# Heap Analysis in the Presence of Collection Libraries

Mark Marron[1] Darko Stefanovic[1] Manuel Hermenegildo[1,2] Deepak Kapur[1]

[1] University of New Mexico
[2] Technical University of Madrid
{marron,darko,kapur}@cs.unm.edu, herme@fi.upm.es

## Abstract

Memory analysis techniques have become sophisticated enough to model, with a high degree of accuracy, the manipulation of simple memory structures (finite structures, single/double linked lists and trees). However, modern programming languages provide extensive library support including a wide range of generic collection objects that make use of complex internal data structures. While these data structures ensure that the collections are efficient, often these representations cannot be effectively modeled by existing methods (either due to excessive analysis runtime or due to the inability to represent the required information).

This paper presents a method to represent collections using an abstraction of their semantics. The construction of the abstract semantics for the collection objects is done in a manner that allows individual elements in the collections to be identified. Our construction also supports iterators over the collections and is able to model the position of the iterators with respect to the elements in the collection. By ordering the contents of the collection based on the iterator position, the model can represent a notion of progress when iteratively manipulating the contents of a collection. These features allow strong updates to the individual elements in the collection as well as strong updates over the collections themselves.

***Categories and Subject Descriptors*** F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages (program analysis)

***General Terms*** Languages, Performance, Verification

***Keywords*** shape analysis, static analysis, collection library

## 1. Introduction

Library-based collections are a fundamental component of modern programming languages and are used extensively in almost any non-trivial program. Substantial work has gone into developing heap analysis tools that can accurately and efficiently analyze simple data structures, mainly lists, trees, and simple cyclic structures [13, 15, 16, 7]. Unfortunately, all of these techniques have aspects that make their use in analyzing large programs that use standard libraries impractical. This is either due to the inability to model the complex data structures (red-black trees, doubly-linked lists with tail pointers, etc.) used in the library code [13, 7] or due to the computational complexity of performing the analysis [15, 16].

An alternative to directly analyzing the code that implements the collection objects is to use the semantics of the collection objects to simulate the effect of each collection operation as an atomic program operation. This approach is frequently used to analyze libraries or other modules [14, 4, 9, 11, 1].

In addition to the model complexity and performance issues that arise when directly analyzing the collection library implementations the semantics based approach allows the modeling of properties specific to each collection type (e.g. sets never contain duplicate elements), the selective modeling of program properties (e.g. modeling sizes of collections without having to track the size of every heap region) and the ability to provide high level semantics for complex operations so that a lightweight analysis can be used effectively (e.g. the semantics of sorting a vector).

Our primary contribution in this paper is a method for representing the semantics of collection libraries and iterators over the collections in a shape analysis framework. The representation that we present for the collection semantics enables the shape analysis to identify individual elements in the collection, allowing them to be strongly updated. The iterator semantics provide a representation for the notion of progress in the processing of the elements in the collections, which allows the shape analysis to accurately model the processing of the collections.

The only property we require from the heap model is the ability to *refine* summarized regions of the heap. The refinement of a summarized region into a set of regions where the relations between them are explicit is critical to identifying individual memory objects and allowing them to be *strongly* updated. The approach presented in this paper can be adapted to the heap models presented in the TVLA (Three-Valued Logic Analysis) based work [16, 7], the graph model in [12], or the UMA (Unified Memory Analysis) model [13]. In order to simplify the construction and to make the examples concrete we focus on the UMA model.

## 2. Example Programs

To gain some insight into how our extensions work and interact with the UMA heap analysis we use the examples in Figure 1. The examples use objects of types, `t1` and `t2`. The `t1` type has a single field `val` that points to objects of type `t2`. The `t2` type is a simple object with no pointer fields. The first code segment is a loop that fills a `set` with objects of type `t1` (all of which have a pointer to the same object in the `val` field). The second example takes the resulting `set` and updates each element to point to the `t2` object that the variable `r` points to.

We are using the `t1` and `t2` types to keep the examples simple. However, the methods presented in this paper can handle similar programs, with the same level of accuracy, where `t1` and/or `t2` are replaced by simple finite structures, lists, trees, or other library collections. The analysis algorithm is also able to analyze our examples when `t1` and/or `t2` are replaced with DAG shaped or cyclic structures, although potentially with reduced accuracy.

