Locking discipline inference and checking

Michael D. Ernst* Alberto Lovato† Damiano Macedonio‡ Fausto Spoto†,‡ Javier Thaine*
*University of Washington, USA †Università di Verona, Italy ‡Julia Srl, Italy
mernst@cs.washington.edu, alberto.lovato@univr.it, damiano.macedonio@juliasoft.com,
fausto.spoto@univr.it, jthaine@cs.washington.edu

Abstract

Concurrency is a requirement for much modern software, but the implementation of multithreaded algorithms comes at the risk of errors such as data races. Programmers can prevent data races by documenting and obeying a locking discipline, which indicates which locks must be held in order to access which data.

This paper introduces a formal semantics for locking specifications that gives a guarantee of race freedom. The paper also provides two implementations of the formal semantics for the Java language: one based on abstract interpretation and one based on type theory. To the best of our knowledge, these are the first tools that can soundly infer and check a locking discipline for Java. Our experiments compare the implementations with one another and with annotations written by programmers.

1. Introduction

Concurrency allows computations to occur inside autonomous *threads*, which are distinct processes that share the same heap memory. Threads increase program performance by scheduling parallel independent tasks on multicore hardware and enable responsive user interfaces [22]. However, concurrency might induce problems such as *data races* (concurrent access to shared data), with consequent unpredictable or erroneous software behavior. Such errors are difficult to understand, diagnose, and reproduce at run time. They are also difficult to prevent: testing tends to be incomplete due to nondeterministic scheduling choices made by the run time, and model-checking scales poorly to real-world code.

The standard approach to prevent data races is to follow a locking discipline while accessing shared data: always hold a given lock when accessing a given shared datum. It is all too easy for a programmer to violate the locking discipline. Therefore, tools are desirable for formally expressing the locking discipline and for verifying adherence to it [9, 28].

The book *Java Concurrency in Practice* [21] (JCIP) proposed the syntax `@GuardedBy` to express a locking discipline and ensure thread-safety. The intention is that when a locking discipline is expressed with `@GuardedBy`, then "No set of operations performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state"; a thread-safe class is one that "use[s] synchronization whenever accessing the [shared, mutable] state". This annotation has been widely adopted; for example, GitHub contains about 35,000 uses of the annotation in 7,000 files.

In an appendix, JCIP proposed a specification for `@GuardedBy`. One of our contributions is our observation that this widely-used specification is ambiguous; indeed, different tools interpret it in different ways [30, 33]. A more important observation is that the specification is incorrect: every interpretation of it permits data races and therefore violates its design goal. Another of our contributions is a formal specification for `@GuardedBy` that satisfies its design goals and prevents data races. (This paper describes the semantics and gives examples, but for reasons of space, the full formal development appears in a technical report [13].) We have also implemented two tools that implement our specification. One tool uses type-checking to validate `@GuardedBy` annotations that are written in Java source code. The other tool uses abstract interpretation to infer valid `@GuardedBy` annotations for unannotated programs. Our techniques are not specific to Java and generalize to other languages. In an experimental evaluation, we compared these tools to one another and to programmer-written annotations. Our evaluation shows that programmers who use the `@GuardedBy` annotation do not necessarily do so consistently with JCIP's rules, and even when they do, their programs still suffer data races.

An informal definition of `@GuardedBy` is that when a programmer writes `@GuardedBy(`E`)` on a program element, then a thread may use the program element only while holding the lock E. Section 2 illustrates important ambiguities in this informal definition. All of these need to be resolved by a formal definition. The most important problem with JCIP's definition is that it provides name protection rather than value protection [8]. Name protection is fine for primitive values, which cannot be aliased in Java, but it allows data races on reference values, which can be freely aliased. The Javadoc for `@GuardedBy` states: "The field or method to which this annotation is applied can only be accessed when holding a particular lock". Value protection is needed in order to prevent data races, not least because the Java Language Specification defines locking in terms of values rather than names [24]. Unfortunately, most tools that check `@GuardedBy` annotations use JCIP's inadequate definition and therefore permit data races. Our definition prevents data races by providing value protection: if a reference r is guarded by E, then for any value v stored in r, v's fields are only accessed while the lock E is held. Checking and inference of this definition requires tracking values v as they flow through the program, because the value may be used through other variables and fields, not necessarily r. Since this is relevant for reference values only, this article considers value protection for reference variables and fields only.

The contributions of this paper include:

- A sound semantics for `@GuardedBy` that guarantees the absence of data races, unlike the interpretation adopted by previous definitions and tools. The semantics is defined in terms of uses of values (objects) rather than uses of names (variables).
- Two independent implementations of the locking discipline semantics for an industrial-strength language, Java: as a modular type analysis and as a whole-program abstract interpretation.
- Case studies of programmers' use of `@GuardedBy` in practice. Pre-

```java
1  public class Fork implements Comparable<Fork> {
2    private static int nextId = 0;
3    private final int id = nextId++;
4    // who is holding the fork, or null if on the table
5    private Philosopher usedBy = null;
6
7    void pickUp(Philosopher philosopher) {
8      this.usedBy = philosopher;
9    }
10
11   void drop() {
12     this.usedBy = null;
13   }
14
15   public int compareTo(@GuardedBy("itself") Fork other) {
16     return id - other.id;
17   }
18
19   public synchronized String toString() {
20     if (usedBy != null)
21       return "fork " + id + " used by " + usedBy.getName();
22     else
23       return "fork " + id + " on the table";
24   }
25 }
```

Figure 1: A fork, possibly held by a philosopher.

vious specifications of the annotation have been vague, and we note places where the programmers' interpretation does not provide a guarantee against data races. Furthermore, we note where programmers have written annotations that are incorrect, illustrating the need for tools like ours.

- A practical comparison of the strengths of the two complementary and independent implementations above: local type-based and global abstract interpretation.

The rest of this paper is organized as follows. Section 2 justifies the need of a locking discipline in concurrent programs. Section 3 describes the checking tool based on a type system. Section 4 presents the inference tool based on abstract interpretation. Section 5 shows experiments with both tools. Section 6 presents related work. Finally, Section 7 concludes.

2. Locking discipline semantics

This section shows how a locking discipline can enforce mutual exclusion and the absence of data races; lays out the design space for a locking discipline semantics; and discusses why such a semantics should provide value protection rather than name protection.

2.1 Dining philosophers example

To illustrate how to specify a locking discipline, consider the traditional dining-philosophers example. More examples are given later. A group of philosophers sit around a table; there is a fork between each pair of philosophers; and each philosopher needs its left and right forks to eat. The locking discipline provides each fork with a lock, and a philosopher must hold the lock in order to use the fork; this guarantees mutual exclusion and the absence of race conditions. (To prevent deadlock, the locks are acquired in increasing order, but that is not a concern of this paper.)

Figure 1 shows Java code for the fork. The fork contains mutable information (which philosopher holds it) in order to demonstrate how a locking discipline can protect the access to its mutable field. A philosopher (Figure 2) is modeled as a thread whose `run` method repeatedly thinks, locks both forks, eats, and unlocks the forks.

