Compositional Pointer and Escape Analysis for Java Programs

John Whaley and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{jwhaley, rinard}@lcs.mit.edu

Abstract

This paper presents a combined pointer and escape analysis algorithm for Java programs. The algorithm is based on the abstraction of points-to escape graphs, which characterize how local variables and fields in objects refer to other objects. Each points-to escape graph also contains escape information, which characterizes how objects allocated in one region of the program can escape to be accessed by another region. The algorithm is designed to analyze arbitrary regions of complete or incomplete programs, obtaining complete information for objects that do not escape the analyzed regions.

We have developed an implementation that uses the escape information to eliminate synchronization for objects that are accessed by only one thread and to allocate objects on the stack instead of in the heap. Our experimental results are encouraging. We were able to analyze programs tens of thousands of lines long. For our benchmark programs, our algorithms enable the elimination of between 24% and 67% of the synchronization operations. They also enable the stack allocation of between 22% and 95% of the objects.

1 Introduction

This paper presents a combined pointer and escape analysis algorithm for Java programs and programs written in similar object-oriented languages. The algorithm is based on the abstraction of points-to escape graphs, which characterize how local variables and fields in objects refer to other objects. Each points-to escape graph also contains escape information, which characterizes how objects allocated in one region of the program can escape to be accessed by another region.

A key concept underlying this abstraction is the goal of representing interactions between analyzed and unanalyzed regions of the program. Points-to escape graphs make a clean distinction between objects and references created within an analyzed region and those created in the rest of the program. They therefore enable a flexible analysis that is capable of analyzing arbitrary parts of the program, with the analysis result becoming more precise as more of the program is analyzed. At every stage in the analysis, the algorithm can distinguish where it does and does not have complete information.

1.1 Analysis Uses

Java presents a clean and simple memory model: conceptually, all objects are allocated in a garbage-collected heap. While useful to the programmer, this model comes with a cost. In many cases it would be more efficient to allocate objects on the stack, eliminating the dynamic memory management overhead for that object. A similar situation holds for the Java synchronization model. Conceptually, every Java object comes with a lock. Each synchronized method ensures that it executes atomically by acquiring and releasing the lock in its receiver object. But the lock overhead is wasted when only one thread accesses the object — the locks are required only when there is a possibility that multiple threads may attempt to access the object simultaneously.

In this paper we discuss the use of our analysis results to eliminate unnecessary synchronization and to enable stack allocation of objects. The basic idea is to use the analysis information to determine when objects do not escape threads and methods. If an object does not escape from its allocating thread to another thread, the compiler can transform the program to eliminate synchronization operations on that object. If an object does not escape a method, the compiler can transform the program to allocate it in that method's activation record instead of in the heap. Our experimental results show that the algorithms can eliminate a significant number of heap allocations and synchronization operations.

1.2 Analysis Properties

Our analysis has several important properties:

- It is an interprocedural analysis. It is designed to combine analysis results from multiple methods to obtain precise points-to and escape information.
- It is a compositional analysis. It is designed to analyze each method once to produce a single parameterized analysis result that can be specialized for use at all of the call sites that may invoke the method.[1]

[1] Recursive methods require an iterative algorithm that may analyze methods multiple times to reach a fixed point.

- It is a partial program analysis in two senses. First, it is designed to analyze each method independently of its callers. Second, it is capable of analyzing a method without analyzing all of the methods that it invokes, with the analysis result becoming more precise as more of the invoked methods are analyzed.

Because the analysis is compositional and can analyze pieces of the program independently of their callers and callees, it is especially appropriate for dynamically loaded programs. Our current implementation runs in a dynamic compiler and analyzes libraries independently of applications. When the compiler loads an application, it retrieves the previously computed analysis results for the libraries and uses these results in the analysis of the application.

1.3 Basic Approach

The analysis is based on an abstraction we call points-to escape graphs. The nodes of this abstraction represent objects; the edges represent references between objects. The abstraction also contains information about which objects escape to other methods or threads. For example, an object escapes if it is returned to an unanalyzed region of the program or passed as a parameter to an unanalyzed method.

For each method, the analysis produces a points-to escape graph that characterizes the points-to relationships created within the method and the interaction of the method with the points-to relationships created by the rest of the program. The analysis represents these interactions, in part, by maintaining a distinction between two kinds of edges: *inside edges*, which represent references created inside the currently analyzed region, and *outside edges*, which represent references created outside the currently analyzed region.

Similarly, there are two kinds of nodes: *inside nodes*, which represent objects created inside the currently analyzed region and accessed via inside edges, and *outside nodes*, which represent objects that are either created outside the currently analyzed region or accessed via outside edges. Outside edges represent interactions in which the analyzed region reads a reference created in an unanalyzed region. Inside edges from outside nodes or nodes reachable from outside nodes represent interactions in which the analyzed region creates a reference that the unanalyzed region may read.

Edges between inside nodes represent references created within the currently analyzed region. If the inside nodes do not escape, they have no interaction with the rest of the program and their edges in the points-to escape graph completely characterize the points-to relationships between the objects represented by these nodes. Because points-to escape graphs record the interactions of methods with their callers, the analysis can recover complete information about objects that escape the currently analyzed method but are recaptured in the caller.

Our analysis is compositional — it analyzes each method independently of its callers. Unlike all other pointer and escape analysis algorithms that we know of, our algorithm is also capable of analyzing a method independently of methods that it may invoke. When the algorithm skips the analysis of a potentially invoked method, it records that all objects passed as parameters to the invoked (but not analyzed) method escape from the scope of the analysis. This escape information allows the algorithm to clearly identify both those objects for which it has complete points-to information and those objects whose points-to relationships may be affected by unanalyzed regions of the program. Our algorithm is designed to analyze arbitrary regions of complete or incomplete programs, obtaining complete information for objects that do not escape the analyzed regions.

1.4 Contributions

This paper makes the following contributions:

- **Analysis Algorithms:** It presents a new combined pointer and escape analysis algorithm. The algorithm is compositional and is designed to deliver useful information without analyzing the entire program.
- **Analysis Approach:** It presents an analysis approach that explicitly differentiates between references and objects from analyzed and unanalyzed regions of the program. This approach allows the algorithm to capture the interactions between these regions, and to clearly identify where it does and does not have complete information about the extracted relationships.
- **Analysis Uses:** It presents two optimizations enabled by the extracted points-to and escape information: synchronization elimination and stack allocation.
- **Experimental Results:** It presents experimental results from a prototype implementation of the algorithms. These results show that the algorithms can eliminate a significant amount of synchronization operations and heap allocations.

The remainder of the paper is organized as follows. Section 2 presents examples that illustrate how the analysis works. Section 3 describes how the analysis represents the program and the analysis objects that the algorithm uses. Section 4 presents the intraprocedural analysis, and Section 5 presents the interprocedural analysis. In Section 6 we present an abstraction relation that characterizes the correspondence between points-to escape graphs and the objects and references that the program creates as it runs. Section 7 presents the synchronization elimination and stack allocation transformations. Section 8 presents the experimental results from our implementation. Section 9 discusses related work; we conclude in Section 10.

2 Example

In this section we present several examples that illustrate how our analysis works.

2.1 Return Values

Figure 1 presents a complex number arithmetic example. The `add` method adds two complex numbers, storing the result in a newly allocated complex number object and returning the new object. The `multiply` method operates similarly, but multiplies the numbers instead of adding them. Because each method returns the new object, the new object escapes from the method.

The `multiplyAdd` method multiplies its two arguments, then returns the sum of the product and the receiver. In this case, only the `add` result escapes — the temporary result from `multiply` is inaccessible outside of the `multiplyAdd` method. It is therefore possible to generate a specialized version of the `multiply` method that expects the storage for the result object to be allocated on the stack of its caller. It would then generate the result into the stack-allocated

```
class complex {
  double x,y;
  complex(double a, double b) { x = a; y = b; }
  complex multiply(complex a) {
    complex product =
      new complex(x*a.x - y*a.y,x*a.y + y*a.x);
    return(product);
  }
  complex add(complex a) {
    complex sum = new complex(x+a.x,y+a.y);
    return sum;
  }
  complex multiplyAdd(complex a, complex b) {
    complex product = a.multiply(b);
    complex sum = this.add(product);
    return(sum);
  }
}
```

Figure 1: Complex Number Example

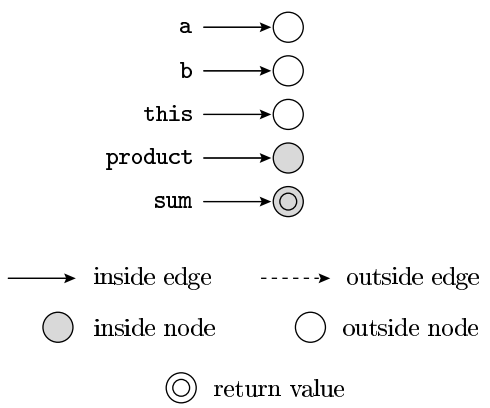

Figure 2: Analysis Result for `multiplyAdd`

object rather than dynamically allocating a new complex object to hold the result.

Figure 2 presents the analysis result for the `multiplyAdd` method. The nodes in this graph represent variables and objects; the edges represent references. The parameter variables `a` and `b` point to nodes that represent the parameter objects; the variable `this` points to a node that represents the receiver. Because these objects are created outside the method, the corresponding nodes are outside nodes. Note that the analysis assumes that the parameters are not aliased. If the analysis result is used at a method invocation site where the parameters are aliased, the mapping algorithm presented in Section 5 merges the two parameter nodes before it uses the analysis result. This mechanism ensures that the algorithm can analyze the program under the assumption that different parameters and object fields are not aliased, but use the analysis results in contexts containing aliases.

The `product` and `sum` variables point to nodes that represent the new complex number objects allocated, respectively, in the `multiply` and `add` methods. The algorithm represents objects allocated within the analyzed region of

```
public class Server extends Thread {
  ServerSocket serverSocket;
  int duplicateConnections;

  Server(ServerSocket s) {
    serverSocket = s;
    duplicateConnections = 0;
  }
  public void run () {
    try {
      Vector connectorList = new Vector();
      while (true) {
        Socket clientSocket =
          serverSocket.accept();
        new ServerHelper(clientSocket).start();
        InetAddress addr =
          clientSocket.getInetAddress();
        if (connectorList.indexOf(addr) < 0)
          connectorList.addElement(addr);
        else duplicateConnections++;
      }
    } catch (IOException e) { }
  }
}
```

Figure 3: Server Example

the program with nodes that correspond to the object creation site. So, for example, `product` points to the node that represents objects created at the object creation site inside `multiply`. Because this node represents objects created inside the analyzed region, it is an inside node.

The node that `product` points to is accessible only via the local variable `product`. We therefore say that this node is *captured*. As soon as the `multiplyAdd` method returns, all of the objects that this node represents will be inaccessible. It is therefore legal to tie the lifetime of the object to the lifetime of the method invocation, and allocate the object on the activation record of the `multiplyAdd` method.

The `multiplyAdd` method returns the object that `sum` points to. Even though the object escapes this method, it may be recaptured by a method above `multiplyAdd` in the call chain. The analysis maintains enough information to recognize such situations.

2.2 Thread-Private Objects

The next example illustrates how the analysis can detect *thread-private* objects, or objects that are accessed by only one thread. The goal is to eliminate synchronization operations on thread-private objects. The code in Figure 3 implements a simple server. The server waits for incoming connection requests on the `serverSocket`. Whenever it gets a request, it creates a new `ServerHelper` thread to service the new connection. The server counts the number of duplicate connections — i.e., the number of times it connects to a client that it has previously connected to.

To correctly compute this count, the server maintains a list of the Internet addresses of clients that it has connected to. Whenever it receives a new request, the server checks the list to determine if it has previously connected to the client. If so, it increments `duplicateConnections`, which is the count of the number of duplicate connections that the server has established. If not, it inserts the Internet address

of the client into the list so that it can detect any future duplicate connections from this client.

In this example, the server uses a `Vector` to hold the list of Internet addresses. This class is *thread safe* — its methods are guaranteed to execute atomically even when concurrently invoked by different threads. Thread safety is an important property. Without it, multithreaded programs can fail in very subtle ways because of *data races*, or unanticipated interactions between threads that concurrently access the same object.

