Semantics for Locking Specifications (extended abstract)

Michael D. Ernst[1], Damiano Macedonio[2], Massimo Merro[2], and Fausto Spoto[2]

[1] Computer Science & Engineering, University of Washington, WA, USA
[2] Dipartimento di Informatica, Università degli Studi di Verona, Italy

Abstract. Lock-based synchronization disciplines, like Java's `@GuardedBy`, are widely used to prevent concurrency errors. However, their semantics is often expressed informally and is consequently ambiguous. This article highlights such ambiguities and overcomes them by formalizing two possible semantics of `@GuardedBy`, using a reference operational semantics for a core calculus of a concurrent Java-like language. It also identifies when such annotations are actual guarantees against data races. Our work aids in understanding the annotations and supports the development of sound tools that verify or infer them.

1 Introduction

Data races are common errors in concurrent programs which occur when a shared data structure is manipulated by different threads, without synchronization, with consequent unpredictable or erroneous software behavior. Such errors are difficult to understand, diagnose, and reproduce. They are also difficult to prevent: testing tends to be incomplete due to nondeterministic scheduling choices made by the runtime, and model-checking scales poorly to real-world code.

The simplest approach to prevent data races is to follow a *lock-based synchronization discipline*: always hold a given lock when accessing a shared data structure. Since a lock can be held by at most one thread at any time, this discipline ensures data-race freedom. However, it is easy to violate a locking discipline, so tools that verify adherence to the discipline are desirable. These tools require a *specification language* to express the intended locking discipline. The focus of this paper is on the formal definition of such a specification language, its semantics, and the guarantees that it gives against data races.

In Java, the most popular specification language for expressing a locking discipline is the `@GuardedBy` [16]. Informally, if the programmer annotates a field f as `@GuardedBy(`E`)` then a thread may access f only while holding the monitor corresponding to the *guard expression* E. The `@GuardedBy` annotation was proposed by Goetz [12] as a documentation convention only, without tool support. It has been adopted by practitioners; GitHub contains about 35,000 uses of the annotation in 7,000 files of distinct projects. Tool support now exists in Java PathFinder [18], the Checker Framework [10], `Houdini/rcc` [1], IntelliJ [22], and Julia [23].

All of these tools, except for [1], rely on the previous informal definition of `@GuardedBy(`E`)` [16]. However, such an informal description is prone to many ambiguities. Suppose a field f is annotated as `@GuardedBy(`E`)`, for some guard expression E. (1) The definition above does not clarify how an occurrence of the self-reference variable `this` in E should be interpreted in client code; this actually depends on the context in which f is accessed. (2) It does not define what an *access* is. (3) It does not say whether a synchronization block must use the guard expression E as written in the annotation or whether a different expression that evaluates to the same value is permitted. (4) It does not indicate whether the lock that must be taken is the value of E at the time of synchronization or that at the time of field access: side effects on E might make a difference here. (5) It does not clarify whether the lock on the guard E must be taken when accessing the field *named* f or the *value* bound to f. The latter ambiguity is particularly important. The interpretation of `@GuardedBy` based on names is adopted in most tools appearing in the literature [18, 22, 23, 1], whereas the interpretation based on values seems to be less common [10, 23]. As a consequence, it is interesting to understand whether and how these two possible interpretations actually protect against data races on the annotated field.

The main contribution of this article is the formalization of two different semantics for annotations of the form `@GuardedBy(`E`)` *`Type`* `f`: a *name-protection* semantics, in which accesses to the annotated *field* `f` need to be synchronized on the guard expression E, and a *value-protection* semantics, in which accesses to a *value* referenced by `f` need to be synchronized on E. The semantics clarify all the above ambiguities, so that programmers and tools know what those annotations mean and which guarantees they entail. We then show that both the name-protection and the value-protection semantics can protect against data races under proper restrictions on the variables occurring in the guard expression. The name-protection semantics requires further constraints — the protected field must not be aliased and the guard expression E must be final (its value must not change during program execution).

Finally, we have used our formalization to extend the Julia static analyzer [23] to check and infer `@GuardedBy` annotations in arbitrary Java code. Our companion paper [11] presents the implementation in Julia together with experiments that show how the tool scales to large real software. Julia allows the user to select either name-protection or value-protection. For instance, in the code of Google Guava [13] (release 18), the programmer put 64 annotations on fields; 17 satisfy the semantics of name protection; 9 satisfy the semantics of value protection; the others do no satisfy any of the two. Julia automatically infers all annotations for name-protection and 5 of those that satisfy the value-protection semantics.

In this extended abstract proofs are omitted; for details see the Appendix.

Outline. Sec. 2 discusses the informal semantics of `@GuardedBy` by way of examples. Sec. 3 introduces a calculus for a concurrent fragment of Java. Sec. 4 gives formal definitions for both the name-protection and value-protection semantics in our calculus. Sec. 5 shows which guarantees they provide against data races. Sec. 6 describes the implementation in Julia. Sec. 7 discusses related work and concludes.

2 Informal Semantics of `@GuardedBy`

This section illustrates the use of `@GuardedBy` by example. Fig. 1 defines an observable object that allows clients to concurrently register listeners. Registration must be synchronized to avoid data races: simultaneous modifications of the `ArrayList` might result in a corrupted list or lost registrations. Synchronization

Fig. 1 This code has a potential data race due to aliasing of the `listeners` field.

```
1  public class Observable {
2    private @GuardedBy(this) List<Listener> listeners = new ArrayList<>();
3    public Observable() {}
4    public Observable(Observable original) { // copy constructor
5      synchronized (original) {
6        listeners.addAll(original.listeners);
7      }
8    }
9    public void register(Listener listener) {
10     synchronized (this) {
11       listeners.add(listener);
12     }
13   }
14   public List<Listener> getListeners() {
15     synchronized (this) {
16       return listeners;
17 } } }
```

is needed in the `getListeners()` method as well, or otherwise the Java memory model does not guarantee the inter-thread visibility of the registrations.

The interpretation of the `@GuardedBy(this)` annotation on field `listeners` requires resolving the ambiguities explained in Sec. 1. The intended locking discipline is that every use of `listeners` should be enclosed within a construct `synchronized` (*container*) `{...}`, where *container* denotes the object whose field `listeners` is accessed (ambiguities (1) and (2)). For instance, the access `original.listeners` in the copy constructor is enclosed within `synchronized (original) {...}`. This contextualization of the guard of synchronized blocks is not clarified in any informal definitions of `@GuardedBy` (ambiguity (3)). Furthermore, it is not clear if a definite alias of `original` can be used as synchronization guard at line 5. It is not clear if `original` would be allowed to be reassigned between lines 5 and 6 (ambiguity (4)). Note that the copy constructor does not synchronize on `this` even though it accesses `this.listeners`. This is safe so long as the constructor does not leak `this`. This paper assumes that an escape analysis [6] has established that constructors do not leak `this`. The `@GuardedBy(this)` annotation on field `listeners` suffers also from ambiguity (5): it is not obvious whether it intends to protect the *name* `listeners` (*i.e.*, the name can be only used when the lock is held) or the value currently bound to `listeners` (*i.e.*, that value can be only accessed when the lock is held). Another way of stating this is that `@GuardedBy` can be interpreted as a *declaration annotation* (a restriction on uses of a name) or as a *type annotation* (a restriction on values associated to that name).

The code in Fig. 1 seems to satisfy the name-protection locking discipline expressed by the annotation `@GuardedBy(this)` for field `listeners`: every use of `listeners` occurs in a program point where the current thread locks its container, and we conclude that `@GuardedBy(this)` name-protects `listeners`. Nevertheless, a data race is possible, since two threads could call `getListeners()` and later access the returned value concurrently. This cannot be avoided when critical sections *leak* guarded data. More generally, name protection does not prevent data races if there are aliases of the guarded name (such as a returned value in our example) that can be used in an unprotected manner. The value-protection semantics of `@GuardedBy` is not affected by aliasing as it tracks accesses to the value referenced by the name, not the name itself.

Fig. 2 Value protection prevents data races; see `itself` in the guard expression.

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

Any formal definition of `@GuardedBy` must result in mutual exclusion in order to ban data races. If f is `@GuardedBy(`E`)`, then at any program point where a thread accesses f (or its value) that thread must hold the lock on E. Let $\mathcal{P}$ be the set of such program points where f is accessed. Mutual exclusion requires two conditions: (i) E can be evaluated at all program points $P \in \mathcal{P}$, and (ii) these evaluations, at a given instant of time, always yield the same value at all $P \in \mathcal{P}$.

Point (i) is syntactic and related to the fact that E cannot refer to variables or fields that are not always in scope or visible at all program points in $\mathcal{P}$. This problem exists for both name protection and value protection, but is more significant for the latter, that is meant to protect values that flow in the program through arbitrary aliasing. For instance, the annotation `@GuardedBy(listeners)` cannot be used for value protection in Fig. 1, since the name `listeners` is not visible outside class `Observable`, but its value flows outside that class through method `getListeners()` and must be protected also if it accessed there. For this, we support a special variable `itself` that refers to the current value of f. For instance, for value protection, the code in Fig. 1 should be rewritten as in Fig. 2.

Point (ii) is semantical and related to the intent of providing a guarantee of mutual exclusion. This point bans the use of a variable in E that, although in scope and visible at every program point in $\mathcal{P}$, might have different values at distinct program points. We need this requirement for both semantics, but it translates into two distinct constraints on the guard E for each semantics. As we will see in Sec. 5, a simple restriction that allows us to satisfy (ii) is to allow only variables `itself`, pointing to the value of the guarded field itself, and variable `this`, pointing to the container of the guarded field, when that container can be identified unambiguously. These two variables have the same value at every program points and this is why we only allow them in E. Moreover, in the semantics for name protection we will require that E only refers to final fields, since the instant of time when the field name is locked and that when the field value gets dereferenced might be arbitrarily away. This latter restriction is not needed for the semantics for value protection, since it requires that a thread holds the lock on the value of a field exactly when that value is accessed.

