A Static Heap Analysis for Shape and Connectivity: Unified Memory Analysis: The Base Framework

Mark Marron, Deepak Kapur, Darko Stefanovic, and Manuel Hermenegildo

University of New Mexico
Albuquerque, NM 87131, USA,
{marron, kapur, darko, herme}@cs.unm.edu

Abstract. Modeling the evolution of the state of program memory during program execution is critical to many parallelization techniques. Current memory analysis techniques either provide very accurate information but run prohibitively slowly or produce very conservative results. An approach based on abstract interpretation is presented for analyzing programs at compile time, which can accurately determine many important program properties such as aliasing, logical data structures and shape. These properties are known to be critical for transforming a single threaded program into a version that can be run on multiple execution units in parallel. The analysis is shown to be of polynomial complexity in the size of the memory heap. Experimental results for benchmarks in the Jolden suite are given. These results show that in practice the analysis method is efficient and is capable of accurately determining shape information in programs that create and manipulate complex data structures.

1 Introduction

Research on automatic thread level parallelization techniques makes extensive use of the *shape* [4,16] of data structures in memory. As an example, in [6] Ghiya used a notion of shape to enable the extraction of *foreach* thread-level parallelism from common heap-based data structures. The notion of shape and sharing can also be used to enable the parallelization of recursive algorithms [15,8]. In many programs the availability of accurate shape information and the application of these two transforms enables the extraction of a substantial portion of the available parallelism. Unfortunately, the applicability of these parallelization techniques has been limited by the difficulty of performing shape analysis with the required level of accuracy. The advent of commonly available multi-processor systems, the slowing of improvements in single threaded processor performance and the increasing use of object oriented languages (which make extensive use of heap allocated memory and rich pointer structures) have renewed interest in shape driven parallelization techniques.

This paper uses an abstract interpretation framework for performing static analysis of programs and introduces a graph based abstract heap model that can represent all the information on aliasing, shape and logical data structures [10] that are required to perform thread level parallelization transformations. Along with accurately representing the required information on shape, aliasing and logically related regions, the framework enables accurate simulation of the evolution of these properties through many

important program idioms, e.g. sorting, copying, destructive reversal, and element insertion/deletion. A theoretical analysis of the runtime and our experience running the method on the Jolden benchmarks indicates that the technique is accurate, efficient and scalable.

A key factor in achieving these results is the use of a novel technique for undoing the *summarization* of information (the analysis must use a bounded representation to summarize unbounded recursive structures). For efficiency, it is important to make the summary representations as compact as possible. However, this summarization may lead to the loss information which is needed to accurately simulate the effect of program statements on the heap model. Seminal work on heap analysis [16] introduced the notion of refinement but the proposed technique results in an exponential runtime (due to the desire to model the program with maximal precision). This paper presents a technique for refinement that sacrifices some accuracy in less common cases to ensure that the worst case exponential time is avoided and that the method is fast in practice.

1.1 Related Work

There are two research activities closely related to the work presented in this paper. One is the research on shape analysis by Ghiya [4] and the second is the TVLA (3-valued Logic Analysis) framework introduced by Reps, Sagiv and Wilhelm [16].

Ghiya's method is efficient and is able to model simple structures in programs that do not use destructive updates. In this work shapes are defined on the entire portion of the heap that is reachable from a variable. This implies that any extraneous sharing of the heap (due to the use of the *singleton* design pattern or sharing of data that is unrelated to the computation that is being parallelized) will result in very conservative results. Further, the analysis is unable to strongly update heap based storage. Thus, the analysis is unable to accurately handle situations where a section of the heap, through destructive updates, temporarily takes on a more general shape and then returns to the original shape (e.g. Tree $\rightarrow$ DAG $\rightarrow$ Tree).

The TVLA framework is very powerful and highly expressive in the sense that it can be used to represent the shape and aliasing properties needed for extracting thread-level parallelism. In addition to being expressive enough to model the relevant program properties, the TVLA framework is able to model the evolution of these properties through destructive updates [17,12] and is able to model shape on a more localized basis. In the TVLA framework destructive updates are handled by allowing the summary representations of recursive data structures to be refined into a number of distinct objects which can be strongly updated. Since there may be ambiguity about how to refine the summarization TVLA enumerates all the possibilities. This results in a potentially exponential runtime and in practice leads to large analysis times. There has been work on reducing the cost of running TVLA or restricted variations of the method [20,7] but they do not eliminate the exponential worst case time and have had mixed results in reducing the execution time on various benchmarks.

To compare the proposed method with existing shape analysis techniques we look at some simple examples with lists and with benchmarks from the Jolden suite. The list benchmarks demonstrate that the proposed method handles simple heap based structures accurately and that in practice it is over an order of magnitude faster than existing

analysis techniques of similar precision. The Jolden tests indicate that the proposed analysis method can determine the correct shape for the majority of heap based data structures even in programs that build and manipulate relatively complex data structures while maintaining an acceptable analysis time.

2 Concrete Domain

Our analysis works on the strongly-statically typed, single-inheritance, thread-free, exception-free, object-oriented imperative core of languages like Java or C#. Using this simplified language enables us to focus on the central issues of the analysis and allows the analysis to be extended to a large class of source languages.

2.1 Concrete Language and Semantics

Our source language MIL (Mid-Level Intermediate Language) is a structured intermediate representation. The language has function and method invocations, a conditional construct (`if ... else if ... else`) and a looping construct with break statements (`do ... while` and `break`). The state modification operations and expressions (load, store and assign along with the standard collection of logical, arithmetic and comparison operators) are in a standard three-address form [11,18].

MIL supports objects and arrays. We use σ to denote the set of all user-defined object types. Each object type, $\upsilon \in \sigma$, has a set of fields F_υ associated with it. The set of all field offsets that are defined in a program is $F = \bigcup\{F_\upsilon | \upsilon \in \sigma\}$. MIL has the primitive types $\rho = \{\texttt{int}, \texttt{float}, \texttt{char}, \texttt{bool}\}$. Arrays can contain either primitive types, ρ, or objects, σ. The set of all legal array types for a program is $\sigma_A = \{\upsilon\,[\,] | \upsilon \in \rho \vee \upsilon \in \sigma\}$. The set of all types in the program is $\tau = \rho \cup \sigma \cup \sigma_A$. We assume that the types of all variables are explicitly declared. Since this paper is focused on the operation of the abstract heap model and the local data flow analysis, we omit any description of how function and method calls are handled.