In Java, each object is associated with a monitor [24, §17.1]. A `synchronized` statement or method locks the monitor, and exiting

```java
public class Philosopher extends Thread {
  private final @GuardedBy("itself") Fork left;
  private final @GuardedBy("itself") Fork right;

  Philosopher(String name, @GuardedBy("itself") Fork left,
                           @GuardedBy("itself") Fork right) {
    super(name);
    // a fixed ordering avoids deadlock
    if (left.compareTo(right) < 0) {
      this.left = left; this.right = right;
    } else {
      this.left = right; this.right = left;
    }
  }

  public void run() {
    while (true) {
      think();
      synchronized (left) {
        left.pickUp(this);
        synchronized (right) {
          right.pickUp(this);
          eat();
          right.drop();
        }
        left.drop();
      }
    }
  }

  private void think() {
    ...
  }

  @Holding({ "left", "right" })
  private void eat() {
    ...
  }
}
```

Figure 2: A philosopher.

the statement or method unlocks the monitor. Java also provides explicit locks, which our theory and implementations handle.

The `@GuardedBy` type qualifiers express the locking discipline. In the semantics that we will introduce in this article, the type qualifier `@GuardedBy("itself")` on a variable's type states that the variable holds a value *v* whose non-`final` fields are only accessed at moments when *v*'s monitor is locked by the current thread.

Our tools infer and verify the `@GuardedBy` annotations in these figures. The `@GuardedBy("itself")` type qualifiers on fields `left` and `right` guarantee that philosophers use their forks only after properly locking them. The unlocked access to the `final` field `id` on line 16 of fig. 1 does not violate the `@GuardedBy("itself")` specification.

2.2 Design space for locking discipline semantics

Recall the informal definition of `@GuardedBy`: when a programmer writes `@GuardedBy(`*E*`)` on a program element, then a thread may use the program element only while holding the lock *E*. This definition suffers the following ambiguities related to the guard expression *E*.

1. May a definite alias of *E* be locked? Given a declaration `@GuardedBy("lock") Object shared;`, is the following permitted?

```java
Object lockAlias = lock;
synchronized (lockAlias) {
  ... use shared ...
}
```

2. Is *E* allowed to be reassigned while locked? Given a declaration `@GuardedBy("lock") Object shared;`, is either of the following permitted?

```
synchronized (lock) {
  lock = new Object();
  ... use shared ...
}

synchronized (lock) {
  ... use shared ...
  lock = new Object();
}
```

What about other side effects to E? Given a declaration `@GuardedBy ("anobject.field") Object shared;`, are the following permitted?

```
synchronized (anobject.field) {
  foo();   // might side-effect anobject and reassign field
  ... use shared ...
}

synchronized (anobject.field) {
  foo();   // might side-effect but not reassign field
  ... use shared ...
}
```

3. Should E be interpreted at the location where it is defined or at the location where it is used? Given a declaration

```
class C {
  @GuardedBy("this") Object field;
  ...
}
```

are the following permitted?

```
C c;
synchronized (this) {
  ... use c.field ...
}
synchronized (c) {
  ... use c.field ...
}
```

The latter use assumes contextualization, such as viewpoint adaptation [12].

The informal definition suffers the same ambiguities in the interpretation of the program element being guarded. These can be summarized by asking, what is a "use" of the shared program element? Is it any occurrence of the variable name or only certain operations; do uses of aliases count, and are reassignment and side effects permitted? More relevantly, does the `@GuardedBy` annotation specify restrictions on uses of a variable name ("name protection"), or restrictions on uses of values ("value protection")?

Current definitions of `@GuardedBy` do not provide guidance about any of the ambiguities regarding the lock expression. Thus, there is a danger that different tools interpret them differently, including unsound interpretations that do not prevent data races. There is also a danger that programmers will assume a different definition than a tool provides, and thus do not obtain the guarantee they expect.

Current definitions of `@GuardedBy` are clearer about what constitutes a use of the program element — any access to (that is, lexical occurrence of) the name. This definition provides name protection, but unfortunately it does not prevent data races. A program that obeys this locking discipline is not thread-safe and may still suffer data races, as illustrated below. Therefore, any definition that provides name protection is in general incorrect, because it does not satisfy the stated goals of the `@GuardedBy` annotation.

```
public class Observable {
  private @GuardedBy("this") List<Listener> listeners
    = new ArrayList<>();
  public Observable() {}
  public Observable(Observable original) { // copy constr.
    synchronized (original) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    synchronized (this) {
      listeners.add(listener);
    }
  }
  public List<Listener> getListeners() {
    synchronized (this) {
      return listeners;
    }
  }
}
```

Figure 3: An implementation of the observer design pattern in which locking is performed on the container `Observable` object. This implementation suffers data races. The implementation satisfies the name-protection semantics for `@GuardedBy`, but not the value-protection semantics.

2.3 Name protection and value protection

To illustrate the differences between name protection and value protection, consider an implementation of the observer design pattern [19], which is a key part of the model-view-controller and other software architectures. Figures 3 and 4 are patterned after the implementation found in the Java JDK. An `Observable` object allows clients to concurrently register listeners. When an event of interest occurs, a callback method is invoked on each listener.

Synchronization is required to avoid data races. Synchronization in the `register` method and copy constructor prevents simultaneous modifications of the `listeners` list, which might result in a corrupted list or lost registrations. Synchronization is needed in the `getListeners()` method as well, or otherwise the Java memory model would not guarantee the inter-thread visibility of the registrations. In fig. 3, synchronization is performed on the container object, and in fig. 4, synchronization is performed on a field.

Figure 3 satisfies all interpretations of the name protection semantics: every use of `listeners` occurs at a program point where the current thread locks its container.[1] Nevertheless, a data race is possible, since two threads could call `getListeners()` and later access the returned value concurrently. This demonstrates that the name protection semantics does not prevent data races. Figure 3 does not satisfy the value-protection semantics, as expected because those semantics prevent data races, because the return type of `getListeners()` is not compatible with the `return` statement. Figure 3 could be made to satisfy the value-protection semantics by annotating the return type of `getListeners()` as `@GuardedBy("this")`, which would force the client program to do its own locking and would prevent data race.

Figure 4 specifies a different locking discipline. First consider the value-protection semantics. `@GuardedBy("itself")` means that all dereferences of the value of `listeners` occur while the current thread locks that value. The annotation on the return type of `getListeners()` imposes the same requirement on clients of `Observable`. The field `listeners` could have been annotated `@GuardedBy("listeners")`, but the syntax for the return type of `getListeners()` would have been more complex, thus the `@GuardedBy("itself")` syntax. Figure 4 also satisfies the name-protection semantics. Depending on how the se-

[1]The program also satisfies an interpretation of `@GuardedBy` that does not do contextualization or viewpoint adaptation, since the constructor is implicitly synchronized on `this`.

```
public class Observable {
  private @GuardedBy("itself") List<Listener> listeners
    = new ArrayList<>();
  public Observable() {}
  public Observable(Observable original) { // copy constr.
    synchronized (original.listeners) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    synchronized (listeners) {
      listeners.add(listener);
    }
  }
  public @GuardedBy("itself") List<Listener> getListeners() {
    synchronized (listeners) {
      return listeners;
    }
  }
}
```

Figure 4: An implementation of the observer design pattern in which locking is performed on the `listeners` field.

```
1  public class K {
2    private K1 x = new K1();
3    private K2 y = new K2();
4    private K1 z;
5    private K2 w;
6
7    public void m() {
8      z = x;
9      w = new K2();
10     synchronized (z) {
11       y = z.f;
12       w = y;
13     }
14     w.g = new Object();
15   }
16 }
17
18 class K1 {
19   K2 f = new K2();
20 }
21
22 class K2 {
23   Object g = new Object();
24 }
```

var	name protection	value protection
`x`	–	`@GB("itself")`
`y`	`@GB("this.x")`	–
`z`	`@GB("itself")`	`@GB("itself")`
`w`	–	–

Figure 5: Comparison of name-protection and value-protection semantics for `@GuardedBy` (abridged as `@GB`).

mantics handles aliasing and side effects, the semantics may prevent data races for clients of this program.

