# Safe Locking for Multi-Threaded Java with Exceptions☆

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

*Dept. of Computer Science, University of Oslo*

**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 lock primitives on re-entrant locks. These operators may lead to runtime 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 a formal, object-oriented calculus which supports non-lexical lock handling and exceptions.

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. Furthermore, the exception handling mechanism complicates the control-flow and thus the analysis.

*Keywords:* Java, multi-threading, lock-based concurrency, non-lexical, re-entrant locks, exceptions, static analysis, type and effect systems

## 1. Introduction

With the advent of multiprocessors, multi-core architectures, and distributed web-based programs, effective parallel programming models and suitable language

☆The work has been partly supported by the EU-project FP7-231620 HATS (Highly Adaptable and Trustworthy Software using Formal Methods).

☆☆The list of authors is given in alphabetical order.

*Email addresses:* `einarj@ifi.uio.no` (Einar Broch Johnsen), `tmtran@ifi.uio.no` (Thi Mai Thuong Tran), `olaf@ifi.uio.no` (Olaf Owe), `msteffen@ifi.uio.no` (Martin Steffen,☆☆)

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 [21] and transactional memory, as formalized in *Transactional Featherweight Java* (TFJ) [16]. 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 (of locks or transactions) 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 [20]. This paper applies that approach to a calculus which supports *non-lexical* lock handling as in Java 5. Our approach guarantees absence of certain erroneous use of locks, in particular, to attempt to release a lock without owning it and to takes a lock without releasing it afterwards, which could lead to a deadlock. We call such a discipline *safe locking*.

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 [20]. 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. Exceptions complicate the sequential control flow by introducing non-local "jumps" from the place where an exception is raised to the one where it is caught and handled (or alternatively "falls through"). Not only does this require to over-approximate thrown (and potentially caught) exceptions, but also, the analysis must keept track of the different lock-status at the points where the exceptions may occur.

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 in our formalization. The paper extends the earlier conference version [17] by guaranteeing lock safety in the presence of exceptions. Furthermore, we include the full type and effect system and the correcness proofs in the work.

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. Section 6 extends the language and the analysis by covering throwing and catching exceptions in the style of Java. Sections 7 and 8 conclude with related and future work.

## 2. A concurrent, object-oriented calculus

The calculus used in this paper is a variant of Featherweight Java (FJ) [12] with concurrency and explicit lock support, but without inheritance and type casts. FJ is an object-oriented core language originally introduced to study typing issues related to Java, such as inheritance, subtype polymorphism, type casts, etc. A number of extensions have been developed for other language features, so FJ is today a generic name for Java-related core calculi. Following [16] and in contrast to the original FJ proposal, we ignore inheritance, subtyping, and type casts, as orthogonal to the issues at hand, but include imperative features such as destructive field updates, furthermore concurrency and lock handling.

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. Without inheritance, a class definition class $C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T};\vec{M}\}$ consists of a name $C$, a list of fields $\vec{f}$ with corresponding type declarations $\vec{T}$ (assuming that all $f_i$'s are different), and a list $\vec{M}$ of method definitions. 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`.

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

Table 1: Abstract syntax

The syntax distinguishes expressions $e$ and threads $t$. A thread $t$ is either a value $v$, the terminated thread `stop`, `error` representing abnormal termination, or sequential composition. The let-construct, as usual, binds $x$ in $t$. We write $fv(t)$ and $fv(e)$ for the free variables of $t$, resp. of $e$. The let-construct generalizes sequential composition: in $\texttt{let}\ x{:}T = e\ \texttt{in}\ t$, $e$ 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 $\texttt{let}\ x{:}T = e\ \texttt{in}\ t$ where the variable $x$ does not occur free in $t$. In the syntax, values $v$ are expressions that can not be evaluated further. In the core calculus, we leave unspecified standard values like booleans and integers, so values are references $r$, variables $x$ , and the unit value $()$. The set of variables includes the special variable this needed to refer to the current object. As for references, 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 are written $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$, the expressions $v.f$ and $v_1.f := v_2$ represent field access and field update respectively. Method calls are written $v.m(\vec{v})$ and object instantiation is $\texttt{new}\ C(\vec{v})$. The language is multi-threaded: $\texttt{spawn}\ t$ starts a new thread which evaluates $t$ in parallel with the spawning thread. The remaining constructs deal with lock handling. The expression $\texttt{new}\ L$ dynamically creates a new lock, which corresponds to instantiating Java's `ReentrantLock` class. The dual operations $v.\,\texttt{lock}$ and $v.\,\texttt{unlock}$ denote lock acquisition and release (the type system makes sure that the value $v$ is a reference to a lock). 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 in the step.

A note on the form of threads and expressions and the use of values may be in order. The syntax is restricted concerning where to use general expressions $e$. For example, the syntax does not allow field updates $e_1.f := e_2$, where the object whose field is being updated and the value used in the right-hand side are represented by general expressions that need to be evaluated first. It would be straightforward to relax the abstract syntax that way. We have chosen this presentation, as it slightly simplifies the operational semantics and the type and effect system later. With that restricted representation, we can get away with a semantics without evaluation contexts, using simple rewriting rules (and the let-syntax). Of course, this is not a real restriction in expressivity. For example, the mentioned expression $e_1.f := e_2$ can easily be expressed by $\mathtt{let}\, x_1 = e_1 \;\mathtt{in}\, (\mathtt{let}\, x_2 = e_2 \;\mathtt{in}\, x_1.f := x_2)$, making the evaluation order explicit. The transformation from the general syntax to the one of Table 1 is standard and known as CPS transformation, i.e., transformation into continuation-passing style.

## 3. Operational semantics

We proceed with the operational semantics of the calculus. The semantics is presented in two stages. The local level, described first, captures the sequential behavior of one thread. Afterwards, we present the behavior of global configurations, dealing with concurrent threads and lock handling.

Local configurations are of the form $\sigma \vdash e$, and local reduction steps of the form $\sigma \vdash e \rightarrow \sigma' \vdash e'$, where $\sigma$ is the *heap*, a finite mapping from references to objects resp. to locks. Re-entrant locks are needed for recursive method calls

### 3.1. Local steps

The local reduction steps are given in Table 2. A thread can access and update the heap through the instance fields. At the local level, a configuration is of the form

$$\sigma \vdash e\ , \tag{1}$$

where $\sigma$ is the *heap*. It represents the mutable state of the program and is shared between all threads. It contains the allocated objects and locks. Thus it is a finite mapping from references to objects or locks, of type $Ref \rightarrow Object + Lock$. We write $\bullet$ for the empty heap. An object is basically a record containing the values for the fields and in addition the name of the class it instantiates. We write $C(\vec{v})$ as short-hand for an instance of class $C$ where the fields contain $\vec{v}$ as values. As convention, the formal parameters of the constructor of a class correspond to the fields of the class, and the constructor is used for one purpose only: to give initial values to the fields. When more explicit, we write $[C, f_1 = v_1, \ldots, f_k = v_k]$ or short $[C, \vec{f} = \vec{v}]$ for an instance of class $C$. Also *locks* are allocated on the heap. Each lock has an identity and is either *free*, or *taken* by one particular thread. We use the value 0 to represent that a lock is not held by any thread, and the pair $p(n)$ for $n \geq 1$

to express that a thread $p$ holds the lock $n$ times. This representation captures *re-entrant* locks: Unlike binary locks, a thread holding a lock can acquire the lock again. By counting the lock keeps track how often a given thread has acquired the lock ("re-entering"). This is needed for recursive method calls. The configurations at the global level later contain more than one thread. To distinguish the threads, they will carry a name, with typical elements $p, p', \ldots$ (for "process identifier").

The heap is *well-formed*, written $\sigma \vdash ok$, if no binding occurs more than once, and furthermore, that all (lock or object) references mentioned in the instance states are allocated in $\sigma$: for object references $o$: if $\sigma(o) = C(\vec{v})$, then $v_i \in dom(\sigma)$ for all $v_i$, where $v_i$ is a lock or an object reference. Finally, we require that the values stored in the instance fields conform to the type-restrictions imposed by the class definition. That is, if $\sigma(o) = C(\vec{v})$, then we require for all values $v_i$ that their type corresponds to the type as the corresponding field of $C$. See also Lemma 8 later.

The reduction steps at the local level are of the form

$$\sigma \vdash e \rightarrow \sigma' \vdash e' \tag{2}$$

and specified in Table 2. The two R-COND rules handle the two cases of conditional expressions in the standard manner. Rules R-FIELD and R-ASSIGN capture field access and field update. In both cases, $\sigma(v)$ refers to the heap $\sigma$ to obtain the instance $C(\vec{v})$. The type system will make sure that the value $v$ is an object reference of appropriate type. The premise $C \vdash \vec{f} : \vec{T}$ states that instances of class $C$ have $\vec{f}$ as fields with respective types $\vec{T}$. Looking up the $i$'th field $f_i$ yields the value $v_i$. In the rule for field update, $\sigma[v_1.f_i \mapsto v_2]$ updates the $i$'th field of the object referenced by $v_1$. In our calculus, there are no uninitialized instance fields and all local variables have defined values. Therefore, we do not have a null pointer as value, which means that in the premise of R-ASSIGN we do not need to check whether $v_1$ is different from the null reference or whether $v_1$ is actually defined in $\sigma$. The rule R-CALL for calling a method uses $C.m$ to determine the body of the method $m$ which is denoted by $\lambda\vec{x}.t$. Remember that we do not consider method overloading, the method call evaluates to that method body, with formal parameters $\vec{x}$ substituted by the actual ones, and with `this` replaced by the identity of the callee. Instantiating a new object in rule R-NEW means to procure a new identity $o$ not in use in the heap and extend the heap with the new object $C(\vec{v})$ bound to that reference. In the premise, $\sigma[o \mapsto C(\vec{v})]$ denotes the heap which coincides with $\sigma$ except for the (fresh) reference $o$ whose value is set to object $C(\vec{v})$.

