Safe Locking for Multi-Threaded Java [⋆]

Einar Broch Johnsen, Thi Mai Thuong Tran, Olaf Owe, and Martin Steffen

Department of Informatics, University of Oslo, Norway
{einarj,tmtran,olaf,msteffen}@ifi.uio.no

Abstract. There are many mechanisms for concurrency control in high-level programming languages. In Java, the original mechanism for concurrency control, based on synchronized blocks, is lexically scoped. For more flexible control, Java 5 introduced non-lexical operators, supporting lock primitives on re-entrant locks. These operators may lead to run-time errors and unwanted behavior; e.g., taking a lock without releasing it, which could lead to a deadlock, or trying to release a lock without owning it. This paper develops a static type and effect system to prevent the mentioned lock errors for non-lexical locks. The effect type system is formalized for an object-oriented calculus which supports non-lexical lock handling. Based on an operational semantics, we prove soundness of the effect type analysis. Challenges in the design of the effect type system are dynamic creation of threads, objects, and especially of locks, aliasing of lock references, passing of lock references between threads, and reentrant locks as found in Java.

1 Introduction

With the advent of multiprocessors, multi-core architectures, and distributed web-based programs, effective parallel programming models and suitable language constructs are needed. Many concurrency control mechanisms for high-level programming languages have been developed, with different syntactic representations. One option is lexical scoping; for instance, `synchronized` blocks in Java, or protected regions designated by an *atomic* keyword. However, there is a trend towards more flexible concurrency control where protected critical regions can be started and finished freely. Two proposals supporting flexible, non-lexical concurrency control are lock handling via the `ReentrantLock` class in Java 5 [13] and transactional memory, as formalized in *Transactional Featherweight Java* (TFJ) [9]. While Java 5 uses `lock` and `unlock` operators to acquire and release re-entrant locks, TFJ uses onacid and commit operators to start and terminate transactions. Even if these proposals take quite different approaches towards dealing with concurrency —"pessimistic" or lock-based vs. "optimistic" or based on transactions— the additional flexibility of non-lexical control mechanisms comes at a similar price: *improper use leads to run-time exceptions and unwanted behavior*.

A static *type and effect* system for TFJ to prevent unsafe usage of transactions was introduced in [12]. This paper applies that approach to a calculus which supports *lock*

[⋆] The author list is written in alphabetical order. The work is partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`).

handling as in Java 5. Our focus is on *lock errors*; i.e., taking a lock without releasing it, which could lead to a deadlock, and trying to release a lock without owning it.

Generalizing our approach for TFJ to lock handling, however, is not straightforward: In particular, locks are re-entrant and have identities available at the program level. Our analysis technique needs to take identities into account to keep track of which lock is taken by which thread and how many times it has been taken. Furthermore, the analysis needs to handle dynamic lock creation, aliasing, and passing of locks between threads. As transactions have no identity at program level and are not re-entrant, these problems are absent in [12]. Fortunately, they can be solved under reasonable assumptions on lock usage. In particular, aliasing can be dealt with due to the following observation: for the analysis it is sound to assume that all variables are non-aliases, even if they may be aliases at run-time, provided that, per variable, each interaction history with a lock is lock error free in itself. This observation allows us to treat soundness of lock-handling *compositionally*, i.e., individually per thread. So the contribution of the paper is a static analysis preventing lock-errors for non-lexical use of re-entrant locks. A clear separation of local and shared memory allows the mentioned simple treatment of aliasing.

The paper is organized as follows. Sections 2 and 3 define the abstract syntax and the operational semantics of our language with non-lexically scoped locks. Section 4 presents the type and effect system for safe locking, and Section 5 shows the correctness of the type and effect system. Sections 6 and 7 conclude with related and future work.

2 A concurrent, object-oriented calculus

Consider a variant of Featherweight Java (FJ) [7] with concurrency and explicit lock support, but without inheritance and type casts. Table 1 shows the abstract syntax of this calculus. A program consists of a sequence $\vec{D}$ of class definitions.Vector notation refers to a list or sequence of entities; e.g., $\vec{D}$ is a sequence $D_1,\ldots,D_n$ of class definitions and $\vec{x}$ a sequence of variables. A class definition $\mathsf{class}\ C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T};\vec{M}\}$ consists of a name C, fields $\vec{f}$ with corresponding types $\vec{T}$ (assuming that all f_i's are different), and method definitions $\vec{M}$. Fields get values when instantiating an object; $\vec{f}$ are the formal parameters of the constructor C. When writing $\vec{f}{:}\vec{T}$ (and in analogous situations) we assume that the lengths of $\vec{f}$ and $\vec{T}$ correspond, and let $f_i : T_i$ refer to the i'th pair of field and type. We omit such assumptions when they are clear from the context. For simplicity, the calculus does not support overloading; each class has exactly one constructor and all fields and methods defined in a class have different names. A method definition $m(\vec{x}{:}\vec{T})\{t\} : T$ consists of a name m, the typed formal parameters $\vec{x}{:}\vec{T}$, the method body t, and the declaration of the return type T. Types are class names C, (unspecified) basic types B, and `Unit` for the unit value. Locks have type `L`, which corresponds to Java's `Lock`-interface, i.e., the type for instances of the class `ReentrantLock`.

The syntax distinguishes expressions e and threads t. A thread t is either a value v, the terminated thread `stop`, `error` representing exceptional termination, or sequential composition. The let-construct generalizes sequential composition: in $\mathtt{let}\,x{:}T = e\ \mathtt{in}\,t$, e is first executed (and may have side-effects), the resulting value after termination is bound to x and then t is executed with x appropriately substituted. (Standard sequential composition $e;t$ is syntactic sugar for $\mathtt{let}\ \ x{:}T = e\ \ \mathtt{in}\ \ t$ where x does not occur

$$\begin{array}{rcll}
D & ::= & \mathsf{class}\ C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T};\vec{M}\} & \text{class definitions}\\
M & ::= & m(\vec{x}{:}\vec{T})\{t\} : T & \text{methods}\\
t & ::= & \texttt{stop} \mid \texttt{error} \mid v \mid \texttt{let}\, x{:}T = e\ \texttt{in}\, t & \text{threads}\\
e & ::= & t \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid v.f \mid v.f := v \mid v.m(\vec{v}) \mid \texttt{new}\ C(\vec{v}) & \text{expressions}\\
 & \mid & \texttt{spawn}\, t \mid \texttt{new}\, L \mid v.\,\texttt{lock} \mid v.\,\texttt{unlock} \mid \texttt{if}\ v.\,\texttt{trylock}\ \texttt{then}\ e\ \texttt{else}\ e & \\
v & ::= & r \mid x \mid () & \text{values}\\
T & ::= & C \mid B \mid \texttt{Unit} \mid \texttt{L} &
\end{array}$$

Table 1. Abstract syntax

free in t.) Values v are expressions that can not be evaluated further. We ignore standard values like booleans and integers, so values are references r, variables x (including this), and the unit value $()$. We distinguish references o to objects and references l to locks. This distinction is for notational convenience; the type system can distinguish both kinds of references. Conditionals $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, field access $v.f$, field update $v_1.f := v_2$, method calls $v.m(\vec{v})$ and object instantiation $\texttt{new}\ C(\vec{v})$ are standard. The language is multi-threaded: $\texttt{spawn}\, t$ starts a new thread which evaluates t in parallel with the spawning thread. The expression $\texttt{new}\, L$ dynamically creates a new lock, like instantiating Java's `ReentrantLock` class. The operations $v.\,\texttt{lock}$ and $v.\,\texttt{unlock}$ denote lock acquisition and release. The conditional $\texttt{if}\ v.\,\texttt{trylock}\ \texttt{then}\ e_1\ \texttt{else}\ e_2$ checks the availability of a lock v for the current thread, in which case v is taken.

