Abstracting Runtime Heaps for Program Understanding

Mark Marron[1] Cesar Sanchez[1,2] Zhendong Su[3] Manuel Fahndrich[4]

[1]IMDEA Software Institute, Spain [2]CSIC, Spain [3]UC Davis, USA [4]Microsoft Research, USA

{mark.marron, cesar.sanchez}@imdea.org, su@ucdavis.edu, maf@microsoft.com

Abstract

Modern programming environments provide extensive support for inspecting, analyzing, and testing programs based on the algorithmic structure of a program. Unfortunately, support for inspecting and understanding runtime data structures during execution is typically much more limited. This paper provides a general purpose technique for abstracting and summarizing entire runtime heaps. It shows how to take advantage of such heap abstractions for interactive debugging, visualization, and memory profiling.

We describe the abstract heap model and the associated algorithms for transforming a concrete heap dump into the corresponding abstract model as well as algorithms for merging, comparing, and computing changes between abstract models. The abstract model is designed to emphasize high-level concepts about heap-based data structures, such as shape and size, as well as relationships between heap structures, such as sharing and connectivity. The focus on high-level heap properties produces an abstraction that is useful for a range of applications.

We demonstrate the utility and computational tractability of the abstract heap model by building a memory profiler. We then use this tool to check for, pinpoint, and correct sources of memory bloat from a test suite including programs from SPEC JVM98 and DaCapo. This evaluation shows that the tool is useful for both finding previously unknown memory problems and in providing additional context for understanding previously reported issues.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Performance

Keywords Data Structure Visualization, Abstraction, Program Understanding

1. Introduction

Modern programming environments provide excellent support for visualizing and debugging code, but inspecting and understanding the high-level structure of the data manipulated at runtime by said code is typically not well supported. Visualizing entire runtime heap graphs is a non-trivial problem, as the number of nodes and edges is typically so large that displaying these graphs directly—even with excellent graph layouts—results in useless jumbles of nodes and edges. As a result, little of interest can be gleaned from such visualizations.

In this paper, we propose an abstract domain for runtime heap graphs that captures many fundamental properties of data structures on the heap, such as shape, connectivity, and sharing, but abstracts away other often less useful details. Abstract heaps can be computed efficiently from a single concrete heap and further summarized/compared with other abstract heap graphs. This summarization allows computing an abstract heap graph for a set of program runs, or from multiple program points in order to get an even more general view of the heap configurations that occur during program execution.

As we show in this paper, the abstract heap graphs we compute are both small enough to visualize and navigate, and at the same time precise enough to capture essential information useful in interactive debugging and memory profiling scenarios.

Overview. This paper addresses the problem of turning large concrete runtime heaps into compact abstract heaps while retaining many interesting properties of the original heap in the abstraction. Our abstraction is safe in the sense that properties stated on the abstract heap graph hold true in the corresponding concrete heaps. To achieve this abstraction safety, we adopt the theory for the design of abstract domains developed in *abstract interpretation* [7, 27]. The theory of abstract interpretation provides a general framework for 1) defining an abstract domain and relating it to possible concrete program states and 2) a method for taking an abstract domain and computing an over-approximation of the collecting semantics for a given program as a static analysis. The static analysis component of the abstract interpretation framework is not relevant here, as we are interested in abstracting runtime heaps. However, the framework for constructing the abstract domains, as well as the properties of operations for comparing ($\sqsubseteq$) and merging ($\widetilde{\sqcup}$) abstract domain elements, allows us to formally describe the relationship of our abstract heap graphs to their concrete counterparts, and to obtain safe operations for comparing and summarizing heaps from different program points or different program runs in a semantically meaningful way. These guarantees provide confidence that all inferences made by examining the abstract model are valid.

Our abstract heap domain encodes a fixed set of heap properties identified in previous work on static heap analysis [10, 21, 22] that are fundamental properties of heaps and can be computed efficiently. These properties include the summarization of recursive and composite data structures, the assignment of shape information to these structures (Tree, Dag), and injectivity of fields (given two distinct objects does the field `f` in each object point to a distinct target). The abstraction is also able to provide information on the number and types of objects in the various data structures, as well as definite non-nullness information for fields and containers.

Our focus on a fixed set of heap properties (as opposed to user defined properties) is what enables our abstraction to be computed efficiently in time $O((Ob+Pt)*log(Ob))$, where Ob is the number of objects and Pt is the number of pointers in the concrete heap. This amounts to a few seconds for concrete heaps containing well over 100K objects. All merge and comparison operations take fractions of a second. The resulting abstraction is still general purpose as the captured heap properties allow us to answer a wide variety of interesting questions on actual client applications.

Sec. 2 defines the precise semantics of our abstraction. Secs. 3 and 4 outline the algorithms for efficiently computing abstract heap graphs and for implementing the abstract merge and comparison

operations. Sec. 5 describes further visual compression techniques that help exploring large graphs. In Sec. 6, we present a case study of debugging memory inefficiencies on a well known benchmark, raytracer from SPEC JVM98 [34]. This section also describes the design of a specialized memory profiling tool, the results of using it find poor memory usage in our benchmarks (including programs from the DaCapo Suite [3]), and a brief report on early industrial experience with the visualization/profiler.

Example. Fig. 1(a) shows a heap snapshot of a simple program that manipulates expression trees. An expression tree consists of binary nodes for `Add`, `Sub`, and `Mult`, and leaf nodes for `Constants` and `Variables`. The local variable `exp` (rectangular box) points to an expression tree consisting of 4 interior binary expression objects, 2 `Var`, and 2 `Const` objects. Local variable `env` points to an array representing an environment of `Var` objects that are shared with the expression tree.

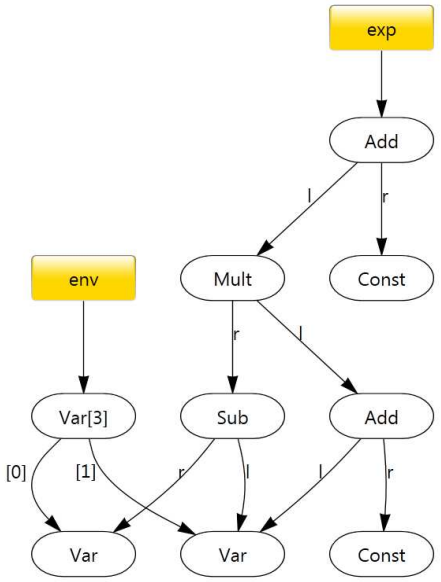

Figure 1(a). A Concrete Heap.

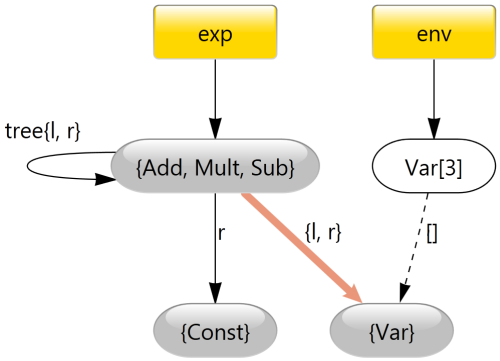

Figure 1(b). Corresponding Abstract Heap.

Fig. 1(b) shows the abstract heap produced by our tools from this concrete heap with the default visualization mode.[1] The abstraction summarizes the concrete objects into three distinct summary nodes in the abstract heap graph: 1) an abstract node representing all interior recursive objects in the expression tree (`Add`, `Mult`, `Sub`), 2) an abstract node representing the two `Var` objects, and 3) an abstract node representing the two `Const` objects. Specific details about the order and branching structure of expression nodes are absent in the abstraction, but other more general properties are still present. For example, the fact that there is no sharing or cycles among the interior expression nodes is apparent in the abstract graph by looking at the self-edge representing the pointers between objects in the interior of the expression tree. The label `tree{l,r}` on the self-edge expresses that pointers stored in the `l` and `r` fields of the objects in this region form a tree structure (i.e., no sharing and no cycles).

The abstract graph maintains another useful property of the expression tree, namely that no `Const` object is referenced from multiple expression objects. On the other hand, several expression objects might point to the same `Var` object. The abstract graph shows possible sharing using wide orange colored edges (if color is available), whereas normal edges indicate non-sharing pointers. Finally, the abstract graph shows that all pointers, besides the ones in the environment array, are definitely non-null as indicated using full vs. dashed lines.

Rudimentary information on the number of objects represented by each node is encoded in the shading. Nodes that always abstract a single object are given a white background while nodes which represent multiple objects are shaded (silver if color is available). Size information of arrays and other containers is encoded by annotating the type label with the container size (`Var[3]` to indicate an array is of length 3).

2. Abstract Heap Graph

We begin by formalizing concrete program heaps and the relevant properties of concrete heaps that will be captured by the abstraction. Later, we define the abstract heap graph and formally relate the abstraction to its concrete heap counterparts using a *concretization* (γ) function from the framework of abstract interpretation.

