The CLOSER: Automating Resource Management in Java

Isil Dillig Thomas Dillig
Computer Science Department
Stanford University
{isil, tdillig}@cs.stanford.edu

Eran Yahav Satish Chandra
IBM T.J. Watson Research Center, USA
{eyahav, satishchandra}@us.ibm.com

Abstract

While automatic garbage collection has relieved programmers from manual memory management in Java-like languages, managing resources remains a considerable burden and a source of performance problems. In this paper, we present a novel technique for automatic resource management based on static approximation of resource lifetimes. Our source-to-source transformation tool, CLOSER, automatically transforms program code to guarantee that resources are properly disposed and handles arbitrary resource usage patterns. CLOSER generates code for directly disposing any resource whose lifetime can be statically determined; when this is not possible, CLOSER inserts conditional disposal code based on interest-reference counts that identify when the resource can be safely disposed. The programmer is only required to identify which types should be treated as resources, and what method to invoke to dispose each such resource. We have successfully applied CLOSER on a moderate-sized graphics application that requires complex reasoning for resource management.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Memory management (garbage collection)

General Terms Languages, Reliability, Verification

Keywords Resource Management, Interest Reachability, Logical Liveness, Higher-level Resource, Resource Interest Graph

1. Introduction

While automatic memory management via garbage collection has enjoyed considerable success in many widely-used programming languages, such as Java and C#, there are a number of situations where a programmer still needs to pay attention to timely resource reclamation. Consider the code in Figure 1, where the underlined code segments deal solely with resource deallocation. For example, should the programmer forget to call `socket.close()` at lines 13 and 30—which apparently serve no functional purpose in the application logic—there would be a leak of a socket resource in the system. While the application-level socket object can be reclaimed by the garbage collector when the encapsulating `BufferPrinter` object becomes unreachable, the associated *system-level* resource is not reclaimed until an explicit call to `close()` or at the end of the application. Conceptually, a call to `close()` removes the system-level socket from a "sockets in use" set. Omitting `close()` could cause a long-running application that repeatedly opens sockets to fail due to resource exhaustion. Graphics applications written using, for example, Eclipse SWT need to manage limited system resources such as fonts and colors in much the same way. For example, in Figure 1, the dispose call at line 26 is necessary to ensure that no font objects are leaked in the program.

```
1  class BufferPrinter {
2    private Buffer buf;
3    private BufferListener listener;
4    private Socket socket;
5
6    public BufferPrinter(Buffer buf) {
7      this.buf = buf;
8      this.listener = new BufferListener(this);
9      buf.addListener(listener);
10   }
11
12   public void setSocket(Socket s) {
13     if (socket!=null) socket.close();
14     this.socket = s;
15     if (!socket.isConnected())
16       socket.connect(...);
17   }
18
19   // Invoked by listener when buf is full. Writes buffer
20   // to socket (if present) and prints buffer on screen.
21   public void bufferFull() {
22     if (socket!=null)
23       socket.getOutputStream().write(buf.getContents());
24     Font font = new Font(...);
25     Util.displayMessage(buf.getContents(), font);
26     font.dispose();
27   }
28
29   public void dispose() {
30     if (socket!=null) socket.close();
31     buf.removeListener(listener);
32   }
33 }
```

Figure 1. Example illustrating resource usage patterns.

The problem of timely resource reclamation is not limited just to finite system-level resources. It occurs whenever an object that no longer serves a useful role cannot be garbage collected because of a reference from another longer-lived object. For instance, in Figure 1, the `BufferListener` object illustrates such an application

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.

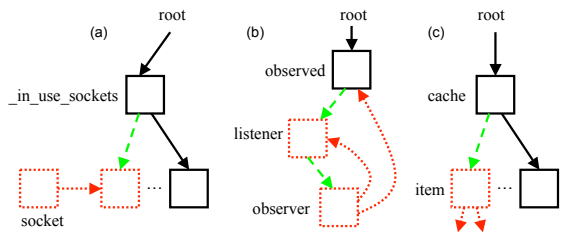

Figure 2. Logical lifetimes vs. Reachability. (a) `Socket` example. (b) `listener` example. (c) `hashtable` example. Dotted lines show the portion whose logical lifetime is over. Dashed links must be removed to enable garbage collection of the logically dead portion.

specific resource. In this code, a `Buffer` object referred to by `buf` is being observed by a `BufferPrinter` instance, which creates a link from the buffer to a listener at line 9. The programmer is expected to explicitly remove the listener from `buf` when the lifetime of this instance of `BufferPrinter` is over; otherwise the listener object will not be garbage collected as long as `buf` is live. Without the unlinking call at line 31, not only would the listener be dragged along through the lifetime of the buffer, but also the instance of `BufferPrinter` since the listener has a reference to it. (This is the well-known *lapsed listener* problem.) A similar performance problem in long-running Java applications occurs when an object placed in a global cache is not removed after its useful lifetime is over. The cache then drags along this useless object, potentially for the entire duration that the application is running.

Figure 2 shows a pattern common to all the above examples. In each case, the object in question is reachable starting from a root, which would prevent a reachability-based approach from reclaiming the object. In each case, an explicit release operation is required to remove a certain link to make the object unreachable, and thus exposed to reclamation. (In case of sockets, removing a system-level socket from the "in use" set is the equivalent of reclamation.) However, to make such a release call requires the knowledge that the object is not live in a *logical* sense: that the relevant behavior of the application is unaffected if the otherwise reachable object is reclaimed. For example, the count of items held by a hash table that implements a cache could change after the removal of an item, but typically a programmer would not consider that a change in relevant program behavior. In the sequel, by "resource", we mean an object of a kind that requires an explicit release in the manner described above; we call such release calls "dispose" calls.

1.1 Limitations of existing approaches

Manual Management Asking programmers to place resource release operations in their code brings back some of the problems of manual memory management. Bugs in placement of resource management operations are rampant: either because of releasing a resource too early, or perhaps more insidious, due to missing release operations along exceptional or less frequently executed paths, resulting in performance degradation. Furthermore, since reasoning about object lifetimes is hard, programmers often adhere to ad hoc design patterns, such as an "owner disposes" protocol often found in graphics libraries. This results in a large number of resources in the program because any object that can transitively reference a primitive graphics object is now also treated as a resource. Consequently, manual resource management code can unnecessarily complicate program logic and distracts the programmer from her main objective. The bug tracking system for Eclipse [2] shows numerous cases of resource leaks in Eclipse's GUI classes, most of which are due to programmer's confusion over dispose logic.

Finalization In Java, finalization invokes clean-up code when the garbage collector determines the memory associated with a resource to be unreachable; one could, for example, write a finalizer for socket to call the `close()` method. However, this strategy suffers from several drawbacks. First, in current JVM implementations, finalizers may not be run until memory is low; but the application may have run out of critical non-memory resources, such as sockets, long before finalizers are run. Second, finalizers are, by design, run asynchronously with respect to the application, making them unsuitable for managing resources when the disposal of a resource affects application semantics. This situation is very common in GUI code, where, for example, disposing of a widget also removes it from a visible display.

Language-based Solutions Java allows programmers to declare certain links as weak references, which are not considered reachability links by the garbage collector. For example, the link from a cache to its constituent items might be implemented as a weak reference. While it works well for caches, where a premature disposal of resource is not detrimental to application semantics, weak references are clearly not a solution to every situation. In C#, a `using` statement provides a scoped resource allocation construct to guarantee automatic resource disposal outside its scope [4]. This works well for situations in which resources are used only in local scope, but it is not a general solution to the problem of timely disposal.

