Automatic Inference of Reference-Count Invariants

David Detlefs
Sun Microsystems
1 Network Drive, Burlington, MA 01803
david.detlefs@sun.com

ABSTRACT

While we know how to efficiently collect short-lived garbage, collecting long-lived garbage usually requires expensive tracing traversals. This paper suggests a static analysis technique that can identify program points that make objects unreachable, allowing automatic insertion of explicit deallocation functions.

1. INTRODUCTION

This paper proposes a new form of compile-time garbage collection. While techniques such as generational garbage collection make it easy to efficiently collect short-lived objects, collection of long-lived objects can be considerably less efficient. Techniques to identify exactly the point in execution at which an object becomes unreachable might significantly decrease collection overhead.

2. ILLUSTRATIVE EXAMPLE

Consider the simple object type shown in Figure 1 (in the Java™ Programming Language). It implements a set of objects using a linked list representation. This example is illustrative of more practically interesting situations: for example, the bucket lists of a closed-address hash table are essentially the same data structure.

The aim of this paper is to suggest for data structures like this that it might be feasible to automatically infer invariants that would allow a compiler to safely insert code to deallocate `Node`s that become unreachable because of `delete`. That is, the compiler would essentially insert a new line "34.5": `free(hd)`.

3. EXAMPLE INVARIANT

What invariant would be necessary to justify the safety of this deallocation? First, we add two new implicit *specification variables* to the program's state space:

- `rc`: a map from objects to integers, indicating the reference count of each object.
- `owner`, a map from objects to objects, indicating the ownership relationship (as in *ownership types* [6]).

The precondition of `free(o)` is that `rc[o] = 1`; I will consider `free` to take its argument by reference, and to have the side effect of setting the argument to `null`.[1]

I believe that tracking `owner` relations can greatly aid program verification (which has much in common with the analysis described in this paper). One of the most difficult aspects of verification of programs in languages with multiple levels of data abstraction is translating a specification of what variables may be modified at one level of abstraction into modifiable variables at a more concrete level (see, e.g., [9]). The `owner` relation can be very useful in this translation. An *abstraction function* defines an abstract variable a of an object in terms of more concrete variables $c_1, ..., c_n$. When objects are involved, this definition may involve fields of sub-objects – in our example, the abstract state of the `SetViaList` object is defined in terms of fields of `Node` objects, but which `Node` objects? I claim that we should restrict the domain of such definitions to those `Node` objects owned by the `SetViaList` object. With this restriction, if a method of object x modifies only fields of objects owned by x, then it modifies the abstract state of x only. A method of x that modifies fields of objects whose ownership is unknown would have to be assumed to modify all abstract fields of all objects, and verification of such a method's modifies clause would therefore fail.

Note that this methodology does *not* necessarily require that owned objects are reachable only from their owner. This methodology allows the possibility of controlled "rep exposure," since exposed objects may not be modified without altering the abstract state of the owner.

A newly allocated object is unowned, and may not be in the domain of any abstraction function. The ownership

[1]Obviously, this is impossible to express in the Java language.

```
 1 | class SetViaList implements Set {
 2 |   private class Node {
 3 |     public Object elem;
 4 |     public Node   next;
 5 |     public Node(Object e; Node n) {
 6 |       elem = e; next = n;
 7 |     }
 8 |   }
 9 |
10 |   private Node head;
11 |
12 |   public SetViaList() { head = null; }
13 |
14 |   public boolean contains(Object o) {
15 |     Node hd = head;
16 |     while (hd != null && !o.equals(hd.elem))
17 |       hd = hd.next;
18 |     return hd != null;
19 |   }
20 |
21 |   public void insert(Object o) {
22 |     if (contains(o)) return;
23 |     // Otherwise...
24 |     Node n = new Node(o, head);
25 |     head = n;
26 |   }
27 |
28 |   public void delete(Object o) {
29 |     Node hd = head;
30 |     Node prev = null;
31 |     while (hd != null) {
32 |       if (o.equals(hd.elem)) {
33 |         if (prev == null) head = hd.next;
34 |         else prev.next = hd.next;
35 |         return;
36 |       } else {
37 |         prev = hd;
38 |         hd = hd.next;
39 |       }
40 |     }
41 |   }
42 | }
```

Figure 1: Set implemented as a linked list.

of an object may be set freely if the precondition that the object is unowned can be proved. A method of an object may reset ownership of objects it owns to unowned.

