Shape Analysis with Reference Set Dominance

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

[1]University of New Mexico, {marron, darko, kapur}@cs.unm.edu
[2]University of California Los Angeles, rupak@cs.ucla.edu

Abstract. Precise modeling of the structure of the heap and how objects are shared between various arrays or data structures is fundamental to understanding the behavior of a program. This paper introduces a novel higher order relation, *reference set dominance*, which subsumes the concept of 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 set dominance*. This concept allows the analysis to track a number of important relations (*must* $=$, and *must* $\subseteq$) between the sets of objects that are the targets of two given sets of references (variables or pointers).
In combination with shape properties, an analysis that tracks reference dominance information can precisely reason about sharing properties on the heap (are the contents of one array a subset of another array?), and how sharing influences the results of destructive updates (does modifying all the objects in one array imply that all the objects in another array are modified as well?). Precisely and efficiently reasoning about these kinds of sharing properties has been beyond the abilities of previous analyses. We show that shape analysis augmented with dominance information is able to precisely model sharing for a large range of data structures in real programs and in contrast to more expressive proposals based on logic languages (e.g., extensions of first-order predicate logic with transitive closure), dominance properties can be efficiently implemented 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 [17]), are known to be too imprecise for many applications. More precise *shape analysis* techniques [1,6–9,12,15,18,19] 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 kind of information, these analyses can precisely model recursive data structures [6, 8,12,15] and composite structures [1,7,12,19] (such as a list of lists or a tree containing pointers to user–defined multi–component objects).

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 pattern 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 objects (for efficiency a class might keep the same set of objects in a *Vector* as well as in a *Hashtable*), and in algorithms that extract and manipulate sub-collections (in *mergesort* with auxiliary storage, the contents of the two partitions are disjoint and their union is the same as the original list) and when performing updates on a set of shared elements (if the array A contains the same set of objects as the array B then applying the function f to every element in A implies that f *must* also have been applied to every element in B).

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

We have implemented a shape analysis framework with dominance information and evaluated the effectiveness of the dominance relations on well known Java benchmarks that manipulate shared heap structures. Our benchmark set consists of a version of the JOlden [3] benchmarks, and several benchmarks from the SPECjvm98 [16] suite, for which we recover precise shape information within less than a minute of running time.

Sharing relations between sets of objects, including dominance relations, can be simulated in general logic languages 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?) along with support for disjunction of reachability relations. However, the introduction of general quantification and disjunction makes abstract reasoning computationally expensive [2]. Instead, as demonstrated in this paper, these properties can be efficiently tracked on top of existing shape analysis techniques with enough accuracy to prove many important sharing properties.

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. The example uses a dummy object of type `Data` which has a single integer field `f`.

The first code fragment allocates an array `A` then fills it with `Data` objects which have random non–negative values stored in their `f` fields. The second loop scans the array for elements that have positive values in the `f` fields and constructs a new `Vector` of these elements. If these loops are analyzed using a standard shape analysis (described in Section 5), we get the abstract heap graph shown in Figure 2(a). In this figure we 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.

```
1 Data[] A = new Data[N];
2 for(int i = 0; i < N; ++i)
3   A[i] = new Data(abs(randInt()));
4 Vector V = new Vector();
5 for(int i = 0; i < A.Length; ++i)
6   if(A[i].f > 0) V.add(A[i]);
7 for(int i = 0; i < V.size(); ++i)
8   V.get(i).f = 0;
```

Fig. 1. Initialize Array, Filter positive values, and Update `f` fields

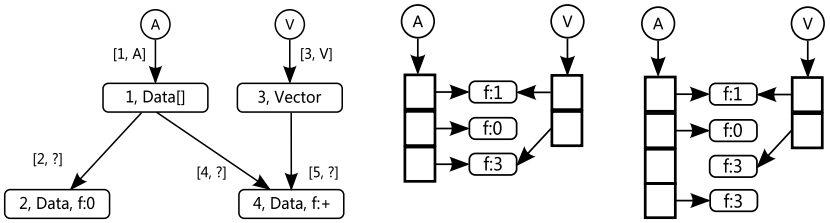


Fig. 2. Abstract model and two possible concrete heaps it represents.