Thus, in Fig. 2 value protection bans data races on `listeners` since the guard `itself` can be evaluated everywhere (point (i)) and always yields the value of

Fig. 3 Mutable guard expressions may lead to data races.

```
public class Observable {
  private @GuardedBy(guard) List<Listener> listeners = new ArrayList<>();
  private Object guard1 = new Object();
  private Object guard2 = new Object();
  public Observable() {}
  public Observable(Observable original) { // copy constructor
    Object guard = guard1;
    synchronized (guard) {
      listeners.addAll(original.listeners);
    }
  }
  public void register(Listener listener) {
    Object guard = guard2;
    synchronized (guard) {
      listeners.add(listener);
} } }
```

`listeners` itself (point (ii)). Here, the `@GuardedBy(itself)` annotation requires all accesses to the value of `listeners` to occur only when the current thread locks the same monitor — even outside class `Observable`, in a client that operates on the value returned by `getListeners()`. In Fig. 3, instead, field `listeners` is `@GuardedBy(guard)` according to both name protection and value protection, but the value of `guard` is distinct at different program points: no mutual exclusion guarantee exists and data races on `listeners` occur.

3 A Core Calculus for Multithreaded Java

Our calculus is a variant of RaceFreeJava [1]. We begin with some preliminary notions. A *partial function* f from A to B is denoted by $f : A \rightharpoonup B$, and its *domain* is $\mathrm{dom}(f)$. The symbol ϕ denotes the empty function; $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$ denotes a function f such that $f(v_i) = t_i$ for $i \in 1..n$; $f[v_1 \mapsto t_1, \ldots, v_n \mapsto t_n]$ denotes the update of f, where $\mathrm{dom}(f)$ is enlarged for every i such that $v_i \notin \mathrm{dom}(f)$. A *poset* is a structure $\langle A, \leq \rangle$ where A is a set and $\leq$ is a partial order. For $a \in A$, we define $\uparrow a \stackrel{def}{=} \{a' : a \leq a'\}$. A *chain* is a totally ordered poset.

3.1 Syntax

Letters $f, g, x, y, \ldots$ range over a set of variables *Var* that includes `this`. Variables identify either local variables in methods or instance variables (*fields*) of objects. Symbols $m, p, \ldots$ range over a set *MethodName* of method names. There is a set *Loc* of memory locations, ranged over by l. Symbols $\kappa, \kappa_0, \kappa_1, \ldots$ range over a set of *classes* (or *types*) *Class*, ordered by a *subclass relation* $\leq$; $\langle \mathit{Class}, \leq \rangle$ is a poset such that for all $\kappa \in \mathit{Class}$ the set $\uparrow\kappa$ is a finite chain. If $m \in \mathit{MethodName}$, then $\kappa.m$ denotes the implementation of m inside class κ, if any. The partial function $\mathit{lookup}(\,) : \mathit{Class} \times \mathit{MethodName} \rightharpoonup \mathit{Class}$ formalizes *Java's dynamic method lookup*, *i.e.* the runtime process of determining the class containing the implementation of a method on the basis of the class of the receiver object: $\mathit{lookup}(\kappa, m) \stackrel{def}{=} \mathit{min}(\uparrow \kappa.m)$ if $\uparrow \kappa.m \neq \emptyset$ and is undefined otherwise, where $\uparrow\kappa.m \stackrel{def}{=} \{\kappa' \in \uparrow\kappa \mid m \text{ is implemented in } \kappa'\}$ is a finite chain since $\uparrow\kappa.m \subseteq \uparrow\kappa$.

Let us provide the syntax of our core language.

Fig. 4 Running example.

```
public class K {
  private C x = new C();
  private C y = new C();
  private C z = new C();
  private Object h = new Object();
  public void m() {
    this.z = this.x;
    synchronized (this.z) {
      this.h = this.z.f;
      this.z = this.y;
    }
    this.z.f = new Object();
  }
}
class C {
  Object f = new Object();
}
```

	name protection	value protection
field `x`	–	`@GuardedBy(itself)`
field `y`	`@GuardedBy(this.x)`	–
field `z`	–	–

$$E ::= x \quad | \quad l \quad | \quad E.f \quad | \quad \kappa\langle f_1 = E_1, \ldots, f_n = E_n \rangle$$

$$C ::= \texttt{let}\ x = E\ \texttt{in}\ C \quad | \quad E.f := E \quad | \quad C; C \quad | \quad \texttt{skip} \quad | \quad E.m(\,) \quad | \\ \texttt{spawn}\ E.m(\,) \quad | \quad \texttt{sync}(E)\{C\} \quad | \quad \texttt{monitor\_enter}(l) \quad | \quad \texttt{monitor\_exit}(l)$$

Expressions Exp, ranged over by E, are given by variables, locations, field accesses, and object allocation, $\kappa\langle f_1 = E_1, \ldots, f_n = E_n \rangle$, to create an object of class κ and initialize each field f_i to the value of E_i. For simplicity, we only have classes and no primitive types, so the only possible *values* are locations.

Commands Com are ranged over by C. *Method bodies*, ranged over by B, are `skip`-terminated commands. Formally, $B ::= \texttt{skip} \mid C; \texttt{skip}$. The set of classes is $\mathit{Class} \stackrel{\text{def}}{=} \{\kappa : \mathit{MethodNames} \rightharpoonup B \mid \text{dom}(\kappa) \text{ is finite}\}$. The binding of fields to their defining class is not relevant in our formalization. Given a class κ and a method name m, if $\kappa(m) = B$ then κ implements m with body B. With "`this`" we denote the standard self-reference variable. In our syntax, self-reference binding is implicit; methods have no formal parameters and/or return value.

Terms containing locations (such as $l.f$ or $\texttt{monitor\_enter}(l)$) cannot be used by the programmer: they are introduced by the semantics.

We write $U\{^{E_1}/_{x_1}, \ldots, ^{E_n}/_{x_n}\}$ to denote the capture-free substitution of expressions E_i, for all free occurrences of x_i, within $U \in \mathit{Com} \cup \mathit{Exp}$, for all $i \in 1..n$.

A *program* is a finite set of classes including a special class *Main* that only defines a method *main* where the program starts: $\mathit{Main} \stackrel{\text{def}}{=} \{\mathit{main} \mapsto B_{\mathit{main}}\}$.

Example 1. Fig. 4 gives our *running example* in Java. In our core language, the body of method `m` is translated as follows: $B_{\texttt{m}}$ = `this.z := this.x; sync(z) {this.h := this.z.f; this.z := this.y}; this.z.f := Object⟨⟩; skip`, with classes $\texttt{K} \stackrel{\text{def}}{=} \{\texttt{m} \mapsto B_{\texttt{m}}\}$, $\texttt{C} \stackrel{\text{def}}{=} \phi$, and $\texttt{Object} \stackrel{\text{def}}{=} \phi$.

3.2 Semantic Domains

Threads, ranged over by T, are constituted by a sequence of commands C and a set $\mathcal{L} \subseteq \mathit{Loc}$ of locations that it currently locks, formally $T ::= \lceil C \rceil \mathcal{L}$. We use letters P and Q to denote a *pool of threads*. Formally, $P, Q ::= T^*$.

A *running program* consists of a pool of threads that share a memory. Initially, a single thread runs the *main* method. The `spawn` $E.m(\,)$ command adds a new thread to the existing ones. A *memory* μ maps a finite set of already allocated memory locations into *objects*.

An object o is a triple containing the object's class, the object's state binding its fields to their corresponding values, and a lock, *i.e.*, an integer counter incremented whenever a thread locks the object (locks are re-entrant).

Definition 1. Objects *and* memories *are defined as* $Object \stackrel{def}{=} Class \times State \times \mathbb{N}$ *and* $Memory \stackrel{def}{=} \{\mu : Loc \rightharpoonup Object \mid \mathrm{dom}(\mu)$ *is finite*$\}$*, with selectors* $class(o) \stackrel{def}{=} \kappa$*,* $state(o) \stackrel{def}{=} \sigma$ *and* $lock^{\#}(o) \stackrel{def}{=} n$*, for every* $o = \langle \kappa, \sigma, n \rangle \in Object$*. We also define* $o[f \mapsto l] \stackrel{def}{=} \langle \kappa,\, \sigma[f \mapsto l], n\rangle$ *and* $lock^{+}(o) \stackrel{def}{=} \langle \kappa,\, \sigma, n{+}1\rangle$ *and* $lock^{-}(o) \stackrel{def}{=} \langle \kappa,\, \sigma, \max(0, n{-}1)\rangle$*.*

The *evaluation of an expression* E in a memory μ, written $[\![E]\!]^{\mu}$, yields a pair $\langle l, \mu' \rangle$, where l is the runtime value of E, and μ' is the memory resulting from the evaluation of E. Given a pair $\langle l, \mu \rangle$ we define $loc\,(\langle l, \mu \rangle) = l$ and $mem\,(\langle l, \mu \rangle) = \mu$.

