HARVARD

School of Engineering
and Applied Sciences

Shape Analysis

CS252r Spring 2011

Outline

- Motivation for shape analysis
- Three-valued logic
- Region-based with tracked locations

Shape analysis

- [Wilhelm, Sagiv, and Reps, CC 2000]
- **Shape analysis**: static program analyses for reasoning about properties of the heap
- Kinds of questions:
  - Null pointers: Is pointer expression maybe null at program point?
  - May-Alias: Can two pointer expressions reference same heap cell?
  - Must-alias: two pointer expression always reference same heap cell
  - Sharing: is there more than one pointer expression referencing a heap cell?
  - Reachability: is the heap cell reachable from a specific variable? any variable?
  - Disjointness: Do two data structures have any common elements?
  - Cyclicity: Can a heap cell be part of a cycle?
- Program understanding, debugging, and verification

Shape analysis

- Shape analysis is flow-sensitive
  - Computes for each point in program "a finite, conservative representation of the heap-allocated data structures that could arise when a path to the program point is executed"
- Finite representation means must be approximate
  - E.g., generally lose info about lengths of lists, depths of trees

Shape Analysis via 3-valued logic

- [Sagiv, Reps, Wilhelm, POPL 99]
- Framework for shape analysis
  - Instantiate by specifying predicates about the heap
  - In concrete execution, these predicates are either true or false
  - In static analysis, approximate the predicates using **3-valued logic**
    - True, False, Don't know

3-valued logic

And	0	I	⊥
0	0	0	0
I	0	I	⊥
⊥	0	⊥	⊥

OR	0	I	⊥
0	0	I	⊥
I	I	I	I
⊥	⊥	I	⊥

3-valued logic

And	0	½	1
0	0	0	0
½	0	½	½
1	0	½	1

OR	0	½	1
0	0	½	1
½	½	½	1
1	1	1	1

Individuals and Predicates

- Universe of individuals U
  - u∈U represents is an abstract location
    - Represents **one** or more concrete locations
    - Each concrete location is represented by exactly one abstract location
- Some predicates
  - pointed-to-by-variable-x(u)
    - Abbreviated to x(u), means that stack variable x points to a concrete location represented by u
  - pointer-component-f-points-to(u_1, u_2)
    - Abbreviated to f(u_1, u_2), means a concrete object rep. by u_1 has field f that points to concrete object rep by u_2
  - sm(u)
    - u is *summary* node, i.e., represents more than 1 concrete location

Meaning of predicates

- $\langle U,\iota\rangle$ is a 3-valued structure
  - U is universe of individuals
  - ι gives valuation to predicates
  - $\iota : p{:}Pred \times U^{arity(p)} \rightarrow \{0, \frac{1}{2}, 1\}$
- A 3-valued structure represents zero or more concrete states
- If formula φ evaluates in $\langle U,\iota\rangle$ to 1, then
  φ holds in every concrete store $\langle U,\iota\rangle$ represents
- If formula φ evaluates in $\langle U,\iota\rangle$ to 0, then
  φ never holds in any concrete store $\langle U,\iota\rangle$ represents
- If formula φ evaluates in $\langle U,\iota\rangle$ to $\frac{1}{2}$, then
  we don't know anything about φ in any concrete store $\langle U,\iota\rangle$ represents

Graphical representation

```
typedef struct node {
  struct node *n;
  int data;
} *List;

/* reverse.c */
#include ''list.h''
List reverse(List x) {
  List y, t;
  assert(acyclic_list(x));
  y = NULL;
  while (x != NULL) {
    t = y;
    y = x;
    x = x->n;
    y->n = t;
  }
  return y;
}
```

S	Structure	Graphical Representation
S_0	**unary predicates:** indiv. x y t sm is **binary predicates:** n	

Figure 2: The three-valued logical structures that describe all possible acyclic inputs to **reverse**.

Graphical representation

```
typedef struct node {
  struct node *n;
  int data;
} *List;

/* reverse.c */
#include ''list.h''
List reverse(List x) {
  List y, t;
  assert(acyclic_list(x));
  y = NULL;
  while (x != NULL) {
    t = y;
    y = x;
    x = x->n;
    y->n = t;
  }
  return y;
}
```

S	Structure	Graphical Representation
S_0	**unary predicates:** indiv. | x | y | t | sm | is **binary predicates:** n	
S_1	**unary predicates:** indiv. | x | y | t | sm | is u_1 | 1 | 0 | 0 | 0 | 0 **binary predicates:** n | u_1 u_1 | 0	x → u_1
S_2	**unary predicates:** indiv. | x | y | t | sm | is u_1 | 1 | 0 | 0 | 0 | 0 u | 0 | 0 | 0 | 1/2 | 0 **binary predicates:** n | u_1 | u u_1 | 0 | 1/2 u | 0 | 1/2	x → u_1 ⋯n⋯→ u, n → u

Figure 2: The three-valued logical structures that describe all possible acyclic inputs to **reverse**.

