Dynamic Data Structure Analysis for Java Programs

Sokhom Pheng Clark Verbrugge
School of Computer Science, McGill University
Montréal, Québec, Canada
{spheng,clump}@cs.mcgill.ca

Abstract

Analysis of dynamic data structure usage is useful for both program understanding and for improving the accuracy of other program analyses. Static analysis techniques, however, suffer from reduced accuracy in complex situations, and do not necessarily give a clear picture of runtime heap activity. We have designed and implemented a dynamic heap analysis system that allows one to examine and analyze how Java programs build and modify data structures. Using a complete execution trace from a profiled run of the program, we build an internal representation that mirrors the evolving runtime data structures. The resulting series of representations can then be analyzed and visualized, and we show how to use our approach to help understand how programs use data structures, the precise effect of garbage collection, and to establish limits on static data structure analysis. A deep understanding of dynamic data structures is particularly important for modern, object-oriented languages that make extensive use of heap-based data structures.

1 Introduction

Data structure, heap and *shape* analysis techniques summarize dynamic data connectivity, with the goal of improving alias analysis [9], automatic parallelization [11], optimizing garbage collection [23], debugging, or as part of a general understanding of program behaviour. Investigation of data structure shape and usage is particularly important for programs which make extensive use of heap data, such as in Java and other object-oriented languages. Static approaches to data structure analysis, however, potentially suffer from overly-conservative approximations, easily induced by temporary data structure inconsistencies during updates and modifications.

In this paper we investigate heap data analysis from the perspective of dynamic analysis. Using complete traces of Java program executions, we reconstruct the entire history of heap-based data as it is changed through program modifications. For smaller programs this allows for the construction of data structure snapshots and animations, visually illustrating evolution of program data, and also encoding a variety of properties of interest, including shape, age of data, node types, connectivity, and so on. For large benchmarks the results of analyses run at each data structure change are graphed to summarize overall behaviour. This permits larger scale investigations of data structure usage, and using a selection of standard Java benchmarks we demonstrate the extraction and analysis of various data that can extend detailed, runtime heap analysis to reasonably sized programs.

Data on number and size of data structures, their general shape, connectedness and entrypoints, all supply useful information on how programs use dynamic data structures, and we show how analysis of such data can provide insights into program behaviour. This includes aspects of data reachability—by analyzing which heap objects are actually reachable from program variables we can easily inspect the extent of and variation in unreachable, garbage data carried through program execution (GC *drag* [22]). A complete tracking of heap data also allows us to determine upper limits on the potential accuracy of a more traditional static, conservative tree/dag/cycle data structure analysis, under different assumptions of available alias analysis information. Most programs in our study are surprisingly simple with respect to heap usage and our results show that static approaches can be quite accurate, at least for common industry benchmarks.

Specific contributions of our work include:

- The design and implementation of a framework for capturing the complete dynamic evolution of data structures in Java programs. Our system includes a wide variety of analyses that expose interesting and useful benchmark properties.
- We compare accurate runtime data structure analysis data with that achievable by both optimal and simple static approaches. This establishes limits on accuracy for static heap analyses.

- We give and discuss experimental results on the actual data structure usage of a number of benchmark programs, including non-trivial programs in the SPEC JVM98 [24] and JOlden [3] suites.

In the next section we discuss background and related work on data structure and dynamic analysis. In Section 3 we describe the general design of our analyzer and the kinds of analyses and information we can gather. Section 4 discusses a simple representation, and Section 5 gives results from more detailed and larger benchmarks. Finally, we discuss future work and conclude in Section 6.

2 Related Work

Our approach combines two main techniques, dynamic analysis and shape analysis. These have historically been relatively orthogonal pursuits, and so we discuss them separately below.

2.1 Shape Analysis

Shape analysis techniques vary from implementing a whole new language for identifying data structures to summarizing them using specialized graphs.

A frequent, and early approach to identifying data structures is to allow the programmer to provide high-level information through program annotations. Hummel et al., for instance, define static annotations to data structures in order to help the compiler identify opportunities for parallelizing transformations [12]. A similar approach is described by Fradet and Le Métayer, who define a new language annotation that integrates the notion of shapes into the C language [8].

Many have tried identifying data structure shape without modifying the source code. Ghiya and Hendren show how the conceptually simple categorization of data structures into *tree*, *DAG*, or *cycle* can be sufficient for compiler optimization [9]. More detailed approaches attempt to model data using various kinds of graph abstractions. Klarlund and Schwartzbach's *graph types* build a representation as a grammar describing data structures having a backbone, such as doubly-linked lists [14]. Wilhelm et al. [25] define *shape graphs* to represent structural properties of data structures. *Three valued logic* has also been applied to express and prove shape properties [15]. Corbera et al. combines static shape graphs with abstract storage graphs to give a more precise shape analysis [4]; their techniques were improved by Navarro et al. by approximating the data structures in a graph combining memory locations having similar patterns [17]. Recently, Hackett and Rugina described a way of breaking down the entire shape abstraction into smaller component and analyzing them separately [10]. Raman and August [20] examine complete dynamic traces to generate abstract shape information; this is a similar general approach to ours, although they concentrate on identifying recursive structures, and derive their information from low-level, binary analysis.

While most work done on shape analysis has been done statically on C code, Bogda and Singh have done some exploratory work on shape analysis for Java code at run-time [1]. They analyze the dynamic call graph to help identify thread local and shared objects in order to perform synchronization removal. This technique focuses on the optimization property more than shape per se, but does illustrate an application of heap connectivity information.

