CS412

Introduction to Compilers

Radu Rugina

Lecture 38: Shape Analysis
05 May 06

## Static Heap Analysis

- Current state-of-the-art in compilers:
  - Error checking limited to type-checking
  - No support for checking for leaks, dangling refs, double frees
- Verification tools:
  - E.g., theorem-provers, model-checkrs
  - Precise, sound verification via shape analysis
  - Expensive, limited to verifying small programs

## Shape Analysis

- Shape analysis = static analysis of heap data structures
- Can automatically determine that a program builds an maintains an unshared and cycle-free heap structure
  - E.g., "your program builds a tree, not a graph"
  - Or "At this point, the list is acyclic & unshared"


## Why Is It Important?

- Many potential applications:
  - Verification: check that the program indeed builds the structure it is supposed to
    - Easier to reason about trees than about graphs
  - Error detection: find memory errors
  - Optimizations: automatic parallelization for tree structures
  - Memory management: enable deallocation of objects at compile-time

## Why Is It Difficult?

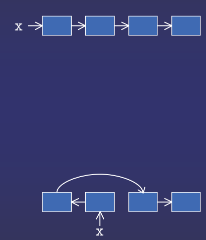
- Reason 1: Unbounded number of heap cells
  - No lexical scopes to bound their lifetimes
  - Think "unbounded numbers of global variables"
- Reason 2: Destructive updates
  - Structure invariants temporarily invalidated
- Reason 3: Inter-procedural interactions, recursion
  - Inter-procedural reasoning is difficult and expensive
  - Main scalability obstacle

## Reference Count Invariants

- Express heap shapes using reference counting:
  - Heap reference count $\leq 1$
  - Distinguish trees from graphs; detect cycles or sharing
  - Invariant indicates no aliasing

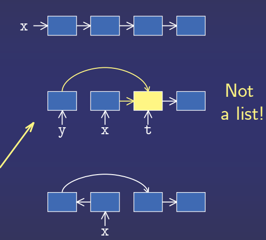
# Maintaining Invariants

```
List *swap(List *x) {
   List *y, *t;
   if (x != NULL &&
       x->next != NULL) {
     y = x;
     x = y->n;
     t = x->n;
     y->n = t;
     x->n = y;
   }
   return x;
}
```




# Breaking Invariants

```
List *swap(List *x) {
   List *y, *t;
   if (x != NULL &&
       x->next != NULL) {
     y = x;
     x = y->n;
     t = x->n;
     y->n = t;
     x->n = y;
   }
   return x;
}
```




# How Shape Analysis Works

- Shape analysis is inherently a dataflow analysis
- Come up with a finite heap abstraction
- Analyze each statement with that abstraction

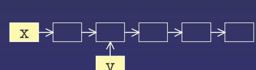
Heap Abstraction Before | `y->n = t` | Heap Abstraction After

# The Dataflow Facts

- Abstract each heap cell separately
  - Local reasoning: analyze heap cells one at a time
  - Easier to build efficient analysis algorithms
- Configuration: (RC, H, M)
  - Abstraction of one heap cell
  - RC = reference counts from variables and fields
  - H = set of expressions that reference the cell (hit)
  - M = expressions that don't reference the cell (miss)
  - Entire heap = finite set of independent configurations

# Example

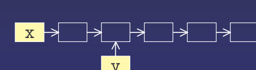
A concrete list:




```
struct list {
  int d;
  struct list *n;
} *x, *y;
```

# Example

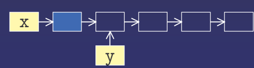
A concrete list:




Abstraction:

## Example

A concrete list:

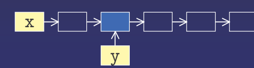



Abstraction:

$(x^1,\ \emptyset,\ \emptyset)$

## Example

A concrete list:



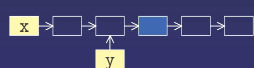

Abstraction:

$(x^1,\ \emptyset,\ \emptyset)$

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

## Example

A concrete list:




Abstraction:

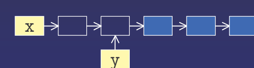
$(x^1,\ \emptyset,\ \emptyset)$

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

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

## Example

A concrete list:




Abstraction:

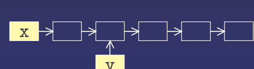
$(x^1,\ \emptyset,\ \emptyset)$

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

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

## Example

A concrete list:




Abstraction:

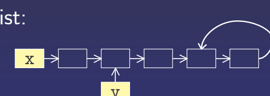
$(x^1,\ \emptyset,\ \emptyset)$

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

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

## Example

A cyclic list:




Abstraction:

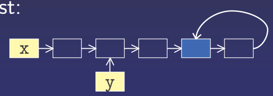
$(x^1,\ \emptyset,\ \emptyset)$

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

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

# Example

A cyclic list:




Abstraction:

$(x^1,\ \emptyset,\ \emptyset)$

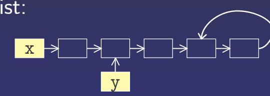
$(y^1n^1,\ \{x\text{->}n\},\ \emptyset)$

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

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

# Example

A cyclic list:




Abstraction:

$(x^1,\ \emptyset,\ \emptyset)$

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

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

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

# Analyzing 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;
}
```

Verify that:
returned list y is acyclic
if input list x is acyclic

List x is acyclic:
$(x^1,\ \emptyset,\ \emptyset)$
$(n^1,\ \emptyset,\ \emptyset)$

# Loop Body Analysis

$\boxed{x^1,\ \emptyset,\emptyset}$

```
t = x->n;

x->n = y;

y = x;

x = t;
```

# Loop Body Analysis

➡ `t = x->n;`

`x->n = y;`

`y = x;`

`x = t;`

$\boxed{x^1,\ \emptyset,\emptyset}$

Local reasoning:

No references from n,t

State remains unchanged

# Loop Body Analysis

➡ `t = x->n;`

`x->n = y;`

`y = x;`

`x = t;`

$\boxed{x^1,\ \emptyset,\emptyset}$

$\downarrow$

$\boxed{x^1,\ \emptyset,\emptyset}$

Local reasoning:

x->n in region L
t in region T

No references from L,T

State remains unchanged

## Loop Body Analysis

```
t = x->n;
x->n = y;   ➡
y = x;
x = t;
```

$x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$

Local reasoning:

No references from y,n

State remains unchanged

## Loop Body Analysis

```
t = x->n;
x->n = y;
y = x;      ➡
x = t;
```

$x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$ → $x^1y^1, \emptyset, \emptyset$

Local reasoning:

x hits cell
y misses cell

Add reference from y

## Loop Body Analysis

```
t = x->n;
x->n = y;
y = x;
x = t;      ➡
```

$x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$ → $x^1y^1, \emptyset, \emptyset$ → $y^1, \emptyset, \emptyset$

Local reasoning:

t misses cell
x hits cell

Remove reference from x

## Loop Body Analysis

```
t = x->n;
x->n = y;
y = x;
x = t;
```

$x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$ → $x^1, \emptyset, \emptyset$ → $x^1y^1, \emptyset, \emptyset$ → $y^1, \emptyset, \emptyset$

## Loop Body Analysis

$n^1, \emptyset, \emptyset$

```
t = x->n;
x->n = y;
y = x;
x = t;
```

## Loop Body Analysis

$n^1, \emptyset, \emptyset$

```
t = x->n;   ➡
x->n = y;
y = x;
x = t;
```

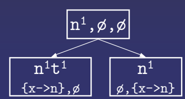
Local reasoning:

t misses cell

Don't know if x->n hits or misses cell…

# Loop Body Analysis

```
➡ t = x->n;
  x->n = y;
  y = x;
  x = t;
```




Local reasoning:

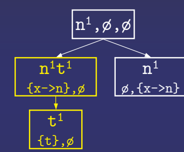
t misses cell

Don't know if x->n hits or misses cell…

Bifurcate and analyze each case!

# Loop Body Analysis

```
  t = x->n;
➡ x->n = y;
  y = x;
  x = t;
```



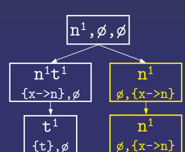

Local reasoning:

y misses cell
x->n hits cell

Remove reference from n

# Loop Body Analysis

```
  t = x->n;
➡ x->n = y;
  y = x;
  x = t;
```



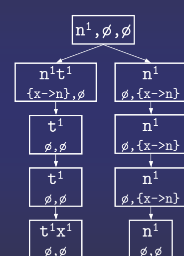

Local reasoning:

y misses cell
x->n misses cell

State remains unchanged

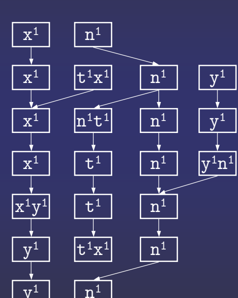
# Loop Body Analysis