The problem with atomic operations is the overhead from the synchronization operations that make the methods execute atomically. In our example, the standard implementations of `indexOf` and `addElement` acquire and release the lock in their receiver object. This overhead is especially regrettable when, as in our example, the `Vector` is a thread-private object.

The analysis of the `run` method in our example proceeds as follows. The algorithm first analyzes the invoked methods to obtain a points-to escape graph for each method. Each graph summarizes how the method affects the points-to relationships between objects and how the objects it creates and accesses escape to other threads or to the caller. In our example, the new `clientSocket` object escapes to the `ServerHelper` thread. The `indexOf` and `addElement` methods may change the objects that the receiver points to, but they do not change the escape status of the receiver.

Based on this information, and on its analysis of the `run` method, the analysis determines that the `connectorList` object is captured within the `run` method, and is accessible only to the `run` method's thread. The synchronization in the `addElement` and `indexOf` methods is therefore redundant. The compiler can generate specialized, synchronization-free versions of these methods. The generated code for the `run` method invokes these specialized versions, and the computation executes without synchronization overhead for the `Vector` object.

It is also possible to address the problem of synchronization overhead for thread-private objects by providing the programmer with two implementations of each class: a thread-safe implementation with synchronization (used for objects shared between threads), and a thread-unsafe implementation with no synchronization (used for thread-private objects). The JDK 1.2 collections API takes this approach. The problem with this approach is that it complicates the API and burdens the programmer with the responsibility of determining which objects are thread safe and which are not. The last problem can become especially severe for programmers maintaining multithreaded programs. If a change makes a previously thread-private object accessible to multiple threads, the programmer must search the program to manually replace the thread-unsafe version with the thread-safe version.

2.3 Recursive Data Structures

We next present an example that illustrates how our algorithm deals with recursive data structures. Let us assume that in the previous example, the server would like to count the number of connections from each client, instead of simply counting the number of duplicate connections. In this case, the server could use the `multiset` class in Figure 4, which implements a multiset as a list of elements. Each element has a count of the number of times it is present in the multiset.

If the server used a `multiset` instead of a `Vector`, the

```
class multisetElement {
  Object element;
  int count;
  multisetElement next;

  multisetElement(Object e, multisetElement n) {
    count = 1;
    element = e;
    next = n;
  }
  synchronized boolean check(Object e) {
    if (element.equals(e)) {
      count++;
      return(true);
    } else return false;
  }
  synchronized multisetElement insert(Object e) {
    multisetElement m = this;
    while (m != null) {
      if (m.check(e)) return this;
      m = m.next;
    }
    return new multisetElement(e, this);
  }
}

class multiset {
  multisetElement elements;
  multiset() {
    elements = null;
  }
  synchronized void addElement(Object e) {
    if (elements == null)
      elements = new multisetElement(e,null);
    else elements = elements.insert(e);
  }
}
```

Figure 4: Multiset Example

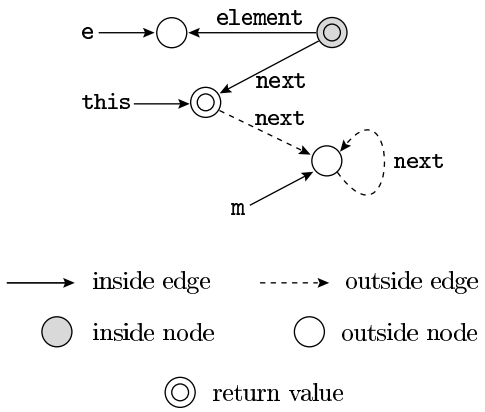

Figure 5: Analysis Result for `insert`

analysis of the `run` method in Figure 3 would still recognize the `multiset` object as thread-private. In this section, we focus on the treatment of recursive data structures in the analysis of the `insert` method for `multisetElements`.

Figure 5 presents the analysis result at the end of the `insert` method. The `this` variable points to the receiver node, which represents the receiver object, while `e` points to a parameter node, which represents the parameter object. Both nodes are outside nodes. The other nodes represent objects created or accessed during the execution of `insert`. The `next` edge from the receiver node points to a node that represents all objects referenced by the `m.next` field at the load statement `m = m.next`, which loads the reference from `m`'s `next` field back into `m`. The edge is an outside edge, because it points to a node that represents an object allocated outside the scope of the analysis of `insert`, and it points to an outside node. In particular, it points to a *load node*, because there is one of these nodes for each load statement in the program.

There is also an outside edge from the load node back to itself. As the loop in the `insert` method walks down the list, it traverses the references in the `next` fields of the nodes. Because all of these references are loaded at the same statement, the outside edges that represent the references point to the same outside node, and there is cycle in the graph. This mechanism ensures that the analysis terminates for programs that manipulate recursive data structures.

The remaining node represents the new `multisetElement` object that `insert` may allocate to hold the new multiset entry. This node has an inside edge (from the `next` field) to the receiver node, and an inside edge (from the `element` field) to the parameter node. These edges represent references to these nodes created during the execution of `insert`.

3 Analysis Algorithm

The algorithm analyzes the program at the granularity of methods. The analysis of each method incorporates information from the method and from some subset of the methods that it invokes. The combination of the currently analyzed method and all of the analyzed methods that it invokes is called the *current analysis scope*.

3.1 Program Representation

The algorithm represents the program using the following analysis objects. There is a set $\mathtt{l} \in \mathtt{L}$ of local variables and a set $\mathtt{p} \in \mathtt{P}$ of formal parameter variables. There is one formal parameter variable for each formal parameter of each method in the program. Together, the local and formal parameter variables make up the set $\mathtt{v} \in \mathtt{V} = \mathtt{L} \cup \mathtt{P}$ of variables.

There is also a set $\mathtt{cl} \in \mathtt{CL}$ of classes and a set $\mathtt{op} \in \mathtt{OP}$ of methods. Each method has a receiver class `cl` and a formal parameter list $\mathtt{p}_0, \ldots, \mathtt{p}_k$. We adopt the convention that parameter $\mathtt{p}_0$ points to the receiver object of the method.

Finally, there is a set $\mathtt{f} \in \mathtt{F}$ of object fields. Object fields are accessed using syntax of the form `v.f`. Static class variables are accessed using syntax of the form `cl.f`.

The algorithm represents the computation of each method using a control flow graph. The nodes of control flow graphs are statements $\mathtt{st} \in \mathtt{ST}$. The algorithm analyzes only those statements that affect the points-to and escape information. We assume the program has been preprocessed so that all statements relevant to the analysis are in one of the following forms:

- A copy statement $\mathtt{l} = \mathtt{v}$.
- A load statement $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$.
- A store statement $\mathtt{l}_1.\mathtt{f} = \mathtt{l}_2$.
- A global load statement $\mathtt{l} = \mathtt{cl}.\mathtt{f}$.
- A global store statement $\mathtt{cl}.\mathtt{f} = \mathtt{l}$.
- A return statement `return l`, which identifies the return value `l` of the method.
- An object creation site of the form $\mathtt{l} =$ `new cl`. There are two kinds of object creation sites:
  - If `cl` inherits from the class `Thread`, then the object creation site is also a thread creation site.
  - If `cl` does not inherit from the class `Thread`, then the object creation site is not a thread creation site.
- A method invocation site of the form $\mathtt{l} = \mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$. Each method invocation site corresponds to a method invocation site $m \in M$.

The control flow graph for each method `op` starts with the enter statement $\mathtt{enter}_{\mathtt{op}}$ and ends with an exit statement $\mathtt{exit}_{\mathtt{op}}$.

The analysis represents the control flow relationships between statements as follows: pred(`st`) is the set of statements that may execute immediately before `st`, and succ(`st`) is the set of statements that may execute immediately after `st`. There are two program points for each statement `st`, the program point $\bullet\mathtt{st}$ immediately before `st` executes, and the program point $\mathtt{st}\bullet$ immediately after `st` executes.

The interprocedural analysis uses call graph information to compute sets of methods that may be invoked at method invocation sites. For each method invocation site m, callees(m) is the set of methods that m may invoke. Given a method `op`, callers(`op`) is the set of method invocation sites that may invoke `op`. The current implementation obtains this call graph information using a variant of class hierarchy analysis [13], but the algorithm can use any conservative approximation to the actual call graph generated when the program runs.

3.2 Object Representation

The analysis represents the objects that the program manipulates using a set $n \in N$ of nodes. There are several kinds of nodes:

- There is the set N_I of inside nodes, which represent objects created within the current analysis scope and accessed via references created within the current analysis scope. There is one inside node for each object creation site; that inside node represents all objects created within the current analysis scope at that site.

  N_T is the set of thread nodes, or inside nodes that correspond to thread creation sites. Since each thread corresponds to an object that inherits from the class `Thread`, thread creation sites are also object creation sites, and $N_T \subseteq N_I$.
- There is the set N_O of outside nodes, which represent objects created outside the current analysis scope or accessed via references created outside the current analysis scope.

- There is the set CL of class nodes. Conceptually, each class node represents a statically allocated object whose fields are the static class variables for the corresponding class.

The set N_O of outside nodes is futher divided into the following kinds of nodes:

- There is a set N_P of parameter nodes. There is one parameter node for each formal parameter in the program. Each parameter node represents the object that its parameter points to during the execution of the analyzed method. When the analysis starts, each of the method's formal parameter variables points to its corresponding parameter node. The receiver object is treated as the first parameter of each method.

- There is a set N_G of global nodes. There is one global node for each static class variable `cl.f` in the program. Each global node represents the objects that its static class variable may point to during the execution of the current method. When the analysis of the current method starts, each accessed static class variable points to its global node.

- There is a set N_L of load nodes. There is one load node for each load statement in the program. When the load statement executes, it loads a value from a field in an object. If the loaded value is a reference, the analysis must represent the object that the reference points to. Each load node represents all of the objects whose references are loaded by the corresponding load statement.

- There is a set N_R of return nodes. There is one return node for each method invocation site in the program. Return nodes are used to represent the return values of invocations of unanalyzed methods.

The analysis represents each array with a single node. This node has a field `elements`, which represents all of the elements of the array. Because the points-to information for all of the array elements is merged into this field, the analysis does not make a distinction between different elements of the same array.

3.3 Points-To Escape Graphs

A points-to escape graph is a quadruple of the form $\langle O, I, e, r\rangle$, where

- $O \subseteq (N \times \mathtt{F}) \times (N_L \cup N_G)$ is a set of outside edges. Outside edges represent references created outside the current analysis scope, either by the caller, by a thread running in parallel with the current thread, or by an unanalyzed invoked method.

- $I \subseteq (\mathtt{V} \cup (N \times \mathtt{F})) \times N$ is a set of inside edges. Inside edges represent references created inside the current analysis scope.

- $e : N \rightarrow 2^M$ is an escape function that records the set of unanalyzed method invocation sites that a node escapes down into.

- $r \subseteq N$ is a return set that represents the set of objects that may be returned by the currently analyzed method.

Both O and I are graphs with edges labeled with a field from F. We define the following operations on nodes of the graphs:

$$\begin{aligned}
\text{edgesTo}(O, n) &= \{\langle n', n\rangle \in O\} \cup \{\langle\langle n', \mathtt{f}\rangle, n\rangle \in O\} \\
\text{edgesFrom}(O, n) &= \{\langle n, n'\rangle \in O\} \cup \{\langle\langle n, \mathtt{f}\rangle, n'\rangle \in O\} \\
\text{edges}(O, n) &= \text{edgesTo}(O, n) \cup \text{edgesFrom}(O, n) \\
O(n) &= \{n'.\langle n, n'\rangle \in O\} \\
O(n, \mathtt{f}) &= \{n'.\langle\langle n, \mathtt{f}\rangle, n'\rangle \in O\}
\end{aligned}$$

The following operation removes a set S of nodes from a points-to escape graph. $\langle O', I', e', r'\rangle = \text{remove}(S, \langle O, I, e, r\rangle)$, where

$$\begin{aligned}
O' &= O - \cup\{\text{edges}(O, n).n \in S\} \\
I' &= I - \cup\{\text{edges}(I, n).n \in S\} \\
e'(n) &= \begin{cases} e(n) & \text{if } n \notin S \\ \emptyset & \text{otherwise} \end{cases} \\
r' &= r - S
\end{aligned}$$

3.4 Reachability and Escaped Nodes

We identify two different kinds of escape information, with the distinction based on when an object becomes accessible outside the current method. If an object was created outside the current analysis scope or was accessed via a reference created outside the current analysis scope, the object is already accessible to other parts of the program. In this case we say that the object *has escaped* or *is escaped*.

