Heap Analysis Design: An Empirical Approach

Mark Marron

Microsoft Research
mark.marron@microsoft.com

Abstract. Despite extensive research the construction of a precise and scalable static heap analysis for object-oriented programs remains an open problem. This paper argues that much of the difficulty is the result of empirically unvalidated and inappropriate design decisions. We examine three of them and determine that in practice: (1) strong updates are not necessary for obtaining precise results (2) fixed naming schemes for defining abstract memory locations are fundamentally limiting and are not required for efficiency, and (3) shape/sharing in the heap is generally simple and can be described using a simple abstract heap model. Using these results we construct a new heap analysis and experimentally demonstrate that it is both precise, capable of supporting program optimizations/tools which require sophisticated information on the structure of the heap, and scalable, allowing the analysis to be run on real world programs such as luindex/lusearch from DaCapo.

1 Introduction

Despite extensive research efforts and large numbers of published papers the construction of a static heap analysis that is both capable of extracting precise information about the program heap and which is scalable to real-world programs remains an open problem. This lack of practical high-accuracy heap analysis tools is a major impediment to research in a range of areas – e.g., memory management, parallelization, test generation, etc. – from across the programming language and software engineering fields. Continued development of points-to analyses, and the creation of clever application specific extensions, have mitigated some of these problems. However, there is a fundamental gap between what can be done when working with variations on classic points-to information vs. what can be done when precise information about sharing in containers, shapes of data structures, and ownership properties are available.

This paper revisits several key assumptions about the design and construction of a shape style analysis in light of recent empirical studies [1, 4, 5, 7, 37]. These studies suggest that much of the conventional wisdom about how a heap analysis should be constructed is incorrect when working with modern object-oriented and garbage-collected languages such as Java or C#.

Based on these insights we construct a hybrid static heap analysis, *Jackalope*, that combines an expressive abstract heap model with a set of efficiently computable abstract transfer functions. This analysis starts with the classic storage shape (or points-to) graph from Chase et. al. [9]. The basic storage shape graph is augmented with additional labels for the *shape* of the regions that the nodes represent and *injectivity* [26] (must not-alias) of the pointers that the edges represent. The analysis allows the creation of an unbounded number of abstract locations and uses a normal form, instead of a fixed naming scheme for abstract heap locations, to ensure termination. The simple flow-sensitive and subset-based transfer functions are based on set-operations and weak updates.

Our experimental evaluation shows that, despite the use of weak updates, the analysis is still able to produce accurate analysis results for the heap properties of interest. The static analysis identifies that in most programs over 80% of regions do not have any internal shape (are atomic) and that 85% or more of the pointer sets identified between regions do not contain any aliased pointers (are injective). These results are near the limits possible with the domain when compared to the runtime analysis results in [4]. The analysis is able to compute these results for large and complex programs, 60K bytecodes and 2000+ methods, using less than 190 seconds and 180MB of memory. We demonstrate how these results translates into both improved results in existing client applications and to show how they can be used to enable previously impractical applications we present three sample clients that utilize the results of the Jackalope analysis – thread-level parallelization [12, 26], static memory reclamation [18], and automated unit-test generation.

In summary this paper makes the following contributions:
- This paper revisits three fundamental issues in the design of a heap analysis — strong updates, abstract memory location definitions, and which shape/sharing properties are modeled — and shows that, for object-oriented programs in managed languages, common choices for these design decisions are inappropriate.
- In light of the first contribution, this paper presents alternative design decisions and uses them to construct a new heap analysis. This analysis provides precise heap structure information which cannot be obtained via existing points-to analyses. The analysis is also able to analyze large/complex programs which are beyond the scope of existing shape analyses.
- We evaluate the precision and scalability of the analysis on well known benchmarks. The evaluation shows that the analysis is able to efficiently extract precise sharing and shape information at a near optimal level, compared to the expected ground truth values, and that the results enable the example optimizations to be applied.

2 Revising Analysis Design Assumptions

We begin by re-examining conventional wisdom around three fundamental decisions in the design of a heap analysis: the use of strong updates, the definition of abstract locations, and the shape/sharing information that is tracked. Based on this re-examination we identify alternative hypotheses to inform the design and construction of a heap analysis.

2.1 Strong Updates

A major issue in a heap analysis is support for *strong updates* [3, 13, 22, 31, 32]. where the analysis can discard the previous contents of the store destination (as opposed to a *weak update* which conservatively merges the old and new content values). It is frequently assumed that aggressively performing strong updates is critical to precision. Seminal papers, such as Sagiv et. al. [32] and Ghiya and Hendren [16], demonstrate the need for strong updates when analyzing programs which destructively transfer ownership. Performing a strong update requires uniquely identifying the memory location that is being modified by the write. This can be handled in a number of ways [6, 13, 32] but fundamentally involves casewise reasoning over the possible alias configurations of the program. The combination of casewise reasoning, possible presence of recursive data structures, and existence of aliasing, lead to rapid increases of the complexity and computational cost of the analysis.

Assumption 1 (Strong Updates) *Strong updates are needed in order to precisely analyze heap shape and sharing properties.*

While this assumption may hold for the type C/C++ programs examined in some of the foundational work on heap analysis it is not clear if it generalizes to programs written in other languages. For example when analyzing a purely functional program performing only weak updates is as precise as aggressively performing strong updates. Thus, a key question is: Under what circumstances are strong updates critical to achieving high precision and how large is the impact of using weak-updates?

To understand the importance of strong updates when analyzing object-oriented programs written in memory managed languages we look to the literature to better understand how these programs construct and modify structures on the heap. Results in [5, 7, 37] show that object-oriented programs exhibit extensive *mostly-functional* behaviors: the use of *final* (or *quiescing*) fields, *stationary* fields, copy construction, and when fields are updated the target is often a newer or freshly allocated object.

For fields which are only assigned to a single time strong updates provides no improvement over weak updates. In the case of a field which is updated to a newly allocated object strong updates provide some benefit by eliminating the old alias but, as shown in the experimental results in this paper, the impact is small. Finally, the case of destructive reorganization of a data structures, the case where strong updates have a large impact, are infrequent in the object-oriented programs we are interested in. Thus, we revise the assumption for the need for strong updates to:

Revised Assumption 1 (Strong Updates) *Strong updates are not critical to obtaining precise shape and sharing information when analyzing object-oriented code in managed languages.*

2.2 Allocation and Heap Abstraction

There is a large divergence in how object allocation is treated and how the heap is abstracted in the standard forms of points-to analysis [2, 29, 35] and standard forms of shape analysis [31, 33]. The general approach taken by points-to analyses creates a fixed number of abstract memory locations based primarily on syntactic features of the program, such as allocation sites, and uses them to abstract the heap. Conversely, shape style analyses generally create a fresh abstract location every time the analysis visits an allocation site and uses an notion of equivalence between abstract locations to constrain the heap model.

