Heap Analysis in the Presence of Collection Libraries

Unified Memory Analysis: Analyzing Collections

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

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

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 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 iterator 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.

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 effectively deal with simple data structures, mainly lists, trees, and very simple cyclic structures [15, 18, 19, 16, 8]. 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 [15, 8] or due to the computational complexity of performing the analysis [18, 19, 16]. The use of these complex structures in the implementation of the general purpose containers in most standard library implementations and the inability to accurately/efficiently model them precludes using the existing techniques when analyzing programs that use library-based generic containers.

An alternative to directly analyzing the code that implements the container objects is to use the semantics of the collection objects to simulate the effect of each container operation as an atomic program operation. This approach is frequently used to analyze libraries or other modules [17, 3, 10, 12].

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 summarized regions back 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 for the heap models presented in the TVLA (Three-Valued Logic Analysis) based work [19, 8], the graph model presented in [14], or the UMA (Unified Memory Analysis) model [15]. In order to simplify the construction and to make the examples concrete we will focus on the UMA model.

Sections 3 and 4 provide a brief introduction of the concrete program model and the UMA heap model. Section 6 introduces the abstract semantics for the collections and extends the heap model with additional types and offsets. Using these new properties Section 7 outlines the abstract program operations for manipulating the containers and the associated iterators.

To assess the effectiveness of the semantic approach to collections we compare the results of analyzing some benchmarks when the library code is analyzed directly and then when using the collection semantics. Our results indicate that the use of the collection semantics is critical to achieving accurate shape analysis results in programs that make use of libraries (we use the shape information to perform thread level parallelization on our benchmarks and achieve an average speedup of 1.44 when running on a 2 processor machine). Further, the use of collection semantics has a minimal impact on the performance of the analysis (analysis times are less than 3 seconds for programs on the order of 1000 LOC that make extensive use of recursion and virtual methods).

