Collecting a Heap of Shapes

Earl T. Barr[1] Christian Bird[2] Mark Marron[3]

[1]UC Davis [2]Microsoft Research [3]IMDEA Software Research

Abstract. A large gap exists between the wide range of admissible heap structures and those that programmers actually build. To understand this gap, we empirically study heap structures and their sharing relations in real-world programs. Our goal is to characterize these heaps. Our study rests on a heap abstraction that uses structural indistinguishability principles to group objects that play the same role. Our results shed light on prevalence of recursive data-structures, aggregation, and the sharing patterns that occur in programs. We find, for example, that real-world heaps are dominated by atomic shapes (79% on average) and the majority of sharing occurs via common programming idioms. In short, the heap is, in practice, a simple structure constructed out of a small number of simple structures. Our findings imply that garbage collection and program analysis may achieve a high return by focusing on simple heap structures.

1 Introduction

The program heap is fundamentally a simple mathematical concept — 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. We seek to fill in this gap and understand more precisely the heap structures in real world programs. We approach these questions via a naturalistic empirical analysis of real-world heaps. The results of this analysis indicate that, in practice, *the heap is a fundamentally simple structure* that is constructed in large part out of a small number of simple structures and sharing idioms.

This result has substantial implications for programming language research, particularly type and annotation systems [5, 6, 9, 17, 28] and the design of static heap analysis techniques [7, 26, 27, 29]. Work in these areas generally considers the heap to be an adversarial setting where the analysis or specification system must effectively handle a wide range of complex heap structures that *could* appear. Our results imply that this pessimistic view does not reflect the reality of how object-oriented programs organize the heap and may artificially limit the scalability and utility of analysis or annotation systems built under this assumption.

A major consideration when studying heap structures is to decide the level of abstraction to employ. A natural idea is to look at how individual objects are related and perhaps shared [10, 15, 17]. We hypothesize that developers actually think primarily in terms of the roles that objects play in a program 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 in the same containers or structures while objects that play different roles are segregated. This allows us to study the heap structures, and relations between them, at a higher level of abstraction that is closer to what the developer envisions.

In this paper, we examine the heaps of real-world programs from the DaCapo suite [4] and find with a high degree of statistical confidence that for object-oriented programs: unique referencing (a sub-relation of ownership [6]) is an important but not dominant organization concept (mean of 43% ± 12% of types), that aggregation is the dominant form of composition (mean of 79% ± 21% of types), and that all types sharing can be well organized by a small set of developer-centric concepts (mean of 89% ± 6% of types can be precisely categorized). We obtain similar results for the structures that are built. These results provide information about the relative importance of set vs. inductive reasoning (aggregate vs. recursive structures) for developing shape analysis techniques [7, 29], and what structural properties occur frequently in real-world code [18] (*i.e.* they must be represented accurately in order to achieve precise results). The results also imply that, while *ownership* [5, 6, 17] is important to the design of object-oriented programs, the strict ownership discipline is often violated. We quantify these violations and how complex the resulting aliasing relations are [20]. We show that the majority of sharing that actually occurs can be categorized using a small number of programming idioms. These results provide possible directions for work on heap annotation and type systems.

This paper makes the following contributions:

- We use runtime sampling to produce a range of *statistically meaningful* measurements of the heaps produced by the DaCapo [4] benchmark, a well-known benchmark that encompasses a real-world object-oriented programs, selected to represent a range of application domains and programming idioms;
- We provide evidence that *components*, as described in [18, 21], usefully and closely correspond to the *roles* that developer assign to objects; and
- We identify a small number of idiomatic sharing patterns that describe the majority of sharing that occurs in practice.

These 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 designs of annotation systems and program analyses.

Section 2 presents the theory that informs that the research questions we address in this study. Section 3 introduces the formalism upon which our heap analysis tool depends and defines the measures we use to answer our research questions. Section 4 describes how we process concrete heaps and compute the data that we analyze. Section 5 presents the results of our study and answers our research questions. Finally, we examine previous work on empirical heap studies in Section 6 and summarize our results in Section 7.

2 Theory

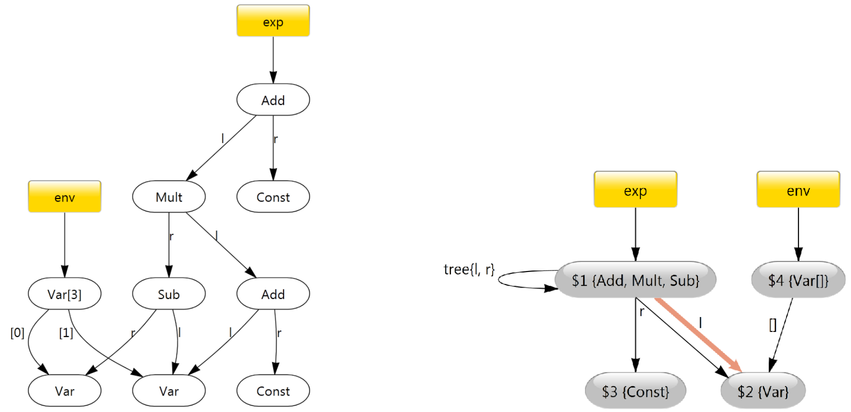
Classically, object-oriented programming provides two ways, (1) inheritance and polymorphism (2) encapsulation and aggregation, for a programmer to transform abstract concepts in the problem space into classes that implement these concepts. A programmer can use inheritance and polymorphism to build up hierarchies of types via the use of the *is-a* relationship. How this is-a relation is used in the organization and construction of real world object-oriented programs has been studied extensively [4, 5, 16]. In this work, we turn to the question of how programmers use encapsulation and aggregation, *i.e. has-a* relation, to organize objects in real world, object-oriented programs. Simply stated, we ask "How do programmers organize the heaps of real world object-oriented programs?"

In this work, we hypothesize that developers often think of objects in terms of the roles they play in the programs [21]. These roles implicitly aggregate objects into conceptually related sets. Thus, the simple view of a heap as a set of concrete objects is insufficient since a programmer may think of all the objects in a graph as one conceptual entity or all the objects in a binary tree as a single tree. Similarly, one cannot simply equate conceptual components with types since programs often use the same type for different purposes. For example, a program may use `List` both to handle collisions in a hash table and to hold the neighbors of a vertex in a graph.

To accommodate these challenges, we build a role-based heap abstraction to mirror the roles a programmer assigns to objects. First, we introduce some terminology. A recursive data structure consists of two sets of objects — the infrastructural objects that realize it, like instances of a ListNode class, and those objects stored in it, like instances of a Profile class. The *backbone* of a recursive data structure comprises the infrastructural objects and their sharing relations. Objects are structurally *indistinguishable* if they 1) are members of the same backbone or 2) have the same type and are stored together. We formalize what "stored together" means in Section 3. We use these two structural indistinguishability principles to partition the concrete heap into *conceptual components*.

