

Identification of Logically Related Heap Regions

Mark Marron¹, Deepak Kapur², and Manuel Hermenegildo¹

¹IMDEA-Software (Spain), {mark.marron, manuel.hermenegildo}@imdea.org

²University of New Mexico (USA), kapur@cs.unm.edu

Abstract. This paper introduces a novel technique for identifying logically related sections of the heap such as recursive data structures, objects that are part of the same multi-component structure, and related groups of objects stored in the same collection/array. When combined with the lifetime properties of these structures, this information can be used to drive a range of program optimizations including pool allocation, object co-location, static deallocation, and region-based garbage collection. The technique outlined in this paper also improves the efficiency of the static analysis by providing a normal form for the abstract models (speeding the convergence of the static analysis).

We focus on two techniques for grouping parts of the heap. The first is a technique for precisely identifying recursive data structures in object-oriented programs based on the types declared in the program. The second technique is a novel method for grouping objects that make up the same composite structure and that allows us to partition the objects stored in a collection/array into groups based on a *similarity* relation. We provide a parametric component in the *similarity* relation in order to support specific analysis applications (such as a numeric analysis which would need to partition the objects based on numeric properties of the fields). Using the *Barnes-Hut* benchmark from the JOlden suite we show how these grouping methods can be used to identify various types of logical structures allowing the application of many region-based program optimizations.

1 Introduction

Identifying and grouping logically related parts of the program heap in an abstract program model is useful both to client optimization applications (which can use the information to perform pool allocation, object co-location, static deallocation, etc.) and in improving the performance of the static data flow analysis (providing a normal form which speeds the convergence of the analysis). This paper presents a novel technique for identifying and grouping these regions in a manner that supports a wide range of client applications and that can be used in practice to produce an efficient static analysis.

Research on object allocation and memory layout has used the notions of logically related structures to improve the spatial locality of objects with similar temporal accesses via techniques such as pool allocation [16, 3] and object co-location [11, 7]. Other work which uses logically related sections of the heap has focused on improving the efficiency of garbage collection. The most obvious application is static deallocation of regions or data structures [4, 12]. There has also been work [14] on using region information to reduce the pause times of garbage collection by only performing the

collection on portions of heap that are likely to contain many dead objects. Similar approaches (when combined with heap based read/write information) can also be used to support parallel garbage collection by statically identifying which parts of the heap can be safely collected without concern for the mutator.

The techniques listed above use a variety of techniques for identifying the region information that is later used in the optimizations phase. The techniques range from simple grouping via the points-to partitions computed using a Steensgaard style analysis [22, 13] to more complex and specialized approaches as done in [16, 12]. However, the technique in this paper offers a significantly higher degree of precision than these approaches and thus the method for identifying regions presented in this paper can be used directly to improve the effectiveness of these optimizations.

In addition to being useful for a range of optimization techniques the region identification technique we present in this paper can be used to improve the performance of various static analysis techniques. This is achieved by using the region identification to define a normal form for the abstract models, reducing the height of the abstract lattice. This use of a normal form can be seen as a pseudo-widening operation used to transform a domain of infinite height (e.g, linked lists of size $0, 1 \dots \infty$) into a finite height lattice (e.g, linked lists that are of size $0, 1, 2$, or some unknown length ω). While this idea is not novel to this paper—symbolic access paths in [6], normalization/merge in [2, 5, 17], and the append left/right rules in [1, 23] or similar rules for inductive synthesis in [10]—the approach we present here is significantly more general. In particular the formalization applies to any type of recursive data structures (as opposed to just lists in [1] or trees [10]), it can precisely model many types of cyclic structures (which are merged in [17, 6]), and it supports more precise grouping of the contents of collections (arrays or collections from `java.util`) than is possible with the methods described in [2, 20] (and which are left out in most other approaches).

We begin with a brief introduction of the parametric labeled storage shape graph (*lssg*) model, Section 2, that we use to illustrate the main contributions of this paper. These contributions as described in Sections 3 and 4 are:

- A method for identifying and grouping recursive data structures.
- A method for identifying groups of objects that form multi-object composite structures.
- A parametric approach to grouping the contents of arrays/collections.

Finally, in Section 6 we use the well known *Barnes-Hut* benchmark from the JOlden suite to illustrate the results of the region analysis and how this information can be used to support some of the optimizations mentioned above.

