Programming Paradigm Driven Heap Analysis

Mark Marron[1] Ondřej Lhoták[2] Anindya Banerjee[1]

[1]IMDEA Software Institute, {mark.marron, anindya.banerjee}@imdea.org
[2]University of Waterloo, olhotak@uwaterloo.ca

Abstract. The computational cost and precision of a shape style heap analysis is highly dependent on the way method calls are handled. This paper introduces a new approach to analyzing method calls that leverages the fundamental object-oriented programming concepts of encapsulation and invariants. The analysis consists of a novel partial context-sensitivity heuristic and a new take on *cutpoints* that, in practice, provide large improvements in interprocedural analysis performance while having minimal impacts on the precision of the results.
The interprocedural analysis has been implemented for .Net bytecode and an existing abstract heap model. 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. Our experimental evaluations show that, despite the use of partial context sensitivity heuristics, the static analysis is able to precisely approximate the ideal analysis results. Further, the results show that the interprocedural analysis heuristics and the approach to cutpoints used in this work are critical in enabling the analysis of large real-world programs, over 30K bytecodes in less than 65 seconds and using less than 130 MB of memory, and which could not be analyzed with previous approaches.

1 Introduction

Understanding the structure and evolution of the program heap is a critical part of understanding the semantics of a program, and thus is a central issue to optimizing, refactoring, or checking a program for errors. A central challenge when analyzing the memory behavior of a program is that memory structures are often long lived and can be passed through many method calls which may both use and modify them. Thus, using context-sensitivity when analyzing method calls is critical to the efficiency and accuracy of a static analysis of the program heap [12, 17, 27, 30]. Context insensitive analysis approaches provide good computational performance. However they often result in overly approximate analysis results. Conversely fully context-sensitive analyses provide precise heap information but are computationally expensive and often intractable for all but trivial programs. Thus, there is interest in the development of heuristics that trade off small amounts of precision for large increases in analysis performance [12, 17]. The general approach to analyzing a program in a context-sensitive manner involves building a memo table of the analysis input and outputs ($\widehat{h}_{in}$, $\widehat{h}_{out}$) for each method in the program. Whenever a method call is encountered the analysis checks if there is an appropriate match in the memo table. If there is a match it is used, otherwise a new entry is added and the needed output model is computed. When analyzing a program in this manner it is critical to minimize the number of entries in these memo tables.

Approaches to minimizing the number of states in the memo tables include defining normal form constructions [16, 31] for the heap models, the introduction of cutpoints [17, 26], and partially context-sensitive call analysis heuristics [12, 27]. However, the introduction of these heuristics (and simplifications) are often done in an ad-hoc manner and can be improved upon by explicitly considering how encapsulation, invariants, and pre/post conditions are used in the construction of object-oriented programs. While very few programs actually contain explicit (much less machine readable) annotations for these properties, empirical results [1, 14] suggest that in general object-oriented programs make heavy implicit use of these design concepts. This paper proceeds under the assumption that the program under analysis has been constructed using basic encapsulation and class invariant concepts but that these invariants (or pre/post conditions) are not made explicit in the code. If this assumption does not hold then the precision of the results will be decreased although they will still be sound.

Under the assumption that a program generally follows good object-oriented design principles there are a number of inferences that can be made about the state of the heap at the entry and exit of method calls. In particular at the entry of a method call every object that is visible to the callee should satisfy its class invariant and the reachable heap state should satisfy the pre-condition of the method. Similarly we know that at the return from the method all objects should again satisfy their class invariants and the caller should not depend on anything that is not ensured by the method post-condition. In an ideal world this would imply that context insensitive analyses would work well but in practice developers do not adhere to this strict use of isolation and uniformity in their programs. To allow for this practical reality we need to take a more relaxed approach to merging and simplifying the input/output models. One way to do this is to use the hypothesis that if a method has a simple implementation then the programmer is more likely to break encapsulation by *peering* into the implementation while for complex methods the programmer will make few assumptions beyond what is implied by the class invariants and pre/post conditions. We can use the call graph structure as a measure of how complex the called method is based on the number of subcalls and recursion.

Contributions. To address the challenges in analyzing real world object-oriented programs with a shape style heap domain this paper makes the following contributions:

- This paper introduces a hybrid call graph structure and abstract domain based context-sensitivity heuristic. This is motivated by features of object-oriented programming paradigms, class invariants, and encapsulation. It uses the structure of the call graph to heuristically estimate which calls are good candidates to apply partially sensitive merging operations and how aggressive the merging should be. This approach provides a framework for understanding why, and to what extent, we can expect the use of partial context-sensitivity to impact the precision of the analysis.
- To support the context-sensitive analysis we propose *unique fresh cutpoints* as an alternative definition for *cutpoints* [17, 26]. These cutpoints allow us to project out the part of the abstract heap model that cannot be accessed by the callee method. In contrast to previous work these cutpoints are always created with fresh names. This eliminates issues with spurious name conflicts and the creation of multiple memo table entries that differ due to naming of cutpoints, while still ensuring termination.

In order to quantify the performance and precision impacts of the interprocedural analysis technique we present an extensive experimental evaluation (Section 4) of well known benchmarks from SPEC JVM98 and DaCapo. The evaluation shows that the interprocedural analysis techniques proposed in this paper have a negligible impact on the precision of the analysis. The base domain used in this paper is capable of precisely expressing the majority of the connectivity, shape, and sharing properties that occur in practice and, despite the use of partial context-sensitivity, the static analysis is able to precisely (with a rate of 80-90%) approximate the ideal results. The approach in this paper enables the analysis of real world programs with an expressive shape style heap domain, requiring less than 65 seconds and 130 MB of memory for programs up to 30K bytecodes. Thus, this work provides a new framework, based on the principles of object-oriented program design, for thinking about interprocedural analysis techniques.

2 Abstract Heap Domain

For concreteness and to enable empirical evaluation it is useful to have a fixed abstract heap model. Thus, for this work we use the *Structural Analysis* domain from [16] which is briefly summarized in this section.