3 Operational semantics

The operational semantics of the calculus is split into steps at the local level of one thread and at the global level. We focus here on global rules which concern more than one sequential thread or lock-manipulating steps. See [10] for the full semantics. Local configurations are written as $\sigma \vdash e$ and local reduction steps as $\sigma \vdash e \to \sigma' \vdash e'$, where σ is the *heap*, a finite mapping from references to objects and locks. Re-entrant locks are needed for recursive method calls. A lock is either *free* (represented by the value 0), or *taken* by a thread p (represented by $p(n)$ for $n \geq 1$ where n specifies that p holds the lock n times). A global configuration $\sigma \vdash P$ consists of a shared heap σ and a "set" of processes, where the processes are given by the following grammar:

$$P ::= \mathbf{0} \mid P \parallel P \mid p\langle t\rangle \qquad \text{processes/named threads} \tag{1}$$

$\mathbf{0}$ represents the empty process, $P_1 \parallel P_2$ the parallel composition of P_1 and P_2, and $p\langle t\rangle$ a process (or named thread), where p is the process identity and t the thread being executed. The binary $\parallel$-operator is associative and commutative with $\mathbf{0}$ as neutral element. Furthermore, thread identities must be *unique*. Global steps are of the form

$$\sigma \vdash P \to \sigma' \vdash P' \ . \tag{2}$$

The corresponding rules are given in Table 2. Rule R-LIFT lifts the local reduction steps to the global level and R-PAR expresses interleaving of the parallel composition of threads. Spawning a new thread is covered in rule R-SPAWN. The new thread gets a fresh identity and runs in parallel with the spawning thread. Rule R-NEWL creates a new lock and extends the heap with a fresh identity l and the lock is initially free. The lock can be taken, if it is free, or a thread already holding the lock can execute the locking statement once more, increasing the lock-count by one (cf. R-LOCK$_1$ and R-LOCK$_2$). The R-TRYLOCK-rules describe conditional lock taking. If the lock l is available for a thread, the expression l. `trylock` evaluates to true and the first branch of the conditional is taken (cf. the first two R-TRYLOCK-rules). Additionally, the thread acquires the lock. If the lock is unavailable, the else-branch is taken and the lock is unchanged (cf. R-TRYLOCK$_3$). Unlocking works dually and only the thread holding the lock can execute the unlock-statement on that lock. If the lock has value 1, the lock is free afterwards, and with a lock count of 2 or larger, it is decreased by 1 in the step (cf. R-UNLOCK$_1$ and R-UNLOCK$_2$). The R-ERROR-rules formalize misuse of a lock: unlocking a non-free lock by a thread that does not own it or unlocking a free lock (cf. R-ERROR$_1$ and R-ERROR$_2$). Both steps result in an `error`-term.

4 The type and effect system

A type and effect system combines rules for *well-typedness* with an *effect* part [1]. Here, effects track the use of locks and capture how many times a lock is taken or released. The underlying typing part is standard (the syntax for types is given in Table 1) and ensures, e.g., that actual parameters of method calls match the expected types for that method and that an object can handle an invoked method.

The type and effect system is given in Table 3 (for the thread local level) and Table 4 (for the global level). At the local level, the derivation system deals with expressions (which subsume threads). Judgments of the form

$$\Gamma; \Delta_1 \vdash e : T :: \Delta_2[\&v] \tag{3}$$

are interpreted as follows: Under the type assumptions Γ, an expression e is of type T. The effect part is captured by the effect or lock contexts: With the lock-status Δ_1 before the e, the status after e is given by Δ_2. The typing contexts (or type environments) Γ contain the type assumptions for variables; i.e., they bind variables x to their types, and are of the form $x_1{:}T_1, \ldots, x_n{:}T_n$, where we silently assume the x_i's are all different. This way, Γ is also considered a finite mapping from variables to types. By $dom(\Gamma)$ we refer to the domain of that mapping and write $\Gamma(x)$ for the type of variable x. Furthermore, we write $\Gamma, x{:}T$ for extending Γ with the binding $x{:}T$, assuming that $x \notin dom(\Gamma)$. To represent the effects of lock-handling, we use *lock environments* (denoted by Δ). At the local level of one single thread, the lock environments are of the form $v_1{:}n_1, \ldots, v_k{:}n_k$, where a value v_i is either a variable x_i or a lock reference l_i, but not the unit value. Furthermore, all v_i's are assumed to be different. The natural number n_i represents the lock status, and is either 0 in case the lock is marked as free, or n (with $n \geq 1$) capturing that the lock is taken n times by the thread under consideration. We use the same notations as for type contexts, i.e., $dom(\Delta)$ for the domain of Δ, further $\Delta(v)$ for looking up the lock status of the lock v in Δ, and $\Delta, v{:}n$ for extending Δ with a new binding, assuming $v \notin dom(\Delta)$. We write $\bullet$ for the empty context, containing no bindings. A

$$\frac{\sigma \vdash t \rightarrow \sigma' \vdash t'}{\sigma \vdash p\langle t\rangle \rightarrow \sigma' \vdash p\langle t'\rangle}\ \text{R-LIFT} \qquad \frac{\sigma \vdash P_1 \rightarrow \sigma' \vdash P_1'}{\sigma \vdash P_1 \parallel P_2 \rightarrow \sigma' \vdash P_1' \parallel P_2}\ \text{R-PAR}$$

$$\frac{p' \textit{ fresh}}{\sigma \vdash p\langle \mathtt{let}\ x{:}T = \mathtt{spawn}\ t'\ \mathtt{in}\ t\rangle \rightarrow \sigma \vdash p\langle \mathtt{let}\ x : T = ()\ \mathtt{in}\ t\rangle \parallel p'\langle t'\rangle}\ \text{R-SPAWN}$$

$$\frac{l \notin dom(\sigma) \qquad \sigma' = \sigma[l \mapsto 0]}{\sigma \vdash p\langle \mathtt{let}\ x{:}T = \mathtt{new\ L\ in}\ t\rangle \rightarrow \sigma' \vdash p\langle \mathtt{let}\ x{:}T = l\ \mathtt{in}\ t\rangle}\ \text{R-NEWL}$$