Figure 4's choice of locking the field rather than the container permits additional flexibility. Consider the following client code:

```
List<Listener> l = new Observable(original).getListeners();
... use l ...
```

At the use of `l`, there is no syntactic handle for the container, and it might even have been garbage-collected.

Regardless of other choices for the semantics of `@GuardedBy`, the name-protection and value-protection variants are not comparable: neither entails the other. In fig. 5, field `x` is declared as `@GuardedBy("itself")`. This annotation holds in the value-protection semantics, since its value is only accessed at line 11 inside a synchronization on itself, but not in name-protection semantics: `x` is used at line 8. Field `y` is `@GuardedBy("this.x")` for name protection but not for value protection: its value is accessed at line 14 via `w`. In some cases the semantics do coincide. Field `z` is `@GuardedBy("itself")` according to both semantics: its name and value are only accessed at line 11, where they are locked. Field `w` is not `@GuardedBy` according to any semantics: its name and value are accessed at line 14.

2.4 Definition of `@GuardedBy`

We can now state our semantics for the `@GuardedBy` annotation. In this article, by *dereference* of a value v we mean the access of a non-`final` field of v. The key idea is that values are protected rather than names, and that dereferences of v are considered uses of v.

Suppose that the type of expression x contains the qualifier `@GuardedBy(`E`)`. A program satisfies the locking discipline if, at program point p where the program dereferences a value that has ever been bound to x, the program holds the lock on the value of expression E. Furthermore, the value of E must not change (in any thread) during the time that the thread holds the lock. The protection is *shallow*, since it applies to the value (reference) in x, not to other references that might be reached from it. There is no restriction on copying values, including passing values as arguments (including as the receiver) or returning values.

This definition resolves the ambiguities noted in section 2.2. A definite alias of the guard expression E is permitted to be locked. The guard expression is not allowed to be reassigned to a different value while locked. Side effects to the guard value are permitted, since they do not affect the monitor. The lock expression undergoes viewpoint adaptation so that it makes sense in the context of use. A use of the program element is a dereference of any value it may hold, regardless of aliasing, reassignment, and side effects.

We have formalized this definition, and also an alternate one that provides name protection, as a structural operational semantics in the style of Plotkin [34]. Our formalization includes a definition of a data race and a proof that our definition prevents data races. For reasons of space, the formal development appears as a technical report [13].

A set of annotations expresses a locking discipline. For an inference tool, a maximal locking discipline is inferred that satisfies the requirements. For a checking tool, the program is verified to check if it satisfies its locking discipline. Every program trivially satisfies the empty locking discipline.

2.5 Definition of `@Holding`

The `@GuardedBy` annotation is sufficient for expressing a locking discipline. Inferring or checking a locking discipline requires reasoning about which locks are held at any given point in the program. Our implementations provide a `@Holding(`E`)` annotation to express these facts explicitly to aid in program comprehension or modular checking.[2] It annotates a method declaration to indicate that when the method is called, the current value of E (possibly viewpoint-adapted) is locked. An example appears on line 60 of fig. 2.

2.6 Other annotations

Our implementations support other features, such as type qualifier polymorphism both without (`@PolyGuardedBy`) and with (`@GuardSatisfied`) a guarantee that all the value's guarding locks are held at the time of the call. For more details about the implementation, see the Lock Checker manual [7].

3. Locking discipline checking

We have implemented a modular static analysis, based on a type system, to verify a programmer-specified locking discipline expressed as Java `@GuardedBy` and `@Holding` annotations. The implementation is publicly available at `http://checker-framework.org/`.

[2] JCIP overloads the name `@GuardedBy` for two distinct purposes as a field annotation and a method precondition. For clarity in this paper, we always refer to the latter as `@Holding`.

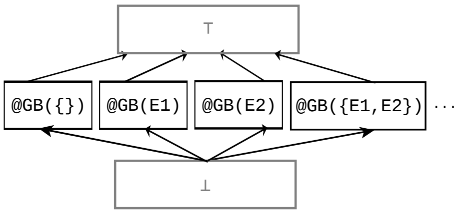

Figure 6: The subtype hierarchy of the locking-discipline type system. `E1` and `E2` are lexically distinct expressions.

If the type-checker issues no warnings for a given program, then it guarantees that the program satisfies the locking discipline; that is, a value that is held in an expression of `@GuardedBy` type in the program is never dereferenced unless the values of all the lock expressions indicated in the `@GuardedBy` annotation are locked by the thread performing the dereference, at the time of the dereference. At run time, a lock expression `E` is held on a given thread at a given time if `java.lang.Thread.holdsLock(E)` evaluates to true on that thread at that time.

Our approach is standard for a static analysis. The goal is to determine facts about values, but the program is written in terms of variables and expressions. Therefore, the analysis computes an approximation (an abstraction) in terms of expressions. Our static analysis simultaneously computes two approximations. (1) The analysis approximates the values that each expression in the program may evaluate to; the abstraction is expressed by the annotations such as `@GuardedBy`. (2) The analysis approximates the locks that the program currently holds; the abstraction is expressed by the annotations such as `@Holding` at method entry and exit.

Both abstractions are sound, so that if the type system approves a program, the program satisfies the locking discipline; however, the abstraction is conservative, so the type system might reject a program that never suffers a race condition at run time.

3.1 Type qualifiers and hierarchy

The type system contains a single parameterized type qualifier, represented by the `@GuardedBy` type annotation. Figure 6 shows the subtype hierarchy. One surprising feature of the type system is that no two `@GuardedBy` annotations are related in the type hierarchy. If *Eset1* $\neq$ *Eset2*, then `@GuardedBy(`*Eset1*`)` and `@GuardedBy(`*Eset2*`)` are siblings in the type hierarchy. It might be expected that `@GuardedBy("x", "y")` *T* is a supertype of `@GuardedBy("x")` *T*. The first type requires two locks to be held, and the second requires only one lock to be held and so could be used in any situation where both locks are held. Our type system conservatively prohibits this in order to prevent type-checking loopholes that would result from aliasing and side effects — that is, from having two references, of different types, to the same data. If our analysis incorporated an analysis of such effects, its type hierarchy could be enriched.

`@Holding` is not part of the type hierarchy because it is a method pre-condition rather than a type qualifier.

3.2 Typing rules

The type system enforces the usual object-oriented subtyping rules at assignments, method calls, overriding method declarations, etc. It also enforces behavioral subtyping [27] for `@Holding` preconditions in overriding method declarations.

Throughout its lifetime, a value is only ever referenced by expressions with the identical `@GuardedBy` type qualifiers (modulo viewpoint adaptation), and this ensures that the value is never dereferenced without the appropriate lock expressions being held.

A field dereference or method invocation type-checks if the receiver type is `@GuardedBy(...)` (that is, not $\top$ or $\perp$), and for field dereferences, all the locks in the type must be held. In addition, a method invocation type-checks only if all the locks mentioned in any `@Holding` precondition are held at the method call.