I claim that predicates involving ownership are useful in formulating *class invariants* for object types whose representation involves pointer-based data structures, such as the set example I have presented. I intend the term *class invariant* to denote a property that holds at the boundaries of all public methods of a class. There is the question of calls between methods: I intend that private helper methods may violate the invariant, but public methods must preserve it. In our example, when `insert` calls the public method `contains`, the inferred invariant must hold at the call, and be preserved by `contains`. Public static methods should preserve such invariants for all objects they modify.

As an example, the following class invariant is a sufficient condition for inferring the required precondition for the introduction of `free` in the `delete` method.

$$
\begin{array}{l}
\forall s : \text{SetViaList} : \\
\quad (\quad s.\text{head} = \text{null} \\
\quad \vee (\quad s.\text{head} \neq \text{null} \wedge \text{owner}[s.\text{head}] = s \\
\qquad \wedge (\forall n : \text{Node} : n \neq \text{null} \wedge \text{owner}[n] = s \Rightarrow \\
\qquad\quad (\quad \text{rc}[n] = 1 \\
\qquad\quad \wedge (\quad n.\text{next} = \text{null} \\
\qquad\quad\quad \vee (\quad n.\text{next} \neq \text{null} \\
\qquad\quad\quad\quad \wedge \text{owner}[n.\text{next}] = s))))))
\end{array}
$$

This says that for any set s, either s is empty ($s.\text{head} = \text{null}$), or else s is non-empty and s is the owner of the head element of the list. In addition, we also have a quantified formula expressing a property true of all non-null `Node`s owned by s: their reference count is exactly one, and their next field is either null or points to another non-null `Node` owned by s. In the latter case, the quantified formula again applies to this node, allowing us to prove that all elements in the list are owned by s and have the desired reference count.

4. INVARIANT INFERENCE

The plan is to automatically infer invariants like the above via abstract interpretation. It is well-known that abstract interpretation can infer various kinds of invariants. One infers the strong properties that hold at given program points along certain executions, then weakens those properties in order to encompass all possible executions. If the properties and weakening strategy are chosen judiciously, then the result may still be strong enough to be useful. I propose to apply abstract interpretation in a novel way: to the entire class, in order to infer invariants of the class. Just as abstract interpretation over a procedure body would assume some start state and begin at the first statement of the procedure, interpretation over the class would start with the constructor(s) of the class, and consider each method a possible successor of the constructor(s) and other methods. In languages with destructors/finalizers, these would be potential successors, but would themselves have no successors. Figure 2 shows the analysis control flow within a class.

4.1 Abstract method execution

In our example the constructor is particularly simple: we would infer the tentative class invariant

$$\forall s : \text{SetViaList} : s.\text{head} = \text{null} \quad .$$

Let's say that next we consider `insert` as a successor. The `this` argument is a `SetViaList`, so we assume that our tentative invariant holds of it; that is, that `this.head` is `null`. As discussed previously, the call to the public `contains` method constitutes a point at which a class invariant must hold. If the calling method `insert` had made modifications before the call, then the state at the point of the call would be merged into the class invariant state, and only the resulting invariant would be assumed on return from the call. In our example, `insert` makes no modifications before calling `contains`, so this concern is not an issue. In addition, `contains` has no side effects, so we can ignore its effects on the analysis of `insert`.

Next, the allocation of a new `Node` yields a new object r, distinct from all currently allocated objects, such that