Our own work here also includes aspects of data structure visualization. Visually representing the heap is an existing concern for debuggers [26], heap analysis tools [19], and visualizing profilers in general [21]. Most shape analysis studies, however, concentrate on the analysis more than depicting the analysis results, although there is recent work on defining structural shape properties suitable for visualization [13].

2.2 Dynamic Analysis

Dynamic program analysis can be performed online, or offline through the analysis of program execution trace files. Given the large resource demands of our precise shape analysis we have focused on the latter technique; inroads have been made to the former [1], however.

Trace extraction from Java programs often relies on the use of the Java's built-in Virtual Machine Profiling Interface (JVMPI), or its new replacement JVMTI (Tracing Interface). Brown et al. describe a framework, STEP, for profiler developers to encode general program trace data in a flexible and compact format [2]. JVMPI is also used by Dufour et al. in the implementation of *J [5], a tool for dynamic analysis of Java programs used to generate Java program metrics [6]. Our work here builds on the *J framework.

The Daikon project from MIT [18] and the Dynamo project from Indiana University [7] both provide online forms of dynamic analysis, differing mostly in usage. Both projects are based on observing runtime values and invariants to perform diverse analyses and optimizations. The Daikon project uses the information to report properties that were true over the observed period, which can then be used for testing and verification for example. Dynamo is a compiler architecture that uses the information to do runtime optimizations. The challenges of efficient online dynamic analysis are quite different from our exhaustive approach to trace analysis, but the invariant-based approach may be a useful basis for determining specific data structure properties.

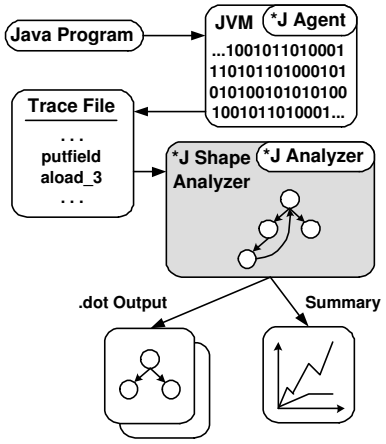

Figure 1. Design overview.

3 Design

We begin with an overview of our design, followed by a description of the properties we can represent in the output, and the analyses that we can perform. Fully accurate dynamic data structure analysis implies significant research challenges in handling and representing large amounts of information; this is discussed further in section 4.1.

The overall flow for our shape analysis system is shown in Figure 1. The first part of the process is data gathering. Java programs are executed in the JVM (Java Virtual Machine), and an attached *J agent produces execution trace files of the running program. Trace files are then fed into the *J shape analyzer. Here the input event trace is used to reconstruct the program data structures and their evolution over time. The *J shape analyzer may apply various analyses such as tree/DAG/cycle analysis, topological shape analysis, etc. The last part of the process is the output representation of the analysis data. Results can be communicated as literal snapshot or animated representations of graph structures, or in the case of larger outputs in more abstract and aggregated forms.

3.1 *J Shape Analyzer

The *J shape analyzer is built on top of *J, a tool for dynamic analysis of Java programs [6, 5]. *J comes in two parts: the first part is the *J agent, which produces trace files containing events obtained from the JVM through the JVMPI (Java Virtual Machine Profiling Interface). The second part is the *J analyzer, a framework for reading trace files and performing different analyses on that data. The *J shape analyzer extends the basic analysis facilities of *J.

For a complete and accurate analysis of runtime data structures we need complete data on heap objects and references, and all values which may be stored in reference fields. *J provides both a complete trace of all instructions executed, and unique identifiers for all objects. We are thus able to reconstruct heap connectivity by tracking which object identifiers are subject and target of reference field writes; this includes reference arrays. The *J shape analyzer reads events from the generated trace file and processes them one by one. For each event processed a corresponding update is applied to an internal structure that mirrors the program's heap nodes and their connectivity. This includes the removal of nodes due to GC. At each of these modification points, analyses are then run to determine the evolving properties of the data structure.

In order to construct an internal data structure that mirrors the program's heap nodes and their connectivity, we model the complete execution of each thread and method by interpreting bytecode events. The system is built following the Java Virtual Machine Specification [16]. Overall, it can be seen as mimicking the behaviour of a JVM in terms of object passing.

We must note that the amount of data that can be acquired through the JVMPI interface in *J is limited. Early events in the virtual machine startup are not available (occurring before JVMPI is initialized), and data from native method executions is not reliably delivered. In our investigations we have restricted our analyses to application code, not startup in order to ensure we have a complete event trace.

3.2 Data Properties

From the mirrored representation of the program data structures we are able to find and show a variety of interesting and useful properties. Certainly type, or other node information can be easily included in most visual representations. We can further determine and encode complex, historical node properties such as relative age of its component nodes, and the data structure can also be examined more abstractly, e.g., in terms of reachability.

3.2.1 Node Information. An example of representing useful node information is shown in Figure 2. Here we display type data textually, and also through colour, which we use to distinguish application from library objects. These kinds of properties and visualizations are straightforward to extend; Figure 2 also shows node *age* through colour: as an object ages, meaning that it lives longer within the program, its colour becomes darker (in Figure 2 this is applied only to application objects, not library objects). Observing age and type can be a useful way of understanding how a structure is constructed; in Figure 2, for example, it is evident that the data structure is mostly built bottom-up, with application nodes near the tree root younger than nodes deeper in the structure.

