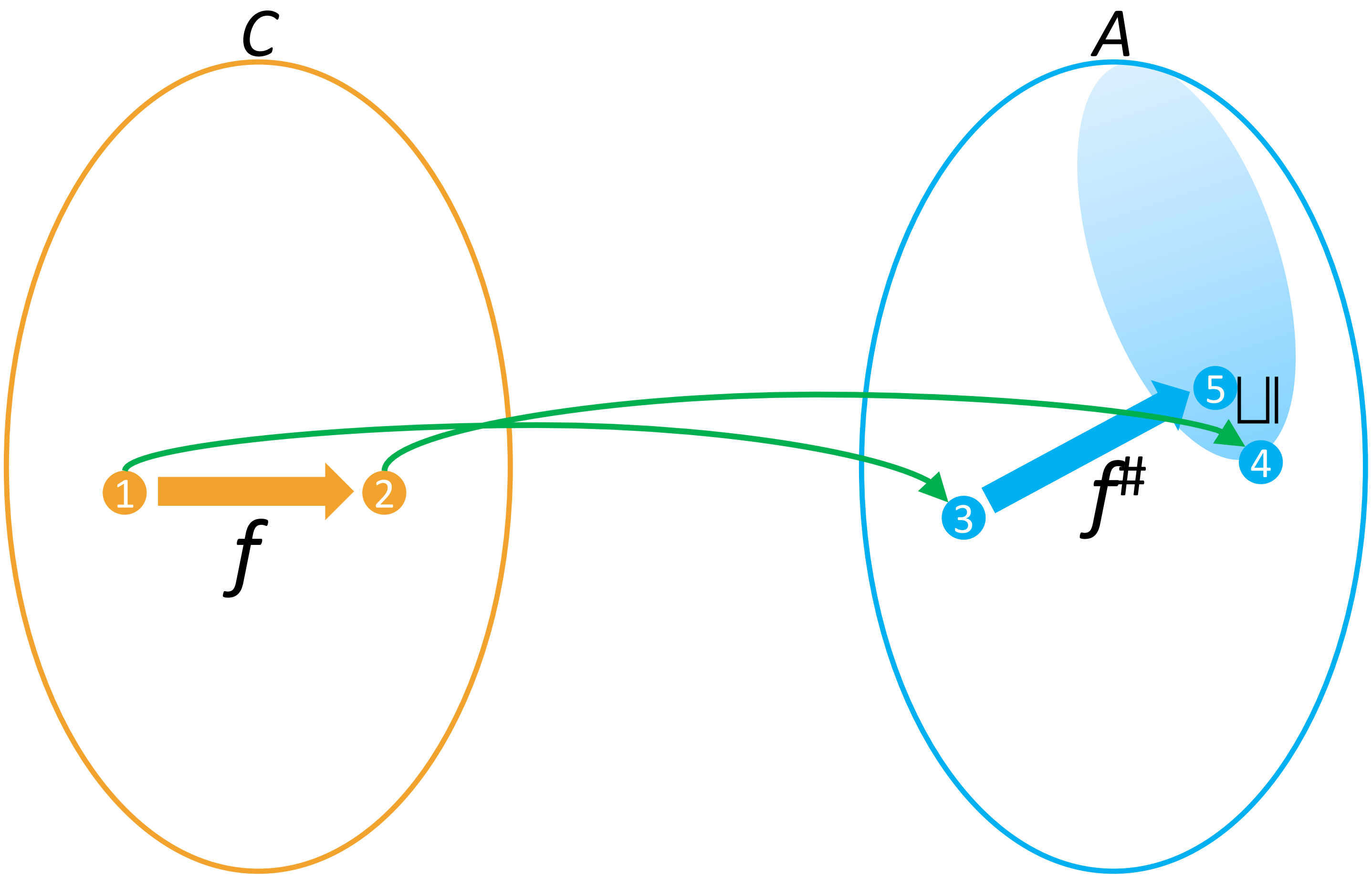


Inducing along the connections



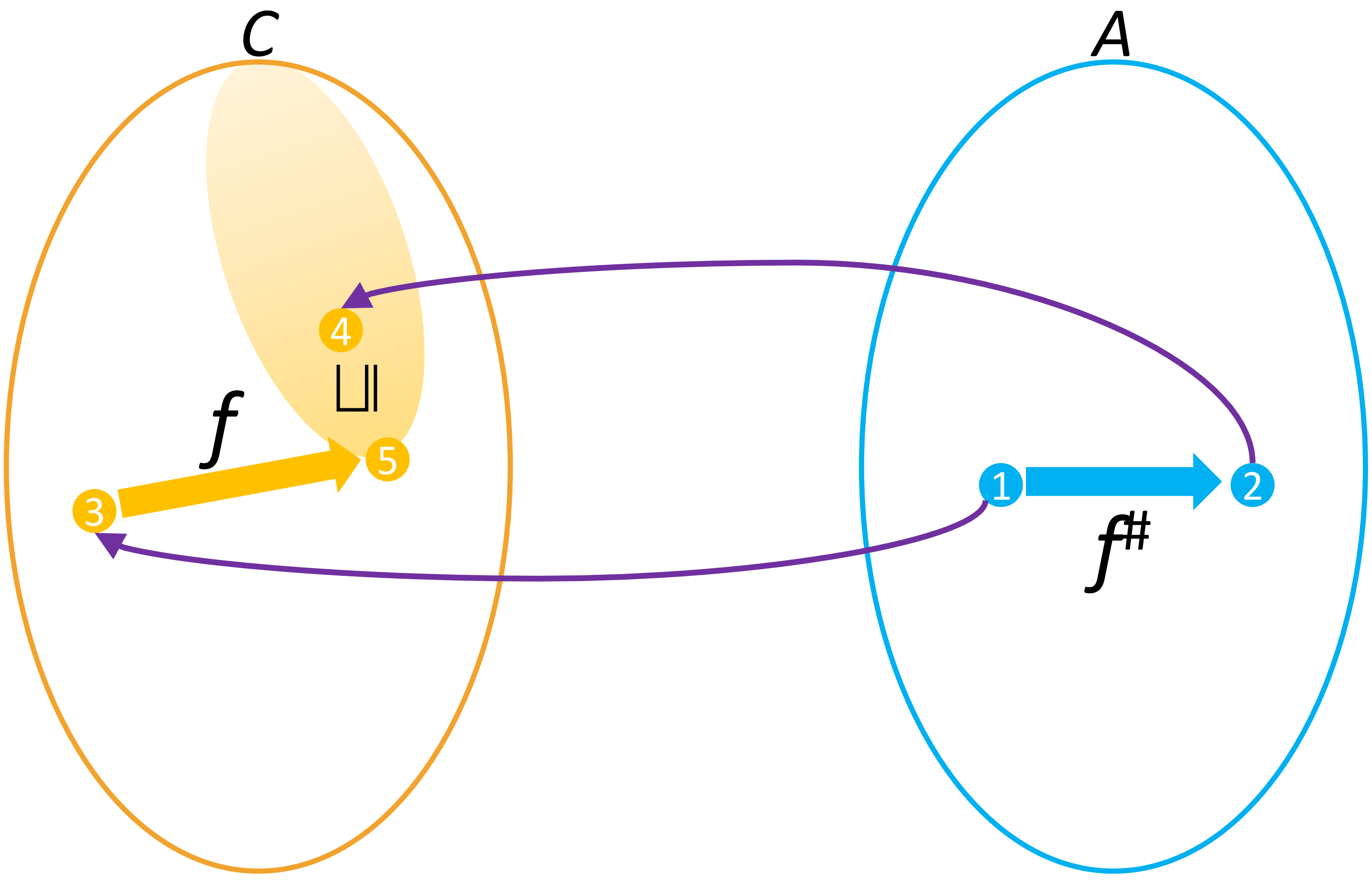
Transformer soundness condition 1

$$\forall c\colon f(c)=c' \Rightarrow \alpha(f^{\#}(c)) \sqsupseteq \alpha(c')$$



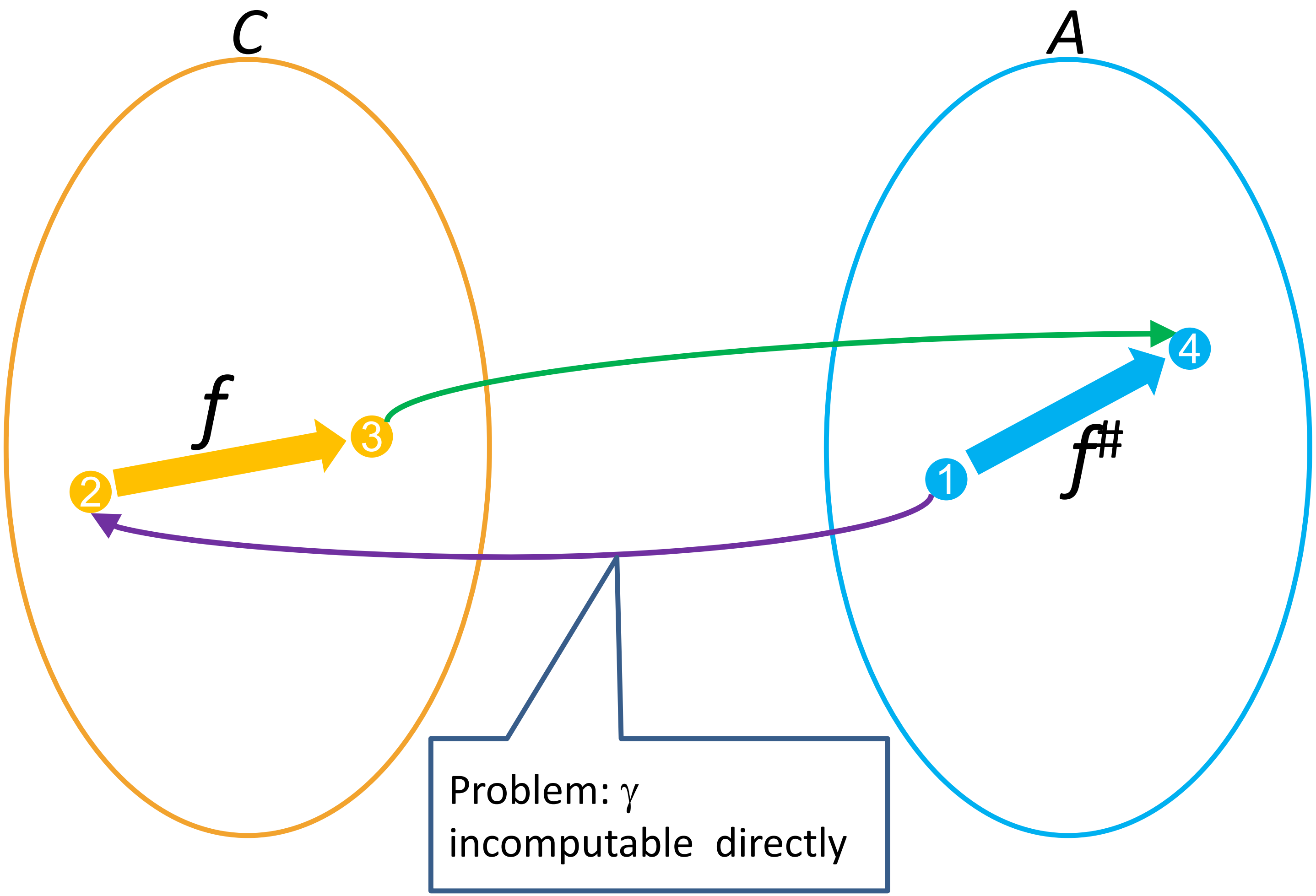
Transformer soundness condition 2

$$\forall a: f^{\#}(a)=a' \Rightarrow f(\gamma(a)) \sqsubseteq \gamma(a')$$



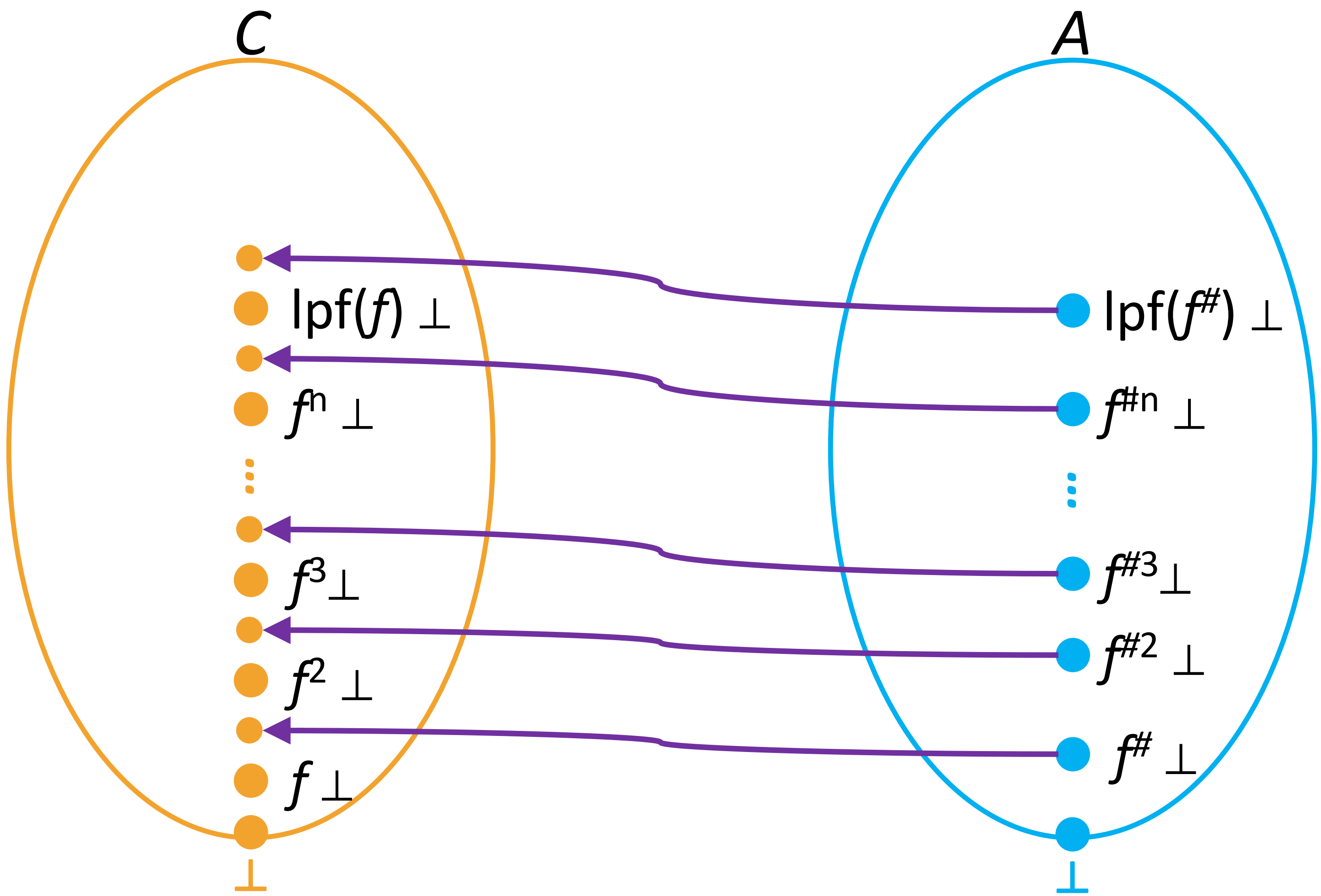
Best (induced) transformer

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



Soundness theorem 1

$$\forall a \in D^A : f(\gamma(a)) \sqsubseteq \gamma(f^{\#}(a)) \quad \Rightarrow \forall a \in D^A : f^n(\gamma(a)) \sqsubseteq \gamma(f^{\#n}(a))$$
$$\Rightarrow \forall a \in D^A : \mathrm{lfp}(f^n)(\gamma(a)) \sqsubseteq \gamma(\mathrm{lfp}(f^{\#n})(a))$$
$$\Rightarrow \mathrm{lfp}(f) \perp \sqsubseteq \mathrm{lfp}(f^{\#}) \perp$$

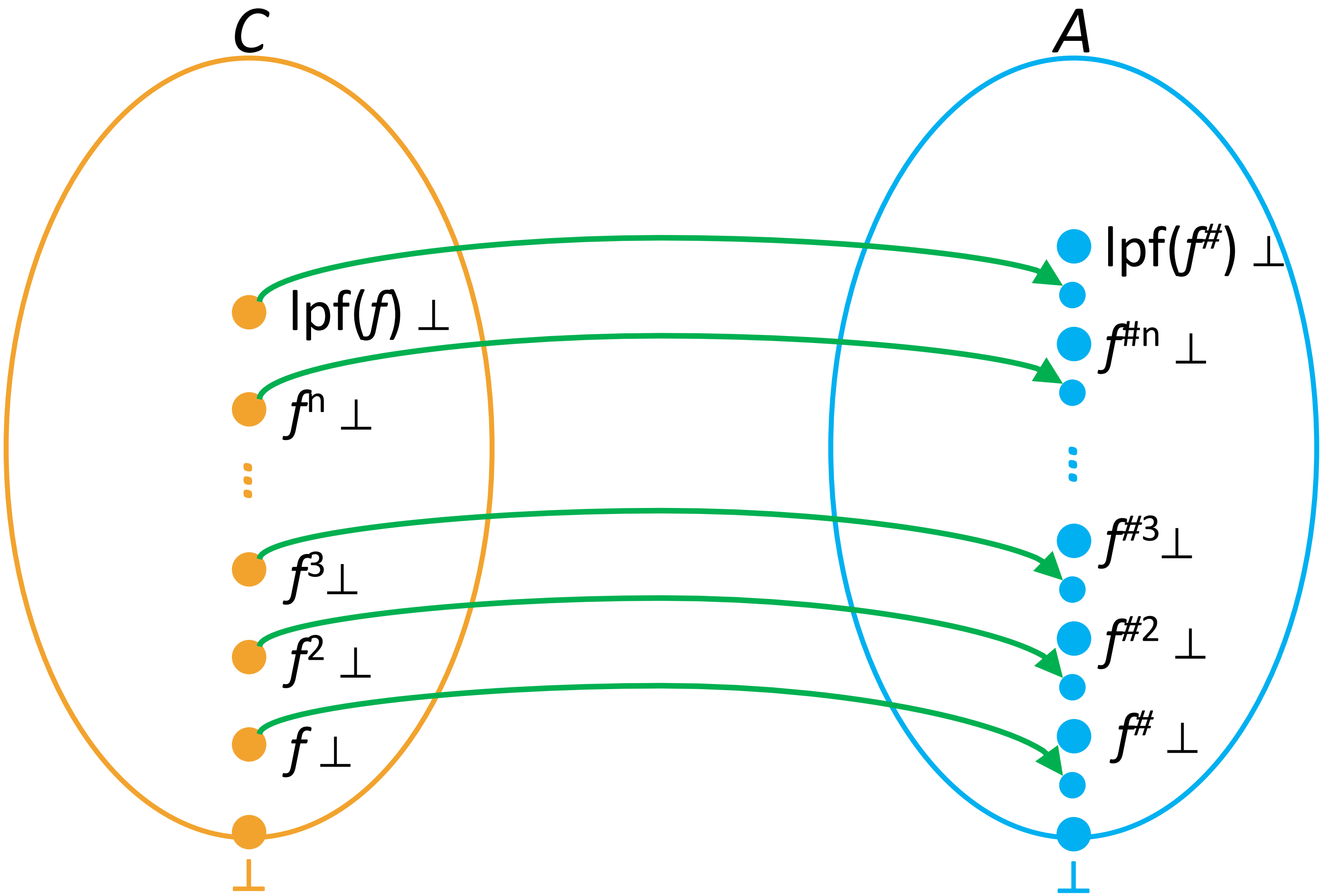


Soundness theorem 2

$\forall c \in D^C : \alpha(f(c)) \sqsubseteq f^\#(\alpha(c)) \Rightarrow \forall c \in D^C : \alpha(f^n(c)) \sqsubseteq f^{\#n}(\alpha(c))$

$\Rightarrow \forall c \in D^C : \alpha(\mathrm{lfp}(f)(c)) \sqsubseteq \mathrm{lfp}(f^\#)(\alpha(c))$

$\Rightarrow \mathrm{lfp}(f) \perp \sqsubseteq \mathrm{lfp}(f^\#) \perp$



Pointer Analysis

- Points-To Analysis
  - may-point-to
  - must-point-to

- Alias Analysis
  - may-alias
  - must-alias

Applications

- Compiler optimizations
  - Method de-virtualization
  - Call graph construction
  - Allocating objects on stack via escape analysis
- Verification & Bug Finding
  - Data race detection
  - Use in preliminary phases
  - Use in verification itself

PWhile syntax

- A primitive statement is of the form
  - x := null
  - x := y
  - x := *y
  - x := &y;
  - *x := y
  - skip

  (where x and y are variables in Var)

Omitted (for now)
- *Dynamic memory allocation*
- *Pointer arithmetic*
- *Structures and fields*
- *Procedures*

Destructive Update: *x = y

- Strong updates
- Weak Updates

Points-to analysis: a simple example

```
p = &x;        {p=&x}
q = &y;        {p=&x ∧ q=&y}
if (?) {
  q = p;       {p=&x ∧ q=&x}
}              {p=&x ∧ (q=&y ∨ q=&x)}
x = &a;        {p=&x ∧ (q=&y ∨ q=&x) ∧ x=&a}
y = &b;        {p=&x ∧ (q=&y ∨ q=&x) ∧ x=&a ∧ y=&b}
z = *q;        {p=&x ∧ (q=&y∨q=&x) ∧ x=&a ∧ y=&b}
```

How would you construct an abstract domain to represent these abstract states?