Rule R-RED captures the basic evaluation step, namely substitution. We use the let-construct to unify sequential composition and local variables. So rule R-LET basically expresses associativity of the sequential composition: Ignoring the local variable declarations, it corresponds to a step from $(e_1; e_2); e_3$ to $e_1; (e_2; e_3)$. Note

$$\sigma \vdash \texttt{let}\ x{:}T = (\texttt{if true then}\ e_1\ \texttt{else}\ e_2)\ \texttt{in}\ t \rightarrow \sigma \vdash \texttt{let}\ x{:}T = e_1\ \texttt{in}\ t \qquad \text{R-COND}_1$$

$$\sigma \vdash \texttt{let}\ x{:}T = (\texttt{if false then}\ e_1\ \texttt{else}\ e_2)\ \texttt{in}\ t \rightarrow \sigma \vdash \texttt{let}\ x{:}T = e_2\ \texttt{in}\ t \qquad \text{R-COND}_2$$

$$\frac{\sigma(v) = C(\vec{v})}{\sigma \vdash \texttt{let}\ x{:}T = v.f_i\ \texttt{in}\ t \rightarrow \sigma \vdash \texttt{let}\ x{:}T = v_i\ \texttt{in}\ t}\ \text{R-FIELD}$$

$$\frac{\sigma(v_1) = C(\vec{v}) \qquad \sigma' = \sigma[v_1.f_i \mapsto v_2]}{\sigma \vdash \texttt{let}\ x{:}T = v_1.f_i := v_2\ \texttt{in}\ t \rightarrow \sigma' \vdash \texttt{let}\ x{:}T = v_2\ \texttt{in}\ t}\ \text{R-ASSIGN}$$

$$\frac{\sigma(v) = C(\vec{v}) \qquad \vdash C.m = \lambda\vec{x}.t}{\sigma \vdash \texttt{let}\ x{:}T = v.m(\vec{v})\ \texttt{in}\ t' \rightarrow \sigma \vdash \texttt{let}\ x{:}T = t[\vec{v}/\vec{x}][v/\mathsf{this}]\ \texttt{in}\ t'}\ \text{R-CALL}$$

$$\frac{o \notin dom(\sigma) \qquad \sigma' = \sigma[o \mapsto C(\vec{v})]}{\sigma \vdash \texttt{let}\ x{:}T = \texttt{new}\ C(\vec{v})\ \texttt{in}\ t \rightarrow \sigma' \vdash \texttt{let}\ x{:}T = o\ \texttt{in}\ t}\ \text{R-NEW}$$

$$\sigma \vdash \texttt{let}\ x{:}T = v\ \texttt{in}\ t \rightarrow \sigma \vdash t[v/x] \qquad \text{R-RED}$$

$$\sigma \vdash \texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2 \rightarrow \sigma \vdash \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2) \qquad \text{R-LET}$$

Table 2: Local semantics

that the reduction relation on the thread-local level is deterministic (up-to the identities of the newly created objects)x.

### *3.2. Global steps*

Next we formalize *global* steps, i.e., steps which concern more than one sequential thread or where the thread identity plays a role (i.e., the lock-manipulating steps). A program under execution contains one or more processes running in parallel and each process is responsible for executing one thread. A global configuration consists of the shared heap and a "set" of processes $P$, which contains the "active" part of the program whereas $\sigma$ contains the "passive" data part. A global configuration thus looks as follows

$$\sigma \vdash P\,, \tag{3}$$

where the processes are given by the following grammar:

$$P \quad ::= \quad \mathbf{0} \ \mid\ P \parallel P \ \mid p\langle t\rangle \qquad \text{processes/named threads} \tag{4}$$

$\mathbf{0}$ 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 $\|$-operator is associative and commutative with $\mathbf{0}$ as neutral element. Furthermore, thread identities must be *unique.* That way, $P$ can also be viewed as finite mapping from thread names to expressions. We allow ourselves to write $dom(P)$ ("domain" of $P$) for the set of all names of threads running in $P$. A new thread (with a fresh identifier) is created by the spawn expression. As the language currently does not cover thread communication (such as using a notify-command and similar), the thread identity is not reflected on the user-level (unlike object and lock references). At run-time, however, the identity of a thread $p\langle t\rangle$ plays a role, because it is important which thread holds a lock. With global configurations as given in equation (3), global steps are consequently of the form

$$\sigma \vdash P \Longrightarrow \sigma' \vdash P' \ . \tag{5}$$

The corresponding rules are given in Table 3. Rule R-LIFT lifts the local reduction steps to the global level and R-PAR expresses interleaving of the parallel composition of threads. By writing $P_1 \parallel P_2$ we implicitly require that the $dom(P_1) \cap dom(P_2) = \emptyset$. Spawning a new thread is covered in rule R-SPAWN. The new thread $p'$ is running in parallel with the spawning thread. The identity $p'$ of the new thread is not returned as value to the spawner; in our language it is not needed. Note that the requirement that the domain in a parallel composition are disjoint entails that only globally new identities are created in the steps of a global program.

The next rules deal with lock-handling. Rule R-NEWL creates a new lock (corresponding to an instance of the `ReentrantLock` class in Java 5) 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 (being free or already in possession of the requesting 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 analogous to R-LOCK$_1$ and R-LOCK$_2$. 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.[1] If the lock has value 1, i.e., the thread holds the lock one time, 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

[1]It may worth mentioning that the decription of Java's `Lock` interface does actually not require that only the thread holding a lock is entitled to release it again. All implementations, however, follow that (natural) discipline. We thank the anonymous reviewer for pointing that out.

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(`error` is not a value, we use it as auxiliary thread $t$).

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

$$\frac{p' \neq p}{\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{spawn}\ t'\ \texttt{in}\ t\rangle \Longrightarrow \vdash p\langle \texttt{let}\ x : T = ()\ \texttt{in}\ t\rangle \parallel p'\langle t'\rangle}\ \text{R-SPAWN}$$

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

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

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

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

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

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

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

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

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

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

Table 3: Global semantics

## 4. The type and effect system

We proceed by presenting the type and effect system combining 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 4 (for the thread local level) and Table 5 (for the global level). At the local level, the derivation system deals with expressions (which subsume threads). Judgments of the form

$$\Gamma; \Delta_1 \vdash e : T :: \Delta_2[\&v] \tag{6}$$

are interpreted as follows: Under the type assumptions $\Gamma$, an expression $e$ is of type $T$. The effect part is captured by the effect or lock contexts: With the lock-status $\Delta_1$ before the $e$, the status after $e$ is given by $\Delta_2$.

The typing contexts (or type environments) $\Gamma$ 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, $\Gamma$ 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 $\Gamma$ with the binding $x{:}T$, assuming that $x \notin dom(\Gamma)$. To represent the effects of lock-handling, we use *lock environments* (denoted by $\Delta$). 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. Since we want to assure that the locks are free at thread termination, the number catches the exact lock balance. If interested only in avoiding exceptions due to improper lock release, the system could be relaxed that $n_1$ represents an static lower bound. We use the same notations as for type contexts, i.e., $dom(\Delta)$ for the domain of $\Delta$, further $\Delta(v)$ for looking up the lock status of the lock $v$ in $\Delta$, and $\Delta, v{:}n$ for extending $\Delta$ with a new binding, assuming $v \notin dom(\Delta)$. We write $\bullet$ for the empty context, containing no bindings. A lock context $\Delta$ corresponds to a local view on the heap $\sigma$ in that $\Delta$ contains the status of the locks from the perspective of one thread, whereas the heap $\sigma$ in the global semantics contains the status of the locks from a global perspective. See also Definition 5 of projection later, which connects heaps and lock contexts. The final component of the judgment from Equation 6 is the value $v$ after the &-symbol. If the type $T$ of $e$ is the type `L` for lock-references,

$$\frac{\Delta \vdash v}{\sigma;\Gamma;\Delta \vdash v : \mathtt{L} :: \Delta \& v}\ \textsc{T-Val}_1 \qquad \frac{\Delta \not\vdash x \qquad \Gamma(x) = T}{\sigma;\Gamma;\Delta \vdash x : T :: \Delta}\ \textsc{T-Val}_2 \qquad \frac{\Delta \not\vdash o \qquad \sigma(o) = C(\vec{v})}{\sigma;\Gamma;\Delta \vdash o : C :: \Delta}\ \textsc{T-Val}_3$$

$$\frac{}{\sigma;\Gamma;\Delta \vdash ()\ : \mathtt{Unit} :: \Delta}\ \textsc{T-Unit} \qquad \frac{}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{stop} : T :: \Delta_2}\ \textsc{T-Stop} \qquad \frac{}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{error} : T :: \Delta_2}\ \textsc{T-Error}$$

$$\frac{\sigma;\Gamma \vdash v' : \mathsf{Bool} \quad \sigma;\Gamma;\Delta_1 \vdash e_1 : T :: \Delta_2[\& v] \quad \sigma;\Gamma;\Delta_1 \vdash e_2 : T :: \Delta_2[\& v]}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{if}\ v'\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T :: \Delta_2[\& v]}\ \textsc{T-Cond}$$

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

$$\frac{\sigma;\Gamma;\Delta \vdash v_1 : C :: \Delta \quad \vdash C.f_i : T_i \quad \sigma;\Gamma;\Delta \vdash v_2 : T_i :: \Delta[\& v_2]}{\sigma;\Gamma;\Delta \vdash v_1.f_i := v_2 : T_i :: \Delta[\& v_2]}\ \textsc{T-Assign}$$

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

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

$$\frac{\vdash C : \vec{T} \to C \qquad \sigma;\Gamma \vdash \vec{v} : \vec{T}}{\sigma;\Gamma;\Delta \vdash \mathtt{new}\ C(\vec{v}) : C :: \Delta}\ \textsc{T-New} \qquad \frac{\sigma;\Gamma;\bullet \vdash t : T :: \Delta' \qquad \Delta' \vdash \mathit{free}}{\sigma;\Gamma;\Delta \vdash \mathtt{spawn}\ t : \mathtt{Unit} :: \Delta}\ \textsc{T-Spawn}$$

$$\frac{\sigma;\Gamma,x{:}\mathtt{L};\Delta_1,x{:}0 \vdash t : T :: \Delta_2 \& v}{\sigma;\Gamma;\Delta_1 \vdash \mathtt{let}\ x{:}\mathtt{L} = \mathtt{new\ L\ in}\ t : T :: \Delta_2 \& v}\ \textsc{T-NewL}$$

$$\frac{\Delta \vdash v \qquad \sigma;\Gamma \vdash v : \mathtt{L}}{\sigma;\Gamma;\Delta \vdash v.\,\mathtt{lock} : \mathtt{L} :: \Delta + v \& v}\ \textsc{T-Lock} \qquad \frac{\Delta \vdash v : n+1 \qquad \sigma;\Gamma \vdash v : \mathtt{L}}{\sigma;\Gamma;\Delta \vdash v.\,\mathtt{unlock} : \mathtt{L} :: \Delta - v \& v}\ \textsc{T-Unlock}$$

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

Table 4: Type and effect system (thread-local)

the type and effect system needs information on which variable resp. which lock reference is returned. If $T \neq \mathtt{L}$, 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 $\sigma$. To prove preservation of well-typedness under reduction ("subject reduction") we need to be able to check also the well-typedness of configurations at run-time. Hence we extend the type and effect judgment from Equation 6 to

$$\sigma;\Gamma;\Delta_1 \vdash e : T :: \Delta_2[\&v] \ . \tag{7}$$

In the subject reduction proofs in Section 5, we split the corresponding preservation argument into a part dealing with the types only and one concentrating on the effects. To do so, we use the judgments $\sigma;\Gamma \vdash e : T$ as shorthand for the one of Equation (7) when ignoring the effect part. Similarly, we write $\Delta_1 \vdash e :: \Delta_2[\&v]$ when ignoring the typing part of the judgement.