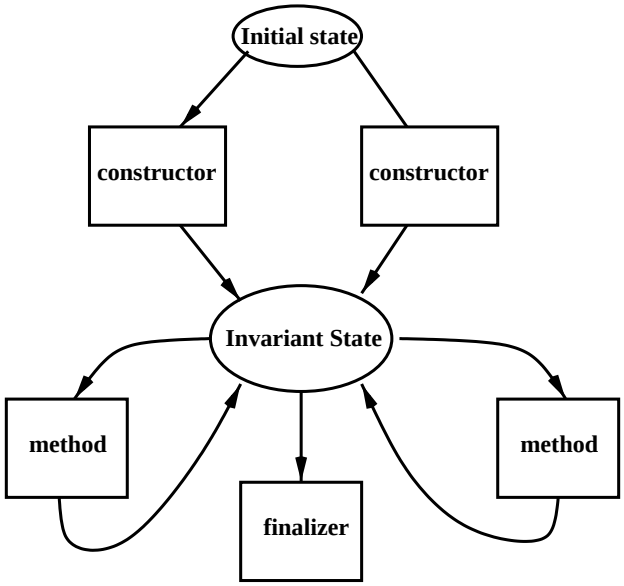

Figure 2: Analysis flow within a class

`owner`$[r]$ = `none` and `rc`$[r] = 1$. This reference is the one due to the local variable `n` in our example.[2] Next we consider the constructor invocation for the new `Node`, essentially "inlining" it into the analysis. We know nothing about the `o` argument beyond its type, so the assignment to `elem` does not augment our view of the program state. We have assumed that `this.head` is `null`, so the `next` field of the new `Node` becomes `null`.

Actually, I glossed over a subtle point in the argument above. We have not discussed any aliasing analysis; the assignment `elem = e` in the `Node` constructor increments the reference count of the current value of `e`; this reference value might also be held by other variables. For example, one could imagine convoluted code in which a `SetViaList` were inserted into itself! In that case, the `elem = e` assignment increments the reference count of the `this` variable of the `insert` method. To handle this issue correctly, I must of course assume that an update of the `rc` map at some reference r leaves `rc` unchanged only at references r' that are provably distinct from r. In my example, this is not a problem, because we're interested only in the reference counts of the `Node` objects, and a reference to a newly allocated `Node` is distinct from all previously-allocated references.

Finally, line 25 sets `head` to the new node. This temporarily increases `rc`$[r]$ to 2, but this quantity goes back to 1 when the local variable `n` becomes a dead variable after its last use at 25. In addition, we apply a heuristic to this assignment, giving it the additional side-effect of making `owner`$[r]$ = `this`. This may seem somewhat *ad hoc*; what rules govern this heuristic, and why are they justified? We infer an ownership change an the first assignment of a reference r to an unowned object to a field of some other object o. If the ownership of o is known, then we make n have the same owner. Otherwise, we assume heuristically that o may be a `this` value for the class under analysis, and make o the owner of r. This last rule is the one invoked in our example. A `SetViaList` that kept the elements in some order, and would therefore insert elements in the interior of the list, would use the previous rule.

[2] At the bytecode level, the initial reference is on the operand stack; this reference is consumed when it is moved to the local variable `n`.

4.2 Invariant Merging

The post-state resulting from this abstract execution of `insert` both violates and extends our tentative class invariant. We need to merge the previous invariant (since it correctly describes the post-state of the constructor) with a predicate describing the post-state of `insert` (from this assumed initial state), to get a predicate describing both. The post-`insert` program state is described by

$$\begin{aligned} & \texttt{this}.\text{head} \neq \text{null} \\ \wedge\ & \text{owner}[\texttt{this}.\text{head}] = \texttt{this} \\ \wedge\ & \text{rc}[\texttt{this}.\text{head}] = 1 \\ \wedge\ & \texttt{this}.\text{head}.\text{next} = \texttt{null} \quad . \end{aligned}$$

We abstract the above predicate over `this`, and merge via disjunction with the previous tentative invariant, to get a new tentative invariant:

$$\begin{aligned} & \forall s : \text{SetViaList} : \\ & \quad (\quad s.\text{head} = \text{null} \\ & \quad \vee (s.\text{head} \neq \text{null} \ \wedge\ \text{owner}[s.\text{head}] = s \\ & \qquad \wedge\ \text{rc}[s.\text{head}] = 1 \wedge s.\text{head}.\text{next} = \text{null})) \quad . \end{aligned}$$

Stating this invariant in prose: the `head` field of a `SetViaList` s is either `null`, or it is a non-`null` `Node` whose owner is s, whose reference count is one, and whose `next` field is `null`.

When the class invariant changes, this changes the start state of all methods; we reach a fixed point only when we have an invariant that all methods preserve. In our case, we could abstractly interpret `insert` again, starting from a state described by the new start state. Since this start state involves a disjunction, this implies several interpretations, one starting from each of the states in the disjunction. We would merge the predicates describing the resulting post-state. In this way, we could add another disjunction describing sets with zero, one, or two elements. At some point we would need to abstract further in hopes of reaching an invariant sufficiently general to constitute a fixed point.