If an object has not escaped but will be returned by the method to its caller, we say that the object *will escape*.

An object has directly escaped the current method in all of the following situations:

- A reference to the object was passed as a parameter into the current method.

- A reference to the object was written into a static class variable.

- A reference to the object was passed as a parameter to an invoked method, and there is no information about what the invoked method did with the object.

- The object is a thread object.

An object has escaped if it is reachable via some sequence of object references from a directly escaped object. An object is *captured* if it has not escaped.

Given our abstraction of points-to escape graphs, we can formalize the concepts of reachability and escaped nodes as follows. A node n' is directly reachable from a node n in a graph O if $n = n'$, $\langle n, n'\rangle \in O$ or $\exists \mathtt{f}.\langle\langle n, \mathtt{f}\rangle, n'\rangle \in O$. A node n' is reachable from a node n in O if $n = n'$ or there exists a sequence $n_0 \circ \cdots \circ n_k$ such that $n = n_0$, $n' = n_k$, and for all $0 \leq i < k$, n_{i+1} is directly reachable from n_i in O. Finally, a node n' is reachable from a set of nodes S in O if there exists a node $n \in S$ such that n' is reachable from n in O. We define reachable(O, S, n) if n is reachable from S in O.

Given a points-to escape graph $\langle O, I, e, r\rangle$, a node n directly escapes if $n \in N_P \cup N_R \cup N_T \cup \mathtt{CL}$ or $e(n) \neq \emptyset$. We define escaped$(\langle O, I, e, r\rangle, n)$ if n is reachable in $O \cup I$ from a node that directly escapes, and captured$(\langle O, I, e, r\rangle, n)$ if not escaped$(\langle O, I, e, r\rangle, n)$.

4 Intraprocedural Analysis

The algorithm uses a dataflow analysis to generate a points-to escape graph at each point in the method. The analysis of each method starts with the construction of the points-to escape graph $\langle O_0, I_0, e_0, r_0 \rangle$ for the first statement in the method. Given a method `op` with formal parameter list $\mathtt{p}_0, \ldots, \mathtt{p}_k$ and accessed static class variables $\mathtt{cl}_1.\mathtt{f}_1, \ldots, \mathtt{cl}_j.\mathtt{f}_j$, the initial points-to escape graph $\langle O_0, I_0, e_0, r_0 \rangle$ is defined as follows.

- In I_0, each parameter $\mathtt{p}_i$ points to its corresponding parameter node $n_{\mathtt{p}_i}$. The formal parameter $\mathtt{p}_0$ points to the receiver object of the method; the node $n_{\mathtt{p}_0}$ represents the receiver object in the analysis of the method.
$$I_0 = \{\langle \mathtt{p}_i, n_{\mathtt{p}_i} \rangle . 0 \leq i \leq k\}$$
- In O_0, each accessed static class variable $\mathtt{cl}_i.\mathtt{f}_i$ points to its corresponding global node $n_{\mathtt{cl}_i.\mathtt{f}_i}$.
$$O_0 = \{\langle \langle \mathtt{cl}_i, \mathtt{f}_i \rangle, n_{\mathtt{cl}_i.\mathtt{f}_i} \rangle . 1 \leq i \leq j\}$$
- For all n, $e_0(n) = \emptyset$.
- $r_0 = \emptyset$.

Note that the algorithm analyzes the method under the assumption that the parameters and accessed static class variables all point to different objects. If the method may be invoked in a calling context in which some of these pointers point to the same object, this object will be represented by multiple nodes during the analysis of the method. In this case, the mapping operation described below in Section 5 will merge the corresponding outside objects when it maps the final analysis result for the method into the calling context at the method invocation site. Because the mapping retains all of the edges from the merged objects, it conservatively models the actual effect of the method.

Once it has finished constructing the initial points-to escape graph, the analysis continues by propagating points-to escape graphs through the statements of the method's control flow graph. The transfer function $\langle O', I', e', r' \rangle = [\![\mathtt{st}]\!](\langle O, I, e, r \rangle)$ defines the effect of each statement `st` on the current points-to escape graph. Most of the statements first kill a set of inside edges, then generate additional inside and outside edges. In this case, the transfer function has the following general form:

$$\begin{aligned} I' &= (I - \mathrm{Kill}_I) \cup \mathrm{Gen}_I \\ O' &= O \cup \mathrm{Gen}_O \end{aligned}$$

Figure 6 graphically presents the rules that determine the sets of generated edges for the different kinds of statements. Each row in this figure contains four items: a statement, a graphical representation of existing edges, a graphical representation of the existing edges plus the new edges that the statement generates, and a set of side conditions. The interpretation of each row is that whenever the points-to escape graph contains the existing edges and the side conditions are satisfied, the transfer function for the statement generates the new edges.

4.1 Copy Statements

A copy statement of the form `l = v` makes `l` point to the object that `v` points to. The transfer function updates I to reflect this change by killing the current set of edges from `l`, then generating additional inside edges from `l` to all of the nodes that `v` points to.

$$\begin{aligned} \mathrm{Kill}_I &= \mathrm{edges}(I, \mathtt{l}) \\ \mathrm{Gen}_I &= \{\mathtt{l}\} \times I(\mathtt{v}) \\ I' &= (I - \mathrm{Kill}_I) \cup \mathrm{Gen}_I \end{aligned}$$

4.2 Load Statements

A load statement of the form $\mathtt{l}_1 = \mathtt{l}_2.\mathtt{f}$ makes $\mathtt{l}_1$ point to the object that $\mathtt{l}_2.\mathtt{f}$ points to. The analysis models this change by constructing a set S of nodes that represent all of the objects to which $\mathtt{l}_2.\mathtt{f}$ may point, then generating additional inside edges from $\mathtt{l}_1$ to every node in this set.

All nodes accessible via inside edges from $\mathtt{l}_2.\mathtt{f}$ should clearly be in S. But if $\mathtt{l}_2$ points to an escaped node, other parts of the program such as the caller or threads executing in parallel with the current thread can access the referenced object and store values in its fields. In particular, the value in $\mathtt{l}_2.\mathtt{f}$ may have been written by the caller or a thread running in parallel with the current thread — in other words, $\mathtt{l}_2.\mathtt{f}$ may contain a reference created outside of the current analysis scope. The analysis uses an outside edge to model this reference. The outside edge points to the load node for the load statement, which is the outside node that represents the objects that the reference may point to.

The analysis must therefore consider two cases: the case when $\mathtt{l}_2$ does not point to an escaped node, and the case when $\mathtt{l}_2$ does point to an escaped node. The algorithm determines which case applies by computing S_E, the set of escaped nodes to which $\mathtt{l}_2$ points. S_I is the set of nodes accessible via inside edges from $\mathtt{l}_2.\mathtt{f}$.

$$\begin{aligned} S_E &= \{n_2 \in I(\mathtt{l}_2).\mathrm{escaped}(\langle O, I, e, r \rangle, n_2)\} \\ S_I &= \cup\{I(n_2, \mathtt{f}).n_2 \in I(\mathtt{l}_2)\} \end{aligned}$$

If $S_E = \emptyset$ (i.e., $\mathtt{l}_2$ does not point to an escaped node), $S = S_I$ and the transfer function simply kills all edges from $\mathtt{l}_1$, then generates inside edges from $\mathtt{l}_1$ to all of the nodes in S.

$$\begin{aligned} \mathrm{Kill}_I &= \mathrm{edges}(I, \mathtt{l}_1) \\ \mathrm{Gen}_I &= \{\mathtt{l}_1\} \times S \\ I' &= (I - \mathrm{Kill}_I) \cup \mathrm{Gen}_I \end{aligned}$$

If $S_E \neq \emptyset$ (i.e., $\mathtt{l}_2$ points to at least one escaped node), $S = S_I \cup \{n\}$, where n is the load node for the load statement. In addition to killing all edges from $\mathtt{l}_1$, then generating inside edges from $\mathtt{l}_1$ to all of the nodes in S, the transfer function also generates outside edges from the escaped nodes to n.

$$\begin{aligned} \mathrm{Kill}_I &= \mathrm{edges}(I, \mathtt{l}_1) \\ \mathrm{Gen}_I &= \{\mathtt{l}_1\} \times S \\ I' &= (I - \mathrm{Kill}_I) \cup \mathrm{Gen}_I \\ \mathrm{Gen}_O &= (S_E \times \{\mathtt{f}\}) \times \{n\} \\ O' &= O \cup \mathrm{Gen}_O \end{aligned}$$

4.3 Store Statements

A store statement of the form $\mathtt{l}_1.\mathtt{f} = \mathtt{l}_2$ finds the object to which $\mathtt{l}_1$ points, then makes the `f` field of this object point to same object as $\mathtt{l}_2$. The analysis models the effect of this assignment by finding the set of nodes that $\mathtt{l}_1$ points to, then generating inside edges from all of these nodes to the nodes that $\mathtt{l}_2$ points to.

$$\begin{aligned} \mathrm{Gen}_I &= (I(\mathtt{l}_1) \times \{\mathtt{f}\}) \times I(\mathtt{l}_2) \\ I' &= I \cup \mathrm{Gen}_I \end{aligned}$$

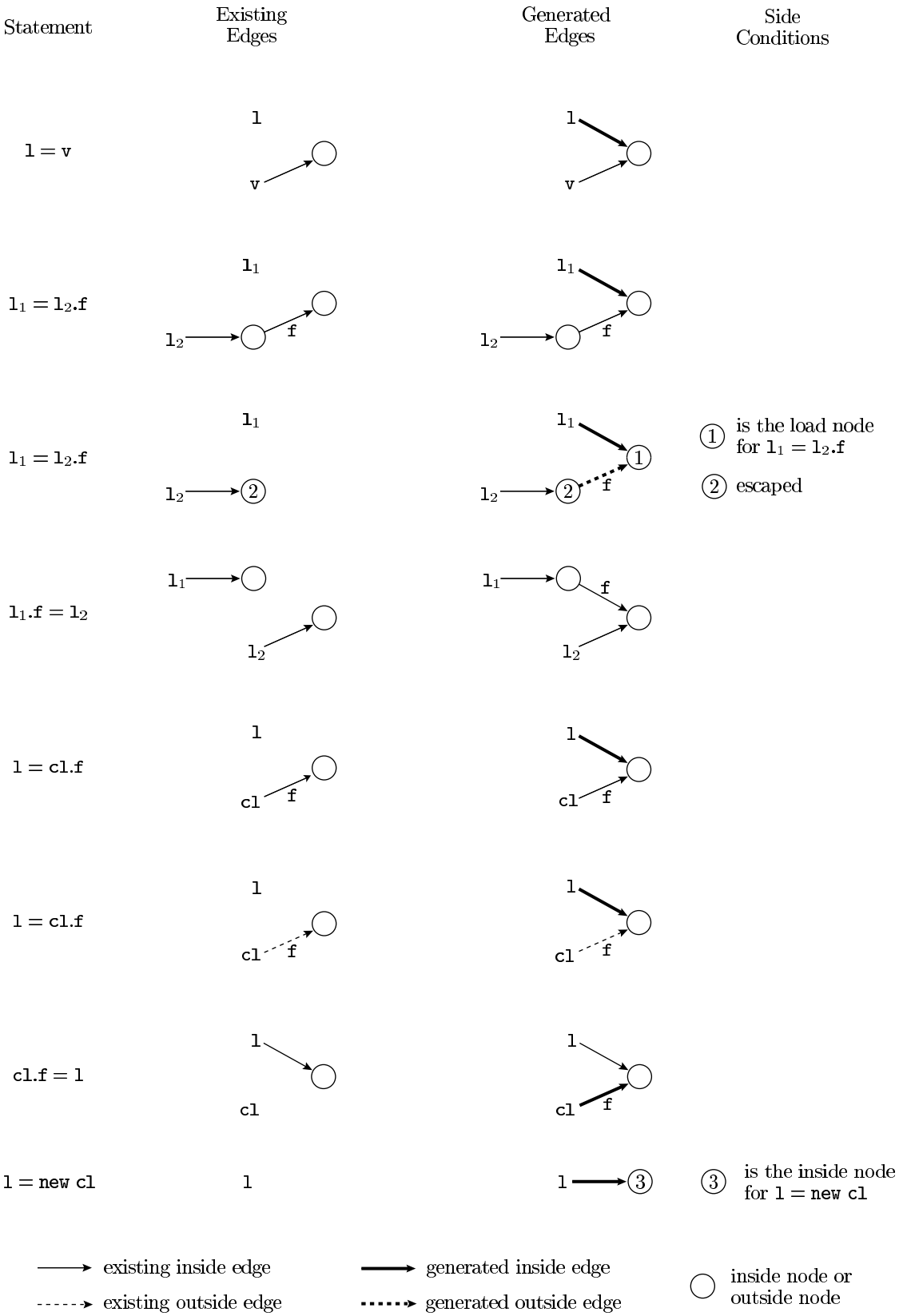

Figure 6: Generated Edges for Basic Statements

4.4 Global Load Statements

A global load statement of the form `l = cl.f` makes `l` point to the same object that `cl.f` points to. The analysis models this change by killing all edges from `l` and generating inside edges from `l` to all of the nodes that `cl.f` points to.