2.1 Concrete Heaps

For the purposes of this paper, we model the runtime state of a program as an environment, mapping variables to values, and a store, mapping addresses to values. We refer to an instance of an environment together with a store as a *concrete heap*. Formally, a concrete heap is a labeled directed graph $(\mathsf{root}, \mathsf{null}, Ob, Pt, Ty)$, where the nodes are formed by the set of heap objects (Ob) and the edges (Pt) correspond to pointers. We assume a distinguished heap object $\mathsf{root} \in Ob$ whose fields are the variables from the environment. This representation avoids dealing with distinct sets of variable locations and makes the formalization more uniform. We also assume a distinguished object null among Ob to model null pointers. The set of pointers $Pt \subseteq Ob \times Ob \times \mathsf{Label}$ connect a source object to a target object with a pointer label from Label. These labels are either a variable name (if the source object is root), a field name (if the source object is a heap object), or an array index (if the source object is an array). Finally, $Ty : Ob \to \mathsf{Type}$ is a map that assigns a concrete program type to each object. We assume the concrete set of types in Type contains at least object types and array types. We use the notation $o_1 \xrightarrow{p} o_2$ to indicate that object o_1 refers to o_2 via pointer label p.

A *region* of memory $C \subseteq Ob \setminus \{\mathsf{null}, \mathsf{root}\}$ is a subset of the concrete heap objects, not containing the root node or null. It is handy to define the set of pointers $P(C_1, C_2)$ crossing from a region C_1 to a region C_2 as:

$$P(C_1, C_2) = \{o_1 \xrightarrow{p} o_2 \in Pt \mid o_1 \in C_1, o_2 \in C_2\}$$

[1] Additional information can be obtained by hovering over the nodes/edges or by restyling for a specific task, as in our case studies in Sec. 6.

2.2 Concrete Heap Properties

We now formalize the set of concrete properties of objects, pointers, and entire regions of the heap that we later use to create the abstract heap graph.

Type. The set of types associated with a region C of the heap is the union of all types of the objects in the region: $\{Ty(o) \mid o \in C\}$.

Cardinality. The cardinality of a region C of the heap is the number of objects in the region $|C|$.

Nullity. A pointer $o_1 \to o_2$ is a definite null pointer if $o_2 = \mathsf{null}$. It is a non-null pointer if $o_2 \neq \mathsf{null}$.

Injectivity. Given two regions C_1 and C_2, we say that pointers labeled p from C_1 to C_2 are *injective*, written $\mathsf{inj}(C_1, C_2, p)$, if for all pairs of pointers $o_1 \xrightarrow{p} t_1$ and $o_2 \xrightarrow{p} t_2$ drawn from $P(C_1, C_2)$, $o_1 \neq o_2 \Rightarrow t_1 \neq t_2$. In words, the pointers labeled p from two *distinct* objects o_1 and o_2 point to *distinct* objects t_1 and t_2.

Shape. We characterize regions of memory C by shape using standard graph theoretic notions of trees and directed-acyclic graphs (dags). The set of graphs that are trees is a subset of the set of graphs that are dags, and dags are a subset of general graphs.

For additional precision, we consider the shape of subgraphs formed from C, and $P(C,C)\downarrow_L$, i.e., the subgraph consisting of objects from C and pointers with labels $l \in L$ only. This way, we can describe, for example, that a tree structure with parent pointers is still a tree if we only consider the left and right pointers, but not the parent pointers.

- The predicate $\mathsf{any}(C, L)$ is simply true for any graph. We use it only to clarify shapes in visualizations that don't satisfy any more restrictive property.
- The predicate $\mathsf{dag}(C, L)$ holds, if the subgraph $P(C,C)\downarrow_L$ is acyclic.
- The predicate $\mathsf{tree}(C, L)$ holds, if $\mathsf{dag}(C, L)$ holds and the subgraph $P(C,C)\downarrow_L$ contains no two edges $o_1 \xrightarrow{p_1} t$ and $o_2 \xrightarrow{p_2} t$ with the same target t, but distinct origins or pointer labels.

As is apparent from this definition, $\mathsf{tree}(C, L)$ implies $\mathsf{dag}(C, L)$, and $\mathsf{dag}(C, L)$ implies $\mathsf{any}(C, L)$. Also note that if $L_1 \subseteq L_2$, then $\mathsf{tree}(C, L_2)$ implies $\mathsf{tree}(C, L_1)$ and similarly for dag.

2.3 Heap Graph Abstraction

An abstract heap graph is an instance of storage shape graphs [5]. More precisely, the abstract heap graphs used in this paper are tuples:

$$(\mathsf{root}, \mathsf{null}, Ob^\#, Pt^\#, Ty^\#, Cd^\#, Ij^\#, Sh^\#)$$

where $Ob^\#$ is a set of abstract nodes (each of which abstracts a region of the concrete heap), and $Pt^\# \subseteq Ob^\# \times Ob^\# \times \mathsf{Label}^\#$ is a set of graph edges, each of which abstracts a set of pointers. Edges are annotated with labels from $\mathsf{Label}^\#$, a set of *abstract labels* (variable names, field labels, or the special label []). The label concretization is defined as follows and generalizes to label sets in the natural way.

$$\gamma_L(l) = \begin{cases} \{[0], [1], \ldots\} & \texttt{if } l == [] \\ \{l\} & \texttt{otherwise} \end{cases}$$

The special label [] abstracts the indices of all array elements. An abstract label l represents the given field or variable l.

As in concrete heap graphs, we distinguish a root node in $Ob^\#$ for modeling the variable environment as fields on root. Another distinguished node null is used to represent the null pointer.

The remaining parts of an abstract heap $(Ty^\#, Cd^\#, Ij^\#, Sh^\#)$ capture abstract properties of the heap graph. $Ty^\# : Ob^\# \mapsto 2^{\mathsf{Type}}$ maps abstract nodes to the set of types of the concrete nodes represented by the abstraction. $Cd^\# : Ob^\# \mapsto Interval$ represents the cardinality of each abstracted region. $Cd^\#$ maps each abstract node n to an abstract numerical interval $[l, u] \in Interval$, where the lowerbound l is a natural number, and u is a natural number or ∞.

The abstract injectivity $Ij^\# : Pt^\# \to \mathsf{bool}$ expresses whether the set of pointers represented by an abstract edge is injective. Finally, the abstract shape $Sh^\#$ is a set of tuples $(n, L, s) \in Ob^\# \times 2^{\mathsf{Label}^\#} \times \{\mathsf{tree}, \mathsf{dag}\}$ indicating the shape s of a region represented by n with edges restricted to L.

2.4 Abstraction Relation

We are now ready to formally relate the abstract heap graph to its concrete counterparts by specifying which heaps are in the concretization of an abstract heap:

$$\begin{aligned}(\mathsf{root}, \mathsf{null}, Ob, Pt, Ty) \in \gamma(\mathsf{root}, \mathsf{null}, Ob^\#, Pt^\#, Ty^\#, Cd^\#, Ij^\#, Sh^\#) \\ \Leftrightarrow \exists \mu.\ \mathsf{Embed}(\mu, Ob, Pt, Ob^\#, Pt^\#) \\ \wedge\ \mathsf{Typing}(\mu, Ob, Ty, Ob^\#, Ty^\#) \\ \wedge\ \mathsf{Counting}(\mu, Ob, Ob^\#, Cd^\#) \\ \wedge\ \mathsf{Injective}(\mu, Pt, Pt^\#, Ij^\#) \\ \wedge\ \mathsf{Shape}(\mu, Pt, Pt^\#, Sh^\#)\end{aligned}$$

A concrete heap is an instance of an abstract heap, if there exists an embedding $\mu : Ob \to Ob^\#$ satisfying the graph embedding, typing, counting, injectivity, and shape relation between the graphs. The auxiliary predicates are defined as follows.

$$\begin{aligned}&\mathsf{Embed}(\mu, Ob, Pt, Ob^\#, Pt^\#) \Leftrightarrow \\ &\qquad \mu(\mathsf{root}) = \mathsf{root} \quad \wedge \quad \mu(\mathsf{null}) = \mathsf{null} \quad \wedge \\ &\qquad \forall o_1 \xrightarrow{p} o_2 \in Pt.\, \exists l.\, \mu(o_1) \xrightarrow{l} \mu(o_2) \in Pt^\# \wedge p \in \gamma_L(l)\end{aligned}$$

The embed predicate makes sure that all edges of the concrete graph are present in the abstract graph, connecting corresponding abstract nodes, and that the edge label in the abstract graph encompasses the concrete edge label. The embedding mapping μ must also map the special objects root and null to their exact abstract counterparts. The mapping of null guarantees that the nullity/non-nullity property is preserved by the embedding: if a concrete pointer is null, its abstract edge must target the abstract null node. Similarly, if there is no edge from a particular abstract source node and label to the null node, then that pointer is guaranteed to be non-null.

