Abstracting Runtime Heaps for Program Understanding

Mark Marron, Member, IEEE, Cesar Sanchez, Member, IEEE, Zhendong Su, Member, IEEE, and Manuel Fahndrich

Presented By: Sameeulla Siddique
Mahesh Chandrappa
Introduction

- Current programming environments provide support for visualizing and debugging code

- Inspecting high-level data structure at runtime is not well supported

- Visualizing entire runtime heap graphs is a significant problem. Solution – get abstract heap graphs
What is an abstract heap??

- Get an abstract domain for runtime heap graphs
- Captures fundamental properties of data structures – shape, connectivity and sharing
- Small enough to visualize and precise enough to capture information useful for debugging and memory profiling
- Computed from a concrete heap and merged/compared with other abstract heap graphs
Example

Fig. 1. A concrete heap dump from an arithmetic expression evaluation program and the corresponding abstract heap visualization produced by the HeapDbg tool.
Nodes of abstract heap

- Node representing all interior recursive objects (Add, Mult, Sub)
- Node representing two Var objects
- Node representing the two Const objects
Basic conventions

- Shows possible sharing using wide orange colored edges
- Normal edges indicate non-sharing pointers
- Pointer nullity (explained later) is shown via full or dashed lines
- Nodes representing multiple objects are shaded (in silver)
- Node representing single objects have a white background
- Size information is annotated to the type label
Overview

- Convert large concrete runtime heaps into compact abstract heaps in $O((Ob+Pt) \times \log(Ob))$
  - $Ob$–number of objects
  - $Pt$–number of pointers

- Properties of abstract heap also hold in the corresponding concrete heaps

- Properties – summarize recursive data structures, assign shape information, injectivity of fields and nullity information

- Later–construction of heap memory profiler and analysis tool
Concrete Heaps

- Instance of the runtime state (environment and store) is referred as concrete heap
- Formally, it’s a labeled directed graph \((\text{root}, \text{null}, \text{Ob}, \text{Pt}, \text{Ty})\)
  - \(\text{Ob}\) – set of heap Objects (forms the nodes)
  - \(\text{Pt}\) – edges which correspond to Pointers
  - \(\text{Ty}\) – map that assigns a Type to each object \((\text{Ty}: \text{Ob} \to \text{Type})\)
- Notation
  - \(o_1 \xrightarrow{p} o_2\)
    - Object \(o_1\) refers to \(o_2\) via pointer label \(p\)
- Region of memory \(C \subseteq \text{Ob} \setminus \{\text{null}, \text{root}\}\)
Concrete Heap Properties

Properties of concrete heap to create abstract heap graph

- **Type** – Union of all types associated with a region $C$
  - $\{\text{Ty}(o) \mid o \in C\}$

- **Cardinality** – Number of objects in the region $C$

- **Nullity** – A pointer $o1 \rightarrow o2$ is null pointer if $o2 = \text{null}$ and vice versa
Concrete Heap Properties (Continued..)

- **Injectivity** – Given two regions $C1$ and $C2$, pointers labeled $p$ from two *distinct* objects $o1$ and $o2$ pointing to *distinct* objects $t1$ and $t2$ are said to be injective.

- **Shape** – Regions of memory $C$ are characterized by shape. Two predicates combined with label sets are used to describe a range of useful heaps.
  - $tree(C,L)$ holds if the subgraph is acyclic and has no cross edges.
  - $any(C,L)$ is simply true for any graph.
Heap Graph Abstraction

- Abstract heap graphs are tuples
  - $(\text{root}, \text{null}, \text{Ob}', \text{Pt}', \text{Ty}', \text{Cd}', \text{Ij}', \text{Sh}')$ where
    - $\text{Ob}'$ is a set of abstract nodes
    - $\text{Pt}'$ is a set of graph edges, each of which abstracts a set of pointers
    - $\text{Ty}'$ is the map which abstracts nodes to the set of types of the concrete nodes
    - $\text{Cd}'$ is the cardinality (number of objects) of a region
    - $\text{Ij}'$ expresses if the set of pointers represented by an abstract edge are injective
    - $\text{Sh}'$ is the set of tuples indicating the shape $s$ of a region
    - Two distinguished nodes from $\text{Ob}'$— roots and $\text{null}$— are used to represent the root node and null pointer
Computing the Abstraction

