Static Ownership Inference for Reasoning Against Concurrency Errors

Ana Milanova
Department of Computer Science
Rensselaer Polytechnic Institute
milanova@cs.rpi.edu

Yin Liu
Department of Computer Science
Rensselaer Polytechnic Institute
liuy@cs.rpi.edu

Abstract

We propose a new approach for reasoning about concurrency in object-oriented programs. Central to our approach is static ownership inference analysis — we conjecture that this analysis has important application in reasoning against concurrency errors.

1 Introduction

Software engineering, compilers and programming languages must evolve to meet the demands for concurrency. Shared memory concurrency in particular is especially difficult because of the many ways different threads can interfere over shared data; in the same time, with the advance of multi-core processors, shared memory concurrency is becoming increasingly relevant. Therefore, research in this direction is becoming increasingly important.

Many new techniques (predominantly dynamic) for detection of concurrency errors such as data races have been proposed recently. However, these techniques focus extensively on error detection and less on understanding the underlying structure of sharing in concurrent programs. Understanding the structure of sharing is important because it may help not only detect but also correct and prevent concurrency errors.

We propose *new static program analysis for reasoning about concurrency in object-oriented programs*. There are two high-level ideas behind our approach.

First, our approach emphasizes a structural view of the program: it reveals the structure of shared objects, and the ways different threads access these shared objects. Our approach highlights the important connection between data and control in object-oriented codes: it reveals object structure and ownership information (e.g., one object "owns" another object) as well as relevant transfer of control (e.g., one object calls a method on another object).

Second, our approach emphasizes the notion of *object ownership*. Informally, an object o *owns* an object o_i if o dominates o_i in the graph of object accesses — that is, all accesses to o_i must go through o. Ownership is useful in reasoning about software security and software quality — it guarantees lack of *representation exposure* and may help prevent serious program errors. Ownership is useful in reasoning about concurrency as well — lack of representation exposure may prevent concurrency errors and may facilitate reasoning about concurrent programs. More concretely, ownership guarantees that appropriate synchronization on a shared object o protects o as well as all objects in the *dominance boundary* of o (i.e., the set of objects that are dominated by o in the graph of object accesses, or in other words, the set of objects that are encapsulated in o); conversely, lack of appropriate synchronization on o may expose concurrency errors such as data races on o as well as on objects nested (sometimes deeply) in the dominance boundary of o.

Our proposed work has the following contributions:

- First, we propose to design and implement algorithms for construction of the *annotated object graph* — a novel representation of program objects and object accesses. The annotated object graph reveals structural information as well as control information.
- Second, we propose a static analysis algorithm for *data race detection* which uses the annotated object graph and ownership information. For programs that exhibit good ownership structure (i.e., programs that exhibit little or no representation exposure), this algorithm may lead to intuitive, fast and accurate detection of data races. The main conjecture of our work is that the annotated object graph and ownership information would prove useful in analyses for race detection, atomicity violation detection, unnecessary synchronization removal, and lock inference.
- Third, we identify several structural patterns of object sharing. We propose a detailed study which will use the annotated object graph and ownership information to examine the occurrence of these patterns in real-world concurrent Java applications.

```
public class Database {
 private ConnectionManager cm;
 public int insert(...)  {
  Connection c=cm.getConnection();
  ...
 }
 public int delete(...)  {...}
}
public class ConnectionManager {
private Map conns=
  Collections.synchronizedMap(new HashMap());
 public void makeConnection(String s) {
  ConnectionSource cs=new ConnectionSource();
  conns.put(s,cs);
 }
 public Connection getConnection(String s) {
  ConnectionSource cs=conns.get(s);
  if (c!=null) return cs.getConnection();
 }
}
public class ConnectionSource {
 private Connection conn;
 private boolean used;
 public Connection getConnection() {
  if (!used) {
   used = true;
   return conn;
  }
 }
}
```

Figure 1. An example from JdbF.

2 Running Example

As an illustrating example, consider the code in Figure 1. This example is taken from [5] and illustrates a data race in a real-world Java application, JdbF. We have modified the code slightly to better illustrate our proposed approach. JdbF is a library and class `Database` provides the interface to clients; clients in multiple threads can access a `Database` object through its public methods `insert` and `delete`. Methods `insert` and `delete` acquire a connection, perform operations on it, and then release this connection; any thread performing operations on a connection must have exclusive access to the connection. Each `Database` object has a private `ConnectionManager` object which handles the database connections. Furthermore, each connection is encapsulated within a `ConnectionSource` object; this is done with the intent to ensure exclusive thread access to a connection. The `ConnectionManager` maintains a private map of `ConnectionSource` objects.