Assumption 2 (Allocation/Abstraction) *A heap analysis can either use a fixed set of abstract locations for scalability (at the cost of precision) or a dynamic allocation and abstraction approach for precision (at the cost of scalability).*

Techniques such as object-sensitivity [29, 34] or recency [3, 9] improve on the precision of simple allocation site based naming schemes but do not solve the fundamental problem of apriori picking a fixed set of abstract locations. This either results in the creation of too few locations to capture important features about how pointers are aliased or too many locations which needlessly increases the computational costs. The alternative of creating a fresh abstract location every time an allocation statement is visited [31, 33] trivially ensures that the heap can be partitioned as finely as needed. However, creating fresh abstract locations at every visit to an allocation can lead to non-termination and poor scalability unless there is a mechanism to merge the locations during the analysis. Thus, the critical question is: Does there exist a *normal form* [6, 9, 27, 33] that provides a fine grained partitioning of the heap when needed for precision but which will effectively merge unneeded abstract locations scalability?

Recent studies of the heap structures in object-oriented programs [4, 27] provide a set of empirically validated concepts to use as a basis for a suitable normal form. This work hypothesizes that developers think of objects in terms of the roles they play in the programs. Further, the formulations in [4, 27] hypothesize that the role of an object can be inferred from where pointers to the object 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. Thus we revise the assumption of how to handle abstract heap locations to:

Revised Assumption 2 (Allocation and Abstraction) *There exists a normal form definition such that a heap analysis can use a dynamic allocation and abstraction approach while being both precise and scalable.*

2.3 Shape and Sharing Representation

The sample optimization clients in Section 6 show that the limited sharing information provided by basic points-to analyses [2, 29, 35] is insufficient for many applications. Conversely, work on shape analysis has developed sophisticated logics, which can track complex shape [6, 24, 31, 32] and sharing relations [14, 17, 26, 28, 31, 32].

Assumption 3 (Shape and Sharing Representation) *A heap analysis needs sophisticated logics to precisely analyze complex recursive data structures and pointers which alias in subtle ways.*

This assumption is clearly correct when considering some classes of programs (particularly low-level code). However, recent studies [1, 4] of object-oriented programs in managed languages show that (1) they use recursive data structures sparingly, (2) that standard library containers are used extensively, and (3) that objects are almost always shared in small number of simple (and idiomatic) ways. Thus, we ask: are there simple heap representations that precisely describe the commonly appearing shape/sharing properties and which can be efficiently operated on during the heap analysis?

The results in [4] show that, after grouping objects by their roles in the program, only a small minority (less than 5% on average) are part of any recursive data structure. The results in [1] confirm this and show that in practice programmers make heavy use of the builtin container libraries. The results in [4] also show that the percentage of objects which were identified as having the same role and contained pointers which were aliased was very small (less than 7% of edges represent aliasing pointers to mutable objects).

These studies imply that the benefit from the use of more than a trivial binary *recursive/non-recursive* model of shape properties is likely to be very small. Further, that when analyzing object-oriented programs, that it is sufficient to extend the basic *storage shape graph* [9] (or points-to graph) with a simple binary *injective/may-pairwise-alias* [4, 26, 27] property to describe sharing. However, given the extensive use of arrays and standard library containers it is critical that the operations on them are handled carefully [14, 28]. Thus we revise the assumption on the sophistication of the sharing and shape description logics to:

Revised Assumption 3 (Shape and Sharing Representation) *Data structures are mostly non-recursive and the majority of the sharing is simple. Thus, simple logics for shape and sharing properties are sufficiently expressive.*

3 Abstract Heap Domain

This section formalizes concrete program heaps and the relevant properties that are tracked by the Jackalope heap analysis. These definitions are designed to support the expression of a range of generally useful properties (e.g., shape, sharing, reachability) which are useful for a wide range of client optimization and error detection applications (such as examined in Section 6).

3.1 Concrete Heaps

The state of a concrete program is modeled using an environment Env mapping from variables to addresses, a store σ mapping from addresses to objects, and a set of objects Objs. We refer to an instance of an environment, a store, and a set of objects as a *concrete heap*. Given a program that defines a set of concrete types (Types) and a set of fields including array indices in $\mathbb{N}$ (Labels) a concrete heap is a tuple $(\mathsf{Env}, \sigma, \mathsf{Objs})$ where:

$$\begin{aligned}
\mathsf{Env} &: \mathsf{Vars} \rightharpoonup \mathsf{Addr}\\
\sigma &: \mathsf{Addr} \to \mathrm{Objects} \cup \{\mathsf{null}\}\\
\mathsf{Objs} &: \{o | o \in \mathrm{Objects}\}\\
&\mathrm{Objects} = \mathsf{ObjID} \times \mathsf{Types} \times (\mathsf{Labels} \rightharpoonup \mathsf{Addr})\\
&\mathsf{ObjID} = \mathbb{N} \quad \mathsf{Addr} = \mathbb{N} \cup \{\bot\}
\end{aligned}$$

Each object o in the set Objs is a tuple consisting of a unique identifier for the object, the type of the object, and a map from field labels to concrete addresses for the fields defined in the object. The set Addr contains a $\bot$ element which out of bound array index labels are mapped to. We assume that the objects in Objs 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 sets Types and Labels.

In the following definitions we use the notation $\mathsf{Ty}(o)$ to refer to the type of a given object. The usual notation $o.l$ to refers to the value of the field (or array index) l in the object. We define a helper function $\mathsf{Flds} : \mathsf{Types} \mapsto \mathcal{P}(\mathsf{Labels})$ to get the set of all fields (or array indices) that are defined for a given type. Finally, for a *non-null pointer* p associated with object o and label l, where $\sigma(o.l) \neq \mathsf{null}$ in the concrete heap $(\mathsf{Env}, \sigma, \mathsf{Objs})$, we use the notation: $p = (o, l, \sigma(o.l))$.