2.1 Concrete Heaps

The state of a concrete program is modeled 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 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 (and array indices), Labels, a concrete heap is a tuple $(\mathsf{Env}, \sigma, \mathsf{Ob})$ where:

$$\begin{aligned} \mathsf{Env} \in \text{Environment} &= \mathsf{Vars} \rightharpoonup \mathsf{Addresses} \\ \sigma \in \text{Store} &= \mathsf{Addresses} \to \text{Objects} \cup \{\mathsf{null}\} \\ \mathsf{Ob} &\in 2^{\text{Objects}} \\ \text{Objects} &= \mathsf{OID} \times \mathsf{Types} \times (\mathsf{Labels} \rightharpoonup \mathsf{Addresses}) \\ &\quad \text{where the object identifier set } \mathsf{OID} = \mathbb{N} \end{aligned}$$

Each object o in the set Ob 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. We use the notation $\mathsf{Ty}(o)$ to refer to the type of an object. The notation $o.l$ refers to the value of the field (or array index) l in the object. It is also useful 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 $\mathsf{Fld} : \mathsf{Types} \mapsto 2^{\mathsf{Labels}}$ to get the set of all fields (or array indices) that are defined for a given type.

In the context of a specific concrete heap, $(\mathsf{Env}, \sigma, \mathsf{Ob})$, a *region* of memory is a subset of concrete heap objects $C \subseteq \mathsf{Ob}$. 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:

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

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

Shape. We characterize the shape of regions of memory using standard graph theoretic notions of trees and general graphs 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 general graphs. Given a region C in the concrete heap $(\mathsf{Env}, \sigma, \mathsf{Ob})$:

- 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 $\mathsf{tree}(C)$ holds if the subgraph $(C, P(C, C, \sigma))$ is acyclic and does not contain any pointers that create cross edges.
- The predicate $\mathsf{none}(C)$ holds if the edge set in the subgraph is empty, $P(C, C, \sigma) = \emptyset$.

2.2 Abstract Heaps

An abstract heap is an instance of a storage shape graph [3]. More precisely, an abstract heap graph is a tuple: $(\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}})$ where:

$$\begin{aligned}
\widehat{\mathsf{Env}} \in \text{Environments} &= \mathsf{Vars} \rightharpoonup \widehat{\mathsf{Addresses}} \\
\widehat{\sigma} \in \text{Stores} &= \widehat{\mathsf{Addresses}} \to \mathsf{Inj} \times 2^{\text{Nodes}} \\
&\quad \text{where the injectivity values } \mathsf{Inj} = \{\mathit{true}, \mathit{false}\} \\
\widehat{\mathsf{Ob}} \in \text{Heaps} &= 2^{\text{Nodes}} \\
\text{Nodes} = &\mathsf{NID} \times 2^{\mathsf{Types}} \times \mathsf{Sh} \times (\widehat{\mathsf{Labels}} \rightharpoonup \widehat{\mathsf{Addresses}}) \\
&\quad \text{where the shape values } \mathsf{Sh} = \{\mathsf{none}, \mathsf{tree}, \mathsf{any}\} \\
&\quad \text{and the node identifier set } \mathsf{NID} = \mathbb{N}
\end{aligned}$$

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 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, NID, 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 [16]. 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 object field with the given name.

As with the concrete 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 ($\widehat{\sigma}$) 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 $\widehat{\mathsf{Fld}} : 2^{\mathsf{Types}} \rightarrow 2^{\widehat{\mathsf{Labels}}}$ 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).

2.3 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:

$$(\mathsf{Env},\sigma,\mathsf{Ob}) \in \gamma((\widehat{\mathsf{Env}},\widehat{\sigma},\widehat{\mathsf{Ob}})) \Leftrightarrow \exists \text{ an embedding } \mu \text{ where}$$
$$\mathsf{Injective}(\mu,\mathsf{Env},\sigma,\mathsf{Ob},\widehat{\mathsf{Env}},\widehat{\sigma},\widehat{\mathsf{Ob}}) \wedge \mathsf{Shape}(\mu,\mathsf{Env},\sigma,\mathsf{Ob},\widehat{\mathsf{Env}},\widehat{\sigma},\widehat{\mathsf{Ob}})$$

A concrete heap is an instance of an abstract heap, if there exists an embedding function $\mu : \mathsf{Ob} \rightarrow \widehat{\mathsf{Ob}}$ which respects the structure and labels of the concrete heap and also satisfies the injectivity and shape relations between the structures.

$$\mathsf{Injective}(\mu,\mathsf{Env},\sigma,\mathsf{Ob},\widehat{\mathsf{Env}},\widehat{\sigma},\widehat{\mathsf{Ob}}) = \forall n_s,n_t \in \widehat{\mathsf{Ob}},\widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n_s)).\,\widehat{\mathsf{Inj}}(\widehat{\sigma}(n_s.\widehat{l})) \Rightarrow$$
$$(\widehat{l} \neq [] \Rightarrow \mathsf{inj}(\mu^{-1}(n_s),\mu^{-1}(n_t),l,\sigma)) \wedge (\widehat{l} = [] \Rightarrow \mathsf{inj}_{[]}(\mu^{-1}(n_s),\mu^{-1}(n_t),\sigma))$$

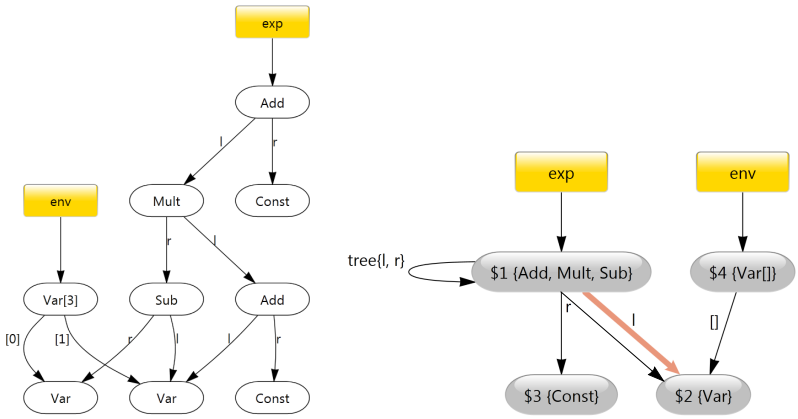
The injectivity relation guarantees that every pointer set marked as injective corresponds to injective (and array injective as needed) pointers between the concrete source and target regions of the heap.

$$\mathsf{Shape}(\mu,\mathsf{Env},\sigma,\mathsf{Ob},\widehat{\mathsf{Env}},\widehat{\sigma},\widehat{\mathsf{Ob}}) = \forall n \in \widehat{\mathsf{Ob}}$$
$$\widehat{\mathsf{Sh}}(n) = \mathsf{tree} \Rightarrow \mathsf{tree}(\mu^{-1}(n,\sigma)) \wedge \widehat{\mathsf{Sh}}(n) = \mathsf{none} \Rightarrow \mathsf{none}(\mu^{-1}(n,\sigma))$$

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

2.4 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.