2 Example Programs

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

We are using the *t1* and *t2* types in this paper to keep the examples simple. However, the methods presented in this paper can handle the examples, with the same level of accuracy, when *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.data = s
    p.insert(q)
```

Update all the elements in the set

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

Fig. 1. Example Code

When performing the analysis of these programs there are a number of properties that we want to identify. The first one is that every element in the *set* must be unique and every element may have a reference to the same value in the *data* field. In the second example the analysis should be able to capture the fact that on each iteration of the loop the *set* element that the iterator refers to has its *data* offset *strongly* updated and that after the loop all the elements in the *set* have been updated. Thus, there are no longer any objects in the set that have a pointer in the *data* field that refers to the same location as the variable *s*.

3 Concrete Programming Model

This section is a review of the relevant aspects of the concrete heap model presented in [15]. The analysis works on the strongly-statically typed, single-inheritance, thread-free, object-oriented imperative core of languages like Java or C#.

3.1 Concrete Language and Semantics

The 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 [13, 20].

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 $\rho = \{$`int`, `float`, `char`, `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.

3.2 Concrete Heap Definition

The concrete heap is modeled as a multigraph with labeled edges where objects and arrays are the vertices and the pointers are labeled directed edges in the graph. The term *cell* is used 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 is, $E \subseteq C \times L \times C$. 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 is defined as 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 $H = \mathscr{P}(H_s)$.

3.3 Heap Properties of Interest

Points-to and Paths. Given cells a, b along with offset o, $(a,o) \rightarrow_p b$ denotes a pointer p that has the label o (is stored at offset o) in a and points to b, $a \rightarrow_p b$ is used to indicate that $\exists$ offset o s.t.

$(a,o) \rightarrow_p b$. Two cells can be connected by a *path* ψ, $(a,o) \rightsquigarrow_\psi b$ indicates that ψ is a 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). Finally, $a \rightsquigarrow_\psi b$ denotes that $\exists o$ s.t. $(a,o) \rightsquigarrow_\psi b$.

Regions of the Heap. A *region* of memory $\mathfrak{R}$ 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 $C_\mathfrak{R} \subseteq C$, let $P_\mathfrak{R} = \{\text{pointer } p \mid \exists a,b \in C_\mathfrak{R}, a \rightarrow_p b\}$. Let $P_c = \{\text{pointer } p \mid \exists a \in C_\mathfrak{R}, x \notin C_\mathfrak{R}, a \rightarrow_p x \oplus x \rightarrow_p a\}$ Then a region is the tuple $(C_\mathfrak{R}, P_\mathfrak{R}, P_c)$.

Connectivity. Connectivity within a region describes how cells in the region are connected. For a region $\mathfrak{R} = (C_\mathfrak{R}, P_\mathfrak{R}, P_c)$ and cells $a,b \in C_\mathfrak{R}$, cells a and b are connected if they are in the same weakly-connected component of the graph $(C_\mathfrak{R}, P_\mathfrak{R})$; cells a and b are disjoint if they are in different weakly-connected components of the graph $(C_\mathfrak{R}, P_\mathfrak{R})$. 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) the cells c,d are connected in the region Z.

Structure Traversals. The basic UMA algorithm uses layout predicates (*Singleton*, *List*, *Tree*, *MultiPath*, and *Cycle*) to define how a program could traverse the memory in a given region. The layout of data structures in memory is an important property for many types of parallelization transformations [4, 5] and the ability to track these layout properties was one of the main goals of the UMA model. This paper does not require any modifications of the *Abstract Layout* domain so we only mention that these properties are tracked and omit a more detailed description.

4 Heap Model

The UMA abstract domain is based on an abstract heap graph model [2, 21, 11]. Each node represents a set of concrete cells 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.

4.1 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 $[0,\infty]$ (*#*). The other is a set of type names that represents all the possible types of the cells 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, `?`, to represent all the array indices. This special offset is used for the edge that represents all the pointers that are stored in the array.

The last of the basic properties is the *Abstract Layout*. This concept is used to abstract the traversals that are admissible for the region of the heap that a given node represents. Thus, a node may have a *Singleton*, *List*, *Tree*, *MultiPath*, or *Cycle* abstract layout.

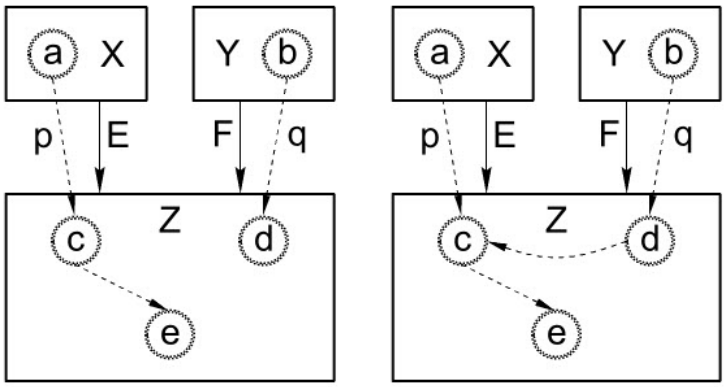
4.2 Pointer Connectivity Properties

The UMA model relies on tracking the potential that two pointers can reach the same cell to drive the tracking of the *Abstract Layouts* and to enable the refinement of the common case heap structures that it encounters. The UMA work defines two properties that track the potential that two pointers can reach the same cell in the region that a particular node represents.