1.2 Overview of our approach

Our approach is to automatically insert resource release calls in an application, allowing programmers to only allocate resources and not worry about releasing them. Our code transformation tool, CLOSER, would, for example, synthesize all of the code underlined in Figure 1; the only information required from a programmer is which types to consider resources along with the corresponding dispose calls. For the program in Figure 1, the programmer only needs to annotate that the `connect` method of `Socket` requires a corresponding call to `close`, that invoking `addListener` requires a corresponding call to `removeListener`, and so on.

CLOSER is based on a static analysis of logical lifetimes of resources. It first identifies which objects other than those already declared as being of a resource type behave like "higher-level" resources. These higher-level resources contain references to other resources, and therefore require clean-up operations at the end of their lifetime. For instance, in Figure 1, CLOSER would infer that `BufferPrinter` is a higher-level resource; it would then synthesize a dispose method for it and invoke this dispose method at appropriate points for every instance of a `BufferPrinter` class.

For tracking resource lifetimes, our analysis uses a new kind of flow- and context-sensitive points-to graph, the *resource interest graph*. The resource interest graph identifies memory locations that must, may, or must-not correspond to resources. In tracking resource lifetimes, it takes into account when inter-object references do or do not imply a logical interest. Based on sharing and lifetime information obtained from the resource interest graph, the analysis infers *strong static* dispose, *weak static* dispose, or *dynamic* dispose as one of the correct dispose strategies and fulfills this responsibility by inserting any necessary dispose operations into the program.

A prerequisite for statically disposing a resource is to identify a unique handle, called a *solicitor* that is responsible for disposing the resource. Sometimes it is not possible to determine a unique solicitor, primarily because resources may be shared, but also because of imprecisions in the static analysis. Real sharing of resources is especially common in GUI code, where the last reference to a resource is defined by some external input, such as a user click. For

example, in SWT applications, two windows may share a font object, and the font has to be disposed when the last window is closed by the user. In such cases, a programmer manually managing resources typically resorts to an ad hoc reference-counting scheme. In such cases of sharing, our technique automatically inserts code for dynamic reference counting, allowing it to work correctly for arbitrary resource usage patterns.

To illustrate these concepts, we briefly explain how CLOSER identifies correct dispose strategies for resources used in Figure 1:

Font: First, CLOSER infers that the font resource created in `bufferFull()` is not shared and identifies the variable `font` as its solicitor. Further, since the font object does not escape its allocating method, CLOSER infers the last use point of `font` in `bufferFull` as the correct static dispose point. Finally, since `font` is known to be a resource, CLOSER can perform a strong (unconditional) static dispose.

Socket: CLOSER first analyzes all call sites of `setSocket` to determine whether the input socket is shared. Assuming it is not, it determines that socket can be (conditionally) statically disposed through its encapsulating `BufferPrinter` object. Further, CLOSER identifies two points where the `close` method must be invoked: (1) before the overwrite to `this.socket` in `setSocket` because the resource interest graph reveals that the old target of `socket` becomes unreachable at this program point, and (2) in the synthesized dispose method. The conditional dispose is needed because the `setSocket` method might not have been called.

Listener: This case is similar to the dispose strategy for the socket. The main difference is that CLOSER needs to take into account the concept of *non-interest* (see Section 2) to infer that `listener` is not shared.

1.3 Advantages and Limitations of our approach

Our approach has the following advantages: (1) Disposal of resources is synchronous to the application, allowing the technique to work correctly even if disposal of a resource affects visible program behavior. (This is in contrast with finalizers.) (2) Our technique is not limited to a fixed class of system-specific resources. Any application-specific type can be declared to be a resource, as long as it obeys certain restrictions. (3) The technique does not require customizing the JVM and is therefore completely portable. (4) Unlike new language constructs that limit sharing patterns, our solution works correctly with arbitrary sharing patterns.

Among the main limitations are: (1) We require programmers to add certain information to be able to deal with cycles in object reference graphs (see Section 4). (2) We require a model of the relevant points-to behavior of methods unavailable for analysis as well as that of dynamically-loaded classes (see Section 8). (3) In this work, we do not attempt to reason about concurrency; our approach suffers from the same problems as other automated synchronous approaches when concurrency is allowed. (See, for example, [10] for a detailed discussion.) (4) We inherit the limitations inherent in path-insensitive flow analysis and the heap abstraction we use; it is possible to do better on both counts at an additional cost.

1.4 Contributions

In this paper, we make the following key contributions:

- We propose a new technique for fully automatic resource management based on source code transformation. Our technique is both powerful enough to allow for arbitrary resource usage patterns and is extensible enough to allow user-defined resources.
- We present a precise static analysis of logical lifetimes of resources. Resource disposal strategy computed based on our static analysis turns out to be very similar to manually written code, showing that the analysis automates the reasoning of resource lifetime very effectively.
- We implement our technique as a feature in Eclipse Java IDE and present a case study on a moderate-sized SWT graphics application. We show that our technique is powerful enough to automate resource management on an application that uses a large number of resources and that requires complex reasoning for correct resource management.

2. Basic Concepts

In this section, we introduce some useful basic concepts.

2.1 Object Liveness and Interest

The lifetime of a resource starts with allocation, followed by a series of uses of the resource, and ends with a reclamation.

DEFINITION 1. **(Liveness)** *An object is live up to and including its unique last use (field access or method invocation), and is dead thereafter.*

An ideal resource management strategy would perform a dispose operation immediately after the last use of a resource. However, it is hard to identify last use precisely either statically or dynamically. The technique often used instead is to approximate last use by the point in time when an object becomes unreachable in an *object reference graph*.

DEFINITION 2. **(Object Reference Graph)** *An object reference graph is a graph representing the heap snapshot at a given instant in the execution of a Java program. Vertices of the graph represent objects and labeled edges represent fields: the edge $o_1 \xrightarrow{f} o_2$ represents the fact that $o_1.f$ points to o_2. The graph contains a distinct root vertex that represents entry points into the heap.*

Examples of object reference graphs appear in Figure 2.

At the time an object becomes unreachable in the object reference graph, its last use has definitely been performed, although it could have been performed sooner. Indeed, many memory management strategies, both dynamic (garbage collection) and static (e.g., [21]), use reachability information to approximate last use. Unfortunately, as is clear from the examples in Figure 2, unreachability in an object reference graph can be a poor solution for timely resource reclamation; we would like to reclaim a resource soon after its *logical* last use.

DEFINITION 3. **(Logical Last Use)** *The logical last use of an object is the unique use such that the programmer deems any succeeding uses prior to reclamation as semantically unimportant.*

For example, in Figure 2(b), after the observer (`BufferPrinter`) is dead, any access from the observed (`Buffer`) to the listener is semantically unimportant and does not constitute a logical use. The judgment of whether an access constitutes a logical use has to be made by the programmer because the notion of logical use depends on program semantics. We cannot approximate logical last use based on the traditional notion of reachability. In order to approximate logical last use better, we introduce a modified form of reachability called *interest reachability*. We distinguish between two kinds of edges in an object reference graph: *interest* and *non-interest* edges. Intuitively, a non-interest edge from an object o_1 to an object o_2 means that the behavior of o_1 is not dependent on the presence or the state of o_2, but the edge is there to serve some other purpose. For example, in a cache implemented as a hash table, a link from the hash table to an item in the table is a non-interest edge; any access to that item by the hash table is logically

insignificant as far as last use is concerned. The notion of non-interest is application-specific and specified by the programmer.

DEFINITION 4. **(Interest Reachability)** *An object o is interest-reachable if there exists a path in the object reference graph from the root to o consisting only of interest edges.*

Just as reachability is used as an approximation to liveness, interest reachability can be used as an approximation to logical liveness. Consider Figure 2 again. The dashed edges in the figure are non-interest links; hence the resources shown in Figure 2 are not interest reachable (or not logically live). However, unless the non-interest edges are removed, the objects are not exposed for reclamation. The purpose of our analysis is to identify all non-interest edges that prevent timely reclamation and insert operations to break such redundant links.