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

Fig. 1. Concrete and Abstract Heap

Figure 1(b) shows the corresponding abstract heap for this concrete heap. To ease discussion each node in a graph is labeled with a unique node id. The abstraction summarizes the concrete objects into three regions 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. The label `tree{l,r}` on the self-edge expresses that pointers stored in the `l` and `r` fields of the objects represented by node 1 form a tree.

The abstract graph also captures the fact that no `Const` object is referenced from multiple expression objects but that 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 (a fact which could not be obtained via points-to analysis).

2.5 Normal Form

Given the definitions for the abstract heap it is clear that the domain is infinite. Thus, we define a normal form that ensures the number of distinct normal form graphs is finite and use this set during the fixpoint computation (see [16] for additional information).

Definition 1 (Normal Form). *We say that the 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 2).*
3. *All equivalent successors are summarized (Definition 4).*
4. *All variable/global equivalent targets are summarized (Definition 5).*

This normal form definition possesses three key properties that ensure finiteness: (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.

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 $\widehat{\mathsf{Ob}}$.

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, τ_1 and τ_2, that have mutually recursive type definitions denoted: $\tau_1 \sim \tau_2$.

Definition 2 (Recursive Structure). *Given two partitions π_1 and π_2 we define the* recursive structure *congruence relation as*[1]*:*

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

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*. Both of these operations depend on the notion of a successor partition which is based on the underlying structure of the abstract heap graph and a general notion of node compatibility: π_1 a successor of π_2 and $\widehat{l} \Leftrightarrow \exists n_2 \in \pi_2 \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n_2.\widehat{l})) \cap \pi_1 \neq \emptyset$.

Definition 3 (Partition Compatibility). *We define the relation Compatible(π_1, π_2) as:* $\text{Compatible}(\pi_1, \pi_2) \Leftrightarrow \bigcup_{n' \in \pi_1} \widehat{\mathsf{Ty}}(n') \cap \bigcup_{n' \in \pi_2} \widehat{\mathsf{Ty}}(n') \neq \emptyset$.

Definition 4 (Equivalent Successors). *Given π_1, π_2 which are successors of π on labels $\widehat{l_1}$, $\widehat{l_2}$ we define the relation π_1, π_2* equivalent successors *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) and two target partitions π_1, π_2 we define the* equivalent targets *relation as:* $\pi_1 \equiv_t^{\Pi} \pi_2 \Leftrightarrow \text{Compatible}(\pi_1, \pi_2) \wedge (r$ *is a static field* $\vee\, \pi_1, \pi_2$ *only have local var predecessors*$)$.

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 (Definition 2). This computation can be done via a standard worklist algorithm that merges partitions that contain equivalent nodes and can be done in $O((N+E) * log(N))$ time where N is the number of abstract nodes in the initial abstract heap, and E is the number of abstract addresses in the heap.

[1] The definition is symmetric on the properties of the nodes, the $\tau_1 \sim \tau_2$ equivalence relation on types, but is not symmetric on the structure of the underlying graph.

3 Interprocedural Analysis

In order to understand how aggressively abstract models can be merged during the analysis and on which methods the precision loss due to the use of partial context-sensitivity will be the smallest we look to the principles of object-oriented program design. Fundamentally, object-oriented programming is designed around the concepts of encapsulation and information hiding.

These concepts can be seen in the use of object invariants and pre/post conditions which allow a client to use a class with only a minimal knowledge of the internal functioning and allow an implementor to build a class that hides most of the internals from the outside world. This requires the caller to ensure the pre-condition at each method call site and allowing the method implementor to assume this condition. Conversely at call exit the implementor is responsible for ensuring that the post-condition is satisfied and the caller is then allowed to assume anything implied by the post-condition. These conditions are augmented by the class invariants which ensure/guarantee that at the method boundaries each object satisfies the class required invariant in addition to properties in the pre/post conditions of the method. Thus, in the ideal case from the standpoint of the program implementation and the static analysis we could assume that for any property P in the program:

1. If the callee depends on a property P holding as part of the pre-condition of the method (or a class invariant) then all callers to the method *must* ensure that P holds.
2. If the caller depends on a property P that is part of the post-condition (or is a class invariant) then every path through the method *must* ensure that P holds.
3. Alternatively a property P may hold on the portion of the heap that is only reachable from the caller but not reachable from the callee in which case the property P will remain unchanged across the call and will not affect the behavior of the callee.

In the context of the heap analysis the first condition (along with scope visibility) ensures that for every property that holds on the program state *accessible* to the callee that either it is part of the precondition and holds for every abstract model or that the callee makes no assumption on the property. Assuming the join operator ($\sqcup$, e.g. [16] for the domain in Section 2) and the domain are sufficiently precise, the merging should have no impact on the accuracy of the analysis. For example if the callee assumes that an argument (say variable `x`) is always *non-null* then all calls to this method must occur in states where `x` is *non-null*. Thus, all abstract models should entail that `x` is *non-null* and the join of all these models should result in a model that also entails that `x` is *non-null*.

From the perspective of the caller the second condition (along with call scope visibility) ensures that for every property P of the program state after the call, either the property P is part of the post-condition and always holds at method exit or it involves parts of the model that were not accessible to the callee and the value of the property is assumed to be unchanged. In the first case, as the property holds for all models at return from the called method, the join at method return loses no information. Alternatively if the property does not involve part of the program state accessible to the callee then the joins will not affect it (since the use of a *project*/*extend* ensure these parts of the abstract model are not involved in the joins). The project and extend operations can be seen as the addition of a *frame rule* [25, 31] for the interprocedural analysis.

3.1 Unique Fresh Cutpoints

One simple and effective way to accomplish the splitting into a reachable and external heap section is to define a project and an extend operation based on *cutpoints* [17, 26]. The cutpoints are special names introduced for abstract storage targets (edges) that cross from the caller only reachable portion of the abstract heap into the callee reachable portion of the abstract heap. However, as noted in previous work, using a fixed set of names leads to the creation of many redundant entries in the memo tables (that differ only in the names of the cutpoints) and inevitably results in the spurious reuse of the same cutpoint name for multiple cross abstract pointers [17, 26]. However, simply creating a fresh name for each cutpoint that is created leads to non-termination in recursive calls due to the constant addition of new names. To avoid these issues we treat the cutpoints as more than syntactic constructs by defining notions of equivalence classes on them and by extending equality on cutpoints beyond a syntactic property.

