CS711 Advanced Programming Languages
Shape Analysis With Tracked Locations

Radu Rugina

22 Sep 2005

Shape Analysis with Local Reasoning

- All previous abstractions:
  - Describe the entire heap at once
  - Makes inter-procedural analysis difficult

- This approach:
  - Idea 1: build shape analysis on top of an underlying pointer analysis
  - Idea 2: Reason locally about one heap cell at a time.

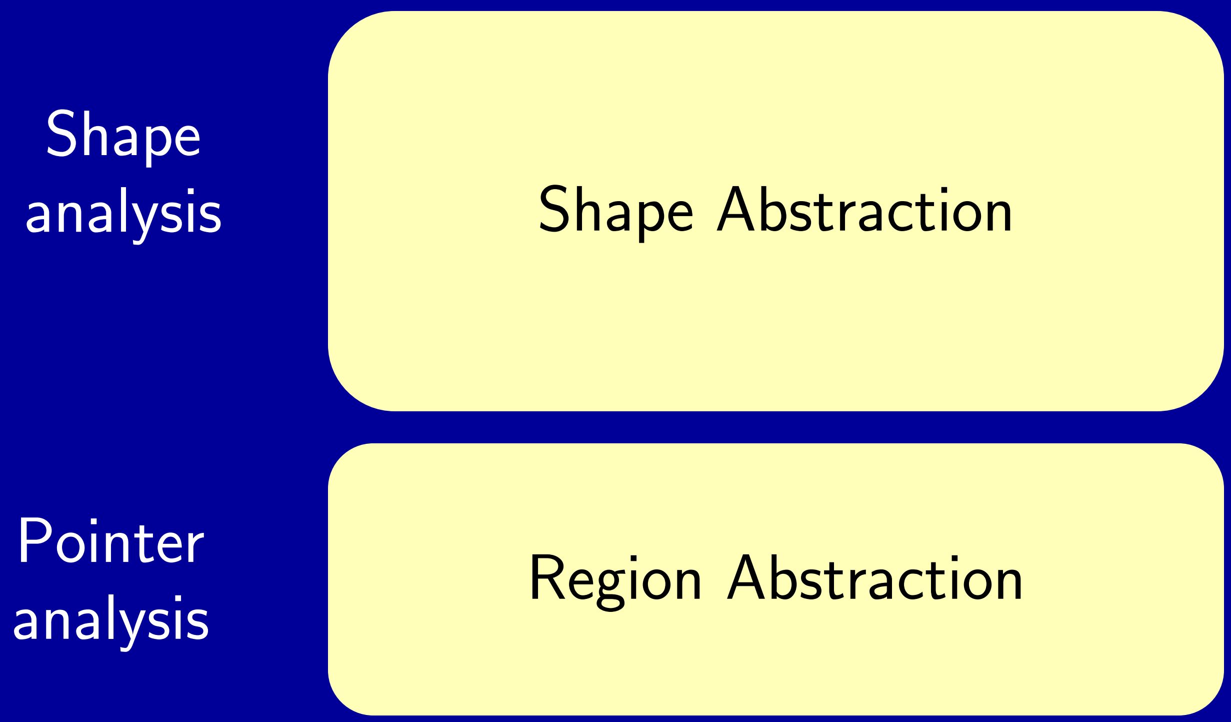
New Memory Abstraction

- Decompose memory abstraction

Heap Abstraction

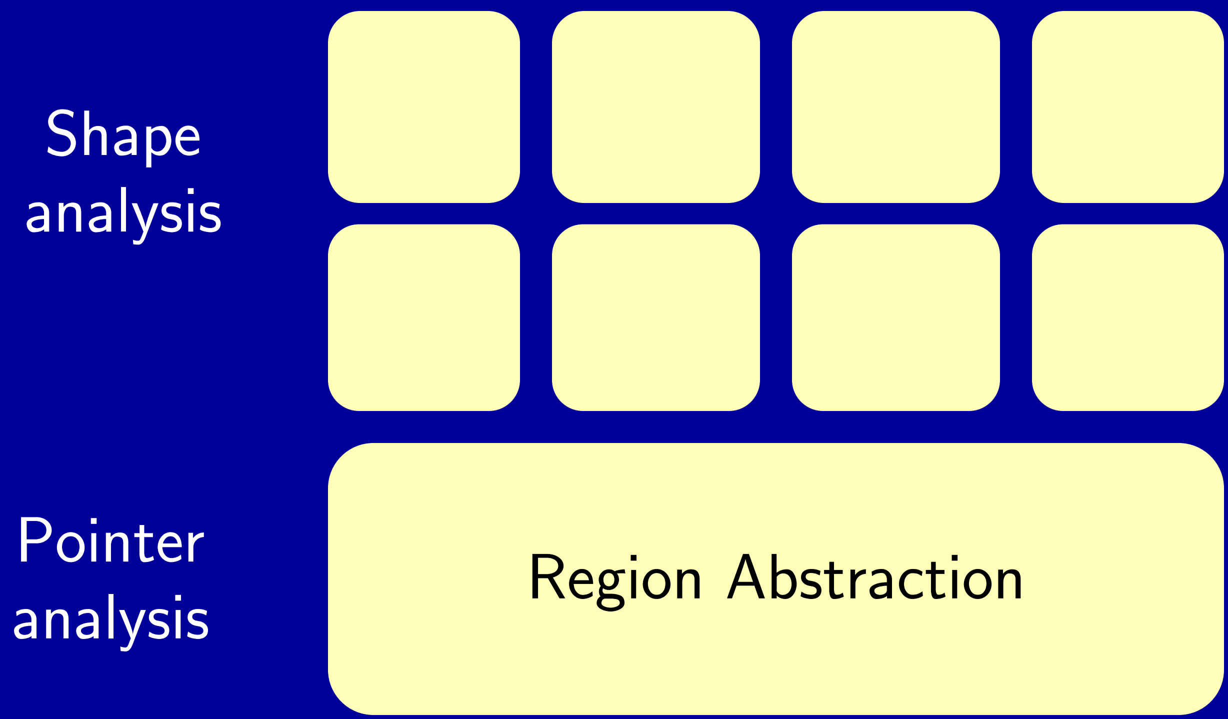
New Memory Abstraction

- Decompose memory abstraction
  - run pointer analysis, then shape analysis



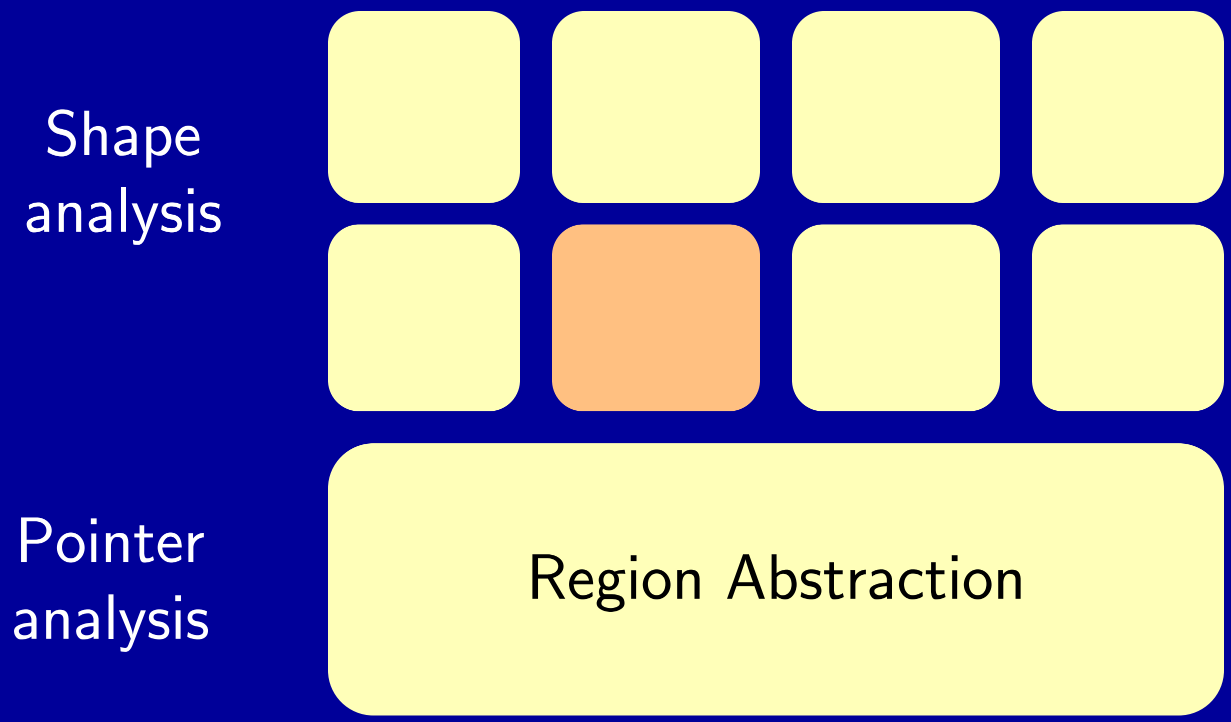
New Memory Abstraction

- Decompose memory abstraction
  - Build shape abstraction using *independent* pieces

