Heapviz: Interactive Heap Visualization for Program Understanding and Debugging

Edward E. Aftandilian
eaftan@cs.tufts.edu

Sean Kelley
sean.kelley@tufts.edu

Connor Gramazio
connor.gramazio@tufts.edu

Nathan Ricci
nricci01@cs.tufts.edu

Sara L. Su
sarasu@cs.tufts.edu

Samuel Z. Guyer
sguyer@cs.tufts.edu

Department of Computer Science
Tufts University
http://www.cs.tufts.edu/r/redline

ABSTRACT

Understanding the data structures in a program is crucial to understanding how the program works, or why it doesn't work. Inspecting the code that implements the data structures, however, is an arduous task and often fails to yield insights into the global organization of a program's data. Inspecting the actual contents of the heap solves these problems but presents a significant challenge of its own: finding an effective way to present the enormous number of objects it contains.

In this paper we present *Heapviz*, a tool for visualizing and exploring snapshots of the heap obtained from a running Java program. Unlike existing tools, such as traditional debuggers, Heapviz presents a global view of the program state as a graph, together with powerful interactive capabilities for navigating it. Our tool employs several key techniques that help manage the scale of the data. First, we reduce the size and complexity of the graph by using algorithms inspired by static shape analysis to aggregate the nodes that make up a data structure. Second, we introduce a dominator-based layout scheme that emphasizes hierarchical containment and ownership relations. Finally, the interactive interface allows the user to expand and contract regions of the heap to modulate data structure detail, inspect individual objects and field values, and search for objects based on type or connectivity. By applying Heapviz to both constructed and real-world examples, we show that Heapviz provides programmers with a powerful and intuitive tool for exploring program behavior.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*; E.1 [**Data Structures**]: Graphs and networks, Lists, stacks, and queues, Trees; I.3.3 [**Computer Graphics**]: Picture/Image Generation—*Display algorithms*

General Terms

Design

Keywords

Software visualization, program understanding, debugging, interactive visualization, graphs

SOFTVIS'10, October 25–26, 2010, Salt Lake City, Utah, USA.

1. INTRODUCTION

> *Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.*
> – Fred Brooks [2]

Understanding modern software has become a significant challenge, even for expert programmers. Part of the problem is that today's programs are larger and more complex than their predecessors, in terms of static code base (lines of code), runtime behavior, and memory footprint. Another problem is that modern applications, such as web-based e-commerce and cloud computing platforms, are constructed by assembling reusable software components, ranging from simple container classes to huge middleware frameworks. In many cases, these components are instantiated dynamically and wired together using techniques such as reflection or bytecode rewriting. These features make it very challenging for any one programmer to obtain a global understanding of the program's state and behavior.

The size and complexity of software is also a major impediment to program understanding tools, particularly those based on static analysis of the code. The programming techniques described above often result in very imprecise information that is of little value to the programmer. Tools that analyze the dynamic behavior of programs have traditionally focused on identifying performance problems rather than on general program understanding [20, 26, 18]. The primary technique currently available for inspecting program state is the debugger, which is extremely painful to use for anything but the smallest data structures.

In this paper we present a new tool called *Heapviz* that is capable of effectively visualizing heap snapshots obtained from running Java programs. By visualizing the actual contents of the heap, we avoid the drawbacks of static analysis tools: the problems caused by dynamic software architectures and the inaccuracy of heap approximation. The main challenge of our approach is the scale of the data: even a modest program can contain an enormous number of objects. We visualize the heap as a graph in which nodes represent objects and edges represent pointers (object references) between them. Our work leverages the Prefuse visualization toolkit [12], which provides a rich set of software tools for building interactive visualizations. Unlike traditional debuggers, Heapviz provides a global view of the data together with powerful interactive capabilities.

Our solution involves two techniques. First, we introduce algorithms for aggregating and abstracting individual objects to create a more succinct summary of the heap. For example, we might display all the elements of a large container using a single representative element. Second, we use Prefuse to provide interactive query and navigation: (a) expand or collapse regions of the heap, (b) inspect individual objects and field values, (c) search for objects (or classes of objects) based on type, and (d) explore the connectivity of the object graph.

We demonstrate Heapviz on both constructed examples and real-world Java benchmark programs to evaluate its effectiveness as a tool for helping programmers visualize and navigate large, pointer-based data structures at a whole-program scale. This ability could greatly increase programmer productivity in many aspects of software construction and maintenance, including finding bugs and memory leaks, identifying opportunities to improve data structures, understanding the overall system architecture and interaction between software components, and helping new members on a development team come up to speed on the code quickly.

2. RELATED WORK

Previous work on program analysis and understanding includes a number of techniques for visualizing the behavior of programs. A large body of prior research has focused primarily on helping programmers navigate and visualize the *code* [4, 29, 6, 24]. As many computing researchers and practitioners have observed, however, understanding the *data structures* of a program is often more valuable. Techniques for determining the structure of data in the heap fall into two main categories: static analysis and dynamic analysis. Static analysis algorithms, such as shape analysis [10, 25], build a compile-time approximation of possible heap configurations. In many cases, however, these abstractions are too imprecise for detailed program understanding and debugging tasks.

Our work is most closely related to dynamic analysis tools that analyze the concrete heap (or a memory trace) in order to present the programmer with a graph or other visual representation of the actual state of the program. Since the main challenge for these tools is managing the scale of the data, the critical feature that distinguishes them is how they aggregate information for the user. Different choices lead to suitability for different tasks. Our specific goal for Heapviz is to help programmers understand the overall organization and structure of data.

Several existing tools provide programmers with an unabstracted graph representation of the concrete heap [32, 8, 31]. Without aggregation or interactive navigation, however, these visualizations do not scale well beyond a few hundred or thousand objects. Pheng and Verbrugge [19] present a tool with two visualizations of the heap. The first is a detailed graph of individual objects and pointers, with no abstraction. Nodes are displayed according to the shape to which they belong (list, tree, or DAG – from Ghiya and Hendren [10]). The second visualization consists of a line graph of the overall heap contents over time broken down by shape category. Their tool focuses on the evolution of the heap over time; Heapviz, on the other hand, aims to make a single snapshot of the heap comprehensible.