The data race occurs on instance field $used$ in class `ConnectionSource`. The following method call sequence can be executed by two threads simultaneously: `Database.insert`, followed by `ConnectionManager.getConnection`, followed by `ConnectionSource.getCon-nection`. Therefore, the read of field $used$, namely `if (!used)`, can be performed temporally next to the write of field $used$, namely `used = true`. There are no ordering constraints on the read and write to field $used$; as a result, each of the threads could obtain access to the same connection.

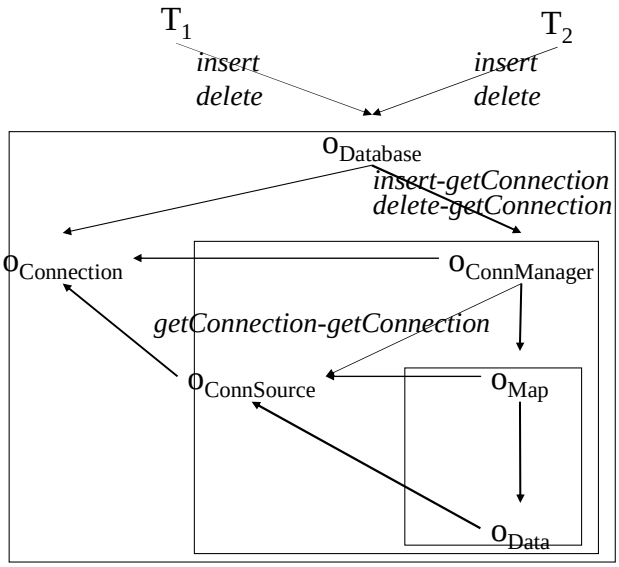

Figure 2. Annotated object graph for Figure 1

3 Annotated Object Graph and Ownership

This section describes the annotated object graph and the ownership information inferred from the object graph. Section 3.1 outlines the construction of the annotated object graph: initially, it describes the object graph — a static structure that represents the accesses between run-time objects; subsequently, it describes the extension to the object graph with annotations that model transfer of control between run-time objects. Section 3.2 describes the ownership information inferred from the object graph. Sections 4 and 5 propose applications of the annotated object graph and ownership information.

3.1 Annotated Object Graph

The object graph is a static structure that approximates run-time accesses between objects. Informally, object o accesses o' if one of the following is true: (i) a field f of o refers to o', or (ii) a method m invoked on receiver o has a local variable that refers to o'. The algorithm for object graph construction is a client of a points-to analysis—it takes as input a points-to graph and produces the object graph as an output. The object graph is a safe approximation of run-time object accesses—that is, if there is a run-

time access edge between two run-time objects, then there is an edge in the object graph between the representatives of these run-time objects.

Figure 2 shows the annotated object graph for the code in Figure 1. The nodes in the object graph are object names (e.g., $o_{Database}$)—there is an object name for each allocation site in the program. Nodes `T1` and `T2` correspond to two threads that access the shared `Database` object $o_{Database}$. The annotations on object graph edges will be explained shortly. Edge $o_{Database} \rightarrow o_{ConnManager}$ represents the access between the `Database` object and its corresponding `ConnectionManager` object; the edge is created by processing an object creation statement in the code (it is not shown in Figure 1). Edge $o_{ConnManager} \rightarrow o_{Map}$ is due to the creation statement in class `ConnectionManager` which creates a new `HashMap` object. Edge $o_{Map} \rightarrow o_{Data}$, the edge between the container object and its internal data array, is due to a creation statement in the code for `HashMap` (this code, part of the standard Java library is not shown here; it is processed by the analysis accordingly). Edge $o_{ConnManager} \rightarrow o_{ConnSource}$ is due to the creation statement in class `ConnectionManager`; this statement creates a `ConnectionSource` object. Furthermore, edges $o_{Map} \rightarrow o_{ConnSource}$ and $o_{Data} \rightarrow o_{ConnSource}$ are due to the statements that pass the `ConnectionSource` object to the `Map` object and then to the data array of the `Map` object. Finally, edges $o_{ConnSource} \rightarrow o_{Connection}$, $o_{ConnManager} \rightarrow o_{Connection}$ and $o_{Database} \rightarrow o_{Connection}$ are due to program statements that pass the `Connection` object to different objects. For example, edge $o_{ConnManager} \rightarrow o_{Connection}$ is due to statement `cs.getConnection()` in `ConnectionManager` which passes $o_{Connection}$ from $o_{ConnSource}$ to $o_{ConnManager}$; similarly, edge $o_{Database} \rightarrow o_{Connection}$ is due to statement `cm.getConnection()` which passes $o_{Connection}$ from $o_{ConnManager}$ to $o_{Database}$.

