Collecting a Heap of Shapes

Earl T. Barr[1,2] Christian Bird[3] Mark Marron[3,4]
[1]University College London, UK [2]UC Davis, USA [3]Microsoft Research, USA [4]IMDEA Software, Spain
e.barr@ucl.ac.uk, {christian.bird, marron}@microsoft.com

ABSTRACT

The program heap is fundamentally a simple mathematical concept — a set of objects and a connectivity relation on them. However, a large gap exists between the set of heap structures that could be constructed and those that programmers actually build. To understand this gap, we empirically study heap structures and sharing relations in large object-oriented programs. To scale and make sense of real world heaps, any analysis must employ abstraction; our abstraction groups sets of objects by role and the aliasing present in pointer sets. We find that the heaps of real-world programs are, in practice, fundamentally simple structures that are largely constructed from a small number of simple structures and sharing idioms, such as the sharing of immutable or unique (*e.g.* singleton) objects. For instance, we find that, under our abstraction, 53–75% of pointers build tree structures and we classify all but 7–18% of aliasing pointers. These results provide actionable information for rethinking the design of annotation systems, memory allocation/collection, and program analyses.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**] ; D.4.8 [**Performance**]: Measurements

General Terms: Measurement

Keywords: Dynamic Analysis, Heap Structure

1. INTRODUCTION

The program heap is fundamentally a simple mathematical concept consisting of a set of objects and a connectivity relation on them. This clean formalism lends itself well to the application of powerful deductive mathematical analyses. However, the formalisms — objects, pointers, types, and fields — that define the program heap in modern object-oriented languages such as Java or C# are fundamentally under-constrained; a large gap exists between the range of heap structures that are admissible under the weak constraints imposed by the type system and the possibly much more limited set of structures that programmers build in practice. Previous work has explored this gap using ownership and dominator relations applied to heaps collected from the commercial operation of large server-side Java applications [27, 28]. In this work, we further fill in this gap using HeapDbg [26], a Daikon-style dynamic invariant discovery tool tailored for the program heap [11]. We apply this tool to the DaCapo benchmarks [3], which encompass object-oriented programs selected to be representative of real client-side applications and , thus, their heap structures. The results of our empirical study of the DaCapo programs, validated against a selection of C# programs, indicate that, in practice, the heap is a fundamentally simple structure that is largely constructed from a small number of shapes and sharing idioms.

This result has substantial implications for programming language research, particularly type and annotation systems, memory management, and the design of static heap analysis techniques. Work in these areas generally considers the heap in an adversarial setting where the analysis, memory management technique, or specification system must effectively handle the entire range of complex heap structures that *could* appear instead of focusing on simple heaps that *do* appear in practice. For example, questions about the prevalence of recursive heap structures and the extent of aliasing between multiple data structures are central issues in the design of a heap analysis. The results in this paper on these questions have already guided the development of a hybrid heap analysis that is both computationally efficient and precise in practice [24]. We believe our results on sharing idioms can achieve similar success informing the design of ownership type systems. Results on the frequency of sharing of immutable objects and isolation have already demonstrated great utility [13]. Our detailed results on immutability/isolation, singletons and other global structures also point to further advances in type and annotation systems; they indicate that lightweight annotations that capture simple sharing relations show promise. As a final example, the results in this paper shed new light on the topic of region based garbage-collection and allocation [6, 18, 22]. In particular, the results on both the number of regions and how they are shared suggest possible avenues for the design of collectors/allocators that operate on individual heap regions. Thus, our results imply that a pessimistic view of the heap frequently does not reflect the reality of how object-oriented programs organize the heap and may artificially limit the utility of systems built under this assumption.

A major consideration when studying heap structures is the level of abstraction to employ. A natural idea is to study

ISSTA ’13, July 15 – 20, 2013, Lugano, Switzerland

how individual objects are connected and shared [16, 19, 21, 23]. We hypothesize that developers actually think primarily in terms of the roles that objects play and the relations between these roles rather than thinking in terms of individual objects. Further, we hypothesize that these relations are encoded in where pointers to the objects that play each role are stored, *i.e.* objects that play the same roles are stored together while objects that play different roles are segregated. This study is built on an abstraction that allows us to study heap structures, and their relations, in terms of roles.

The results in this paper show, with a high degree of statistical confidence, that for object-oriented programs: local ownership (a variant of ownership [7, 23]) is an important but not dominant organization concept (mean of 51% ±12% of types), that aggregation is the dominant form of composition (mean of 85% ±12% of types), and that a small set of developer-centric concepts organize all type sharing (mean of 88% ±4% of types can be precisely categorized). In particular, we show that the majority of sharing that actually occurs can be categorized using a small number of programming idioms, *viz.* contained (*i.e.* locally shared), global, unique, and immutable objects.

This paper makes the following contributions:

- We use runtime sampling to measure and statistically analyze the heaps produced by the DaCapo benchmarks [3] and a selection of C# programs;
- We provide evidence that *components*, as described in [26], usefully and closely correspond to the *roles* that developers assign to objects; and
- We identify a small number of sharing patterns, which are related to common programming idioms, such as the sharing of unique and immutable objects, that describe the majority of sharing that occurs in practice.

Our results confirm some commonly held beliefs about the heap — programmers avoid sharing and builtin containers are preferred to custom implementations — and provide actionable information — strict ownership is *not* the dominant form of organization and sharing generally occurs in a small number of idiomatic ways — for rethinking the design of annotation systems, the allocation/collection of memory, and the design of program analyses.

2. THEORY

Object-oriented programming provides two ways for a programmer to transform abstract concepts into classes: *is-a* and *has-a* relations. In this work, we focus on the question of how programmers use the *has-a* relation, *i.e.* encapsulation and aggregation, to organize objects in object-oriented programs. Simply stated, we ask "How do programmers organize the heaps of real world object-oriented programs?"

We hypothesize that developers often think of objects in terms of the roles they play in programs. These roles implicitly aggregate objects into conceptually related sets, as when a developer thinks of objects as a class or a collection of value and expression nodes simply as an "abstract syntax tree". Thus, we utilize a role-based heap abstraction, which mirrors the roles a programmer assigns to objects, based on the *has-a* relation plus the standard notions of recursive data-structure identification [8, 21, 27, 32], predecessors [2, 5, 8, 25, 33], and grouping container contents [5, 9, 25].

Conceptual Components. In our formulation, objects are structurally *indistinguishable* if they 1) are members of the same *spine* [32] of a recursive structure or 2) have the same type and are stored together. Informally, objects are stored together when they are stored in the same container or when they have the same type and share a predecessor.

These structural indistinguishability principles were first formalized and realized in *HeapDbg*, the heap analysis tool on which this empirical study is based [26]. Formally, the HeapDbg abstraction models the state of a program heap with an environment, mapping variables to addresses, and a store, mapping addresses to objects. An instance of an environment paired with a store is a *heap*. Given $\mathsf{ProgramTypes}$, the types a program uses, HeapDbg defines the set of concrete labels in the program, $\mathsf{StorageLabels}$, as the set of all member fields and array indices in the program. HeapDbg then constructs a heap as the tuple $(\mathsf{Env}, \sigma, \mathsf{Ob})$ where

$$\begin{aligned}\mathsf{Env} \in \text{Environment} &= \mathsf{Vars} \rightharpoonup \mathsf{Addresses}\\ \sigma \in \text{Store} &= \mathsf{Addresses} \rightarrow \text{Objects} \cup \{\mathsf{null}\}\\ \text{Objects} &= \mathsf{ProgramTypes} \times (\mathsf{StorageLabels} \rightharpoonup \mathsf{Addresses})\\ \mathsf{Ob} &\in 2^{\text{Objects}}.\end{aligned}$$

Each object $o \in \mathsf{Ob}$ pairs the type of the object with a map from the object's field labels to addresses. HeapDbg assumes that the objects in Ob and the variables in the environment Env, as well as the values stored in them, are well-typed. The usual notation $o.l$ refers to the value of the field (or array index) l in the object o.