Definition 2 (Evaluation of Expressions). *The evaluation function has the type* $[\![\ ]\!] : (Exp \times Memory) \rightharpoonup (Loc \times Memory)$ *and is defined as:*

$$[\![l]\!]^{\mu} \stackrel{\text{def}}{=} \langle l,\, \mu \rangle \qquad [\![E.f]\!]^{\mu} \stackrel{\text{def}}{=} \langle state(\mu'(l))(f),\, \mu' \rangle,\ \textit{where}\ [\![E]\!]^{\mu} {=} \langle l,\, \mu' \rangle$$

$$[\![\kappa\langle f_1{=}E_1, .., f_n{=}E_n\rangle]\!]^{\mu} \stackrel{\text{def}}{=} \langle l,\, \mu_n[l \mapsto \langle \kappa,\, \sigma, 0\rangle]\rangle,\ \textit{where}$$

(1) $\mu_0 = \mu$ *and* $\langle l_i,\, \mu_i \rangle = [\![E_i]\!]^{\mu_{i-1}}$*, for* $i \in [1..n]$

(2) l *is* fresh *in* μ_n*, that is* $\mu_n(l) \uparrow$

(3) $\sigma \in State$ *is such that* $\sigma(f_i) = l_i$ *for* $i \in [1..n]$*, while* $\sigma(y)\uparrow$ *elsewhere.*

We assume that $[\![\ ]\!]$ *is undefined if any of the function applications is undefined.*

In the evaluation of the object creation expression, a fresh location l is allocated and bound to an unlocked object whose environment σ binds its fields to the values of the corresponding initialization expressions.

3.3 Structural Operational Semantics

We define a *reduction semantics* on *configurations* of the form $\langle P,\, \mu \rangle$. We write $\langle P,\, \mu \rangle \rightarrow \langle P',\, \mu' \rangle$ for representing an execution step. We write $\rightarrow^{*}$ to denote the reflexive/transitive closure of $\rightarrow$, and $\rightarrow^{i}$ for i consecutive reduction steps.

Table 1 deals with sequential commands. Rule [seq] assumes that the first command is not of the form `spawn` $E.p(\ )$; this case is treated separately. In rule [invoc] the receiver E is evaluated and the method implementation is looked up from the dynamic class of the receiver. The body of the method is then executed after binding `this` to the receiver.

Table 2 focuses on concurrency and synchronization. The spawn of a new method is similar to a method call, but the method body runs in its own new thread with an initially empty set of locked locations. Note that if a sequence of commands starts with a `spawn` then rule [spawn] is the only rule which can be used. In rule [sync] the location l associated to the guard E is computed; the computation can proceed only if a lock action is possible on l. The lock will be released only at the end of the critical section C. Rule [acquire-lock] models the entering of the monitor of an unlocked object. Rule [reentrant-lock] models Java's *lock reentrancy*. Rule [decrease-lock] decreases the lock counter of an object that

Table 1 Structural operational semantics for sequential commands.

$$\frac{[\![E]\!]^{\mu} = \langle l,\, \mu' \rangle}{\langle \lceil \texttt{let}\ x = E\ \texttt{in}\ C \rceil \mathcal{L},\, \mu \rangle \ \rightarrow \langle \lceil C\{^{l}/_{x}\} \rceil \mathcal{L},\, \mu' \rangle}\ [\text{let}]$$

$$\frac{[\![E]\!]^{\mu} = \langle l,\, \mu' \rangle \quad [\![E']\!]^{\mu'} = \langle l',\, \mu'' \rangle \quad o = \mu(l) \quad o' \stackrel{def}{=} o[f \mapsto l'] \quad \mu''' \stackrel{def}{=} \mu''[l \mapsto o']}{\langle \lceil E.f := E' \rceil \mathcal{L},\, \mu \rangle \ \rightarrow \langle \lceil \texttt{skip} \rceil \mathcal{L},\, \mu''' \rangle}\ [\text{field-ass}]$$

$$\frac{\langle \lceil C_1 \rceil \mathcal{L},\, \mu \rangle \ \rightarrow \langle \lceil C_1' \rceil \mathcal{L}',\, \mu' \rangle \quad C_1 \neq \texttt{spawn}\ E.p()}{\langle \lceil C_1; C_2 \rceil \mathcal{L},\, \mu \rangle \ \rightarrow \langle \lceil C_1'; C_2 \rceil \mathcal{L}',\, \mu' \rangle}\ [\text{seq}]$$

$$\frac{-}{\langle \lceil \texttt{skip}; C \rceil \mathcal{L},\, \mu \rangle \ \rightarrow \langle \lceil C \rceil \mathcal{L},\, \mu \rangle}\ [\text{seq-skip}]$$

$$\frac{[\![E]\!]^{\mu} = \langle l,\, \mu' \rangle \quad \kappa' = \mathit{lookup}(\mathit{class}(\mu'(l)), m) \quad \kappa'(m) = B}{\langle \lceil E.m(\,) \rceil \mathcal{L},\, \mu \rangle \ \rightarrow \langle \lceil B\{^{l}/_{\texttt{this}}\} \rceil \mathcal{L},\, \mu' \rangle}\ [\text{invoc}]$$

still remains locked, as it was locked more than once. When the lock counter reaches 0, rule [release-lock] can release the lock of the object. Rule [thread-pool] lifts the execution to a pool of threads.

Definition 3 (Operational Semantics of a Program). *The initial configuration of a program is* $\langle P_0,\, \mu_0 \rangle$ *where* $P_0 \stackrel{def}{=} \lceil B_{main}\{^{l_{init}}/_{\texttt{this}}\} \rceil \emptyset,\ \mu_0 \stackrel{def}{=} \{l_{init} \mapsto \langle \mathit{Main},\, \phi, 0 \rangle\}$ *and* $\mathit{Main} = \{\mathit{main}\ \mapsto\ B_{main}\}$ *The* operational semantics *of a program is the set of traces of the form* $\langle P_0,\, \mu_0 \rangle \ \rightarrow^* \langle P,\, \mu \rangle$.

Example 2. The implementation in Ex. 1, becomes a program by defining B_{main} as: $\texttt{K}\langle\, \texttt{x} = \texttt{C}\langle \texttt{f} = \texttt{Object}\langle\rangle\rangle,\ \texttt{y} = \texttt{C}\langle \texttt{f} = \texttt{Object}\langle\rangle\rangle,\ \texttt{z} = \texttt{C}\langle \texttt{f} = \texttt{Object}\langle\rangle\rangle,\ \texttt{h} = \texttt{Object}\langle\rangle\rangle\texttt{.m}();\texttt{skip}.$
The operational semantics builds the following maximal trace from $\langle P_0,\, \mu_0 \rangle$ that, for convenience, we divide in eight macro-steps:

1. $\rightarrow^* \langle \lceil l\texttt{.z := }l\texttt{.x; sync(z)\{}l\texttt{.h := }l\texttt{.z.f; }l\texttt{.z := }l\texttt{.y\}; }l\texttt{.z.f := Object}\langle\rangle\texttt{; skip; skip} \rceil \emptyset,\, \mu_1 \rangle$
   with $\mu_1 \stackrel{def}{=} \mu_0[l \mapsto o, l_1 \mapsto o_1, l_2 \mapsto o_2, l_3 \mapsto o_3, l_4 \mapsto o_4, l_1' \mapsto o_4, l_2' \mapsto o_4, l_3' \mapsto o_4]$;
   $o \stackrel{def}{=} \langle \texttt{K},\ \{\texttt{x} \mapsto l_1, \texttt{y} \mapsto l_2, \texttt{z} \mapsto l_3, \texttt{h} \mapsto l_4\}, 0\rangle;\ o_i \stackrel{def}{=} \langle \texttt{C},\ \{\texttt{f} \mapsto l_i'\}, 0\rangle$, for $i \in 1..3$;
   $o_4 \stackrel{def}{=} \langle \texttt{Object},\ \phi, 0\rangle$
2. $\rightarrow^* \langle \lceil \texttt{sync(z)\{}l\texttt{.h := }l\texttt{.z.f; }l\texttt{.z := }l\texttt{.y\};}l\texttt{.z.f := Object}\langle\rangle\texttt{; skip; skip} \rceil \emptyset,\, \mu_2 \rangle$
   with $\mu_2 \stackrel{def}{=} \mu_1[l \mapsto o[\texttt{z} \mapsto l_1]]$
3. $\rightarrow^* \langle \lceil l\texttt{.h := }l\texttt{.z.f; }l\texttt{.z := }l\texttt{.y; monitor\_exit(}l_1\texttt{); }l\texttt{.z.f := Object}\langle\rangle\texttt{; skip; skip} \rceil \{l_1\},\, \mu_3 \rangle$
   with $\mu_3 \stackrel{def}{=} \mu_2[l_1 \mapsto \mathit{lock}^+(o_1)]$
4. $\rightarrow^* \langle \lceil l\texttt{.z := }l\texttt{.y; monitor\_exit(}l_1\texttt{); }l\texttt{.z.f := Object}\langle\rangle\texttt{; skip; skip} \rceil \{l_1\},\, \mu_4 \rangle$
   with $\mu_4 \stackrel{def}{=} \mu_3[l \mapsto o[\texttt{z} \mapsto l_1][\texttt{h} \mapsto l_1']]$
5. $\rightarrow^* \langle \lceil \texttt{monitor\_exit(}l_1\texttt{); }l\texttt{.z.f := Object}\langle\rangle\texttt{; skip; skip} \rceil \{l_1\},\, \mu_5 \rangle$
   with $\mu_5 \stackrel{def}{=} \mu_4[l \mapsto o[\texttt{z} \mapsto l_2,\ \texttt{h} \mapsto l_1']]$
6. $\rightarrow^* \langle \lceil l\texttt{.z.f := Object}\langle\rangle\texttt{; skip; skip} \rceil \emptyset,\, \mu_6 \rangle$, with $\mu_6 \stackrel{def}{=} \mu_5[l_1 \mapsto o_1]$
7. $\rightarrow^* \langle \lceil \texttt{skip} \rceil \emptyset,\, \mu_7 \rangle$, with $\mu_7 \stackrel{def}{=} \mu_6[l_2 \mapsto o_2[\texttt{f} \mapsto l_2''],\ l_2'' \mapsto o_4]$.

