

# Uniqueness Inference for Compile-Time Object Deallocation \*

Sigmund Cherem and Radu Rugina

Computer Science Department  
Cornell University  
Ithaca, NY 14853

{siggi, rugina}@cs.cornell.edu

## Abstract

This paper presents an analysis and transformation for individual object reclamation in Java programs. First, we propose a uniqueness inference algorithm that identifies variables and object fields that hold unique references. The algorithm performs an intraprocedural analysis of each method, and then builds and solves a set of inter-procedural uniqueness dependencies to find the global solution. Second, our system uses the uniqueness information to automatically instrument programs with explicit deallocation of individual objects. A key feature of the transformation is its ability to deallocate entire heap structures, including recursive structures, when their root objects are freed. This is achieved by generating object destructors that recursively free all of the unique object fields. Our experiments show that the analysis is effective at reclaiming a large fraction of the objects at a low analysis cost.

**Categories and Subject Descriptors** D.3.4 [Processors]: Compilers, Memory management; F.3.2 [Semantics of Programming Languages]: Program Analysis

**General Terms** Languages, Performance.

**Keywords** Uniqueness inference, compile-time memory management, individual object deallocation, program transformations.

## 1. Introduction

Compile-time memory management approaches use static analysis to conservatively estimate object lifetimes and reclaim objects as soon as they are no longer needed. Existing compile-time memory management techniques include: *stack allocation* [21, 9], which places objects on the run-time stack so that they are automatically reclaimed as methods return; *region inference* [18, 4, 7] which groups objects into regions and deallocates regions all at once; and *individual object reclamation* [17, 5, 10], which instruments programs with free statements to reclaim single object instances. This paper is concerned with the last approach. Compared to stack or region allocation, individual object deallocation has several appealing properties. First, it requires a standard malloc-free runtime support. No other runtime mechanisms, such as region allocation

or manipulating region handles, needs to be supported. Second, it requires less changes to programs compared to passing region handles across methods. Finally, it is more flexible than region and stack allocation, because it allows objects allocated at the same site to have different lifetimes and be collected at different points. However, freeing an entire heap structure when its root becomes unreachable has been a challenging problem for previous individual object deallocation approaches.

In this paper we use the notion of *uniqueness* to enable the deallocation of single objects in Java programs. A reference is unique if it is unaliased, i.e., no other memory location points to the same object. When a unique reference is overwritten or becomes dead, the target object can be freed.

This paper makes two contributions. First, it proposes a novel *uniqueness inference* algorithm that computes variable and field uniqueness in a flow-sensitive fashion, in the presence of temporary stack and heap sharing. The analysis combines a local analysis of each method with a lightweight global uniqueness constraint system. Second, the paper shows how to use the uniqueness information in a *free transformation*. In particular, the analysis uses field uniqueness information to generate *destructor methods* that recursively traverse and deallocate entire heap structures at once. We give a brief overview of each of these contributions below.

### 1.1 Uniqueness Analysis

Our analysis computes unique information for variables and fields. We use a relaxed notion of uniqueness that doesn't require variable of fields to be unique throughout their lifetime. More precisely, uniqueness is *flow-sensitive*, i.e., variables and fields may be unique at some points but not at others. Furthermore, uniqueness can be *recovered* in spite of temporary stack or heap sharing. In particular, *unique fields* can be temporarily shared with local variables or even other fields inside a method; however, they must be truly unique at method boundaries (method calls, entry, and exit points). We sometimes use the term "weakly unique field" to emphasize the fact a unique field may be temporarily shared.

Both flow-sensitivity and uniqueness recovery are achieved using a must-alias dataflow analysis that precisely tracks sharing of fields and variables at each point. In addition, temporary stack sharing is also permitted for the duration of method calls, when unique references are passed to non-escaping parameters. The necessary escape information is provided by an existing escape analysis [6].

The uniqueness inference algorithm that we propose in this paper consists of two steps:

- A *local must-alias dataflow analysis*. Each method is analyzed exactly once. The analysis computes must-alias sets at each program point. The analysis results are *parameterized* on the (unknown) uniqueness of parameters and fields at method entry.

\* This work was supported in part by NSF grant CNS-0406345.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'07, October 21–22, 2007, Montréal, Québec, Canada.  
Copyright © 2007 ACM 978-1-59593-893-0/07/0010...\$5.00.

- *A global constraint-based uniqueness propagation.* The analysis uses the parameterized dataflow results to derive constraints of the form  $f_1 \rightarrow f_2$ , showing that field or parameter  $f_2$  is unique only if  $f_1$  is. The analysis then performs a simple traversal of the constraint graph to propagate sharing information.

Once the global constraints are solved, the analysis can determine, for each program point  $p$  and each variable or field  $x$ , whether  $x$  is truly unique at  $p$ . Because the global constraint resolution component is very lightweight, the analysis is mostly modular.

## 1.2 Free Transformation

After computing the uniqueness information, the compiler inserts free statements to deallocate single objects whenever unique field references are overwritten; or when unique variables die. The analysis ensures that field deallocation is performed only at points where fields are truly unique.

The uniqueness analysis also provides a general solution to the structure deallocation problem. Our solution is based on destructor methods. The destructor of an object is called right before the object is freed, and is responsible for deallocating all of the object's unique fields. If the fields themselves have destructors, those will be recursively called. Hence destructors automatically “walk” the entire structure and deallocate all objects that are transitively reachable through unique references. The recursive nature of destructors makes them capable of deallocating recursive data structures, too.

We have implemented the proposed analysis and transformation in Soot [19] and applied it to the SPECjvm98 benchmarks. The compiler automatically instruments programs with free statements and destructor methods. Transformed programs are executed in a modified version of the Jikes RVM [2] that supports explicit object deallocation. On average, the uniqueness analysis takes about 7 seconds per benchmark and the instrumented programs free about 60% of the total memory. Variable uniqueness accounts for reclaiming 40% of the memory and field uniqueness for the remaining 20%. The proposed analysis is both efficient enough to be practical, and precise enough to reclaim a substantial amount of memory.

The rest of the paper is organized as follows. Section 2 presents several examples. Section 3 presents the uniqueness analysis and Section 4 presents the free transformation. Next, Section 5 presents experimental results. Finally, we discuss related work in Section 6 and conclude in Section 7.

## 2. Examples

This section presents several examples that illustrate the features of the uniqueness analysis and transformation.

### 2.1 Unique Fields

Figures 1 and 2 show code fragments from *jess*, one of the SPECjvm98 benchmarks. The code in Figure 1 shows a class `Token` containing a set of “facts” stored in the array field `facts`. The constructor at lines 4-7 builds a new token with one initial fact, and method `AddFact` at lines 8-17 adds one fact to the token, resizing the array if necessary. The code in italics is inserted automatically by our compiler.

Our analysis determines that the field `facts` in class `Token` is unique, i.e., it holds the only reference to the array object it points to. At lines 5 and 14 this field is assigned fresh, unaliased new arrays. Note that after the assignment `facts = vv` at line 14, `facts` is unique because `vv` is dead and is no longer an alias of `facts`. The other accesses to `facts` at line 6 and 16 (as well as in other parts of the program) read the array contents, but do not create aliases of this field. Hence, they do not affect its uniqueness.

