# Dominance: Modeling Heap Structures with Sharing

Mark Marron
U. of New Mexico
marron@cs.unm.edu

Rupak Majumdar
UC Los Angeles
rupak@cs.ucla.edu

Darko Stefanovic
U. of New Mexico
darko@cs.unm.edu

Deepak Kapur
U. of New Mexico
kapur@cs.unm.edu

August 27, 2007

**Abstract**

A number of papers have used predicate languages over sets of abstract locations to model the heap (decorating a heap graph with the predicates, or in conjunction with an access path abstraction). In this work we introduce a new predicate, *dominance*, which is a generalization of aliasing and is used to model how objects are shared in the heap (e.g. do two lists contain the same set of objects?) and how sharing influences the results of destructive updates (e.g. if all the objects in one list are modified does that imply that all the objects in another list are modified?). The dominance relation is introduced in the context of a graph-based heap model but the concept is general and can be incorporated in other frameworks as a high-level primitive.

The motivation for introducing a higher-order predicate to model sharing is based on the success of existing approaches which use connectivity, reachability, and sharing predicates to perform shape analysis using high-level graph-based models. Our approach provides an analysis technique that is efficient and scalable while retaining a high level of accuracy. This is in contrast to proposals in the literature based on using logical languages (usually built on top of first-order predicate logic) which either are highly expressive and very general (resulting in a computationally expensive analysis) or utilize only fragments of the full logic (which reduces the computational cost but results in a much less powerful analysis).

The approach presented in this paper has been implemented and run on a variation of the Jolden benchmarks. The information extracted using the analysis has been used to parallelize these programs with a near optimal speedup for 7 of the 9 benchmarks. The runtimes are small enough to make the analysis practical (a few seconds for each program).

## 1 Introduction

Precise reasoning about the structure of the program heap is crucial to understanding the behavior of a given program (particularly for programs written in imperative 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 [23]), are known to be too imprecise for many verification or optimization applications. More precise *shape analysis* techniques [21, 24, 1, 10, 16, 9] have been proposed when more accurate information is desired. These analyses recover precise information by distinguishing heap cells based on extra *instrumentation predicates* that encode reachability or separation information. By a suitable choice of instrumentation predicates, these

analyses can precisely model recursive data structures [21, 8, 16, 10] and composite structures [1, 16, 9] (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 sharing properties (and by negation, separation properties) of *individual* pointers. That is, the fundamental question asked of the abstract heap representations is whether a particular pointer may/must point to an object satisfying some combination of instrumentation predicates, or whether two pointers may possibly point to the same object. 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 is overly restrictive (and imprecise) when it comes to more complex subset relationships among sets of shared objects. Such reasoning arises in analyzing programs that use multiple views of the same collection of objects or that rely on the knowledge that updating all the objects in one collection implies (based on sharing of the contents) that all the objects in another collection are also updated.

Take for example the representation of a function in a compiler. The function body may contain a list of all instructions in the function and a list of all the basic blocks in the function. Each basic block would have a list of all the instructions in the basic block. Thus, there are two views of the instructions in the function. Further, during processing (perhaps in preparation for inlining) we may wish to replace all occurrences of a variable, say $x$ with a fresh variable $x'$. If the programmer has done this via a *foreach* loop over the list of instructions in the function, the analysis should determine that $x$ no longer appears in any of the instructions and further that this information is projected from the mapping over the list of instructions in the function into the lists of instructions in the basic blocks. That is, the analysis should be able to determine that none of the basic blocks contain an instruction that refers to the variable $x$.

The reasoning required here involves properties over *sets* of pointers or objects rather than individual pointers: we would like to know whether the *set* of instruction objects pointed to by a basic block are all contained in the set of instruction objects in the function, and therefore, by virtue of updating all instructions in the function, all instructions in a basic block are also updated.

We introduce *dominance relations* that enable existing shape analysis approaches to support such subset reasoning via a number of high-level predicates. Fundamentally, the dominance property is a generalization of aliasing. While aliasing tracks equality between the targets of variables or single pointers, dominance tracks the set relations (subset, equality) between the targets of variables and sets of pointers. Thus, the dominance relation subsumes aliasing and allows the analysis to track situations where two sets of references (pointers/variables) must refer to the same set of objects.

Such sharing relations between sets of objects can be simulated by introducing quantification with a "forall-exists" quantifier structure (i.e. for all objects pointed to by a reference in set $A$, does there exist a reference in set $B$ pointing to the same object?), or by adding explicit *reachability* or transitive closure predicates.[1] However, the introduction of general quantification or transitive closure makes abstract reasoning complicated, harder to scale, and dependent on the set of heuristics implemented in the underlying theorem prover. Instead, as demonstrated by the experimental evaluation in this paper, the dominance relations can be efficiently encoded and provide extensive information about various sharing properties. We believe that the logical languages used in [21, 1, 10, 9] can be extended with similar dominance properties, allowing them to simulate this limited (but useful) sharing information, without significantly increasing the computational complexity of the analysis.

[1]The use of reachability information to discharge the list-filter example was pointed out by Viktor Kuncak.

Initialize a List

```
list lo = new list<t1>()
t2 a = new t2()

while( * )
  t1 v = new t1()
  v.val = * ? a : null
  lo.push_back(v)
```

Filter elems that point to `a`

```
list ls = new list<t1>()
iterator i = lo.begin()
while(i.isValid())
  t1 t = i.get().val
  if(t == a)
    ls.add_back(t)
  i.advance()
```

Nullify the `val` fields

```
iterator i = ls.begin()

while(i.isValid())
  i.get().val = null
  i.advance()
```

Figure 1: Example Code

To make our proposal concrete and to enable empirical evaluation of the effectiveness of the dominance relation we extend an existing heap analysis framework, proposed in [16, 17], with dominance information. This extended analysis has been implemented and successfully run on a collection of well known parallelization benchmarks. The results on the benchmarks are extremely encouraging. Using the information provided by the heap analysis to drive the parallelization of the programs we achieve near optimal speedups for 7 out of the 9 benchmark programs. This analysis, to the best of our knowledge, is the only analysis method (to date) that is capable of providing the heap information needed to parallelize several of these benchmarks. Further, the analysis times are less than 4 seconds per program analyzed (most less than 2 seconds).

## 2 Example Programs

Consider the program segments shown in Figure 1, consisting of three subprograms manipulating lists: list construction, filtering elements from the list created in the first program into a sublist, and the update of the contents of the sublist from the second subprogram. The example uses objects of two types, `t1` and `t2`. The `t1` type has a single field `val` that points to objects of type `t2`.

The first code segment allocates an object *o* of type `t2` and assigns the variable `a` to refer to *o*. Then, it fills a list with objects of type `t1` each of which contains either a pointer to the object *o* or contains the `null` value in the `val` field. The second code segment scans the list `lo` for elements that have references to the object *o* and constructs a new list `ls` of all these elements. The final code segment updates all the elements in the list `ls` to have `null` pointers in the `val` field.

At the end of this example code, we want the analysis to determine the following properties: that all the objects in the sublist `ls` have had their `val` field set to `null` (there are no longer any references to the object *o* in the sublist) and that there are no objects in the original list `lo` that have a reference to *o*.

Using reachability, connectivity, and shape information various shape analysis techniques [1, 16, 9] can determine that after the execution of the first code segment the list container `lo` contains pointers to objects type of `t1` and each of the pointers in `lo` points to a unique object. Furthermore, it can be determined that each of the `t1` objects may have a pointer to the object *o*.

These predicates are not however sufficient to verify all the properties of interest that arise in the second code segment, that the objects in the list `ls` are a subset of the objects in the original list `lo`; and furthermore,

that every object with a *non-null* `val` field in `lo` must also be in the list `ls`.