Initialize a Set

```
set p = new set()
t1 q
t2 s = new t2()
for(int i = 0; i < M; ++i)
    q = new t1()
    q.val = s
    p.insert(q)
```

Update all the elements in the set

```
t2 r = new t2()
iterator i = p.begin()
while(i.isValid())
    (i.get()).val = r
    i.advance()
```

**Figure 1.** Example Code

In both examples the analysis should determine that every element in the `set` is unique (although the elements may reference the same object in the `val` field). In the second example the analysis should capture the fact that on each iteration of the loop the element that the iterator refers to has its `val` offset updated and after the loop all the elements in the `set` have been updated. Thus, there are no longer any objects in the set with pointers in the `val` field that refer to the same object as the variable `s`.

## 3. Heap Model

The UMA [13] abstract domain is based on an abstract heap graph model [3, 17, 10]. Each node represents a region of the heap and each edge represents a set of pointers. The UMA model uses a number of instrumentation domains that, when added to the nodes and edges in the abstract heap graph allows connectivity to be tracked more accurately, enables the modeling of shape and enables the refinement of nodes in the heap model.

***Regions of the Heap.*** A *region* of memory $\Re$ is a subset of the objects/arrays in memory, all the pointers that connect them and all the cross region pointers that start or end in this region. Given $C_{\Re} \subseteq \{$objects/arrays in memory$\}$, let $P_{\Re} = \{$pointer $p \mid \exists a,b \in C_{\Re}, p$ is stored in $a$ and points to $b\}$. Let $P_c = \{$pointer $p \mid \exists a \in C_{\Re}, x \notin C_{\Re}, p$ is stored in $a$ and points to $x \oplus p$ is stored in $x$ and points to $a\}$. Then a region is the tuple $(C_{\Re}, P_{\Re}, P_c)$.

***Connectivity.*** Connectivity within a region describes how objects/arrays in the region are connected. For a region $\Re = (C_{\Re}, P_{\Re}, P_c)$ and objects $a,b \in C_{\Re}$, objects $a$ and $b$ are connected if they are in the same weakly-connected component of the graph $(C_{\Re}, P_{\Re})$; objects $a$ and $b$ are disjoint if they are in different weakly-connected components of the graph.

### 3.1 Basic Properties

The UMA model uses a number of simple properties to augment the nodes and edges. The most basic is the numeric abstraction, which has two values, exactly one (*1*) and the range $[1,\infty]$ (*#*). The other is a set of type names that represents all the possible types of the objects/arrays that the node represents.

Next we have the offsets. Each edge in the model represents a set of pointers and each pointer has an offset (label) associated with it. The UMA model allows the offsets to be any of the field identifiers declared in the program or a special offset, `?`. This special offset is used to represent pointers which are stored in an array.

The last of the basic properties is the *Abstract Layout*. This concept is used to represent the possible memory layouts that a region of the heap may have. 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.

### 3.2 Pointer Connectivity Properties

The UMA model relies on tracking the potential that two pointers can reach the same location in a region of memory to drive the tracking of the *Abstract Layouts* and to enable the refinement of the common case heap structures that it encounters.

***Connected Edges.*** The first property is when two pointers are represented by different edges in the heap model. Given the concretization operator $\gamma$ and two edges $e_1, e_2$ that are incoming edges to the node $n$ (end at $n$), the predicate that defines *inConnected* in the abstract domain is: $e_1, e_2$ are *inConnected* 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. $(p_1$ ends at $a) \wedge (p_2$ ends at $b) \wedge$ $(a$, $b$ connected in $\gamma(n))$.

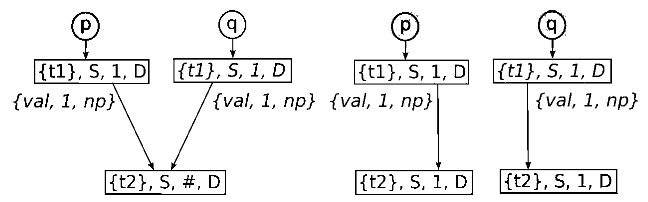
***Interfering Pointer Edges.*** The second property is for the case where the pointers of interest are represented by the same edge in the abstract model. To model this, the *interfere* property is introduced. An edge $e$ represents interfering pointers if there exist pointers $p,q \in \gamma(e)$ such that the objects that $p,q$ point to are connected. A two-element lattice, $np < ip$, $np$ for edges with all non-interfering pointers and $ip$ for edges with potentially interfering pointers is used to represent the interference property.