The simplified model shows the variable `A` referring to a node with an *id* tag of 1 which abstracts an object of type `Data[]`. There may be many pointers stored in this array (these pointers are abstracted by the edges with the *id*'s 2, 4) since these pointers are stored in an arrays we give them the special storage *offset* `?`. Since there are two outgoing edges 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` and `f:+` indicates the values of the integer fields using a simple *sign* domain [5], 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. 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 these pointers refer to objects abstracted by node 4.

Based on this information either of the concrete heaps shown in Figure 2(b) or 2(c) are consistent with this model (i.e., they are valid concretizations). In Figure 2(b) we see that array `A` contains 3 `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 but this object is not in `V`. However, this program model 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 order to precisely represent these desired *must* sharing relations between various sets of pointers stored in the array and vector we need to extend the model with additional information.

While there are shape analysis techniques [7] that can determine, using reachability and connectivity information, that after the execution of the first two loops the `Data`

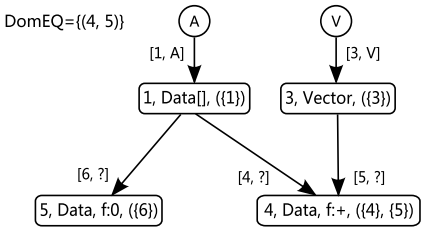


Fig. 3. Model With Dominance Information

objects in the vector `V` are a subset of the objects in the array they cannot model the stronger property; that every `Data` object with a *positive* `f` field in the array *must* also be in the vector `V`.

The second property is critical to the ability to determine the effect of the third code fragment, where all of the elements in the vector `V` have their `f` field set to 0. If the analysis was unable to determine that every object in `A` with a non–zero `f` field *must* be in the vector `V` then it will be unable to determine that setting the `f` field of every element in `V` to 0 implies that the `f` field of every object in `A` *must* also be 0.

To accomplish this our analysis tracks *dominance equivalence* relations on the heap (in addition to connectivity and shape properties). That is, we track if two edges abstract sets of references that *must* always refer to exactly the same set of objects and given a node which sets of edges abstract references such that every object represented by the node is referred to by a reference in this set.

These additional dominance properties allow the analysis to precisely model the result of the construction and filter loops in our example, the model enhanced with the dominance properties is shown in Figure 3. In Section 4 we define and add two dominance relations to the model. The `DomEQ` relation which tracks which edges abstract sets of references that always refer to the same sets of objects, and for each node we add a list of sets of edges such that every object abstracted by the node is referred to by a references 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 implies 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 either has the value 0 stored in the `f` field or it is also 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. A detailed example of how the analysis of the update loop and the subset computation is accomplished is in Section 6.

3 Concrete Heap and Reference Set Dominance

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 en-

vironment and the store together as the concrete heap, which is treated as a labeled, directed multi–graph (V,O,R) where each $v \in V$ is a variable, each $o \in O$ is an object on the heap and each $r \in R$ is a reference (either a variable reference or a pointer between objects). The set of references $R \subseteq (V \cup O) \times O \times L$ 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). For a reference $(a,b,p) \in R$, we use the notation $a \xrightarrow{p} b$ to indicate that the object (or variable) a refers to b via the field name (or variable identifier) p.

A *region* of memory $\mathfrak{R} = (C,P,R_{in},R_{out})$ consists of a subset $C \subseteq O$ of the objects in the heap, all the pointers $P = \{(a,b,p) \in R \mid a,b \in C \wedge a \xrightarrow{p} b\}$ that connect these objects, the references that enter the region $R_{in} = \{(a,b,r) \in R \mid a \in (O \setminus C) \wedge b \in C \wedge a \xrightarrow{r} b\}$ and similarly a set of references that exit the region $R_{out} = \{(a,b,r) \in R \mid a \in C \wedge b \in (O \setminus C) \wedge a \xrightarrow{r} b\}$.