The rules of Table 4 are mostly straightforward. 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 $\Delta_1$, and the effect of method $m$ is the change from $\Delta_1$ to $\Delta_2$ (see also the rule T-METH for method definitions later, which requires that the domains of $\Delta_1$ and of $\Delta_2$ are equal and correspond to the lock parameters of $m$). 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 the types as stored in $\Gamma$ (for variables) or in $\sigma$ (in case of object references and where the type corresponds to the class, see T-VAL$_3$) and have no effect (cf. the T-VAL-rules). We write $\Delta \vdash v{:}n$ is $v$ has lock balance $n$ in $\Delta$ and $\Delta \vdash v$ if we are not interested in that value (as in rule T-VAL$_1$), i.e., $\Delta \vdash v$ is synonymous to $v \in dom(\Delta)$. The unit value `unit` is of type `Unit` and has no effect. The stop-expression as well as the error-expression have any type and an arbitrary effect (cf. rules T-STOP and T-ERROR), which reflects that the state after the stop or after the error expression is never reached and that the type system formalizes "partial correctness" assertions. 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, as well as the "return value" $v$. 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. Field update (as field look-up) 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. Note that the assignment can update fields containing lock references, i.e., re-directing a field from pointing to one lock to another. By allowing this and especially in the presence of race conditions and interference, the analysis cannot track the exact lock balance of shared fields therefore. The analysis is nonetheless sound, as the rule T-FIELD starts the thread-local analysis of the corresponding local variable with a count 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 $\Delta_2$ of $e$ is used in the pre-condition when checking $t$, and the post-condition $\Delta_3$ 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 $\Delta_2$. 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 $\Delta_1'$ 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 $\Delta_1'$ of the method definition as the *required* lock balances and $\Delta_1$ the *provided* ones at the control point before the call. For the post-conditions, $\Delta_2'$ can be seen as the promised post-condition and $\Delta_2$ 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 $\Delta_1'$ 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 $\Delta_1$ before the call. In the premises, we formalize those checks and calculations as follows:

**Definition 1.** *Assume two lock environments $\Delta_1$ and $\Delta_2$. 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 $\Delta_2 \vdash v : n_2$. Given $dom(\Delta_1) = dom(\Delta_2)$, 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 $\Delta'$, where $\Delta'(v) = 1$ if $\Delta(v) = 0$, and $\Delta'(v) = n+1$, if $\Delta(v) = n$. $\Delta - v$ is defined analogously.*

The type system assures that that the lock balances are always non-negative. In particular, the substraction $\Delta - v$ never leads to negative balances. This is assured by corresponding premises of the typing rules T-CALL and T-UNLOCK. For the effect part of method specifications $C.m :: \Delta_1 \to \Delta_2$, the lock environments $\Delta_1$ and $\Delta_2$ represent the pre- and post-conditions for the lock parameters and hence $dom(\Delta_1) = dom(\Delta_2)$. As for the method specications, however, the difference of $\Delta_1 - \Delta_2$, where $\Delta_1$ is the pre-condition and $\Delta_2$ the post-condition, may be negative. 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 2** (Method parameters and aliasing)**.** . *Consider the following code:*

Listing 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 $\Delta_1$ should state that the lock stored in $x_1$ should have at least value* 1*, and the same for $x_2$, i.e.,*

$$\Delta_1 = x_1{:}1, x_2{:}1 \tag{8}$$

*With $\Delta_1$ as pre-condition, the effect type system accepts the method of Listing 1 as type correct, because the effects on $x_1$ and $x_2$ are checked* individually. *Assume that at run-time, the actual parameters, say $l_1$ and $l_2$ happen to be* not aliases *in a call $o.m(l_1, l_2)$ with $l_1 \neq l_2$, and each of them satisfies the precondition of Equation 8* individually, *i.e., at run-time, the lock environment $\Delta_1' = \Delta_1[l_1/x_1][l_2/x_2]$ i.e.,*

$$\Delta_1' = l_1{:}1, l_2{:}1 \ . \tag{9}$$

*Now 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 $\Delta_1$ 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 3** (Substitution for lock environments)**.** *Given a lock environment $\Delta$ of the form $\Delta = v_1{:}n_1, \ldots, v_k{:}n_k$, with $k \geq 0$, and all the natural numbers $n_i \geq 0$. Remember that each value $v_i$ is either a variable or a lock reference and all the $v_i$'s are assumed to be different and that we assume the order of the bindings $v_i{:}n_i$ to be irrelevant. The result of the* substitution *of a variable $x$ by a value $v$ in $\Delta$ 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, $\Delta' = \Delta$.*

We apply substitution "point-wise" also to judgments, i.e., writing $(\sigma;\Gamma;\Delta_1 \vdash t : T :: \Delta_2[\&v'])[v/x]$ is understood as $\sigma;\Gamma[v/x];\Delta[v/x] \vdash t[v/x] : T :: \Delta[v/x][\&v'[v/x]]$. Note that $\sigma$ is un-affected by the substitution, and furthermore, in abuse of notation, the substitition on $\Gamma$, $t$, and $v'$ is interpreted as "standard" substitution, i.e., the replacement of variable $x$ by $v$. For the lock environments $\Delta_1$ and $\Delta_2$, the substitution is given by Definition 3.

**Example 4** (Aliasing)**.** *The example continues from Example 2, i.e., we are given the method definition of Listing 1. Listing 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 local variables $x_1$ and $x_2$) has a balance of* 1*, as required in $m$'s precondition, and the method body individually unlocks each of them once. As a note: in the code snippets, we do not use the let-construct for defining the value of a local variable, but use more conventional syntax with the silent understanding that the variable's scope extends till the end of the shown expression.*

Listing 2: Method call, no aliasing

```
x1 := new L;
x2 := new L;          // x1 and x2: no aliases
x1.lock; x2.lock;
o.m(x1,x2);
```

*As explained earlier, nothing is wrong with aliasing as such. If we change the code of the call site by making $x_1$ and $x_2$ aliases, the code could look as follows:*

Listing 3: Method call, aliasing

```
x1 := new L;
x2 := x1;     // x1 and x2: aliases
x1.lock; x2.lock;
o.m(x1,x2);
```

*Again, there is* no *run-time error, because after executing* $x_1$`.lock` *and* $x_2$`.lock`*, the actual balance of the single lock stored in* $x_1$ *as well as in* $x_2$ *is* 2*, which means, the two unlocking operations in the body of m cause no lock error. We will show the corresponding type and effect derivation later in Example 6, after explaining the corresponding rules.* □

Back to the rules of Table 4. The identity of a new thread is irrelevant, i.e., spawning a new thread carries type `Unit`, and a freshly instantiated object carries the class it instantiates as type (cf. T-SPAWN and T-NEW). 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 $\bullet$.[2] The reason is that the new thread starts without holding any lock (cf. R-SPAWN of the semantics in Table 3). As an aside: this is one difference of the effect system formalized here for lock handling from the one dealing with transactions in [20]. A new thread here does not inherit the locks of its spawning thread, whereas in the transactional setting with multi-threaded and nested transactions, the new thread starts its life "inside" the transactions of its spawner. Note further that the premise of T-SPAWN requires 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 $\Delta' \vdash \mathit{free}$. Formally, $\Delta' \vdash \mathit{free}$ is defined as follows: if $\Delta \vdash v$ then $\Delta \vdash v{:}0$. Typing for new locks is covered by T-NEWL. Giving back the fresh identity of the lock, the expression is typed by the type of locks `L`. As for the effect, the pre-context $\Delta_1$ is extended by a binding for the new lock initially assumed to be free, i.e., the new binding is $x{:}0$. The last three rules cover handling of an existing lock. The two operations for acquiring and releasing a lock, `lock` and `unlock`, 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). If the lock is free (i.e., $v{:}0$), the

[2] We overload the symbol $\bullet$ to represent empty type contexts as well as empty lock contexts and also the empty heap.

counter is increased to $v{:}1$ afterwards. If the lock is taken (i.e., $v{:}n$) by the current thread, the lock counter is increased to $v{:}n+1$. We abbreviate that counting up the lock status for a lock $v$ (assuming $\Delta \vdash v$) by $\Delta + v$ in the premise of T-LOCK. 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 $\Delta_1$ 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 4 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 4.

The effect system at the local level uses lock environments to approximate the effect of the expression on the locks (cf. Equation 7). Lock environments $\Delta$ 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 $\sigma$, 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 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 5).

**Definition 5** (Projection)**.** *Assume a heap $\sigma$ with $\sigma \vdash ok$ and a thread $p$. The* projection *of $\sigma$ onto $p$, written $\sigma \downarrow_p$ is inductively defined as follows:*

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

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. Note, however, that the projection is needed in the type and effect analysis, not in the semantics. In the reduction relation when dealing with lock handling we can obviously *not* have a thread-local view on the lock; after all, locks are *meant* to be shared to coordinate the behavior of different threads. In contrast, for the effect system, the local perspective is possible, i.e., it is possible to work with the above definition of projection. 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 thread projected to can actually take the lock, it makes a statement about what happens after the thread has taken the lock. If the local thread can take the lock, the lock must have been 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 into account the effect of other threads. 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 5, formalizing judgments of the form

$$\sigma \vdash P : ok\ , \tag{10}$$

where $P$ is given as in Equation 4.

$$\frac{}{\sigma \vdash \mathbf{0} : ok}\ \text{T-EMPTY} \qquad \frac{\sigma \vdash P_1 : ok \qquad \sigma \vdash P_2 : ok}{\sigma \vdash P_1 \parallel P_2 : ok}\ \text{T-PAR} \qquad \frac{\forall i.\ \vdash M_i : ok}{\vdash C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T};\vec{M}\} : ok}\ \text{T-CLASS}$$

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

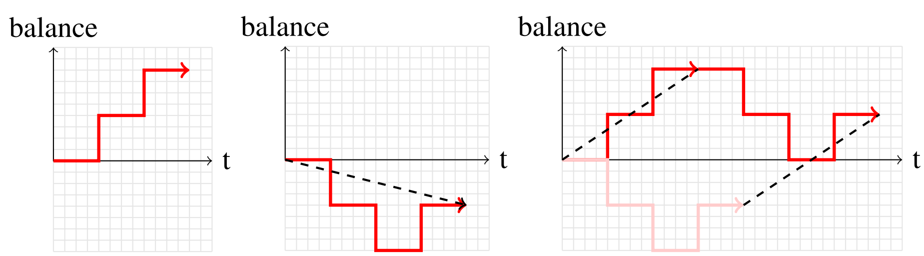
$$\frac{\begin{array}{c}\vdash C.m : \vec{T} \to T :: \Delta_1 \to \Delta_2 \qquad \bullet;\vec{x}{:}\vec{T},\mathsf{this}{:}C;\Delta_1 \vdash t : T :: \Delta_2,\Delta_2' \\ \Delta_2' \vdash \mathit{free} \qquad \mathit{dom}(\Delta_1) = \mathit{dom}(\Delta_2) = \mathit{locks}(\vec{x}{:}\vec{T}) \qquad T \neq \mathtt{L}\end{array}}{\vdash C.m(\vec{x}{:}\vec{T})\{t\} : ok}\ \text{T-METH}$$