$$\frac{\sigma(l) = 0 \qquad \sigma' = \sigma[l \mapsto p(1)]}{\sigma \vdash p\langle \mathtt{let}\ x : T = l.\,\mathtt{lock}\ \mathtt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \mathtt{let}\ x : T = l\ \mathtt{in}\ t\rangle}\ \text{R-LOCK}_1$$

$$\frac{\sigma(l) = p(n) \qquad \sigma' = \sigma[l \mapsto p(n+1)]}{\sigma \vdash p\langle \mathtt{let}\ x : T = l.\,\mathtt{lock}\ \mathtt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \mathtt{let}\ x : T = l\ \mathtt{in}\ t\rangle}\ \text{R-LOCK}_2$$

$$\frac{\sigma(l) = 0 \qquad \sigma' = \sigma[l \mapsto p(1)]}{\sigma \vdash p\langle \mathtt{let}\ x : T = \mathtt{if}\ \ l.\,\mathtt{trylock}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \mathtt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \mathtt{let}\ x : T = e_1\ \mathtt{in}\ t\rangle}\ \text{R-TRYLOCK}_1$$

$$\frac{\sigma(l) = p(n) \qquad \sigma' = \sigma[l \mapsto p(n+1)]}{\sigma \vdash p\langle \mathtt{let}\ x : T = \mathtt{if}\ \ l.\,\mathtt{trylock}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \mathtt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \mathtt{let}\ x : T = e_1\ \mathtt{in}\ t\rangle}\ \text{R-TRYLOCK}_2$$

$$\frac{\sigma(l) = p'(n) \qquad p \neq p'}{\sigma \vdash p\langle \mathtt{let}\ x : T = \mathtt{if}\ l.\,\mathtt{trylock}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\ \mathtt{in}\ t\rangle \rightarrow \sigma \vdash\ p\langle \mathtt{let}\ x : T = e_2\ \mathtt{in}\ t\rangle}\ \text{R-TRYLOCK}_3$$

$$\frac{\sigma(l) = p(1) \qquad \sigma' = \sigma[l \mapsto 0]}{\sigma \vdash p\langle \mathtt{let}\ x : T = l.\,\mathtt{unlock}\ \mathtt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \mathtt{let}\ x : T = l\ \mathtt{in}\ t\rangle}\ \text{R-UNLOCK}_1$$

$$\frac{\sigma(l) = p(n+2) \qquad \sigma' = \sigma[l \mapsto p(n+1)]}{\sigma \vdash p\langle \mathtt{let}\ x : T = l.\,\mathtt{unlock}\ \mathtt{in}\ t\rangle \rightarrow \sigma' \vdash p\langle \mathtt{let}\ x : T = l\ \mathtt{in}\ t\rangle}\ \text{R-UNLOCK}_2$$

$$\frac{\sigma(l) = p'(n) \qquad p \neq p'}{\sigma \vdash p\langle \mathtt{let}\ x : T = l.\,\mathtt{unlock}\ \mathtt{in}\ t\rangle \rightarrow \sigma \vdash p\langle \mathtt{error}\rangle}\ \text{R-ERROR}_1$$

$$\frac{\sigma(l) = 0}{\sigma \vdash p\langle \mathtt{let}\ x : T = l.\,\mathtt{unlock}\ \mathtt{in}\ t\rangle \rightarrow \sigma \vdash p\langle \mathtt{error}\rangle}\ \text{R-ERROR}_2$$

Table 2. Global semantics

lock context Δ corresponds to a local view on the heap σ in that Δ contains the status of the locks from the perspective of one thread, whereas the heap σ in the global semantics contains the status of the locks from a global perspective. See also Definition 3 of projection later, which connects heaps and lock contexts. The final component of the judgment from Equation 3 is the value v after the &-symbol. If the type T of e is the type L for lock-references, type effect system needs information in which variable resp. which lock reference is returned. If $T \neq$ L, that information is missing; hence we write [&v] to indicate that it's "optional". In the following we concentrate mostly on the rules dealing with locks, and therefore with an &v-part in the judgment.

At run-time, expressions do not only contain variables (and the unit value) as values but also references. They are stored in the heap σ. To check the well-typedness of configurations at run-time, we extend the type and effect judgment from Equation 3 to

$$\sigma;\Gamma;\Delta_1 \vdash e : T :: \Delta_2[\&v] \tag{4}$$

The rules of Table 3 are mostly straightforward. We concentrate on the rules relevant for lock handling, the full set of rules is shown in [10]. To define the rules, we need two additional auxiliary functions. We assume that the definition of all classes is given. As this information is static, we do not explicitly mention the corresponding "class table" in the rules; relevant information from the class definitions is referred to in the rules by $\vdash C : \vec{T} \to C$ (the constructor of class C takes parameters of types $\vec{T}$ as arguments; the "return type" of the constructor corresponds to C), $\vdash C.m : \vec{T} \to T :: \Delta_1 \to \Delta_2$ (method m of class C takes input of type $\vec{T}$ and returns a value of type T). Concerning the effects, the lock status of the parameters must be larger or equal as specified in the pre-condition Δ_1, and the effect of method m is the change from Δ_1 to Δ_2. Similarly, $\vdash C.f : T$ means that the field f of instances of class C is of type T. Because fields simply contain values, they have no effect.

Values have no effect and thus $\Delta_1 = \Delta_2$ (cf. the rule T-VAL). A conditional expression is well-typed with type T if the conditional expression is a boolean and if both branches have the common type T. Also for the effect, rule T-COND insists that both branches are well-typed with the same pre- and post-condition. Field update in rule T-ASSIGN has no effect, and the type of the field must coincide with the type of the value on the right-hand side of the update.

For looking up a field containing a lock reference (cf. T-FIELD), the local variable used to store the reference is assumed with a lock-counter of 0. Rule T-LET, dealing with the local variable scopes and sequential composition, requires some explanation. First, it deals only with the cases not covered by T-NEWL or T-FIELD, which are excluded by the first premise. The two recursive premises dealing with the sub-expressions e and t basically express that the effect of e precedes the one for t: The post-condition Δ_2 of e is used in the pre-condition when checking t, and the post-condition Δ_3 after t in the premise then yields the overall postcondition in the conclusion. Care, however, needs to be taken in the interesting situation where e evaluates to a lock reference: In this situation the lock can be referenced in t by the local variable x *or* by the identifier which is handed over having evaluated e, i.e., via v' in the rule. Note that the body is analysed under the assumption that originally x, which is an alias of v', has the lock-counter 0. The last side condition deals with the fact that after executing e, only *one* lock reference can be handed over to t, all others have either been existing *before* the let-expression or become "garbage" after e, since there is no way in t to refer to them. To avoid hanging locks, the rule therefore requires that all lock values *created* while executing e must end free, i.e., they must have a lock count of 0 in Δ_2. This is formalized in the predicate $FE(\Delta_1,\Delta_2,v)$ in the rule's last premise where $FE(\Delta_1,\Delta_2,v)$ holds if $\Delta_2 = \Delta_1', \vec{v}{:}\vec{0}, v{:}n$ for some Δ_1' such that $dom(\Delta_1') = dom(\Delta_1)$ or $dom(\Delta_1', v{:}n) = dom(\Delta_1)$.

