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

Shape analysis

Shape Abstraction

Pointer analysis

Region Abstraction
New Memory Abstraction

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

[Diagram showing shape and pointer analysis leading to region abstraction]
New Memory Abstraction

• Decompose memory abstraction
  – Build shape abstraction using \textit{independent} pieces
Configuration:
- Talk about one location: the “tracked location”
- No knowledge about other locations
Configurations

- Reference counts from each region
- Hit expressions
- Miss expressions
Example Abstraction

Concrete Memory:

Region Abstraction

Shape Abstraction
Example Abstraction

Concrete Memory:

Region Abstraction

Shape Abstraction

(X₁, \{x\}, \emptyset)

(L¹Y¹, \{x→n,y\}, \emptyset)

(L¹, \emptyset, \{x→n\})
Cyclic Structures

Concrete Memory:

Region Abstraction

Shape Abstraction

(L^1, \phi, \{x\rightarrow n\})

(L^2, \phi, \{x\rightarrow 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;
}
List Reversal

- Region abstraction:
  - $(X^1, \{x\}, \emptyset)$ describes list head
  - $(L^1, \emptyset, \emptyset)$ describes tail
t = x->n;

x->n = y;

y = x;

x = t;
Loop Body Analysis

t = x->n;

x->n = y;

y = x;

x = t;
List *reverse(List *x) {
    List *t, *y;
    y = NULL;
    while (x != NULL) {
        t = x->next;
        x->next = y;
        x->next = y;
        y = x;
        x = t;
    }
    return y;
}
List *reverse(List *x) {
    List *t, *y;
    y = NULL;
    while (x != NULL) {
        t = x->next;
        x->next = y;
        x->next = y;
        y = x;
        y = x;
        x = t;
    }
    return y;
}
List *reverse(List *x) {
    List *t, *y;
y = NULL;
    while (x != NULL) {
        t = x->next;
        x->next = y;
        x->next = y;
        y = x;
        y = x;
        x = t;
    }
    return y;
}
Cyclic Input

\[
x \xrightarrow{\text{reverse}} y
\]
Cyclic Input
Cyclic Input

Analysis:

x

reverse

y

X^1  L^1  L^2

Y^1  L^1  L^2
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

![Diagram showing input and output with `foo()`]

- 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

Abstraction at a call site

Abstraction at a different site

Reuse!
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:

<table>
<thead>
<tr>
<th></th>
<th>SSH</th>
<th>SSL</th>
<th>binutils</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines</td>
<td>18.6 KLOC</td>
<td>25.6 KLOC</td>
<td>24.4 KLOC</td>
</tr>
<tr>
<td>Reported Bugs</td>
<td>26</td>
<td>13</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>4</td>
<td>24</td>
</tr>
<tr>
<td>Total Time</td>
<td>45 sec</td>
<td>22 sec</td>
<td>44 sec</td>
</tr>
<tr>
<td>Points-to Shape</td>
<td>16 sec</td>
<td>13 sec</td>
<td>6 sec</td>
</tr>
<tr>
<td></td>
<td>29 sec</td>
<td>9 sec</td>
<td>38 sec</td>
</tr>
</tbody>
</table>
## Comparison

<table>
<thead>
<tr>
<th>Analysis/Year</th>
<th>Implemented?</th>
<th>Inter-Procedural?</th>
<th>size(LOC), time(sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jones, Muchnick / 1979</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chase, Wegman, Zadeck / 1990</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Ghiya, Hendren / 1996</td>
<td>YES</td>
<td>YES</td>
<td>3.3 K, n/a</td>
</tr>
<tr>
<td>Sagiv, Reps, Wilhelm / 1996</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sagiv, Reps, Wilhelm / 1999</td>
<td>no</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lev-Ami, Reps, Sagiv, Wilhelm / 2000</td>
<td>YES</td>
<td>no</td>
<td>&lt; 30, 295</td>
</tr>
<tr>
<td>Dor, Rodeh, Sagiv / 2000</td>
<td>YES</td>
<td>no</td>
<td>&lt; 30, 2128</td>
</tr>
<tr>
<td>Rinetzky, Sagiv / 2001</td>
<td>YES</td>
<td>YES</td>
<td>&lt; 30, 1028</td>
</tr>
<tr>
<td>Jeannet, Loginov, Reps, Sagiv / 2004</td>
<td>YES</td>
<td>YES</td>
<td>&lt; 30, 222</td>
</tr>
<tr>
<td>Yahav, Ramalingam / 2004</td>
<td>YES</td>
<td>YES</td>
<td>1.3K, 12881</td>
</tr>
<tr>
<td>Hackett/Rugina / 2005</td>
<td>YES</td>
<td>YES</td>
<td>25 K, 45</td>
</tr>
</tbody>
</table>
Summary

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