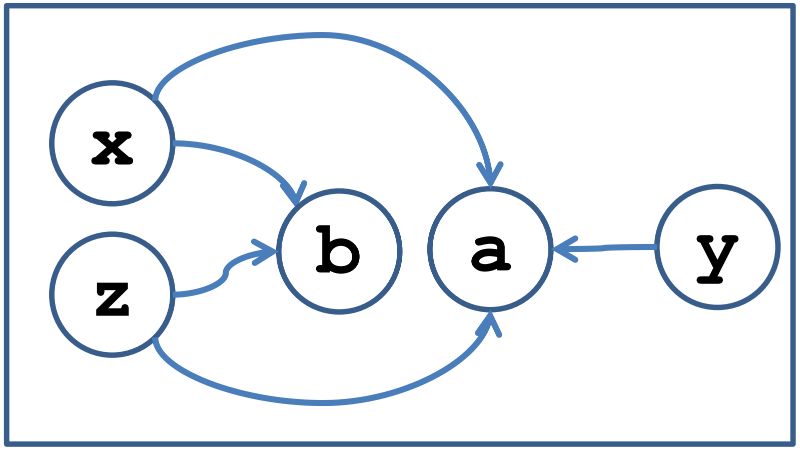
PWhile operational semantics

- **State** : (Var→Z) ∪ (Var→Var∪{null})
- ⟦ x = y ⟧ s =
- ⟦ x = *y ⟧ s =
- ⟦ *x = y ⟧ s =
- ⟦ x = null ⟧ s =
- ⟦ x = &y ⟧ s =

Andersen: Flow-insensitive analysis

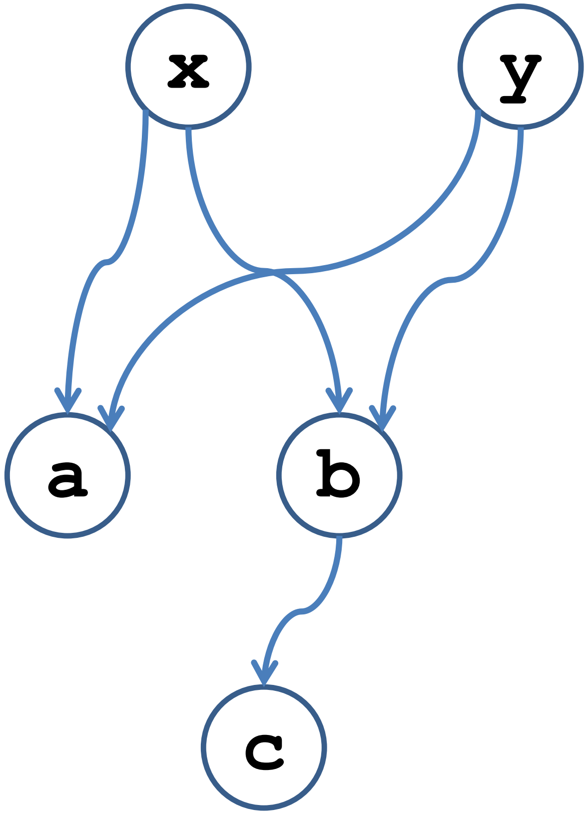
```
L1: x = &a;
L2: y = x;
L3: x = &b;
L4: z = x;
L5:
```

L1-5

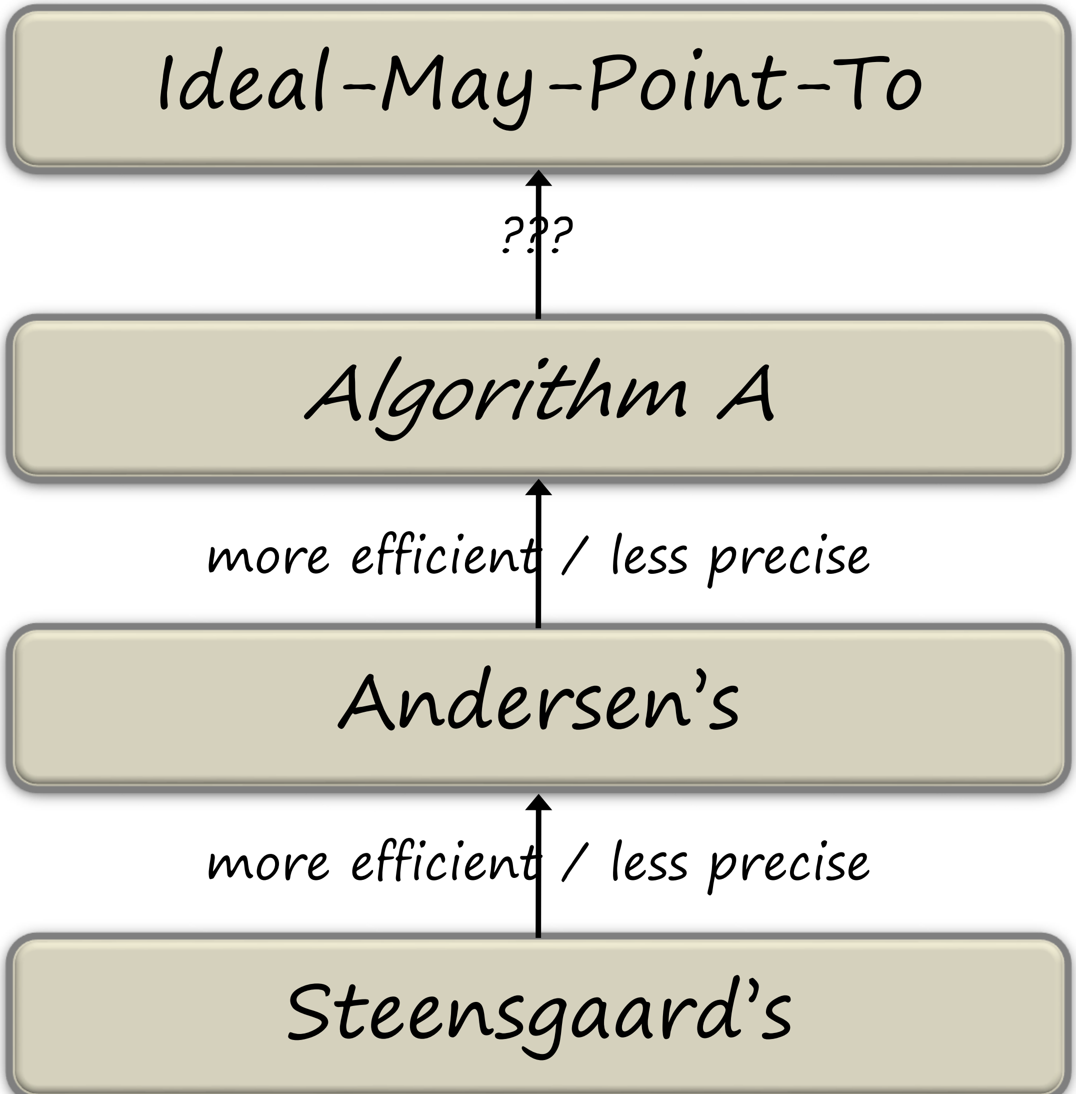


Steensgaard's Flow-insensitive analysis

```
L1: x = &a;
L2: y = x;
L3: y = &b;
L4: b = &c;
L5:
```



May-points-to analyses



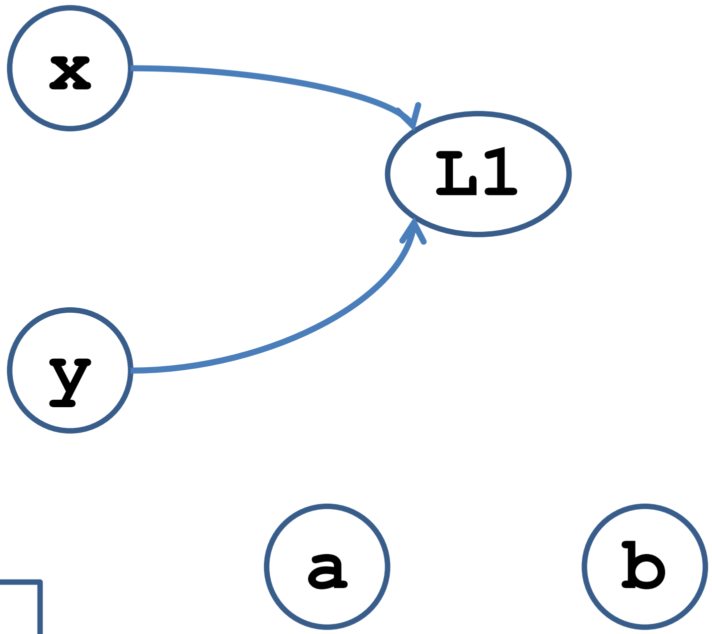
Handling memory allocation

- s: x = new () / malloc ()
- Assume, for now, that allocated object stores one pointer
  - s: x = malloc (sizeof(void*))
- Introduce a pseudo-variable V_s to represent objects allocated at statement s, and use previous algorithm
  - Treat s as if it were "x = &V_s"
  - Also track possible values of V_s
  - Allocation-site based approach
- Key aspect: V_s represents a set of objects (locations), not a single object
  - referred to as a summary object (node)

Dynamic memory allocation example

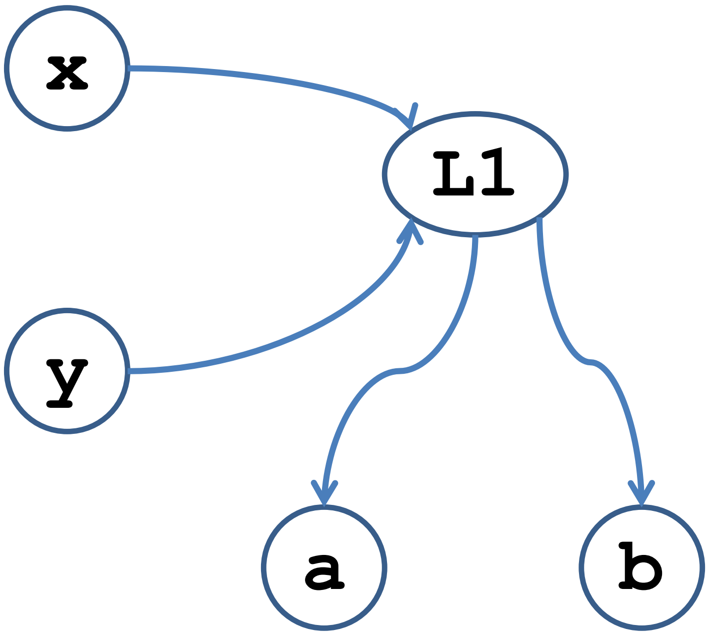
```
L1: x = new O;
L2: y = x;
L3: *y = &b;
L4: *y = &a;
```

How should we handle these statements



Summary object update

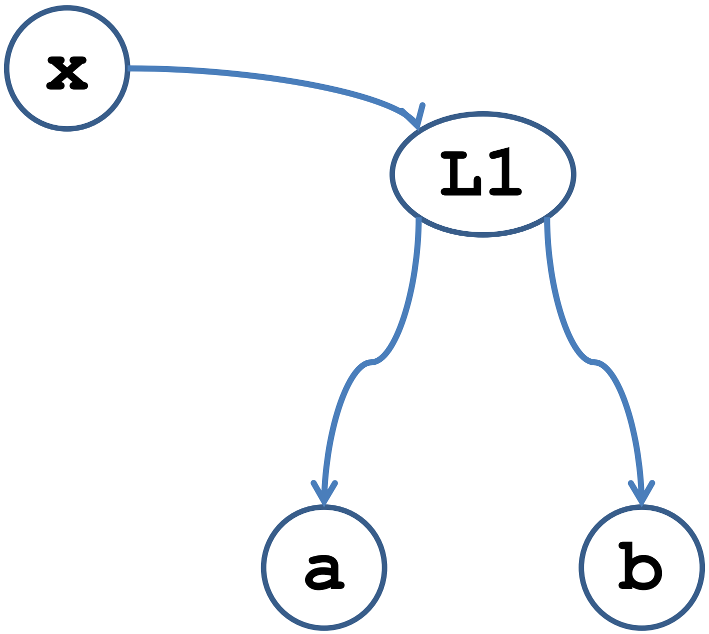
```
L1: x = new O;
L2: y = x;
L3: *y = &b;
L4: *y = &a;
```



Object fields

- Field-insensitive analysis

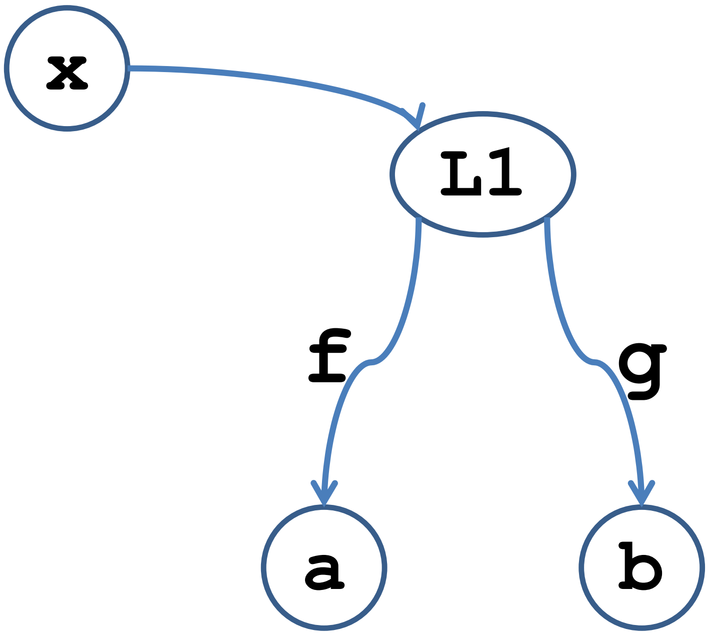
```
class Foo {
     A* f;
     B* g;
}
L1: x = new Foo()

x->f = &b;

x->g = &a;
```



Object fields

- Field-sensitive analysis

```
class Foo {
      A* f;
      B* g;
}
L1: x = new Foo()

x->f = &b;

x->g = &a;
```



Other Aspects

- Context-sensitivity
- Indirect (virtual) function calls and call-graph construction
- Pointer arithmetic
- Object-sensitivity

Shape Analysis

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;
    }
```

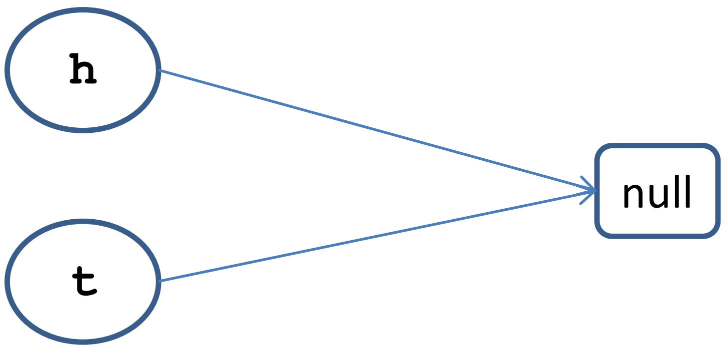
```
// Singly-linked list
// data type.
class SLL {
  int data;
  public SLL n; // next cell

  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;
    }
```

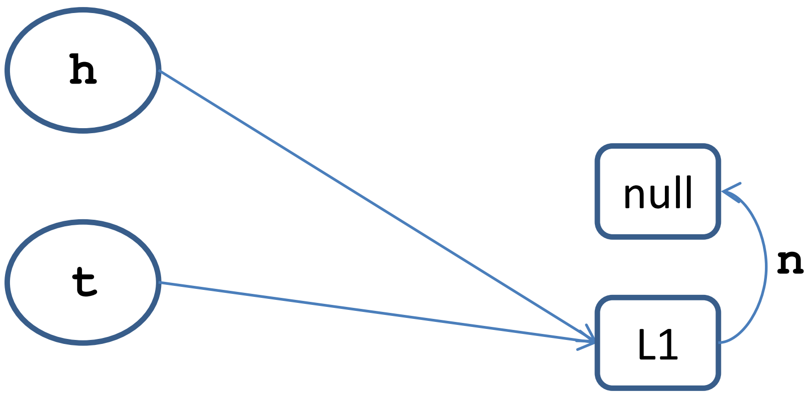
Flow&Field-sensitive Analysis

```
     // 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;
     }
```



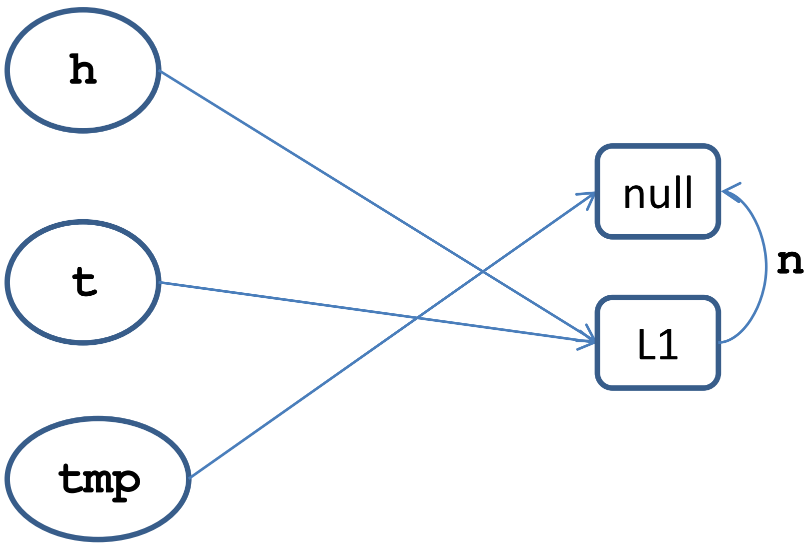
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



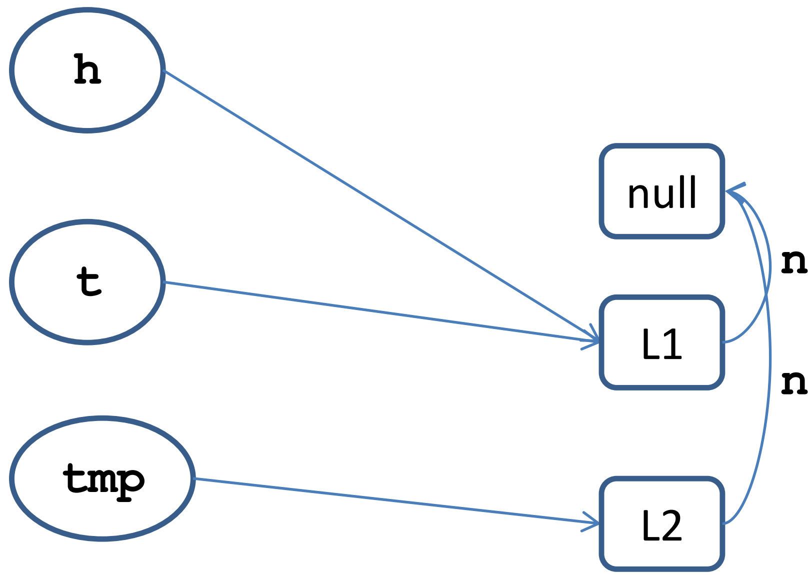
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



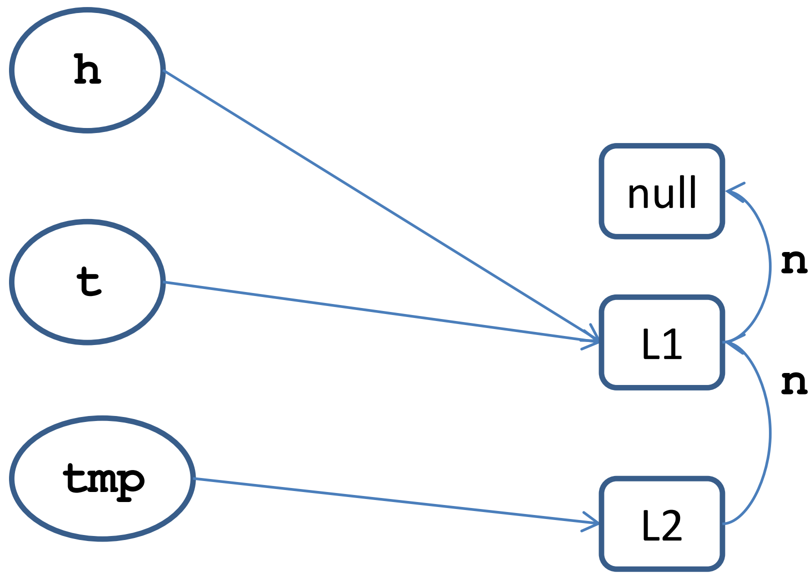
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



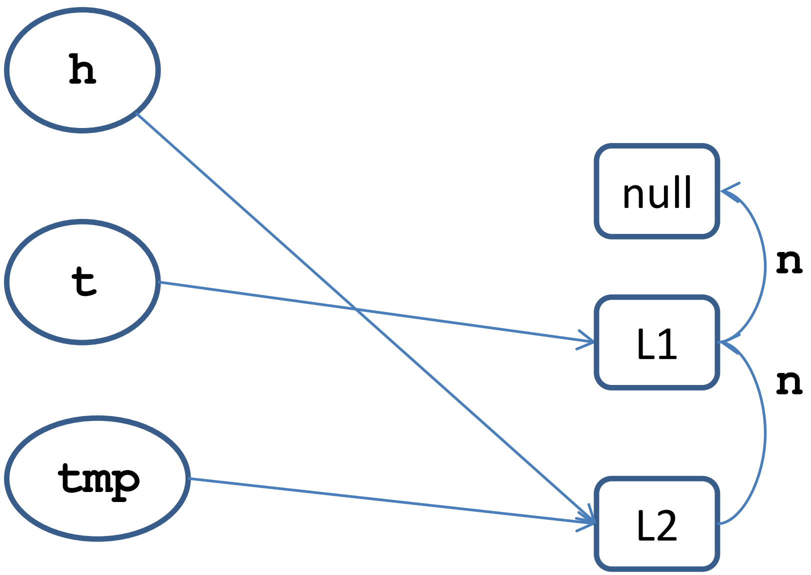
Flow&Field-sensitive Analysis