Figure 1 shows the output of *HeapDbg* [21], a conceptual component visualization tool; it illustrates how we apply these principles to the concrete heap of a program that manipulates arithmetic expression trees. Figure 1(a) shows a concrete heap snapshot as computed by the sampling framework for a simple program that manipulates expression trees. 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 backbone 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



(a) A Concrete Heap.

(b) Corresponding Abstract Heap.

Fig. 1. Running example.

are not abstracted into a single object because their type distinguishes them. Since no principle applies to the environment array `env`, it acquires its own node $4. The edges represent possible sets of pointers and their associated field labels. The parallel edges from $1 to $2 are discussed below in Section 2.3.

2.1 Heap Structure

Given a set of 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 and set based reasoning in shape analysis tools [7, 18, 29]. This also provides insight into the role that recursive structures and container libraries play in the design of programs [3] specifically: 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?

In our running example, Figure 1(b), we have four conceptual components the recursive expression tree in node $1, the set of `Var` objects in node $2, the `env` array in node $4, and the `Const` objects into node $3. These components include one recursive structure, the tree in $1, and one builtin collection object, the array in $4.

2.2 Ownership

Encapsulation is a fundamental concept in OOP and has traditionally been expressed as a binary property in terms of *ownership* [6]. This strict definition with transitivity leads to the same issues as encountered in the classic `const` problem where use of `const` in one location leads to required cascading uses throughout. In this work we utilize the slightly weaker notion of *unique-ref* [17] that specifies that the address of an object is stored in at most one memory location although objects transitively reachable from it may be shared. For us, an object that has a unique-ref is *locally owned*. This definition allows some data is to be hidden, while other data may be shared, i.e. locally owned but *not* necessarily (transitively) owned.

Questions about ownership, local ownership, and sharing are fundamental throughout research in programming language design [5, 6, 9, 28] and program analysis research [7, 17, 18, 26, 29]. Despite a number of valuable empirical studies [10, 12, 14, 15, 17, 21, 22, 24] the question of what sharing is actually present in real-world programs and why this sharing occurs is still an open question. In the case of programming language design there is substantial interest in developing type or annotation systems that can express rich encapsulation and exposure properties. The construction of an annotation or type system for sharing that can be applied to real-world programs remains and open problem. Similarly, in the area of static heap analysis there has been a substantial amount of work on possible approaches for modeling sharing but it remains unclear what sharing is actually present in real-world programs and thus is critical to producing useful results.

In this study, we hypothesize that ownership in real object-oriented programs is important but that a non-trivial amount of sharing also occurs. Further, we hypothesize that this sharing idiomatic and that much of it can be classified by a small number of common programming idioms. Thus, we first want to understand how common local ownership is within a program.

Research Question 2: What percentage of objects are locally owned?

2.3 Sharing

For the objects that are shared we want to understand more about how this sharing occurs. There are two possible ways sharing could occur 1) there are objects in several different components that all contain pointers to the same object or 2) there are multiple objects in the same component that 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 in the program. This distinction provides insight the degree to which a codebase localizes interactions with shared objects.

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

A set of pointers that do not alias is *injective*, *i.e.* the set is a one-to-one map of pointers to objects. Figure 1(b) shows that the `Const` objects are always locally owned, since it has a single (narrow), injective in-edge. In contrast, several expression objects might point to the same Var object; Figure 1(b) depicts this aliasing (*non-injectivity*) using wide, orange colored edges, if color is available. The Var exhibits both types of sharing (the node has multiple incoming *cross* edges). Multiple objects within the expr, the tree component, alias objects with Var; in addition to the aliasing from expr, the environment array env also points to objects within Var.

To understand why sharing does occur we examine the *non-injective* edges and *cross* edges that occur through the lens of a number of common programming idioms.

Research Question 4: What percentage of sharing involves 1) immutable objects, 2) singleton or intern table objects, or 3) *contained* objects, *i.e.* sharing is contained in an immediately enclosing object?

The first idiom we look at is the sharing of immutable objects such as strings, which are always immutable in C# and Java. When the objects are known to be immutable, developers are much less concerned about sharing them and in fact for performance reasons often do so intentionally. Another common idiom is the use of the singleton design pattern or, very similarly, an intern table that maps objects that are equal based on value type to objects that are equal on reference identity. Common examples of this are the String.intern method in Java or compiler symbol tables. Our final idiom is based on the notion that a key role of many classes is to aggregate and provide the appropriate views of the contained data. This often requires the resulting objects to store the data in multiple ways. For example a class may store its object 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, the shared objects are recaptured by a unique dominator that is no more that two pointer dereferences away. We hypothesize that, in practice, these three types of sharing dominate the sharing that occurs in real-world programs, so we ask:

Research Question 5: What percentage 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 (1) that it is possible to relate idiomatic code designs to the heap structures they produce and (2) 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 is capable of expressing the concepts used in this study will be able to precisely (and compactly) annotate (analyze) the features that appear in real-world programs. Further, since these system would be built on a small number of concepts and designed to reflect programmer intent, these systems should be relatively simple to implement in an abstract domain and both simple and intuitive for programmers to use.

2.4 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*[1] *partition the heap.* If this hypothesis does not hold and our conceptual components poorly approximate developer intent, then we would expect them to contain unrelated objects and the resulting measurements of their properties to produce low information, indeterminate values. We present evidence for this hypothesis as part of the empirical analysis in Section 5.4.

3 Formalism

As is standard, we model the state of a program heap with an environment, mapping variables to addresses, and a store, mapping addresses to objects. We refer to an instance of an environment together with a store as a *heap*. Given a program that defines a set of types, $\mathsf{ProgramTypes}$, we define the set of concrete labels in the program, $\mathsf{StorageLabels}$, as the set of all member fields and array indices in the program. We then construct a heap as a tuple $(\mathsf{Env}, \sigma, \mathsf{Ob})$ where:

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

Each object $o \in \mathsf{Ob}$ is a tuple consisting of the type of the object and a map from field labels to addresses for the fields defined in the object. We assume that the objects in Ob and the variables in the environment Env, as well as the values stored in them, are well typed according to the store (σ) and the types/labels in the sets $\mathsf{ProgramTypes}$ and $\mathsf{StorageLabels}$.

In the following definitions, $\mathsf{Type}(o)$ refers to the type of an object. The usual notation $o.l$ to refers to the value of the field (or array index) l in the object. To simplify and clarify our definitions, we define a *non-null pointer p* associated with an object o and a label as l in a specific heap $(\mathsf{Env}, \sigma, \mathsf{Ob})$ as $p = (o, l, \sigma(o.l))$ where $\sigma(o.l) \neq \mathsf{null}$. We define a helper function $\mathsf{Field}(\mathit{type})$ to return the set of all fields that are defined for a given type (or array indices for an array type).