Definition 1 (Reference Set Dominance). *Let $\mathfrak{R} = (C,P,R_{\text{in}},R_{\text{out}})$ be a region, and let $R_s \subseteq R_{\text{in}}$ and $R'_s \subseteq R_{\text{in}}$. We define the* targets *of these reference sets, $T_s = \{o \in C \mid \exists a \in O, r \in L \text{ s.t. } (a,o,r) \in R_s\}$ and $T'_s = \{o' \in C \mid \exists a' \in O, r' \in L \text{ s.t. } (a',o',r') \in R'_s\}$. We define the following relations:*

1. R_s dominates R'_s *if* $T'_s \subseteq T_s$.
2. R_s, R'_s *are* dominance equal *if* $T_s = T'_s$.
3. R_s region dominates $\mathfrak{R}$ *if* $C \subseteq T_s$.

We note that if R_s *region dominates* $\mathfrak{R}$, where $\mathfrak{R} = (C,P,R_{in},R_{out})$, then for any $R'_s \subseteq R_{in}$ then R_s *dominates* R'_s. Which is useful later for compactly representing simple (and common) subset relations.

4 Abstract Dominance

Our abstract heap domain is based on the *storage shape graph* [4] approach. An *abstract storage graph* is a tuple of the form $(\hat{V},\hat{N},\hat{E})$, where $\hat{V}$ is a set of abstract nodes representing the variables, $\hat{N}$ is a set of abstract nodes (each of which abstracts a region $\mathfrak{R}$ 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 abstracts a set of pointers, and $\hat{L}$ is a set of abstract storage *offsets* (variable names, field offsets or the special offset *?* for references stored in arrays/collections). We extend this definition with a set of additional relations $\hat{U}$ that further restrict the set of concrete heaps that each *storage shape graph* abstracts. The *labeled storage shape graphs (lssg)*, which we refer to simply as *abstract graphs*, are tuples of the form $(\hat{V},\hat{N},\hat{E},\hat{U})$.

Definition 2 (Valid Concretization of a *lssg*). *A given concrete heap h is a* valid concretization *of a* labeled storage shape graph *g if there are functions Π_v, Π_o, Π_r such that the following hold:*

- *$\Pi_v : V \mapsto \hat{V}$ and is 1–1, $\Pi_o : O \mapsto \hat{N}$ and $\Pi_r : R \mapsto \hat{E}$ are functions.*
- *h, Π_v, Π_o, Π_r satisfy all the relations in $\hat{U}$.*
- *h, Π_v, Π_o, Π_r are* connectively consistent *with g.*

Where h, Π_v, Π_o, Π_r *are* connectively consistent *with* g *if:*

- $\forall\, o_1, o_2 \in O$ *s.t.* $(o_1, o_2, p) \in R$, $\exists\, e \in \hat{E}$ *s.t.* $e = \Pi_r((o_1, o_2, p))$, e *starts at* $\Pi_o(o_1)$, *ends at* $\Pi_o(o_2)$ *and* $e.\text{offset} = p$.
- $\forall\, v \in V$, $o \in O$ *s.t.* $(v, o, v) \in R$, $\exists\, e \in \hat{E}$ *s.t.* $e = \Pi_r((v, o, v))$, e *starts at* $\Pi_v(v)$, *ends at* $\Pi_o(o)$ *and* $e.\text{offset} = v$.

In Section 2 we implicitly introduced several relations, *type* and *sign*, which are relations on the nodes in the abstract graph. To check if a given concrete heap h and maps Π_v, Π_o, Π_r satisfy these relations (and the other properties we use) we need to look at the pre–images of the nodes and edges in the abstract graph g under the maps Π_v, Π_o, Π_r. We use the notation $h \downarrow_g e$ to indicate the set of references in the concrete heap h that are in the pre–image of e under the maps and similarly, $h \downarrow_g n$, to indicate the region of the heap that is the image of n under the maps.

