Shape Analysis with Reference Set Relations

Mark Marron[1], Rupak Majumdar[2], Darko Stefanovic[3], and Deepak Kapur[3]

[1]IMDEA-Software, `mark.marron@software.imdea.org`
[2]University of California Los Angeles, `rupak@cs.ucla.edu`
[3]University of New Mexico, {`darko, kapur`}`@cs.unm.edu`

Abstract. Tracking subset relations between the contents containers on the heap is fundamental to modeling the semantics of many common programing idioms such as applying a function to a subset of objects and maintaining multiple views of the same set of objects. We introduce a relation, *must reference sets*, which subsumes the concept of *must-aliasing* and enables existing shape analysis techniques to efficiently and accurately model many types of containment properties without the use of explicit quantification or specialized logics for containers/sets. We extend an existing shape analysis to model the concept of *reference sets*. Reference sets allow the analysis to efficiently track a number of important relations (*must*-$=$, and *must*-$\subseteq$) between objects that are the targets of sets of references (variables or pointers). We show that shape analysis augmented with reference set information is able to precisely model sharing for a range of data structures in real programs that cannot be expressed using simple must-alias information. In contrast to more expressive proposals based on logic languages (e.g., extensions of first-order predicate logic with transitive closure or the use of a decision procedure for sets), reference sets can be efficiently tracked in a shape analyzer.

1 Introduction

Precise reasoning about the structure of the program heap is crucial to understanding the behavior of a given program, particularly for object-oriented languages. Traditional *points-to* analyses, which calculate sharing properties based on coarse aggregations of the heap (for example by coalescing all cells from the same allocation site and ignoring program flow [15]), are known to be too imprecise for many applications. More precise *shape analysis* techniques [1, 5, 6, 9, 13, 16–19] have been proposed when more accurate information is desired. These analyses recover precise information by distinguishing heap cells based on additional reachability, allocation site, or type information. Using this additional information, these analyses can precisely model recursive data structures [5, 19] and composite structures [1, 6, 18].

Most work on shape analysis has focused on existential (*may*) sharing properties (and by negation, separation properties) of pointers or variables—the fundamental question asked of the abstract heap representations is whether two abstract references *may* represent pointers that *alias* each other. While this is often enough to prove many sophisticated properties of data structures that have limited amounts of sharing or where the sharing is simple (e.g., variable aliasing), the reasoning becomes overly restrictive (and imprecise) for more complex subset relationships among sets of shared objects. Such relationships arise in programs that use multiple views of the same collection of

```
01 Vector V = new Vector();                    05 for(int i = 0; i < A.length; ++i) {
02 Data[] A = new Data[N];                     06   Data d = A[i];
03 for(int i = 0; i < N; ++i)                  07   if(d.f > 0) V.add(d);
04   A[i] = new Data(abs(randInt()));          08 }

                    09 for(int i = 0; i < V.size(); ++i) {
                    10   Data d = V.get(i);
                    11   d.f = 0;
                    12 }
```

Fig. 1. Initialize array (lines 3-4), Filter values (lines 5-7), and Update `f` fields (lines 9-11)

objects (for efficiency, a class might keep the same set of objects in a *Vector* and in a *Hashtable*) or when performing updates on a set of shared elements (filter-map and subset-remove loops, where a sub-collection is first computed then operated on).

We introduce *reference set* relations that track set relations (*must*-$=$, and *must*-$\subseteq$) between the targets of sets of variables/pointers in the concrete program. Thus, *must reference set* information is stronger than, and subsumes *must-aliasing* (which only tracks *must*-$=$ between pairs of variables/pointers). We show that when an existing shape analysis is extended with two simple relations to track the most commonly occurring reference set relations it can efficiently and precisely model many sharing properties in the program, and also model how these properties affect the behavior of the program.

Sharing relations between sets of objects, including reference set relations, can be modeled by extending the analysis with a theory for sets [8] or by quantification with a "forall-exists" quantifier structure (i.e., for all objects pointed to by a reference in array *A*, does there exist a reference in array *B* pointing to the same object?). However, the introduction of additional theories or using more general logics (with quantification and disjunction) makes reasoning computationally expensive. Instead, as demonstrated in this paper, many sharing properties can be efficiently tracked on top of an existing shape analysis with enough accuracy to prove many important sharing relationships.

2 Example and Motivation

Consider the three loops in Figure 1: array initialization, filtering elements into a sub-collection, and updating the contents of the sub-collection. For simplicity the example uses a dummy class `Data` with a single integer field `f`.

The first code fragment allocates an array `A` and then fills it with `Data` objects with random non-negative values stored in their `f` fields. The second loop scans the array for elements that have strictly positive values in the `f` fields and constructs a new vector `V` of these elements. The third loop sets the `f` field of every element in the vector `V` to zero. If these loops are analyzed using one of the existing shape analysis that can model collections, such as [12], we get the abstract heap graph shown in Figure 2(a) at the end of the second loop. In this figure we have simplified the edge/node labels to focus on the concept of how *must* sharing relations between sets of objects can be used to precisely model the behavior of a program.