In the context of a specific concrete heap, $(\mathsf{Env}, \sigma, \mathsf{Objs})$, a *region* of memory is a subset of concrete heap objects $C \subseteq \mathsf{Objs}$. It is useful to define the set $P(C_1, C_2, \sigma)$ of all non-null pointers crossing from region C_1 to region C_2 as:

$$P(C_1, C_2, \sigma) = \{(o_s, l, \sigma(o_s.l)) \mid o_s \in C_1 \wedge \sigma(o_s.l) \in C_2\}$$

Injectivity. Given regions C_1, C_2 in the heap $(\mathsf{Env}, \sigma, \mathsf{Objs})$ and the set of non-null pointers with the label l from C_1 to C_2, a frequent and important question is: do any of the pointers in this set alias? To answer this question we define the property of *injectivity.*

Definition 1 (Injective). *A set of non-null pointers P_l with label l from C_1 to C_2 is* injective*, $\mathsf{inj}(C_1, C_2, l, \sigma)$, if: $\forall (o_s, l, o_t), (o'_s, l, o'_t) \in P_l \,.\, o_s \neq o'_s \Rightarrow o_t \neq o'_t$.*

As a special case we consider non-null pointer sets stored in arrays in C_1. We define the property of *array injectivity* which includes the assertion that any pair of pointers stored in two different locations in the same array do not alias.

Definition 2 (Array Injective). *A set of non-null pointers $P_{[]}$ from arrays in C_1 which point-to objects in C_2 is* array injective*, $\mathsf{inj}_{[]}(C_1, C_2, \sigma)$, if: $\forall\ (o_s, i, o_t)$, $(o'_s, i', o'_t) \in P_{[]} \,.\, (o_s \neq o'_s \vee i \neq i') \Rightarrow o_t \neq o'_t$.*

These definitions capture the general case of an injective relation being defined from a set of objects and fields to a set of 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. The notion of injectivity asserts a complete absence of aliasing which is the strongest possible assertion wrt. aliasing on the set.

Lemma 1 (Injectivity is Transitive). *Given* 3 *regions* C_1*,* C_2*, and* C_3 *with the* injective *pointer sets* $P_l(C_1, C_2, \sigma)$ *and* $P_{l'}(C_2, C_3, \sigma)$*. If* $o_a, o_b \in C_1$ *and* $o_a \neq o_b$ *then the injectivity of the first pointer set implies that* $o_a.l \neq o_b.l$ *in* C_2*. Similarly the injectivity of the second pointer set implies that* $o_a.l.l' \neq o_b.l.l'$ *in* C_3*. Thus, the pointer path,* $P_l(C_1, C_2, \sigma) \circ P_{l'}(C_2, C_3, \sigma)$*, is injective.*

The transitive property of injectivity (and array injectivity) allows us to use local injective information to reason about disjointness on full heap *access paths*.

Shape. We characterize the shape of regions of memory using two simple categories: sets of objects without any internal connectivity and sets of objects with some unknown internal connectivity.

- The predicate $\mathsf{none}(C, \sigma)$ holds if $P(C, C, \sigma) = \emptyset$.
- The predicate $\mathsf{any}(C, \sigma)$ holds if $P(C, C, \sigma) \neq \emptyset$.

3.2 Abstract Heap

An abstract heap graph is a tuple: $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Objs}})$ where:

$$\widehat{\mathsf{Env}} : \mathsf{Vars} \rightharpoonup \widehat{\mathsf{Addr}}$$

$$\widehat{\sigma} : \widehat{\mathsf{Addr}} \to \mathsf{Inj} \times \mathcal{P}(\text{Nodes})$$

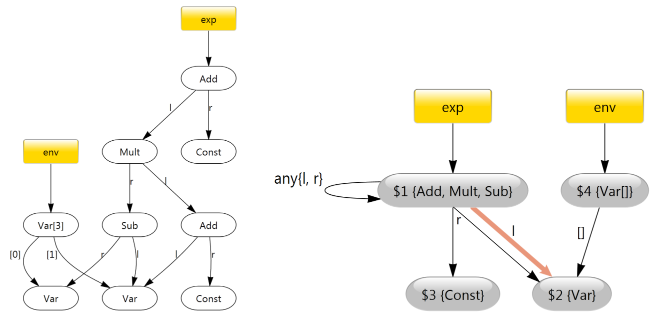
$$\widehat{\mathsf{Objs}} : \{n | n \in \text{Nodes}\}$$

$$\text{Nodes} = \mathsf{NodeID} \times \mathcal{P}(\mathsf{Types}) \times \mathsf{Sh} \times (\widehat{\mathsf{Labels}} \rightharpoonup \widehat{\mathsf{Addr}})$$

$$\mathsf{Sh} = \{\mathsf{none}, \mathsf{any}\} \quad \mathsf{Inj} = \{\mathit{true}, \mathit{false}\} \quad \mathsf{NodeID} = \mathbb{N} \quad \widehat{\mathsf{Addr}} = \mathbb{N} \cup \{\bot\}$$

The abstract store ($\widehat{\sigma}$) maps from abstract addresses to tuples consisting of the injectivity associated with the abstract address and a set of target nodes. Each node n in the set $\widehat{\mathsf{Objs}}$ is a tuple consisting of a unique identifier for the node, a set of types, a shape tag, and a map from abstract labels to abstract addresses. The use of an infinite set of node identity tags, NodeID, allows for an unbounded number of nodes associated with a given type/allocation context allowing the local analysis to precisely represent freshly allocated objects for as long as they appear to be of special interest in the program (as defined via the normal form in Section 4 and used in the transfer functions in Section 5). The abstract labels ($\widehat{\mathsf{Labels}}$) are the field labels and the special label []. The special label [] abstracts the indices of all array elements (i.e., array smashing). Otherwise an abstract label $\widehat{l}$ represents the named member field.

As with the objects we introduce the notation $\widehat{\mathsf{Type}}(n)$ to refer to the type set associated with a node. The notation $\widehat{\mathsf{Shape}}(n)$ is used to refer to the shape



(a) A Concrete Heap. (b) Corresponding Abstract Heap.

Fig. 1. Concrete and Abstract Heap

property, and the usual $n.\widehat{l}$ notation to refer to the abstract value associated with the label $\widehat{l}$. Since the abstract store ($\widehat{\sigma}$) now maps to tuples of *injectivity* and node target information we use the notation $\widehat{\mathsf{Inj}}(\widehat{\sigma}(\widehat{a}))$ to refer to the *injectivity* and $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{a}))$ to refer to the set of possible abstract node targets associated with the abstract address. We define the helper function $\widehat{\mathsf{Flds}} : \mathcal{P}(\mathsf{Types}) \rightarrow \mathcal{P}(\widehat{\mathsf{Labels}})$ to refer to the set of all abstract labels that are defined for the types in a given set (including [] if the set contains an array type).