Conceptual Component A conceptual component in memory $C \subseteq \mathsf{Ob}$ is simply a subset of the heap objects. As defined below in Section 3.1, our abstraction computes a partition of Ob and maps each partition to a conceptual component. Where clear from context, we refer to conceptual components simply as components.

Component Types Given a component of the heap C we can define the type set associated with it as the types of all objects that are contained in the component: $\mathsf{Type}_c(C) = \{\mathsf{Type}(o) \mid o \in C\}$.

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

Component Pointer Injectivity Given two disjoint components C_1 and C_2 in the heap $(\mathsf{Env}, \sigma, \mathsf{Ob})$ and the non-null pointers with the label l from C_1 to C_2,

$$\mathsf{inj}(C_1, C_2, l, \sigma) \Leftrightarrow o_s \neq o'_s \Rightarrow \sigma(o_s.l) \neq \sigma(o'_s.l),$$

for all pairs of non-null pointers $(o_s, l, \sigma(o_s.l))$ and $(o'_s, l, \sigma(o'_s.l))$. As a special case for array objects, we have $\mathsf{inj}_{[]}(C_1, C_2, \sigma) \Leftrightarrow i \neq j \Rightarrow \sigma(o_s[i]) \neq \sigma(o_s[j])$, for all pairs of non-null pointers $(o_s, i, \sigma(o_s.i))$ and $(o_s, j, \sigma(o_s.j))$ where i, j are valid array indices.

These definitions capture the general case of an injective relation from a set of objects and fields to target objects. They also capture the special, but important case of arrays where each index in an array contains a pointer to a distinct object. We say a set of component pointers P between the components C_1 and C_2 is *injective*, which we denote $\mathsf{Inj}(P)$, if the pointers have the same label l and $\mathsf{inj}(C_1, C_2, l, \sigma)$ or they are stored in arrays and $\mathsf{inj}_{[]}(C_1, C_2, \sigma)$.

A set of pointers that do not alias is *injective*, *i.e.* the set is a one-to-one map of pointers to objects. Figure 1(b) shows that the

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. Given a component C then:

$\mathsf{any}(C)$ holds for any graph. We use it as the most general shape that does not satisfy a more restrictive property.
$\mathsf{dag}(C)$ holds if the subheap restricted to C is acyclic.
$\mathsf{tree}(C)$ holds if $\mathsf{dag}(C)$ holds and the subheap restricted to C contains no pointers that create cross edges.
$\mathsf{none}(C)$ holds if the edge set in the subheap restricted to C is empty.

3.1 Identifying Conceptual Components

To identify the components in a program heap we want to group together the objects that play the same conceptual roles in the program. Our hypothesis is that, in object-oriented programs, our indistinguishability principles (Section 2) determine if two objects are conceptually indistinguishable and therefore have the same conceptual roles, and belong in the same conceptual component. In more detail, our indistinguishability principles are (1) objects in the same recursive data-structure belong to the same component; (2) objects that are stored in the same container or are pointed to by fields of objects in the same conceptual component are indistinguishable (3) unless the objects have different types then there is one conceptual component per type. Here, principles 2 and 3 decompose the single, more abstract storage principle in Section 2.

This organizational hypothesis, which is also used in [18, 21], forms a set of equivalence relations on the objects and connectivity from the underlying heap structure. Thus, we can formulate the identification of conceptual components as a congruence closure computation. To construct the closure we first build a map from the objects in the heap to equivalence sets (conceptual components) using a Tarjan union-find structure. Formally, $\Pi : \mathsf{Ob} \to \{\pi_1, \ldots, \pi_k\}$ where $\pi_i \in 2^{\mathsf{Ob}}$ and $\{\pi_1, \ldots, \pi_k\}$ *partition* Ob. We start with a partition per object, then apply the following equivalence relations on this set of equivalence classes until the set of partitions is closed under the relations. At this point the resulting equivalence classes are the desired conceptual components.

The first equivalence relation that defines conceptual components identifies objects that are part of the same recursive data structure. It examines the type system of the program under analysis and identifies all the types, τ_1 and τ_2 whose definitions are mutually recursive; we denote such types $\tau_1 \sim \tau_2$. Our goal is to distinguish types that represent unbounded recursive structures (*e.g.* a linked-list) from types that represent relations among a finite set of different concepts whose definitions are mutually referential (*e.g.* an iterator and its container reference each other).

To distinguish types in unbounded structures from finite mutual dependencies we use a simple heuristic: the types τ_1 and τ_2 are part of the same unbounded recursive type definition, written $\tau_1 \simeq \tau_2$, if $\tau_1 \sim \tau_2 \wedge (\forall \tau \in \{\tau' | \tau' \sim \tau_1 \wedge \tau' \sim \tau_2\}$ where τ is a builtin library type $\vee$ the least common super-type of τ, τ_1, τ_2 is a user-defined type). This definition determines whether all the types in the recursive set (excluding any builtin types such as arrays or hashsets) have a common and nontrivial super-type. If this holds then they all expose some uniform representation that makes them conceptually equivalent.

Definition 1 (Recursive Structure). *Given two partitions π_1 and π_2, the* unbounded recursive structure *congruence relation is* $\pi_1 \equiv_r^{\Pi} \pi_2 \Leftrightarrow \exists \tau_1 \in \mathsf{Type}_c(\pi_1), \exists \tau_2 \in \mathsf{Type}_c(\pi_2)$ *s.t.* $\tau_1 \simeq \tau_2 \wedge \exists o \in \pi_1, \exists l \in \mathsf{Field}(o)$ *s.t.* $\sigma(o.l) \in \pi_2$.

The other part of congruence closure computation identifies conceptual components that have equivalent successors. The partition π_1 is a *successor* π_2 on l iff $\exists o \in \pi_2, \exists l \in \mathsf{Field}(o)$ s.t. $\sigma(o.l) \in \pi_1$. To define equivalent successors, we define the relation $\text{Compatible}(\pi_1, \pi_2) \Leftrightarrow \mathsf{Type}_c(\pi_1) \cap \mathsf{Type}_c(\pi_2) \neq \emptyset$.

Definition 2 (Equivalent Successors). *For the partition π with successors π_1 on label l_1 and π_2 on l_2, π_1 and π_2 are* equivalent successors *when* $\pi_1 \equiv_s^{\Pi} \pi_2 \Leftrightarrow (l_1 = l_2 \vee l_1, l_2 \in \mathbb{N}) \wedge \text{Compatible}(\pi_1, \pi_2)$.