To accomplish this we extend the abstract model with a mapping from an unbounded set of cutpoint names, $\mathsf{CPNames}$, to the abstract addresses $\widehat{\mathsf{CP}} \in \text{CPEnvironment} = \mathsf{CPNames} \rightharpoonup \widehat{\mathsf{Addresses}}$, and we extend the definition of the abstract store to be $\widehat{\sigma} : \widehat{\mathsf{Addresses}} \to \{\mathit{true}, \mathit{false}\} \times (2^{\widehat{\mathsf{Ob}}} \cup \mathsf{CPNames})$. We use *fresh* names for each cutpoint we introduce but to avoid the termination issues we add them such that each abstract heap location has at most one cutpoint associated with it. Finally, we add an additional clause to the normal form computation in Definition 1 to handle the cutpoints. For equality matching we check for the existence of an isomorphism between the cutpoint name sets in the two graphs.

Definition 6 (Cross References and Cutpoints). *Given an abstract heap, $\widehat{h}_{\text{in}}$, to a method call and the set of abstract nodes reachable from the callee scope $C \subseteq \widehat{\mathsf{Ob}}$ then:*

- *An abstract location $\widehat{l}$ in node n is a cross pointer if $n \notin C \wedge \widehat{\mathsf{Trgts}}(\widehat{\sigma}(n.\widehat{l})) \cap C \neq \emptyset$.*
- *An abstract location v is a cross variable if $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{Env}}(v))) \cap C \neq \emptyset$.*
- *A cutpoint c_p is redundant if $\widehat{\mathsf{Trgts}}(\widehat{\sigma}(\widehat{\mathsf{CP}}(c_p))) \cap C \neq \emptyset$.*

The project operation works by partitioning the set of nodes in the graph into two sets based on their reachability from the local variables. For each node in the reachable set that has a cross pointer from a location in the unreachable set, a cross variable, or a redundant cutpoint we add a fresh cutpoint which refers to the node. Next we associate this newly created cutpoint with all the cross references to the node (replacing the node target with the name of the freshly created cutpoint). The resulting abstract heap has at most one cutpoint which refers to any node and as each cutpoint name that is added is fresh we are ensured that there are no name collisions. We note that in our heap model there are no relational properties between the incoming pointers (cutpoints) to a node. In domains that do associate relational properties with pointers the same technique can be applied via a generalization to one cutpoint per *equivalence set* of cross pointers.

Additionally, as the particular domain we are using does not perform strong updates [16] we can obtain further speedups in the analysis by taking advantage of the fact that the input heap model is always an under-approximation of the output heap model. So in the project operation defined here we want to extract the callee reachable

portion of the heap and also preserve the entire heap structure (extended with the cutpoint information) in the remaining heap model. If the underlying domain performed strong updates to externally visible heap locations then we would need to, slightly, alter the definition to fully partition the heap structure as done in [17, 26].

$\texttt{project} : \overrightarrow{v}, (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}, \widehat{\mathsf{CP}}) \rightsquigarrow (\widehat{\mathsf{Env}}_e, \widehat{\sigma}_e, \widehat{\mathsf{Ob}}_e, \widehat{\mathsf{CP}}_e), (\widehat{\mathsf{Env}}_r, \widehat{\sigma}_r, \widehat{\mathsf{Ob}}_r, \widehat{\mathsf{CP}}_r)$ where

$\widehat{\mathsf{Ob}}_{reach} = \{n \mid n \in \widehat{\mathsf{Ob}} \wedge n \text{ reachable from a callee variable in } \overrightarrow{v}\}$

$(\widehat{\mathsf{Ob}}_r, \widehat{\mathsf{Ob}}_e) = (\widehat{\mathsf{Ob}}_{reach}, \widehat{\mathsf{Ob}})$

$(\widehat{\mathsf{Env}}_r, \widehat{\mathsf{Env}}_e) = (\{[v \mapsto \widehat{a}_v] \mid v \in \overrightarrow{v}\}, \widehat{\mathsf{Env}})$

$(\widehat{\sigma}_r, \widehat{\sigma}_e) = (\{[\widehat{a} \mapsto \widehat{\sigma}(\widehat{a})] \mid \widehat{a} \text{ a reachable address from a callee variable}\}, \widehat{\sigma})$

$(\widehat{\mathsf{CP}}_r, \widehat{\mathsf{CP}}_e) = (\emptyset, \widehat{\mathsf{CP}})$

$\forall n_r \in \widehat{\mathsf{Ob}}_r$ let c_p be fresh cutpoint name for n

$\forall n_e \in \widehat{\mathsf{Ob}}_e, \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n_e))$ if $n_e, \widehat{l}$ is cross pointer then

$\widehat{\sigma}_e(n_e.\widehat{l}) = (\widehat{\mathsf{Inj}}(\widehat{\sigma}_e(n_e.\widehat{l})), \widehat{\mathsf{Trgts}}(\widehat{\sigma}_e(n_e.\widehat{l}) \cup \{c_p\})$

$\forall v \in Dom(\widehat{\mathsf{Env}})$ if v is cross variable then

$\widehat{\sigma}_e(\widehat{\mathsf{Env}}_e(v)) = (true, \widehat{\mathsf{Trgts}}(\widehat{\sigma}_e(\widehat{\mathsf{Env}}_e(v))) \cup \{c_p\})$

$\forall c'_p \in Dom(\widehat{\mathsf{CP}})$ if c'_p is redundant then

$\widehat{\sigma}_e(\widehat{\mathsf{CP}}_e(c'_p)) = (true, \widehat{\mathsf{Trgts}}(\widehat{\sigma}_e(\widehat{\mathsf{CP}}_e(c'_p))) \cup \{c_p\})$

$\widehat{\mathsf{CP}}_r = \widehat{\mathsf{CP}}_r \cup \{[c_p \mapsto \widehat{a_{c_p}}]\}$ where $\widehat{a_{c_p}}$ a fresh address

$\widehat{\sigma}_r = \widehat{\sigma} \cup \{[\widehat{a_{c_p}} \mapsto \{n\}]\}$

$\texttt{extend:}\ (\widehat{\mathsf{Env}}_e, \widehat{\sigma}_e, \widehat{\mathsf{Ob}}_e), (\widehat{\mathsf{Env}}_r, \widehat{\sigma}_r, \widehat{\mathsf{Ob}}_r) \rightsquigarrow (\widehat{\mathsf{Env}}', \widehat{\sigma}', \widehat{\mathsf{Ob}}')$ where

$\widehat{\mathsf{Ob}}' = \widehat{\mathsf{Ob}}_e \sqcup \widehat{\mathsf{Ob}}_r$