The notion of interest in our model is categorical rather than temporal; a link between any two objects is either an interest or a non-interest link but cannot transition from one to the other. Non-interest links may not transition to interest links because it would otherwise not be safe to dispose an interest-unreachable resource when it is still reachable through non-interest links. Similarly, interest links cannot transition to non-interest links due to difficulties with the semantics of reference counting. However, loss of interest can still be indirectly expressed by setting a field to null.

Specifying Non-Interest In our system, every inter-object reference is considered to be an interest edge unless annotated otherwise by the programmer. Non-interest is declared using annotations on instance variables. For example,

```
class Buffer {
  @NonInterest
  BufferListener listener;
}
```

declares that any instance of class `Buffer` has a non-interest link to its `listener` field. In our system, classes inherit non-interest annotations from their supertypes. In the sequel, when we mention last use, we mean logical last use, unless said otherwise.

2.2 Resources

We now give a precise definition of a resource and discuss how to annotate resources in CLOSER.

DEFINITION 5. **(Resource)** *A resource r is an instance of any type whose specification requires that if a method m is called with r as the receiver or a parameter, then a matching method m' is called after the (logical) last use of r. Methods m and m' establish and tear down a non-interest link to the resource respectively.*

Since calling method m' fulfills an obligation incurred by calling m, we refer to m as an *obligating* method and to m' as the *fulfilling* method. According to Definition 5, a `Socket` is a resource because calling the `connect` method obligates a corresponding call to `close` after the socket's last use. Definition 5 does not require that the obligating and fulfilling methods be defined in the class type of the resource. For example, although `BufferListener` from Figure 1 is considered a resource, the `addListener` and `removeListener` methods are defined by the `Buffer` class.

Specifying Resources CLOSER allows programmers to define application-specific resources using simple annotations on obligating methods. For example, the annotation

```
@Obligation(obligates = "removeListener", resource = "1")
public void addListener(BufferListener l)
```

declares that calling `addListener` incurs an obligation to call `removeListener`, and that the first parameter (i.e., the listener) becomes a resource as a result of calling `addListener`. The special parameter 0 declares the `this` pointer, which is the receiver object of a method to be the resource. We refer to any resource declared explicitly by the programmer as a *primitive resource*. We write $\mathit{Obligation}(C, m_1) = \langle m_2, i \rangle$ to denote that method m_1 of class C is obligating, that m_2 is the corresponding fulfilling method, and that the i'th argument to m_1 is the resource. The notation $\mathit{Obligation}(C, m_1) = \emptyset$ indicates that method m_1 is not obligating. Classes inherit obligations from their parent types and we require that overridden methods do not declare additional obligations.

3. The Resource Interest Graph

To allow static analysis of resource lifetimes, we introduce a variation of a local points-to graph, called a *resource interest graph*.

DEFINITION 6. **(Resource Interest Graph)** *A* resource interest graph*, or RIG, for a method m at a given program point is a tuple $\langle V, E, \sigma_V, \sigma_E \rangle$ where V denotes a finite set of abstract memory locations used in m and E denotes a set of directed edges between these memory locations labeled with a field selector f. σ_V is a mapping from memory locations to a value from $\{0, 1, \top\}$, where 0 indicates that memory location l is not a resource, 1 indicates that l* must *be a resource, and $\top$ indicates that l* may *be a resource. σ_E is a mapping from edges to $\{\text{true}, \text{false}\}$, where* false *indicates a non-interest edge and* true *indicates an interest edge.*

The computation of RIG proceeds by intra and interprocedural propagation of the following information: (1) Π, an environment mapping program variables to a set of abstract locations, `Location`, (2) Γ, an environment with signature `Location` $\times$ `Field` $\rightarrow$ `Location`, defining nodes V and edges E of the RIG, (3) A, which maps each abstract location to a variable $(\alpha_1, \alpha_2, \ldots)$ ranging over values $\{0, 1, \top\}$, and (4) C, a set of global constraints between variables in A. The constraints appearing in C are conjunctions of two kinds of atomic constraints: $\alpha = c$, where c is a constant, or $\alpha_1 \sqsupseteq \alpha_2$.

The first six rules of Figure 3 describe the intraprocedural analysis for a subset of the Java language with typing judgments of the form $\Gamma, \Pi, A, C \vdash s : \Gamma', \Pi', A', C'$, with unprimed and primed symbols representing the state before and after the statement. (Figure 3 omits while statements since loops can be treated as tail-recursive functions.) Intraprocedural propagation tracks the environments Π and Γ in the expected way, using rules of flow-sensitive points-to analysis. At control-flow join points, the four pieces of information are merged, with C from the two paths conjoined. Allocation of an object creates a corresponding new variable α in A. If this abstract location is already known to be a resource, α is set to 1 and to 0 otherwise. (For this reason, the RIG computation goes through a fix point as more resource types are discovered; we will have more to say about this later.) Finally, when an obligating method is encountered, if parameter i of the call is a resource, then a fresh variable α in A is set to 1 if the corresponding actual parameter maps to a unique abstract location, and $\top$ otherwise. The body of the obligating method is also analyzed for side-effects using interprocedural propagation just like ordinary methods.

Interprocedural analysis (the last two rules of Figure 3) is summary-based. For each method, we compute a points-to summary, and additionally, a *method resource summary* (MRS), which defines constraints C between variables occurring in A at the entry of the method and A' at the exit. The purpose of MRS is to help ensure consistency among the variables in A that correspond to the same abstract location but in different local points-to graphs. Computation of method summaries is presented in the case METHOD DEFINITION in Figure 3. We use the notation $C \downarrow E$ to yield the strongest necessary condition for C ranging only over the variables in E. We define Π_{arg^*} to be the set of memory locations transitively

reachable from the arguments and return value so that the summary of a method only refers to memory locations that either have a valid mapping in the calling context or that are freshly allocated. In the METHOD CALL rule, the notation $\vdash_{\tau_i} m(\vec{C}\ \vec{v}) : \Gamma^{\tau_i,m}$ means that the points-to summary associated with method m of class τ_i is $\Gamma^{\tau_i,m}$. The points-to summary is a partition of the exit points-to graph Γ_{exit} of m containing only those locations transitively reachable from the arguments and return value. Since the points-to summary of the callee is defined in terms of the abstract memory locations in the callee's local heap, we define an operation $Proj_m(l)$ which maps location l used in the caller's local heap to the corresponding memory location l' in the local heap of callee m. Analogously, the operation $Proj_m^{-1}(l)$ maps a location in the local heap of the callee to an abstract location in the calling context. Note that locations allocated by the callee do not have a previous mapping; hence $Proj_m^{-1}(l) = l$ for fresh allocations. We also extend the projection and inverse projection operations to the points-to summary in the expected way. The points-to graph after the call is obtained by composing the points-to graph before the call with the points-to summary, $\Gamma \circ Proj^{-1}(\Gamma^m)$. The meaning of $\Gamma_1 \circ \Gamma_2$ is that locations referred to by Γ_2 are assigned the edges given by Γ_2, except for *summary* nodes (nodes representing a set of concrete memory locations), which also preserve outgoing edges given by Γ_1, and any location $l \in (\Gamma_2 \backslash \Gamma_1)$ preserves its old targets.

In Rule METHOD DEFINITION, MRS's of callee methods are used to enforce consistency constraints between the σ_V values of memory locations named differently in different local heaps but that correspond to the same concrete memory location. For example, the constraint $\alpha_i \sqsupseteq \alpha_i'$ enforces that the σ_V value of a location at the entry point of the callee is consistent with the σ_V value of the corresponding location at any call site. The constraint $\alpha_i''' \sqsupseteq \alpha_i''$ is similar, but enforces such consistency constraints between callee locations at the exit point and the corresponding locations in the calling context.