The simplified model shows the variable `A` referring to a node with an *id* tag of 1 (a unique identifier given to each node/edge to simplify the discussions of the figures) which abstracts an object of type `Data[]`. There may be many pointers stored in this

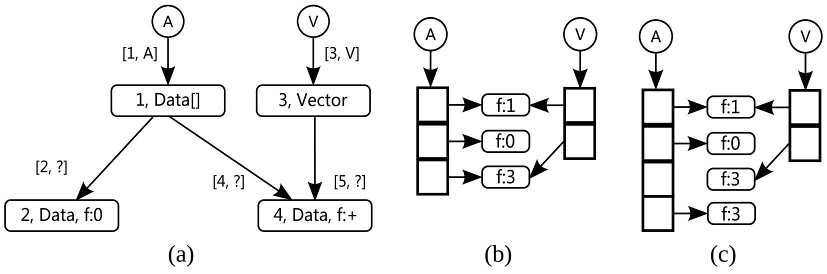


Fig. 2. Abstract model and two possible concrete heaps.

array (these pointers are abstracted by the edges with the *id*'s 2, 4); since these pointers are stored in an array we give them the special storage *offset* `?` (indicating that they are stored at an indeterminate index in the array/container). The two outgoing edges indicate that the pointers stored in the array may either refer to objects abstracted by node 2 or to objects abstracted by node 4. The notation `f:0`, `f:+`, and `f:0+` indicates the values of the integer fields using a simple *sign* domain [4], where `f:0` in node 2 indicates that all the objects that are abstracted by this node have the value 0 stored in the `f` field while the `f:+` entry in node 4 indicates that all the objects abstracted by that node have values in the range $[1, \infty)$ in their `f` fields (and `f:0+`, used later, indicates f values in the range $[0, \infty)$). Figure 2(a) also shows the variable `V` which has an edge to a node abstracting a `Vector` object. The pointers stored in this vector are abstracted by edge 5 and they refer to objects abstracted by node 4.

Based on this information both of the concrete heaps shown in Figure 2(b) and 2(c) are consistent with this model (i.e., they are valid concretizations). In Figure 2(b) we see that array `A` contains three `Data` objects (some of which have 0 field values and some of which have positive values), the first and third of which are also stored in the `Vector V` (which only contains objects with positive values). This heap is clearly a possible result of the construction and filter loops in our example. If we look at the concrete heap shown in Figure 2(c) it is apparent that this program state is infeasible since the contents of `V` are not a subset of `A` and there is a `Data` object in `A` with a positive field value that is not in `V`. However, this concrete heap is consistent with the information provided by the abstract graph model, as the fact that edges 4 and 5 end at the same node *only* means that there *may* exist an object that is referred to by both a pointer abstracted by edge 4 and a pointer abstracted by edge 5. In particular, the abstraction is too weak to prove that at the end of the third loop, every element in `A` has the value zero in the `f` field.

Thus, in order to precisely represent the desired *must* sharing relations between various sets of pointers stored in the array and vector we need to extend the graph model with additional information. The analysis presented in this paper extends a standard shape analysis by tracking two *reference set equivalence* relations on the heap. The first relation is on pairs of abstract edges, which tracks pairs of edges such that the sets of references abstracted by the two edges *must* always refer to exactly the same set of objects. The second relation is on edges and nodes, and tracks edges that abstract a set of references such that all of the objects abstracted by a node are pointed to by one of the references in the set.

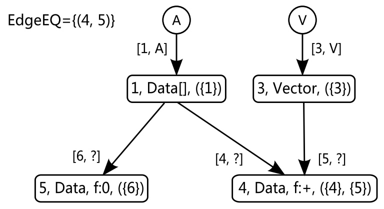


Fig. 3. Abstract Graph With Reference Set Information

These reference set properties allow the analysis to precisely model the result of the construction and filter loops in our example. The model enhanced with the reference set properties is shown in Figure 3. We have made two additions to the model in Figure 2(a). First, the `EdgeEQ` relation tracks which edges abstract references that always refer to the same sets of objects. Second, for each node we add a list of sets of edges such that every object abstracted by the node is referred to by a reference represented by one of the edges in the set. Intuitively, these additional relations tell us that the set of objects referred to by references abstracted by edge 4 is equal to the set of objects referred to by references abstracted by edge 5. This information and the structure of the graph imply that every object stored in the vector `V` *must* also be stored in `A` and also that if an object is stored in `A` it must be either abstracted by node 5 (and have the value 0 stored in the `f` field) or by node 4 (and be stored in `V`, which as desired, excludes the concrete heap in Figure 2(c) from the set of feasible concretizations).

This last property then allows us to precisely model the third loop in the running example. In particular we know that since every object in `A` with a non-zero `f` field is stored in `V` we can infer that if every object in `V` has the `f` field set to 0 then after the loop every object in `A` will have 0 in the `f` field.

3 Concrete and Abstract Heaps

3.1 Concrete Heap and Reference Set Relations

The semantics of memory are defined in the usual way, using an *environment*, mapping variables into values, and a *store*, mapping addresses into values. We refer to the environment and the store together as the concrete heap, which is represented as a labeled, directed multi-graph (V,O,R) where V is a set of *variables*, O is a set of *objects* on the heap, and $R \subseteq (V \cup O) \times O \times L$ a set of *references*, where L is the set of storage location identifiers (a variable name in the environment, a field identifier for references stored in objects, or an integer offset for references stored in arrays/collections).

A *region* of memory $\mathfrak{R} = (C,P,R_{in},R_{out})$ consists of a subset $C \subseteq O$ of the objects on the heap, all the references $P = \{(a,b,p) \in R \mid a,b \in C \wedge p \in L\}$ that connect these objects, the references that enter the region $R_{in} = \{(a,b,r) \in R \mid a \in (V \cup O) \setminus C \wedge b \in C \wedge r \in L\}$, and references exiting the region $R_{out} = \{(a,b,r) \in R \mid a \in C \wedge b \in O \setminus C \wedge r \in L\}$. Note that $\mathfrak{R}$ is determined by C, and we say a region $\mathfrak{R}$ is *induced by* a set C of objects.

Given a region $\mathfrak{R} = (C,P,R_{in},R_{out})$ and a set of references $R_s \subseteq R_{in}$ we define the function: $\mathit{Target}(R_s) = \{o \in C \mid \exists a \in (V \cup O), r \in L \text{ s.t. } (a,o,r) \in R_s\}$.

