Structural Analysis:

Combining Shape Analysis Information with Points-To Analysis Computation

Mark Marron

IMDEA Software Institute, Spain

mark.marron@imdea.org

Abstract

This paper introduces a new hybrid memory analysis, *Structural Analysis*, which combines an expressive shape analysis style abstract domain with efficient and simple points-to style transfer functions. By using insights from empirical studies of runtime heap structures and the programmatic idioms used in modern object-oriented languages we construct an analysis with the following characteristics: (1) it can express a rich set of structural, shape, and sharing properties which are not provided by a classic points-to analysis and that are useful for optimization and error detection applications (2) it uses efficient, weakly-updating, set-based transfer functions which enable the analysis to be more robust and scalable than a shape analysis and (3) it can be used as the basis for a scalable interprocedural analysis that produces precise results in practice.

The analysis has been implemented for .Net bytecode and using this implementation we evaluate both the runtime cost and the precision of the results on a number of well known benchmarks and real world programs. When compared to the results of a perfect oracle for the benchmarks we see that, despite the use of weak updates and absence of case splitting/materialization, the analysis produces information that is near the limit (80-90% accurate) of what is possible with our chosen abstract domain. Further, the analysis is capable of analyzing programs larger than any reported general purpose shape analysis and is faster than some points-to analyses on these programs, never taking longer than 70 seconds or using more than 150 MB of memory. Thus, this work presents a new type of memory analysis that advances the state of the art with respect to expressive power, precision, and scalability and represents a new area of study on the relationships between and combination of concepts from shape and points-to analyses.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages–Program Analysis

General Terms Languages, Performance

Keywords Structural Analysis, Program Understanding, Static Analysis

1. Introduction

Techniques for analyzing the memory structures created and operated on by a program have generally fallen into two families: Points-To (or alias) Analysis and Shape Analysis. These approaches lie at far ends of the spectrum of analysis cost and precision. In particular points-To analyses track very simple properties, usually little more than points-to set information, and the transfer functions which simulate the effects of various program statements use simple and efficient set operations. At the other end of the spectrum, shape analyses track a range of rich heap properties and generally utilize computationally complex transfer functions, involving materialization operations, case splitting, and strong updates. While individually each of these areas has seen intensive research, there has been little work in exploring the vast area between these two points in the cost-precision spectrum or in merging concepts from these analysis approaches. A major reason for this separation is the issue of weak vs. strong updates and the associated machinery of case splitting and materialization. In particular a critical question is: Are strong updates a critical component of a shape style analysis or is it possible to compute precise shape, sharing, etc. information with an analysis that uses simpler and more efficient transfer functions?

Recent empirical work on the structure and behavior of the heap in modern object-oriented programs has shed light on how heap structures are constructed [1, 47], the configuration of the pointers and objects in them [3], and their invariant structural properties [31, 36]. These results affirm several common assumptions about how object-oriented programs are designed and how the heap structures in them behave. In particular [1, 3, 47] demonstrate that object-oriented programs exhibit extensive *mostly-functional* behaviors: making extensive use of *final* (or *quiescing*) fields, *stationary* fields, copy construction, and when fields are updated the new target is frequently a newer (often freshly allocated) object. The results in [31, 36] provide insight into what heuristics can be used to effectively group sections of the heap based on how they are used in the program, what types of invariants hold for these structures, and how universal these invariants are in practice. The results show that, in practice, object-oriented programs tend to organize objects on the heap into well defined groups based on their roles in the program and that the relationships between these groups tend to be relatively stable, particularly with respect to structural organization, reachability, and sharing properties.

The information provided by these empirical studies provide the central design principles that guide the construction of the heap analysis in this paper. The prevalence of mostly functional behavior implies that the domain and transfer functions can, generally, handle writes as weak updates without large precision losses. However, to precisely handle object initialization and the frequent case of updating a field to point to a newly (or very recently) allocated object, the domain should model such objects with extra care. Previous experience with *Context-Sensitive* dataflow analysis has shown that the number of contexts that are created is a critical factor in performance [29, 32, 44]. To improve the speed at which the analysis converges to a fixpoint the abstract heap domain and normal form representation should have natural (and compact) encodings for commonly occurring and relatively invariant heap properties. Finally, given that object-oriented programs are not completely functional, there will be cases where the simplified abstract transfer functions introduce imprecision. Thus, the abstract heap domain should provide strong disjointness and isolation properties between the vari-

ous parts of the heap. These properties serve to both minimize the impact of any imprecision that is introduced and to prevent cascading of this imprecision. As an additional benefit a notion of disjointness allows the use of frame rules [20, 41].

1.1 Contributions

The main practical contribution of this paper is the construction of a novel static heap analysis, *Structural Analysis*, that combines a rich shape analysis style abstract heap model with efficiently computable points-to analysis style abstract transfer functions. The resulting hybrid memory analysis is able to precisely identify various structures in memory and to track sharing, shape, and reachability relations on them (in practice $80-90\%$ accurate when compared to our analysis results oracle). In addition to producing precise results the analysis is capable of analyzing real world programs, which are beyond the capabilities of existing shape analyses, and requires less time than even a points-to analysis on some of these programs (always less than 70 seconds and 150 MB of memory).

The main theoretical contribution of this paper is the initial exploration of a new area of memory analysis that lies between and is based on the combination of concepts from work on shape analysis and points-to analysis. This paper identifies and examines a number of general principles that are derived from empirical studies of the heap in real programs and that are central to the construction of these style of hybrid analysis approaches. The information from these studies combined with the empirical results from the analysis constructed in this paper show that strong updates (and associated machinery) are *not critical* and that in practice weak updates are sufficient for computing large amounts of useful shape and sharing information in real world object-oriented programs. Thus, this work opens new possibilities for exploring the relationships between shape and points-to analyses and represents a new approach to building scalable and precise memory analysis tools.

Technical Contributions. This paper contains a number of technical contributions involving the design of the domain, normal form, and transfer functions. The abstract domain (Sec. 2) is based on the classic storage shape graph approach and is able to express a rich set of commonly occurring and generally useful properties including, structure identification, reachability, sharing, and shape. Additionally, due to the implicit disjointness information in the graph structure, the resulting abstract heap model possess strong separability and isolation characteristics that limit the propagation of imprecision. The normal form (Sec. 3) is defined in terms of an efficient congruence closure computation, $O((N+E)*log(N))$ where N is the number of nodes in the shape graph and E is the number of edges. This congruence relation is based on the structures identified in the empirical studies and enables the analysis to rapidly converge to a fixpoint without either a large loss of information on the domain properties of interest or the generation of large amounts of irrelevant detail. The points-to style transfer functions (Sec. 5) are based on set-operations and weak updates. In practice they precisely model the heap properties of interest and are efficiently computable, $O(N+E)$ worst case but in practice are near constant time. In order to quantify the performance and precision of this analysis we present an extensive experimental evaluation (Sec. 6) of several well known benchmarks including programs from SPEC JVM98 and DaCapo. This evaluation includes both the timing and memory use characteristics of the analysis as well as a rigorous evaluation of the precision of the results. The evaluation shows that the analysis results are both precise and, despite the extensive use of context sensitivity via call-graph cloning and type information, the interprocedural analysis is scalable.

2. Abstract Heap Domain

We begin by formalizing concrete program heaps and the relevant properties that will be captured by the abstraction. Later, we define the abstract heap and formally relate the abstraction to the concrete heaps using a *concretization* (γ) function from the framework of abstract interpretation [6, 40]. These definitions are designed to support the expression of a range of generally useful properties (e.g., shape, sharing, reachability) that are common in shape analysis [5, 10, 35] and that are useful for a wide range of client optimization and error detection applications.