$\widehat{\mathsf{Env}}' = \widehat{\mathsf{Env}}_e + [v_{ret} \mapsto \widehat{\mathsf{Env}}_r(v_{ret})]$

$\widehat{\sigma}' = \widehat{\sigma}_e \sqcup \widehat{\sigma}_r$

$\forall c_p \in Dom(\widehat{\mathsf{CP}}_r), \widehat{a} \in Dom(\widehat{\sigma}')$

if $c_p \in \widehat{\mathsf{Trgts}}(\widehat{\sigma}'(\widehat{a}))$ then

$\widehat{\sigma}' = \widehat{\sigma}' + [\widehat{a} \mapsto (\widehat{\mathsf{Inj}}(\widehat{\sigma}'(\widehat{a})), \widehat{\mathsf{Trgts}}(\widehat{\sigma}'(\widehat{a})) \setminus \{c_p\} \cup \widehat{\sigma}_r(\widehat{\mathsf{CP}}_r(c_p)))]$

$\widehat{\mathsf{CP}}' = \widehat{\mathsf{CP}}_e$

After analyzing the callee method body we need to recombine the two partitions of the abstract heap. The extend operation handles this recombination. It proceeds by taking the union of the contents of the two abstract heaps (the callee reachable and only caller reachable partitions) and then replacing the cutpoint names inserted in the project operation with the needed values. Since we preserved the entire original heap structure in the project operation we cannot simply union the abstract nodes and environments (since the same node or address mapping may appear in both). Instead we must perform

a join on these structures by taking the union of any elements that appear in only one of the abstract heaps and performing a pairwise join on the properties [16] of any duplicate nodes or environment mappings.

In the normal form computation of the abstract heap graph, Section 2, the normal form is based on the construction of equivalence classes based on predecessor relations and the identification of recursive data structures. As the introduction of cutpoints adds a new source of possible predecessors we need to extend the normal form to deal with these as well. We note that cutpoints can be introduced in both calls to simple acyclic parts of the call-graph as well as to parts which may contain recursive calls.

In order to preserve as much information as possible we want to only merge nodes that have the same sets of cutpoints referring to them but in order to ensure termination we must ensure that this condition does not result the violation of conditions 2 through 4 in Definition 1. For example it would be possible for a recursive call to allocate a new node and add it to a list before making a recursive call with the list as an argument. This would result in a new node with a distinct cutpoint being created at each call and the abstract graph having no upper bound on its size. We note that the only way this can happen is with cutpoints added in recursive calls, any non-recursive path though the call graph has a finite length and thus only creates a finite number of cutpoints. Thus, we can strongly distinguish nodes in the graph based on cutpoints added in non-recursive calls and only weakly distinguish them based on cutpoints from recursive calls. Thus we update the *Compatible* relation Definition 3 to distinguish on cutpoints in addition to types, where $NonRecCP(\pi)$ returns the set of all cutpoints that refer to a node in the given partition and were introduced in a non-recursive call:

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

With these definitions we can now split and rejoin the heap into the section that is relevant to a callee method and the part that is guaranteed to be unchanged across a method call as in [17, 26]. The use of *unique fresh cutpoints* and associated constructions eliminates the problem of creating unneeded memo contexts or spurious name collisions via the use of fresh names. Further, the normal form and uniqueness requirements still ensure termination.

3.2 Context-Sensitivity Heuristics

As most programs deviate from the very strict uses of invariants and pre/post conditions described previously we want to selectively relax how aggressively we merge different call contexts based on how strongly a method adheres to the ideal version of information hiding within the implementation. A simple hypothesis is that the complexity of the called method will have a large impact on the amount of additional assumptions, beyond the invariant and pre/post conditions, that the programmer may depend on. For example simple leaf methods such as getters or setters can often be called in many unrelated states and have very broad pre/post conditions associated with them. Similar cases hold for other methods that are near the leaves in the call graph that can easily have their implementations examined or that may have very broad pre/post conditions. Conversely

methods that are more complex, higher in the call graph and particularly those that are part of large recursive call structures, are much more likely to have strict pre/post conditions and the callers of these methods generally avoid making additional assumptions about the behavior of the called method. This leads to the following heuristic for selecting which methods should be handled with full context-sensitivity and which methods could be handled in a partially context-sensitive manner.

Definition 7 (Context Compatibility). *A given abstract heap* $\widehat{h}_{\text{in}}$ *is* compatible *with a previously memoized input* $\widehat{h}_{\text{memo}}$ *for the call to the method m if:*

- *The method m is in an acyclic part of the call graph and* $\widehat{h}_{\text{in}} \simeq \widehat{h}_{\text{memo}}$.
- *The method m is in a cyclic part of the call graph and* $\widehat{h}_{\text{in}} \approx \widehat{h}_{\text{memo}}$.

The $\simeq$ operations is fundamentally an extension of the basic equality operation on the abstract heaps from [16] to include the cutpoint set names. Given two abstract heaps $(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1 \widehat{\mathsf{CP}}_1)$ and $(\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2, \widehat{\mathsf{CP}}_2)$ we first determine if they are structurally isomorphic (i.e., if there is an isomorphism, ϕ on the graph structures that respects variable and field labels), then we check that all abstract node and store properties in $(\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2, \widehat{\mathsf{CP}}_2)$ have the same values in $(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1, \widehat{\mathsf{CP}}_1)$ under the isomorphism. Finally, we check is there is an isomorphism ϕ_{cp} between the cutpoints that respects the reachability relations on the graphs.

$$
\begin{aligned}
&(\widehat{\mathsf{Env}}_1, \widehat{\sigma}_1, \widehat{\mathsf{Ob}}_1) \simeq (\widehat{\mathsf{Env}}_2, \widehat{\sigma}_2, \widehat{\mathsf{Ob}}_2) \Leftrightarrow \exists \phi, \phi_{cp} \\
&\quad \forall n \in \widehat{\mathsf{Ob}}_1 \,.\, \widehat{\mathsf{Ty}}(n) = \widehat{\mathsf{Ty}}(\phi(n)) \wedge \widehat{\mathsf{Sh}}(n) = \widehat{\mathsf{Sh}}(\phi(n)) \\
&\qquad \wedge \forall \widehat{l} \in \widehat{\mathsf{Fld}}(\widehat{\mathsf{Ty}}(n)) \,.\, \widehat{\mathsf{Inj}}(\widehat{\sigma}_1(n)) = \widehat{\mathsf{Inj}}(\widehat{\sigma}_2(\phi(n))) \\
&\qquad \wedge \forall \text{ cutpoints } c_p \in \widehat{\mathsf{CP}}_1 \,.\, \widehat{\mathsf{Trgts}}(\widehat{\sigma}_2(\phi_{c_p}(c_p))) = \{\phi(n') \mid n' \in \widehat{\mathsf{Trgts}}(\widehat{\sigma}_1(c_p))\}
\end{aligned}
$$