$$\begin{aligned} \text{Kill}_I &= \text{edges}(I, \texttt{l}) \\ \text{Gen}_I &= \{\texttt{l}\} \times (O \cup I)(\texttt{cl}, \texttt{f}) \\ I' &= (I - \text{Kill}_I) \cup \text{Gen}_I \end{aligned}$$

4.5 Global Store Statements

A global store statement of the form `cl.f = l` makes the static class variable `cl.f` point to the same object as `l`. The analysis models this change by generating inside edges from `cl.f` to all of the nodes that `l` points to.

$$\begin{aligned} \text{Gen}_I &= \{\langle \texttt{cl}, \texttt{f} \rangle\} \times I(\texttt{l}) \\ I' &= I \cup \text{Gen}_I \end{aligned}$$

4.6 Object Creation Sites

An object creation site of the form `l = new cl` allocates a new object and makes `l` point to the object. The analysis represents all objects allocated at a specific creation site with the creation site's inside node n. The transfer function models the effect of the statement by killing all edges from `l`, then generating an inside edge from `l` to n.

$$\begin{aligned} \text{Kill}_I &= \text{edges}(I, \texttt{l}) \\ \text{Gen}_I &= \{\langle \texttt{l}, n \rangle\} \\ I' &= (I - \text{Kill}_I) \cup \text{Gen}_I \end{aligned}$$

4.7 Return Statements

A return statement `return l` specifies the return value for the method. The immediate successor of each return statement is the exit statement of the method. The analysis models the effect of the return statement by updating r to include all of the nodes that `l` points to.

$$r' = I(\texttt{l})$$

4.8 Exit Statements

The transfer function for an exit statement $\texttt{exit}_{\texttt{op}}$ produces the final analysis result for the method. This result is used, at all call sites that may invoke the method, to compute the effect of the method on the analysis of its caller. The primary activity of the transfer function is to remove information from the points-to escape graph that should not be visible to the caller. The algorithm first computes the set S of nodes that are not reachable from the static class variables, the parameters, or the return values. It then removes these nodes from the points-to escape graph.

$$\begin{aligned} &S = \{n \in N. \text{not reachable}(O \cup I, \texttt{CL} \cup \texttt{P} \cup r, n)\} \\ &\langle O', I', e', r' \rangle = \text{remove}(S, \langle O, I, e, r \rangle) \end{aligned}$$

The analysis uses the reachability information at method exit points to bound object lifetimes. If an inside node is not reachable from the static class variables, the parameters, the return values, or a thread object, then all of the objects it represents (i.e., all of the objects allocated at the corresponding object allocation site within the current analysis scope) are inaccessible both to the method's caller and to other threads. When the method returns, these objects are inaccessible to the rest of the computation. It is therefore legal to allocate these objects in the activation record of the method and to access the objects without synchronization.

4.9 Control-Flow Join Points

To analyze a statement, the algorithm first computes the join of the points-to escape graphs flowing into the statement from all of its predecessors. It then applies the transfer function to obtain a new points-to escape graph at the point after the statement. The join operation is defined as follows. $\langle O_1, I_1, e_1, r_1 \rangle \sqcup \langle O_2, I_2, e_2, r_2 \rangle = \langle O, I, e, r \rangle$, where $O = O_1 \cup O_2$, $I = I_1 \cup I_2$, $\forall n \in N. e(n) = e_1(n) \cup e_2(n)$, and $r = r_1 \cup r_2$. The corresponding partial order for points-to escape graphs is $\langle O_1, I_1, e_1, r_1 \rangle \sqsubseteq \langle O_2, I_2, e_2, r_2 \rangle$ if $O_1 \subseteq O_2$, $I_1 \subseteq I_2$, $\forall n \in N. e_1(n) \subseteq e_2(n)$, and $r_1 \subseteq r_2$. The bottom points-to escape graph is $\langle \emptyset, \emptyset, e_\perp, \emptyset \rangle$, where $e_\perp(n) = \emptyset$ for all n.

The analysis of each method produces analysis results $\alpha(\bullet\texttt{st})$ and $\alpha(\texttt{st}\bullet)$ before and after each statement `st` in the method's control flow graph. The analysis result α satisfies the following equations:

$$\begin{aligned} \alpha(\bullet\texttt{enter}_{\texttt{op}}) &= \langle O_0, I_0, e_0, r_0 \rangle \\ \alpha(\bullet\texttt{st}) &= \sqcup\{\alpha(\texttt{st}'\bullet).\texttt{st}' \in \text{pred}(\texttt{st})\} \\ \alpha(\texttt{st}\bullet) &= [\![\texttt{st}]\!](\alpha(\bullet\texttt{st})) \end{aligned}$$

The final analysis result of method `op` is the analysis result at the program point after the exit node, i.e., $\alpha(\texttt{exit}_{\texttt{op}}\bullet)$. As described below in Section 5.3, the analysis solves these equations using a standard worklist algorithm.

4.10 Strong Updates

Our analysis models the execution of statements that update memory locations by adding edges to the nodes that represent the updated locations. There are two possible kinds of updates: weak updates, which leave the existing edges in place, and strong updates, which remove the existing edges. Because strong updates leave fewer edges in the points-to escape graph, they may produce more precise analysis results. In the analysis presented so far, updates to local variables are strong and all other updates are weak.

The analysis can legally perform a strong update whenever the updated node is captured and represents exactly one updated memory location when the program runs. For a store statement of the form $\texttt{l}_1.\texttt{f} = \texttt{l}_2$, this condition is satisfied whenever $\texttt{l}_1$ points to a single captured node n, n represents a single object, and `f` represents a single location in that object. The last condition is satisfied unless `f = elements`, the special field identifier for array elements discussed in Section 3.2.

4.10.1 Strong Updates for Singular Nodes and Fields

The node n is *singular* if it represents one object. This is the case, for example, if n is an inside node that corresponds to a statement executed at most once within the current analysis scope. The field `f` is singular if it represents a single location within an object. Given these definitions, we can provide the following definition of I' for the transfer function of a store statement $\texttt{l}_1.\texttt{f} = \texttt{l}_2$. With this definition, the transfer function for store statements performs strong updates.

$$\begin{aligned} \text{Kill}_I &= \begin{cases} \text{edgesFrom}(n, \texttt{f}) & \text{if } I(\texttt{l}_1) = \{n\},\ n, \texttt{f} \text{ singular,} \\ & \quad \text{and captured}(\langle I, O, e, r \rangle, n) \\ \emptyset & \text{otherwise} \end{cases} \\ \text{Gen}_I &= (I(\texttt{l}_1) \times \{\texttt{f}\}) \times I(\texttt{l}_2) \\ I' &= (I - \text{Kill}_I) \cup \text{Gen}_I \end{aligned}$$

4.10.2 Summary Nodes

It is possible to extend the object representation so that each inside node would represent only the last object allocated at the corresponding allocation site. All other nodes allocated at this site would be represented by a summary node. With this approach, the analysis would perform strong updates for all inside nodes, and weak updates only for summary nodes. Similar approaches have been proposed for intraprocedural shape analysis algorithms [24, 12].

4.11 An Optimization for Static Class Variables

In the absence of information about how parallel threads access static class variables, the extracted analysis information imposes no limit on the lifetimes or points-to relationships of objects reachable from these variables. The implemented compiler therefore adopts a more compact representation for the points-to relationships involving nodes that represent these objects. Instead of recording the specific set of static class variables that may point to a node, the analysis simply records that at least one does. This representation reduces the size of the points-to escape graph without reducing the amount of useful information.

5 Interprocedural Analysis

At each method invocation site, the analysis has the option of either skipping the site or analyzing the site. If it analyzes the site, it collects the final analysis results from all of the potentially invoked methods, maps each analysis result into the points-to escape graph from the program point before the method invocation site, then merges the mapped results to derive the points-to escape graph at the point after the method invocation site. If the analysis skips a method invocation site, it marks all of the parameters as escaping down into the site.

5.1 Skipped Method Invocation Sites

The transfer function for a skipped method invocation site is defined as follows. Given a skipped method invocation site m of the form $\mathtt{l} = \mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$ with return node n_R and a current points-to escape graph $\langle O, I, e, r\rangle$, the points-to escape graph $\langle O, I', e, r'\rangle = [\![m]\!](\langle O, I, e, r\rangle)$ after the site is defined as follows:

$$I' = (I - \text{edges}(I, \mathtt{l})) \cup \{\langle \mathtt{l}, n_R\rangle\}$$

$$e'(n) = \begin{cases} e(n) \cup \{m\} & \text{if } \exists 0 \leq i \leq k. n \in I(\mathtt{l}_i) \\ e(n) & \text{otherwise} \end{cases}$$

The return node n_R is an outside node used to represent the return value of the invoked method.

5.2 Analyzed Method Invocation Sites

Given an analyzed method invocation site m of the form $\mathtt{l} = \mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$ and a current points-to escape graph $\langle O, I, e, r\rangle$, the new points-to escape graph $\langle O', I', e', r'\rangle = [\![m]\!](\langle O, I, e, r\rangle)$ after the site is defined as follows:

$$\langle O', I', e', r'\rangle = \sqcup\{\text{map}(m, \langle O, I, e, r\rangle, \mathtt{op}).\mathtt{op} \in \text{callees}(m)\}$$

We next present the mapping algorithm for method invocation sites. We assume a method invocation site m of the form $\mathtt{l} = \mathtt{l}_0.\mathtt{op}(\mathtt{l}_1, \ldots, \mathtt{l}_k)$ and an invoked method $\mathtt{op}$ with formal parameter list $\mathtt{p}_0, \ldots, \mathtt{p}_k$ and accessed static class variables $\mathtt{cl}_1.\mathtt{f}_1, \ldots, \mathtt{cl}_j.\mathtt{f}_j$. There are three points-to escape graphs involved in the algorithm:

- The *old graph* is the points-to escape graph $\langle O, I, e, r\rangle$ at the point before the method invocation site.
- The *incoming graph* is the final analysis result $\langle O_R, I_R, e_R, r_R\rangle = \alpha(\mathtt{exit}_{\mathtt{op}}\bullet)$ for the invoked method.
- The mapping algorithm produces the *new graph* $\langle O_M, I_M, e_M, r_M\rangle = \text{map}(m, \langle O, I, e, r\rangle, \mathtt{op})$.