I propose *abstraction over ownership* as an appropriate generalization technique. Given a predicate such as the one shown above, this technique would heuristically recognize conjunctions stating that some term (such as s.head in our example above) is non-null and has a given owner and also has some other properties. It would propose the generalization that *all* non-null instances of the type of the term (i.e., `Node` in our example) that have the given owner also have the other properties. In the truncated example we've given this process would infer the invariant:

$$
\begin{array}{l}
\forall s : \text{SetViaList} : \\
\quad (\quad s.\text{head} = \text{null} \\
\quad \vee (\quad s.\text{head} \neq \text{null} \ \wedge \ \text{owner}[s.\text{head}] = s \\
\qquad \wedge (\forall n : \text{Node} : n \neq \text{null} \wedge \text{owner}[n] = s \Rightarrow \\
\qquad\quad \text{rc}[n] = 1 \wedge n.\text{next} = \text{null}))) \quad .
\end{array}
$$

This invariant is not correct: it is not the case that all `Nodes` have null `next` fields. This is a result of applying this generalization too early, before sufficiently many possibilities have been elaborated. Consider applying this technique to the more complicated "concrete" predicate we would have obtained from executing `insert` once more from start states described by the last tentative invariant: we would obtain a predicate describing sets with zero, one, or two elements. This predicate describes all the possible situations: in the two-element case there would be a conjunction saying that the first element of the list was a non-null `Node` owned by the set, and that its next field was non-null and also owned by set. From this possibility we would merge (after abstraction over ownership) to the desired invariant we gave originally, in which the next field of a `Node` is null or names another `Node` with the same owner.

The other method that has side effects is `delete`, so we must verify that `delete` preserves the invariant. Here there are some details we need to resolve. In the case where the element being deleted is found, we need to ensure that the modifications made to the data structure do not violate the invariant. In this case, we update either the `head` pointer of the set object or the `next` pointer of one of the nodes (lines 33 and 34 in Figure 1). This pointer updating will decrease the reference count of the deleted node to zero, and also, if `hd.next` is non-null, increase the reference count of that node to two. (By the invariant, we knew that both reference counts were one at the start of the method.) This latter fact would seem to violate the invariant; we need a technique to do "compile-time deallocation," recognizing that the inserted call to `free(hd)` updates the `rc` map not only for `hd`, but also decrements reference counts of objects recursively reachable from `hd`, recursively freeing them if they become zero. This may create an intractable program verification problem; if so, it would be perfectly safe to bound this recursion at, say, a single level. The reference counts would be incorrect, but conservatively: they might prevent a deallocation that could be permitted, but they will not permit an incorrect deallocation. In our example, a single level would mean decrementing but not freeing the reference count of `hd.next`, which is exactly what we require to re-establish the invariant.

Also, the invariant would seem also to fail for the freed node, since it is a non-null `Node` owned by s, but its reference count is no longer one. We will solve this problem by having `free(o)` have the further side effect of setting `owner[o]` to `none`. The specification of `void free(Object o)` is now:

$$
\begin{array}{l}
\textbf{Requires:} \ \text{rc}[o] = 1 \\
\textbf{Modifies:} \ o, \text{rc}, \text{owner}, \forall f : f \text{ fieldof } o : \text{rc}[o.f] \\
\textbf{Ensures:} \ \text{rc}'[o] = 0 \ \wedge \ \text{owner}'[o] = \text{none} \\
\quad \wedge (\forall f : f \text{ fieldof } o : \text{rc}'[o.f] = \text{rc}[o.f] - 1) \\
\quad \wedge o' = \text{null} \quad .
\end{array}
$$

I have again glossed over another subtle issue in the discussion of the `delete` method. As discussed previously, when we modify the `rc` map at some expression e that evaluates to a reference value r, we are also modifying `rc` at other expressions e' that also evaluate to r. So again there is the question of aliasing: when (for example) we increment the reference count of a deleted node's successor, how do we know that the list doesn't have a cycle, allowing the successor node to be the successor of more than one node? The answer is contained in the reference count invariant itself. Since we know that the successor node's reference count was one before the update, we know there is no such aliasing. In general, if t is some object, and we know that `rc`$[t] = k$, and we are in a state where k *distinct locations* contain t, then we can conclude that no other variable can be equal to t. (To put this idea on a more formal footing: distinct local variables are distinct locations; all local variables are distinct from object fields; distinct object fields in the same object are distinct locations; and any pair of fields in distinct objects are distinct locations.)