2 Concrete Heap and Labeled Storage Shape Graph

We begin by reviewing the abstract graph model that we build on in this work (although the concepts presented in this paper can be easily applied in other approaches such as those that rely on separation logic [1, 10, 23]). In previous work [17–20] this model is used to precisely perform shape and sharing analysis on a range of Java 1.4 programs. While the properties discussed therein are critical to precisely analyzing these

programs, we do not need all of this information in order to perform region identification and grouping. Thus, to simplify the discussion and to focus on the novel concepts in this paper we present a simplified version of the model.

2.1 Concrete Semantics

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 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 them, the references that enter the region $R_{in} = \{(a, b, r) \in R \mid a \in (O \cup V) \setminus C \wedge b \in C \wedge a \xrightarrow{r} b\}$ and references exiting the region $R_{out} = \{(a, b, r) \in R \mid a \in C \wedge b \in O \setminus C \wedge a \xrightarrow{r} b\}$.

2.2 Storage Shape Graph Abstraction

Our abstract heap domain is based on the *storage shape graph* [2] 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 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 1 (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:

- $\Pi_v : V \mapsto \hat{V}$, $\Pi_o : O \mapsto \hat{N}$ and $\Pi_r : R \mapsto \hat{E}$ are functions (and Π_v is 1–1).
- h, Π_v, Π_o , and Π_r satisfy all the relations in \hat{U} .
- h, Π_v, Π_o , and 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 $e.offset = p$.
- $\forall v \in V, o \in O$ 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 $e.offset = v$.

To check if a given concrete heap h and maps Π_v, Π_o, Π_r satisfy a given relation in \hat{U} 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. Similarly, we use $h \downarrow_g n$, to indicate the region of the heap that is the image of n under the maps.

2.3 Label Relations (in \hat{U})

Type. For the *type* relation, we add a relation $(n, \{\tau_1, \dots, \tau_k\})$ (we use the shorthand $n.type = \{\tau_1, \dots, \tau_k\}$) 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 $(n, \{\tau_1, \dots, \tau_k\})$ iff $\{\text{typeof}(o) \mid \text{object } o \in h \downarrow_g n\} \subseteq \{\tau_1, \dots, \tau_k\}$.

Linearity. The *linearity* relation is used to track the number of objects in the region abstracted by a given node or the number of references abstracted by a given edge. The *linearity* property has 2 values: 1 indicating a cardinality of $[0, 1]$ or ω indicating a cardinality of $[0, \infty)$. Given a node n where $h \downarrow_g n = (C, P, R_{in}, R_{out})$ then:

$$|C| \in \begin{cases} [0, 1] & \text{if } n.linearity = 1 \\ [0, \infty) & \text{if } n.linearity = \omega \end{cases}$$

Similarly for an edge e where $h \downarrow_g e = \{r_1, \dots, r_j\}$ then:

$$|\{r_1, \dots, r_j\}| \in \begin{cases} [0, 1] & \text{if } e.linearity = 1 \\ [0, \infty) & \text{if } e.linearity = \omega \end{cases}$$

Abstract Layout. To approximate the shape of the structures present in the region that a node abstracts, the analysis uses *abstract layout* properties $\{(S)ingleton, (L)ist, (T)ree, (M)ultiPath, \text{ and } (C)ycle\}$. The *(S)ingleton* property states that there are no pointers between any of the objects abstracted by the node, given a node n where $h \downarrow_g n = (C, P, R_{in}, R_{out})$ then $P = \emptyset$. The other properties correspond to the standard definitions for list, tree, DAG, and cyclic structures in the literature [9, 18, 17, 1].

2.4 Sample Heap and Abstract Graph Model.

Figure 1 shows a linked list of length 3 or more (left) and the representation of this list in the abstract domain with the objects that represent it grouped into regions (right). In the abstract domain each edge is labeled with a unique identifier, an abstract storage *offset*, and a *linearity* label. The nodes are labeled with a unique identifier, a *type* label, a *linearity* label and a *layout* label.

In Figure 1(b) we see that the variable `l` refers to node 1 which represents a single (*linearity* is 1) ListNode (LN) object at the head of the linked list. There is a single edge (edge 2) out of the node representing the single (again *linearity* 1) `(n)ext` pointer, which ends at node 2. This node represents the tail of the list (the self `n` edge and *(L)ist layout*) which may contain many objects (*linearity* is ω).