Definition 1 (Reference Set Relations). *Given a region* $\mathfrak{R} = (C,P,R_{\text{in}},R_{\text{out}})$, *reference sets* $R_s \subseteq R_{\text{in}}$ *and* $R'_s \subseteq R_{\text{in}}$, *we define the following relations:*

Reference Contains $R'_s \preceq R_s$ *if* $\mathrm{Target}(R'_s) \subseteq \mathrm{Target}(R_s)$.
Reference Equivalent $R'_s \sim R_s$ *if* $\mathrm{Target}(R'_s) = \mathrm{Target}(R_s)$.
Region Covers $R_s \triangleright \Re$ *if* $C \subseteq \mathrm{Target}(R_s)$.

Aliasing of two references x, y in the concrete heap is equivalent to the reference set relation $\{x\} \sim \{y\}$, thus the *concrete reference set relations* subsume the standard notion of aliasing.

3.2 Abstract Graphs

Our abstract domain is based on the *storage shape graph* [2,3] approach. Let $\hat{L}$ be a set of abstract storage *offsets* (variable names, field offsets, or special offsets for references stored in arrays/collections) which are related to the storage locations L by an abstraction function $\alpha_{\text{offset}} : L \mapsto \hat{L}$. A *storage shape graph (ssg)* is a tuple of the form $(\hat{V}, \hat{N}, \hat{E})$, where $\hat{V}$ is a set of nodes representing the variables, $\hat{N}$ is a set of nodes (each of which intuitively abstracts a region $\Re$ of the heap), and $\hat{E} \subseteq (\hat{V} \cup \hat{N}) \times \hat{N} \times \hat{L}$ are the graph edges, each of which intuitively abstracts a set of references.

Definition 2 (Valid Concretization of a *ssg*). *A given concrete heap* $h = (V, O, R)$ *is a* valid concretization *of a* labeled storage shape graph $g = (\hat{V}, \hat{N}, \hat{E}, \hat{U})$ *if there are functions* $\Pi_v : V \mapsto \hat{V}$, $\Pi_o : O \mapsto \hat{N}$, $\Pi_r : R \mapsto \hat{E}$ *such that* Π_v *is 1-1, and*

– *for all* $(o_1, o_2, p) \in R$ *with* $o_1, o_2 \in O$, *if* $\Pi_r(o_1, o_2, p) \equiv (n_1, n_2, l)$, *then* $n_1 = \Pi_o(o_1)$, $n_2 = \Pi_o(o_2)$, *and* $l = \alpha_{offset}(p)$.
– *for all* $(v, o, v) \in R$ *with* $v \in V$ *and* $o \in O$, *if* $\Pi_r(v, o, v) \equiv (n_1, n_2, l)$, *then* $n_1 = \Pi_v(v)$, $n_2 = \Pi_o(o)$, *and* $l = v$.

We say (Π_v, Π_o, Π_r) witness *that* h *is a valid concretization of* g. *We introduce the following notation for pre-images of nodes and edges of an ssg:*

– *We write* $h \downarrow_g e$ *for the set* $\{r \in R \mid \Pi_r(r) = e\}$ *of references in the concrete heap* h *that are in the pre-image of* $e \in \hat{E}$ *under* Π_r.
– *We write* $h \downarrow_g n$ *for the concrete region* $\Re$ *induced by the set* $\{o \in O \mid \Pi_o(o) = n\}$.

In our analysis, we extend ssg's with a set of additional instrumentation predicates that restrict the set of valid concretizations of an ssg. Let U denote a set of relations (called *instrumentation predicates*) on concrete objects and references, and let $\hat{U}$ denote instrumentation relations on the nodes and edges of an ssg, with $u : U \to \hat{U}$ a 1-1 map between them. A *labeled storage shape graphs (lssg)* is a tuple $(\hat{V}, \hat{N}, \hat{E}, \hat{U})$ where $(\hat{V}, \hat{N}, \hat{E})$ is a ssg and $\hat{U}$ is a set of relations over $\hat{N}$ and $\hat{E}$. In the following, we refer to lssg's simply as *abstract graphs*. A concrete heap h is a valid concretization of an lssg $(\hat{V}, \hat{N}, \hat{E}, \hat{U})$ if h is a valid concretization of the ssg $(\hat{V}, \hat{N}, \hat{E})$ through the functions Π_v, Π_o, Π_r, and additionally, for each $p \in \hat{U}$, nodes $n_1, \ldots, n_k \in \hat{N}$, and edges $e_1, \ldots, e_l \in \hat{E}$, if $(n_1, \ldots, n_k, e_1, \ldots, e_l) \in p$ holds, then each tuple in $\{(o_1, \ldots, o_k, r_1, \ldots, r_l) \mid o_i \in h \downarrow_g n_i, i \in \{1, \ldots, k\}, r_j \in h \downarrow_g e_j, j \in \{1, \ldots, l\}\}$ is in $u^{-1}(p)$.

For example, in Section 2 we introduced two instrumentation relations *type* and *sign*. Formally, for a set $\{\tau_1, \ldots, \tau_k\}$ of object types, we add an instrumentation relation $\mathtt{Type}[\{\tau_1, \ldots, \tau_k\}] \subseteq \hat{N}$ to $\hat{U}$ corresponding to the relation $\lambda o.\mathtt{typeof}(o) \in \{\tau_1, \ldots, \tau_k\}$ on objects, and require that for each $n \in \mathtt{Type}[\{\tau_1, \ldots, \tau_k\}]$ we have that each object $o \in h \downarrow_g n$ satisfies $\mathtt{typeof}(o) \in \{\tau_1, \ldots, \tau_k\}$. The *sign* relation can be similarly defined.