The second of these properties is critical to the ability to determine the effect of the third code segment where all of the elements in the sublist have their `val` field set to `null`. If the analysis was unable to determine that all the objects in `lo` with a *non-null* `val` field must be in the sublist `ls` then it will be unable to determine that *nullifying* the `val` field of every element in `ls` implies that the `val` field of every object in the list `lo` must also be *null*.

To accomplish this our analysis will track (in addition to predicates dealing with the connectivity and shape of data structures) *dominance relations* on the heap. That is, we track how various sets of references share the objects that they point to. Given two sets of references (variables/pointers) $R_1$ and $R_2$ we want to track the set relations $(\subseteq, \supseteq, =)$ between the sets of objects referred to by the references in $R_1$ and in $R_2$.

**Sharing and Dominance:** The execution of the second code segment demonstrates how the dominance relation enables the proposed approach to model the sharing relationships between sets of pointers. After the loop exits, the contents of the list `ls` are a subset of the contents of the list `lo` (and conversely every element in the list `lo` that has a *non-null* `val` field must be in `ls`).

Figure 4(f) shows how the heap model presented in this paper is able to use the concept of dominance to precisely model this information. The reader would notice that the edge from the list `ls` (which has the label "7") refers to the same node as the edge labeled "3" from the list `lo`. If this was the only information available about the connectivity properties, the analysis can only infer that there are some pointers in the list `ls` that point to the same objects as some of the pointers in the list `lo`. But the analysis would be unable to infer the desired subset relations. With the introduction of the dominance properties as discussed in Section 3, the proposed method is able to determine that the pointers abstracted by the edge "7" *must* point to the same set of objects as the pointers abstracted by the edge "3" (the *DomEQ* relation contains the pair (3,7) which indicates this).

Furthermore, since edge "7" is the only edge out of the list `ls` this means that all the pointers stored in the list *must* be abstracted by edge "7". The objects stored in the list `ls` are thus a subset of the objects stored in the list `lo`.

The list `lo` has two outgoing edges "3" and "8". The target set of the pointers abstracted by edge "3" is equal to the set of objects stored in the list `ls`, edge "8" refers to a node that only represents objects with a *null* `val` field (there are no outgoing edges). Thus, the proposed approach can infer that if an object is stored in the list `lo` and it has a *non-null* `val` field, then it is also stored in the list `ls`.

**Modification and Dominance:** The second example code segment demonstrated how the *dominance* predicates enable the heap model to accurately represent how objects are shared. We also want to be able to model how the updates of a set of objects is reflected in other heap structures that share these objects. The third example code segment illustrates this.

In the third code segment, the contents of the sublist `ls` are modified by setting the `val` field of every object in it to `null`. Because of the sharing of these objects with the original list `lo`, these updates are reflected in it as well.

During the analysis of the third code segment the abstract state shown in Figure 5(d) is identified as a safe approximation of all possible executions of the loop body. The analysis knows that there is the element that the iterator `i` currently refers to (this object is represented by the node pointed to by the edge with the

label “14”), there is some set of objects that still need to be processed (these objects are represented by the node pointed to by edge “9”) and some set of objects that have already been processed (these are represented by the node pointed to by edge “10”). The contents of the original list are partitioned into these nodes plus an extra node (pointed to by edge “8”) which represents objects that are in `lo` but not in `ls`.

After the analysis has determined that the abstract heap in Figure 5(d) represents all the possible concrete states that may occur during the execution of the loop body it combines this abstract state information with the information implied by the exit test to determine the possible program states at the loop exit. In this case the analysis assumes that the `isValid` exit test returns *false*. This means that we are done processing the contents of the list `ls` (the iterator is at the end of the list) which implies that there is no current element to process and there are no unprocessed elements. This allows the analysis to delete edges “9” and “14” (which represent the remaining elements and the current element respectively).

Without the additional knowledge that the set of objects referred to by pointers abstracted by edge “9” must equal the set of objects referred to by pointers abstracted by edge “11” the analysis must conservatively assume that edge “11” and the node it refers to may remain in the heap graph. The result is that after the loop the analysis has (very) conservatively concluded that edge “11” represents pointers in `lo` which point to objects that have *non-null* `val` fields.

However, the dominance predicate enables the analysis to determine that if edge “9” does not exist (has an empty concretization) then edge “11” must also have an empty concretization and can be removed from the model. This of course means that the node that edge “11” refers to is removed as well. A similar series of deductions can be performed for edges “14” and “15”. Thus the dominance information enables the analysis to determine that there are no pointers in the list `lo` that refer to objects with *non-null* `val` fields. The result is shown in Figure 5(e) where the analysis has, as desired, determined that there are no longer any objects in `ls` that may refer to the same object as the variable `a`.

# 3 Dominance Relation

## 3.1 Concrete Heap and Concrete Dominance

We illustrate the notion of dominance on a core sequential imperative language with memory allocation and destructive updates. We assume our language is statically typed, and treat generic collections (lists and sets) and iterators over them as basic datatypes present in the language. The semantics of the language is 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 of the program.

We model the concrete heap as a labeled, directed multi-graph $(V,E)$ where each vertex $v \in V$ is an object in the store or a variable in the environment, and each labeled directed edge $e \in E$ represents a pointer between objects or a reference from a variable to an object. Each edge is given a label that is either an identifier from the program or an integer $i \in \mathbb{N}$ (the integers label the pointers stored in the collections). For an edge $(a,b) \in E$ labeled with $p$, we use the notation $a \xrightarrow{p} b$ to indicate that the object $a$ points to $b$ via the field name (or identifier) $p$.

A *region* of memory $\mathfrak{R}$ is a subset of the objects in memory, with all the pointers that connect these objects and all the cross-region pointers that start or end at an object in this region. Formally, let $C \subseteq V$ be a subset of objects, and let $P_i = \{p \mid \exists a,b \in C, a \xrightarrow{p} b\}$ and $P_c = \{p \mid \exists a \in C, x \notin C, a \xrightarrow{p} x \vee x \xrightarrow{p} a\}$ be

respectively the set of internal and cross-region pointers for $C$. Then a region is the tuple $(C, P_i, P_c)$. For a region $\mathfrak{R} = (C, P_i, P_c)$ and objects $a, b \in C$, we say $a$ and $b$ are *connected* in $\mathfrak{R}$ if they are in the same weakly-connected component of the graph $(C, P_i)$. Objects $a$ and $b$ are *disjoint* in $\mathfrak{R}$ if they are in different weakly-connected components of the graph.

**Concrete Dominance.** Given an edge $e \in E$ in the concrete heap we define the function *refTarget* to return the object that $e$ points to. Given a set of edges $E' \subseteq E$ in the concrete heap, we define the concrete dominance set (*domSetC*) of $E'$ as $\mathit{domSetC}(E') = \{o \in V \mid \exists e \in E', \mathit{refTarget}(e) = o\}$. Finally, given a region $\mathfrak{R} = (C, P_i, P_c)$ and some set of edges $E' \subseteq E$ in the concrete heap we define $E'$ dominates $\mathfrak{R}$ iff $\mathit{domSetC}(E') = C$.

Notice that using the *domSetC* definition we can define aliasing by $e_1 \equiv_{\text{alias}} e_2$ iff $\mathit{domSetC}(\{e_1\}) = \mathit{domSetC}(\{e_2\})$.

### 3.2 Abstract Heap and Abstract Dominance

Let $(V, E)$ be a concrete heap. An *abstract heap* [4, 13, 21] for $(V, E)$ is a labeled, directed multi-graph $(\hat{V}, \hat{E})$ such that there is an abstraction function $\alpha : V \to \hat{V}$ and for all $v \xrightarrow{p} v' \in E$, there is an edge $\alpha(v) \xrightarrow{p} \alpha(v')$ in $\hat{E}$. Conversely, we assume there are *concretization functions* $\gamma_v : \hat{V} \to 2^V$ and $\gamma_p : \hat{E} \to 2^E$ such that $(\alpha, \gamma_v)$ form a Galois connection [5], and $\gamma_p(\hat{u}, \hat{v}) = \{(u, v) \in E \mid \alpha(u) = \hat{u}, \alpha(v) = \hat{v}\}$. We extend $\gamma_v$ and $\gamma_p$ to sets of nodes and edges in the natural way. We call $(V, E)$ a concretization of $(\hat{V}, \hat{E})$. An abstract heap $(\hat{V}, \hat{E})$ can have many concretizations. We say an abstract heap $(\hat{V}, \hat{E})$ *represents* the concrete heap $(V, E)$ if the abstraction function $\alpha$ maps $(V, E)$ to $(\hat{V}, \hat{E})$.