A *conceptual component* $C \subseteq \mathsf{Ob}$ is a partition of the heap objects, built by applying congruence closure to a formalization of the structural indistinguishability principles. Using conceptual components as nodes, we then build a storage-shape [5] (or points-to) graph whose edges are sets of pointers between objects in conceptual components. HeapDbg extends this basic model with *injectivity* (non-aliasing) annotations on the edges and *shape* annotations on the nodes [26].

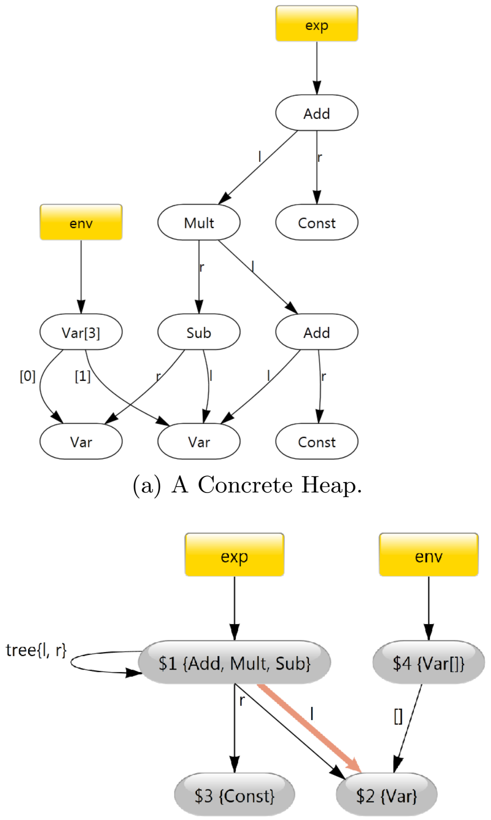
Pointer Injectivity. We can view a set of pointers as a function that maps one set of objects to another. A function is injective, or one-to-one, when it maps each distinct element in its domain to a distinct element in its range. The edges in the conceptual component graph abstract just such sets of pointers. When the pointers an edge abstracts are all *pairwise unaliased*, that edge is *injective*; otherwise, it contains aliasing pointers and is *non-injective*.

In the heap $(\mathsf{Env}, \sigma, \mathsf{Ob})$, the set of non-null pointers from C_1 to C_2 is $P_+(C_1, C_2, \sigma) \Leftrightarrow \{(o, l, \sigma(o.l)) \mid o \in C_1 \wedge \sigma(o.l) \in C_2\}$. Since C_2 is a partition of the concrete heap, it is nonempty and does not contain null by definition, so $\sigma(o.l) \neq \mathsf{null}$. Given two distinct conceptual components C_1 and C_2 in the heap $(\mathsf{Env}, \sigma, \mathsf{Ob})$, the non-null pointers with the label l from C_1 to C_2 are *injective*:

$$\mathsf{inj}(C_1, C_2, l, \sigma) \Leftrightarrow \forall (o_s, l, o_t), (o'_s, l, o'_t) \in P_+(C_1, C_2, \sigma) : \\ o_s \neq o'_s \Rightarrow o_t \neq o'_t.$$

This definition of the injectivity of the pointer sets that form edges in the graph of conceptual components elegantly generalizes to arrays. The key to this generalization is to replace labels with indices, then require distinct indices to point to distinct objects.

The notion of injectivity is very strong. It asserts an absence of aliasing among a set of pointers, and perhaps counter-intuitively, it holds for the vast majority of the non-



(a) A Concrete Heap.

(b) Corresponding Abstract Heap.

Figure 1: Running example.

null pointer sets HeapDbg identifies (Section 4). Thus, the ability to model this precisely is crucial for precisely capturing real world heap structures.

Shape. We characterize the shape of components using standard graph theoretic notions of trees and directed-acyclic graphs (dags), treating the objects as vertices in a graph and non-null pointers as the (labeled) edge set. In this style of definition, 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 the conceptual component C

any(C) holds for any graph.
dag(C) holds if the subheap restricted to C is acyclic.
tree(C) holds if dag(C) holds and the subheap restricted to C contains no pointers that create cross edges, *i.e.* pointers to the same object.
none(C) holds if the edge set restricted to C is empty.

2.1 Illustrative Example

Figure 1 shows the output of HeapDbg; it illustrates how we apply these principles to the concrete heap of a program that manipulates arithmetic expression trees[1]. Figure 1(a) shows a concrete heap snapshot HeapDbg computes on this program. The expression nodes have `l` and `r` operand fields. The local variable `exp` points to an expression tree consisting of four interior binary expression objects and four leaves — two `Var` and two `Const` objects. For efficiency, the program has interned all variable names into the `env` array to avoid string comparison during expression evaluation.

Figure 1(b) shows the graph of conceptual components and relations between them as produced by the application of the indistinguishability principles. To ease discussion, we label each node graph with a unique id. The abstraction summarizes the concrete objects into four components, which become nodes in the graph: 1) a node representing all interior recursive objects in the expression tree (*viz.* `Add`, `Mult`, `Sub`), 2) a node representing the two `Var` objects, 3) a node representing the two `Const` objects, and 4) a node representing the environment array. The recursive spine indistinguishability principle groups the four expression objects into node $1 in Figure 1(b). The container indistinguishability principle groups the two `Var` objects into node $2 and the two `Const` into node $3. They are not abstracted into a single component because their type distinguishes them. Since no principle applies to the environment array `env`, it acquires its own node $4. The edges represent sets of pointers and their associated field labels. The edges into node $2 are discussed in *Ownership* and *Research Question 3*.

2.2 Research Questions

Given the conceptual components, a natural question is

Research Question 1: What proportion of conceptual components are simple *vs.* recursive?

Here, a conceptual component is simple when it is a set, without internal relations, and complex when it abstracts objects that form structures such as trees or cyclic graphs. Answering this question provides insight into the relative importance of inductive *vs.* set based reasoning in shape analysis tools [2, 9]. It also provides insight into the role that recursive structures and container libraries play in the design of programs; specifically, it answers the question "Are simple recursive structures defined and used frequently or do programmers tend to define a small number of application specific recursive structures and otherwise avoid recursive definitions in favor of builtin collections?"

Ownership Encapsulation is a fundamental concept in OOP and has traditionally been expressed as a binary property in terms of *ownership* [7], *i.e.* all paths from the root of a system to an object must pass through that object's owner. This strict definition with transitivity leads to the same issues as encountered in the classic `const` problem where use of `const` in one location cascades into its required use throughout a program. Here, we utilize the slightly weaker notion of *local ownership* (similar to [23]). A set of pointers that do not alias is *injective*, *i.e.* the set is a one-to-one map of pointers to objects. A conceptual component is *locally owned* if it has a single, injective in-edge. Informally, a single pointer points to each of the objects in a locally owned component, even if objects transitively reachable from one of these objects may be shared. Under this definition, transitive sharing does not obscure the fact that some data may be encapsulated in a locally owned object. Figure 1(b) shows that the concrete `Const` objects are always locally owned, since they are contained in a single conceptual component with a single (narrow), injective in-edge.