Partitioning the list into these two nodes captures several important attributes. First we have kept the head of the list (which may be modified though the variable `l`) distinct,

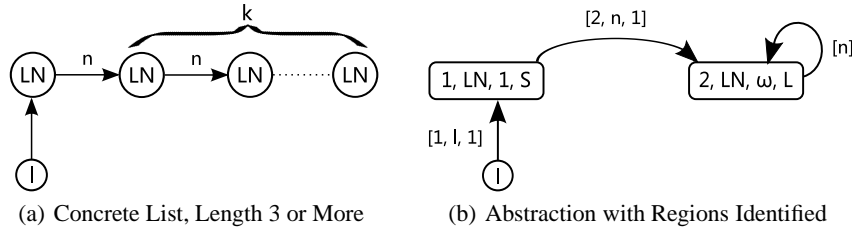


Fig. 1. A Linked List and Desired Abstraction with Regions Identified

giving more opportunities to the analysis for precisely modeling the effects of later program statements. Next, the grouping has produced a compact representation for the list structure which has a substantial impact on the efficiency of the analysis. Finally, we have grouped all of the objects that make up the list into two nodes (the head and the tail, nodes 1 and 2) and as we will see later if there are other unrelated lists in the program the abstraction will generate separate nodes for each of these lists. Thus, the information needed by the various optimization techniques we are interested in is preserved (objects in the same structures are grouped together while disjoint structures in the concrete heap are kept separate in the abstract model).

3 Recursive Components

The first contribution in this paper is a generalized method for identifying parts of the abstract heap graph that may represent a single recursive data structure and how these parts should be grouped together (e.g. using multiple nodes to represent the head and tail sections of the linked list). While the basic approach to identifying potentially recursive structures is a straightforward examination of the type information and connectivity properties of the program based on the symbolic access paths in [6] there are a number of subtle but important modifications that are needed maintain the desired level of precision in the results, which we describe in Subsection 3.2.

3.1 Statically Recursive Types

We can identify the types in a program that may be recursive by looking at the type graph for the program. This *static program type graph* has a node for each type that is declared and for each pair of types τ, τ' there is an edge from τ to τ' if τ has a field of type (or supertype) τ' . From this construction we can identify types that are recursive (based on the static type information) as follows:

Definition 2 (Statically Recursive Types). For a given program and types τ, τ' :

1. τ, τ' are statically recursive iff in the static program type graph $(\tau \neq \tau' \wedge \tau, \tau'$ are in the same strongly connected component) $\vee (\tau = \tau'$ and there is a self edge).
2. τ is a statically recursive type iff $\exists \tau'$ s.t. τ, τ' are statically recursive.

3.2 Recursive Structure Identification Refinements

In much of the past work on region identification [17, 16, 6, 1, 10] this static type information has been used (in various ways) to determine if two objects are part of the same recursive data structure. However, this can result in overly approximate region identification in three important classes of heap structures. Below we describe these and how we can modify our concept of recursive structures to characterize them.

Safe Nodes. In order to accurately simulate the effects of various program statements it is critical to precisely model the targets of variable references. Consider removing an element from a linked list where we have multiple variables pointing into the same list structure. In order to preserve the listness property after the removal we must keep track of the relative positions of the variable references into the list structure and the effects of the assignment statements on the objects referred to by the variables. Thus, even though all of these objects make up the same recursive list structure, we want to use multiple nodes to represent it (one node for each location in the list that is being modified in a addition to nodes representing the list tail or other segments).

To identify these important objects which need to be modeled independently we introduce the notion of *safe nodes*. We say a node is safe if it represents an interesting point in a recursive data structure (a point where the program is accessing a specific node in the data structure though a variable reference, as in the above example, or a non-recursive data structure pointing into specific locations in the recursive structure) and we keep these nodes distinct from any other recursive components.

If we have a recursive data structure and we store references to important points in it via another data structure we want to be able to maintain the relations between these specific points in a data structure for when they are accessed later in the program. This is a generalization of maintaining the precise locations of variable references into a recursive data structure. This is important to analyzing situations of the form: a method returns a `Pair` object containing two `ListNode` objects and we want to remove all the elements in the list between the `first` and `second` entries of the `Pair`. If the analysis does not maintain the order relation between the targets of the `first` and `second` reference fields in the list structure we cannot accurately model the effects of the remove operation (e.g., we would conservatively assume that the target of the `second` field could come before the target of the `first` field in the list).