There are special rules for the polymorphic type qualifier `@GuardSatisfied`: any type may be assigned to it, usually by being passed as an argument to a formal parameter of type `@GuardSatisfied`. Such an expression may always be dereferenced. By contrast, an expression of type `@PolyGuardedBy`, the general polymorphic type, may never be dereferenced. No lock may be released within the dynamic scope of an expression with `@GuardSatisfied` type.

The full type-checking rules are listed and explained in the Lock Checker manual [7], which also explains additional features of the implementation.

3.3 Held lock expressions analysis

The Lock Checker conservatively and flow-sensitively estimates the lock *expressions* that are held at each point in a program. That is, it computes a set of expressions whose locks are definitely held. This process can be viewed as local type inference.

The Lock Checker considers a lock expression held starting when

- the lock expression is used to acquire a lock, or
- a `@Holding` annotation asserts that the lock is held.

The Lock Checker does not track aliasing (different lock expressions that evaluate to the same value); it only considers the exact lock expression used to acquire a lock to be held. The Lock Checker considers a lock expression no longer held when

- the lock is released (explicitly or due to scoping), or
- the lock expression may be side-effected.

The analysis makes conservative approximations about when the lock expression may be side-effected. For example, if a call is made to a method not explicitly annotated as being side-effect free, then the call is considered to side-effect any mutable lock expression.

3.4 Modular analysis and libraries

Our type analysis is modular: it analyzes each procedure in isolation. This makes the analysis scalable and permits separate compilation. A modular analysis requires a summary for each procedure that is called by the one being analyzed. The Lock Checker uses the programmer-written annotations as this specification.

To verify uses of Java's monitor locks, the annotations as described so far are sufficient. Because monitor locks are held throughout the dynamic scope of a `synchronized` statement or invocation of a `synchronized` method, a routine cannot affect the locks held, from the point of view of the caller, and the `@Holding` method annotation can specify a single set of held locks. For explicit locks, the summary needs to be able to indicate different locks held on method entry and method exit. For an analysis focused on deadlocks, the summaries need to be even more complex [40], but deadlock detection and prevention is outside the scope of this paper.

The use of `@GuardSatisfied` as the default annotation means that most code, including external libraries, does not need any annotations except near locking operations. The Lock Checker ships with annotations for relevant parts of the JDK.

4. Locking discipline inference

Our abstract-interpretation-based, whole-program inference uses four static analyses to infer `@GuardedBy` annotations (fig. 7), as described in this section. Inference of `@Holding` is based on similar

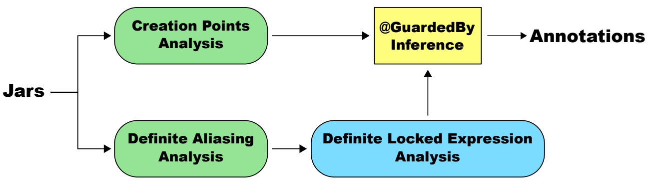

Figure 7: The structure of the abstract interpretation inference of `@GuardedBy` annotations.

lines	variable/field	creation points
8,12,16	`this`	{`Fork@80`}
20,21,23	`this`	{`Fork@80`}
16	`other`	{`Fork@80`}
35,37	`this`	{`Philosopher@84`}
-	`Philosopher.left`	{`Fork@80`}
-	`Philosopher.right`	{`Fork@80`}
21,21,23	arg. to `String.concat`	$\{\kappa, \pi\}$
21	arg. to `Thread.getName`	{`Philosopher@84`}

Figure 8: Julia's creation points analysis of our example. Creation point π stands for a generic creation point inside the Java library code; κ stands for an object held in the constant pool.

techniques but is simpler. Since data race protection holds for guards E that keep their value constant in each thread of the program, our inference tool only infers E made up of final fields and the special variable `itself`, that refers to the same value being protected. In particular, Julia never infers guards E using other variables.

4.1 Creation points analysis

Creation points analysis is an instance of *class analysis* [39]. Julia uses a concretization of Parlsberg and Schwarzbach's class analysis [32, 37]. For each variable and field of reference type, creation points analysis infers an overapproximation of the set of program points where the value bound to that variable or field might have been created. This is a *concretization* since it does not track types of values, but rather their creation point, from which the type can be derived.

Figure 8 shows the result of Julia's creation points analysis at some selected points of the program of figs. 1 and 2 and a client program that creates forks and philosophers and starts the philosopher processes. It reports where the values of the variables at those program points and of the fields of the objects have been created by a `new` statement. For instance, the figure shows that variable `other` at line 16 contains a value of type `Fork` that can only be created in the driver program. The same holds for the values held in fields `left` and `right` of all `Philosopher` objects in memory. Figure 8 also reports the creation points of the objects passed to the Java library, including the implicit argument (receiver) of `getName`, that will be needed later. Note that, in Java bytecode, those arguments are held in stack variables, hence the creation points analysis computes that information. In this simple example, the approximation is always a singleton, but in general it could be a set of creation points. If the line numbers are dropped from column *creation points*, one gets a class analysis. That extra information makes it into a creation points analysis.

4.2 Definite aliasing analysis

This analysis infers, at each program point, definite aliasing between local variables and other local variables or expressions [31]. *Definite* means that that aliasing must hold at the program point, however it is reached. In particular, we are interested in the definite

lines	definite aliases of locked value
19	{`this`}
44	{`this.left`}
46	{`this.right`}

lines	definite aliases of the container of the field
8,12,20	{`this`}

Figure 9: Julia's expression aliasing analysis of our example.

aliases of the values that are used in the `synchronized` statements in our example. Those values are held in a stack variable in bytecode, whose definite aliases are shown in fig. 9, as computed by the analysis. Note that the approximation is semantic. For instance, the analysis would not change if one modified the code at line 44 into

```
Fork f = left;
synchronized (f) ...
```

Later, it will be useful to know the definite aliases of the container E in each field access expression $E.f$ where f is a non-`final` field. Figure 9 provides that information for our example as well.

4.3 Definite locked expressions analysis

This analysis computes, at each program point, a set of expressions that are definitely locked by the current thread at that point. It uses the result of the definite aliasing analysis as a prerequisite. It works as a data flow analysis. Namely, let L_p be a set of definite locked expressions at each given program point p. The analysis builds an inclusion constraint for every statement. In most cases, those constraints just propagate the approximation, such as from line 42 to line 43:

$$L_{42} \supseteq L_{43}$$

Where a synchronization occurs, the set of definitely locked expressions is instead enlarged with the definite aliases of the locked value, as previously computed by the definite alias analysis (fig. 9). This is the case at line 44:

$$L_{44} \cup \{\texttt{this.left}\} \supseteq L_{45}$$

At the end of the synchronization, the analysis builds a constraint that conservatively kills all definitely locked expressions whose type is compatible with that of the unlocked expression, such as at line 50 of our example:

$$L_{50} \setminus \{l \in L_{50} \mid l \text{ has type } \texttt{Fork}\} \supseteq L_{51}$$

The analysis is interprocedural. Namely, definitely locked expressions are renamed at method call, such as at line 45, to implement parameter passing:

$$\left\{ l \begin{bmatrix} a_1 \mapsto \texttt{this} \\ a_2 \mapsto \texttt{philosopher} \end{bmatrix} \;\middle|\; \begin{array}{l} l \in L_{45}, \\ \text{the rec. of } \texttt{pickUp} \\ \text{is definitely aliased to } a_1, \\ \text{the par. of } \texttt{pickUp} \\ \text{is definitely aliased to } a_2 \end{array} \right\} \supseteq L_7$$

A large simplification of the analysis comes from the fact that Java enjoys the property that it does not allow one to write code where a callee unlocks a lock taken by its caller, nor to lock a value without unlocking it before returning to the caller (section 3.4). Hence method calls can be safely approximated as no-operations:

$$L_{43} \supseteq L_{44}$$

lines	definitely locked expressions
8,12,20,21,23	{this}
16,35,37	{}

Figure 10: Julia's definite locked expressions analysis of our example.