Table 5: Type and effect system (global)

In the rules, we assume that $\sigma$ 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 $\Delta_1$ for that check, the projection of the current heap $\sigma$ is taken. The code $t$ must be well-typed, i.e., carry some type $T$. As for the post-condition $\Delta_2$, we require that the thread has given back all the locks, postulated by $\Delta_2 \vdash \mathit{free}$. 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. The rules that the domains of $\Delta_1$ and $\Delta_2$ correspond exactly to the lock parameters of the method, where $locks(\vec{x}{:}\vec{T})$ is the set of formal parameters of the method of lock-type. This is expressed using the function *locks* which extract from the formal parameters those dealing with locks. 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 $\Delta_2, \Delta_2'$ of the body, $\Delta_2'$ contains local lock variables *other* than the formal lock parameters (which are covered by $\Delta_2$). The premise $\Delta_2' \vdash free$ requires that the lock counters of $\Delta_2'$ 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 1. Assume two methods $m$ and $n$, where $m$ calls $n$, i.e., $m$ is of the form

$$m()\{\ldots; x.n()\ldots\}\ .$$

Let us assume the methods operate on one single lock, whose behavior is illustrated by the first two sub-figures of Figure 1. The history in Figure 1(a) is supposed to represent the lock behavior $m$ up to the point where method $n$ is called, and Figure 1(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 $\Delta_1$ 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




(a) Method $m$ (b) Method $n$ (c) Method $n$

Figure 1: Lock balance of methods $m$ and $n$

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 3. 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 1(c) for combining the two histories of Figures 1(a) and 1(b). Finally, a class definition $\mathsf{class}\ C(\vec{f}{:}\vec{T})\{\vec{f}{:}\vec{T};\vec{M}\}$ 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).

**Example 6** (Aliasing)**.** *Revisiting Example 4 and the code of Listing 3, an analysis of the corresponding expression gives rise to the following derivation*

$$\dfrac{\dfrac{\dfrac{\Delta_1 \vdash x_1}{\Delta_1 \vdash x_1 : \mathtt{L} :: \Delta_1 \& x_1} \qquad \dfrac{(12)}{(\Delta_2 \vdash x_1.\,\mathtt{lock}; x_2.\,\mathtt{lock}; o.m(x_1,x_2) :: \Delta_0, x_1{:}0, x_2{:}0)([x_1/x_2])}}{\Delta_1 \vdash \mathtt{let}\ x_2{:}L = x_1\ \mathtt{in}\ x_1.\,\mathtt{lock}; x_2.\,\mathtt{lock}; o.m(x_1,x_2) :: \Delta_0, x_1{:}0}\ \text{T-LET}}{\Delta_0 \vdash \mathtt{let}\ x_1{:}L = \mathtt{new}\ L\ \mathtt{in}\ \mathtt{let}\ x_2{:}L = x_1\ \mathtt{in}\ x_1.\,\mathtt{lock}; x_2.\,\mathtt{lock}; o.m(x_1,x_2) : \Delta_0, x_1{:}0}\ \text{T-NEWL} \tag{11}$$

*where $\Delta_1 = \Delta_0, x_1{:}0$ and $\Delta_2 = \Delta_1, x_2{:}0$. In the derivation, we concentrate on the effect part, omitting the (conventional) part for typing. In particular, we leave out $\sigma$ and $\Gamma$ from the judgment. We assume $C$ to be the type/class of object $o$, i.e., $\sigma(o) = C(\vec{v})$ for some $\vec{v}$. In the right-premise of the instance of the let-rule, the judgment $(\Delta_1, x_2{:}0 \vdash x_1.\,\mathtt{lock}; x_2.\,\mathtt{lock}; o.m(x_1,x_2))([x_1/x_2])$ corresponds to $\Delta_0, x_1{:}0 \vdash x_1.\,\mathtt{lock}; x_1.\,\mathtt{lock}; o.m(x_1,x_1)$ after the substitution. From this sub-goal, the derivation continues as follows:*

$$\dfrac{\Delta_1 \vdash x_1.\,\mathtt{lock} :: \Delta_0, x_1{:}1 \qquad \dfrac{\Delta_0, x_1{:}1 \vdash x_1.\,\mathtt{lock} :: \Delta_0, x_1{:}2 \qquad \dfrac{\begin{array}{c}\vdash C.m : \mathtt{L} \times \mathtt{L} \to T :: \Delta_1' \to \Delta_2' \\ \Delta_0, x_1{:}2 \geq \Delta_1'[x_1/x_1'][x_1/x_2']\end{array}}{\Delta_0, x_1{:}2 \vdash o.m(x_1,x_1) :: \Delta_0, x_1{:}0}\ \text{T-CALL}}{\Delta_0, x_1{:}1 \vdash x_1.\,\mathtt{lock}; o.m(x_1,x_1) :: \Delta_0, x_1{:}0}}{\Delta_1 \vdash x_1.\,\mathtt{lock}; x_1.\,\mathtt{lock}; o.m(x_1,x_1) :: \Delta_0, x_1{:}0} \tag{12}$$

*The specification of method $m$ (from some class $C$) from Listing 1 is $\vdash C.m : \mathtt{L} \times \mathtt{L} \to T :: \Delta_1' \to \Delta_2'$ with $\Delta_1' = x_1'{:}1, x_2'{:}1$ and $\Delta_2' = x_1'{:}0, x_2'{:}0$. Furthermore, the following equalities hold:*

$$\begin{array}{rcl}\Delta_1'[x_1/x_1'][x_1/x_2'] & = & x_1{:}2 \\ (\Delta_2' - \Delta_1') & = & x_1'{:}(-1), x_2'{:}(-1) \\ (\Delta_2' - \Delta_1')[x_1/x_1'][x_1/x_2'] & = & x_1{:}(-2) \\ (\Delta_0, x_1{:}2) + (\Delta_2' - \Delta_1')[x_1/x_1'][x_1/x_2'] & = & x_1{:}0\,.\end{array}$$

*Note also that if we changed the example by replacing the second locking statement of Listing 3 from* $x_2$`.lock` *to* $x_1$`.lock`*, the analysis would reject the program, even if at run-time, the program would not show any error.* □

## 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 proof proceeds in two parts: one dealing with the typing part first, and afterwards dealing with the effect part. Both parts are further split into the treatment of local transitions and the one for global transitions (Lemmas 10 and 11 for the typing and 16 and 17 for the effects). With preservation of well-typedness under reduction proven, the desired results is straightforward: "well-typed programs don't go wrong" in our case, no thread releases a lock it does not own nor will there be hanging locks at thread termination (Theorems 19 and 20).

The first two lemmas deal with aspects of type preservation during local evaluation. Lemma 7 shows preservation of typing under substitution. Lemma 8 shows preservation of typing when updating the heap, i.e., replacing the value of a field of an object (in a type-consistent manner). Both lemmas are needed for subject reduction afterwards.

**Lemma 7** (Substitution)**.** *If* $\sigma;\Gamma,x{:}T_2 \vdash e_1 : T_1$ *and* $\sigma;\Gamma \vdash e_2 : T_2$*, then* $\sigma;\Gamma \vdash e_1[e_2/x] : T_1$.

*Proof.* Straightforward. □

**Lemma 8.** *Let* $\sigma \vdash ok$*.*

1. *Assume* $\sigma;\Gamma \vdash e : T$ *and further* $\sigma;\Gamma \vdash r : C$ *with* $C \vdash f_i : T_i$ *and assume a value* $v$ *of the same type as field* $f_i$*, i.e.,* $\sigma;\Gamma \vdash v : T_i$*. Let* $\sigma' = \sigma[r.f_i \mapsto v]$*. Then:*
   (a) $\sigma' \vdash ok$*.*
   (b) $\sigma';\Gamma \vdash e : T$*.*
2. *Let* $\sigma' = \sigma[r \mapsto C(\vec{v})]$ *where* $r \notin dom(\sigma)$*. Assume* $\vdash \mathsf{class}\, C(\vec{f}{:}\vec{T})\{\vec{f} : \vec{T};\vec{M}\} : ok$ *and* $\vdash C : \vec{T} \to C$ *and furthermore* $\sigma;\Gamma \vdash \vec{v} : \vec{T}$*. Then* $\sigma' \vdash ok$*.*

*Proof.* Straightforward. □

**Lemma 9** (Weakening)**.** *If* $\sigma;\Gamma \vdash e : T$*, then* $\sigma;\Gamma,x'{:}T' \vdash e : T$*.*

*Proof.* Generalize the weakening property slightly to: If $\sigma;\Gamma_1,\Gamma_2 \vdash e{:}T$, then $\sigma;\Gamma_1,x'{:}T',\Gamma_2 \vdash e : T$. Proceed by induction on the typing derivation. Most cases are immediate or by straightforward induction. In particular, the base case T-VAL$_2$ for $\sigma;\Gamma_1,\Gamma_2 \vdash y : T'$ is immediate, observing that $\sigma;\Gamma_1,\Gamma_2 \vdash y : T'$ implies $\sigma;\Gamma_1,\Gamma_2 \vdash y$ by the premise of the rule, which further implies $x' \neq y$. We show only the case for the let-construct.

*Case:* T-LET: $\sigma;\Gamma_1;\Gamma_2 \vdash \mathtt{let}\, x{:}T_1 = e \,\mathtt{in}\, t : T_2$
where $\sigma;\Gamma_1,\Gamma_2 \vdash e : T_1$ and $(\sigma;\Gamma_1,\Gamma_2,x{:}T_1 \vdash t : T_2)[v'/x]$, where $v'$ is (optionally) the value in which $e$ gives back its result, in case $T_1 = \mathtt{L}$. If $T_1 \neq \mathtt{L}$, the substitution is empty. By induction, we get $\sigma;\Gamma_1,x'{:}T',\Gamma_2,x{:}T_1 \vdash t : T_2$, which implies the result with T-LET. In case $T_1 = \mathtt{L}$, the second premise $(\sigma;\Gamma_1,\Gamma_2,x{:}T_1 \vdash t : T_2)[v'/x]$ equals $\sigma;\Gamma_1[v'/x],\Gamma_2[v'/x] \vdash t[v'/x] : T_2$. By induction, this implies $\sigma;\Gamma_1[v'/x],x'{:}T',\Gamma_2[v'/x] \vdash t[v'/x] : T_2$, which is the same as $(\sigma;\Gamma_1,x'{:}T',\Gamma_2,x : \mathtt{L} \vdash t : T_2)[v'/x]$, and thus with T-LET

$$\frac{\sigma;\Gamma_1,x'{:}T',\Gamma_2 \vdash e : \mathtt{L} \qquad (\sigma;\Gamma_1,x'{:}T',\Gamma_2,x{:}\mathtt{L} \vdash t : T_2)[v'/x]}{\sigma;\Gamma_1,x'{:}T',\Gamma_2 \vdash \mathtt{let}\, x{:}\, \mathtt{L} = e \,\mathtt{in}\, t : T_2}$$

as required.