2.2 Concrete Heap Definition

The concrete heap is modeled as a multi-graph with labeled edges where objects and arrays are the vertices and the pointers are labeled directed edges in the graph. We use the term *cell* to indicate either an object or an array on the heap and *offset* to indicate the field or array index that a pointer is stored at in a cell. Thus, the set of edge labels (offsets) is, $L = F \cup \mathbb{N}$. Edges are modeled as a relation on the cells and the labels. Given a set of cells C and the set of labels L the edge relation $E \subseteq C \times L \times C$. Variables are modeled as a partial map from variable names to cells. Given a set of variables, V, the variable map is a function, $V_m : V \mapsto C$. The set of all concrete heaps (which we define as being the heap graph plus the program variable map) is, $H_s = \mathscr{P}(C) \times \mathscr{P}(E) \times \{V_m\}$ and the concrete domain $H = \mathscr{P}(H_s)$.

2.3 Heap Properties of Interest

Points-to and Paths. Given cells a, b and offset o, $(a,o) \rightarrow_p b$ denotes a pointer p that has the label o (is stored at offset o) a and points to b. We use $a \rightarrow_p b$ to indicate that $\exists$

offset o s.t. $(a,o) \rightarrow_p b$. Two cells can be connected by a *path* ψ. We use $(a,o) \rightsquigarrow_\psi b$ to indicate the sequence of pointers $\langle p_1 \ldots p_n \rangle$ s.t. p_1 has the label o, starts at cell a, p_n points to b and $\forall p_i, p_{i+1}$ in the path p_i ends at the same cell, c_i, that p_{i+1} begins at ($\exists o'$ s.t. p_{i+1} is stored at o' in c_i). Define $a \rightsquigarrow_\psi b$ to denote that $\exists o$ s.t. $(a,o) \rightsquigarrow_\psi b$. We abuse the notation $\phi \subseteq P$ to denote that all the pointers in the path ϕ are contained in the set of pointers P.

Regions of the Heap. A *region* of memory $\Re$ is a subset of the cells in memory, all the pointers that connect these cells and all the cross region pointers that start or end at a cell in this region. Given $C \subseteq \{c \mid c \text{ is a cell in memory}\}$, let $P = \{\text{pointer } p \mid \exists a,b \in C, a \rightarrow_p b\}$. Let $P_c = \{\text{pointer } p \mid \exists a \in C, x \notin C, a \rightarrow_p x \oplus x \rightarrow_p a\}$ Then a region is the tuple (C,P,P_c).

Connectivity. Connectivity within a region describes how cells in the region are connected. For a region $\Re = (C,P,P_c)$ and cells $a,b \in C$, cells a and b are connected if they are in the same weakly-connected component of the graph (C,P); cells a and b are disjoint if they are in different weakly-connected components of the graph (C,P). Figure 2 shows examples of connected and disjoint concrete heaps. In Figure 2(a) the cells c,d are disjoint in the region Z, while in Figure 2(b) and Figure 2(c) the cells c,d are connected in the region Z.

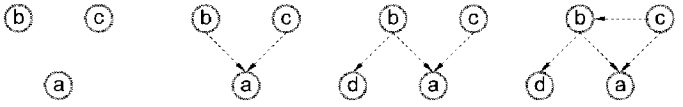
Structure Traversals. An important property for program transformations is the layout of data structures in memory [4,5]. The idea is to track the layout of the heap as it appears to a program traversing a data structure. Ghiya considered the shape of the section of the heap that could be accessed starting from each variable.

Our heap analysis identifies logically related sections of the heap (regions). To improve the accuracy of the shape information we define data structure layouts on these logically related regions instead of the entire section of the heap reachable from a given variable. Given a region $\Re = (C,P,P_c)$, we can define several layout predicates on the graph (C,P) to indicate what kinds of traversal patterns a program can use to navigate through the data structures in the region. A region admits a traversal type if there is a subregion that satisfies the corresponding layout predicate. Note that these traversals are not mutually exclusive and that *Tree* traversal $\Rightarrow$ *List* traversal $\Rightarrow$ *Singleton* traversal. In the following definitions, let a,b be cells and ϕ,ψ be paths.

- Cycle Traversal iff $\exists$ graph $(C',P'), C' \subseteq C, P' \subseteq P$ s.t. $\exists a \in C', \phi \subseteq P'$ s.t. $a \rightsquigarrow_\phi a$.
- MultiPath Traversal iff $\exists$ graph $(C',P'), C' \subseteq C, P' \subseteq P$ s.t. $\exists a,b \in C', \phi,\psi \subseteq P'$ s.t. $(a \neq b) \wedge (\phi \neq \psi) \wedge (a \rightsquigarrow_\phi b) \wedge (a \rightsquigarrow_\psi b) \wedge (C',P')$ does not admit a Cycle Layout.
- Tree Traversal iff $\exists$ graph $(C',P'), C' \subseteq C, P' \subseteq P$ s.t. $(\exists a \in C', a$ has 2 or more successors in $C') \wedge (C',P')$ does not admit a Cycle or Multipath Layout.
- List Traversal iff $\exists$ graph $(C',P'), C' \subseteq C, P' \subseteq P$ s.t. $(\forall a \in C', a$ has one or zero successors in $C') \wedge (\exists b \in C', b$ has one successor in $C') \wedge (C',P')$ does not admit a Cycle or Multipath Layout.
- Singleton Traversal holds for all regions.