Given an abstract heap $(\hat{V}, \hat{E})$, we define two dominance relations on it: *edge dominance*, which relates two abstract edges, and *node dominance*, which relates a set of abstract edges to an abstract heap 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 for all concrete heaps $(V, E)$ that are concretizations of $(\hat{V}, \hat{E})$, we have $\mathit{domSetC}(\gamma_p(e)) = \mathit{domSetC}(\gamma_p(e'))$, where $\gamma_p$ is the edge concretization function.

**Node Dominance** Given a set of edges $\hat{E}' \subseteq \hat{E}$ and an abstract node $n \in \hat{V}$, we define $\hat{E}'$ to *node dominate* $n$, written $\hat{E}' \propto n$, iff for all concretizations $(V, E)$ of $(\hat{V}, \hat{E})$, we have $\mathit{domSetC}(\gamma_p(\hat{E}')) = \gamma_v(n)$.

Proposition 1 summarizes a number of useful facts about edge and node dominance.

**Proposition 1** *Let $n, n'$ be abstract heap nodes, $e, e'$ abstract edges, and $E$, $E'$ sets of abstract edges.*

1. *If $\{e\} \propto n$ and $\gamma_p(e) = \emptyset$ then $\gamma_v(n) = \emptyset$ and $\gamma_p(e') = \emptyset$ for all $e'$ pointing to $n$.*
2. *If $e \equiv_{dom} e'$ and $\gamma_p(e) = \emptyset$ then $\gamma_p(e') = \emptyset$.*
3. *If $\{e\} \propto n$ and $\{e'\} \propto n$ then $e \equiv_{dom} e'$.*
4. *If $\{e\} \propto n$ and $|\gamma_p(e)| = 1$ then $|\gamma_v(n)| = 1$.*
5. *If $E' \subset E$ and $E' \propto n$ then $E \propto n$.*

We have restricted the definition of the dominance *equivalence* relation on edges to equality on the sets of objects that must be referred to in the concrete domain. More generally, we could define dominance subset relations, where the set of objects pointed to by the concretization of one edge is a subset of the set of objects pointed to by the concretization of the second. Our experiments with the edge dominance relation indicate that modeling the full range of relations over sets of edges provides only a small boost in accuracy at a substantial computational cost. Thus, we restrict the edge dominance relation to strict equality on single edges.

# 4 Extended Abstract Heap Domain

The heap model we work with is a variation on the classic heap graph where each node represents a region of the heap or a variable and each edge represents a set of pointers or a variable target. The nodes and edges are decorated with a number of instrumentation predicates to improve the accuracy of the analysis (see [16, 17] for more details).

## 4.1 Instrumentation Properties

The heap model uses a simple numeric abstraction, which has two values, exactly 1 and the range $[0,\infty]$ (written #), to represent the sizes of various components.

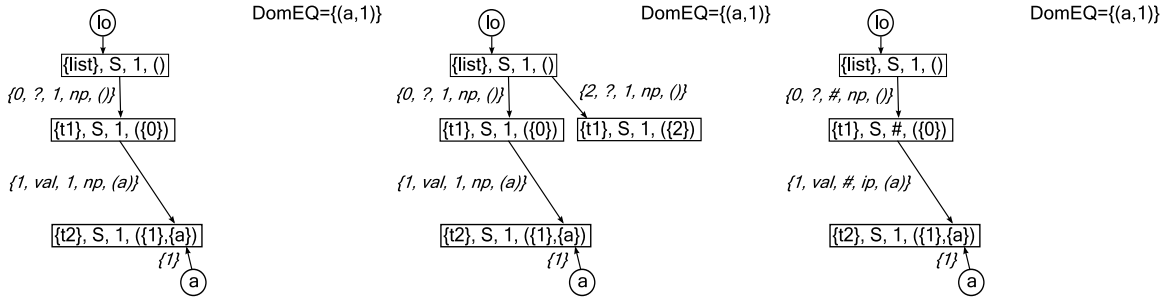
Each edge in the abstract graph represents a set of pointers. To track the labels of pointers that the edge represents the model uses the field identifiers declared in the program or a special offset that is used for identifying the pointers in a collection. These special offsets are ? (which represents an arbitrary set of pointers in the collection), @ (which identifies the single pointer that a given iterator refers to), B@ (which represents the set of pointers that come before the iterator in the collection iteration order), and A@ (which represents the set of pointers that come after the iterator).

To track the connectivity and *shape* of the region a given node abstracts, the analysis uses an *Abstract Layout* property. The possible layouts are *Singleton*, *List*, *Tree*, *MultiPath*, or *Cycle*. Of particular interest are the *Singleton* layout, which indicates that there are no pointers between any of the objects in the region, and the *List* layout, which indicates that each object has at most one pointer to another object in the region. The other shapes correspond to the standard definitions for Trees, Dags, and Cycles in the literature.

The heap model uses two properties to track the potential that two references can reach the same memory location in the region that a particular node represents.

The first property is for references that are represented by different edges in the heap model. Given two edges $e_1, e_2$ that are incoming edges to the node $n$, the predicate that defines *inConnected* in the abstract domain is: $e_1, e_2$ are *inConnected* with respect to $n$ if it is possible that $\exists r_1 \in \gamma_p(e_1) \wedge \exists r_2 \in \gamma_p(e_2) \wedge \exists a, b \in \gamma_v(n)$ s.t. ($r_1$ refers to $a$) $\wedge$ ($r_2$ refers to $b$) $\wedge$ ($a$, $b$ connected).

The second property is for the case where the references are represented by the same edge. To model this the *interfere* property is introduced. An edge $e$ represents interfering pointers if there may exist references $r_1, r_2 \in \gamma_p(e)$ such that the objects that $r_1, r_2$ refer to are connected. A two-element lattice, $np < ip$, *np* for edges with all non-interfering references and *ip* for edges with potentially interfering references, is used to represent the interference property.




(a) After First Analysis Iteration (b) After Second Analysis Iteration (c) Fixpoint and Done

Figure 2: Initializing List

## 4.2 The Heap Graph

Each node in the graph either represents a region of the heap or a variable. The variable nodes are simply labeled with the variable that they represent. The nodes representing the regions each contain a record that tracks the types of the concrete objects that a node represents (*types*), the number of objects (either 1 or #) that may be in the region represented by this node (*count*), and the abstract layout of a node (*layout*). Each node also tracks the connectivity relation between each pair of incoming edges. A binary relation *connR* is used to track the *inConnected* relation. Although the connectivity relation is a property of the nodes, for readability in the figures we associate the information with the edges. Finally, for each node we have a field *nodeDom*, which is a list of the sets of edges that dominate the node. Thus, each node is represented as a record of the form `[types layout count nodeDom]`.

As in the case of the nodes, each edge contains a record that tracks domain information about the edge. The *offset* component indicates the offsets (labels) of the references that are abstracted by the edge. The number of references that this edge may represent is tracked with the *maxCut* property. The *interfere* property 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 *connR* relation. Since the variable edges always represent single references and the offset label is implicitly the name of the variable the record simply contains the *connR* information or is omitted entirely if the *connR* relation is empty. To simplify the discussion of the examples each edge also has a unique integer label. The pointer edges in the figures are represented as records {`label offset maxCut interfere connto`}.