At bytecode level, the same property is enforced by the Java Virtual Machine and a violation leads to an `IllegalMonitorStateException`. However, the implementation of this check is not mandatory. For simplicity, we assume that it is implemented or that the analyzed code is generated from Java.

In general, programmers do not modify the value of the expressions that they use as locks, such as `this.left`, and this is the case in our example. However, the analysis copes with the unusual case of field updates that affect the locked expressions. For instance, if line 45 were modified to

```
left = right;
```

and `left` made non-`final`, then Julia would build a constraint that conservatively kills all potentially affected locked expressions:

$$L_{45} \setminus \{l \in L_{45} \mid \texttt{left} \text{ occurs in } l\} \supseteq L_{46}$$

However, the analysis would be unsound if field updates were allowed from a concurrent thread. For this reason, we preferred to keep the analysis sound and, like some other work, only allow `final` fields in the inferred definitely locked expressions.

After inclusion constraints have been built for each pair of consecutive statements and from callers to callees, the analysis computes a fixpoint of the resulting set-constraint. Since this is a definite analysis, a maximal fixpoint is computed. The result for our example, projected on some program points, is shown in fig. 10.

4.4 Inference of the `@GuardedBy` Property

Once the three previous supporting analyses have been performed, Julia infers `@GuardedBy` annotations for fields and method parameters (fig. 7). According to the definition of `@GuardedBy(E)`, this amounts to verifying that the non-`final` fields of all possible values ever held in those fields or parameters are only accessed in a program point where E is locked by the current thread. Julia uses creation points as a conservative approximation of the identity of run-time values. Objects created at distinct creation points must be distinct, while the converse might not hold. Namely, it uses the following algorithm to infer the `@GuardedBy` annotations for a field or parameter x:

1. it uses the creation points analysis to determine an overapproximation C of the creation points of the values ever held in x;
2. it computes the set of program points $A = \{p \mid$ a non-`final` field f is accessed at p as $E_p.f$ and the set $C^p_{E_p}$ of all possible creation points of E_p at p is such that $C^p_{E_p} \cap C \neq \emptyset\}$;
3. for each $p \in A$, it computes the set of expressions
   $$L_p = \{E[E_p \mapsto \texttt{itself}] \mid E \text{ is a definite alias of } E_p \text{ at } p \text{ and } E \text{ is definitely locked at } p\};$$
4. it computes $L = \bigcap_{p \in A} L_p$;
5. it infers the annotations `@GuardedBy("E")` for each $E \in L$ where no variable occurs.

Consider for instance field `left` in fig. 2. According to the creation point analysis (fig. 8), we have $C = \{\texttt{Fork@80}\}$. Access to non-`final` fields occur as `this.usedBy` at lines 8, 12, 20 and we have $C^8_{\texttt{this}} = C^{12}_{\texttt{this}} = C^{20}_{\texttt{this}} = \{\texttt{Fork@80}\}$ (fig. 8). Hence $A = \{8, 12, 20\}$. At those program points, `this` is obviously a definite alias of itself (fig. 9). According to fig. 10, the expression `this` is always locked at 8, 12 and 20. Then $L_p = \{\texttt{this}[\texttt{this} \mapsto \texttt{itself}]\} = \{\texttt{itself}\}$ for each $p \in A$, and hence $L = \{\texttt{itself}\}$. Therefore, Julia infers the annotation `@GuardedBy("itself")` for field `left`.

4.5 Calls to library methods

The algorithm sketched in section 4.4, at its step 2, requires to check all program points A where a non-`final` field in accessed. This includes the program points inside the libraries as well. Hence the inference of `@GuardedBy("itself")` for field `left` above should be corrected by considering in A also the program points outside the application shown in figs. 1 and 2 and the driver program. However, as already sketched in section 3.4, a simplifying and computationally effective alternative solution is to consider only program points A inside the application under analysis, as long as we also include in A the program points where a value is passed to the libraries. That is, point 2 of the algorithm from section 4.4 can be modified to

2. it computes the set of program points $A = \{p$ in the application $\mid$ a non-`final` field f is accessed at p as $E_p.f$ *or an expression E_p is passed as an argument to libraries* and the set $C^p_{E_p}$ of all possible creation points of E_p at p is such that $C^p_{E_p} \cap C \neq \emptyset\}$;

By applying this inference algorithm to all fields and method parameters in figs. 1 and 2 and the driver program, Julia infers the `@GuardedBy` annotations reported in the same figures.

5. Experiments

We performed experiments to understand how programmers currently use `@GuardedBy` and to evaluate the utility of our semantics. Our implementations of the abstract-interpretation-based inference for locking disciplines (section 4) and of the type-system-based checker for locking disciplines (section 3) were written by different people and they share no code, so the fact that they agree provides extra confidence that they correctly implement the semantics.

5.1 Subject programs and methodology

We chose 14 open-source subject programs that use locking (fig. 11). The programmers had partially documented the locking discipline in 5 of them. We counted not only `@GuardedBy` and `@Holding`[3] annotations but also commented annotations and English comments containing the string "guard". The programmers may have used comments in order to obtain the benefits of documenting a locking discipline without adding a compile-time and run-time dependency on the `@GuardedBy` annotation. However, the documented locking discipline may be incorrect because it was not checked by any tool.

We determined a goal set of correct annotations, which are all the annotations whose locking discipline the program obeys. To determine this set, we manually analyzed every annotation written by the programmer or inferred by Julia.[4] We retained every annotation from either set such that the program is guaranteed not to suffer a data race on the annotated program element. Then, we compared the

[3] Our implementations use the type annotation `@GuardedBy` and the method annotation `@Holding`. The programmer-written annotations overload the syntax `@GuardedBy` for both purposes, but for clarity this paper treats the programs as if the programmer had distinguished the two purposes by writing some annotations as `@GuardedBy` and some as `@Holding`.

[4] There might exist other correct annotations that neither Julia, the original programmer, nor we are aware of.

Project	Version	LoC	Programmer-written `@GuardedBy`	Programmer-written `@Holding`	Inference time
BitcoinJ	0.12.2	102458	46	14	238
Derby Engine	10.11.1.1	119594	12	9	4077
Eclipse ECJ	4.4	161701	0	0	924
Guava	18.0	118190	64	72	621
Jetty Server	9.2.6.v20141205	59611	0	0	109
Velocity	1.7	54549	0	0	94
Zookeeper	3.4.6	75475	0	0	118
Catalina	8.0.15	121959	0	0	472
Coyote	8.0.15	71527	1	0	110
Dbcp	8.0.15	53181	16	0	84
Jasper	8.0.15	67380	0	0	105
Jni	8.0.15	32682	0	0	49
Util	8.0.15	42115	0	0	58
Websocket	8.0.15	39928	0	0	75

Figure 11: Subject programs. The last 7 programs are components of Tomcat. *LoC* is the approximate number of lines of code reached by Julia during the analysis. It is the count of the entries in the line number table of each class included in the analysis, plus 3 for each method or constructor. *Inference time* is measured in seconds.