Figure 1 shows several concrete heaps; the cells are the circles labeled with letters and the edges represent pointers. Since we are interested in the most general way a program could traverse a region of the concrete heap we must assume that a program

variable could begin its traversal of the region at any of the cells in the region. Thus, the figures omit the program variables. Figure 1(a) shows a concrete heap with three cells (a,b,c). Since there are no edges connecting these cells the only way a program can traverse them is by individually referencing each cell. Figure 1(b) shows a concrete heap that admits a *List* traversal (both $b \rightarrow a$ and $c \rightarrow a$). It also admits a *Singleton* traversal since a program can always treat the cells as if they were disconnected. Figure 1(c) shows a concrete heap that admits a *Tree* traversal (b,a,d) as well as *List* and *Singleton* traversals. Finally, Figure 1(d) adds an edge, $c \rightarrow b$ that changes the region to admit a *MultiPath* traversal (c,b,a).



(a) Singleton (b) List (c) Tree (d) MultiPath

Fig. 1. Concrete Heaps, Admissible Traversals and Layout Types for the Regions

3 Abstract Domain

The abstract domain is based on an abstract heap graph model [2,19,9]. Each node represents a set of concrete cells and each edge represents a set of pointers. The model provides a natural framework for representing connectivity, aliasing, and region identification information. This section introduces a number of instrumentation domains that when added to the nodes and edges in the abstract heap graph allow aliasing and connectivity to be tracked more accurately and enable the modeling of shape.

Numeric Quantities. The only requirement we place on the numeric abstraction is that it differentiates the case where the value is exactly one and the case where the value is in the range $[0,\infty]$. This gives the binary domain $1 < \#$ (unknown), where 1 represents the interval [1, 1] and $\#$ represents the interval [0, ∞]. Given this domain, $a \sqcup a' = 1$ if $a = a' = 1$ and $\#$ otherwise. In the later algorithms we also need an interpretation, $\tilde{+}$, for $+$. This is given by, $a \tilde{+} a' = \#$.

Types. Each node represents a set of cells and each cell is either an object (has type $\upsilon \in \sigma$) or an array ($\upsilon \in \sigma_A$). Since MIL has dynamic method invocation as well as type casting it is important to model the types of cells that a given node might represent. The domain for representing the types of each node is $\mathscr{P}(\sigma \cup \sigma_A)$. As usual the join operation $\sqcup$ is $\cup$ and the $\leq$ relation is $\subseteq$.

Offsets. Each edge in the model represents a set of pointers and each pointer has an offset (label) associated with it. Since there are only a finite number of fields in a given program the model can be completely sensitive with respect to field offsets (by construction two pointers with different offsets are never represented by the same edge). However, there may not be a bound on the size of arrays. So, we treat arrays as having a single offset, ?, that contains a summary of all the elements that may be in the array. Thus, the offsets that are used in the field sensitive parts of the analysis is the set $F \cup \{?\}$.

Abstract Layout. Each node, n, in the graph represents a region, $\mathfrak{R}$ on the heap. To track the traversals that may be admissible in the region $\mathfrak{R}$ that n represents we use a set of layout types $Layouts = \{Singleton, List, Tree, MultiPath, Cycle\}$.

- if n has a *Singleton* Layout, then $\mathfrak{R}$ only admits *Singleton* traversals.
- if n has a *List* Layout, then $\mathfrak{R}$ only admits *Singleton* or *List* traversals.
- if n has a *Tree* Layout, then $\mathfrak{R}$ only admits *Singleton*, *List* or *Tree* traversals.
- if n has a *MultiPath* Layout, then $\mathfrak{R}$ only admits *Singleton*, *List*, *Tree* or *MultiPath* traversals.
- if n has an *Cycle* Layout, then any traversal pattern may be admissible in $\mathfrak{R}$.

This definition leads naturally to the order: $Singleton < List < Tree < MultiPath < Cycle$. Then $l \sqcup l'$ is $max(l, l')$. Examples are shown in Figure 1.

Connectivity. Given the concretization operator γ and two edges e_1, e_2 that start or end at the node n, the predicates that define connectivity in the abstract domain are:

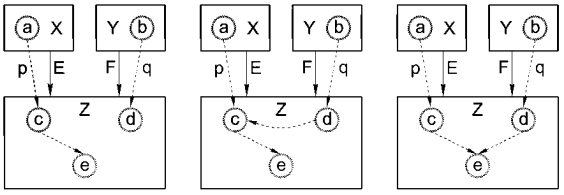
- e_1, e_2 connected with respect to n if: $\exists p_1 \in \gamma(e_1) \wedge \exists p_2 \in \gamma(e_2) \wedge \exists a, b \in \gamma(n)$ s.t. (p_1 starts or ends at a) $\wedge$ (p_2 starts or ends at b) $\wedge$ (a, b connected).
- e_1, e_2 disjoint with respect to n if: $\forall p_1 \in \gamma(e_1) \wedge \forall p_2 \in \gamma(e_2) \wedge \forall a, b \in \gamma(n)$ (p_1 starts or ends at a) $\wedge$ (p_2 starts or ends at b) $\Rightarrow$ a, b are disjoint.

Edges e_1, e_2 are *outConnected* if: $\exists$ n s.t. (e_1, e_2 are out edges from n) $\wedge$ (e_1, e_2 are connected in n).

Edges e_1, e_2 are *inConnected* if: $\exists$ n s.t. (e_1, e_2 are in edges to n) $\wedge$ (e_1, e_2 are connected in n).

Figure 2 shows overlays of the abstract and concrete heaps. The concrete cells and pointers are shown as dotted circles and lines while the abstract nodes and edges are represented with solid boxes and lines. Edge E is an abstraction of pointer p, edge F is an abstraction of pointer q. Node Z abstracts cells c, d, e. Nodes X, Y abstract cells a, b respectively. In Figure 2(a) we can see that the targets of p, q (cells c, d) are disjoint. By the definition of the connectivity abstraction, edges E and F are also disjoint with respect to Z. In Figure 2(b) there is an additional pointer which connects cells d, c. This means that c, d are connected and in the abstraction, E, F are connected with respect to Z and thus E, F are also *inConnected*. Finally, Figure 2(c) shows the case where cells c, d are connected indirectly (but according to the definition they are still connected). Thus E, F are also *inConnected*.