Knowing that `facts` is unique, the compiler performs the following transformations:

```

1: class Token {
2:     ValueVector[] facts; /* unique */
3:     int size = 0;

4:     Token(ValueVector firstFact) {
5:         facts = new ValueVector[5];
6:         facts[size++] = firstFact;
7:     }

8:     void AddFact(ValueVector fact) {
9:         if (size >= facts.length) {
10:            ValueVector[] vv;
11:            vv = new ValueVector[size+3];
12:            System.arraycopy(facts,0,vv,0,size);
13:            free(facts);
14:            facts = vv;
15:        }
16:        facts[size++] = fact;
17:    }

18:     /* Destructor that frees unique fields */
19:     void destroy() {
20:         free(facts);
21:     }
22: }

```

**Figure 1.** Example 1: class `Token`. The code in italics is inserted by our compiler. The destructor `destroy()` frees the array field when the current object is freed.

```

23: void runTestsVaryRight(Token lt) {
24:     for (int i=0; i<right.size(); i++) {
25:         Token rt = right.elementAt(i);
26:         Token nt = appendToken(lt,rt);
27:         if (runTests(lt, rt, nt)) {
28:             for (int j=0; j<succ.length; j++) {
29:                 Successor s = succ[j];
30:                 s.node.CallNode(nt, s.callType);
31:             }
32:         }
33:         else if (nt != null) {
34:             nt.destroy();
35:             free(nt);
36:         }
37:     }
38: }

```

**Figure 2.** Example 2: a method that uses `Token` objects. The code in italics is inserted by our compiler.

- **Field deallocation.** At line 14, the unique field `facts` is being updated. Hence the last reference to the array object is removed and the array can be freed. The compiler inserts a statement `free(facts)` before the assignment.
- **Destructors methods.** The compiler adds a destructor method `destroy()` to class `Token`. The destructor of a class frees all of unique fields of that class. If the unique fields themselves have destructors, their destructors are recursively called before deallocating the fields (this is not the case in this example). The destructor method `destroy` will be called each time a `Token` object is freed in the program.

```

1: Id lookup(String s, boolean s_unique) {
2:   Id i = table.get(s);
3:   if (i == null) {
4:     i = new Id(s);
5:     table.put(s, i);
6:   }
7:   else if (s_unique)
8:     free(s);
9:   return i;
10: }

```

Figure 3. Example 3: Calling-context-sensitive deallocation.

## 2.2 Flow-Sensitive Variable Uniqueness

Figure 2 shows a method `runTestsVaryRight` that manipulates `Token` objects. The method uses several token variables: `lt`, `rt`, and `nt`. Of particular interest is token `nt`, which is assigned at line 26 the return value of `appendToken(lt,rt)`. Method `appendToken` is an *allocator method* that always returns fresh objects. Therefore, `nt` is unique right after the assignment.

Next, `nt` is passed to the call `runTests(lt,rt,nt)` at line 27. During this call, `nt` is temporarily shared with the parameter of `runTests` and possibly with other local variables of that method. However, `runTests` doesn't *escape* its last parameter, i.e., doesn't store this reference into memory locations that outlive the call. Hence, `nt` is again unique after the call. On the true branch of the if statement, `nt` is passed to `CallNode` (line 30). This method, however, escapes its first parameter, placing this reference into a long-lived heap field. Hence, `nt` is no longer unique at the end of the true branch. On the false branch, though, `nt` remains unique and also becomes dead. The compiler automatically inserts a statement `free(nt)` at line 35 to deallocate the object, but only after calling its destructor `nt.destroy()` that will free its unique fields.

Note that the uniqueness of variable `nt` is a flow-sensitive property: `nt` is unique at lines 27 and 34, but not at lines 31 or 37. The compiler identifies these facts using a dataflow analysis that tracks the aliasing state of `nt` through the program, identifying the points where `nt` is unique. Escape information and information about allocator methods is provided by an escape analysis that we have developed in prior work [6].

## 2.3 Calling-Context Uniqueness

In addition to path-dependence, uniqueness may also depend on the calling context of the current method. Distinguishing between calling contexts where arguments are unique or shared can provide additional opportunities for memory reclamation.

Consider the example in Figure 3 taken from the *javac* benchmark. The code uses an if statement: on the true branch, the object passed through parameter `s` is placed into a long-lived table via a put operation; on the false branch, the object is not used. The ability to deallocate `s` depends on the context in which method `lookup` is called: the object can be safely deallocated on the false branch only when the method is passed a unique reference into this parameter. Note that callers of `lookup` cannot deallocate this object because they would not know which branch is taken inside `lookup`.

Our compiler identifies such a situation and augments the method signature with an boolean flag indicating whether `s` is unique when calling this method. The compiler automatically augments all calls to `lookup` with the appropriate value at each call site. Finally, freeing `s` is performed only after testing this flag.

## 2.4 Weakly-Unique Fields and Temporary Heap Sharing

Certain data structures, such as linked lists or trees, have fields that are unique most of the time, except during destructive operations

```

1: class Elem {
2:   Elem next; /* unique */
3:   Object data;
4:
5:   void destroy() {
6:     if (next != null) {
7:       next.destroy();
8:       free(next);
9:     }
10:  }
11: }

```

```

12: class List {
13:   Elem head; /* unique */
14:
15:   void add(Object o) {
16:     Elem e = new Elem();
17:     e.data = o;
18:     e.next = head;
19:     head = e;
20:   }
21:
22:   void reverse() {
23:     Elem x = head;
24:     Elem c,p = null;
25:     while (x != null) {
26:       c = x.next;
27:       x.next = p;
28:       p = x;
29:       x = c;
30:     }
31:     head = p;
32:   }
33:
34:   void destroy() {
35:     if (head != null) {
36:       head.destroy();
37:       free(head);
38:     }
39:   }
40: }

```

Figure 4. Example 4: a linked list example. Destructors recursively deallocate the entire list.

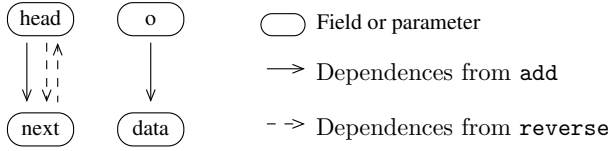
such as insertion or removal operations when these fields might be temporarily shared with other variables or even other heap fields. Our analysis allows unique fields to be temporarily shared, but requires them to be unique at method boundaries.

Figure 4 shows a linked list example. Class `Elem` represents list elements having fields `data` and `next`. Class `List` points to the list head element via field `head`. For simplicity, we consider just two methods for `List`: method `add`, which inserts an element at the beginning of the list; and `reverse`, which reverses the list. In this example, both `head` and `next` are weakly-unique: they are unique at all points except during `add` and `reverse`.

To identify weakly-unique fields, the compiler performs an intra-procedural dataflow analysis of each method. Figure 5 shows the analysis of method `add`; the analysis for `reverse` is similar, but requires a fixpoint computation for the analysis of the loop. The analysis result at each point is shown in the right part of Figure 5 and consists of several must-alias sets at each program point. The expressions in each set point to the same object instance; furthermore, they represent *all* references to that object. Each set can be thought of as an abstraction of an object instance – the object

<pre> 1: void add(Object o) { 2:   Elem e = new Elem(); 3:   e.data = o; 4:   e.next = this.head; 5:   this.head = e; 6:   //all locals are dead 7: }</pre>	<pre> {e}⊥ {e}⊥ {e}⊥ {this.head, e}⊥ {* .head}⊥</pre>	<pre> {* .head}head {* .head}head {* .head}head {e.next, this.head}head {e.next}head {* .next}head</pre>	<pre> {* .head}head {this.head}head {this.head}head {* .head}head</pre>	<pre> {o}° {o}° {e.data, o}° {e.data, o}° {e.data, o}° {* .data}°</pre>
(a)		(b)	(c)	(d)

**Figure 5.** Analysis of method `add`. The computed alias sets are shown at each line. Superscripts represent conditions regarding the uniqueness of field `head` and parameter `o` at method entry. The columns show alias sets for different objects: (a) for the new object; (b) for the head of the current list; (c) for the head of another list; (d) for the parameter object.



**Figure 6.** Uniqueness dependences for `add` and `reverse`.

that the set points to. When a set contains a single reference, that reference is unique.

The elements in each set are either variables ( $x$ ) or field expressions, which include variable-field expressions ( $x.f$ , the field  $f$  of variable  $x$ ), anonymous-field expressions ( $*.f$ , the field  $f$  of an arbitrary object), and negated-field expressions ( $\bar{x}.f$ , i.e., the field  $f$  of an object other than  $x$ ). For instance, the set  $\{e.next, this.head\}$  at line 4 shows that `e.next` and `this.head` point to the same object instance, and there are no other references to that object. Similarly,  $\{* .head\}$  at line 1 models an arbitrary object that is pointed to only by `head`.