5.2.1 Overview

Conceptually, the algorithm first builds an initial mapping $\mu : N \to 2^N$ from the nodes of the incoming graph to the nodes of the old graph. For each outside node n in the incoming graph, $\mu(n)$ is the set of nodes from the old graph that n represents during the analysis of the invoked method $\mathtt{op}$. This initial value of μ maps global nodes, parameter nodes, and load nodes to their corresponding nodes in the new graph. It builds the mapping for load nodes by tracing correlated paths in the old graph and the incoming graph. Each path in the old graph consists of a sequence of inside edges; the corresponding path in the incoming graph consists of the sequence of outside edges that represent the corresponding inside edges during the analysis of the method.

The algorithm then builds the new points-to escape graph $\langle O_M, I_M, e_M, r_M\rangle = \text{map}(m, \langle O, I, e, r\rangle, \mathtt{op})$. The algorithm starts by initializing the new graph to the old graph $\langle O, I, e, r\rangle$. It then uses the mapping μ to map nodes and edges from the incoming graph into the new graph. The mapped nodes represent objects allocated or accessed by the invoked method. The mapped edges represent references that the invoked method created or read. During the mapping process, edges are mapped from pairs of nodes in the incoming graph to corresponding pairs of nodes in the new graph. In this case we say that the nodes in the new graph *acquire* the edges from the nodes in the incoming graph.

As part of the mapping process, the algorithm extends μ so that if the algorithm maps a node n from the incoming graph into the new graph, $n \in \mu(n)$. In this case we say that n is *present* in the new graph.

We next describe the roles that the different kinds of nodes in the incoming graph play in the analysis, and discuss how these roles affect each node's presence in the new graph, its effect on the mapping μ, and the edges that are mapped into the new graph.

- $n \in N_P$: If n is a parameter node, it represents the nodes that the corresponding actual parameter points to. These nodes should acquire n's edges, but n itself should not be present in the new graph.

  $\mu(n)$ is the set of nodes in the current graph that the corresponding actual parameter points to.

- $n \in N_G$: If n is a global node, it represents the objects that the corresponding static class variable points to. These nodes should acquire n's edges, and n should be present in the new graph. Note that n is also escaped in the the new graph.

  $\mu(n)$ is the set of nodes in the current graph that the corresponding static class variable points to. Note that $n \in \mu(n)$.

- $n \in N_L$: If n is a load node, all of the nodes that it represents should acquire its edges. If n may represent an object created outside the analysis scope of the caller or an object accessed via a reference created outside

that scope, it and its edges should also be mapped into the new graph.

The basic idea is that n should be present in the new graph only if an escaped node would point to it — there should be no outside edges from captured nodes. The analysis determines if n should be present by examining all of the nodes in the incoming graph that point to n. There are two cases:

- If at least one of these nodes (say n') is present and escaped in the new graph, n should also be present, and there should be an outside edge in the new graph from n' to n.
- If at least one of these nodes is mapped to an escaped node (say n') from the old graph, n should be present, and there should be an outside edge in the new graph from n' to n.

In both cases, n is escaped in the new graph. $\mu(n)$ includes the set of nodes in the old graph that n represented during the analysis of the method. And $n \in \mu(n)$ if n is present in the new graph.

- $n \in N_I$: If n is an inside node, it and its inside edges should be present in the new graph if the objects that it represents are reachable in the caller. The analysis determines if n should be present by examining all of the nodes in the incoming graph that point to it. As for load nodes, there are two cases:
  - If at least one of these nodes (say n') is present in the new graph, n should also be present, and there should be an edge in the new graph from n' to n.
  - If at least one of these nodes represents a node (say n') in the old graph, n should be present, and there should be an edge in the new graph from n' to n.

  A primary difference between load nodes and inside nodes is that load nodes are present in the new graph only if they are escaped in that graph. Inside nodes are present if they are reachable, and can be either escaped or captured in the new graph.

  If an inside node n is present in the new graph, $\mu(n) = \{n\}$. Otherwise $\mu(n) = \emptyset$.
- $n \in N_R$: If n is a return node (these nodes represent return values from invoked but unanalyzed methods), the conditions are the same as for inside nodes.

5.2.2 Constraints for the Mapping Algorithm

Figures 7 and 8 present a formal specification of the constraints that the mapping μ and the new points-to escape graph $\langle O_M, I_M, e_M, r_M \rangle$ must satisfy. The constraints are specified as a set of inference rules that the final mapping and points-to escape graph must satisfy.[2]

Conceptually, many of these inference rules start with an existing mapping and set of edges, then generate additional mappings and edges. Figure 9 graphically presents the generated mappings and edges for some of the crucial rules. Each row in this figure contains four items: an inference rule, a graphical representation of existing edges and mappings, a graphical representation of the existing edges and mappings plus the new edges and mappings that the rule generates, and a set of side conditions. The nodes on the bottom row of each graphical representation are from the incoming graph, while the nodes on the top row are from the new graph. The interpretation of each row is that if the existing edges and mappings are present in the incoming graph and new graph, and the side conditions are true, then the inference rule generates the new edges and mappings.

$$\frac{0 \leq i \leq k, n \in I(\mathtt{l}_i)}{n \in \mu(n_{\mathtt{p}_i})} \quad (1)$$

$$\frac{1 \leq i \leq j}{\mathtt{cl}_i \in \mu(\mathtt{cl}_i)} \quad (2)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_R, \langle\langle n_3, \mathtt{f}\rangle, n_4\rangle \in I, \; n_3 \in \mu(n_1), n_1 \notin N_I}{n_4 \in \mu(n_2)} \quad (3)$$

Figure 7: Initial Rules for μ

$$\begin{array}{cc} O \subseteq O_M & I - \text{edges}(\mathtt{l}) \subseteq I_M \\ e(n) \subseteq e_M(n) & r \subseteq r_M \end{array} \quad (4)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_R}{(\mu(n_1) \times \{\mathtt{f}\}) \times \mu(n_2) \subseteq I_M} \quad (5)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in I_R, n \in \mu(n_1), n_2 \in N_I \cup N_R}{n_2 \in \mu(n_2)} \quad (6)$$

$$\frac{\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \in O_R, n \in \mu(n_1), \; \text{escaped}(\langle O_M, I_M, e_M, r_M\rangle, n)}{\langle\langle n, \mathtt{f}\rangle, n_2\rangle \in O_M, n_2 \in \mu(n_2)} \quad (7)$$

$$\frac{n \in r_R \cap (N_I \cup N_R)}{n \in \mu(n)} \quad (8)$$

$$\frac{e_R(n) \neq \emptyset, n' \in \mu(n)}{m \in e_M(n')} \quad (9)$$

$$\cup\{\mu(n).n \in r_R\} \subseteq I_M(\mathtt{l}) \quad (10)$$

Figure 8: Rules for μ and $\langle O_M, I_M, e_M, r_M \rangle$

5.2.3 Initial Rules

The initial set of contraints, Rules 1 through 3, set up the initial mapping μ so that all outside nodes in the incoming graph are mapped to the nodes in the old graph that they represent during the analysis of the invoked method `op`.

- Rule 1 maps each parameter node $n_{\mathtt{p}_i}$ to the nodes $I(\mathtt{l}_i)$ in the old graph that it represents during the analysis of the method.[3]
- Rule 2, along with Rule 3, ensures that each accessed global node is mapped to the nodes in the old graph

[2] Each inference rule is written in the standard form

$$\frac{c_1, \ldots, c_k}{c'_1, \ldots, c'_j}$$

which imposes the constraint that if all of $c_1, \ldots, c_k$ are true, then all of $c'_1, \ldots, c'_j$ must be true.

[3] Recall that the method invocation site has actual parameters $\mathtt{l}_0, \ldots, \mathtt{l}_k$, the method has formal parameters $\mathtt{p}_0, \ldots, \mathtt{p}_k$, and $n_{\mathtt{p}_i}$ is the parameter node for $\mathtt{p}_i$.

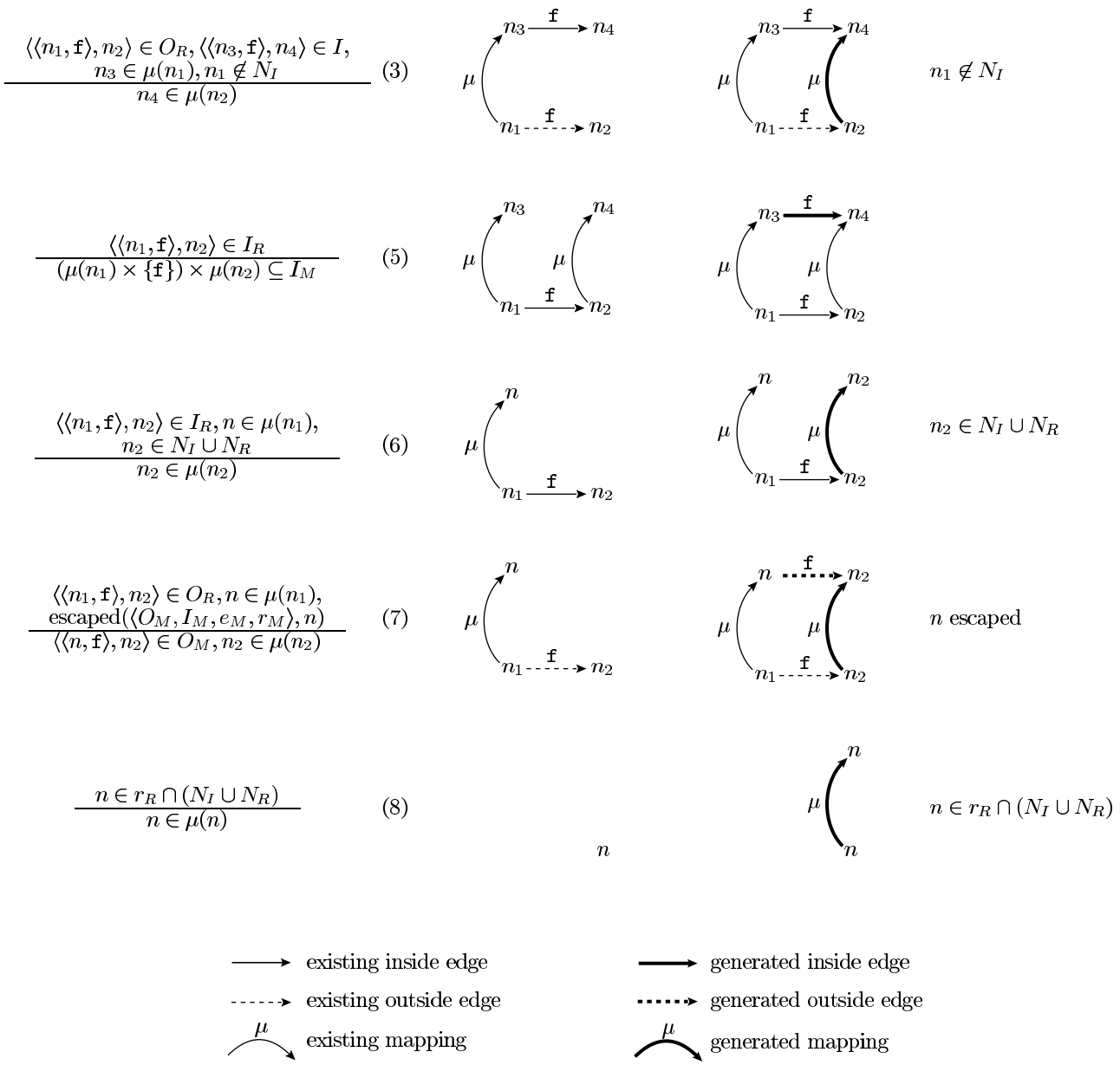

Figure 9: Generated Edges and Mappings for Inference Rules

that it represents during the analysis of the method.[4]

- Rule 3 maps outside nodes to the nodes that they represent during the analysis of the method, matching outside edges in the incoming graph with corresponding inside edges in the old graph. The constraint starts with a node n_1 from the incoming graph that already maps to a node n_3 in the old graph. It then matches edges from these two nodes to find a load node n_2 from the incoming graph that represents a node n_4 in the new graph during the analysis of the method. The constraint maps n_2 to n_4.