The first property is when the pointers are represented by different edges in the heap model. Given the concretization operator γ 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).

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 *inConnected*.

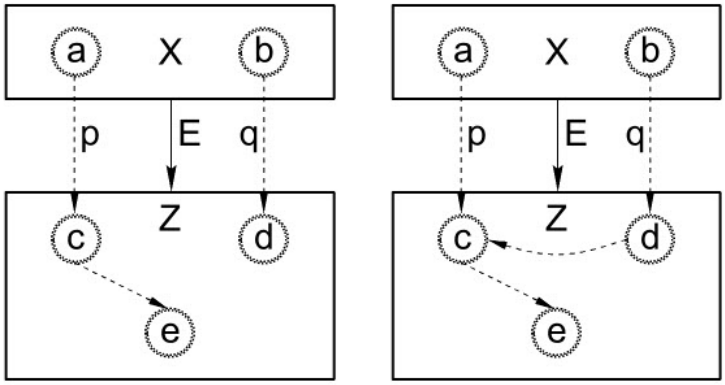


(a) E, F Disjoint in Z (b) E, F Connected in Z

Fig. 2. Concrete and Abstract Connectivity

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 cells 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.

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.



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

Fig. 3. Concrete Connectivity and Abstract Interference

4.3 The Heap Graph

Each node in the graph contains a record that tracks the types of the concrete cells that a node represents (*types*), the total number of cells 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 [15] 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 *de* if all the in edges must be disjoint and *ce* 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 relation. Thus, each node is represented as a record of the form `[types layout size connB]`.

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 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 for the heap graphs. This 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 in the heap graph.

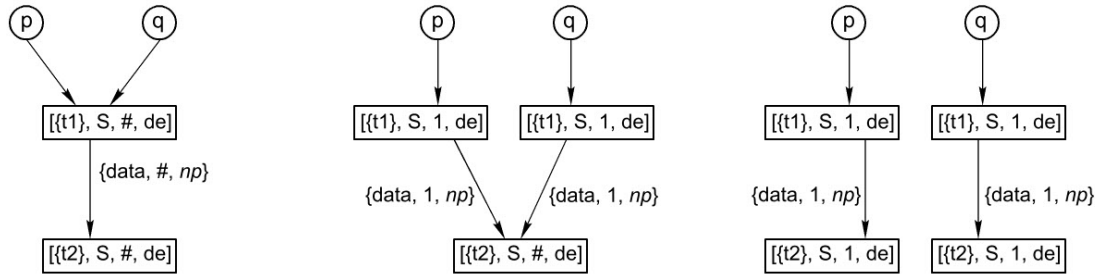
The program analysis is then performed using sets of the heap graphs to represent the possible program states at each point in the program.

5 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 if it is too aggressive. 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 (or variables) for a given node are disjoint. In this case we know that each of the edges represents a set of pointers which point into a disjoint sub-region of the region represented by the node. In this case the algorithm expands each sub-region into a separate node in the abstract graph (one for each disjoint edge).

Consider the case in Figure 4(a) where the variables p and q point to the same node (a node representing cells of type *t1*, with a *Singleton* layout, that may represent any number of cells and assume we know that all the incoming variables are disjoint). Since the variables *p* and *q* refer to disjoint sections of the node we can partition this summary node into two distinct nodes (one representing section reachable from *p* and one representing the section reachable from *q*). The partitioning results in Figure 4(b). Note that since the newly created nodes each only have a single variable pointing to them and have *Singleton* layouts. Thus, we know that the node can represent at most one cell and set the size to 1. Then since the node is *size* 1 the outgoing edges (except edges with the *?* offset) can only have a single pointer in them. Thus, the outgoing edges must have a *maxCut* of 1 and also be *np*. Since the edge that was split contained all non-interfering pointers the two edges incident to the node representing the cells with type *t2* cannot be *inConnected* (thus is marked *de*). This now allows us to apply refinement again—the results are shown in Figure 4(c).



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

Fig. 4. Refinement of a region with disjoint sub-regions

6 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 4.1.

- 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 container) 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 container) 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 [7] 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.

The Built-in Collections. The core language from Section 3.1 is extended with a simplified standard collection library, which provides *vectors*, *sets*, and *maps*. These collection objects support the standard assortment of insert, delete, copy, union, addBack, lookup, etc. operations. The collection library also has iterator objects that allow indexing on the containers. The iterator objects support indexing (*advance*), deletion (*erase*), an *isValid* test (returns false if the iterator is off the end of the collection or if the collection has been modified) and of course a *get* operator. The iterators can be initialized by calling *begin* on a collection to get an iterator that refers to the first element in a collection or by using the *find* function which returns an iterator referring to the item with the desired property (or an *invalid* iterator if no such element exists).

Modifications to the Model. To model the collections and iterators we need to extend the abstract domain from Section 4 with some additional properties. The most basic extension is to add the types *vector*, *set*, *map* and *iterator* to the type system. We also add the labels (*@*, *B@*, *A@*) that we introduced to the set of edge offsets. 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. Conveniently, our modifications have only a minimal impact on the UMA algorithm. We only need to modify the refinement methods, the node join and the node combine algorithms. Before we look at the algorithms we need to update, 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 in Algorithm 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 ← ?;
```