### 3.3 The Heap Graph

Each node in the graph contains a record that tracks the types of the objects/arrays that a node represents (*types*), the total number of objects/arrays that may be in the region represented by this node (*size*), and the abstract layout of a node (*layout*). Each node also needs to track the connectivity relation between each pair of incoming edges. In [13] a binary relation $connR \subseteq E \times E$ is used to track the *inConnected* relation. However, for this work it is sufficient to use a simple binary domain (*connB*), where *connB* is $D$ if all the in edges must be disjoint and $C$ if any of the in edges may be connected. In this work we assume that the variables may be connected to any edge or variable in the node they refer to and thus are ignored in the *connB* binary predicate. Thus, each node is represented as a record of the form `[types layout size connB]`.

Each edge contains a record that tracks domain information about the edge. 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. Thus, in the figures each edge is represented as a record `{offset maxCut interfere}`.

The abstract heap domain is restricted via a normal form. The normal form ensures that the heap graph remains finite, that all the outgoing edges from a node have unique labels, and that there are no unreachable nodes. The graph is kept finite by ensuring that any recursive structure (structures that involve recursive object types) are represented by a finite number of nodes (see [13] for a more complete description of how this is done). The program analysis is then performed using sets of the heap graphs to represent the possible program states at each point in the program.



(a) Disjoint Edges (b) Refined

**Figure 2.** Refinement of a region with disjoint sub-regions

## 4. Refinement

During the dataflow 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. To minimize this accuracy loss the UMA algorithm uses a technique that (for several common cases) undoes the summarization by transforming a summary node into a number of nodes (and edges) such that the relationships between variables and regions are more explicit.

There are currently three cases that the UMA algorithm refines. For this paper the only case that is relevant is when all the incoming edges for a given node are disjoint. In this case we know that each of these edges represents a set of pointers which point into a disjoint sub-region of the region represented by the node. Thus, the algorithm can expand each sub-region into a separate node in the abstract graph (one for each disjoint edge).

Consider the case in Figure 2(a) where the the two edges with the `val` offsets refer to the same node (which is a node representing cells of type `t2`, with a *Singleton* layout, that may represent any number of objects, and all the incoming edges are *disjoint*). Since the incoming edges refer to disjoint sections of the node we can partition this summary node into two distinct nodes. The partitioning results in Figure 2(b). Note that the newly created nodes each only have a single incoming edge representing at most one pointer and they have *Singleton* layouts. Thus, the node can represent at most one object and the size is set to 1.

## 5. Domain Extensions For Collections

The fundamental idea for modeling the collections and iterators is to classify the pointers that are stored in a collection into four categories based on their relation to any iterators that are acting on the collection. Based on this classification we create a special *offset* for each category, just as was done for arrays in Section 3.1.

- Pointers that have an unknown relation to the active iterator or when there is no active iterator for this collection. Edges representing pointers in this category are given the label *?*.
- The single pointer that the iterator is currently at in the collection. The edge representing this pointer is given the label *@*.
- Pointers that come before (in whatever iterator order is specified by the collection) the location that the iterator is at. Edges representing pointers from this class are given the label *B@*.
- Pointers that come after (in whatever iterator order is specified by the collection) the location that the iterator is at. Edges representing pointers from this class are given the label *A@*.

This scheme for classifying the pointers in a collection is a specific case of the *partitioning functions* that are used in [6] to partition arrays of scalars. The definition we use is only precise when there is a single iterator that is active in a collection. In the case of multiple iterators simultaneously indexing through a collection our partition must conservatively assume that any relation could hold between the positions of the iterators. The use of more flexible *partitioning functions* would allow our analysis to partition the pointers in a collection even when multiple iterators are being used to index through the collection. However, the use of more general *partition functions* substantially complicates the analysis and we expect that most of the time only a single iterator will be active in a collection. Based on this assumption we opted for the fixed partition.

***Modifications to the Model.*** To model the collections and iterators we need to extend the abstract domain from Section 3 with some additional properties. The most basic extension is to add the types `list`, `vector`, `set`, `map` and `iterator` and the standard assortment of built-in functions to the primitive types and functions that the analysis understands. We introduce the labels (*@*, *B@*, *A@*) to represent the partitions introduced by the iterators in collections. Finally, we want to be able to determine which (if any) iterator variable is currently active in a given collection. To do this we add an *iter* field to the record that represents collection nodes. The *iter* field is either a variable name, indicating that the iterator with the given name is being used to partition the pointers in the collection or * to indicate that no iterator variable is currently being used to partition the collection.