4 Instrumentation Predicates

4.1 Abstract Reference Sets

We introduce two instrumentation relations that allow us to track many useful properties of the heap: *abstract edge equivalence*, which relates two abstract edges, and *abstract node coverage*, which relates a set of abstract edges to an abstract node.

Abstract Edge Equivalence Given two edges $e, e' \in \hat{E}$, we say e is *edge equivalent* to e', written $e \hat{\sim} e'$, iff every valid concretization h of the abstract graph g must satisfy $(h \downarrow_g e) \sim (h \downarrow_g e')$.

Abstract Node Coverage Given a set of edges $E_c \subseteq \hat{E}$ and an abstract node $n \in \hat{N}$ we say E_c *node covers* n, written $E_c \hat{\triangleright} n$, iff every valid concretization h of the abstract graph g must satisfy $\bigcup\{h \downarrow_g e' \mid e' \in E_c\} \triangleright (h \downarrow_g n)$.

Proposition 1. *Given lssg $g = (\hat{V}, \hat{N}, \hat{E}, \hat{U})$, a valid concretization h of g, $n, n' \in \hat{N}$, and $e, e' \in \hat{E}$.*

1. *If $\{e\} \hat{\triangleright} n$ and $h \downarrow_g e = \emptyset$ then $h \downarrow_g n = \emptyset$ and $h \downarrow_g e' = \emptyset$ for all e' ending at n.*
2. *If $e \hat{\sim} e'$ and $h \downarrow_g e = \emptyset$ then $h \downarrow_g e' = \emptyset$.*
3. *If $\{e\} \hat{\triangleright} n$ and $\{e'\} \hat{\triangleright} n$ then $e \hat{\sim} e'$.*
4. *If $E_c \hat{\triangleright} n$, $E_s \subseteq \{e_s \mid e_s$ ends at $n\}$ then $\bigcup\{h \downarrow_g e_s \mid e_s \in E_s\} \preceq \bigcup\{h \downarrow_g e_c \mid e_c \in E_c\}$.*

Given the definition for *abstract edge equivalence* we can express the standard concept of *must-aliasing* of edges e_1 and e_2 as a special case of the *abstract edge equivalence relation*: e_1 and e_2 *must alias* iff e_1, e_2 each represent a single reference and $e_1 \hat{\sim} e_2$.

We restricted the definition of the *abstract reference* relations to equivalence of edges plus a special relation on nodes. This allows us to track the most common occurrences of reference equivalence (the edge $\hat{\sim}$ relation) and subset relations (the $\hat{\triangleright}$ relation and Proposition 1). We could define a more general relation, where subset relations between sets of edges are tracked. However, this formulation requires tracking a binary relation on the power set of $\hat{E}$, which is undesirable from a computational standpoint.

4.2 Additional Instrumentation Predicates

In addition to tracking type properties of the nodes, and the edge/node abstract reference set relations defined above, the nodes and edges of storage graphs are augmented with the following instrumentation relations introduced in previous work [11].

Linearity. The *linearity* relation is used to track the maximum number of objects in the region abstracted by a given node or the maximum number of references abstracted by a given edge. The *linearity* property has two values: 1, indicating a cardinality of $[0, 1]$, or ω, indicating any cardinality in the range $[0, \infty)$.

Connectivity and Interference. We use two instrumentation relations to track the potential that two references can reach the same heap object in the region that a particular node represents. For this paper we use simplified versions and refer the reader to [11] for a more extensive description of these relations.

Given a concrete region $\Re = (C, P, R_{in}, R_{out})$ and we say objects $o, o' \in C$, are *related* in $\Re$ if they are in the same *weakly-connected*[1] component of the graph (C, P).

To track the possibility that two incoming edges e, e' to the node n abstract references that reach *related* objects in the region abstracted by n we introduce the *connectivity* relation. We say e, e' are *connected* with respect to n if there *may* $\exists(a, o, r) \in (h \downarrow_g e), (a', o', r') \in (h \downarrow_g e')$ s.t. $o, o' \in (h \downarrow_g n) \wedge$ (o, o' are *related*). Otherwise we say the edges are *disjoint*.

To track the possibility that a single incoming edge e to the node n abstracts multiple references that reach the same object in the region abstracted by n we introduce the *interfere* relation. An edge e represents *interfering* pointers (*ip*) if there *may* $\exists(a, o, r), (a', o', r') \in (h \downarrow_g e)$ s.t. $(a, o, r) \neq (a', o', r') \wedge$ (o, o' are *related*). Otherwise we say the edge represents all *non-interfering* pointers (*np*).

Pictorial Representation. We represent abstract graphs pictorially as labeled, directed multi-graphs. Each node in the graph either represents a region of the heap or a variable. The variable nodes are labeled with the variable that they represent. The nodes representing the regions are represented as a record `[id type scalar linearity nodeCover]` that tracks the instrumentation relations for the object types (*type*), the simple scalar domain (*scalar*), the number of objects represented by the node (*linearity*, omitted when it is the default value 1), and the edge sets that cover the node (*nodeCover*).

Each edge contains a record that tracks additional information about the edge. The edges in the figures are represented as records {`id offset linearity interfere connto`}. The *offset* component indicates the offsets (abstract storage location) of the references that are abstracted by the edge. The number of references that this edge may represent is tracked with the *linearity* relation. The *interfere* relation tracks the possibility that the edge represents references that interfere. Finally, we have a field *connto* which is a list of all the other edges/variables that the edge may be connected to according to the *connected* relation. Again to simplify the figures we omit fields that are the default domain value (*linearity* = 1, *interfere* = *np*, *connto* = $\emptyset$).