The remaining cases are by straightforward induction. □

**Lemma 10** (Subject reduction (local))**.** *Assume $\sigma;\Gamma \vdash e : T$ and $\sigma \vdash$ok. If $\sigma \vdash e \rightarrow \sigma' \vdash e'$, then $\sigma' \vdash$ok and $\sigma';\Gamma \vdash e' : T$.*

*Proof.* The proof proceeds by straightforward induction on the rules of Table 2. In the cases for R-COND$_i$, for fields look-up, for field update, method calls, and for instantiating a new object, the reduction rule is of the general form $\sigma \vdash \mathtt{let}\, x{:}T = e \,\mathtt{in}\, t \rightarrow \sigma' \vdash \mathtt{let}\, x{:}T = e' \,\mathtt{in}\, t$. By the well-typedness assumption $\sigma \vdash \mathtt{let}\, x{:}T = e \,\mathtt{in}\, t : T'$ we obtain by inverting rule T-LET that

$$\sigma;\Gamma \vdash e : T \tag{13}$$

and $\sigma;\Gamma \vdash t : T'$, where $\Gamma = x{:}T$. It suffices to show that for $e'$ after the step, $\sigma;\Gamma \vdash e' : T$ in all the mentioned cases. Whence the result follows by T-LET.

*Case:* R-COND$_1$: $\sigma \vdash \mathtt{let}\, x{:}T = (\mathtt{if\ true\ then}\, e_1 \,\mathtt{else}\, e_2) \,\mathtt{in}\ t \rightarrow \sigma \vdash \mathtt{let}\, x{:}T = e_1 \,\mathtt{in}\, t$

Assumption (13) means $\sigma \vdash \mathtt{if\ true\ then}\, e_1 \,\mathtt{else}\, e_2 : T$, which gives by inverting rule T-COND that $\sigma;\Gamma \vdash e_1 : T$, i.e., $e_1 = e'$. Furthermore, the steps is side-effect free, i.e., $\sigma$ does not change, from which well-formedness of the heap after the step follows. The case for R-COND$_2$ works symmetrically.

*Case:* R-FIELD: $\sigma \vdash \texttt{let}\, x{:}T = v.f_i \,\texttt{in}\, t \to \sigma \vdash \texttt{let}\, x{:}T = v_i \,\texttt{in}\, t$, where $\sigma(v) = C(\vec{v})$ and $C \vdash f_i : T_i$ by the premises of that rule. The assumption $\sigma \vdash ok$ implies that $\sigma; \bullet \vdash v_i : T_i$, and hence by weakening (Lemma 9) $\sigma;\Gamma \vdash v_i : T_i$, as required. Well-formedness of the heap after the step is trivial as $\sigma$ is unchanged in the step.

*Case:* R-ASSIGN: $\sigma \vdash \texttt{let}\, x{:}T = (v'.f_i := v_2) \,\texttt{in}\, t \to \sigma' \vdash \texttt{let}\, x{:}T = v_2 \,\texttt{in}\, t$, where $\sigma' = \sigma[v_1.f_i \mapsto v_2]$. From the well-typedness assumption $\sigma;\Gamma \vdash v_1.f_i := v_2 : T_i$, we get by inverting rule T-ASSIGN $\sigma;\Gamma \vdash v_1 : C$ where $C \vdash f_i : T_i$, and furthermore $\sigma;\Gamma \vdash v_2 : T_i$. The heap $\sigma'$ after the step is of the form $\sigma' = \sigma[v_1.f_i \mapsto v_2]$. Since the field $f_i$ and the value $v_2$ are of the same type, $\sigma \vdash ok$ implies $\sigma' \vdash ok$ (Lemma 8(1a)). Furthermore $\sigma;\Gamma \vdash v_2 : T_i$ implies $\sigma';\Gamma \vdash v_2 : T_i$ (cf. Lemma 8(1b)), as required.

*Case:* R-CALL: $\sigma \vdash \texttt{let}\, x{:}T = v.m(\vec{v}) \,\texttt{in}\, t' \to \sigma \vdash \texttt{let}\, x{:}T = t[\vec{v}/\vec{x}][v/\mathsf{this}] \,\texttt{in}\, t'$
By looking up the class table, we get the method body $\vdash C.m = \lambda\vec{x}.t$. From the well-typedness assumption $\sigma;\Gamma \vdash v.m(\vec{v}) : T$ and by inverting rule T-CALL we get the types for the arguments $\sigma;\Gamma \vdash \vec{v} : \vec{T}$, for the callee $\sigma;\Gamma \vdash v : C$, and for the called method as declared in the class $\vdash C.m : \vec{T} \to T$. Preservation of typing under substitution from Lemma 7 gives $\sigma;\Gamma \vdash t[\vec{v}/\vec{x}][v/\mathsf{this}] : T$ as required. The heap $\sigma$ is unchanged in the step and hence still well-formed afterwards. The result follows by T-LET.

*Case:* R-NEW: $\sigma \vdash \texttt{let}\, x{:}T = \texttt{new}\, C(\vec{v}) \,\texttt{in}\, t \to \sigma' \vdash \texttt{let}\, x{:}T = o \,\texttt{in}\, t$, where $\sigma'$ extends $\sigma$ by allocating the new instance $C(\vec{v})$, i.e, $\sigma' = \sigma[o \mapsto C(\vec{v})]$. The assumption of well-typedness $\sigma;\Gamma \vdash \texttt{new}\, C : C$ before the step gives (by inverting rule T-NEW) as type for the constructor method $\vdash C : \vec{T} \to C$, i.e., $\vec{T}$ are the types of the constructor arguments (and thus the fields), and furthermore, $\sigma,\Gamma \vdash \vec{v} : \vec{T}$. Well-typedness after the step, i.e., $\sigma';\Gamma \vdash o : C$, follows by rule T-VAL$_3$ and since $\sigma'(o) = C$. As for well-formedness of $\sigma'$: the object reference $o$ is fresh, which implies that well-formedness is preserved in the step (cf. Lemma 8(2)).

*Case:* R-RED: $\sigma \vdash \texttt{let}\, x{:}T' = v \,\texttt{in}\, e \to \sigma \vdash e[v/x]$
The well-typedness assumption $\sigma;\Gamma \vdash \texttt{let}\, x{:}T' = v \,\texttt{in}\, e : T$ implies $\sigma;\Gamma, x{:}T' \vdash e : T$, and the result follows by preservation of typing under substitution (Lemma 7).

*Case:* R-LET: $\sigma \vdash \texttt{let}\, x_2{:}T_2 = (\texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, e) \,\texttt{in}\, t \to \sigma \vdash \texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, (\texttt{let}\, x_2{:}T_2 = e \,\texttt{in}\, t)$
By induction, using rule T-LET. □

Next we prove subject reduction also for global configurations.

**Lemma 11** (Subject reduction (global)). *If $\sigma \vdash P : ok$ and $\sigma \vdash P \Longrightarrow' \vdash P'$ where the reduction step is given* not *by one of the two* R-ERROR*-rules, then $\sigma' \vdash P' : ok$.*

*Proof.* By induction on the reduction rules of Table 3. The two R-ERROR-rules are excluded by assumption.

*Case:* R-LIFT: $\sigma \vdash p\langle t\rangle \Longrightarrow' \vdash p\langle t'\rangle$,
with $\sigma \vdash t \to \sigma' \vdash t'$. Remember that a thread $t$ is an expression $e$ as well in the grammar of Table 1. The assumption $\sigma \vdash p\langle t\rangle$ :*ok* implies by inverting rule T-THREAD $\sigma;\bullet \vdash t : T$, for some type $T$. Subject reduction on the local level (Lemma 10) gives $\sigma';\bullet \vdash t' : T$ and the result $\sigma' \vdash p\langle t'\rangle : ok$ follows by T-THREAD.[3]

*Case:* R-PAR: $\sigma \vdash P_1 \parallel P_2 \Longrightarrow' \vdash P_1' \parallel P_2$.
By straightforward induction.

*Case:* R-SPAWN: $\sigma \vdash p\langle \mathtt{let}\, x{:}T' = \mathtt{spawn}\, t' \;\mathtt{in}\; t\rangle \Longrightarrow \vdash p\langle \mathtt{let}\, x{:}T' = () \;\mathtt{in}\; t\rangle \parallel p'\langle t'\rangle$
The well-typedness assumption for the configuration before the step, inverting rules T-THREAD, gives $\sigma;\bullet \vdash \mathtt{let}\, x{:}T' = \mathtt{spawn}\, t' \;\mathtt{in}\; t : T$ for some type $T$ and further by inverting T-LET and using T-SPAWN $\sigma;\bullet \vdash \mathtt{spawn}\, t' : \mathtt{Unit}$ (i.e., $T' = \mathtt{Unit}$) and $\sigma;x{:}\,\mathtt{Unit} \vdash t : T$, and still further by inverting T-SPAWN $\sigma;\bullet \vdash t' : T''$ (for some type $T''$). The result then follows with the help of T-PAR, T-THREAD, T-LET, and T-UNIT. The case for T-NEWL works similarly.

*Case:* R-LOCK$_1$: $\sigma \vdash p\langle \mathtt{let}\, x{:}T = l.\,\mathtt{lock}\;\mathtt{in}\; t\rangle \to \sigma' \vdash p\langle \mathtt{let}\, x{:}T = l \;\mathtt{in}\; t\rangle$
The case works similar as the previous ones, observing that both $l.\,\mathtt{lock}$ and $l$ are of type $L$ by rule T-LOCK. Rule R-LOCK$_2$ and the R-UNLOCK-rules work similarly.

*Case:* R-TRYLOCK$_1$: $\sigma \vdash p\langle \mathtt{let}\, x{:}T = \mathtt{if}\;\; l.\,\mathtt{trylock}\;\mathtt{then}\; e_1 \;\; \mathtt{else}\; e_2 \;\mathtt{in}\; t\rangle \Longrightarrow' \vdash p\langle \mathtt{let}\, x{:}T = e_1 \;\mathtt{in}\; t\rangle$
The well-typedness assumption for the configuration before the step implies that the trylock sub-expression is well-typed as well, i.e., $\sigma;\Gamma \vdash \mathtt{if}\; l.\,\mathtt{trylock}\;\mathtt{then}\; e_1 \;\mathtt{else}\; e_2 : T$, which in turn implies by the typing premises of rule T-TRYLOCK $\sigma;\bullet \vdash e_1 : T$, from which the result follows, using Lemma 8(1b). The cases for R-TRYLOCK$_2$ and R-TRYLOCK$_3$ work similarly. □

Next we prove subject reduction for the effect part of the system of Tables 4 and 5.