```
     // 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;
     }
```



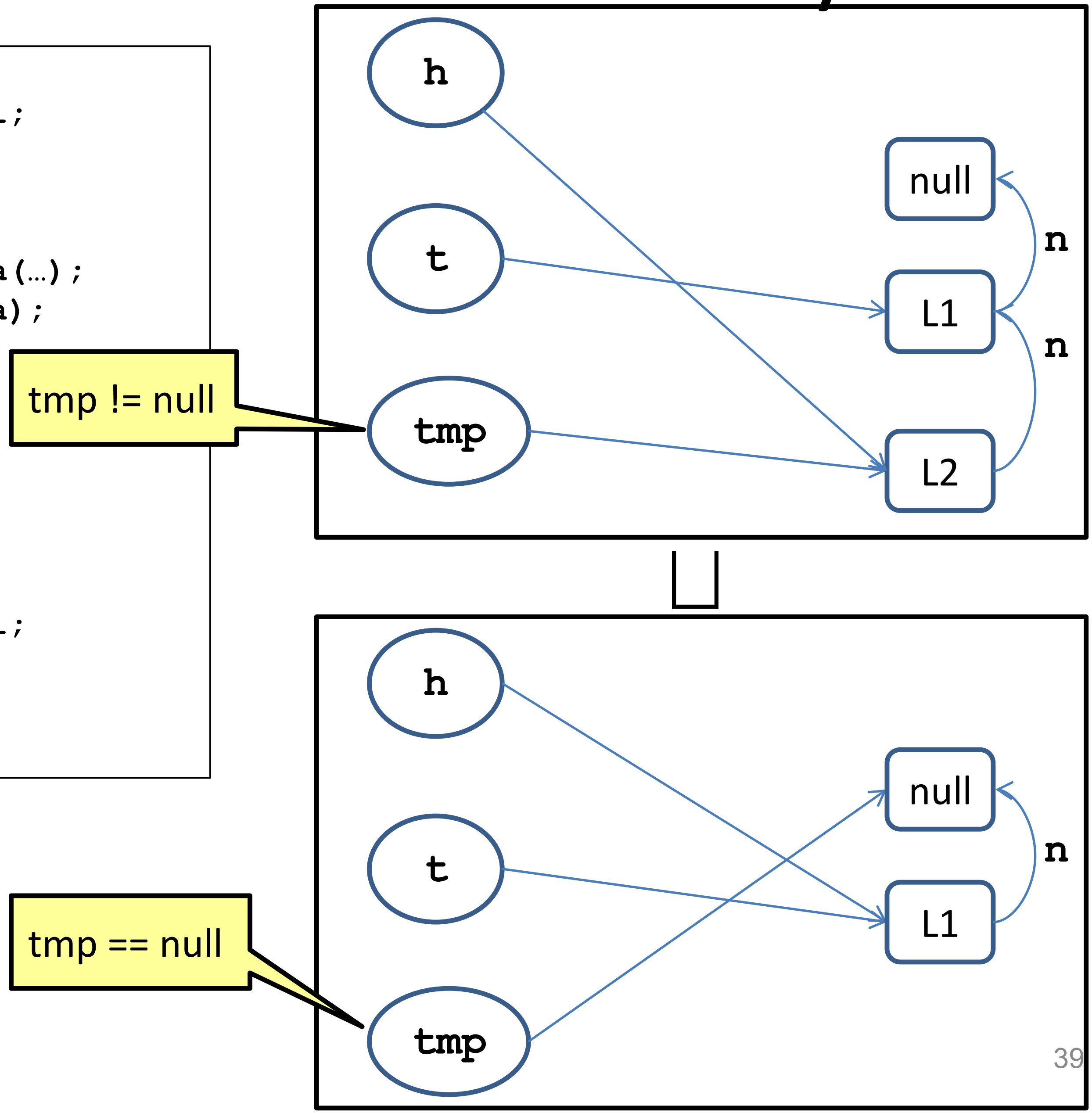
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



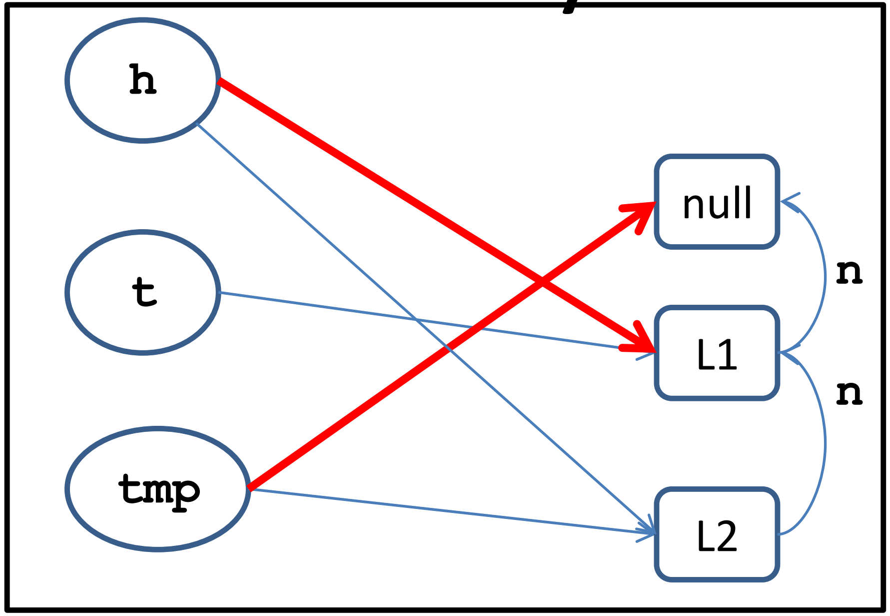
Flow&Field-sensitive Analysis

```
    // 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;
    }
```

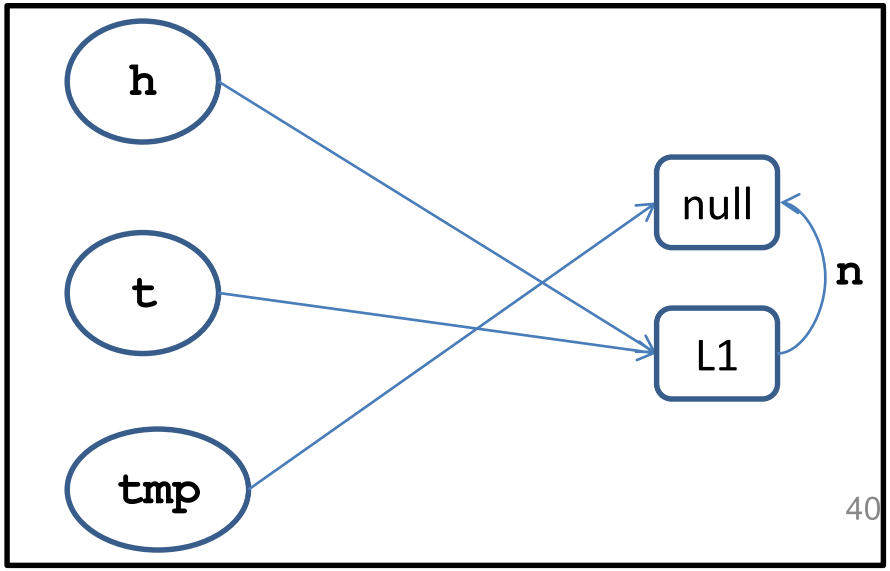


Flow&Field-sensitive Analysis

```
    // 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;
    }
```

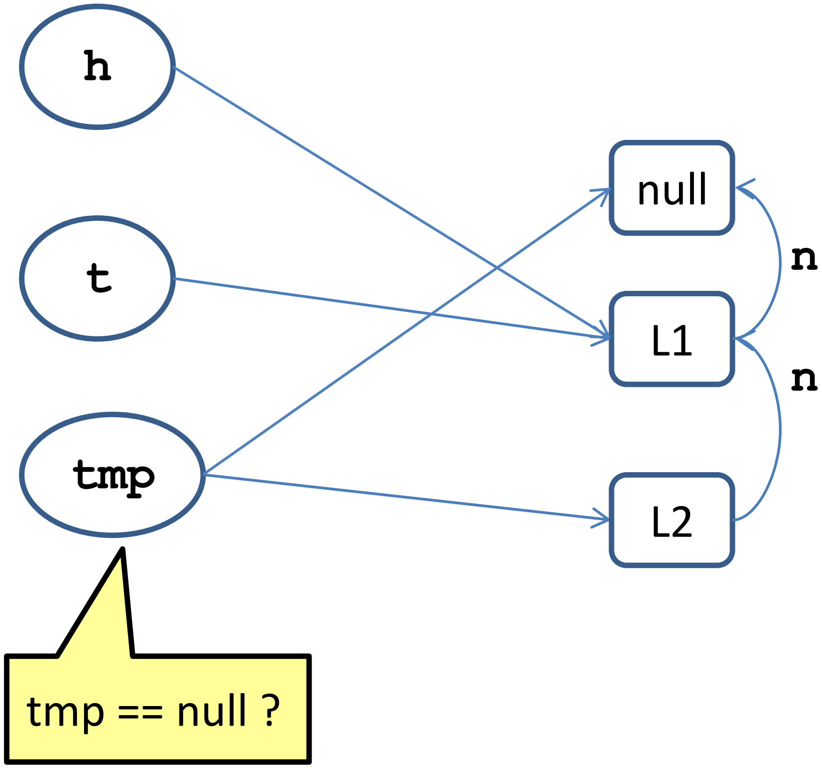


⊔



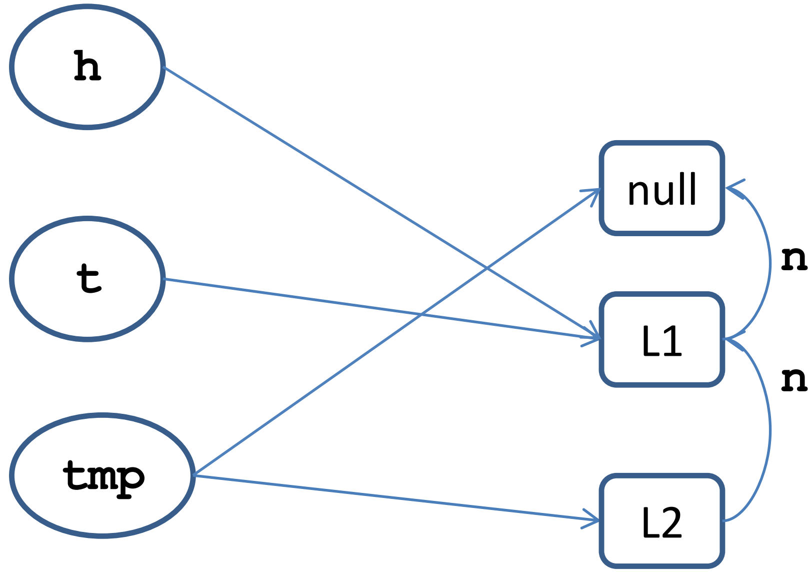
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



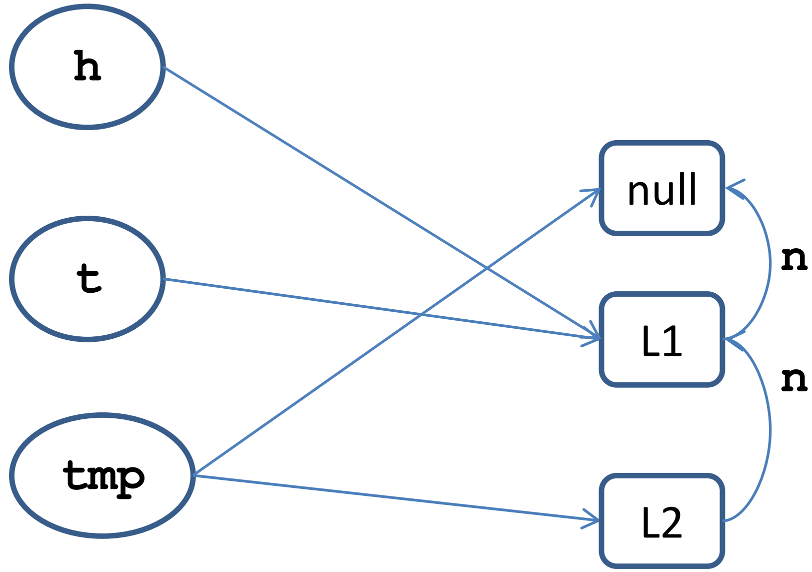
Flow&Field-sensitive Analysis

```
    // 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

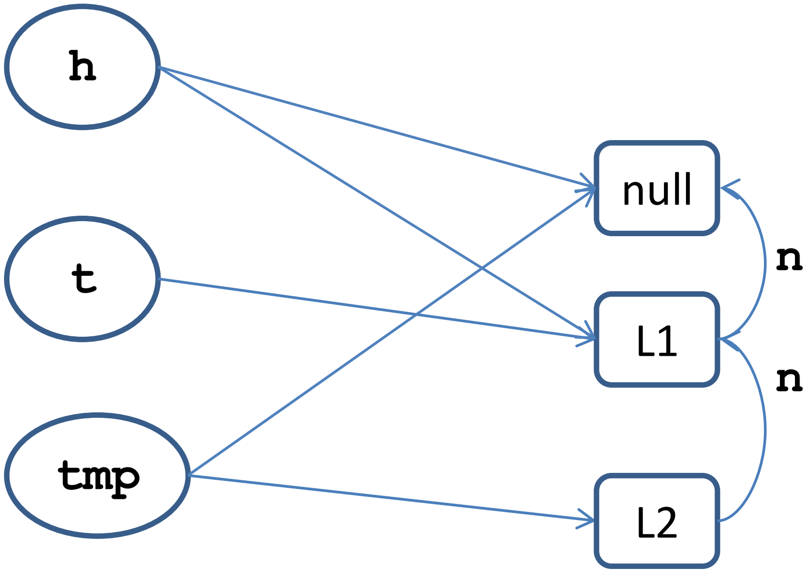
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

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



Possible null dereference!

Flow&Field-sensitive Analysis

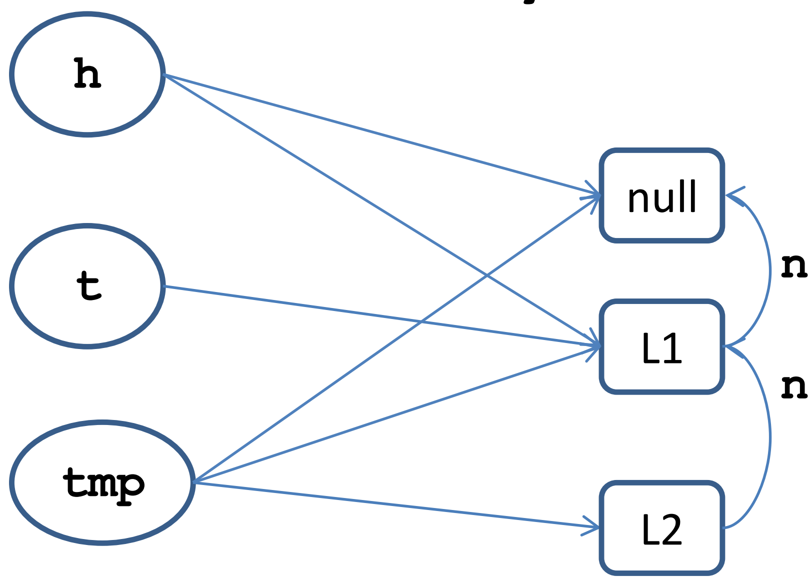
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

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



Fixed-point for first loop

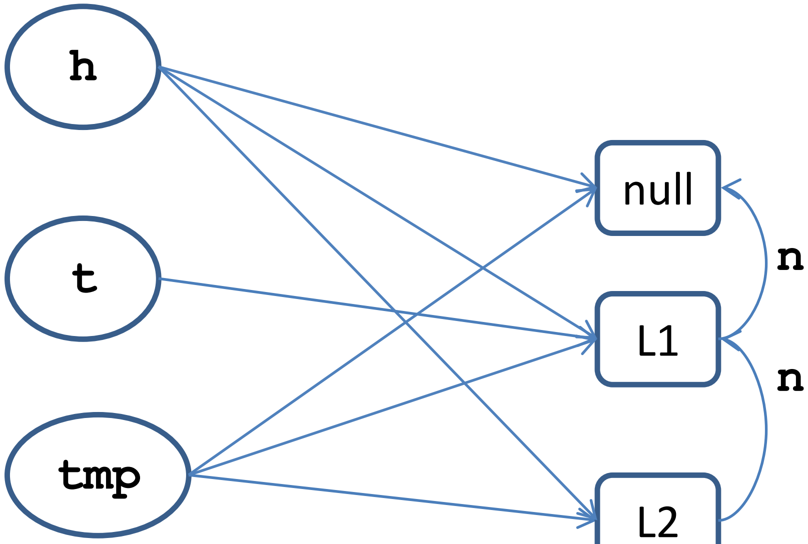
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