***Modifications to the Dataflow Operators.*** Our modifications have only a minimal impact on the UMA algorithm and we only need to modify the node join algorithm. First, we define a simple function that takes a node and if it is currently partitioned on an iterator forgets all the partition and iterator information. The procedure to *forget* this information is shown in Alg. 1.

```
Algorithm 1: forgetIterator
  input : n a node
  if n has an active iterator then
      n.iter ← *;
      foreach out edge e do
          if e.offset ∈ {B@, @, A@} then e.offset ← ?;
```

When performing the join operation we check if the nodes that are being joined are from different *contexts* (graphs) then if the *iter* fields are the same we retain the iterator information, otherwise we forget it by calling the *forgetIterator* algorithm. This is safe since in both heap graphs the nodes being joined are partitioned by the same iterator variable and thus the joined node must be partitioned by the iterator variable. Since the edges in the UMA model represent *may* exist pointers, the edges from the collections are correctly handled.

## 6. Modeling Iterator and Collection Operations

In this section we look at how the various collection methods are implemented. Even our simplified collection library has a non-trivial number of methods to manipulate the various collection objects and the associated iterators. Thus, we focus on describing the most interesting methods. For simplicity we assume that all of the out edges for any given node have unique labels (no nodes have *ambiguous* edges).

***Forget and Clear Iterators.*** Our library collection semantics assumes that if the collection contents are modified then any active iterators are invalidated. To model the invalidation of an iterator we use a method, *clearIterator*, which first invokes the *forgetIterator* method to erase the iterator and associated edge partition. Then the *clearIterator* method joins all the *ambiguous edges*. This ensures that the collection will have (at most) a single edge with the label *?*.

***Insertion and Deletion.*** For the insert operation we first call the *clearIterator* method. Next we add an edge from the collection to the object that we want to add to the collection and we set the label of this edge as *?*.

The *delete* operation for our collection library takes an iterator and removes the element referred to by that iterator from the collection. To model this we remove the edge with @ label (which strongly deletes the iterator target from the collection).

***Iterator Initialization and Get.*** The most common way to initialize an iterator is to get an iterator to the first element (with respect to the collection's iteration order) of a collection. The `begin` method in our collection library is used to do this. To simulate the effect of this operation in the heap graph (Alg. 2) we use the *clearIterator* method to forget the partitioning of any other iterators on the collection. Then we create two edges: one is used to represent the element in the collection that the iterator refers to, the other edge is used to represent all the elements that come after the element referred to by the iterator. Since the iterator must refer to the first element in the collection (with respect to iteration order) we do not need an edge to represent elements that come before the iterator. Then, we see if the *?* edge has the interfere property *ip*. If it does we set the node that represents the contents of the collection as having *inConnected* edges (since the newly created edges must be connected), otherwise it is left unchanged. Finally, we delete the *?* edge.

**Algorithm 2**: iteratorBegin

```
input : $n$ a collection node, $v$ an iterator variable
$n.clearIterator()$;
if $n$ does not have an edge with label ? then return;
$e_? \leftarrow$ the edge with label ?;
$n_t \leftarrow$ endpoint of $e_?$;
$e_@ \leftarrow newEdge(@, 1, np)$;
$e_{A@} \leftarrow newEdge(A@, e_?.maxCut, e_?.interfere)$;
add edges $e_@$ and $e_{A@}$ from $n$ to $n_t$;
if $e_?$.interfere = ip then $n_t.connB \leftarrow C$;
delete edge $e_?$;
```

The `get` operator can be treated as a simple field load off the special field @. Using this approach passes all the work onto the existing UMA framework which performs the appropriate operation.

***Iterator Advance.*** After initializing an iterator we often want to advance it through the collection (the `advance` method) and use the `isValid` test to check if the iterator still refers to a valid point in the collection.

The advance method needs to re-label the existing edge with the @ label to have the *B@* label and create a new edge with the @ label that is parallel to the edge with the *A@* label (if such an edge exists). This is shown in Alg. 3, which assumes that the given iterator is valid and is the current active iterator for the collection.