The computed dataflow facts are parameterized by the uniqueness of parameter `o` and field `head` at the beginning of the method. The analysis tags each dataflow fact with a uniqueness condition, shown using superscripts. For instance, the tagged set  $\{e.next, this.head\}^{head}$  indicates that `e.next` and `this.head` are the only aliases to their target, but only if field `head` was unique at the beginning of the method.

The analysis initializes the dataflow facts at the beginning of the method with an alias set for each parameter and each field that the method explicitly loads. Each set is tagged with the appropriate condition:  $\{* .head\}^{head}$  represents an arbitrary head element, and  $\{o\}^\circ$  represents the parameter. The analysis then computes how these sets change in the method body. The analysis also builds new alias sets at allocation sites, such as `e = new Elem()` at line 2, where the alias set  $\{e\}^\perp$  is created. The superscript  $\perp$  indicates that the alias set does not depend on the uniqueness of any parameter or field. At line 4, when the program loads from `this.head`, we perform a case analysis, splitting the imprecise set  $\{* .head\}^{head}$  into  $\{this.head\}^{head}$  (the head of the current list) and  $\{\bar{this}.head\}^{head}$  (the head of some other list). After that, precise alias sets can be computed for each of the two cases.

Finally, the analysis inspects the sets at the end of the function. If a set containing field  $f_1$  is tagged with another field  $f_2$ , the analysis derives a uniqueness dependence  $f_2 \rightarrow f_1$  showing that the uniqueness of  $f_1$  depends on that of  $f_2$ . In other words, if  $f_2$  is shared, then so is  $f_1$ . Figure 6 shows the uniqueness constraints for this example. The analysis generates two constraints: `head`  $\rightarrow$  `next` due to the set  $\{* .next\}^{head}$  at line 6; and `o`  $\rightarrow$  `data` due to

$\{* .data\}^\circ$ . The analysis of `reverse` (not shown here due to lack of space) will indicate that `head` is shared if and only if `next` is.

Assuming that no other methods access `head` or `next`, the analysis concludes that these are unique everywhere except the bodies of `add` and `reverse`. The uniqueness of `data` depends on whether the calls to `add` pass unique references for `o`: if there exists a call site where `o` is passed a shared reference, then node `o` will be marked shared, and the uniqueness dependence `o`  $\rightarrow$  `data` will make the field `data` shared, too.

## 2.5 Recursive Destructors and Recursive Deallocation

Once the compiler identifies that the fields `head` and `next` are weakly unique, the compiler inserts a destructor in each class, as shown in Figure 4. The destructor of `List` (lines 34-39) destructs and frees the head element; and the destructor of `Elem` (lines 5-10) destructs and frees the next list element. As a result, whenever a `List` object is freed, its destructor will “walk” the entire list and recursively free all of the list elements. If the `data` field is determined to be unique, then it can also be freed in the `List` destructor. Our transformation ensures that destructors are never called in contexts where weakly unique fields are temporarily shared.

## 3. Uniqueness Analysis

This section describes the uniqueness analysis algorithm in detail.

### 3.1 Language

We assume a simple program model where each method is represented using a flow graph whose nodes are assignments of the form:

$$loc = exp$$

The left and right hand side expressions have the following syntax:

$$\begin{aligned}
loc &::= x \mid x.f \\
exp &::= loc \mid \text{null} \mid \text{new } C \mid m(y_0, \dots, y_n)
\end{aligned}$$

Here,  $x \in \mathcal{V}$  ranges over variables,  $f \in \mathcal{F}$  over fields,  $C$  over classes, and  $m \in \mathcal{M}$  over methods. Variables include method parameters  $\mathcal{V}_P \subset \mathcal{V}$ , denoted  $p_0, \dots, p_n$ . For virtual methods,  $p_0$  is the reference to the receiver object. Locations are either variables or instance field accesses  $x.f$ . Arrays are modeled as a special field “[ ]”, and all static fields are modeled with a special global variable  $g$ . To simplify the presentation, we assume that all expressions have reference types, and that methods always return a value. Other Java constructs are modeled in the standard way: a throw statement “throw  $x$ ” is modeled just like an assignment of  $x$  to a static field; catching an exception “catch (Exception  $x$ )” is represented as loading the exception from a static field to a local variable; and a return statement “return  $x$ ” is modeled as an assignment to a special return variable: “ret =  $x$ ”. Constructors are treated the same as all other methods; they are called right after the object is allocated.

### 3.2 Uniqueness Abstraction

The uniqueness analysis uses a two-level memory abstraction: a global flow-insensitive abstraction, and a local flow-sensitive abstraction. The combination of these abstractions allows us to describe precise uniqueness information.

**Global Abstraction** The global abstraction consists of two components: a set  $U$  describing unique references, and a relation  $E$  describing uniqueness restrictions.

- The set  $U \subseteq \mathcal{V}_P \cup \mathcal{F}$  holds parameters and fields that are weakly unique. Fields and parameters not in  $U$  are conservatively assumed to be shared at all program points. Fields and parameters in  $U$  must be unique at the entry and exit of methods that access them. Fields in  $U$  may be shared at the boundaries of methods that don't access them; those methods are specified by the relation  $E$  below. Thus, temporary sharing of elements in  $U$  is expressed by the local abstraction and by the relation  $E$ .
- The relation  $E \subseteq \mathcal{M} \times U$  matches methods and unique fields. A pair  $(m, f) \in E$  indicates that  $f$  is not accessed (read or written) in  $m$ , but might be shared at the boundaries  $m$ , in spite of being labeled unique ( $f \in U$ ). Hence,  $E$  can be thought of modeling "exceptions" to the uniqueness property.

**Local Abstraction** For each method, the local flow-sensitive abstraction consists of a set of tagged sets  $A$  per program point. Every tagged set  $s^t \in A$  contains expressions of the form:

$$e ::= x \mid x.f \mid \bar{x}.f \mid *.f$$

where  $x \in \mathcal{V}$  are variables,  $x.f$  are field expressions that occur in load and store statements of the method,  $\bar{x}.f$  represents an expression ending in field  $f$ , whose source is not  $x$ ; and,  $*.f$  is a single field expression whose source is unknown. Each set  $s$  represents an object instance and the expressions in the set represent references to that object. The tag  $t$  gives additional meaning to this set. The possible tags  $t$  on a set are:

$$t ::= \perp \mid f \mid p_i \mid \top$$

A tag  $\perp$  means that  $s^\perp$  contains *all* references to the object it represents. In particular, when a set  $s^\perp$  contains one single expression, that expression holds a unique reference to the object. The tag  $f$  says that  $s^f$  contains all references to the object *only if* the field  $f$  was unique at the entry of the method and at all method calls. Similarly,  $s^{p_i}$  contains all references to the object if  $p_i$  was unique at the method's entry and all method calls. A tag  $\top$  means that not all references to the object are known, hence the object may contain additional references.

**Combined Abstraction** The combination of these abstractions can express when a field, variable or parameter is in fact truly unique. Consider a program point  $p$  in a method  $m$ , an let  $A$  be the local abstraction at that program point. Given a variable or parameter  $x$ , we can say that  $x$  is unique at  $p$  when the object pointed by  $x$  has no additional references:

$$\begin{aligned} \text{uniqueVar}(x, p) &= \forall s^t \in A. \\ x \in s &\Rightarrow (s = \{x\}) \wedge t \in U \cup \{\perp\} \end{aligned}$$

This is when all sets  $s^t$  in the local abstraction that mention  $x$  contain a single expression ( $s = \{x\}$ ) and the tag indicates there are no additional references to it ( $t \in U \cup \{\perp\}$ ).

Similarly, a field  $f$  is unique at a program point  $p$ , if an object  $o$  pointed by the field  $f$  at that point has no additional references:

$$\begin{aligned} \text{uniqueField}(f, p) &= (m, f) \notin E \wedge \\ &[\forall s^t \in A, z \in (\mathcal{V} \cup \{*\}) . \\ & z.f \in s \Rightarrow (s = \{z.f\}) \wedge t \in U \cup \{\perp\}] \end{aligned}$$

The abstraction  $A_0$  at the entry of a method is defined by:

$$\begin{aligned} A_0 &= \{ \{p_i\}^{p_i} \mid p_i \text{ is an argument} \} \\ &\cup \{ \{*.f\}^f \mid f \text{ appears on some load stmt} \} \\ &\cup \{ \{g\}^\top \} \end{aligned}$$

**Figure 7.** Analysis Initialization: includes a set for each external reference used in the method.

Besides possible references shown in the local abstraction  $A$ , we check that the location has no references from callers of the current method by ensuring  $(m, f) \notin E$ .