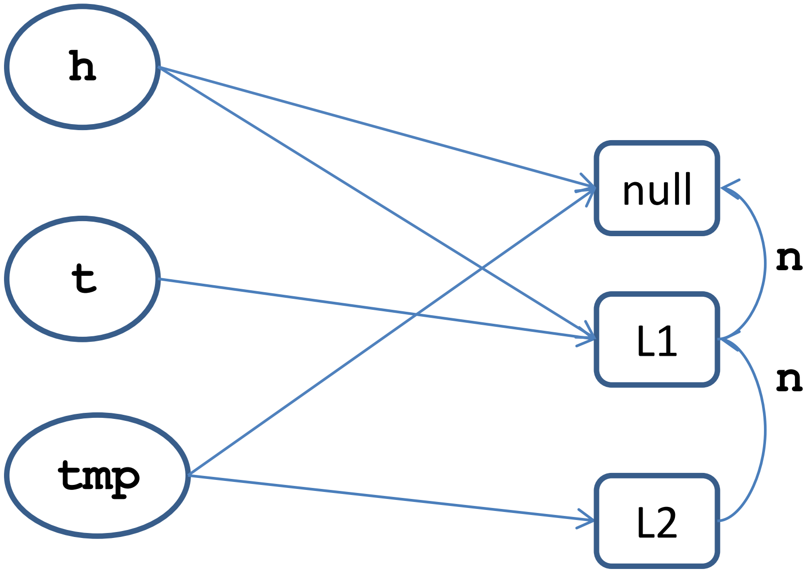
Flow&Field-sensitive Analysis

```
    // 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

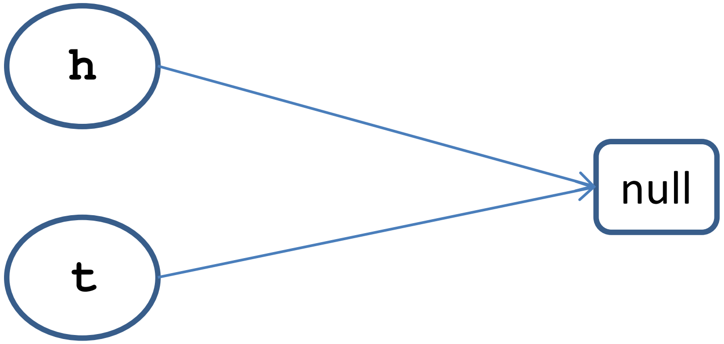
- Pointer analysis abstract all objects allocated at same program location into one summary object. However, objects allocated at same memory location may behave very differently
  - E.g., object is first/last one in the list
  - Add extra instrumentation predicates to distinguish between objects with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

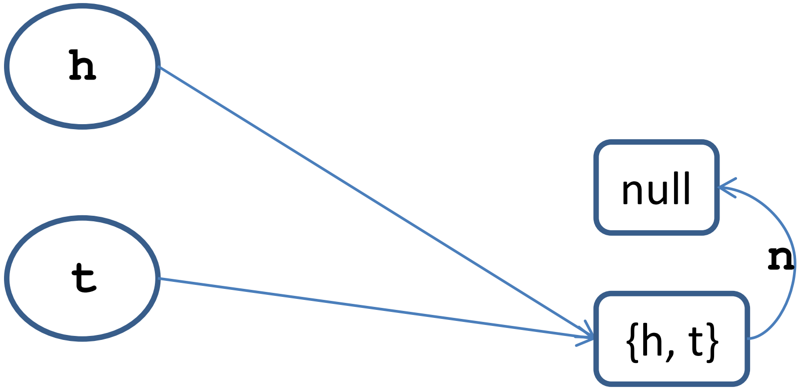
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



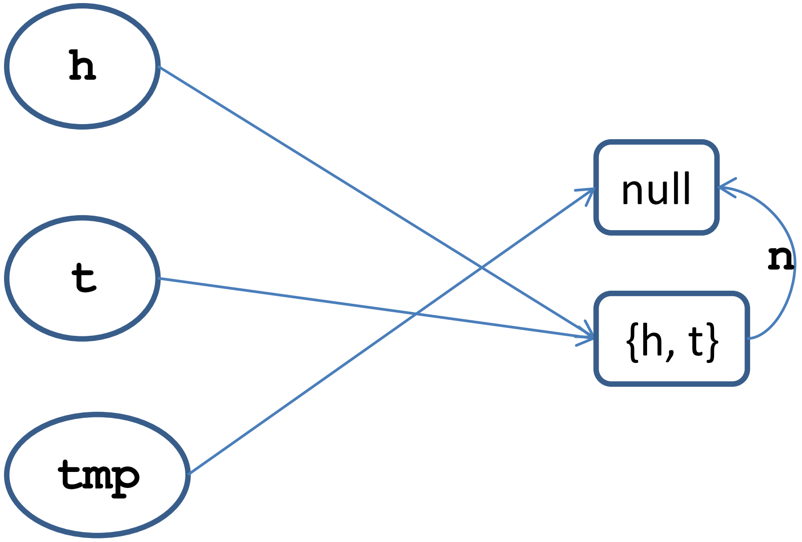
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



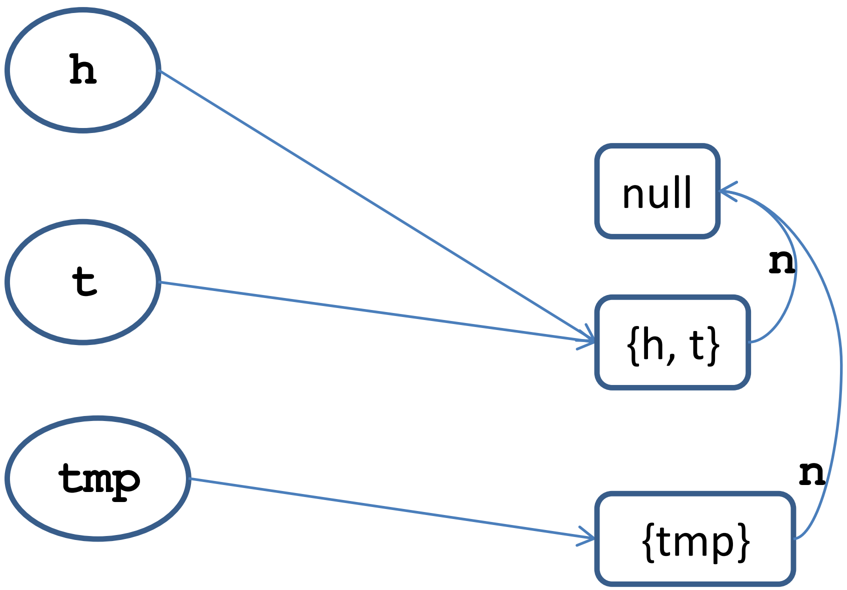
Flow&Field-sensitive Analysis

```
    // 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

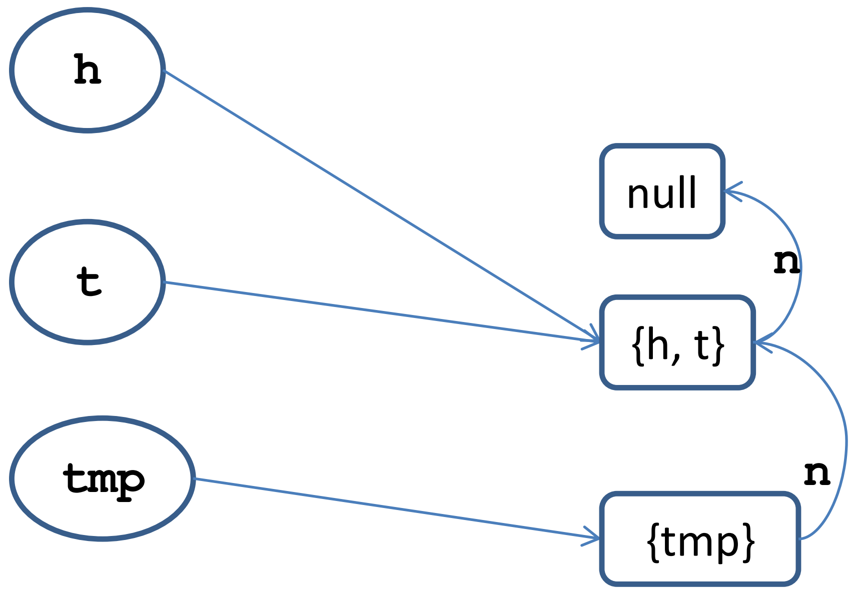
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

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



No null dereference

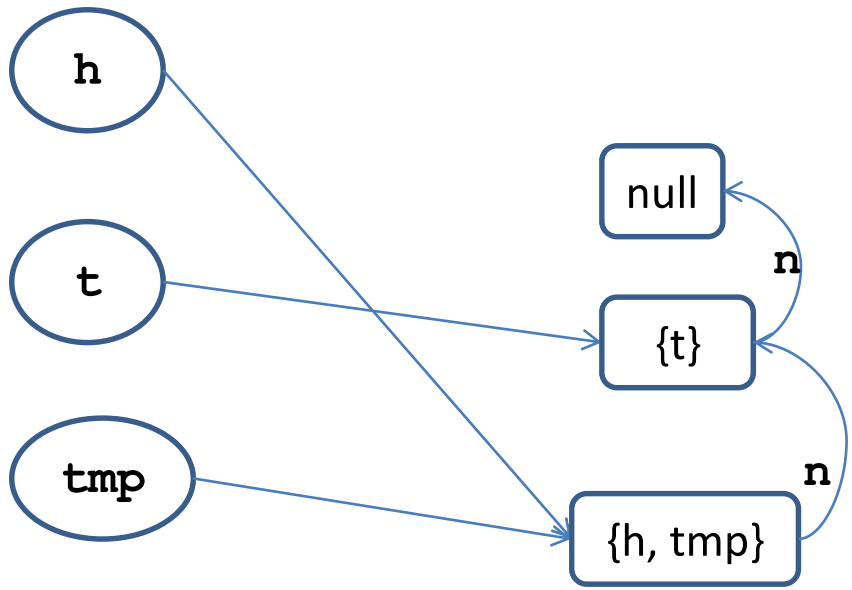
Flow&Field-sensitive Analysis

```
     // 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;
     }
```



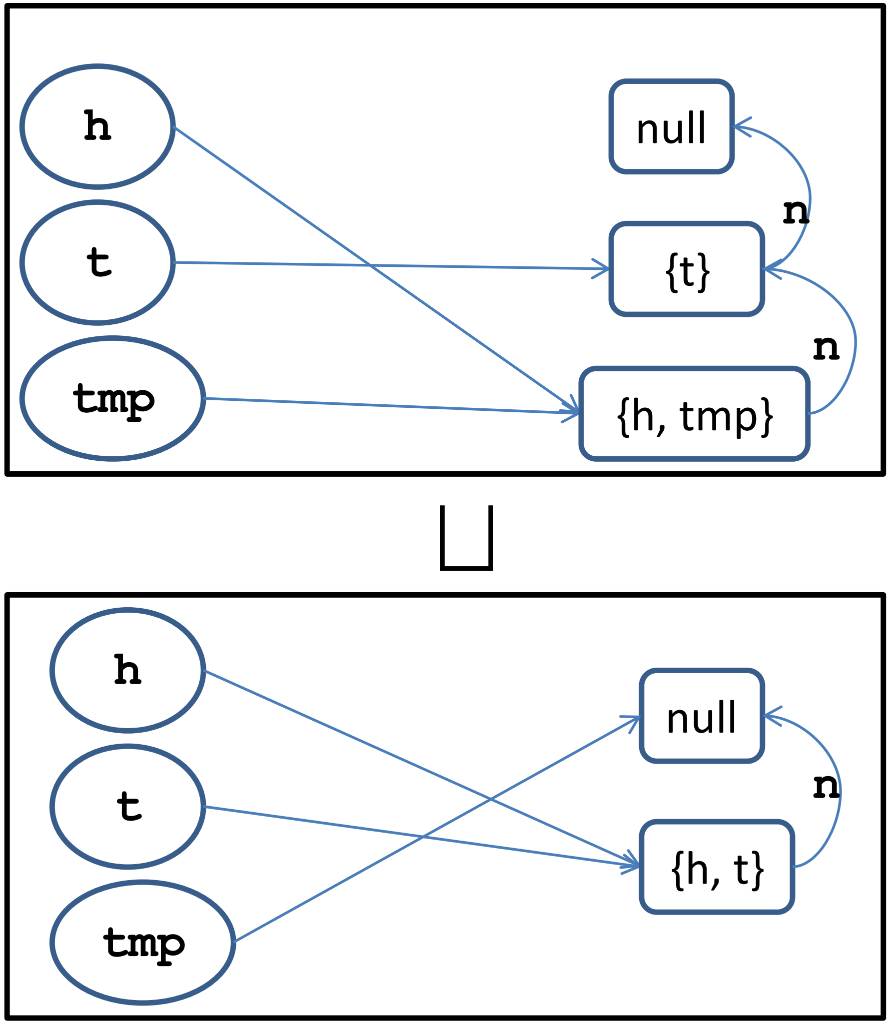
Flow&Field-sensitive Analysis

```
     // 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;
     }
```



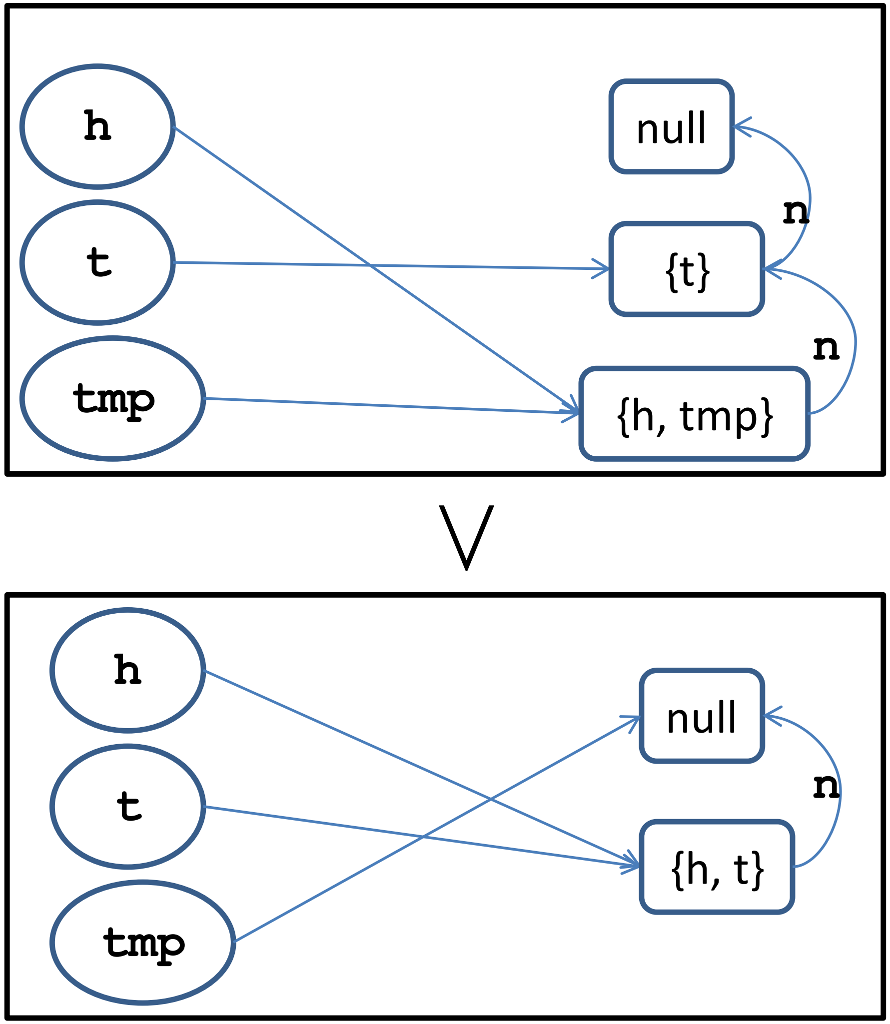
Flow&Field-sensitive Analysis

```
     // 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

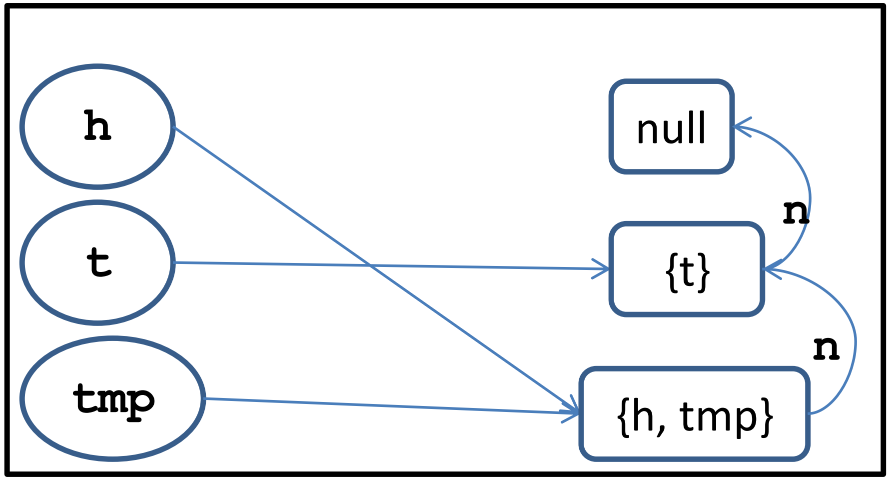
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

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



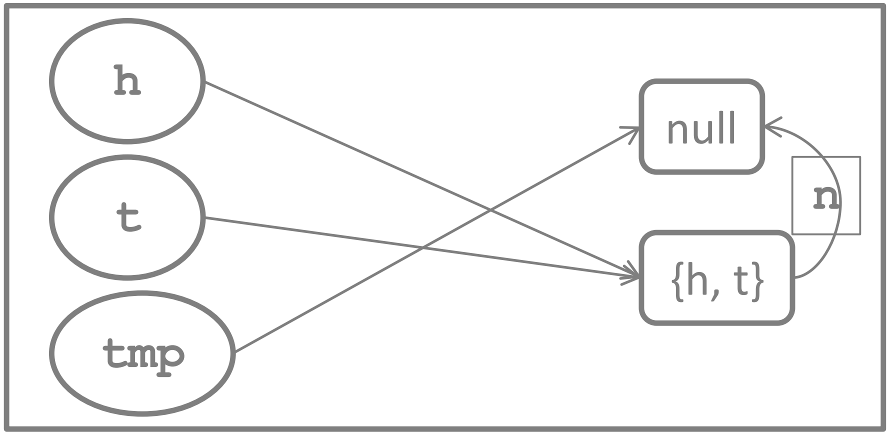
Do we need to analyze this shape graph again?

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;
    }
```

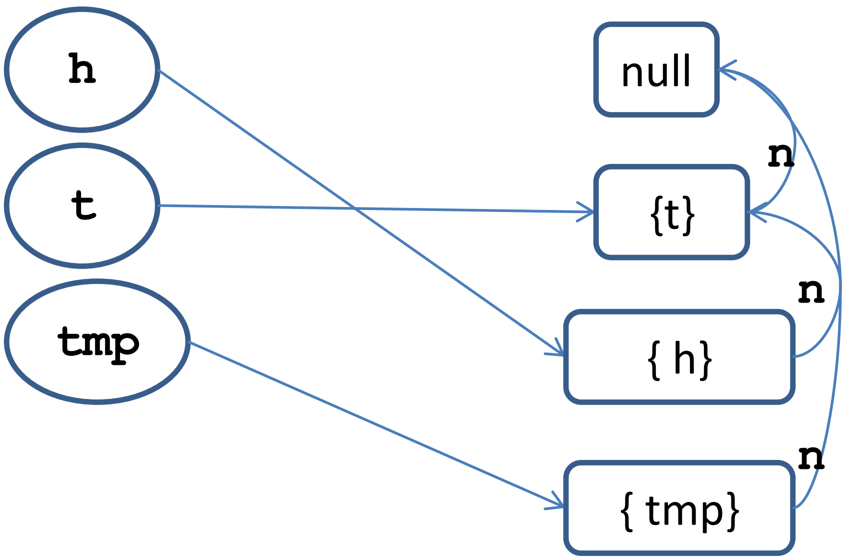


⊑



Flow&Field-sensitive Analysis

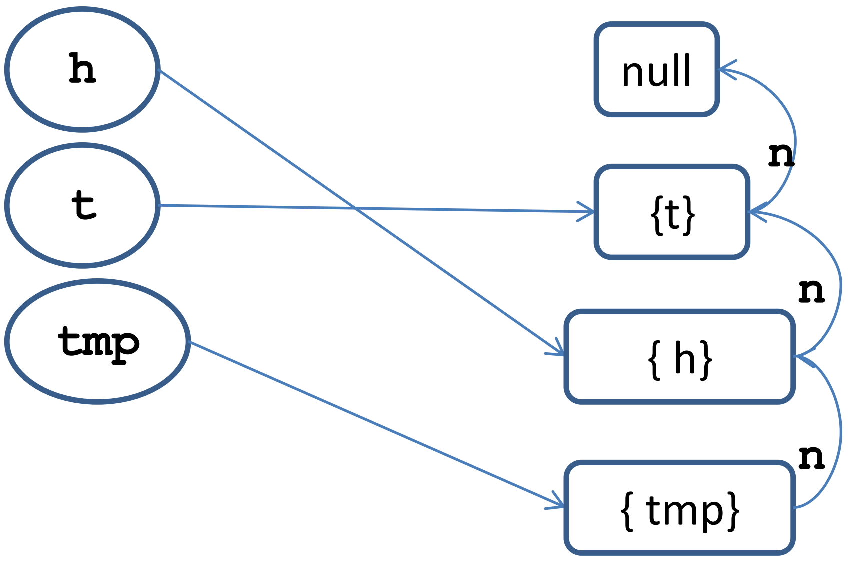
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