There are two things that should be checked for during the refinement operation. First, edges with the *B@* or *A@* labels should not have their *maxCut* or *interfere* properties changed (just like with the *?* label for arrays). Second, if we refine a node that has an iterator active in it we cannot know which of the refined nodes the collection should go in and thus which of the refined nodes will have the active iterator after the refinement. Thus, when refining a node we need to apply the *forgetIterator* algorithm before splitting it into multiple nodes.

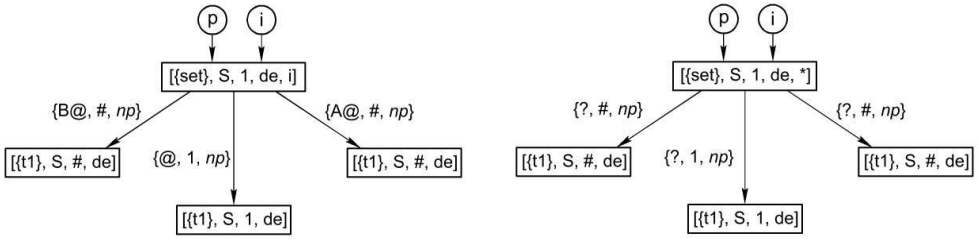
The other modifications that need to be made are in the node join and merge operations. In the case of the merge algorithm we conservatively forget all the information about the iterators in the two nodes that are being merged. For the join operation we want to be slightly more careful. If the nodes 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 node is partitioned by the same iterator and thus the joined node must be partitioned by the iterator. Since the edges in the UMA model represent *may* exist pointers, the edges from the collections are correctly handled by the existing model algorithms.

7 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*).

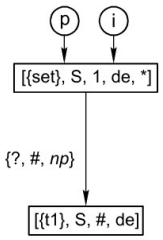
Forget and Clear Iterators. Our library collection semantics assumes that if the collection contents are modified then any active iterators are invalidated. To model the invalidation of an iterator we use a method, *clearIterator*, which first invokes the *forgetIterator* method to erase the iterator 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 *?*.

Figure 5 shows an example of the use of the *clearIterator* method. In Figure 5(a) we have a set that has the iterator *i* as its active iterator. We assume that there are some unknown number of elements that come before the iterator, represented by the edge with the *B@* label and some



(a) Iterator at some position in set

(b) Iterator forgotten



(c) Ambiguous edges joined

Fig. 5. Clear Iterator

unknown number of elements that come after the iterator, represented by the edge with the *A@* label. Then we have the single entry that the iterator is at, which is represented by the @ label. Note that the fact that the iterator *i* is the active iterator for the set, denoted by the *i* in the node record.

In Figure 5(b) the *forgetIterator* method has been applied. The set is now marked as having an unknown active iterator, the *iter* entry is now *. All the edges have been set to the *?* offset, which indicates an unknown position with respect to any iterators. Finally, Figure 5(c) shows the results of joining all the ambiguous edges from the set node (all the edges with the *?* offset and their target nodes have been summarized).

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

For the *delete* operation we have two versions. The first is when we are deleting an element from a collection based on equality. In this case if the node that is being removed has *size* of *1* and the edge from the container is *np* we can delete the incoming edge (strongly deleting the element from the collection), otherwise we must conservatively do nothing. Finally, we call the *clearIterator* method. In the case of deleting the element that is referred to by a given iterator, we can remove the edge with @ label (which strongly deletes the target) and then we call the *clearIterator* method.

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 we first call the *clearIterator* method to forget the partitioning of any other iterators on the collec-

tion. 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 outlines how this is done.