There are three transformation phases to compute an abstract graph from a given concrete heap

- Recursive data structures are identified and collapsed
- Nodes that represent objects in the same logical heap region are merged based on equivalent edges originating from same abstract node
- Abstract properties like cardinality, injectivity and shape are computed for the abstract edges and nodes
Definition 1 (Same recursive data structure objects)

Two distinct objects o1, o2 are part of the same recursive data structure if there is a reference o1→o2 in the heap and the types of the two objects are part of the same recursive type definition Ty(o1) ~ Ty(o2)

Definition 2 (Equivalent on abstract predecessors)

Given two pointers o1→o2 and o1’→o2’ we say that their target nodes are equivalent whenever: The labels agree l=l’ and the target nodes have some types in common
Steps in abstracting computation

(a) Initial Partition.
(b) Merge Recursive Data Structures.
(c) Merge Predecessors.
The final map for the objects is

\[
\mu^{-1} = \begin{cases} 
    n_1 & \mapsto \{o_1, o_2, o_4, o_5\} \\
    n_3 & \mapsto \{o_3, o_6\} \\
    n_7 & \mapsto \{o_7, o_8\} \\
    n_9 & \mapsto \{o_9\} 
\end{cases}
\]

- Type, cardinality and nullity
- Injectivity – abstract edge representing the set \(n_1 \rightarrow n_7\) is not injective. Edge representing the set \(n_1 \rightarrow n_3\) is injective
- Shape – As there are no cross or back edges, the layout for this is \(tree\{l,r\}\)
Merge and Comparison Operations

- **Compare**
  - First determine the structural equality of the abstract graphs $g_1$ and $g_2$ by computing an isomorphism, followed by an implication check that all abstract edge and node properties in $g_2$ cover the equivalent node and edge properties of $g_1$

- **Merge**
  - Produce the union of the two graphs by taking the union of node and edge sets from the graphs
  - Merge the global root objects
  - Use Definition 1 and Definition 2 to merge nodes and edges, until no more changes are occurring
Goals:
1) the cost of computing abstract heaps for real programs
2) the feasibility of visualizing the abstract graphs, and
3) whether the abstract graphs produced are precise enough for understanding the program’s behavior and to identify and correct various defects.
Implementation

- Authors implemented the algorithms for computing and manipulating the abstract heap graph.
- They used DGML (XML based format for graphs) graph format and its associated viewer was Visual Studio 2010.
- Advantages of using DGML and viewer
  - flags nodes of heap with poor memory utilization.
  - highlight edges that object overhead.
  - apply heat-color map to nodes based on the amount of memory the objects they represent are using.
Implementation: Memory Profiling Tool

- Authors used memory profiling to achieve their goals.

What it does?
- To monitor memory use and compute heap snapshot with associated abstraction at points where high memory is used during execution.
- Authors implemented simple post-processing operations on the abstract graphs which allow the DGML viewer to flag nodes (regions) of the heap that display common types of poor memory utilization.
- The properties we identify are
  - percentage of memory used,
  - small object identification,
  - sparse container or small containers
Implementation: Memory Profiling Tool

- **Percentage of memory**
  uses a heat map, coloring any nodes that contain more than 5, 15, or 25 percent of the heap

- **Small object identification**
  highlights any nodes where the object overheads (assumed to be 4 bytes per object) are more than half the actual data stored in the objects

- **Sparse container or small containers**
  The poor collection utilization property highlights nodes that represent regions which are containers and, for which all the containers are either very small (contain three or fewer elements) or are more than half empty (over half the entries are null)
Ray Tracer : Extended Case Study

- Ray Tracer program renders a user-provided scene. Running this program in our memory profiler, we obtain snapshots of an abstract heap (~168K objects ~4MB)

- Root nodes of the heap are the argument variables of the method

- ‘this’ variable refers to a scene object

- Octree field of ‘this’ variable represents a space decomposition tree structure, which is also referred by multiple dominated regions

- Larger nodes with chevrons are dominator reduced nodes that represents multiple dominated regions
Node #19 represents a large amount of memory, 107K objects representing nearly half of the total live heap.