In order to track the edge dominance equality we use a global equivalence relation on the edges/variables, which tracks the dominance equality relations (this is labeled `domEQ` in the figures).

The local data flow analysis is performed using a *Hoare Power Domain* [18, 22] 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 [15] for more details.

### 4.3 Example 1

To clarify how these properties are combined and how they represent the heap properties that we are interested in we look at the heap model during the analysis of the first example program, Figure 1. To simplify the discussion we assume that the analysis knows that the loop body must be executed at least once and that on the first iteration the `v.val = a` branch is taken. These assumptions allow us to represent the result of analyzing each loop as a single graph instead of requiring additional graphs for special cases (e.g. the loop body is never executed).

In Figure 2(a) we show the graph that results from following the `v.val = a` path on the first pass of the analysis through the loop body. The variable `lo` points to a node which abstracts the `list` object and the variable `a` points to the node which abstracts an object of type `t2` that the elements in the list will reference. The outgoing edge from the `list` node (which is labeled by the number "0") is at the special offset `?`, represents at most one pointer (*maxCut* is 1) and since the edge represents at most one pointer the interfere property must be *np* (since the definition of interfence requires 2 pointers). This edge points to a region containing a single object of type `t1` (*count* is 1) and there are no inter-region pointers (the layout is *Singleton*). Finally, we note that since there is only one incoming edge (edge "0") and there are no inter-region connections, all of the live objects in the region must be pointed to by a pointer represented by edge "0". Thus we have edge "0" as a *dominator* for the node.

The edge representing the pointer created by the assignment `v.val = a` is represented by the edge with the label "1". Since the pointer represented by this edge is connected to the variable `a` we add the variable to the list of references that the pointers represented by this edge may be connected to. When we assign the `val` field to point to the same object as `a` we know that the pointer represented by this edge and the variable `a` must be dominance equal (thus the entry in the DomEQ equivalence relation) and since `a` dominates the target node we know the newly created edge also dominates the node.

Figure 2(b) shows one of the heap models that occur at the end of the second iteration of the analysis (in this case we have assumed that the `v.val = null` branch was taken). In this figure a new node to represent the newly allocated object has been added to the graph model and the edge "2" has been added to represent the pointer in the collection that refers to the newly created object. Again we know that since there is only one incoming edge ("2") to this node this edge must dominate the node.

After several iterations thought the loop body (and the application of the normalization operator, Section 5.5) the analysis will identify the graph in Figure 2(c) as representing the result of all possible executions of the loop body. There is the list variable `lo` which points to a node of type `list`. There is a single edge (with the label "0") from the node representing all of the pointers stored in the list. This edge has the special offset `?` (it represents all the pointers in the collection), represents an unknown number of pointers (*maxCut* is #), and all the pointers abstracted by this edge must refer to unique objects in the region that the target node abstracts (*interfere* is *np*).

The edge "0" points to a node which abstracts objects of type `t1`, there may be many objects in this region (*count* is #), and there are no inter-region pointers (the layout is *Singleton*). Finally we know that the region is dominated by the pointers represented by the edge "0". The pointers stored at the `val` offset are represented by the edge "1". Since the pointers represented by this edge may point to connected portions of the target region (in fact we know they must alias) the edge *interfere* property is *ip*. Since the pointers in the "1" edge may be connected to the target of the variable `a`, the variable `a` is included in the connectivity relation for the edge (and by symmetry the edge "1" is in the connectivity relation for the variable `a`).

Finally, the region containing the `t2` objects is dominated by the edge "1" as well as by the variable `a`. Again by using our facts about the dominance relation we know that edge "1" is dominance equal to `a` and thus this is included in the *DomEQ* relation.

# 5 Model Operations

Now that we have defined the heap domain that we are working with and seen how the properties in the abstract domain represent various properties of the concrete heap we define the domain and simulation operations for performing the heap analysis. The domain operations are *safe* approximations of the concrete program operations (thus the analysis safely approximates the semantics of the program). For brevity we omit proofs of these safety properties (which rely on basic case-wise reasoning about the graph structure and the instrumentation properties).

## 5.1 Tests

When performing tests we generate one version of the abstract heap for each possible outcome. For a nullity test of a variable we create one model in which the variable *must* be *null* and one model in which the variable is *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). Finally, we perform the focus operation, Section 5.5, on any nodes that may have been affected. Algorithm 1 gives the code for this operation.

**Algorithm 1**: Assume Var Null

**input** : graph $g$, var $v$
$E_s \leftarrow$ all edges that represent the targets of $v$;
$E_{\text{null}} \leftarrow \emptyset$;
**for** *edge* $e \in E_S$ **do**
    $E_{\text{null}} \leftarrow E_{\text{null}} \cup \{e' | e' \equiv_{\text{dom}} e\}$;
    $n \leftarrow$ the target node of $e$;
    **if** $e \propto n$ **then** $E_{\text{null}} \leftarrow E_{\text{null}} \cup \{$all incoming edges to $n\}$;
$T_n \leftarrow \{n | \exists e \in E_{\text{null}}, e$ is an in edge to $n\}$;
**for** *edge* $e \in E_{null}$ **do**
    $g$.removeEdge($e$);
**for** *node* $n \in T_n$ **do**
    $g$.focusNode($n$);

Similarly when the analysis assumes that the `isValid` test is *false* it assumes that the edge representing the current pointer the iterator refers to (edge with label `@`) and the edge which represents all pointers that are stored after the current iterator location (with label `A@`) have empty concretizations. Then, as with the nullity test, we use the dominance information to identify other edges and nodes that must also have empty concretizations and remove all of these from the heap model.

In the case of an equality comparison between two *non-null* variables `x == y` we can strengthen the information we have in the models that represent the true and false branches. For both the *true* and *false* cases we begin by ensuring that `x` and `y` have unique targets by merging (Section 5.3) all the possible target nodes into a single node. In the case where we assume this test returns true, if `x` and `y` refer to different nodes in the graph then we can assume this flow path is infeasible. If not, we add the fact that $\texttt{x} \equiv_{\text{dom}} \texttt{y}$ to the model. In the case where we assume this test must be false then we can check if the relation $\texttt{x} \equiv_{\text{dom}} \texttt{y}$ holds and if it does we can rule this path out as being infeasible.

## 5.2 Assign, Load, Store

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

**Load.** The load operation (`x = y.f`) is more interesting as we may need to deal with ambiguous targets of `y.f` (there may be multiple targets of `y` and each of these targets may have multiple outgoing edges with the label `f`) and we may need to refine the node (transform a summary node into a more explicit representation, Section 5.4) referred to by `y.f` before we actually update the target of the variable `x`. For simplicity our current implementation (Algorithm 2) resolves the problems of ambiguous variable targets by merging them into a single node/edge.

**Algorithm 2**: Load

```
input : graph g, var x, var y, field f
if y has multiple targets then merge all the targets into a single node;
if multiple edges at y.f then merge all edges into a single edge;
g.refineLoad(the unique target of y.f);
if y.f is the null value then
    nullify x;
    return ;
e ← the unique edge at y.f;
assign x to refer to the target of e;
if e.maxCut = 1 then
    set x dominance equal to e;
    if e dominates the target node then set x dominates target node;
```

**Store.** The store operation (`x.f = y`) is simpler than the load operation as it does not need to deal with the resolution of ambiguous field targets or the refinement of target nodes. As with the load operation we begin by ensuring that there is a unique target node that `x` refers to. The analysis then determines if the location at `x.f` can be strongly updated. If the node ($n$) referred to by `x` is of *count* 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 for each possible target of `y`. Just as in the case of the loads these new edges are each dominance equal to `y` and if `y` dominates a node then so does the newly created edge.