Updating formula

- Key idea: track state of formula at each program point.
  - Just like dataflow

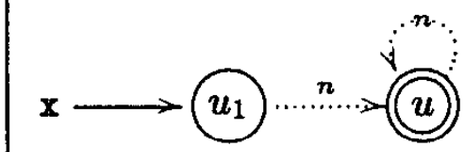
S_2

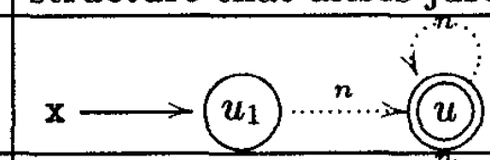
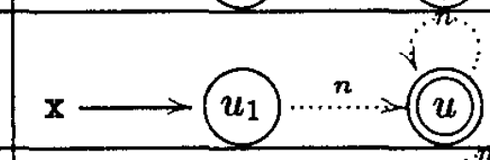
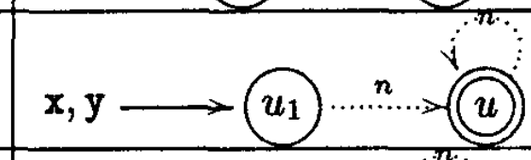
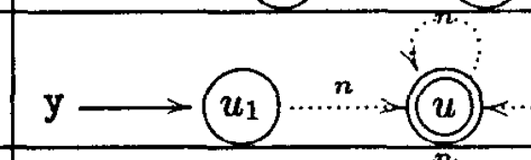
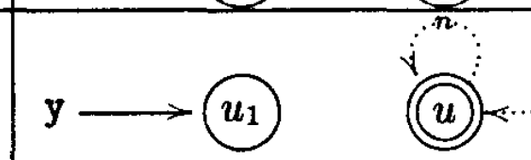
unary predicates:

indiv.	x	y	t	sm	is
u_1	1	0	0	0	0
u	0	0	0	1/2	0

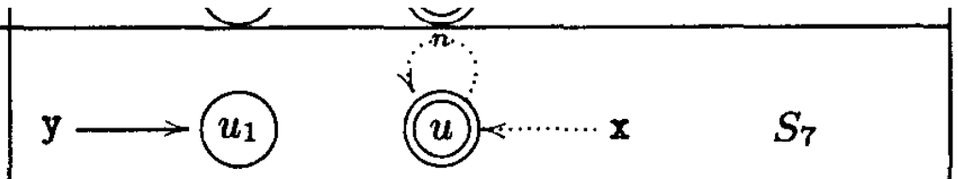
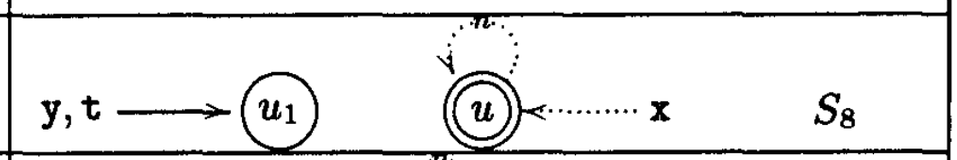
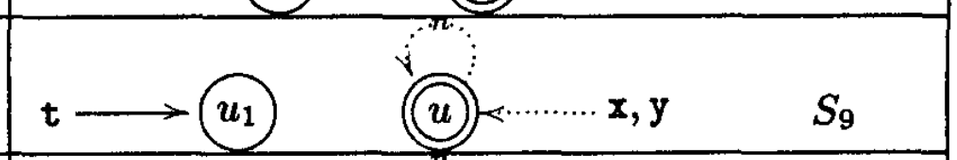
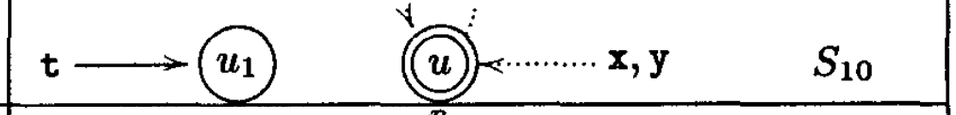
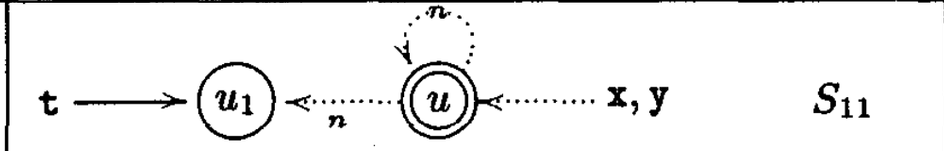
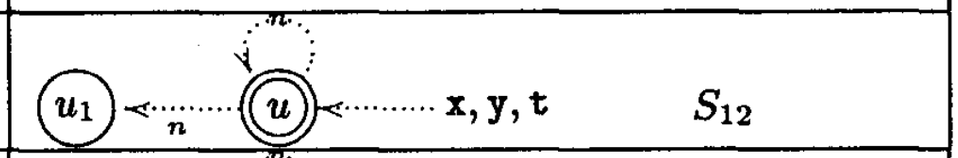
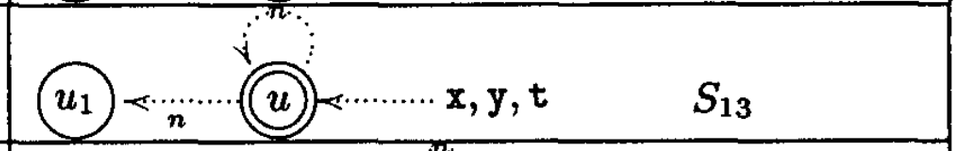
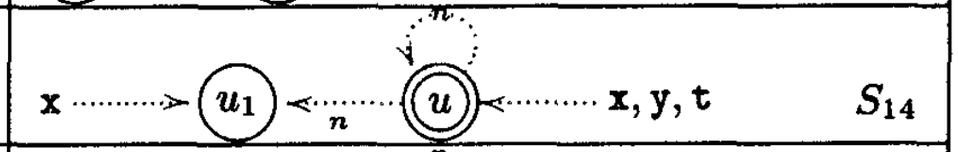
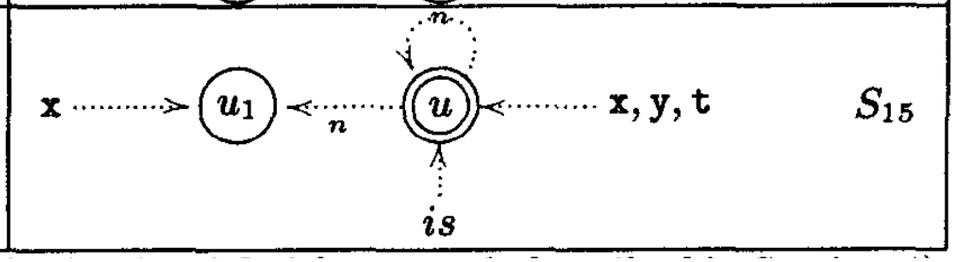
binary predicates:

n	u_1	u
u_1	0	1/2
u	0	1/2



statement	formula	structure that arises just after statement
st_1: `y = NULL;`	$y'(v) = 0$	 S_3
st_2: `t = y;`	$t'(v) = y(v)$	 S_4
st_3: `y = x;`	$y'(v) = x(v)$	 S_5
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	 S_6
st_5: `y->n = t;`	$n'(v_1, v_2) = (n(v_1, v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = is(v) \wedge \exists v_1, v_2 : \begin{pmatrix} v_1 \neq v_2 \wedge n(v_1, v) \wedge n(v_2, v) \\ \wedge \neg y(v_1) \wedge \neg y(v_2) \end{pmatrix}$ $\vee (t(v) \wedge \exists v_1 : n(v_1, v) \wedge \neg y(v_1))$	 S_7

Updating formula

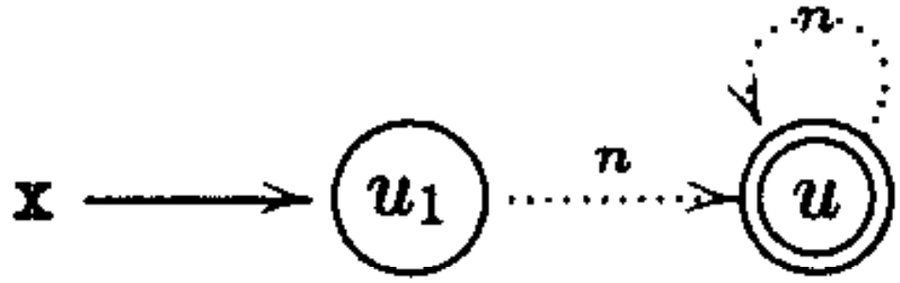
st_5: `y->n = t;`	$n'(v_1,v_2) = (n(v_1,v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = is(v) \wedge \exists v_1, v_2 : \begin{pmatrix} v_1 \neq v_2 \wedge n(v_1,v) \wedge n(v_2,v) \\ \wedge \neg y(v_1) \wedge \neg y(v_2) \end{pmatrix}$ $\vee (t(v) \wedge \exists v_1 : n(v_1,v) \wedge \neg y(v_1))$	 S_7
st_2: `t = y;`	$t'(v) = y(v)$	 S_8
st_3: `y = x;`	$y'(v) = x(v)$	 S_9
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1,v)$	 S_{10}
st_5: `y->n = t;`	$n'(v_1,v_2) = (n(v_1,v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = is(v) \wedge \exists v_1, v_2 : \begin{pmatrix} v_1 \neq v_2 \wedge n(v_1,v) \wedge n(v_2,v) \\ \wedge \neg y(v_1) \wedge \neg y(v_2) \end{pmatrix}$ $\vee (t(v) \wedge \exists v_1 : n(v_1,v) \wedge \neg y(v_1))$	 S_{11}
st_2: `t = y;`	$t'(v) = y(v)$	 S_{12}
st_3: `y = x;`	$y'(v) = x(v)$	 S_{13}
st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1,v)$	 S_{14}
st_5: `y->n = t;`	$n'(v_1,v_2) = (n(v_1,v_2) \wedge \neg y(v_1)) \vee (y(v_1) \wedge t(v_2))$ $is'(v) = is(v) \wedge \exists v_1, v_2 : \begin{pmatrix} v_1 \neq v_2 \wedge n(v_1,v) \wedge n(v_2,v) \\ \wedge \neg y(v_1) \wedge \neg y(v_2) \end{pmatrix}$ $\vee (t(v) \wedge \exists v_1 : n(v_1,v) \wedge \neg y(v_1))$	 S_{15}