Rules 1 through 3 map outside nodes only. Rule 6, however, may map an inside node into the new graph. It is important that Rule 3 does not match outside edges from these inside nodes against inside edges in the old graph. The reason for this restriction is that inside nodes represent objects that were allocated inside the current analysis scope. These objects did not exist when the analyzed method was invoked. Any outside edges from inside nodes therefore represent references created either by threads running in parallel with the current analysis scope, or by unanalyzed methods invoked by the current analysis scope. In either case the reference did not exist when the method was invoked. Because all inside edges in the old graph existed when the analyzed method was invoked, the outside edge in the incoming graph did not represent these edges.

5.2.4 Remaining Rules

Rule 4 initializes the new graph to include the old graph. Rule 5 maps inside edges from the incoming graph into the new graph. There is an inside edge between two nodes in the new graph if there is an inside edge between their two corresponding nodes in the incoming graph.

Rules 6 through 8 map nodes from the incoming graph into the new graph. Rule 6 maps an inside or return node into the new graph if it is reachable from a node that is already mapped into the new graph. Rule 7 maps a load node into the new graph if it is reachable from an escaped node that is already mapped into the new graph. Finally, Rule 8 maps an inside or return node into the new graph if it is returned by the method.

Rule 9 ensures that all nodes passed into unanalyzed methods are marked appropriately in the escape function. Rule 10 makes `l` (the local variable that is assigned to the value that the method returns) point to the nodes that represent the return value of the method.

5.2.5 Constraint Solution Algorithm

Figures 10 and 11 present an algorithm for solving the constraint system in Figures 7 and 8. The algorithm directly reflects the structure of the inference rules. At each step it detects an inference rule antecedent that becomes true, then takes action to ensure that the consequent is also true.

The mapNode(n_1, n) procedure is invoked whenever the algorithm maps a node n_1 from the incoming graph to a node n in the new graph. It first updates the escape function $e_M(n)$ to reflect any new escape information. It then maps new inside edges both to and from n to reflect the corresponding inside edges from the incoming graph. Rule 5 is the only constraint that maps inside edges from the incoming graph into the new graph. Because all insertions of inside edges into the new graph take place inside mapNode, it is responsible for ensuring that the new graph contains the right inside edges.

The control structure of the algorithm is based on three worklists. Each worklist corresponds to one of the rules that map nodes from the incoming graph to the new graph. Each worklist entry contains a node n from the new graph and an edge $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ from the incoming graph. All worklist entries have the property that $n \in \mu(n_1)$.

When the algorithm processes the worklist entry, it checks to see if it should update the mapping for n_2. The details of the check depend on the specific worklist.

- The worklist W_E corresponds to Rule 3, which matches outside edges from the incoming graph against inside edges in the old graph. All edges in this worklist are outside edges. To process an entry from this worklist, the algorithm matches the edge $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ from the worklist against all corresponding inside edges $\langle\langle n, \mathtt{f}\rangle, n_4\rangle$ in the old graph. For each corresponding edge it maps n_2 to n_4.

- The worklist W_I corresponds to Rule 6, which maps inside nodes into the new graph. All edges in this worklist are inside edges. To process an entry from this worklist, the algorithm extracts the node n_2 that the worklist edge points to, and maps n_2 into the new graph.

- The worklist W_O corresponds to Rule 7, which maps outside nodes into the new graph. All edges in this worklist are outside edges. To process an entry from this worklist, the algorithm inserts a corresponding outside edge $\langle\langle n, \mathtt{f}\rangle, n_2\rangle$ into the new graph and maps n_2 into the new graph.

There is a slight complication in the algorithm. As the algorithm executes, it periodically maps inside edges from the incoming graph into the new graph. Whenever a node n_1 is mapped to n, the algorithm maps each inside edge $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ from the incoming graph into the new graph. This mapping process inserts a corresponding inside edge from n to each node that n_2 maps to; i.e., to each node in $\mu(n_2)$. The algorithm must ensure that when it completes, there is one such edge for each node in the final set of nodes $\mu(n_2)$. But when the algorithm first maps $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ into the new graph, $\mu(n_2)$ may not be complete. In this case, the algorithm will eventually map n_2 to more nodes, increasing the set of nodes in $\mu(n_2)$. There should be edges from n to all of the nodes in the final $\mu(n_2)$, not just to those node that were present in $\mu(n_2)$ when the algorithm mapped $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle$ into the new graph.

The algorithm ensures that all of these edges are present in the final graph by building a set $\delta(n_2)$ of delayed actions. Each delayed action consists of a node in the new graph and a field in that node. Whenever the node n_2 is mapped to a new node n' (i.e., the algorithm sets $\mu(n_2) = \mu(n_2) \cup \{n'\}$), the algorithm establishes a new inside edge for each delayed action. The new edge goes from the node in the action to the newly mapped node n'. These edges ensure that the final set of inside edges satisfies the constraints.

5.3 Global Fixed-Point Analysis Algorithm

Figure 12 presents the global fixed-point algorithm that the compiler uses to solve the combined intraprocedural and interprocedural dataflow equations. It maintains a worklist of pending statements. At each step, it removes a statement

[4] Recall that the method accesses static class variables $\mathtt{cl}_0.\mathtt{f}_0, \ldots, \mathtt{cl}_j.\mathtt{f}_j$, and that $n_{\mathtt{cl}_i.\mathtt{f}_i}$ is the global node for $\mathtt{cl}_i.\mathtt{f}_i$.

```
mapNode(n1, n)
    if ⟨n1, n⟩ ∉ D then
        μ(n1) = μ(n1) ∪ {n}
        D = D ∪ {⟨n1, n⟩}
        Update eM(n) to satisfy Rule 9
        if (eR(n1) ≠ ∅) then eM(n) = eM(n) ∪ {m}
        Update IM to satisfy Rule 5
        IM = IM ∪ (δ(n1) × {n})
        for all ⟨⟨n1, f⟩, n2⟩ ∈ IR do
            IM = IM ∪ {⟨n, f⟩} × μ(n2)
            δ(n2) = δ(n2) ∪ {⟨n, f⟩}
        Update worklists for Rules 3,6 and 7
        WE = WE ∪ ({n} × edgesFrom(OR, n1))
        if (n1 = n) then
            WI = WI ∪ ({n} × edgesFrom(IR, n1))
            WO = WO ∪ ({n} × edgesFrom(OR, n1))
```

Figure 10: Procedure for Mapping One Node to Another

```
Initialize worklists
WE = WI = WO = ∅
Initialize new graph to satisfy Rule 4
⟨OM, IM, eM, rM⟩ = ⟨O, I − edges(l), e, r⟩
Map parameter nodes to satisfy Rule 1
for 0 ≤ i ≤ k do
    for all n ∈ I(l_i) do mapNode(n_{P_i}, n)
Map classes to satisfy Rule 2
for 1 ≤ i ≤ j do mapNode(cl_i, cl_i)
Map return values to satisfy Rule 8
for all n ∈ rR ∩ (NI ∪ NR) do mapNode(n, n)
done = false
while not done do
    done = true
    if choose ⟨n3, ⟨⟨n1, f⟩, n2⟩⟩ ∈ WE such that
            n1 ∉ NI then
        WE = WE − {⟨n3, ⟨⟨n1, f⟩, n2⟩⟩}
        Map nodes to satisfy Rule 3
        for all ⟨⟨n3, f⟩, n4⟩ ∈ I do mapNode(n2, n4)
        done = false
    else if choose ⟨n, ⟨⟨n1, f⟩, n2⟩⟩ ∈ WI such that
            n2 ∈ NI ∪ NR then
        WI = WI − {⟨n, ⟨⟨n1, f⟩, n2⟩⟩}
        Map inside node to satisfy Rule 6
        mapNode(n2, n2)
        done = false
    else if choose ⟨n, ⟨⟨n1, f⟩, n2⟩⟩ ∈ W0 such that
            escaped(⟨OM, IM, eM, rM⟩, n) then
        WO = WO − {⟨n, ⟨⟨n1, f⟩, n2⟩⟩}
        Map outside edge and outside node
        to satisfy Rule 7
        OM = OM ∪ {⟨⟨n, f⟩, n2⟩}
        mapNode(n2, n2)
        done = false
Update IM(l) to satisfy Rule 10
IM(l) = ∪{μ(n).n ∈ rR}
```

Figure 11: Constraint Solution Algorithm for μ and $\langle O_M, I_M, e_M, r_M \rangle$

```
Initialize analysis results
for all st ∈ ST do
    α(•st) = α(st•) = ⟨∅, ∅, e⊥, ∅⟩
for all op ∈ OP do
    α(•enter_op) = ⟨O0, I0, e0, r0⟩
Initialize the worklist
W = {enter_op.op ∈ OP}
while (W ≠ ∅) do
    Remove a statement from worklist
    W = W − {st}
    Process the statement
    α(•st) = ⊔{α(st'•).st' ∈ pred(st)}
    α(st•) = [[st]](α(•st))
    if α(st•) changed then
        Put potentially affected statements
        onto worklist
        W = W ∪ succ(st)
        if st ≡ exit_op then
            W = W ∪ callers(op)
```

Figure 12: Fixed-Point Analysis Algorithm

and updates the analysis results before and after the statement. If the analysis result after the statement changed, it inserts all of its successors (or for exit nodes, all of the callers of its method) into the worklist.

The order in which the algorithm analyzes methods can have a significant impact on the analysis. For non-recursive methods, a bottom-up analysis of the program yields the full result with one analysis per method. For recursive methods, the analysis results must be iteratively recomputed within each strongly connected component of the call graph using the current best result until the analysis reaches a fixed point.

It is possible to extend the algorithm so that it initially skips the analysis of method invocation sites. If the analysis result is not precise enough, it can incrementally increase the precision by analyzing method invocation sites that it originally skipped. The algorithm will then propagate the new, more precise result to update the analysis results at affected program points.

6 Abstraction Relation

In this section, we characterize the correspondence between points-to escape graphs and the objects and references created during the execution of the program. A key property of this correspondence is that a single concrete object in the execution of the program may be represented by multiple nodes in the points-to escape graph. We therefore state the properties that characterize the correspondence using an *abstraction relation*, which relates each object to all of the nodes that represent it.

As the program executes, it creates a set of concrete objects $o \in \mathrm{C}$ and a set of references $r \in R \subseteq (\mathtt{V} \times \mathrm{C}) \cup ((\mathtt{CL} \times \mathtt{F}) \times \mathrm{C}) \cup ((\mathrm{C} \times \mathtt{F}) \times \mathrm{C})$ between objects. At each point in the execution of the program, it is possible to define the following sets of references and objects:

- R_C is the set of references created by the current execution of the current method and all of the analyzed methods that it invokes.

- R_R is the set of references read by the current execution of the current method and all of the analyzed methods that it invokes.
- C_R is the set of objects reachable from the local variables, static class variables, and parameters by following references in $R_C \cup R_R$.
- $R_I = R_C \cap (\mathrm{C}_R \times F \times \mathrm{C}_R)$ is the set of inside references. These are the references represented by the set of inside edges in the analysis.
- $R_O = (R_R \cap (\mathrm{C}_R \times F \times \mathrm{C}_R)) - R_I$ is the set of outside references. These are the references represented by the set of outside edges in the analysis.

It is always possible to construct an abstraction relation $\rho \subseteq \mathrm{C} \times N$ between the objects and the nodes in the points-to escape graph $\langle O, I, e, r\rangle$ at the current program point. This relation relates each object to all of the nodes in the points-to escape graph that represent the object during the analysis of the method. The abstraction relation has all of the properties described below.