### 5.3 Merge/Split Nodes and Edges

In several domain operations we need to transform a portion of the heap graph into a single node (to remove ambiguity or to ensure a finite domain) and in others we want to transform single nodes into a more explicit sub-graph representation (so that we can accurately simulate the effects of various program statements). For this paper we focus on the effects the merge and split operations have on the dominance properties and only informally mention how the other properties are handled as needed in the figures. More precise definitions of the merge/split operations can be found in [16].

**Merge Nodes.** From the definition of dominance equality it is clear that when two nodes $(n_1, n_2)$ are replaced by a summary node $(n_s)$ any edges that were dominance equal before the summarization are dominance equal after the summarization. For the node dominance relation we can infer that if some set of edges $E_1 \propto n_1$ and some other set of edges $E_2 \propto n_2$ then $E_1 \cup E_2 \propto n_s$.

**Merge Edges.** The summarization of two edges $(e_1, e_2)$ into a single edge $(e_s)$ is handled by conservatively assuming that all dominance equality relations which involve either $e_1$ or $e_2$ no longer hold and replacing each occurrence of $e_1$ or $e_2$ in the node dominance set with $e_s$.

**Split Nodes.** The node split operation does not affect the edge dominance equality relation so we only need to update the information on node dominance. To do this we restrict the relations that held for the original node $(n_o)$ to the set of edges that are incident to the newly created node $(n_i)$. That is, if some set of edges $\{e_1, \ldots, e_k\} \propto n_o$ (note that by definition $n_i$ represents a subset of the objects represented by $n_o$) then $\{e_1, \ldots, e_k\} \propto n_i$, thus whatever subset of $\{e_1, \ldots, e_k\}$ refers to $n_i$ must dominate the $n_i$.

**Split Edges.** When splitting an edge $(e_s)$ into multiple new edges $(e_1, \ldots, e_k)$ we conservatively assume that all dominance equality relations which involve $e_s$ no longer hold and we replace each occurrence of $e_s$ in the node dominance sets with $e_1, \ldots, e_k$.

### 5.4 Refinement

The refinement 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.

Our refinement operation on *Singleton* nodes is restricted to handle the following cases 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 *inConnected* relation.

- If there is a single edge $e_c$ that is *connected* to every other edge and all other edges are pairwise *disjoint*.

While these two cases are limited our experimentation with more complex partitioning schemes indicates that these cases cover a majority of the situations encountered and the improvements in accuracy from using more complex approaches are minimal.

In the first case where we have several disjoint partitions we define the *refineDisjointEdges* method, which creates a new node for each partition. In the second case we define the *refineSingleConn* method, which creates a new node for each of the edges except $e_c$ and then splits $e_c$ so that it points to each of the newly created nodes (and thus the possibility that it is connected to any one of the other edges is preserved).

There are two uses for the refinement operation. The first is when normalizing the heap graph, Algorithm 4; the second is when simulating the load operation, Algorithm 2. Algorithm 3 shows the code for the refinement operation. In this algorithm we first try to split all the disjoint partitions. Then we attempt to apply the second *Singleton* splitting rule to the partition. Finally, if possible, we apply the *List*/*Tree* refinement operations as described in [16]. Since we do not need to refine lists or tree nodes in our examples we omit a detailed description of this operator but for completeness it is included in the algorithm.

**Algorithm 3**: Refine Load

**input** : graph $g$, node $n$
$E_p \leftarrow$ partition of the incoming edges to $n$;
**if** $E_p$ *has 2 or more partitions* **then**
    $g$.refineDisjointEdges($n, E_p$);
    **for** *each newly created node* $n'$ **do**
        $g$.focusNode($n'$);
*allcp* $\leftarrow \exists e'$ s.t. all edges *conn* to $e'$;
*owdisjoint* $\leftarrow$ all edges except $e'$ are pairwise *disjoint*;
**if** *allcp* $\wedge$ *owdisjoint* **then**
    $g$.refineSingleConn($n, e'$);
    **for** *each newly created node* $n'$ **do**
        $g$.focusNode($n'$);
**if** *n.layout* List *or* Tree *and single in edge of size 1* **then**
    $g$.refineListTree($n$);
    **for** *each newly created node* $n'$ **do**
        $g$.focusNode($n'$);

Figure 3 shows the result of initializing the iterator `i` and refining the heap graph from Figure 2. We initialized the iterator `i` to refer to the first element of the list `lo`, splitting the edge with the label `?` into an edge representing the single element referred to by the iterator (labeled `@`) and an edge representing all the other pointers in the collection (which must come after the current element in iteration order `A@`). We have also split the node which represents the `t1` objects into a node representing the object targeted by the `@` edge and a node representing the objects targeted by the `A@` edge.

Since we know that the `?` edge dominated the node that was split we know that the edges with the `@` and the `A@` labels must dominate the resulting nodes (e.g. edge "3" dominates the node it refers to and edge "4" dominates the node it refers to). Further we know that the edge with the `@` label represents a single pointer (since it represents the unique element in the collection that the iterator refers to) and, since it dominates the

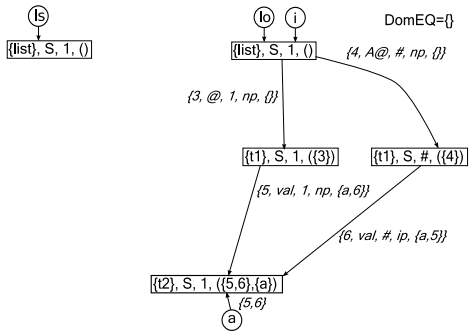



Figure 3: Iterator Begin and Refined Contents

node it refers to, that node must represent at most one object.

When splitting the node which represents the `t1` objects we split the outgoing edge as well: the edge with the label "1" from Figure 2 has been split into the edges with the labels "5" and "6". Since the original edge (the edge with the label "1") abstracted potentially interfering pointers we must assume that the resulting edges could (and in this case do) refer to the same object. Again since the original edge dominated the node it ended at, the edges resulting from the split must also dominate the node (thus we replaced the $\{1\}$ in the node dominance set with the entry $\{5,6\}$).

## 5.5 Normal Form

The normal form for the heap graphs enables efficient equality comparison and provides a simple method for defining the heap join operation.

**FocusNode.** The *focusNode* method is used to infer and make explicit information that is implicitly represented by various combinations of the domain properties. Given a node $n$, a set of incoming edges $E^i = \{e^i_1, \ldots, e^i_m\}$ and a set of outgoing edges $E^o = \{e^o_1, \ldots, e^o_n\}$ the focus operator updates the nodes and edges using the following heuristics:

- If $n.layout = Singleton$ then $n.nodeDom \leftarrow n.nodeDom \cup \{E^i\}$.
- If $\{e^i_x\} \propto n \wedge \{e^i_y\} \propto n$ then $e^i_x \equiv_{\text{dom}} e^i_y$.
- If $\exists e^i_x, e^i_x.maxCut = 1 \wedge \{e^i_x\} \propto n$ then $n.count \leftarrow 1$.
- If $n.count = 1$ then for each $e^i_x \in E^i$ if $e^i_x.interfere = np$ $e^i_x.maxCut \leftarrow 1$.
- If $n.count = 1$ then for each $e^o_x \in E^o$ if $e^o_x.offset \notin \{\texttt{?}, \texttt{B@}, \texttt{A@}\}$ then $e^o_x.maxCut \leftarrow 1$ and $e^o_x.interfere \leftarrow np$.
- If $E^i = \emptyset$ then $n$ is irrelevant and should be removed.

**Ambiguous Edges.** When comparing two graphs $g_1, g_2$ for equality we need to compute a graph isomorphism between the two graph models. Since the comparison operation must be fast we must minimize backtracking. Starting from the variables we want to be able to pick an edge in $g_1$ and match it unambiguously (as much as possible) to an edge in $g_2$. The easiest way to do this is to ensure that each out edge for a given node has a unique offset. However, there are many cases where this can lead to overly aggressive merging and a loss of important information (in our subset example, Figure 4(f), the two `?` edges from `lo`).