Finally, we use a global equivalence relation on the edges which tracks the abstract edge equivalence relations (`EdgeEQ` in the figures).

5 Abstract Operations

We now define the most important and interesting dataflow transfer functions for the abstract graph domain, including how the reference set relations are updated. The domain operations are *safe* approximations of the concrete program operations. For brevity we omit proofs of these safety properties (which rely on simple case-wise reasoning about the graph structure and the instrumentation relations). For these algorithms we also assume that all the variables have unique targets (in practice this is done by creating one new abstract graph for each possible variable target, where in each new graph the variable of interest has a unique target).

[1] Two objects are weakly-connected if there is a (possibly non-empty) path between them (treating all edges as undirected).

5.1 Operations

Variable Nullity. When performing tests we generate one version of the abstract graph for each possible outcome. For the nullity test of a variable we create one abstract graph in which the variable *must* be *null* and one abstract graph in which the variable *must* be *non-null*. In the case where the variable is assumed to be *null* we are asserting that the concretization of the edge that represents the variable target is empty. Thus, if the variable edge covers ($\hat{\triangleright}$) a node we infer that the node does not represent any objects and all the other incoming edges must also have empty concretizations. Similarly any edge that is $\hat{\sim}$ to the edge representing the variable target must also have an empty concretization (and can be removed from the graph).

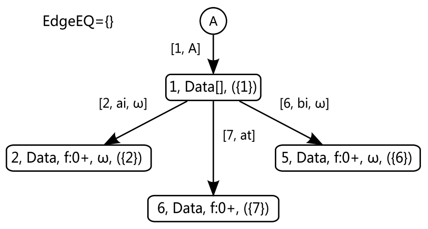
Algorithm 1: Assume Var Null (`v == null` is *true*)

```
input : graph $g$, var $v$
$e_v \leftarrow$ the edge representing the target of $v$;
$n \leftarrow$ the target node of $e_v$;
if $e_v \hat{\triangleright} n$ then $E_{\text{null}} \leftarrow$ {all incoming edges to $n$};
else $E_{\text{null}} \leftarrow \{e' | e' \hat{\sim} e_v\}$;
for edge $e \in E_{null}$ do
    $g$.removeEdge($e$);
```

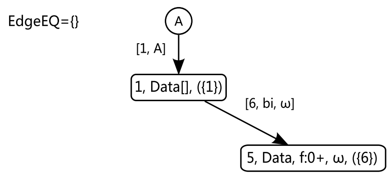
Indexing Bounds. In order to analyze nontrivial programs that manipulate arrays and collections we must be able to accurately model the effects of programs that use integer indexed loops to traverse them. To do this we use several special names for the edges that represent the pointers stored in arrays/collections. The name *?* indicates elements at arbitrary indices in an array when it is not being indexed through, *at* represents the distinct element at the index given by the indexing variable, *bi* represents all the elements stored at indices less than the indexing variable, and *ai* represents all the elements stored at indices greater than the indexing variable.

In order to simulate the effect of the test, `i < A.Length`, we again create two new abstract graphs, one where the test result is *true* and one where the test result is *false*. The true result does not provide any additional information that is applicable in our heap domain so we do not need to do anything. The false result indicates that the indexing variable now refers to an index larger than the array size. This implies that there are no elements stored at indices equal to or greater than the current value of the indexing variable, which means that the edges with *offsets at* and *ai* must have empty concretizations and can be eliminated from the abstract graph. Further, as with the variable nullity test we can use the reference set relation information to eliminate other edges and nodes that must also have empty concretizations.

Figure 4(a) shows the most general abstract heap that arises when using simple integer indexing in a loop (to focus on the loop indexing we assume the body is empty) to traverse an array as initialized in lines 3-4. In this figure we have three outgoing edges from node 1, the edge with offset *bi* (edge 6) which represents all the elements at indices less than `i` (elements that have been processed), the edge with the offset *at* (edge 7) which represents the single element stored at index `i` (the element currently being processed), and the edge with offset *ai* (edge 2) which represents all of the elements at indices greater than `i` (elements not yet processed).



(a) `for(i = 0; i < A.Len; ++i)`



(b) `i < A.Len` is False

Fig. 4. Integer Indexing and Test

Figure 4(b) shows the abstract graph that results from assuming the test, `i < A.Length`, is false. In this figure the analysis has determined that since the indexing variable (`i`) is off the end of the array all of the elements in the array must be stored at indices less than `i` and that edges 2, 7 have empty concretizations. This allows the analysis to remove them and since these edges *cover* ($\hat{\triangleright}$) nodes 2, 6 respectively we can infer that these nodes have empty concretizations and can be removed as well.

Load. The field load operation (`x = y.f`) first computes which node is the target of the expression `y.f`, creating a more explicit representation as needed (Subsection 5.2). Then it adds an edge from `x` to this node and if the storage location of `y.f` is unique then we know the target of `x` must be equal to the target of `y.f` (and the edges representing them are $\hat{\sim}$ and have the same $\hat{\triangleright}$ properties).