As for method calls in rule T-CALL, the premise $\vdash C.m : \vec{T} \to T :: \Delta_1' \to \Delta_2'$ specifies $\vec{T} \to T$ as the type of the method and $\Delta_1' \to \Delta_2'$ as the effect; this corresponds to looking up the definition of the class including their methods from the class table. To be well-typed, the actual parameters must be of the required types $\vec{T}$ and the type of the call itself is T, as declared for the method. For the effect part, we can conceptually think of the pre-condition Δ_1' of the method definition as the *required* lock balances and Δ_1 the *provided* ones at the control point before the call. For the post-conditions, Δ_2' can be seen as the promised post-condition and Δ_2 the actual one. The premise $\Delta_1 \geq \Delta_1'[\vec{v}/\vec{x}]$ of the rule requires that the provided lock status of the locks passed as formal parameters must be larger or equal to those required by the precondition Δ_1' declared for the method. The lock status *after* the method is determined by adding the effect (as the *difference* between the promised post-condition and the required pre-condition) to the provided lock status Δ_1 before the call. In the premises, we formalize those checks and calculations as follows:

Definition 1. *Assume two lock environments Δ_1 and Δ_2. The sum $\Delta_1 + \Delta_2$ is defined point-wise, i.e., $\Delta = \Delta_1 + \Delta_2$ is given by: $\Delta \vdash v : n_1 + n_2$ if $\Delta_1 \vdash v : n_1$ and $\Delta_2 \vdash v : n_2$. If $\Delta_1 \vdash v : n_1$ and $\Delta_2 \not\vdash v$ then $\Delta \vdash v : n_1$, and dually $\Delta \vdash v : n_2$, when $\Delta_1 \not\vdash v$ and $\Delta_2 \vdash v : n_2$. The comparison of two contexts is defined point-wise, as well: $\Delta_1 \geq \Delta_2$ if $dom(\Delta_1) \supseteq dom(\Delta_2)$ and for all $v \in dom(\Delta_2)$, we have $n_1 \geq n_2$, where $\Delta_1 \vdash v : n_1$ and $\Delta_2 \vdash v : n_2$. The difference $\Delta_1 - \Delta_2$ is defined analogously. Furthermore we use the following short-hand: for $v \in dom(\Delta)$, $\Delta + v$ denotes the lock context Δ', where $\Delta'(v) = 1$ if $\Delta(v) = 0$, and $\Delta'(v) = n+1$, if $\Delta(v) = n$. $\Delta - v$ is defined analogously.*

For the effect part of method specifications $C.m :: \Delta_1 \rightarrow \Delta_2$, the lock environments Δ_1 and Δ_2 represent the pre- and post-conditions for the lock parameters. We have to be careful how to *interpret* the assumptions and commitments expressed by the lock environments. As usual, the formal parameters of a method have to be unique; it's not allowed that a formal parameter occurs twice in the parameter list. Of course, the assumption of uniqueness does not apply to the *actual* parameters; i.e., at run-time, two different actual parameters can be *aliases* of each other. The consequences of that situation are discussed in the next example.

Example 1 (Method parameters and aliasing). Consider the following code:

Listing 1.1. Method with 2 formal parameters

```
m(x1:L, x2:L) {
   x1.unlock;x2.unlock
}
```

Method m takes two lock parameters and performs a lock-release on each one. As for the effect specification, the precondition Δ_1 should state that the lock stored in x_1 should have at least value 1, and the same for x_2, i.e.,

$$\Delta_1 = x_1{:}1, x_2{:}1 \tag{5}$$

With Δ_1 as pre-condition, the effect type system accepts the method of Listing 1.1 as type correct, because the effects on x_1 and x_2 are checked *individually.* If at run-time the actual parameters, say l_1 and l_2 happen to be *not aliases*, and if each of them satisfies the precondition of Equation 5 *individually,* i.e., at run-time, the lock environment $\Delta_1' = \Delta_1[l_1/x_1][l_2/x_2]$ i.e.,

$$\Delta_1' = l_1{:}1, l_2{:}1 \tag{6}$$

executing the method body does not lead to a run-time error. If, however, the method is called such that x_1 and x_2 become aliases, i.e., called as $o.m(l,l)$, where the lock value of l is 1, it results in a run-time error. That does *not* mean that the system works *only* if there is no aliasing on the actual parameters. The lock environments express *resources* (the current lock balance) and if x_1 and x_2 happen to be aliases, the resources must be *combined.* This means that if we substitute in Δ_1 the variables x_1 and x_2 by the same lock l, the result of the substitution is

$$\Delta_1' = \Delta_1[l/x_1][l/x_2] = l{:}(1+1)$$

i.e., l is of balance 2. □

This motivates the following definition of substitution for lock environments.

Definition 2 (Substitution for lock environments). *Given a lock environment Δ of the form $\Delta = v_1{:}n_1,\ldots,v_k{:}n_k$, with $k \geq 0$, and all the natural numbers $n_i \geq 0$. The result of the* substitution *of a variable x by a value v in Δ is written $\Delta[v/x]$ and defined as follows. Let $\Delta' = \Delta[v/x]$. If $\Delta = \Delta'', v{:}n_v, x{:}n_x$, then $\Delta' = \Delta'', v{:}(n_v + n_x)$. If $\Delta = \Delta'', x{:}n$ and $v \notin dom(\Delta'')$, then $\Delta' = \Delta'', v{:}n$. Otherwise, $\Delta' = \Delta$.*

Example 2 (Aliasing). The example continues from Example 1, i.e., we are given the method definition of Listing 1.1. Listing 1.2 shows the situation of a *caller* of m where first, the actual parameters are *without* aliases. Before the call, each lock (stored in the fields f_1 and f_2) has a balance of 1, as required in m's precondition, and the method body individually unlocks each of them once.

Listing 1.2. Method call, no aliasing

```
f1 := new L;
f2 := new L;          // f1 and f2: no aliases
f1.lock; f2.lock;
o.m(f1,f2);
```

If we change the code of the call site by making f_1 and f_2 aliases, setting $f_2 := f_1$ in the second line, instead of f_2 :=`new L`, again there is *no* run-time error, as after executing f_1`.lock` and f_2`.lock`, the actual balance of the single lock stored in f_1 as well as in f_2 is 2. □

Why aliasing f_1 and f_2 in the situation of Example 2 is unproblematic is illustrated in Figure 1. Figures 1(a) and 1(b) show the change of two different locks over time, where the y-axis represents the lock balance. The behavior of each lock is that it starts at a lock-count of zero, counts up and down according to the execution of `lock` and `unlock` and at the end reaches 0 again. It is important that at no point in time the lock balance can be *negative,* as indicated by the red, dashed arrow in the left sub-figure.