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



No null dereference

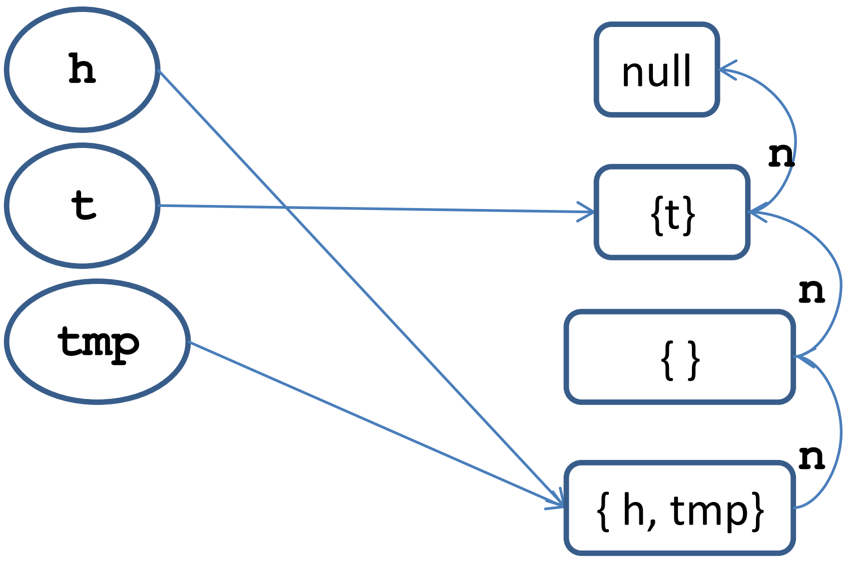
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



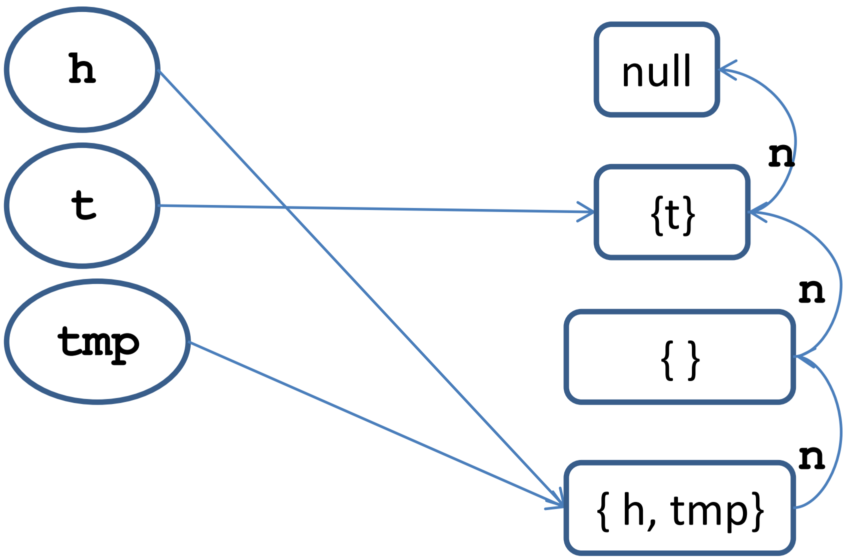
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



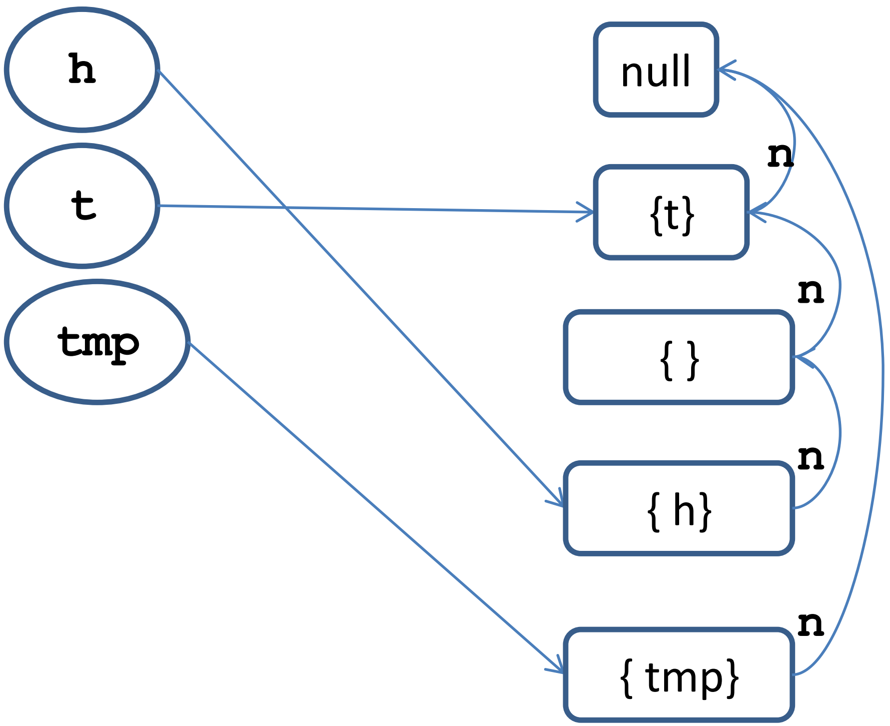
Flow&Field-sensitive Analysis

```
     // 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

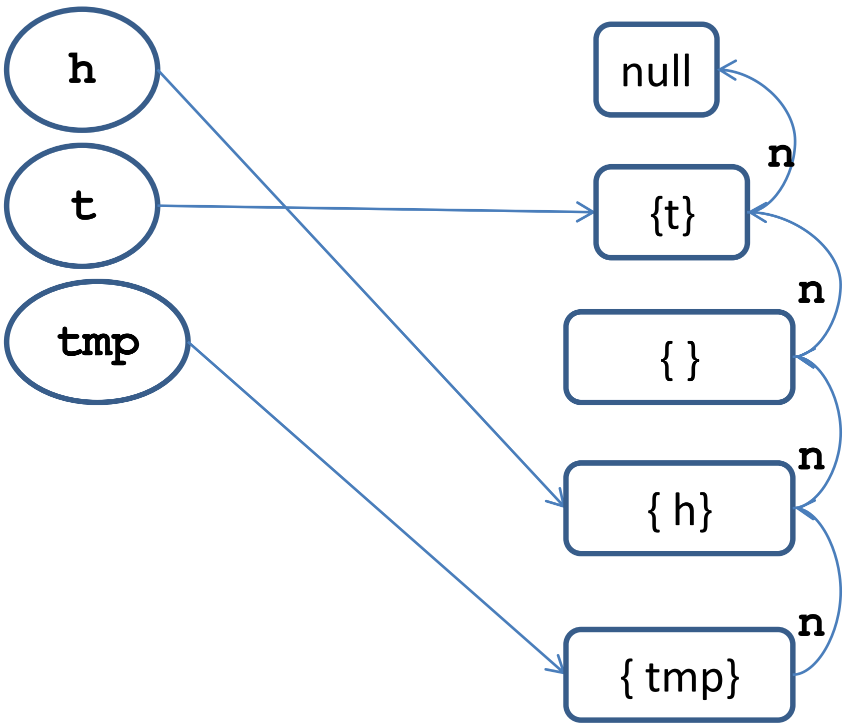
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

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



No null dereference

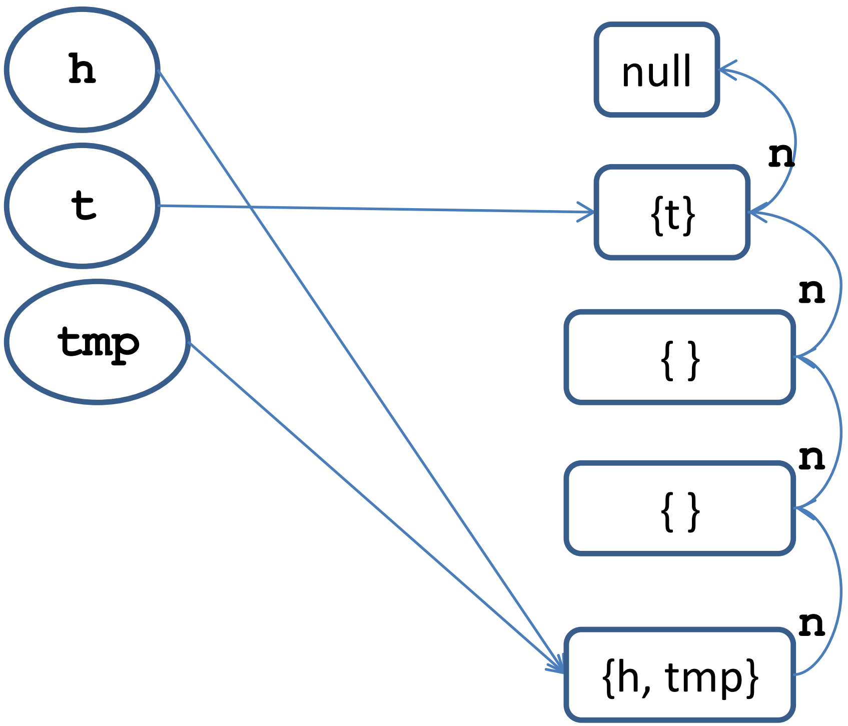
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



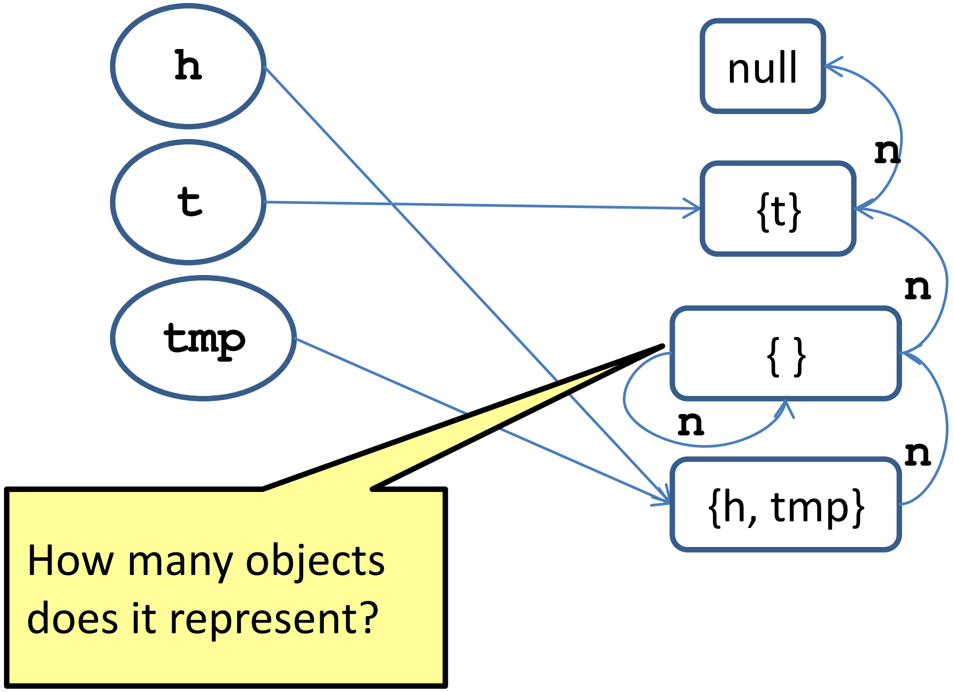
Flow&Field-sensitive Analysis

```
    // 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

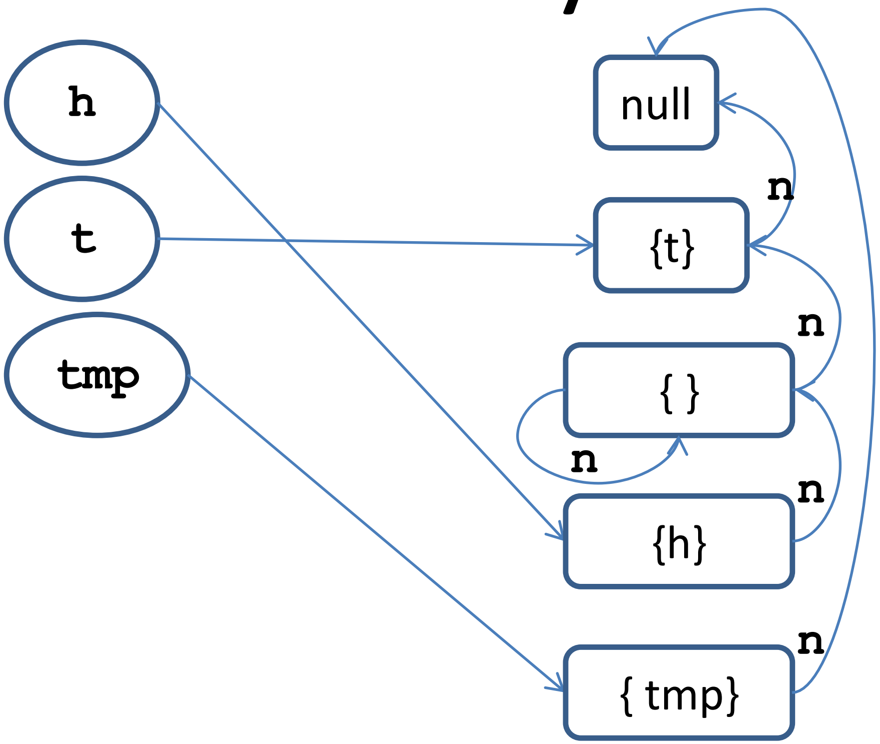
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

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



Summarization

Flow&Field-sensitive Analysis

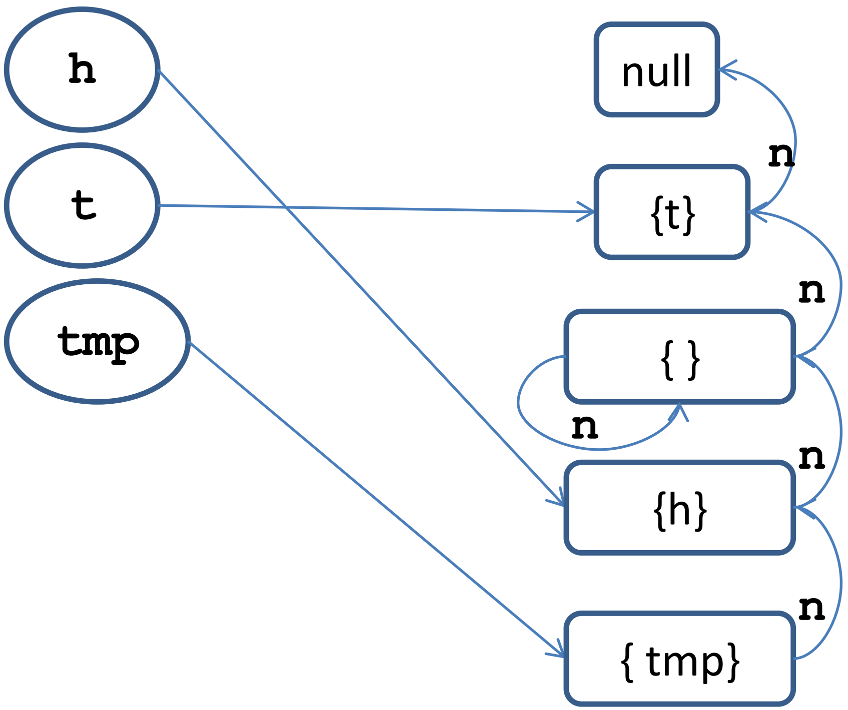
```
     // Build a list
     SLL h=null, t = null;
L1:  h=t= new SLL(-1);
     SLL tmp = null;
     while (…) {
       int data = getData(…);
L2:    tmp = new SLL(data);
       tmp.n = h;
       h = tmp;
     }

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



No null dereference

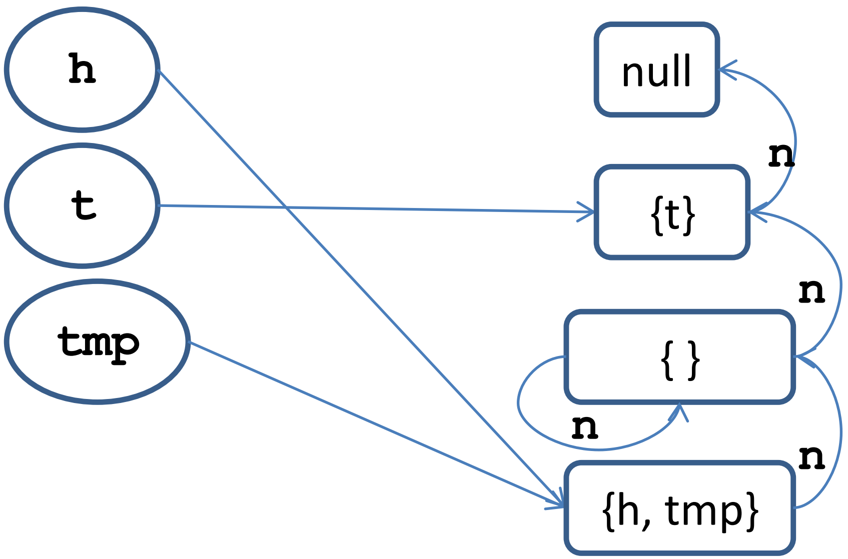
Flow&Field-sensitive Analysis

```
    // 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

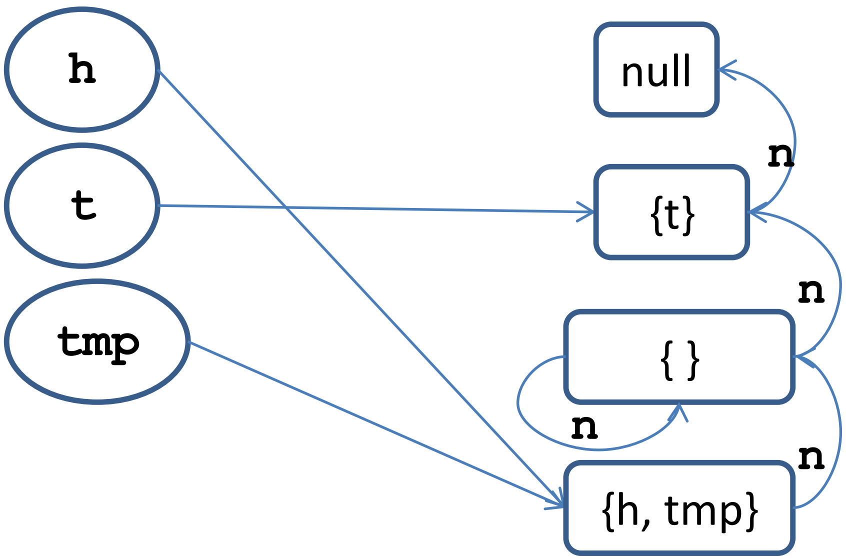
```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

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



Fixed-point for first loop

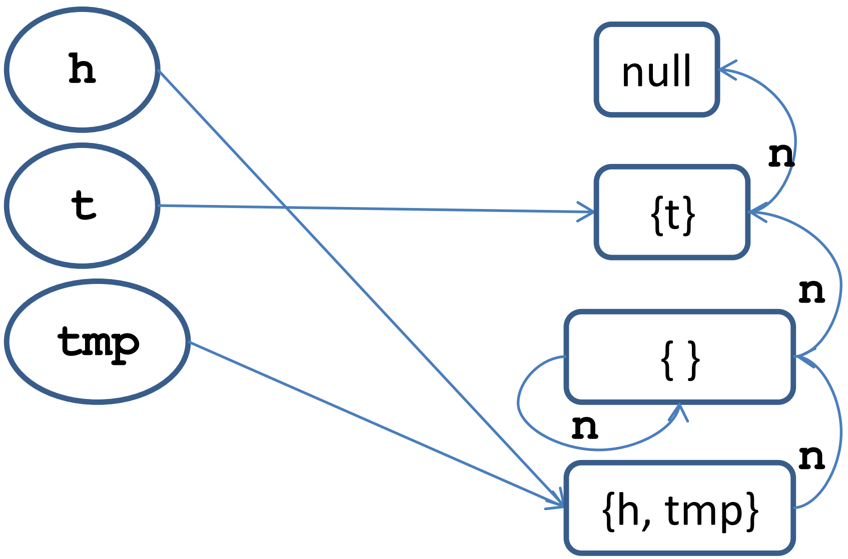
Flow&Field-sensitive Analysis