4.3 Summary of Invariant Inference

This generalization technique is clearly the most speculative part of this proposal, and details still need to be worked out. It may, for example, seem overly heuristic. Note, however, that in techniques like this we can always just "give up," generalizing fully to the trivial invariant `true`. Since this application is intended to introduce an optimization, failure means only that the optimization may not be applied in this case.

One might worry that I've oversimplified the exposition of the set example by carefully choosing the order in which methods where chosen for interpretation. I don't think so: the order should not matter. The `contains` method has no side effects, and therefore has no effect on any class invariant. If we interpret `delete` immediately after the constructor, it has no side effects. If we interpret it after one execution of insert (i.e., with zero or one elements in the set), it either leaves the state unchanged or deletes the single element. In either case, the result state is covered by the tentative invariant. The same is true after two invocations of insert, until we reach the generalized invariant that is a fixed point of the analysis.

5. OTHER EXAMPLES

In this section I will sketch some other situations in which this technique might apply.

As mentioned previously, a closed hash table is simply several instances of this data structure ("bucket lists") pointed to from an array. The invariant inference would require an extra level of abstraction over the indices of the array, but would essentially be the same. A binary tree implementation of a set or mapping is also very similar to the `SetViaList` example. There is (conventionally) an internal node type. Instances of this node type are private to the tree (i.e,, we may infer that they are owned by the tree), and have reference count one, as a class invariant.

We do not intend this technique to be confined to data

structures whose elements all have reference count one. We could imagine elaborating the tree example by having every non-root node contain a reference to its parent. The invariant to be inferred would become more complicated, essentially saying that all tree nodes are owned by the top-level set or mapping object, that all non-root, non-leaf nodes n (those with non-null child pointers) have $\texttt{rc}[n] = 3$, and that leaf nodes l (those with null child pointers) have $\texttt{rc}[l] = 1$. But the inference mechanism should (I hope) be essentially the same. A similar example is a circular doubly linked list, pointed to by some outer owner object. Here the desired invariant is more complicated. Reference counts of owned nodes are either 3, for the list node pointed to by the owner object, or else 2 for all other nodes. List pointer fields in both directions point to other nodes owned by the same owner. It will be interesting to see whether these techniques can infer the desired invariants for these more complicated examples.

6. OBJECT COMPOSITION

Suppose that class type `X` has a field `f` of type `Set`. But `X` accesses this sub-object only through its interface; `X` doesn't know that (say) the underlying implementation of the `Set` is `SetViaList`. Using techniques like the ones we describe above to analyze `X`, it may be possible to identify points at which field `x.f` (for `x` of type `X`) becomes unreachable. In this case, we would like to delete not only the outermost `SetViaList` object, but also all of the `Node` objects it owns. (Note, of course, that we cannot safely delete the set elements referenced by those nodes.)

We can achieve this by giving every object an implicit *destructor*. The purpose of this is only to deallocate private state; this is accessible only to the language implementation, and is separate from any language-level finalization mechanism. (I should mention that classes with GC-initiated finalizers should not be explicitly deallocated – the finalization mechanism usually creates an implicit reference to the object that should be counted in the object's reference count when it is first allocated.) The default destructor implementation is to do nothing. For classes such as `SetViaList` for which we've been able to infer an interesting invariant, however, we may be able to use this invariant to synthesize the desired destructor. The idea would be to recognize conditions on objects reachable from owner objects as starting points for loops, and ownership invariants as loop generators. That is, it's not hard to imagine going from our inferred class invariant to

```
class SetViaList implements Set {
  ...
  private void destroy() {
    Node t = s.head;
    while (t != null) {
      // loop invariant: rc[t] = 1 && owner[t] = s
      Node next = t.next;
      free(t);
      t = next;
    }
  }
```

Note that the precondition for `free` is established by the loop invariant.

7. CAVEATS

This paper tries to work out the techniques in enough detail to convince the reader that it could be mechanized; however, I should be clear that this is a paper design and we have implemented very little of this yet. Reviewers had a number of questions about the technique, some of which I answered in the revised text, others of which I address here.