Questions about ownership, local ownership, and sharing are fundamental throughout research in programming language design [4, 7, 12], memory management [14, 15, 22],

[1] The code for this program can be downloaded from [17].

and program analysis [2, 9, 23, 33]. Despite a number of valuable studies [16, 19, 21, 23, 27, 30], the question of what sharing is actually present in real-world programs and why this sharing occurs is still an open question. In programming language design, there is substantial interest in developing type or annotation systems that can express rich sharing, encapsulation, and exposure properties relevant to real-world programs. Sharing (non-sharing) information can also improve both the layout and eventual collection of object structures.

In this study, we hypothesize that ownership in object-oriented programs is important but that a non-trivial amount of sharing also occurs. Thus, we first want to understand how common local ownership is.

> **Research Question 2:** What proportion of objects are locally owned?

Sharing. Sharing occurs when objects in different components contain pointers to the same object or when multiple objects in the same component contain pointers to the same object. In the first case, the sharing likely involves objects of multiple types or at least objects that play different roles in the program; in the second case, the sharing likely involves objects of a single type that all play the same roles.

> **Research Question 3:** What proportion of sharing occurs between objects in the same conceptual component *vs.* across conceptual components?

In Figure 1(b), several expression objects point to the same `Var` object; this aliasing (*non-injectivity*) is depicted using wide, orange edges, if color is available. Multiple incoming edges to a node are *cross* edges. The node \$2, which abstracts the `Var` objects, exhibits both types of sharing and therefore has cross edges: multiple objects within the tree component `expr` alias `Var` objects with \$2 and the environment array `env` also points to `Var` objects within \$2.

To understand why sharing occurs, we examine the non-injective and cross edges through the lens of common programming idioms. Our first idiom is based on the notion that a key role of many classes is to aggregate and provide appropriate views of the contained data. This often requires the resulting objects to store data in multiple ways. For example a class may store the same objects in both a `List` and a `HashSet`. Objects in such a class are shared but a single class closely manages their sharing. We consider sharing to be localized if, in all cases, a unique dominator recaptures the shared objects within no more than two pointer dereferences; we call such recaptured objects *contained* objects. Another common idiom is the use of objects, like singletons (unique) or intern tables (global), which map objects, typically strings, to references which are then used in place of the object for efficient storage or equality testing. The final idiom we look at is the sharing of immutable objects such as strings in C# and Java. When the objects are known to be immutable, developers are much less concerned about sharing them and often do so intentionally for performance reasons. First, we classify sharing in terms of these idioms:

> **Research Question 4:** What proportion of sharing involves 1) contained, 2) global, 3) unique, or 3) immutable objects?

We hypothesize that, in practice, these types of sharing dominate the sharing in real-world programs, and ask:

> **Research Question 5:** What proportion of sharing relationships remain unclassified?

The answer to this question has direct implications for the design of both annotation (or type) systems and static heap analysis tools. If much of the heap remains unclassified, then more expressive (and unappealing to practitioners) annotations will be needed and static heap analysis tools must be both deep and broad. If, on the other hand, our classification scheme captures most of the sharing in the heap, we will have shown that it is possible to relate idiomatic code designs to the heap structures they produce and that, in practice, programmers form and combine the components in a small number of simple and often idiomatic ways. This means that an annotation system or analysis tool that captures these idioms will be able to precisely and compactly annotate (analyze) the features that dominate real-world heaps. Further, since these systems would be built on a small number of concepts and designed to reflect programmer intent, they should be simple and intuitive for programmers to use and relatively easy to implement in a static (or dynamic) heap analysis tool.

Abstraction Hypothesis This work empirically explores how developers translate informal design specifications into class definitions. The following hypothesis underpins our analysis: *Conceptual components, defined using our indistinguishability principles, accurately*[2] *partition the heap.* If this hypothesis does not hold then we would expect the partitions to contain unrelated objects and the resulting measurements of their properties to produce low information, indeterminate values. However, the results in Section 4 show very strong biases towards high accuracy properties. Thus, we have confidence that the conceptual component partitioning correctly identifies and abstracts the relevant parts of the heap.

3. METHODOLOGY

This work explores what structures real world programs build; in particular, we are interested in features that express developer intent, *e.g.* class invariants. This is why we based this study on HeapDbg, a Daikon-style dynamic invariant discovery tool tailored for the program heap [17, 26]. As HeapDbg operates on .Net bytecode, we must first translate Java programs into .Net bytecode using the ikvm compiler [20] before applying it.

HeapDbg extracts heap information, at program points and from those parts of the heap that are involved in these invariants. These points are typically the entry/exit of public methods and, in the heap, all objects reachable from method parameters and in-scope static fields. A heap snapshot is the set of locations reachable from static roots and the parameters of the current method call. At the entry of every public method, HeapDbg injects sampling code whose firing computes, abstracts, and aggregates a heap snapshot. Since extracting heap snapshots at each method call is impractical, HeapDbg uses a per-method randomized approach with an exponential backoff. When the current heap snapshot and

[2] Here, we use the definition of accuracy from measurement theory, *i.e.* closeness of a measurement to the actual value.

previously taken snapshots differ with respect to the components and the relations on these components, HeapDbg samples frequently; when no new information is discovered, it reduces the sampling rate. On smaller runs, we compared the results obtained by sampling uniformly at random with the results from the exponential backoff approach and found that the uniform sampling approach produced results that were no more useful.

To compute the likely pre/post invariant heap states for each method in the analyzed program, our profiler 1) copies the current heap state, 2) computes the corresponding conceptual component graph, 3) compares this graph with the previously seen graphs to update the sample rates, and 4) if it is new, adds the graph to the accumulated set.

3.1 Measurement

A program may use some of its classes quite heavily; for example, instances of a point class usually dominate the heap of a raytracer implementation. If we weighted our measurements by object frequency, our results would be heavily biased toward the features of such classes, which tend to be simple. To avoid this bias, we ignore object (and class use) frequency by discarding component graphs that are subgraphs of a larger, previously seen component graph. For each program, HeapDbg therefore produces a set of structurally distinct snapshots. Our analysis generates labeled graphs, over which checking for subgraphs is quadratic in the worst case [26]. In Figure 1(b), we discard the graph computed for the `Const` objects since it is a subgraph of the graph computed for the `Add` class. Finally, we take the set of component graphs produced by the runtime sampling and compute the measurements presented in Section 4.

We measure properties both in terms of both nodes/edges (for designing a static analysis) and types/fields (for constructing an annotation system). Our sampling methodology ensures that each retained snapshot is distinct. To compute a ratio for a single program, we compute the ratio of all nodes/edges that have a given property over the total number of nodes/edges across all the distinct snapshots that HeapDbg retained from an analysis run. For types, we report the ratio of the number of types to the total number of instantiated types. Depending on the property, its satisfaction may require *any* or *all* of the nodes that contain a given type to satisfy it. All of our properties obey a linear order: for edge category, *tree* $\preceq$ *cross* $\preceq$ *back* and, for sharing, *injective* $\preceq$ *non-injective.* We use the convention that the property of a type/field is the least upperbound of *all* nodes/edges of that type/field. In Figure 1(b), the `r` field is associated with 2 edges, a TreeEdge and a CrossEdge; thus under our construction, `r` is a CrossField.