A number of existing heap visualization tools focus primarily on identifying memory utilization problems, such as memory bloat and memory leaks. The main difference between Heapviz and these tools is that they give up much of the detail of the heap organization necessary to understand how the data structures work.

De Pauw et al. [5] present a tool aimed at finding spurious object references. The tool collapses the heap graph by aggregating groups of objects with similar reference patterns. It also supports interactive navigation, expanding and contracting of aggregated nodes. While similar in spirit, this tool is focused on finding spurious references and requires some programmer intervention to identify which references to record and display.

Several tools aggregate objects by ownership [21, 13, 17]. These tools first analyze the heap and build a model of object ownership, then aggregate objects that have similar patterns of ownership, type, or lifetime. The visualization typically presents the abstracted graph with annotations that summarize the properties of the aggregated nodes. The DYMEM memory visualization tool [23] shows the heap as a strict tree, duplicating subtrees as necessary, and uses an elaborate coloring scheme to indicate the amount of memory used and owned by groups of objects. These tools are often not well-suited for general program understanding, however, since they abstract away internal organization and individual objects.

Demsky and Rinard [7] present a heap visualization based on a dynamic analysis of object *roles*, or patterns of use in the program. The tool generates a role transition diagram, which focuses on object lifecycles, rather than the overall organization of the data. While this tool presents a unique view, scalability appears to be a concern for large programs.

Most closely related to Heapviz is the work of Marron et al. [15]. They process the heap graph using an abstract function previously developed for use in a sophisticated static pointer analysis. The analysis attempts to preserve information about internal structure (such as sharing) whenever nodes are collapsed.

3. SYSTEM OVERVIEW

In this section, we provide a brief overview of the architecture of Heapviz. We go into further detail in Sections 4 and 5. The full Heapviz pipeline (Figure 1) consists of three major parts:

1. **JVM + HPROF.** Generates a heap snapshot from a running Java program using the HPROF tool [18] provided by Sun with the Java Development Kit.
2. **Heap analyzer.** Parses the heap snapshot, builds a

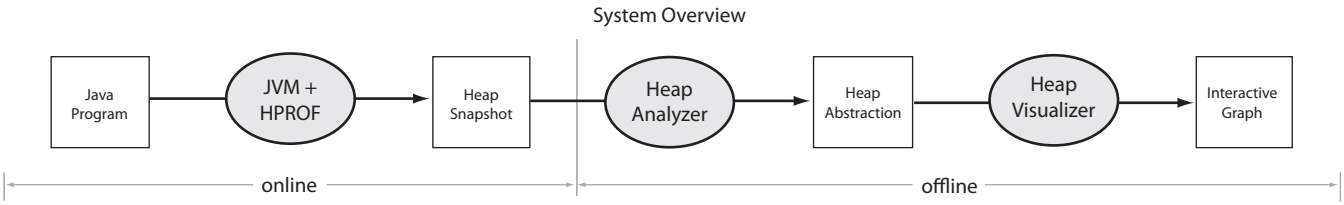

Figure 1: The Heapviz pipeline. The JVM and HPROF generate a heap snapshot from a running Java program. Our heap analyzer then parses this snapshot, builds a graph representation of the concrete heap, summarizes the graph, and outputs a heap abstraction. Our heap visualizer reads the heap abstraction and displays the graph, enabling the user to explore it interactively.

graph representation of the concrete heap, summarizes the graph, and outputs the resulting heap abstraction.

3. **Heap visualizer.** Reads the heap abstraction and displays the (possibly large) graph, allowing the user to explore it interactively.

Our *heap analyzer* parses the heap snapshot and builds a graph representation of the concrete heap. It then summarizes the concrete heap using the algorithm described in Section 4. In addition, it computes a dominator tree, which the visualization tool uses for layout. We call the combination of the summarized graph and dominator tree a *heap abstraction*. Finally, the heap analyzer outputs the heap abstraction to GraphML [11], an XML-based graph format.

Our *heap visualizer* reads the heap abstraction from the GraphML file and displays the summarized graph. As described further in Section 5, it displays the graph using a radial layout and uses the dominator tree to decide where in the hierarchy to place each node. The heap visualizer allows the user to explore the graph interactively, selecting nodes to view the contents of their fields, expanding and collapsing nodes, and displaying the connections among the objects in the heap.

4. HEAP ANALYSIS

4.1 Heap Snapshot

Our heap analysis starts with a heap snapshot obtained from a running Java program. The heap snapshot tells us which objects are currently in the heap, what their field values are, including the pointer values, and the values of all root references (static variables and stack references). In addition, the heap snapshot provides the runtime type of each object instance and the number of bytes needed to represent each object.

We use Sun's HPROF tool [18] to generate heap snapshots. HPROF is an agent that connects to a host Java virtual machine and uses the JVM Tool Interface [30] to enumerate all the objects in the program's heap. HPROF outputs the snapshot in a well-documented binary format that is supported by many third-party profiling and memory analysis tools. HPROF runs on top of any JVM that supports the JVM Tool Interface, so it is independent of JVM implementation.

One of our goals is to allow the programmer to view the heap at any point in the program's execution. To support this capability we provide a mechanism for dumping a heap snapshot from within the program itself. Since the source to HPROF is provided with the Java Development Kit, we modify it to include a class `Dumper` with a static method `dumpHeap()` that generates the heap snapshot when called. The programmer adds a call to `dumpHeap()` in the source code at the point where a heap visualization is desired.

4.2 Heap Analyzer

4.2.1 Input Format

Our heap analyzer takes the heap snapshot from HPROF and parses it into a sequence of records. Of interest are the class records, which tell us details about the types of the objects in the heap snapshot, the object instance records, which give the types of these objects and their field values, and the root records, which tell us which heap objects are pointed to by root pointers (stack references and static references). Using these data, the heap analyzer builds a graph representation of the program heap, mapping object instances to graph vertices, pointer fields to graph edges, and roots to entry points in the graph.

4.2.2 Summarization Algorithm

Typical Java programs may contain 100,000, 1,000,000, or more live objects at any given point in program execution [1]; drawing all these objects would make the visualization too cluttered to comprehend and too slow to interact with. Our heap analyzer *summarizes* the graph to make visualization manageable.