Definition 3 (Safe Node). A node n is safe if it is a node of linearity 1 and either of the following hold:

1. \exists variable v that refers to n .
2. \exists edge e s.t. e starts at a node n_s where $\forall \tau_s \in n_s.\text{type}, \tau \in n.\text{type}, \tau_s, \tau$ are not statically recursive.

Connectivity Awareness. Consider a program with the object types τ_1, τ_2, τ_3 which are mutually recursive on the `n` field. If we have the abstract heap graph in Figure 2 we can see that the 2nd and 3rd nodes in the list are statically recursive according to the definitions above but that there is no complete recursive structure present in the program (no type appears multiple times in the same structure). This can occur frequently with

pre 1.5 Java collections as their contents are untyped (they may contain any type of object) and are thus statically recursive with all other types.

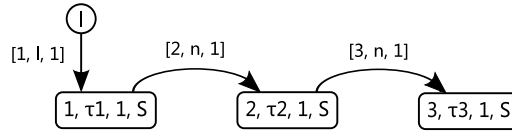


Fig. 2. Recursive Types But No Complete Structure

To avoid this problem we perform a connectivity aware detection of the recursive structures which takes into account the connectivity information of the heap and ensures that we only consider two nodes as being recursive if they are part of a *complete recursive structure*. Which ensures only nodes that are in repeating and uninteresting parts of a recursive data structure are grouped into a single region.

Definition 4 (Complete Recursive Structure). Two nodes n, n' are part of a complete recursive structure if:

\exists edge e from n to n' , $\exists n_\tau$ and a path from n' to n_τ s.t. none of n, n', n_τ or the nodes on the path are safe, and $n.type \cap n_\tau.type \neq \emptyset$.

Recursive vs. Back Pointers. Many programs use back pointers causing the above definition to identify any cyclic structure as recursive, since trivially every node can reach itself and thus every type appears multiple times. This causes the grouping of cycles in the graph into single nodes with the *layout (C)ycle*, which can lead to substantial imprecision. Figure 3 shows an example of such a heap. We can see that even though the heap structure is finite, the back edge will cause our recursive component definition to group the 2nd and 3rd nodes into the same recursive component.

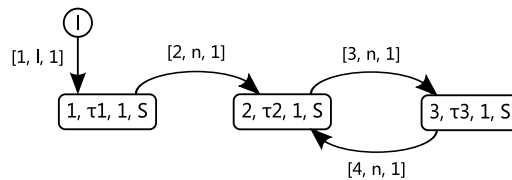


Fig. 3. Recursive Cycle

To address this problem (and other similar problems that arise when identifying unbounded recursive structures when cyclic structures are present) we modify the recursive definition to ignore back edges when determining if two nodes are recursive.

3.3 Recursive Node Definition

Given the above scenarios and the proposed solutions for handling them we get the following final definition for determining if two nodes are recursive.

Definition 5 (Recursive Nodes). *Given the function depth which returns the depth of a node in the graph, nodes n, n' (where $n \neq n'$) are recursive if:*

\exists edge e from n to n' , neither of n, n' are safe and $\exists n_\tau$ s.t. there is a (possibly empty) path $\psi_r \langle (n_1^s, n_1^e) \dots (n_k^s, n_k^e) \rangle$ from n' to n_τ s.t. $\forall (n_i^s, n_i^e) \in \psi_r, \text{depth}(n_i^s) < \text{depth}(n_i^e)$ (where depth is the depth of the node in the graph), $\forall (n_i^s, n_i^e),$ neither n_i^s or n_i^e is safe and, $n.\text{type} \cap n_\tau.\text{type} \neq \emptyset$.

4 Composite Components and Array/Collection Grouping

The second contribution of this paper is a method to identify composite structures and equivalence classes of the objects stored in arrays or collections. This is done by defining a parametric predicate for determining if two nodes represent *equivalent* regions of the heap. The method presented in this section is effective for the application of identifying heap regions based on simple connectivity information (and is sufficient for our optimization applications) but the parametric component allows for the predicate to be tailored to support other applications as well (for example if we are using a numeric domain we can extend it to keep objects in an array with non-zero values in a given field distinct from objects that must have a zero in this field [18]).

We introduce a notion of *equivalence* of two nodes that captures our intuition of when two nodes n, n' abstract similar regions of the concrete heap. Since the *equivalence* predicate is used to determine the maximum number of out edges each node may have, we can improve efficiency by minimizing the number of equivalence classes created by this relation. The tradeoff between precision and performance that we have found to be acceptable is determined by the following conditions: (1) are all the types represented by the nodes non-recursive (or may both nodes represent recursive types) and (2) what variables can access the objects in the regions abstracted by the nodes?