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;
$e_?$ ← the edge with label *?*;
n_t ← endpoint of $e_?$;
$e_@$ ← $newEdge(@, 1, np)$;
$e_{A@}$ ← $newEdge(A@, e_?.maxCut, e_?.interfere)$;
add edges $e_@$ and $e_{A@}$ from n to n_t;
if $e_?$.interfere = ip **then** $n_t.connB \leftarrow ce$;
delete edge $e_?$;

The other possibility for initializing an iterator is to use a find method, which will return an iterator to some arbitrary element in the collection. This case is almost identical to the *begin* method, except that we add an edge with the *B@* label to account for any elements that may come before the iterator.

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. Once we have initialized an iterator we usually want to advance it through the collection (the *advance* method) and then use the *isValid* test to check if the iterator still points to a valid point in a collection.

The advance method only needs to re-label the existing edge with the @ label to have the *B@* label and then 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 Algorithm 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;
$e_{A@}$ ← the edge with label *A@*;
n_t ← endpoint of $e_{A@}$;
re-label the edge with label @ to have label *B@*;
$e_@$ ← $newEdge(@, 1, np)$;
add edge $e_@$ from n to n_t;
if $e_{A@}$.interfere = ip **then** $n_t.connB \leftarrow ce$;

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. 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. Algorithm 4 presents method in detail. 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 ← the target of *i*;
if *i is not the active iterator for n* **then** return;
n.eraseEdgeWithOffset(@);
n.eraseEdgeWithOffset(*A*@);

8 Examples

8.1 Initialize the Set

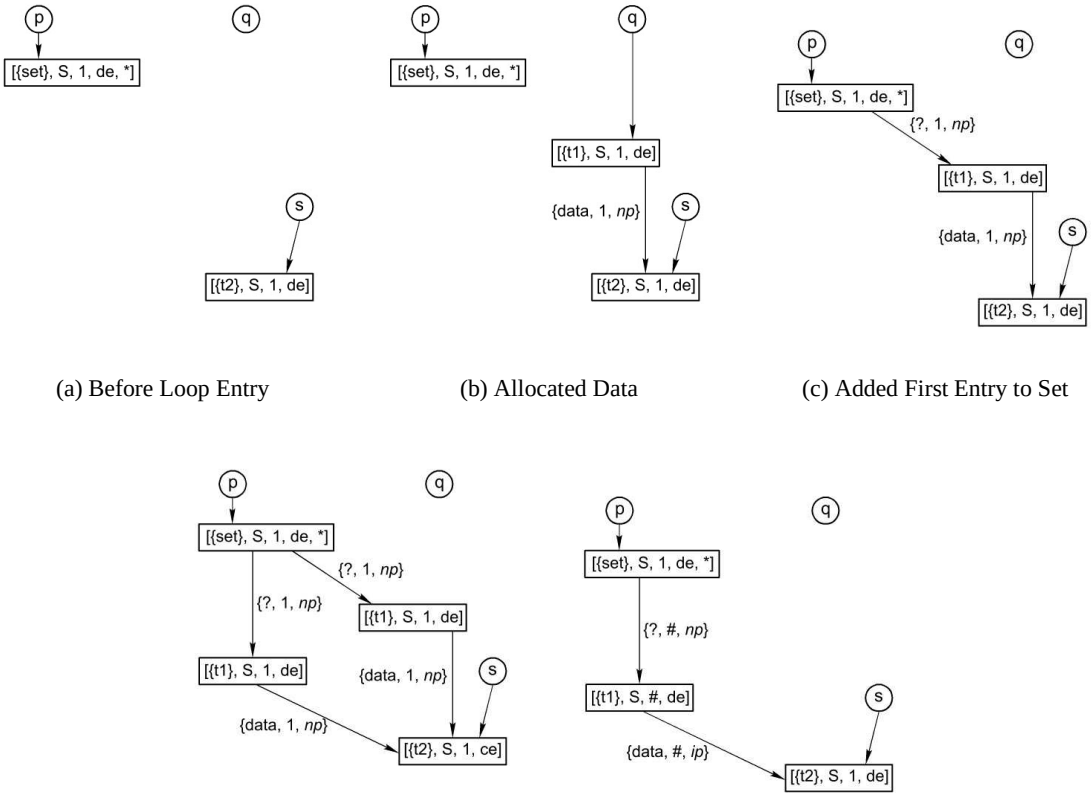
The set insertion example, Figure 1, demonstrates how the insertion operation works and provides an opportunity to review some of the properties from the UMA algorithm that are important to the operation of our collection analysis. Figure 6(a) shows the abstract state at the entry of the *for* loop. 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. In Figure 6(b) the first of the *t1* objects has been allocated and has had the *data* field set (since variable connectivity is ignored the *connB* term is *D* in the node *s* points to). Since we just allocated the object that the node represents we know it has a *size* of *1* and a *Singleton* layout. Since the *t2* node only has a single incoming edge (which by definition cannot be connected to another incoming edge) the *connB* relation is still *de*.