2.1 Concrete Heaps

For the purposes of this paper, we model the state of a program in a standard way where there is 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 *concrete heap*. Given a program that defines a set of concrete types, Type, and a set of fields (and array indices), Labels, defined in the types, we construct a concrete heap as a tuple $(\mathsf{Env},\sigma,\mathsf{Ob})$ where:

$$\mathsf{Env} : \mathsf{Vars} \rightharpoonup \mathsf{Addresses}$$
$$\sigma : \mathsf{Addresses} \rightarrow \mathsf{Ob} \cup \{\mathsf{null}\}$$
$$\forall o \in \mathsf{Ob}\,.\, o \text{ is a tuple } (\tau, \mathsf{Labels} \rightharpoonup \mathsf{Addresses})$$
$$\text{where } \tau \in \mathsf{Type}$$

Each object o in the set Ob is a tuple consisting of the type of the object and a map from field labels to concrete 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 Type 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. It is also useful to be able to refer to a *non-null pointer* as a specific structure in a number of definitions. Therefore we define a *non-null pointer* p associated with an object o and a label as l in a specific concrete 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 $\mathsf{Fld}(\mathit{type})$ to get the set of all fields that are defined for a given type (or array indices for an array type).

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

$$P(C_1,C_2) = \{(o_s,l,\sigma(o_s.l)) \mid \exists o_s \in C_1, l \in \mathsf{Fld}(\mathsf{Ty}(o_s))\,.\,\sigma(o_s.l) \in C_2\}$$

2.2 Concrete Heap Properties

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

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

Injectivity. Given two regions C_1 and C_2, we say that the non-null pointers with the label l from C_1 to C_2 are *injective*, written $\mathsf{inj}(C_1,C_2,l)$, if for all pairs of non-null pointers (o_s,l,o_t) and (o'_s,l,o'_t) drawn from $P(C_1,C_2)$, $o_s \neq o'_s \Rightarrow o_t \neq o'_t$. As a special case when we have an array object, we say the non-null pointer set $P(C_1,C_2)$ is *array injective*, written, $\mathsf{inj}_{[]}(C_1,C_2)$, if for all pairs of non-null pointers (o_s,i,o_t) and (o_s,j,o'_t) drawn from $P(C_1,C_2)$ and i, j valid array indices, $i \neq j \Rightarrow o_t \neq o'_t$.

These definitions capture the general case of an injective relation being defined from a set of objects and fields to 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.

Shape. We characterize the shape of regions of memory using standard graph theoretic notions of trees and directed-acyclic graphs (dags) treating the objects as vertices in a graph and the non-null pointers as defining the (labeled) edge set. We note that 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 region C then:

- The predicate $\mathsf{any}(C)$ is true for any graph. We use it as the most general shape that doesn't satisfy a more restrictive property.
- The predicate $\mathsf{dag}(C)$ holds, if the subgraph $(C, P(C,C))$ is acyclic.
- The predicate $\mathsf{tree}(C)$ holds, if $\mathsf{dag}(C)$ holds and the subgraph $(C, P(C,C))$ contains no pointers that create cross edges.
- The predicate $\mathsf{none}(C)$ holds, if the edge set in the subgraph is empty, $P(C,C) = \emptyset$.

As is apparent from this definition, $\mathsf{none}(C)$ implies $\mathsf{tree}(C)$, $\mathsf{tree}(C)$ implies $\mathsf{dag}(C)$, and $\mathsf{dag}(C)$ implies $\mathsf{any}(C)$.

2.3 Abstract Heap

An abstract heap is an instance of a storage shape graph [5]. More precisely, an abstract heap graph is a tuple: $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})$ where:

$$\widehat{\mathsf{Env}} : \mathsf{Vars} \rightharpoonup \widehat{\mathsf{Addresses}}$$
$$\widehat{\sigma} : \widehat{\mathsf{Addresses}} \to \widehat{\mathsf{Inj}} \times 2^{\widehat{\mathsf{Ob}}}$$
$$\text{where } \widehat{\mathsf{Inj}} = \{\mathit{true}, \mathit{false}\}$$
$$\forall n \in \widehat{\mathsf{Ob}}.\, n \text{ is a tuple } (\widehat{\tau}, \widehat{\zeta}, \widehat{\mathsf{Label}} \rightharpoonup \widehat{\mathsf{Addresses}})$$
$$\text{where } \widehat{\tau} \in 2^{\mathsf{Type}} \wedge \widehat{\zeta} \in \{\mathsf{none}, \mathsf{tree}, \mathsf{dag}, \mathsf{any}\}.$$

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 $\widehat{\mathsf{Ob}}$ is a tuple consisting of a set of types, a shape tag, and a map from abstract labels to abstract addresses. The abstract labels ($\widehat{\mathsf{Label}}$) are the field labels and the special label []. The label concretization is defined by:

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

The special label [] abstracts the indices of all array elements (i.e., array smashing). Otherwise an abstract label $\widehat{l}$ represents the given object field with the given name.

As with the objects we introduce the notation $\widehat{\mathsf{Ty}}(n)$ to refer to the type set associated with a node. The notation $\widehat{\mathsf{Sh}}(n)$ is used to refer to the shape 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 $\widehat{\mathsf{Inj}}(\widehat{\sigma}(x))$ to refer to the *injectivity* and $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(x))$ to refer to the set of possible abstract node targets associated with the abstract address. We define the helper function $\widehat{\mathsf{Fld}}(\{\mathit{type}_1, \ldots, \mathit{type}_k\})$ 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).

2.4 Abstraction Relation

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

$$\begin{aligned}(\mathsf{Env}, \sigma, \mathsf{Ob}) \in \gamma((\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})) \\ \Leftrightarrow \exists \mu\,.\, \mathsf{Embed}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \\ \wedge\, \mathsf{Typing}(\mu, \mathsf{Ob}, \widehat{\mathsf{Ob}}) \\ \wedge\, \mathsf{Injective}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \\ \wedge\, \mathsf{Shape}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})\end{aligned}$$

A concrete heap is an instance of an abstract heap, if there exists an embedding function $\mu : \mathsf{Ob} \to \widehat{\mathsf{Ob}}$ satisfying the graph embedding, typing, injectivity, and shape relations between the structures. The auxiliary predicates are defined as follows.

$$\begin{aligned}&\mathsf{Embed}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) = \\ &\quad \forall v \in \mathsf{Vars}\,.\, \mu(\sigma(\mathsf{Env}(v))) \in \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v))) \\ &\quad \wedge\, \forall o_s \in \mathsf{Ob} \text{ and non-null pointers } p = (o_s, l, o_t) \\ &\qquad \exists \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(\mu(o_s)))\,.\, \mu(o_t) \in \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\mu(o_s).\widehat{l})) \wedge l \in \gamma_{\mathsf{L}}(\widehat{l})\end{aligned}$$

The embed predicate makes sure that all of the objects and pointers of the concrete heap are present in the abstract heap graph, connecting corresponding abstract nodes, and that the store and labels in the abstract graph respect the concrete store and labels. The embedding must also preserve any variable mappings.

$$\mathsf{Typing}(\mu, \mathsf{Ob}, \widehat{\mathsf{Ob}}) = \forall n \in \widehat{\mathsf{Ob}}, o \in \mu^{-1}(n)\,.\, \mathsf{Ty}(o) \in \widehat{\mathsf{Ty}}(n)$$

The typing relation guarantees that the type $\mathsf{Ty}(o)$ for every concrete object o is in the set of types of the abstract node $\widehat{\mathsf{Ty}}(n)$ associated with o.

$$\begin{aligned}&\mathsf{Injective}(\mu, \mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) = \\ &\quad \forall n_s, n_t \in \widehat{\mathsf{Ob}}, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n_s))\,.\, \widehat{\mathsf{Inj}}(\widehat{\sigma}(n_s.\widehat{l})) \Rightarrow \\ &\qquad \text{if } \widehat{l} = [] \text{ then } \mathsf{inj}_{[]}(\mu^{-1}(n_s), \mu^{-1}(n_t)) \\ &\qquad \wedge\, \forall l \in \gamma_{\mathsf{L}}(\widehat{l})\,.\, \mathsf{inj}(\mu^{-1}(n_s), \mu^{-1}(n_t), l)\end{aligned}$$

The injectivity relation guarantees that every pointer set marked as injective corresponds to injective (and array injective as needed) pointers between the concrete source and target regions of the heap. We note that this definition is restricted to the subset of labels that are type consistent with the declared types and field sets.

$$\begin{aligned}&\mathsf{Shape}(\mathsf{Env}, \sigma, \mathsf{Ob}, \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) = \\ &\quad \forall n \in \widehat{\mathsf{Ob}}\,.\, \widehat{\mathsf{Sh}}(n) = \mathsf{dag} \Rightarrow \mathsf{dag}(\mu^{-1}(n)) \\ &\qquad \wedge\, \widehat{\mathsf{Sh}}(n) = \mathsf{tree} \Rightarrow \mathsf{tree}(\mu^{-1}(n)) \\ &\qquad \wedge\, \widehat{\mathsf{Sh}}(n) = \mathsf{none} \Rightarrow \mathsf{none}(\mu^{-1}(n))\end{aligned}$$

The shape relation guarantees that for every node n, the concrete subgraph $\mu^{-1}(n)$ abstracted by node n satisfies the corresponding concrete shape predicates.