Recursive Similarity. Two nodes are *recursive similar* if they both abstract all non-recursive types or they both may abstract an object with a recursive type. An example of why this is important is the common construction of k-ary trees using arrays/collections to hold either a recursive subtree or a non-recursive leaf object.

Definition 6 (Recursive Similarity). *Given nodes n, n' and the statically recursive type information, n, n' are recursive similar iff either of the following holds:*

1. $(\exists \tau \in n.\text{type}, \tau \text{ is statically recursive}) \wedge (\exists \tau' \in n'.\text{type}, \tau' \text{ is statically recursive})$
2. $(\nexists \tau \in n.\text{type}, \tau \text{ is statically recursive}) \wedge (\nexists \tau' \in n'.\text{type}, \tau' \text{ is statically recursive})$

An example of this situation is in the `bh` program discussed in Section 6. This program performs a n-body gravitational simulation on a set of `Body` objects using the

fast-multipole approach. This technique builds a space decomposition tree made up of `Cell` objects each of which has a reference to a `Vector` containing references to other `Cell` objects in the space decomposition tree or to the `Body` objects. Using good OOP style the `Cell` and `Body` objects both inherit from an abstract `Node` class. Thus, if we did not distinguish between the recursive `Cell` objects which make up the tree structure and the leaf `Body` objects the analysis would end up grouping the tree (with the `Cell` objects) and the leaf objects (the `Body` objects) into the same region. However, by distinguishing regions based on their *recursive similarity* we can avoid this and ensure that the tree structure and the leaf objects are grouped into different regions.

Reference Similarity. If we have two nodes n, n' and the objects abstracted in the region by n are all stored in an array A and all the objects in the region abstracted by n' are stored in array A and a second array B then it is reasonable to assume that the programmer has partitioned these objects differently for some reason. Thus, we want to preserve this information by keeping the nodes distinct, we show this situation in Figure 4. We can ensure that the information on which collections and variables refer to which sets of objects is maintained by using the following definition of *reference similarity*.

Definition 7 (Reference Similarity). We say two nodes n, n' are *reference similar* if given the set of in edges to n , $E_{in} = \{e_1^n \dots e_k^n\}$, the set of in edges to n' , $E'_{in} = \{e_1^{n'} \dots e_k^{n'}\}$, and the set of variables that can reach node n , $V_r = \{v_1^n \dots v_r^n\}$, the set of variables that can reach node n' , $V'_r = \{v_1^{n'} \dots v_s^{n'}\}$, the following holds:

$$(\{e.offset \mid e \in E_{in}\} = \{e'.offset \mid e' \in E'_{in}\}) \wedge (V_r = V'_r)$$

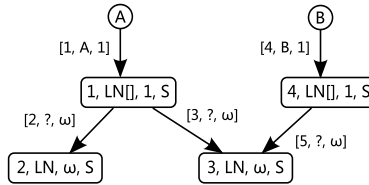


Fig. 4. Nodes 2, 3 Not Reference Similar (based on variable reachability)

This definition ensures that if two nodes are treated differently with respect to the types of objects they are stored in or the variables that reach them then they are kept separate. In Figure 4 nodes 2 and 3 are not *reference similar* since node 2 is reachable from variable A while node 3 is reachable from both variables A and B.

Parametric Node Equivalence. In addition to using the structural information provided by the *recursive similar* and *reference similar* relations we can also provide a parametric component to the grouping operation to support the needs of more specific types of analysis. In some of our related work [18] we have used examples that involve tracking the numeric field values of objects stored in each region. In order to achieve the required

level of precision it is important to avoid grouping regions with differing field values even if the objects they represent are stored in the same array. Thus our definition allows parametric similarity properties to support specialized analyses.

Definition 8 (Equivalent Nodes/Edges). *Given the above definitions we define edge equivalence. Given a node n and two out edges e, e' which start at node n and end at nodes n_e and $n_{e'}$ respectively we say e, e' are equivalent if:*

1. $e.offset = e'.offset$
2. $n_e, n_{e'}$ are recursive similar
3. $n_e, n_{e'}$ are reference similar
4. $n_e, n_{e'}$ are equivalent for all parametric similarity relations

5 Region Identification and Grouping

Using the above definitions for identifying recursive structures, composite structures and grouping the contents of collections/arrays we define the method for constructing the logically related regions. Once we have identified a set of nodes that represent a logically related region, based on our region predicates, we need to replace them with a single node that *safely* approximates the properties of the nodes in the set.