Connecting the figures to the example above, the two lock histories correspond to the situation of Listing 1.2 where f_1 and f_2 are *no* aliases. When they *are* aliases, the overall history looks as shown in Figure 1(c). This combined history, clearly, satisfies the mentioned condition for a lock behavior: it starts and ends with a lock balance of 0 and it never reaches a negative count as it is simply the "sum" of the individual lock histories.

The identity of a new thread is irrelevant, i.e., spawning carries type `Unit` (cf. T-SPAWN). Note for the effect part of T-SPAWN that the pre-condition for checking the thread t in the premise of the rule is the empty lock context $\bullet$. The reason is that the new thread starts without holding any lock. Note further that for the post-condition of the newly created thread t, all locks that may have been acquired while executing t must have been released again; this is postulated by $\Delta' \vdash \mathit{free}$. Typing for new locks is covered by T-NEWL. As for the effect, the pre-context Δ_1 is extended by a binding for the new lock initially assumed to be free, i.e., the new binding is x:0. The two operations for acquiring and releasing a lock carry the type `L`. The type rules here are formulated on the thread-local level, i.e., irrespective of any other thread. Therefore, the lock contexts also contain no information about which thread is currently in possession of a non-free lock, since the rules are dealing with one local thread only. The effect of taking a lock is unconditionally to increase the lock counter in the lock context by one (cf. Definition 1). Dually in rule T-UNLOCK, $\Delta - v$ decreases v's lock counter by one. To do so safely, the thread must hold the lock before the step, as required by the premise $\Delta \vdash v : n+1$. The expression for tentatively taking a lock is a two-branched conditional. The first branch e_1 is executed if the lock is held, the second branch e_2 is executed if not. Hence, e_1 is analysed in the lock context $\Delta_1 + v$ as precondition, whereas e_2 uses Δ_1 unchanged (cf. T-TRYLOCK). As for ordinary conditionals, both branches coincide concerning their type and the post-condition of the effects, which in turn also are the type, resp. the post-condition of the overall expression.

The type and effect system in Table 3 dealt with expressions at the local level, i.e., with expression e and threads t of the abstract syntax of Table 1. We proceed analysing the language "above" the level of one thread, and in particular of global configurations as given in Equation 1.

The effect system at the local level uses lock environments to approximate the effect of the expression on the locks (cf. Equation 4). Lock environments Δ are thread-local views on the status of the locks, i.e., which locks the given thread holds and how often. In the reduction semantics, the locks are allocated in the (global) heap σ, which contains the status of all locks (together with the instance states of all allocated objects). The thread-local view can be seen as a *projection* of the heap to the thread, as far as the

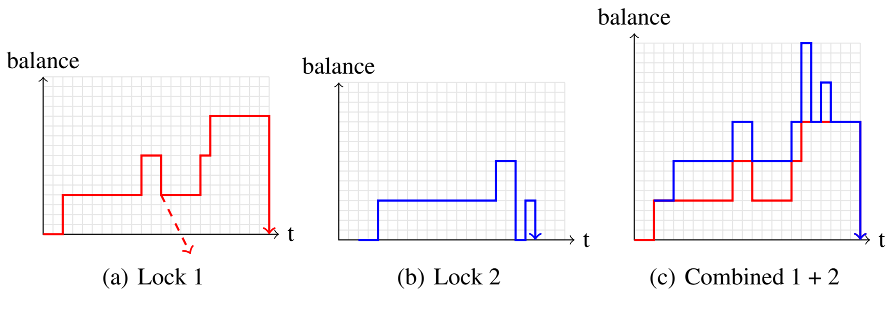

Fig. 1. Two lock histories

$$\frac{\sigma;\Gamma \vdash v : \mathtt{L} \qquad \Delta \vdash v}{\sigma;\Gamma;\Delta \vdash v : \mathtt{L} :: \Delta \& v}\ \text{T-VAL} \qquad \frac{\sigma;\Gamma \vdash v : \mathsf{Bool} \quad \sigma;\Gamma;\Delta_1 \vdash e_1 : T :: \Delta_2 \& v \quad \sigma;\Gamma\Delta_1 \vdash e_2 : T :: \Delta_2 \& v}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T :: \Delta_2 \& v}\ \text{T-COND}$$

$$\frac{\sigma;\Gamma \vdash v' : C \quad \vdash C.f : \mathtt{L} \quad \sigma;\Gamma,x{:}\mathtt{L};\Delta_1,x{:}0 \vdash t : T :: \Delta_2 \& v}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{let}\ x : \mathtt{L} = v'.f\ \mathtt{in}\ t : T :: \Delta_2 \& v}\ \text{T-FIELD}$$

$$\frac{\sigma;\Gamma;\Delta \vdash v_1 : C :: \Delta \quad \vdash C.f_i : T_i \quad \sigma;\Gamma;\Delta \vdash v_2 : T_i :: \Delta \& v_2}{\sigma;\Gamma;\Delta \vdash v_1.f := v_2 : T_i :: \Delta \& v_2}\ \text{T-ASSIGN}$$

$$\frac{e \notin \{\mathtt{new}\,\mathtt{L},\ v.f\} \qquad \sigma;\Gamma;\Delta_1 \vdash e : T_1 :: \Delta_2 \& v' \qquad (\sigma;\Gamma,x{:}T_1;\Delta_2,x{:}0 \vdash t : T_2 :: \Delta_3 \& v'')[v'/x] \qquad FE(\Delta_1,\Delta_2,v')}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{let}\ x : T_1 = e\ \mathtt{in}\ t : T_2 :: \Delta_3[v'/x] \& v''[v'/x]}\ \text{T-LET}$$

$$\frac{\begin{array}{c}\vdash C.m = \lambda\vec{x}.t \qquad \sigma;\Gamma \vdash \vec{v} : \vec{T} \qquad \sigma;\Gamma \vdash v : C \qquad \vdash C.m : \vec{T} \to T :: \Delta_1' \to \Delta_2' \\ \Delta_1 \geq \Delta_1'[\vec{v}/\vec{x}] \qquad \Delta_2 = \Delta_1 + (\Delta_2' - \Delta_1')[\vec{v}/\vec{x}] \qquad T \neq \mathtt{L}\end{array}}{\sigma;\Gamma;\Delta_1 \vdash v.m(\vec{v}) : T :: \Delta_2}\ \text{T-CALL}$$

$$\frac{\sigma;\Gamma,x{:}\mathtt{L};\Delta_1,x{:}0 \vdash t : T :: \Delta_2 \& v}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{let}\ x{:}\mathtt{L} = \mathtt{new}\ \mathtt{L}\ \mathtt{in}\ t : T :: \Delta_2 \& v}\ \text{T-NEWL} \qquad \frac{\sigma;\Gamma;\bullet \vdash t : T :: \Delta' \qquad \Delta' \vdash \mathit{free}}{\sigma;\Gamma;\Delta \vdash \mathtt{spawn}\ t : \mathtt{Unit} :: \Delta}\ \text{T-SPAWN}$$

$$\frac{\Delta \vdash v \qquad \sigma;\Gamma \vdash v : \mathtt{L}}{\sigma;\Gamma;\Delta \vdash v.\ \mathtt{lock} : \mathtt{L} :: \Delta + v \& v}\ \text{T-LOCK} \qquad \frac{\Delta \vdash v : n+1 \qquad \sigma;\Gamma \vdash v : \mathtt{L}}{\sigma;\Gamma;\Delta \vdash v.\ \mathtt{unlock} : \mathtt{L} :: \Delta - v \& v}\ \text{T-UNLOCK}$$

$$\frac{\sigma;\Gamma \vdash v : \mathtt{L} \qquad \sigma;\Gamma;\Delta_1 + v \vdash e_1 : T :: \Delta_2 \& v' \qquad \sigma;\Gamma;\Delta_1 \vdash e_2 : T :: \Delta_2 \& v'}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{if}\ v.\ \mathtt{trylock}\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T :: \Delta_2 \& v'}\ \text{T-TRYLOCK}$$