Figure 6(c) shows the effect of the *insert* operation. There is now 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 container) it must represent a single pointer stored in the container (thus having $maxCut = 1$, $interfere = np$ and $offset = ?$). Finally, since the variable *q* is dead at this point we mark it as dead and nullify it. This is the state of the abstract heap at the end of the first iteration of the loop.

If we proceed to the end of the second iteration of the loop the abstract heap model will be in the state shown in Figure 6(d). In this figure we have allocated and inserted another element into the set (again at an unknown offset, *?*). The *data* offset of this object has been set to refer to the same node that the variable *s* points to. Since there are two incoming edges that may be connected, the target node has the *connB* component set to *ce*, indicating that the *data* edges may be *inConnected*.

Since the abstract heap model in Figure 6(d) is not in normal form (the node representing the *set* has ambiguous edges) it needs to be normalized (see Sction 4.3). This results in the abstract heap in Figure 6(e).

The two nodes with type *t1* have been combined into a new summary node. The two edges with the labels *?* have been joined and are now represented by an edge that is labeled $\{?,\#,np\}$ since the edge represents more than one pointer and we know the edge represents pointers that cannot interfere. Finally, the two edges with the labels *data* have been joined and are now represented by an edge that is labeled $\{data,\#,ip\}$ since the edge represents more than one pointer and the edge represents pointers that may interfere (since the edges that were joined were *inConnected*). Running through the loop again produces the same result, thus we know we have covered all possible iterations of the loop and are done. As desired the analysis was able to determine that each entry in the set points to a unique object (the *?* edge is *up*), each of the objects of type *t1* may refer to the same object of type *t2* (since the *data* edge is *ip*), and this may be the same object the variable *s* points to (since the variable and the edge end at the same node).



(a) Before Loop Entry (b) Allocated Data (c) Added First Entry to Set

(d) Added Second Entry to Set (e) Normalize and Fix Point

Fig. 6. Add Elements to a Set Container

8.2 Update the Set

The second example from Figure 1 traverses all elements in the *set* that was built in the first example and updates the *data* field of each object to refer to the same object as the variable *r*. Figure 7(a)

shows the heap model after allocating a second object of type *t2*, which is pointed to by *r*. Figure 7(b) shows the state of the abstract heap after initializing the iterator. We have set the iterator variable *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 with the label *A@*). When we initialized the iterator the unknown edge *?* was *np* which meant that the newly created edges (@ and *A@*) could not be *inConnected*. This allowed the refinement method to 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, since the @ edge has *maxCut* of size 1 and it points to a *Singleton* node the refinement algorithm can safely assume that the target node of the @ edge has *size* 1 as well.

This allows the node to be strongly updated when the assignment is done. The result is shown in Figure 7(c). 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 7(d), which is the state of the abstract heap at the end of the first abstract loop iteration.

The second assignment is again able to strongly update the target of the collection entry, resulting in the state in Figure 7(e). Note that the *connB* flag in the node pointed to by *r* is set to *ce* to denote that the edges are *inConnected*.

We again advance the iterator, which gives the state in Figure 7(f). Since we now have two edges with the label *B@* we need to combine their targets into a summary node and join the edges. This results in the abstract heap shown in Figure 7(g).