As an example for the *type* relation, we add a relation $(n, \{\tau_1, \ldots, \tau_k\})$ to $\hat{U}$ for each node in $\hat{N}$, where τ_j are types in the program and say: h, Π_v, Π_o, Π_r satisfies $(n, \{\tau_1, \ldots, \tau_k\})$ iff $\{\texttt{typeof}(o) \mid \text{object } o \in h \downarrow_g n\} \subseteq \{\tau_1, \ldots, \tau_k\}$. The *sign* relation can be similarly defined and a number of other useful properties are added in Section 5.

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

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

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

Proposition 1. *Let* $g = (\hat{V}, \hat{N}, \hat{E}, \hat{U})$ *be an abstract heap,* h *a valid concretization of* g, $n, n' \in \hat{N}$ *be abstract nodes,* $e, e' \in \hat{E}$ *abstract edges, and* $E, E' \subseteq \hat{E}$ *sets of abstract edges.*

1. *If* $\{e\} \propto n$ *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* $e \equiv_{dom} e'$ *and* $h \downarrow_g e = \emptyset$ *then* $h \downarrow_g e' = \emptyset$.
3. *If* $\{e\} \propto n$ *and* $\{e'\} \propto n$ *then* $e \equiv_{dom} e'$.
4. *If* $\{e\} \propto n$ *and* $|h \downarrow_g e| = 1$ *then* $|h \downarrow_g n| = 1$.

We have restricted the definition of the *abstract dominance* relations to equality of edges plus a special relation on nodes. We could define a more general relation, where subset relations between the targets of sets of edges are tracked. However, this later formulation would require tracking a binary relation on the power set of $\hat{E}$ which is undesirable from a computational standpoint.

5 Full Abstract Heap Graph Definition

In order to analyze programs with the desired level of precision we need to use a number of other relations that have been introduced in previous work [12–14] to the graph model.

5.1 Additional Instrumentation Properties and Relations

In addition to tracking edge and node dominance relations, the nodes and edges of storage graphs are augmented with the following instrumentation relations.

Linearity. To model the number of objects abstracted by a given node (or references by an edge) we use a *linearity* property which has two possible values: 1, which indicates that the node (edge) concretizes to either 0 or 1 objects (references), and the value ω, which indicates that the node (edge) concretizes to any number of objects (references) in the range $[0,\infty)$.

Abstract Layout. To approximate the shape of the region a node abstracts, the analysis uses the *abstract layout* properties $\{$*(S)ingleton, (L)ist, (T)ree, (M)ultiPath, (C)ycle*$\}$. The *(S)ingleton* property states that there are no pointers between any of the objects abstracted by the node. The *(L)ist* property states each object has at most one pointer to another object in the region. The other properties correspond to the standard definitions for trees, DAGs, and cycles.

Connectivity and Interference. The heap model uses two 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 [13] for a more extensive description of these relations.

Given a concrete region $\Re = (C, P, R_{in}, R_{out})$ and objects $a, b \in C$, we say a and b 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 the same object in the region abstracted by n we introduce the *connectivity* relation. We say e, e' are *connected* with respect to n if there *may* $\exists (a,b,r) \in (h \downarrow_g e), (a',b',r') \in (h \downarrow_g e')$ s.t. $b, b' \in (h \downarrow_g n) \wedge$ (b, b' 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* $\exists (a,b,r), (a',b',r') \in (h \downarrow_g e)$ s.t. $(a,b,r) \neq (a',b',r') \wedge$ (b, b' are *related*). Otherwise we say the edge represents all *non–interfering* pointers (*np*).

5.2 Heap 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 layout linearity nodeDom]` that tracks the instrumentation properties for the object types (*type*), the simple scalar domain (*scalar*), the *layout*, the number of objects represented by the node (*linearity*), and the edge sets that dominate the node (*nodeDom*). To simplify the figures we omit fields from the labels when they are the default domain values (*layout* = *S*, *linearity* = 1).

As in the case of the nodes, each edge contains a record that tracks additional information about the edge. The edges in the figures are represented as records {`id`

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

`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 dominance equivalence relations (`DomEQ` in the figures).