3.3 Example Heap

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

Figure 1(b) shows the corresponding normal form (see Section 4) abstract heap for this concrete heap. To ease discussion we label each node in a graph with a unique node id ($id). The abstraction summarizes the concrete objects into three regions. The regions are represented by the nodes in the abstract heap graph: (1) a node representing all interior recursive objects in the expression tree (`Add`, `Mult`, `Sub`), (2) a node representing the two `Var` objects, and (3) a node representing the two `Const` objects. The edges represent possible sets of non-null cross region pointers associated with the given abstract labels.

Details about the order and branching structure of expression tree nodes are absent (shown via the self-edge with the *any* label). However, the abstract graph maintains other useful sharing properties of the expression tree, namely that no `Const` object is referenced from multiple expression objects. On the other hand, several expression objects might point to the same `Var` object. The abstract graph shows this possible non-injectivity using wide orange colored edges (if color is available), whereas normal edges indicate injective pointers. Similarly the edge from node 4 (the `env` array) to the set of `Var` objects represented by node 2 is injective, not shaded and wide. This implies that there is no aliasing between the pointers stored in the array, i.e. every index in the array contains a pointer to a unique object (which is critical to understanding the semantics of any loops that operate on the contents of this array).

4 Normal Form

The abstract heap domain is infinite which allows substantial flexibility when defining the transfer functions and more precise results when analyzing straight line blocks of code. However, this is problematic when defining merge/equality operations and can result in the final analysis having an unacceptably large computational cost. Thus, we define an efficiently computable normal form to ensure that the set of normal form abstract heaps for any given program is *finite* and that the abstract heaps in this set can easily be merged and compared.

To guide the construction of the normal form definition we utilize the hypothesis that developers think of objects in terms of the roles they play in the programs. Our approach for identifying roles of objects builds on the concepts in [4] and is based on the *has-a* pointer to relation plus the standard notions of recursive data-structure identification [6, 11, 20, 24, 27, 30, 31], predecessors [6, 9, 11, 27, 33], and grouping the contents of containers [9, 14, 17, 27, 28].

Definition 3 (Normal Form). *An abstract heap is in normal form iff:*
1. *All nodes are reachable from a variable or static field.*
2. *All recursive structures are summarized (Definition 4).*
3. *All equivalent successors are summarized (Definition 6).*
4. *All variable equivalent targets are summarized (Definition 7).*

The normal form definition has three key properties: (1) the abstract normal form heap graphs have a bounded depth, (2) each node has a bounded out degree, and (3) for each node the possible targets of the abstract addresses associated with it are unique wrt. the label and the types in the target nodes. The first two properties ensure that the number of abstract heaps in the normal form set are finite, while the third enables efficient merge and compare operations on the bounded labeled graphs.

4.1 Equivalence Partitions

The properties (*recursive structures*, *equivalent successors*, and *equivalent targets*) are defined in terms of congruence between abstract nodes. Thus, the transformation of an abstract heap into the corresponding normal form is fundamentally a congruence closure computation over the nodes in the abstract heap followed by merging the resulting equivalence sets. We build a map from the abstract nodes to equivalence sets (partitions) $\mathit{\Pi} : \widehat{\mathsf{Objs}} \to \{\pi_1, \ldots, \pi_k\}$ where $\pi_i \in \mathcal{P}(\widehat{\mathsf{Objs}})$ and $\{\pi_1, \ldots, \pi_k\}$ are a pairwise disjoint cover of $\widehat{\mathsf{Objs}}$. Initially $\mathit{\Pi}$ consists of singleton sets i.e., $\forall n \in \widehat{\mathsf{Objs}} \,.\, \mathit{\Pi}(n) = \{n\}$.

The first part of the normal form computation identifies nodes that may be part of unbounded depth structures. This is accomplished by examining the type system for the program and identifying all the types that are part of the same recursive type definitions. We say types τ_1 and τ_2 are *recursive* ($\tau_1 \sim \tau_2$) if they are part of the same recursive type definition.

Definition 4 (Recursive Structure). *Given partitions π_1 and π_2 we define the* recursive structure *congruence relation as:* $\exists \tau_1 \in \bigcup_{n_1 \in \pi_1} \widehat{\mathsf{Type}}(n_1),\ \tau_2 \in \bigcup_{n_2 \in \pi_2} \widehat{\mathsf{Type}}(n_2) \,.\, \tau_1 \sim \tau_2 \wedge \exists n \in \pi_1,\ \widehat{l} \in \widehat{\mathsf{Flds}}(\widehat{\mathsf{Type}}(n)) \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap \pi_2 \neq \emptyset$.

The other part of the normal form computation is to identify any partitions that have *equivalent successors* and variables that have *equivalent targets*. Both of these operations depend on the notion of a successor partition which is based on the underlying structure of the abstract heap graph in a standard way:

$$\pi_1 \text{ successor of } \pi_2, \widehat{l} \Leftrightarrow \exists n_2 \in \pi_2 \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n_2.\widehat{l})) \cap \pi_1 \neq \emptyset$$

Definition 5 (Partition Compatibility). *We define the partition compatibility relation as:* $\mathrm{Compatible}(\pi_1, \pi_2) \Leftrightarrow \bigcup_{n' \in \pi_1} \widehat{\mathsf{Type}}(n') \cap \bigcup_{n' \in \pi_2} \widehat{\mathsf{Type}}(n') \neq \emptyset$.

Definition 6 (Equivalent Successors). *Given π_1, π_2 which are successors of π on labels $\widehat{l_1}$, $\widehat{l_2}$ we define the* equivalent successors *relation as:* $\widehat{l_1} = \widehat{l_2} \wedge \mathrm{Compatible}(\pi_1, \pi_2)$.

Definition 7 (Equivalent on Targets). *Given a variable v and two partitions π_1, π_2 where v refers to a node in π_1 and a node in π_2 we define the* equivalent targets *relation as:* π_1, π_2 *only have var predecessors* $\wedge\, \mathrm{Compatible}(\pi_1, \pi_2)$.

Using these relations we can efficiently compute the congruence closure over an abstract heap producing the partitions for the normal form abstract heap (Definition 4). This computation is done via a standard worklist algorithm that merges partitions that contain equivalent nodes.