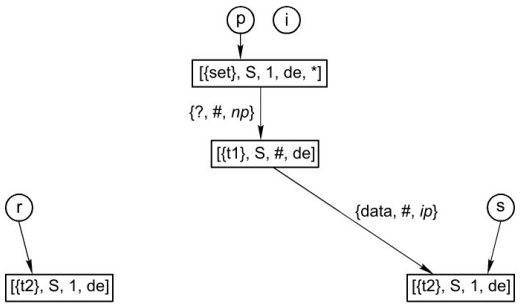
In Figure 7(g) we have some unknown number of pointers before the current iterator which all point to unique objects of type *t1* (the edge is *np*) and each of these objects has a reference stored in their *data* field, which (may) point to the same object as the variable *r*. Then we have the single 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 7(g) 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 show in Figure 7(g) we get the result shown in Figure 7(h). 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.

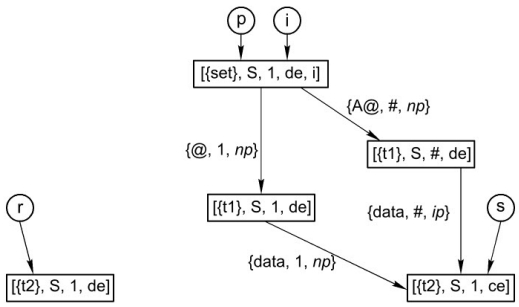
9 Performance

Theoretical Performance. The base UMA analysis has a worst case runtime of $O(n^2k^4)$ for any of the model operations, where n is the number of nodes in the abstract heap graph and k is the max number of incident edges for any node. All of the extensions in this paper involve the creation, deletion, or joining of a constant number of edges. This implies that the runtime of the extended method is also $O(n^2k^4)$ per model operation.

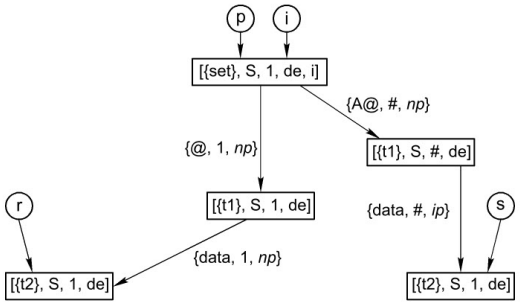
Experimental Results. To examine the effectiveness of our semantic model for the collections we decided to look at a variation of the Jolden [1] suite of benchmarks. The Jolden suite contains a number of pointer intensive kernels that are parallelizable using shape based approaches [6, 5, 9]. However, the implementation in [1] does not use any of the Java collection libraries. Thus, we