Table 3. Type and effect system (thread-local)

locks are concerned. This projection is needed to connect the local part of the effect system to the global one (cf. T-THREAD of Table 4).

Definition 3 (Projection). *Assume a heap σ with $\sigma \vdash ok$ and a thread p.*[1] *The* projection *of σ onto p, written $\sigma \downarrow_p$ is inductively defined as follows:*

$$\begin{aligned}
\bullet \downarrow_p &= \bullet \\
(\sigma, l{:}0) \downarrow_p &= \sigma \downarrow_p, l{:}0 \\
(\sigma, l{:}p(n)) \downarrow_p &= \sigma \downarrow_p, l{:}n \\
(\sigma, l{:}p'(n)) \downarrow_p &= \sigma \downarrow_p, l{:}0 \qquad \text{if } p \neq p' \\
(\sigma, o{:}C(\vec{v})) \downarrow_p &= \sigma \downarrow_p \ .
\end{aligned}$$

Note the case where a lock l is held by a thread named p' different from the thread p we project onto, the projection makes l free, i.e., l:0. At first sight, it might look strange that the locks appears to be locally free where it is actually held by another thread. The reason is that the type system captures a *safety* property about the locks and furthermore that locks ensure *mutual exclusion* between threads. Safety means that the effect type system gives, as usual, no guarantee that the local thread can actually take the lock, it makes a statement about what happens after the thread has taken the

[1] Cf. the technical report [10] for the standard definition of $\sigma \vdash ok$.

lock. If the local thread can take the lock, the lock must be free right before that step. The other aspect, namely mutual exclusion, ensures that for the thread that has the lock, the effect system calculates the balance without taking the effect of other thread into account. This reflects the semantics as the locks of course guarantee mutual exclusion. As locks are manipulated *only* via $l.$ `lock` and $l.$ `unlock`, there is no interference by other threads, which justifies the local, *compositional* analysis.

Now to the rules of Table 4, formalizing judgments of the form

$$\sigma \vdash P : ok\ , \tag{7}$$

where P is given as in Equation 1. In the rules, we assume that σ is well-formed, i.e., $\sigma \vdash ok$. The empty set of threads or processes $\mathbf{0}$ is well-formed (cf. T-EMPTY). Well-typedness is a "local property" of threads, i.e., it is compositional: a parallel composition is well-typed if both sub-configurations are (cf. T-PAR). A process $p\langle t\rangle$ is well-typed if its code t is (cf. T-THREAD). As precondition Δ_1 for that check, the projection of the current heap σ is taken. . As for the post-condition Δ_2, we require that the thread has given back all the locks, postulated by $\Delta_2 \vdash free$. The remaining rules do not deal with run-time configurations $\sigma \vdash P$, but with the static code as given in the class declarations/definitions. Rule T-METH deals with method declarations. The first premise looks up the *declaration* of method m in the class table. The declaration contains, as usual, the argument types and the return type of the method. Beside that, the effect specification $\Delta_1 \to \Delta_2$ specifies the pre- and post-condition on the lock parameters. We assume that the domain of Δ_1 and Δ_2 correspond exactly to the lock parameters of the method. The second premise then checks the code of the method body against that specification. So t is type-checked, under a type and effect context extended by appropriate assumptions for the formal parameters $\vec{x}$ and by assuming type C for the self-parameter this. Note that the method body t is checked with an empty heap $\bullet$ as assumption. As for the post-condition Δ_2, Δ_2' of the body, Δ_2' contains lock variables *other* than the formal lock parameters (which are covered by Δ_2). The last premise requires that the lock counters of Δ_2' must be free after t. The role of the lock contexts as pre- and post-conditions for method specifications and the corresponding premises of rule T-CALL are illustrated in Figure 2. Assume two methods m and n, where m calls n, i.e., m is of the form $m()\{\ldots; x.n()\ldots\}$. Let's assume the methods operate on one single lock, whose behavior is illustrated by the first two sub-figures of Figure 2. The history in Figure 2(a) is supposed to represent the lock behavior m up to the point where method n is called, and Figure 2(b) gives the behavior of n in isolation. The net effect of method n is to decrease the lock-count by one (indicated by the dashed arrow), namely by unlocking the lock twice but locking it once afterwards again. It is not good enough as a specification for method n to know that the overall effect is a decrease by one. It is important that at the point where the method is called, the lock balance must be *at least* 2. Thus, the effect specification is $\Delta_1 \to \Delta_2$, where Δ_1 serves as precondition for all formal lock parameters of the method, and T-CALL requires current lock balances to be larger or equal to the one specified. The type system requires that the locks are handed over via parameter passing and the connection between the lock balances of the actual parameters with those of the formal ones is done by the form of substitution given in Definition 2. The actual value of the lock balances after the called method n is then determined by the lock balances before the call plus the net-effect of that method. See Figure 2(c) for combining the two histories of Figures 2(a) and 2(b). Finally, a class definition class $C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T};\vec{M}\}$ is dealt with in rule T-CLASS, basically checking that all method definitions are well-typed. For a program (a sequence of class definitions) to be well-typed, all its classes must be well-typed (we omit the rule).

5 Correctness

We prove the correctness of our analysis. A crucial part is subject reduction, i.e., the preservation of well-typedness under reduction. (The full proofs can be found in [10].)

Next we prove subject reduction for the effect part of the system of Tables 3 and 4.

Lemma 1 (Substitution). *Let x be a variable of type* `L` *and l be a lock reference If $\Delta_1 \vdash t :: \Delta_2 \& v$, then $\Delta_1[l/x] \vdash t[l/x] :: \Delta_2[l/x] \& v[l/x]$.*

The next lemma expresses that given a lock environment Δ_1 as precondition for an expression e such that the effect of e leads to a post-condition of Δ_2, e is still well-typed if we assume a Δ_1' where the lock balances are increased, and the corresponding post-condition is then increased accordingly.