Using the *recursive structure* relation and the *equivalent successor* relations we can efficiently compute the congruence closure of for a concrete heap which yields the desired conceptual components. In many contexts, it is useful to view the conceptual components and component pointers as a graph in which each conceptual component (set of objects) is a node and each set of component pointers is an edge.

Definition 3 (Conceptual Component Graph). *Given a concrete heap* $(\mathsf{Env}, \sigma, \mathsf{Ob})$ *and the partition Π, a* conceptual component *graph is $G = (N, E)$ where*

- $\forall \pi_i \in \{\pi_1, \ldots, \pi_k\}, \exists n \in N$
- $\forall$ *sets of component pointers* $P = \{(n_s, n_t, l) \mid n_s, n_t \in N \wedge l \in \mathsf{StorageLabels}\}$, $\exists e \in E$ *corresponding to P where π_s corresponds to n_s and π_t corresponds to n_t.*

Example Heap Components. We can see how this technique for identifying conceptual components works by applying it to the heap in Figure 1(a). The computation of the equivalence classes for the objects identifies the objects with the types `Add`, `Sub`, and `Mult` as belonging to the same partition since they are part of the same *recursive structure*. The grouping of these objects then causes the `Const` type objects (nodes 4, 7) to be identified as *equivalent successors* of the tree partition. Finally, either due to the tree partition or the fact that the `Var[]` node references all the `Var` type objects triggers the identification of all the `Var` type partition as *equivalent successors*. This results in the conceptual component graph shown in Figure 1(b).

3.2 Features Under Study

We next define predicates over the conceptual component graph $G = (N, E)$ and define the features that we measure and use to evaluate the research questions posed in Section 2. These measures are computed as percentages for both the of nodes (or edges) in the graphs that satisfy a property and for the types (or fields) that satisfy a property. We classify our predicates into four classes, based on their focus (nodes *vs.* edges) and their locality (purely local *vs.* involving multiple nodes/edges). To provide intuition for each definition, we refer back to our running example in Section 2.

Single Component Properties A node n in the component graph represents a single conceptual component of a concrete heap. The types and the shape of the data structure that the component represents are frequently of interest in program analysis/specification. We consider a type to be *immutable* when it is a known primitive type such as `String`, `Int`, *etc.* which is immutable according to the language definition. A builtin type is any type from the standard JDK libraries.

A node $n \in N$ is:

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

In our running example (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[]`. Our example contains no singleton objects (and thus no singleton nodes or types). We use the global shared property to identify sets of objects that are intended to be globally shared, such as values stored in an intern table or some set of special sentinel objects. In Figure 1, the region containing the `Var` objects is a globally shared region since it represents only `Var` objects stored in a container (the array) to which the static field `env` refers. Thus, our examples contains one globally shared node and one globally shared type.

A node $n \in N$ has shape:

Atomic iff $\mathsf{Shape}_c(n) = \mathsf{none}$, *i.e.*, there are no pointers between any of the objects.
Linear iff $\mathsf{Shape}_c(n) = \mathsf{tree} \vee \mathsf{Shape}_c(n) = \mathsf{dag}$, *i.e.*, there is a tree or a dag structure.
Cyclic iff $\exists$ a strongly-connected component of pointers in the objects in n.

Figure 1 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. None of the nodes in Figure 1 have self-edges labeled *any* so it has no cyclic nodes or types.

Cross Component Pointer Set Properties We next distinguish between sets of pointers that are internal to a single component and those that cross between distinct components. We also want to extract information on the prevalence of *injective* and *non-injective* pointer sets.

An edge $e \in E$ is:

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 aliases.

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. The example 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 the `l` and `[]` fields, the example contains two injective fields.
The edge $e \in E$ s.t. e is external and $\neg\mathsf{Inj}(e)$ is

NonInjectiveToImmutable iff its target node n_t is an immutable node.
NonInjectiveToSingleton iff its target node n_t is a singleton node.
NonInjectiveToGlobal iff its target node n_t is a globally shared node.

Our example (Figure 1) does not contain any immutable or singleton components, so it has no edges (or fields) that interfere only on immutable or singleton objects. The `Var` region is globally shared, so our example contains a non-injective edge, the `l` edge from the expression tree to `Var`. Thus, we have one *NonInjectiveToGlobal* edge. Since all the other edges are either *Injective* or *Internal*, we have one *NonInjectiveToGlobal* field.

Ownership Properties Next we define the ownership properties and in-degree of the nodes. Let $d^+ : N \to \mathbb{N}$ denote the in-degree, including self edges, of a node. For all nodes $n \in N$, when $d^+(n) = 1$ and its single in-edge e is injective ($\mathsf{Inj}(e)$), the node n is *locally owned* and its in-edge e is a *local owner*.

When an conceptual component is locally owned, a single (unique) pointer points to each of the objects in it. This can be seen as the dual of local owner fields where, instead of asking if a location contains a local owner pointer to a target object, we ask if there exists a unique location that contains a pointer to a target object. In our running example, Figure 1, two nodes — of type `Var[]` and `Const` — are locally owned. Since the edge with the `r` label ending at the node containing the `Const` objects is injective and is the node's only in-edge, this edge and the `l` field are local owners.

Graph Structure Properties Next we define predicates to capture the graph theoretic properties of the component graphs, particularly sharing information.

An edge $e \in E$ that ends at node n_t is:

TreeEdge iff e is external and $d^+(n_t) = 1$.
CrossEdge iff e is external and $d^+(n_t) > 1$ where DFS from the program root set would label e a *cross* edge.
BackEdge iff e is an external edge and $d^+(n_t) > 1$ where DFS from the program root set would label e a *back* edge.
CrossToImmutable iff e is a cross edge and n_t is immutable.
CrossToSingleton iff e is a cross edge and n_t is a singleton.
CrossToGlobal iff e is a cross edge and n_t is globally shared.
CrossToLocalEscape iff e is a cross edge and $\exists n_d \in N$ 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 and encapsulated.

Figure 1 contains three *TreeEdges* — the local `exp` variable edge, the static field `env`, and the `r` edge pointing to the `Const` objects. It 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 *CrossToGlobal* fields. Our example does not contain any back edges, immutable nodes or singletons, so it does not illustrate the *BackEdge*, *CrossToImmutable*, or *CrossToSingleton* features.