I have steered clear of multi-threading issues: many of the assumptions made are valid only if there is no concurrency between methods of a class. This could be enforced in a multithreaded setting by requiring all public methods of the owning object to be synchronized.

Obviously, we cannot support public data fields that participate in an invariant, since such fields may be modified arbitrarily. Actually, this caveat applies to class-local analyses; a whole-program analysis might be able to treat each modification of a public field as a "mini-method," and show that such modifications preserve an invariant. The motivating example in this paper has been simple enough to avoid calls to methods outside the analyzed class (or its private classes, such as `Node`), and thus has appeared as a class-local analysis. In more complicated examples with such calls, the analysis would become more like a whole-program analysis. However, calls to methods outside the analyzed class could be treated with much less precision. Essentially we are only interested in whether the call can modify state relevant to the invariant we are inferring. If so, then it should be "inlined" into the analysis. If not, it can be ignored (as a call with arbitrary side-effects on variables outside the scope of the analysis).

8. RELATED WORK

In this section I survey various categories of related work.

After a fairly long period of neglect by the memory management community, there has been a recent resurgence of interest in reference counting techniques. Large commercial applications often have large amounts of long-lived, infrequently updated heap data. For such applications (at least for their long-lived data), reference counting is attractive because its cost is proportional to pointer mutation rates, rather than to total heap size. This work includes work by Bacon *et al.* [1], Levanoni and Petrank [16], and Blackburn and McKinley [2].

There are other forms of compile-time garbage collection. Escape analysis [3, 18, 8, 22] identifies when objects reachable only from a single thread's stack become free. Region inference [20, 14] (essentially) identifies points at which all objects allocated within a given stack frame become free, and deallocates them *en masse*. The analysis we propose addresses orthogonal issues.

Shape analysis [19] is clearly related to this work, though it uses different methods. These methods have more in common with traditional compiler analyses than the kind of abstract interpretation/program verification techniques we

have advocated here. The tradeoff is more well-understood worst-case analysis times for shape analysis, but perhaps at the cost of less generality in the kinds of invariants that can be inferred; this remains to be tested. The Role Analysis work of Kuncak *et al.* [15] is also related. Here the programming language is extended to allow the programmer to embed certain invariants as part of types, which are verified as part of type-checking. This system probably enables expression of the invariants necessary to enable the explicit deallocation optimization, but requires them to be explicitly expressed in a new language rather than inferred in an existing language.

The other side of this spectrum are automatic program verification systems, such as ESC [10, 13] and Prefix [7]. These sorts of tools have not usually been used to verify, much less infer, invariants as complicated as the ones we discuss. (Though the Houdini assistant of ESC [12] makes some efforts in the inference direction.) Our introduction of `owner` and `rc` as automatically updated state variables is a contribution that might be useful in the verification framework, as well. Bourdoncle [4] also used abstract interpretation techniques to infer program invariants, though again invariants considerably less elaborate than the ones we're contemplating here.

Linear types [21] are essentially a method for enforcing `rc[x] = 1` properties via the type system. The Vault system [11] allows a mixing of linear and nonlinear types, with a controlled dynamic "focus" operation allowing an object of nonlinear type to be considered temporarily linear, with associated type guarantees ensuring that no possible aliases can be used to modify the object in the region of focus. When linear type restrictions do not make the programming of a data structure unnatural, it is probably simpler to express such properties in a type system, rather than via the relatively complicated invariants we have shown in this work. However, the invariants are also clearly more general: they need not apply only to `rc[x] = 1` properties, as the doubly-linked list example suggests. Also, invariant annotations largely allow existing languages to be used unchanged except for the annotations.

The `owner` state variable concept obviously owes much to the concept of ownership types [6, 5]. As discussed previously, having the ownership map as an updatable state component may simplify verification of some programs: transferring ownership of a node from one object to another seems to become simpler, since it does not require a change in the type of the object. Another difference in our contribution is noting the interaction of ownership and program verification. Invariants phrased as quantification over all objects reachable from a given object are notoriously difficult to reason about mechanically (see [17]). I have some hope that invariants phrased as quantification over all objects owned by a given object may be considerably more tractable.