### 3.3 Uniqueness Dataflow Analysis

The goal of the dataflow analysis is to compute the local abstraction at each program point. Partial information computed by this dataflow analysis will be used later in Section 3.4 to construct the global abstraction.

Even though our formalisms are presented at the level of an entire abstraction  $A$ , our analysis operates at the level of individual sets and tracks each set independently from the others.

Figure 7 presents the initial abstraction  $A_0$  at the entry of a method. The abstraction includes a set for each external reference read in the method (i.e., parameters and fields loaded). An initial set  $\{*.f\}^f$  describes objects pointed by a reference from field  $f$ . If  $f$  is unique, this set represents in fact all references to the object.

At each statement, the analysis determines how the abstraction changes using set transformers. Figure 8 presents the analysis transformer function for assignments.

Consider the abstraction  $A$  before an assignment  $e_1 = e_2$ . For each set  $s^t$  in  $A$ , the analysis first makes sure that it knows if  $e_1$  and  $e_2$  reference the object described by  $s^t$ . We use the relation  $F$  to make this information precise. The analysis then applies a set transformer  $T$  to the resulting sets. The set transformer updates the information in these sets to incorporate the effects of the assignment. Finally, we update the tag information using  $I$ , and we introduce new sets according to the specification in  $G$ . To simplify the presentation, we omit the tag annotation whenever it is unchanged by the set transformer.

To illustrate how the analysis works, let's consider several kinds of assignments:

- Variable nullification  $x = \text{null}$ . The analysis removes  $x$  from its set, and adjusts each set that contains an expression  $x.f$  (or  $\bar{x}.f$ ), for any field  $f$ . It would be unsafe to just remove  $x.f$  from those sets. To keep the algorithm simple, the analysis doesn't attempt to reason about  $x$ 's aliases, and instead it conservatively represents this reference using  $*.f$ .
- Allocation  $x = \text{new } C$ . The analysis first nullifies  $x$ , and generates a new set  $\{x\}$  to represent the new object.
- Copy  $x = y$ . After nullifying  $x$ , the analysis adds it to the set that contains  $y$ .
- Field nullify  $y.f = \text{null}$ . The analysis simply removes  $y.f$  from the sets where it appears. Note that nullifying  $y.f$  might also nullify another expression  $z.f$  if  $y$  and  $z$  are aliased. For this reason we introduce the case analysis described with the function  $F_{y.f}$ . Consider when  $s = \{z.f\}$ , the operation  $F_{y.f}$  will transform  $s$  into two sets:  $\{y.f\}$  and  $\{\bar{y}.f\}$ . Finally the set transformer  $T$  produces two sets,  $\{\}$  and  $\{\bar{y}.f\}$ , as a result.
- Field store  $y.f = x$ . The analysis nullifies  $y.f$  and then adds it to the set that contains  $x$ .
- Field load  $x = y.f$ . The analysis transforms sets at loads by moving  $x$  to the sets referenced by  $y.f$ .

For an assignment  $e_1 = e_2$  the transfer function is

$$\llbracket e_1 = e_2 \rrbracket A = (I \circ T \circ F_{e_2} \circ F_{e_1}(A)) \cup G$$

where:

$F_e$  is a relation that performs a case analysis on the expressions  $e$ :

$$\begin{aligned} F_{x.f} &= \{(s, s) \mid x.f \in s \vee \bar{x}.f \in s\} \\ &\cup \{(s, s - \{z.f\} \cup \{x.f\}) \mid z.f \in s \wedge x.f \notin s\} \\ &\cup \{(s, s - \{z.f\} \cup \{\bar{x}.f\}) \mid z.f \in s \wedge x.f \notin s\} \\ &\quad \text{where } z \in (\mathcal{V} \cup \{*\}) \\ F_e &= \{(s, s) \mid \text{for any } s\} \text{ when } e = x \text{ or } e \notin \text{loc} \end{aligned}$$

$T$  is a relation that transforms sets. For any field  $f \in \mathcal{F}$ :

$$\begin{aligned} T &= \{(s, s - \{e_1 \mid e_1 \neq g\} \\ &\quad - \{x.f, \bar{x}.f \mid e_1 \equiv x\} \\ &\quad \cup \{e_1 \mid e_2 \in s\} \\ &\quad \cup \{*.f \mid (x.f \in s \vee \bar{x}.f \in s) \wedge e_1 \equiv x\})\} \end{aligned}$$

$I$  updates the tag information:

$$\begin{aligned} I &= \{(s^t, s^\top) \mid g \in s\} \\ &\cup \{(s^t, s^\top) \mid x.\square \in s\} \\ &\cup \{(s^t, s^\top) \mid x.f \in s \wedge z.f \in s \wedge x \neq z\} \\ &\cup \{(s^t, s^t) \mid \text{otherwise}\} \end{aligned}$$

$G$  are new sets generated by the assignment:

$$G = \{ \{e_1\}^\perp \mid e_2 \equiv \text{new} \}$$

**Figure 8.** Transfer function for assignments. We lift all transformer  $H$  to its power-domain as follows:  $H(A) = \{H(s) \mid s \in A\}$ . All transformers except  $I$  keep the original tag on each set.

The transformer  $I$  evaluates if some set  $s$  should be considered a shared object, for example if it was stored in a static field, in an array or too many fields point into it. In all such cases, the analysis updates the set tag to  $\top$ .

At any program point the abstraction contains no pair of similar sets. Two sets are similar if they have the same tag and the same set and number of variables and fields:

$$\begin{aligned} s_1^{t_1} \sim s_2^{t_2} &\Leftrightarrow t_1 = t_2 \wedge s_1 \cap \mathcal{V} = s_2 \cap \mathcal{V} \wedge \\ &\forall f. |\{x.f \mid x.f \in s_1\}| = |\{x.f \mid x.f \in s_2\}| \end{aligned}$$

The join operation of two similar sets maintains the original tag and merges the expressions as follows:

$$s_1 \sqcup s_2 = ((s_1 \cap s_2) \cup \{*.f \mid x.f \in s_1 \wedge y.f \in s_2\})$$

Finally at merge points, our analysis computes the following abstraction:

$$\begin{aligned} A_1 \sqcup A_2 &= \{s_1 \sqcup s_2 \mid s_1 \sim s_2 \wedge s_1 \in A_1 \wedge s_2 \in A_2\} \\ &\cup \{s_1 \mid s_1 \in A_1 \wedge \forall s_2 \in A_2. s_1 \approx s_2\} \\ &\cup \{s_2 \mid s_2 \in A_2 \wedge \forall s_1 \in A_1. s_1 \approx s_2\} \end{aligned}$$

Method calls are handled specially, using method summaries to describe the effects of the call. Figure 9 shows the rules to handle a method call  $x = m(y_0, \dots, y_n)$  in our analysis. The analysis uses five predicates to describe the effects of the call, an implementation of these predicates is presented later in Section 3.5.2:

1.  $fresh(m)$ : method  $m$  returns a fresh new object or null.
2.  $returned(m, p_i)$ : method  $m$  returns the same object referenced by the formal  $p_i$ .
3.  $stores(m, p_i)$ : method  $m$  may create external references to the object pointed by formal parameter  $p_i$ .
4.  $reads(m, f)$ : calls to method  $m$  may read field  $f$ .
5.  $retShared(m)$ : the return value of method  $m$  is shared.