The environment Γ computed as above gives V and E of the RIG. A least solution to C determines the mapping σ_V. (These constraints can be easily solved, e.g., in the dataflow framework, hence we omit a lengthy description of how to solve the constraints. See Example 1 for a simple illustration.) The rules in Figure 3 omit σ_E since the notion of interest is type-based and $\sigma_E(l_1 \times f \to l_2)$ can be determined directly from the static type of l_1.

EXAMPLE 1. Consider the following two call sites of the `setSocket` method from Figure 1:

```
public void foo(BufferPrinter bf) {
  Socket s = new Socket();   //  α_foo,s = 0
  s.bind(...);
  s.connect(...);            //  α'_foo,s = 1
  bp.setSocket(s);           //  α_setSocket,s ⊒ α'_foo,s
}
public void bar(BufferPrinter bf) {
  Socket s = new Socket();   //  α_bar,s = 0
  s.bind(...);
  bp.setSocket(s);           //  α_setSocket,s ⊒ α_bar,s
}
```

The α variables in environment A and the constraints C generated by each statement are shown as comments in the above code fragment. The least solution for $\alpha_{setSocket,s}$ is $\top$, which expresses that the argument of `setSocket` may or may not correspond to a resource.

4. Higher-Level Resources

When a class τ has a field that points to a resource, estimating the liveness of that resource can be difficult, because any method in τ's public interface can be called as long as an instance o_τ is live. But if o_τ uniquely encapsulates the resource, the lifetime of o_τ is

GET FIELD

$$\frac{\Pi \vdash v_2 : \{l_1, .., l_n\} \quad l_i' = \begin{cases} \textit{fresh} & \Gamma(l_i, f) = \emptyset \\ \Gamma(l_i, f) & \textit{otherwise} \end{cases} \quad \Pi' = \Pi[v_1 \leftarrow \bigcup_i l_i']}{\Gamma, \Pi, A, C \vdash v_1 = v_2.f : \Gamma, \Pi', A, C}$$

PUT FIELD

$$\frac{\Pi \vdash v_1 : \{l_1, .., l_n\} \quad \Gamma' = \Gamma[(l_i, f) \leftarrow \Pi(v_2)]}{\Gamma, \Pi, A, C \vdash v_1.f = v_2 : \Gamma', \Pi, A, C}$$

ASSIGN

$$\frac{\Pi' = \Pi[v_1 \leftarrow \Pi(v_2)]}{\Gamma, \Pi, A, C \vdash v_1 = v_2 : \Gamma, \Pi', A, C}$$

ALLOC

$$\frac{\Pi' = \Pi[v \leftarrow l] \quad (l\ \textit{fresh}) \quad A' = A[l \leftarrow \alpha_l]\ (\alpha_l\ \textit{fresh}) \quad C' = C \wedge \begin{cases} \alpha_l = 1 & T\textit{'s constructor obligating} \\ \alpha_l = 0 & \textit{otherwise} \end{cases}}{\Gamma, \Pi, A, C \vdash v = \textit{new } T() : \Gamma, \Pi', A', C'}$$

IF STATEMENT

$$\frac{\Gamma, \Pi, A, C \vdash s_1 : \Gamma', \Pi', A', C' \quad \Gamma, \Pi, A, C \vdash s_2 : \Gamma'', \Pi'', A'', C''}{\Gamma, \Pi, A, C \vdash \textit{if}(*)\ \textit{then}\ s_1\ \textit{else}\ s_2 : \begin{array}{l} \Gamma' \cup \Gamma'', \Pi' \cup \Pi'', \\ A' \cup A'', C' \wedge C'' \end{array}}$$

OBLIGATING METHOD CALL

$$\frac{\textit{Type}(v) = \tau \quad \textit{Obligation}(\tau, m) = \langle m', i \rangle \quad \Pi \vdash v_i : \{l_1, ..., l_n\} \quad A' = A[l_i \leftarrow \alpha_i]\ (\alpha_i\ \textit{fresh}) \quad C' = C \wedge \begin{cases} \alpha_i = 1\ \textit{if } n = 1 \\ \alpha_i = \top\ \textit{otherwise} \end{cases}}{\Gamma, \Pi, A, C \vdash v.m(\vec{v}) : \Gamma, \Pi, A', C'}$$

METHOD DEFINITION

$$\frac{\textit{MRS}(\tau, m) = [A, A'; C] \quad \Pi = \{v_i \leftarrow l_i\}\ (l_i\ \textit{fresh}) \quad \emptyset, \Pi, A, \textit{true} \vdash s : \Gamma', \Pi', A', C' \quad C \leq (C' \downarrow \{A \cup A'\}) \quad \Gamma \subseteq (\Gamma' \downarrow \Pi^*_{arg})}{\vdash_\tau m(\vec{T}\ \vec{v})\{\vec{T}\ \vec{y};\ s\} : \Gamma}$$

METHOD CALL

$$\frac{\Pi \vdash v_2 : \{l_1, ..., l_n\} \quad \textit{Type}(l_i) = \tau_i \quad \vdash_{\tau_i} m(\vec{v}) : \Gamma^{\tau_i,m} \quad \Gamma' = \Gamma \circ \textit{Proj}^{-1}(\bigcup_i \Gamma^{\tau_i,m}) \quad \textit{MRS}(m) = \bigsqcup_i \textit{MRS}(\tau_i, m) = [A^m, A^{m'}; C^m] \quad A^m \vdash l_{m,i} : \alpha_i \quad A \vdash \textit{Proj}^{-1}(l_{m,i}) : \alpha_i' \quad A^{m'} \vdash l_{m',i} : \alpha_i'' \quad A' = A[\textit{Proj}^{-1}(l_{m',i}) \leftarrow \alpha_i'''] \ (\alpha_i'''\ \textit{fresh}) \quad C' = C \wedge (\alpha_i \sqsupseteq \alpha_i') \wedge (\alpha_i''' \sqsupseteq \alpha'') \wedge C^m}{\Gamma, \Pi, A, C \vdash v_2.m(\vec{v}) : \Gamma', \Pi, A', C'}$$

Figure 3. Inference Rules Defining the Resource Interest Graph

an upper bound to the resource's lifetime. In fact, a very common resource management policy in object-oriented programming is to implement a `dispose` method for such a class, and invoke the dispose method at the end of the lifetime of o_τ, similar to the use of destructors in C++. The dispose method in turn disposes its nested resources, possibly triggering cascading disposes.

Since class τ now requires calling a dispose method, we call τ a *higher-level resource*. Given a set of primitive resources annotated

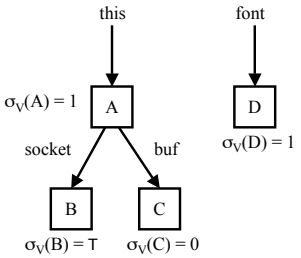

Figure 4. RIG at the program point in `bufferFull()` just before `font.dispose()`

by the user, our analysis infers the set of all transitive higher-level resources and synthesizes `dispose` methods for each such higher-level resource. For any given class τ, we say that the *dispose responsibility* for τ ($DR(\tau)$) is a set of fields $f_1, \ldots, f_n$ such that τ is responsible for disposing each f_i.

DEFINITION 7. **(Higher-level Resource)** Any instance of class τ for which $DR(\tau) \neq \emptyset$ is called a *higher-level resource*. Allocating an object of type τ obligates a call to `dispose`.