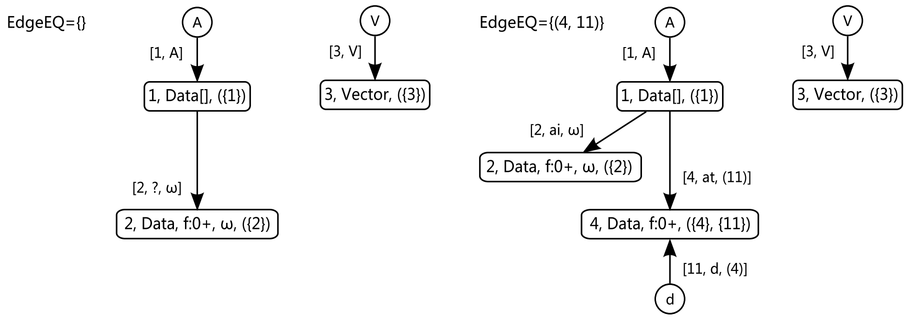
Algorithm 2: Load (`x = y.f`)

```
input : graph $g$, var $x$, var $y$, field $f$
nullify $x$;
if $y.f \neq$ null then
    $g$.materialize(the unique target of $y.f$);
    $n \leftarrow$ target node of $y$;
    $e \leftarrow$ the unique edge at $y.f$;
    assign $x$ to refer to the target of $e$;
    if $n.linearity = 1$ then
        $n' \leftarrow$ the target node of $e$;
        set edge representing $x \hat{\sim}$ to $e$;
        if $e \hat{\triangleright} n'$ then set edge representing $x \hat{\triangleright} n'$;
```

5.2 Materialization

The materialization operation [13] is used to transform single summary nodes into more explicit subgraph representations. For the example in this paper we only need a simple version of *Singleton* materialization which is restricted to handle the following case and otherwise conservatively leave the summary region as it is: if the incoming edges can be partitioned into two or more equivalence classes based on the *connected* instrumentation relation. Once we have identified a node and the edge partitions we create a new node for each partition.

Figure 5(a) shows the heap abstract graph that captures all of the possible states at line 4 of the example program. The variable `A` refers to a node with the identifier 1, which represents a `Data[]` array, and we know it represents at most one array (the default omitted *linearity* value of 1). This array may have multiple pointers stored in it, represented by the *linearity* value ω in the edge with id 2. Each of these pointers refers to a unique `Data` object since the edge has the omitted default *interfere* value of *np*. The `f:0+` entry indicates that all objects abstracted by node 2 have values in the range $[0,\infty)$ in their `f` fields. Finally, based on the $\{2\}$ entry of the *nodeCover* set for the node 2, we know that each object is referred to by a pointer abstracted by edge 2.



(a) Result From Initialization Loop (b) Load of $A[i]$ (when i = 0)

Fig. 5. Load of $A[0]$ on result of first loop

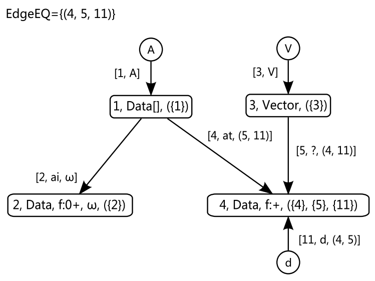
The result of the load, `d = A[i]` when $i = 0$ during the analysis of the first iteration of the filter loop (line 6), is shown in Figure 5(b). In this figure we have split edge 2 from Figure 5(a) into two edges, one representing the pointer stored at index 0 (edge 4, with offset `at`) and one representing all the pointers stored at indices $[1,\infty)$ (edge 2, with offset `ai`). We have also split the node which represents the `Data` objects into node 4 representing the object targeted by the pointer in `A[0]` and node 2 representing the objects targeted by the pointers stored at the other indices in the array.

Since we know that the edge that was split (edge 2) $\hat{\triangleright}$ the node that was split (node 2) we know that the resulting edges in Figure 5(b) must $\hat{\triangleright}$ the resulting nodes (edge 2 $\hat{\triangleright}$ node 2 and edge 4 $\hat{\triangleright}$ the node 4). Further we know that edge 4 represents a single pointer (it represents the single pointer at `A[0]`) and, since it $\hat{\triangleright}$ node 4, that node must represent at most one object (the default omitted *linearity* value of 1).

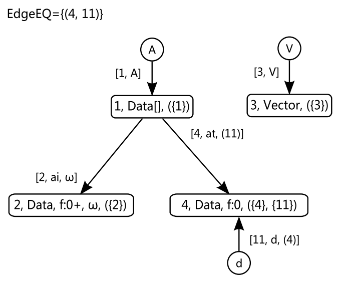
Finally, we have set the target of the variable `d` to be the same as the target of the edge that represents the pointers stored in `A[0]`. Based on the load algorithm we set the new edge (edge 11) to be $\hat{\sim}$ to edge 4 and since edge 4 $\hat{\triangleright}$ node 4, we know that edge 11 $\hat{\triangleright}$ node 4 as well.

6 Examples

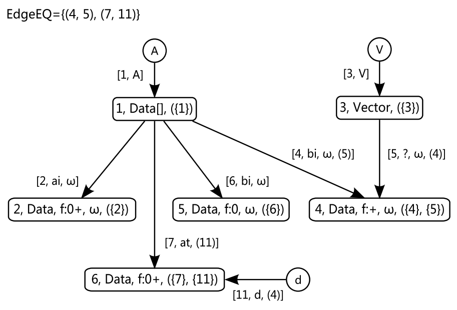
Filter Loop Example. The filter loop (lines 5-7) demonstrates how the analysis uses reference set information and the control flow predicate (`d.f > 0`) to infer additional information about the heap, in particular that the set of objects stored in `V` *must* equal the set of objects with positive `f` fields in `A`. To simulate the effect of the test (`d.f > 0`) on the state of the program we create two abstract graphs, one for the result when test result is true and one when the test result is false.