	Goal	Programmer-written					Inference			Type-checking		
			name		value			value			value	
Project	#	#	P%	R%	P%	R%	#	P%	R%	#	P%	R%
BitcoinJ	47	46	87	85	30	30	7	100	15	6	100	86
Derby Engine	16	12	83	63	58	44	6	100	38	6	100	100
Eclipse ECJ	6	0	-	0	-	0	6	100	100	6	100	100
Guava	22	64	19	55	14	41	5	100	23	5	100	100
Jetty Server	1	0	-	0	-	0	1	100	100	1	100	100
Velocity	4	0	-	0	-	0	4	100	100	4	100	100
Zookeeper	5	0	-	0	-	0	5	100	100	5	100	100
Catalina	2	0	-	0	-	0	2	100	100	2	100	100
Coyote	24	1	100	4	0	0	23	100	100	23	100	100
Dbcp	20	16	88	70	56	45	6	100	30	6	100	100
Jasper	7	0	-	0	-	0	7	100	100	7	100	100
Jni	1	0	-	0	-	0	1	100	100	1	100	100
Util	4	0	-	0	-	0	4	100	100	4	100	100
Websocket	9	0	-	0	-	0	9	100	100	9	100	100

Figure 12: Experimental results for `@GuardedBy` annotations. The table gives the number of goal, programmer-written, inferred, and verified annotations, and the precision and recall for each approach along with whether the annotations are interpreted according to the name protection or value protection semantics. Computations whose denominator is zero are reported as "-".

goal annotations to both the programmer-written and the inferred annotations. This comparison was not syntactical: annotations that are conceptually the same or are expressing the same thing are considered equal.

As is standard for an information retrieval problem [36], we report results in terms of precision (number of correct reported annotations divided by total number of reported annotations) and recall (number of correct reported annotations divided by total number of goal annotations). Precision and recall are percent measurements between 0% and 100% inclusive, and larger numbers are better.

5.2 Inference experiments

We used Julia to infer the locking discipline in terms of `@GuardedBy` and `@Holding` with value protection semantics.[5]

Experimental results for `@GuardedBy` annotations appear in fig. 12, and results for `@Holding` appear in fig. 13. Programmers made significant numbers of mistakes (as shown by low precision) and omitted significant numbers of annotations (as shown by low recall). Results for the two annotations are reported separately because there are two possible interpretations for `@GuardedBy` (providing value protection or name protection), but `@Holding` means the same thing in both semantics.

Programmer mistakes. In every program where programmers documented a locking discipline, programmers wrote incorrect annotations that express a locking discipline that the code does not satisfy.

For example, Guava's `LocalCache` and `MapMakerInternalMap` classes incorrectly use `Segment.this` as a guard expression. Julia infers the correct guard `this`.

In other cases, a lock is acquired only around write accesses but not read access to a variable. This can lead to corrupted data being read for data larger than 32 bits, and even for 32-bit data it can lead to inconsistencies between multiple reads of a variable. An example is in the Guava class `SerializingExecutor`: the field `private boolean isThreadScheduled` is annotated as `@GuardedBy("internalLock")`, but it is read without protection at line 135, despite being always written after acquiring the lock.

The most common programmer mistake, however, was creating external aliases to a value. If a reference to a variable's value leaks, then a data race can occur even if a lock is held whenever the variable is read or written. In other words, in the presence of aliasing the value-protection semantics provides no guarantee. This is a natural problem, given the lack of automated checking and even the lack of a mention of the danger of aliasing in references such as JCIP [21]. An example is BitcoinJ field `PaymentChannelClient.conn`. It is always accessed holding a lock inside the class, but the field is initialized with a parameter of a `public` constructor. So there exists an external alias to the object that can potentially be used to access the object without protection.

Programmer omissions. The private BitcoinJ method `PaymentChannelServer.truncateTimeWindow(long)` is inferred to be `@Holding("lock")`, and is indeed called always with `lock` held. Nevertheless, the programmer didn't write the annotation.

In Apache Velocity, a template engine, Julia finds four objects that are `@GuardedBy("itself")`: the field `XPATH_CACHE`, in `XPathCache`, is accessed in a `synchronized(XPATH_CACHE)` block; the field `SimplePool pool`, in `ParserPoolImpl`, uses methods `put` and `get` of `SimplePool`, that modify the object's state inside a `synchronized(this)` block; the receivers of the same two methods are thus guarded as well.

Inference mistakes. Julia's output was correct: its precision is always 100%, just as for any sound tool.

Inference omissions. There are two general reasons that Julia fails to infer a correct programmer-written locking discipline: either (1) the program's correctness is too subtle for Julia to reason about, or (2) the locking discipline is inexpressible in the value-protection semantics.

(1) Julia incompleteness: Julia misses 1 `@Holding` in Derby Engine and 16 in Guava because methods in the `Monitor`, `AbstractService`, and `ServiceManager` classes use complex reasoning, ensuring for instance that a call to a method happens only in flows of execution where the lock is held by the executing thread. At the moment Julia does not understand these tricks.

Julia only allows `itself` and final fields as a guard expression. This is a sufficient but not necessary condition to ensure that the guard expression evaluates to the same value throughout the scope of the guard (section 2.4). Locks used by programmers usually

[5] Julia has two modes and can also infer annotations for name protection, but this article focuses on value protection.

Project	Goal	OGoal	Programmer-written Written	Correct	P%	R%	OR%	Abstract interpretation Inferred	Correct	P%	R%	Type-based analysis Verified	Correct	P%	R%
BitcoinJ	113	45	14	14	100	12	31	113	113	100	100	112	112	100	99
Derby Engine	121	13	9	7	78	6	54	120	120	100	99	119	119	100	99
Eclipse ECJ	1	0	0	0	-	0	100	1	1	100	100	1	1	100	100
Guava	126	45	72	38	53	30	84	110	110	100	87	110	110	100	100
Jetty Server	4	0	0	0	-	0	100	4	4	100	100	4	4	100	100
Velocity	20	0	0	0	-	0	100	20	20	100	100	20	20	100	100
Zookeeper	16	0	0	0	-	0	100	16	16	100	100	16	16	100	100
Catalina	98	0	0	0	-	0	100	98	98	100	100	98	98	100	100
Coyote	13	0	0	0	-	0	100	13	13	100	100	13	13	100	100
Dbcp	18	0	0	0	-	0	0	18	18	100	100	16	16	100	89
Jasper	2	0	0	0	-	0	100	2	2	100	100	2	2	100	100
Jni	1	0	0	0	-	0	100	1	1	100	100	1	1	100	100
Util	4	0	0	0	-	0	100	4	4	100	100	4	4	100	100
Websocket	4	0	0	0	-	0	100	4	4	100	100	4	4	100	100

Figure 13: Experimental results for `@Holding` annotations. This table lists the number of annotations written by the programmer, inferred by Julia, and verified by the Lock Checker, as well as the precision (P%) and recall (R%) percentages. *Goal* is the number of goal annotations. *OGoal* (for "omission-tolerant goal") is the number of goal annotations whose guard expression the programmer used elsewhere, and *OR%* is the programmer recall based on the omission-tolerant goal set.

use variables in guard expressions (sometimes correctly, sometimes incorrectly). As future work, we plan to support the container `this` in guard expressions, which still protects against data races if it is never aliased.