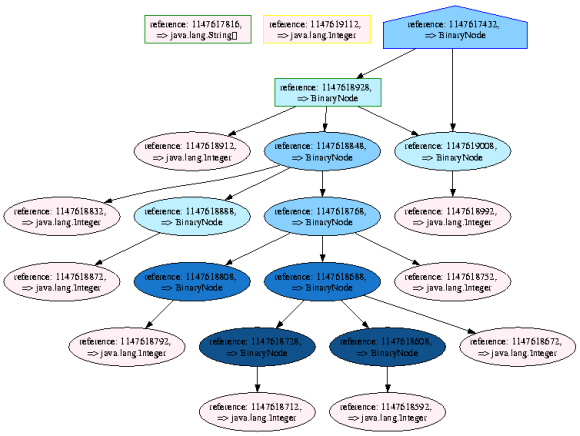

Figure 2. A data structure showing the aging property. Nodes are coloured according to their age (and type); all leaf nodes here are library objects, and all internal nodes application objects. The rectangular and pentagonal nodes are entry points.

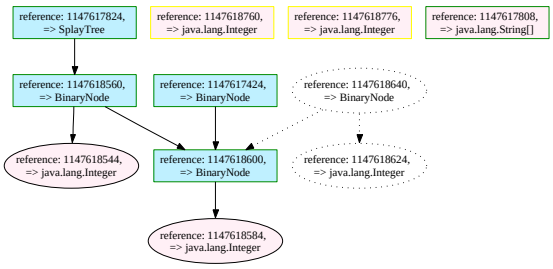

Figure 3. Showing garbage nodes in the data structure. Here unreachable nodes are drawn in dotted lines.

3.2.2 Graph Information. Non-local properties of data structures can also calculated; graph reachability, for instance, is easily determined in our system. By tracking all object references we also know the set of all root objects, or entry points to the structure. Root objects include static variables, live local variables, and live method parameters. Thus by comparing the transitive closure of references with the set of all allocated but currently uncollected objects we can determine the set of dead objects, not reachable from the root set. This information can be visualized, showing the exact amount and (remaining) connectivity of dead, garbage objects the heap contains. Figure 3 shows a visualization of a data structure containing garbage data. Here dead objects are drawn with dotted lines, and we can easily see how many there are and exactly how they are connected to each other and to the rest of the structure. Understanding how much data is carried in this way can be useful for garbage collector optimization [22].

3.3 Analyses

As well as basic node and connectivity information, the *J shape analyzer has all necessary information to support the implementation of various analyses, including different summary and shape graph approaches, topological shape analysis, etc. We have implemented a basic tree/DAG/cycle analysis as a proof of principle, and also to investigate the quality and utility of this simple data structure categorization.

3.3.1 Tree/DAG/Cycle. Dynamically, a tree/DAG/cycle categorization is quite trivial to compute. From each entry point we simply do a depth-first search to determine whether the nodes reachable from that entry point represent a tree, a DAG or a cyclic graph. This process has one important practical caveat: single, unconnected nodes are considered trees. While this is true in a technical sense, many programs make extensive use of single node objects, and this obfuscates any understanding of more realistic tree usage. For this reason we actually make use of a 4-way categorization, with single nodes distinct from trees.

Note that a given data structure may appear differently from different perspectives: it is common to think of data structures as connected graphs, but analysis information can be distinct for each entry point (reference variable), or generic to the entire connected data structure. Figure 2 shows examples of distinct tree and DAG entry points into the same connected structure. In most of our work we use entry point information as fine grain data; connected data structure information, however, is also determined.

3.3.2 Purity. Lastly, in order to measure the potential accuracy of a static analysis of the same program, we also define a *purity* metric on all data structure references.

Definition Let "$\sqsubseteq$" be a partial order on data structure shapes; e.g., tree $\sqsubseteq$ DAG $\sqsubseteq$ cycle. If the shape computed from a particular reference r at each heap change forms a sequence $s_1, s_2, \ldots, s_n$ where $s_i \in \{\text{tree}, \text{DAG}, \text{cycle}\}$ and n is the total number of heap changes observable from r. then r is *pure* if $s_i \sqsubseteq s_{i+1}$ for all $i = 0..n-1$.

Data structure purity is meant to capture the relative ability of a static shape analysis to accurately determine shape. If despite any changes the data structure is perceived to have the same, constant shape then static analysis may be able to give an accurate shape designation. If, however, the data structure shape changes then any static shape result is necessarily an approximation. Of course data structures are built incrementally—all data structures evolve from trees (single nodes). To avoid considering nearly all DAGs and cycle references as impure we categorize *monotonic* references, ones that never progress downward in shape order, as pure.

Purity thus over-approximates the accuracy of a static approach.

Based on the above, we compute two measurements on our runtime data. The entry point purity determines purity for each runtime reference. This provides a rough upper bound for static approaches, corresponding to the presence of perfect alias (points-to) information. Less than perfect alias information implies a need to merge information for multiple entry points, necessarily reducing, or at least not improving accuracy.

In the absence of good alias information, a static shape analysis can minimally separate references according to the static class type. To see how well even such a simple approach can determine data structure shape we compute a type-based purity metric; here, shape data for runtime fields with the same static signature are merged together. Purity is then determined from changes in the merged entity.

4 Literal Representations

We have two main ways to represent the data gathered. The most obvious and direct representation of data structure evolution is as series of literal snapshots of the encoded data structures, as in Figures 2 and 3. In this model the information from various analyses is encoded in the visual output, as described above for age and reachability. Shape (tree/DAG/cycle) categorization is also encoded: if the reachable nodes form a tree then the entry point is drawn as a rectangle, if the structure is a DAG then the entry point is drawn as a "house shape" (pentagon), and for cyclic structures a hexagon entry point is used.