- Reachable objects are represented by their allocation sites. If o was created at an object creation site within the current execution of the current method or analyzed methods that it invokes, and o is reachable (i.e. $o \in \mathrm{C}_R$), $n \in \rho(o)$, where n is the object creation site's inside node.
- Each object is represented by at most one inside node:
  - $n_1, n_2 \in \rho(o)$ and $n_1, n_2 \in N_I$ implies $n_1 = n_2$
- All outside references have a corresponding outside edge in the points-to escape graph:
  - $\langle \mathtt{v}, o\rangle \in R_O$ implies $O(\mathtt{v}) \cap \rho(o) \neq \emptyset$
  - $\langle\langle \mathtt{cl}, \mathtt{f}\rangle, o\rangle \in R_O$ implies $O(\mathtt{cl}, \mathtt{f}) \cap \rho(o) \neq \emptyset$
  - $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \in R_O$ implies $(\rho(o_1) \times \{\mathtt{f}\}) \times \rho(o_2) \cap O \neq \emptyset$
- All inside references have a corresponding inside edge in the points-to escape graph:
  - $\langle \mathtt{v}, o\rangle \in R_I$ implies $I(\mathtt{v}) \cap \rho(o) \neq \emptyset$
  - $\langle\langle \mathtt{cl}, \mathtt{f}\rangle, o\rangle \in R_I$ implies $I(\mathtt{cl}, \mathtt{f}) \cap \rho(o) \neq \emptyset$
  - $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \in R_I$ implies $(\rho(o_1) \times \{\mathtt{f}\}) \times \rho(o_2) \cap I \neq \emptyset$
- If an object is represented by a captured node, it is represented by only that node:
  - $n \in \rho(o)$ and captured$(\langle O, I, e, r\rangle, n)$ implies $\rho(o) = \{n\}$

  Given this property, we define that an object is captured if it is represented by a captured node. All references to captured objects are either from local variables or from other captured objects:
  - $n \in \rho(o)$, captured$(\langle O, I, e, r\rangle, n)$, and $\langle \mathtt{v}, o\rangle \in R$ implies $\mathtt{v} \in \mathtt{L}$
  - $n_2 \in \rho(o_2)$, captured$(\langle O, I, e, r\rangle, n_2)$, and $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \in R$ implies $\exists n_1 \in N.\rho(o_1) = \{n_1\}$ and captured$(\langle O, I, e, r\rangle, n_1)$

  These properties ensure that captured objects are reachable only via paths that start with the local variables. If an object is captured at a method exit point, it will therefore become inaccessible as soon as the method returns.
- The points-to information in the points-to escape graph completely characterizes the references between objects represented by captured nodes:
  - captured$(\langle O, I, e, r\rangle, n_1)$, captured$(\langle O, I, e, r\rangle, n_2)$, $n_1 \in \rho(o_1)$, $n_2 \in \rho(o_2)$ and $\langle\langle n_1, \mathtt{f}\rangle, n_2\rangle \notin I$ implies $\langle\langle o_1, \mathtt{f}\rangle, o_2\rangle \notin R$

7 Optimizations

For optimization purposes, the two most important properties are that the absence of edges between captured nodes guarantees the absence of references between the represented objects, and that captured objects are inaccessible when the method returns. In this section we discuss two transformations, stack allocation of objects and synchronization elimination, that the escape information enables.

7.1 Synchronization Elimination

If an object does not escape a thread, it is legal to remove the lock acquire and release operations from synchronized methods that execute on that object. The benefit of this transformation is the elimination of synchronization overhead [5]

To apply the transformation, the compiler generates a specialized, synchronization-free version of each synchronized method that may execute on a captured object. At each call site that invokes the method only on captured objects, the compiler generates code that invokes the synchronization-free version.

An issue that the compiler must deal with is the distance in the call graph between the method that captures the object and the synchronized method. In addition to removing synchronization operations from the synchronized method, the compiler must also generate specialized versions of all methods in the call chain from the capturing method to the method containing the call site that invokes the synchronization-free version. Our specialization policy imposes no limit on the number of specialized methods — it applies the transformation whenever possible.

A final problem is finding the paths in the call graph that go from captured objects to synchronized methods. The compiler solves this problem by augmenting the analysis to record, for each node in the points-to escape graph, paths in the call graph that lead to synchronized operations on the objects represented by that node. For captured objects, the compiler can use this information to trace a path from the captured object down to the call sites that invoke synchronized methods on that object.

7.2 Stack Allocation of Objects

If an object does not escape a method, it can be allocated on the stack instead of in the heap. One benefit of this transformation is the elimination of garbage collection overhead. Instead of being processed by the collector, the object

[5] To preserve the semantics of the Java memory model on machines that implement weak memory consistency models, the compiler may need to insert memory barriers at the original lock acquire and release sites when it removes the acquire and release constructs.

will be implicitly collected when the method returns and the stack rolls back. Another benefit is better memory locality because the stack frame will tend to be resident in the cache. Finally, if the compiler can precisely compute the location of the object on the stack, it can generate code that accesses the object fields directly in the stack frame.

The largest potential drawback of stack allocation is that it may increase memory consumption by extending the lifetime of objects allocated on the stack. Because this problem may be especially acute for allocation sites that create a statically unbounded number of objects on the stack, our implementation allocates objects on the stack only if the allocation site will be executed at most once per invocation of the *allocating method*, or the method that contains the object allocation site.

In the simplest case, a captured object does not escape its allocating method. In this case, the compiler can simply replace the normal object allocation instruction with a special instruction that allocates the object on the heap.

If an object escapes its allocating method, but is recaptured within a caller (direct or indirect), it may be possible to allocate the object on the stack of the method that captures the object. In addition to requiring that stack-allocated objects come from allocation sites that are executed at most once per invocation of the allocating method, the compiler also requires that the allocating method be invoked at most once for each call site in the captured method that leads to the allocating method.

If the allocation site satisfies this restriction, the compiler generates a specialized version of the allocating method. Instead of allocating the object itself, the specialized version takes, as a parameter, a pointer to preallocated space in the capturing method's activation record. The allocation site is transformed to initialize the object at the given location, rather than to allocate it in the heap. As for the synchronization elimination transformation described above, the compiler must generate specialized versions of all methods on the call chain from the capturing method to the allocating method.

For this transformation to interoperate correctly with the rest of the system, the garbage collector must recognize stack-allocated objects. The recognition mechanism is simple — it examines the address of the object to determine if it is allocated on a stack or in the heap. During a collection, the collector still scans stack-allocated objects normally. But it does not attempt to move or collect stack-allocated objects.

8 Experimental Results

We have implemented a combined pointer and escape analysis based on the algorithm described in Sections 4 and 5. We implemented the analysis in the compiler for the Jalapeño JVM [8], a Java virtual machine written in Java with a few unsafe extensions for performing low-level system operations such as explicit memory management and pointer manipulation.

8.1 Compiler Structure

The analysis is implemented as a separate phase of the Jalapeño dynamic compiler, which operates on the Jalapeño intermediate representation. To analyze a class, the algorithm loads the class, converts its methods into the intermediate representation, then analyzes the methods. The final analysis results for the methods are written out to a file. This approach provides excellent support for dynamically loaded programs. It allows the compiler to analyze a large, commonly used package such as the Java Class Libraries once, then reuse the analyze results every time a program is loaded that uses the package. It also supports the delivery of preanalyzed packages. Instead of requiring the analysis to be performed when the package is first loaded into a customer's virtual machine, a vendor could perform the analysis as part of the release process, then ship the analysis results along with the code.

When the Jalapeño compiler generates code for a dynamically loaded class, it uses the information in the analysis file to apply the stack allocation and synchronization elimination optimizations described above.[6] One unusual feature of the system is that the analysis is applied to the compiler and virtual machine during the bootstrap process. The entire system, including the dynamic compiler and the virtual machine as well as the applications, therefore executes with the optimizations applied.

8.2 Benchmark Set

Our benchmark set includes four programs. Javac[7] is the standard Java compiler. Javacup[8] is a yacc-like parser generator for Java. Javalex[9] is a lexical analyzer generator for Java. Pbob is a transaction-processing benchmark designed to quantify the performance of simple transactional server workloads written in Java.

We chose these applications in part because they are all complete programs in regular use. We therefore expect them to exhibit realistic object creation and access patterns. They were also chosen to test the scalability of our analysis. With the associated (and analyzed) class libraries, all three programs are on the order of tens of thousands of lines of code, and many existing pointer analysis algorithms are impractical for programs of this size. Figure 13 presents the number of lines of code and the total sizes of the class files for these applications. JVM is the Jalapeño virtual machine.

Benchmark	Lines of Code	Class File Size (bytes)
javac	-	87,801
javacup	10,574	157,057
javalex	8,159	95,229
pbob	18,370	253,752
JVM	70,536	456,494

Figure 13: Benchmark Characteristics

We staged the analysis of the applications as follows. We first analyzed the virtual machine and the standard libraries, writing the analysis results out to the appropriate files. We then analyzed the application code, reusing the analysis results from the standard libraries.

Benchmark	Number of Synchronization Operations Before Optimization	Number of Synchronization Operations After Optimization
javac	4,583,013	2,925,653
javacup	1,004,409	330,952
javalex	2,038,945	1,058,456
pbob	205,198,941	155,764,374

Figure 14: Number of Synchronization Operations Before and After Optimization

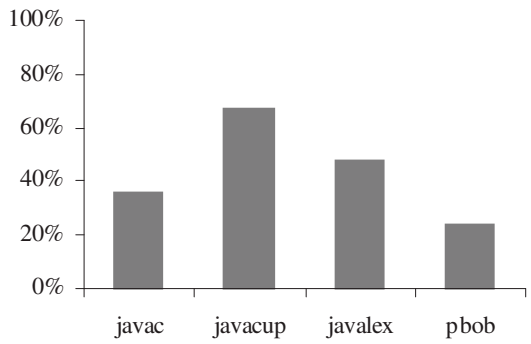

Figure 15: Percentage of Eliminated Synchronization Operations

8.3 Synchronization Elimination

Figure 14 presents the dynamic number of synchronization operations for all of the applications both before and after the synchronization elimination optimization. Figure 15 plots the percentage of eliminated synchronization operations. The analysis achieves reasonable results, eliminating between 24% to 64% of the synchronization operations.

8.4 Stack Allocation

Figure 16 presents the dynamic number of heap allocated objects for all of the applications both before and after the stack allocation optimization. Figure 17 plots the percentage of objects allocated on the stack. Once again, the analysis achieves reasonable results, allocating between 22% and 95% of the objects on the stack.

Figure 18 presents the total size of the heap allocated objects for all of the applications both before and after the stack allocation optimization. Figure 17 plots the percentage of memory allocated on the stack instead of in the heap. The relative amount of object memory allocated on the stack is closely correlated with the number of objects allocated on the stack.

[6] The current version of the Jalapeño JVM does not support stack allocation. All objects are therefore allocated on the heap. The compiler does, however, generate all of the specialized methods necessary to support stack allocation. The compiler fully implements the synchronization elimination optimization.

[7] See the package sun.tools.javac in the standard Java distribution.

[8] Available at www.cs.princeton.edu/ appel/modern/java/CUP/

[9] Available at www.cs.princeton.edu/ appel/modern/java/JLex/

Benchmark	Number of Heap-Allocated Objects Before Optimization	Number of Heap-Allocated Objects After Optimization
javac	2,845,485	2,033,525
javacup	1,913,594	1,495,141
javalex	4,389,452	214,803
pbob	56,708,490	39,751,260

Figure 16: Number of Heap-Allocated Objects Before and After Optimization

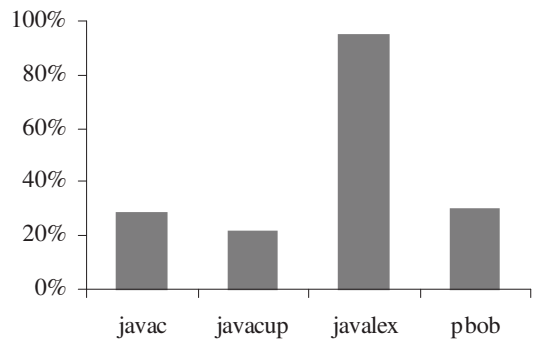

Figure 17: Percentage of Objects Allocated on the Stack

Benchmark	Size of Heap-Allocated Objects Before Optimization	Size of Heap-Allocated Objects After Optimization
javac	81,573,448	60,843,884
javacup	54,089,120	43,038,640
javalex	107,347,648	8,887,836
pbob	25,950,641,024	18,046,027,560

Figure 18: Total Size of Heap-Allocated Objects Before and After Optimization (bytes)

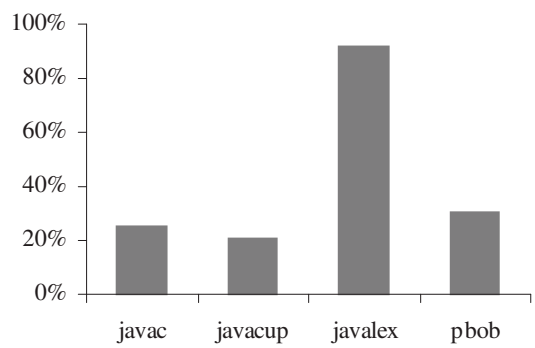

Figure 19: Percentage of Object Memory Allocated on the Stack

9 Related Work

In this section, we discuss several areas of related work: pointer analysis, escape analysis, and synchronization optimizations.