To avoid this we relax the uniqueness requirement to include some information about the target of the edge as well as the edge offset. That is, two edges are ambiguous if they have the same offset and their target nodes are *equivalent* under some relation. We heuristically choose to use equality of the multi-sets of incoming edge offsets. Thus, in the subset example one of the nodes has the incoming offsets $\{?\}$ and the other node has the incoming offsets $\{?, ?\}$ and thus they are not joined when the heap graph is normalized.

**Recursive Sections.** The idea behind the recursive definition is to identify sections of the heap that are regular and can be merged with a minimal loss of information while preserving the important transition points in the heap. First we define what it means for two regions to be recursive. Given a type system it is trivial to identify the potentially recursive types. Simply merging all recursive types is too aggressive as it can lead to the loss of information on the transitions between recursive and non-recursive segments of the heap. To prevent this loss of information we introduce safe nodes which represent the important transition points in the heap. A node $n$ is safe if either of the following holds:

- There is a variable pointing to $n$.
- There is an incoming edge $e = (n', n)$ and $\forall t \in n.\text{types}$, $\nexists t' \in n'.\text{types}$, s.t. $t$ and $t'$ are recursively related.

Now we can define what it means for two nodes $n_1$, $n_2$ to be recursive:

- There is an edge connecting $n_1$, $n_2$, $\exists e = (n_1, n_2)$.
- Neither of $n_1$ or $n_2$ is safe.
- $\exists n'$ s.t. there is a path in the graph from $n_2$ to $n'$ that does not contain any safe nodes and $n_1.types \cap n'.types \neq \emptyset$.

Based on this definition we see that if a recursive section is broken by an interesting transition (either a variable or a transition from a non-recursive component is pointing into it) then the recursive cycle is left expanded. Thus, the analysis can accurately track the relations between variables and the transition points in the heap.

**Normalize.** The normalization routine is the repeated application of several steps until there are no longer any changes in the heap graph model, Algorithm 4. In the algorithm the nodes are processed in topological order to speed convergence.

**Algorithm 4**: Normalize Graph

**input** : graph $g$
Remove all unreachable nodes from $g$;
**while** *g is changing* **do**
  **while** $\exists$ *node n s.t. n the focusNode operation can be applied* **do**
    $g$.focusNode($n$);
  **while** $\exists$ *node n s.t. n has disjoint in edge partitions* **do**
    $g$.refineDisjointEdges($n$, partition of the in edges to $n$);
  **while** $\exists$ *nodes* $n, n'$ *s.t.* $n, n'$ *are recursive* **do**
    $g$.mergeNodes($n, n'$);
  **while** $\exists$ *node n s.t. n has ambiguous edges* $e, e'$ **do**
    $g$.mergeEdges($e, e'$);

**Heap Graph Equality.** To compare two graphs for equality we fist compute a graph isomorphism between the two heap graphs (this is efficient since the normal form effectively eliminates any ambiguity in the matching, in our experimental results no equality comparison encountered required backtracking and had at most one isomorphism). For each possible isomorphism we do a pairwise comparison of each node and edge property for each pair of nodes/edges that are related under the isomorphism.

**Heap Join.** To join two heaps we first forget all the dominance information in both heap models. Then the algorithm computes the union of the two graphs and merges the variable nodes. Finally, the resulting graph is normalized. This results in only the dominance information that is implied by the focus operation being maintained. While this loss of dominance information could be avoided by a more complex join operation our experimental results indicate that in conjunction with the *Hoare Power Domain* the actual information loss from the use of this simple join is minimal.

# 6 Examples

Now that we have defined the data flow and program simulation operations we look at how they work with the dominance properties to enable the analysis to accurately model the effects of the remaining two code segments from Figure 1.

## 6.1 Example 2

Figure 4 shows part of the analysis of the second example code segment from Figure 1. In Figure 4(a) the analysis performed the load operations required by the statement `i.get().val` and assigned the value to the variable `t`. Since the `val` edge (edge "5") in this path represents at most one pointer (*maxCut* = 1) we know that the dominance set of the edge "5" and the variable `t` must be the same (thus the *DomEQ* relation has been updated with this fact).

Figure 4(b) shows the abstract heap after assuming the *false* branch was taken, indexing the iterator and loading the new value into `t`. The model shows that the object from the first iteration has been added into the sublist (`ls`) by adding the edge "7" to represent the pointer from the list (`ls`) to the node representing

the object that has been added. Since we know this pointer must refer to the same object as the pointer abstracted by the edge "3" we know their dominance sets are equal, thus the dominance set of "7" $\equiv_{\text{dom}}$ "3" and additionally that edge "7" dominates the node. Indexing the iterator results in the `@` edge being relabeled `B@` (the pointer is now before the current iterator position) and a new `@` edge (edge "8") has been split out to represent the pointer that the iterator refers to after the *advance* operation.

Figure 4(c) shows the result of assuming that the test against `null` returns *true*. In this case we learn that `t` is `null` which implies that its dominance set is empty. Based on the dominance equality information we have this implies that edge "9" also has an empty concretization (it is `null`).

The result of again indexing the iterator is shown in Figure 4(d). If we were to proceed through the loop several more times we would reach the state shown in Figure 4(e). In this figure we see that there may be many elements in the sublist and many elements that are not added to the sublist (represented by the edges with the `B@` label, "3" and "8" 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 "3" must have been added to the sublist and thus is also referred to by a pointer represented by edge "7". This implies that edge "7" is dominance equivalent to edge "3" and thus both edges "3" and "7" must dominate their target node.

If we assume the `isValid` test returns false then the `@` and `A@` edges must be null and can be eliminated, Figure 4(f). We know that set of edges {"5", "6", "10"} dominate the node and we now assume that the concretization of "6" and "10" are both empty, thus it must be the case that "5" alone dominates the node. As desired the analysis has determined that all the objects with a non-null `val` field have been stored in the sublist `ls`.

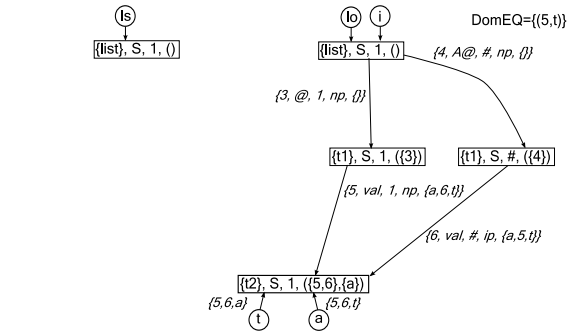
## 6.2 Example 3

Figure 5 shows how the analysis models the strong update of every element in the sublist (the third example code segment) and how the dominance relation enables the analysis to determine that this update implies there are no more elements in the original list that have *non-null* `val` references.

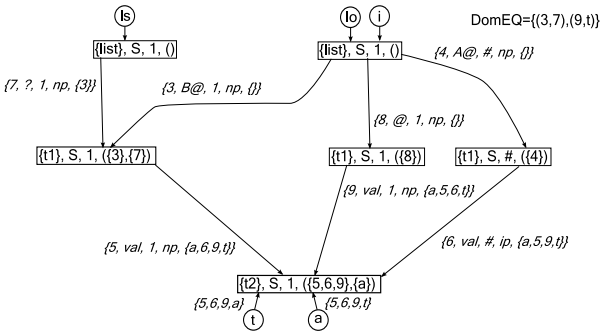
Figure 5(a) shows the state of the abstract heap after the initialization of the list iterator. In this figure we have drawn the edges representing pointers in the original list as dotted lines to help clarify which edge belongs to which collection. It is important to note that when the node representing the contents of the list `ls` was refined the edges from both lists were split and:

- Since edge "3" and edge "7" both dominated the node the resulting split edges ("9", "11"), ("10", "12") dominate their respective split nodes. The focus operation can then infer that these newly created edges are also dominance equal.
- The focus operations infers that since edge "12" is *np* and it is equivalent to "10" which must be of *maxCut* = 1 then edge "12" must also be of *maxCut* = 1.