By performing our analyses at each structure modification we obtain an evolving view of the data, appropriate for animation. Snapshot animation itself, however, is surprisingly difficult, even with external tools. In order to have a nice animation of the snapshots, we need to be able to incrementally add/subtract nodes and edges to an existing drawing while ensuring existing nodes and edges do not move. This preserves the location of nodes between snapshots, making node identity trivially obvious as frames change. Current open source and commercial tools for graph layout, however, focus on optimal, static representations, and do not in general attempt to locate nodes in the same place between drawings. This results in animation frames where graphs in successive frames may bear little visual relation to each other, and thus are not useful as a visual replay of data structure behaviour. Improvements to this situation are part of our future work.

4.1 Scaling Concerns

While it is quite natural, if not trivial, to represent data structure evolution as series of literal snapshots of the encoded data structures, it is not feasible as a general approach to most benchmarks. The large data sets that must be manipulated in the context of the analyzer impose strong constraints on the style of presentation, and also on the kind of data that can be gathered.

Tiny, test programs modify data structures only a relatively small number of times. More realistic programs, however, can perform a very large number of updates; the Jess benchmark from SPECjvm98, for instance, performs more than 48 million heap modifications. Examining all these snapshots is physically unrealistic for humans, and can generate significant analysis costs as well. Instead of generating snapshots for each modification we therefore only generate a snapshot every nth changes, for different n depending on the scale of investigation required. This trades off space for accuracy, but can also help in reducing the computational cost of the analysis.

Unfortunately, many programs also produce large data structures, whether or not they are modified frequently. Even a simple program such as BiSort from the JOlden benchmark suite generates more than 120 thousand objects—far too many objects for a drawing tool to handle, or to meaningfully show on a screen or in an animation. Interactive visualization techniques can improve this situation, but it is clear that animations, and even representative snapshots are appropriate for only very small programs. For the non-trivial benchmarks we analyze in the subsequent section we have thus concentrated on alternative representations that draw only reduced, aggregate information on data structure properties, and not the data structures themselves.

5 Experiments

We have analyzed a number of benchmarks from the SPECjvm98 and JOlden benchmark suites. Below we describe the programs analyzed, and present analysis examples based on the various data gathered using our framework. These discussions demonstrate both the kind of data we can collect, and also how it relates to relevant program features and behaviour.

The JOlden programs illustrate our analysis on non-trivial, but small and easily verified programs, whereas the SPECjvm98 benchmarks represent larger programs with more complex heap usage. For space reasons we cannot show results for the entirety of both benchmarks suites, or for all defined analyses. Here we discuss BiSort, Barnes-Hut, and TSP (Travelling Salesman Problem) from JOlden, and Jess, MpegAudio, and Compress from SPECjvm98. All benchmarks are run in Sun's 1.4.0 JVM, server mode (128M heap); the SPEC benchmarks are run at size 10 for Compress and MpegAudio, and size 100 for Jess.

5.1 Data and Analyses

Non-trivial benchmarks are not amenable to literal data structure representations, and so we present aggregated data from the analyses run in the shape analyzer at each data structure modification. We use data from three main analyses: a tree/DAG/cycle shape classification, reachability analysis, and purity.

Shape classification data is based on the number of entry points that reach single-node, tree, DAG, and cycle type data structures, plotted over time. For portability of results time is measured abstractly, as either bytecodes executed, or in terms of number of data structure modifications. To compress the visual representation our graphed data is sometimes a sampled subset; sample periods vary up to every 100K updates, and are indicated in the individual descriptions.

Reachability is given both in terms of the number of live versus dead objects, and in terms of number of connected structures. The former makes it easy to see general trends in volume of data and garbage, and for a limited visual inspection of GC drag. The latter gives a better impression of the number of connected data structures (of size at least 2) actually used in the program.

Purity data is used in two forms, entry point purity and type-based purity, as described in section 3.2.

5.1.1 BiSort. BiSort performs two bitonic sorts, one forward and one backward. It works in two phases. The first phase is the tree construction, and the second phase is the sorting.

In Figure 4, we can easily see the first phase, where the tree is being constructed. A number of single nodes are allocated, and then consumed by construction of the base tree. At about 1/3 of the way through execution the program enters its second phase; here many changes are performed on the tree, and the number of tree structures becomes quite variable. As the tree is modified the data types fluctuate between DAG types and tree types in a complementary fashion: nodes are being rearranged, and not copied or deleted. Note that there are not in fact as many disjoint structures as the number of trees and DAGs would indicate; call chains allow for the stack to contain multiple entry points to the same structure, magnifying the apparent number of structures. This is more evident in the top graph of Figure 5—in the second phase there is only ever 1 or 2 connected structures.