We add annotations to edges to model transfer of control between objects. Annotation m_1-m_2 on object graph edge $o_1 \rightarrow o_2$ means that method m_1 invoked on receiver o_1 calls method m_2 on receiver o_2. Notation $o_1.m_1$-$o_2.m_2$ abbreviates this information. For example, annotation `insert`-`getConnection` on edge $o_{Database} \rightarrow o_{ConnManager}$ denotes that method `insert` called on receiver $o_{Database}$ calls method `getConnection` on receiver $o_{ConnManager}$. These annotations abstract away calls through `this` — they shows only method calls that transfer control from one object to another.

3.2 Ownership

Informally, we say that an object o *owns* object o_i if o immediately dominates o_i on the object graph—that is, all accesses of o_i must go through its owner o. Again informally, the *dominance boundary* of o is the portion of the object graph that is dominated by o; it includes the objects owned by o as well as the boundaries of the objects owned by o. Figure 2 shows the ownership structure of our running example. The dominance boundary of the top-level object $o_{Database}$ includes all other objects, namely $o_{ConnManager}$, o_{Map}, o_{Data}, $o_{ConnSource}$ and $o_{Connection}$; it is easy to see that $o_{Database}$ dominates these objects (i.e., all accesses to these objects must go through $o_{Database}$). The dominance boundary of $o_{ConnManager}$, which is nested within the boundary of $o_{Database}$ includes objects $o_{ConnSource}$, o_{Map} and o_{Data}. Finally, the dominance boundary of o_{Map} which is nested within the boundary of $o_{ConnManager}$ includes object o_{Data}.

4 Reasoning Against Data Races

The main intuition behind our analysis is that in order to have a data race on an object o_n we must first have an object race on the owner of o_n, o_k; conversely, there is no data race on o_n if the accesses to o_k are appropriately synchronized, because the accesses to o_n are protected by synchronization on its owner o_k. Further, in order to have an object race on o_k we must have an object race on the owner of o_k, and so on, until we reach an object race on the "central" object o.

We illustrate our proposed algorithm for race detection by two examples. Consider the following pair of accesses: ($o_{ConnSource}$.`getConnection` (a read to field *used* of $o_{ConnSource}$), $o_{ConnSource}$.`getConnection` (a write to field *used* $o_{ConnSource}$)). Method `getConnection` is unsynchronized and the pair of accesses constitutes a potential data race, unless the accesses are appropriately protected by synchronization on $o_{ConnSource}$'s owner. There is a pair of paths, p_1, p_2 from the owner of $o_{ConnSource}$, $o_{ConnManager}$, to $o_{ConnSource}$; we have $p_1 = p_2$: $o_{ConnManager}$.`getConnection`–$o_{ConnSource}$.`getConnection`. `getConnection` in `ConnectionManager` is unsynchronized and therefore there is a potential object race on $o_{ConnManager}$. Next, there is a pair of paths p_1, p_2 from the owner of $o_{ConnManager}$, $o_{Database}$, to $o_{ConnManager}$; we have p_1=p_2: $o_{Database}$.`insert`—$o_{ConnManager}$.`getConnection` where again, `insert` is unsynchronized. Since $o_{Database}$ is the central object which is directly accessed by different threads, the algorithm reports the race:

`T1`–
$o_{Database}$.`insert` – $o_{ConnManager}$.`getConnection`–
$o_{ConnSource}$.`getConnection` // read of *used*

`T2`–
$o_{Database}$.`insert` – $o_{ConnManager}$.`getConnection`–
$o_{ConnSource}$.`getConnection` // write of *used*

As another example consider potential race ($o_{Data}[\,]$ (array element read), $o_{Data}[\,]$ (array element write)). The algorithm looks for related object race on the owner o_{Map}. It considers p_1: $o_{Map}.get - o_{Data}[\,]$ (i.e., an array read), and p_2: $o_{Map}.put - o_{Data}[\,]$ (i.e., an array write). However, `get` and `put` are synchronized on o_{Map} and the potential race is proven safe: the accesses to o_{Data} are protected by synchronization on its owner o_{Map}.