2.5 Example Heap

Fig. 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.

Fig. 1(b) shows the corresponding normal form (see Sec. 3) abstract heap for this concrete heap. To ease discussion we label each node in a graph with a unique node id ($id). The abstraction

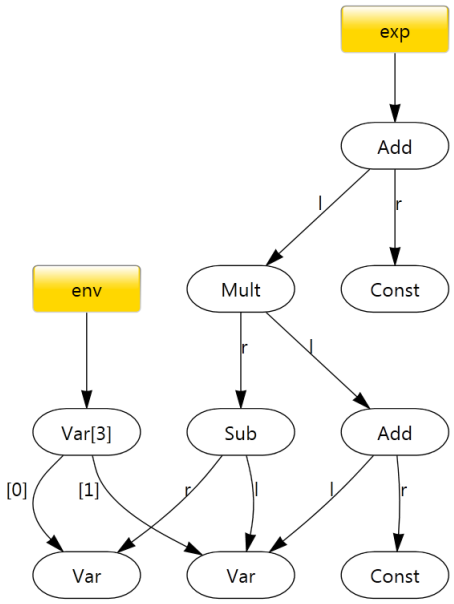

Figure 1(a). A Concrete Heap.

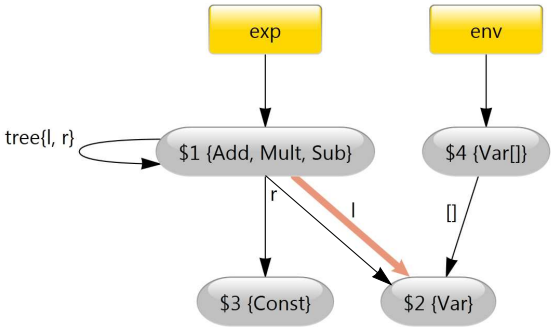

Figure 1(b). Corresponding Normal Form Abstract Heap.

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 nodes are absent but other more general properties are still present. For example, the fact that there is no sharing or cycles among the interior expression nodes is apparent in the abstract graph by looking at the self-edge representing the pointers between objects in the interior of the expression tree. The label `tree{l,r}` on the self-edge expresses that pointers stored in the `l` and `r` fields of the objects in represented by node 1 form a tree structure (i.e., no sharing and no cycles).

The abstract graph maintains another useful property of the expression tree, namely that no `Const` object is referenced from multiple expression objects. On the other hand, several expression objects might point to the same `Var` object. The abstract graph shows 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. Additionally, the abstract heap, via a combination of reachability, shape, and sharing information, shows there is no aliasing on any distinct pair of paths starting from `exp` and ending with a dereference of the `r` field. This can be deduced from the fact that node 1 is a tree layout, so there is no aliasing internally on either the `l` or `r` fields, and that both outgoing edges `r` edges are *injective* (narrow and unshaded). Since we know all paths through the tree do not alias (lead to different objects) this implies the final dereferences of the `r` fields, which can only contain injective pointers to `Const` or `Var` objects, do not alias either.

This example illustrates the expressiveness of the abstract domain constructed in this section which is capable of computing per region and per field information on reachability (via the graph structure), shape (e.g., the tree region), and sharing (e.g., no aliasing in the `env` array). Thus it is capable of expressing properties that are needed for the introduction of thread-level parallelism [10], object co-location [13], pool allocation [26], incremental GC [19], static deallocation [14], etc. As many of these approaches were designed to work with the limited information provided by a (often context insensitive) points-to analysis, the precise points-to information in the model (due to the full call graph cloning) combined with the shape and injectivity information provides improvements to both the baseline effectiveness of the techniques and opportunities for using the additional information for further refinements.

3. Normal Form

Given the definitions for the abstract heap it is clear that the domain is infinite. This allows substantial flexibility when defining the transfer functions and more precise results when analyzing straight line blocks of code. However, it is problematic when defining the merge/equality operations and can result in the final analysis having an unacceptably large computational cost. To prevent this we define an efficiently computable normal form, $O((N+E)*log(N))$ where N is the number of nodes in the abstract heap graph and E is the number of edges. The normal form ensures 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.

The normal leverages the idea that locally (within a basic block or method call) invariants can be broken and subtle details are critical to program behavior but before/after these local components invariants should be restored. The basis for the normal form, and the selection of what are important properties to preserve, comes from studies of the runtime heap structures produced in object-oriented programs [31, 36]. Thus we know that, in general, these definitions are well suited to capturing the fundamental structural properties of the heap that are of interest while simplifying the structure of abstract heaps and discarding superfluous details.

DEFINITION 1 (Normal Form). *We say that the abstract heap is in normal form iff:*

- *All nodes are reachable from a variable or static field.*
- *All recursive structures are summarized (Def. 2).*
- *All equivalent successors are summarized (Def. 4).*
- *All variable/global equivalent targets are summarized (Def. 5).*

That is there are no unreachable nodes and structurally the abstract heap represents the congruence closure of the recursive structure, equivalent successor, and equivalent target relations.

While the normal form definition is fundamentally driven by heuristics derived from empirical studies of the heap structures in real programs (and thus one could imagine a number of variants) there are three key properties that it possesses: (1) the resulting abstract heap graph has 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 allows us to define efficient merge and compare operations (Sec. 4).

3.1 Equivalence Partitions

As each of the properties (*recursive structures*, *ambiguous successors*, and *ambiguous targets*) are defined in terms of, congruence between abstract nodes the transformation of an abstract heap into the corresponding normal form is fundamentally the computation of a congruence closure over the nodes in the abstract heap followed by merging the resulting equivalence sets. Thus, we build a map from the abstract nodes to equivalence sets (partitions) using a Tarjan union-find structure. Formally $\Pi : \widehat{\mathsf{Ob}} \to \{\pi_1, \ldots, \pi_k\}$ where $\pi_i \in 2^{\widehat{\mathsf{Ob}}}$ and $\{\pi_1, \ldots, \pi_k\}$ are a *partition* of $\widehat{\mathsf{Ob}}$. The union-find structure can also be used to maintain the set of all the types associated with the nodes in a partition ($\bigcup_{n\in\pi} \widehat{\mathsf{Ty}}(n)$). Initially the partition is set as a singleton (i.e., $\forall n \in \widehat{\mathsf{Ob}} \,.\, \Pi(n) = \{n\}$).

Recursive Structures. The first step in computing the normal form is to identify any nodes that may be parts of unbounded depth structures. This is accomplished by examining the type system for the program that is under analysis and identifying all the types that are part of the same recursive type definitions. This is a commonly used technique [2, 7, 30] and ensures that any heap graph produced has a finite depth. 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 2 (Recursive Structure). *Given two partitions π_1 and π_2 we define the* recursive structure *congruence relation as:*

$$\begin{aligned}\pi_1 \equiv_r^{\Pi} \pi_2 \Leftrightarrow \\ \exists \tau_1 \in \textstyle\bigcup_{n_1\in\pi_1} \widehat{\mathsf{Ty}}(n_1), \tau_2 \in \bigcup_{n_2\in\pi_2} \widehat{\mathsf{Ty}}(n_2) \,.\, \tau_1 \sim \tau_2 \\ \wedge \exists n \in \pi_1, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap \pi_2 \neq \emptyset\end{aligned}$$

Equivalent Successors and Targets. The other part of the normal form computation is to identify any partitions that have *equivalent successors* and variables that have *equivalent targets*.