For the $\approx$ operation, given $\widehat{h}_{mrg} = \widehat{h}_{in} \sqcup \widehat{h}_{memo}$, then $\approx$ is defined as: $\widehat{h}_{in} \approx \widehat{h}_{memo} \Leftrightarrow$ $\forall v \in \widehat{\mathsf{Env}}_{memo}$ the points-to set in $\widehat{h}_{mrg}$ for v is identical to the points-to set for v in $\widehat{h}_{memo}$. Formally, given $T_{memo} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}_{memo}(\widehat{\mathsf{Env}}_{memo}(v)))$ and $T_{mrg} = \widehat{\mathsf{Trgts}}(\widehat{\sigma}_{mrg}(\widehat{\mathsf{Env}}_{mrg}(v)))$ which we assume always are empty or have size zero (Algorithm 1) or one then:

$$
\begin{aligned}
&T_{memo} \equiv T_{mrg} \Leftrightarrow |T_{mrg}| = |T_{memo}| \wedge \forall n_{mrg} \in T_{mrg}, n_{memo} \in T_{memo} \\
&\qquad \widehat{\mathsf{Ty}}(n_{mrg}) \cap \widehat{\mathsf{Ty}}(n_{memo}) \neq \emptyset \\
&\qquad \wedge \mathit{InDegree}(n_{mrg}) = \mathit{InDegree}(n_{memo}) \wedge \mathit{InVars}(n_{mrg}) = \mathit{InVars}(n_{memo})
\end{aligned}
$$

This test is based on the hypothesis that at the entry of complex method calls the programmer makes very few assumptions about the behavior of the method beyond what is provided by the pre/post conditions. Thus, if two abstract heaps have the same aliasing relations on the argument variables then any differences are in the internal heap structures and are unimportant details that are irrelevant to the overall behavior of the program. This matching enables the analysis to remain fully-context sensitive for simple methods that may not follow a strict pre/post condition protocol with while more complex (and recursive methods) that are likely to have more strict pre/post conditions will be treated with less context-sensitivity.

3.3 Complete Partial Context-Sensitive Call Analysis

The overall interprocedural analysis algorithm is based on a simple worklist approach where methods are taken from and added to a pending worklist as new input models are seen or the result of a child call is updated. During the analysis when a call to a method m is encountered with the input described by the abstract heap $\widehat{h}$ the interprocedural analysis looks at the memoized results in the memotable for the given method. If we find an entry that matches with the input model we return the memoized result model otherwise a new model is added to the memotable for later analysis.

Definition 8 (Memo Table Representation). *For each method m in the program we maintain a list of memoized analysis models* $[\lambda_1, \ldots \lambda_k]$ *where* $\lambda_i = ((\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}, \widehat{\mathsf{CP}})_i^{\text{in}}, (\widehat{\mathsf{Env}}, \widehat{\sigma}, \widehat{\mathsf{Ob}}, \widehat{\mathsf{CP}})_i^{\text{out}})$.

The *AnalyzeCall* method (Algorithm 1) takes a method call and a receiver object. The first step ensures that each argument variable has at most one abstract target. Next we find the set of possible method implementations for the virtual call. For each of these possibilities we analyze the callee method body (or fetch the memoized result) with the given input abstract heap. The possible implementation methods are then analyzed as needed (via the *AnalyzeBody* method) with the argument abstract model and the results are accumulated to produce the result abstract model ($\widehat{h}_f$).

The *EnsureUniqueAbstractTarget* method simply checks each argument variable and does case splitting if there is more than one target. This splitting operation is useful in preventing ambiguity in the caller scope from propagating into the callee scope which improves the effectiveness of the memoization. The *GetImpls* algorithm takes a call signature c_{sig} and an abstract heap $\widehat{h}$. It then looks up every possible method target of that call signature given the possible types of the receiver object (`this`).

Algorithm 1: AnalyzeCall

input : program p, caller m_{from}, call sig. c_{sig}, abstract heap $\widehat{h}$
output: abstract heap $\widehat{h}_f$
$\widehat{h}_f \leftarrow \bot$;
heapset $\leftarrow$ EnsureUniqueAbstractTarget($\widehat{h}$);
for $\widehat{h}_i \in$ *heapset* **do**
  methodset $\leftarrow$ p.GetImpls(c_{sig}, $\widehat{h}_i$);
  for $m \in$ *methodset* **do**
    $\widehat{h}_t \leftarrow$ AnalyzeBody(m, $\widehat{h}_i$);
    $\widehat{h}_f \leftarrow \widehat{h}_f \sqcup \widehat{h}_t$;
return $\widehat{h}_f$;

The *AnalyzeBody* method begins by examining an approximate call graph computed by the frontend (*IsComplexCall*) and selects one of the matching strategies, for simple

acyclic components or for calls that are part of a cyclic component of the call graph, and selects the appropriate matching algorithm, $\approx$ or $\simeq$, to use in searching for a match.

Algorithm 2: AnalyzeBody

input : method m, abstract heap $\widehat{h}$
output : abstract heap $\widehat{h}_f$
if *IsComplexCall(m)* **then**
    (*cmplx*, *match*) $\leftarrow$ (*true*, $\approx$);
else
    (*cmplx*, *match*) $\leftarrow$ (*false*, $\simeq$);
memotable $\leftarrow$ MemoTableFor(m);
($\widehat{h}_e$, $\widehat{h}_r$) $\leftarrow$ project(m.Args, $\widehat{h}$.Copy(), *cmplx*);
if memotable.*HasMatchFor*($\widehat{h}_r$, match) **then**
    $\widehat{h}_{acc} \leftarrow$ *memotable*.GetMatchMergeIfNeeded($\widehat{h}_r$, *cmplx*, *match*);
else
    $\widehat{h}_{acc} \leftarrow$ *memotable*.AddNewEntry($\widehat{h}_r$);
$\widehat{h}_f \leftarrow$ extend($\widehat{h}_e$, $\widehat{h}_{acc}$);
return $\widehat{h}_f$;

If we are matching via the $\approx$ operation then the analysis also updates memoized input state as the merge of the matched memoized input model and the new argument model. The algorithm for handling the matching and merge if needed, *GetMatchMergeIfNeeded*, is shown in Algorithm 3. The two helper functions *UniqueifyCutpoints* and *RemapCutpoints* are used to ensure that after the join the one cutpoint per node property is maintained and then to ensure that the set of cutpoints used in the input model match the set of cutpoints that appear in the output model.