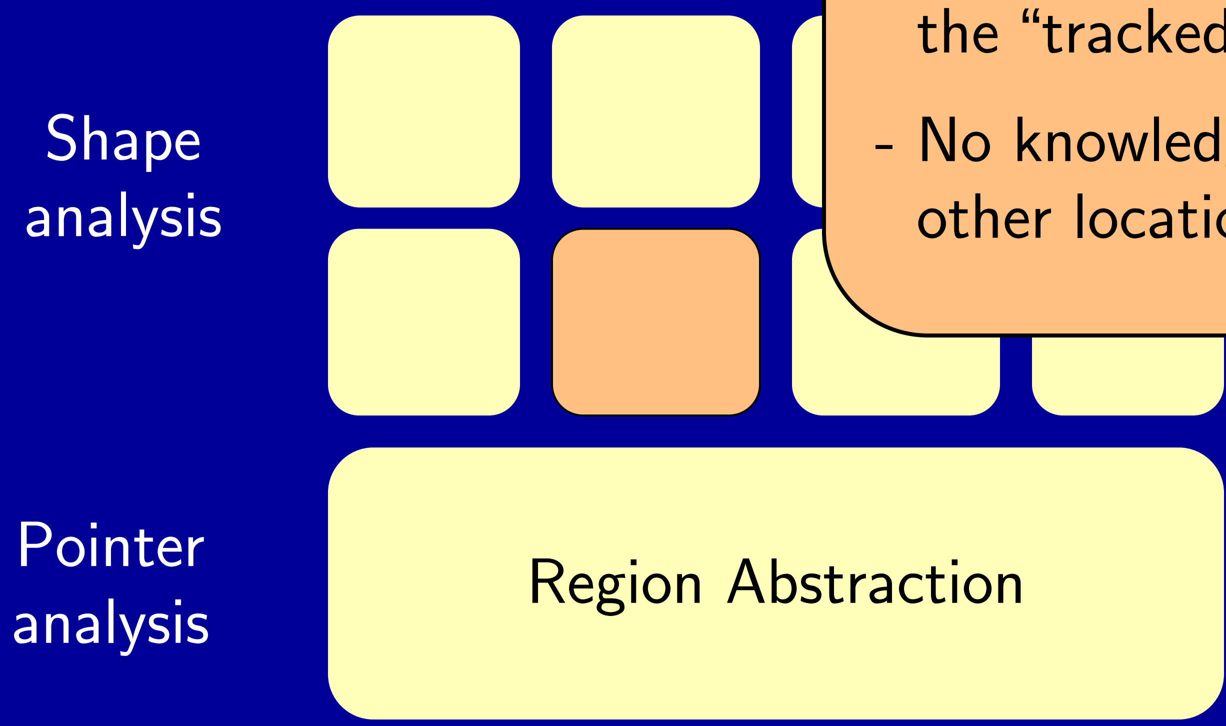


New Memory Abstraction

- Decompose memory abstraction
  - Build shape abstraction using *independent* pieces



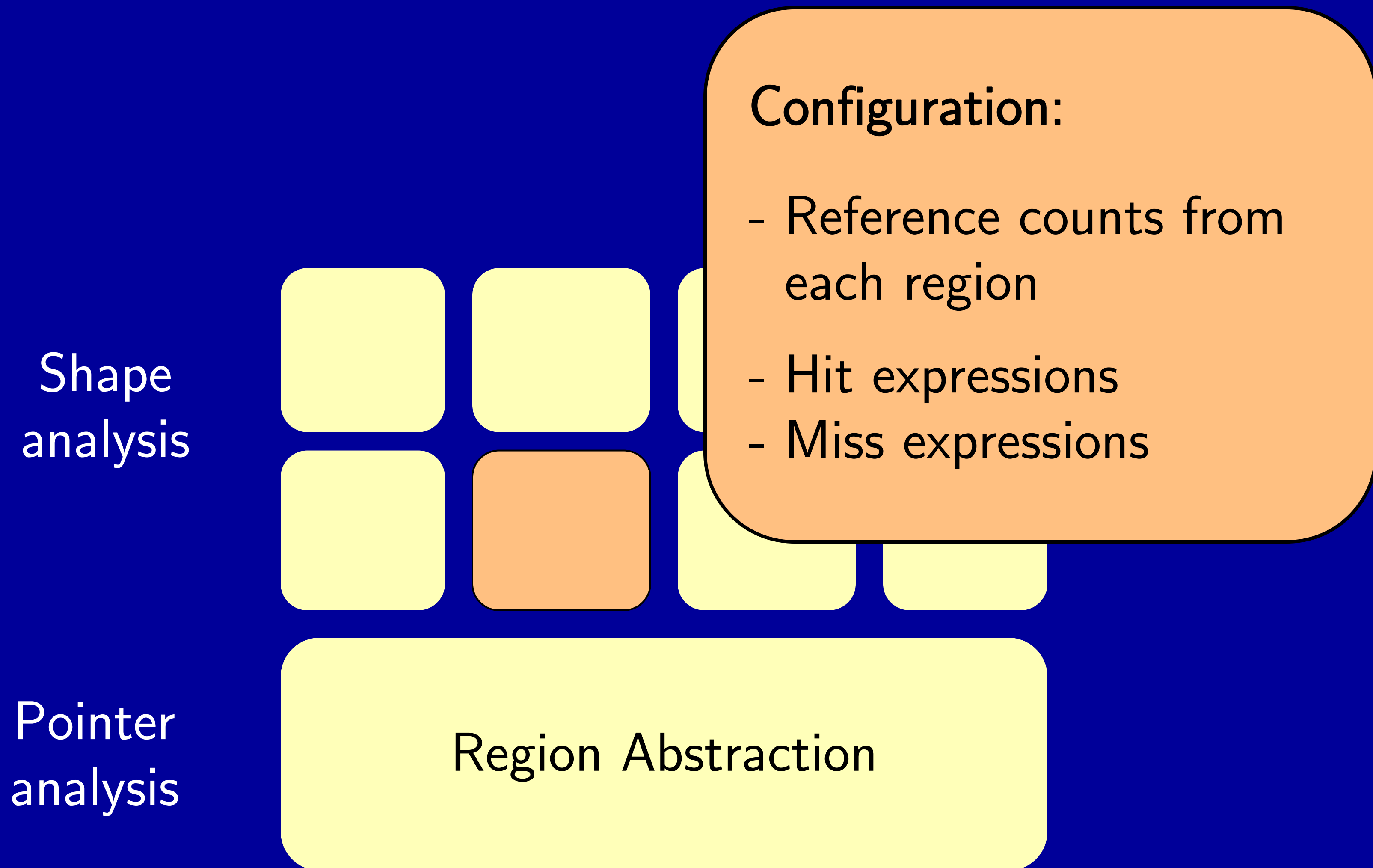
Configurations



Configuration:

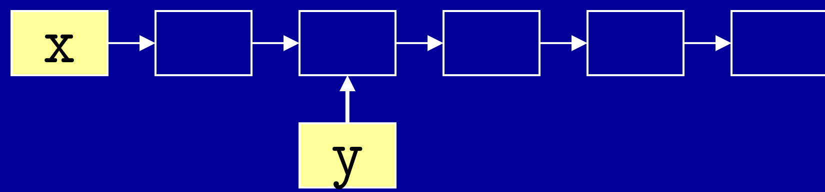
- Talk about one location: the “tracked location”
- No knowledge about other locations

Configurations

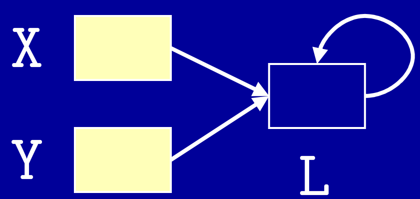


Example Abstraction

Concrete Memory:



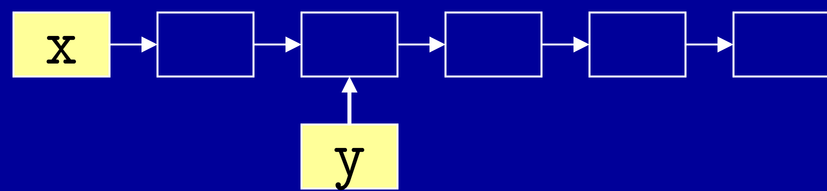
Region Abstraction



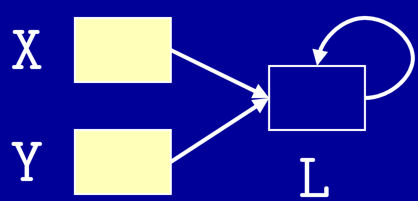
Shape Abstraction

Example Abstraction

Concrete Memory:



Region Abstraction



Shape Abstraction

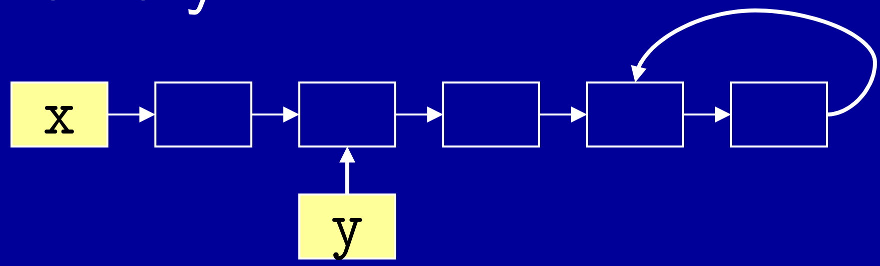
$(X^1, \{x\}, \emptyset)$

$(L^1Y^1, \{x\text{->}n, y\}, \emptyset)$

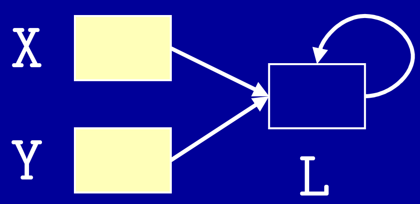
$(L^1, \emptyset, \{x\text{->}n\})$

Cyclic Structures

Concrete Memory:



Region Abstraction



Shape Abstraction

$(X^1,\ \{x\},\ \emptyset)$

$(L^1Y^1,\ \{x\text{->}n, y\},\ \emptyset)$

$(L^1,\ \emptyset,\ \{x\text{->}n\})$

$(L^2,\ \emptyset,\ \{x\text{->}n\})$

Analysis Example: List Reversal

```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {
     t = x->n;
     x->n = y;
     y = x;
     x = t;
   }
   return y;
}
```

Given acyclic list `x`:
is returned list `y` acyclic?

List Reversal

- Region abstraction:

$$\begin{tikzcd} \mathtt{X} \arrow[dr] & \\ \mathtt{Y} \arrow[r] & \mathtt{L} \arrow[loop right] \\ \mathtt{T} \arrow[ur] & \end{tikzcd}$$

- Acyclic list $\mathtt{x}$, two configurations:
  - $(\mathtt{X}^1, \{\mathtt{x}\}, \emptyset)$ describes list head
  - $(\mathtt{L}^1, \emptyset, \emptyset)$ describes tail

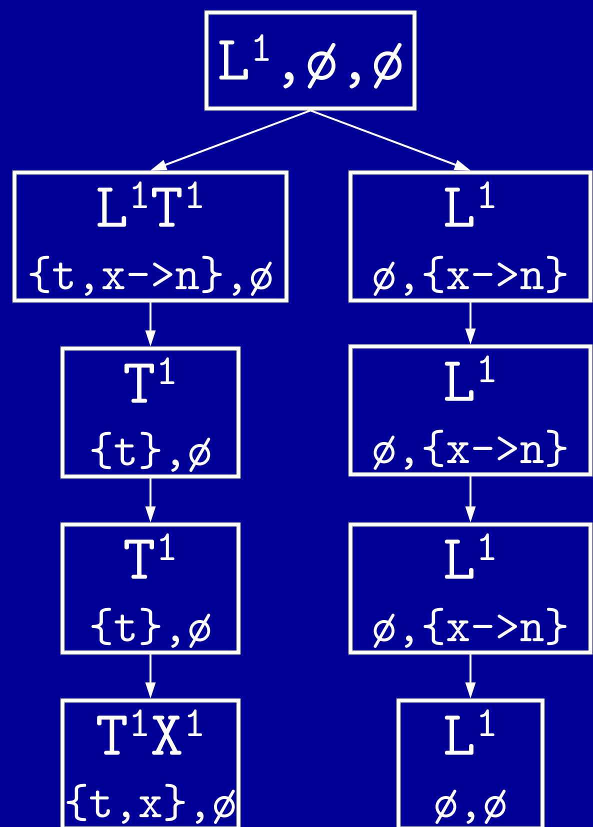
Loop Body Analysis

```
t = x->n;

x->n = y;

y = x;

x = t;
```