**Algorithm 3**: iteratorAdvance

```
input : $n$ a node that represents a collection
if $n$ does not have an edge with label @ then return;
$e_{A@} \leftarrow$ the edge with label $A@$;
$n_t \leftarrow$ endpoint of $e_{A@}$;
re-label the edge with label @ to have label $B@$;
$e_@ \leftarrow newEdge(@, 1, np)$;
add edge $e_@$ from $n$ to $n_t$;
if $e_{A@}$.interfere = ip then $n_t.connB \leftarrow C$;
```

***IsValid.*** In the `isValid` method we want to (when possible) propagate the knowledge that on a given path `isValid` returned *true* or *false* and update the model to represent this information. If we take a branch that can only be executed when a given iterator is invalid then we want to update our model to reflect this information (Alg. 4). To do this we have two cases. If the given iterator is not the active iterator we do nothing. If the given iterator is the active iterator we only need to delete the edges with the @ label and the edges with the *A@* label. The *eraseEdgeWithOffset* removes the edge with a given offset from the abstract heap graph. Our current abstraction has no way to represent that an iterator must be valid so in the case that `isValid` returns *true* we do not do anything.

**Algorithm 4**: isValid$_{false}$

```
input : $i$ an iterator
$n \leftarrow$ the target of $i$;
if $i$ is not the active iterator for $n$ then return;
$n$.eraseEdgeWithOffset(@);
$n$.eraseEdgeWithOffset($A@$);
```

## 7. Examples

***Initialize the Set*** The set insertion example, Figure 1, demonstrates how the insertion operation works and provides an opportunity to develop some intuition into how the UMA algorithm works.

Figure 3(a) shows the abstract domain at the end of the first loop iteration. The variable `p` points to the `set` object and the variable `s` points to the object of type `t2` that all the elements in the set will reference (since variable connectivity is ignored the *connB* term is *D*). The first of the `t1` objects has been allocated and has had the `val` field set. Since we just allocated the object that the node represents we know it has a *size* of *1* and a *Singleton* layout.

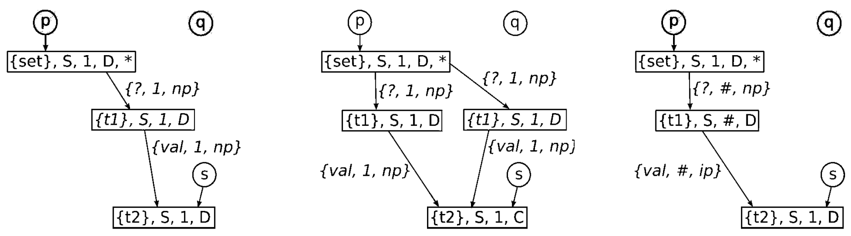
We also created an edge from the `set` object to the `t1` object. Since this edge was just created (by a store to an unknown location in the collection) it must represent a single pointer stored in the collection ($maxCut = 1$, $interfere = np$ and $offset = ?$). Since the variable `q` is dead at this point we explicitly nullify it.

Figure 3(b) shows the state of the heap model at the end of the second iteration. Another element has been allocated and inserted into the set. The `val` offset of this object has been set to refer to the same node that `s` points to. Since there are now two incoming edges that may be connected, the target node has the *connB* component set to *C*, indicating the `val` edges may be *inConnected.*

Since the abstract heap in Figure 3(b) is not in normal form (the `set` node has *ambiguous* edges) it needs to be normalized (see Section 3.3). This results in the abstract heap in Figure 3(c).

The two nodes with type `t1` have been combined into a new summary node. The edges with the labels *?* have been joined and are represented by an edge labeled $\{?, \#, np\}$ since the edge represents more than one pointer and the pointers cannot interfere. Finally, the edges with the labels `val` have been joined and are represented by an edge that is labeled $\{$`val`$, \#, ip\}$ since the edge represents more than one pointer and the pointers may interfere (the edges that were joined were *inConnected*). Running through the loop again produces the same result, thus we have covered all possible iterations of the loop and are done.

***Update the Set*** The second example from Figure 1 traverses all elements in the `set` (from the first example) and updates the `val` field of each object to refer to the same object as `r`. Figure 4(a) shows the state of the abstract heap after allocating a second object of type `t2` and initializing the iterator. We have set the iterator `i` to point to the `set` object, created a new edge to represent the single entry the iterator refers to (the edge with label @) and a new edge to represent the entries that come later in the iteration order (the edge



(a) Added First Entry (b) Added Second Entry (c) Normalize

**Figure 3.** Add Elements to a Set Container

with the label *A@*). When initializing the iterator the unknown edge *?* is *np* which means that the newly created edges (@ and *A@*) can not be *inConnected*. Thus, the refinement method can split the node that represents the `t1` objects into two nodes (one representing the heap reachable from the @ edge and one representing the heap reachable from the *A@* edge). Additionally, the @ edge has *maxCut* of size 1 and points to a *Singleton* node, thus the refinement algorithm can safely assume that the target has *size* 1 as well.