Our summarization algorithm is designed to reduce the size of the graph while retaining the relationship among nodes. Each node in the summary graph represents a set of nodes in the concrete graph with the same runtime type and similar connectivity, and the edges in the summary graph represent sets of edges in the concrete graph. It works by merging nodes in the concrete graph according to a set of rules, and repeatedly applying those rules until it reaches a fixed point.

The rules for merging are:

1. **If there exists a reference from object o_1 to object o_2, and o_1 and o_2 are of the same type, merge o_1 and o_2.** This rule merges the recursive backbone of a data structure (e.g. the nodes of a linked list or the nodes in a tree).

2. **If objects o_1 and o_2 have the same set of predecessor objects (objects that point to o_1 or o_2) and are of the same type, merge o_1 and o_2.** This rule merges sets of objects that have the same type and the same connectivity (e.g. the objects contained by a data structure).

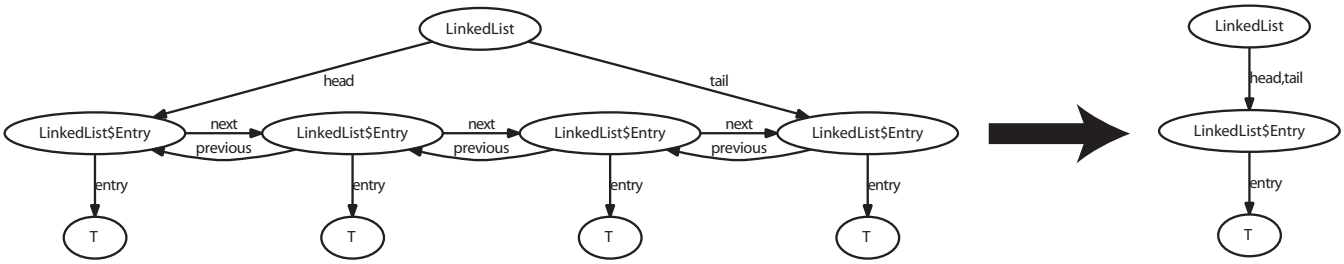

Figure 2: An example of applying our summarization algorithm to a linked list. Our algorithm first summarizes all `LinkedList$Entry` objects into a single node, then summarizes all `T` objects into a single node. Note that the summary would look the same regardless of the number of `T` objects in the linked list.

With these two simple rules, our system can compress very large graphs into more manageable ones. Consider the linked list data structure of Figure 2. This linked list contains four objects of type `T`. Our summarization algorithm first merges all the `LinkedList$Entry` objects into a single object using rule 1, and then all the `T` objects into a single object using rule 2. Note that the summarized graph looks the same no matter how many elements the linked list contains — whether 4, 400, or 40,000. However, if the linked list contains objects of different types, the summarized graph will contain separate nodes for the set of objects of each type.

4.2.3 Dominators

In additional to the graph summary, our heap analyzer also computes dominance information for all nodes in the graph. In graph theory, a node d dominates a node n if every path from the root to n must pass through d. Dominance is widely used in compilers for control flow graphs, and prior work has also applied dominance to heap snapshots [16]. In this work, we use dominance only to decide the layout of the graphs on the screen; the actual dominator tree is not displayed by default. We use dominance to establish the hierarchy of objects needed for a tree-based graph layout, as proposed by Falke et al. [9] for software dependency graphs. We discuss our graph layout scheme further in Section 5.

We use Cooper, Harvey, and Kennedy's dominance algorithm [3] to compute dominance. Its $O(n^2)$ asymptotic complexity is worse than the near-linear algorithm of Lengauer and Tarjan [14], but through well-engineered data structures it performs better in practice.

Dominance analysis requires the graph to have a single root, but the Java heap may have many roots (stack references and static variables). Therefore, before computing the dominator tree, we add to the graph a single "fake root" node that points to the roots of the Java heap. This "fake root" does not represent any real Java object, so we label it as such in the visualization.

4.2.4 Output Format

We write the summarized graph and dominance information in GraphML format [11] for the heap visualizer to import and display.

5. HEAP VISUALIZATION

Given a heap graph from the analyzer described in the previous section, Heapviz creates an interactive radial layout. The input file consists of one set of nodes and two sets of edges – a *pointer* set describing the connections that actually exist in the heap, and a *dominance* set describing conceptual parent-child relations between nodes; this is stored in GraphML format. The goals of the visualization are twofold: (1) to create an intuitive display of summarized heap data, and (2) to create an interactive environment where heap data can be easily explored. Heapviz builds upon the Prefuse toolkit [12]. Although our implementation focuses on facilitating program understanding and debugging, our visualization environment may be extended for other usage scenarios.

5.1 Layout

The heap abstraction includes dominance information for all nodes in the graph; the immediate dominator relationship defines a tree over the heap graph. This tree is at the heart of the layout algorithm. From among many algorithms tailored to display trees, we selected one that produces a radial graph because of its space-efficient expansion of the tree in all directions. Efficient use of screen space is of utmost importance, as even after summarization the graph may contain several hundred nodes. Radial graphs also avoid the restrictions presented by normal tree layouts. Normal tree layouts have many expectations accompanying them, such as the lack of back edges, something not guaranteed in the graphs Heapviz produces. To place emphasis on levels of the graph, the user can determine the depth of nodes by displaying concentric rings around the root node of the layout.

5.2 Visual Encoding

Visual encoding within the radial graph focuses primarily on edge and node differentiation to facilitate pattern recognition and to accentuate filters the user may have enacted upon the display. Edges appear in three colors: blue if they are a pointer edge, red if they are a dominance edge, and purple if they are both. Nodes also have three possibilities for both their fill color – indicating they are selected, not selected, or matching the current search query – and their outline color – indicating whether they are expanded, collapsed, or have no children in the dominance tree and thus cannot be either (these six states that dictate node coloration are described in the following section). Additionally, nodes display their object type on mouse hover.