4 Methodology

The main objective in this work is to understand, in a general sense, what kinds of structures real world programs build; in particular, we are interested in features that express developer intent, *i.e.* class invariants. Thus, HeapDbg extracts heap information, at program points and from those parts of the heap, that are involved in these invariants. In standard object-oriented program design these points are the entry/exit of public methods and, in the heap, all objects reachable from the parameters and in-scope static

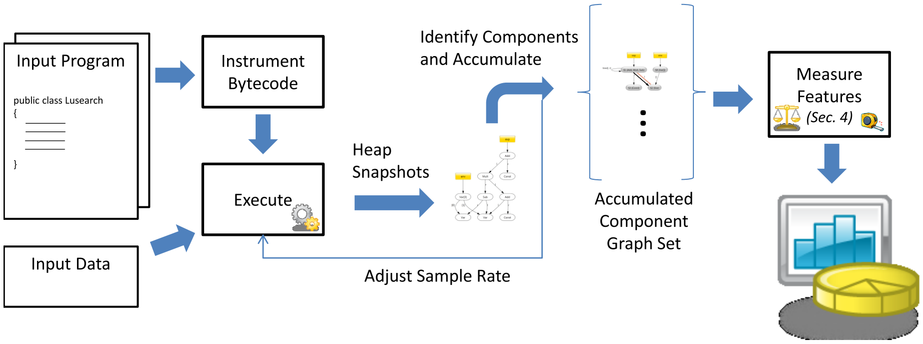


Fig. 2. Overview of the study methodology.

fields. As the runtime analysis infrastructure we use in this paper operates on .Net bytecode, we translate the Java programs into .Net bytcode using the ikvm compiler [13].

Figure 2 shows the workflow of the execution, sampling, collection, and measurements of the heaps produced for a input program. At the entry and exit of every public method in the program, the profiler rewrites the program's .Net assembly with sampling code to compute heap snapshots and to identify the conceptual components in the snapshot, as depicted by the *Instrument Bytecode* box in Figure 2. A snapshot of the heap is the portion reachable from the parameters of a the current method call and from static roots. The rewriter is based on the heap profiler work from [11]. Since extracting heap snapshots at each method call is impractical we use a per-method randomized approach with an exponential backoff, which occurs when the current heap snapshot and previously taken snapshots differ with respect to the components and the relations on these components. If we detect that the program may have entered a new phase of computation, *viz.* the current snapshot differs from all previously taken snapshots, then we add the component graph corresponding to the current heap snapshot to the accumulated set of component graphs and reset the sampling rate (the control edge from the accumulate flow edge to the execution in Figure 2). On smaller runs we compared the results obtained by sampling uniformly at random with the results from the exponential backoff approach and found that, in addition to having a much larger overhead, the uniform sampling approach produced results that were no more useful.

We ran the profiler tool to compute likely heap invariants for each class declared in the program. To avoid biasing the results heavily toward features of simple classes, we exclude any heap invariant graphs that are subgraphs in a larger component graph. Thus, in the *Identify Components and Accumulate* step in Figure 2, we check if the newly extracted component graph is actually a subgraph of some other previously seen state (and visa-versa). If so, we discard the redundant information. In the case of our example in Figure 1(b), we discard the graph computed for the `Const` objects since this component graph is a subgraph of the component graph computed for the `Add` (or `Sub`/`Mult`) class. Finally, we take the set of likely invariant component graphs for the program and compute the measurements described in Section 3.

5 Evaluation

The DaCapo suite is designed to exercise a wide range of memory system behaviors [4]; thus, its authors selected programs and inputs from a number of real-world applications to represent a range of application domains and to cover many different heap structure behaviors. These features make its benchmarks ideal choices for this work. To perform our study, we translated six programs from DaCapo into .Net bytcode using the ikvm compiler [13] (ikvm is not able to process the remaining DaCapo benchmarks). All of the programs we examine in this section are (1) user-space applications, (2) relatively mature and well tested, (3) implemented in an object oriented fashion, and (4) in a language with garbage collection. Thus, it is not clear whether these results generalize to domains such as low-level systems code, languages which use other programming paradigms (*e.g.* functional programming languages), programs in environments that do not have fully automatic memory management (*e.g.* as C++), or programs not written in a standard object-oriented programming style.

We use inferential quantitative methods (t-tests, confidence intervals, tests of binomial distributions, and chi-square tests [8]) and generalize our results. These techniques indicate of there is statistical significance (i.e. if the results from the sample are simply program-specific noise or are indicative of a more general trend) and they rely on an underlying assumption of a random sample. Thus, our conclusions are only valid for the population of programs for which DaCapo is representative. The fact that we are able to identify statistically significant trends in a sample of just six programs indicates that our data does indeed contain a strong signal. As we collect observations of all of the types in the six programs and the number of types range from hundreds to thousands, one might be tempted to treat the set of all types as a sample of the population of all types in programs of which the DaCapo suite is representative. Such a large sample would certainly provide more statistical power and perhaps tighter confidence bounds. However, such an approach would fatally neglect one of the core assumptions of most statistical techniques, that of independence. The set of types so constructed would not be not independent of each other (they are related and in the same program) and are certainly not drawn at random from the larger population of types in all programs of which DaCapo is representative.

As the majority of our research questions relate to the effective categorization of sharing in object-oriented programs we use t-tests extensively to compare different categories or classifications of fields and edges to determine if particular categories are more prevalent than others. Where appropriate, we compute the confidence bounds on the proportion of edges, fields, types, *etc.* that fall under a particular classification. The confidence bound for a classification indicates an interval that with 95% certainty contains the true mean proportion of that classification across the population of all programs of which DaCapo is representative.

5.1 Properties of Conceptual Components

We begin our evaluation with a number of basic descriptive statistics for the conceptual components that are produced. While not directly applicable to our research questions in

Benchmarks		Types			Conceptual Components		
Name	Objects	Total	Singleton%	GloballyShared%	Total	Immutable%	System%
antlr	∼12K	63	0.0	0.0	606	25.5	26.9
chart	∼189K	119	15.9	3.3	198	19.5	27.2
fop	∼120K	450	57.3	0.7	531	7.5	8.2
luindex	∼2K	48	4.1	12.5	87	15.0	20.6
pmd	∼178K	186	0.5	0.0	146	20.0	21.0
xalan	∼40K	355	13.8	1.9	451	6.9	11.1

Table 1. Descriptive statistics of heap features.

Section 2, these results provide context when answering these questions and insight into how the conceptual components relate to the objects that make up a concrete heap.

A basic property of interest is the amount of compression that is obtained from the grouping of heap objects into conceptual components and how this number relates to the number of distinct types used by the program. Table 1 shows the number of objects in the largest heap snapshot seen for each program along with the number of object types which are allocated and the number of conceptual components in the resulting component graph. This table shows that the number of components is only loosely correlated to the number of objects in the heap and that even in the largest cases the heap can be precisely decomposed into a relatively small number of components. This result indicates that even for large programs the partition is able to identify a relatively small number of components and to produce a relatively compact representation. Further, there are anywhere from 1 to 6 times as many components as there are types, but the range is usually from 1 to 2 times as many. This indicates that objects of the same type are frequently used in multiple distinct roles even within the same program (and heap snapshot).