Given an RIG defined in Section 3, we can determine dispose responsibilities for each class τ according to the following rule:

$$\frac{(l_1 \times f \rightarrow l_2) \in E \quad \sigma_E(l_1 \times f \rightarrow l_2) = \mathit{true} \quad \sigma_V(l_2) \sqsupseteq 1 \quad \mathit{Type}(l_1) = \tau}{f \in DR(\tau)}$$

The above rule says that if there is an *interest* edge labeled with field selector f from location l_1 with static type τ to another location l_2 which may be a resource (i.e., $\sigma_V(l_2) \sqsupseteq 1$), we add f to the set of potential dispose responsibilities of τ. Since the ALLOC rule in Figure 3 depends on whether τ is inferred to be a resource, the RIG construction and higher-level resource inference are mutually dependent and must co-fixpoint.

EXAMPLE 2. Figure 4 shows the RIG for the program point in the method `bufferFull()` in our running example, just before the statement `font.dispose()`. $\sigma_V(B) = \top$ as a result of solving the constraints C globally, which takes into account calling contexts of `setSocket()` (see Example 1). $\sigma_V(C) = 0$ because `Buffer` is not a resource. The pointer analysis adds fields only lazily, so the `listener` field is not shown. `BufferPrinter` is a higher-level resource, hence $\sigma_V(A) = 1$. This fact is determined based on the rule above, as it concludes that `socket` $\in DR($`BufferPrinter`$)$. The local variable `font` points to a resource, thus $\sigma_V(D) = 1$. This is because the constructor of `Font` is an obligating method, and the resource unconditionally flows to this program point.

Obviously, the notion of higher-level resources is only meaningful when a partial order on resource types can be defined. However, for a cyclic RIG, it is possible that $f : \tau_2 \in DR(\tau_1) \wedge f' : \tau_1 \in DR(\tau_2)$ (i.e., τ_1 is responsible for disposing τ_2 and vice versa). This situation arises in two cases: (1) in unbounded data structures, and (2) in the case of *back pointers* (e.g., see Figure 2b). For this reason, it is necessary to impose some restriction on the kinds of cycles that may arise in the RIG. For unbounded data structures, we require that they be implemented through the Java collection interface, allowing us to reason about the entire data structure as a whole rather than individually about its constituent parts. This restriction also allows us to easily traverse unbounded data structures using the `Iterator` interface. For back pointers, we require that the programmer define a partial order on dispose responsibilities by annotating appropriate edges as non-interest edges. For example, in the case of listeners, the cycle can be broken by annotating the edge from the listener to the observer as a non-interest edge.

5. Solicitor Inference

Given a resource r used in the program, there are three possible ways in which r may be disposed:

- Strong static dispose, which requires no dynamic checks and disposes the resource directly by calling the fulfilling method.
- Weak (conditional) static dispose, which checks whether the object actually became a resource at runtime.
- Dynamic dispose, which requires that we keep an interest edge count to the resource at run-time.

A prerequisite for disposing a resource statically, either through strong or weak dispose, is to identify a *solicitor* for it. The solicitor of a resource r is a referent to r which has the unique responsibility to dispose r. The solicitor has to obey the following requirements:

1. If o is a solicitor of resource r, there may not be another object o' which also disposes r.
2. If o is a solicitor of r, o is guaranteed to dispose r for all possible execution traces.

Clearly, determination of a solicitor requires analysis of relative lifetimes of the resource as well as the various referents to it. Note that when a resource r is truly shared, it is not possible to find a solicitor for r that obeys the above requirements. If r is a truly shared resource, then the object disposing r necessarily varies across different execution traces, violating (2) above.

In this section, we present a technique for inferring a solicitor for each resource, if one exists. Our technique first infers a *solicitor candidate* for a given resource r (written *SC(r)*) at every program point using the reachability information encoded in the local RIG. Then, given a set of inferred solicitor candidates, we perform a simple dataflow analysis to determine the actual solicitor.

We define a *path* $\mathcal{P}$ as a triple $\langle l, f_1 \circ \ldots \circ f_n, \mathit{May/Must} \rangle$ consisting of a memory location l, a sequence π of field labels $f_1 \circ \ldots \circ f_n$, and an element of the set $\{\mathit{May}, \mathit{Must}\}$. A *must path* indicates that the path has a unique target, while a *may path* can have multiple targets. For every location r such that $\sigma_V(r) \sqsupseteq 1$ (i.e., a possible resource), we compute a set of paths $\mathit{Paths}(r)$ that reach resource r. The set $\mathit{Paths}(r)$ describes all memory locations that transitively reference r at a given program point. This can be computed from the RIG according to the rules given in Figure 5. In addition, we add a default path $\mathcal{P}_{AR} = \langle AR, \epsilon, \mathit{Must} \rangle$ because the activation record has a reference to every resource used in a method. We use $\mathcal{P}$ (or $\mathcal{P}_i$) to mean any path other than $\mathcal{P}_{AR}$.

To compute *SC(r)* at a given program point, we exhaustively apply the following simplification rules to $\mathit{Paths}(r)$:

1. Let $\mathcal{P}_1 = \langle l_1, \pi_1, \_ \rangle \in \mathit{Paths}(r)$ and let $\mathcal{P}_2 = \langle l_2, \pi_2, \_ \rangle \in \mathit{Paths}(r)$ such that π_2 is a proper suffix of π_1. If $\langle l_1, \pi', \_ \rangle \in \mathit{Paths}(l_2)$, we unify $\mathcal{P}_1$ and $\mathcal{P}_2$ and choose $\mathcal{P}_1$ as the equivalence class representative. (The idea is that the existence of the longer path implies the existence of the shorter path.)
2. Let $\mathcal{P}_{\mathit{Must}} = \langle l, \pi, \mathit{Must} \rangle \in \mathit{Paths}(r)$ and $\mathcal{P}_{\mathit{May}} = \langle l, \pi, \mathit{May} \rangle \in \mathit{Paths}(r)$. We unify $\mathcal{P}_{\mathit{Must}}$ and $\mathcal{P}_{\mathit{May}}$ and choose $\mathcal{P}_{\mathit{Must}}$ as the equivalence class representative. (The existence of a must path to the resource implies the existence of a may path.)
3. Let $\mathcal{P} = \langle l, \pi, \mathit{Must} \rangle \in \mathit{Paths}(r)$. We unify $\mathcal{P}_{AR}$ with $\mathcal{P}$ and choose $\mathcal{P}$ as the equivalence class representative. (Whenever there is a must-path to the resource through a higher level resource, any local handles to the resource delegate dispose responsibility to the higher level resource.)

$$\frac{}{\langle r, \emptyset, \mathit{May}\rangle \in \mathit{Paths}(r)} \qquad \frac{\exists l_i.\ l_i \in \Gamma(l, f_1) \wedge \sigma_E(l \times f_1 \rightarrow l_i) = \mathit{true} \wedge \langle l_i, S, \mathit{May}\rangle \in \mathit{Paths}(r)}{\langle l, f_1 \circ S, \mathit{May}\rangle \in \mathit{Paths}(r)}$$

$$\frac{}{\langle r, \emptyset, \mathit{Must}\rangle \in \mathit{Paths}(r)} \qquad \frac{\forall l_i.\ l_i \in \Gamma(l, f_1) \wedge \sigma_E(l \times f_1 \rightarrow l_i) = \mathit{true} \wedge \langle l_i, S, \mathit{Must}\rangle \in \mathit{Paths}(r)}{\langle l, f_1 \circ S, \mathit{Must}\rangle \in \mathit{Paths}(r)}$$

Figure 5. Inductive rules for computing *Paths*(r)

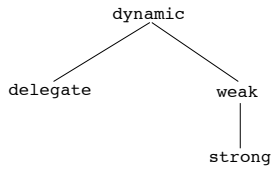

Figure 6. Partial order for *DS(r)*

After applying the above simplification rules to $\mathit{Paths}(r)$ computed for program point p, if the cardinality of the resulting set is greater than 1, we conclude there is no solicitor for r and write $\mathit{SC}_p(r) = \bot$. Intuitively, if the resulting set contains two paths $\mathcal{P}_1$ and $\mathcal{P}_2$, there must be at least two independent higher-level resources initiating the chain of calls leading to the disposal of resource r. Hence, we conclude there is no unique solicitor for r. If the resulting set contains some path $\mathcal{P}$ as well $\mathcal{P}_{AR}$, then the resource only conditionally escapes to a higher-level resource. This implies that in some execution traces, the dispose responsibility for the resource belongs to the activation record, while in other cases, the dispose responsibility belongs to the object to which r escapes, again requiring dynamic reference counting in our approach. On the other hand, if the set obtained after applying the simplification rules is a singleton $\{\langle l, \pi, \mathit{Must}\rangle\}$, we conclude $\mathit{SC}_p(r) = l.\pi$. Note that neither the singleton $\{\langle l, \pi, \mathit{May}\rangle\}$ nor $\emptyset$ may be the outcome of applying the simplification rules to the original $\mathit{Paths}(r)$.