5.3 Interaction

The visualization supports a variety of interaction styles, listed below according to their familiarity, with the most application-specific interactions last. The supplemental video

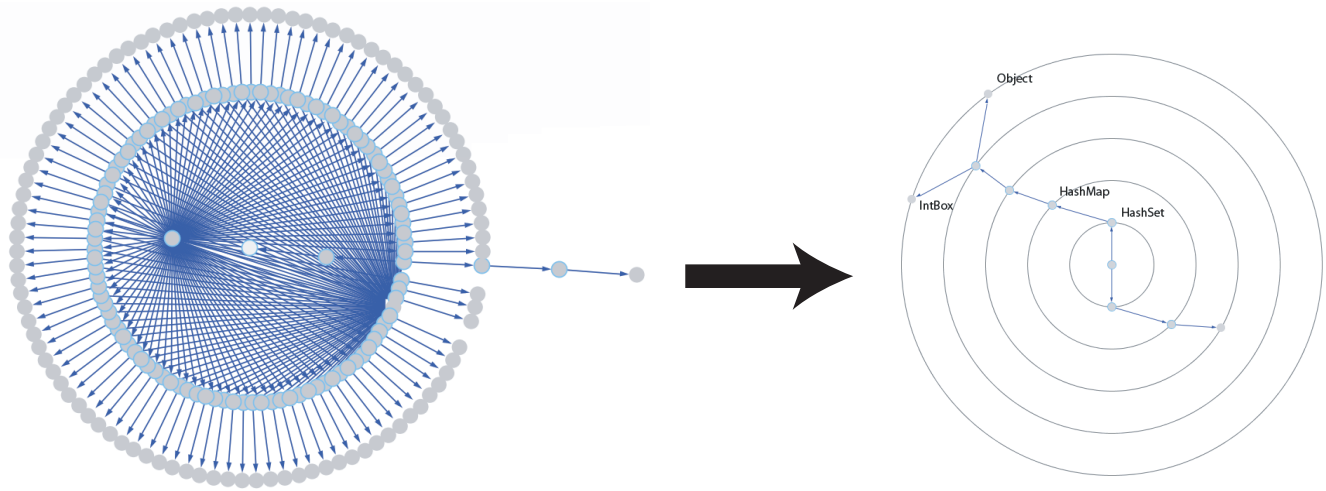

Figure 3: A Heapviz visualization of a `HashSet` containing 100 objects. The graph on the left is unsummarized, and the one on the right is summarized. The supplemental video demonstrates the interactive capabilities of our system using this visualization example.

demonstrates how the user can interact with Heapviz. Because our work relies on the user's being able to explore the graph interactively, we recommend that the reader view the video to have a better understanding of how Heapviz works and how it can be used.

5.3.1 *Canvas Movement*

The user is able to pan the view around the visualization, zoom in and out by arbitrary distances, and zoom the display to fit the entire graph. Additionally, the graph can be laid out relative to a node of the user's choosing, recentering the view on that node and bringing the entire graph back towards the new center.

5.3.2 *Selection and Dragging*

Nodes may be added to or removed from the current selection set either individually or by subtrees (as defined by the dominance tree). Once nodes are selected they can be dragged. By default, dragged nodes maintain their distance from the root node of the layout; however, the user may enable free movement of nodes.

5.3.3 *Search*

Heapviz provides the user with a search bar that performs an incremental search (search-as-you-type) over the names, member variable names, and member variable values of nodes. Nodes that fulfill the query are highlighted as they are found, a feature that reveals patterns of where particular objects or values may be found in the heap. Alternatively, searching can help the user quickly identify a particular node he or she would like to investigate.

5.3.4 *Field View*

Nodes in the graph have a variety of attributes that can be displayed to the user: member variable names, member variable values, number of instances (for summarized nodes) and size in bytes. When the user selects a node, Heapviz displays all node attributes that apply to the selected node. This allows the user to inspect the instance values of any Java object.

5.3.5 *Expanding and Collapsing*

The user can interactively collapse and expand nodes in the Heapviz graph. Only nodes that have children in the dominance tree can be expanded or collapsed. A node that dominates an entire subtree can be said to represent that subtree; the ability to expand (show) or collapse (hide) that subtree behind the dominating node offers the user a way both to reduce unwanted visual clutter and to conceptually simplify the graph.

5.3.6 *Edge Visibility Toggles*

The user is able to individually enable or disable the display of the two edge sets via a set of toggles. Dominance edges can provide revealing information about conceptual connections between data structures when the user is unfamiliar with the program; pointer edges show the actual structure of the object graph, and thus are useful for both program understanding and debugging.

6. CASE STUDIES

We now present the results of visualizing data structures in several Java programs and use these as a basis for discussion of Heapviz's strengths and weaknesses. First, we show two constructed examples (micro-benchmarks) built using standard container classes. Second, we explore two real-world benchmarks, _209_db [27] and SPEC JBB 2000 [28].

6.1 Constructed Examples

We first consider two examples constructed from standard data structures from the Java class library. In both cases, Heapviz can help users understand how a data structure is implemented without looking at the source code.

6.1.1 *HashSet*

Consider a set data structure: a collection that contains no duplicate elements and no ordering. One can implement a set using a hash table, which maps keys to values. The keys of the hash table are the elements of the set; the values are irrelevant but must be present. The Java class library