9.1 Pointer Analysis

Pointer analysis for sequential programs is a relatively mature field. Flow-insensitive analyses, as the name suggests, do not take statement ordering into account, and typically use some form of constraint-based analysis to produce a single points-to graph that is valid across the entire program [2, 27, 26]. Flow-insensitive analyses extend trivially from sequential programs to multithreaded programs. Because they are insensitive to the statement order, they trivially model all of the interleavings of the parallel executions.

9.1.1 Flow-Sensitive Analyses

Flow-sensitive analyses take the statement ordering into account, typically using a dataflow analysis to produce a points-to graph or set of alias pairs for each program point [25, 22, 28, 18, 10, 20]. One approach analyzes the program in a top-down fashion starting from the `main` procedure, reanalyzing each potentially invoked procedure in each new calling context [28, 18]. This approach exposes the compiler to the possibility of spending significant amounts of time reanalyzing procedures. It also commits the compiler to an analysis of the entire program and is impractical for situations in which the entire program is not available.

Another approach analyzes the program in a bottom-up fashion, extracting a single analysis result for each procedure. The analysis reuses the result at each call site that may invoke the procedure. Sathyanathan and Lam present an algorithm for programs with non-recursive pointer data types [25]. This algorithm supports strong updates and extracts enough information to obtain fully context-sensitive results in a compositional way. Chatterjee, Ryder, and Landi describe an approach that extracts multiple analysis results; each result is indexed under the calling context for which it is valid [9]. Our approach extracts a single analysis result for calling contexts with no aliases, and merges nodes for calling contexts with aliases. The disadvantage of this approach is that it may produce less precise results than approaches that maintain information for multiple calling contexts. The advantage is that it leads to a simpler algorithm and smaller analysis results.

Many pointer and shape analysis algorithms are based on the abstraction of points-to graphs [24, 18, 28]. The nodes in these graphs typically represent memory locations, objects, or variables, and the edges represent references between the represented entities. A standard technique is to use invisible variables [20, 18, 28] to parameterize the analysis result so that it can be used at different call sites. Like outside objects, invisible variables represent entities from outside the analysis context. But the relationships between invisible variables are typically determined by the aliasing relationships in the calling context or by the type system. In our approach, the relationships between outside objects are determined by the structure of the analyzed method — each load statement corresponds to a single outside object. This approach leads to a simple and efficient treatment of recursive data structures. Another difference between the approach presented in this paper and other previously presented approaches is the explicit distinction between inside and outside edges.

9.1.2 Pointer Analysis for Multithreaded Programs

Rugina and Rinard recently developed a flow-sensitive pointer analysis algorithm for multithreaded programs [23]. The key complication in this algorithm is characterizing the interaction between multiple threads and how this interaction affects the points-to relationship. The analysis presented in this paper simply sidesteps this issue. It generates precise results only for objects that do not escape. If an object escapes to another thread, our algorithm is not designed to maintain precise information about its points-to relationships. It is important to realize that this is a key design decision that allows us to obtain good escape analysis results with what is essentially a local analysis.

Corbett describes a shape analysis algorithm for multithreaded Java programs [12]. The algorithm can be used to find objects that are accessible to a single thread or objects that are accessed by at most one thread at any given time. The goal is to use this information to reduce the size of the state space explored by a model checker. The algorithm is intraprocedural, and does not address the problem of analyzing programs with multiple procedures.

9.2 Escape Analysis

There has been a fair amount of work on escape analysis in the context of functional languages [4, 3, 29, 14, 15, 5, 19]. The implementations of functional languages create many objects (for example, cons cells and closures) implicitly. These objects are usually allocated in the heap and reclaimed later by the garbage collector. It is often possible to use a lifetime or escape analysis to deduce bounds on the lifetimes of these dynamically created objects, and to perform optimizations to improve their memory management.

Deutsch [14] describes a lifetime and sharing analysis for higher-order functional languages. His analysis first translates a higher-order functional program into a sequence of operations in a low-level operational model, then performs an analysis on the translated program to determine the lifetimes of dynamically created objects. The analysis is a whole-program analysis. Park and Goldberg [3] also describe an escape analysis for higher-order functional languages. Their analysis is less precise than Deutsch's. It is, however, conceptually simpler and more efficient. Their main contribution was to extend escape analysis to include lists. Deutsch [15] later presented an analysis that extracts the same information but runs in almost linear time. Blanchet [5] extended this algorithm to work in the presence of imperative features and polymorphism. He also provides a correctness proof and some experimental results.

Baker [4] describes an novel approach to higher-order escape analysis of functional languages based on the type inference (unification) technique. The analysis provides escape information for lists only. Hannan also describes a type-based analysis in [19]. He uses annotated types to describe the escape information. He only gives inference rules and no algorithm to compute annotated types.

9.3 Synchronization Optimizations

Diniz and Rinard [16, 17] describe several algorithms for performing synchronization optimizations in parallel programs. The basic idea is to drive down the locking overhead by coalescing multiple critical sections that acquire and release the same lock multiple times into a single critical section that acquires and releases the lock only once. When possible, the algorithm also coarsens the lock granularity by using locks

in enclosing objects to synchronize operations on nested objects. Plevyak and Chien describe similar algorithms for reducing synchronization overhead in sequential executions of concurrent object-oriented programs [21].

Several research groups have recently developed synchronization optimization techniques for Java programs. Aldrich, Chambers, Sirer, and Eggers describe several techniques for reducing synchronization overhead, including synchronization elimination for thread-private objects and several optimizations that eliminate synchronization from nested monitor calls [1]. Blanchet describes a pure escape analysis based on an abstraction of a type-based analysis [6]. The implementation uses the results to eliminate synchronization for thread-private objects and to allocate captured objects on the stack. Bogda and Hoelzle describe a flow-insensitive escape analysis based on global set inclusion constraints [7]. The implementation uses the results to eliminate synchronization for thread-private objects. A limitation is that the analysis is not designed to find captured objects that are reachable via paths with more than two references.

Choi, Gupta, Serrano, Sreedhar, and Midkiff present a compositional dataflow analysis for computing reachability information [11]. The analysis results are used for synchronization elimination and stack allocation of objects. Like the analysis presented in this paper, it uses an extension of points-to graphs with abstract nodes that may represent multiple objects. It does not distinguish between inside and outside edges, but does contain an optimization, deferred edges, that is designed to improve the efficiency of the analysis. The approach classifies objects as globally escaping, escaping via an argument, and not escaping. Because the primary goal was to compute escape information, the analysis collapses globally escaping subgraphs into a single node instead of maintaining the extracted points-to information. Our analysis retains this information, in part because we anticipate further thread interaction analyses (for example, extensions of existing pointer analysis algorithms for multithreaded programs [23]) that will use this information.

10 Conclusion

This paper presents a new combined pointer and escape analysis algorithm for object-oriented programs. The algorithm is designed to analyze arbitrary parts of complete or incomplete programs, obtaining complete information for objects that do not escape the analyzed parts.

We have implemented the algorithm in the IBM Jalapeño virtual machine, and applied the analysis information to two optimization problems: synchronization elimination and stack allocation of objects. For our benchmark programs, our algorithms enable the stack allocation of between 22% and 95% of the objects. They also enable the elimination of between 24% and 67% of the synchronization operations.

In the long run, we believe the most important concept in this research may turn out to be designing analysis algorithms from the perspective of extracting and representing interactions between analyzed and unanalyzed regions of the program. This concept leads to representations, such as the points-to escape graphs presented in this paper, that make a clean distinction between information created locally within an analyzed region and information created in the rest of the program outside the analyzed region.

These representations support flexible analyses that are capable of analyzing arbitrary parts of the program, with the analysis result becoming more precise as more of the program is analyzed. At every stage of the analysis, the algorithm can distinguish where it does and does not have complete information. As developers continue to move to a model of dynamically loaded, component-based software, we believe this general approach will become increasingly relevant and compelling.

Acknowledgements

The authors would like to acknowledge discussions with Radu Rugina and Darko Marinov regarding pointer and escape analysis. The second author would like to acknowledge discussions with Mooly Sagiv regarding pointer, escape, and shape analysis. The first author would like to acknowledge Jong-Deok Choi for introducing him to the concept of using escape information for synchronization elimination, while he was an IBM employee in the summer of 1998. The authors would like to thank the developers of the IBM Jalapeño virtual machine and the MIT Flex project for providing the compiler infrastructure in which this research was implemented, and the anonymous reviewers for their comments.

References

[1] J. Aldrich, C. Chambers, E. Sirer, and S. Eggers. Static analyses for eliminating unnecessary synchronization from java programs. In *Proceedings of the 6th International Static Analysis Symposium*, September 1999.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] B. Goldberg and Y. Park. Escape analysis on lists. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, pages 116–127, July 1992.

[4] H. Baker. Unifying and conquer (garbage, updating, aliasing ...) in functional languages. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 218–226, 1990.

[5] B. Blanchet. Escape analysis: Correctness proof, implementation and experimental results. In *Proceedings of the 25th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1998. ACM, ACM, New York.

[6] B. Blanchet. Escape analysis for object oriented languages. application to java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[7] J. Bodga and U. Hoelzle. Removing unnecessary synchronization in java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[8] M. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeño dynamic optimizing compiler for java. In *Proceedings of the ACM SIGPLAN 1999 Java Grande Conference*, June 1999.

[9] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *Proceedings of the 26th Annual ACM Symposium on the Principles of Programming Languages*, San Antonio, TX, January 1999.

[10] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual Symposium on Principles of Programming Languages*, Charleston, SC, January 1993. ACM.

[11] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[12] J. Corbett. Using shape analysis to reduce finite-state models of concurrent java programs. In *Proceedings of the International Symposium on Software Testing and Analysis*, March 1998.

[13] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[14] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the 17th Annual ACM Symposium on the Principles of Programming Languages*, pages 157–168, San Francisco, CA, January 1990. ACM, ACM, New York.

[15] A. Deutsch. On the complexity of escape analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997. ACM, ACM, New York.

[16] P. Diniz and M. Rinard. Lock coarsening: Eliminating lock overhead in automatically parallelized object-based programs. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, pages 285–299, San Jose, CA, August 1996. Springer-Verlag.

[17] P. Diniz and M. Rinard. Synchronization transformations for parallel computing. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, pages 187–200, Paris, France, January 1997. ACM, New York.

[18] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Program Language Design and Implementation*, pages 242–256, Orlando, FL, June 1994. ACM, New York.

[19] J. Hannan. A type-based analysis for block allocation in functional languages. In *Proceedings of the Second International Static Analysis Symposium*. ACM, ACM, New York, September 1995.

[20] W. Landi and B. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, June 1992.

[21] J. Plevyak, X. Zhang, and A. Chien. Obtaining sequential efficiency for concurrent object-oriented languages. In *Proceedings of the 22nd Annual ACM Symposium on the Principles of Programming Languages*, San Francisco, CA, January 1995. ACM, New York.

[22] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[23] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the SIGPLAN '99 Conference on Program Language Design and Implementation*, Atlanta, GA, May 1999.

[24] 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.

[25] P. Sathyanathan and M. Lam. Context-sensitive interprocedural pointer analysis in the presence of dynamic aliasing. In *Proceedings of the Ninth Workshop on Languages and Compilers for Parallel Computing*, San Jose, CA, August 1996. Springer-Verlag.

[26] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, January 1997.

[27] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, FL, January 1996.

[28] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995. ACM, New York.

[29] Y. Tang and P. Jouvelot. Control-flow effects for escape analysis. In *Workshop on Static Analysis*, pages 313–321, September 1992.