Component Summarization. Before we present the complete region identification/normalization algorithm we describe how the summary nodes are computed. To simplify the computation we perform the summarization in a pairwise manner. When summarizing two nodes, n and n' , there are three possibilities. The first is that there are no edges between the nodes, there are only edges in one direction between nodes (from n to n' or n' to n , but not both) and when there are edges from n to n' and from n' to n .

If there are no edges between the nodes we use the *mergeNoEdge* method to compute the summary representation. This method is a simple component-wise operation where the updated *type* label is the union of the two *type* sets, the *linearity* value is ω and the *layout* is the max (the most general) of the two *layout* labels. The case where there are edges from n to n' and from n' to n (*mergeBothWay*) is similar except we always assume the *layout* of the summary node is (*C*)ycle.

The *mergeEdgeOneWay* operation (Algorithm 1) on a pair of nodes that have connecting edges is more complicated. In particular we need to account for the fact that the edge(s) connecting nodes n and n' will affect the *layout* of the new summary node.

Algorithm 1: mergeOneWay

```

input : graph  $g, n, n'$  nodes,  $ebt$  set of edges from  $n$  to  $n'$ 
 $n.types \leftarrow n.types \cup n'.types;$ 
 $n.linearity \leftarrow \omega;$ 
 $n.layout \leftarrow combineLayout(n.layout, n'.layout, ebt);$ 
remap all edges incident to  $n'$  to be incident to  $n$ ;
deleteNode( $g, n'$ );

```

The algorithm *combineLayout*(l, l', ebt), is based on a case analysis of the *layout* that results from the possible combinations of the *layouts* for n, n' along with the total number of pointers represented by *ebt*. We enumerate the possible combinations of the *ebt* edges and the *layout* labels and then for each case we use the semantics of the edge and *layout* properties to determine the most general *layout* type that may result from this particular case. For example if we have two (*S*)ingleton nodes connected by an edge of *linearity* 1 then the most general *layout* for a node that summarizes these nodes and the edge is a (*L*)ist.

To merge two arbitrary nodes n, n' we use Algorithm 2 which selects the appropriate method for merging two nodes based on the existence of edges between them.

Algorithm 2: mergeNode

```

input : node  $n, n'$ , graph  $g$ 
if  $\exists$  edges from  $n$  to  $n'$  and  $n'$  to  $n$  then
    mergeBothWay( $g, n, n'$ );
else if  $\exists$  edges from  $n$  to  $n'$  then
    mergeOneWay( $g, n, n', \{e \mid e \text{ from } n \text{ to } n'\}$ );
else if  $\exists$  edges from  $n'$  to  $n$  then
    mergeOneWay( $g, n', n, \{e \mid e \text{ from } n' \text{ to } n\}$ );
else
    mergeNoEdge( $g, n', n$ );

```

Region Identification/Normalization Algorithm. Once we have the above methods for computing summary nodes for a pair of nodes in the graph we can define the final region identification algorithm. The resulting region grouped model is also a convenient normal form ensuring that the static analysis terminates as the infinite set of *labeled storage shape graphs* is a finite set under the normal form (recursive structures are represented by a bounded number of nodes and each node has a bounded number of out edges, for space we omit a formal proof).

The algorithm is a straightforward iterative identification of pairs of nodes/edges that should be grouped and the replacement of these structures by a summary representation until a fixpoint is reached. After this method terminates the abstract graph model will have all the logically related regions identified and grouped according to the characterizations in Sections 3 and 4.

6 Case Study and Experimental Evaluation

Barnes-Hut Case Study. Our version of the `bh` program performs a gravitational interaction simulation on a set of bodies (the `Body` objects) using a *fast-multipole* technique with a space decomposition tree. The tree is represented using `Cell` objects each of which has a `Vector` containing references to other `Cell` objects or references to the `Body` objects. The program also keeps two `java.util.Vector` objects for accessing the bodies, `bodyTab` and `bodyTabRev`. The positions (`pos`), velocities (`vel`)

Algorithm 3: groupRegions

```
input : graph  $g$ 
while  $g$  is changing do
  while  $\exists$  node  $n$  with edges  $e, e'$  s.t.  $e \neq e' \wedge e, e'$  are equivalent edges do
    mergeNode(target of  $e$ , target of  $e'$ ,  $g$ );
     $e$ .linearity  $\leftarrow \omega$ ;
    deleteEdge( $g, e'$ );
  while  $\exists$  nodes  $n, n'$  that are recursive do
    mergeNode( $g, n, n'$ );
```

and acceleration (acc) values of the bodies are represented with composite structures consisting of a MathVector object and a double[] .