Instrumentation predicates

- Consider formula

$$\varphi(v) = \exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$$

  - "There are at least two different objects pointing to v"



- What does $\varphi(u)$ evaluate to, for shape graph above?
  - With $v_1 = u_1$, $v_2 = u$, we have

$$n(u_1, u) \wedge n(u, u) \wedge u_1 \neq u \equiv \frac{1}{2} \wedge \frac{1}{2} \wedge 1 \equiv \frac{1}{2}$$

  - Implies that tail of linked list might be shared
  - But this is not the case for a linked list!

Instrumentation predicates

- Maintain precision by using **instrumentation predicates**
  - predicate is(u) represents truth of predicate for nodes represented by abstract location
    - Is Shared
- is(u)=0 implies that S_2 can only represent acyclic lists

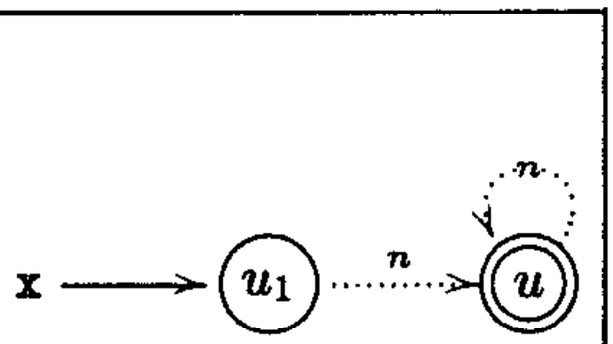
S_2

unary predicates:

indiv.	x	y	t	sm	is
u_1	1	0	0	0	0
u	0	0	0	1/2	0

binary predicates:

n	u_1	u
u_1	0	1/2
u	0	1/2

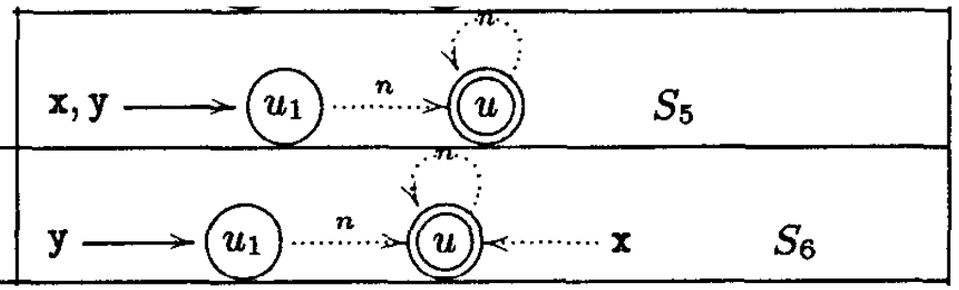


Other useful instrumentation predicates

Pred.	Intended Meaning	Purpose	Ref.
$is(v)$	Do two or more fields of heap elements point to v?	lists and trees	[2], [19]
$r_x(v)$	Is v (transitively) reachable from pointer variable x?	separating disjoint data structures	[19]
$r(v)$	Is v reachable from some pointer variable (i.e., is v a non-garbage element)?	compile-time garbage collection	
$c(v)$	Is v on a directed cycle?	ref. counting	[11]
$c_{f.b}(v)$	Does a field-f dereference from v, followed by a field-b dereference, yield v?	doubly-linked lists	[7], [16]
$c_{b.f}(v)$	Does a field-b dereference from v, followed by a field-f dereference, yield v?	doubly-linked lists	[7], [16]

Focus for precision

- Once the value of a formula is ½, it can be easy to lose precision.

st_4: `x = x->n;`	$x'(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$	

- **Focusing** may allow us to maintain precision
- Key idea: if update formula evaluates to ½, try instantiating it to 0 and 1
  - Focus attention on each of the possible cases
  - May need to make sure rest of structure is consistent

Focus example

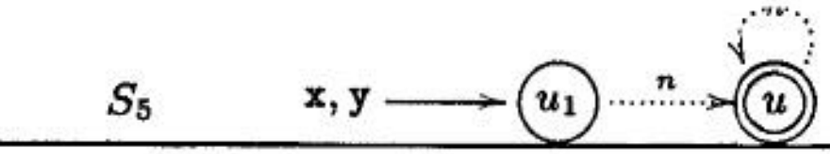
input struct.	S_5 
focus formulae	$\{\varphi_x^{st_4}(v) = \exists v_1 : x(v_1) \wedge n(v_1, v), \varphi_y^{st_4}(v) = y(v), \varphi_t^{st_4}(v) = t(v)\}$

Figure 5: The first application of the improved transformer for statement st_4: `x = x->n` in `reverse`.