The bottom graph of Figure 5 gives an indication of how well a static analysis could do in identifying the data structure shape, assuming perfect alias information. Most references are pure, but the second phase of execution contains several impure variables due to the tree modifications. Statically these references would have to be considered

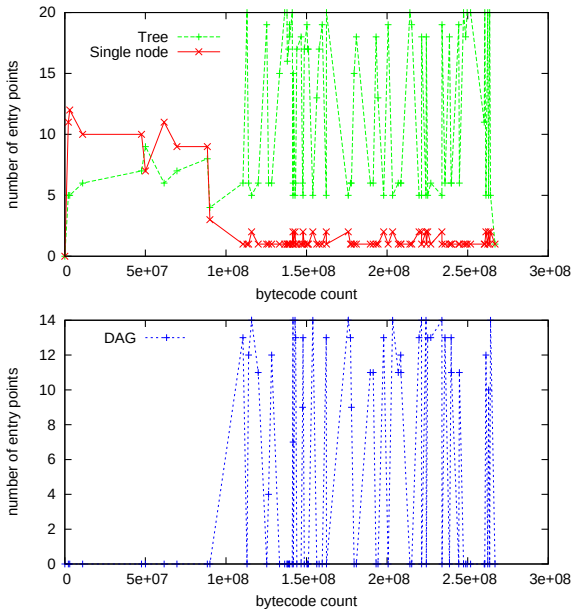

Figure 4. BiSort analysis results by bytecode for every 10k updates. The top figure shows single nodes and trees over bytecodes executed, and the bottom figure shows DAGs. There are no cycles in BiSort.

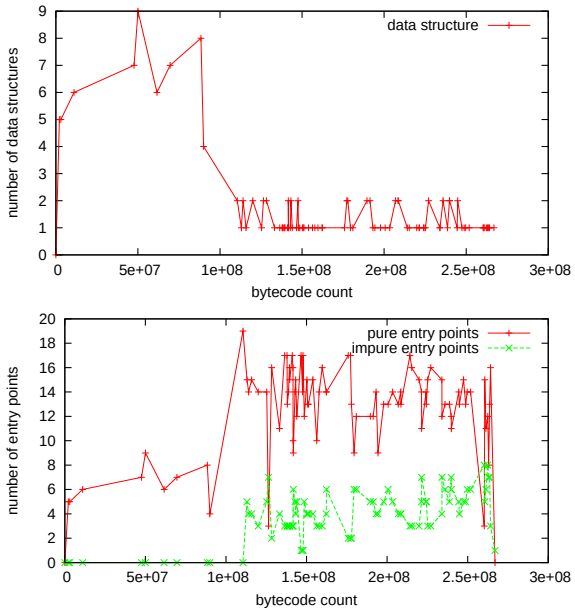

Figure 5. BiSort analysis showing the number of connected data structures (top) and the number of pure vs. impure entry points (bottom) for every 10k updates.

DAGs. Less optimal alias information may spread this conservative approximation.

5.1.2 Barnes-Hut. Barnes-Hut solves the classic N-body

gravitational attraction problem. Barnes-Hut also works in two phases; first tree construction, where a quad-tree is built, and second a force computation, where the tree is traversed. From the graphs in Figure 6 it is evident that this program is quite dynamic in behaviour, including either frequent data structure changes or allocations. As with BiSort there are no cyclic data structures at all. This is unsurprising for tree-based programs, but is also informative: it implies, for instance, that the quadtree does not make use of parent pointers in child nodes.

The phases are not obvious in the shape information, but are clearly shown in the GC results graph of Figure 7. The large spikes in number of dead objects indicate a rapid accumulation of garbage data, and the short-lived nature of the spikes suggests this is temporary data, quickly collected. The frequent variation in number of tree and DAG entry points is in this case mainly due to the use of allocated, intermediate data; purity data (not shown) in terms of both entry points and types shows all references pure, further supporting the conclusion that the shapes of existing data structures are not generally altered.

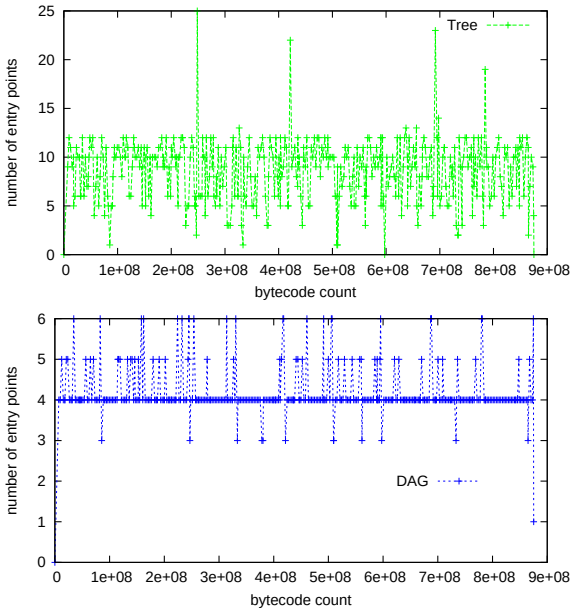

Figure 6. Barnes-Hut analysis results by bytecode for every 1k updates. On the top is shown the number of tree entry points over time (bytecodes executed), and on the bottom the number of DAGs. Again, there are no cyclic structures.

5.1.3 Travelling Salesman Problem. TSP computes an estimate of the best Hamiltonian circuit for the Travelling Salesman Problem. There are two clear phases evident in both graphs of Figure 8: a short initial phase constructing the problem, and a longer phase of analysis.