The singleton and globally shared columns in Table 1 list the percentages of types that are singletons or interned in global tables. The percentages in these columns indicate that, not surprisingly, the singleton and global intern table design patterns occur with non-negligible frequency and are a significant feature of the heap structure of some benchmarks. The 95% confidence interval for singleton types is 0%–38%; for globally shared types, it is 0%–8%. Last two columns report the percentage of components known to contain only immutable objects or only objects from the builtin libraries. These results highlight the importance of both the use of immutable objects (the 95% confidence interval for the mean number of immutable objects is 8%–24%) and the prevalent use of builtin library objects (11%–28% of the components).

Now we explore what sorts of heap structures conceptual components form, first answering **RQ1:** *What proportion of conceptual components are simple* vs. *recursive?* Figure 3 shows the ratios of purely 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 relative number of nodes identified by the abstraction that have the given shape.

As can be seen, atomic shapes dominate the other, more complex components. To assess the level of significance of this finding, we used a two sample t-test to compare the proportions of atomic shapes to all other shapes (i.e., linear and cyclic shapes). Our sample size of six 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 statistically significant degree ($p \ll 0.01$), with a 95% confidence interval for the proportion of shapes that are atomic of $79\% \pm 21\%$. Having established that atomic shapes are most prevalent, we also examined the proportion of cyclic shapes to determine if they are more common that linear shapes. Although they appear more frequently in our six project sample, a t-test did not show a statistically significant difference ($p = 0.07$). We performed a similar analysis for our examination of ratios of shapes per component. Again, atomic shapes dominate all other shapes by a large degree ($p \ll 0.01$). In fact the confidence interval for the proportion of atomic shapes is $98.5\% \pm 1.5\%$. Neither cyclic nor linear shapes were more prevalent than the other to a statistically significant degree. In short, the answer to **RQ1** is that simple conceptual components dominate recursive components: for types, the proportion of atomic shapes is $79\% \pm 21\%$; for components, $98.5\% \pm 1.5\%$.

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 encourage composition of classes in layers and often provide extensive container libraries which are used in lieu of custom list and search tree structures. These features lead to heaps that feature relatively few recursive structures. Similarly, the ability to define a general abstract `BaseNode` class, on which a recursive structure is based, in conjunction with the ability to easily subclass it leads to construction of a relatively few, complex and application-specific recursive structures out of many types, *e.g.* `pmd` builds an abstract syntax tree from the code that it analyzes.

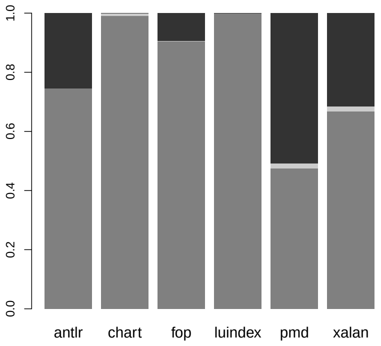
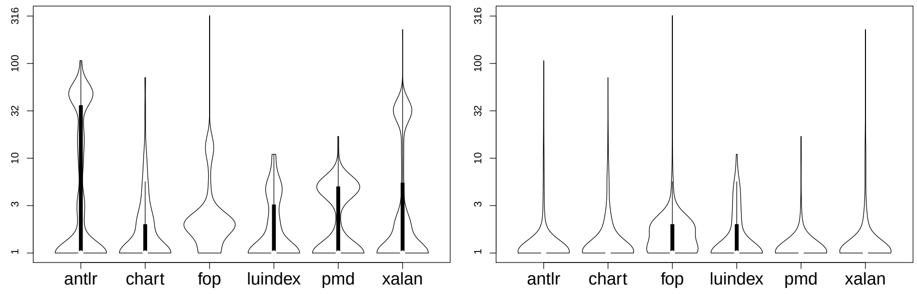


Fig. 3. Percentage of types whose shape is cyclic ■, linear □, or atomic ▨.

5.2 Ownership

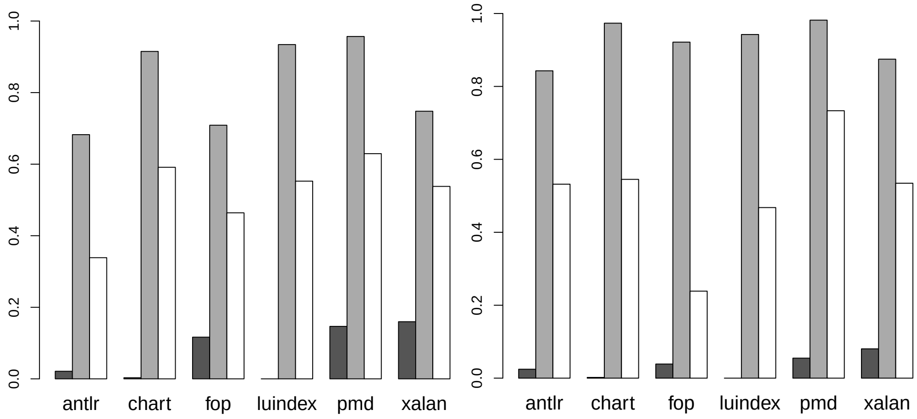
Figure 4 shows the distributions (via violin plots[2]) of in-degree over the components and the types in them. While the number of components and types with in-degree k decreases fairly rapidly as k increases there are a nontrivial number of components (types) with high in-degree. One factor in this is large recursive structures (with many types in the recursive structure) that have many references to them. Thus, even though the in-degree of the individual objects is low, the overall in-degree of the structure they are in is high. A second contributing factor is the high degree of sharing of singleton (and other global)

[2] A violin plot is similar to a box plot in that it enables the comparison of distributions, but gives a more detailed view of the shape of the distribution. Each plot is essentially a probability density distribution. See Wikipedia `http://en.wikipedia.org/wiki/Violin_plot`



(a) In-degree distribution per type.
(b) In-degree distribution per component.

Fig. 4. In-degree distributions (log-scale).



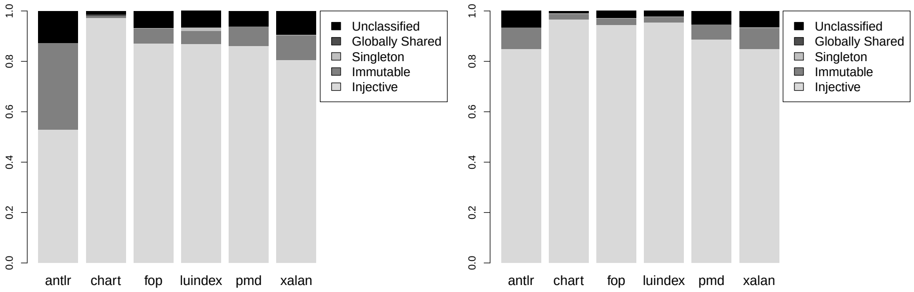
(a) Property per field.
(b) Property per component edge.