The successor (predecessor) relation for the node partitions is the natural definition based on the underlying structure of the abstract heap graph:

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

Next we define the basic equivalence relation on the nodes that forms the basis of the congruence relation on the graph.

DEFINITION 3 (Partition Compatibility). *Given partitions π_1 and π_2 we define the relation Compatible(π_1, π_2) as:*

$$\text{Compatible}(\pi_1, \pi_2) \Leftrightarrow \bigcup_{n'\in\pi_1} \widehat{\mathsf{Ty}}(n') \cap \bigcup_{n'\in\pi_2} \widehat{\mathsf{Ty}}(n') \neq \emptyset$$

Given the successor and compatibility relations we can define the congruence relations for nodes that are either both successors of the same partition on that are both targets of the same local variable (or static field).

DEFINITION 4 (Equivalent Successors). *For a partition π and successors π_1, π_2 on labels $\widehat{l}_1$, $\widehat{l}_2$ respectively we define the* equivalent successors *relation as:*

$$\pi_1 \equiv_s^{\Pi} \pi_2 \Leftrightarrow \widehat{l}_1 = \widehat{l}_2 \wedge \text{Compatible}(\pi_1, \pi_2)$$

DEFINITION 5 (Equivalent on Targets). *Given a root r (a variable or a static field) two target partitions π_1, π_2 we define the* equivalent targets *relation as:*

$$\begin{aligned}\pi_1 \equiv_t^{\Pi} \pi_2 \Leftrightarrow \text{Compatible}(\pi_1, \pi_2) \wedge \\ (r \textit{ is a static field} \vee \pi_1, \pi_2 \textit{ only have local var predecessors})\end{aligned}$$

Using the *recursive structure* relation and the *equivalent successor (target)* relations we can efficiently compute the congruence closure over an abstract heap producing the corresponding normal form abstract heap (Def. 2). This computation can be done via a standard worklist algorithm [39] for grouping equivalent nodes where merging two partitions may create a new opportunity for merging. Whenever partitions are merged we add any other partitions that may be effected by the merge back onto the worklist. Due to the properties of congruence closure algorithms and the union-find data structure, we can know that this implementation can be done such that each partition can enter the work list at most $log(N)$ times, where N is the number of abstract nodes in the initial abstract heap, and if E is the number of abstract addresses in the heap then the complexity of computing the partitions is $O((N+E) * log(N))$.

3.2 Computing Summary Nodes

After partitioning the nodes in the graph with the congruence closure computation we need to merge all the nodes in each partition into a summary node. The resulting summary node should safely summarize the properties of the all the nodes in the partition. Similarly, we may need to update target and injectivity information for the summary nodes in the abstract store. Given a node partition (π) that we want to replace with a new summary node (n_s), we can use the following functions to compute the abstract properties for each summary node and the new abstract store $\widehat{\sigma}_s$:

$$\begin{aligned}\forall \pi \in &Img(\Pi) \\ &n_s = (\sqcup_{type}(\pi), \sqcup_{shape}(\pi), lmap) \\ &\quad lmap = \{[\widehat{l} \mapsto \widehat{a}_{\widehat{l}}] \mid \widehat{l} \in \widehat{\mathsf{Fld}}(\sqcup_{type}(\pi)), \widehat{a}_{\widehat{l}} \text{ a fresh address}\} \\ &\widehat{\sigma}_s = MergeStore(\widehat{\sigma}_s, \widehat{l}, \pi) \text{ for each } \widehat{l} \in \widehat{\mathsf{Fld}}(\sqcup_{type}(\pi))\end{aligned}$$

Once this merge is complete we can update the information on the abstract addresses associated with each variable in $\widehat{\mathsf{Env}}$ by replacing any nodes in the target sets with the appropriate newly created summary nodes.

Type. The abstract type information is simply the union of corresponding type sets from the nodes in the partition.

$$\sqcup_{type}(\pi) = \bigcup_{n\in\pi} \widehat{\mathsf{Ty}}(n)$$

Shape. The *Shape* information is more difficult to merge as it depends both on the shapes of the individual nodes that are being grouped and also on the connectivity properties between them. We first perform a traversal of the subgraph of the partition and the (non-self) abstract targets between them. Then based on the discovery of back, cross, or tree references (in a graph theoretic sense) and if any of these abstract storage location are *not injective* we compute the shape as $\sqcup_{shape}(\pi) = struct(\pi) \sqcup \bigsqcup_{n\in\pi} \widehat{\mathsf{Sh}}(n)$ where

$struct(\pi)$ is defined:

$$
\begin{aligned}
&struct(\pi) = \\
&\quad \textsf{any if } \exists n \in \pi, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, n.\widehat{l} \textit{ creates a Back Edge in } \pi \setminus \{n\} \\
&\quad \textsf{dag if } \exists n \in \pi, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, n.\widehat{l} \textit{ creates a Cross Edge in } \pi \\
&\qquad\qquad \vee \neg \widehat{\mathsf{Inj}}(\widehat{\sigma}(n.\widehat{l})) \\
&\quad \textsf{tree if } \forall n \in \pi, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, n.\widehat{l} \textit{ creates a Tree Edge in } \pi \\
&\quad \textsf{none if } \textit{No Internal Edges Exist}
\end{aligned}
$$

Injectivity and Abstract Targets. Given a mapping from the partitions to the new summary nodes, $\Phi : Img(\Pi) \rightarrow \{n_{s_1}, \ldots, n_{s_k}\}$, 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:

$$
\begin{aligned}
&MergeStore(\widehat{\sigma}_s, \widehat{l}, \pi) = \widehat{\sigma}_s + [\widehat{a_{\widehat{l}}} \mapsto (inj, trgts)] \\
&\text{where} \\
&\quad trgts = \{\Phi(\Pi(n')) \mid n' \in \textstyle\bigcup_{n \in \pi} \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l}))\} \\
&\quad inj = \forall n \in \pi \,.\, \widehat{\mathsf{Inj}}(\widehat{\sigma}(n.\widehat{l})) \wedge \forall n' \in \pi \setminus \{n\} \,.\, inj_{\widehat{l}}(n, n') \\
&\quad inj_{\widehat{l}}(n_1, n_2) = \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n_1.\widehat{l})) \cap \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n_2.\widehat{l})) = \emptyset
\end{aligned}
$$

Injectivity is the logical conjunction of the injectivity of all the source label locations, and that the respective targets sets of the nodes that are merged do not overlap. 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.

From the definitions of the summary node computations and the update of the abstract store locations the preservation of the safety of the abstraction is straight forward to check via case enumeration. In particular each of the operations consists of a simple join on a set of values, as given by the partition, and some simple additional computation on the local structure of each partition. It is also clear that each partition is processed once in the normal form computation (and similarly the addresses in the abstract store are each only visited a constant number of times). Thus, the cost of computing the summaries can be done in linear time. Finally, as the congruence closure over given a graph is unique the resulting normal form graph, as defined here, is also unique.

3.3 Normal Form on Example Heap

We can see how this normal form works by using it to transform the concrete heap in Fig. 1(a) into its normal form abstract representation. This can be done by first 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 Fig. 2.

The normal form partition for the abstract heap in Fig. 2 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. 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*.

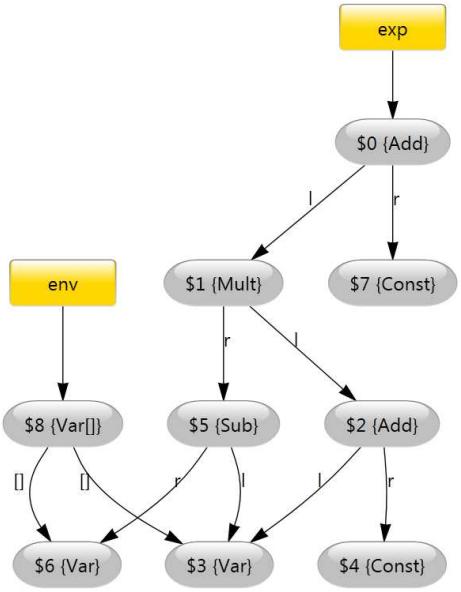

Figure 2. Isomorphic Abstract Heap.