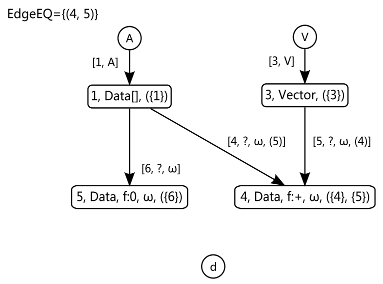
(a) Assert Test is True



(b) Assert Test is False



(c) Fixed Point of Loop Analysis



(d) `i < A.Length` False, Exit Loop

Fig. 6. Filter Loop Analysis

Figure 6(a) shows the abstract graph that results from assuming that the test `d.f > 0` is *true* (on the first iteration of the loop, $i = 0$) and the entry is added to the `Vector` `V`. Since the test succeeds and we know `d` must refer to the single object abstracted by node 11 (default omitted *linearity* value of 1) we can update the scalar information to show that the `f` field must be greater than 0 (the `f:+` label). We have updated the graph structure by adding the edge 5 to represent the pointer that is stored into the vector object. Since we know this pointer refers to the same object as `d`, which is represented by edge 4, we add the entry (4, 5) to the *EdgeEQ* relation and since edge 4 $\hat{\triangleright}$ node 4 we know that edge 5 also $\hat{\triangleright}$ node 4.

Figure 6(b) shows the abstract graph that results from assuming that the test `d.f > 0` is *false* (on the first iteration of the loop, $i = 0$) and the entry is not added to `V`. Since the test fails and again we know `A[i]` refers to a single object we update the scalar information to show that the `f` field must equal to 0 (the `f:0` label).

Figure 6(c) shows the fixed point abstract graph which represents all the states that are generated in the loop. We see that there may be many elements in the vector `V` and many elements that are not added to the vector (represented by the edges with the `bi` labels, 4 and 6 respectively). Since we tracked the $\hat{\triangleright}$ relation of each individual object as it was processed we know that every object referred to by a pointer represented by edge 4 must have been added to the vector `V` and thus is also referred to by a pointer represented by edge 5. This implies that edge 5 $\hat{\sim}$ edge 4 and both edge 4 $\hat{\triangleright}$ node 4 and edge 5 $\hat{\triangleright}$ node 4.

Figure 6(d) shows the result of assuming that `i < A.Length` returns *false*. The `at` and `ai` edges (edges 7, 2) must have empty concretizations and can be eliminated (as they abstract the pointer stored at index `i` and pointers stored at indices larger than `i`). As desired the analysis has determined that all the objects with a non-zero `f` field have been stored in the vector `V` (since node 5 only abstracts objects with 0 in the `f` field and edge 4 $\hat{\sim}$ edge 5).

Update Loop Example. For brevity we omit descriptions of how the reference set information is propagated during the individual operations of the update loop (lines 9-11) and focus on how this information is used to improve the precision of the analysis results at the loop exit. The fixed point abstract graph for the loop body is shown in Figure 7(a). In this figure we see that the there are potentially many pointers that come before the current index position in the vector `V` (edge 10 with *offset* `bi`, all of which point to objects with 0 in the `f` field). It also indicates that the edges representing the current index location (edge 8 with *offset* `at`) and the set of pointers that come after the current index position (edge 5 with *offset* `ai`) cover ($\hat{\triangleright}$) their target nodes (nodes 4, 8).

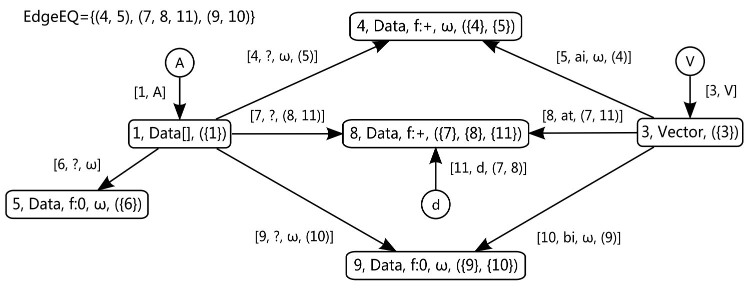
If the exit test (`i < V.size()`) is *false* then we can infer that there are no entries in the vector at indices that are greater than or equal to `i`. This implies that the edges `at` and `ai` (edges 8, 5) have empty concretizations since they represent pointers stored at indices greater than or equal to `i`. Based on the $\hat{\sim}$ relations (4, 5) and (7, 8, 11) this implies that edges 4, 7 and 11 have empty concretizations as well.

The result of this inference is shown in Figure 7(b). After the test (and the removal of the edges/nodes) there are no longer any pointers to objects with non-zero `f` fields in the vector `V` or the array `A`. Thus, the loop has successfully determined that all the objects in the vector `V` must be updated and further, this update information has been reflected in the original array `A` (i.e., there is no object in `A` that had a non-zero field that was not updated in the loop). As desired the analysis has determined that all of the objects in the array `A` have the value 0 stored in their `f` fields after the filter/map loops.

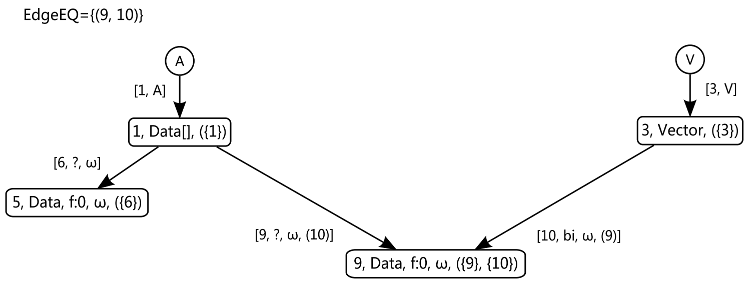
7 Experimental Evaluation

We have implemented a shape analyzer based on the instrumentation relations and reference set information presented in this paper. We use a number of benchmarks[2] from our version of the Jolden suite [7], two programs from SPECjvm98 [14], and two programs (exp and interpreter) written as challenge problems. The JOlden suite contains

[2] Benchmark/Analysis code is available at `www.software.imdea.org/~marron/`.