Many of our research questions turn on whether two sample sets are drawn from different populations. To answer these question, we first checked that our data was normal via a Kolmogorov-Smirnov test [10] and then performed a t-test using the standard threshold of $p \leq 0.05$ for statistical significance. When we report a confidence interval on the average of a measure across all of the programs, we sum the measurements, computed as described above, for each program and divide by the number of programs. This unweighted average prevents a program with a disproportionately large heap from biasing the results. To avoid repetitive graphs, we do not show figures for all the statistical analyses we performed, as many questions are similar.

4. EVALUATION

HeapDbg, on which this study rests, was developed for use with .Net bytecode to understand memory usage and sharing problems in C# programs. Unfortunately, the C# programs that we initially considered for this study presented two problems. First, many of these programs are unfamiliar to the larger research community and some are proprietary. Second, since these programs have not been used in previous studies, it would have been prohibitively difficult to meaningfully compare our results with existing work on program optimization, analysis, specification, *etc.* Since we wanted to study programs both familiar and available to a broad segment of the research community so that our results could be meaningfully compared with previous work, we partially reworked HeapDbg to analyze programs from the well-known DaCapo [3] suite.

The DaCapo suite is designed to be representative of realistic program workloads with an emphasis on client-side applications. Its authors selected programs and representative, real-world inputs to span a range of application domains, including text searching, database work, XML document processing, program analysis, *etc.* As a result, the DaCapo suite exhibits many different heap structures and code behaviors. To perform our study, we translated ten programs from the DaCapo suite into .Net bytecode using the ikvm compiler [20] (we only omit DaCapo programs that ikvm was unable to compile) and ran them on the DaCapo suite's default inputs.

Shapes Here, we explore what sorts of heap structures conceptual components contain. For this purpose, we define the following predicates on the conceptual components. Given the abstract heap graph $G = (N, E)$, the conceptual component $n \in N$ has shape

Atomic iff $\mathsf{Shape}(n) = \mathsf{none}$, *i.e.*, there are no pointers between any of the objects in n.

Linear iff $\mathsf{Shape}(n) = \mathsf{tree} \lor \mathsf{Shape}(n) = \mathsf{dag}$[3].

Cyclic iff $\exists$ an SCC of the objects in n.

To clarify these examples, consider Figure 1. It contains three nodes with atomic shape — the nodes representing the `Const`, `Var`, and `Var[]` objects — and three corresponding atomic shape types. It contains one node whose shape is linear, the node with the self edges (labeled *tree(l, r)*) that represents the `Add`, `Mult`, and `Sub` objects. Since this node represents multiple types, there are three linear shape types. No nodes in Figure 1 have self-edges labeled *any*, so it has no cyclic nodes or types.

Armed with these predicates, we can measure the proportion of components of each shape, and answer **RQ1:** *What proportion of conceptual components are simple* vs. *recursive?* Figure 2 shows the ratios of compositional data structures (Atomic), simple recursive data structures such as trees or dags (Linear), and more complex cyclic structures (Cyclic). We measure these ratios both in terms of the number of types that appear — at any time — in a recursive structure and the number of nodes that have the given shape. In our analysis, a type inherits its graph theoretic properties from the components in which it participates; for instance, the degree of a type is the maximum degree of all components in which it participates.

As can be seen, atomic shapes dominate the other, more complex components. To assess the significance of this find-

[3] Linear does not reduce to $\mathsf{Shape}(n) = \mathsf{dag}$, since trees are undirected and a dag may contain an undirected cycle.

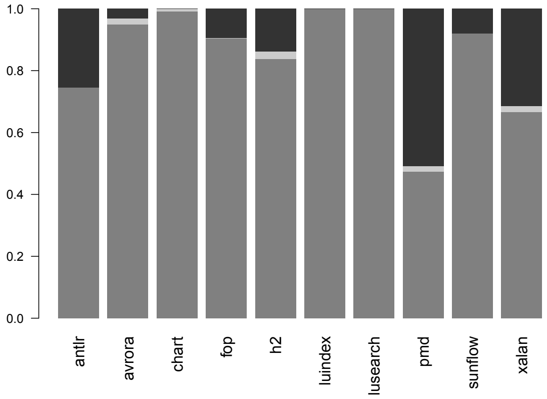


Figure 2: Percentage of types whose shape is cyclic ■, linear □, or atomic ▨.

ing, we compared the proportions of atomic shapes to linear and cyclic shapes with a two sample t-test. Our sample size of ten projects is fairly small for inferential analysis, but still allows for significant results if the differences are extreme with low variance. Such is the case for our type shape analysis where we found that atomic shapes dominate to a statistically significant degree ($p \ll 0.01$), with a 95% confidence interval for the proportion of shapes that are atomic of 85%±12%. Turning to comparing the proportion of cyclic to linear shapes, a t-test shows that cyclic shapes occur more often than linear shapes to a statistically significant degree ($p = 0.03$).

When we analyze the ratios of shapes per component, atomic shapes again dominate all other shapes to a large degree ($p \ll 0.01$, not shown). In fact, the confidence interval for the proportion of atomic shapes over conceptual components is 98.5%±1%. In contrast to type shapes, neither cyclic nor linear shapes over components were more prevalent than the other to a statistically significant degree ($p > 0.05$). In short, the answer to **RQ1** is that simple, conceptual components dominate recursive components: for types, the proportion of atomic shapes is 85%±12%; for components, 98.5%±1%.

Although the use of recursive structures in the program is limited to a few components, these components can involve a large number of types (*e.g.*, antlr, pmd, and xalan). This result is not surprising as object-oriented programming languages 1) often provide extensive container libraries which are used in lieu of custom list and tree structures and 2) support for *is-a* relations that allow an application to closely model underlying problem domain relations in the data structures, as seen in the abstract syntax tree in pmd.

Graph Structure and Ownership Ownership is a structural property of graphs. Let $d^+ : N \to \mathbb{N}$ denote the in-degree, including self edges, of a node. To explore ownership, we first classify the edge $e \in E$ that ends at n_t as

Internal iff e is a self edge.

External iff e is not a self edge.

Injective iff e is external and $\mathsf{Inj}(e)$, *i.e.*, e contains no aliasing concrete pointers, where Inj is defined in Section 2.

TreeEdge iff e is external and $d^+(n_t) = 1$.

CrossEdge iff e is external and $d^+(n_t) > 1$.

BackEdge iff e is an external edge a DFS from the program root set would label e a *back* edge.

Figure 1, our running example, contains only one internal edge, the self-edge on the expression tree `exp` and, since this edge represents pointers in the `l` and `r` fields of `Exp`, it contains two *Internal* fields. It contains four external edges: the three outgoing edges from `exp` and the edge representing the pointers in the `Var[]`. However, since the `l` and `r` fields also appear as *Internal* fields, there is only one external field. It contains four *Injective* edges, the local `exp` variable edge, the static field `env`, the edge representing the pointers stored in the environment array that refer to `Var` objects and the pointers in the expression tree that refer to constant objects. Since these edges represent pointers stored in the `r` and `[]` fields, the example contains two injective fields. It contains three *TreeEdges* — the local `exp` variable edge, the static field `env`, and the `r` edge pointing to the `Const` objects.

Defined using these predicates, the node n is *locally owned* when its in-edge e is *Injective* and a *TreeEdge*. When a conceptual component is locally owned, a unique pointer points to each of the objects in it. In Figure 1, two nodes — of type `Var[]` and `Const` — are locally owned. Since the edge with the `r` label ending at `$3`, the node containing the `Const` objects, is injective and is `$3`'s only in-edge `r` *locally owns*, or is the *local owner* of, `$3`.