**Lemma 12** (Substitution and ordering)**.** *Assume $v \in dom(\Delta_1)$, $x \in dom(\Delta_1')$ and $x \notin dom(\Delta_1)$ and $y \notin dom(\Delta_1')$. If $\Delta_1 \geq \Delta_1'[v/x]$, then $\Delta_1[l/y] \geq \Delta_1'[v/x][l/y] = \Delta_1'[v[l/y]/x]$.*

---

[3]The other two premises of T-THREAD requiring that the thread has not reached an error state after the step and that the locks are all free in the post-condition are not part of subject reduction as far as the *typing* is concerned.

*Proof.* We start by proving the inequation $\Delta_1[l/y] \geq \Delta_1'[v/x][l/y]$. By definition of $\leq$ on lock contexts, $dom(\Delta_1) \supseteq dom(\Delta_1'[v/x])$, and for all bindings in $\Delta_1'[v/x]$, the corresponding lock counter value is larger or equal than the corresponding value in $\Delta_1$.

If $y \notin dom(\Delta_1)$, the result is immediate, since also $y \notin dom(\Delta_1'[v/x])$. Assume otherwise that $y \in dom(\Delta_1)$ but $y \notin dom(\Delta_1'[v/x])$, If $l \in dom(\Delta_1)$ the result follows from the fact that $dom(\Delta_1[l/y]) \subseteq dom(\Delta_1)$ and that the lock balance of $y$ in $\Delta_1$ is non-negative, i.e., $\Delta_1 \vdash y{:}n$ means $n \geq 0$. If $l \notin dom(\Delta_1)$, the result is immediate. If finally $y \in dom(\Delta_1)$ and $y \in dom(\Delta_1'[v/x])$, the result is immediate again by the definition of substitution.

For the equality $\Delta_1'[v/x][l/y] = \Delta_1'[v[l/y]/x]$, first observe that $\Delta_1'[v/x][l/y] = \Delta_1'[l/y][v[l/y]/x]$, and the result follows from the assumption that $y \notin dom(\Delta_1')$. $\square$

**Lemma 13** (Substitution)**.** *If $FE(\Delta_1, \Delta_2, v)$, then $FE(\Delta_1[l/x], \Delta_2[l/x], v[l/x])$.*

*Proof.* By definition $\Delta_2 = \Delta_1', \vec{v}{:}\vec{0}, v{:}n$ for some $\Delta_1'$, where we have to distinguish the following two cases:

*Case: $dom(\Delta_1) = dom(\Delta_1')$*
If $x \in dom(\Delta_1) = dom(\Delta_1')$, the result is immediate. If $x \in \vec{v}$, we distinguish further, whether $l \in dom(\Delta_1)$ or not. If not, $\Delta_2[l/x] \vdash l{:}0$, as required. If $l \in dom(\Delta_1)$, then also $l \in dom(\Delta_1')$ and hence it is not a new value in $\Delta_2[l/x]$ compared to $\Delta_1[l/x]$, hence again $FE(\Delta_2[l/x], \Delta_1[l/x], v)$. Finally, for $x = v$: the case where $x = v \in dom(\Delta_1)$ is covered above. If $x \notin dom(\Delta_1)$, the case is immediate observing that the $x$, specifying which value in $\Delta_2$ need *not* to be zero in $FE(\Delta_1, \Delta_2, x)$ is replaced by $l$ in $FE(\Delta_1[l/x], \Delta_2[l/x], l)$.

*Case: $dom(\Delta_1', v{:}n) = dom(\Delta_1)$*
Similarly. $\square$

**Lemma 14** (Substitution)**.** *Let $x$ be a variable of type* L *and $l$ be a lock reference. Let further be $x$ different from all formal parameters of all methods in the program. If $\Delta_1 \vdash t :: \Delta_2\&v$, then $\Delta_1[l/x] \vdash t[l/x] :: \Delta_2[l/x]\&v[l/x]$.*

*Proof.* We are given $\Delta_1 \vdash t :: \Delta_2\&v$. Proceed by induction on the typing derivation.

*Case:* T-VAL$_1$: $\Delta \vdash v :: \Delta\&v$
where $\Delta \vdash v$. In case, the value $v = x$, we have $\Delta[l/x] \vdash l$ by Definition 3, and thus $\Delta[l/x] \vdash l :: \Delta[l/x]\&l$ by T-VAL$_1$. If $v = y$ where $y \neq x$ or $v = l'$ for some lock reference $l'$, the assumption $\Delta \vdash v$ implies also $\Delta[l/x] \vdash v$, and the case follows by T-VAL$_1$ again.

The cases for T-VAL$_2$ and for T-VAL$_3$, i.e., for values different from lock references or corresponding variables are straightforward. Likewise the cases for T-UNIT, T-STOP, and T-ERROR.

*Case:* T-COND: $\Delta_1 \vdash \texttt{if}\ v'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: \Delta_2[\&v]$ .
As for the value $v'$: the type system assures $v'$ to be of boolean type. Hence, $v' \neq x$ and $v' \neq l$, and $v'$ is thus unaffected by the substitution. By the premises of the rule we further have $\Delta_1 \vdash e_1 :: \Delta_2[\&v]$ and $\Delta_1 \vdash e_2 :: \Delta_2[\&v]$ by sub-derivations. Thus by induction and with rule T-COND, we get

$$\frac{(\Delta_1 \vdash e_1 :: \Delta_2[\&v])[l/x] \qquad (\Delta_1 \vdash e_2 :: \Delta_2[\&v])[l/x]}{(\Delta_1 \vdash \texttt{if}\ v'\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: \Delta_2[\&v])[l/x]}$$

which concludes the case.

*Case:* T-FIELD: $\Delta_1 \vdash \texttt{let}\ x'{:}\mathrm{L} = v'.f_i\ \texttt{in}\ t :: \Delta_2\&v''$
with the premise $\Delta_1, x'{:}0 \vdash t :: \Delta_2\&v''$. Induction yields $(\Delta_1, x'{:}0 \vdash t :: \Delta_2\&v'')[l/x]$. Observing that $x \neq x'$ since $x'$ is a local variable, the case follows with T-FIELD.

*Case:* T-ASSIGN: $\Delta \vdash v_1.f_i := v_2 :: \Delta[\&v_2]$
with premises $\Delta \vdash v_1 :: \Delta$ and $\Delta \vdash v_2 :: \Delta[\&v_2]$. The case follows by induction and T-ASSIGN.

*Case:* T-LET: $\Delta_1 \vdash \texttt{let}\ x'{:}T_1 = e\ \texttt{in}\ t :: \Delta_3[v'/x']\&v''[v'/x']$
with premises $\Delta_1 \vdash e :: \Delta_2\&v'$ and $(\Delta_2, x'{:}0 \vdash t :: \Delta_3\&v'')[v'/x']$ and where $e \notin \{\texttt{new L}, v'.f\}$. Induction on those two premises gives $(\Delta_1 \vdash e :: \Delta_2\&v')[l/x]$ and $(\Delta_2, x'{:}0 \vdash t :: \Delta_3\&v'')[v'/x'][l/x]$. Let $v = v'[l/x]$, then

$$[v'/x'][l/x] = [l/x][v'[l/x]/x'] = [l/x][v/x'] \tag{14}$$

and $x \neq x'$ further gives $(\Delta_2[l/x], x'{:}0 \vdash t[l/x] :: \Delta_3[l/x]\&v''[l/x])[v/x']$. Thus by T-LET

$$\frac{\Delta_1[l/x] \vdash e[l/x] :: \Delta_2[l/x]\&v \qquad (\Delta_2[l/x], x'{:}0 \vdash t[l/x] :: \Delta_3[l/x]\&v''[l/x])[v/x']}{\Delta_1[l/x] \vdash \texttt{let}\ x'{:}T_1 = e[l/x]\ \texttt{in}\ t[l/x] :: \Delta_3[l/x][v/x']\&v''[l/x][v/x']}$$

Using equation (14) again gives $(\Delta_1 \vdash \texttt{let}\ x'{:}T_1 = e\ \texttt{in}\ t :: \Delta_3[v'/x']\&v''[v'/x'])[l/x]$, as required. Finally $FE(\Delta_1, \Delta_2, v')$ implies $FE(\Delta_1[l/x], \Delta_2[l/x], v'[l/x])$ (cf. Lemma 13), which concludes the case.

*Case:* T-CALL: $\Delta_1 \vdash v.m(\vec{v}) :: \Delta_2$
with $\Delta_1 \geq \Delta_1'[\vec{v}/\vec{x}]$ and $\Delta_2 = \Delta_1 + (\Delta_2' - \Delta_1')[\vec{v}/\vec{x}]$ and were the method's interface specification is given by $C.m \vdash \Delta_1' \rightarrow \Delta_2'$. By Lemma 12, $\Delta_1[l/x] \geq \Delta_1'[\vec{v}[l/x]/\vec{x}]$. Note that by assuming that $x$ is different from all formal parameters, the condition for $x$ in that lemma is satisfied. The same assumption gives that $(\Delta_2' -$

$\Delta'_1)[\vec{v}[l/x]/\vec{x}] = (\Delta'_2 - \Delta'_1)[\vec{v}/\vec{x}][l/x]$. Hence $\Delta_1[l/x] + (\Delta'_2 - \Delta'_1)[\vec{v}[l/x]/\vec{x}]$ equals $\Delta_2[l/x]$, and therefore by T-CALL

$$\frac{\Delta_1[l/x] \geq \Delta'_1[\vec{v}[l/x]/\vec{x}] \qquad \Delta_2[l/x] = \Delta_1[l/x] + (\Delta'_2 - \Delta'_1)[\vec{v}[l/x]/\vec{x}]}{\Delta_1[l/x] \vdash v.m(\vec{v}[l/x]) :: \Delta_2[l/x]}$$

which concludes the case.

*Case:* T-NEW: $\Delta \vdash \texttt{new}\, C(\vec{v}) :: \Delta$
Immediate

*Case:* T-SPAWN: $\Delta \vdash \texttt{spawn}\, t :: \Delta$
By straightforward induction on the premise of that rule and T-SPAWN. It's easy to see that $\Delta' \vdash \mathit{free}$ implies $\Delta'[l/x] \vdash \mathit{free}$.

*Case:* T-NEWL: $\Delta_1 \vdash \texttt{let}\, x'\texttt{: L =new L in}\, t :: \Delta_2$
with $\Delta_1, x'{:}0 \vdash t :: \Delta_2$ as premise. Since $x'$ is a local variable, $x \neq x'$. Hence by induction $\Delta_1[x/l], x'{:}0 \vdash t[x/l] :: \Delta_2[x/l]$, from which the case follows by T-NEWL.