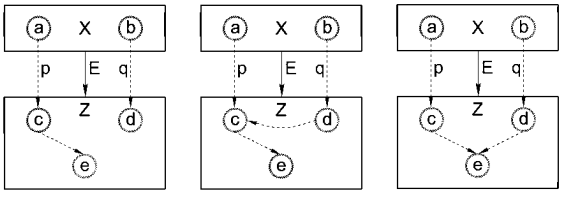
Interference. Each graph edge represents a set of inter-region pointers. When combining nodes, it is important to know if all the pointers that the edge represents point into disjoint subregions or if there may exist a cell that two or more pointers may be able to reach and thus they *interfere*. An edge e represents interfering pointers if there exist pointers $p, q \in \gamma(e)$ such that the cells that p, q point to are connected. We use a two-element lattice, $np < ip$, np for edges with all non-interfering pointers and ip for edges with potentially interfering pointers. This abstraction is a complement to the connectivity relation. The connectivity relation tracks reachability information between the start or end cells of pointers represented by different edges while interference tracks reachability information between the end cells of pointers represented by the same edge.



(a) Disjoint in Z (b) Connected in Z (c) Connected in Z

Fig. 2. Concrete and Abstract Connectivity

In Figure 3, Edge E is an abstraction of pointers p and q, node Z abstracts cells c, d, e, and X abstracts cells a and b. In Figure 3(a) the targets of p, q (cells c, d) are disjoint. Thus, the pointers do not interfere and the edge, E, that abstracts them should be *np*. In Figure 3(b) there is an additional pointer which connects cells d, c. This means that c and d are connected and edge E should be *ip*. In Figure 3(c) the cells c, d are connected indirectly. Thus, the edge E is again *ip*.



(a) Non-interfering (b) Interfering (c) Interfering

Fig. 3. Concrete Connectivity and Abstract Interference

Nodes. The types of the concrete cells that a node represent are stored in a set called *types*. To track the total number of cells that may be in the region represented by this node we use the *size* property. The internal layout of a node is represented by the *layout* component. Finally, we introduce a binary relation $connR \subseteq E \times E$ to track the connectivity of the edges that are incident to this node. If $(e_1, e_2) \in connR$ then e_1, e_2 are connected with respect to this node otherwise e_1, e_2 are disjoint with respect to this node. The abstract domain for the nodes, $N = \mathscr{P}(\sigma \cup \sigma_A) \times Layouts \times \{1, \#\}, \times \mathscr{P}(E \times E)$ and each node in the graph is represented as a record of the form `[types layout size]`. For clarity we omit a representation of the *connR* relation, as the inclusion of this information complicates the figures substantially. In the cases where the connectivity relation is of interest we will mention it in the description of the figure.

Edges. As in the case of the nodes, we combine several component abstractions to create the edge abstraction. The *offset* component indicates the offsets (labels) of the pointers that are abstracted by the edge. The number of pointers that this edge may represent is tracked with the *maxCut* property. The *interfere* property tracks the possibility that the edge represents pointers that interfere. The domain of the edges is,

$E = (F \cup \{?\}) \times \{1, \#\} \times \{np, ip\}$, and each edge is represented as a record {`offset maxCut interfere`}.

Graph. The domain for the abstract heap graphs is the set $G \subseteq \mathscr{P}(N) \times \mathscr{P}(E) \times \{V_n\} \times \{M_e\}$. The function $V_n : V \mapsto N$ is a partial map from variable names to nodes in the heap graph, which represents the targets of the variables. The function $M_e : E \rightarrow N \times N$ defines the structure of the graph by mapping edges e to the pair of nodes (n_s, n_e) such that e begins at n_s and ends at n_e. We use the notation $M_e(e) = (*, n)$ or $M_e(e) = (n, *)$ in the case were we do not care about the identity of the start/end node of the edge.

We restrict the abstract domain by defining a normal form for heap graphs. This normal form simplifies the structure of the abstract domain and it has several properties that improve the accuracy of the analysis.

First, we define what it means for two nodes to to be *recursive* (for this work we assume single level recursion but the definitions can be generalized [3]). This definition is used to make the abstract heap domain finite for a given program. If we limit the maximum size of the graph structure then, since the domains for the nodes and the edges are finite, the number of graphs is finite. This is done by forcing recursive structures to have bounded representations. Define two nodes $n, n' \in N$ to be *recursive* if:

- $\exists e \in E$ s.t. $M_e(e) = (n, n')$.
- $n.types \cap n'.types \neq \emptyset$.
- $\nexists$ variable v s.t. $V_n(v) = n \vee V_n(v) = n'$.

Another useful concept is that of ambiguous edges. We would like to be able to assume that given an offset and a node there is a unique outgoing edge that is incident to this node with that offset. Define a node n as having an ambiguous offset if: $\exists e, e' \in E$ s.t. $e \neq e' \wedge M_e(e) = (n, *) \wedge M_e(e') = (n, *) \wedge e.offset = e'.offset$.

A graph $g = (N, E, V_n, M_e)$ is in normal form if:

- It has no unreachable nodes: $\forall n \in N, \exists$ variable v s.t. $V_n(v) = n \vee (V_n(v) = n' \wedge \exists$ path ϕ s.t. $n' \rightsquigarrow_\phi n)$.
- It has no recursive nodes: $\nexists n_1, n_2 \in N$ s.t. n_1, n_2 are recursive.
- It has no ambiguous edges: $\nexists n \in N$ s.t. n has an ambiguous offset.
- No refinement rules can be applied, See Section 5.

4 Example: Building a List

We use two examples to demonstrate our analysis, Figure 4. The first is a loop that constructs a linked list. The second example copies a linked list (and is the subject of Section 7). We assume that the datatypes `ListNode` and `DataNode` have been defined. In an actual program the data elements in the list might be other data structures (lists, trees, arrays, etc) or composite user defined objects. However, the exact nature of the data components of the list does not fundamentally alter the behavior of the algorithm on our examples. Thus, for simplicity we use `DataNode` as a dummy type to represent whatever data is of interest. `ListNode` has a `next` field which points to the next node in the list and a `data` field which points to a `DataNode`.