```
    // 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;
    }
```



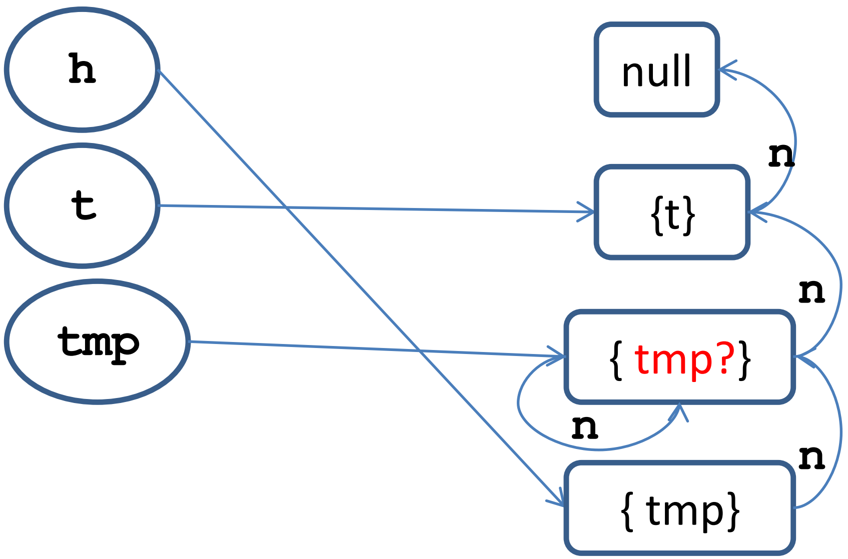
Flow&Field-sensitive Analysis

```
     // 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;
     }
```



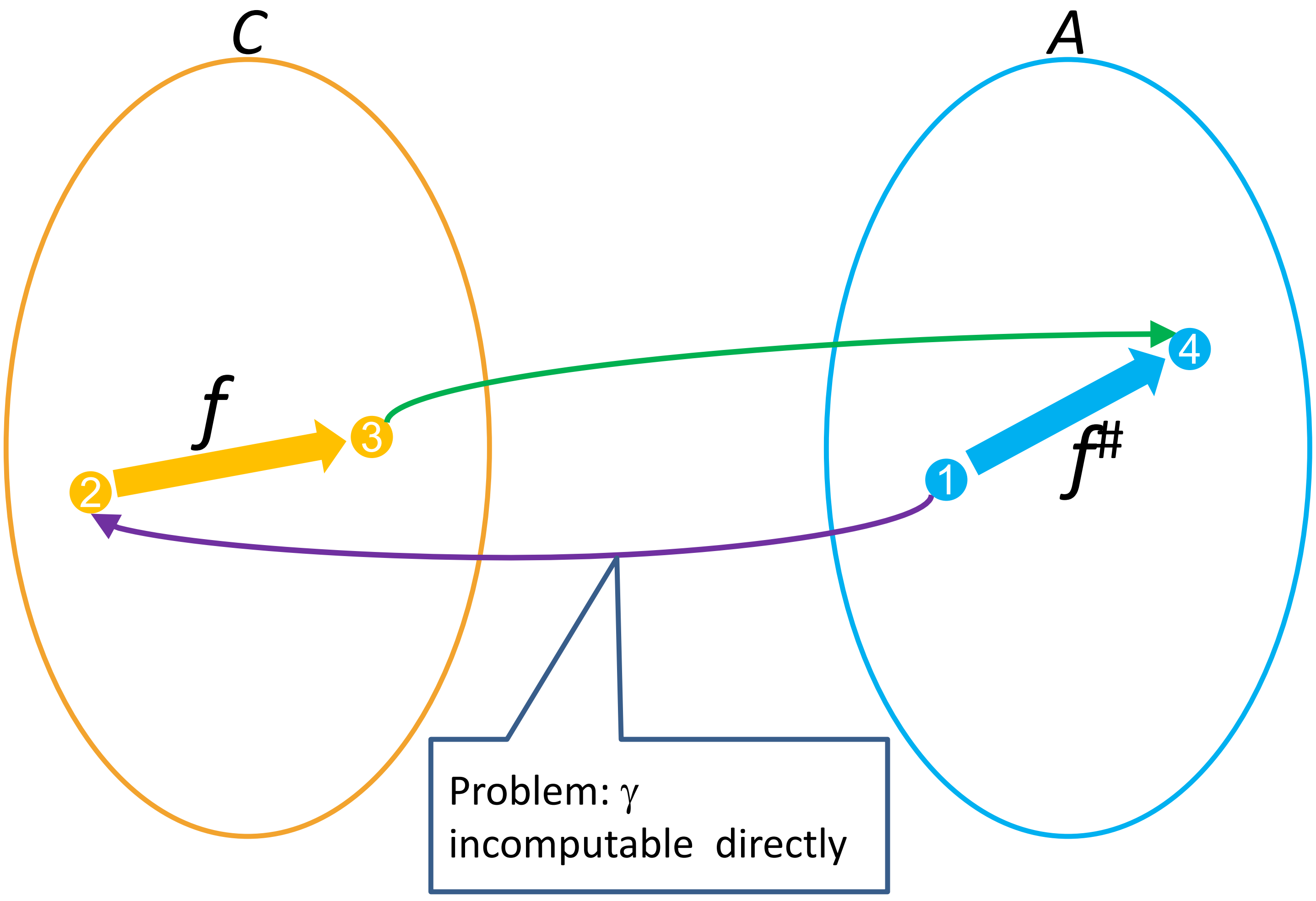
Flow&Field-sensitive Analysis