Focus example

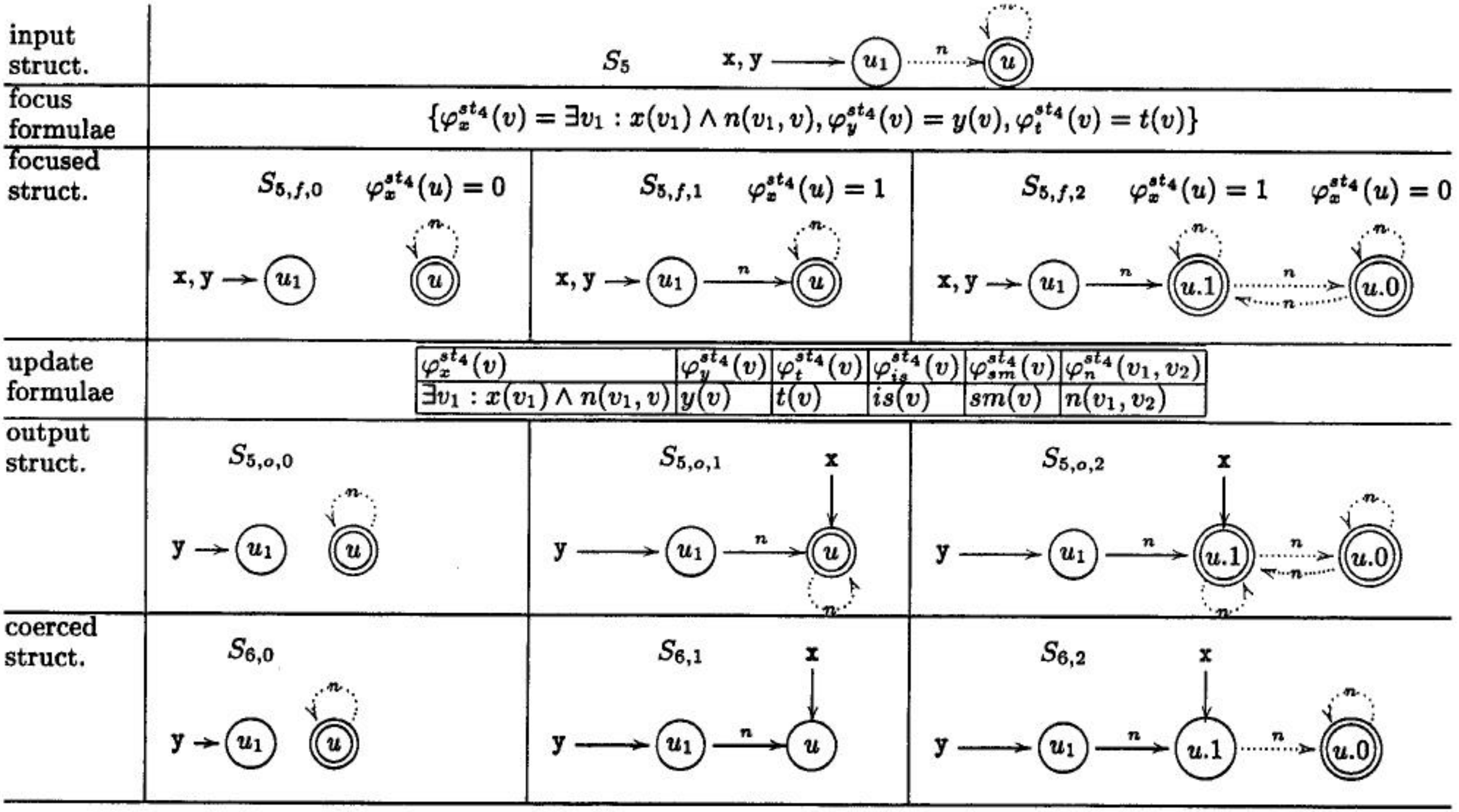


Figure 5: The first application of the improved transformer for statement st_4: `x = x->n` in reverse.

Region-based shape analysis with tracked locations

- Hackett and Rugina, POPL 05
- Key idea: reason about **one location at a time**
- Allows a decomposition of a state into a set of tracked locations
  - Reason about each tracked location independently of others
  - Better scalability, compact representation, context-sensitive analysis
  - No need to merge abstractions, or keep multiple abstractions of entire heap
  - Easier on-demand and incremental algorithms