Build a List

```
ListNode p, q
p = null
for(int i = 0; i < M; ++i)
    q = new ListNode()
    q.data = new DataNode
    q.next = p
    p = q
```

Copy a List (in reverse, for simplicity)

```
ListNode q, x, t
x = p
q = null
while(x != null)
    t = q
    q = new ListNode()
    q.next = t
    q.data = x.data
    x = x.next
```

Fig. 4. List Example Code

Figure 5(a) shows the state of the abstract heap after allocating the `ListNode` (abbreviated `LN`). The variable `q` points to a node of type `ListNode` and since we just allocated the object that this node represents we know that the node represents exactly one cell and has a *Singleton* layout (abbreviated `S`). Figure 5(b) shows the state of the heap after allocating and assigning the data object, a cell of type `DataNode` (`DN`). The data node is also a node of size one with a *Singleton* layout. The connecting edge is stored at the `data` offset and since it was just created it must represent a single pointer and be *np*. Figure 5(c) shows the heap at the end of the first loop iteration: `p` points to the newly created list entry and `q` is nullified since it is dead.

Figure 5(d) shows the abstract heap at the end of the second loop iteration. New nodes represent the `ListNode` and `DataNode` cells allocated in this iteration. The newly allocated list entry has been put at the head of the list and the old list (shown dotted) is linked in with an edge stored at the `next` offset. If we were to continue, the heap abstraction would grow in an unbounded manner. To prevent this, we normalize the abstract heap. This is described in detail in Section 6 but for this example the important point is that we merge the two `ListNode` nodes into a single summary node that represents the combined information from these two nodes and the edge between them. By looking at the edge connecting the two nodes and the internal layouts we can determine that the internal layout of the summary node is *List* (abbreviated `L`) since we have two *Singleton* regions connected by an edge of size one. Since each region is of size one the summary region must be of size larger than one, represented by # in our abstract domain. Finally, we update the internal connectivity information for the summary node. In particular, the two edges are *outConnected*. The state of the heap after this merge is shown in Figure 5(e).

After combining the list nodes we have ambiguous targets (two out edges from the same node with the same label, `data`) This ambiguity is removed by merging the potential targets into a single summary node and by combining the edges that refer to these targets into a single summary edge. Merging these nodes is similar to the merge of the list nodes except that the two incoming edges are *disjoint*. After merging the nodes we merge the two edges. Since the summary edge represents two pointers its *maxCut* is #. To determine the value of the *interfere* property we check if either edge is *ip* or if the targets of the edges are *inConnected*. Because the edges pointed to disjoint nodes they

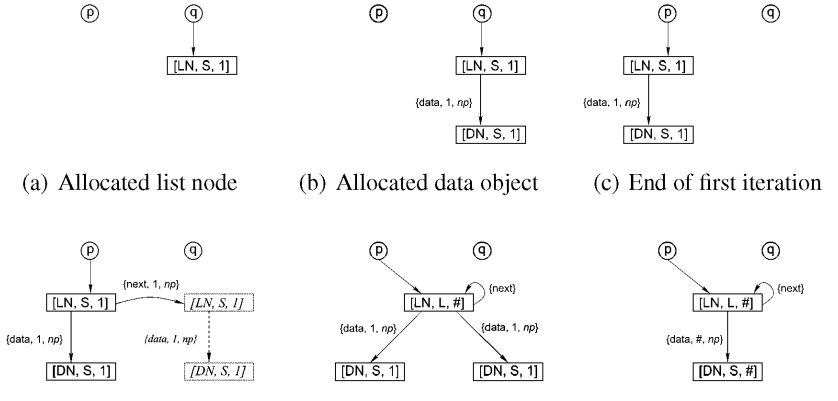
are not *inConnected* and therefore cannot *interfere*. Thus, the interference property of the summary edge is *np*. The result is shown in Figure 5(f), which is also the fixed point for the analysis of the loop.

5 Refinement

During the data flow analysis portions of the abstract heap graph are summarized into single nodes to improve efficiency and to eliminate unbounded recursive data structures. This summarization can cause a substantial loss of accuracy if it is too aggressive. We define a method that (for the most common cases encountered) allows us to undo the summarization by transforming a summary node into a number of nodes (and edges) so that relationships between variables and regions of the heap can be more accurately modeled.

There are three layout types that we refine. The first is a node that represents several disjoint regions of the concrete heap. In this case we expand each sub-region into a separate node in the abstract graph. The second is a list node with a single incoming edge. In this case we make explicit the unique memory location that the variable must refer to in the list structure. The third is a tree with a single incoming edge. This case is analogous to the list so we do not discuss it separately.

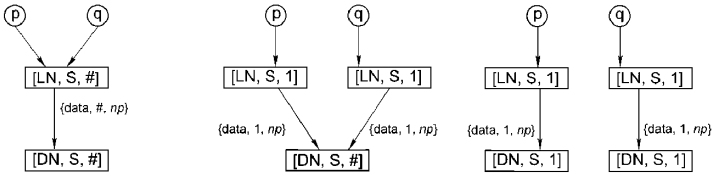
Disjoint Region Separation. It is possible for a single summary node to represent several entirely disjoint regions. If this is the case then there is a partition of incoming edges (from variables or pointers) based on the *inConnected* relationship. Using this partition we transform the node into a number of new nodes, each new node representing a single element from partition of edges in the original node. An important special case is when a node has a *Singleton layout* and there is a single incoming edge of *maxCut* 1. If a node has these properties we can safely assume that the node represents a single cell, which enables strong updates in later analysis steps.



(a) Allocated list node (b) Allocated data object (c) End of first iteration

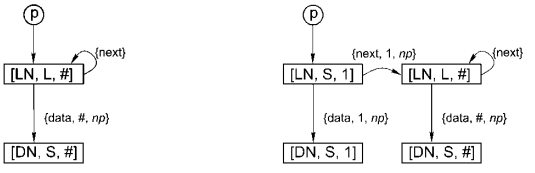
(d) End of second iteration (e) First normalization step (f) Finished

Fig. 5. Building a linked list



(a) Summarized Singleton (b) Partition Pointers (c) Partition pointer edges

Fig. 6. Refinement of a region with disjoint sub-regions



(a) Summarized List (b) Refined List

Fig. 7. Refinement of a node with a list layout

Consider the case in Figure 6(a) where the variables p and q point to the same node, and assume that the edges from p and q are not *inConnected*. Partitioning results in Figure 6(b) where the summary node has been partitioned based on the *inConnected* relation from the variables. Since the edge that was split contained all non-interfering pointers the two edges incident to the node representing the *DataNode* cells cannot be *inConnected*. This now allows us to apply refinement again—the results are shown in Figure 6(c).