Lemma 2 (Weakening). *If $\Delta_1 \vdash e :: \Delta_2$, then $\Delta_1 + \Delta \vdash e :: \Delta_2 + \Delta$.*

Lemma 3 (Subject reduction (local)). *Let $\sigma \vdash t$ be well-typed. Assume further $\Delta_1 \vdash t :: \Delta_2 \& v$ where $\Delta_1 = \sigma\downarrow_p$ for a thread identifier p and $\Delta_2 \vdash free$. If $\sigma \vdash t \rightarrow \sigma' \vdash t'$, then $\Delta_1' \vdash t' :: \Delta_2' \& v'$, with $\Delta_1' = \sigma\downarrow_p$ and with $\Delta_2' \vdash free$.*

Lemma 4 (Subject reduction (global)). *If $\sigma \vdash P : ok$ and $\sigma \vdash P \rightarrow \sigma' \vdash P'$ where the reduction step is* not *one of the two* R-ERROR*-rules, then $\sigma' \vdash P' : ok$.*

Lemma 5. *Let $P = P' \parallel p\langle t\rangle$. If $\sigma \vdash P : ok$ then $\sigma \vdash P \not\rightarrow \sigma \vdash P' \parallel p\langle$*`error`*$\rangle$.*

The next result captures one of the two aspects of correct lock handling, namely that never an exception is thrown by inappropriately unlocking a lock.

Theorem 1 (Well-typed programs are lock-error free). *Given a program in its initial configuration $\bullet \vdash P_0 : ok$. Then it's not the case that $\bullet \vdash P_0 \longrightarrow^* \sigma' \vdash P \parallel p\langle$*`error`*$\rangle$.*

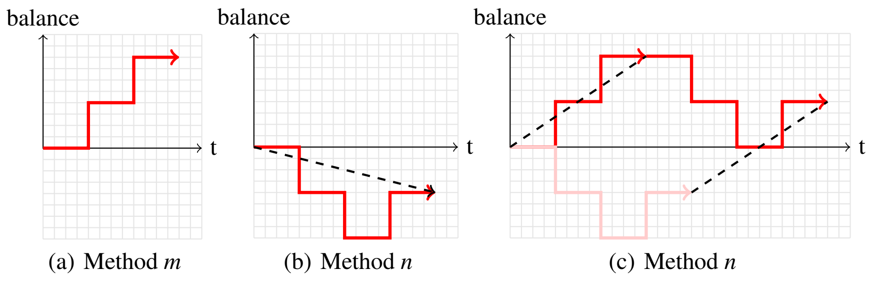

Fig. 2. Lock balance of methods m and n

The second aspect of correct lock handling means that a thread should release all locks before it terminates. We say, a configuration $\sigma \vdash P$ has a *hanging lock* if $P = P' \parallel p\langle \texttt{stop} \rangle$ where $\sigma(l) = p(n)$ with $n \geq 1$, i.e., one thread p has terminated but there exists a lock l still in possession of p.

Theorem 2 (Well-typed programs have no hanging locks). *Given a program in its initial configuration $\bullet \vdash P_0 : ok$. Then it's not the case that $\bullet \vdash P_0 \longrightarrow^* \sigma' \vdash P'$, where $\sigma' \vdash P'$ has a hanging lock.*

6 Related work

Our static type and effect system ensures proper usage of non-lexically scoped locks in a concurrent object-oriented calculus to prevent run-time errors and unwanted behaviors. As mentioned, the work presented here extends our previous work [12], dealing with transactions as a concurrency control mechanism instead of locks. The extension is non-trivial, mainly because locks have user-level identities. This means that, unlike transactions, locks can be passed around, can be stored in fields, and in general aliasing becomes a problem. Furthermore, transactions are not "re-entrant". See [11] for a more thorough discussion of the differences. There are many type systems and formal analyses to catch already a compile time various kinds of errors. For multi-threaded Java, static approaches so far are mainly done to detect data races or to guarantee freedom of deadlocks, of obstruction or of livelocks, etc. There have been quite a number of type-based approaches to ensure proper usage of resources of different kinds (e.g., file access, i.e., to control the opening and closing of files). See [6] for a recent, rather general formalization for what the authors call the resource usage analysis problem (the paper discusses approaches to safe resource usage in the literature). Unlike the type system proposed here, [6] considers type *inference* (or type reconstruction). Their language, a variant of the λ-calculus, however, is sequential. [15] uses a type and effect system to assure deadlock freedom in a calculus quite similar to ours in that it supports thread based concurrency and a shared mutable heap. On the surface, the paper deals with a different problem (deadlock freedom) but as *part* of that it treats the same problem as we, namely to avoind releasing free locks or locks not owned, and furthermore,

$$\frac{}{\sigma \vdash \mathbf{0} : ok}\ \textsc{T-Empty} \qquad \frac{\sigma \vdash P_1 : ok \qquad \sigma \vdash P_2 : ok}{\sigma \vdash P_1 \parallel P_2 : ok}\ \textsc{T-Par} \qquad \frac{\forall i.\ \vdash M_i : ok}{\vdash C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T};\vec{M}\} : ok}\ \textsc{T-Class}$$

$$\frac{\Delta_1 = \sigma\downarrow_p \qquad \sigma;\bullet;\Delta_1 \vdash t : T :: \Delta_2 \qquad t \neq \texttt{error} \qquad \Delta_2 \vdash \mathit{free}}{\sigma \vdash p\langle t \rangle : ok}\ \textsc{T-Thread}$$

$$\frac{\vdash C.m : \vec{T} \to T :: \Delta_1 \to \Delta_2 \qquad \bullet;\vec{x}{:}\vec{T},\mathsf{this}{:}C;\Delta_1 \vdash t : T :: \Delta_2,\Delta_2' \qquad \Delta_2' \vdash \mathit{free}}{\vdash C.m(\vec{x}{:}\vec{T})\{t\} : ok}\ \textsc{T-Meth}$$

Table 4. Type and effect system (global)