The transfer function for a method call is:

$$\llbracket x = m(y_0, \dots, y_n) \rrbracket A = (I_m \circ T_m(A)) \cup G_m$$

where:

$$\begin{aligned} T_m &= \begin{cases} T_{x=y_i} & \text{returned}(m, p_i) \\ \{(s, s)\} & \text{otherwise} \end{cases} \\ I_m &= \{(s^t, s^\top) \mid y_i \in s \wedge stores(m, p_i)\} \\ &\cup \{(s^t, s^\top) \mid \exists z. z.f \in s \wedge |s| \geq 2 \wedge reads(m, f)\} \\ &\cup \{(s^t, s^t) \mid \text{otherwise}\} \\ G_m &= \{ \{x\}^\perp \mid fresh(m) \} \\ &\cup \{ \{x\}^\top \mid retShared(m) \} \end{aligned}$$

**Figure 9.** Transfer function for a method call. The predicates  $fresh(m)$ ,  $stores(m, p_i)$ ,  $reads(m, f)$ ,  $returned(m, p_i)$ , and  $retShared(m)$  summarize the effects of the call.

The transfer function for the method call behaves like an allocation site if the callee returns a fresh new object. Otherwise, it acts like an assignment  $x = y_i$  if the method returns the same object pointed by its formal argument  $p_i$ . Since method calls are points where the program leaves a method boundary, the transfer function sometimes updates the tag of objects that are accessed during the call. This includes objects pointed by an actual  $y_i$ , and the method is known to add references to  $p_i$ . Or similarly, the object is pointed by some reference via field  $f$ , which is used during the call.

### 3.4 Global Constraints

The global abstraction is generated from inspecting the results of the flow-sensitive analysis. The compiler deduces a set of constraints from the analysis results. From solving the constraints, we can compute the global abstraction.

**Weakly Unique References ( $U$ ).** Figure 10 presents the possible constraints generated by our analysis. The constraints are formulated in terms of the tags used to annotate the dataflow facts. Hence the constraints mention parameters, fields,  $\perp$  and  $\top$ . The tag  $\top$ , static references and array fields are assumed to be shared. The most precise solution to the constraints will always include  $\perp \in U^*$ . The set of weakly unique fields and parameters is  $U = U^* - \{\perp\}$ .

Since the analysis allows temporarily sharing within method boundaries, constraints are generated when leaving a method, either at the exit point or when calling other methods. Consider a set  $s_e^t$  in the abstraction at the exit point of a method. If  $s_e^t$  contains two or more elements, we must consider all its references as shared. If  $s_e^t$  contains a single reference, say  $x.f$ , we generate the constraint  $t \notin U^* \Rightarrow f \notin U^*$ . This constraint models three cases. First, if the set is tagged  $\perp$ , then it was derived from an allocation site and  $x.f$  is a unique reference. Second, if the set is tagged with a field  $h$ , then the field  $f$  is weakly unique only if  $h$  is also unique. Finally, if the set is tagged  $\top$ , then  $x.f$  is considered to be shared.

A similar, but more precise reasoning can be done at method calls. Unlike constraints at the exit point, the analysis doesn't make fields shared if the set  $s_e^t$  before a call represents an object that is not accessed during the call. Virtual calls are modeled by using an additional constraint: a parameter of a child method is shared when the same parameter is shared the parent method.

We solve these constraints with a simple reachability algorithm. We construct a graph where nodes are tags and edges represent constraints. Implication constraints  $a \notin U^* \Rightarrow b \notin U^*$  are represented by an edge  $a \rightarrow b$ . Sharing constraints  $a \notin U^*$  are

The set of weakly unique fields and parameters is  $U = U^* - \{\perp\}$ , where  $U^*$  is the maximal set that satisfies the following constraints:

Constraints for  $\top$ , arrays and static fields:

$$\top \notin U^*, \quad \square \notin U^*, \quad g \notin U^*$$

For each set  $s_e^t \in A$  at the end of each method:

$$\begin{aligned} p_i &\notin U^* && \text{if } |s_e| \geq 2 \wedge p_i \in s_e \\ f &\notin U^* && \text{if } |s_e| \geq 2 \wedge x.f \in s_e \\ t &\notin U^* \Rightarrow p_i \notin U^* && \text{if } s_e = \{p_i\} \\ t &\notin U^* \Rightarrow f \notin U^* && \text{if } s_e = \{x.f\} \end{aligned}$$

For each set  $s_c^t \in A$  before a call to  $m(y_0, \dots, y_n)$ :

$$\begin{aligned} p_i^m &\notin U^* && \text{if } |s_c| \geq 2 \wedge y_i \in s_c \\ f &\notin U^* && \text{if } |s_c| \geq 2 \wedge x.f \in s_c \wedge \text{reads}(m, f) \\ (m, f) &\in E && \text{if } |s_c| \geq 2 \wedge x.f \in s_c \wedge \neg \text{reads}(m, f) \\ t &\notin U^* \Rightarrow p_i^m \notin U^* && \text{if } s_c = \{y_i\} \\ t &\notin U^* \Rightarrow f \notin U^* && \text{if } s_c = \{x.f\} \wedge \text{reads}(m, f) \end{aligned}$$

Constraints modeling virtual calls:

$$p_i^p \notin U^* \Rightarrow p_i^c \notin U^* \quad c \text{ overrides } p$$

Constraints modeling uniqueness exceptions:

$$(m, f) \in E \Rightarrow (m', f) \in E \quad m \text{ calls } m' \text{ or } m' \text{ overrides } m$$

**Figure 10.** Global constraint system.

represented by a special edge  $\top \rightarrow a$ . Weakly unique fields and parameters are those not reachable from  $\top$ .

**Uniqueness Restrictions ( $E$ ).** Figure 10 also presents constraints for the uniqueness exceptions relation  $E$ . Consider an object  $o$  represented by  $s_c^t$  before a call to  $m$ . If  $o$  is pointed by a field  $f$  and by some additional references, but it is not accessed during the call, then we impose that  $(m, f) \in E$ . Moreover, if a method  $m$  satisfies  $(m, f) \in E$ , so does any method  $m'$  that is transitively invoked by  $m$  or that overrides  $m$ . Hence, if  $(m, f) \notin E$ , an object pointed by a field  $f$  within a method  $m$  cannot have additional references from callers of  $m$ . Based on the observation that  $\text{reads}(m, f)$  implies  $(m, f) \notin E$ , our implementation approximates  $E$  using  $\text{reads}$ .

### 3.5 Enhancements and Analysis Details

This section presents several enhancements and details of the algorithm presented so far.

#### 3.5.1 Variable Liveness

Live variable information can be used to improve the analysis: when a variable  $x$  becomes dead at a program point, the uniqueness analysis models this fact using a nullification  $x = \text{null}$ . Nullifications lead to smaller alias sets and allow the compiler to deallocate objects earlier.

Liveness information also helps identifying when uniqueness is being transferred between variables: if a unique variable  $x$  is copied into another variable  $y$ , and  $x$  becomes dead after this use, then the uniqueness of  $x$  is passed on to  $y$ . Passing unique references to methods is usually done in this manner. Thus, liveness information plays an important role in identifying unique method parameters.

#### 3.5.2 Using Escape Summaries

As discussed earlier, we use the method escape summaries developed in our previous work [6] to approximate the effects of method calls. We briefly describe the method summaries and discuss how they are used in the current uniqueness analysis.

For a given constant  $k$ , a  $k$ -level method summary assigns a sequence of  $k$  attributes to each parameter and to the return value.

The  $i$ -th attribute in the sequence models objects reachable through  $i$  levels from the parameter. Objects beyond the  $k$ -levels are marked as escaped using a  $\top$  value. Objects denoted with the same attribute may be aliases. We call this sequence of attributes the signature of a method. For example, the method:

```
void set (x) { this.f = x; }
```

has the following 2-level signature:

$$\text{set} : (\alpha_1, \alpha_2) \times (\alpha_2) \rightarrow \perp$$

Here,  $\alpha_1$  corresponds to the receiver object, and  $\alpha_2$  to any of the objects that is pointed by some field of the receiver. The signature indicates that the parameter  $x$  may alias a field of the receiver (they share the attribute  $\alpha_2$ ) and, hence, it indicates that the parameter may be stored in the receiver.

More generally, the signatures can also describe points-to relations, accessed fields and read/write effects. We have omitted these features to simplify our presentation.