Refinement on Lists. Refinement on lists is more complex than refinement of disjoint regions. Since disjoint region refinement is applied before list or tree refinement we know that all the incoming edges to the given list node may be connected. Further, if there are multiple incoming edges we cannot determine an ordering for them in the list, so we only consider lists with a single incoming edge with a *maxCut* of 1.

Figure 7(a) shows a list with one incoming variable. Figure 7(b) shows the most general way in which a list can be referred to by a single program variable; there is a single cell that the variable points to and a section of the list after this cell. We can safely ignore the section of the list before the cell that the variable refers to since it is unreachable and therefore cannot affect the program in any way. Since the data edge contains all non-interfering pointers we apply the disjoint region separation rule to the data components of the list.

6 Dataflow Operators

This section describes the principal algorithms used in the analysis. We first address merging nodes and edges. Then we define the normalization routines for nodes and

graphs. Finally, we use these operations to build the heap graph upper bound and comparison operations. Detailed descriptions of some algorithms and proofs for the required safety properties are omitted; see [13] for more details.

Edge Join (Algorithm 1). The edge join method is only well defined when two edges start at the same offset in the same node and end at the same node. The method checks the end connectivity information to determine how the component abstraction should be combined. If the edges are *inConnected* then the pointers that these edges represent may interfere and we set the summary edge as *ip*, otherwise we take the join of the *interfere* types of the edges. For the rest of the components that are used to represent an edge, we can simply combine them component-wise with respect to the possibility that these edges originated in separate graphs. That is, when we join two heap graphs that are from separate flow paths in the program we know that there can be no interaction between edges from different control contexts. The edge join algorithm uses the function *updateInternalConnInfoEdgeJoin*(n_s, n_e, e_a, e_b) to update the internal connectivity info in n_s and n_e to represent the fact that e_a now represents pointers from e_a and e_b.

Algorithm 1. Join Edges ($\sqcup_e$)

input : g the heap graph, e_a, e_b edges, n_s,n_e the nodes, e_a,e_b start and end at
if (e_a, e_b *from the same context*) **then** e_a.maxCut $\leftarrow e_a$.maxCut $\widetilde{+}$ e_b.maxCut;
else e_a.maxCut $\leftarrow e_a$.maxCut $\sqcup$ e_b.maxCut;
if e_a,e_b *are inConnected* **then** e_a.interfere $\leftarrow ip$;
else e_a.interfere $\leftarrow e_a$.interfere $\sqcup_{\text{interfere}}$ e_b.interfere;
updateInternalConnInfoEdgeJoin(n_s, n_e, e_a, e_b);
deleteEdge(g, e_b);

Node Join and Combine. When summarizing two nodes, n_a and n_b, there are three possibilities. The first is neither node can reach the other. In this case we *join* them. If there are only edges in one direction between nodes, from n_a to n_b or n_b to n_a, then we *combine* them. If there are edges from n_a to n_b and from n_b to n_a, then we replace n_a, n_b with a single node n_c that is a safe approximation of n_a, n_b.

Figures 5(e) and 5(f) show that the node join is a purely component-wise operation. The combine operation on a pair of nodes that have a connecting edge is more complicated, as it can be seen in Figures 5(d) and 5(e), where the two nodes with type `ListNode` are combined into a single summary node. In particular we need to account for the fact that the edge(s) connecting nodes n_a and n_b will affect the layout and the internal connectivity in the new summary node.

The algorithm *combineLayout*(l_a, l_b, ebt), is based on a case analysis of the internal layout that results from the possible combinations of layouts for n_a, n_b along with the total number of pointers represented by *ebt* and the potential that any pointers in the edges represented by *ebt* interfere. We enumerate the possible combinations of the *ebt* edges and the layout types. 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.

Algorithm 2. Combine Nodes ($\widetilde{+}_{\text{node}}$)

input : graph g, n_a, n_b nodes, *ebt* set of edges from n_a to n_b
n_a.type $\leftarrow$ n_a.type $\cup$ n_b.type;
n_a.size $\leftarrow$ n_a.size $\widetilde{+}$ n_b.size;
n_a.layout $\leftarrow$ combineLayout(n_a.layout, n_b.layout, *ebt*);
n_a.connR $\leftarrow$ combineConnR(n_a.connR, n_b.connR, *ebt*);
remap all edges incident to n_b to be incident to n_a;
deleteNode(g, n_b);

Algorithm 3. combineLayout

input : l_a, l_b layout types, *ebt* set of edges from n_a to n_b
output: the layout of the combined node
$mayInterfere \leftarrow \bigvee\{e \in ebt | e.\text{interfere} = ip\}$;
$totalCut \leftarrow \sum\{e \in \text{ebt} \mid e.\text{maxCut}\}$;
$notSingletons \leftarrow s_a \neq \text{Singleton} \wedge s_b \neq \text{Singleton}$;
$isDAGgraph \leftarrow totalCut > 1 \wedge notSingletons$;
$l_r \leftarrow l_a \sqcup_{\text{layout}} l_b$;
case $(mayInterfere \vee isDAGgraph)$ **return** $l_r \sqcup_{layout} MultiPath$;
case $(l_a = List)$ **return** $l_r \sqcup_{layout} Tree$;
case $(l_a = l_b = Singleton)$ **return** $l_r \sqcup_{layout} List$;
otherwise return l_r;

The *combineConnR* function updates the internal connectivity information in n_a to reflect that it now represents the combined regions for n_a and n_b. This involves computing the binary connectivity relation for all the edges that are incident to the new summary node based on the connectivity information in the argument nodes n_a, n_b, and the edges that connect the argument nodes, *ebt*.

Normalization/Join Operators. To normalize a node we check if there are two edges that start at this node and have the same offset. If they exist and they end at different nodes, we merge the target nodes and then join the edges. If they already end at the same node, we just join the edges.