(a) Fixed Point Update of Loop



(b) After Loop Exit

Fig. 7. Fixpoint and Exit of Map Loop

pointer-intensive kernels (taken from high performance computing applications). We have modified the suite to use modern Java programming idioms. The benchmarks raytrace and db are taken from SPECjvm98 (with minor modifications to remove test harness code and threading).

Benchmarks *exp* and *interpreter*, our two internally developed benchmarks, are a basic arithmetic expression evaluator and an interpreter for the computational core of Java. The exp program contains a variety of heap analysis challenges (non-trivial heap structures with and without sharing, copy traversals of the structures and destructive traversals of the structures), and is still small enough to understand. The interpreter program is a large program with varied heap structures, from a large well defined tree structure in the AST, symbol and local variable tables, a call stack of pending call frames, and a very poorly defined cyclic structure in the internal model of the heap built by the interpreter (thus the heap analysis must be both precise and able to deal with ambiguity efficiently). It also has substantial amounts of sharing (variables, method signatures and objects on the interpreters internal representation of the heap are shared in multiple structures). Because of these characteristics we believe these programs are excellent challenge problems for this area of research.

The analysis algorithm was written in C++ and compiled using MSVC 8.0. The analysis was run on a 2.6 GHz Intel quad-core machine with 2 GB of RAM (although memory consumption never exceeded 150 MB). Detailed information on the interprocedural dataflow analysis methods used can be found in [10].

We compare the analysis results when using the *reference set relations* described in this paper and when using a basic equivalence-based field-sensitive must points to

Benchmark	LOC	Alias Time	Ref. Time
em3d	1103	0.09s	0.11s
health	1269	1.55s	1.87s
bh	2304	0.72s	0.91s
db	1985	0.68s	1.07s
raytrace	5809	15.5s	15.9s
exp	3567	152.3	161.8
interpreter	15293	114.8	119.3

Fig. 8. Alias Time reports the analysis time with basic *must-alias* tracking while Ref. Time reports the analysis time using *reference set relations*. LOC is for the normalized program representation including library stubs required by the analysis.

relation on the abstract graph edges. In each of these benchmarks when using the reference set relations we see a moderate increase in runtime which varies based on the quantity of subset relations generated by the program (with the largest increase in db, which represents an in-memory database and views of this database via arrays). Each of these benchmarks possess some instances of data structures where the use of reference set relations allows the analysis to extract information that was not possible with simple aliasing information. In some cases this information is not particularly useful (in em3d the analysis discovers that the there are 2 vectors each of which refers to every element in one of the halves of a bipartite graph). However, in most of the programs the reference set information provides potentially valuable information. For example, in bh the analysis discovers that the leaves of the space decomposition trees are always a subset of a given vector, in db we know the set of entries in each view is a subset of the entire database, and in interpreter the analysis determines that each variable symbol is interned in a special table and that all live stack frame objects must be stored in a single list container. In addition the analysis is able to precisely model (as in the running example) most of the filter-map and subset-remove type loops that occur.

8 Conclusion

In this paper we introduced reference set relations, a novel concrete heap property that subsumes the concept of must-aliasing and allows us to compactly express a wide range of must-sharing relations (*must*-$=$ and *must*-$\subseteq$) between arrays, collections, and heap data structures. By extending an existing shape analysis with two simple relations to track the most commonly occurring reference set relations (equality via the *abstract edge equivalence* property, $\hat{\sim}$, and subset relations, indirectly, via the *abstract node cover* property and $\hat{\triangleright}$) we can model many useful sharing properties. As demonstrated by the experimental evaluation, this approach has a small impact on computational costs when compared with classic *must-aliasing* and allows the tracking of a much richer set of heap sharing properties. This work also highlights the strength of the labeled storage shape graph approach, which partitions the heap into conceptually homogeneous regions. This partitioning enables even relatively simple concepts such as the *reference set relations* presented in this work to extract rich information from the program (and conversely may enable the efficient use of strong decision procedures by limiting the complexity of the verification conditions encountered during program analysis).

Acknowledgments. he first author was supported by EU FP7 NoE S-Cube 215483 and FET IST-231620 *HATS*, Spanish MICINN project 2008-05624/TIN *DOVES* and Madrid Regional project S-0505/TIC/0407 *PROMESAS*. The second author was supported in part by the NSF grants CCF-0546170 and CCF-0702743. This material is based upon work supported by the National Science Foundation under grant CCF-0540600. This research is supported in part by the National Science Foundation under grants CCF-0540600, CCF-0546170, and CCF-0702743. The authors thank Mooly Sagiv and Roman Manevich for their useful comments on a preliminary version of this work.

References

1. 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.
2. D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.
3. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.
4. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
5. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
6. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
7. Jolden Suite. http://www-ali.cs.umass.edu/DaCapo/benchmarks.html.
8. V. Kuncak and M. C. Rinard. Decision procedures for set-valued fields. *Proc. Abstract Interpretation of Object-Oriented Languages*, 2005.
9. T. Lev-Ami, N. Immerman, and S. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.
10. M. Marron, O. Lhotak, and A. Banerjee. Call-site heuristics for scalable context-sensitive interprocedural analysis. Report at: `www.software.imdea.org/~marron/`, July 2009.
11. M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE*, 2008.
12. M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.
13. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.
14. Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.
15. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
16. T. Wies, V. Kuncak, P. Lam, A. Podelski, and M. C. Rinard. Field constraint analysis. In *VMCAI*, 2006.
17. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.
18. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *CAV*, 2008.
19. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *PLDI*, 2008.