*Case:* T-LOCK: $\Delta \vdash v.\,\texttt{lock} :: \Delta + v$
If $v \neq x$, the case is straightforward. If $v = x$, we have $\Delta[l/x] \vdash l$. Thus $\Delta[l/x] \vdash l.\,\texttt{lock} :: \Delta[l/x] + l$ by T-LOCK, since $\Delta[l/x] + l = (\Delta + x)[l/x]$.

The case for T-UNLOCK works analogously to T-LOCK.

*Case:* T-TRYLOCK: $\Delta_1 \vdash \texttt{if}\, v.\,\texttt{trylock then}\, e_1\, \texttt{else}\, e_2 :: \Delta_2$
If $v \neq x$, the case follows by straightforward induction. If $v = x$, we get by induction on the second premise that $(\Delta_1 + x)[l/x] \vdash e_1 :: \Delta_2[l/x]$. Since again $(\Delta_1 + x)[l/x] = \Delta_1[l/x] + l$, the case follows by induction also on the third premise and T-TRYLOCK. □

The next lemma expresses that given a lock environment $\Delta_1$ as precondition for an expression $e$ such that the effect of $e$ leads to a post-condition of $\Delta_2$, $e$ is still well-typed if we assume a larger pre-condition where the lock balances are increased, and the corresponding post-condition is then increased accordingly.

**Lemma 15** (Weakening)**.** *If $\Delta_1 \vdash e :: \Delta_2$, then $\Delta_1 + \Delta \vdash e :: \Delta_2 + \Delta$.*

*Proof.* Straightforward. □

**Lemma 16** (Subject reduction (local))**.** *Assume $\sigma \vdash ok$ (i.e., $\sigma$ is well-formed) and $t$ is well-typed with $\sigma$, i.e., $\Gamma;\sigma \vdash t : T$ where $\Gamma$ is empty and for some type $T$. 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 $\sigma \vdash t \to \sigma' \vdash t'$, then $\Delta'_1 \vdash t' :: \Delta'_2 \& v'$, with $\Delta'_1 = \sigma \downarrow_p$ and with $\Delta'_2 \vdash free$.*

*Proof.* The proof proceeds by straightforward induction on the rules of Table 2, concentrating on the effect part (the typing part is covered by Lemma 10).

For the proof that $\Delta_1' = \sigma' \downarrow_p$ after the step, observe that for all local steps, $\sigma'$ is unchanged compared to $\sigma$ as far as the locks are concerned. Remark further that in all the cases below, $\Delta_1' = \Delta_1$.

*Case:* R-COND$_1$: $\sigma \vdash \texttt{let}\ x{:}T = \texttt{if}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t \to \sigma \vdash \texttt{let}\ x{:}T = e_1\ \texttt{in}\ t$

By straightforward induction, and R-COND$_2$ for the second branch works analogously.

*Case:* R-FIELD: $\sigma \vdash \texttt{let}\ x{:}\mathrm{L} = v'.f_i\ \texttt{in}\ t \to \sigma \vdash \texttt{let}\ x{:}\mathrm{L} = l\ \texttt{in}\ t$

By assumption, we are given $\Delta_1 \vdash \texttt{let}\ x{:}\mathrm{L} = v'.f_i\ \texttt{in}\ t :: \Delta_2 \& v$. Inverting the type rule T-FIELD for locks containing fields gives

$$\frac{\Delta_1, x{:}0 \vdash t :: \Delta_2 \& v}{\Delta_1 \vdash \texttt{let}\ x{:}\mathrm{L} = v'.f_i\ \texttt{in}\ t :: \Delta_2 \& v}\ \text{T-FIELD} \tag{15}$$

By the substitution Lemma 14, the premise implies $(\Delta_1, x{:}0 \vdash t :: \Delta_2 \& v)[l/x]$. The result follows by rule T-LET and T-REF as follows:

$$\frac{\dfrac{\Delta_1 \vdash l}{\Delta_1 \vdash l :: \Delta_1 \& l}\ \text{T-REF} \qquad (\Delta_1, x{:}0 \vdash t :: \Delta_2 \& v)[l/x]}{\Delta_1 \vdash \texttt{let}\ x{:}\mathrm{L} = l\ \texttt{in}\ t :: \Delta_2[l/x] \& v[l/x]}\ \text{T-LET} \tag{16}$$

Note that in the step, the heap remains unchanged and likewise, the context $\Delta_1$ remains unchanged in the step. Note further that $\Delta_2 \vdash \mathit{free}$ implies that also $\Delta_2[l/x] \vdash \mathit{free}$ (as the sum of two free locks ($x$ and $l$) is still free.

*Case:* R-ASSIGN: $\sigma \vdash \texttt{let}\ x{:}T = o.f := v\ \texttt{in}\ t \to \sigma' \vdash \texttt{let}\ x{:}T = v\ \texttt{in}\ t$

By the well-typedness assumption, we are given by inverting the rules T-LET and T-ASSIGN that

$$\frac{\Delta_1 \vdash o.f := v :: \Delta_1 \& v \qquad (\Delta_1, x{:}0 \vdash t :: \Delta_2)[v/x]}{\Delta_1 \vdash \texttt{let}\ x{:}T = o.f := v\ \texttt{in}\ t :: \Delta_2[v/x]} \tag{17}$$

The result then follows with T-LET and T-VAL$_1$:

$$\frac{\Delta_1 \vdash v :: \Delta_1 \& v \qquad (\Delta_1, x{:}0 \vdash t :: \Delta_2)[v/x]}{\Delta_1 \vdash \texttt{let}\ x{:}T = v\ \texttt{in}\ t :: \Delta_2[v/x]} \tag{18}$$

Note that T-ASSIGN allows to update a field containing a lock reference, independent of whether the lock is free or not.

*Case:* R-CALL: $\sigma \vdash \texttt{let}\, x{:}T = v.m(\vec{v})\ \texttt{in}\, t' \rightarrow \sigma \vdash \texttt{let}\, x{:}T = t[\vec{v}/\vec{x}][v/\mathsf{this}]\ \texttt{in}\, t'$. By the well-typedness assumption $\Delta_1 \vdash v.m(\vec{v}) :: \Delta_2$. Note that we do not allow that the method call gives back a lock (cf. rule T-METH), hence the judgment does not mention a value in which a lock is given back. By inverting rule T-CALL, we get for the effect specification of the called method $m$ that $\vdash C.m :: \Delta_1' \rightarrow \Delta_2'$ and for the method body $\vdash C.m = \lambda\vec{x}.t$ (as premise of rule R-CALL). As the whole program is well-typed, we know from the premise of rule T-METH that the body $t$ of the called method conforms to the given effect specification, which means

$$\Delta_1' \vdash t :: \Delta_2'$$

Using the substitution Lemma 14 for effects, this gives $\Delta_1'[\vec{v}/\vec{x}][v/\mathsf{this}] \vdash t[\vec{v}/\vec{x}][v/\mathsf{this}] :: \Delta_2'[\vec{v}/\vec{x}][v/\mathsf{this}]$ which implies

$$\Delta_1'[\vec{v}/\vec{x}] \vdash t[\vec{v}/\vec{x}][v/\mathsf{this}] :: \Delta_2'[\vec{v}/\vec{x}] \tag{19}$$

since $\mathsf{this}$ is an object reference. From the premise $\Delta_1 \geq \Delta_1'[\vec{v}/\vec{x}]$ of T-CALL, the result

$$\Delta_1 \vdash t[v/\mathsf{this}][\vec{v}/\vec{x}] :: \Delta_2 \tag{20}$$

follows by Lemma 15 by the following calculation: Let $\Delta_1'' = \Delta_1'[\vec{v}/\vec{x}]$ and $\Delta_2'' = \Delta_2'[\vec{v}/\vec{x}]$. We are given from the premise of rule T-CALL that $\Delta_1 \geq \Delta_1''$. Thus, we can define $\Delta = \Delta_1 - \Delta_1''$ (cf. Definition 1). Another premise of T-CALL gives

$$\Delta_2 = \Delta_1 + (\Delta_2'' - \Delta_1'') \tag{21}$$

The above equation (19) is equivalent to

$$\Delta_1'' \vdash t[\vec{v}/\vec{x}][v/\mathsf{this}] :: \Delta_2'' \tag{22}$$

which gives by the mentioned weakening lemma

$$\Delta_1'' + \Delta \vdash t[\vec{v}/\vec{x}][v/\mathsf{this}] :: \Delta_2'' + \Delta \tag{23}$$

which gives the judgement of equation (20), as required.

*Case:* R-NEW: $\sigma \vdash \texttt{let}\, x{:}T = \texttt{new}\, C(\vec{v})\ \texttt{in}\, t \rightarrow \sigma' \vdash \texttt{let}\, x{:}T = o\ \texttt{in}\, t$, where $\sigma'$ extends $\sigma$ by allocating the new instance $C(\vec{v})$, i.e, $\sigma' = \sigma[o \mapsto C(\vec{v})]$. We are given by the well-typedness assumption (by rule T-NEW and T-LET) that $\Delta \vdash \texttt{new}\, C(\vec{v}) :: \Delta$, i.e., $\Delta_1 = \Delta_2 = \Delta$, and the result for the expression after the step follows by T-VAL$_2$ and T-LET again.

*Case:* R-RED: $\sigma \vdash \texttt{let}\, x{:}\mathrm{L} = l \,\texttt{in}\, t \rightarrow \sigma \vdash t[l/x]$.
By the well-typedness assumption, we are further given

$$\frac{\Delta_1 \vdash l :: \Delta_1 \& l \qquad (\Delta_1, x{:}0 \vdash t :: \Delta_2 \& v)[l/x]}{\Delta_1 \vdash \texttt{let}\, x{:}\mathrm{L} = l \,\texttt{in}\, t :: \Delta_2[l/x] \& v[l/x]}\ \text{T-LET} \tag{24}$$

where $\Delta_1 = \sigma\downarrow_p$ and $\Delta_2[l/x] \vdash \mathit{free}$. Since the heap $\sigma$ remains unchanged in the step, the pre-context $\Delta_1'$ for after the reduction step is required to equal $\Delta_1$. The result $\Delta_1 \vdash t[l/x] :: \Delta_2' \& v'$ for some appropriate $\Delta_2'$ and $v'$ follows immediately from the second premise setting $\Delta_2' = \Delta_2[l/x]$ and observing that $(\Delta_1, x{:}0)[l/x] = \Delta_1$, as the lock reference exists in $\Delta_1$ already.