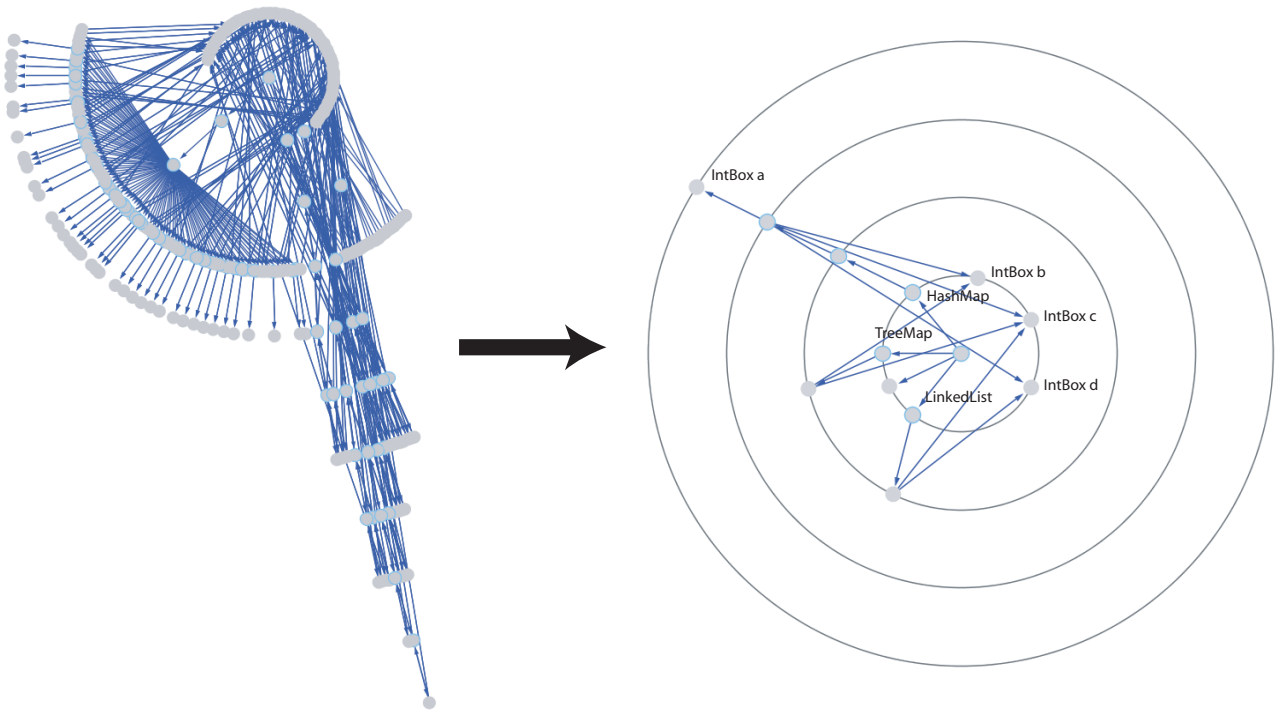

Figure 4: A Heapviz visualization of three data structures: a `TreeMap`, a `HashSet`, and a `LinkedList`. Each data structure contains a subset of `IntBox` objects. The unsummarized graph is on the left, and the summarized graph is on the right. Heapviz reveals the sharing among different data structures without requiring the user to look at the program source code.

includes a `HashSet` data structure that implements a set in this way.

Figure 3 shows a Heapviz visualization of a Java `HashSet` containing one hundred elements. The unsummarized graph shows the overall structure of the `HashSet` but is too cluttered to show the specifics of particular nodes. The summarized graph is much more manageable, with only ten nodes. In this, we see that the `HashSet` points to a `HashMap` (a Java hash table). All `IntBox` elements (our own implementation of boxed integers) are collapsed into a single node pointed to by a `HashSet$Entry` node. The user may see how many concrete nodes are summarized in one summary node by clicking on the node in the visualization. In this view, we notice that all `HashSet$Entry` nodes point to a single concrete `Object` node. This `Object` is the sentinel value in the hash table to ensure that a particular key is in the hash table and set. The implementor has chosen to save space and time by using the same `Object` as the value for all keys, and in fact we were unaware of this implementation detail until we viewed this Heapviz graph.

6.1.2 *Overlapping Elements*

Consider a program that contains multiple data structures. These data structures each contain some subset of the data objects in the program, and these subsets may overlap. That is, of the objects in data structure x, some are pointed to only by x, some by x and y, and some by x and z. Heapviz can explain this situation.

Figures 4 and 5 show such a program. This program has three data structures, a `TreeMap`, a `HashSet`, and a `LinkedList`, using the standard implementations in the Java class library. These data structures contain `IntBox` objects, some of which are shared by two or all three data structures. The unsummarized graph on the left in Figure 4 gives too much detail to discover this sharing, but the summarized graph makes the sharing clear. Each `IntBox` node in the summarized graph represents a set of concrete nodes with the same set of concrete predecessor nodes. Thus the summarized graph explains the sharing of `IntBox` nodes among the different data structures. For example, we can see that the `IntBox` objects represented by node a are pointed to only by the `HashMap`, but the ones represented by nodes c and d are pointed to by both the `HashMap` and the `LinkedList`. Thus we can determine that all `IntBox` objects in the `LinkedList` are also in the `HashMap`, but not all of the ones in the `HashMap` are in the `LinkedList`. In the interactive visualization, the user can click on nodes to see exactly how many `IntBox` nodes are shared among the data structures, and how many are only in the `HashMap`.

From the nodes and edges in Figure 4, without looking at the source code of the program, we can determine that:

1. There are three data structures in the program: a `TreeMap`, a `HashSet`, and a `LinkedList`.
2. Each data structure contains objects of type `IntBox`.
3. Some `IntBox` objects (the ones represented by node a) are pointed to by only the `HashMap`. The user could find the exact number by clicking on node a.
4. Some `IntBox` objects are shared by all three data structures (node c), and others are shared by only two

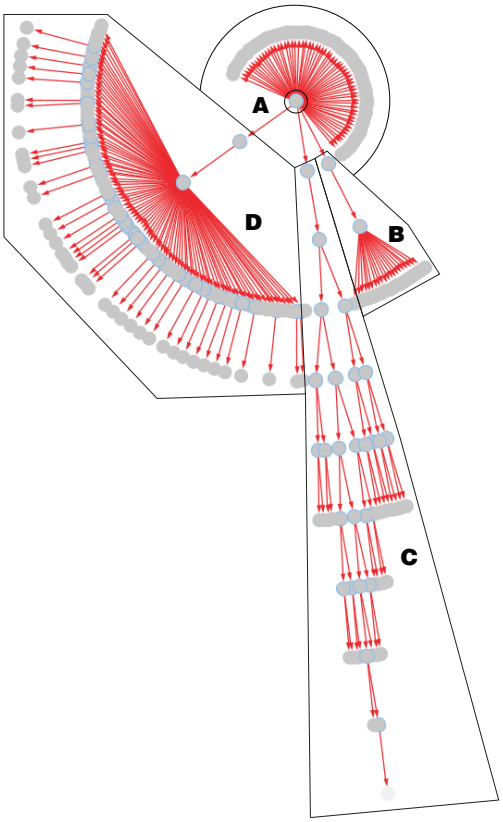

Figure 5: The same heap as in Figure 4, but showing only the dominance edges Heapviz uses for graph layout. The dominance tree clearly separates the three data structures – a `HashSet` (D), a `TreeMap` (C), and a `LinkedList` (B). Some objects (A) are shared among multiple data structures and thus are dominated only by the root.