do not leave any locks hanging. The language of [15] is more low-level in that it supports pointer dereferencing, whereas our object-oriented calculus allows shared access on mutable storage only for the fields of objects and especially we do not allow pointer dereferencing. Pointer dereferencing makes the static analysis more complex as it needs to keep track of which thread is actually responsible for lock-releasing in terms of read and write permissions. We do not need the complicated use of ownership-concepts, as our language is more disciplined dealing with shared access: we strictly separate between *local* variables (not shared) and shared fields. In a way, the content of a local variable is "*owned*" by a thread; therefore there is no need to track the current owner across different threads to avoid bad interference. Besides that, our analysis can handle *re-entrant locks,* which are common in object-oriented languages such as Java or C$^\sharp$, whereas [15] covers only binary locks. The same restriction applies to [16], which represents a type system assuring race-freedom. Gerakios et.al. [5] present a uniform treatment of region-based management and locks. A type and effect system guarantees the absence of memory access violations and data races in the presence of region aliasing. The main subject in their work is regions, however, not locks. Re-entrant locks there are just used to protect regions, and they are implicit in the sense that each lock is associated with a region and has no identity. The regions, however, have an identity, they are non-lexically scoped and can be passed as arguments. The safety of the region-based management is ensured by a type and effect system, where the effects specify so-called region *capabilities.* Similar to our lock balances, the capabilities keep track of the "status" of the region, including a count on how many times the region is accessed and a *lock* count. As in our system, the static analysis keeps track of those capabilities and the soundness of the analysis is proved by subject reduction (there called "preservation"). [4] uses "flow sensitive" type qualifiers to statically correct resource usage such as file access in the context of a calculus with higher-order functions and mutable references. Also the Vault system [3] uses a type-based approach to ensure safe use of resources (for C-programs). Laneve et. al. [2] develop a type system for statically ensuring proper lock handling also for the JVM, i.e., at the level of byte code as part of Java's bytecode verifier. Their system ensures what is known as *structured locking*, i.e., (in our terminology), each method body is balanced as far as the locks are concerned, and at no point, the balance reaches below 0. As the work does not consider non-lexical locking as in Java 5, the conditions apply *per method* only. Extending [14], Iwama and Kobayashi [8] present a type system for multi-threaded Java programs on the level of the JVM which deals with non-lexical locking. Similar to our system, the type system guarantees absence of lock errors (as we have called it), i.e., that when a thread is terminated, it has released all its acquired locks and that a thread never releases a lock it has not previously acquired. Unlike our system, they cannot deal with method calls, i.e., the system analyses method bodies in isolation. However, they consider type *inference.*

7 Conclusion

We presented a static type and effect system to prevent certain errors for non-lexical lock handling as in Java 5. The analysis was formalized in an object-oriented calculus in the style of FJ. We proved the soundness of our analysis by subject reduction. Challenges for the static analysis addressed by our effect system are the following: with dynamic lock creation and passing of lock references, we face *aliasing* of lock references, and due to dynamic thread creation, the effect system needs to handle concurrency.

Aliasing is known to be tricky for static analysis; many techniques have been developed to address the problem. Those techniques are known as alias or pointer analyses, shape analyses, etc. With dynamic lock creation and since locks are *meant* to be shared, one would expect that a static analysis on lock-usage relies on some form of alias analysis. Interestingly, aliasing poses no real challenge for the specific problem at hand, under suitable assumptions on the use of locks and lock variables. The main assumption restricts passing the lock references via instance fields. Note that to have locks *shared between threads,* there are basically only two possible ways: hand over the identity of a lock via the thread constructor or via an instance field: it's not possible to hand the lock reference to another thread via method calls, as calling a method continues executing in the *same* thread. Our core calculus does not support thread constructors, as they can be expressed by ordinary method calls, and because passing locks via fields is more general and complex: passing a lock reference via a constructor to a new thread means locks can be passed only from a parent to a child thread. The effect system then enforces a *single-assignment* policy on lock fields. The analysis also shows that this restriction can be relaxed in that one allows assignment concerning fields whose lock status corresponds to a *free* lock. Concerning passing lock references within *one* thread, parameter passing must be used. The effect specification of the formal parameters contains information about the effect of the lock parameters. We consider the restriction *not* to re-assign a lock-variable as a natural programming guideline and common practice.

Like aliasing, concurrency is challenging for static analysis, due to interference. Our effect system checks the effect of interacting locks, which are some form of shared variables. An interesting observation is that locks are, of course not just shared variables, but they synchronize threads for which they ensure mutual exclusion. Ensuring absence of lock errors is thus basically a *sequential* problem, as one can ignore interference; i.e., a parallel program can be dealt with *compositionally*. See the simple, compositional rule for parallel composition in Table 4. The treatment is similar to the effect system for TFJ dealing with transactions instead of locks. However, in the transactional setting, the local view works for a different reason, as transactions are *not shared* between threads.

The treatment of the locks here is related to type systems governing *resource usage*. We think that our technique in this paper and a similar one used in our previous work could be applied to systems where run-time errors and unwanted behaviors may happen due to improperly using syntactical constructs for, e.g., opening/closing files, allocating/deallocating resources, with a non-lexical scope. Currently, the exceptional behavior due to lock-mishandling is represented as one single `error`-expression. Adding a more realistic exception mechanism including exception handling and finally-clauses to the calculus is a furthrer step in our research. Furthermore we plan to implement the system for empirical results. The combination of our two type and effect systems, one for TFJ [12] and one for the calculus in this paper, could be a step in setting up an integrated system for the applications where locks and transactions are reconciled.

Acknowledgements We are grateful to the very thorough anonymous reviewers for giving helpful and critical feedback and useful pointers to the literature.

References

1. T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Presse, 1999.
2. G. Bigliardi and C. Laneve. A type system for JVM threads. In *In Proceedings of 3rd ACM SIGPLAN Workshop on Types in Compilation (TIC2000)*, page 2003, 2000.
3. R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proceedings of the 2001 ACM Conference on Programming Language Design and Implementation*, pages 59–69, June 2001.
4. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2002.
5. P. Gerakios, N. Papaspyrou, and K. Sagonas. A concurrent language with a uniform treatment of regions and locks. In *Programming Language Approaches to Concurrency and Communication-eCentric Software EPTCS 17*, pages 79–93, 2010.
6. A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.
7. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '99*, pages 132–146. ACM, 1999. In *SIGPLAN Notices*.
8. F. Iwama and N. Kobayashi. A new type system for JVM lock primitives. In *ASIA-PEPM '02: Proceedings of the ASIAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 71–82, New York, NY, USA, 2002. ACM.
9. S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, Aug. 2005.
10. E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java. Technical Report (revised version) 402, University of Oslo, Dept. of Computer Science, Jan. 2011. `www.ifi.uio.no/~msteffen/publications.html#techreports`. A shorter version (extended abstract) has been presented at the NWPT'10.
11. T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe typing for transactional vs. lock-based concurrency in multi-threaded Java. In S. B. Pham, T.-H. Hoang, B. McKay, and K. Hirota, editors, *Proceedings of the Second International Conference on Knowledge and Systems Engineering, KSE 2010*, pages 188 – 193. IEEE Computer Society, Oct. 2010.
12. T. Mai Thuong Tran and M. Steffen. Safe commits for Transactional Featherweight Java. In D. Méry and S. Merz, editors, *Proceedings of the 8th International Conference on Integrated Formal Methods (iFM 2010)*, volume 6396 of *Lecture Notes in Computer Science*, pages 290–304 (15 pages). Springer-Verlag, Oct. 2010. An earlier and longer version has appeared as UiO, Dept. of Comp. Science Technical Report 392, Oct. 2009 and appeared as extended abstract in the Proceedings of NWPT'09.
13. S. Oaks and H. Wong. *Java Threads*. O'Reilly, Third edition, Sept. 2004.
14. R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.
15. K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *APLAS 2008*, volume 5356 of *Lecture Notes in Computer Science*, pages 155–170. Springer-Verlag, 2008.
16. T. Terauchi. Checking race freedom via linear programming. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 1–10, New York, NY, USA, 2008. ACM.