Algorithm 3: GetMatchMergeIfNeeded

input : memoized values $[\lambda_1, \ldots \lambda_k]$, abstract heap $\widehat{h}$, bool *cmplx*, compare *match*
output : abstract heap $\widehat{h}_f$
for $\lambda \in [\lambda_1, \ldots \lambda_k]$ **do**
    if match($\widehat{h}$, λ^{in}) **then**
        if cmplx **then**
            $\lambda^{in} \leftarrow \lambda^{in} \sqcup \widehat{h}$;
            cpremap $\leftarrow$ *UniqueifyCutpoints*(λ^{in});
            $\lambda^{out} \leftarrow \lambda^{out} \sqcup \widehat{h}$;
            RemapCutpoints(*cpremap*, λ^{out});
        return λ^{out};

4 Experimental Evaluation

Our benchmark suite consists primarily of direct C# ports of commonly used Java benchmarks from Jolden [9], the db and raytracer programs from SPEC JVM98 [28], and the luindex and lusearch programs from the DaCapo suite [9]. Additionally we have analyzed the heap abstraction code, runabs, from [19]. In practice we translate the .Net assemblies to a simplified IR (intermediate representation) which allows us to remove most .Net specific idioms from the core analysis and to perform specialized analyses on the standard libraries [20]. 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. The domain, operations, and data flow analysis algorithms are all implemented in C# and are publicly available.[2]

Client Applications

The analysis in this paper tracks general classes of properties that have shown, in past work, to be both relevant and useful in a wide range of client applications [5–8, 11, 14, 18]. However, we have performed additional small scale implementations and case studies with the analysis results to ensure that the particulars of the domain defined in this paper are useful for these types of optimization and program understanding applications. These case studies include:

- The introduction of thread-level parallelization, as in [18], to obtain a 3× speedup for bh on our quad-core machine.
- Data structure reorganization to improve memory usage and automated leak detection, as in [19], to obtain over a 25% reduction in live memory in raytrace.
- The computation of ownership information for object fields, as in [14], identifying ownership properties for 22% of the fields and unique memory reference properties for 47% of the fields in lusearch.

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 more general measurement of the possible imprecision due to the partial context-sensitivity heuristics than the use of a specific client application (which may hide precision losses that *happen* not to matter for the particular client).

Quantitative Precision

We define precision relative to a hypothetical *perfect analysis* which uses the same abstract domain from Section 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

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

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.

One possible concern with this approach is that the base abstract domain may be very coarse, i.e., $\bigsqcup_{h \in \{h_1,\ldots,h_k\}} \alpha(h)$ is always $\top$ or another very imprecise value. To account for this we report the average percentage of properties (shape and injectivity) that the runtime result marks as precise (none or tree and *injective*) in the models (the *Runtime Precise Rate* group in Table 1). This table shows that in practice the domain achieves a very high rate of precise identification of *shape* values, on average over 90% or more of nodes are precisely identified (the *Shape* column), and a similarly high rate of precise *injectivity* values, on average nearly 90% of the edges are identified as being *injective* (the *Injectivity* column). For reference the example abstract heap, Figure 1(b), abstract heap would have a 100% precise *shape* rate and a 75% precise *injectivity* rate. Thus, we can see that in general the base domain is exceptionally effective in representing the heap properties we are interested and is an effective baseline for comparison.

	Static Match Rate			Runtime Precise Rate	
Benchmark	Region	Shape	Injectivity	Shape	Injectivity
power	100%	100%	100%	100%	100%
bh	100%	90%	87%	100%	100%
db	100%	100%	81%	100%	100%
raytracer	80%	85%	83%	89%	98%
luindex	95%	95%	82%	100%	91%
lusearch	93%	90%	84%	96%	89%
runabs	97%	98%	87%	94%	90%

Table 1. Static Match is percentage of each property (at method entries) which is accurately predicted by the static analysis when compared to *perfect analysis*. Runtime Precise is the percentage of properties that the *perfect analysis* captures precisely.

Table 1 shows the results of comparing the results from our *perfect analysis* with the results from the static analysis described in this paper. In this table we report the percentage of properties in the static analysis results that are the same as reported by the runtime analysis for regions, shapes, and injectivity values. The region percentage (the *Region* column) is the 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). These results show that the analysis is able to extract a large percentage of the properties that can be expressed via the abstract domain (i.e. the static analysis is able to answer any client query on aliasing or shape as accurately as possible given the underlying domain for 80% to 90% of queries). Running the static analysis without the partial context-sensitivity heuristics provides a slight increase in the accuracy (3% in the best case) and running the analysis without the use of the unique-cutpoints is infeasible (per Table 2). Thus, the use of the fresh cutpoints and context-sensitivity heuristics result in only small losses in precision in practice.

Analysis Performance

We next examine the time and memory requirements of analyzing a program using the method described in this paper. Table 2 shows the cost of running several analysis variations. The cost of analyzing a program with full context-sensitivity and without the use of the fresh cutpoints is generally infeasible (the No Opt column in Table 2) with the heap domain we are using (all but two of the benchmarks time out). The next variation of interprocedural analysis we examine (the Project column) uses full context-sensitivity but applies the project/extend with fresh cutpoints as described in this paper. The Optimized column uses the partial context-sensitivity heuristics described in this paper and the fresh cutpoint project/extend operations. For each benchmark we list the number of bytecode instructions 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.

Benchmark Statistics			No Opt		Project		Optimized	
Name	Insts	Methods	Time	Mem	Time	Mem	Time	Mem
power	3298	320	1.21s	$\leq$20MB	0.11s	$\leq$20MB	0.09s	$\leq$20MB
bh	3723	351	5.38s	72MB	0.61s	$\leq$20MB	0.42s	$\leq$20MB
db	2873	315	-	-	0.21s	$\leq$20MB	0.21s	$\leq$20MB
raytrace	9808	476	-	-	12.11s	40MB	6.72s	32MB
luindex	26852	246	-	-	20.15s	70MB	12.1s	53MB
lusearch	33632	272	-	-	197.79s	489MB	64.3s	130MB
runabs	27875	253	-	-	12.84s	68MB	10.4s	60MB

Table 2. Statistics and aggregate performance of the *No Opt*, *Project*, and *Optimized* analysis. For the timeout we use a limit of 10 min. or 500MB of memory.