(nodes b and d). Again, the exact number can be found by clicking on the appropriate summary node.

5. The `HashMap` contains all `IntBox` objects in the program. The other data structures each contain a strict subset of the `IntBox` objects.

Heapviz makes clear the sharing among different data structures without requiring the user to look at the program source code.

6.2 Real Examples

Now we assess Heapviz's visualization of two real-world benchmarks, _209_db [27] and SPEC JBB 2000 [28]. We consider how Heapviz is successful in helping us understand the data structures in _209_db and why it is less successful in summarizing the graph from SPEC JBB 2000.

6.2.1 _209_db

_209_db is a database benchmark from the SPEC JVM 98 benchmark suite [27]. It performs multiple database operations on an in-memory database. We took a heap snapshot of an execution of _209_db just after the database is built from an input file and before any operations are performed on the database. Figure 6 shows the summarized visualization on the bottom left, with the inset zoomed to show the program's primary data structure.

Before summarization, the graph contained 294,002 objects; after summarization, it contained 254. By zooming in on the nodes in the graph that summarize the most concrete nodes, we can see the primary data structure in the program: an object of type `spec.benchmarks._209_db.Database`. This database contains two `Vector` objects (`Vector`s are growable arrays of objects), which each contain strings or database entries. The node that represents all database entries in this database summarizes 15,332 nodes. The database entries then each contain a `Vector` of strings; the total number of these strings in the database is 122,656.

By inspecting the source code of _209_db, we can explain what these data mean. One of the two `Vector` objects pointed to by the database is used to store a format string that describes the records included in each entry. The other `Vector` holds the database entries. Each database entry uses a `Vector` to hold strings for each record: name, address, city, state, and so on. Though we had to look at the source code to understand this program, the visualization quickly showed us the primary data structure in the program and its high-level structure.

However, this example shows some of the limitations of Heapviz. Even though Heapviz was able to reduce the size of the graph by three orders of magnitude, the graph is still somewhat large and difficult to understand at a glance. In particular, it is difficult to pick out the important nodes in the graph, since nodes that summarize one concrete node are displayed the same way as nodes that summarize many concrete nodes. We discuss possible solutions to these issues in Section 7.

6.2.2 SPEC JBB 2000

The SPEC JBB 2000 [28] benchmark emulates a three-tier client-server system, with the database replaced by an in-memory tree and clients replaced by driver threads. The system models a wholesale company, with warehouses serving different districts and customers placing orders. We took a heap snapshot of SPEC JBB 2000 during the `destroy()` method of the `District` class in order to understand the sharing of `Order` objects stored by the `District`.

A full visualization of the SPEC JBB 2000 heap snapshot is shown in Figure 7. From the 117,819 objects in the concrete heap at this point in program execution, we produce a summarized graph of 7578 nodes, a reduction of 93.5%. Although some large data structures are visible after application of the summarization algorithm, the graph is still too visually complex.

The characteristics of this data provide some clues as to limitations of the summarization algorithm. SPEC JBB 2000 represents an extreme form of the simple program discussed in Section 6.1.2, with a dense web of connections among what we could consider the "leaves" of the data structures – objects that represent the individual elements of the program, such as customers and orders. In SPEC JBB, many of these "leaves" point to other "leaves" and have a one-to-one relationship. This one-to-one pointer mapping gives each of these "leaves" a unique predecessor set, preventing them from being summarized.

For example, consider the `Order` objects in SPEC JBB. An `Order` object points to the `Customer` who made that `Order`,