$$\mathsf{Typing}(\mu, Ob, Ty, Ob^\#, Ty^\#) \Leftrightarrow \forall o \in Ob.\, Ty(o) \in Ty^\#(\mu(o))$$

The typing relation guarantees that the type $Ty(o)$ for every concrete object o is in the set of types $Ty^\#(\mu(o))$ of the abstract node $\mu(o)$ of o.

$$\mathsf{Counting}(\mu, Ob, Ob^\#, Cd^\#) \Leftrightarrow \forall n \in Ob^\#.\, |\mu^{-1}(n)| \in Cd^\#(n)$$

The counting relation guarantees that for each abstract node n, the set of concrete nodes $\mu^{-1}(n)$ abstracted by n has a cardinality in the numeric interval $Cd^\#(n)$.

$$\begin{aligned}&\mathsf{Injective}(\mu, Pt, Pt^\#, Ij^\#) \Leftrightarrow \forall (n_1, n_2, l) \in Pt^\#. \\ &\qquad Ij^\#(n_1, n_2, l) \Rightarrow \forall p \in \gamma_L(l).\, \mathsf{inj}(\mu^{-1}(n_1), \mu^{-1}(n_2), p)\end{aligned}$$

The injectivity relation guarantees that every pointer set marked as injective corresponds to injective pointers between the concrete source and target regions of the heap.

$$\begin{aligned}&\mathsf{Shape}(\mu, Pt, Pt^\#, Sh^\#) \Leftrightarrow \\ &\qquad \forall (n, L, \mathsf{dag}) \in Sh^\#.\, \mathsf{dag}(\mu^{-1}(n), \gamma_L(L)) \\ &\qquad \wedge\ \forall (n, L, \mathsf{tree}) \in Sh^\#.\, \mathsf{tree}(\mu^{-1}(n), \gamma_L(L))\end{aligned}$$

Finally, the shape relation guarantees that for every abstract shape tuple (n, L, s), the concrete subgraph $\mu^{-1}(n)$ abstracted by node n restricted to labels L satisfies the corresponding concrete shape predicate s (either tree or dag).

2.5 Visual Representation of Abstract Heap Graphs

In the iconography for our abstract graph visualizations, the screen shots in Fig. 1(a), Fig. 1(b), and Sec. 6, we leverage a number of conventions to convey information.

An edge (root, o, p) whose source is the root node represents the content of variable p. Instead of drawing a root node with such edges, we simply draw a variable node p and an unlabeled edge to o. Thus, the root node is never drawn, as it does not appear as the target of any edge in concrete or abstract graphs.

The set of abstract types of an abstract node is represented as the label of the abstract node. Shape information is represented as labels on the recursive self edges of abstract nodes. An abstract node with cardinality 1 is represented by a white background. Other cardinalities are represented with shaded abstract nodes.

We do not draw explicit edges which only point to null. If an edge is associated with a label that contains both pointers to null and pointers to other heap objects we fold the possibility into the edge by using a dashed edge instead of a full edge. Finally, injective edges are represented with normal thin edges, whereas non-injective edges are represented by wide edges (and if color is available are also highlighted with the color orange).

3. Computing the Abstraction

This section describes the computation of the abstract graph from a given concrete heap. The transformation is performed in three phases. 1) recursive data structures are identified and collapsed based on identifying cycles in the type definitions, 2) nodes that represent objects in the same logical heap region based on equivalent edges originating the same abstract node are merged, and finally 3) abstract properties like cardinality, injectivity, and shape are computed for the abstract edges and nodes.

Initially, we associate with each concrete object o_i an abstract partition n_i representing an equivalence using a Tarjan union-find structure. The mapping μ from concrete objects to abstract partitions is given at any point in time by: $\mu(o_i) = \mathsf{ecr}(n_i)$, i.e., by the equivalence class of the original n_i associated with o_i. The union-find structure maintains the reverse mapping μ^{-1} providing the set of concrete objects abstracted by a node. The abstract type map $Ty^{\#}$ can be maintained efficiently in the union-find structure as well.

Fig. 2 shows the initial state of these equivalence partitions for our example from Fig. 1(a) (one partition per object, plus the roots, and a special partition for null). Each node is labeled with its partition id and the types of the objects in that partition.

3.1 Recursive Data Structure Identification

The first abstraction identifies parts of the heap graph that represent unbounded depth recursive data structures. The basic approach consists of examining the type information in the program and the heap connectivity properties [2, 9, 20, 21].

We identify the parts of a recursive data structure via the type definitions in the program. We define a binary relation on the type definitions where $\tau_1 \triangleright \tau_2$, if:

- τ_1 has a field with type τ_2, or
- τ_1 is a container, which holds elements of type τ_2, or
- τ_1 is a supertype of τ_2.

DEFINITION 1 (Same Data Structure Types). *Types τ_1, τ_2 are types in the same data structure, written $\tau_1 \sim \tau_2$ iff there exists a sequence $\tau_1, \ldots, \tau_2, \ldots, \tau_1$ where:*

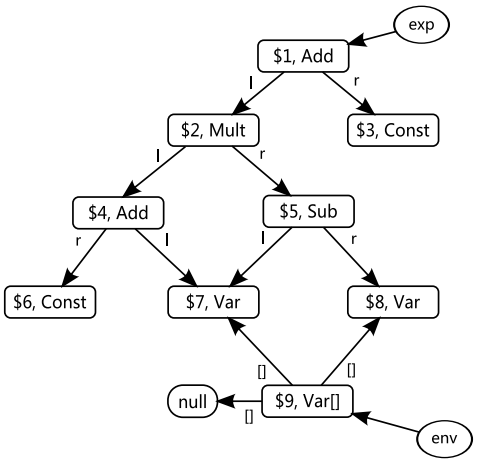

Figure 2. Initial Partition of Concrete Heap

- *for all τ_i, τ_{i+1} $\tau_i \triangleright \tau_{i+1}$, and*
- *there exists a τ_j in the sequence such that τ_j has a field of type τ_s or τ_j is a container of elements type τ_s, and $\forall$ types τ in the sequence, τ_s is a supertype of τ.*

DEFINITION 2 (Same Data Structure Objects). *Two distinct objects o_1, o_2 are* part of the same data structure *if there is a reference $o_1 \xrightarrow{p} o_2$ in the heap and the types of the two objects are in the same data structure $Ty(o_1) \sim Ty(o_2)$.*

This definition differs from a classic formulation of *recursive types* in that it attempts to distinguish between recursive type definitions that are intended to represent *unbounded* data structures and those that are intended to represent *bounded* depth structures (e.g., a linked list vs. a simple parent pointer). In our experiments, the above definition seems to perform this distinction well.

The recursive components are thus identified by visiting each pointer $o_i \rightarrow o_j$ in the heap and if o_i and o_j are in the same data structure according to Def. 2, then we union the corresponding abstract nodes n_i and n_j.

Fig. 3 shows the result of merging *Same Data Structure Nodes* on the initial partitions shown in Fig. 2. The algorithm identified objects $1, 2, 4, 5$ (the `Add`, `Sub`, and `Mult` objects from the interior of the expression tree) as being part of the recursive data structure and replaced them with a single representative summary node.

3.2 Grouping on Abstract Predecessors

Next we group objects based on predecessor partitions. The motivation for this abstraction can be seen in Fig. 3 where `Var` objects in partitions `7` and `8` represent "variables in the environment". There's no need to distinguish them as they are both referenced from the environment array. Similarly, the two constant objects referenced from the recursive component both represent "constants in the expression tree".

We use the following predicate for determining if two objects are equivalent based on where the pointers to them are stored.

DEFINITION 3 (Equivalent on Abstract Predecessors). *Given two pointers $o_1 \xrightarrow{l} o_2$ and $o_1' \xrightarrow{l'} o_2'$ where $\mu(o_1) = \mu(o_1')$, we say that their target nodes are* equivalent *whenever:*

- *The labels agree $l = l'$, and the target nodes have some types in common, i.e., $Ty^{\#}(\mu(o_2)) \cap Ty^{\#}(\mu(o_3')) \neq \emptyset$.*

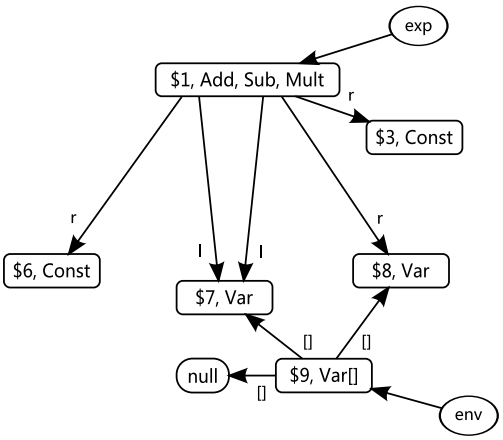

Figure 3. After Merging on *Same Data Structure*

The algorithm for grouping equivalent objects on abstract predecessors is based on a worklist, as merging two partitions may create a new opportunity for merging. The worklist consists of cross partition pointers that may have *equivalent* target objects, per Def. 3. When processing a cross partition pointer from the worklist we check if the pointer label and target object satisfies the *equivalent abstract predecessors* relation with any other cross partition pointer originating from the same partition. If there is such a pointer (and the target objects are in different partitions) then these partitions are merged and all pointers incident to the merged partitions are added to the worklist. Due to the properties of the Tarjan union-find algorithm, each cross partition pointer set can enter the work list at most $log(N)$ times, where N is the number of abstract partitions that can be merged, and E is the number of pointers. Thus the complexity of this step is $O(E * log(N))$.