Our semantics lets us formalize some properties on the soundness of the locking mechanism, that we report in Appendix B. Here we just give a key property used in our proofs, that states that two threads never lock the same location (*i.e.*, object) at the same time. It is proved by induction on the length of the trace.

Proposition 1 (Locking correctness). *Let* $\langle P_0,\, \mu_0 \rangle \ \rightarrow^* \langle \lceil C_1 \rceil \mathcal{L}_1 ... \lceil C_n \rceil \mathcal{L}_n\ ,\, \mu \rangle$ *be an arbitrary trace. For any* $i, j \in \{1 \ldots n\}$, $i \neq j$ *entails* $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$.

Table 2 Structural operational semantics for concurrency and synchronization.

$$\frac{[\![E]\!]^{\mu} = \langle l,\, \mu'\rangle \quad \kappa' = \mathit{lookup}(\mathit{class}(\mu'(l)), p) \quad \kappa'(p) = B}{\langle \lceil \mathtt{spawn}\ E.p(\,); C\rceil \mathcal{L},\, \mu\rangle \ \rightarrow \langle \lceil B\{^{l}/_{\mathtt{this}}\}\rceil \emptyset . \lceil C\rceil \mathcal{L},\, \mu'\rangle}\ [\text{spawn}]$$

$$\frac{[\![E]\!]^{\mu} = \langle l, \mu'\rangle}{\langle \lceil \mathtt{sync(}E\mathtt{)\{}C\mathtt{\}}\rceil \mathcal{L},\, \mu\rangle \ \rightarrow \langle \lceil \mathtt{monitor\_enter(}l\mathtt{)}; C; \mathtt{monitor\_exit(}l\mathtt{)}\rceil \mathcal{L},\, \mu'\rangle}\ [\text{sync}]$$

$$\frac{\mathit{lock}^{\#}(\mu(l)) = 0 \quad \mathcal{L}' \stackrel{\mathit{def}}{=} \mathcal{L} \cup \{l\} \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{+}(\mu(l))]}{\langle \lceil \mathtt{monitor\_enter(}l\mathtt{)}\rceil \mathcal{L},\, \mu\rangle \ \rightarrow \langle \lceil \mathtt{skip}\rceil \mathcal{L}',\, \mu'\rangle}\ [\text{acquire-lock}]$$

$$\frac{l \in \mathcal{L} \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{+}(\mu(l))]}{\langle \lceil \mathtt{monitor\_enter(}l\mathtt{)}\rceil \mathcal{L},\, \mu\rangle \ \rightarrow \langle \lceil \mathtt{skip}\rceil \mathcal{L},\, \mu'\rangle}\ [\text{reentrant-lock}]$$

$$\frac{\mathit{lock}^{\#}(\mu(l)) > 1 \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{-}(\mu(l))]}{\langle \lceil \mathtt{monitor\_exit(}l\mathtt{)}\rceil \mathcal{L},\, \mu\rangle \ \rightarrow \langle \lceil \mathtt{skip}\rceil \mathcal{L},\, \mu'\rangle}\ [\text{decrease-lock}]$$

$$\frac{\mathit{lock}^{\#}(\mu(l)) = 1 \quad \mathcal{L}' \stackrel{\mathit{def}}{=} \mathcal{L} \setminus \{l\} \quad \mu' \stackrel{\mathit{def}}{=} \mu[l \mapsto \mathit{lock}^{-}(\mu(l))]}{\langle \lceil \mathtt{monitor\_exit(}l\mathtt{)}\rceil \mathcal{L},\, \mu\rangle \ \rightarrow \langle \lceil \mathtt{skip}\rceil \mathcal{L}',\, \mu'\rangle}\ [\text{release-lock}]$$

$$\frac{\langle T,\, \mu\rangle \ \rightarrow \langle P,\, \mu'\rangle}{\langle P_1.T.P_2,\, \mu\rangle \ \rightarrow \langle P_1.P.P_2,\, \mu'\rangle}\ [\text{thread-pool}]$$

4 Two Semantics for `@GuardedBy` Annotations

This section gives two distinct formalizations for locking specifications of the form `@GuardedBy(`E`)` *`Type`* `f`, where E is a guard expression allowed by the language, possibly using a special variable `itself` that stands for the protected field `f`.

In a *name-protection* interpretation, a thread must hold the lock on the value of the guard expression E whenever it *accesses* (reads or writes) the *name* of the guarded field `f`. Def. 4 formalizes the notion of *accessing an expression* when a given command is executed. For our purposes, it is enough to consider a single execution step; thus the accesses in $C_1; C_2$ are only those in C_1. When an object is created, only its creating thread can access it. Thus field initialization cannot originate data races and is not considered as an access. The access refers to the value of the expression, not to its lock counter, hence `sync(`E`){`C`}` does not access E. For accesses to a field f, Def. 4 keeps the exact expression used for the container of f, that will be used in Def. 5 for the contextualization of `this`.

Definition 4 (Expressions Accessed). *The set of expressions accessed in a single execution step is defined as follows:*

$$\begin{array}{rcl@{\qquad}rcl}
\mathrm{acc}(l) & \stackrel{\mathit{def}}{=} & \emptyset & \mathrm{acc}(\kappa\langle f_1{=}E_1, \ldots, f_n{=}E_n\rangle) & \stackrel{\mathit{def}}{=} & \bigcup_{i=1}^{n} \mathrm{acc}(E_i) \\
\mathrm{acc}(\mathtt{let}\ x = E\ \mathtt{in}\ C) & \stackrel{\mathit{def}}{=} & \mathrm{acc}(E) & \mathrm{acc}(E.f) & \stackrel{\mathit{def}}{=} & \mathrm{acc}(E) \cup \{E.f\} \\
\mathrm{acc}(C_1; C_2) & \stackrel{\mathit{def}}{=} & \mathrm{acc}(C_1) & \mathrm{acc}(E.f := F) & \stackrel{\mathit{def}}{=} & \mathrm{acc}(E.f) \cup \mathrm{acc}(F) \\
\mathrm{acc}(E.m(\,)) & \stackrel{\mathit{def}}{=} & \mathrm{acc}(E) & \mathrm{acc}(\mathtt{spawn}\ E.m(\,)) & \stackrel{\mathit{def}}{=} & \mathrm{acc}(E) \\
\mathrm{acc}(\mathtt{monitor_enter(}l\mathtt{)}) & \stackrel{\mathit{def}}{=} & \emptyset & \mathrm{acc}(\mathtt{monitor_exit(}l\mathtt{)}) & \stackrel{\mathit{def}}{=} & \emptyset \\
\mathrm{acc}(\mathtt{sync(}E.f\mathtt{)\{}C\mathtt{\}}) & \stackrel{\mathit{def}}{=} & \mathrm{acc}(E) & \mathrm{acc}(\mathtt{sync(}x\mathtt{)\{}C\mathtt{\}}) & \stackrel{\mathit{def}}{=} & \emptyset \\
\mathrm{acc}(\mathtt{skip}) & \stackrel{\mathit{def}}{=} & \emptyset & \mathrm{acc}(\mathtt{sync(}l\mathtt{)\{}C\mathtt{\}}) & \stackrel{\mathit{def}}{=} & \emptyset
\end{array}$$

$$\mathrm{acc}(\mathtt{sync(}\kappa\langle f_1 = E_1, \ldots, f_n = E_n\rangle\mathtt{)\{}C\mathtt{\}}) \stackrel{\mathit{def}}{=} \mathrm{acc}(\kappa\langle f_1{=}E_1, \ldots, f_n{=}E_n\rangle).$$

We say that C accesses *a field f if and only if $E.f \in \mathrm{acc}(C)$, for some E.*

Def. 5 formalizes when a field f is name-protected by `@GuardedBy(`E`)` in a program. In Sec. 2 we have discussed the reasons for using the special variable `itself` in the guard expressions when working with a value-protection semantics. In the name-protection semantics, `itself` denotes just an alias of the accessed name: `@GuardedBy(itself)` *`Type`* `f` is the same as `@GuardedBy(f)` *`Type`* `f`.

Definition 5 (Name-protection `@GuardedBy`). *A field f in a program is* name protected *by* `@GuardedBy(`E`)` *if and only if for any trace of that program*

$$\langle P_0,\, \mu_0\rangle \quad \to^* \;\langle P_1.T.P_2,\, \mu\rangle \;\to \langle P_1.\hat{T}.P_2,\, \hat{\mu}\rangle$$

where $T = \lceil C\rceil\mathcal{L}$, whenever C accesses f, i.e. $E'.f \in \mathrm{acc}(C)$, for some E', with $[\![E']\!]^{\mu} = \langle l',\, \mu'\rangle$ and $l'' = state(\mu'(l'))f$, we have $loc([\![E\{^{l',l''}/_{\texttt{this},\texttt{itself}}\}]\!]^{\mu'}) \in \mathcal{L}$.

Def. 5 evaluates the guard expression E at those program points where f is accessed, in order to verify that its lock is held by the current thread. Thus, E is evaluated in a memory μ' obtained by the evaluation of the container of f, that is E'. Actually, we evaluate E only after having replaced the occurrences of the variable `this` with l', *i.e.* the evaluation of E', and the occurrences of `itself` with l'', *i.e.* the evaluation of f.

Example 3. In Ex. 2, field `y` is name protected by `@GuardedBy(this.x)`. It is accessed during the 5th macro-step, when $[\![\texttt{this.x}\{^{l}/_{\texttt{this}}\}]\!]^{\mu_4} {=} [\![l.\texttt{x}]\!]^{\mu_4} {=} \langle l_1,\, \mu_4\rangle$, and l_1 is locked. Fields `x` and `z` are name protected by `@GuardedBy(`E`)`, for *no* E, as they are accessed at macro-step 2, when no location is locked.