(2) Value protection semantics inflexibility: The only example that seems a genuine value-protection programmer-written annotation and not inferred by Julia is that related to the static field `private static Timer _timer; //@GuardedBy("EvictionTimer.class")`. This field is always accessed in `synchronized` static methods, it never escapes, and is assigned with the code

```
_timer = AccessController.doPrivileged(
  new PrivilegedNewEvictionTimer());
```

The `doPrivileged` method is native, and executes the `run` method of the `PrivilegedNewEvictionTimer` class, that simply returns a new `Timer` object. The guard refers to the class object and is permitted under the value-protection semantics.

5.2.1 Omission-tolerant results

We computed precision and recall for programmer-written `@Holding` annotations, whose results are shown in fig. 13, but we computed two sets for the programmer recall numbers. First, we determined the overall recall, based on the full set of goal annotations. Second, we determined the recall based on a reduced set of goal annotations. The reduced, or omission-tolerant, set contains only `@Holding` annotations whose guard expressions appar in `@GuardedBy` annotations that the programmer wrote. This latter metric considers only locks that the programmer deemed significant enough to document.

The rationale for reporting two different measurements is that there are two different reasons that a `@Holding` annotation might be missing from the programmer-written set:

- The programmer wrote `@GuardedBy` on some variable *v* but omitted `@Holding(v)`. This incomplete specification of the locking discipline for *v* is a programmer error. For example, the programmer correctly annotated the unary method `Wallet.maybeUpgradeToHD` as `@Holding("keychainLock")`, but didn't annotate the no-argument overloaded version.

- The programmer omitted `@GuardedBy` on some variable *v* and also omitted `@Holding(v)`. It is conceivable that the programmer only intended to write specifications for some guarded variables and intentionally omitted the `@GuardedBy` annotation on other variables. The OR% measurement assumes every such omission was intentional, even though the practice is undesirable because someone calling or modifying the code could misuse it.

  For example, Julia infers `@Holding("enumConstantCache")` for Guava's private method `Enums.populateCache`, which needs it for a call to `put`. Indeed, the only invocation of `populateCache` is in a `synchronized (enumConstantCache)` block. Nevertheless, the programmer did not annotate it as `@GuardedBy("enumConstantCache")`.

5.3 Type-checking experiments

5.3.1 Methodology

In order to run the type-checking approach, we performed the following steps for each target program:

1. Remove all the programmer-written `@GuardedBy` and `@Holding` annotations from the program's source code. Leave all programmer-written `@SuppressWarnings` annotations, as they are trusted.

2. Insert the Julia-inferred annotations in the program's source code. These use the value-protection semantics, which is what the Lock Checker verifies.

3. Repeatedly run the Lock Checker and edit the annotations or the code to eliminate the warning (e.g., add a missing annotation), until the Lock Checker issues no more warnings.

A set of `@GuardedBy` and `@Holding` annotations is verified by the Lock Checker if the Lock Checker issues no warnings when only those annotations are present in the source code.

5.3.2 Type-checking results

Figures 12 and 13 describe the annotations that were verified by the Lock Checker.

Overall, there were 5 annotations that were inferred by Julia but could not be verified by the Lock Checker. Only one was due to a difference in the tools' abstraction (static approximations to the semantics). That annotation is `@Holding("#1.lock")` on BitcoinJ's method `Transaction.isConsistent(TransactionBag, boolean)`, where `#1` refers to the first parameter of the method. Since `TransactionBag` is an interface, the expression `#1.lock` is not legal Java (interfaces cannot contain fields) and cannot be processed by the Lock Checker. Julia's whole-program, closed-world analysis determined that every implementing class has a field named `lock`. If the `TransactionBag` interface were modified to include a `getLock()` method, the Lock Checker would be able to resolve the expression `#1.getLock()`.

The other 4 differences were due to limitations of the Java 8 language syntax. Julia inferred a `@GuardedBy` annotation on the receiver parameter of a method declaration of an anonymous inner class, and `@Holding` annotations on the constructors of anonymous inner classes. These parts of the program are implicit — they cannot be written in the Java source code. Therefore, there was no way to communicate this information to the Lock Checker, which reads and verifies annotations in source code. An example is that Julia inferred that the constructor of the anonymous class within method `PaymentChannelClient.incrementPayment` should be annotated with `@Holding("#1.lock")` and its receiver with `@GuardedBy("itself.lock")`.

5.4 Abstract interpretation vs. type-checking

The abstract interpretation approach allows a codebase to be annotated from scratch, producing valuable documentation and also permitting the type-checking approach to verify the absence of bugs relevant to the locking discipline described by these annotations, whereas a pure type-checking approach can only verify annotations already present in the code. In a codebase completely free of annotations, the type-checking approach will issue no warnings, regardless of any bugs that might be present.

Type-checking is more naturally adequate to a compositional analysis. This entails, for instance, that if an annotated program type-checks, then the locking strategy holds for the program and will still hold for future extensions of the program. Hence this will stay true also if a class is extended and methods overridden, as long as the overriding methods conservatively match the annotations of the overridden ones.

A specific observation is the need to extend the syntax of the type system to handle expressions that are not legal Java, such as `#1.lock` as described in section 5.3.2. We also observed the need to extend the Java language itself to make explicit locations that were previously implicit. Java 8 already made an important step toward this by permitting a programmer to optionally write the receiver formal parameter explicitly — a change that was motivated by the desire to write type annotations on the receiver [26]. That capability was essential in our case studies, where many annotations on type qualifiers were inferred.

It is interesting that these new syntax limitations became clear only with the integration with an inference tool that inferred all possible guards, even though the Lock Checker has been publicly available in the Checker Framework distribution since June 2009 (over 70 monthly releases before the current writing). We speculate that this is an issue of "out of sight, out of mind": Java programmers didn't think about annotations on those locations and so they did not specify them. The programmers also may have simply suppressed type-checking errors related to those locations.

6. Related work

Despite the need for a formal specification for reasoning about Java's concurrency and for building verification tools [9, 28, 4], we are not aware of any previous tool built upon a formalization of the semantics of Java's concurrency annotations.

Warlock [38] was an early tool that checked user-written specifications of a locking discipline, including annotations for variable guards and locks held on entry to functions. ESC/Modula-3 [11] and ESC/Java [17] provided similar syntax and checked them via verification conditions and automated theorem-proving, an approach also applied to other concurrency problems [16]. All these tools are unsound and do checking rather than inference.

Our approach is a pure, flow-sensitive type system. A heavier-weight alternative is a type-and-effect system, which can prevent not just race conditions but also deadlocks [14, 1]. It can associate guards not just with variables but also with specific side effects [29].

Most approaches, including ours, explicitly associate each variable with a lock that guards access to it. An alternative is to use ownership types and make each field fields are protected by its owner, which is not necessarily the object that contains it [6, 10]. This approach is somewhat less flexible, but it can leverage existing object encapsulation specifications and can be extended to prevent deadlocks [5].

These concepts can also be expressed using fractional permissions [41]. Grossman [25] extended type-checking for data races to Cyclone, a lower-level language, but did not implement or experimentally evaluate it.

Previous inference techniques include unsound dynamic inference of lock types [2, 35] and Sound inference via translation to propositional satisfiability, for most of Java [15]. By contrast, our approach is sound, more precise, and more scalable. Improving our aliasing analysis [3] would improve the recall of our implementations.