```
t = x->n;
x->n = y;
y = x;
x = t;
```



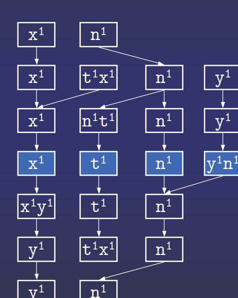

# Analysis Result

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

    t = x->next;

    x->next = y;

    y = x;

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



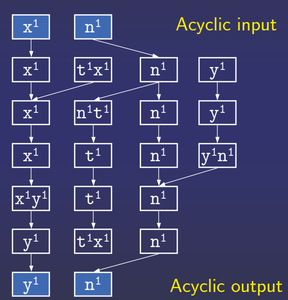

# Analysis Result

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

    t = x->next;

    x->next = y;

    y = x;

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

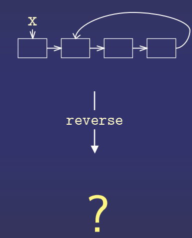


## Property Verified

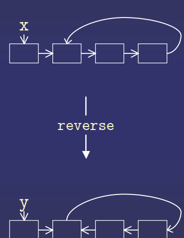
```
List *reverse(List *x) {
    List *t, *y;
    y = NULL;
    while (x != NULL) {

        t = x->next;

        x->next = y;

        y = x;

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



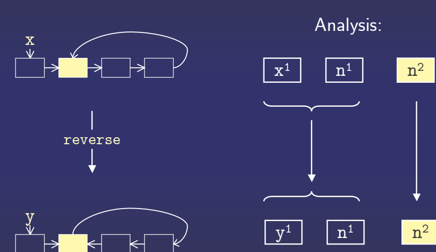

## Cyclic Input




## Cyclic Input




## Cyclic Input

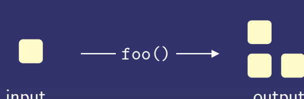



## Initialization

- Inject a new configuration at each `malloc`:
  - After: `x = malloc(...)`
  - Create: $(x^1, \emptyset, \emptyset)$
- Then track the new configuration
  - Bifurcate when necessary

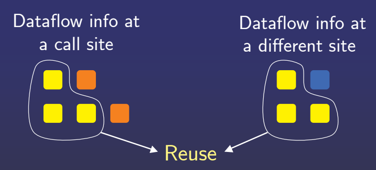
## Inter-Procedural Analysis

- Context-sensitive analysis
- Procedure summaries: map each input configuration set of corresponding output configurations

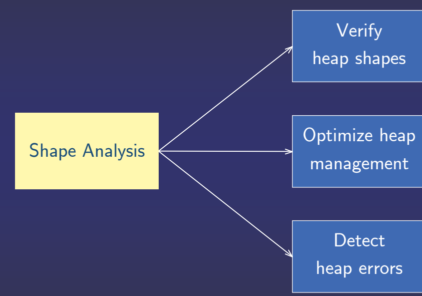


## Inter-Procedural Analysis

- Efficient: reuse previous analyses of functions
  - Match individual configurations, not entire heap abstractions
  - Works even if there is only partial overlap




## Applications




## Application 1

- Check that acyclic shape is maintained
- Singly linked lists
  - Handles standard list manipulations: `insert, append, swap, reverse, insertion_sort, quicksort`
- Doubly linked lists
  - Does not identify structural invariants

Verify heap shapes | Optimize heap management | Detect heap errors

## Application 2

- For garbage collected languages (e.g., Java)
- Static reclamation of heap objects
  - Compile-time program transformation
  - Insert "free" statements
  - Desirable for real-time and embedded systems
- Implementation:
  - Reduces memory watermark by 50%
  - Low run-time overhead (2% on average)

Verify heap shapes | Optimize heap management | Detect heap errors

## Application 3

- For languages with explicit de-allocation (e.g., C)
- Extend configurations: (RC, H, M, F)
  - F = "freed" flag
- Dangling pointer access $*e$ if:
  - $e$ may hit a configuration with F = true
  - Same for double free's
- Memory leak if:
  - A configuration has all reference counts zero
  - And F flag is false

Verify heap shapes | Optimize heap management | Detect heap errors

## Application 3

- Bug finding tool
- Analyzed `ssh, ssl, binutils`

| | |
|---|---|
| Size per app.: | 18 to 25 KLOC |
| Analysis time: | 1.7 KLOC/sec |
| Alloc sites: | 197 (7% cut off) |
| Warnings: | 96 |
| Leaks found: | 38 |

Verify heap shapes | Optimize heap management | Detect heap errors

# Summary

- New approach to static heap analysis
  - Local reasoning about heap cells
- Applications:
  - Verification of heap shapes
  - Analyze manipulations of recursive structures
  - Finding heap-related bugs in larger programs
  - Memory management transformations