An alternative semantics for `@GuardedBy` protects the values held in a fields rather than the field name. In this *value-protection* semantics, a field f is `@GuardedBy(`E`)` if wherever a thread dereferences a location l eventually bound to f, it holds the lock on the object obtained by evaluating E at that point. In object-oriented parlance, *dereferencing a location* l means accessing the object stored at l in order to read or write a field. In Java, accesses to the lock counter are synchronized at a low level and the class tag is immutable, hence their accesses cannot give rise to data races and are not relevant here. Dereferences (Def. 6) are very different from accesses (Def. 4). For instance, statement `v.f := w.g.h` accesses expressions `v`, `v.f`, `w`, `w.g` and `w.g.h` but dereferences only the locations held in `v`, `w` and `w.g`: locations bound to `v.f` and `w.g.h` are left untouched. Def. 6 formalizes the set of locations dereferenced by an expression or command to access some field and keeps track of the fact that the access is for reading ($\Rightarrow$) or writing ($\Leftarrow$) the field. Hence dereference tokens are $l.f{\Leftarrow}$ or $l.f{\Rightarrow}$, where l is a location and f is the name of the field that is accessed in the object held in l.

Definition 6 (Dereferenced Locations). *Given a memory μ, the dereferences in a single reduction are defined as follows:*

$$\mathrm{deref}(l)^{\mu} \stackrel{\mathit{def}}{=} \emptyset \qquad \mathrm{deref}(E.f)^{\mu} \stackrel{\mathit{def}}{=} \{loc\,([\![E]\!]^{\mu})\,.f{\Rightarrow}\} \cup \mathrm{deref}(E)^{\mu}$$

$$\mathrm{deref}(\kappa\langle f_1 = E_1, \ldots, f_n = E_n\rangle)^{\mu} \stackrel{\mathit{def}}{=} \textstyle\bigcup_{i=1}^{n} \mathrm{deref}(E_i)^{\mu}$$

$$\mathrm{deref}(\texttt{let}\ x = E\ \texttt{in}\ C)^{\mu} \stackrel{\mathit{def}}{=} \mathrm{deref}(E)^{\mu} \qquad \mathrm{deref}(\texttt{skip})^{\mu} \stackrel{\mathit{def}}{=} \emptyset$$

$$\mathrm{deref}(\texttt{sync(}E\texttt{)\{}C\texttt{\}})^{\mu} \stackrel{\mathit{def}}{=} \mathrm{deref}(E)^{\mu} \qquad \mathrm{deref}(C_1; C_2)^{\mu} \stackrel{\mathit{def}}{=} \mathrm{deref}(C_1)^{\mu}$$

$$\mathrm{deref}(\texttt{monitor\_enter(}l\texttt{)})^{\mu} \stackrel{\mathit{def}}{=} \emptyset \qquad \mathrm{deref}(\texttt{monitor\_exit(}l\texttt{)})^{\mu} \stackrel{\mathit{def}}{=} \emptyset$$

$$\mathrm{deref}(E.f := E')^{\mu} \stackrel{\mathit{def}}{=} \{loc\,([\![E]\!]^{\mu})\,.f{\Leftarrow}\} \cup \mathrm{deref}(E')^{\mu}$$

$$\mathrm{deref}(E.m(\,))^{\mu} \stackrel{\mathit{def}}{=} \mathrm{deref}(E)^{\mu} \qquad \mathrm{deref}(\texttt{spawn}\ E.m(\,))^{\mu} \stackrel{\mathit{def}}{=} \mathrm{deref}(E)^{\mu}$$

We define $derefloc(C)^{\mu} \stackrel{\mathit{def}}{=} \{l \mid \exists f\ s.t.\ l.f{\Leftarrow} \in \mathrm{deref}(C)^{\mu} \vee l.f{\Rightarrow} \in \mathrm{deref}(C)^{\mu}\}$.

Def. 7 formalizes when a field f is value-protected by `@GuardedBy(`E`)` in a program. Intuitively, for any execution trace t we collect the set $\mathcal{F}$ of locations that have ever been bound to a guarded field f in t. Then, we require that whenever a thread dereferences one of those locations, that thread must hold the lock on the object obtained by evaluating the guard E.

Definition 7 (Value-protection `@GuardedBy`). *A field f in a program is* value-protected *by* `@GuardedBy(`E`)` *if and only if for any trace of that program*

$$\langle P_0,\, \mu_0\rangle \;\to^i \langle P_i,\, \mu_i\rangle = \langle P.T.Q,\, \mu_i\rangle \;\to \langle P.T'.Q,\, \mu_{i+1}\rangle \;\to \ldots$$

letting $T{=}\lceil C\rceil\mathcal{L}$; letting $\mathcal{F} = \bigcup_{j>0}\{state(\mu_j(l))f \mid l{\in}\,\mathrm{dom}(\mu_j) \wedge state(\mu_j(l))f{\downarrow}\}$ be the set of locations eventually associated to field f; letting $\Delta_f = derefloc(C)^{\mu_i} \cap \mathcal{F}$ be those locations in $\mathcal{F}$ dereferenced at the $i{+}1$-th step of the trace above. Then, for every $l \in \Delta_f$ it follows that $loc\left(\llbracket E\{^l/_{\mathtt{itself}}\}\rrbracket^{\mu_i}\right) \in \mathcal{L}$.

Note that $\mathcal{F}$ contains all locations eventually bound to f, at any time, in the past or the future, not just those bound in the last configuration $\langle P_i,\, \mu_i\rangle$. This is because value-protection `@GuardedBy(`E`)` is a kind of type annotation that predicates on the values held in the annotated field, and the properties of such values must remain unchanged as they flow through the program.

Note alse that the only variable allowed in the guard expression E is `itself`. This is because there is no value that we can bind to the container `this` of the guarded value (in Definition 5, instead, we had the value of E'). It is actually possible that the value of the guarded field f might be held in more fields of distinct containers, hence the unique identification of the value of *the* container `this` becomes impossible here.

Example 4. In Ex. 2 field `x` is value protected by `@GuardedBy(itself)`. This because $\Delta_{\mathtt{x}} = \{l_1\}$ and l_1 is dereferenced only at macro-step 4, when the corresponding object o_1 is accessed to obtain the value of its field f. At that program point, l_1 is locked by the current thread. Fields `y` and `z` are value protected by `@GuardedBy(`E`)`, for *no* E, since $\Delta_{\mathtt{y}} = \{l_2\}$, $\Delta_{\mathtt{z}} = \{l_1, l_2\}$, and l_2 is dereferenced at macro-step 7, when the thread holds no locks.

The two semantics for `@GuardedBy` are incomparable: neither entails the other. In Ex. 2 field `x` is value protected by `@GuardedBy(itself)`, but is not name protected. Field `y` is name protected by `@GuardedBy(this.x)`, but is not value protected.

5 Protection against Data Races

In this section we provide sufficient conditions that ban data races when `@GuardedBy` annotations are satisfied, in either of the two versions. Intuitively, a *data race* occurs when two threads dereference the same location l, at the same time, to access a field of the object stored at l, and at least one modifies the field.

Definition 8 (Data race). *Let $\langle P_0,\, \mu_0\rangle \;\to^* \langle P,\, \mu\rangle = \langle P_1.T_1.P_2.T_2.P_3,\, \mu\rangle$, with $T_i = \lceil C_i\rceil\mathcal{L}_i$, for $i \in 1..2$. A* data race *occurs at a location l during the access to some field f in $\langle P,\, \mu\rangle$, only if*

- $\langle P,\, \mu\rangle \;\to \langle P_1.T_1'.P_2.T_2.P_3,\, \mu'\rangle$*, for some* $T_1' \neq T_1$
- $\langle P,\, \mu\rangle \;\to \langle P_1.T_1.P_2.T_2'.P_3,\, \mu''\rangle$*, for some* $T_2' \neq T_2$

where $l.f{\Leftarrow} \in \mathrm{deref}(C_1)^{\mu}$ *and* ($l.f{\Leftarrow} \in \mathrm{deref}(C_2)^{\mu}$ *or* $l.f{\Rightarrow} \in \mathrm{deref}(C_2)^{\mu}$*).*

In Sec. 2 we said that accesses to fields (or their value) that are `@GuardedBy(`E`)` occur in mutual exclusion if the guard E is such that it can be evaluated at distinct program points and its evaluation always yields the same value. This implies that E cannot contain local variables as they cannot be evaluated at distinct program points. Thus, we restrict the variables that can be used in E. In particular, `itself` can always be used since it refers to the location being dereferenced. For the name-protection semantics, `this` can also be used, since it refers to the container of the guarded field, as long as it can be uniquely determined; for instance, if there is no aliasing. Indeed, Sec. 2 shows that name protection without aliasing restrictions does not ban data races, since it protects the name but not its value, that can be freely aliased and accessed through other names, without synchronization. In a real programming language, *aliasing* arises from assignments, returned values, and parameter passing. Our simple language has no returned values and only the implicit parameter `this`.

Definition 9 (Non-aliased fields). *A field f is* non-aliased *in a program if and only if for any trace $\langle P_0, \mu_0\rangle \rightarrow^* \langle P, \mu\rangle$ of that program, there are no l', l'', and g such that $state(\mu(l'))f = state(\mu(l''))g$, and $l' = l''$ entails $f \neq g$.*

Field aliasing can be inferred through a may-alias analysis (that is, a must-non-alias analysis) [3] or prevented by syntactic restrictions, as currently done by Julia. Although the precision of this aliasing analysis might in principle affect the precision of the results, it must be said that programmers who use name protection do not alias the protected fields. When they do it, the field is not actually data race free, hence simple syntactic restrictions are enough in practice.