TSP is our first presented benchmark to actually include

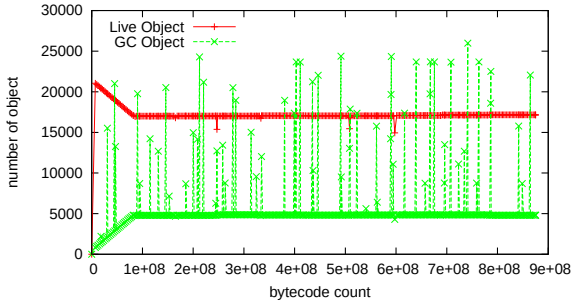

Figure 7. Barnes-Hut GC results by bytecode for every 1k updates, showing the number of live and dead objects over bytecodes executed.

cyclic data structures. There are also a very large number of tree data structures, orders of magnitude more than single nodes, DAGS, or cycles. In fact the algorithm mainly builds trees, and the few cycles can be attributed to a double-linked threading of trees forming partial solutions to the input problem.

TSP, relative to Barnes-Hut, generates minimal garbage, although references are also uniformly pure. This suggests a mainly static heap structure; however, since the number of entry points in different shape categories does fluctuate the data structures clearly do change. In this program the use of heap data at different stages in the computation is well-separated—entry points used in processing and generating the tree structures are disjoint from those used for DAGs and for cyclic structures.

5.1.4 Jess. Jess produces a lot of structures. Most of them, however, are single node objects, as shown in Figure 9. There are no cycles, and there is a rhythmic pattern of tree/DAG construction. This behaviour roughly corresponds with the algorithm and input, which does repeated, tree-based searches to solve a given combinatorial problem.

Memory usage in Jess is more complicated than in the JOlden programs. From Figure 10 we can see that a large number of objects are dead, usually many more than are live at any one time. Moreover, while the live set is overall stable, the number of dead nodes seems to have a general upward slant, increasing over time.

We believe this to be an artifact of heap adaptation. Jess allocates a lot of temporary objects (single nodes), some of which have non-trivial lifetimes. The heap pressure due to the use of temporary object allocations results in the heap being expanded to accommodate the perceived memory requirements. However, the core, necessary and retained data is not increasing, and a larger heap merely provides more room for garbage to accumulate. In this situation the amount of drag increases as the heap increases, suggesting that more aggressive GC rather than increasing heap size may result in more efficient execution.

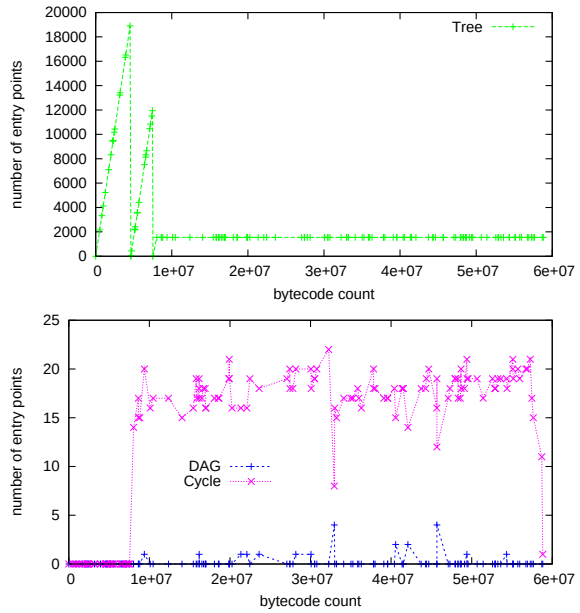

Figure 8. TSP analysis results by bytecode for every 1k updates. On the top are trees, and on the bottom DAGs and cycles.

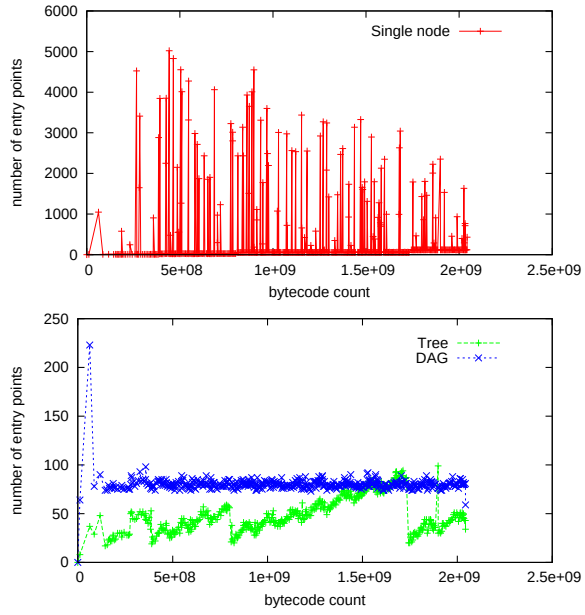

Figure 9. Jess analysis results by bytecode for every 100k updates. The top graph shows single nodes, the bottom shows trees and DAGS. There are no cycles.

5.1.5 Compress. Most of the benchmarks produce extremely similar graphs whether the time axis is formed of bytecode executions, or expressed in terms of data structure modifications: data structure updates are quite regular.

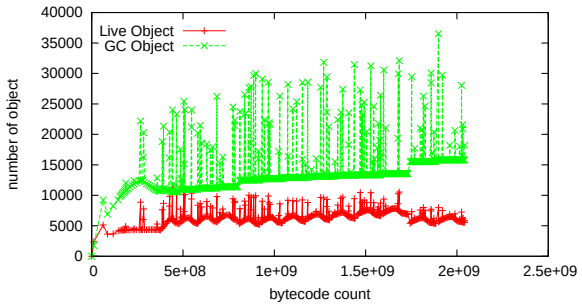

Figure 10. Jess GC results by bytecode for every 100k updates.

Compress shows this is not always the case. In the bottom graph of Figure 11 the number of tree and DAG entry points are plotted against total number of data structure updates. The results show quite regular behaviour, with three obvious phases of execution, each consisting of two sub-phases. This correlates nicely with the known behaviour of Compress under our input parameters, which is to compress and decompress three files.