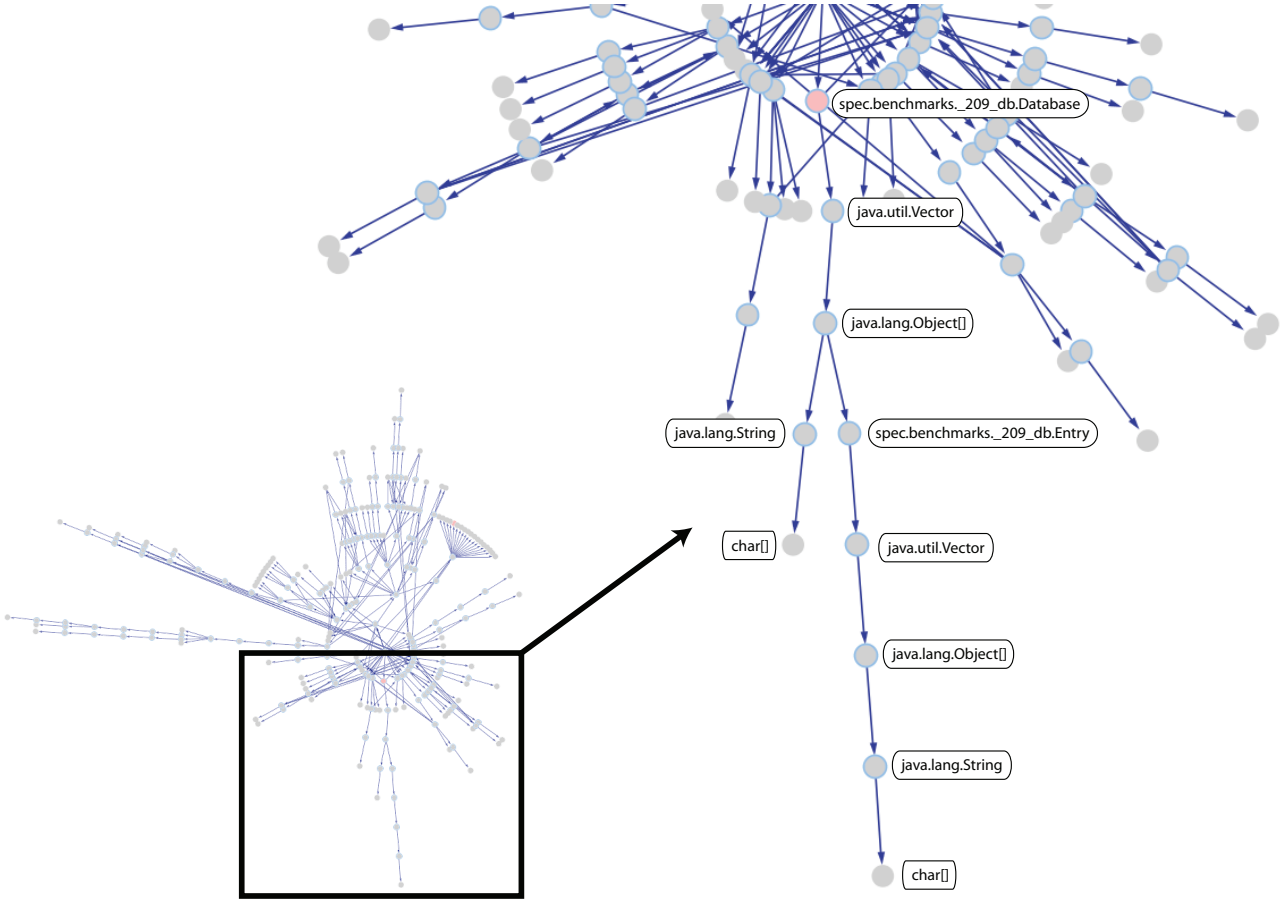

Figure 6: A summarized visualization of the _209_db benchmark. The bottom left shows the full graph, with the inset zoomed to show the program's primary data structure, a database. This database contains two `Vector` objects, each containing strings or database entries. The node representing all entries in this database summarizes 15,332 nodes. Each database entry contain a `Vector` of strings; the total number of these strings in the database is 122,656.

and a `Customer` object points to the last `Order` the `Customer` made. Thus, all `Customer` objects point to different `Order` objects, which then point back to the `Customer` object. This situation is shown in Figure 8.

Because `Customer` objects point to different `Order` objects, any `Order` pointed to by a `Customer` has a unique predecessor set and will not be merged with any other `Order`, unless the `Customer` objects are merged first. But because these `Order` objects point back to different `Customer` objects, those `Customer` objects also have unique predecessor sets and will not be merged. As a result, our algorithm cannot merge nodes that exhibit this one-to-one structure. We see this exact behavior in our graph: None of the three hundred `Order` objects are summarized because each of them points to and is pointed to by a different `Customer` object. We discuss possible solutions to this problem in Section 7.

7. FUTURE WORK

We have demonstrated Heapviz's performance on large programs; however, our summarization algorithm cannot greatly compress the size of graphs for programs containing a large number of nodes that have unique predecessor sets. With highly-connected graphs, such as the SPEC JBB 2000 benchmark (Figure 7), it is difficult to see the results of a query, even with highlighting of the relevant nodes. Our current summarization algorithm applies the same set of rules for merging nodes to all heap graphs, regardless of complexity. We plan to experiment with adjustable levels of detail, in which the summarization algorithm applies increasingly-powerful abstraction rules to a graph until it reaches a certain threshold of complexity, measured in number of nodes and edges. For example, Heapviz currently preserves sharing information among nodes, but for complex graphs like SPEC JBB 2000 it could discard sharing information to produce a more manageable visualization.

On a similar note, our visualization currently allows users to collapse and expand nodes based on the dominator tree. Another possibility would be to expand and collapse the concrete nodes that a particular summary node represents. This approach would require care to ensure that the graph does not become too complex when a summary node is expanded.

Heapviz currently supports searching for nodes by type, but a full-fledged query language would allow users to search for objects that meet specific criteria. We envision a query language based on first-order logic, similar to the heap as-

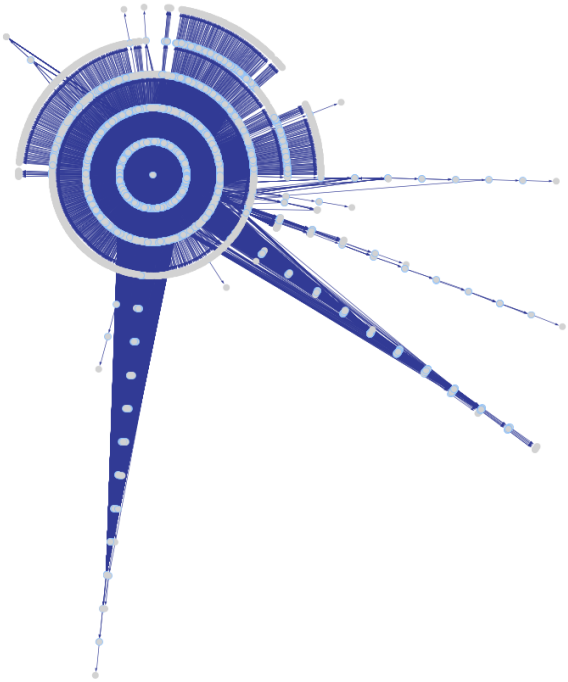

Figure 7: A summarized visualization of the SPEC JBB 2000 benchmark, which contains 117,819 objects in the concrete heap at this point in program execution. The summarized graph contains 7578 nodes. Although some large data structures are visible after this significant reduction, the graph is still cluttered.

sertion language proposed by Reichenbach et al. [22]. Such a language would allow queries such as "Which objects of type `String` have `length` ≥ 5?" and "Which objects are reachable from object `a`?" Similarly, we could support a path query language to highlight paths through the heap that match certain criteria.

Finally, because our long-term goal with Heapviz is to produce a tool that will be useful to developers, we plan to release Heapviz to the developer community. In addition, we plan to conduct formal user studies to determine whether outside users find Heapviz useful for program understanding and debugging tasks.

8. CONCLUSIONS

We have presented a tool for helping programmers analyze, visualize, and navigate heap snapshots from running Java programs. Heapviz enables users to navigate large, pointer-based data structures at a whole-program scale. We have introduced a heap analyzer, which parses a heap snapshot, builds a graph representation, and applies algorithms to create a summarized heap abstraction. We have demonstrated how to navigate this abstraction with a heap visualizer which supports several interaction styles, including detailed field view, expanding and collapsing of nodes, edge visibility toggles, and search.