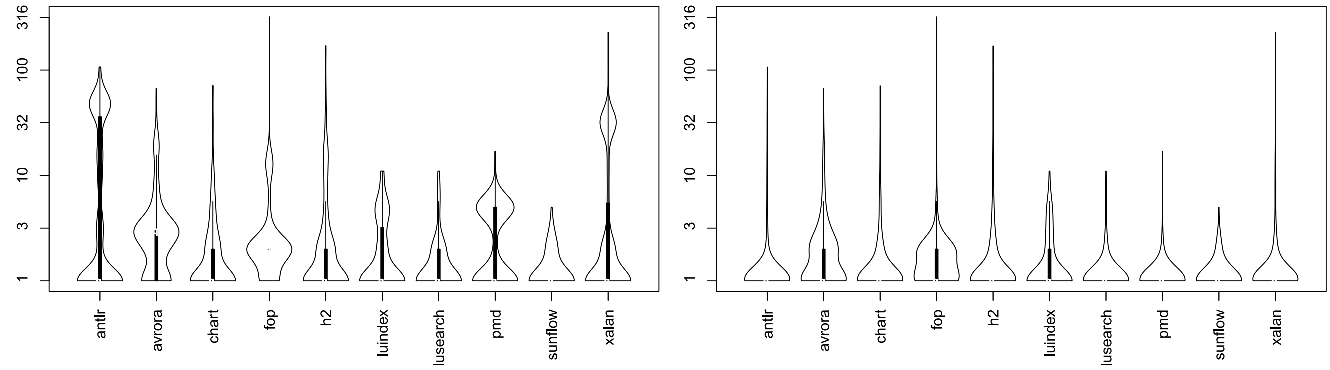
Figure 3 shows the distributions, via violin plots[4], of in-degree over the components and the types in them. While the number of components and types with in-degree k decreases fairly rapidly as k increases, a nontrivial number of components (types) with high in-degree occur. One reason is large recursive structures, with many types in the recursive structure, that have many in-edges to them. Thus, even though the in-degree of the individual objects is low, the overall in-degree of the structure of which they are members is high. A second contributing factor is large numbers of unique objects. The high portion of in-degree 2 types in Figure 3(a) is a result of this kind of structure. One outlier program, fop, has a large number of unique type objects that are also stored in dictionaries. Figure 3(a) provides initial insight into the value and limitations of using local ownership to describe heap structures in real programs.

To understand how much ownership (Section 2) exists that researchers (and eventually practitioners) can expect to exploit to improve program analysis or to build annotation systems, we examine its prevalence to answer **RQ2:** *What proportion of objects are locally owned?*

Consider the components and types with in-degree 1. If their in-edge is injective, then we know a single pointer points to each object in the component or type. The fraction for types is around 51% on average (confidence interval 39%–63%), showing that local ownership is the organizing principle for many parts of the heap. However, the fact that the remaining 37%–61% of the types in the program have references to them stored in multiple locations shows that the principle of local ownership does not dominate real world heap structures. We see higher ratios of in-degree 1 with a confidence interval of 42%–66% for the components.

Figure 4 shows the percentage of fields (edges) in the heaps that are involved in the internal structure of a conceptual component and represent pointers that are always local owner pointers. Edges are created from each snapshot; their origin is a concrete field in their source conceptual component. Here, we classify a field as a lattice join on the pointer classifications of the edges in which the field participates.

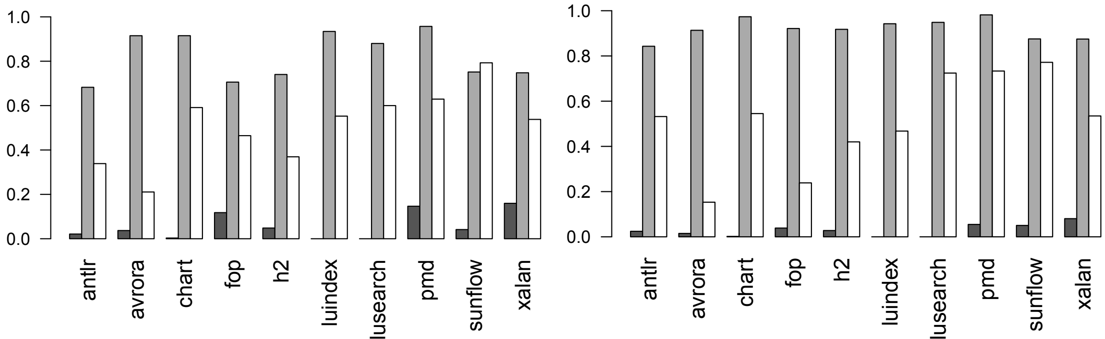
[4] A violin plot is similar to a box plot in that it compares distributions, but gives a more detailed view of the shape of the distribution.



(a) In-degree distribution per type.

(b) In-degree distribution per component.

Figure 3: In-degree distributions (log-scale).



(a) Property per field.

(b) Property per component edge.

Figure 4: Percentage of pointers that are internal ■, nonnull ▨, or owners □.

For instance, there are two edges labeled 'r' in our running example, Figure 1. One edge is a tree edge and the other is a cross edge; because cross subsumes tree, we classify 'r' as a cross field. From these figures, we see that overall most of the connectivity in the program is through pointers (fields) between conceptual components instead of within a single component. Figure 4(a) shows that on average 50% of the fields in the program always contain pointers that locally own their target object.

Thus, our answer to **RQ2** is that local ownership, both in terms of fields and edges as well as types and components, is an important but not dominant organizing principle for data structures in object oriented programs. For fields, the calculated confidence interval on the true mean based on our ten project sample is 39%–63%. This ratio translates almost equivalently into the ratio of edges that represent these types of pointers, a mean of 51% with a confidence interval on the true mean of 36%–66%.

Sharing The non-dominance of ownership brings us to the issue of why and how sharing occurs in practice. Our component graphs represent sharing in two ways: either a node has an in-edge that is non-injective or it has multiple in-edges. To capture some of the most common sharing idioms we classify a conceptual component $n \in N$ as

Immutable iff $\forall \tau \in \mathsf{Type}(n), \tau$ is immutable.
System iff $\forall \tau \in \mathsf{Type}(n), \tau$ is a builtin.
Unique iff $|\mathsf{Type}(n)| = 1$ and $\exists$ a static field with a pointer to n and $\forall n' \in N - \{n\}, \mathsf{Type}(n) \cap \mathsf{Type}(n') = \emptyset$.
Global iff $|\mathsf{Type}(n)| = 1$ and $\exists$ a static field that holds a pointer to a container object that holds pointers to n and $\forall n' \in N - \{n\}, \mathsf{Type}(n) \cap \mathsf{Type}(n') = \emptyset$.