4.2 Computing Summary Nodes

After partitioning the nodes in the graph with the congruence closure computation we merge the nodes in each partition into a summary node. We also need to update target and injectivity information for the abstract store. Given set of nodes, π, that we want to replace with a new summary node, n_s, we compute the following abstract properties for the summary node and store $\widehat{\sigma}_s$:

$$n_s = (\text{fresh } \mathsf{NodeID}, \sqcup_{type}(\pi), \sqcup_{shape}(\pi), \{[\widehat{l} \mapsto \widehat{a_{\widehat{l}}}] \mid \widehat{l} \in \widehat{\mathsf{Flds}}(\sqcup_{type}(\pi)), \widehat{a_{\widehat{l}}} \text{ is fresh}\})$$
$$\widehat{\sigma}_s = \mathit{MergeStore}(\widehat{\sigma}_s, \widehat{l}, \pi, n_s) \text{ for each } \widehat{l} \in \widehat{\mathsf{Flds}}(\sqcup_{type}(\pi))$$
$$\sqcup_{type}(\pi) = \bigcup_{n \in \pi} \widehat{\mathsf{Type}}(n)$$

Shape. The *Shape* information depends on the shapes of the individual nodes that are being grouped and the connectivity properties between them. We perform a traversal of the subgraph of the partition to see if there are any internal edges in the partition. Then based on the discovery of any internal edges in this subgraph we compute the shape as $\sqcup_{shape}(\pi) = struct(\pi) \sqcup \bigsqcup_{n \in \pi} \widehat{\mathsf{Shape}}(n)$ where: $struct(\pi)$ is: none if no internal edges exist and any otherwise.

Injectivity and Abstract Targets. Given a mapping from the partitions to the new summary nodes, $\Phi : Img(\Pi) \to \{n_{s_1}, \ldots, n_{s_k}\}$, then for each label, $\widehat{l}$, and abstract address, $\widehat{a}_{\widehat{l}}$, that may appear in a summary node, n_s, we set the values in the abstract store as:

$$MergeStore(\widehat{\sigma}_s, \widehat{l}, \pi, n_s) = \widehat{\sigma}_s + [n_s.\widehat{l} \mapsto (inj, trgts)] \text{ where}$$

$$trgts = \{\Phi(\Pi(n')) \mid n' \in \textstyle\bigcup_{n \in \pi} \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l}))\}$$

$$inj = \forall n \in \pi \,.\, \widehat{\mathsf{Inj}}(\widehat{\sigma}(n.\widehat{l})) \wedge \forall n' \in \pi \setminus \{n\} \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n'.\widehat{l})) = \emptyset$$

Injectivity is the logical conjunction of the injectivity of all the source label locations, and that the respective targets sets are disjoint. In the case where the target sets do overlap, i.e., two distinct nodes have abstract labels/addresses that contain the same node, the resulting address may not only be associated with injective pointers. Thus, the injectivity value is conservatively set to *false* (i.e., *not injective*). The target set is simply the remapping of the old nodes in the target sets to the appropriate newly created summary nodes.

4.3 Normal Form on Example Heap

We can see how this normal form works by using it to transform the concrete heap in Figure 1(a) into its normal form abstract representation. This is done by creating an abstract heap graph that is isomorphic to the concrete heap (i.e., create a node for each concrete object and set the appropriate targets in the abstract store for each concrete pointer). The resulting isomorphic abstract heap is shown in Figure 2. The normal form partition identifies the nodes with the `Add`, `Sub`, and `Mult` types as being in the same partition (they are part of the same *recursive structure*). The presence of this partition will then cause all of the nodes with `Const` type (nodes 4, 7) to be identified as *equivalent successors* of the tree partition.

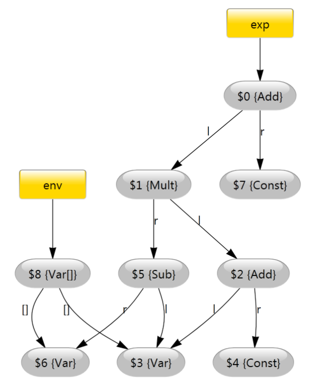


Fig. 2. Isomorphic Abstract Heap.

Finally, either due to the tree partition or the fact that all the nodes with `Var` type (nodes 3, 6) have references to them from node 8 (the `Var[]`) will cause all the partitions associated with `Var` types being identified as *equivalent successors*. The final partitioning after the congruence closure is:

$$\pi_1 = \{n_0, n_1, n_2, n_5\}, \pi_2 = \{n_3, n_6\}, \pi_3 = \{n_4, n_7\}, \pi_4 = \{n_8\}$$

The *Shape* for the partitions containing the `Var`, `Const` and `Var[]` nodes are trivial to compute as there are no internal references between the nodes in these partitions. The *shape* computation for the partition (π_1) containing the nodes in the expression structure requires a traversal of the four nodes, and as there are internal edges the layout is `any`.

For the abstract address associated with the expression tree partition (π_1) and the label `l` there are two nodes (n_2 and n_5) that refer to the same node (n_3) in partition π_2. Thus this abstract storage location is set as not injective (*false*). However, for the label `r` from partition π_1 the target sets are disjoint and thus the injectivity in the abstract store is set as injective (*true*). Similarly, the store location for the label `[]` out of the partition π_4 representing the targets of the pointers stored in the `env` array is set as injective. This results in the normal form abstract heap shown in Figure 1(b).

5 Abstract Transfer Functions

The transfer functions for the statements that are the most interesting from the standpoint of memory analysis are shown in Table 1. To focus on the central ideas we ignore null-pointer dereferences, array out-of-bounds errors, etc. The interprocedural analysis is context-sensitive (wrt. equality of abstract heap states) on calls to acyclic portions of the call graph. We refer to [25] for a full discussion of the how project/extend operations and method calls are handled.

Allocate. The allocation operation creates a fresh node (abstract memory location) at every visit to an allocation statement. The creation of a fresh node for each visit to an allocation site is critical to allowing the analysis to later model stores into/of this object and the impact on injectivity and shape precisely. Of course the creation of a new node at each visit to an allocation site creates a potential problem with the termination of the analysis as the abstract heap state may grow without bound. However, by applying the normal form operation from Section 4 at each control flow join point and at each call site we can be sure of the termination of the analysis (as the set of normal form graphs is finite).