Fig. 4 shows the result of performing the required merge operations on the partitions from Fig. 3. The algorithm has merged the two `Var` regions into a new summary region (since the objects represented by partitions 7 and 8 in Fig. 3 are both referred to from the same array). Similarly the two `Const` partitions from Fig. 3 have been merged as they are both stored in the same recursive data structure (the expression tree).

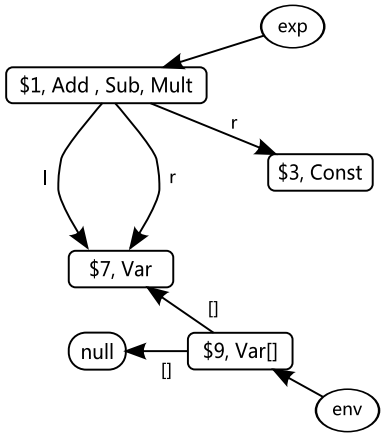

Figure 4. Merged Equivalent Predecessors.

Fig. 4 differs from the original abstract graph shown in Fig. 1(b) where there is only one edge between the expression tree node and the variables node. The reason is that, despite the underlying abstraction being a multi-graph, our visualization application collapses all multi-edges as they frequently lead to poor graph layouts and only rarely provide useful extra information to the developer.

Also, note that we still have explicit references to null and that these were not merged according to Def. 3, since we associate no types with the abstract null object.

3.3 Computation of Abstract Properties

After the merging of nodes and edges is complete, the abstraction process finally computes the values of all abstract graph properties.

Type, Cardinality, and Nullity. The abstract type map $Ty^{\#}$ has already been computed as part of the union-find operation on abstract nodes. Similarly, the union-find operation computes the exact cardinality, which results in a precise interval value $[i,i]$ if a node abstracts exactly i objects. The nullity information is represented as explicit edges to the null abstract object.

Injectivity. The *Injectivity* information for an abstract edge $n_1 \xrightarrow{l} n_2$ is computed by iterating over all pointers from objects o_i represented by n_1 to objects o_j represented by n_2 with label p compatible with l. We determine if every concrete target object is referenced at most once, in which case the abstract edge is *injective*. Otherwise, the edge is *not injective*.

Shape. The fundamental observation that enables interesting shape predicates to be produced for the abstract graphs is that the shape properties are restricted to the subgraphs represented by an abstract node. In addition, we allow the examination of a variety of further subgraphs by restricting the set of labels considered in the subgraph. Restricting the label set allows e.g., to determine that the $\{\mathtt{l},\mathtt{r}\}$ edges in a tree actually form a tree, even though there are also parent pointers `p`, which if included would allow no interesting shape property to be determined. Selecting the particular subsets of edge labels to consider in the subgraph selection is based on heuristics. We can start with all labels to get an overall shape and use that computation to guess which labels to throw out and try again. For small sets of labels, all combinations can be tried.

3.3.1 Abstract Properties of Running Example

After partitioning the heap as shown in Fig. 4 the final map for the objects is:

$$\mu^{-1}\begin{cases} n_1 : \{o_1, o_2, o_4, o_5\} \\ n_3 : \{o_3, o_6\} \\ n_7 : \{o_7, o_8\} \\ n_9 : \{o_9\} \end{cases}$$

Thus, for Fig. 1(b) we determine the abstract edge representing the cross partition pointer set $n_1 \xrightarrow{l} n_7$ is *not injective*, since it abstracts the two concrete pointers $o_4 \xrightarrow{l} o_7$ and $o_5 \xrightarrow{l} o_7$ both refer to the same `Var` object o_7. On the other hand, since the two `Const` objects o_3, o_6 are distinct, the algorithm will determine that edge representing the cross partition pointer set $n_1 \xrightarrow{r} n_3$ is *injective*.

The *Shape* of nodes for partitions 3, 7, and 9 is straightforward to compute as there are no edges internal to the regions. The *shape* computation for the node representing partition 1 requires a traversal of the four objects. As there are no cross or back edges the layout for this is `tree{l,r}`.

4. Merge and Comparison Operations

Many program analysis and understanding tasks require the ability to 1) accumulate abstract graphs, and 2) compare abstract graphs (both from the same program execution and across executions). For example, to support computing differences in the heap state during

profiling activities. So we cannot simply track object identities and use them to control the merge and compare operations. Thus, the definitions must be entirely based on the abstract graph structure.

4.1 Compare

Formally, the order between two abstract graphs $g_1 \sqsubseteq g_2$ can be defined via our abstraction relation from Sec. 2.4 as

$$g_1 \sqsubseteq g_2 \Leftrightarrow \forall h.h \in \gamma(g_1) \Rightarrow h \in \gamma(g_2)$$

However, this is not directly computable. Instead, we implement an approximation of this relation that first determines the structural equality of the abstract graphs by computing an isomorphism, followed by an implication check that all abstract edge and node properties in g_2 cover the equivalent node and edge properties of g_1.

To efficiently compute the subgraph isomorphism between g_1 and g_2 we use a property of the abstract graphs established by Def. 3. From this definition we know that every pair of out edges from a node either differ in the *label* or have the same *label* but non-overlapping sets of *types* in the nodes they refer to. Thus, to compute an isomorphism between two graphs we can simply start pairing the local and global roots and then from each pair match up edges based on their *label* and *type* sets, leading to new pairings. This either results in an isomorphism map, or it results in a pair of nodes reachable from the roots along the same path that have incompatible edges. Any such edge differences can then be reported.

With the subgraph isomorphism ϕ, we define the ordering relation:

$$\begin{aligned} g_1 \sqsubseteq_\phi g_2 \Leftrightarrow & \\ & \forall n \in Ob_1^\#.Ty_1^\#(n) \subseteq Ty_2^\#(\phi(n)) \\ & \wedge \forall n \in Ob_1^\#.Cd_1^\#(n) \sqsubseteq Cd_2^\#(\phi(n)) \\ & \wedge \forall \phi(e) \in Pt_2^\#.Ij_2^\#(\phi(e)) \Rightarrow Ij_1^\#(e) \\ & \wedge \forall(\phi(n), L_2, s_2) \in Sh_2^\#.\exists(n, L_1, s_1) \in Sh_1^\#.L_2 \subseteq L_1 \wedge s_1 \sqsubseteq s_2 \end{aligned}$$

Note how abstract shape predicates are contra-variant in the label set L. In other words, if a shape property holds for the subgraph based on L_1, then it holds for the smaller subgraph based on the smaller set L_2.

4.2 Merge

The merge operation takes two abstract graphs and produces a new abstract graph that is an over approximation of all the concrete heap states that are represented by the two input graphs. In the standard abstract interpretation formulation this is typically the least element that is an over approximation of both models. However, to simplify the computation we do not enforce this property (formally we define an *upper approximation* instead of a *join*). Our approach is to leverage the existing definitions from the abstraction function in the following steps.