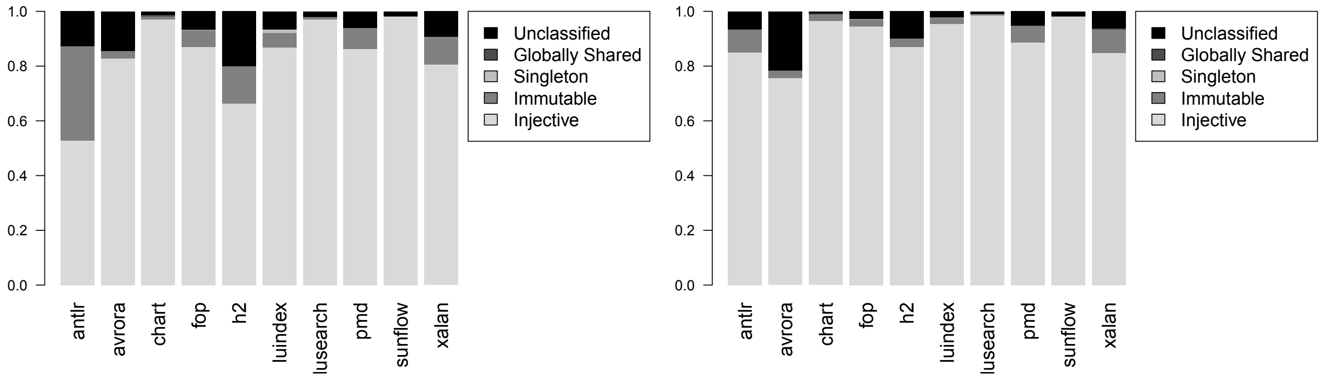
In Figure 1, none of the nodes contain immutable types so there are no immutable nodes. It contains one system type and one system node, the node representing the `Var[]`. The example contains no unique objects (thus no unique nodes or types). In Figure 1, the component containing the `Var` objects is globally shared, it only represents `Var` objects stored in an array to which the static field `env` refers. Thus, the example contains one globally shared node and one globally shared type.

When the edge $e = (n_s, n_t) \in E$ is external and $\neg\mathsf{Inj}(e)$ (*i.e.* non-injective), it can be further classified as

NonInjectiveToImmutable iff n_t is immutable.
NonInjectiveToUnique iff n_t is unique.
NonInjectiveToGlobal iff n_t is globally shared.

Figure 1 does not contain any immutable or unique components, so it has no edges (or fields) that interfere only on immutable or unique objects. The `Var` component is globally shared, so the example contains a non-injective edge, the `l` edge. Thus, we have one *NonInjGlobal* edge and one *NonInjGlobal* field.

Figure 5 classifies the injectivity of field and component edge pointers. This figure shows the ratio of fields and edges that always contain *injective* pointers to those fields and edges that at some point contain a *non-injective* pointers. In the *non-injective* case, it further classifies the sharing in terms of what is shared: immutable, unique, global, or otherwise unclassified objects. These figures show that most fields and edges are *injective* and, when they are not, it is frequently because they are sharing an immutable object. These two cases cover approximately 90% of all fields (with a confidence interval of 87%–96%) and 95% of all edges (con-



(a) Injectivity per field.

(b) Injectivity per component edge.

Figure 5: The percentage of field and edge pointers that are injective or non-injective, where the non-injective pointers are further subclassified into those that point at immutable, unique, global, or unclassified objects.

fidence interval of 90%–99%). The addition of unique and global objects pushes these numbers up a few percent.

We now turn to how multiple nodes and edges (type and field definitions) combine into larger structures on the heap. The first measure we examine looks at the prevalence of tree, cross, and back edges (Defined in Figure 4). We then use programming idioms to further breakdown the cross edges. Table 1 again shows field- and edge- centric classifications in the conceptual component graphs of each program. The far right columns shows the confidence interval for the mean proportion of the occurrence of each category in our sample. In most cases (with avrora as a clear exception), tree edges account for a slim majority of the fields (edges) in the heaps; in fact, a t-test indicates that tree offsets are the most common category of edge to a statistically significant degree ($p < 0.05$). Further these results show that cross edges appear quite frequently and dominate back edges to a statistically significant degree ($p < 0.05$).

We are now in a position to answer **RQ3:** *What proportion of sharing occurs between objects in the same conceptual component* vs. *across conceptual components?*. The non-dominance of local ownership in Figure 4 means that sharing is occurring regularly — the 95% confidence interval of the true mean of sharing is 37%–61% for fields and 34%–63% for edges. Tree edges dominate both the edge and field views of pointers. The only way a tree field or edge can exhibit sharing is through non-injectivity because tree edges are, by our definition, the only in-edge to their target. Thus, the high degree of injectivity in Figure 5 indicates that the sharing that we have observed is mostly due to components whose in-degree is greater than one, *viz.* either because of incoming cross or back edges. Therefore, most of the sharing we observe spans different conceptual components, as the prevalence of cross edges, the two rows marked † in Table 1, makes clear. Thus, we compute the raw count of cross fields (edges) over the raw count of shared fields (edges) to answer **RQ3**. Of all sharing, 77%-87% (the confidence interval for field) and 67%–77% (edge) occur between objects in different conceptual components while only 18%–42% (field) and 12%–28% (edge) occur between objects in the same conceptual component. A t-test indicates that sharing occurs more frequently between objects in the different conceptual components to a statistically significant degree ($p < 0.05$).

Although most of our benchmarks use back pointers sparingly (10% or less in most cases), antlr and avrora in particular make extensive use of them. Some standard idioms we observed in the code include implicit `this` pointers in inner class definitions, the observer pattern, and parent pointer idioms. Unfortunately, our abstraction currently does not capture the must-alias semantics needed to precisely categorize these back-pointers. Thus, we leave further investigation of back pointers to future work.

Classified Sharing Having answered **RQ3**, we turn to **RQ4:** *What proportion of sharing involves 1) contained 2) global, 3) unique, or 4) immutable objects?* To answer this question, we introduce additional edge predicates. The cross edge $e \in E$ that ends at node n_t is

CrossToImmutable iff n_t is immutable.
CrossToUnique iff n_t is unique.
CrossToGlobal iff n_t is globally shared.
CrossToContained iff $\exists n_d \in N$ s.t. n_d *dominates* n_t and the longest acyclic path from n_d to n_t has two or fewer edges, *i.e.* the sharing is highly localized.

Intuitively, cross edges represent aliasing in the abstract graph, as opposed to aliasing (non-injectivity) in the pointer set abstracted into a single abstract edge. Figure 1 has one `TreeField`, the static field `env`; the `r` field is not a tree field since its label also appears on non-tree edges. It has three *CrossEdges* all ending at `Var`. Thus, it contains three *CrossFields* in the heap — the `l, r` fields as well as the `[]` field from the array. It contains a *globally shared* node, `Var`, and the cross edges that end at this node, *viz.* the `l, r` edges from the expression tree, and the `[]` edge from the `Var[]`. Thus, we have three *CrossToGlobal* edges and, since all the other edges labeled with `l`, `r`, or `[]` are either *TreeEdges* or are internal tree edges, we have three *CrossGlobal* fields. Our example does not contain any back edges, immutable or unique nodes, so it does not illustrate the *BackEdge*, *CrossImmutable*, or *CrossUnique* features.

Table 1 shows a classification of cross edges based on their involvement in programmatic idioms, again as a function of both field declarations and edges in the conceptual component graph. We did not collect statistics on back edges and leave their further investigation for future work. Thus, our answer to **RQ3** is restricted to cross fields and edges because they are the most general, *i.e.* when a field participates in both a cross and a tree edge, its classification is cross. Pointers to immutable objects account for the largest fraction of cross edges, as the rows designated with $\star$ make clear. Indeed, Figure 5 shows that immutable objects are a

Table 1: Sharing classification per field and per component edge. The per field and per edge columns sum to 100%. The cross field and cross rows groupings further categorize the cross edges using the properties defined in Table 4; their columns do not always sum to their cross row entry marked † because the sharing categories are not disjoint. The '*' designates a category that occurs more frequently than the categories below it (within a grouping) to a statistically significant degree.