The *no opt* approach is unable to handle most of the benchmarks, so we do not discuss it further. When compared to the *project* approach the *optimized* technique is a factor of 2-4 faster in all of the smaller benchmarks and enables the analysis to complete the larger program where the *project* analysis times out (lusearch). Both the *project* and *optimized* analysis require very little memory to analyze the simpler programs (less than 20MB for most of them). However, as the size and complexity of the program increases we see that the *optimized* analysis improves memory usage as well.

To understand the source of this difference we examine the impact of the *optimized* analysis on the maximum number of memo entries for any method and the average number of memo entries per method (shown in Table 3). Looking at the results in the table the impact on the number of memo entries per method is quite significant. This reduction occurs in the maximum number of entries and in the average number per method. This has a huge impact on the number of comparisons that need to be done to test if a given call model is memoized and in the number of times each method body is analyzed. As the program gets larger this reduction has a large impact in the amount of time and memory needed to analyze a program. We note that the improvement is

small on the simpler benchmarks but become increasingly important as the complexity increases in db and raytrace, and is critical to successfully analyzing luindex, lusearch, and runabs.

Benchmark	Project			Optimized		
Name	Avg Entries	Max Entries	Max Iters.	Avg Entries	Max Entries	Max Iters.
power	1.03	2	3	1.02	2	3
bh	1.46	5	3	1.11	5	2
db	1.80	5	1	1.80	5	1
raytrace	2.36	15	4	2.14	15	4
luindex	2.54	21	7	2.46	17	6
lusearch	11.60	100	17	2.08	22	16
runabs	2.15	14	4	2.11	13	4

Table 3. Comparison of the *Project* and *Optimized* analysis on model set size and memo table entries. *Avg Entries* is the average number of entries for each method memotable, *Max Entries* is the maximum number of entries, and *Max Iters.* is the maximum number of times a memo table entry is analyzed before it reaches a fixpoint.

These results emphasize the robustness of the scaling of the interprocedural analysis with respect to growth in the memo table sizes. In particular the average number of memo entries is nearly constant regardless of the size of the program and thus the memory (time) required by the analysis scales with the number of contexts and number of iterations required to reach a fixpoint.

5 Related Work

There has been a substantial amount of work on techniques for speeding up interprocedural dataflow analysis. These range from methods for efficiently encoding call contexts using BDD's [13, 29], to heuristics that alter the level of context sensitivity based on structural properties of the call or flow graph [12, 21, 24], to methods for generating partial call context tokens [10, 22, 30]. The work in this paper draws on the general concepts of context heuristics presented in [12, 15, 21, 23, 24] and work on project/extend operations [17, 26], to efficiently and precisely manage the challenges present when performing interprocedural heap analysis on object-oriented programs.

This work differs substantially from previous work in a number of respects. One of the key insights into this work is that there exist specific points in the program where the programmer makes generalizing assumptions about the state of the program and thus most of the differences between abstract heaps are not critical to understanding the later behavior of the program. This idea has been exploited implicitly in previous work which often applies some set of heuristics for merging states either at call or local control flow joins [17, 21, 23, 31] but there is little empirical or conceptual justification (beyond the experimental results) for why the specific set of join points selected is a good choice.

Recent work on modular shape analysis [2, 4] has also shown promise in scaling to large programs. However these approaches place substantial restrictions on the programs

that are being analyzed. Techniques in [2] do not allow generalized sharing in the heap structures while the techniques in [4] do not handle programs that contain recursive data structures. Further, the programs that these approaches have been demonstrated on contain relatively shallow and small data structures and the code does not heavily employ recursion or dynamic dispatch. In contrast the approach in this paper does not place any constraints on the programs under analysis and has been demonstrated on a range of programs that use large and complex data structures that are connected in a variety of ways. The benchmark programs used in this paper also extensively employ dynamic dispatch and contain non-trivial recursive traversals of heap structures.

6 Conclusion

This work introduced and provided justification for a novel partial context-sensitivity heuristic, and a new take on *cutpoints* that produce a computationally efficient analysis which is still able to produce precise results in practice. The impact of these contributions was demonstrated by integrating them, and an existing high precision heap analysis domain, into a complete context-sensitive dataflow analysis. Using this algorithm we analyzed a number of well known object-oriented programs. The results provide empirical evidence that the use of fresh cutpoints and partial-context sensitivity heuristics based on principles of object-oriented program design drastically reduce the time and memory required to analyze a program (in a number of cases it is faster than many state of the art context-sensitive points-to analyses). Further, these performance improvements were obtained without a substantial degradation in the precision of the results. Thus, we believe the interprocedural analysis presented in this paper represents an important technical advancement in the state of the art in precise and scalable heap analysis techniques and also introduces a new way to think about how program development concepts can be leveraged in the design of static analysis techniques.

References

[1] E. Barr, C. Bird, and M. Marron. Collecting a Heap of Shapes. Technical Report MSR-TR-2011-135, Microsoft Research, Dec. 2011.

[2] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 2011.

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

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

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

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

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

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

[9] Jolden Suite. http://www-ali.cs.umass.edu/DaCapo/.
[10] N. Jones and S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *POPL*, 1979.
[11] C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI*, 2005.
[12] C. Lattner, A. Lenharth, and V. Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *PLDI*, 2007.
[13] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Method.*, 18(1), 2008.
[14] K.-K. Ma and J. Foster. Inferring aliasing and encapsulation properties for Java. In *OOPSLA*, 2007.
[15] R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.
[16] M. Marron. Structural analysis: Combining shape analysis information with points-to analysis computation. arXiv:1201.1277v1 [cs.PL], 2012.
[17] M. Marron, M. Hermenegildo, D. Stefanovic, and D. Kapur. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, 2008.
[18] M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.
[19] M. Marron, C. Sanchez, Z. Su, and M. Fahndrich. Abstracting runtime heaps for program understanding. `http://heapdbg.codeplex.com/`, 2011.
[20] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.
[21] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP*, 2005.
[22] A. Milanova, A. Rountev, and B. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, 2002.
[23] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *JLP*, 13(2/3), 1992.
[24] T. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In *FSTTCS*, 2007.
[25] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, 2002.
[26] N. Rinetzky, J. Bauer, T. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.
[27] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. In *POPL*, 2011.
[28] Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.
[29] J. Whaley and M. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, 2004.
[30] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.
[31] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.