Figure 5 shows the abstract heap model built and used in the stepSystem method of the benchmark (the listing below), where the space decomposition tree is recomputed (the makeTree method), the body-body interactions are computed (the loop with the hackGravity method), and the new acceleration information is propagated (the vprop method).

```
public void stepSystem() {
  this.makeTree(nstep);

  Iterator bi = this.bodyTabRev.iterator();
  while (bi.hasNext())
    ((Body) bi.next()).hackGravity(rsize, root);

  vprop(this.bodyTabRev, nstep);
}
```

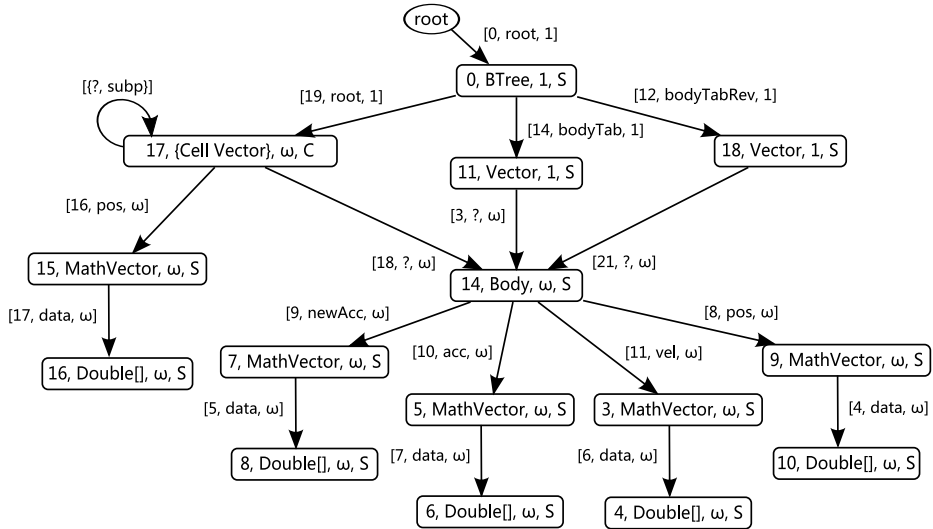


Fig. 5. Abstract Heap in BH

As we can see in Figure 5 the region identification algorithm is able to correctly identify and group all the major components in the overall heap structure. The space decomposition tree is grouped into the region represented by node 17 (although the analysis has overly conservatively assumed the structure may have a *(C)yclic* layout) while the leaf `Body` objects are represented separately by node 14. The analysis has also grouped the composite `MathVector/double[]` structures and has maintained the separation of these structures when they abstract distinct structures and are stored in different types or in different fields.

The `bh` program has many opportunities to apply the optimizations discussed in the introduction [16, 11, 7, 12, 14]. In particular the information computed by the analysis in this paper enables opportunities that could not be previously exploited due to a lack of sufficiently precise region identification. For example we can determine that the space decomposition tree (node 17) and the `Body` objects (node 14) are good candidates for pool allocation [16] (and collection) while the `MathVector/double[]` structures are good candidates for co-location [11, 7]. In examining the `vprop` method, which updates the new position and velocity information for each `Body`, we can use the region information to determine that the `MathVector` objects stored in the different fields of the `Body` objects can be effectively region allocated and collected [14]. If we include *sharing* information as described in [19] we can determine precisely when these objects are dead and immediately reclaim them instead of waiting for the collector [12].

These transformations allow for the efficient collection (by collecting individual objects or entire pools) of all the dead objects created during this main computation portion and for the location of temporally related objects into contiguous parts of memory. Thus, this benchmark demonstrates how the precision of the region analysis presented in this paper enables the application of a number of powerful program optimizations that reduce the memory requirements, reduce garbage collection costs, and to improve the performance of the program.