The local data flow analysis is performed using a *disjunctive power domain* [10] over these graphs. Interprocedural analysis is performed in a context-sensitive manner and the procedure analysis results are memoized. At each call site the set of graphs is joined into a single graph and the call is analyzed using this graph as the context; see [11] for more details.

5.3 Abstract Operations

We now define the dataflow transfer functions for our abstract graph domain, including how the dominance information is updated. The domain operations are *safe* approximations of the concrete program operations. For brevity we omit proofs of these safety properties (which rely on straight forward case–wise reasoning about the graph structure and the instrumentation relations/properties). For these algorithms we also assume that all the variables have unique targets (in practice this is done by creating a new model for each possible variable target as needed).

Tests: 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 model in which the variable *must* be *null* and one model 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 dominates a node we infer that the node does not represent any live objects and all the other incoming/outgoing edges must also have empty concretizations. Similarly any edge that is $\equiv_{\text{dom}}$ to the variable must also have an empty concretization (and can be removed from the graph). Algorithm 1 gives the code for this operation.

In the case of an equality comparison (`x == y`) between two variables which *may* be *non-null* we can strengthen the information we have in the models that represent the true and false branches. In the case where we assume the test returns true, if `x` and `y` refer to different nodes in the graph then the only way they can be equal is if both variables are `null`, otherwise we add the fact that `x` $\equiv_{\text{dom}}$ `y` to the model. In the case where we assume the test must be false then we can check if the relation `x` $\equiv_{\text{dom}}$ `y` and if it does we can rule this path out as being infeasible.

Variable Assignment: The variable assignment operation (`x = y`) does not need to perform any complex manipulations on the heap since these operations were done during previous analysis steps as needed. Thus, we simply update the target node and dominance information for the variable `x` to be the same as the variable `y`.

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 $e_v \propto n$ **then**
    $E_{\text{null}} \leftarrow \{\text{all incoming edges to } n\}$;
else
    $E_{\text{null}} \leftarrow \{e_v\} \cup \{e' | e' \equiv_{\text{dom}} e_v\}$;
for *edge* $e \in E_{null}$ **do**
    g.removeEdge(e);

Load: The 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 5.4). Then it adds an edge from `x` to this node and if the target of `y` is unique (represents a single unique object on the heap) then we know the target of `x` must be equal to the target of `y.f`.

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 $n.linearity = 1 \wedge e.linearity = 1$ **then**
        set x dominance equal to e;
        if *e dominates the target node* **then** set x dominates target node of e;

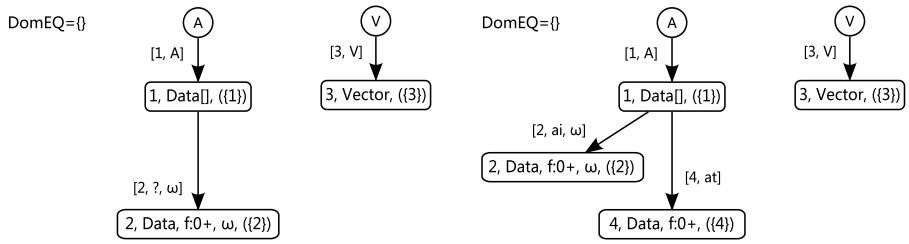
Store: The store operation (`x.f = y`) begins by determining if the location at `x.f` can be strongly updated. If the node (n) referred to by `x` is of *linearity* 1 then it is possible to do so. In this case there is at most one object that is live in the target region so either `x` points to this object or `x` is *null*, since we assume the latter case cannot happen (it would be a *null* pointer dereference) we know `x` refers to the single object that the node represents. Thus, we can erase any edges with the `f` field. Once we have completed the testing and removal of any edges stored at the field `f` we create a new edge representing the newly stored pointer (which refers to the same thing as `y`). Just as in the case of the loads the new edge is dominance equal to `y` and if `y` dominates a node then so does the newly created edge.