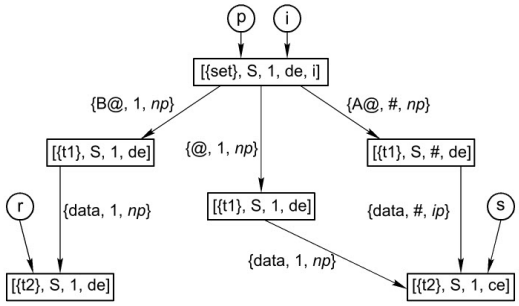
(a) Before Loop Entry



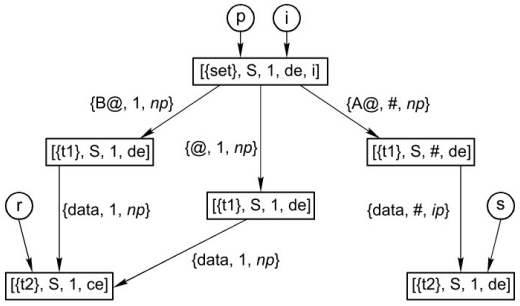
(b) Initialized Iterator



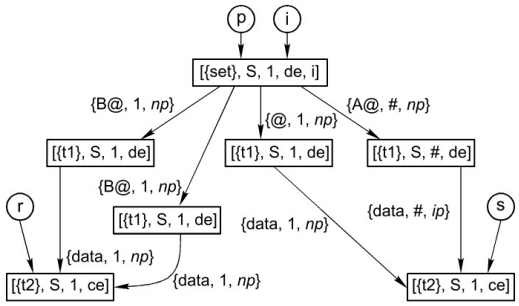
(c) Update the Entry



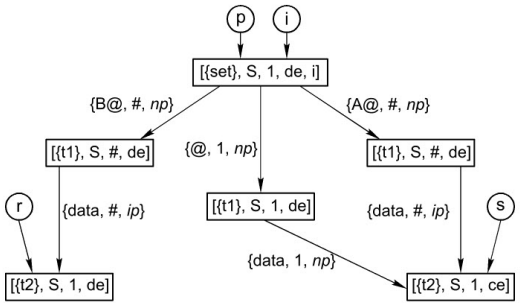
(d) Advance Iterator



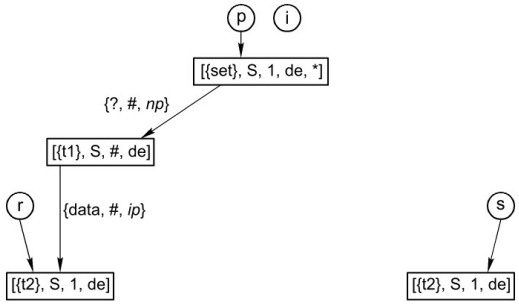
(e) Second Update



(f) Advance Again



(g) Normalize and Fix Point



(h) Interpret Exit Test

Fig. 7. Update Data in the Set

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

Fig. 8. Benchmarks

selected five of the benchmarks, and updated them to use the collection libraries where possible (we also addressed the major issues in health [22]).

We ran the original UMA algorithm with the library code inlined so that it was analyzed by the UMA algorithm. We then ran the UMA algorithm using the collection semantics. To compare the accuracy of the results we report if the algorithm was able to correctly determine the shape information for the data structures created by the programs and the performance improvement that was obtained by parallelizing the programs based on this shape information. We use three categories for the accuracy of the shape analysis. *Yes* (Y) is used when the shape analysis is able to provide the correct shape information for all of the relevant heap structures in the program. *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 was compiled using gcc 3.3.5. The benchmarks were run on a 2.2 GHz Intel (Dual Core) PentiumD 2.8 GHz machine with 1 GB of RAM (although memory consumption never exceeded 5 MB). The results are show in Figure 8. The parallelization benchmarks were run with the default inputs from [1] on the same machine using the Sun Java 1.5 JVM.

The results in Figure 8 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 containers. 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 increase in analysis time when using the collection semantics is due to the refinement and analysis of sections of the heap graph that the base UMA analysis was unable to expand. Thus, the slowdown is due to the more accurate representation of the heap and is not caused by the implementation of the collection semantics.

10 Conclusion

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

To handle the manipulation of these containers 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 container 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. This extended model is capable of identifying the individual elements in the containers, 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 container. This allows the heap analysis to accurately track destructive update operations that involve containers and their contents, which is critical to obtaining accurate analysis results when dealing with imperative programs.

Our experimental results show that the analysis can achieve substantially more accurate results when using the semantics based approach than when attempting to analyze the library code directly. Further, the extensions to the analysis are all polynomial time (which is the same as for the model operations in the basic UMA method) and in practice analyzing code with library collections is efficient enough to be useful in optimization applications.

References

1. B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in Java. In *PACT*, 2001.
2. D. R. Chase, M. N. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.
3. M. Codish, S. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *POPL*, 1993.
4. 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.
5. R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *POPL*, 1998.
6. R. Ghiya, L. J. Hendren, and Y. Zhu. Detecting parallelism in C programs with recursive data structures. In *CC*, 1998.
7. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, 2005.
8. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.
9. L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE TPDS*, 1(1), 1990.
10. T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *FSE*, 2005.
11. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of lisp-like structures. In *POPL*, 1979.
12. V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 2006.
13. C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, 2004.
14. T. Lev-Ami, N. Immerman, and S. Sagiv. Abstraction for shape analysis with fast and precise transformers. In *CAV*, 2006.
15. M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *LCPC*, 2006.
16. N. Rinetzky, J. Bauer, T. W. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.
17. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, 2001.
18. S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL*, 1996.
19. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, 1999.
20. R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON*, 1999.
21. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.
22. C. Zilles. Benchmark health considered harmful. In *Computer Arch. News*, 2001.