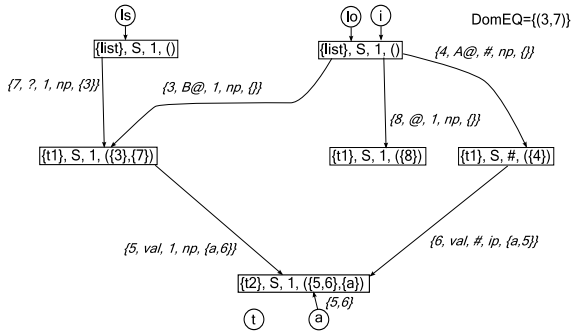
Figure 5(b) shows the result of nullifying the `val` target for the object referred to by the iterator. Note that the analysis was able to strongly nullify the `val` field since the node has a *count* = 1, indicating that the node represents at most a single object. Before the assignment we have the edge set {"13","14"} as dominating their target node. When the assignment occurs we nullify edge "14", thus after the assignment



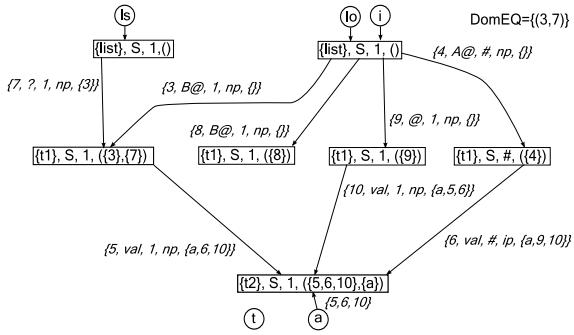

(a) Temp Var Loaded



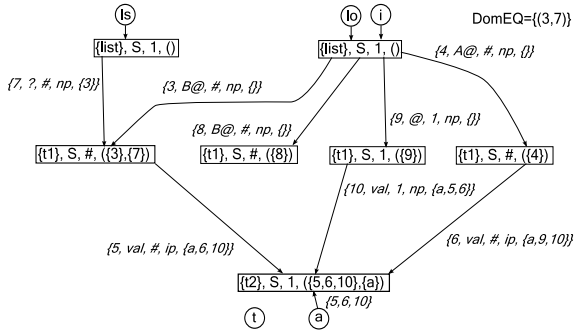

(b) Added To Subset and Indexed



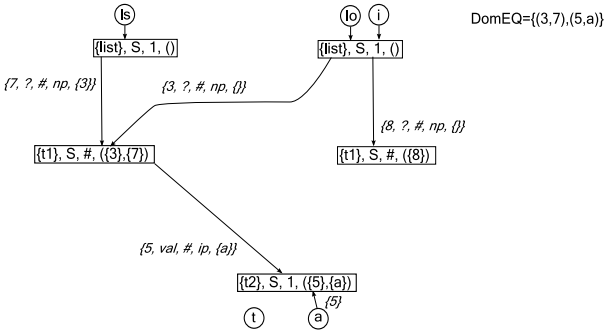

(c) Assert Temp Var is Null




(d) Not Added to Subset and Indexed




(e) Fixed Point of Loop Analysis




(f) IsValid False Exit Loop

Figure 4: Computing the sublist of all elements that refer to the same object as a

we cannot be sure if edge "13" still dominates the node so we must conservatively assume it does not. Figure 5(c) shows the result of indexing the iterator and again based on the dominance relations of the edges being split the analysis is able to infer the *maxCut* and dominance relations of the newly created nodes/edges.

Figure 5(d) shows the state of the abstract heap after several iterations of analyzing the loop body. In this figure we see that the there are now potentially many pointers that come before the current iterator position (`B@`) in the list `ls` (all of which point to objects with a *null* `val` field) and that the edges representing the current iterator location (`@`) and the set of pointers that come after the current iterator position (`A@`) dominate their respective target nodes. This abstract heap is also the fixed point of the loop analysis.

Finally, Figure 5(e) shows the result of interpreting the `isValid` test as *false*. The assumption that the edges "9" and "14" have empty concretizations implies that edges "11" and "15" also have empty concretizations (based on the dominance equal relation). After the test (and the removal of the edges/nodes) there are no longer any pointers to the object referred to by the variable `a`. Thus, the loop has successfully strongly updated all of the objects in the sublist `ls` and this strong update information has also been reflected in the original list `lo`.

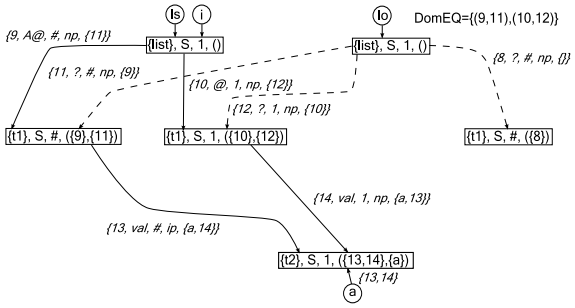
# 7 Experimental Evaluation

The proposed approach has been implemented and the effectiveness and efficiency of the analysis have been evaluated on a number of examples (some micro-benchmarks and a number of parallelization kernels). For the micro-benchmarks we used two small list and tree manipulation routines as well as the example presented in this paper. The list and tree benchmarks call a range of list/tree procedures (copy, search, insert, remove, filter, recursive subtree swap) from several calling contexts (data elements shared and unshared at call sites) on singly linked lists and binary trees. The parallelization kernels are a variation of the Jolden [2] suite. The Jolden suite contains pointer-intensive kernels that make use of recursive procedures, inheritance, and virtual methods. The implementation in [2] is a mostly verbatim translation of the original C Olden benchmarks [3], which have some known issues [25]. We modified the suite to use modern Java programming idioms and addressed major concerns raised in the literature about the suite.

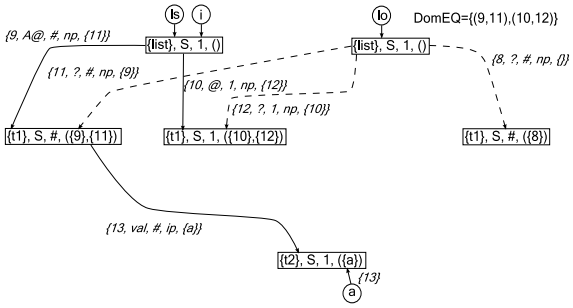
The analysis algorithm was written in C++ and compiled using MSVC 8.0. The analysis as well as the parallelization benchmarks were run on an Intel dual-core PentiumD 2.8 GHz machine with 1 GB of RAM (although memory consumption never exceeded 20 MB for any of the benchmarks).

To assess the accuracy of the analysis results we report, in the *ShapeT* column of Table 7, if our algorithm correctly determined the shape information for the data structures created by the programs. To compare these results with other work on shape analysis we list the most accurate results from the related literature in the *ShapeO* column. We use three categories for the accuracy of a shape analysis. Y(es) means the analysis was able to provide shape 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 of a substantial portion of the heap data structures.