This allows the node to be strongly updated when the assignment is done. The result is shown in Figure 4(b). When the iterator is advanced we set the current @ edge to have the label *B@* and split a new out edge from the current *A@* edge. The result of this is shown in Figure 4(c), which is the state of the abstract heap at the end of the first abstract loop iteration.

The state of the heap model at the end of the second iteration is shown in Figure 4(d). The assignment was able to strongly update the target of the `val` field of the object referred to by the iterator, note that the *connB* flag in the node pointed to by `r` is set to *C* to denote that the edges are *inConnected*. The iterator advance has indexed the current iterator position, splitting out a new @ edge and resulting in two edges with the label *B@*. Thus, we need to combine their targets into a summary node and join the edges. This results in the abstract heap shown in Figure 4(e).

In Figure 4(e) we have some unknown number of pointers before the current iterator which all point to unique objects of type `t1` (the edge is *np*) and each of these objects has a reference stored in their `val` field, which (may) point to the same object as the variable `r`. Then we have the single element currently referred to by the iterator and some number of pointers that come after the iterator, which refer to the objects that have not been updated. The state shown in Figure 4(e) is also the repeated state of the abstract loop execution so we are done processing the loop body.

If we apply the exit test condition, `isValid`, which erases the edges with labels @ and *A@*, to the state shown in Figure 4(e), we get the result shown in Figure 4(f). Note that there are no longer any references from the objects in the `set` to the region of the heap pointed to by `s`: each element in the set was strongly updated and by modeling the progress of the iterator we determined that the contents of the collection have been strongly updated.

## 8. Performance

To evaluate the utility of the semantic model for the collections we examine a variation of the Jolden [2] benchmarks. The Jolden suite contains a number of pointer intensive kernels that are parallelizable using shape based approaches [5, 8]. The implementation in [2] does not utilize the Java collection libraries. Thus, we selected five of the benchmarks, and updated them to use the collection library `lists` and `vectors` instead of singly-linked lists and arrays (we also addressed the major issues in health [18]).

| JoldenWC | UMA Base | | | UMA Lib | | |
|---|---|---|---|---|---|---|
| Benchmark | Time | Shape | Speedup | Time | Shape | Speedup |
| bh | 2.58s | N | NA | 2.83s | P | 1.02 |
| em3d | 0.06s | N | NA | 0.11s | Y | 1.88 |
| health | 1.24s | P | NA | 1.56s | Y | 1.15 |
| power | 0.09s | Y | 1.68 | 0.38s | Y | 1.68 |
| tsp | 0.08s | P | 1.51 | 0.10s | Y | 1.51 |
| Overall | 4.05s | 1/2/2 | 1.23 | 4.98s | 5/1/0 | 1.44 |

**Figure 5.** Analysis and Parallelization Benchmarks

We ran the original UMA algorithm with the library code inlined so that it was analyzed directly. We then ran the algorithm using the collection semantics. To compare the accuracy of the results we report if the algorithm was able to determine the shape information for the data structures created by the programs and the performance improvement that was obtained by parallelizing based on this information. We use three categories for the accuracy of the shape analysis. *Yes* (Y) is used when the analysis is able to provide the correct shape information for all of the relevant heap structures. *Partial* (P) 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 (which may not matter for parallelization). *No* (N) is used when the analysis failed to correctly identify the shape of a substantial portion of the heap data structures.