The next step in our technique is to infer the actual solicitor from the set of solicitor candidates inferred at each program point. Intuitively, we need to ensure that the inferred solicitor candidates "agree" for different program points. The only apparent exception to this consistency requirement occurs when the inferred solicitor is the *AR* at one program point ρ_1 and some other object $l.\pi$ at another program point ρ_2, and ρ_2 is a control flow successor of ρ_1. This apparent inconsistency can be resolved through a transfer of dispose responsibility from the *AR* to the higher-level resource.

EXAMPLE 3. Consider the following statement in our running example:

```
this.listener = new BufferListener(this);
```

The allocated `BufferListener` initially has *AR* as its solicitor, but when it is assigned to the field `this.listener`, the current instance `this` becomes the solicitor.

Note that the only possible transfer of dispose responsibility is from the *AR* to a higher level resource. Dispose responsibility can never be transferred from one higher-level resource to another or from a higher-level resource to the *AR* because in our model higher-level resources are always deemed responsible for disposing of their nested resources.

To determine the actual solicitor for a given resource, we perform a simple forward data flow analysis defined by the following transfer function and join operator:

$$S_{\mathit{out}}(r) = \begin{cases} \mathit{SC}(r) \sqcup S_{\mathit{in}}(r) & \mathit{SC}(r) \neq \mathit{AR} \ \vee \\ & (\mathit{SC}(r) = \mathit{AR} \wedge \mathit{Used}(r)) \\ S_{\mathit{in}}(r) & \mathit{SC}(r) = \mathit{AR} \wedge \neg \mathit{Used}(r) \end{cases}$$

where *SC(r)* indicates the inferred solicitor candidate at the current program point, $\mathit{Used}(r)$ means resource r is read or written to by the current statement, and the join operator is defined as $l_1.S_1 \sqcup l_2.S_2 = \bot$ if $l_1 \neq l_2$ or $S_1 \neq S_2$, and $l_1.S_1$ otherwise. The inferred solicitor for a resource r is then given by $S_{\mathit{out}}(r)$ for the exit point of the method. In the above rule, notice the importance of the predicate $\mathit{Used}(r)$: For r to be statically disposed, all local handles to r must be dead after any interest links from a higher-level resource to r are broken.

EXAMPLE 4. Consider the `bufferFull` method from Figure 1. The analysis infers *AR* and `this` as the solicitors for the font and socket objects respectively.

6. Fulfilling Dispose Responsibilities

In the previous section, we discussed how to infer solicitors for every abstract memory location that may correspond to a resource. However, since our goal is ultimately to insert correct dispose management calls into source code, we need to formalize how the dispose strategy for each abstract memory location translates to the correct dispose strategy for variables in program text. In this section, we discuss how to assign a dispose strategy to each program variable using the concept of solicitors from Section 5. We write $\mathit{DS}(v) \in \{\texttt{dynamic}, \texttt{delegate}, \texttt{strong}, \texttt{weak}\}$ to mean that the dispose strategy for variable `v` is one of (1) `dynamic`, that is to dynamically dispose the referent of `v` using reference counting, (2) `delegate`, that is to leave the disposal of `v`'s referent to some other variable, (3) `strong`, that is to statically dispose `v`'s referent using `v.dispose()`, and (4) `weak`, that is to statically but conditionally dispose `v`'s referent using a `Manager` class.

While every program variable references exactly one concrete memory location at run-time, the static points-to information defines a mapping from every variable `v` to a set of possible abstract memory locations that `v` may point to. Hence, in order to determine the correct dispose strategy for `v`, we first compute $\mathit{DS}(v, l)$ which yields the correct dispose strategy for `v` *assuming it points to location* l. We can then compute the dispose strategy for `v` as:

$$\mathit{DS}(v) = \bigsqcup_i \mathit{DS}(v, l_i)$$

for every location l_i in `v`'s points-to set. The join operator used above is defined according to the partial order in Figure 6.

To compute $\mathit{DS}(v, l_i)$, we assume that the result of every fresh memory allocation is assigned to a freshly declared variable. We write $\mathit{Handle}(l) = v$ to indicate that variable `v` captures the result of allocating l. We also assume that the handle is not overwritten so that it can be used for disposing l. Without these assumptions, we may need to generate additional temporary variables used exclusively for the purpose of disposing the resource. (The implementation does not make these assumptions.) We can then compute $\mathit{DS}(v, l_i)$ as follows:

$$\mathit{DS}(v, l) = \begin{cases} \texttt{dynamic} & \mathit{Solicitor}(l) = \bot \wedge \sigma_V(l) \sqsupseteq 1 \\ \texttt{strong} & \mathit{Solicitor}(l) = \mathit{AR} \wedge \mathit{Handle}(l) = v \\ & \wedge\ \sigma_V(l) = 1 \\ \texttt{weak} & \mathit{Solicitor}(l) = \mathit{AR} \wedge \mathit{Handle}(l) = v \\ & \wedge\ \sigma_V(l) = \top \\ \texttt{delegate} & \mathit{otherwise} \end{cases}$$

7. Synthesis of Dispose Methods

In this section, we discuss how to synthesize dispose methods for higher-level resources and how to dispose fields that are overwritten in methods of a class. Determining the right dispose strategy for a field is a non-modular property because it requires reasoning about every instance of the class. We write $\mathcal{I}(\tau)$ to denote any instance of

type τ and $DS(\tau, f)$ to denote the correct dispose strategy for field f. We compute $DS(\tau, f)$ as follows:

$$DS(\tau, f) = \begin{cases} \texttt{dynamic} & \exists \mathcal{I}(\tau).\ \mathit{Solicitor}(\mathcal{I}(\tau)) = \bot \\ & \wedge\ f \in \mathit{DR}(\tau) \\ \\ \texttt{weak} & \forall \mathcal{I}(\tau).\ \mathit{Solicitor}(\mathcal{I}(\tau)) \neq \bot \\ & \wedge\ f \in \mathit{DR}(\tau) \end{cases}$$

A resource field f of class τ is dynamically disposed if there is any instance o of τ such that $o.f$ does not have a unique solicitor. In contrast to local variables, fields can neither be delegated nor strongly statically disposed. Fields may not be delegated because higher level resources are always responsible for the disposal of their nested resources unless annotated explicitly as non-interest links by the programmer. We choose never to strongly dispose fields since they are often manipulated through setters, and correct disposal of the resource often requires checking whether the setter method was called. For example, in Figure 1, we may not unconditionally dispose the socket because the `setSocket` method of `BufferPrinter` may not have been called. Although it may be possible to unconditionally dispose fields in some cases (e.g., if the field is allocated in the constructor and does not escape), our implementation only weakly disposes instance variables. [1]

EXAMPLE 5. Consider the following use of the `BufferPrinter` class from Figure 1:

```
Logger log = new Logger();
BufferPrinter bp = new BufferPrinter(log.getBuffer());
log.setPrinter(bp);
```

Suppose `setPrinter` establishes an interest link from the `Logger` object l to the `BufferPrinter` object b. CLOSER infers that `Logger` is a higher-level resource, that the solicitor for l is the activation record, and that l can be strongly disposed using handle `log`. The strong dispose call `log.dispose` inserted by CLOSER disposes b, which in turn disposes the native font and socket resources and the application-level `BufferListener`. The dispose strategy for variable `bp` is `delegate`, since the solicitor for the memory location referenced by `bp` is already disposed through l.