9. CONCLUSION

Programs are coming to have more and more live data in the heaps. Usually, much of this data is long-lived and rarely-modified. There exist many good techniques for efficiently collecting short-lived data, but collecting long-lived data that becomes garbage in a timely manner, without requiring full tracing traversals of very large live data sets, remains a challenge. As discussed above, techniques such as reference counting, whose cost is proportional to mutation rates, have undergone a resurgence of interest. This paper suggests that static analysis techniques may make it possible to get some of the benefits of reference counting without the runtime overhead.

10. TRADEMARKS

Java is a trademark or registered trademark of Sun Microsystems, Inc., in the United States and other countries.

11. ACKNOWLEDGMENTS

I would like to thank the anonymous reviewers for their helpful comments. This paper is in some measure a plea to find out what else I need to read to contribute in this area, and the reviewers helped greatly in that regard.

12. REFERENCES

[1] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the coffee breaks: A non-intrusive multiprocessor garbage collector. Submitted for publication; available at http://www.research.ibm.com/people/d/dfb/papers.html, 2000.

[2] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior referene counting: Fast garbage collection without a long wait. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[3] Bruno Blanchet. Escape analysis for object oriented languages. application to Java™. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 20–34, 1999.

[4] F. Bourdoncle. Assertion-based debugging of imperative programs by abstract interpretation. In Ian Sommerville and Manfred Paul, editors, *Software Engineering—ESEC '93*, volume 717 of *Lecture Notes in Computer Science*, pages 501–516. Springer-Verlag, 1993.

[5] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: preventing data races and deadlocks. *ACM SIGPLAN Notices*, 37(11):211–230, November 2002.

[6] Chandrasekhar Boyapati, Barbara Liskov, and Liuba Shrira. Ownership types for object encapsulation. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL-03)*, volume 38, 1 of *ACM SIGPLAN Notices*, pages 213–223, New York, January 15–17 2003. ACM Press.

[7] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Software Practice and Experience*, 30(7):775–802, June 2000.

[8] Jong-Deok Choi, M. Gupta, Maurice Serrano, Vugranam C. Sreedhar, and Sam Midkiff. Escape analysis for Java. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 1–19, Denver, CO, October 1999. ACM Press.

[9] David L Detlefs, K. Rustan M. Leino, and Greg. Nelson. Wrestling with rep exposure. Technical Report SRC-RR-156, Hewlett Packard Laboratories, July 29 1998.

[10] David L. Detlefs, K. Rustan M. Leino, James B. Saxe, and Greg Nelson. Extended static checking. Technical Report SRC Research Report 159, Compaq Systems Research Center, December 1998.

[11] Manuel Fahndrich and Robert DeLine. Adoption and focus: practical linear types for imperative programming. In Cindy Norris and Jr. James B. Fenwick, editors, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI-02)*, volume 37, 5 of *ACM SIGPLAN Notices*, pages 13–24, New York, June 17–19 2002. ACM Press.

[12] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/java. Technical Report SRC-TN-2000-003, Hewlett Packard Laboratories, December 20 2000.

[13] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. *ACM SIGPLAN Notices*, 37(5):234–245, May 2002.

[14] Niels Hallenberg, Martin Elsman, and Mads Tofte. Combining region inference and garbage collection. In *Proceedings of SIGPLAN 2002 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 141–152, Berlin, June 2002. ACM Press.

[15] Viktor Kuncak, Patrick Lam, and Martin Rinard. Role analysis. *ACM SIGPLAN Notices*, 37(1):17–32, January 2002.

[16] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[17] Greg Nelson. Verifying reachability invariants of linked structures. In *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, pages 38–47. ACM, Jan 1983.

[18] Young Gil Park and Benjamin Goldberg. Reference escape analysis: Optimizing reference counting based on the lifetime of references. In *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 26, 9 of *ACM SIGPLAN NOTICES*, pages 178–189, New York, June 17–19 1991. ACMPress.

[19] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[20] Mads Tofte and Lars Birkedal. A region inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):724–767, July 1998.

[21] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods, Sea of Galilee, Israel*, pages 347–359. North Holland, April 1990.

[22] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *OOPSLA'99 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 34(10) of *ACM SIGPLAN Notices*, pages 187–206, Denver, CO, October 1999. ACM Press.