5.4 Materialization

The materialization operation is used to transform single summary nodes into more explicit subgraph representations. The operation is defined for nodes with *List*, *Tree*, or *Singleton* layouts and is further restricted based on the number of incoming edges to the node and the connectivity relations of these edges. For the example this paper we only need a simple version of *Singleton* materialization. Full definitions for the other operations can be found in [12].

Our materialization operation on *Singleton* nodes and 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 2 or more equivalence classes based on the *connected* relation. Once we have identified a node and the edge partitions we create a new node for each partition.

When iterating through arrays/collections with an integer variable, `i`, we split the edges with the *?* offset (which represent the contents of the array/collection) into multiple edges with three special *offsets* that partition the pointers based on their position relative to the index variable `i`. The offset `at` (at index) is used for the edge which represents the single reference stored in index `i`. The offset `bi` (before index) is used for edges that represent pointers stored in indices less than `i`. Finally, the offset `ai` (after index) is used for edges that represent pointers stored in indices greater than `i`.



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

Fig. 4. Load of $A[0]$ on result of first loop

Figure 4(a) shows the heap model 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 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 *nodeDom* set for the node 2, we know that each object abstracted by node 2 is referred to by a pointer abstracted by edge 2.

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

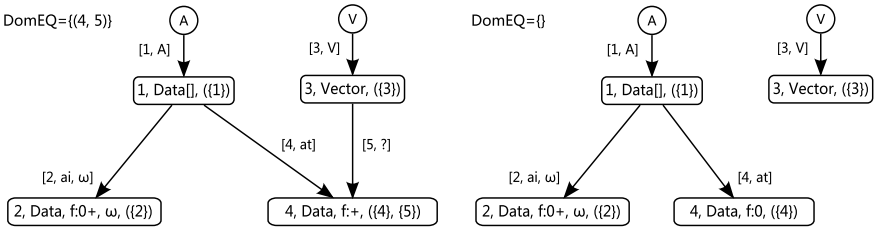
2, with offset `ai`). We have also split the node which represents the `Data` objects into a node representing the object targeted by the pointer in index 0 (node 4) and a node representing the objects targeted by the pointers stored in the other indices (node 2).

Since we know that the edge that split edge dominated the node that was split we know that the resulting edges in Figure 4(b) must dominate the resulting nodes (edge 2 dominates the node it refers to and edge 4 dominates the node it refers to). Further we know that edge 4 represents a single pointer (it represents the single pointer at `A[i]`) and, since it dominates the node it refers to, that node must represent at most one object (the default omitted *linearity* value of 1).

6 Examples

6.1 Filter Loop Example

The filter loop (lines 5-6) demonstrates how the analysis uses dominance information and the control flow predicate (`A[i].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`.



(a) Assert Test is True (b) Assert Test is False

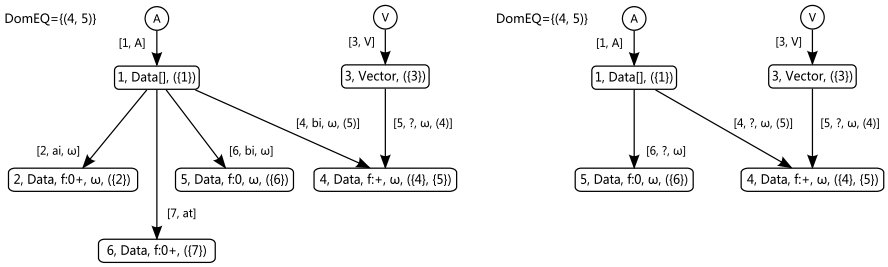
Fig. 5. True and False Conditional Results

In Figure 4(b) we show the result of evaluating the expression `A[i]` when `i` $= 0$. To simulate the effect of the test (`A[i].f > 0`) on the state of the program we create two new models; one for when the condition is true and one for when the condition is false.

Figure 5(a) shows the heap model that results from assuming that the test `A[i].f > 0` is *true* and the entry is added to the `Vector V`. Since the test succeeds and we know `A[i]` refers to a single object (the node has the 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 structural information 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 `A[i]`, which is represented by edge 4, we add the entry (4, 5) to the *DomEQ* relation and since edge 4 dominates node 4 we also know that edge 5 also dominates node 4.