Memory regions

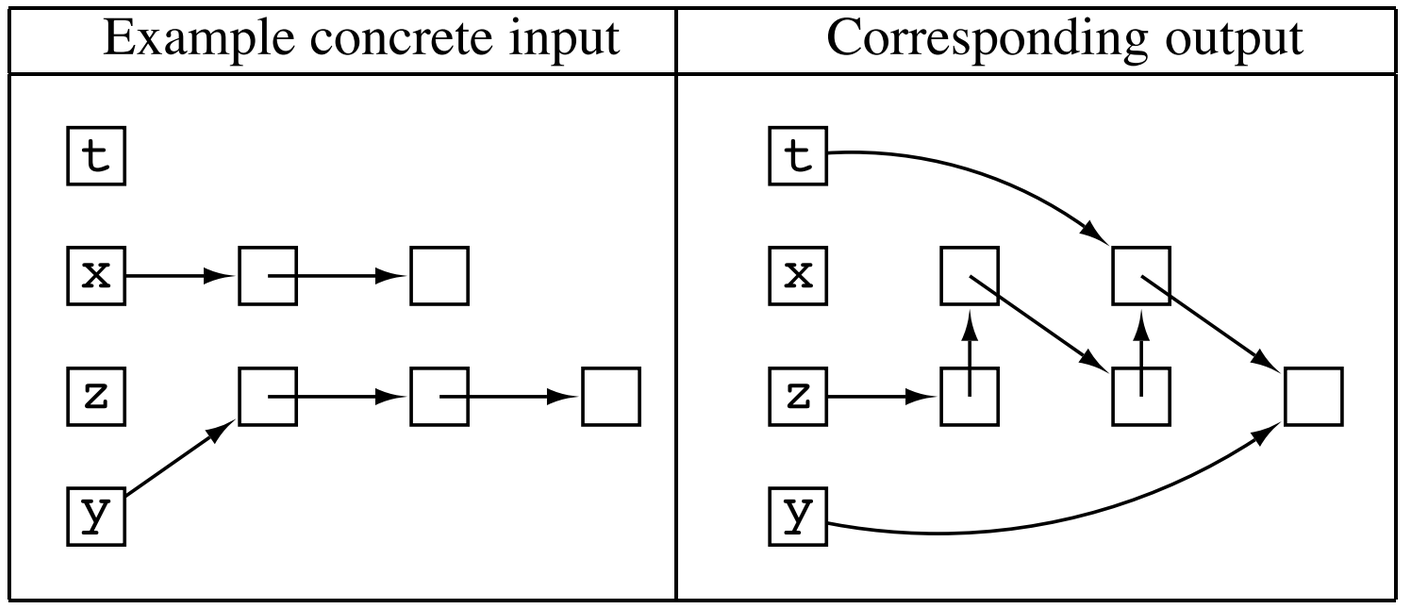
- Analysis builds on top of a **region analysis**
  - Each region represents a set of concrete locations
  - Each concrete location represented by exactly one region
  - Points-to relation over regions must be sound
  - Can use a variety of region analyses
    - E.g., flow-sensitive or insensitive
    - In paper, they use a flow-insensitive, context-sensitive analysis that uses an intra-procedural unification-based analysis, and uses procedure summaries for an interprocedural analysis

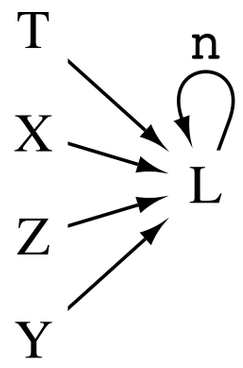
Configurations and shape abstractions

- A **configuration** is (i, (e^+, e^-))
  - i is **index**, a function from regions to {0,…,k,∞}
    - i(r) = How many locations in r point to tracked location
      - ∞ means ≥ k+1
  - e^+ is **hit set**: expressions that definitely refer to tracked location
  - e^- is **miss set**: expressions that definitely do not refer to tracked location
- A **shape abstraction** is a set of configurations
  - at most one configuration for each index
  - each concrete location should be represented by at least one configuration
  - Treat shape abstraction as partial function from indexes to hit/miss sets

Example

```
 1:  typedef struct list {
 2:     struct list *n;
 3:     int data;
 4:  } List;
 5:
 6:  List *splice(List *x, List *y) {
 7:    List *t = NULL;
 8:    List *z = y;
 9:    while(x != NULL) {
10:      t = x;
11:      x = t->n;
12:      t->n = y->n;
13:      y->n = t;
14:      y = y->n->n;
15:    }
16:    return z;
17:  }
```



Region Points-to Component	Shape Component: Configurations for input memory	Shape Component: Configurations for output memory
	$(X^1, \{\texttt{x}\}, \emptyset)$ $(Y^1, \{\texttt{y}\}, \emptyset)$ $(L^1, \emptyset, \emptyset)$	$(Z^1, \{\texttt{z}\}, \emptyset)$ $(T^1\, L^1, \{\texttt{t}\}, \emptyset)$ $(Y^1\, L^1, \{\texttt{y}\}, \emptyset)$ $(L^1, \emptyset, \emptyset)$

Intra-procedural analysis

$$(a_1 \sqcup a_2)(i) = \begin{cases} a_1(i) & \text{if } i \notin dom(a_2) \\ a_2(i) & \text{if } i \notin dom(a_1) \\ a_1(i) \sqcup a_2(i) & \text{if } i \in dom(a_1) \cap dom(a_2) \end{cases}$$

where $(e_1^+, e_1^-) \sqcup (e_2^+, e_2^-) = (e_1^+ \cap e_2^+, e_1^- \cap e_2^-)$
and $\bot \sqcup (e^+, e^-) = (e^+, e^-) \sqcup \bot = (e^+, e^-)$