Load. The load operation is a direct translation of the concrete semantics where the target set that is stored into the variable is the union of the target sets of the appropriate node sets. However, as variable locations always contain a single pointer we can strongly update the target set and set the *injective* value to *true*.

$$
\begin{aligned}
&v = \texttt{new type:}\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Objs}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Objs}}') \text{ where}\\
&\quad n = (\text{fresh } \mathsf{NodeID}, \mathsf{type}, \mathsf{none}, \{\widehat{l} \to \widehat{a_{\widehat{l}}} \mid \widehat{l} \in \widehat{\mathsf{Flds}}(\{\mathtt{type}\}), \widehat{a_{\widehat{l}}} \text{ is fresh}\})\\
&\quad \widehat{\sigma}' = \widehat{\sigma} + [\widehat{\mathsf{Env}}(v) \mapsto (\mathit{true}, \{n\})] + \{[n.\widehat{l} \mapsto (\mathit{true}, \emptyset)] \mid \widehat{l} \in \widehat{\mathsf{Flds}}(\{\mathtt{type}\})\}\\
&\quad \widehat{\mathsf{Objs}}' = \widehat{\mathsf{Objs}} \uplus \{n\}
\end{aligned}
$$

$$
\begin{aligned}
&v = v'.\widehat{l}\text{:}\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Objs}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Objs}}) \text{ where}\\
&\quad v'_{trgts} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v')))\\
&\quad \widehat{\sigma}' = \widehat{\sigma} + [\widehat{\mathsf{Env}}(v) \mapsto (\mathit{true}, \bigcup_{n \in v'_{trgts}} \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})))]
\end{aligned}
$$

$$
\begin{aligned}
&v.\widehat{l} = v'\text{:}\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Objs}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Objs}}) \text{ where}\\
&\quad v_{trgts} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v))) \quad v'_{trgts} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v')))\\
&\quad \forall n \in v_{trgts}\,.\text{ if } n \in v'_{trgts} \text{ then } \widehat{\mathsf{Shape}}(n) \leftarrow \mathsf{any}\\
&\qquad \widehat{\sigma}' = \widehat{\sigma} + [n.\widehat{l} \mapsto (\mathit{inj}, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cup v'_{trgts})]\\
&\qquad\quad \mathit{inj} = \widehat{\mathsf{Inj}}(\widehat{\sigma}(n.\widehat{l})) \wedge \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap v'_{trgts} = \emptyset
\end{aligned}
$$

Table 1. Abstract Semantics

Store. The store operation plays a central role in the analysis as it is where special care needs to be taken to update the injectivity and shape information. It first gathers all the possible objects that may be stored into (v_{trgts}) and all the possible objects that we may be storing references to (v'_{trgts}). In the update step we compute new values for the possible shape, the new target node set, and the new injectivity value. The shape information is handled by checking if the node we are storing into is in the set of possible targets. If it is then conservatively set the shape to the top value (any) otherwise the shape is unchanged.

The update to the abstract store involves taking the union of the old target set and the new target set (weakly updating the target set) and computing a new injectivity value. There are two cases we need to check to determine the new injectivity value. The first is if the old injectivity value was *false*, in which case we leave it as *false*. The second is if the new target set and the old target set overlap, in which case we cannot guarantee that the address is only associated with injective pointers. Again in this case we conservatively set the result as not injective. If neither of these cases occur then we mark the abstract address as containing injective pointers (i.e., the injective value is *true*).

6 Implementation and Evaluation

The domain, operations, and data flow analysis algorithm are implemented in C#. Our test machine is an Intel Core2 class processor at 2.5 GHz with 2 GB of RAM available. Our benchmark set consists of the power and bh programs from Jolden, the db and raytracer programs from SPEC JVM98, and the luindex and lusearch programs from the DaCapo suite [7]. Additionally, we use the runtime heap abstraction code, runabs, from [27]. The code for the benchmarks and Jackalope analysis tool are available online.[1]

6.1 Analysis Precision and Computational Cost

In evaluating the precision of the analysis we want to (1) minimize any biases present in the evaluation metric and (2) ensure that there is a ground truth to compare the results of the static analysis against. To do this we selected four properties described in previous work [4, 23, 26] which describe fundamental shape and sharing properties of a program heap.

- The prevalence of non-recursive data structures (*Atomic*) via the percentage of nodes that are *not* any or part of a strongly-connected component.
- The prevalence of objects that are not shared by other objects in *different* abstract heap locations (*Unique-In*) via the percentage of unique in-edges.
- The prevalence of objects that are not shared by objects in the *same* abstract heap location (*Injective*) via the percentage of edges which are injective.
- The prevalence of pointers which are not aliased in the local scope (*Local-Owner*) via the percentage of edges which are both *unique-in* and *injective*.

By measuring properties directly, instead of by indirectly through the results of a client application, we avoid biases introduced by the sensitivity or and idiosyncrasies of the selected clients. The reported values are computed using the abstract heap graphs at the entry of each method and aggregating (*Max* or *Average*) over all methods in the program.

The first two columns in Table 2 provide insight into the size of the largest abstract heaps in terms of the maximum number of Nodes/Edges seen in any abstract heap graph. The next four columns show the percentages of nodes/edges in the heap graphs that are *Atomic*, *Unique-In*, *Injective*, and *Local-Owners* respectively. The definitions used for shape and injectivity are two valued with a strong bias. In both definitions one value, none or *injective*, is maximally *precise* (i.e., there is no logically stronger shape or aliasing statement that can be made). Conversely, the other values, any or *non-injective*, are fully general. The analysis reports over 75% (and often over 90%) of heap regions as atomic. Similarly, the analysis reports a *unique-in* edge rate of over 60% in all benchmarks except raytracer and over 78% of edges as *injective*.

Overall, the raw rates for these values are, for most of the metrics/programs, quite high. Thus, we know that the analysis is capable of precisely resolving the shape and sharing of substantial parts of the program heap despite the substantial simplifications made in the analysis. However, it is not clear if the remaining nodes/edges truly do not satisfy the given property for some concrete execution of the program or if it is a result of imprecision in the analysis, for example due

[1] Source code and benchmarks are available at: `http://jackalope.codeplex.com/`

Benchmark	Max Nodes	Max Edges	Atomic%	Unique-In%	Injective%	Local-Owner%
power	13	12	75%	70%	83%	70%
bh	20	21	94%	83%	93%	82%
db	15	12	100%	81%	85%	73%
raytracer	50	70	83%	43%	84%	39%
luindex	136	177	99%	67%	86%	66%
lusearch	258	410	97%	64%	82%	62%
runabs	50	65	94%	60%	78%	50%

Table 2. The first two columns show the max nodes/edges seen in any abstract graph. The other columns show the percent of nodes which are *Atomic*, the percent of edges which represent *Injective* pointers, which are *Unique-In* edges to the target node, and the percentage of edges which represent *Local-Owner* pointers.

to weak-updates. To understand how close to optimal the results produced by the static analysis are we want to select a *ground truth* for comparison which represents an absolute limit on these rates. We can obtain an estimate of the absolute via the runtime profiling results [27]. As the static analysis cannot report rates larger than the runtime profiling results (otherwise it is unsound!) we can use the runtime results to estimate how much room there is for precision improvements in the the static analysis. According to the results the values for luindex are: 71% as unique-in, 92% as injective, and 69% of edges as local-owners. These values show that the expected room for further improvement is small and that the results produced by the analysis are near the limits of what is possible.