$$\begin{array}{c} \boxed{X^1,\{x\},\emptyset} \\ \downarrow \\ \boxed{X^1,\{x\},\emptyset} \\ \downarrow \\ \boxed{X^1,\{x\},\emptyset} \\ \downarrow \\ \boxed{X^1Y^1,\{x,y\},\emptyset} \\ \downarrow \\ \boxed{Y^1,\{y\},\emptyset} \end{array}$$

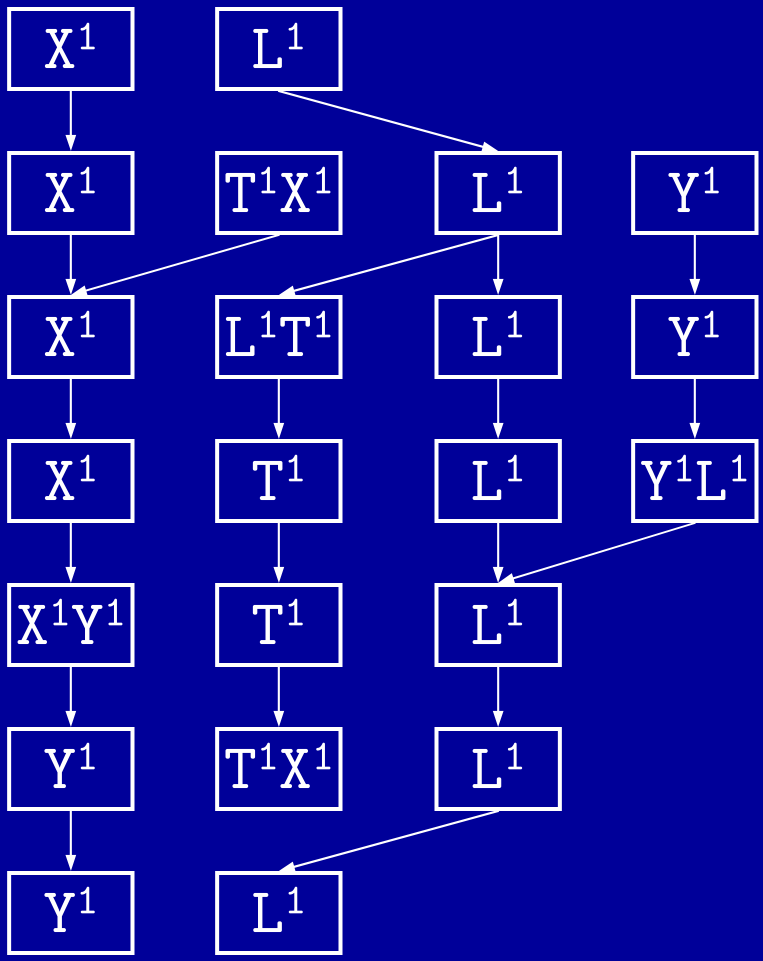
Loop Body Analysis

```
t = x->n;

x->n = y;

y = x;

x = t;
```



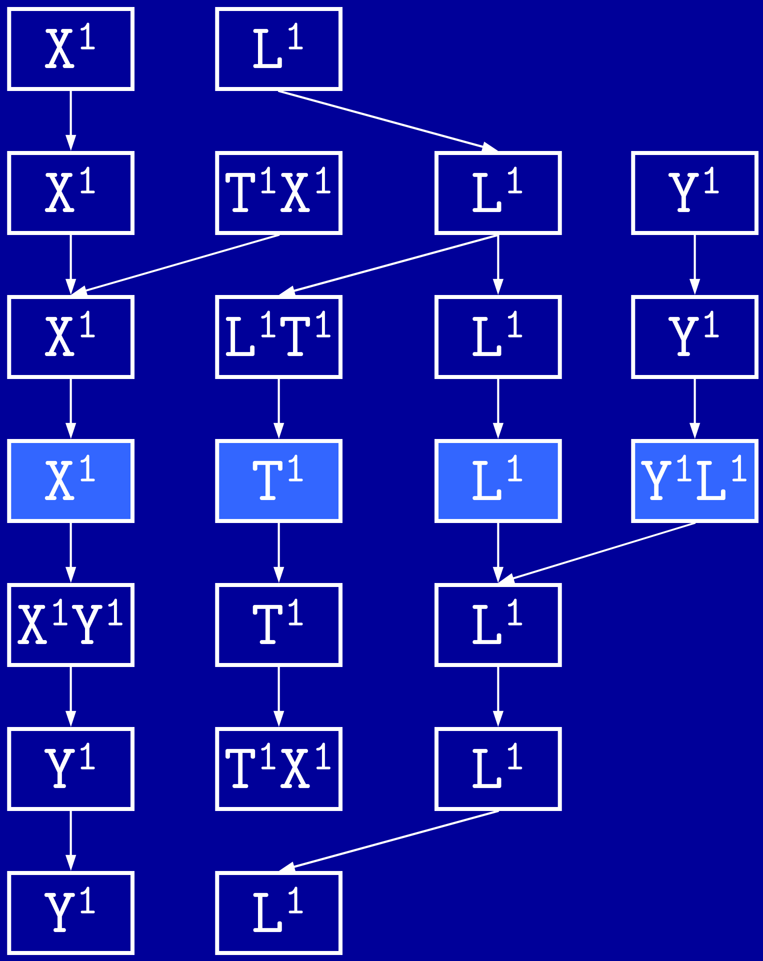
Analysis Result

```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {

     t = x->next;

     x->next = y;

     y = x;

     x = t;
   }
   return y;
}
```