However, as discussed in Sec. 2, to ensure the soundness of the name-protection semantics we need a further assumption: the value of the guard expression must not change during program execution.

Definition 10 (Final expressions). *An expression E where the only allowed variables are* `this` *and* `itself` *is said to be* final *in a program if for every trace $\langle P_0, \mu_0\rangle \rightarrow^i \langle P_i, \mu_i\rangle$ of that program, for all $0\leq p\leq q\leq i$ and for all $l, l' \in \mathrm{dom}(\mu_p)$, $[\![E\{^{l,l'}/_{\texttt{this,itself}}\}]\!]^{\mu_p} = \langle l_1, \mu_1\rangle$ and $[\![E\{^{l,l'}/_{\texttt{this,itself}}\}]\!]^{\mu_q} = \langle l_2, \mu_2\rangle$ entails $l_1{=}l_2$.*

We can now prove that, for non-aliased fields and final guard expressions, the name-protection semantics of `@GuardedBy` protects against data races.

Theorem 1 (Name-protection semantics vs. data race protection). *Let E be a final expression in a program, and f be a* non-aliased *field that is name protected by* `@GuardedBy(`E`)`*. Let E contain no variable distinct from* `itself` *and* `this`*. Then, no data race can occur at those locations bound to f, at any execution trace of that program.*

As argued in Sec. 2, the assumptions on non-aliased fields and final guard expressions are not necessary in the value-protection semantics as this locking discipline protects directly the value of the guarded field f.

Theorem 2 (Value-protection semantics vs. data race protection). *Let E be an expression in a program, and f be a field that is value-protected by* `@GuardedBy(`E`)`*. Let E have no variable distinct from* `itself`*. Then no data race can occur at those locations bound to f, during any execution of the program.*

Both results are proved by contradiction, by supposing that a data race occurs and showing that two threads would lock the same location, against Prop. 1

6 Implementation in Julia

The Julia analyzer infers `@GuardedBy` annotations. The implementation has been completed after the theory of this article being formalized. At the same time, the theoretical results have been inspired by actual case studies analyzed by Julia, as it is often the case when research must formalize a property that is already used in real code. The user selects name-protection or value-protection semantics.

As said in Sec. 2, and formalized in Sec. 4, a field f is `@GuardedBy(`E`)` if, at *all* program points P where f is accessed (for name protection) or one of its locations is dereferenced (for value protection), the value of E is locked by the current thread. The inference algorithm of Julia builds on two phases: (i) compute the set $\mathcal{P}$ of program points where f is accessed; (ii) find expressions E locked at all program points $P \in \mathcal{P}$.

Point (i) is obvious for name protection, since accesses to f are syntactically apparent in the program. For value protection, the set $\mathcal{P}$ is instead undecidable, since there might be infinitely many objects potentially bound to f at runtime, that flow through aliasing. Hence Julia overapproximates the set $\mathcal{P}$ by abstracting objects into their *creation point* in the program: if two objects have distinct creation points, they must be distinct. The number of creation points is finite, hence the approximation is finitely computable. Julia implements creation points analysis as a concretization of the class analysis in [21], where objects are abstracted in their creation points instead of just their class tag.

Point (ii) uses the *definite aliasing* analysis of Julia, described in [19]. At each `synchronized(`G`)` statement, that analysis provides a set L of expressions that are definitely an alias of G at that statement (*i.e.*, their values coincide there, always). Julia concludes that the expressions in L are locked by the current thread after the `synchronized(`G`)` and until the end of its scope. Potential side-effects might however invalidate that conclusion, possibly due to concurrent threads. Hence, Julia only allows in L fields that are never modified after being defined, which can be inferred syntactically for a field. For name protection, viewpoint adaptation of `this` is performed on such expressions (Def. 5). These sets L are propagated in the program until they reach the points in P. The expressions E in point (ii) are hence those that belong to L at *all* program points $\mathcal{P}$.

Since `@GuardedBy(`E`)` annotations are expected to be used by client code, E should be visible to the client. For instance, Julia discards expressions E that refer to a private field or to a local variable that is not a parameter, since these would not be visible nor useful to a client.

The supporting creation points and definite aliasing analyses are sound, hence Julia soundly infers `@GuardedBy(`E`)` annotations that satisfy the formal definitions in Sec. 4. Such inferred annotations protect against data races if the sufficient conditions in Sec. 5 hold for them.

More detail and experiments with this implementation can be found in [11]. There, we have analyzed 15 large open-source programs, including parts of Eclipse and Tomcat, for a total of $1,290,060$ non-blank lines of code. Julia has often inferred the annotations already present in code (if any), while the annotations not inferred by Julia have often been proved to be programmers mistakes (either fields that are not actually guarded as expected, or they are guarded in a way that do not prevent data races).

7 Conclusions, Future and Related Work

Coming back to the ambiguities sketched in Sec. 1, we have clarified that: (1) `this` in the guard expression must be interpreted as the container of the guarded field and consistently contextualized (Def. 5). (2) An access is a field use for name protection (Def. 4 and 5). A value access is a dereference (field get/set or method call) for value protection; copying a value is not an access in this case (Def. 6 and 7). (3) The value of the guard expression must be locked when a name or value is accessed, regardless of how it is accessed in the synchronized block (Def. 5 and 7). (4) The lock must be held on the value of the guard expression as evaluated at the access to the guarded field (name or value) (Def. 5 and 7). (5) Either the *name* or the *value* of a field can be guarded, but this choice leads to very different semantics. Namely, in the *name-protection* semantics, the lock must be held whenever the field's name is accessed (Def. 4 and 5). In the *value-protection* semantics, the lock must be held whenever the field's value is accessed (Def. 6 and 7), regardless of what expression is used to access the value. Both semantics yield a guarantee against data races, though name protection requires an aliasing restriction on the field and final guard expressions (Th. 1 and 2).

This work could be extended by enlarging the set of guard expressions that protect against data races. For instance, it could be extended with static fields. We believe that the protection results in Sec. 5 still hold for them. Another aspect to investigate is the scope of the protection against data races. In this article, a single location is protected (Def. 8), not the whole tree of objects reachable from it: our protection is shallow rather than deep. Deep protection is possibly more interesting to the programmer, since it relates to a data structure as a whole, but it requires to reason about boundaries and encapsulation of data structures.

The work of Abadi et al. [1] is the closest to ours. It proposes a type system for detecting data races in Java programs by introducing `@GuardedBy` type annotations, according to a name-preservation semantics. Theoretical results are stated on a significant concurrent subset of Java, RaceFreeJava, which shares many similarities with our calculus. The main result of the paper is that well-typed programs do not have data races. This result relies on a few constraints: (i) like us, in `GuardedBy(`E`)` annotations, E must be final, so `this` is the only admitted variable in E; (ii) unlike us, in blocks `sync(`E`){`C`}`, E must be final; (iii) unlike us, field updates are admissible (typable) only if they are guarded by some final expression; (iv) unlike us, Java lock reentrancy is not admitted; (v) unlike us, they overcome the limitation (i) by extending the type system to allow fields of a class to be protected by locks external to the class. Note that non-aliasing is not required in [1], although this seems to be a consequence of the (quite) strong requirement (iii) that field updates can *only* occur on guarded fields.

We refer to [1] for a careful review of tools developed for detecting data races.

There are many other formalizations of concurrent fragments of Java, such as [2, 8]. Our goal here is the semantics of annotations such as `@GuardedBy`. Hence we kept the semantics of the language to the minimum core needed for the formalization of those program annotations. Another well-known formalization is Featherweight Java [15], a functional language that provides a formal kernel of sequential Java. It does not include threads, nor assignment. Thus, it is not adequate to formalize data races, which need concurrency and assignments. A similar argument applies to Middleweight Java [5] and Welterweight Java [20]. The need of a formal specification for reasoning about Java's concurrency and for building verification tools is recognized [9, 17, 7] but we are not aware of any formalization of the semantics of Java's concurrency annotations.

Our formalization will support tools based on model-checking such as Java PathFinder [18] and Bandera [14, 4], on type-checking such as the Checker Framework [10] and `Houdini/rcc` [1], or on abstract interpretation such as Julia [23].