The five predicates used by the analysis in Section 3.3 can be computed from the  $k$ -level summaries as follows:

- $\text{fresh}(m)$  is true when the return value is unaliased with any other reachable object. This holds if the return attribute at level 0 is not  $\top$  and doesn't occur elsewhere in the signature.
- $\text{returned}(m, p_i)$  is true when a single parameter is aliased with the return value. This holds if the attribute of  $p_i$  at level 0 is not  $\top$ , and the only other position in which it occurs is the return attribute at level 0.
- $\text{stores}(m, p_i)$  is true when the parameter could be stored in a field reference that outlives the lifetime of  $m$ . This may hold if the attribute of  $p_i$  at level 0 is either  $\top$  or occurs elsewhere in the signature. In the method `set` above, for example, the signature reveals that the parameter  $x$  may be stored since its attribute ( $\alpha_2$ ) appears also in the receiver attribute ( $\alpha_1, \alpha_2$ ).
- $\text{reads}(m, f)$  is true if  $f$  is accessed during a call to  $m$ . This can be answered when field information is available in the signature. In particular  $\text{reads}(m, f)$  is true if the field  $f$  appears in any points-to relation of the signature. When the signatures contains a  $\top$  value,  $\text{reads}(m, f)$  holds when  $f$  is listed the method's accessed fields.
- $\text{retShared}(m)$  is true if the return value of  $m$  is shared, i.e., when the return attribute at level 0 is either  $\top$ , it also appears at levels greater than 0, or it appears more than twice.

#### 3.5.3 Open World Semantics for Library Analysis

To be able to analyze library classes separately, once for all benchmarks, we use a *open world* semantics where the analysis conservatively assumes that method parameters and public fields are shared. Hence, in library code, uniqueness properties can be derived only for private fields and local variables.

#### 3.5.4 Java Features

The compiler uses a conservative treatment for exceptions, multi-threading, and calls by reflection. All of these constructs have complex control-flow that the uniqueness analysis cannot easily reason about. All objects that are thrown, passed to child threads, or passed as arguments to calls by reflection are conservatively marked as shared by the uniqueness analysis and the transformation will not attempt to free them.

## 4. Transformation

This section presents how to use the results of the uniqueness analysis to enable the deallocation of objects. The compiler uses

the predicates  $uniqueVar(x, p)$  and  $uniqueField(f, p)$  presented in Section 3.2 to free objects referenced by unique variables and unique fields. We denote by  $\bullet stmt$  and  $stmt\bullet$  the program points before and after a statement, respectively.

#### 4.1 Free on Unique Variables

The compiler inserts a `free(x)` instruction in the program right after the last use of a unique variable  $x$ . In other words, the `free` is inserted when a unique variable dies. There are two situations when this can happen:

- *A unique variable dies after a statement  $stmt$ .* The compiler inserts `free(x)` after the statement when  $uniqueVar(x, \bullet stmt)$  holds at the point before the statement, and  $x$  becomes dead on  $stmt$ .

An exception to this rule is when the statement is a method call,  $x$  is passed as a parameter to the call and the callee expects a unique parameter in  $x$ 's position. In such case, the callee *captures* the unique reference and the free will be inserted later inside the callee.

- *A unique variable dies on an outgoing control edge.* If a unique variable dies on one, but not all outgoing branches, then it can be freed on the one branch where it dies. More precisely, if there is a control edge from statement  $stmt$  to statement  $stmt'$ , the compiler inserts a `free(x)` statement just before  $stmt'$  if:  $x$  is unique after  $stmt$ , i.e.  $uniqueVar(x, stmt\bullet)$ ;  $x$  is live after  $stmt$ ; and  $x$  is dead before  $stmt'$ .

#### 4.2 Free on Unique Fields

In the case of a store assignment  $x.f = e$ , the compiler inserts a `free(x.f)` instruction right before the store when it determines that  $f$  is truly unique at that point:  $uniqueField(f, \bullet(x.f = e))$ .

#### 4.3 Adding Destructors and Calls to Destructors

The analysis also takes advantage of field uniqueness information to enable the recursive deallocation of entire structures. For each class that has at least a unique field, the compiler adds a *destructor* method `destroy()` that frees each unique field.

Finally, for each `free(x)` instruction such that  $x$ 's class has a destructor method, the compiler introduces a call to the destructor  $x.destroy()$ . Such calls must be guarded by checks that ensure  $x$  is not null. Destructors can occur within destructors. This enables the automatic deallocation of entire structures, including the deallocation of recursive structures such as lists or trees.

The compiler can insert calls to  $x.destroy()$  only if the fields traversed by the destructor are truly unique at the point before the `free(x)` instruction. To check this, we compute a set  $D_f$  of fields accessed by each destructor method. The set  $D_f$  includes all fields directly freed by the destructor, but also those fields accessed by destructors called within the destructor itself. The call to  $x.destroy()$  is added only when all fields  $f \in D_f$  are truly unique, i.e.  $uniqueField(f, \bullet(free(x)))$ .

#### 4.4 Calling-Context Uniqueness Transformation

We extend our transformation to support calling-context uniqueness, as illustrated in the example from Section 2.3. Our compiler performs such a transformation via a restricted form of uniqueness polymorphism: we will perform different actions depending on the uniqueness of the method arguments. We define a predicate  $condUniqueVar(x, p)$  to formalize when a variable  $x$  is conditionally unique at program point  $p$ . We say  $x$  is conditionally unique if it is not truly unique, but it would be if all method parameters were unique:

$$condUniqueVar(x, p) = \neg uniqueVar(x, p) \wedge \forall s^t \in A. x \in s \Rightarrow (s = \{x\}) \wedge t = p_i \in \mathcal{V}_P - U \cup \{\perp\}$$

Whenever a dead variable  $x$  satisfies  $condUniqueVar(x, p)$  we introduce a conditional deallocation:

```
if (p1.unique && ... && pn.unique) free(x);
```

where  $p_1, \dots, p_n$  are the set of parameters on which  $x$ 's uniqueness depends on; and  $p_i.unique$  is a boolean argument added to the method signature.

Each call to an extended method must be updated to use the new method signature. We pass a `true` value for  $p_i.unique$  when the corresponding actual parameter  $y_i$  is both unique and dead at the call site. Otherwise, we pass a `false` parameter. Note this transformation cannot be combined with the open world semantics from Section 3.5.3. Thus, we don't apply this transformation to library code.

## 5. Results

We have implemented the analysis presented in this paper in a prototype compilation system for Java built using the Soot infrastructure (version 2.2.2) [19]. Free-instrumented programs generated by our compiler were executed in Jikes RVM (version 2.3.5) [2]. The run-time system of Jikes has been extended to support the explicit deallocation of objects via `free`. We evaluated benchmark programs from SPECjvm98 using the GNU Classpath Java libraries (version 0.14) [8]. Experiments were performed on a 2 GHz Pentium machine with 1GB of memory and running Linux FC 6.

### 5.1 Experimental Methodology

Our compiler performs a bytecode-level transformation that compiles standard Java bytecodes into programs that contain `free` instructions and `destroy()` destructor methods. We applied our compiler to the SPECjvm98 benchmarks. We also analyzed library classes from the *java.lang*, *java.util*, and *java.io* packages using the *open world* semantics explained in Section 3.5.3.

**Instrumenting Libraries** Since Jikes RVM itself is written in Java instrumenting Java library code in Jikes is a non-trivial task. To include the transformed libraries, one must rebuild Jikes with the extended libraries. Jikes is compiled with the help of an external JVM, and some of the library classes are used and compiled in this process.

This compilation strategy introduces two main difficulties. First, the external JVM does not recognize `free` instructions. Hence, we can't include `free` statements directly in the library code. Our solution to this problem was to replace the `free(x)` instructions in the libraries with indirect calls that are resolved dynamically.

A second difficulty is introduced from calling destructor methods. The external virtual machine loads some critical classes from its internal library (e.g. Object, String, Thread), and some other classes are taken from the transformed libraries. The internal classes have no extensions and thus have no destructor methods. Hence, we must ensure that the transformed libraries do not call destructors of such critical classes.

We have been able to integrate the transformed libraries in Jikes RVM baseline compiler. However, the Jikes RVM build process fails when compiling the optimizing compiler with the extended libraries. We haven't determined the reason for this problem.