Classification		antlr	avrora	chart	fop	h2	luindex	lusearch	pmd	sunflow	xalan	Confidence Interval
Per Field	Tree	56%	25%	63%	56%	52%	59%	69%	73%	83%	59%	49%–71%*
	Cross†	25%	24%	31%	36%	29%	38%	29%	25%	16%	27%	24%–32%*
	Back	19%	51%	6%	8%	2%	3%	2%	17%	12%	14%	2%–23%
Per Edge	Tree	62%	42%	66%	48%	55%	61%	80%	81%	88%	63%	53%–75%*
	Cross†	26%	38%	31%	51%	32%	38%	19%	18%	12%	28%	21%–37%*
	Back	11%	20%	3%	1%	13%	1%	1%	1%	0%	9%	1%–11%
Cross Field	Contained	2%	3%	10%	6%	3%	11%	7%	6%	8%	4%	4%–8%
	Global	0%	0%	0%	3%	0%	1%	0%	0%	0%	0%	0%–1%
	Unique	0%	0%	3%	1%	0%	7%	0%	0%	0%	0%	0%–3%
	Immutable*	21%	5%	5%	10%	12%	11%	9%	16%	1%	14%	6%–15%
	Unclassified	4%	17%	16%	16%	15%	14%	13%	4%	8%	10%	8%–15%
Cross Edge	Contained	0%	6%	4%	2%	2%	10%	2%	2%	8%	3%	2%–6%
	Global	0%	0%	0%	1%	0%	5%	0%	0%	0%	0%	0%–2%
	Unique	0%	0%	1%	2%	0%	2%	0%	0%	0%	0%	0%–6%
	Immutable*	17%	8%	6%	24%	12%	12%	7%	12%	1%	17%	7%–16%
	Unclassified	9%	28%	21%	5%	19%	16%	9%	5%	4%	9%	7%–18%

major source of the cross edge sharing that occurs in practice (confidence interval 6%–15% of fields and 7%–16% of the edges). The "Contained" category shows that our relatively simple definition of localized sharing captures contained objects and shows that they form an important structure in these programs, accounting for 4%–8% on a field basis and 2%–6% of the edges. Finally, unique and globally shared objects are the least frequent, comprising 0%–4% (field) and 0%–8% (edge). Thus, our answer to **RQ4** is, of all sharing, immutable objects account for the majority of sharing (via a t-test with $p < 0.05$), with unique, global, and contained objects representing a smaller amount of the sharing.

Unclassified Sharing Combined, our answers to **RQ3** and **RQ4** conclusively show that, although the sharing relations in the heap can be arbitrarily complex in theory, they are overwhelmingly simple in practice and can be mapped back to common development idioms. The fact that our sampling methodology is biased against frequently instantiated classes, which tend to be simple, further strengthens this result. One simple threat to this result is if our abstraction failed to classify a large percentage of sharing relationships, so we conclude with **RQ5:** *What proportion of sharing relationships remain unclassified?* In short, **RQ5** is a measure of the effectiveness of our abstraction. For instance, good classification coverage is necessary to use these results as a basis for designing annotations to express the simple structures that our results indicate dominate heaps.

When we consider how much sharing information our current categorization scheme captures in Table 1, we see that our abstraction captures at least 72% of fields and in many cases, over 90% of the sharing relations of fields. The confidence interval for the mean proportion of relations that our approach leaves unclassified is 8%–15%. For edges, the breakdown is more variable but the mean proportion of unclassified edges is only 7%–18%. Further, our analysis is unaware of user-defined immutable types or the sharing of a unique or global object not captured by our current measurements, so some portion of the sharing we report as uncategorized is actually simple and well-behaved. While further study is warranted to investigate other common sharing patterns, this study demonstrates that a surprisingly large percentage of the heap in real world programs exhibits relatively simple structure.

Conceptual Component Accuracy Our results rest on accurately identifying a programmer's intended groupings of heap objects. We used the structural indistinguishability principles from Section 2 to approximate these grouping as equivalence classes of heap objects. Thus, our results rest on the abstraction hypothesis: *conceptual components, defined using our indistinguishability principles, accurately partition the heap.* Our abstraction could fail in two ways: it could generate conceptual components that lose structural information or it could generate components that do not reflect programmer intent.

If our abstraction lost structural information, we posit that our results would be much noisier, because it would tend to group together unrelated objects. However, this is not the case: our results contain strong signal for each of the reported measures, the measurements correlate with widely used program analysis concepts such as aliasing and shape and they are ordered in terms of their information content. For example the tree measurement contains more information than a dag measurement — *i.e.* it is a more restrictive property as $\mathsf{tree}(n) \Rightarrow \mathsf{dag}(n)$. Also, in most of our reported measures, the simpler (from a programmer standpoint) outcome dominates the other outcomes to a statistically significant degree. For example, the *injective* result dominates the lower information content *non-injective* result even though in a uniformly generated random partitioning we would expect the *non-injective* result to be more frequent. Finally, we note that we can precisely describe most of the remaining sharing with a small set of categories motivated by programming idioms. This fact provides strong evidence that our abstraction is accurately capturing features of the heaps that real-world programs build.

It is possible that, although the computed conceptual components effectively capture actual heap structures, these structures do not correspond to how a developer thinks about a program's heap. Although further study is needed, we present two reasons to believe that our indistinguishability principles do capture developer intent. First, HeapDbg, the tool that realizes the heap abstraction on which this study rests, has successfully been used to identify and fix memory issues in the DaCapo benchmarks, as well as in production software [26]. For instance, DaCapo's chart creates a large array of XYCord objects each of which consists of two boxed doubles; HeapDbg's visualization made this memory bloat obvious. Developers also reported that the HeapDbg tool was useful for program understanding and that it generally grouped objects into components in the expected ways. For instance, a Microsoft developer said "I found simply scanning around the structure graph to be very interesting and found a number of places where it did not match my understanding of the code. However, on further investigation most of these mismatches were due to bugs in the program which were causing unintended sharing."

The combination of quantitative evidence presented above and the qualitative developer experience with HeapDbg both support our hypothesis that our abstraction accurately captures roles and sharing in heap structures.

Threats to Validity We have used inferential statistical tests and analysis such as t-tests and confidence intervals to make conclusions about a large set of Java programs based on observations from a smaller set. A potential threat to external validity for any study is that the sample examined is not representative of the larger population and thus the results do not generalize.

To mitigate this threat, we have chosen to examine a benchmark that has been independently selected as a representative set of open source, client-side Java programs and which has been used in a large number number of prior studies. Although we do not include the full set of DaCapo benchmarks (due to limitations in ikvm) the set we do analyze contains programs covering maximal, minimal, and median values of the *principal component* studies done in [3] and the ownership/uniqueness properties in [23]. Thus, we believe the set we use covers the behaviors in the suite.

A potential confounding factor for our results is that the cross compilation step (Java to .Net via ikvm) could substantially alter the heap properties we are interested in. To check this possibility we took versions of luindex and lusearch which had been, by hand, ported directly from Java to C# [29]. The results for these native C# implementations are identical to the DaCapo versions in the *atomic* node rates. For luindex and lusearch respectively, the injectivity rates are 89% and 97%, the cross edge rates are 42% and 25%, and the unclassified cross edge rates are 13% and 11%. As these values are within a few percent of the results for the DaCapo versions, we conclude that the cross compilation did not impact our results to any significant degree.

All of the programs we examined are mature, well-tested and implemented in object-oriented languages with garbage collection. Thus, it is not clear whether these results generalize to domains such as low-level systems code, languages that use other programming paradigms (*e.g.* functional programming languages), programs in environments without fully automatic memory management (*e.g.* as C++), or programs that violate object-oriented programming conventions.