To normalize a heap graph we normalize all the nodes, then apply the refinement rules to all the nodes that they can be applied to and finally we compress all the recursive nodes in the graph. This process is repeated until the heap graph is no longer changing.

To compute the upper approximation for two heaps, we first normalize both heaps and mark which graph each edge and node belonged to originally. Then we take variables with the same name and union their targets. Once this is done the resulting graph is normalized.

Heap Graph Equivalence. Defining equivalence on the heap graphs is simple if we require that they are in normal form. This implies that each abstract storage location has a unique edge and we can compare the graphs for structural equality and equality of the data in the nodes and edges.

Algorithm 4. Normalize Node

input : node n, graph g, $n \in g$
output: None
while $\exists$ *offset o with more than 1 edge* **do**
    $e_1, e_2 \leftarrow$ two edges with offset o;
    $n_1 \leftarrow$ endpoint of e_1;
    $n_2 \leftarrow$ endpoint of e_2;
    if $n_1 \neq n_2$ **then**
        if $\exists$ *edges from* n_1 *to* n_2 *and* n_2 *to* n_1 **then**
            replace n_1, n_2 with the $\top$ from the lattice of nodes;
        else if $\exists$ *edges from* n_1 *to* n_2 **then**
            combine(g, n_1, n_2);
        else if $\exists$ *edges from* n_2 *to* n_1 **then**
            combine(g, n_2, n_1);
        else
            join(g, n_1, n_2);

    $\sqcup_e(e_1, e_2, g)$;

Algorithm 5. Normalize Graph

input : graph g
output: None
Remove all unreachable nodes from g;
while g *is changing* **do**
    while $\exists$ *node n s.t. n can be normalized* **do**
        normalize(n, g);

    while $\exists$ *node n s.t. n can be refined* **do**
        apply the applicable refinement rule to n;

    while $\exists$ *nodes* n, n' *that are recursive* **do**
        combineNodes(g, n, n');

Algorithm 6. Heap Graph Upper Bound, $\tilde{\sqcup}$

input : graph g_a, g_b
output: None
$g_{an} \leftarrow$ normalize(g_a);
set all nodes/edges in g_{an} as context a;
$g_{bn} \leftarrow$ normalize(g_b);
set all nodes/edges in g_{bn} as context b;
$g_{res} \leftarrow g_{an} \cup g_{bn}$;
normalizeGraph(g_{res});

7 Example: Copying a List

During the copy operation (Figure 4) there are several attributes that we want to preserve: the source list should be unaffected, the copy should be a list, and, if the source list contained all independent data elements so should the copy. For simplicity assume that we know that the source list is already pointed to by `p`. Figure 8(a) shows the list at the start of the copy. Figure 8(b) shows the results at the end of the first loop iteration. The head of the list has been copied; `t` is nullified and `x` has been indexed down the list. Note that in indexing down the list we refined the list on the `next` edge so that the node that `x` refers to is made explicit (the node is a singleton of size 1). We show the newly materialized list and data node using dotted lines.

Figure 8(c) shows the heap during the second iteration of the loop after creating the new list node and assigning it to point to the next data node in the source list. At the end of the loop 8(d) we have again indexed the variable `x`. We now have recursive nodes (for simplicity assume that we know that keeping `p`, `q` refined does not matter—if we keep them refined the result is the same, it just takes an extra loop iteration and results in a larger graph). Thus, we compress them during normalization. The resulting graph shown in Figure 8(e). This is the fixed point of the loop and if we interpret the exit condition we see that the result of the copy loop is the heap graph in Figure 8(f).

8 Performance

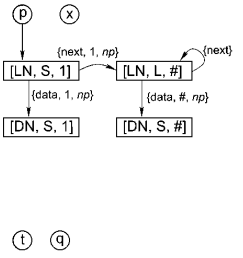
Theoretical Performance. In order to analyze a program, the model presented in this paper can plugged into any dataflow analysis framework. The total cost of analyzing the program is affected by the cost of the model operations and the runtime of the dataflow framework that is chosen. In this paper we do not assume a specific framework so our runtime analysis only looks at the cost of the model operations.

We assume that the number of nodes in the abstract heap graph is n, that each node has at most k edges and there are t user defined types. The most expensive part of running the heap model is the graph normalization step, so we only present the analysis for this and the node combine operation, which is the dominant cost of the normalization algorithm.

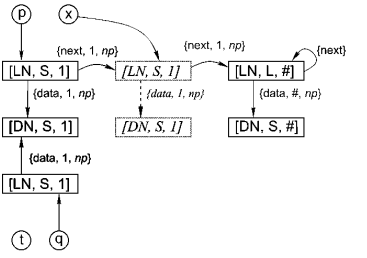
The execution of *Combine Nodes* (Algorithm 2), requires combining the type sets, $O(t)$, remapping the incident edges, $O(k)$, calling *combineLayout* (which computes the shape of the combined nodes), $O(k)$ and calling the *computeConnR* method (which computes the transitive closure of the two connectivity relations), $O(k^3)$. Thus, The total time is $O(t+k+k+k^3)$. If we assume that t is a small constant, the time to normalize a node is $O(k^3)$.

The graph normalization step requires:

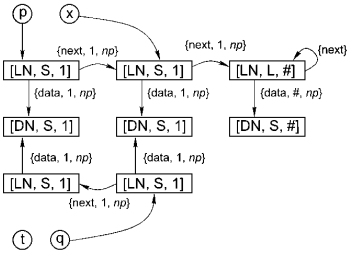
- Removing all the unreachable sections of the heap graph, $O(nk)$.
- Normalizing each node, visit each edge of each node and potentially combine edges, $O((nk)k^3)$.
- Refining all possible nodes, visit each node and potentially refine it, $O(nk)$.
- Removing all recursive nodes, visit each node and potentially combine two nodes, $O((nk)k^3)$.



