Type-Based Race Detection for Java (summary)

Cormac Flanagan
Compaq Systems Research Center
130 Lytton Ave.
Palo Alto, CA 94301
flanagan@pa.dec.com

Stephen N. Freund[*]
Department of Computer Science
Stanford University
Stanford, CA 94305-9045
freunds@cs.stanford.edu

1 Introduction

A race may occur in a concurrent program when two threads access a shared memory location at the same time. This situation often causes unintended behavior ranging from memory corruption to execution failure, and races may be difficult to locate and fix, even after their effects have been observed.

To avoid race conditions, programmers often adopt a programming discipline in which shared resources are guarded by locks. Before accessing any shared structure, the necessary lock must be acquired. This discipline ensures that no two threads ever access the same resource at the same time. Using locks in this fashion shifts the problem of preventing races to one of enforcing the locking discipline.

This summary describes a static analysis technique that supports this locking strategy in concurrent Java programs[1]. The analysis technique, which is presented in the form of the static type system, was designed with the following desirable features:

1. a sound formal foundation
2. low programmer overhead
3. the ability to check a reasonable set of programming idioms

The following section describes an extension to the Java type system that captures locking information; the third section describes a prototype implementation, and the fourth section summarizes our experiences with it.

2 Type System and Annotation Language

We build on an existing race-free type system for a concurrent object calculus [FA99]. To preserve compatibility with standard Java compilers, the additional type information used in the analysis is written in special comments in the code, similar to those of escjava [LSS99, DLNS98]. For example, the following class is a monitored counter:

```
class Counter {
  private int c = 0 /*# guarded_by this */;
  private void set(int x) /*# requires this */ {
    c = x;
  }
  public void increment() {
    synchronized(this) {
      set(c+1);
    }
  }
}
```

The `guarded_by` annotation on the field `c` indicates which lock must be held to access that field, and the requires clause on the `set` method indicates which lock (or locks) must be held prior to invoking that method. To typecheck a program, a conservative approximation of the set of locks held at each program point is determined, and the checker then verifies whether the constraints expressed in the annotations are satisfied on each field access and method invocation.

As part of this verification process, the analysis needs to determine whether a specific lock is in the lock set, which requires some notion of equality between lock names. Since our analysis cannot rely on run-time values, we approximate run-time value equivalence with syntactic equality. This approximation is sufficient for most of the programs we studied.

Another common programming idiom is to create unsynchronized classes and require the client to provide the necessary synchronization. This type of class may be expressed in our type system using classes parameterized by lock names, as shown below:

```
class Counter/*# {ghost Object o} */ {
  private int value = 0  /*# guarded_by o */;
  private void set(int x) /*# requires o */ {
    value = x;
  }
  public void increment() /*# requires o */ {
    set(value+1);
  }
}

Object mutex = new Object();
Counter/*#{mutex}*/ c = new Counter/*#{mutex}*/();
```

It is often the case that a significant fraction of a concurrent program does not use synchronization at all. To avoid the need to require locks on objects that

[*]This work was completed while the author was employed at the Compaq Systems Research Center.
[1]see [FF00] for a full discussion of this work.

Program	Lines Of Code	Programmer Time (hrs)	Annotations	Races Found
`java.util.Hashtable`	440	0.5	60	0
`java.util.Vector`	430	0.5	10	1
`java.io.*`	16,000	16.0	139	4
Ambit	4,500	4.0	38	0
WebL	20,000	12.0	358	5

Table 1: Programs analyzed using `rccjava`.

are not shared between threads, we introduce the notion of a `thread_local` class into the type system. A `thread_local` class is a class whose instances are never shared between threads, indicated with the annotation `thread_local` on the class declaration. This type of class requires no synchronization on field accesses, and a class may be thread local only if:

1. no instances of the class are stored in fields of a shared class
2. the class is not a subclass of java.lang.Thread

The first requirement is checked with a simple escape analysis.

One final feature added to the type system is an escape mechanism to circumvent the analysis when it is too restrictive. As usual, it is the programmer's responsibility to ensure the validity of each use of these escapes.

3 Implementation

`Rccjava`, a prototype type checker, was implemented as an extension to an existing Java front-end. The main additions to the standard Java type checker were the algorithm to compute lock sets, the notion of syntactic equality, and classes parameterized by lock names. Several annotation inference techniques were also incorporated into the implementation in order to reduce the number of annotations required for large programs, including a technique to determine whether an unannotated class is `thread_local` or `thread_shared`.

4 Experimental Results

The prototype implementation was used to check race conditions in a number of programs. The test cases include two representative single classes, the standard `Hashtable` and `Vector` classes, and several large programs, including `java.io`, the Java input/output package (version 1.1) [Jav98]; Ambit, an implementation of a mobile object calculus [Car97]; and an interpreter and run-time environment for WebL, a language for automating web-based tasks [KM98].

Table 1 summarizes the number of annotations required by `rccjava`. The large number of annotations in Hashtable may be attributed to the use of type parameters which require an annotation on each reference to a parameterized type name. The two larger examples required approximately 20 annotations per thousand lines of code. One race was found in the Vector class, and several races were found in the WebL code.

5 Conclusions and Further Work

The initial experiments with `rccjava` indicate that it is a useful tool for detecting races. While more difficult to use than dynamic tools like Eraser [SBN+97], it is not subject to the same coverage problems as those tools. In addition, the annotation overhead is lower than some other static analysis techniques, such as using escjava.

The most important direction for future work is to reduce the annotation requirements. We are currently exploring better annotation inference algorithms and the possibility of using feedback from dynamic tools to help infer annotations.

References

[Car97] Luca Cardelli. Mobile ambient synchronization. Technical Report 1997-013, Digital Systems Research Center, Palo Alto, CA, July 1997.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[FA99] Cormac Flanagan and Martín Abadi. Object types against races. In *Proceedings of CONCUR*, August 1999.

[FF00] Cormac Flanagan and Stephen N. Freund. Type-based race detection for Java. In *Proceedings of ACM Conference on Programming Language Design Implementation*, June 2000.

[Jav98] JavaSoft. Java Developers Kit, version 1.1. http://java.sun.com, 1998.

[KM98] Thomas Kistler and Johannes Marais. WebL – a programming language for the web. *Computer Networks and ISDN Systems*, 30:259–270, April 1998.

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata. Checking Java programs via guarded commands. Technical Report 1999-002, Compaq Systems Research Center, Palo Alto, CA, May 1999.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.