Table 3 shows the number of bytecode instructions, classes, and methods for each program after being translated into our internal representation. These numbers differ from previous work (e.g. [7]) due to porting of the benchmarks to C# and the processing done by the analysis when loading the .Net bytecode. This processing removes code which will never be executed (via a type based call-graph) and replaces several of the types/methods in the standard libraries with simplified versions or specialized domain operations [14, 28].

The *Analysis Cost* columns in Table 3 show the aggregate performance of the analysis on the benchmark set. These results show that the analysis described in this work is quite scalable and capable of analyzing complex programs. In the case of luindex the analysis requires only 10.7 seconds and up to 188.1 seconds for lusearch. The large jump in runtime, despite similar program sizes, is connected to the increased number of fixpoint iterations in lusearch and the increase larger number of methods in recursive call cycles. Table 3 also shows that the analysis requires at most 180 MB for any program.

6.2 Impact with Client Applications

The Barnes-Hut n-body algorithm operates on a set of `Body` objects representing planets, stars, etc. The `Body` objects use `MathVector` objects to represent the position, velocity and acceleration. During a force update computation a space decomposition tree is built based on the objects in the `bodies` array. Next the

Benchmark Statistics				Fixpoint			Analysis Cost	
Name	Insts (K)	Classes	Methods	Loop	Method	SCC	Time (sec)	Mem (MB)
power	9.9K	48	407	3	4	4	0.4s	15MB
bh	10.3K	50	442	4	6	4	0.7s	18MB
db	12.8K	57	529	4	1	0	0.8s	16MB
raytracer	16.2K	71	582	6	9	3	14.2s	25MB
luindex	52.5K	249	2259	6	10	4	10.7s	53MB
lusearch	60.0K	275	2479	6	17	17	188.1s	180MB
runabs	55.8K	261	2491	5	14	5	9.6s	64MB

Table 3. Program sizes in instructions (*Insts*), *Classes*, and *Methods*. Max fixpoint iterations for *Loops* and *Methods*. *SCC* is the number of methods in recursive cycles in the call-graph. Aggregate analysis costs, *Time* in seconds and *Memory* in MB.

program computes the new acceleration of each `Body` based on the force interactions with the other bodies. Then the `vp` method (below) is used to update the position and velocity fields of each `Body` object based on the new acceleration.

```
class Body : Node {MathVector pos, vel, acc, tacc; ...}
class MathVector {double[] data; ...}

public static void vp(Body[] bv, float dt) {
  for(int i = 0; i < bv.Count; ++i) {
    bv[i].pos = ComputeNewPos(bv[i].pos, bv[i].vel, dt);
    bv[i].vel = ComputeNewVel(bv[i].vel, bv[i].acc, dt);
    bv[i].acc = bv[i].tacc.CloneMV();
  }
}
```

The abstract heap that is constructed by the analysis for the program state at the entry of the `vp` method is shown in Figure 3. Of particular interest is that (1) all the edges in the graph end at different nodes and (2) all of the edges in the figure, including the edge labeled *[]* which represents the pointers in the `Body[]`, are narrow and uncolored (injective). These two features of the abstract heap imply that (1) there is no aliasing between pointers abstracted by different edges, i.e., stored in different fields and (2) there is no aliasing between pointers abstracted by the same edge, i.e., all pointer sets are injective (array injective).

Thread Level Parallelism The first client we examine is automatic thread-level parallelization of basic `for` loops. The first part of the condition definition ensures that the array contents do not alias. The third condition ensures that all modified locations are at the current loop iteration index (and do not introduce new aliases). The final condition ensures that reads are from un-modified locations or locations written in the current loop iteration.

Optimization 1 (Basic Array Loop Parallelization) *A loop which is iterating over an array A, using monotone index variable i, can be parallelized if:*

- *$\forall j \neq i$ then $A[j]$ and $A[i]$ do not alias ($A[j] \neq A[i]$).*
- *Writes are of the form `A[i].f = y` where subheaps for y and A are disjoint.*
- *Reads are from fields which are never modified or are of the form `x = A[i].f`.*

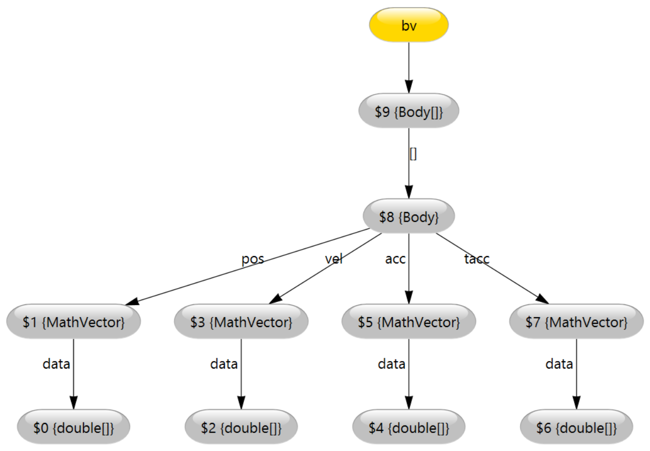


Fig. 3. Abstract Heap at Entry of `vp` Method.

The relevant heap properties needed to perform parallelization can be checked directly using the heap information provided by the Jackalope analysis. To see that for all $i \neq j$ the `Body` objects that are modified in the access paths, $bv[i]$ and $bv[j]$, are not equal we look at the injectivity of the edge representing the array contents. The edge representing the pointers stored in the array, labeled with *[]*, is array injective. By the definition of array injectivity, Definition 2, $i \neq j$ implies $bv[i] \neq bv[j]$. Further, the implementations of `ComputeNewPos` and `ComputeNewVel` return fresh `MathVectors`. Since the analysis creates new abstract locations for each visit to a `new` operation and only merges them later, the analysis trivially discovers that on all the assignments the right-hand side values are freshly escaping allocations in the current loop iteration. Based on these results we know the `vp` loop can be safely thread parallelized. After parallelizing the `vp` loop, and a parallel loop in the `stepSystem` method, with the C# *Task Parallel Library* we obtain a 2× speedup on our 4-core processor.