To further understand how these issues might impact our results we looked at two proprietary C# programs. The codebases were recently implemented, use modern C# language features, and have non-trivial architectures and data structure usage. The atomic rates for these programs of 93% and 97% is similar to our DaCapo results. Edge injectivity was slightly lower than for the DaCapo programs at 74% and 78%. However, the incidence of cross edges, 52% and 43%, along with the rate of unclassified cross edges, 36% and 32%, was higher than expected. When we shared these results with the developers they explained that the observed sharing involved user-defined immutable or lazily initialized objects, in addition to singletons or intern tables that were not stored in static fields. After providing manual immutable/singleton/intern annotations on the types the resulting injectivity/sharing values were in-line with the results from the DaCapo benchmarks. Thus, the sharing idioms described in this work appear to be generally applicable. However, this result illustrates how different languages (and development practices) may realize these concepts differently and highlights the potential value of having a type/annotation system or program analysis specifically for these properties.

Internal validity is related to how well associations or correlations are indicative of causal effects. The goal of this study has been to empirically examine characteristics of program heaps rather then look at causes, so it does not suffer from threats to its internal validity. Finally, a study has construct validity when its conclusions are based on the correct use of measures and analyses. This largely rests on the validity of the use of conceptual components as meaningful partition of a concrete heap. We addressed this threat in the previous section. No oracle exists for heap abstraction, so our approach and results rest on a particular abstraction of the heap and our study may suffer from construct validity to the degree that the reader does not accept our abstraction.

5. RELATED WORK

A variety of questions about the structure of the program heap have been explored in previous empirical studies. Often these studies have focused on the *shape* of the data structures that appear in the programs and use type [21] or reachability from root locations [30] to define the sets of objects over which to compute shape information. This, relatively coarse, decomposition of the heap results in lower resolution information than our approach. Mitchell *et al.* [27, 28] and Potanin *et al.* [31] look at the heap through the lens of ownership and dominator structures. In contrast this work uses conceptual components and injectivity as introduced by Marron *et al.* in [25, 26] which enable us to extract information on injective/non-injective pointer sets and to differentiate between sharing via single or multiple components.

At a high-level, the general approach to abstraction used in this paper is standard: identify recursive structures and define predecessor and equality relations to partition the heap [5, 8, 33]. This general approach has been realized in many different variations [2, 21, 25, 27, 28], among others. However, details of the definitions used can produce a wide range of different heap models. The abstraction used in this work elucidates relatively unexplored features and does so with greater precision than previous work. This study examines heap features via a heap abstraction [25, 26] that employs equivalence and predecessor relations, based on field, type, and immediate predecessors, to capture structural features, notably pointer injectivity and shape. In contrast, Mitchell *et al.* use a predecessor relation based on type information and dominators to group objects which merges object structures into equivalence classes much more aggressively. For example, Mitchell *et al.*'s abstraction, applied to the SpecJVM Raytracer (mtrt), merges 4 cyclic structures, 4 list structures, and 30 atomic sets of objects into a single component (via their dominator-based abstraction), as shown in [27, Figure 4c]. Our abstraction preserves this structural information. This increased resolution enables us to count List/Cycle/Atomic shapes and their sharing relations, instead of reporting a single cyclic structure [26].

The work in [19] performs an extensive evaluation of reachability in the context of understanding object lifetime for garbage-collection applications. The paper [3] introducing the DaCapo benchmarks (used in this work) includes an extensive evaluation of both general properties of the benchmarks and how they allocate and use memory.

Work by Hackett and Aiken in [16] explores how aliasing is used in systems software and, much like the work in this paper, relates the observed aliasing relations to concepts in the source code. Their work focuses on aliasing on individual pointers instead of larger scale conceptual components and therefore does not explore as wide a range of properties as the work presented in this paper. Work by Ma and Foster [23] explores a rich set of sharing and structural annotations and develops a static analysis to extract them. Their empirical study employs their static analysis to construct a conservative over-approximation of actual program behavior and identify the prevalence of various properties. Thus, their work provides a *lower* bound (possibly a very conservative one) on these numbers while our work uses runtime sampling to compute an *upper* bound (which we believe is quite precise) for the prevalence of the properties measured. Abi-Antoun and Aldrich computed ownership domain information whose quality was evaluated by developers [1].

6. FUTURE WORK AND CONCLUSION

In an effort to understand the heaps of real-world programs, we analyzed the heap structures of a number of DaCapo applications. We found that the organization of heap structures is fairly simple, with the vast majority made up of atomic shapes and that approximately half of all data structures on the heap are locally owned. Sharing occurs between conceptual components more often than within them and although a high proportion (37% to 61%) of objects are shared, this sharing is frequently of immutable objects or, in smaller proportions, unique or global objects. In practice, sharing occurs via fairly simple and common development idioms. Our abstraction classifies a large majority of sharing relations (89% of fields and 87% of edges) and partitions the heap into categories that 1) show clear statistical differences in occurrence, 2) model simple and common programming practices, and 3) are useful and intuitive to practitioners. These results call into question the commonly held belief that the heap exhibits intricate sharing and show, rather, that the heap is, in practice, a fundamentally simple structure which is primarily constructed from a small number of basic structures and sharing idioms. Finally, our results have actionable implications for rethinking the design of annotation systems, memory management, and program analyses.

7. ACKNOWLEDGMENTS

We would like to thank the reviewers for their constructive feedback on earlier drafts of this paper. This research was supported in part by the NSF, grant 0964703. This paper's content does not necessarily reflect the position or policy of the government; no official endorsement should be inferred.

8. References

[1] M. Abi-Antoun and J. Aldrich. A field study in static extraction of runtime architectures. In *PASTE*, 2008.

[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] N. R. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *OOPSLA*, 2010.

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

[6] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *ISMM*, 2004.

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

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

[9] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

[10] S. Dowdy, S. Wearden, and D. Chilko. *Statistics for research.* John Wiley & Sons, third edition, 2004.

[11] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *SCP*, Dec. 2007.

[12] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

[13] C. Gordon, M. Parkinson, J. Parsons, A. Bromfield, and J. Duffy. Uniqueness and reference immutability for safe parallelism. In *OOPSLA*, 2012.

[14] S. Guyer, K. McKinley, and D. Frampton. Ulterior reference counting: Fast garbage collection without a long wait. In *OOPSLA*, 2003.

[15] S. Guyer, K. McKinley, and D. Frampton. Free-Me: A static analysis for automatic individual object reclamation. In *PLDI*, 2006.

[16] B. Hackett and A. Aiken. How is aliasing used in systems software? In *FSE*, 2006.

[17] Heap abstraction code. http://heapdbg.codeplex.com/.

[18] M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *OOPSLA*, 2003.

[19] M. Hirzel, J. Henkel, A. Diwan, and M. Hind. Understanding the connectivity of heap objects. In *ISMM*, 2002.

[20] ikvm. http://www.ikvm.net/.

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

[22] C. Lattner and V. Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.

[23] K.-K. Ma and J. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, 2007.

[24] M. Marron. Heap analysis design: An empirical approach. In *Submission*, 2012.

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

[26] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting runtime heaps for program understanding. *IEEE TSE*, 2013.

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

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

[29] Nlucene. http://nlucene.sourceforge.net/.

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

[31] A. Potanin, J. Noble, and R. Biddle. Checking ownership and confinement: Research articles. *Concurrency and Computation: Practice and Experience*, 2004.

[32] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

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