**Method summaries** As mentioned earlier, we use method escape summaries [6] to approximate method call effects. Method summaries have two parameters:  $k$ , the heap depth limit, and  $b$ , the field branching factor limit. In these experiments, we have used  $k = 2$  and  $b = 2$  for the libraries, and  $k = 4$  and  $b = 15$  for the benchmarks.

Program	App. size		Analysis
	Classes	/ Meth.	time (sec)
compress	12	44	0.6
jess	151	690	6.6
raytrace	25	176	2.9
db	3	34	1.9
javac	176	1190	19.4
mtrt	26	180	3.0
jack	56	316	14.1
Total SPEC	449	2630	48.5
Libraries	538	5426	29.7

**Table 1.** Analysis times

For native methods, we manually wrote escape signatures that describe their behavior. We manually identified methods called by reflection and marked their formal arguments as shared. Due to the imprecision of the escape analysis, some method summaries were overly imprecise, for example for `Object.equals` and `Hashtable.get`. We manually changed these summaries to express the fact that these methods do not escape their arguments.

## 5.2 Compilation Statistics

Table 1 presents the program sizes and analysis running times. The library statistics are only for the packages we have analyzed. The analysis times are given in seconds and indicate that the analysis is efficient and scales well. On average, the analysis takes 0.08 seconds per class. This time is similar to the average time needed to load a class file, which is about 0.1 seconds. Method summaries were computed previously, analysis times to compute such summaries is around 30 seconds for the entire Java libraries [6]. These numbers are not included in Table 1.

Table 2 counts the transformations made by our compiler. The columns show the number of frees on variables, frees on field expressions, destructor methods, and calls to destructors. Variable deallocations are an order of magnitude more frequent than deallocation on field expressions. Only 5% of the variables are freed using the calling-context uniqueness transformation (column *CC*). The libraries don't include this transformation because of the *open world* semantics. About 70% of the frees are inserted after a statement, the other 30% are inserted on control flow edges, i.e. the deallocations occurs in a single branch of a conditional.

Many destructors are never called. A few destructors were not called because the condition  $(m, f) \notin E$  failed. For example, 7 calls in *javac* were omitted for this reason. In contrast, *jess* and *db* have destructors that are called in several program points.

The right portion of the table shows the total number of non-static reference fields, and the number of those identified as unique. On average, about 22% of the fields in the SPECjvm98 programs are identified as unique. The results show that unique references are not uncommon in some of these programs, for example, around half of the application fields are unique in *jess* and *compress*.

## 5.3 Memory Savings

Table 3 shows the amount of memory collected with the free statements inserted by our compiler. Note that all of the deallocated objects are freed when they are still reachable from variables or fields, hence a dynamic collector could not have collected them earlier.

The first column in the table shows the total memory allocated by a program, and the subsequent columns show the percentage of this memory collected by our analysis. Each column adds some features with respect to the previous column, hence numbers increase from left to right.

Program	Frees on		Num Dest.	Dest. Called	Fields	
	<i>x</i>	(CC) <i>x.f</i>			total/unique	
compress	6	(0)	14	8	9	19 12
jess	289	(26)	40	17	41	84 40
raytrace	107	(5)	8	5	2	41 7
db	53	(1)	8	1	4	7 1
javac	373	(21)	29	16	2	200 25
mtrt	106	(5)	8	5	2	41 7
jack	197	(8)	14	5	1	64 11
Total SPEC	1237	(66)	121	57	61	456 103
Libraries	946	—	73	27	33	640 54

**Table 2.** Counts of instructions added and unique fields identified.

By just freeing unique variables, the system can collect about 40% of the memory in these programs (column *Var + CC*). This number includes savings from the calling-context uniqueness transformation, which only contributes for 4% of the savings in *javac*. By adding deallocation of field expressions *x.f* the analysis can reclaim an additional 15% of memory (column *+Field*). This improvement is most noticeable in *compress* and *db*. In *compress* our tool recognized two weakly unique array fields that are initialized, temporarily shared, and nullified inside a loop. In *db* the analysis also recognized a weakly unique array field. The field is often allocated before performing an operation, and nullified after the operation completes. The analysis successfully deallocates the arrays each time the fields are nullified.

The column *+App Dest* shows the savings of including destructors in the application code. The benefit of using destructors is illustrated by the *jess* program, where our analysis can claim an additional 36% of the allocated memory. This destructor is exactly the example we illustrated earlier in Section 2.

Deallocations within the libraries slightly improve the results in *javac* and *jack*, where an additional 2% of memory is reclaimed (column *+Lib*). Calling the generated library destructors from the application code didn't improve the results. We believe the savings from the libraries are quite low because fields of commonly used classes are not identified as unique. This is partially because the *open world* semantics are too conservative. For example, the array field of `Vector` is not private, and thus is not labeled unique.

To illustrate the potential of destructors in the libraries, we have manually extended the `String` and `StringBuffer` classes with destructors (column *+String annot*). The destructors reclaim the character array used in these classes. Our analysis does not generate these destructors because the arrays are not always unique in the GNU library implementation, but their sharing can be modeled.

Generally the character arrays are unique, but when calling `StringBuffer.toString` (typically the last operation performed on a buffer) the implementation sometimes shares the `StringBuffer` array with the new `String`, instead of duplicating the array. When sharing the array, the `StringBuffer` sets a boolean flag indicating that its array is shared. Hence, the array in `StringBuffer` can be freed when the sharing flag is false. The array in `String` can be freed when the `StringBuffer` that created the string is dead.

Our manual extensions consisted of: adding a destructor to `StringBuffer` that conditionally frees the array based on the shared flag, calling this destructor whenever `StringBuffers` are freed, adding a destructor to `String` that unconditionally frees the array, and calling this destructor only when it is safe to do so. These changes had an important impact in *javac* and *jack*. In *javac* we observed a 18% improvement: 7% from adding the destructor of `StringBuffer` and 11% from `String`. Interestingly, 5% of the `String` savings are possible because of the calling-context

Program	Total Mem (Mb)	% Mem Freed				
		Var +CC	+Field	+App Dest	+Lib	+String annot
compress	111	0%	80%	80%	80%	80%
jess	305	26%	26%	62%	62%	62%
raytrace	161	80%	80%	80%	80%	80%
db	81	55%	80%	80%	80%	80%
javac	240	16%	16%	16%	18%	36%
mtrt	170	75%	76%	76%	77%	77%
jack	314	30%	30%	30%	32%	54%
Avg.	—	40%	55%	60%	61%	67%

**Table 3.** Memory reclaimed by analysis feature.

uniqueness deallocation. In *jack* we saw a 15% improvement from `StringBuffer` and 7% from `String`.

**Impact of Method Summaries** Reducing the precision of the summaries can have a considerable negative impact. This is because with less precision, the analysis assumes that more method arguments escape and more fields are accessed during method calls. For example, generating application signatures with  $k = 2$  will reduce the average savings from 60% down to 49%. If we remove the summaries completely, the overall savings would drop to 22%.

#### 5.4 Performance Impact

We measured the runtime overhead of our transformations using a version of JikesRVM with an optimizing compiler which inlines free statements. Since we haven’t been able to integrate the libraries in the optimizing compiler version of JikesRVM, we evaluated the runtime overhead without library transformations.

To evaluate the performance impact, we have compared the running times of the programs with explicit deallocation generated by our compiler against running those programs in a system with no memory reclamation (where frees are ignored and garbage collection is turned off).

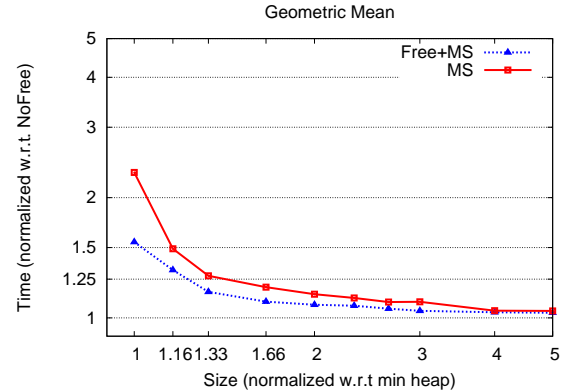
The overhead of instrumenting programs with frees and destructors was less than 6% for all applications and around 1.07%, on average. For *jess* explicit memory reclamation actually improved the overall performance by 5%. We believe that this is due to data locality. We have also measured the relative overhead of adding destructors on top of having only free statements. On average such overhead is 0.3%. We believe this overhead is low because our applications have few calls to destructors.