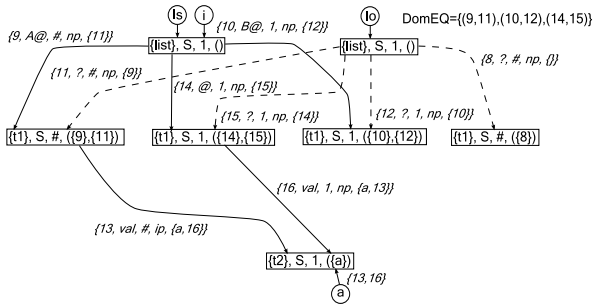
In Table 7 the results for the micro-benchmarks are separated from the other results. Since we have not been able to run a number of the existing heap analysis techniques on the micro-benchmarks or the example used in this paper we report *NA* for the *ShapeO* column. However, based on our hand simulation of the various analysis techniques in the literature [6, 20, 21, 14, 11, 8, 1, 10, 9] we believe that all of these



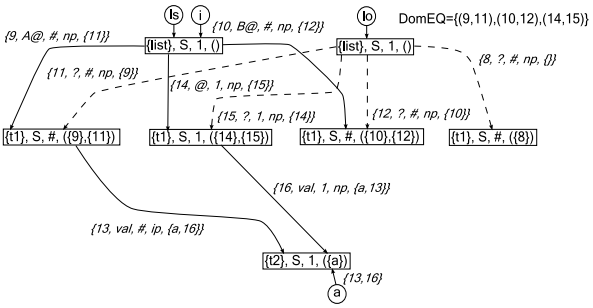

(a) Initialize Iterator Split Edges



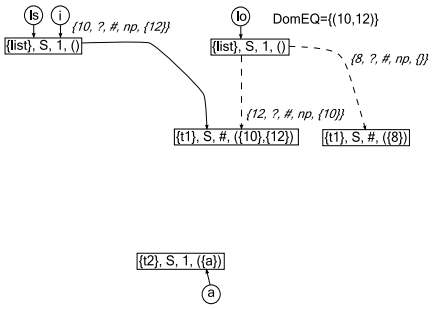

(b) Nullify `val` field




(c) Index Iterator




(d) Fixed Point of Loop




(e) IsValid False Exit Loop

Figure 5: Update `val` field of all elements in the sublist

| Benchmark | Stmts | Methods |
|---|---|---|
| example | 52 | 4 |
| tree | 247 | 11 |
| list | 250 | 12 |
| bisort | 260 | 13 |
| em3d | 333 | 13 |
| mst | 457 | 32 |
| tsp | 510 | 13 |
| perimeter | 621 | 41 |
| health | 643 | 21 |
| voronoi | 981 | 63 |
| power | 1352 | 29 |
| bh | 1616 | 58 |
| Overall | 7322 | 310 |

Figure 6: Benchmark Statistics

approaches are unable to satisfactorily handle the example program.

We used the sharing information from the shape analysis to parallelize the benchmarks by using multiple threads in loops and calls [7, 19, 12] to exploit the two cores of the test machine. The speedup column in table 7 shows the results. In 7 of 9 benchmarks we achieve near-optimal speedup. The mst and perimeter benchmarks cannot be parallelized using the information provided by the analysis and the shape-driven parallelization techniques since they build and traverse recursive cyclic structures (a tree with parent pointers; and an unstructured graph), which our analysis does not currently model accurately (the features are represented as generic cycles and are not expanded by the refinement operation).

The Olden benchmark suite was introduced in 1995 as a challenge problem to assess how parallelizing compilers and parallel architectures would be able to compile and execute programs that make extensive use of dynamically allocated data structures. In the intervening years a substantial number of papers have used benchmarks from this suite to evaluate proposed parallelization techniques. Our survey of the literature indicates that despite interest in the problem, the present work (to the best of our knowledge) is the only analysis that has identified useful parallelization opportunities in two of the benchmarks (em3d and the corrected version [25] of health), and the only heap analysis (to date) that can identify the heap properties needed to successfully parallelize four of the benchmarks (em3d, health, voronoi, and bh).

Although the benchmarks are in fact smallish kernels, our results represent the only general-purpose heap analysis technique that is able to address a number of the features in this set of challenge problems (which use a range of heap structures from lists to trees to bipartite graphs and perform a range of destructive operations on these structures). Given the speed with which the analysis is able to produce the information needed for the parallelization and the large speedup that is obtained in the benchmarks (1.60 over all of the benchmarks and 1.76 if we exclude the benchmarks, mst and perimeter, which utilize properties that we have not yet added to the analysis), we are pleased with the accomplishment.

| Benchmark | Time | Speedup | ShapeT | ShapeO |
|---|---|---|---|---|
| example | 0.05s | NA | Y | NA |
| tree | 0.52s | NA | Y | NA |
| list | 0.10s | NA | Y | NA |
| bisort | 0.41s | 1.72 | Y | Y |
| em3d | 0.09s | 1.75 | Y | N |
| mst | 0.11s | 1.00 | Y | Y |
| tsp | 0.15s | 1.84 | Y | Y |
| perimeter | 1.35s | 1.00 | P | Y |
| health | 1.22s | 1.76 | Y | N |
| voronoi | 1.93s | 1.68 | Y | N |
| power | 0.17s | 1.93 | Y | Y |
| bh | 3.61s | 1.75 | P | N |
| Overall | 9.71s | 1.60 | 7/2/0 | 5/0/4 |

Figure 7: ShapeT is the shape results for the analysis in this paper ShapeO is the best reported in the literature. The notation 7/2/0 indicates the shape was correctly determined for 7 of the Olden based benchmarks, partially for 2, and there were no benchmarks where the analysis failed to provide useful information.

# 8 Conclusion

This paper introduced the concept of *dominance*, which enables predicate decoration approaches to model many of the sharing and dependence properties that are required to track how objects are stored in multiple collections (or data structures), and how modifications to the contents in one collection affect other collections that have overlapping sets of data elements. Being able to model these properties is critical to ensuring that programs which build, share, and modify non-trivial data structures can be accurately analyzed. Our generic formulation of the *dominance* relation shows that it is fundamentally a generalization of the concept of aliasing and provides the ability to model a range of important sharing properties.

The example in this paper provided a demonstration of how dominance information enabled our heap analysis to discover several important facts about how a set of objects was shared by two lists; we also demonstrated that the dominance information enables the analysis to determine how the modification of elements in the sublist affected the contents of the original list. Our experimental results using the dominance relation are very positive. The analysis was able to analyze benchmarks that build and manipulate a variety of data structures. In addition to small list/tree manipulation benchmarks, our benchmark set includes a number of kernels that were originally designed as challenge problems for automatic parallelization. Our heap analysis was able to provide sufficient information to successfully parallelize all but 2 of benchmarks we examined, including several that (to the best of our knowledge) cannot be successfully parallelized using other proposed shape analysis methods.

We believe that the dominance property introduced in this paper is a simple concept that captures fundamental properties of how objects can be shared and that the definition provided can be adapted to a variety of heap models. We thus believe that the proposed approach meets to a considerable extent, if not completely, a challenge posed in [1], where it is stated “[Sharing] as far as we know, is beyond current automatic shape

analysis." and continues "In general real-world systems programs contain much more complex data structures than those usually found in papers on shape analysis and handling the full range of these structures efficiently and precisely presents a significant challenge." The success of the proposed approach on our example program and the benchmarks is such an illustration.

Encouraged by the range of program structures that our analysis is capable of handling, the utility of the information provided by our model and the speed with which this information was computed we are working on applying the analysis 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] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.

[3] M. C. Carlisle and A. Rogers. Software caching and computation migration in Olden. *J. Parallel and Distributed Computing*, 1996.

[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] 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.

[7] R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *CC*, 1998.

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

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

[10] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.

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

[12] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE TPDS*, 1(1), 1990.

[13] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In *POPL*, 1979.

[14] T. Lev-Ami, N. Immerman, and S. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.

[15] M. Marron, M. Hermenegildo, D. Stefanovic, and D. Kapur. Efficient context-sensitive shape analysis with graph based heap models. Tech. report, CS Dept., Univ. of New Mexico, Mar 2007.

[16] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.

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

[18] G. D. Plotkin. A powerdomain construction. *SIAM J. Computing.*, 1976.

[19] R. Rugina and M. C. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPOPP*, 1999.

[20] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.

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

[22] M. B. Smyth. Power domains and predicate transformers: A topological view. In *ICALP*, 1983.

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

[24] R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *CC*, 2000.

[25] C. Zilles. Benchmark health considered harmful. In *Computer Arch. News*, 2001.