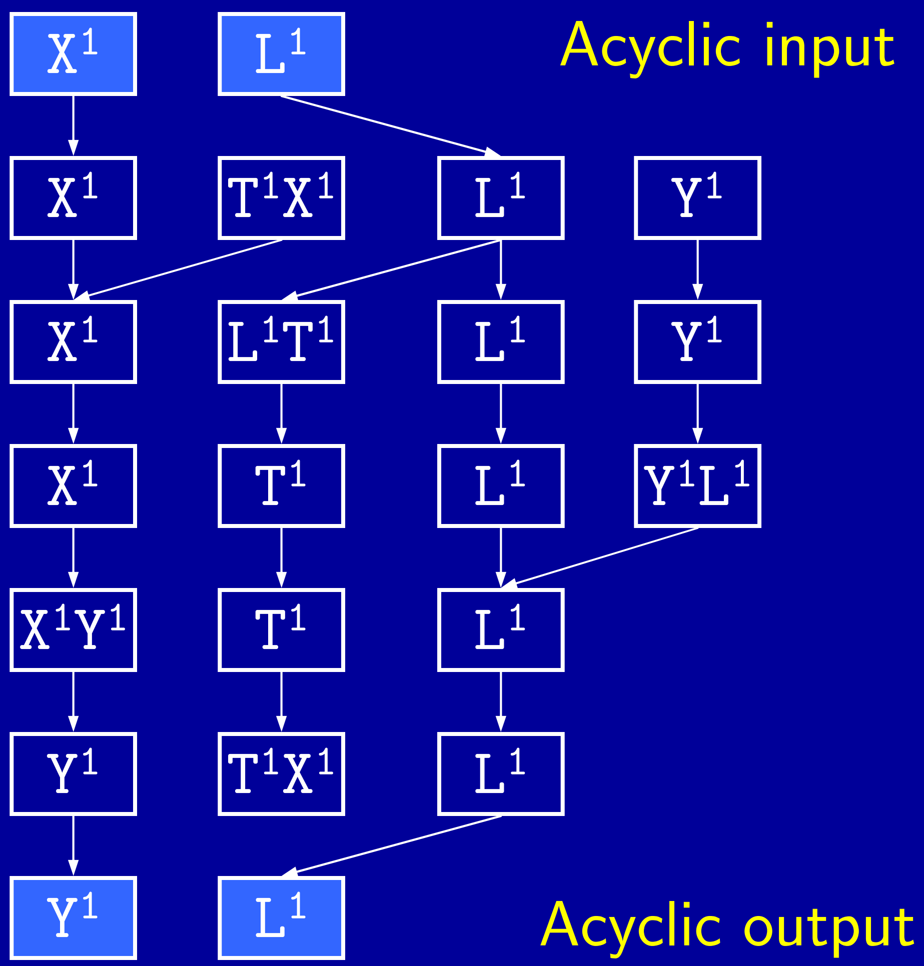
Analysis Result

```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {

     t = x->next;

     x->next = y;

     y = x;

     x = t;
   }
   return y;
}
```