Given two abstract heap graphs, g_1 and g_2 of the form $g_i = (\mathsf{root}_i, \mathsf{null}_i, Ob_i^\#, Pt_i^\#, Ty_i^\#, Cd_i^\#, Ij_i^\#, Sh_i^\#)$ we can define the graph, g_3, that is the result of their merge as follows. First we produce the union of the two graphs by simply adding all nodes and edges from both graphs. Once we have taken the union of the two input graphs we merge all the variable/static roots that have the same names. Then we use Def. 2/3 to zip down the graph merging nodes and edges until no more changes are occurring. During the merging steps we build up two mappings $\eta_1 : g_1 \to g_3$ and $\eta_2 : g_2 \to g_3$ from nodes (edges) in the original graphs, g_1 and g_2 respectively, to the nodes (edges) in the merged graph. Using these mappings, we define safe upper approximations of all the graph properties:

$$\begin{aligned} Ty_3^\#(n) &= \bigcup_{n_1 \in \eta_1^{-1}(n)} Ty_1^\#(n_1) \cup \bigcup_{n_2 \in \eta_2^{-1}(n)} Ty_2^\#(n_2) \\ Cd_3^\#(n) &= \sum_{n_1 \in \eta_1^{-1}(n)} Cd_1^\#(n_1) \sqcup \sum_{n_2 \in \eta_2^{-1}(n)} Cd_2^\#(n_2) \\ Ij_3^\#(e) &= (|\eta_1^{-1}(e)| = |\{n_2 \mid n_1 \xrightarrow{l} n_2 \in \eta_1^{-1}(e)\}|) \\ &\quad \wedge (|\eta_2^{-1}(e)| = |\{n_2 \mid n_1 \xrightarrow{l} n_2 \in \eta_2^{-1}(e)\}|) \\ &\quad \wedge \bigwedge_{e_1 \in \eta_1^{-1}(e)} Ij_1^\#(e_1) \wedge \bigwedge_{e_2 \in \eta_2^{-1}(e)} Ij_2^\#(e_2) \end{aligned}$$

The set of types associated with the result is just the union of all types abstracted by the node in both graphs. The cardinality is more complicated to compute. It computes the abstract sums over intervals from all nodes abstracted from the input graphs separately, and then joins the resulting interval (or depending on the application widens as defined in [7]). Injectivity is the logical conjunction of the injectivity of all the source edges, provided that all the edges in the respective graphs that are merged had different target nodes (the equality of the edge and target sets). When merging two injective edges from the same graph cannot, in the case that they target the same node, guarantee that the resulting set of edges is injective, and if we encounter this we conservatively assume the result edge is *not injective*.

For computing the resulting *shape* predicates we need to take into account not only the shape properties of the original graphs, but also the connectivity among the input nodes that map to the same merged node in the joined graph. Define $\mathsf{dag}_\mu(n, L, \mu, g)$ and $\mathsf{tree}_\mu(n, L, \mu, g)$:

$$\begin{aligned} \mathsf{dag}_\mu(n, L, \mu, g) \Leftrightarrow & \; Pt_{g\downarrow_{\mu^{-1}(n),L}}^\# \text{ is acyclic} \\ & \wedge \forall n' \in \mu^{-1}(n).\exists L' \supseteq L(n', L', \mathsf{dag}) \in Sh_g^\# \\ \mathsf{tree}_\mu(n, L, \mu, g) \Leftrightarrow & \; \mathsf{dag}_\mu(n, L, \mu, g) \\ & \wedge |Pt_{g\downarrow_{\mu^{-1}(n),L}}^\#| = 0 \\ & \wedge \forall n' \in \mu^{-1}(n).\exists L' \supseteq L(n', L', \mathsf{tree}) \in Sh_g^\# \end{aligned}$$

where $Pt_{g\downarrow_{\mu_i^{-1}(n),L}}^\#$ is the subgraph of $Pt_g^\#$ made up of nodes that map to n under μ and non-self[2] edges incident to them and restricted to labels L. Note that tree can only be inferred, if the set of joined nodes from a single graph are not connected in the joined subgraph. Now, the abstract shapes for the merged graph can be computed as follows:

$$\begin{aligned} (n, L, \mathsf{dag}) \in Sh_3^\# &\Leftrightarrow \mathsf{dag}_1(n, L) \wedge \mathsf{dag}_2(n, L) \\ (n, L, \mathsf{tree}) \in Sh_3^\# &\Leftrightarrow \mathsf{tree}_2(n, L) \wedge \mathsf{tree}_2(n, L) \end{aligned}$$

5. Additional Reduced and Interactive Views

While the abstract heap graph presented thus far produces models that scale in size with the number of logical regions in the program— independently of heap size and loosely correlated with the number of types used in the program—the graphs are often still too large to visualize and explore effectively. A second issue, particularly in a debugger scenario, is that after identifying a region of interest the developer wants to zoom into a more detailed view of the objects that make up the region.

While the DGML viewer [11] we use is quite effective at zooming, slicing, and navigating though large graphs we can directly address the above two issues by providing additional support for

[2] Self-edges need not be considered as they are already accounted for in the shape.

zooming between abstraction levels: the developer can zoom incrementally from a very high level view based on *dominators* in the abstract heap graph, defined in Sec. 2.3, all the way down to individual objects in the concrete heap without losing track of the larger global context of the heap structure[3].

Dominator Reduced Graph Given an abstract heap graph we can compute *dominator* information in a fairly standard way [26]. We deviate slightly since we want to ensure that *interesting* nodes which are directly pointed to by variables, and nodes that are immediate neighbors of these nodes remain expanded. In our experience this heuristic seems to strike a nice balance between collapsing large portions of the graph, to aid in quickly getting a general overview of the heap, while preserving structure around local variables, which are frequently of particular interest and we want extra detail on. This can be done by simply asserting that all of the nodes we want to keep expanded do not have any non-self dominators (equivalently ignoring all in-edges to these nodes during dominator computation). Using our modified dominator computation we can replace every node *n* (which has not been marked *interesting*) and all of the nodes $n_1^d \ldots n_k^d$ that *n dominates* with a single *reduced node*. This simple transformation results in a substantial reduction in the size of the graph while preserving much of the large scale heap structure and, since we can track the set of abstract graph nodes that each *reduced node* corresponds to, we can move easily between the two views of the heap. Furthermore, since the notion of *domination* and *ownership* [6] are closely related, this reduction has a natural relation with the developer's concept of ownership encapsulation of heap structures. This view is conceptually similar to the approach taken in [23, 24], although the dominator construction is on the abstract graph, where data structures have already been identified and grouped, instead of on the concrete heap graph.

Individual Object Zoom When looking at a graph that represents an abstraction of a single heap state (e.g., in an interactive debugger) it is very useful to be able to zoom down from the level of individual regions to examine the individual objects that make up a region. One approach for this is to simply expand a node in the abstract graph into the concrete object graph it represents. However, for large structures (e.g., a list with 2000 entries) this can produce an intractably large graph. An alternative is to mark individual objects as *interesting* and then implement the abstraction function such that these objects are always represented as distinct nodes (i.e., never merged). Then as the user drills down into a data structure, similar to what is done in existing debuggers, we can recompute the abstraction for the data structure that is being explored marking the appropriate nodes as *interesting* so they can be individually inspected.

6. Implementation and Evaluation

One goal of the present work is to build a general purpose tool for understanding the heap structures that a program is building and manipulating. To evaluate the utility of our abstraction, we examine 1) the cost of computing abstract heaps from realistically sized heaps of real programs, 2) the feasibility of visualizing the abstract graphs directly, and 3) whether the abstract graphs produced are precise enough for understanding the program's behavior and to identify various types of defects and properties.

6.1 Profiler Implementation and Benchmarks

We implemented the algorithms for computing and manipulating abstract heap graphs in C#. In order to visualize the resulting graphs we use the DGML [11] graph format and the associated viewer in Visual Studio 2010. This graph format and viewer support conditional style macros to control changes between the levels of abstraction described in this paper, and to perform selective highlighting of nodes/edges with given properties. In particular we can highlight edges that represent *non-injective* pointers, or we can apply a *heat-color* map to the nodes based on the amount of memory the objects they represent are using.

In order to evaluate the utility of the abstraction in the inspection and understanding of heap related problems (and in their solutions) we implemented a memory profiler tool. This profiler rewrites a given .Net assembly with sampling code to monitor memory use and to compute heap snapshots, along with the associated abstractions, at points where memory use is at a high point in the execution profile. The rewriter is based on the Common Compiler Infrastructure (CCI) [4] framework. As performing full heap abstractions at each method call would be impractical we use a per-method randomized approach with an exponential backoff based on the total reachable heap size (as reported by the GC). If we detect that the program may have entered a new phase of computation, the reachable heap size grows or shrinks by a factor of $1.5\times$ from the previous threshold, then we begin actively taking and abstracting heap snapshots. A snapshot of the heap is the portion reachable from the parameters of a single method call and from static roots. Again, depending on the size of the snapshot relative to the total reachable heap size, we either save the snapshot as likely capturing some interesting heap state or, if it is very small, discard it and increase the random backoff for the method that produced it. This use of random backoff sampling based on GC reported memory use and snapshot size results in a program that outputs between 2 and 10 snapshots from a program execution and execution is around $20\times$ to $100\times$ slower than the uninsturmented program. We compared the results obtained by sampling uniformly at random and found that, in addition to having a much larger overhead, the uniform sampling approach produced results that were no more useful for memory debugging then the backoff sampling approach.