Our transformations can be combined with a garbage collector to reduce the overall run-time overhead. Figure 11 presents the performance comparison of combining frees with mark and sweep (Free+MS) against running the applications without any frees (MS). The plot shows the geometric mean of the results obtained from each application. The plot is normalized with respect to the minimum size used to run the applications with MS. Overall, it shows that on average an application can run 8% faster when using frees, and up to 33% faster when restricted to small heaps.

These numbers briefly illustrate the potential of a hybrid approach. However, integrating static memory reclamation and modern dynamic collectors (e.g. copying, generational) is a non-trivial task. This problem is not in the scope of this paper, but could be a possible direction of future work.

## 6. Related Work

**Linear Types and Unique References** There have been multiple proposals to extend programming languages with the concept of unique references or linear types [20, 14, 12]. The values of such references can be used only once; hence, the variables being read



**Figure 11.** Performance comparison using a mark and sweep collector with and without frees.

are “destroyed” and can no longer be used. This is a strong notion of uniqueness that disallows temporary sharing. Several researchers have relaxed this notion by allowing temporary stack sharing. For instance, *lent* parameters in AliasJava [1] and *borrowed* parameters proposed by Boyland [3] allow unique variables to be temporarily shared for the duration of a method call. Similarly, the *alias* declaration in Cyclone [12] introduces scoped regions where temporary sharing to the unique reference is allowed. None of the above approaches allow temporary heap sharing for unique references.

**Aliasing Inference** Most of the above work is concerned with proposing language features, and less with static analyses to infer them. Aldrich et al. [1] present an algorithm capable of inferring *unique* and *lent* references. Unlike our analysis, their algorithm is flow-insensitive and cannot identify references that are unique at some, but not all of the points in the program. Their analysis cannot identify the flow-sensitive uniqueness in Figure 2 or the weakly unique fields in Figure 4.

In work performed concurrently with ours, Ma and Foster developed UNO [13], a system to infer reference uniqueness and object ownership for Java. They also divide the inference problem in two phases: a local intra-procedural flow-sensitive analysis and an inter-procedural constraint-based analysis. Unlike our system, UNO’s local analysis is a may-alias analysis and does not perform strong updates on heap fields. As a result, unique field references can never have heap sharing. In contrast, our local analysis is a must-alias analysis that allows unique references to have temporary heap sharing.

**Shape Analysis** Shape analyses [22] are advanced static analyses capable of identifying the “shapes” of recursive heap structures, such as trees or acyclic lists, and distinguish them from similar structures with cycles or sharing. Shapes are closely related to the notion of uniqueness since unique references are the distinguishing aspect that characterizes the lack of sharing or cycles. Shape analyses use powerful abstractions and dataflow analysis algorithms to identify that shapes are preserved in spite of destructive operations that might cause temporary sharing. Existing shape analyses abstractions include shape graphs [15], 3-valued logic structures [16], or local abstractions of single cells [11, 5]. The must-alias analysis presented in this paper is, in fact, a form of shape analysis. The alias set abstraction can be regarded as an abstraction of the object that the aliases point to; hence, it is a form of the tracked cell abstraction similar in spirit to those that we proposed in our earlier work [11, 5]. Alias sets are, however, a simpler representation that captures reference counts, hit, and miss expressions all at once.

**Compile-Time Memory Management** Approaches to compile-time memory management include stack allocation of objects [21, 9], region-based memory management [18, 4, 7], and individual object deallocation [17, 5, 6, 10]. As mentioned at the beginning of the paper, individual object reclamation allows more flexible object lifetimes, requires fewer program changes, and little run-time support. Reclaiming entire structures is more challenging, but the work in this paper proposes a promising solution to this problem.

Individual object deallocation using shape analysis has been explored in the work of Shaham et al. [17] and in our previous work [5]. The analysis of Shaham is based on shape analysis using 3-valued logic and is intra-procedural only. Our shape analysis [5] is inter-procedural and tracks single object instances through the program. The tracked object analysis is less restrictive (it doesn't require uniqueness invariants to hold at method boundaries), but is significantly more expensive. Experiments on the same set of benchmarks show that the inference analysis in this paper is an order of magnitude faster (49 seconds instead of 11 minutes to analyze all the benchmarks) while enabling similar memory savings. For one benchmark (*jess*), the analysis proposed in this work can reclaim substantially more memory via object destructors. The current analysis is more efficient because the must-alias analysis is only intra-procedural.

We have previously developed a simple uniqueness analysis that computes uniqueness only for variables and cannot identify unique fields [6]. That analysis is less powerful and reclaims less memory. The escape and effect summaries developed in that work are however used in our current, enhanced uniqueness analysis. Guyer et al. [10] also proposed a static analysis for individual object reclamation. They combine flow insensitive points-to analysis to compute connectivity method summaries, with a flow-sensitive liveness analysis. Due to the points-to abstraction, their analysis cannot free objects placed in heap fields.

Except for [5], none of the above approaches to individual object deallocation can free entire heap structures; the analysis in [5] cannot deallocate recursive structures.

## 7. Conclusions

We have presented a novel, scalable uniqueness inference algorithm that computes uniqueness information for variables and fields. The analysis is flow sensitive and can recover uniqueness in spite of temporary stack and heap sharing. We have applied the uniqueness analysis to the compile-time deallocation of single objects using statements and destructor methods for collecting entire heap structures with unique references. Our results shows that the uniqueness analysis is scalable and enables the reclamation of a large fraction of the allocated memory.

## References

- [1] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Seattle, WA, November 2002.
- [2] B. Alpern, D. Attanasio, A. Cochi, D. Lieber, S. Smith, T. Ngo, and J. Barton. Implementing Jalapeño in Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.
- [3] J. Boyland. Alias burying: Unique variables without destructive reads. *Software Practice and Experience*, 31(6):533–553, 2001.
- [4] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the International Symposium on Memory Management*, Vancouver, Canada, October 2004.
- [5] S. Cherem and R. Rugina. Compile-time deallocation of individual objects. In *Proceedings of the International Symposium on Memory Management*, Ottawa, Canada, June 2006.
- [6] S. Cherem and R. Rugina. A practical escape and effect analysis for building lightweight method summaries. In *Proceedings of the International Conference on Compiler Construction*, Braga, Portugal, March 2007.
- [7] W. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, Washington, DC, June 2004.
- [8] GNU Classpath. <http://www.gnu.org/software/classpath/>.
- [9] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *Proceedings of the International Conference on Compiler Construction*, Berlin, Germany, April 2000.
- [10] S. Guyer, K. McKinley, and D. Frampton. Free-me: A static analysis for automatic individual object reclamation. In *Proceedings of the ACM Conference on Program Language Design and Implementation*, Ottawa, Canada, June 2006.
- [11] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, Long Beach, CA, January 2005.
- [12] M. Hicks, G. Morrisett, D. Grossman, and T. Jim. Experience with safe manual memory-management in Cyclone. In *Proceedings of the International Symposium on Memory Management*, Vancouver, Canada, October 2004.
- [13] K. Ma and J. Foster. Inferring aliasing and encapsulation properties for Java. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Montreal, Canada, October 2007.
- [14] N. Minsky. Towards alias-free pointers. In *Proceedings of the European Conference on Object-Oriented Programming*, July 1996.
- [15] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.
- [16] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3), May 2002.
- [17] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proceedings of the International Static Analysis Symposium*, San Diego, CA, June 2003.
- [18] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 1994.
- [19] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON '99*, Toronto, Canada, November 1999.
- [20] P. Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.
- [21] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.
- [22] R. Wilhelm, M. Sagiv, and T. Reps. Shape analysis. In *Proceedings of the International Conference on Compiler Construction*, Berlin, Germany, April 2000.