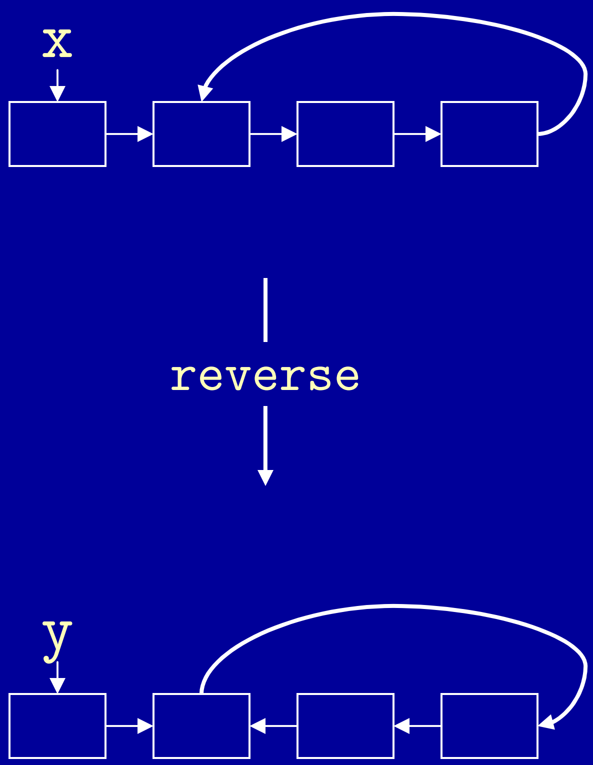


Property Verified

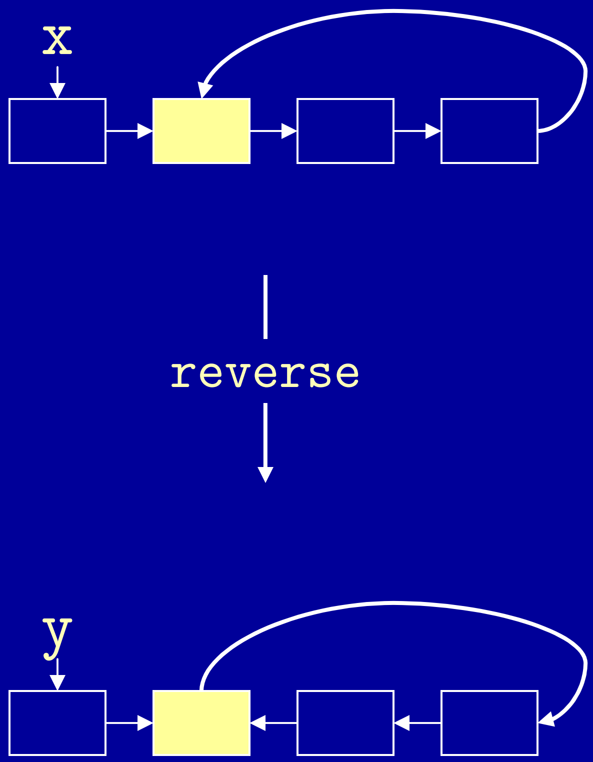
```
List *reverse(List *x) {
   List *t, *y;
   y = NULL;
   while (x != NULL) {

     t = x->next;

     x->next = y;

     y = x;

     x = t;
   }
   return y;
}
```



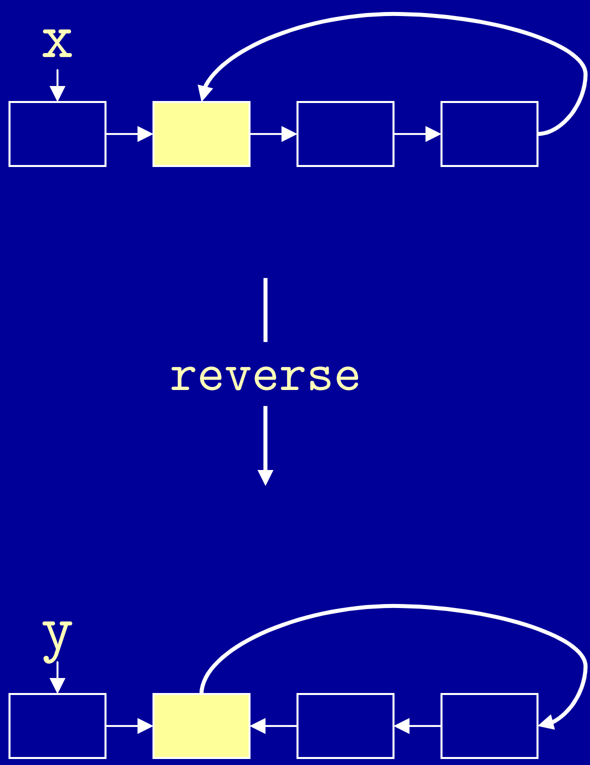
Cyclic Input



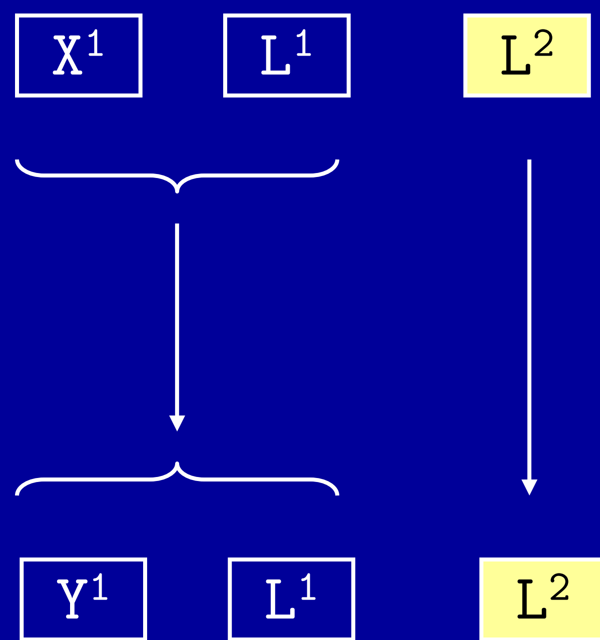
Cyclic Input



Cyclic Input



Analysis:



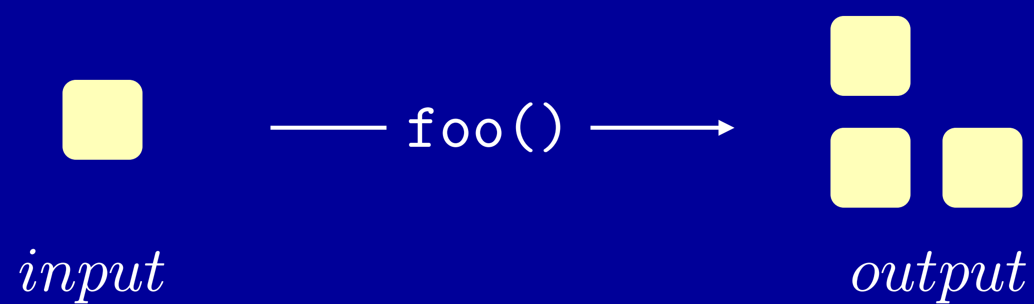
Analysis Algorithm

- Phase 1: Pointer Analysis
  - Flow-insensitive, unification-based
  - Context-sensitive

- Phase 2: Shape Analysis
  - Intra and inter-procedural
  - Flow-sensitive, context-sensitive
  - Granularity of configurations

Inter-Procedural Shape Analysis

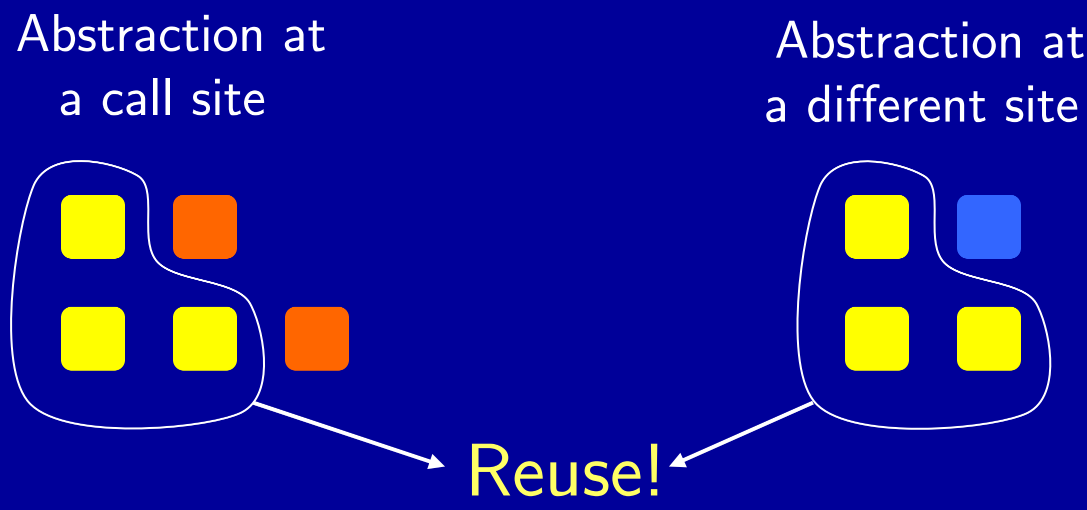
- Context-sensitive analysis
- Summary input = a configuration
- Summary output = set of configurations that correspond to the input



- Tag configurations with the input they originated from
  - Output = retrieve configurations with the desired tag

Inter-Procedural Shape Analysis

- Efficient: reuse previous analyses of functions
  - Match individual configurations!
    - Not entire heap abstractions
  - Works even if there is only partial redundancy



Detecting Memory Errors

- For languages with explicit de-allocation
  - `free(`e`)` de-allocates cell referenced by e
- Extend configurations with one bit:
  has the tracked cell been de-allocated?
  - `malloc()` sets bit to false
  - `free()` sets bit to true
  - Keep tracking cells even after de-allocation

Reference counts
Hit expressions
Miss expressions

Freed flag

Detecting Memory Errors

- Dereference $*e$ may be unsafe if:
  - Expression e may reference the tracked locations
  - And tracked location is marked as de-allocated
  - Catches double frees: `free`(e) checked as $*e$
- A potential memory leak occurs if:
  - The tracked location has all reference counts zero
  - And not marked as de-allocated
  - Allocated in the current function

Implementation

- Implementation for C programs in SUIF
- Singly linked lists
  - Handles standard list manipulations:

    `insert`, `append`, `swap`, `reverse`, `quicksort`, `insertionsort`.
- Doubly linked lists
  - Does not identify structural invariants

Implementation

- Tested tool on three larger programs:

	SSH	SSL	binutils
Lines	18.6 KLOC	25.6 KLOC	24.4 KLOC
Reported Bugs	26 10	13 4	58 24
Total Time Points-to Shape	45 sec 16 sec 29 sec	22 sec 13 sec 9 sec	44 sec 6 sec 38 sec

Comparison

Analysis/Year	Implemented?	Inter-Procedural?	size(LOC), time(sec)
Jones, Muchnick / 1979	no		
Chase, Wegman,Zadeck / 1990	no		
Ghiya, Hendren /1996	YES	YES	3.3 K, n/a
Sagiv, Reps,Wilhelm /1996	no		
Sagiv, Reps,Wilhelm /1999	no		
Lev-Ami, Reps, Sagiv, Wilhelm/2000	YES	no	<30, 295
Dor, Rodeh, Sagiv/2000	YES	no	<30, 2
Rinetzky, Sagiv /2001	YES	YES	<30, 1028
Jeannet, Loginov, Reps, Sagiv /2004	YES	YES	<30, 222
Yahav, Ramalingam /2004	YES	YES	1.3K, 12881
Hackett/Rugina /2005	YES	YES	25 K, 45

Summary

- Shape analysis:
  - Needed for precise analysis of heap structures
  - Necessarily flow-sensitive
  - Not scalable until recently