8. Implementation

We have built a prototype implementation of the resource management technique proposed in this paper as an Eclipse plug-in. To use our system, the user annotates resources and non-interest links using the standard Java 1.5 annotation syntax. During software development, the programmer may omit any resource management code and later invoke our plug-in to insert dispose code. If desired, the programmer can inspect and change the automatically resource managed code; this does not require any extra knowledge because all the instrumentation is in standard Java code. From a software engineering point of view, allowing the programmer to inspect changes to the source code is desirable because she can understand and, if required optimize, the instrumented code without running the application. In contrast to dynamic "black-box" techniques, such as modifications to the JVM, CLOSER appears transparent to the programmer, allowing her to interact directly with modifications to the source code.

Static Analysis Implementation The underlying static analysis of CLOSER is built on the IBM T.J. Watson Libraries for Analysis (WALA) which provides the basic functionality for performing static analysis on Java byte code [5]. The source code transformation component of the system utilizes the Eclipse JDT framework, which provides libraries for rewriting Java source code [3]. The most important restriction of our current prototype implementation is that it requires the user to manually refactor all inner classes.

In earlier sections, we implicitly assumed only closed programs because determining the dispose strategy for fields requires knowledge about every instance of a class. Unfortunately, the entire program may not always be available for analysis. In such cases, we require manual specifications concerning the relevant behavior of missing methods, such as relevant points-to annotations e.g., conceptually similar to AliasJava [6]. For example, in our case study (see Section 9), we needed to annotate the relevant side-effects of native methods called by the Eclipse SWT library.

Dynamic Instrumentation All dynamic instrumentation is accomplished by calling static methods of a `Manager` class, which tracks resource reference counts for dynamically disposed resources and other run-time information necessary for weakly disposing a resource. Unlike traditional reference counting schemes which track the number of physical links to a memory location, our reference counting technique differentiates between interest and non-interest links. We refer to this modified version of reference counting as *interest counting*. CLOSER inserts increment and decrement count statements before and after every assignment involving a variable that may reference a resource if (i) the inferred dispose strategy for that resource is `dynamic` and (ii) for put and get field instructions, there is no non-interest annotation between the parent type and field. Our dynamic instrumentation does not rely on modifying the JVM; we insert increment and decrement count statements directly into source code.

The `Manager` class is also involved in weak static dispose events. In particular, if a resource is conditionally acquired, CLOSER generates code to save the resource at calls to obligating methods and queries whether the resource has been acquired before disposing it. In some special cases of weak static dispose, a simple null check before the dispose may be sufficient.

9. Case Study: Automating an SWT Graphics Showcase Application

To evaluate our proposed resource management strategy, we applied CLOSER to an SWT Graphics Showcase Example application available through the Eclipse documentation [1]. This 7,440 line typical GUI-based Java application illustrates different graphics capabilities of the SWT library and is a good target for evaluating CLOSER because it is both large enough to be realistic and also small enough to allow us to remove all the existing resource management code and refactor inner classes (see Section 8). Furthermore, this application uses 67 different kinds of resources and has reasonably complex resource management logic. In this section, we report on our preliminary experiences and compare the code produced by CLOSER with the original manually written code.

Figure 7 compares various statistics between the original application and the resulting instrumented code. Figure 8 presents the number of manual annotations required by CLOSER. In the application we analyzed, the user has to annotate only five primitive resources; CLOSER then infers 34 other primitive resource types due to inheritance and 28 higher-level resources. These 28 inferred higher-level resources correspond exactly to the classes designated as higher-level resources by the programmer. This one-to-one correspondence between the resources in the original and instrumented code (row 1 of Figure 7) indicates that the notion of higher-level resources proposed in this paper is a natural fit for resource management in object-oriented languages.

[1] In contrast to variables, when a field is overwritten with itself (e.g., for some statement `this.f = f`, the old and new `f`'s alias one another), the overwritten resource may be prematurely disposed. Hence, static disposal of fields must explicitly check for this special case.

	Original	Instrumented
# Resources	67	67
# Strong Static Dispose	116	117
# Weak Static Dispose	14	63
# Dynamic Dispose	0	0
# Number of Resource Bugs	1	0
# Lines of Resource Mgmt Code	316	356
Resource Mgmt Code to Application Size Ratio	4.2%	4.9%

Figure 7. Comparison Between Manually vs. Automatically Resource Managed Applications

In the original application, the programmer instruments 116 strong static dispose calls compared with the 117 calls inserted by CLOSER. Incidentally, the extra dispose call inserted by our tool revealed a resource leak in the original application, where the dispose call on a `Transpose` object (a resource in the SWT library) was forgotten by the programmer.

The number of weak static dispose calls instrumented by CLOSER is larger than the number weak dispose calls present in the original code (63 vs. 14). All redundant weak dispose calls inserted by CLOSER are due to our path-insensitive analysis. Without exception, every single redundant weak dispose call inserted by CLOSER was due to the use of the following programming pattern:

```
private void paint() {
  if(image == null) {
    image = new Image(...);
  }
}
```

In frequently invoked methods, such as `paint` in the above example, the programmer uses the condition `image == null` to detect the first invocation of the method and allocates the resource only then. Since the image is overwritten only when the instance variable `image` is null, it is clearly not necessary to dispose `image` before the overwrite. Since our tool is not path-sensitive, it does not detect this correlation and inserts the following redundant weak static dispose code:

```
private void paint() {
  if(image == null) {
    if(image!=null){
      image.dispose();
    }
    image = new Image(...);
  }
}
```

Since this code is invoked exactly once (i.e., for the first invocation of `paint`), the overhead of this redundant dispose call is negligible, albeit a visual disturbance in the generated code. However, it would be easy to implement a peephole optimizer to detect such cases.

There were no shared resources in the application we analyzed, and hence no manual reference counting was present in the original code. CLOSER also did not detect any spuriously shared resources and was able to infer a unique solicitor for every resource used in the application.

While we only instrument the SWT graphics showcase application, CLOSER also needs to analyze all transitively called methods of the SWT and Java libraries, summing up to a total of 8,836 methods (or several hundred thousand lines of byte code) which take approximately 11:08 minutes to analyze on a single core. Since the underlying libraries (e.g., SWT) typically do not change during software development, the large bulk of this computation can be memoized to substantially decrease running time.

Resource Annotations	5
Non-Interest Annotation	9
Annotated Points-to Information	16

Figure 8. Number of Required Annotations

Even though all the byte code of the SWT and Java libraries are available, some low-level library functions call native methods which CLOSER cannot reason about. Unfortunately, some native methods create important points-to links and need to be annotated with the appropriate points-to information. We found 16 such instances of native library calls that required manual annotation.