The advantage of this approach is that it structures the search: search starts from the innermost boundary and proceeds outwards. If the program is well-structured into deeply nested dominance boundaries, this approach may help reduce the search space and lead to fast and accurate race detection.

5 The Structure of Object Sharing

The annotated object graph and ownership information can be used to reason about the structure of accesses to shared objects. We identify three structural patterns.

First, we consider *thread owned* objects—these objects are owned by their creating thread. Thread ownership is a stronger property than thread locality in escape analysis [2, 7]. We conjecture that a large number of objects identified as non-thread-local would be identified as thread-owned.

Second, we consider *central shared* objects o—these objects are accessed *directly* by two or more threads (i.e., the `run` methods of two different threads call methods on o). Central shared objects typically have deep dominance boundaries—i.e., although thread access occurs centrally through o, a race condition could occur on an object deep within the dominance boundary of o. An example of such an object is $o_{Database}$. Central shared objects, and the structures that are accessed through them, are easy to reason about, as shown in the previous section. We conjecture that this case is especially relevant for the analysis of open programs such as JdbF.

Third, we consider *distributed shared* objects o. The access to o is distributed in the following sense. Let o be created by a thread `T1`, and let o be passed to the boundary of another object, say o' where o' is created and accessed by thread `T2`. Object o is accessed by `T1`, and indirectly, through o', by `T2`. As an example, one may have a collection object o created and directly accessed by thread `T1` (i.e., by calling `add`, etc. on o), and another collection object o' which is owned by thread `T2`. In order to copy the elements of collection object o into o' (i.e., by calling `addAll`) one needs to pass o to o'. Thread `T1` accesses o by calling `add`; in the same time `T2` accesses o' calling `addAll` and within `addAll` o' accesses o, which may lead to problematic accesses. Distributed shared objects are difficult to reason about.

6 Related Work

There is a large amount of work on detection of concurrency errors. The trend is notably towards dynamic and hybrid approaches [8, 6], which may be due to better scalability compared to existing static approaches.

Our work focuses on scalable static analysis, which is underrepresented in research on concurrency. Static analysis has important advantages over dynamic analysis: (i) it avoids the run-time overhead of instrumentation, (ii) it is safe, and (iii) it can help understand the underlying structure of sharing in a concurrent program and not only detect, but correct and prevent concurrency errors. Our ownership inference analyses are precise and scalable [3] and we conjecture that they could be used in analysis of large concurrent programs. To the best of our knowledge the state-of-the-art in scalable static race detection is Chord [5]. Chord's approach is different from ours; it uses object-sensitive points-to analysis [4] to reason about object accesses while we propose to use ownership analysis for this purpose. Ownership analysis may be less expensive than object-sensitive points-to analysis. Most importantly, ownership may present a more intuitive way of reasoning about shared objects.

It is known that ownership is a useful concept in reasoning about concurrency. Von Praun and Gross [9] use the notion of thread ownership in the context of dynamic object race detection and Boyapati et al. [1] use a similar notion in the context of an ownership type system. The main contribution of our work lies in the use of object ownership in the context of scalable static analysis.

References

[1] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[2] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *OOPSLA*, pages 1–19, 1999.

[3] Y. Liu and A. Milanova. Ownership and immutability inference for UML-based object access control. In *ICSE*, pages 323–332, 2007.

[4] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM TOSEM*, 14(1):1–42, 2005.

[5] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *PLDI*, pages 308–319, 2006.

[6] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *FSE*, 2008.

[7] A. Rountev, A. Milanova, and B. G. Ryder. Points-to analysis for Java using annotated constraints. In *OOPSLA*, pages 43–55, 2001.

[8] K. Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[9] C. von Praun and T. Gross. Object race detection. In *OOPSLA*, pages 70–82, 2001.