The top graph shows the same data plotted with respect to bytecodes executed. Here the phases are considerably less evident—the time spent compressing and decompressing each file is clearly uneven. Regularity of changes is a useful property for adaptive program optimization; Compress is quite deterministic in the sequence of actions executed, but duration of program phases, a large part of predicting behaviour, is here an input property.

5.1.6 MpegAudio. MpegAudio demonstrates the potentially large effect of good alias analysis on a static shape analysis. The top graph of Figure 12 shows that while there are a large number of entry points, they are entirely pure. However, a shape analysis that relies on less precise alias information may not be as successful as this suggests—the bottom graph of Figure 12 shows that when minimal alias data is available there are proportionally many impure references.

5.2 Overall

Dynamic data structure analysis can illustrate a variety of program behaviours. Most obviously execution phases are clearly visible in most of our graphs—programs, especially industry benchmarks, tend to behave in relatively regular ways, and data-centric algorithms show a corresponding regularity in data manipulations. Regularity is also seen in the kind of data used: the composition of trees, dags and cycles shows that while most of our benchmarks do perform numerous data structure modifications, they do not generally tend to be complex in their usage—there are surprisingly few cyclic structures found in our suite. In fact, there

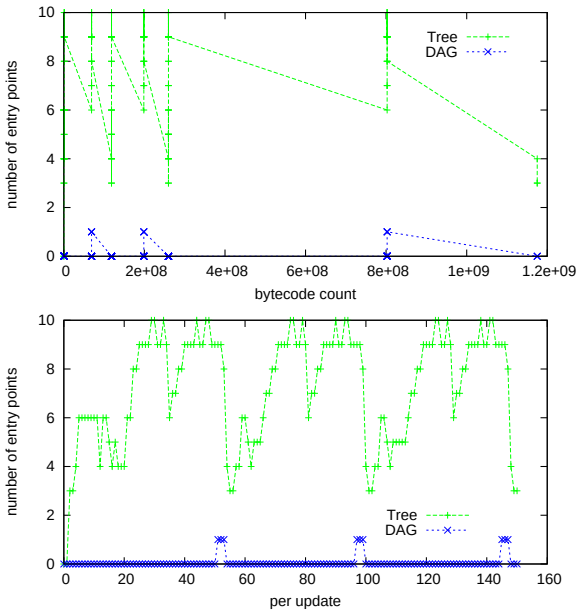

Figure 11. Compress shape (tree and DAG) analysis result by both bytecodes executed (above) and number of heap updates (below).

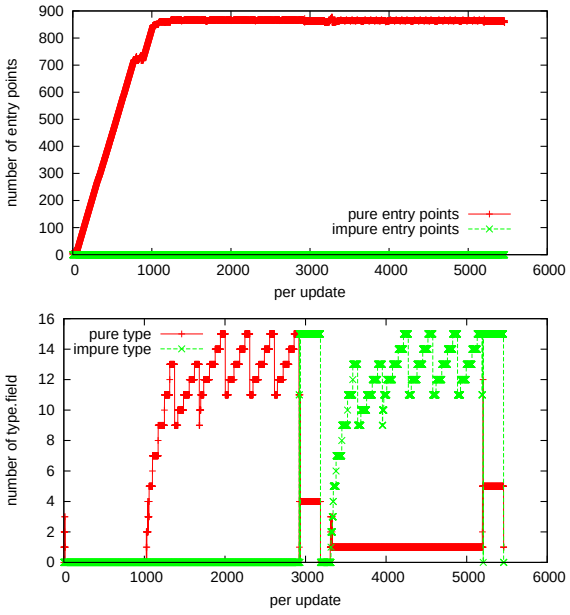

Figure 12. MpegAudio purity results per data structure change. The top graph shows number of pure versus impure entry points, and the bottom graph shows the purity of fields merged over all objects of the same class type. Updates are used as a time axis in this case to emphasize the two phases.

are surprisingly few actual connected structures (as opposed to entry points) in most programs. This lack of complexity in data structure usage is further supported by our purity data—certainly some entry points in some benchmarks do vary in the shape found, but most entry points are in fact pure, with many of our programs even having 100% of entry points pure. This is less well reflected in type-based purity, indicating the importance of alias information, but is still quite encouraging for static approaches.

Of course data structure analysis does not reveal all behaviours, not will it be equally effective on all programs. Scaling concerns require program data be summarized, and this is necessarily a lossy process; interesting events may be missed due to sampling interval size or selection. This is evident in several of our GC graphs, where a reduction in the number of live objects is not always matched by an equal rise in garbage objects. Scientific and other programs that make little use of dynamic data structures will obviously not show phases or other program behaviour through a heap analysis. Java programs in general are quite heap intensive, but this is at least partially evident in, for example, our results from Compress (Figure 11), where the heap is not altered for quite large portions of execution time. In the case of Compress the lack of heap activity is still a useful observation, but in less heap-intensive programs most or all of a program execution may not discernible.

6 Future Work & Conclusions

Dynamic data structure analysis has the ability to show detailed information on various aspects of program behaviour. This can help identify program characteristics, heap usage, and provide general understanding of any calculable static or evolving dynamic data structure property, advancing various optimization, understanding, and analysis goals. By comparing our runtime data with that achievable through static means we have been able to verify that static approaches have potential to be quite accurate, at least for many of our example programs. This suggests that despite the potential complexity of heap activity static interpretations may be generally sufficient for understanding how heap data is organized.