For all $s \in S_{asgn},\ s_a \in S_{alloc},\ s_e \in S_{entry},\ i \in I$:

[JOIN] $\quad Res(\bullet s)\, i = \bigsqcup_{s' \in pred(s)} Res(s' \bullet)\, i$

[TRANSF] $\quad Res(s \bullet)\, i = \bigsqcup_{i' \in I} ([\![s]\!](\rho, (i', Res(\bullet s)\, i')))\, i$

[ALLOC] $\quad Res(s_a \bullet)\, i_a \sqsupseteq h_a,$ where $[\![s_a]\!]^{gen}(\rho) = (i_a, h_a)$

[ENTRY] $\quad Res(\bullet s_e)\, i \sqsupseteq a_o\, i$

Splice example

```
 8: z = y;

 9: while (x != NULL)

10:   t = x;

11:   x = t->n;

12:   t->n = y->n;

13:   y->n = t;

14:   y = y->n->n;

16: return z;
```

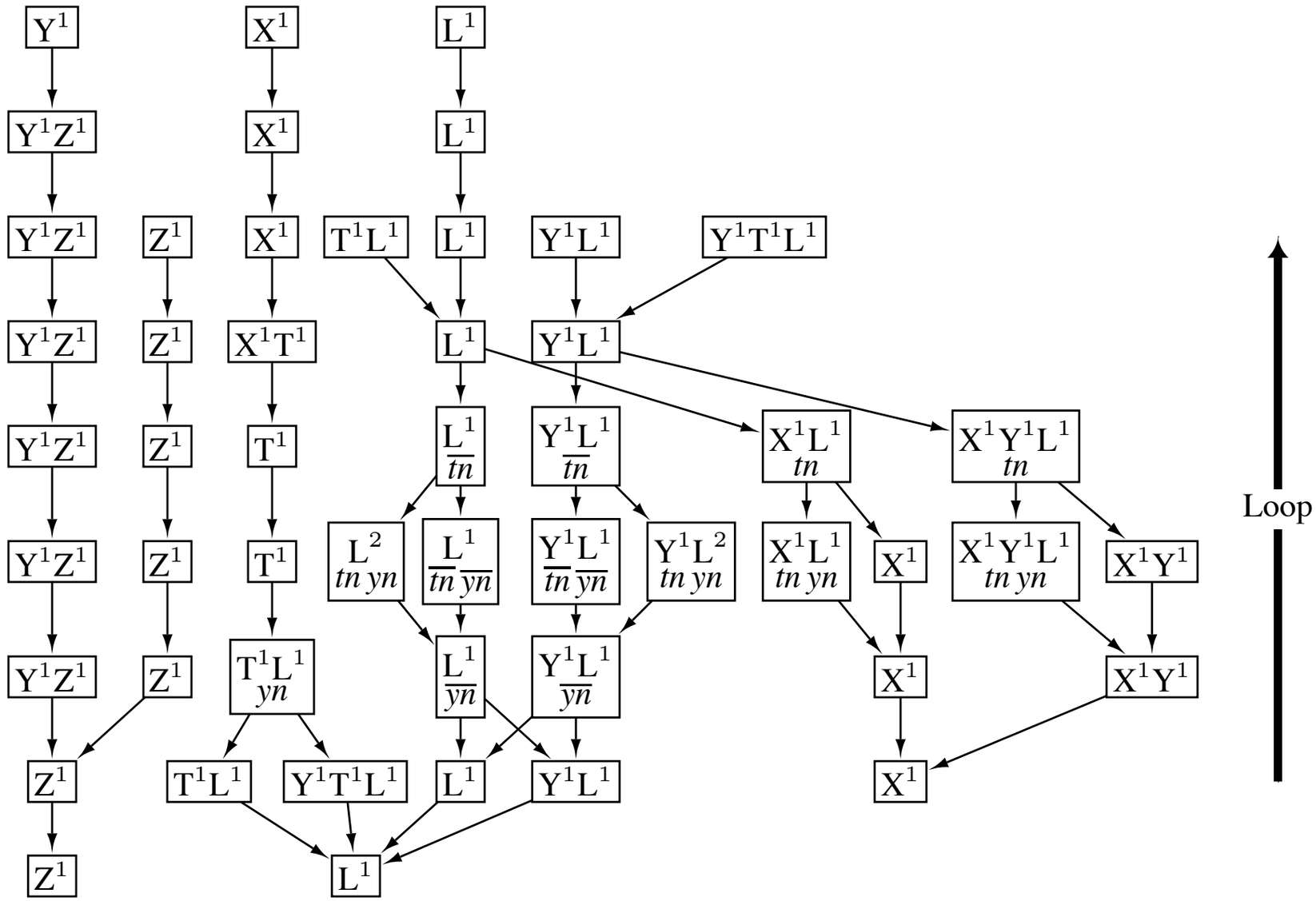


Figure 5: Shape analysis results for `splice`. Boxes represent configurations and edges show how the state of the tracked location changes during the execution. We only show field access expressions in the hit and miss sets. We use the abbreviations: *tn*≡ `t->n` and *yn*≡ `y->n`, and we indicate miss expressions using overlines. For readability, back edges from configurations at the end of the loop to the corresponding configurations at the beginning of the loop are omitted.