Thus the final partitioning after the congruence closure is:

$$
\mu^{-1} \begin{cases} \pi_1 : \{n_0, n_1, n_2, n_5\} \\ \pi_2 : \{n_3, n_6\} \\ \pi_3 : \{n_4, n_7\} \\ \pi_4 : \{n_8\} \end{cases}
$$

Given this set of partitions the computation of the various properties is straight forward. The *Shape* for the partitions containing the `Var`, `Const` and `Var[]` nodes is 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 no internal cross or back edges the layout for this is `tree`.

In computing the new summary abstract store properties 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 to 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 to *true* (*injective*). 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 Fig. 1(b).

4. Domain Operations

Given the normal form in Sec. 3 we can define an efficiently computable abstract equality operation ($\widehat{=}$) and upper approximation ($\widetilde{\sqcup}$) operator on the *normal form* abstract heaps. Since the set of normal form abstract heaps is finite (for a given program) we do not need a widening operator. Both operations can be performed efficiently, $O(N+E)$ for equality and $O((N+E) * log(N))$ for the upper approximation.

Abstract Equality. To enable efficient comparison we only define equality on the normal forms of the abstract heap states. The abstract equality relation we construct has the property:

$$
\begin{aligned}
&(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1) \widehat{=} (\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2) \Rightarrow \\
&\qquad\qquad \gamma((\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1)) = \gamma((\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2))
\end{aligned}
$$

Since the set of normal form abstract graphs we use in the fixpoint computation is finite this is sufficient to guarantee termination and safety of the analysis.

Given two abstract heaps $(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1)$ and $(\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2)$ we first determine if they are structurally isomorphic (i.e., if there is an isomorphism on the graph structures that respects variable and field labels), then we check that all abstract node and store properties in $(\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2)$ have the same values in $(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1)$ under the isomorphism.

To efficiently compute the needed isomorphism we use a property of the abstract graphs established by the normal form definition (Def. 1). By this definition we know that each node is reachable from a root location (a local variable or a static field), thus if an isomorphism exists it can be found by matching from the roots. Further, we know that for each abstract address in the store if there is more than one element in the target set then each of these targets must have non-overlapping sets of *types* (from the definition of *Compatible*, Def. 3). Thus, to compute an isomorphism between two graphs we can simply start pairing the local and static roots and then process the abstract structure in a breadth first manner, pairing up nodes based on abstract labels and type sets of the targets, leading to new pairings. This either results in an isomorphism between the two structures, ϕ, or it reaches a point where no match is possible and fails without backtracking.

If we find an isomorphism ϕ then we check the equivalence of the abstract nodes and store as follows:

$$(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1) =_\phi (\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2) \Leftrightarrow$$
$$\forall n \in \widehat{\mathsf{Ob}}_1 \,.\, \widehat{\mathsf{Ty}}(n) = \widehat{\mathsf{Ty}}(\phi(n)) \wedge \widehat{\mathsf{Sh}}(n) = \widehat{\mathsf{Sh}}(\phi(n))$$
$$\wedge \forall \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, \widehat{\mathsf{Inj}}(\widehat{\sigma}_1(n)) = \widehat{\mathsf{Inj}}(\widehat{\sigma}_2(\phi(n)))$$

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

Given two abstract heaps, $(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1)$ and $(\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2)$ we can define the abstract heap that is the result of their merge as follows. First we produce the union of the two abstract heaps by taking the union of the abstract node sets and the abstract stores in the usual way. From this union store we can compute the corresponding normal form as described in Sec. 3.

$$(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1) \widehat{\sqcup} (\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2) =$$
$$\mathit{Normalize}(\widehat{\mathsf{Env}}_m, \widehat{\sigma}_m, \widehat{\mathsf{Ob}}_1 \uplus \widehat{\mathsf{Ob}}_2) \text{ where}$$
$$\widehat{\mathsf{Env}}_m = \{[v \mapsto \widehat{a_v}] \mid v \in \mathit{Dom}(\widehat{\mathsf{Env}}_1 \cup \widehat{\mathsf{Env}}_2), \widehat{a_v} \text{ a fresh address}\}$$
$$\widehat{\sigma}_m = \widehat{\sigma}_1 \uplus \widehat{\sigma}_2 \uplus \{[\widehat{a_v} \mapsto (\mathit{true}, \mathit{trgts}_v)] \mid [v \mapsto \widehat{a_v}] \in \widehat{\mathsf{Env}}_m\}$$
$$\mathit{trgts}_v = \widehat{\mathsf{Trgts}}(\widehat{\sigma}_1(\widehat{\mathsf{Env}}_1(v))) \cup \widehat{\mathsf{Trgts}}(\widehat{\sigma}_2(\widehat{\mathsf{Env}}_2(v)))$$

5. Abstract Transfer Functions

Given the expressive *Shape Analysis Style* domain defined in Sec. 2.3 the next step is to define a set of transfer functions that simulate the effects of various program statements on the abstract heaps. Our goal is to construct these definitions in a *Points-To Analysis Style*, using weak updates and simple set operations while still precisely modeling the effects of each statement on the heap state. In order to focus on the fundamental aspects of the analysis we present the results on a simple object-oriented language with the standard set of allocation, load, and store operations. However, in practice the approach can be extended in a natural way to handle a much richer language. Our implementation for .Net bytecode (Sec. 6) handles features such as struct types, references to the stack, limited forms of multi-threading, pointers to the interior of objects, and function pointers.

Table 1 shows the transition semantics for both the concrete heap model (left column) and abstract heap model (right column) for the statements that are the most interesting from the standpoint of memory analysis. In order to focus on the central ideas we ignore issues with null-pointer dereferences, array out-of-bounds errors, etc. In most cases the abstract transfer functions are the natural translations of the concrete semantic operations, and are very similar to the set of transfer functions seen in a standard points-to analysis [38, 44, 48]. However, there are a number of important differences from a standard formulation of points-to analysis transfer functions, of particular interest are the `allocation`, `store`, and `invocation` operations.

Allocate. The definition of the allocation operation plays a key role in the functioning of the analysis. As opposed to the usual points-to definition which will reuse nodes in the abstract heap based on some context token, ranging from simple allocation type or line number through sophisticated object-sensitive constructions, our definition of the allocation operation always creates a fresh node. In this sense the definition closely resembles the constructions used in shape style analyses.

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. Any finite naming scheme creates situations where there will be spurious reuse of a node, which will cause the loss of injectivity and/or shape information (e.g., in the store operation or the normal form summary computation). 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 Sec. 3 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 graphs that are in normal form is finite.

Load. The load operation is mostly a simple 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 fields and objects. However, since a variable location always contains a single pointer we can strongly update the target set and always set the associated pointers as being *injective*.

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 target nodes. If it is then we may be modifying the shape of the data structure represented by the node we are updating. While, it is possible to perform additional checks to be more precise in how the store affects the shape information we have opted to simply set the shape to the top value (`any`) in the case that a self store occurs. If there is no self store then the shape is unchanged.

The update to the abstract store involves taking the union of the old target set and the new target set (we weakly update 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

$v = \texttt{alloc type}\colon\ (\mathsf{Env}, \sigma, \mathsf{Ob}) \rightsquigarrow (\mathsf{Env}, \sigma', \mathsf{Ob}')$ where
$o = (\texttt{type}, \{l \to a_l \mid l \in \mathsf{Fld}(\texttt{type}), a_l \text{ fresh address}\})$
$\sigma' = \sigma + [\mathsf{Env}(v) \mapsto o]$
$\quad + \{[o.l \mapsto \mathsf{null}] \mid l \in \mathsf{Fld}(\texttt{type})\}$
$\mathsf{Ob}' = \mathsf{Ob} \uplus \{o\}$

$v = \widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Ob}}')$ where
$n = (\texttt{type}, \mathsf{none}, \{\widehat{l} \to \widehat{a_l} \mid \widehat{l} \in \widehat{\mathsf{Fld}}(\{\texttt{type}\}), \widehat{a_l} \text{ fresh address}\})$
$\widehat{\sigma}' = \widehat{\sigma} + [\widehat{\mathsf{Env}}(v) \mapsto (\mathit{true}, \{n\})]$
$\quad + \{[n.\widehat{l} \mapsto (\mathit{true}, \emptyset)] \mid \widehat{l} \in \widehat{\mathsf{Fld}}(\{\texttt{type}\})\}$
$\widehat{\mathsf{Ob}}' = \widehat{\mathsf{Ob}} \uplus \{n\}$