In order to help the developer quickly identify structures of interest in the abstract heap we have implemented a number of simple post-processing operations on the abstract graphs which allow the DGML viewer to flag nodes (regions) of the heap that display common types of poor memory utilization [25]. The properties we support are percentage of memory used, *small object* identification, *sparse container* or *small containers*, and *over-factored classes*. The memory percentage property uses a heat map, coloring any nodes that contain more than 5%, 15%, or 25% of the heap respectively. The small object property highlights any nodes where the object overheads (assumed to be 8 bytes per object) are larger than the actual data stored in the objects. The poor collection utilization property highlights nodes that represent regions which are containers and all of them are either all very small (contain 3 or fewer elements) or are more than half empty (over half the entries are null pointers). While the first three properties are fairly standard, the final property, over-factored classes, is a less well known issue. We consider a structure overfactored if (1) there exists a node *n* that consists of small objects and (2) *n* has a single incoming edge that is *injective* (i.e., each object represented by the node *n* is uniquely owned by another object). These two features appear commonly when the objects represented by the node *n* could be merged with the objects that have the unique pointers to them (i.e., there are class definitions which can be merged) or when the objects represented by *n* could be better implemented as *value types* (i.e., `structs` in C#). The `Face[]` and `Point` objects in the raytracer case study in Sec. 6.2 are an example of this.

From the viewpoint of a userspace tool handling the types provided by the base class or system libraries, e.g., the Base Class Library (BCL) for .Net or the `java.*` in Java, are an impor-

[3] In a way that is similar to the *semantic zoom* of [8].

tant consideration. For user space applications the internal structure of say, `FileStream` or `StringBuilder` is not interesting, We identify these objects by simply examining the namespace of the type and treat them as single opaque objects. However, some classes in these libraries have features that are relevant to userspace code even though the details of the internal representation are not of particular interest. Examples of these types are `List<T>` or `Dictionary<K,V>`, which we treat as ideal algebraic data structures, showing the links to the contained elements but still treating the internal implementations as opaque.

For this paper we converted two benchmarks from SPEC JVM98 [34] and six programs from DaCapo suite [3] to .Net bytecode using the ikvmcompiler [14][4]. We also took the code for the CCI framework, used to implement the rewriter component of the heap profiler. This set of benchmarks has well-studied heap structures and memory use patterns. As a result, we were able to examine how well our abstraction results matched the expected structures. As the DaCapo suite contains a number of large and complex programs these results also provide information on how the extraction and comparison operations perform on large heaps.

Implementations of the algorithms in this paper are available online [12]. In addition a simplified implementation usable from a web browser is accessible at RiSE4Fun [32].

6.2 Raytracer: Extended Case Study

In this section we study the raytracer program from SPEC JVM98. The program implements a simple single threaded raytracer which renders a user provided scene. Using this example, we illustrate how the heap visualization looks for a well know program, and how this information can be used in a debugging type scenario to investigate memory use.

Running this program in the heap profiler, we obtain as one of the snapshots an abstract heap from the entry of the `shade`. This resulting abstract heap represents ∼168K objects (a total of ∼4MB of memory). Applying the heap graph abstraction followed by the dominator reduction produces the visualization shown in Fig. 5. This figure shows the *entire* user visible heap structure for the program while preserving most structural features of interest. In this heap we see the root nodes `this`, `tree`, and `eyeRay` representing the argument variables to the method and the static field `Scene.sceneLines`. The `this` variable refers to a `scene` object. This object has a field `octree` that represents a space decomposition tree structure which is also referred to by the `tree` argument variable. The larger nodes with the *chevron* are *dominator reduced* nodes that represent multiple dominated regions and can be expanded to inspect the internal structure in more detail.

The raytracer octree space decomposition structure is represented by the dominator reduced node labeled `#20`. It is directly noticeable that there are pointers from this data structure to `ObjNode` objects, represented by node `#7`. The shape `tree{nextLink}` of node `#7` indicates that this is a list (a tree with out-degree 1). The list in turn contains shapes (`SphereObj`, `TriangleObj`, ...) that are in the associated quadrants of the space decomposition structure. This list is used to easily enumerate all the shapes that appear in a given quadrant. There are also references from objects in the space decomposition tree structure to the dominator reduced node `#19`, which contains more information on the composite structure of `Face` objects.

Memory Use. Memory usage is an important concern for many applications. There are many reasons why an application may use more memory than is really required. Common problems in object-oriented, garbage collected languages are *leaks* [16], where unused objects are still reachable, and *bloat* [25], where encapsulation and layering has added an excessive overhead.

Ideally, upon debugging, a programmer would like to see what types are using the most memory and where these objects are being used. Our visualization uses the conditional styling support in the DGML renderer to color nodes based on the percentage of total used live memory. It also includes the number of objects of a node into its label. In the running example two nodes are highlighted: `#1` and `#19`.

Node `#1` represents a set of strings stored in a static array (`Scene.sceneLines`). This is a common problem in GC languages: if a static field is used to cache objects, then these objects are reachable from the viewpoint of the collector. These static fields may be defined and used in a very distant part of the source code, and then it is not clear for the programmer when the objects are no longer needed.

Hovering the mouse over node `#1` displays additional information about the memory consumed by the objects in the region (in this case 0.5MB or ∼12% of the memory in use). Based on the graph we can easily determine that the only reference to this set of objects comes from the static field `Scene.sceneLines`. A quick inspection of the source code associated with this static field shows that the strings represent the contents of the target render file. The file is read at program start and the strings are used to initialize the scene. After that, the strings are never used again and thus represent a memory leak. Clearing the static field after initialization is sufficient to free the otherwise wasted memory.

Node `#19` is more interesting. This node represents much more memory, ∼107K objects using 2.2MB of memory, which accounts for 55% of the total live heap. By expanding the node `#19` we zoom to the abstract graph (Fig. 6) representing the internal structure of the dominator reduced node. This graph reveals node (`$48`), abstracting a region of ∼18K `Face` objects, node (`$23`), abstracting a region of ∼18K `Point[]`, and node (`$49`), abstracting a region of ∼72K `Point` objects. The raytracer program is known to have poor *memory health* [25], in the sense that it exhibits a high rate of object overhead associated with a large number of very small objects. The `Point` objects here are a major factor in that.

At first glance it may not be clear how to reduce the overhead of these `Point` objects. However, turning on the *over-factored* highlighting or inspecting the *injectivity* information in Fig. 6, provides additional guidance. The edge from node `$23` to node `$49`—representing all the pointers stored in the arrays—is shown as a normal edge and not shaded and wide. Therefore, the set of pointers abstracted by the edge is *injective* and each index of each array points to a unique `Point` object. Given this likely ownership relation and the fact that all of the arrays are of length 4 it seems that flattening the *Face* data structure would reduce memory use substantially (i.e., this satisfies our conditions for being an *over factored* structure and would be flagged as such if we turned on the highlighting for the property).

By studying the source code for the `Face` class we can see that these ownership and length properties do in fact hold universally. Thus, we can flatten each `Point[4]` and associated `Point` objects into a `float[12]`. This transformation eliminates one object header per `Point` object (at 8 bytes each) and the 4 pointers stored in the `Point[4]` (at 4 bytes per pointer). Given that we have ∼72K `Point` objects and ∼18K `Point[]`, this change works out to ∼0.86MB of savings or ∼21% of the total live heap. Using similar reasoning we could further flatten the `float[12]` arrays into the `Face` implementations for another ∼0.22MB of savings, or another ∼5% of the live heap.

After these refactorings, the live memory consumption is reduced by a total of ∼1.58MB which is around 39% of the memory that was used by the baseline implementation.

[4] Unfortunately, ikvm is not able to process the remaining DaCapo benchmarks.

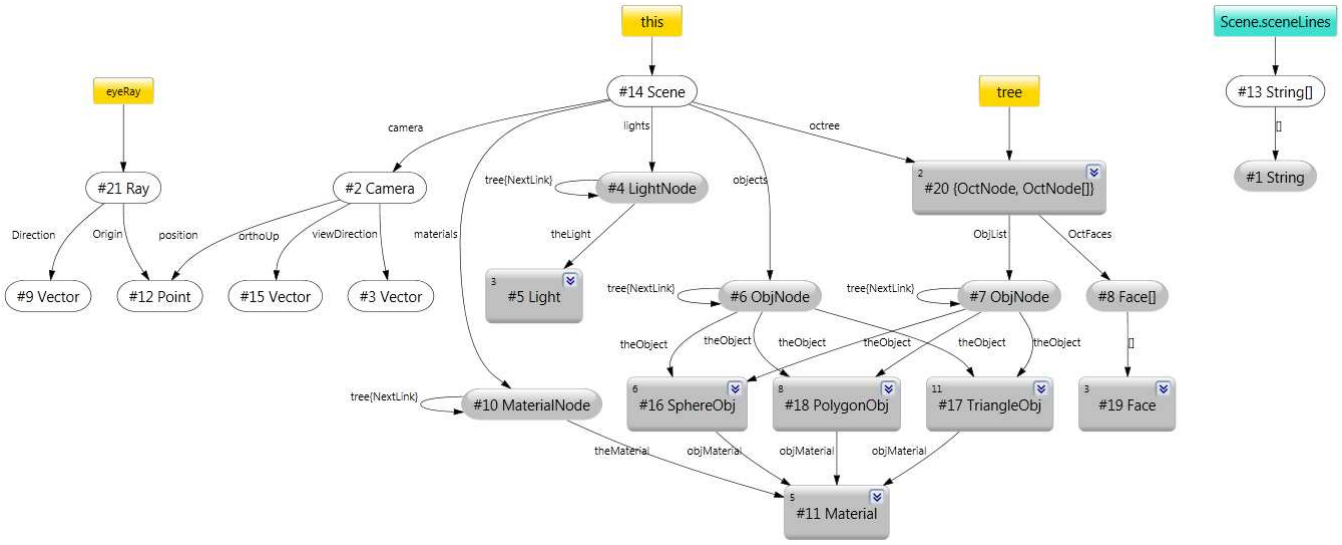

Figure 5. Debugger snapshot of `Shade` method in the `Scene` class, abstracting ∼168K objects.