By expanding the node #19 we get the internal structure of the dominator reduced node.

This reveals node ($48), abstracting a region of 18K Face objects, node ($23), abstracting a region of 18K Point[], and node ($49), abstracting a region of 72K Point objects.
Industrial Experience

- Have a small number of industrial users.
- Even though time to run the profiler is long, it solves the major problems.
- Useful to manually explore the graph to see what the data structures in the programs look like and if this matches their intuition.
- Memory Inefficiency and reachability items are the two major problems solved using this technique.
# Computational Costs

## TABLE 1
AbsNodes and Reduced Are the Number of Nodes in the Largest Abstract and Dominator Reduced Graphs

<table>
<thead>
<tr>
<th>Bench</th>
<th>Objects</th>
<th>AbsNode</th>
<th>Reduced</th>
<th>AbsTime</th>
<th>EqTime</th>
<th>MergeTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>raytracer</td>
<td>~168K</td>
<td>48</td>
<td>21</td>
<td>1.37s</td>
<td>0.04s</td>
<td>0.11s</td>
</tr>
<tr>
<td>antlr</td>
<td>~12K</td>
<td>606</td>
<td>201</td>
<td>0.41s</td>
<td>0.03s</td>
<td>0.11s</td>
</tr>
<tr>
<td>chart</td>
<td>~189K</td>
<td>198</td>
<td>110</td>
<td>3.22s</td>
<td>0.09s</td>
<td>0.21s</td>
</tr>
<tr>
<td>fop</td>
<td>~120K</td>
<td>531</td>
<td>150</td>
<td>2.67s</td>
<td>0.11s</td>
<td>0.41s</td>
</tr>
<tr>
<td>luindex</td>
<td>~2K</td>
<td>87</td>
<td>36</td>
<td>0.50s</td>
<td>0.01s</td>
<td>0.02s</td>
</tr>
<tr>
<td>pmd</td>
<td>~178K</td>
<td>146</td>
<td>28</td>
<td>4.11s</td>
<td>0.09s</td>
<td>0.15s</td>
</tr>
<tr>
<td>xalan</td>
<td>~40K</td>
<td>451</td>
<td>127</td>
<td>2.42s</td>
<td>0.07s</td>
<td>0.17s</td>
</tr>
</tbody>
</table>

*AbsTime is the largest time needed to compute the abstraction of a concrete heap. EqTime and MergeTime are the largest times to compare or merge any pair of abstract heaps.*
The running time scales very closely to the asymptotic complexity of $O(E \times \log (N))$.

Implementation and the construction of a full shadow heap for each snapshot results in a large memory overhead.

For huge programs this memory overhead is a nightmare.

Solution: restructuring eliminates the large cost of creating a full shadow heap and allows the concrete heap to be processed incrementally.
Related Work

Other Works:
Uses descriptive model i.e. they have not considered informations such as injectivity or shape.
Some work uses related idea i.e taking concrete heaps from a program and inferring the types and basic shapes of heap structure only.
The approach to abstraction also differs in some of the work.
This paper: This technique does not only give compact runtime heaps but also provides more control over abstractions.
Conclusion

- A new runtime technique for program understanding, analysis and debugging.
- Compact representation of runtime heap allows effective visualization and navigation.(Heap Abstraction)
- Abstract Heap graph provides interesting information on heap structure, and identifying and correcting memory use related defects.
- Can also be used for thread races, interactive debugging, refactoring for parallelism, interactive debugging, and computing runtime.
References

QUESTIONS?
Thank You !!