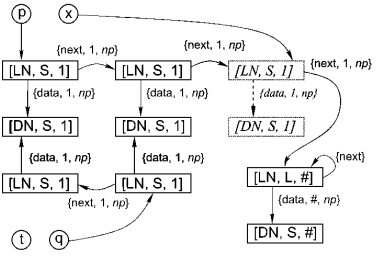
(a) Start of the method



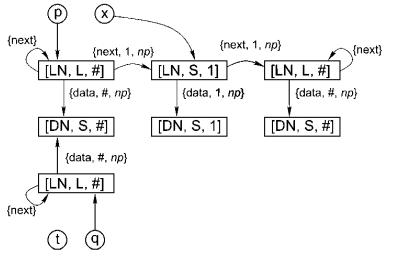
(b) First loop iteration



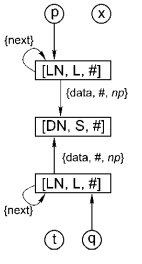
(c) Create copy node in the second iteration



(d) End of second loop iteration



(e) Normalization



(f) Finished

Fig. 8. Copying a linked list (in reverse order)

These operations need to be done until none can be applied. Since the refine operation can only be applied to a node twice and the combine operation replaces two nodes by a single node (which cannot be refined), the algorithm cannot continue for more than $O(n)$ iterations. Thus the total time for the normalization routine is $O(n) * O((nk)k^3) = O(n^2k^4)$.

Benchmarks. In this section we compare the runtime cost of the UMA (Unified Memory Analysis) method with TVLA (tvla-2-alpha) and a simple flow-sensitive equality-based

points-to method (which is not capable of shape analysis but provides a performance baseline). Then, we examine the accuracy and utility of the information that the UMA analysis method provides. All measurements were made on a Pentium M 1.5 GHz laptop with 1 GB of RAM.

We use two sets of benchmarks. The first is a number of simple list manipulation methods that are useful for validating that the information computed by this analysis is accurate. These benchmarks include list insertion, deletion, find and copy operations. The goal is to ensure that the listness and data independence properties are preserved through all of these operations. The first entry in Figure 9 shows the runtime for TVLA, the points-to and the UMA analysis. In all of the simple list tests, our analysis is able to determine that the result of each list operation is a region with the *List* shape.

List Analysis Times

Benchmark	Copy	Find	Insert	Delete	Reverse
TVLA	NA	0.91s	1.52s	8.00s	3.01s
Points-to	0.05s	0.03s	0.04s	0.06s	0.03s
UMA	0.25s	0.10s	0.15s	0.19s	0.13s

Jolden Analysis Times and Shape Results

Benchmark	bisort	em3d	health	mst	power	treeadd	tsp
Points-to	2.10s	1.48s	12.20s	0.54s	0.42s	0.16s	0.70s
UMA	12.30s	6.90s	40.90s	5.70s	4.20s	1.80s	5.08s
Accurate	Partial	Yes	Partial	Yes	Yes	Yes	Yes

Fig. 9. Benchmark Results

The second set of benchmarks is from the Jolden suite [1] (we have not finished implementing the virtual method dispatch analysis, so bh, perimeter and voronoi are omitted from the table). This set of benchmarks is designed to test how well the analysis method scales to non-trivial code sizes and as a first test for the ability of the heap analysis method to provide useful shape information for parallelization transforms. Current work on interprocedural versions of TVLA indicate that even simple programs take upwards of 30s to analyze [14] and no results for programs as complex as the Jolden suite have been published so we omit the TVLA analysis from the table. The second entry in Figure 9 shows the time to run the analysis on each of the benchmarks and indicates if the analysis was able to correctly determine the shape information required to perform basic thread level *foreach* and *recursive tree* parallelization. In the table we have two categories for the accuracy of the shape analysis. *Yes* is used when the shape analysis was able to provide the correct shape information for all of the relevant heap structures in the program. *Partial* indicates that the analysis was able to determine the correct shape for some of the heap data structures but that some important properties were missed.

There were no cases where the analysis failed to produce a non-trivial amount of useful information on data structure shapes. In the cases where the UMA algorithm is unable to provide completely accurate information for parallelization the causes can be traced back to the simple modeling of arrays (health) or the crude technique we are currently using for interprocedural analysis in recursive functions (bisort and health).

9 Conclusion and Future Work

This paper presented a graph-based heap model that can be used with a standard data flow framework to analyze the evolution of the heap during program execution. The

model is shown to be capable of representing heap properties (aliasing, shape and logical data structure identification) that are needed to extract thread level parallelism from single threaded programs. The paper then outlined the model operations required to perform the program analysis. A key component of the operations was the use of a refinement operator that enables the accurate simulation of important program operations (copying, reversing, destructive updates, etc.). Unlike Ghiya's work where extremely conservative approximations must be made in the presence of destructive updates, the proposed model is able to retain enough information to provide meaningful shape information even when destructive updates are being performed. Theoretical analysis shows that all the program operations on the model are $O(k^4)$ and the upper bound/equality operations are $O(n^2k^4)$ where n is the number of nodes in the heap graph and k is the number of edges incident to a node. This polynomial runtime is due to our conservative refinement operator (which only refines unambiguous cases) which is in contrast to the TVLA refinement operator (which resolves ambiguity by enumerating all possible cases).

The method has been implemented and run on several benchmarks. The first set of benchmarks is designed to test the ability of the analysis method to model fundamental list operations. The method analyzed this set quickly while discovering all the relevant list properties. Next, we analyzed several codes from the Jolden benchmark suite. Analysis times on these benchmarks scaled acceptably given that a simplistic and fully context-sensitive interprocedural analysis method was used. The method correctly identified the shapes (*Singleton*, *List*, *Tree*, *MultiPath*, *Cycle*) for almost all of the data structures in the programs.

These results are a critical step toward the goal of transforming modern single threaded programs that make extensive use of pointer rich, heap based structures into multithreaded parallel programs. Our future work is focused on improving the accuracy, performance and scope of this analysis technique. We identified recursive procedures that rely on destructive updates as a major issue in accurately modeling shape and handling these cases is the next step in our research. The method is local in the sense that all abstract program operations only refer to and modify small portions of the heap, we plan to utilize this to enable memoization and localization of procedure calls, both of which are crucial to improving scalability. Since modern programming languages make extensive use of built in collection libraries (hashtables, trees with parent pointers, iterators, etc.) we are working on how to model these important data structures and generic programming concepts.