Static Memory Collection. The second optimization we examine is compiler directed static memory reclamation. Guyer et. al. [18] describe a technique for statically inserting frees based on the results of a specialized points-to analysis. A simplified version of the optimization is based on the intuition that if the memory location being overwritten contains the only reference to the target object (the target object has a reference count of one) then it can be reclaimed.

Optimization 2 (Basic Reclamation at Store) *Object o pointed to by $x.f$ ($A[i]$) can be explicitly reclaimed at the statement* `x.f = y` (`A[i] = y`) *if, on all executions of the statement, the pointer stored in $x.f$* (`A[i]`) *prior to the assignment is the unique reference to o.*

Using the combination of in-degree and injectivity information provided by the Jackalope analysis we can identify the stores in the `vp` loop as safe for explicit static reclamation. Since the in-degree to the target nodes of each left-hand side (`bv[i].pos`) is 1 and the single incoming edge is injective (no pointers abstracted by the edge alias) we know that each object abstracted by the target node is referred to a single pointer (i.e., the one abstracted by the `bv[i].pos` edge). Thus, we know the store satisfies the requirement for adding explicit reclamation. By recursively walking the abstract heap graph structure, looking at nodes representing sets of uniquely owned objects, we can also infer that the associated `double[]` arrays can be freed as well (i.e., `bv[i].pos.data`). These results enable the explicit reclamation of objects at all three assignment statements in the loop and enable an additional 8% of all allocated memory to be collected statically when compared to [18].

Unit-Test Generation. The last client application we examine is the automatic generation of unit-tests using the results of the Jackalope analysis. Tools such as Pex [36] or Korat [8] can be effectively used to automatically generate test inputs for methods with well constrained inputs such as scalar parameters or precisely specified heap structures. However, when run on methods with larger heap structures that lack precise specifications (as is the case in most programs) they often return large numbers of false positives and have low coverage of the code in the method under test. Running Pex on the `vp` method (via Pex Explore) results in 4 false-positives (null pointer exceptions), 1 passing test, and only achieves 33% block coverage in the code under test.

However, the information needed to generate high-coverage and low false-positive test cases can be extracted almost directly from the abstraction/concretization relations in Section 3. Each node corresponds to a existentially quantified set of objects and each edge to an existentially quantified set of pointers. The additional *injective* and `none` abstract properties correspond to assertions on these sets. The constraint system is output in the form of an executable constructor which Pex can explore and fill in the scalar fields in the objects that are constructed. When Pex is run on this combined setup and test code for the `vp` method the results are 0 failing test cases (which implies 0 false positives), 3 passing test cases, and 100% block coverage of the code under test.

7 Related Work

Much of the related literature on shape and points-to analysis was reviewed in Section 2. Thus, we focus on work that has explored other alternative heap analysis designs. Work on memory analysis by Latter et. al. [21] is based on a modular approach which first builds local shape graphs for each method via a local flow-insensitive points to analysis, and then merges (and clones as needed) these local graphs via a context-sensitive interprocedural analysis to produce the final result. Work in [15, 19] mixes shape and points-to analysis by first partitioning the heap into regions via a flow-insensitive points-to analysis followed by performing shape analysis based on these partitions. The work of Ghiya and Hendren [16] uses points-to and basic reachability predicates to compute shape information. The work by Ma and Foster [23] uses a predicate resolution algorithm to, on-demand, compute more complex sharing properties than can be provided by the base points to analysis. Similarly the work in [10] adds a form of reference counting to a heap analysis to track unique reference information.

8 Conclusion

This paper presented the empirically motivated design and construction of a heap analysis. A key insight was the need to revisit three critical decisions in the analysis design in the context of object-oriented program written in memory managed languages. Based on this study we constructed the Jackalope heap analysis under the hypotheses that: (1) strong updates are not required for precise analysis results, (2) fixed naming schemes, such as allocation or object-sensitivity, are fundamentally limiting, and (3) the vast majority of the shape/sharing which appears in practice is simple. Experimental evaluation of the resulting heap analysis showed that in practice it was precise, near the limit of what is possible even with a hypothetical perfect analysis, for a range of fundamental ownership and aliasing metrics. The ability to successfully apply the example client optimizations demonstrate the utility of the analysis results. Scalability was demonstrated via the analysis of several large and complex programs, 60K+ bytecodes and 2000+ methods, using less than 190 seconds and 180MB of memory.

References

[1] S. Albiz and P. Lam. Implementation and use of data structures in Java programs. http://patricklam.ca/dsfinder/, 2010.

[2] L. Anderson. Program analysis and specialization for the c programming language. Tech. report, PhD. Thesis, DIKU, report 94/19, 1994.

[3] G. Balakrishnan and T. Reps. Recency-abstraction for heap-allocated storage. In *SAS*, 2006.

[4] E. Barr, C. Bird, and M. Marron. Collecting a heap of shapes. In *ISSTA*, 2013.

[5] W. Benton and C. Fischer. Mostly-functional behavior in Java programs. In *VMCAI*, 2009.

[6] 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.

[7] 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.

[8] C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on Java predicates. In *ISSTA*, 2002.

[9] D. Chase, M. Wegman, and K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[10] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *ISMM*, 2006.

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

[12] D. Dig, M. Tarce, C. Radoi, M. Minea, and R. Johnson. Relooper: Refactoring for loop parallelism in Java. In *OOPSLA*, 2009.

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

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

[15] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *PLDI*, 2011.

[16] R. Ghiya and L. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, 1996.

[17] S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.

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

[19] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.

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

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

[22] O. Lhoták and K.-C. Chung. Points-to analysis with efficient strong updates. In *POPL*, 2011.

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

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

[25] M. Marron, O. Lhoták, and A. Banerjee. Programming paradigm driven heap analysis. In *CC*, 2012.

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

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

[28] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.

[29] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, 2005.

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

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

[32] S. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.

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

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

[35] B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.

[36] N. Tillmann and J. de Halleux. Pex-White box test generation for .NET. In *TaP*, 2008.

[37] C. Unkel and M. Lam. Automatic inference of stationary fields: A generalization of Java's final fields. In *POPL*, 2008.