Figure 5(b) shows the heap model that results from assuming that the test `A[i].f > 0` is *false* 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).



(a) Fixed Point of Loop Analysis (b) `i < A.Length` False, Exit Loop

Fig. 6. Fixpoint and Exit of Filter Loop

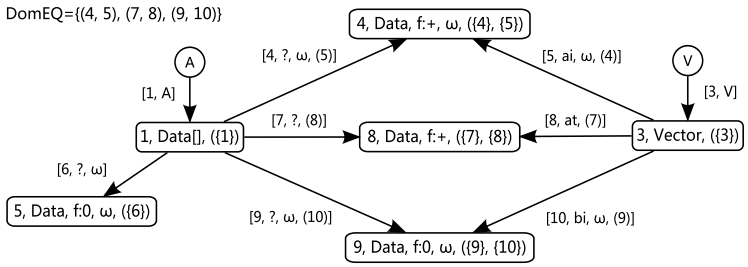
Figure 6(a) shows the fixpoint model which represents all the states that are generated in the loop. We see that there may be many elements in the vector `V` and many elements that are not added to the vector (represented by the edges with the `bi` labels, 4 and 6 respectively). Since we tracked the dominance 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 is dominance equivalent to edge 4 and thus both edges 4 and 5 must dominate node 4.

If we assume the `i < A.Length` test returns false then 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`), Figure 6(b). Thus, 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 $\equiv_{\text{dom}}$ edge 5).

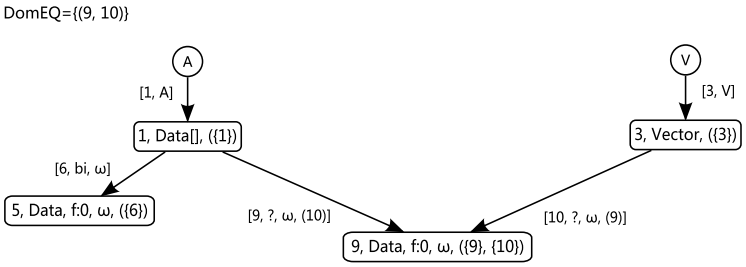
6.2 Update Loop Example

For brevity we omit descriptions of how the dominance information is propagated during the individual operations of the update loop (lines 7-8) and focus on how this information is used to improve the precision of the analysis results at the exit of the loop. The fixpoint model 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`) dominate their respective 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



(a) Fixed Point Update of Loop



(b) After Loop Exit

Fig. 7. Fixpoint and Exit of Map Loop

in indices greater than or equal to `i`. Based on the dominance equality relations (4, 5) and (7, 8) this implies that edges 4 and 7 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 the objects with non–zero `f` fields in the vector `V` or the array `A`. Thus, the loop has successfully strongly updated all of the objects in the vector `V` and this strong update information has been reflected in the original array `A`. 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 properties and dominance equivalences presented in this paper and evaluated the effectiveness and efficiency of the analysis on programs from SPECjvm98 [16] and the entire non–trivial JOlden [3] suite. The JOlden suite contains pointer–intensive kernels that make use of recursive procedures, inheritance, and virtual methods. We modified the suite to use modern Java programming idioms. The benchmarks `raytrace` (modified to be single threaded) and `db` are taken from SPECjvm98.

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 4 GB of RAM (although memory consumption never exceeded 120 MB).