Type systems have been applied to other concurrency problems, such as atomicity [18]. Deadlocks are generally handled by imposing a lock ordering: if all locks are acquired in the given order, then no deadlock occurs. Recent type systems permit the lock ordering not to be static throughout the program [20, 23].

7. Conclusion

A locking discipline makes concurrent programming manageable. Used properly, it guarantees the lack of data races. Used improperly (with vague definitions or no mechanical checking), it is error-prone at best and misleading at worst.

Case studies of significant real-world code shows that programmers often make mistakes: they write locking-discipline specifications that their programs do not follow, and they fail to write ones that their programs do follow. Programmers seem to often assume a name-protection semantics for the locking-discipline specifications, not realizing that it does not provide a guarantee against data races. We have shown that the value-protection semantics is more restrictive and possibly harder to use; but the more accurate documentation and the reduction in bugs should be worth it.

Two popular analysis approaches are abstract interpretation and type-checking. We have implemented one of each type of tool. Each tool is based on a firm semantic foundation that guarantees no data races (modulo standard assumptions, such as regarding reflection). Abstract interpretation is generally assumed to be more powerful and precise, but we have quantified the differences, leading to insights about the analysis approaches. In the future, others will be able to make a more informed decision between the approaches. The two tools have completely independent implementations, and the fact that their results agree, up to differences in their underlying analysis, lends confidence to our semantics and implementations. The inference can also find bugs when a value is usually but not always accessed when a lock is taken. Our tools are scalable, robust, and publicly available, so programmers can take advantage of them today.

References

[1] M. Abadi, C. Flanagan, and S. N. Freund. Types for safe locking: Static race detection for Java. *ACM Transactions on Programming Languages and Systems*, 28(2):207–255, March 2006.

[2] R. Agarwal, A. Sasturkar, and S. D. Stoller. Type discovery for Parameterized Race-Free Java. Technical Report DAR-04-16, Computer Science Department, SUNY at Stony Brook,

September 2004.

[3] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI 2003, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, pages 129–140, San Diego, CA, USA, June 9–11, 2003.

[4] D. Bogdanas and G. Rosu. K-Java: A complete semantics of Java. In *ACM SIGPLAN-SIGACT POPL*, pages 445–456, Mumbai, India, 2015.

[5] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2002)*, pages 211–230, Seattle, WA, USA, October 28–30, 2002.

[6] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001)*, pages 56–69, Tampa Bay, FL, USA, October 14–18, 2001.

[7] *The Checker Framework Manual: Custom pluggable types for Java*, version 1.9.4 edition, August 2015.

[8] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 48–64, Vancouver, BC, Canada, October 20–22, 1998.

[9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby. Expressing Checkable Properties of Dynamic Systems: the Bandera Specification Language. *STTT*, 4(1):34–56, 2002.

[10] D. Cunningham, S. Drossopoulou, and S. Eisenbach. Universes for race safety. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP)*, pages 20–51, Lisbon, Portugal, September 3, 2007.

[11] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, December 18, 1998.

[12] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, pages 28–53, Berlin, Germany, August 1–3, 2007.

[13] M. Ernst, D. Macedonio, M. Merro, and F. Spoto. Semantics for locking specifications. Technical Report UW-CSE-15-09-01, University of Washington Department of Computer Science and Engineering, Seattle, WA, USA, September 2015.

[14] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *PLDI 2000, Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 219–232, Vancouver, BC, Canada, June 18–23, 2000.

[15] C. Flanagan and S. N. Freund. Type inference against races. In *Proceedings of the Eleventh International Symposium on Static Analysis, SAS 2004*, pages 116–132, Verona, Italy, August 26–28, 2004.

[16] C. Flanagan, S. N. Freund, and S. Qadeer. Thread-modular verification for shared-memory programs. In *11th European Symposium on Programming*, pages 262–277, Grenoble, France, April 2002.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002, Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245, Berlin, Germany, June 17–19, 2002.

[18] C. Flanagan and S. Qadeer. A type and effect system for atomicity. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 338–349, New Orleans, LA, January 15–17, 2003.

[19] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[20] P. Gerakios, N. Papaspyrou, and K. Sagonas. A type and effect system for deadlock avoidance in low-level languages. In *TLDI 2011: The sixth ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 15–28, Austin, TX, USA, January 25, 2011.

[21] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java Concurrency in Practice*. Addison-Wesley, 2006.

[22] A. Göransson. *Efficient Android Threading*. O'Reilly Media, June 2014.

[23] C. S. Gordon, M. D. Ernst, and D. Grossman. Static lock capabilities for deadlock freedom. In *TLDI 2012: The seventh ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 67–78, Philadelphia, PA, USA, January 28, 2012.

[24] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley. *The Java Language Specification*. Addison Wesley, Boston, MA, Java SE 8 edition, 2014.

[25] D. Grossman. Type-safe multithreading in Cyclone. In *TLDI 2003: The ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 13–25, New Orleans, LA, USA, January 18, 2003.

[26] JSR 308 Expert Group. Annotations on Java types. `http://download.oracle.com/otndocs/jcp/annotations-2014_01_08-pfd-spec/`, January 8, 2014. Proposed Final Draft.

[27] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[28] B. Long and B. W. Long. Formal Specification of Java Concurrency to Assist Software Verification. In *IPDPS*, page 136, Nice, France, April 2003.

[29] Y. Lu, J. Potter, and J. Xue. Structural lock correlation with ownership types. In *22nd European Symposium on Programming*, pages 391–410, Rome, Italy, March 19–22, 2013.

[30] NASA. Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf`.

[31] D. Nikolic and F. Spoto. Definite Expression Aliasing Analysis for Java Bytecode. In *9th International Colloquium on Theoretical Aspects of Computing (ICTAC 2012)*, pages 74–89, Bangalore, India, September 2012.

[32] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 146–161, Phoeniz, AZ, USA, October 1991.

[33] V. Pech. Concurrency is Hot, Try the JCIP Annotations. `http://jetbrains.dzone.com/tips/concurrency-hot-try-jcip`, February 2010.

[34] G. D. Plotkin. A structural approach to operational semantics. *Journal of Logic and Algebraic Progamming*, 60-61:17–139, July–December 2004.

[35] J. Rose, N. Swamy, and M. Hicks. Dynamic inference of polymorphic lock types. In *Workshop on Concurrency and Synchronization in Java Programs (CSJP)*, pages 18–25, St. John's, Newfoundland, Canada, July 25–26, 2004.

[36] G. Salton. *Automatic Information Organization and Retrieval*. McGraw-Hill, 1968.

[37] F. Spoto and T. P. Jensen. Class Analyses as Abstract Interpretations of Trace Semantics. *ACM Trans. Program. Lang. Syst.*, 25(5):578–630, 2003.

[38] N. Sterling. Warlock: A static data race analysis tool. In *Proceedings of the Winter 1993 USENIX Conference*, pages 97–106*, San Diego, CA, USA, January 5–29, 1993.

[39] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, pages 281–293, Minneapolis, MN, USA, October 15–19, 2000.

[40] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.

[41] Y. Zhao and J. Boyland. Assuring lock usage in multithreaded programs with fractional permissions. In *ASWEC'09: 20th Australian Software Engineering Conference*, pages 277–286, Gold Coast, Australia, April 15–17, 2009.