References

1. M. Abadi, C. Flanagan, and S.N. Freund. Types for safe locking: Static race detection for Java. *ACM Trans. Program. Lang. Syst.*, 28(2):207–255, 2006.
2. E. Ábrahám-Mumm, F. S. de Boer, W. P. de Roever, and M. Steffen. Verification for Java's Reentrant Multithreading Concept. In *FOSSACS'02*. Springer, 2002.
3. L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.
4. Bandera. About Bandera. `http://bandera.projects.cis.ksu.edu`.
5. G. M. Bierman and M. J. Parkinson. Effects and Effect Inference for a Core Java Calculus. In *WOOD*, pages 82–107, 2003.
6. B. Blanchet. Escape Analysis for Java: Theory and Practice. *ACM TOPLAS*, 25(6):713–775, 2003.
7. D. Bogdanas and G. Rosu. K-java: A Complete Semantics of Java. In *ACM SIGPLAN-SIGACT POPL*, pages 445–456, Mumbai, India, 2015.
8. P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. From Sequential to Multi-Threaded Java: An Event-Based Operational Semantics. In *AMAST'97*, 1997.
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. W. Dietl, S. Dietzel, M. D. Ernst, K. Muslu, and T. W. Schiller. Building and Using Pluggable Type-Checkers. In *ICSE'11*, 2011.
11. M. D. Ernst, A. Lovato, D. Macedonio, F. Spoto, and J. Thaine. Locking discipline inference and checking. In *ICSE'16*, Austin, TX, USA, 2016.
12. B. Goetz, T. Peierls, J. Bloch, and J. Bowbeer. *Java Concurrency in Practice*. Addison Wesley, May 2006.
13. Google. Guava: Google Core Libraries for Java 1.6+. `https://code.google.com/p/guava-libraries`.
14. J. Hatcliff and M. B. Dwyer. Using the Bandera Tool Set to Model-Check Properties of Concurrent Java Software. In *CONCUR'01*, pages 39–58. Springer, 2001.
15. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A Minimal Core Calculus for Java and GJ. *ACM TOPLAS*, 23(3):396–450, 2001.
16. Javadoc for `@GuardedBy`. `https://jsr-305.googlecode.com/svn/trunk/javadoc/javax/annotation/concurrent/GuardedBy.html`.
17. B. Long and B. W. Long. Formal Specification of Java Concurrency to Assist Software Verification. In *IPDPS'03*. IEEE Computer Society, 2003.
18. NASA. Java PathFinder. `http://babelfish.arc.nasa.gov/trac/jpf`.
19. D. Nikolic and F. Spoto. Definite Expression Aliasing Analysis for Java Bytecode. In *ICTAC*, pages 74–89, Bangalore, India, September 2012.
20. J. Östlund and T. Wrigstad. Welterweight Java. In *TOOLS'10*, pages 97–116, 2010.
21. J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. In *OOPSLA*, volume 26(11) of *ACM SIGPLAN Notices*, pages 146–161. ACM, November 1991.
22. V. Pech. Concurrency is Hot, Try the JCIP Annotations. `http://jetbrains.dzone.com/tips/concurrency-hot-try-jcip`, February 2010.
23. Julia Srl. The Julia Static Analyzer. `http://www.juliasoft.com/julia`.

A Proofs of Sec 5

Theorem 3 (Name-protection semantics vs. data race protection). *Let E be a final expression in a program, and f be a non-aliased field that is name protected by* `@GuardedBy(`E`)`*. Let E contain no variable distinct from* `itself` *and* `this`*. Then, no data race can occur at those locations bound to f, at any execution trace of that program.*

Proof. The proof is by contradiction. We recall that P and Q range over thread pools and T ranges over threads. Let $\langle P_0, \mu_0\rangle \rightarrow^i \langle P_i, \mu_i\rangle$ be an arbitrary trace of our program. By Def. 8, if a data race occurred in $\langle P_i, \mu_i\rangle$, at some location l, bound to f,[3] then there would be Q_1, $T_1 = \lceil C_1 \rceil \mathcal{L}_1$, Q_2, $T_2 = \lceil C_2 \rceil \mathcal{L}_2$ and Q_3, such that $\langle P_i, \mu_i\rangle = \langle Q_1.T_1.Q_2.T_2.Q_3, \mu_i\rangle$ and

- $\langle P_i, \mu_i\rangle \rightarrow \langle Q_1.T_1'.Q_2.T_2.Q_3, \mu_1\rangle$, for some $T_1' \neq T_1$
- $\langle P_i, \mu_i\rangle \rightarrow \langle Q_1.T_1.Q_2.T_2'.Q_3, \mu_2\rangle$, for some $T_2' \neq T_2$

with $l.g\!\Leftarrow\, \in \mathrm{deref}(C_1)^\mu$ and ($l.g\!\Leftarrow\, \in \mathrm{deref}(C_2)^\mu$ or $l.g\!\Rightarrow\, \in \mathrm{deref}(C_2)^\mu$), for some field g. That is g is a field accessed by both threads T_1 and T_2 to dereference the location l, and at least one of these two accesses is an assignment.

As $l \in \mathrm{dom}(\mu)$ and f is a non-aliased field (there is no other field pointing at l) both T_1 and T_2 must have previously accessed f, to get access to l. For instance, thread T_1 accessed the expression $E_1.f$ at time p, and thread T_2 accessed the expression $E_2.f$ at time q, with $p \leq q \leq i$.

As f is name protected by `@GuardedBy(`E`)` (Def. 5) both threads T_1 and T_2 hold the lock on the value (*i.e.* the location) of E at time p and q, respectively. Formally,

- at time p, $[\![E_1]\!]^{\mu_p} = \langle l_p, \mu_p'\rangle$ and the location $loc\left([\![E\{^{l_p,l}/_{\texttt{this},\texttt{itself}}\}]\!]^{\mu_p'}\right)$ is locked by T_1 (we recall that `itself` refers to f, and f is non-aliased)
- at time q, $[\![E_2]\!]^{\mu_q} = \langle l_q, \mu_q'\rangle$ and the location $loc\left([\![E\{^{l_q,l}/_{\texttt{this},\texttt{itself}}\}]\!]^{\mu_q'}\right)$ is locked by T_2 (we recall that `itself` refers to f, and f is non-aliased).

As f is a non-aliased field (Def 9) it cannot have two different containers, thus $l_p = l_q = \hat{l}$.

Without loss of generality, thread T_1 never released the lock between step p and i. In fact, if $p < i$ then only possibility is that T_1 at time p (when the expression $E_1.f$ is accessed) the execution of a let construct of the form `let` $x = E_1.f$ `in` $(\ldots y.g \ldots)$ (where x and y are placeholders for the same value l) started within a critical section where the value of E, call it l_E^1, is already locked. And at time i, when the field g is accessed to dereference l, l_E^1 is still locked by T_1, that is $l_E^1 \in \mathcal{L}_1$ (the case $p = i$ is straightforward). Thus,

$$l_E^1 = loc\left([\![E\{^{\hat{l},l}/_{\texttt{this},\texttt{itself}}\}]\!]^{\mu_p'}\right)$$

[3] Formally, there is $l_f \in \mu_i$ such that $state(\mu_i(l_f))f = l$.

With a similar reasoning for T_2 we derive there is $l_E^2 \in \mathcal{L}_2$ such that

$$l_E^2 = loc\left([\![E\{{}^{\hat{l},l}/_{\texttt{this},\texttt{itself}}\}]\!]^{\mu'_q} \right)$$

However, as E is final it follows that:

$$l_E^1 = l_E^2 = loc\left([\![E\{{}^{\hat{l},l}/_{\texttt{this},\texttt{itself}}\}]\!]^{\mu'_p} \right) = loc\left([\![E\{{}^{\hat{l},l}/_{\texttt{this},\texttt{itself}}\}]\!]^{\mu'_q} \right) \quad .$$

By Prop. 1 this is not possible as two different threads cannot lock the same location at the same time.

The requirement on the absence of aliasing is not necessary when working with a value-protection semantics.

Theorem 4 (Value-protection semantics of `@GuardedBy` vs. data race protection). *Let E be an expression in a program, and f be a field that is value-protected by `@GuardedBy(`E`)`. Let E contain no variable distinct from `itself`. Then no data race can occur at those locations bound to f, during any execution trace of the program.*

Proof. Again, the proof is by contradiction and it is similar to the previous one. Let $\langle P_0, \mu_0\rangle \rightarrow^i \langle P_i, \mu_i\rangle$ be an arbitrary trace of our program. By Def. 8, if a data race occurred in $\langle P_i, \mu_i\rangle$, at some location l, bound to f, then there would be Q_1, $T_1 = \lceil C_1 \rceil \mathcal{L}_1$, Q_2, $T_2 = \lceil C_2 \rceil \mathcal{L}_2$ and Q_3, such that $\langle P_i, \mu_i\rangle = \langle Q_1.T_1.Q_2.T_2.Q_3, \mu_i\rangle$ and

- $\langle P_i, \mu_i\rangle \rightarrow \langle Q_1.T'_1.Q_2.T_2.Q_3, \mu_1\rangle$, for some $T'_1 \neq T_1$
- $\langle P_i, \mu_i\rangle \rightarrow \langle Q_1.T_1.Q_2.T'_2.Q_3, \mu_2\rangle$, for some $T'_2 \neq T_2$

with $l.g \Leftarrow \in \mathrm{deref}(C_1)^\mu$ and ($l.g \Leftarrow \in \mathrm{deref}(C_2)^\mu$ or $l.g \Rightarrow \in \mathrm{deref}(C_2)^\mu$), for some field g. That is g is a field accessed by both threads T_1 and T_2 to dereference the location l, and at least one of these two accesses is an assignment.

As f is value protected by `@GuardedBy(`E`)` (Def. 7) both threads T_1 and T_2 hold the lock on the value (*i.e.* the location) of E at time i. Formally,

- $loc\left([\![E\{{}^{l}/_{\texttt{itself}}\}]\!]^{\mu_i}\right) \in \mathcal{L}_1$
- $loc\left([\![E\{{}^{l}/_{\texttt{itself}}\}]\!]^{\mu_i}\right) \in \mathcal{L}_2$.

Again, by Prop. 1, this is impossible, as two threads cannot lock the same location at the same time.

B Properties of the Operational Semantics

Let us provide a few properties showing the soundness of both the locking and unlocking mechanisms of our operational semantics.

Two different threads never lock the same location. First we need a technical lemma.

Lemma 1. *Given an arbitrary execution trace*

$$\langle P_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil C_1 \rceil \mathcal{L}_1 \ldots \lceil C_n \rceil \mathcal{L}_n \,,\, \mu\rangle$$

if $\mathit{lock}^{\#}(\mu_k(l)) = 0$ *for some* $l \in \mathrm{dom}(\mu_k)$ *then* $l \notin \mathcal{L}_i$*, for any* $i \in 1..n$*.*

Proof. By a simple induction on the length k of the trace.