Experimental Evaluation. We have implemented a shape analyzer based on the region identification methods and instrumentation properties presented in this paper and evaluated the effectiveness and efficiency of the analysis on programs from SPECjvm98 [21] and a version of the JOlden [15] 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 (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 struc-

Benchmark	LOC	Description	Analysis Time	Region Correct
bisort	560	Tree w/ Mod	0.26s	Yes
mst	668	Cycle w/ Struct.	0.12s	Yes
tsp	910	Tree to Cycle	0.15s	Yes
em3d	1103	Bipartite Graph	0.31s	Yes
perimeter	1114	Tree w/ Parent Ptr	0.91s	Yes
health	1269	Tree w/ Mod	1.25s	Yes
voronoi	1324	Cycle w/ Struct	1.80s	Yes
power	1752	Lists of Lists	0.36s	Yes
bh	2304	N-Body Sim w/ Mod	1.84s	Yes
db	1985	Shared/Mod Arrays	1.42s	Yes
raytrace	5809	Shared/Cycle/Tree	37.09s	Yes

Fig. 6. LOC is for the normalized program representation including library stubs required by the analysis. Analysis Time is the analysis time for the analysis in seconds.

tures, 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.¹

To assess the accuracy of the analysis, we report, in the *Region Correct* column of Table 6, the results of the region identification presented in this paper. In all of the programs we examined the analysis was able to identify and precisely group the heap into logically related regions (as with the *bh* case study).

Region analysis in the context of shape analysis is a relatively unstudied problem. Most work on region analysis has been based on points-to or reachability based analyses [12, 16, 7, 8, 13, 14]. Other notable work includes [5] which uses a graph based approach (similar to this paper) but has a much less general and precise grouping method, and the related work on separation logic based analysis places restrictions on the types of structures built [1, 23] (only allow linked lists) or on the method of construction [10] (limits the how recursive structures can be built). Thus, many of these benchmarks cannot be precisely analyzed with other existing methods, including *bh*, *em3d*, *voronoi*, and *raytrace*, which all have substantial opportunities for the application of various region based optimization (similar to the *bh* case study).

Our experiments demonstrate that the proposed region identification method can be used to precisely identify and group logically related regions of the heap (recursive data structures, composite structures composed of multiple objects and the contents of arrays/collections). Further this information can be computed efficiently. In fact the normal form that results from this grouping greatly improves the performance of the analysis by restricting the size of the abstract graph domain (which speeds the convergence of the analysis). 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

¹ See www.cs.unm.edu/~marron/software/software.html for benchmark code, examples of the analysis results, and an executable analysis demo.

information for a range of optimization applications that rely on region information and we are currently working on improving the practicality of the analysis by developing on techniques to scale it to larger programs.

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. Cherm and R. Rugina. Region analysis and transformation for Java programs. In *ISMM*, 2004.
4. S. Cherm and R. Rugina. Compile-time deallocation of individual objects. In *ISMM*, 2006.
5. S. Chong and R. Rugina. Static analysis of accessed regions in recursive data structures. In *SAS*, 2003.
6. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k -limiting. In *PLDI*, 1994.
7. J. Dolby and A. Chien. An automatic object inlining optimization and its evaluation. In *PLDI*, 2000.
8. D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *CC*, 2000.
9. R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *POPL*, 1996.
10. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
11. S. Z. Guyer and K. S. McKinley. Finding your cronies: static analysis for dynamic object colocation. In *OOPSLA*, 2004.
12. S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *PLDI*, 2006.
13. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.
14. M. Hirzel, A. Diwan, and M. Hertz. Connectivity-based garbage collection. In *OOPSLA*, 2003.
15. Jolden Suite. <http://www-ali.cs.umass.edu/DaCapo/benchmarks.html>.
16. C. Lattner and V. Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. 2005.
17. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.
18. M. Marron, R. Majumdar, D. Stefanovic, and D. Kapur. Shape analysis with dominance equivalence. In *Submission*, 2008.
19. M. Marron, M. Méndez-Lojo, M. Hermenegildo, D. Stefanovic, and D. Kapur. Sharing analysis of arrays, collections, and recursive structures. In *PASTE (To Appear)*, 2008.
20. M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. In *PASTE*, 2007.
21. Standard Performance Evaluation Corporation. JVM98 Version 1.04, August 1998. <http://www.spec.org/jvm98>.
22. B. Steensgaard. Points-to analysis in almost linear time. In *POPL*, 1996.
23. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.