For each of the benchmarks we provide a brief description of some of the major structures/features that are in the program. We mention the major data structures used

Benchmark	LOC	Description	Analysis Time	ShapeO	ShapeND	ShapeD
bisort	560	Tree w/ Mod	0.29s	Y	P	Y
mst	668	Cycle w/ Struct.	0.15s	Y	Y	Y
tsp	910	Tree to Cycle	0.19s	Y	Y	Y
em3d	1103	Bipartite Graph	0.44s	N	P	Y
perimeter	1114	Tree w/ Parent Ptr	1.23s	Y	P	P
health	1269	Tree w/ Mod	1.40s	N	Y	Y
voronoi	1324	Cycle w/ Struct	2.03s	N	Y	Y
power	1752	Lists of Lists	0.45s	Y	Y	Y
bh	2304	N-Body Sim w/ Mod	2.11s	N	P	P
db	1985	Shared/Mod Arrays	0.97s	N	P	Y
raytrace	5809	Shared/Cycle/Tree	42.91s	N	P	Y

Fig. 8. ShapeO is the best analysis reported in the literature, ShapeND is the shape results of the baseline analysis without dominance information, and ShapeD is the analysis in this paper with dominance information. LOC is for the normalized program representation including library stubs required by the analysis. Analysis Time is the analysis time for ShapeD in seconds.

(Trees, Lists of Lists, Cycles, etc.) and if the program heavily modifies the data structures (w/ Mod). Some of the benchmarks have slightly more nuanced structures, mst and voronoi which build globally cyclic structures that have significant local structure, bh which has a complex space-decomposition tree and sharing relations, and raytrace which builds a large multi–component structure which has cyclic structures, tree structures and substantial sharing throughout. We also note that tsp, and voronoi begin with tree structures and process them building up a final cyclic structure during the program. These benchmarks thus exercise a wide range of features in the analysis based on: the types of structures built, modification of these structures, sharing of the structures, use of multi–component structures, and the use of arrays/collections.[2]

To assess the accuracy of the analysis, we report, in the *ShapeO*, *ShapeND*, *ShapeD* columns of Table 8, the results of various shape analysis techniques. The *ShapeO* lists the most accurate results from the related literature [1, 6–9, 15, 18, 19], the *ShapeND* column is the result for the baseline analysis from this paper (dominance information disabled) and the *ShapeD* column is the result for the analysis from this paper with dominance information enabled. We use three categories for the accuracy of the analysis. Y(es) means the analysis was able to provide shape and sharing information for all of the relevant heap structures in the program. P(artial) means the analysis was able to determine the precise shape for some of the data structures but that some important properties were missed. N(o) means the analysis failed to precisely identify the shape/sharing information for a substantial portion of the heap data structures.

The benchmarks raytrace, db and bh (which to the best of our knowledge have not been analyzed using other shape analysis techniques) use a range of complex data structures, destructive operations, and forms of sharing between data structures. In these benchmarks the information provided by the dominance relations is critical to precisely modeling shape, connectivity and sharing properties. These benchmarks are also a factor of 2-4 larger, and build much more complex structures, than what is usually reported

[2] See `www.cs.unm.edu/~marron/software/software.html` for examples of the analysis results and an executable analysis demo.

in the literature. Thus, the small runtimes (less than a minute per benchmark) indicate that even with the addition of the dominance relations the analysis is computationally tractable.

Our experiments demonstrate that dominance equivalence can be used to efficiently and precisely analyze common programming idioms to build, share, and modify non-trivial data structures. Based on these results we believe that the proposed approach presents a basis for a heap analysis that can be used in practice to provide detailed heap information for a range of optimization and verification applications.

Acknowledgements. The authors thank Mooly Sagiv, Roman Manevich, and Amer Diwan for their useful comments on earlier versions of this paper.

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. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, 2004.
3. B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.
4. D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.
5. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.
6. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
7. S. Gulwani and A. Tiwari. An abstract domain for analyzing heap-manipulating low-level software. In *CAV*, 2007.
8. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
9. T. Lev-Ami, N. Immerman, and S. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.
10. R. Manevich, S. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, 2004.
11. M. Marron, M. Hermenegildo, D. Stefanovic, and D. Kapur. Efficient context-sensitive shape analysis with graph based heap models. In *CC*, 2008.
12. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.
13. M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *Submission*, 2008.
14. M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.
15. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.
16. Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. http://www.spec.org/jvm98.
17. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
18. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.
19. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. OHearn. Scalable shape analysis for systems code. In *CAV*, 2008.