Proposition 2 (Locking correctness). *Given an arbitrary execution trace*

$$\langle P_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil C_1 \rceil \mathcal{L}_1 \ldots \lceil C_n \rceil \mathcal{L}_n \,,\, \mu\rangle$$

then for any $i, j \in \{1 \ldots n\}$*,* $i \neq j$ *entails* $\mathcal{L}_i \cap \mathcal{L}_j = \emptyset$*.*

Proof. By induction on the length k of the trace. Suppose to have:

$$\langle P_0,\, \mu_0\rangle \;\rightarrow^k \langle \lceil C_1 \rceil \mathcal{L}_1 ... \lceil C_i \rceil \mathcal{L}_i ... \lceil C_n \rceil \mathcal{L}_n \,,\, \mu_k\rangle \;\rightarrow \langle \lceil C_1 \rceil \mathcal{L}_1 ... P ... \lceil C_n \rceil \mathcal{L}_n \,,\, \mu_{k+1}\rangle.$$

Here the k+1-th step is due to the i-th thread. By inductive hypothesis we know that after k steps the sets $\mathcal{L}_j$, for $1 \leq j \leq n$, are pairwise disjoint. Let us prove that after the k+1-th step this property is preserved. Let proceed by case analysis on the reduction rule applied to derive the k+1-th step.

- Let the step be due to an application of one rule among [let], [field-ass], [seq-skip], [invoc], [sync], [reentrant-lock], [decrease-lock] (possibly together with rule [seq]). Then $P = \lceil C_i' \rceil \mathcal{L}_i$, for some C_i', and the set of locked location of the i-th thread remained unchanged, thus the property is trivially preserved.
- Let the step be due to an application of rule [spawn]. Thus,

  $$P = \lceil B\{^{l}/_{\mathtt{this}}\} \rceil \emptyset \,.\, \lceil C_i' \rceil \mathcal{L}_i$$

  for some B, l and C_i'. In this case a new thread has been spawn with an empty set of locked locations. Again the property is trivially maintained.
- Let the step be due to an application of rule [acquire-lock] (possibly together with rule [seq]). Thus, $P = \lceil C_1' \rceil \mathcal{L}_i'$ with $\mathcal{L}_i' = \mathcal{L}_i \cup \{l\}$, for some l such that $\mathit{lock}^{\#}(\mu_k(l)) = 0$. Then, by Lemma 1 the property is preserved also after the k+1-th step.
- Let the step be due to an application of rule [release-lock] (possibly together with rule [seq]). Then, $P = \lceil C_i' \rceil \mathcal{L}_i'$ for some C_i' and $\mathcal{L}_i' = \mathcal{L}_i \setminus \{l\}$, for some l. In this case, as the lockset of the i-th thread becomes smaller the property is preserved.

When a thread starts its execution it does not hold any lock:

Proposition 3 (Thread initialization vs. locking). *Let*

$$\langle P_0,\, \mu_0\rangle \;\rightarrow^* \langle \lceil C_1 \rceil \mathcal{L}_1 ... \lceil C_i \rceil \mathcal{L}_i ... \lceil C_n \rceil \mathcal{L}_n \,,\, \mu\rangle \;\rightarrow \langle \lceil C_1 \rceil \mathcal{L}_1 ... P ... \lceil C_n \rceil \mathcal{L}_n \,,\, \mu'\rangle$$

be an arbitrary trace where $C_i = \mathtt{spawn}\, E.p(\,); C_i'$*, for some* E, p *and* C_i'*, then*

- $[\![E]\!]^\mu = \langle l,\, \mu' \rangle$
- $\kappa' = lookup(class(\mu'(l)), p)$
- $\kappa'(p) = B$
- $P = \lceil B\{^{l}/_{\mathtt{this}}\} \rceil \emptyset \,.\, \lceil C'_i \rceil \mathcal{L}_i$.

Proof. This is a direct consequence of the semantic rules [spawn], the only one which can be applied to perform the reduction step.

When a thread terminates it does not keep locks on locations:

Proposition 4 (Thread termination vs. locking). *Let*

$$\langle P_0,\, \mu_0 \rangle \;\to^* \langle \lceil C_1 \rceil \mathcal{L}_1 \ldots \lceil C_n \rceil \mathcal{L}_n \,,\, \mu \rangle$$

be an arbitrary run where $C_i = \mathtt{skip}$*, then* $\mathcal{L}_i = \emptyset$.

Proof. By induction on the length of the reduction.

A thread may not lock a location by mistake:

Proposition 5 (Locking). *Let*

$$\langle P_0,\, \mu_0 \rangle \;\to^* \langle \lceil C_1 \rceil \mathcal{L}_1 \ldots \lceil C_n \rceil \mathcal{L}_n \,,\, \mu \rangle \;\to \langle \lceil \hat{C}_1 \rceil \hat{\mathcal{L}}_1 \ldots \lceil \hat{C}_m \rceil \hat{\mathcal{L}}_m \,,\, \hat{\mu} \rangle$$

be an arbitrary run. Then $\bigcup_{j=1}^{n} \mathcal{L}_j \;\subset\; \bigcup_{j=1}^{m} \hat{\mathcal{L}}_j$*, if and only if*

- *there is* $i \in 1..n$ *such that* $C_i = \mathtt{monitor\_enter}(l); C'_i$*, for some* l *and* C'_i
- $lock^{\#}(\mu(l)) = 0$ *and* $lock^{\#}(\hat{\mu}(l)) = 1$
- $m = n$ *and* $\hat{\mathcal{L}}_j = \mathcal{L}_j$ *for every* $j \in \{1 \ldots n\} \setminus \{i\}$
- $\hat{\mathcal{L}}_i = \mathcal{L}_i \uplus \{l\}$.

Proof. By case analysis on the rules applied to perform the reduction step.

Reentrant locks are allowed: only threads that already own the lock on an object can synchronize again on that object.

Proposition 6 (Reentrant locking). *Given an arbitrary run*

$$\langle P_0,\, \mu_0 \rangle \;\to^* \langle \lceil C_1 \rceil \mathcal{L}_1 \ldots \lceil C_n \rceil \mathcal{L}_n \,,\, \mu \rangle$$

where $l \in \bigcup_{j=1}^{n} \mathcal{L}_j$*, for some* l*, and* $C_i = \mathtt{monitor\_enter}(l); C'_i$*, for some* $i \in 1..n$ *and* C'_i*. Then*

$$\langle \lceil C_1 \rceil \mathcal{L}_1 ... \lceil C_i \rceil \mathcal{L}_i ... \lceil C_n \rceil \mathcal{L}_n \,,\, \mu \rangle \;\to \langle \lceil C_1 \rceil \mathcal{L}_1 ... \lceil C'_i \rceil \mathcal{L}'_i ... \lceil C_n \rceil \mathcal{L}_n \,,\, \mu' \rangle$$

if and only if

- $l \in \mathcal{L}_i$
- $lock^{\#}(\mu'(l)) = lock^{\#}(\mu(l)) + 1$
- $\mathcal{L}'_i = \mathcal{L}_i$.

Proof. By case analysis on the rule applied to perform the reduction. Here the only rule which can be applied is [reentrant-lock].

Locks on locations are never released by mistake:

Proposition 7 (Lock releasing). *Let*

$$\langle P_0,\, \mu_0\rangle \;\rightarrow^*\, \langle \lceil C_1\rceil \mathcal{L}_1 \ldots \lceil C_n\rceil \mathcal{L}_n\,,\, \mu\rangle \;\rightarrow \langle \lceil \hat{C}_1\rceil \hat{\mathcal{L}}_1 \ldots \lceil \hat{C}_m\rceil \hat{\mathcal{L}}_m\,,\, \hat{\mu}\rangle$$

be an arbitrary run. Then $\bigcup_{j=1}^{n} \mathcal{L}_j \supset \bigcup_{j=1}^{m} \hat{\mathcal{L}}_j$*, if and only if*

- *there is* $i \in 1..n$ *such that* $C_i = \texttt{monitor\_exit}(l); C'_i$*, for some* l *and* C'_i
- $lock^{\#}(\mu(l)) = 1$ *and* $lock^{\#}(\hat{\mu}(l)) = 0$
- $m = n$ *and* $\hat{\mathcal{L}}_j = \mathcal{L}_j$ *for every* $j \in \{1 \ldots n\} \setminus \{i\}$
- $\mathcal{L}_i = \hat{\mathcal{L}}_i \uplus \{l\}$.

Proof. By case analysis on the rule applied to perform the reduction. Here the only possible rule is [release-lock].

Unlocking always happens after some locking: it may release the lock or not, depending on the number of previous lockings.

Proposition 8 (Unlocking). *Let*

$$\langle P_0,\, \mu_0\rangle \;\rightarrow^*\, \langle \lceil C_1\rceil \mathcal{L}_1 ... \lceil C_i\rceil \mathcal{L}_i ... \lceil C_n\rceil \mathcal{L}_n\,,\, \mu\rangle \;\rightarrow \langle \lceil C_1\rceil \mathcal{L}_1 ... \lceil C'_i\rceil \mathcal{L}'_i ... \lceil C_m\rceil \mathcal{L}_m\,,\, \mu'\rangle$$

be an arbitrary run where $C_i = \texttt{monitor\_exit}(l); C'_i$*, for some* l*, then*

- $l \in \mathcal{L}_i$
- *if* $lock^{\#}(\mu(l)) > 1$ *then* $\mathcal{L}'_i = \mathcal{L}_i$ *else* $\mathcal{L}_i = \mathcal{L}'_i \uplus \{l\}$
- $lock^{\#}(\mu'(l)) = lock^{\#}(\mu(l)) - 1$.

Proof. By case analysis on the rules applied to perform the reduction. Here there are two possible rules: [decrease-lock] and [release-lock].