Heapviz builds on a body of prior work on tools for debugging, static analysis, and data structure visualization. Our system makes several key contributions. Unlike traditional debuggers, we provide both an overview and detail; Heapviz provides the global view of the actual heap contents, as well as the ability to examine detail on demand with the field view. This variable level of detail supports programmer productivity in many common usage scenarios, including finding bugs and memory leaks, identifying data structures that could be improved, and understanding the overall system architecture. We hope that further exploration of domain-specific usage scenarios will spark the development of new analysis and visualization techniques.

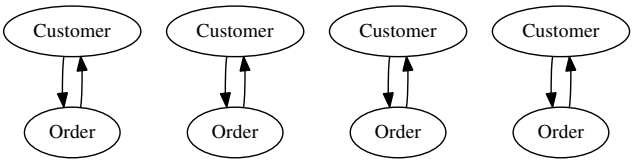

Figure 8: An example of the one-to-one mapping between "leaf" elements that we observed in SPEC JBB 2000. No matter what other objects point to these objects, they will not be summarized because the one-to-one structure guarantees that each has a unique predecessor set.

Acknowledgements

This research has been partially supported by the National Science Foundation under grant CCF-0916810. The authors would like to thank Ben Schwalb, Mark Marron, Audrey Girouard, and the anonymous reviewers for their helpful discussions and comments.

9. REFERENCES

[1] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Eliot, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 169–190, 2006.

[2] F. P. Brooks, Jr. *The Mythical Man-Month.* Addison-Wesley, 1975.

[3] K. D. Cooper, T. J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software—Practice and Experience*, (4):1–10, 2001.

[4] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. Visualizing the behavior of object-oriented systems. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 326–337, 1993.

[5] W. De Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in Java. In *The European Conference on Object-Oriented Programming*, pages 116–134, 1999.

[6] W. De Pauw and J. M. Vlissides. Visualizing object-oriented programs with Jinsight. In *The European Conference on Object-Oriented Programming*, pages 541–542, 1998.

[7] B. Demsky and M. Rinard. Automatic extraction of heap reference properties in object-oriented programs. *IEEE Transactions on Software Engineering*, pages 305–324, 2009.

[8] A. S. Erkan, T. J. VanSlyke, and T. M. Scaffidi. Data structure visualization with LaTeX and Prefuse. In *ITiCSE '07: Proceedings of the 12th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 301–305, New York, NY, USA, 2007. ACM.

[9] R. Falke, R. Klein, R. Koschke, and J. Quante. The dominance tree in visualizing software dependencies. In *VISSOFT '05: Proceedings of the 3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*, page 24, Washington, DC, USA, 2005. IEEE Computer Society.

[10] R. Ghiya and L. J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *ACM Symposium on the Principles of Programming Languages*, pages 1–15, 1996.

[11] Graph Drawing Steering Committee. The GraphML file format, 2010. `http://graphml.graphdrawing.org`, Retrieved April 30, 2010.

[12] J. Heer. Prefuse: a toolkit for interactive information visualization. In *CHI '05: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 421–430, 2005.

[13] T. Hill, J. Noble, and J. Potter. Scalable visualizations of object-oriented systems with ownership trees. *Journal of Visual Languages and Computing*, pages 319–339, June 2002.

[14] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[15] M. Marron, C. Sanchez, and Z. Su. High-level heap abstractions for debugging programs. 2010. `http://software.imdea.org/~marron/papers/publications.html`, Retrieved April 30, 2010.

[16] N. Mitchell. The runtime structure of object ownership. In *The European Conference on Object-Oriented Programming*, pages 74–98, 2006.

[17] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *The European Conference on Object-Oriented Programming*, pages 77–97, 2009.

[18] K. O'Hair. HPROF: A heap/CPU profiling tool in J2SE 5.0, 2004. `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`, Retrieved April 30, 2010.

[19] S. Pheng and C. Verbrugge. Dynamic data structure analysis for Java programs. In *Proceedings of IEEE International Conference on Program Comprehension*, pages 191–201, 2006.

[20] Quest. JProbe Memory Debugger. `http://www.quest.com/jprobe/`, Retrieved July 22, 2010.

[21] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *WODA '06: Proceedings of the 2006 International Workshop on Dynamic Systems Analysis*, pages 57–64, 2006.

[22] C. Reichenbach, , N. Immerman, Y. Smaragdakis, E. E. Aftandilian, and S. Z. Guyer. What can the GC compute efficiently? A language for heap assertions at GC time. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2010.

[23] S. Reiss. Visualizing the Java heap to detect memory problems. In *VISSOFT '09: Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis*, pages 73–80, 2009.

[24] S. P. Reiss and M. Renieris. Jove: Java as it happens. In *ACM Symposium on Software Visualization*, pages 115–124, 2005.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM Symposium on the Principles of Programming Languages*, pages 105–118, 1999.

[26] G. Sevitsky, W. De Pauw, and R. Konuru. An information exploration tool for performance analysis of java programs. In *TOOLS '01: Proceedings of the Technology of Object-Oriented Languages and Systems*, page 85, Washington, DC, 2001. IEEE Computer Society.

[27] Standard Performance Evaluation Corporation. *SPECjvm98 Documentation*, release 1.03 edition, 1999.

[28] Standard Performance Evaluation Corporation. *SPECjbb2000 Documentation*, release 1.01 edition, 2001.

[29] M.-A. D. Storey and H. A. Muller. Manipulating and documenting software structures using shrimp views. In *ICSM '95: Proceedings of the International Conference on Software Maintenance*, page 275, Washington, DC, 1995. IEEE Computer Society.

[30] Sun Microsystems. JVM tool interface, 2010. `http://java.sun.com/javase/6/docs/platform/jvmti/jvmti.html`, Retrieved April 30, 2010.

[31] J. Sundararaman and G. Back. HDPV: Interactive, faithful, in-vivo runtime state visualization for C/C++ and Java. In *SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization*, pages 47–56, New York, NY, USA, 2008. ACM.

[32] T. Zimmerman and A. Zeller. Visualizing memory graphs. *Lecture Notes in Computer Science - Software Visualization*, 2269:533–537, 2002.