Fig. 5. Percentage of pointers that are internal ■, nonnull ▨, or owners □.

objects. The high portion of in-degree 2 types in Figure 4(a) is a result of this kind of structure. The outlier program, fop, has a large number of singleton type objects that are also stored in dictionaries. This figure provides some initial insight into the value and limitations of using local ownership to describe heap structures in real programs.

To understand how much ownership (Section 3) 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 percentage 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 the types is around 50% on average (confidence interval 31%–55%), showing that being locally owned is used as the organizing principle for many parts of the heap. However, the fact that the remaining 45%–70% of the types in the program have references to them stored in multiple locations shows that the principle of local ownership captures a large but not dominate portion of real world heap structures. We see higher ratios of in-degree 1 with a confidence interval of 51%–76% for the components.



(a) Injectivity per field.

(b) Injectivity per component edge.

Fig. 6. The percentage of field and edge pointers that are injective or non-injective, where the non-injective pointers are further subclassified into immutable, singleton, non-singleton globals, or unclassified.

Figure 5 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 define a field to be a lattice join on the pointer classifications of the edges in which the field participates. 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. This matches our observation in Figure 3, now from the viewpoint of fields and component edges instead of types and components, that many programs have a central recursive data structure built using a small number of fields which is then augmented with a large number of simpler aggregate structures. Figure 5(a) shows that on average 50% of the fields in the program always contain pointers that are the local owners of 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 six project sample is 42%–63%. This ratio of translates over 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 34%–68%.

5.3 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 multiple in-edges.

Figure 6 classifies the injectivity (*i.e.* interference) of field and component edge pointers. Figure 6 shows the ratio of fields and edges that always contain *injective* pointers to those fields and edges that at some point contain an *non-injective* pointers. In the *non-injective* case, it further classifies the sharing is into sharing occurs on an

Classification	antlr	chart	fop	luindex	pmd	xalan	Confidence Interval
Per Field							
Tree	56%	63%	57%	59%	73%	59%	54% – 68%*
Cross†	25%	31%	35%	38%	25%	27%	24% – 36%*
Back	19%	6%	8%	3%	2%	14%	1% – 16%
Per Edge							
Tree	63%	66%	48%	61%	81%	63%	53% – 75%*
Cross†	26%	31%	51%	38%	18%	28%	20% – 44%*
Back	11%	3%	1%	1%	1%	9%	0% – 9%
Cross Field							
LocalEscape	2%	10%	6%	11%	6%	4%	3% – 10%
Global	0%	0%	3%	1%	0%	0%	0% – 2%
Singleton	0%	3%	1%	7%	0%	0%	0% – 5%
Immutable⋆	21%	5%	10%	11%	16%	14%	7% – 19%
Unclassified	4%	16%	16%	15%	4%	10%	5% – 17%
Cross Edge							
LocalEscape	0%	4%	2%	10%	2%	3%	0% – 7%
Global	0%	0%	1%	5%	0%	0%	0% – 3%
Singleton	0%	1%	19%	2%	0%	0%	0% – 11%
Immutable⋆	17%	6%	24%	12%	12%	17%	8% – 21%
Unclassified	9%	21%	5%	12%	5%	9%	4% – 18%

Table 2. Sharing classification per field and per component edge. The per field and per edge rows sum to 100%. The cross field and cross rows further categorize the cross edges using the graph structure properties defined in Section 3.2; they do not always sum to their corresponding entry in their cross row marked † because of rounding. The '*' designates a category that occurs more frequently than the categories below it (within each grouping) to a statistically significant degree.

immutable object, a singleton, globally shared non-singleton, and a catch all for otherwise unclassified sharing. 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 nearly 90% of all fields (with a confidence interval of 89%–96%) and 95% of all edges (confidence interval of 93%–98%). The addition of the singleton and globally shared types 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 uses the standard graph theory definitions of tree, cross, and back edges. We then using programming idioms to further breakdown the cross edges. Table 2 again shows both field and edge centric breakdowns of the various ratios the classifications of the fields and edges in the conceptual component graph of each program. The far right columns shows the confidence interval for the mean proportion of occurrence of each category based on our six project sample. In most cases, 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 ($p \ll 0.05$). Further these results show that cross edges appear quite frequently and dominate back edges to a statistically significant degree ($p \ll 0.05$).

We are now in a position to answer **RQ3:** *What percentage of sharing occurs between objects in the same conceptual component* vs. *across conceptual components?* Our analysis groups pointers that originate in one conceptual component into a single edge, while assigning pointers from different conceptual component to different edges, as defined in Section 3. The non-dominance of local ownership in Figure 5 means that sharing is occurring regularly — the 95% confidence interval of the true mean of sharing is 37%–59% for fields and 32%–66% for edges. Tree edges dominate both edge and field views of pointers. The only way a tree field or edge can exhibit sharing is through non-injectivity, because they are, by definition, the only in-edge to their target. Thus, the high degree of injectivity in Figure 6 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. Thus, most of the sharing we observe must span different conceptual components, as the prevalence of cross edges, the two rows marked † in Table 2, makes clear. Thus, we compute the raw count of cross fields (edges) over the raw count of shared fields (edges) and answer **RQ3**. Of all sharing, 48%-82% (field) and 59%–71% (edge) occur between objects in different conceptual components while only 18%–52% (field) and 29%–41% (edge) occur between objects in the same conceptual component. A t-test indicates that sharing occurs more frequently between objects in the different conecptual components to a statistically significant degree ($p \ll 0.05$).

Having answered **RQ3**, we turn to **RQ4:** *What percentage of shared conceptual components represent 1) immutable objects, 2) singleton or intern table objects, or 3)* contained *objects,* i.e. *sharing is contained in an immediately enclosing object?* Table 2 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 because they are relatively uncommon; we 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 6 shows that immutable objects are a major source of the sharing that occurs in practice (confidence interval 18%–71% of all sharing for fields and 28%–66% for edges). The LocalEscape category shows that even our relatively simple definition of localized sharing captures *contained* objects and shows that they form an important structure in these programs, accounting for 11%–31% of the edges on a per field (field) basis and 0%–23% of the edges (edge). Finally, singletons and globals are the least frequent, comprising 0%–16% (field) and 0%–27% (edge) of the sharing. Thus, our answer to **RQ4** is, of all sharing, immutable objects account for the majority of sharing conceptual components (via a t-test with $p \ll 0.05$), with singletons, globals, and contained objects making up a smaller, but still non-trivial (up to 31%) amount of the 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. 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 percentage 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 simple annotations to express the simple and common structures that our results indicate dominate heaps.