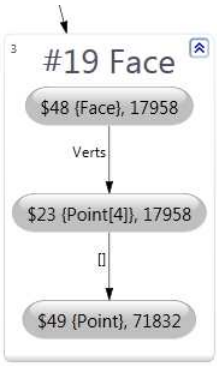

Figure 6. Memory Bloat

This case study shows how the multi-level abstraction allows the developer to navigate around the heap at the desired level of detail, zoom-in and out of specific areas of interest, all while maintaining the larger context. This ability to put the problem in context and interactively explore the heap is critical to aiding the developer in quickly identifying the source of a problem, understanding the larger context, and thus being confident in formulating a remedy.

6.3 Evaluation With Profiler

A number of recent papers have identified and explored memory use issues in the DaCapo benchmark suite. Hence, we decided to evaluate the effectiveness of the abstraction techniques described in this paper by using our profiling tool to analyze several programs from the DaCapo suite and our CCI based rewriter for memory utilization issues.

After running the profiler we inspected the output abstract graphs to find nodes (regions) that contained potential problems and then to determine what (if anything) could be done to resolve the issues or if the memory use appeared appropriate. This was done via manual inspection of the graph, the use of the heap inspection and highlighting tools in the profiler, and inspecting the associated source code. In all cases at most 7 nodes were colored by the profiler tools and the total time to inspect the graph, identify the relevant structures, inspect the associated source code, and determine if the memory use was generally appropriate ranged from 5 minutes to 10 minutes. Also, as we had not previously worked with the code, sometimes we needed to spend additional time to understand more about the intent of the classes and their structure in order to fully determine if the code could be successfully refactored and how. This was particularly important when multiple classes/subclasses were used to build recursive data structures. However, this inspection never required more than an additional 15 to 20 minutes.

Antlr. For the Antlr benchmark, the tool reports one of the larger heaps being reachable from a method in the `JavaCodeGenerator` class. We inspected this heap with our visualization turning on the memory use heat map, we were able to quickly identify one dominator node as containing around 72% of the reachable memory. This region was dominated by a set of `RuleSymbols` each of which stores information representing various aspects of the parser. Further inspection did not reveal any obvious memory use problems or obvious areas where data structures could be refactored to substantially improve memory utilization. These findings match those of previous studies of the benchmark which is not known to have any reported memory leaks and is reported to have good utilization of memory. In particular [25] reports a good *health* score for this benchmark.

Chart. For the Chart benchmark our tool reports the largest heaps being reachable from a method in the `JFreeChart` class. The coloring highlights a region that is dominated by a set of `XYSeries` objects. If we zoom into this region we see the structure shown in Fig. 7. The memory heat map coloring highlights the regions containing the `XYDataItem` and the `Double` objects they own. By hovering over these we see that they consume about 3MB of heap space. The actual data contained in these objects (in particular the `Double` objects) is small compared to the object overhead and since there is an ownership relation between each of the `XYDataItem` objects and the Double objects it points to (the edges are injective, i.e. not shaded and wide). This indicates that we may be able to inline these structures to save space. An inspection of the `XYDataItem` class shows that it declares the `x/y` fields as `Number` types to allow for some level of polymorphism. So we

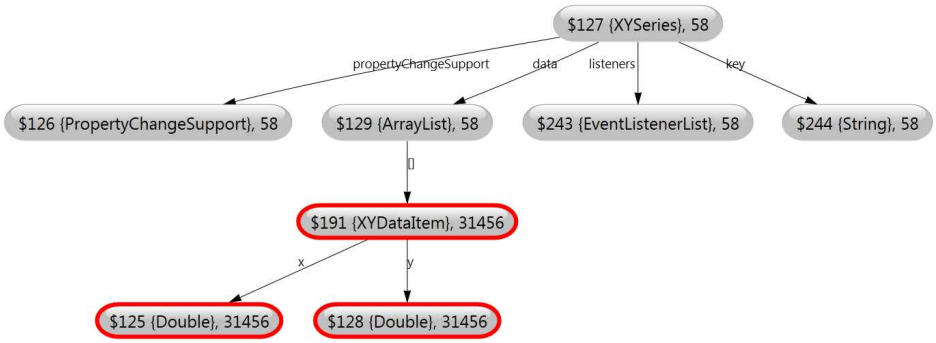

Figure 7. Chart Memory Use

need to subclass our new flattened classes to allow for storing both integer and floating point `x,y` pairs. This minor refactoring results in a savings of around 1MB, which is around 25% of the total live memory. Further savings are possible if more aggressive refactoring is done. To the best of our knowledge this memory issue has not been reported in previous work. This scenario shows again how the additional context around the memory use problem supports the identification and implementation of a solution.

FOP. For the fop benchmark the tool reports the largest heap being reachable from a method in the `Page` class. The highlighted region consists of a large number of objects that contains various parts of the document to render, for example, the `WordArea` and `TableArea` objects. After a some inspection of the source code we concluded that the data structure was not particularly amenable to refactoring. As reported in [16], we note that the data structure is needed later in the computation and thus is not a leak.

PMD. For the pmd program, our tool reports one of the larger heaps in the `JavaParser` class. The section highlighted by the memory utilization coloring is a summary node. This node uses over 10MB of memory and consists of a data structure which is a tree via the `children` field and container offsets, along with a parent back pointer on the `parent` field. This data structure represents the AST of the program that is being analyzed. Hovering over the node reports that it represents more than 50 types (with names like `ASTExpression` and `ASTPrimitiveType`) that all inherit from the `SimpleNode` class. On inspection we see that this base class has many data fields (line numbers, the `children` array, the `parent` field, etc.) which the subclasses add to with additional AST specific information. Given this structure we did not see any obviously poor memory use or memory leaks. This appears to contradict [25] which reports a high rate of object header overhead in this benchmark. However, we note that in this case the overhead is actually encoding important structural information about the AST that is being processed. Thus, this case study demonstrates how the visualization can be used to augment the information provided by existing analysis tools (and in this case may be useful in improving the quality of their reports by giving additional structural and contextual information).

CCI. For the CCI based rewriter program our tool showed that in general the code is well balanced in terms of memory used in various parts of the heap and the data structures are well designed. However, by turning on the highlighting for sparse arrays we quickly found one possible location for improvement. As part of reading a DLL, a `PEFileToObjectModel` object is created which initializes an array of `MethodDefinition` entries based on the number of declared methods in the DLL (and similarly for fields, types, etc.). These arrays are then filled on demand as the various names are accessed. However, if only a few names from a DLL are used these arrays can be quite sparse. For the particular case under study one DLL that is loaded contains 25,090 entries for methods but only 7 methods are referenced (thus there is a substantial space overhead for this case). Additional investigation indicated that the developer was aware of the potential space costs associated with this choice but had decided that, in this case, the overhead in memory usage was an acceptable cost to pay in order to have a simple and time efficient implementation. Further experimentation would be needed to determine if this space overhead is consistently high (e.g. the overhead is dependent on the number of methods actually accessed). However, this example shows how even the relatively simple metrics supplied by the profiler when combined with the ability to directly visualize the overall heap structure allows the rapid identification of memory use issues. As, in this case, in the span of 5 minutes we, having no previous experience with the internals of CCI, were able to identify this (good) candidate for reducing memory use in the program.

6.4 Industrial Experience

We have a small number of industrial users who have been evaluating the tools described in this paper on large production codebases. The initial feedback mirrors our experiences. Users have reported that while the time to run the profiler (and associated abstractions) is non-trivial it is not a major issue in using the tools. Similar to our case studies they have identified and fixed an range of memory inefficiency issues. They have also reported that they found it useful to manually explore the graph to see what the data structures in the programs look like and if this matches their intuition. In cases where there was a mismatch it was often due to unintentional sharing which had not yet appeared as a bug (e.g., partial copying). For both the memory inefficiencies and the reachability items the users indicate that they felt the issues would have been significantly more difficult to find and fix without a tool along like the one presented in this paper. In particular the general view is that the ability to see the global context of the memory state and the natural grouping of objects into data structures was very useful.