$v = v'\colon\ (\mathsf{Env}, \sigma, \mathsf{Ob}) \rightsquigarrow (\mathsf{Env}, \sigma', \mathsf{Ob})$ where
$\sigma' = \sigma + [\mathsf{Env}(v) \mapsto \sigma(\mathsf{Env}(v'))]$

$v = v'\colon\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Ob}})$ where
$\widehat{\sigma}' = \widehat{\sigma} + [\widehat{\mathsf{Env}}(v) \mapsto \widehat{\sigma}(\widehat{\mathsf{Env}}(v'))]$

$v = v'.l\colon\ (\mathsf{Env}, \sigma, \mathsf{Ob}) \rightsquigarrow (\mathsf{Env}, \sigma', \mathsf{Ob})$ where
$o = \sigma(\mathsf{Env}(v'))$
$\sigma' = \sigma + [\mathsf{Env}(v) \mapsto \sigma(o.l)]$

$v = v'.\widehat{l}\colon\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Ob}})$ where
$v'_{trgts} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v')))$
$\widehat{\sigma}' = \widehat{\sigma} + [\widehat{\mathsf{Env}}(v) \mapsto (\mathit{true}, \bigcup_{n \in v'_{trgts}} \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})))]$

$v.l = v'\colon\ (\mathsf{Env}, \sigma, \mathsf{Ob}) \rightsquigarrow (\mathsf{Env}, \sigma', \mathsf{Ob})$ where
$o = \sigma(\mathsf{Env}(v))$
$\sigma' = \sigma + [o.l \mapsto \sigma(\mathsf{Env}(v'))]$

$v.\widehat{l} = v'\colon\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Ob}})$ where
$v_{trgts} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v)))$
$v'_{trgts} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v')))$
$\forall n \in v_{trgts}$. if $n \in v'_{trgts}$ then $\widehat{\mathsf{Sh}}(n) \leftarrow \mathsf{any}$
$\widehat{\sigma}' = \widehat{\sigma} + [n.\widehat{l} \mapsto (\mathit{inj}, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cup v'_{trgts})]$
where $\mathit{inj} = \widehat{\mathsf{Inj}}(\widehat{\sigma}(n.\widehat{l})) \wedge \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap v'_{trgts} = \emptyset$

$v = m(\overrightarrow{v}')\colon\ (\mathsf{Env}, \sigma, \mathsf{Ob}) \rightsquigarrow (\mathsf{Env}, \sigma', \mathsf{Ob}')$ where
$\mathsf{Env}_m = \{[\mathit{param}_i \mapsto a_i] \mid \mathit{param}_i \in m, a_i \text{ a fresh address}\}$
$\sigma_m = \sigma + \{\mathsf{Env}_m(\mathit{param}_i) \mapsto \sigma(\mathsf{Env}(v'_i)) \mid \mathit{param}_i \in m\}$
$(\mathsf{Env}_{ret}, \sigma_{ret}, \mathsf{Ob}_{ret}) = \mathsf{Apply}(m, \mathsf{Env}_m, \sigma_m, \mathsf{Ob}_m)$
$\sigma' = \sigma_{ret} + [\mathsf{Env}(v) \mapsto \sigma_{ret}(\mathsf{Env}_{ret}(v_{ret}))]$
$\mathsf{Ob}' = \mathsf{Ob}_{ret}$

$v = m(\overrightarrow{v}')\colon\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \rightsquigarrow (\widehat{\mathsf{Env}}, \widehat{\sigma}', \widehat{\mathsf{Ob}}')$ where
$\widehat{\mathsf{Env}}_m = \{[\mathit{param}_i \mapsto \widehat{a}_i] \mid \mathit{param}_i \in m, \widehat{a}_i \text{ a fresh address}\}$
$\widehat{\sigma}_m = \widehat{\sigma} + \{\widehat{\mathsf{Env}}_m(\mathit{param}_i) \mapsto \widehat{\sigma}(\widehat{\mathsf{Env}}(v'_i)) \mid \mathit{param}_i \in m\}$
$(\widehat{\mathsf{Env}}_{ret}, \widehat{\sigma}_{ret}, \widehat{\mathsf{Ob}}_{ret}) = \widehat{\mathsf{Apply}}(m, \widehat{\mathsf{Env}}_m, \widehat{\sigma}_m, \widehat{\mathsf{Ob}}_m)$
$\widehat{\sigma}' = \widehat{\sigma}_{ret} + [\widehat{\mathsf{Env}}(v) \mapsto \widehat{\sigma}_{ret}(\widehat{\mathsf{Env}}_{ret}(v_{ret}))]$
$\widehat{\mathsf{Ob}}' = \widehat{\mathsf{Ob}}_{ret}$

$\texttt{return } v\colon\ (\mathsf{Env}, \sigma, \mathsf{Ob}) \rightsquigarrow (\mathsf{Env}', \sigma', \mathsf{Ob})$ where
$\mathsf{Env}' = \mathsf{Env} + [v_{ret} \mapsto a_{ret}], a_{ret} \text{ a fresh address}$
$\sigma' = \sigma + [\mathsf{Env}(v_{ret}) \mapsto \sigma(\mathsf{Env}(v))]$

$\texttt{return } v\colon\ (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}) \rightsquigarrow (\widehat{\mathsf{Env}}', \widehat{\sigma}', \widehat{\mathsf{Ob}})$ where
$\widehat{\mathsf{Env}}' = \widehat{\mathsf{Env}} + [v_{ret} \mapsto \widehat{a_{ret}}], \widehat{a_{ret}} \text{ a fresh address}$
$\widehat{\sigma}' = \widehat{\sigma} + [\mathsf{Env}(v_{ret}) \mapsto \widehat{\sigma}(\widehat{\mathsf{Env}}(v))]$

Table 1: Concrete Semantics (left) and Abstract Semantics (right)

conservatively 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*).

Method Call. For simplicity we assume that each method call can be statically resolved to a single target but in practice the analysis handles dynamic dispatch in the usual way of resolving the possible types of the receiver object, performing the analysis of each possible target, and then combining the results. Otherwise for the method call operation we perform the usual steps of constructing a fresh environment for the callee method body, calling a helper function ($\widehat{\texttt{Apply}}$) to perform the analysis of the callee, and integrating the results back into the local method scope. The structure and key aspects of the interprocedural analysis its operation are outlined here and we refer to [34] for more detail. The interprocedural analysis is fully context sensitive on calls to acyclic portions of the call graph, performing full call graph cloning on each method call for each new call state. On calls involving cyclic components of the call graph the analysis performs partial call graph cloning based on the *Compatibility*, Def. 3, of the arguments of the call. In practice this is done via a memotable of analysis input states (abstract heaps) and results which are re-analyzed as needed with new input states as in [32, 48].

DEFINITION 6 (Memo Table Representation). *For each method m in the program we maintain a list of memoized analysis states* $[\lambda_1, \ldots \lambda_k]$ *where each* $\lambda_i = ((\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})^{\text{in}}_i, (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})^{\text{out}}_i)$.

When a call to a method *m* is encountered with the input described by the abstract heap, $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})$, we look at the memo table entries, $[\lambda_1, \ldots \lambda_k]$, that we have previously encountered when analyzing the method body. If we find an entry $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})^{in}_i$ that matches with $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})\downarrow_m$, which is the abstract heap at the call site projected into the scope of *m*, we return the memoized result state $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})^{out}_i$ [32, 41, 48]. If not then we create a new entry in the table for $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})\downarrow_m$ and begin analysis on *m* with the new input. We refer to [32, 34] for a discussion of the matching and project/extend techniques used.