When we consider how much sharing information our current categorization scheme captures in Table 2, we see that our abstraction captures at least 75% of fields and in some cases over 90% of the sharing relations of fields. The confidence interval for the mean proportion of relations that our approach leaves unclassified is 5%–17%. For edges, the breakdown is more variable but the mean proportion of unclassified edges is only 4%–18%. Further, our analysis is unaware of used-defined immutable types or the sharing of singleton or global not defined in Section 3.2, so some portion of the sharing we report is 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.

5.4 Conceptual Component Accuracy

Our results rest on accurately identifying a programmer's intended, role-based groupings of heap objects. We used our structural indistinguishability principles introduced in Section 2 and formalized in Section 3 to approximate these grouping as equivalence classes of heap objects. Thus, our results rest on 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 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 *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 capturing actual 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, int the *HeapDbg* work [21], the tool that realizes our heap abstraction has been successfully used to identify and fix memory issues in the DaCapo benchmarks. Second, we informally interviewed developers who had used the HeapDbg tools. They reported that HeapDbg was useful for finding memory bloat in production software. They also found HeapDbg to be useful for program understanding and that it generally grouped objects into components in the expected ways: "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." These testimonials provide further confidence that the structures on which we performed our measurements were good approximations of the developers' intent and conceptualization of the heap structures in their programs.

The combination of quantitative evidence provided by the high information content of the conceptual component graphs on which we perform our measurements and the initial qualitative developer experience with the HeapDbg visualization tool provides evidence supporting our hypothesis.

5.5 Threats to Validity

There are a number of possible confounding factors that may impact this study as well as several limitations on what can be inferred from the results. There are of course the standard set of problems that can confound an empirical study such as bugs in the implementation or mistakes performed in analysis. However, there are a number of issues that are particularly relevant for the study done in this paper.

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 empirical 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 real-world Java programs and which has been used in a number of prior studies. Internal validity is related to how well associations or correlations are indicative of causal effects. The goal of this study has been more 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 our computation 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.

6 Related Work

A variety of questions about the structure of the program heap have been explored in previous empirical studies using runtime analysis. Often these studies have focused on the *shape* of the data structures [3, 14, 24] that appear in the programs and use type or reachability from root locations to define the sets of objects over which to compute shape information. This, relatively coarse, decomposition of the heap resulted in lower resolution information [24] than our approach extracts (although the results are broadly similar), while Jump et. al. [14] and Albiz et. al. [3] focused primarily on library data structures. Mitchell et. al. [22, 23] and Noble et. al. [25] look at the heap through the lens of ownership and dominator structures; in constract, our work uses conceptual components and injectivity as introduced by Marron in [19–21]. The work in [12] performs an extensive evaluation of reachability in the context of understanding object lifetime for garbage-collection applications. The original paper [4] introducing the DaCapo benchmarks (used in this work) also includes an extensive evaluation of both general properties of the benchmarks and a variety of information on how these programs allocate and use memory.

Other empirical studies have been based on static analysis information [10, 17] or on using runtime information to explore the precision of various static heap analysis approaches [15]. The work by Hackett and Aiken in [10] explores how aliasing is used in systems software and, much like the work in this paper, attempts to relate the types of aliasing relations that are seen to concepts in the source code. This work is focused specifically on aliasing and does not explore as wide a range of properties as the work here and also looks only at relations between individual objects instead of larger scale conceptual components. Work by Ma and Foster [17] explores a rich set of sharing and structural annotations that would be suitable for use as method pre/post conditions and develops a static analysis to extract these conditions. Their empirical study employs their static analysis, by construction a conservative over-approximation of actual program behavior, to compute the prevalence of various relations. Thus, their work provides an *lower* bound (possibly a very conservative on) 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. Finally, Linag et. al. [15] compare runtime aliasing information with the results of several static points-to analyses to evaluate the absolute precision of which they are capable and how this impacts a number of client optimization applications. Empirical studies of heap structures have also been performed from a developer perspective: Work by Abi-Antoun and Alderich has used static analysis to compute possible ownership domain information that is then evaluated via end developers [1, 2].

7 Future Work and Conclusion

In an effort to understand the heaps of real-world programs have, we have analyzed the heap structures of a number of DaCapo applications. Using an empirical and inferential statistical approach, we were able to identify properties (*e.g.* confidence intervals on categories of sharing) and relationships (*e.g.* the dominance of atomic shapes over linear and cyclic) that were statistically significant and thus generalize to the population of programs of which DaCapo is representative. We found that the shape of heap structures are fairly simple, with the vast majority made up of atomic shapes and that approximately half of all data structures on the heap have local ownership. Sharing occurs between conceptual components more often than within them and although a high proportion (up to 66%) of objects are shared, the majority of sharing is occurs over immutable types, with a smaller proportion of singletons and globals. That is, in practice, sharing occurs via fairly simple and common development idioms. Lastly, our abstraction not only classifies a large majority (up to 75% of fields and 90% of edges), but also partitions the heap into categories that 1) show clear differences in occurrence, 2) model simple and common programming practices, and 3) are useful and intuitive to practitioners. These findings have implications for future research directions. Work on technologies such as garbage collection and other dynamic program analysis will achieve a high return by focusing on simple heap structures. Our results imply that simple annotations that are more accessible to practitioners will be capable of expressing most heap structures and sharing that occur in practice and that program analyses may improve their scalability by viewing the heap as a weaker adversary without losing much precision.

References

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

[2] M. Abi-Antoun, N. Ammar, and T. LaToza. Questions about object structure during coding activities. In *CHASE*, 2010.

[3] S. Albiz and P. Lam. Implementation and use of data structures in Java programs. In *Tech. Report.*, 2011.

[4] 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.

[5] N. R. Cameron, J. Noble, and T. Wrigstad. Tribal ownership. In *OOPSLA*, pages 618–633, 2010.

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

[7] I. Dillig, T. Dillig, and A. Aiken. Symbolic heap abstraction with demand-driven axiomatization of memory invariants. In *OOPSLA*, 2010.

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

[9] M. Fahndrich and R. DeLine. Adoption and focus: practical linear types for imperative programming. *SIGPLAN Not.*, 37:13–24, May 2002.

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

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

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

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

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

[15] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, 2010.

[16] B. Liskov. Data abstraction and hierarchy. In *OOPSLA*, 1987.

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

[18] M. Marron. Structural analysis: Combining shape analysis information with points-to analysis computation. In *Submission*, 2012.

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

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

[21] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting runtime heaps for program understanding. In *Submission*, 2012.

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

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

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

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

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

[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, 2011.

[28] P. Wadler. Linear types can change the world! In *Programming Concepts and Methods*. North, 1990.

[29] H. Yang, O. Lee, J. Berdine, C. C. andByron Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.