We believe that the results of our case study as presented in Figures 7 and 8 are satisfactory in the following respects:

- Our tool requires reasonably few annotations to work correctly in a reasonably large application with over 7,400 lines of code.
- CLOSER generates code that is very similar (in fact, almost identical) to hand-written code, except in the few situations where the programmer checks for edge cases. This suggests that the automatically generated code should be almost as efficient as hand-written code in the vast majority of cases.
- CLOSER does not cause a substantial code bloat. As shown in Figure 7, the resource management code increases by only 40 lines (0.7% of the whole application).
- Our tool relieves the programmer from the burden of manually managing resources. In the application we analyzed, the programmer needs to write 316 lines less code (4.2% of the entire application), allowing her to focus on the more important programming tasks than disposing resources.
- Most importantly, resource management automated by CLOSER is correct by construction, making it impossible to leak or double dispose resources. The existence of a single resource bug in the SWT showcase graphics application written by expert Java programmers highlights the error-prone nature of manual resource management and the potential usefulness of a tool like CLOSER.

10. Related Work

Static resource management: Shaham et. al. [22] present an approach for static garbage collection in Java programs. Their approach employs shape analysis to approximate the last time an object is used and inject appropriate memory deallocation code. Their approach is based on shape analysis and does not scale to large programs. Cherem and Rugina [12] follow the same approach but employ a more scalable shape analysis. Our approach has two advantages: (i) Since resources are used in a more limited regime than arbitrary memory locations, our approach can adopt a less precise and more scalable analysis. (ii) We allow the user to specify non-interest, making a distinction between physical and logical last use. In the above approaches reading a field of the managed object is considered a use. As a result, native resources associated with an object will be released when all pointers to it are no further dereferenced. Our approach allows native resources associated with an object to be released earlier, even when the object itself remains live for a longer period of time.

Escape analysis (e.g., [9]) allows heap objects to be allocated on the stack when they do not escape the scope of a method. In principle, this technique can be applied to dispose resources (and is in fact parallel to our local disposal policy). While this technique has been shown to be useful, it is limited to objects that do not

escape their allocating method. Our approach also applies in cases where the resource does escape method boundaries.

In region-based memory management [23, 7, 18], the lifetime of an object is predicted at compile-time. An object is associated with a memory region, and the allocation and deallocation of each region are inferred automatically at compile time. While this approach can be applied to manage resources, our approach permits finer-grained resource management.

Static error detection and verification: Among bug finding tools, Livshits [20] uses simple and scalable static analyses to detect SWT resource leaks in Eclipse. A number of other static tools target detection or prevention of memory and resource leaks [19, 13, 14, 16, 15, 24]. In principle, most of these approaches are capable of detecting cases where an object is leaked or double disposed. In contrast with our approach, all of these techniques require the programmer to write manual resource management code. Furthermore, fixing the errors detected by these tools is known to be a hard problem on its own. Finally, some of these approaches either require additional (potentially cumbersome) annotations or restrict the class of programs that may be written, e.g. by restricting aliasing [14, 13, 16].

Dynamic error detection: Dynamic tools such as Tracematches [8], and MOP [11] are also able to detect violation of typestate properties, and in particular detect resource leaks. In [11], JavaMOP was used to successfully detect a number of resource leaks in Eclipse. In contrast to dynamic bug-finding techniques which are necessarily unsound, our tool is correct by construction.

Hybrid approaches: Similar to CLOSER, [17] also uses a combination of points-to and liveness analysis to reduce the overhead of garbage collection by automatically inserting deallocation code into the program. However, since CLOSER is mainly geared towards resource instead of memory management, there are some important differences: First, [17] mainly targets static deallocation of short-lived objects in local scope, whereas CLOSER must also track objects whose life times span multiple methods because many resources have non-local scope. Second, CLOSER distinguishes between reachability via interest and non-interest links; reasoning based only on physical reachability would not be adequate for solving the resource drag problem.

11. Acknowledgements

We would like to thank the anonymous reviewers and Suhabe Bugrara for their valuable feedback and useful suggestions.

References

[1] http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.platform.doc.isv/whatsNew/platform_isv_whatsnew.html. See section titled SWT graphics showcase.

[2] Bug reports for Eclipse projects. See bugs.eclipse.org/bugs.

[3] Eclipse Java Development Tools. See www.eclipse.org.

[4] The `using` statement. See C# Language Specification, msdn.microsoft.com.

[5] Watson Libraries for Program Analysis (WALA). Available at wala.sf.net.

[6] ALDRICH, J., KOSTADINOV, V., AND CHAMBERS, C. Alias annotations for program understanding. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (New York, NY, USA, 2002), ACM, pp. 311–330.

[7] ALEXANDER AIKEN, M. F., AND LEVIEN, R. Better static memory management: Improving region-based analysis of higher-order languages. In *Proc. ACM Conf. on Programming Language Design and Implementation* (June 1995).

[8] ALLAN, C., AVGUSTINOV, P., CHRISTENSEN, A. S., HENDREN, L., KUZINS, S., LHOTÁK, O., DE MOOR, O., SERENI, D., SITTAMPALAM, G., AND TIBBLE, J. Adding trace matching with free variables to aspectj. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (New York, NY, USA, 2005), ACM, pp. 345–364.

[9] BLANCHET, B. Escape analysis for object oriented languages. application to Java^{tm}. In *OOPSLA* (Denver, 1998).

[10] BOEHM, H. Destructors, finalizers, and synchronization. *ACM SIGPLAN Notices 38*, 1 (2003), 262–272.

[11] CHEN, F., AND ROŞU, G. MOP: An Efficient and Generic Runtime Verification Framework. In *Object-Oriented Programming, Systems, Languages and Applications(OOPSLA'07)* (2007).

[12] CHEREM, S., AND RUGINA, R. Compile-time deallocation of individual objects. In *ISMM '06: Proceedings of the 2006 international symposium on Memory management* (New York, NY, USA, 2006), ACM Press, pp. 138–149.

[13] DELINE, R., AND FAHNDRICH, M. Enforcing high-level protocols in low-level software. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM Press, pp. 59–69.

[14] DELINE, R., AND FÄHNDRICH, M. Adoption and focus: Practical linear types for imperative programming. In *Proc. ACM Conf. on Programming Language Design and Implementation* (Berlin, June 2002), pp. 13–24.

[15] FINK, S., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. Effective typestate verification in the presence of aliasing. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis* (New York, NY, USA, 2006), ACM Press, pp. 133–144.

[16] FOSTER, J. S., TERAUCHI, T., AND AIKEN, A. Flow-sensitive type qualifiers. In *Proc. ACM Conf. on Programming Language Design and Implementation* (Berlin, June 2002), pp. 1–12.

[17] GUYER, S., MCKINLEY, K., AND FRAMPTON, D. Free-Me: a static analysis for automatic individual object reclamation. *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation* (2006), 364–375.

[18] HALLENBERG, N., ELSMAN, M., AND TOFTE, M. Combining region inference and garbage collection. In *Proc. ACM Conf. on Programming Language Design and Implementation* (Berlin, 2002), pp. 141–152.

[19] HEINE, D. L., AND LAM, M. S. A practical flow-sensitive and context-sensitive c and c++ memory leak detector. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), ACM, pp. 168–181.

[20] LIVSHITS, V. B. Turning Eclipse against itself: Finding bugs in Eclipse code using lightweight static analysis. Eclipsecon '05 Research Exchange, Mar. 2005.

[21] SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. Heap profiling for space-efficient java. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (New York, NY, USA, 2001), ACM Press, pp. 104–113.

[22] SHAHAM, R., YAHAV, E., KOLODNER, E. K., AND SAGIV, S. Establishing local temporal heap safety properties with applications to compile-time memory management. In *SAS* (2003), R. Cousot, Ed., vol. 2694 of *Lecture Notes in Computer Science*, Springer, pp. 483–503.

[23] TOFTE, M., AND TALPIN, J.-P. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proc. ACM Symp. on Principles of Programming Languages* (1996), pp. 188–201.

[24] WEIMER, W., AND NECULA, G. Finding and preventing run-time error handling mistakes. *Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications* (2004), 419–431.