One interesting issue is what to do in the case of a recursive call when we may have a matching input but the memoized output value has never been computed. A common approach is to simply return the bottom domain value ($\bot$) for this case. The bottom value — for us the empty heap — is always a safe under approximation of the results but using it generally leads to a large number of fixpoint computation iterations. However, we know that the input abstract heap is also an under approximation of whatever the resulting output abstract heap will be. This is a result of the fact that all of the transfer functions are weakly updating wrt. heap locations and we do not do case splitting, thus any domain property that holds on the caller reachable heap at the entry of the method will always hold on the caller reachable portion of the heap at the exit of the method. So for the initial match we can simply return a copy of the input abstract heap. This often substantially reducing the number of iterations required to reach a fixpoint.

Computation. All of the transfer functions we have defined can be computed in time linear in the size of the heap model that they operate on. But the local operations (allocate, assign, load, store) are even more efficient as they are implemented in terms of simple set/graph operations which only examine the nodes (and perhaps immediate neighbors) that are the targets of the variables that they operate on. Thus, these local operations are linear in the number of targets and neighbor nodes (abstract addresses) which is, in general, a small fraction of the total number of nodes (abstract addresses) in the abstract heap.

6. Implementation and Evaluation

We have implemented the analysis described in this paper for a large set of the .Net bytecode language including struct types, references to the stack, limited forms of multi-threading, pointers to the interior of objects, and function pointers. In practice we first translate from .Net bytecode to an intermediate representation similar to the IR used in the LLVM compiler [25]. The translation from .Net to our IR is a mostly a 1-1 mapping but the use of the internal IR allows us remove most .Net specific idioms from the core analysis and allows some pre-processing to simplify later analysis steps. Our benchmarks are C# implementations of programs from Jolden [22], the db and raytracer programs from SPEC JVM98 [45], the luindex and lusearch programs from the DaCapo suite [22], and the heap abstraction code from [34], runabs. The domain, operations, and data flow analysis algorithms are all implemented in C# and are publicly available.[1]

One important consideration from the viewpoint of an analysis tool that is intended to operate on userspace programs are the types provided by the base class or system libraries, e.g., the Base Class Library (BCL) for .Net or the `java.*` in Java. For user space applications the internal structure of say, `FileStream` or `StringBuilder` is not interesting, so we treat these as single opaque objects. However, some classes in these libraries have features that are relevant to userspace code even though the details of the internal representation are not of particular interest. Examples of these types would be `List<T>` or `Dictionary<K,V>`, which we treat as ideal algebraic data structures, tracking the contained elements but treating the internal implementations as opaque. Our .Net translation system identifies these builtin types and methods invoked on them, replacing the actual implementations with either simplified versions or with special semantic operations as in [8, 37]. This special handling of builtin operations is very useful in improving the performance and precision of the analysis, but comes at the cost of additional work to implement support for large libraries.

Our test machine is an Intel i7 class processor at 2.66 GHz with 2 GB of RAM available. We use the standard 32 bit .Net JIT and runtime framework provided by Windows 7. As the analysis never consumes more than 150 MB of memory or takes more than 70 seconds we utilize the default parameters for the JIT and runtime.

6.1 Analysis Performance

Table 2 examines the cost of running the analysis in this paper. For each benchmark we list the number of bytecode instructions, the number of classes, and the number of methods that each program contains after being translated into the internal IR. These numbers exclude much of the code that would normally be part of the runtime system libraries. This is due to the fact that during the translation from .Net bytecode to the internal IR code which is never referenced is excluded. Additionally for the builtin types/methods that are used the implementations are often replaced by simplified versions or specialized domain operations.

The last two columns of Fig. 2 show the aggregate performance of the analysis on the benchmark set. The timing measurements exclude the time required to startup and read/transform the source program into the internal IR. These performance results show that the analysis described in this work is quite efficient and capable of analyzing complex programs. Despite the fact that the analysis is highly-context sensitive and has an expressive shape style domain the overall time and memory needed to analyze the programs is

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

Benchmark Statistics				Analysis Cost	
Name	Insts	Types	Methods	Time	Mem
power	3,298	43	320	0.09s	11MB
health	2,062	44	329	0.14s	12MB
bh	3,723	45	351	0.42s	14MB
db	2,873	42	315	0.21s	12MB
raytracer	9,808	65	476	6.72s	32MB
luindex	26,852	246	1747	12.1s	53MB
lusearch	33,632	272	1919	64.3s	130MB
runabs	27,875	253	1894	10.4s	60MB

Table 2: Benchmark statistics and aggregate performance of the analysis on them.

Benchmark	Max Iters	Avg. Entries	Max Nodes
power	3	1.02	29
health	4	1.09	23
bh	5	1.11	32
db	1	1.80	14
raytracer	4	2.24	61
luindex	6	2.46	102
lusearch	16	2.88	170
runabs	4	2.11	70

Table 3: *Max Iters* is the maximum number iterations taken to reach a fixpoint for any method/input abstract heap, *Avg. Entries* is the average number of memotable entries associated with each method, and *Max Nodes* is the max number of nodes in any abstract heap during the analysis.

quite small (even when compared to state of the art object-sensitive points-to analyses). However, the analysis runtime and cost only has a minimal correlation with the size of the program. Despite very similar numbers of instructions and methods bh takes over four times as long as power, and similarly for luindex and lusearch.

In the case of luindex (a fairly direct translation of the Java version from the DaCapo suite) the analysis requires only 12 seconds while recently reported results on context-sensitive points to analyses [44] reports analysis times ranging between 67 and 179 seconds depending on the amount and type of object-sensitivity used (and 37 seconds with an insensitive analysis). But more importantly, as memory use frequently is a major scalability wall, are the low memory requirements. Despite performing the equivalent of full call graph cloning for large parts of the analysis and being partially context sensitive on the remainder, the analysis presented in this paper uses less than 150 MB of memory when analyzing any of the benchmarks. We note that existing shape style approaches do not currently scale to programs of this size/complexity. While the work in [4, 8, 9, 49] has been used to analyze large programs, the C/C++ programs that have been analyzed do not use heap allocated data structures, recursion, and dynamic dispatch as extensively as the Java/C# programs here. Additionally, these techniques also place restrictions on the types of heap structures that the programs may create, either limitations on sharing [4, 49], or on the presence of recursive data structures [8, 9].

One major reason for the scalability of the analysis is that the normal form in Sec. 3 has been constructed to create equivalence classes that closely mirror the heap structures which appear in object-oriented programs. This ensures that the analysis quickly converges to a fixpoint and avoids generating large numbers of spurious and uninteresting contexts (entries in the memo tables). Table 3 shows information on the number of memo table entries produced for methods during the analysis. The first column in the table shows the maximum number of analysis iterations of a method body (with a given input abstract heap) required to reach a fixpoint state. As can be seen even this maximum value is relatively small (16 in the worst case). Additionally when looking at the average number of contexts created per method in the program we see that, even with the aggressive creation of memo table entries, the average is less than 3. Finally, the size of the abstract heaps is large enough to precisely resolve useful structure but not so large that it is computationally problematic.

The runtimes from Tab. 2 correlate well with the *MaxIters* and *AvgEntries* columns. Thus we see the performance impacts of reducing the number of memo table entries and number of fixpoint iterations. This highlights the value of the normal form which is able to quickly push the abstract heaps toward invariant states, thus producing a small set of input abstract states for each method which quickly stabilizes during the fixpoint computation. This shows that the normal form is, in combination with the efficient operations for the local transfer functions, critical to the low memory use and rapid completion of the analysis.

6.2 Quantitative Precision

The analysis in this paper tracks properties that have shown, in past work, to be both relevant and useful [9, 10, 12–14, 26, 44]. However, we want to examine the quantitative precision of the analysis in a way that is free from biases introduced by the selection of a particular client application. Thus, we examine the precision of the static analysis relative to a hypothetical perfect analysis which uses the same abstract domain. This notion of precision is a better basis for examining the impact of the possible imprecision of the abstract transfer functions and normal form on the analysis results than the use of a specific client application (which may hide precision losses that *happen* not to matter for the particular client).