6.5 Computational Costs

In this section, we evaluate the cost of extracting and comparing abstract heap graphs during the execution of the debugger and profiler tools. We report the maximum times and sizes in the results. The timings were obtained on a 2.4GHz Intel Core2 series processor (single threaded) with 2GB of memory.

Fig. 8 contains the sizes of the largest abstract representations produced during the runs of the profiler tool. The first column lists the benchmark and the second column the number of objects in the largest concrete heap snapshot that was encountered. The following

Bench	Objects	AbsNode	Reduced
db	∼153K	12	10
raytracer	∼168K	48	21
antlr	∼12K	606	201
chart	∼189K	198	110
fop	∼120K	531	150
luindex	∼2K	87	36
pmd	∼178K	146	28
xalan	∼40K	451	127
cciwriter	∼53K	923	112

Figure 8. Max graph sizes.

Bench	Objects	AbsTime	CmpTime
db	∼153K	1.37s	0.01s
raytracer	∼168K	2.79s	0.04s
antlr	∼12K	0.41s	0.03s
chart	∼189K	3.22s	0.09s
fop	∼120K	2.67s	0.11s
luindex	∼2K	0.50s	0.01s
pmd	∼178K	4.11s	0.09s
xalan	∼40K	2.42s	0.07s
cciwriter	∼53K	2.20s	0.09s

Figure 9. Max times for operations.

columns are the size of the largest abstract heap graph produced for any heap snapshot (*AbsNode*), and the size of the corresponding dominator reduced representation from Sec. 5 (*Reduced*). Some of these sizes seem to be at (or beyond) the upper end of what can be conveniently visualized. However, our experience with in Sec. 6.2 shows the combination of the conditional graph styles, the ability to zoom between levels of detail, and the navigational tools provided by the DGML viewer made inspecting and understanding the relevant parts of the graphs quite easy.

The next issue we wanted to evaluate was the computational costs of performing the abstractions and comparison operations. The first two columns in Fig. 9 list the benchmark and size of the largest concrete heap encountered (excluding objects from the `System.*` or `java.*` namespace). The *AbsTime* column shows the maximum time taken to abstract a concrete heap during the profiler run while the *CmpTime* column shows the maximum time taken to compare an abstract heap to a previous version.

To enable easy experimentation and implementation the current abstraction implementation creates a complete shadow copy of the concrete heap during abstraction. Despite this large constant time overhead, the cost of computing the abstractions is quite tractable. The running time scales very closely to the asymptotic complexity of $O(E * log(N))$ from Sec. 3. The current implementation computes the abstraction inside the process that is instrumented, so it was not possible to precisely measure the exact memory overhead of the abstraction operations. However, using the difference in the total memory consumed by the process as reported by the system monitor indicates a factor of a $40\times$ increase in memory use (never exceeding 800MB). For our applications, including the calculation of the shadow heap creation, this overhead is quite acceptable. However, in other applications where the underlying heap may be hundreds of MB this overhead may not be acceptable. We note that the algorithm in Sec. 3 can be restructured to compute the equivalence classes online during the heap walk, thus eliminating the need to create a full shadow heap or even create an explicit equivalence class entry for each object in the heap. This algorithm will be evaluated in future research.

7. Related Work

Developing debugger support for the program heap is an ongoing active research area. The work in [36] outlines many of the basic issues that arise when attempting to visualize concrete program heaps and [29] presents some abstractions to help alleviate some of these issues. There is a large body of work on techniques to improve the efficiency and effectiveness of debugging [13, 18, 19, 28, 31, 35]. Work in [1] takes the same general approach as in this work but focuses on the interactive aspects of visualizing the heap, in particular on how to allow the developer to inspect individual objects as part of a larger structure.

Work by Mitchell et. al. [23, 24] has a number of similarities to the work in this paper. Both approaches use a set of grouping heuristics to partition structures in the heap and then extract information about the partitions, but the partitioning strategy and information extracted differ substantially. Our work uses recursive structures and predecessor ownership to identify equivalence classes of objects/data while [23, 24] focus on dominator relations between objects. We note that this results in the same asymptotic cost as the work in this paper. Given this difference of grouping heuristics there is also a natural difference in the focus on what type of information is extracted. In particular, the abstraction in this paper is designed to aid programmer understanding of the structure and connectivity of various heap structures and so it explicitly extracts information on shape, edge injectivity, pointer nullity, container sizes, in addition to information on the sizes of various data structures. While some of these properties can, in some cases, be reconstructed using *fanout* and object count information, the majority of the information computed in [23, 24] focuses the specific task of identifying memory inefficiencies in large Java programs.

There is a substantial amount of work on the development of heap models for use in static program analysis [2, 10, 17, 33]. Whereas program analysis is concerned with computability and obtaining enough precision at reasonable cost, the main challenge in abstracting runtime heaps is to obtain very small models that can be visualized, while retaining many useful properties of the original heap. We believe though that insights in static heap analysis can inform the abstractions of runtime heaps and vice versa. For example, it would be interesting to provide programmers with more control over the abstractions produced via instrumentation predicates [2, 33].

The approach in [17] uses a less descriptive model than the one presented in this paper for example, it does not consider information such as injectivity or shape. Work in [15, 30] use a related idea of taking a concrete heap from a C/C++ or Java program and inferring the types [30] or basic shapes [15] of heap structures.

8. Conclusion

This paper introduces a new runtime technique for program understanding, analysis and debugging. The abstraction of heap graphs presented attempts to construct a very small representation of the runtime heap in order to allow effective visualization and navigation, while retaining crucial high-level properties of the abstracted heap, such as edge relations and shape of various subgraphs. The construction of the abstraction ensures that the abstract graph is a *safe* representation of the concrete heap, allowing the programmer (or other tools) to confidently reason about the state of the memory by looking at the abstract representation. Furthermore, the use of an abstract domain enables the stable comparison and merging of abstract heap graphs obtained at different program points or different program runs. Our benchmarks and case studies demonstrate that abstract heap graphs can be efficiently computed, contain interesting information on the heap structure, and provide valuable information for identifying and correcting memory use related defects.

Given the utility of the abstraction in this task we believe there are a number of other applications including thread races, refactoring for parallelism, or interactive debugging, where this type of abstraction and understanding would be useful.

References

[1] E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. Su, and S. Guyer. Heapviz: interactive heap visualization for program understanding and debugging. In *SOFTVIS*, 2010.

[2] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[3] S. Blackburn, R. Garner, C. Hoffman, A. Khan, K. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis (2006-mr2). In *OOPSLA*, 2006.

[4] Common Compiler Infrastructure. `http://ccimetadata.codeplex.com`.

[5] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[6] D. Clarke, J. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, 1998.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[8] R. DeLine and K. Rowan. Code canvas: zooming towards better development environments. In *ICSE*, 2010.

[9] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond *k*-limiting. In *PLDI*, 1994.

[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 *POPL*, 1996.

[11] DGML Specification. `http://schemas.microsoft.com/vs/2009/dgml`.

[12] Heap abstraction code. `http://www.codeplex.com/heapdbg`.

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

[14] ikvm. `http://www.ikvm.net/`.

[15] M. Jump and K. McKinley. Dynamic shape analysis via degree metrics. In *ISMM*, 2009.

[16] M. Jump and K. S. McKinley. Cork: dynamic memory leak detection for garbage-collected languages. In *POPL*, 2007.

[17] C. Lattner, A. Lenharth, and V. S. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.

[18] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI*, 2005.

[19] C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff. Sober: statistical model-based bug localization. *SIGSOFT*, 30(5), 2005.

[20] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, 2005.

[21] M. Marron, D. Kapur, and M. Hermenegildo. Identification of logically related heap regions. In *ISMM*, 2009.

[22] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.

[23] N. Mitchell. The runtime structure of object ownership. In *ECOOP*, 2006.

[24] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *ECOOP*, 2009.

[25] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. In *OOPSLA*, 2007.

[26] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[27] F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., 1999.

[28] W. D. Pauw and G. Sevitsky. Visualizing reference patterns for solving memory leaks in java. In *ECOOP*, 1999.

[29] S. Pheng and C. Verbrugge. Dynamic data structure analysis for Java programs. In *ICPC*, 2006.

[30] M. Polishchuk, B. Liblit, and C. Schulze. Dynamic heap type inference for program understanding and debugging. In *POPL*, 2007.

[31] A. Potanin, J. Noble, and R. Biddle. Snapshot query-based debugging. In *ASWEC*, 2004.

[32] RiSE4Fun. `http://rise4fun.com/HeapDbg`.

[33] S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.

[34] Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.

[35] A. Zeller. Isolating cause-effect chains from computer programs. In *FSE*, 2002.

[36] T. Zimmermann and A. Zeller. Visualizing memory graphs. In *Software Visualization*, 2001.