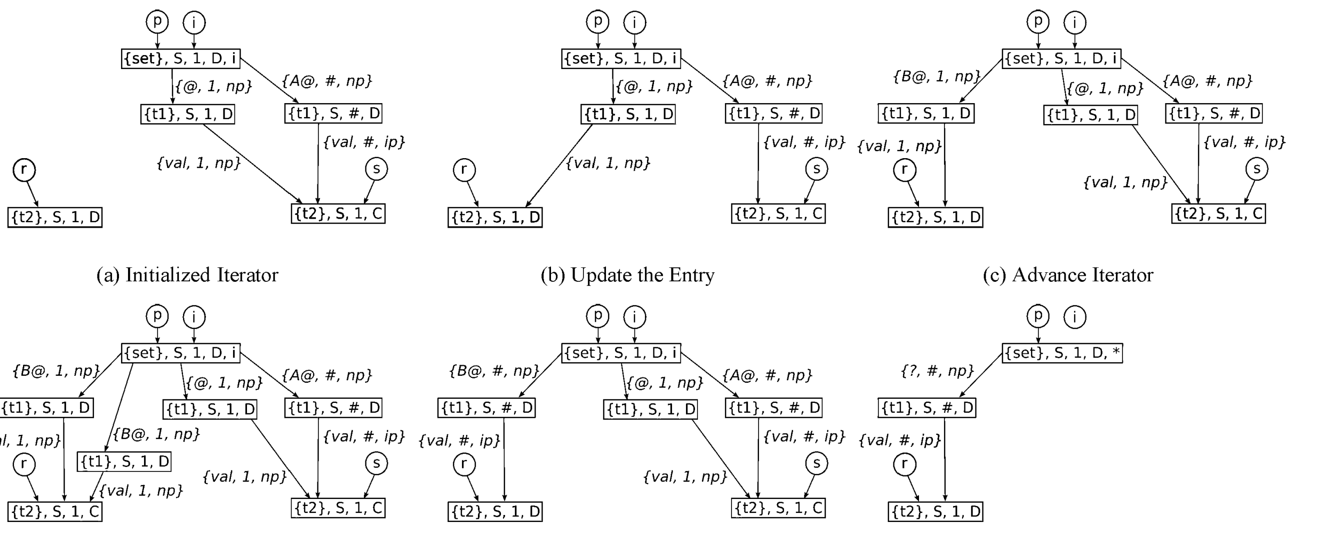
The UMA algorithm is written in C++ and compiled with gcc 3.3.5. The benchmarks were run on an Intel (Dual Core) PentiumD 2.8 GHz machine with 1 GB of RAM. The parallelization benchmarks were run with the default inputs from [2] on the same machine with the Sun Java 1.5 JVM.

The results in Figure 5 indicate that the use of semantics to model the collection objects results in much more accurate results than attempting to directly analyze the actual implementation of the collections. On our test system the maximum speedup is 2 and we did not employ any transformations other than parallelizing recursive tree calls and foreach parallelization. Given these constraints the average speedup of 1.44 indicates that, in general, the analysis is able to accurately model the connectivity of the program heap.

The increased analysis time when using the collection semantics is due to the refinement of sections of the heap graph that the base analysis is unable to expand, i.e. is due to a more accurate representation of the heap and not the implementation of the semantics.

## 9. Conclusion

This paper presented a technique for extending an existing heap analysis to handle various types of generic collection objects. Instead of attempting to extend the range of data structures that the target analysis understands our analysis treats the collections and iterators over the collections as opaque objects. By ignoring the internal representation of the collections we avoided the issues of model complexity and computational intractability.



(a) Initialized Iterator (b) Update the Entry (c) Advance Iterator

(d) End of Second Iteration (e) Normalize and Fix Point (f) Interpret Exit Test

**Figure 4.** Update Data in the Set

To handle the manipulation of these collections we introduced a partition scheme using the iterators in a collection. The partition is based on the idea that an iterator splits the elements in the collection into three classes (before the current position, the single element at the current position and elements after the current position). We then extended the UMA heap analysis with the semantics required to model the collections and iterators. The extended model is capable of identifying individual elements in the collections, performing strong updates on the individual elements and, by using the partition induced by the iterators, is able to model the iterative processing of the collection. This allows the heap analysis to accurately track destructive update operations that involve collections and their contents, which is critical to obtaining accurate analysis results when dealing with imperative programs.

The experimental results show that the analysis can achieve substantially more accurate results by using a semantics based approach instead of analyzing the library code directly. Further, the analysis is efficient enough to be of practical use in optimization and error detection applications.

## References

[1] F. Bueno, M. G. de la Banda, M. Hermenegildo, K. Marriott, G. Puebla, and P. J. Stuckey. A model for inter-module analysis and optimizing compilation. *LNCS*, 2001.

[2] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.

[3] D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[4] M. Codish, S. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *POPL*, 1993.

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

[6] D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, 2005.

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

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

[9] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *FSE*, 2005.

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

[11] V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE TSE*, 2006.

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

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

[14] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, 2001.

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

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

[17] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.

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