We define precision relative to a hypothetical *perfect analysis* which uses the same abstract domain from Sec. 2 but that is able to perfectly predict the effects of every program operation. Since we cannot actually build such an analysis we approximate it by collecting and abstracting the results of concrete executions. By definition this collection of results from the concrete execution is an under approximation of the universal information we want to compute, and in the limit of execution of all possible inputs is identical. Formally, given a method and a set of concrete heaps $\{h_1, \ldots, h_k\}$ and a set of abstract heaps $\{\widehat{h}_1, \ldots, \widehat{h}_j\}$ we can compute differences between $\bigsqcup_{h \in \{h_1,\ldots,h_k\}} \alpha(h)$ and $\bigsqcup\{\widehat{h}_1, \ldots, \widehat{h}_j\}$. This gives an unbiased measure of how close our results are to the optimal solution, wrt. the abstract domain we are working with in a way that is independent of peculiarities of a client application or other analysis technique.

Table 4 shows the results of this comparison on our benchmarks. For the numbers in this table we compared the results from our *perfect analysis* with the results from the static analysis described in the paper. In this table we further refine the comparison to be property specific by reporting the average percentage, over all nodes (or abstract addresses) in all graphs for all methods in the program, precisely identified: regions, shapes, or injectivity values. The region percentage (the *Region* column) is number of nodes that can be exactly matched between the statically computed and ideal result structure. Using this matching we then compute the percentage of the *shape* and *injectivity* properties that are precisely identified by the static analysis (the *Shape* and *Injectivity* columns).

Overall the results show that the analysis is able to extract a large percentage of the properties that can be expressed via the selected abstract domain (in general with a rate of 80% to 90%). In general the normal form and points-to style abstract transfer functions result in only small losses in precision when analyzing the behavior of the program and the effects of various operations on the state of the heap. In inspecting the places where the analysis

Benchmark	Region	Shape	Injectivity
power	100%	100%	100%
health	72%	100%	65%
bh	100%	90%	87%
db	100%	100%	81%
raytracer	80%	85%	83%
luindex	95%	95%	82%
lusearch	93%	90%	84%
runabs	97%	98%	87%

Table 4: Average accuracy of analysis results when compared to *perfect analysis*. Reported as a percentage of each property correctly predicted by the static analysis.

does lose precision we often find small blocks of code operating in a nontrivial way on some set of objects. An example of this are the benchmarks power, which has extensive *mostly-functional* behavior and our analysis is able to analyze it perfectly. Conversely, the outlying benchmark health performs extensive transfer of ownership among a number of lists in the program. In this case our analysis loses a substantial amount of sharing information (identifying the true injectivity state precisely for only 65% of the abstract store locations). In all the cases we inspected, such as health, it would be possible to apply more powerful analysis techniques such as [9, 49] to these slices of code/heap structures to eliminate the precision losses. This refinement process could be done either as a post processing step or online during the analysis.

7. Related Work

There is a large body of existing work in the areas of both points-to and shape analysis and this work has led to a number of practical and widely used analysis techniques. Rather than attempt to cover the entirety of previous work (which even for the area of points-to analysis requires a full paper to do justice to [18]) we focus specifically on where this analysis sits in the spectrum of memory analysis techniques and how it ties in with other work in the area.

From the viewpoint of the analysis in this paper work on points-to analysis can be seen as falling into two categories, flow-insensitive, and flow-sensitive. Flow-insensitive analyses involve an inherently different set of tradeoffs than the analysis in this paper. These analyses fundamentally prioritize speed and scalability over precision and thus are much faster but produce much less sophisticated information [16, 46]. In particular these approaches can now scale to millions of lines of code with analysis times on the order of a few seconds or less [16]. The second class of points-to style analyses are more precise, tracking information in a flow-sensitive manner [17, 28] and often employing techniques to track information in a way that is sensitive to different call sites, either via a context-sensitive or object-sensitive approach [23, 38, 44]. While these analyses are more precise than flow insensitive points-to analyses they cannot express general shape or sharing properties. However, due to the way that context is tracked they can produce more precise points-to information in some cases than the analysis in this paper and it is an open question if object sensitive techniques can be used to improve on the results in this paper. Somewhat surprisingly these context (or object) sensitive analyses can be slower (and use more memory) than the analysis in this paper.

Work on memory analysis by Latter et. al. [24, 27] 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. Due to the modular and flow insensitive nature of the analysis it is very efficient, capable of analyzing large C++ programs in seconds. The use of a flow-insensitive and a local points-to analysis limits the range of properties that can be extracted and the precision of the analysis. However, as the focus of this work was scalability (instead of expressivity) it provides an interesting contrast in design decisions to the hybrid analysis proposed in this paper. Similarly the work of Hackett and Rugina [15] 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 within these partitions. The work of Ghiya and Hendren [10] is of particular relevance to the work in this paper as it uses points-to and basic reachability predicates to compute shape information and in Sec. 4.3 notes the challenges of using weak updates when analyzing shape properties.

There is an extensive body of work on shape analysis [5, 9, 11, 12, 21, 37, 42, 43, 49], and while the work in presented in this paper eschews the use of materialization and case splitting in the abstract transfer functions, it borrows heavily from existing work in the design of the abstract domain and in the selection of properties it encodes. In particular the domain in this paper is based on the basic *storage shape graph* construction [5], which is then augmented with additional information on data structure shape [10] and sharing information (injectivity) [37]. However, as opposed to using a partitioning scheme based on type or allocation site as done in [5] (or in most work on points-to analysis) the approach in this paper always creates a fresh node in the graph during the *allocation* operation. This node is then grouped into other data structures as needed using a normal form operation based on connectivity and a set of equivalence relations on the properties of the nodes [30, 33, 49]. The simplicity of the transfer functions in this work, as opposed to the more sophisticated shape analysis transfer functions, results in a much faster and more scalable analysis at the cost of a small amount of precision.

8. Conclusion

This paper introduced *Structural Analysis*, a novel memory analysis technique based on the combination of a shape analysis style abstract heap model, a normal form driven by empirical studies of heap structures in real-world object-oriented programs, and a set of points-to analysis style transfer functions. The resulting hybrid memory analysis is able to precisely identify various structures in memory and to track sharing, shape, and reachability relations on them. At the same time the simple points-to style transfer functions and congruence closure based normal form allow the analysis to efficiently process the effects of various program statements and quickly converge to a final fixpoint (despite using extensive call-graph cloning in the interprocedural analysis). We believe that the combined scalability and precision, plus the hybrid shape and points-to analysis structure presents both immediate benefits and unique opportunities for future research. The development of an expressive and scalable heap analysis is a valuable contribution wrt. the wide range of other research that depends on information about the program heap. However, we also believe further work in the area of hybrid analysis approaches, such as adding object-sensitivity or integrating aspects from SMT or separation logic based approaches, will be fruitful areas of investigation. As such we believe the analysis presented in this paper represents the introduction of a significant new class of heap analysis and represents an important advancement in the state of the art in precise and scalable heap analysis techniques.

References

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

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

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

[4] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

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

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

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

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

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

[10] 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.

[11] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.

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

[13] S. Guyer and K. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *OOPSLA*, 2004.

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

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

[16] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *PLDI*, 2007.

[17] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL*, 2009.

[18] M. Hind. Pointer analysis: haven't we solved this problem yet? In *ISSTA*, 2001.

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

[20] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[21] J. Jenista, Y. Eom, and B. Demsky. Using disjoint reachability for parallelization. In *CC*, 2011.

[22] Jolden Suite. `http://www-ali.cs.umass.edu/DaCapo/`.

[23] N. Jones and S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL*, 1982.

[24] C. Lattner and V. Adve. Data Structure Analysis: An Efficient Context-Sensitive Heap Analysis. Technical Report UIUCDCS-R-2003-2340, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Apr 2003.

[25] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.

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

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

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

[29] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 2008.

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

[31] M. Marron, E. Barr, and C. Bird. Collecting a heap of shapes. In Preparation, 2011.

[32] M. Marron, M. Hermenegildo, D. Stefanovic, and D. Kapur. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, 2008.

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

[34] M. Marron, O. Lhotak, and A. Banerjee. Scalable interprocedural analysis. In *Submission*, 2011.

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

[36] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting runtime heaps for program understanding. In *Submission*, 2011.

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

[38] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 2005.

[39] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 1980.

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

[41] N. Rinetzky, J. Bauer, T. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.

[42] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *POPL*, 2011.

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

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

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

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

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

[48] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.

[49] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *CAV*, 2008.