Transfer function for assign

$[\![e_o \leftarrow e_1]\!](\rho, (i, (e^+, e^-))) :$

case $(\mathcal{D}[\![e_0]\!](\rho, (i, (e^+, e^-))), \mathcal{D}[\![e_1]\!](\rho, (i, (e^+, e^-)))$ **of**

$(v_0 \in \{-, +\},\ v_1 \in \{-, +\}) \Rightarrow$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+, e^-, v_0 = +, v_1 = +)$

$(?,\ v_1 \in \{+, -\}) \Rightarrow$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+ \cup \{e_0\}, e^-, \text{true}, v_1 = +) \sqcup$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+, e^- \cup \{e_0\}, \text{false}, v_1 = +)$

$(v_0 \in \{-, +\},\ ?) \Rightarrow$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+ \cup \{e_1\}, e^-, v_0 = +, \text{true}) \sqcup$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+, e^- \cup \{e_1\}, v_0 = +, \text{false})$

$(?,\ ?) \Rightarrow$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+ \cup \{e_0, e_1\}, e^-, \text{true}, \text{true}) \sqcup$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+ \cup \{e_0\}, e^- \cup \{e_1\}, \text{true}, \text{false}) \sqcup$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+ \cup \{e_1\}, e^- \cup \{e_0\}, \text{false}, \text{true}) \sqcup$
  $\mathit{assign}(e_0, e_1, \rho, i, e^+, e^- \cup \{e_0, e_1\}, \text{false}, \text{false})$

Figure 10: Transfer function for assignments $[\![e_0 \leftarrow e_1]\!]$.

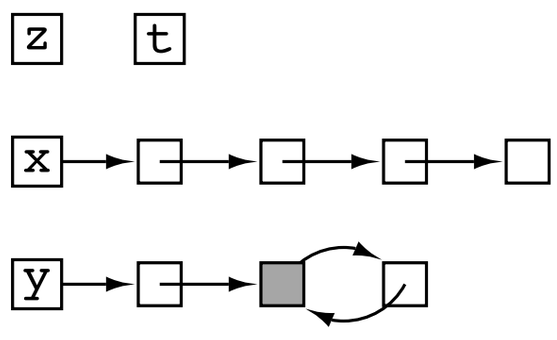
$\mathit{assign}(e_0, e_1, \rho, i, e^+, e^-, b_0, b_1) :$

```
1   r = L[[e0]](ρ)
2   if (b0 ∧ ¬b1) then
3     if (i(r) ≤ k) then S_i = { i[r ↦ i(r) − 1] }
4                   else S_i = { i[r ↦ k], i[r ↦ ∞] }
5   else if (¬b0 ∧ b1) then
6     if (i(r) < k) then S_i = { i[r ↦ i(r) + 1] }
7                   else S_i = { i[r ↦ ∞] }
8   else S_i = { i }
9
10  e_n^+ = {e ∈ e^+ | S[[e]]_v(ρ, r) ∨ (S[[e]]_l(ρ, r) ∧ b1)}
11  e_n^- = {e ∈ e^- | S[[e]]_v(ρ, r) ∨ (S[[e]]_l(ρ, r) ∧ ¬b1)}
12
13  if (S[[e0]]_l(ρ, r) ∧ b1)  then e_n^+ ∪= {e0}
14  if (S[[e0]]_l(ρ, r) ∧ ¬b1) then e_n^- ∪= {e0}
15
16  if (S[[e0]]_l(ρ, r) ∧ S[[e1]]_l(ρ, r)) then
17      e_n^+ ∪= ( e_n^+[*e0/*e1] ∩ E'_p )
18      e_n^- ∪= ( e_n^-[*e0/*e1] ∩ E'_p )
19
20  e_n^- = {e ∈ e_n^- | ∀i' ∈ S_i . i'(L[[e]]ρ) = 0}
21
22  return (S_i, (e_n^+, e_n^-))
```

Figure 11: Helper function *assign*.

Interprocedural analysis

- Context-sensitive interprocedural analysis
- A context for a procedure is a single configuration, output is a set of contexts
  - Fine granularity helps scalability
    - Less redundant computation
    - Allows "incremental" analysis
      - E.g., now calling splice with a cyclic list
        - Just one new configuration: L^2



Uses and limitations

- Can be used for memory error detection
  - Double frees, dangling pointer access, memory leak
- Spurious configurations
  - Configuration that represents concrete states that cannot occur at runtime
  - Better decision procedure would help
- Complex structural invariants
  - e.g., double linked lists
- Sensitive to how program is written
  - e.g., `x = t->n` vs `x = x->n` treated differently, since analysis doesn't know x = t
- Exponential

Verification vs. inference

- Separation logic has shown a lot of success at verifying programs that destructively update heap
- To what extent can separation logic be used in inference of heap properties?