Our framework design and experience demonstrates the feasibility of this technique, and also highlights the research challenges involved. Extracting and reconstructing data structure changes is itself a non-trivial effort, with further complexity provided by the need for appropriate, scalable representations. Literal representations are natural and provide maximal information, but are not practical for real program investigations. More abstract analysis data from larger programs, however, can still provide useful and interesting information on program behaviour, while maintaining much of the accuracy provided by a dynamic, runtime analysis.

There are a great many potential future directions for this work. More, and larger benchmark programs would of course be useful, as would an examination of bench-

marks under different inputs. Our results here suggest data structure usage is often quite simple; further experimental evidence is needed, however, to make strong, general conclusions. Other languages and environments would also be interesting to consider. We have primarily focused on Java programs, due to the relative simplicity of using JVMPI to trace program execution. With appropriate tracing facilities we could extend our approach to other languages and environments. We are also interested in evaluating the efficacy and accuracy of more detailed shape analysis techniques, such as those based on compact graph abstractions [17], or shape types [25].

Visualization improvements are many of course. We have been most recently working on improving animation quality by adapting existing tools to support incremental, if perhaps sub-optimal, graph drawing. Integration of good animation with interactive visualization techniques can help alleviate some of the scaling concerns with literal representations, and can also be the basis for useful educational and debugging tools. Further, novel visualizations that compactly summarize graph properties are also important, and a combined approach that allows inspection of both literal and more abstract representations of heap activity would greatly assist the understanding of how programs use data structures.

Acknowledgements

This research has been supported by the le Fonds Québécois de la Recherche sur la Nature et les Technologies and the Natural Sciences and Engineering Research Council of Canada.

References

[1] J. Bogda and A. Singh. Can a shape analysis work at runtime? In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium*. USENIX, 2001.

[2] R. Brown, K. Driesen, D. Eng, L. Hendren, J. Jorgensen, C. Verbrugge, and Q. Wang. STEP: A framework for the efficient encoding of general trace data. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Anaylsis for Software Tools and Engineering (PASTE)*, New York, New York, United States, Nov. 2002. ACM Press.

[3] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java controller. In *PACT01*, pages 280–291, Barcelona, Spain, Sept. 2001.

[4] F. Corbera, R. Asenjo, and E. Zapata. New shape analysis and interprocedural techniques for automatic parallelization of C codes. *Int. J. Parallel Program.*, 30(1):37–63, 2002.

[5] B. Dufour. Objective quantification of program behaviour using dynamic metrics. Master's thesis, McGill University, Montréal, Québec, Canada, 2004.

[6] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic metrics for Java. In *Proceedings of the ACM SIGPLAN 2003 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 149–168, 2003.

[7] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *International Conference on Software Engineering*, pages 213–224, 1999.

[8] P. Fradet and D. L. Métayer. Shape types. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 27–39, New York, NY, USA, 1997.

[9] 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 *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–15, New York, NY, USA, 1996.

[10] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 310–323, New York, NY, USA, 2005.

[11] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. In *IEEE Transaction on Parallel and Distributed Systems, Vol. 1, No. 1*, pages 35–47, January 1990.

[12] J. Hummel, L. J. Hendren, and A. Nicolau. Abstract description of pointer data structures: an approach for improving the analysis and optimization of imperative programs. *ACM Lett. Program. Lang. Syst.*, 1(3):243–260, 1992.

[13] D. Johannes, R. Seidel, and R. Wilhelm. Algorithm animation using shape analysis: visualising abstract executions. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 17–26, New York, NY, USA, 2005.

[14] N. Klarlund and M. I. Schwartzbach. Graph types. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 196–205, New York, NY, USA, 1993.

[15] T. Lev-Ami and M. Sagiv. TVLA: a system for implementing static analyses. In J. Palsberg, editor, *Proceedings of the 7th International Static Analysis Symposium (SAS'00)*, number 1824 in LNCS, pages 280–302, 2000.

[16] T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison Wesley, 1996.

[17] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata, and E. Zapata. A new dependence test based on shape analysis for pointer-based codes. In *LCPC '04: Proceedings of the 17th International Workshop on Languages and Compilers for Parallel Computing*, 2004.

[18] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July-23 2001.

[19] T. Printezis and R. Jones. GCspy: an adaptable heap visualisation framework. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 343–358, New York, NY, USA, 2002.

[20] E. Raman and D. I. August. Recursive data structure profiling. In *Memory Systems Performance Workshop (MSP'05)*, Chicago, Illinois, June 2005.

[21] S. P. Reiss and M. Renieris. Jove: Java as it happens. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 115–124, New York, NY, USA, 2005.

[22] N. Röjemo and C. Runciman. Lag, drag, void and use - heap profiling and space-efficient compilation revisited. In *ICFP '96: Proceedings of the first ACM SIGPLAN international conference on Functional programming*, pages 34–41, New York, NY, USA, 1996.

[23] R. Shaham, E. K. Kolodner, and M. Sagiv. On the effectiveness of GC in Java. In *ISMM '00: Proceedings of the 2nd international symposium on Memory management*, pages 12–17, New York, NY, USA, 2000.

[24] SPEC Corporation. The SPEC JVM Client98 benchmark suite. `http://www.spec.org/jvm98/jvm98/`, 1998.

[25] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Computational Complexity*, pages 1–17, 2000.

[26] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization*, pages 191–204, 2001.