```
     // 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;
     }
```

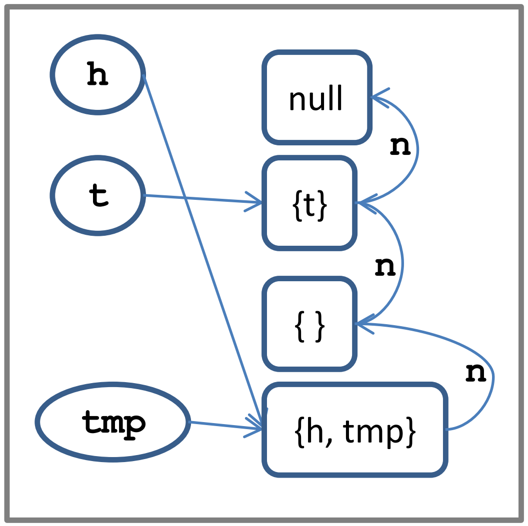


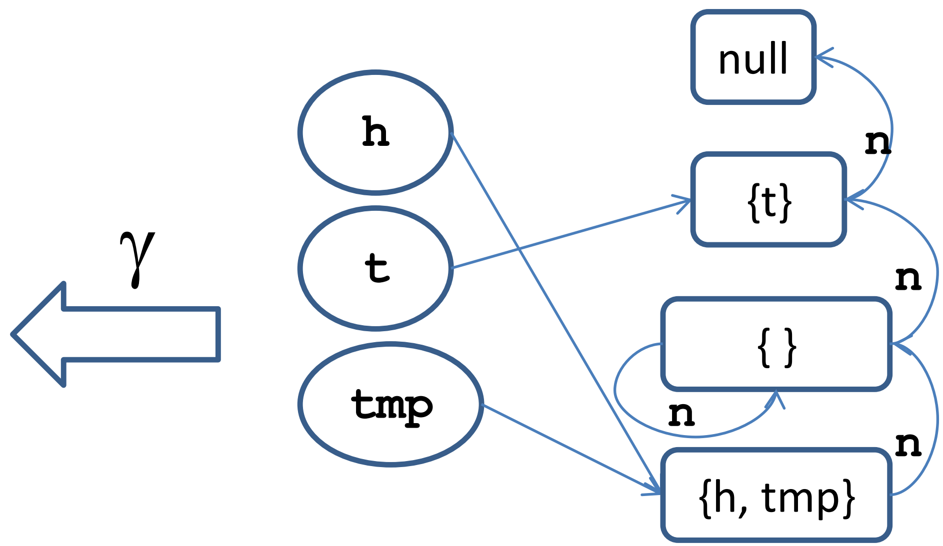
Best (induced) transformer

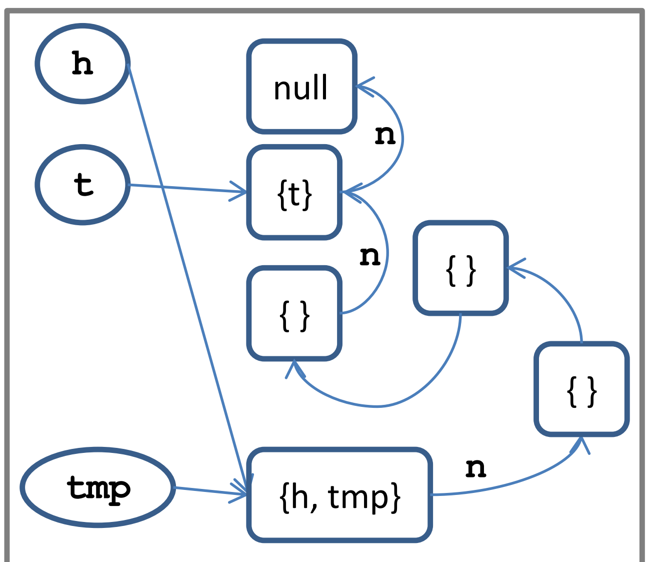
$$f^{\#}(a) = \alpha(f(\gamma(a)))$$



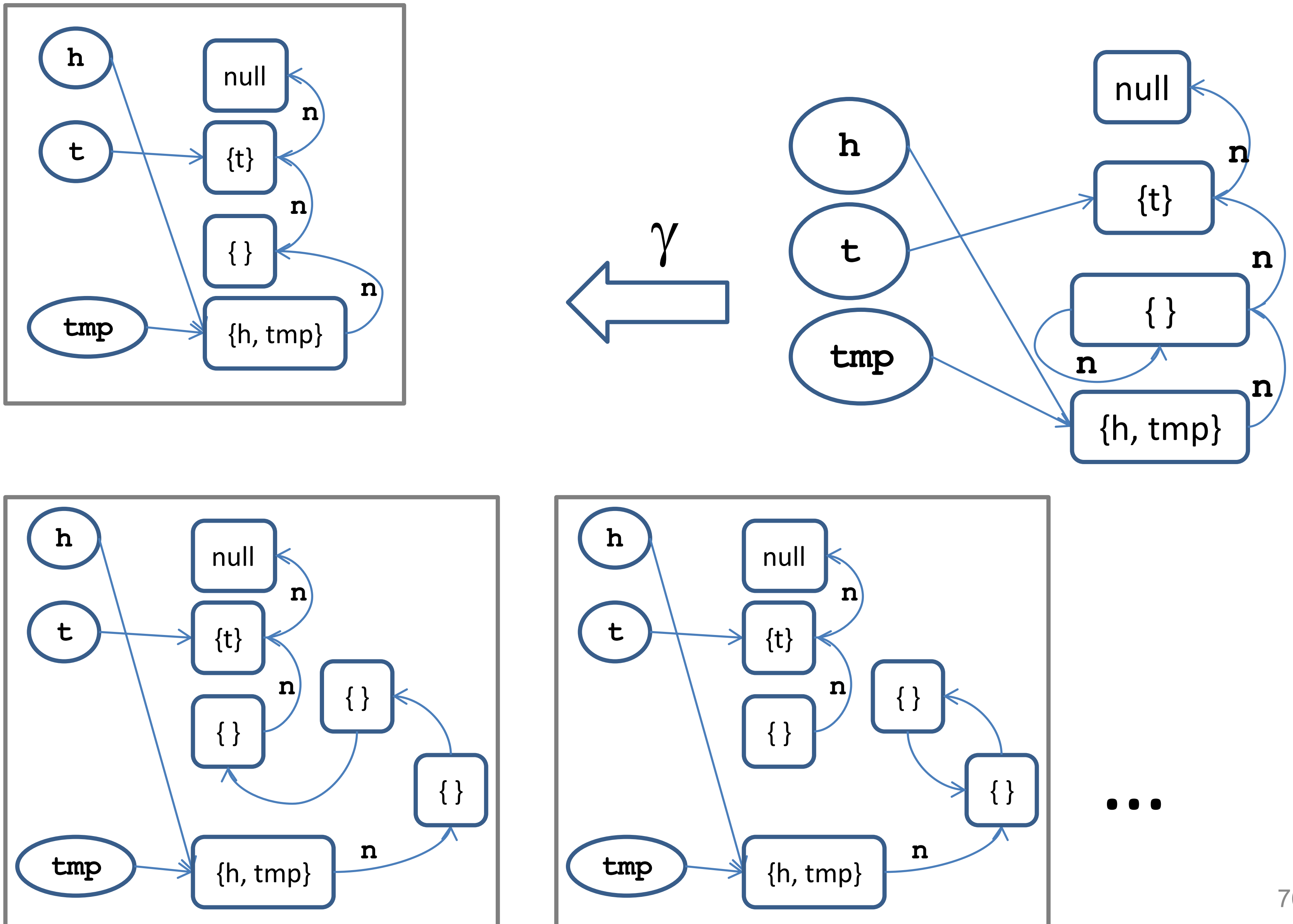
Best transformer for tmp=tmp.n



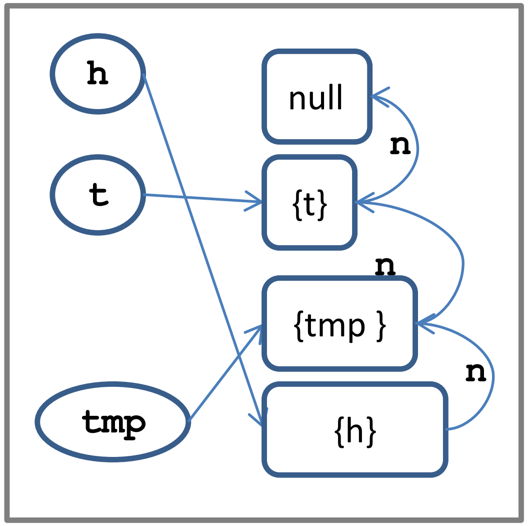


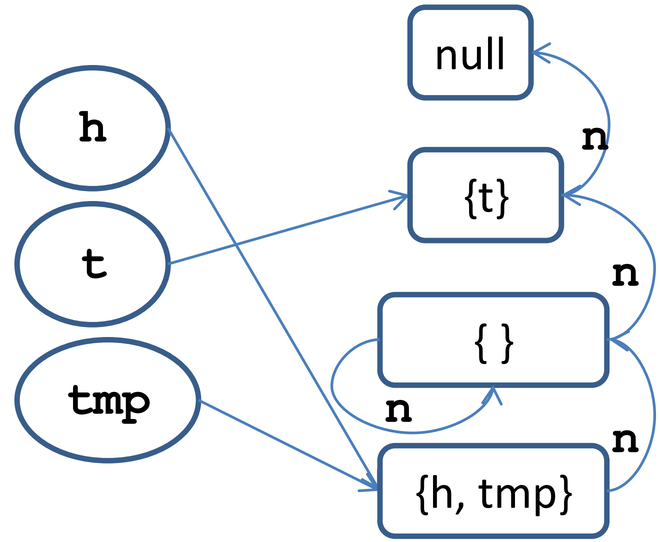


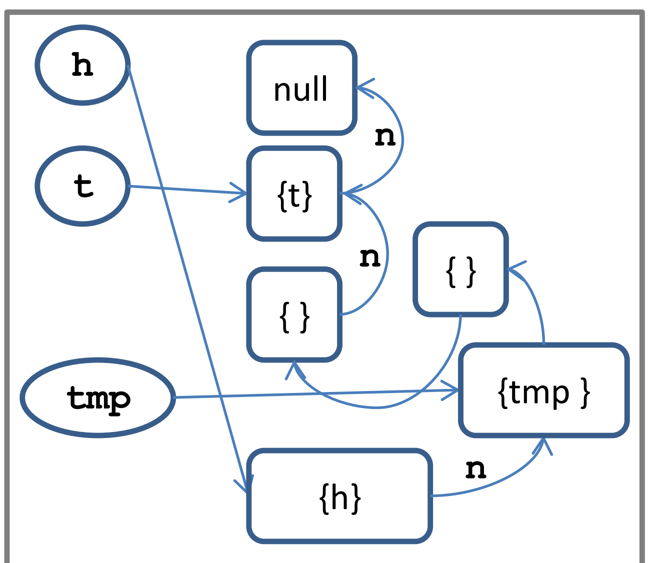
Best transformer for tmp=tmp.n: γ

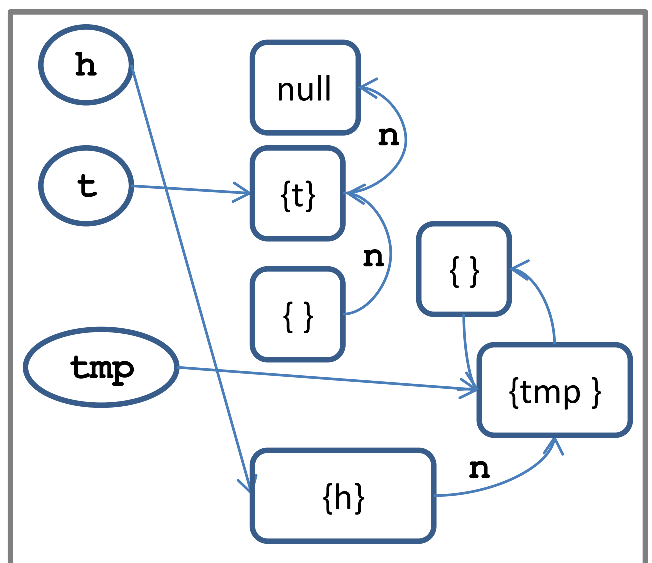


Best transformer for ⟦tmp=tmp.n⟧



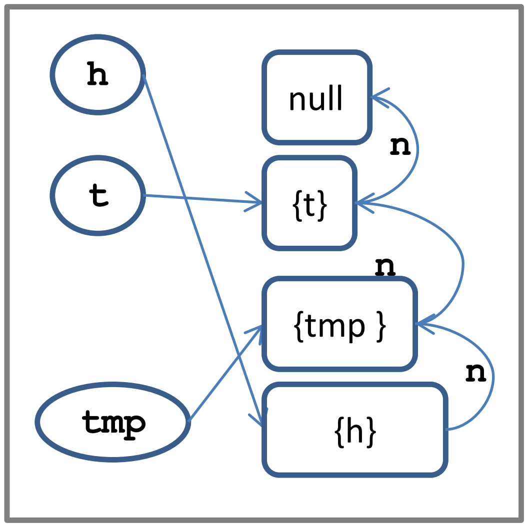


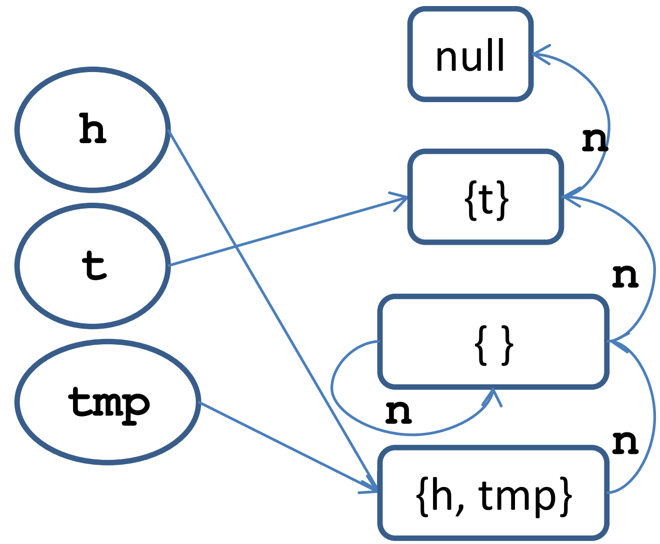


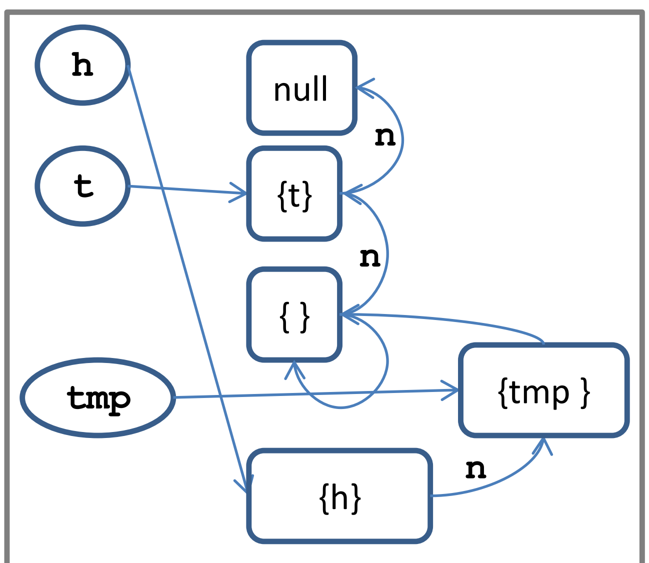


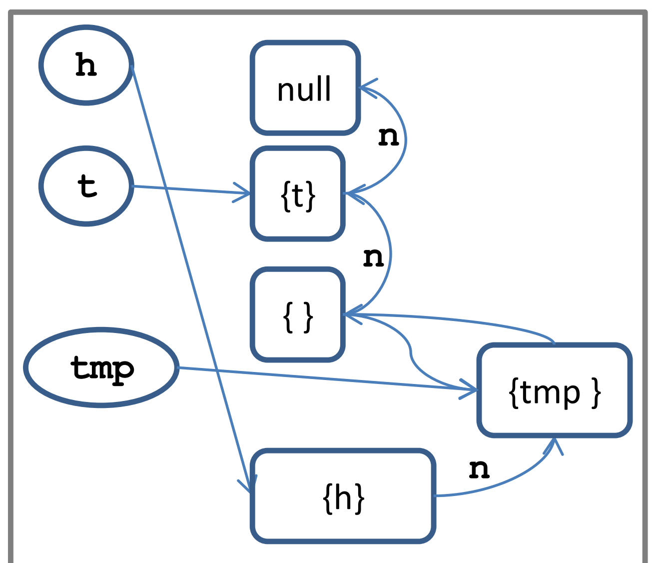
...

Best transformer for tmp=tmp.n: α







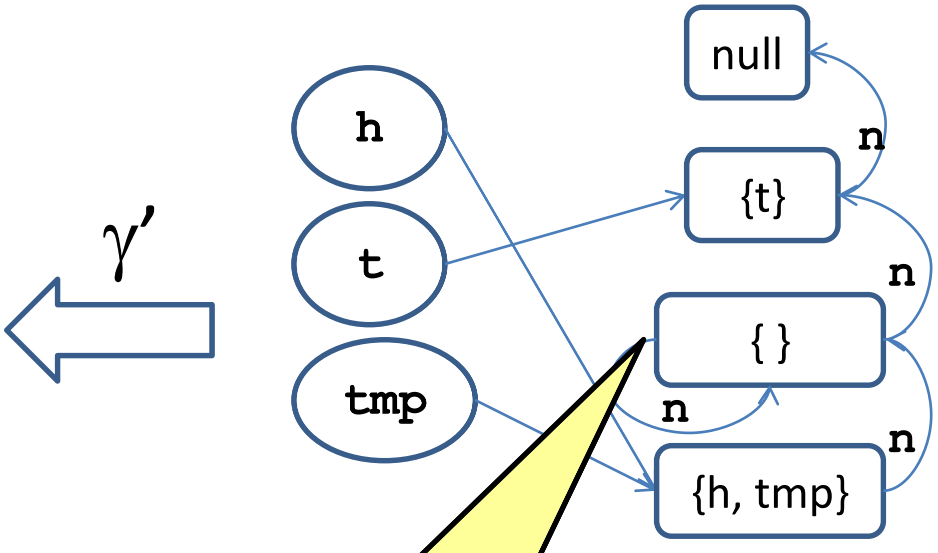


...

Handling updates on summary nodes

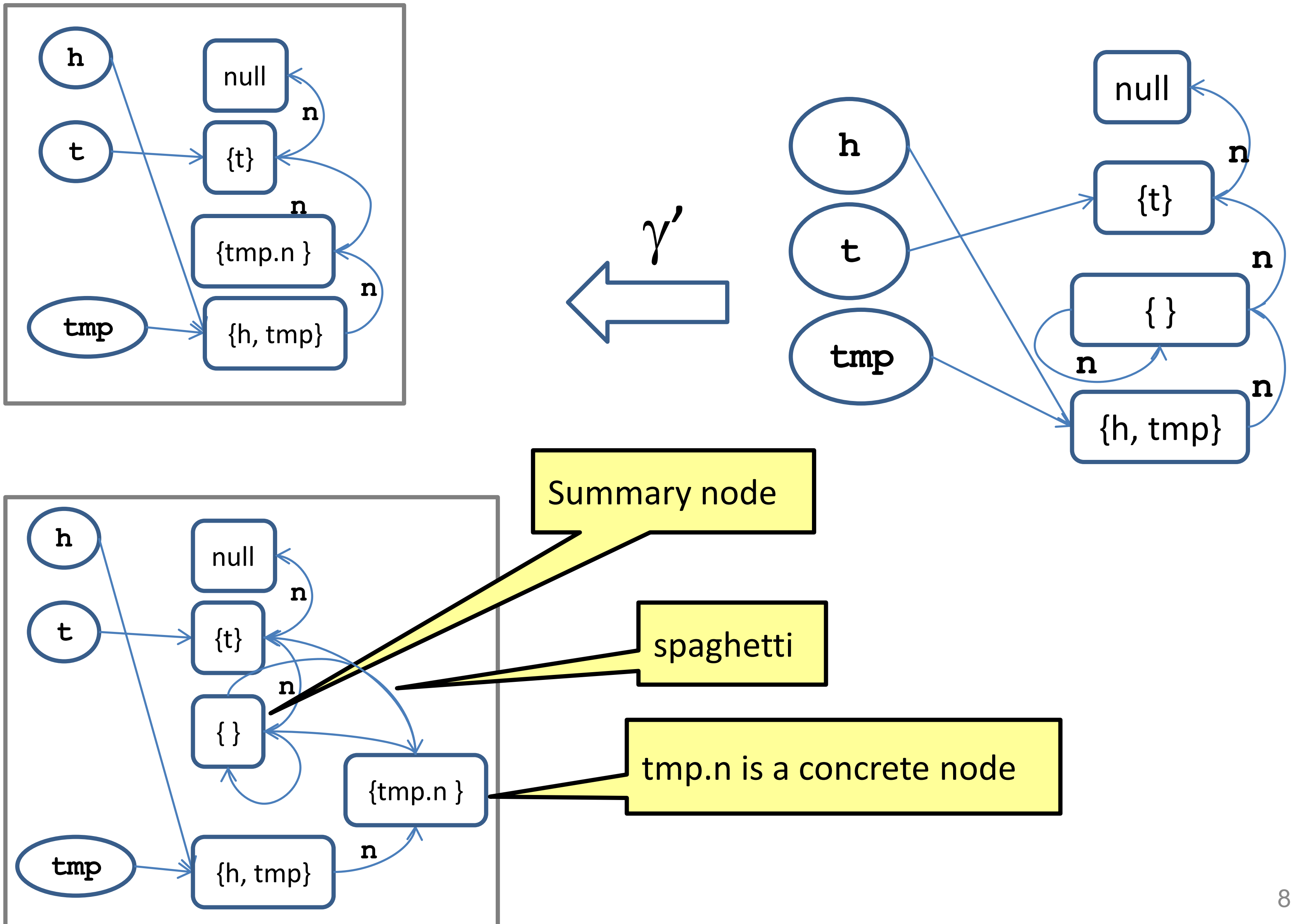
- Transformers accessing only concrete nodes are easy
- Transformers accessing summary nodes are complicated
- Can’t concretize summary nodes – represents potentially unbounded number of concrete nodes
- We need to split into cases by “materializing” concrete nodes from summary node
  - Introduce a new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

Transformer for tmp=tmp.n: γ'

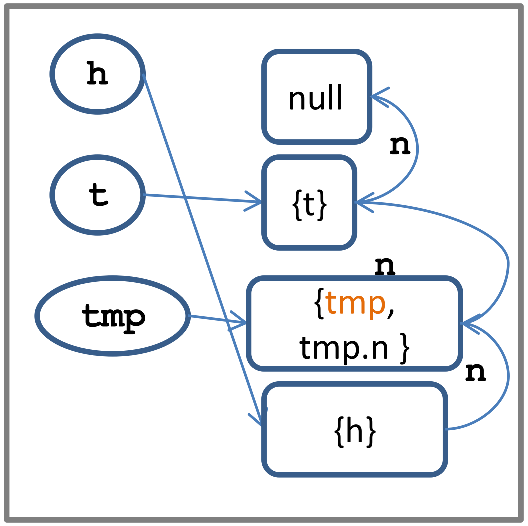


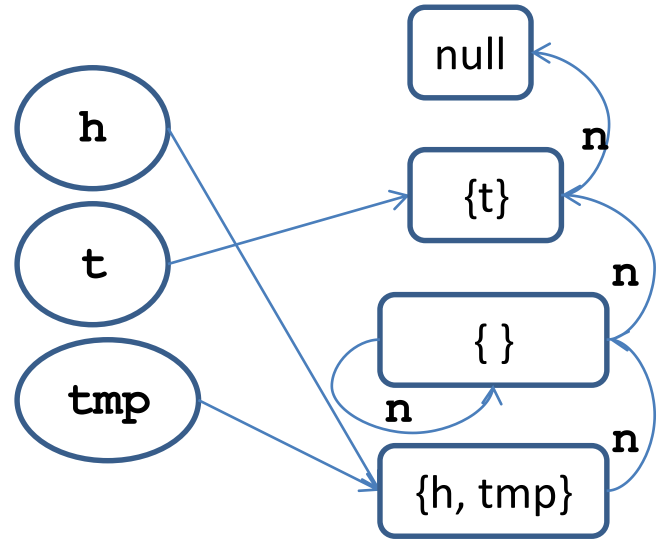
Case 1: Exactly 1 object.
Case 2: >1 objects.

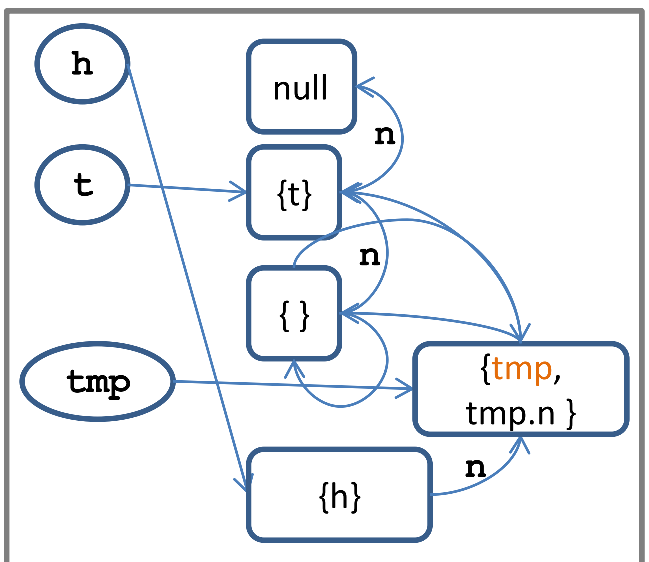
Transformer for tmp=tmp.n: γ'



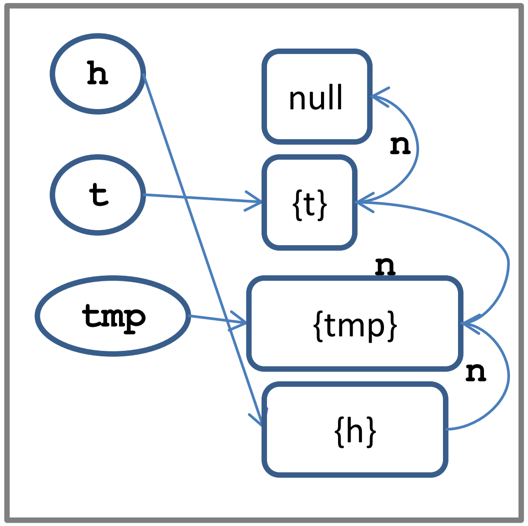
Transformer ⟦tmp=tmp.n⟧

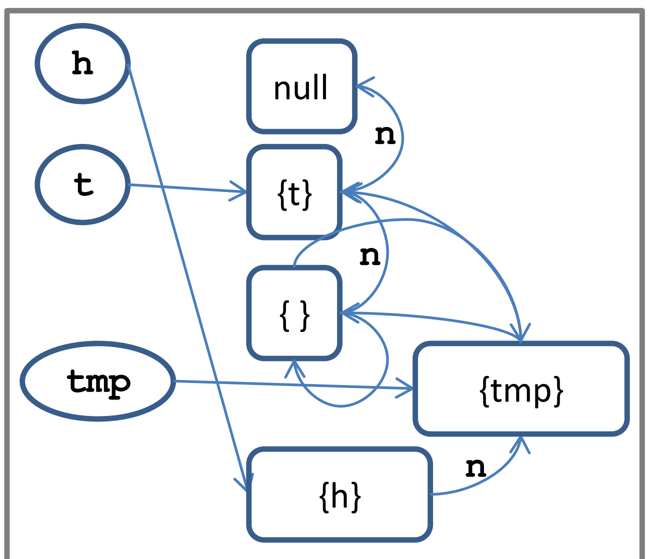






Transformer for tmp=tmp.n: α

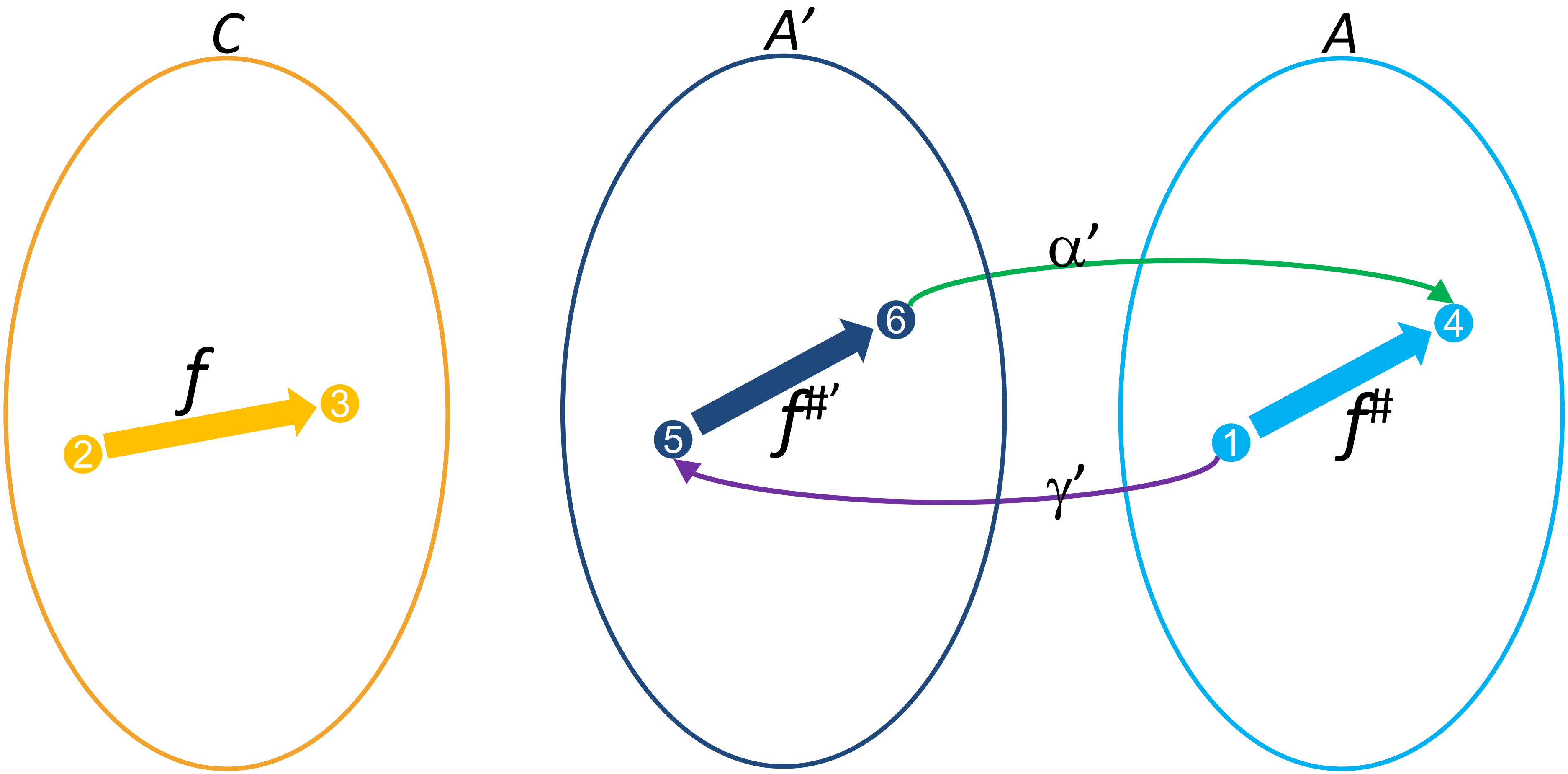





```
    // Build a list
    SLL h=null, t = null;
L1: h=t= new SLL(-1);
    SLL tmp = null;
    while (…) {
      int data = getData(…);
L2:   tmp = new SLL(data);
      tmp.n = h;
      h = tmp;
    }

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

Transformer via partial-concretization

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



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

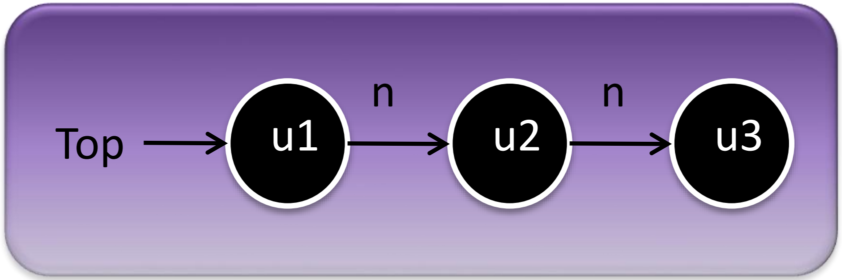
Shape Analysis via 3-valued Logic

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

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

Concrete State

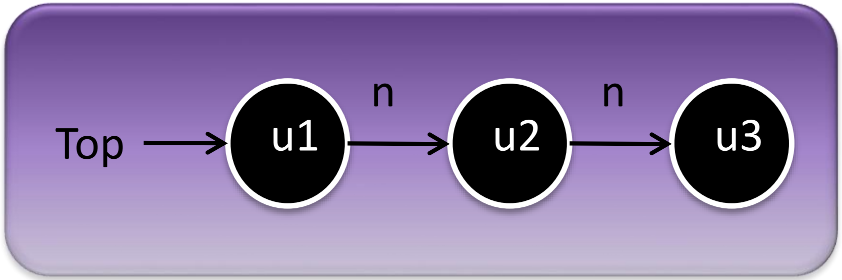
- represent a concrete state as a two-valued logical structure
  - Individuals = heap allocated objects
  - Unary predicates = object properties
  - Binary predicates = relations
- parametric vocabulary



(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

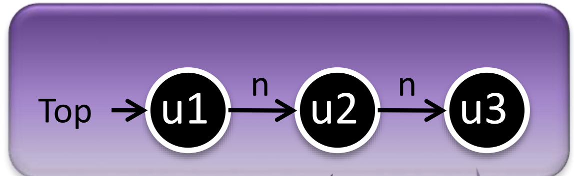


- U = { u1, u2, u3}
- P = { Top, n }

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));
```

$\exists w: x(w)$

$\exists w: x(w)$

```
 Top = x;

}
```

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

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

Top != null

$\exists w:Top(w)$ 1

No node precedes Top

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

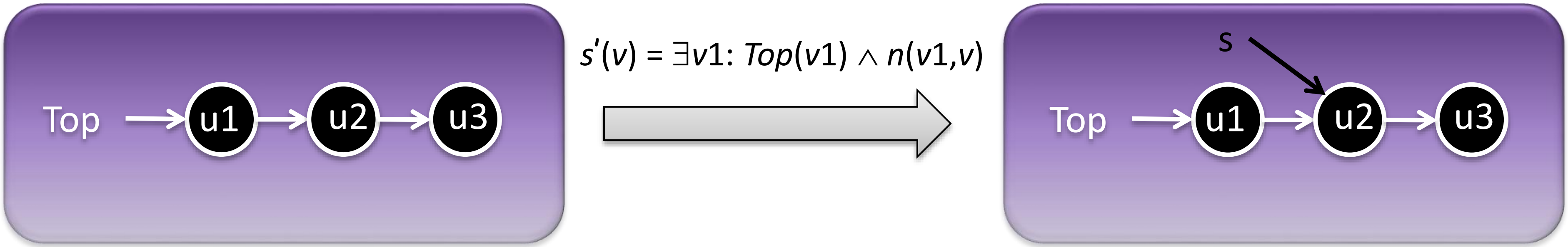
No Cycles

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

Concrete Interpretation Rules

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

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

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

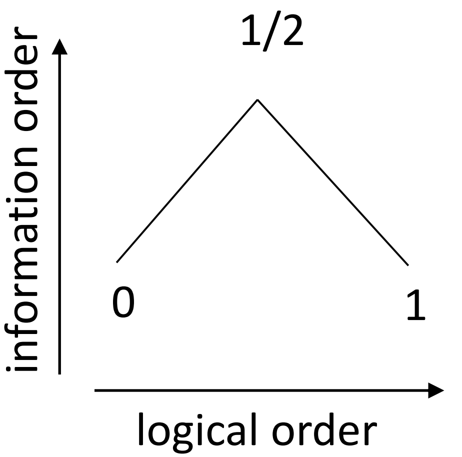
Collecting Semantics

- 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*

    $p^0() \rightarrow \{0,1, \mathit{1/2}\}$

    $p^1(v) \rightarrow \{0,1, \mathit{1/2}\}$

    $p^2(u,v) \rightarrow \{0,1, \mathit{1/2}\}$
- 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 $S \sqsubseteq^{f} S'$

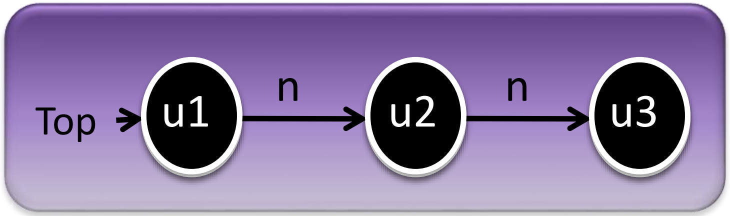
Tight Embedding

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

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

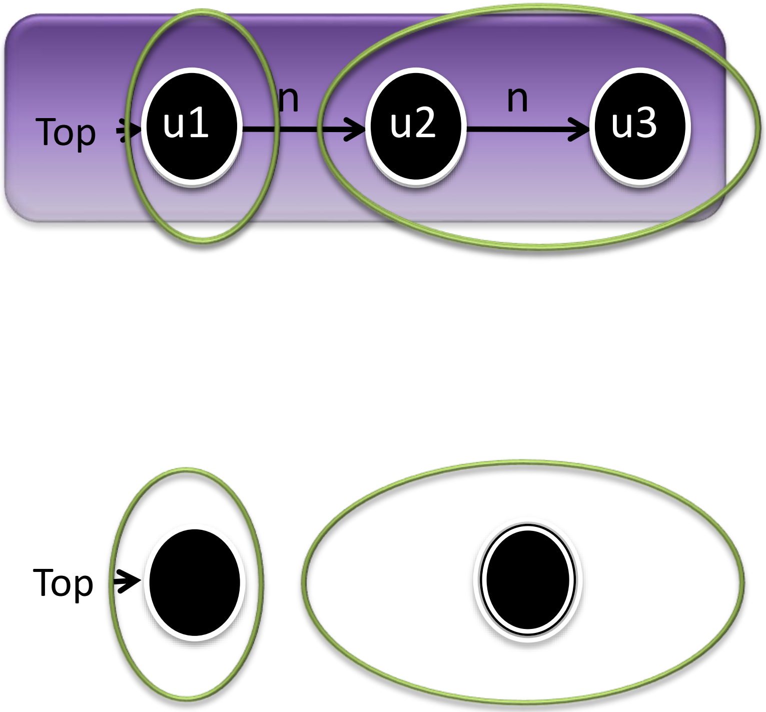
- One way to get tight embedding is canonical abstraction

Canonical Abstraction



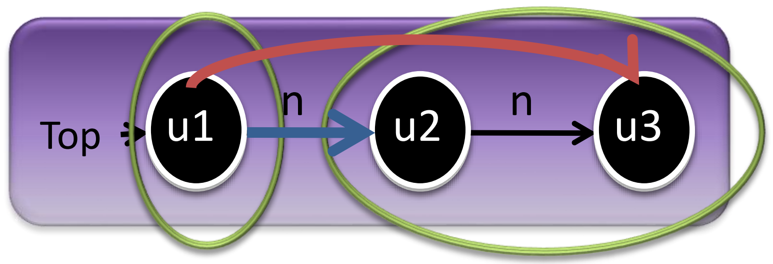
[Sagiv, Reps, Wilhelm, TOPLAS02]

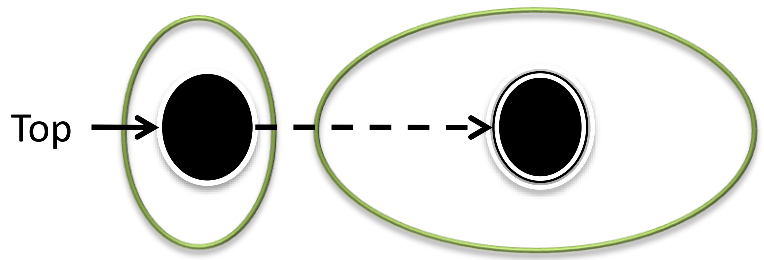
Canonical Abstraction



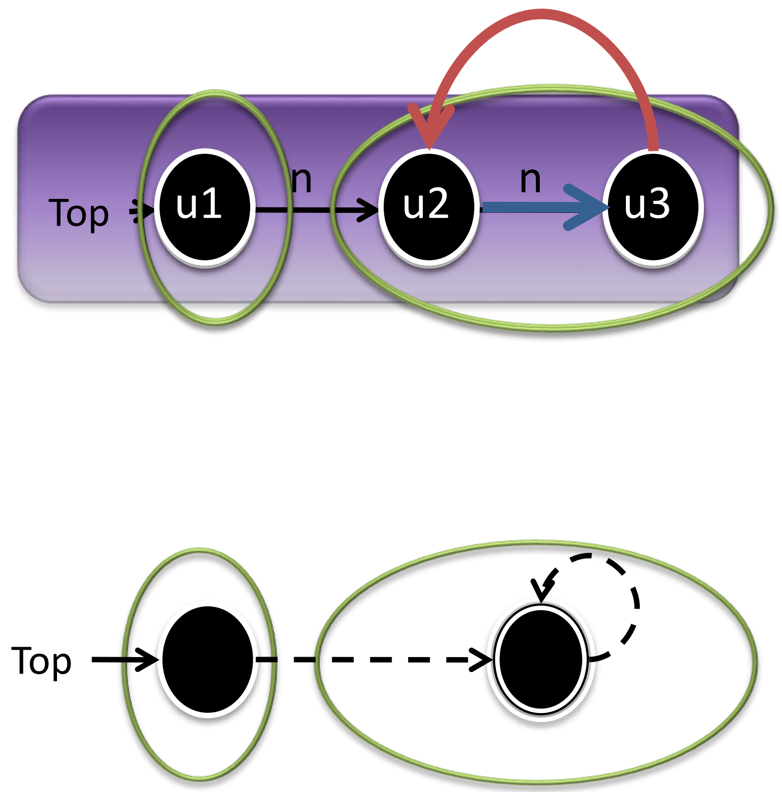
[Sagiv, Reps, Wilhelm, TOPLAS02]

Canonical Abstraction

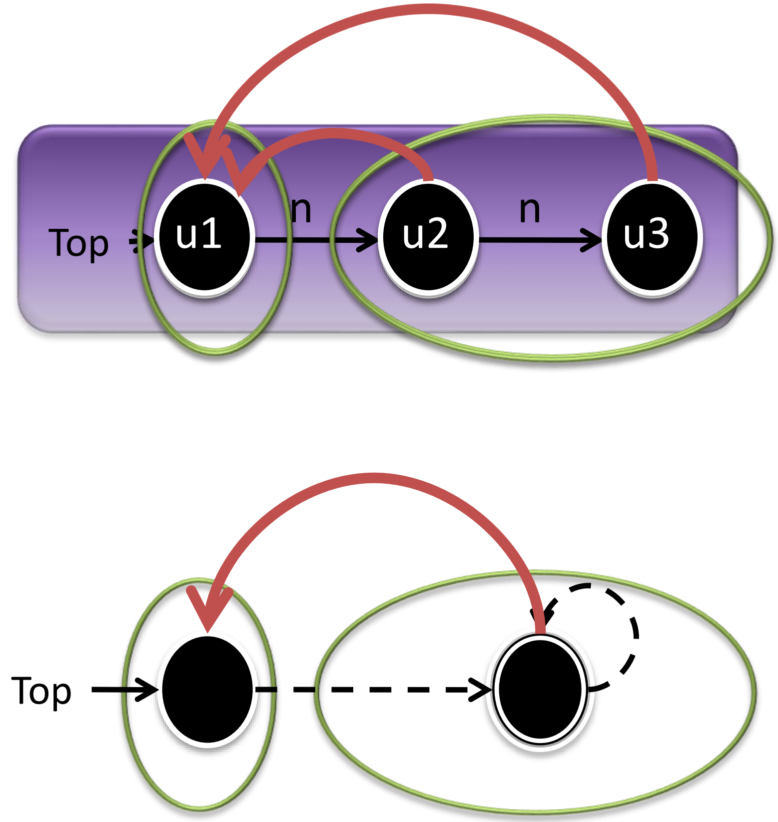




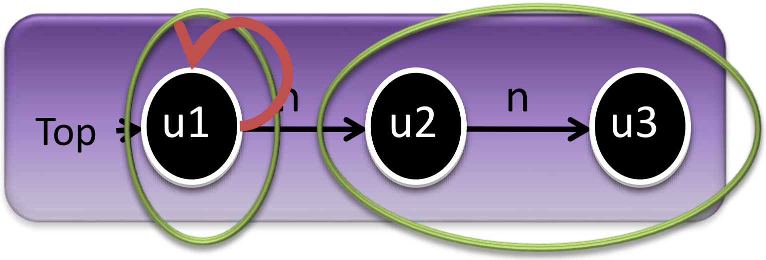
Canonical Abstraction

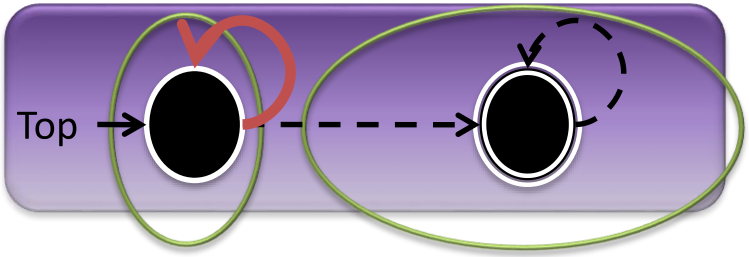


Canonical Abstraction



Canonical Abstraction





Canonical Abstraction (β)

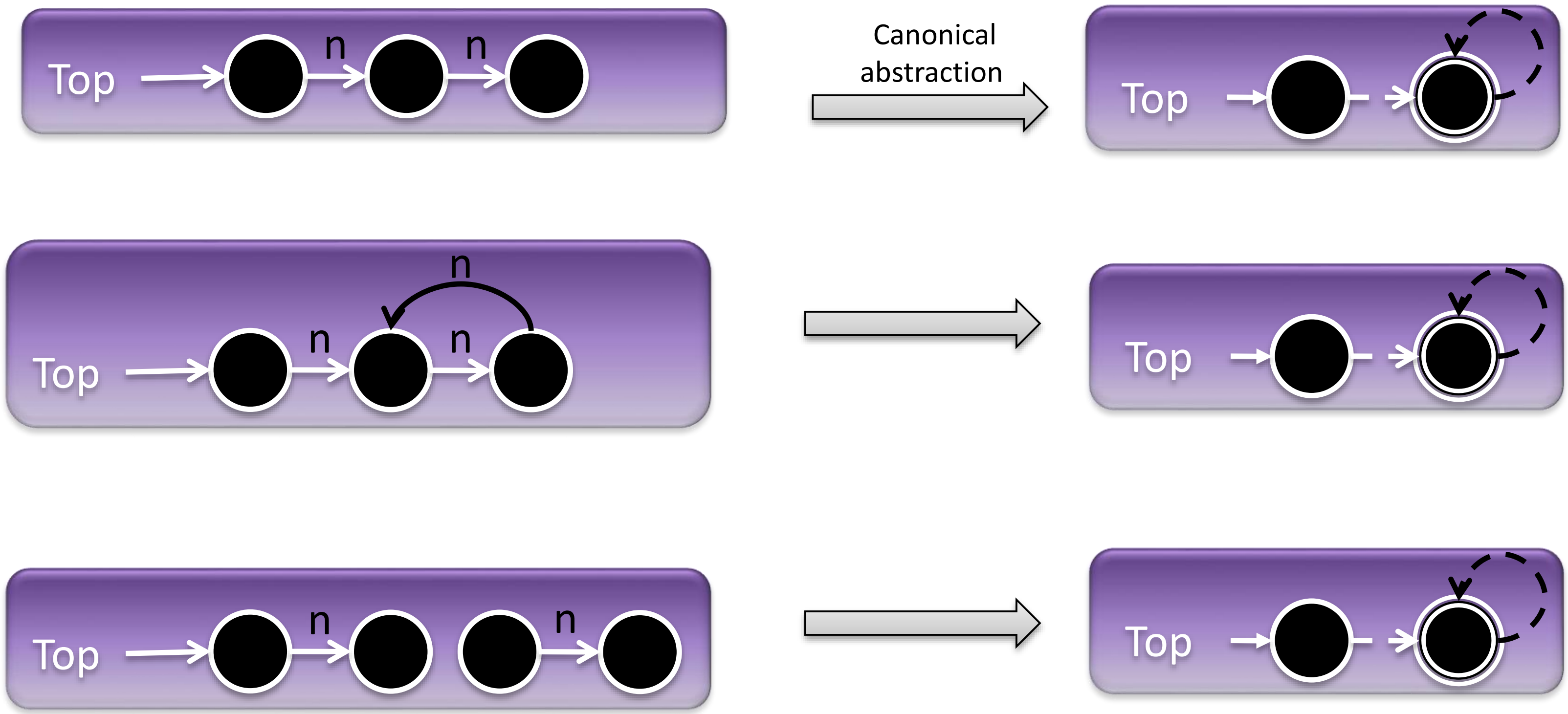
- Merge all nodes with the same unary predicate values into a single summary node
- Join predicate values

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

- 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

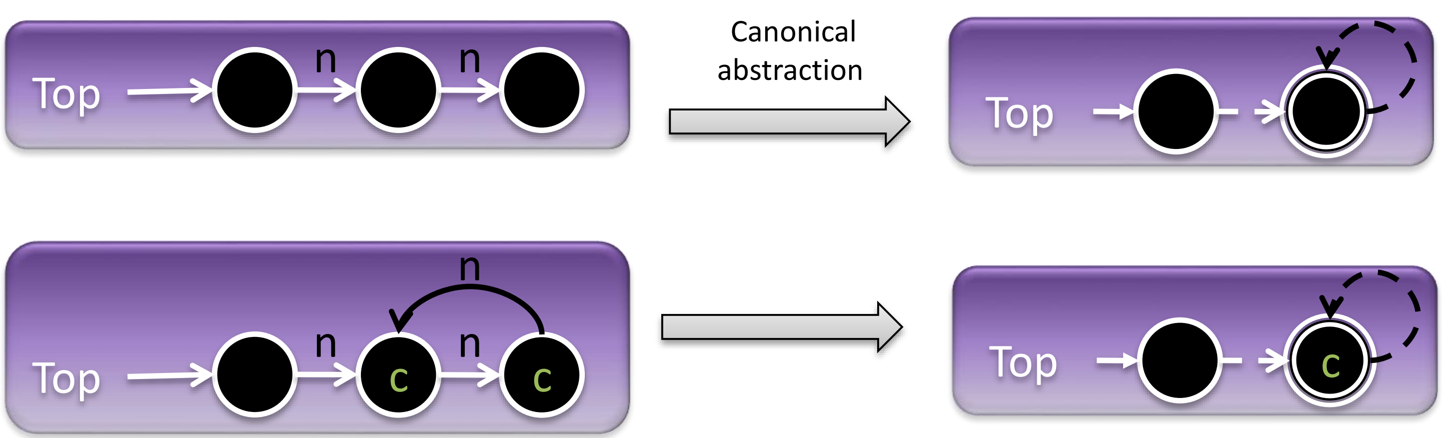


Instrumentation Predicates

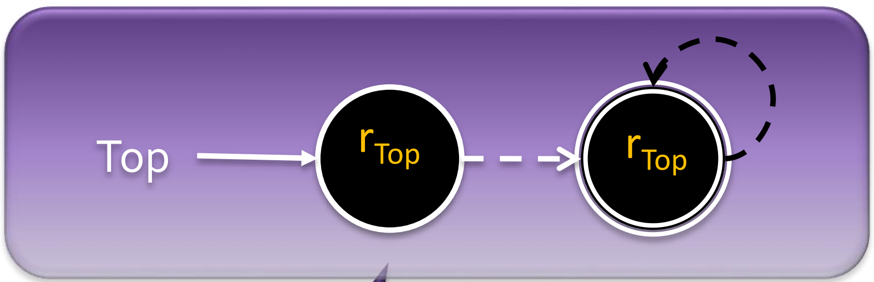
- Record additional derived information via predicates

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

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



Embedding Theorem: **Conservatively** Observing Properties



No Cycles

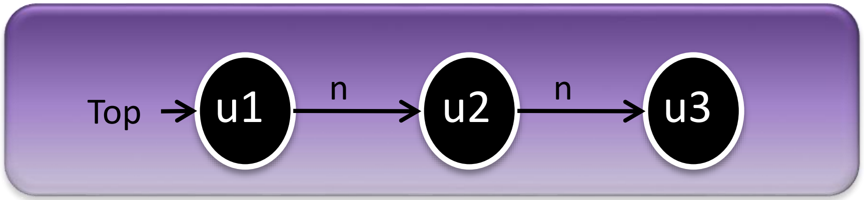
$\neg\exists v1,v2: n(v1, v2) \wedge n^*(v2, v1)$ 1/2

No cycles (derived)

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

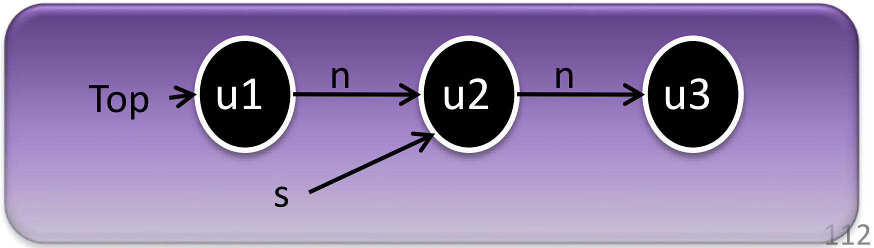
Operational Semantics

```
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;
}
```

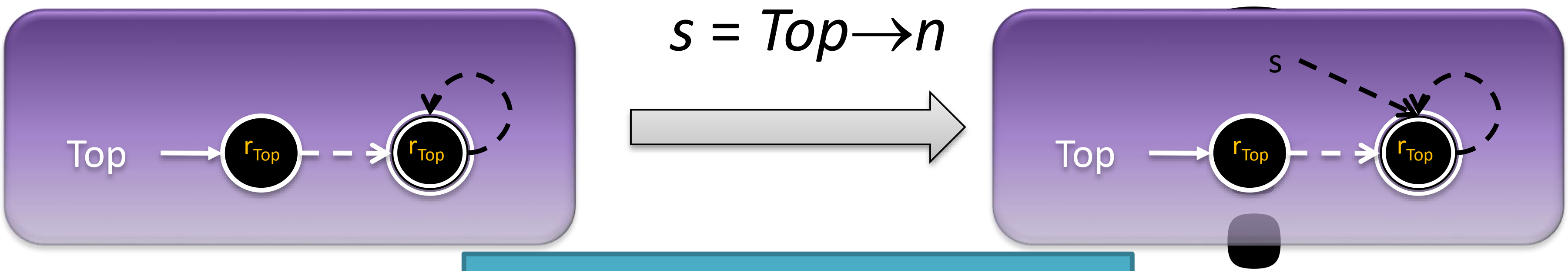


⟦s = Top->n⟧

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



Abstract Semantics



⟦s = Top->n⟧

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

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