*Case:* R-LET: $\sigma \vdash \texttt{let}\, x_2{:}\mathrm{L} = (\texttt{let}\, x_1{:}\mathrm{L} = e_1 \,\texttt{in}\, t_1) \,\texttt{in}\, t_2 \rightarrow \sigma \vdash \texttt{let}\, x_1{:}\mathrm{L} = e_1 \,\texttt{in}\, (\texttt{let}\, x_2{:}\mathrm{L} = t_1 \,\texttt{in}\, t_2)$
By the well-typedness assumption, we are given by inverting the rule T-LET two times:

$$\frac{\dfrac{\Delta_1 \vdash e_1 :: \Delta_2 \& v_1 \quad (\Delta_2, x_1{:}0 \vdash t_1 :: \Delta_3 \& v_2)[v_1/x_1]}{\Delta_1 \vdash \texttt{let}\, x_1{:}\mathrm{L} = e_1 \,\texttt{in}\, t_1 :: \Delta_3' \& v_2'} \qquad (\Delta_3', x_2{:}0 \vdash t_2 :: \Delta_4 \& v_3)[v_2'/x_2]}{\Delta_1 \vdash \texttt{let}\, x_2{:}\mathrm{L} = (\texttt{let}\, x_1{:}\mathrm{L} = e_1 \,\texttt{in}\, t_1) \,\texttt{in}\, t_2 :: \Delta_4' \& v_3'} \tag{25}$$

where $\Delta_3' = \Delta_3[v_1/x_1]$ and $v_2' = v_2[v_1/x_1]$, and furthermore $\Delta_4' = \Delta_4[v_2'/x_2]$ and $v_3' = v_3[v_2'/x_2]$. Additionally, we have $FE(\Delta_1, \Delta_2, v_1)$ and $FE((\Delta_1, \Delta_3', v_2')$ as premises of the two instances of the let-rule.

Since the heap $\sigma$ remains unchanged in the step, the pre-context $\Delta_1'$ for after the reduction step is required to equal $\Delta_1$. Well-typedness for the program after the step is derived using T-LET two times as follows:

$$\frac{\Delta_1 \vdash e_1 :: \Delta_2 \& v_1 \qquad \dfrac{(\Delta_2, x_1{:}0 \vdash t_1 :: \Delta_3 \& v_2)[v_1/x_1] \quad (\Delta_3', x_2{:}0 \vdash t_2' :: \Delta_4 \& v_3)[v_2'/x_2]}{(\Delta_2, x_1{:}0 \vdash \texttt{let}\, x_2{:}\mathrm{L} = t_1 \,\texttt{in}\, t_2)[v_1/x_1] :: (\Delta_4 \& v_3)[v_2'/x_2]}}{\Delta_1 \vdash \texttt{let}\, x_1{:}\mathrm{L} = e_1 \,\texttt{in}\, (\texttt{let}\, x_2{:}\mathrm{L} = t_1 \,\texttt{in}\, t_2) :: (\Delta_4 \& v_3)[v_2'/x_2]} \tag{26}$$

where $\Delta_2' = \Delta_2[v_1/x_1]$ and $t_2' = t_2[v_1/x_1]$. Note that since $x_1$ does not occur in $t_2$, we have $t_2' = t_2$, i.e., the upper-most premise is covered, as well, by the corresponding premise from Equation 25.

The two premises concerning the return values

$$FE(\Delta_1, \Delta_2, v_1) \quad \text{and} \quad FE((\Delta_2, x_1{:}0)[v_1/x_1], \Delta_3[v_1/x_1], v_2[v_1/x_1]) \tag{27}$$

are proven as follows: The first one follows directly from the given derivation (25). Observing that $\Delta_2 \vdash v_1$, the second assertion is equal to

$$FE(\Delta_2, \Delta_3', v_2') \tag{28}$$

Since $dom(\Delta_2) \supseteq dom(\Delta_1)$, the assertion (28) follows from $FE(\Delta_1, \Delta_3', v_2')$ directly from the definition of $FE$. $\square$

**Lemma 17** (Subject reduction (global))**.** *If $\sigma \vdash P : ok$ and $\sigma \vdash P \Longrightarrow' \vdash P'$ where the reduction step is* not *given by one of the two* R-ERROR*-rules, then $\sigma' \vdash P' : ok$.*

*Proof.* By induction (for R-PAR) on the reduction rules of Table 3, using local subject reduction for the reduction from Lemma 16 (for T-LIFT). Apart from rule T-PAR which deals with the parallel composition of two threads, each rule covers one step of one thread $p$ (which in case of R-SPAWN spawns a second one). In all rules except T-PAR we set $\Delta_1 = \sigma \downarrow_p$, as given in the premise of rule T-THREAD.

*Case:* R-LIFT: $\sigma \vdash p\langle t\rangle \Longrightarrow' \vdash p\langle t'\rangle$,
with $\sigma \vdash t \to \sigma' \vdash t'$ from the premise of R-LIFT. Remember that a thread $t$ is an expression $e$ as well in the grammar of Table 1. A reduction step on the local level (as in the premise of R-LIFT) does not change any lock. The assumption $\sigma \vdash p\langle t\rangle : ok$ implies by inverting rule T-THREAD $\Delta_1 \vdash t :: \Delta_2$ (concentrating on the effect part), where $\Delta_1 = \sigma \downarrow_p$, holds as pre-condition and $\Delta_2 \vdash free$ afterwards. The lock environment $\Delta_1$ represents the lock status from the perspective of thread $p$ (cf. Definition 5). Subject reduction on the local level (Lemma 16) gives $\Delta_1' \vdash t' :: \Delta_2'$, where $\Delta_1' = \sigma' \downarrow_p$ (which implies for the local steps that $\Delta_1' = \Delta_1$). Furthermore, local subject reduction gives $\Delta_2' \vdash free$. Since the local step does not affect the locks in the heap, $\sigma \downarrow_p = \sigma' \downarrow_p = \Delta_1$, and the result $\sigma' \vdash p\langle t'\rangle : ok$ follows by T-THREAD.

$$\frac{\Delta_1' = \sigma' \downarrow_p \qquad \Delta_1' \vdash t' :: \Delta_2' \qquad t' \neq \texttt{error} \qquad \Delta_2' \vdash free}{\sigma' \vdash p\langle t'\rangle : ok} \text{ T-THREAD} \tag{29}$$

*Case:* R-PAR: $\sigma \vdash P_1 \parallel P_2 \Longrightarrow' \vdash P_1' \parallel P_2$
where $\sigma \vdash P_1 \Longrightarrow' \vdash P_1'$. By straightforward induction and mutual exclusion in the sense that each thread can manipulate only locks it owns or free locks: By the well-typedness assumption and the premises of T-PAR, we know $\sigma \vdash P_1 : ok$ and $\sigma \vdash P_2 : ok$. By induction thus $\sigma' \vdash P_1' : ok$. Since $P_1$ in the step from $\sigma \vdash P_1 \Longrightarrow' \vdash P_1'$ cannot change locks held by processes in $P_2$, also $\sigma' \vdash P_2 : ok$, so the result follows by rule T-PAR:

$$\frac{\sigma' \vdash P_1' : ok \qquad \sigma' \vdash P_2 : ok}{\sigma' \vdash P_1' \parallel P_2 : ok} \text{ T-THREAD} \tag{30}$$

*Case:* R-SPAWN: $\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{spawn}\ t'\ \texttt{in}\ t\rangle \Longrightarrow \vdash p\langle \texttt{let}\ x{:}T = ()\ \texttt{in}\ t\rangle \parallel p'\langle t'\rangle$

The well-typedness assumption for the configuration before the step, inverting rule T-THREAD, T-LET, and T-SPAWN, we obtain the following derivation tree:

$$\dfrac{\dfrac{\dfrac{\bullet \vdash t' :: \Delta' \quad \Delta' \vdash free}{\Delta_1 \vdash \texttt{spawn}\ t' :: \Delta_1}\ \text{T-SPAWN} \qquad \Delta_1 \vdash t :: \Delta_2}{\Delta_1 \vdash \texttt{let}\ x{:}T = \texttt{spawn}\ t'\ \texttt{in}\ t :: \Delta_2}\ \text{T-LET} \qquad \Delta_2 \vdash free}{\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{spawn}\ t'\ \texttt{in}\ t\rangle : ok}\ \text{T-THREAD}$$

Note that to check $t'$ in the left upper premise, the lock context as precondition is empty. The result then follows by T-UNIT, T-LET, T-THREAD, and T-PAR. Note further that spawning a new thread does not return a lock reference as value; hence the typing rule for let does not have to deal with substitution.

$$\dfrac{\dfrac{\dfrac{\Delta_1 \vdash\ () :: \Delta_1 \quad \Delta_1 \vdash\ t :: \Delta_2}{\Delta_1 \vdash \texttt{let}\ x{:}T = ()\ \texttt{in}\ t :: \Delta_2}\ \text{T-LET}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = ()\ \texttt{in}\ t\rangle : ok} \qquad \dfrac{\Delta_1' \vdash t' ::\ \Delta' \quad \Delta_1' = \sigma\downarrow_{p'}}{\sigma \vdash p'\langle t'\rangle : ok}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = ()\ \texttt{in}\ t\rangle \parallel p'\langle t'\rangle : ok}$$

For the validity of $\Delta_1' \vdash t' ::\ \Delta'$ in the premise of T-THREAD in the upper right sub-goal of the derivation: note that the new thread $p'$ does not own any lock immediately after creation. This means, the projection $\sigma\downarrow_{p'} = \Delta_1'$ is the empty context $\bullet$ and this covered by the left upper sub-goal of the derivation from the assumption.

*Case:* R-NEWL: $\sigma \vdash p\langle \texttt{let}\ x{:}\mathrm{L} = \texttt{new L in}\ t\rangle \to \sigma' \vdash p\langle \texttt{let}\ x{:}\mathrm{L} = l\ \texttt{in}\ t\rangle$, where $l$ is fresh and $\sigma' = \sigma[l \mapsto 0]$. The case works rather similar to the one for R-FIELD for subject reduction on the local level: By assumption we are given $\Delta_1 \vdash \texttt{let}\ x{:}\mathrm{L} = \texttt{new L in}\ t :: \Delta_2$. Inverting the type rules T-THREAD and T-NEWL gives

$$\dfrac{\dfrac{\Delta_1, x{:}0 \vdash t :: \Delta_2}{\Delta_1 \vdash \texttt{let}\ x{:}\mathrm{L} = \texttt{new L in}\ t :: \Delta_2}\ \text{T-NEWL}}{\sigma \vdash p\langle \texttt{let}\ x{:}\mathrm{L} = \texttt{new L in}\ t\rangle : ok}\ \text{T-THREAD} \tag{31}$$

where $\Delta_1 = \sigma\downarrow_p$ and $\Delta_2 \vdash free$. By the substitution Lemma 14, the premise implies $(\Delta_1, x{:}0 \vdash t :: \Delta_2)[l/x]$. The result follows by rules T-VAL$_1$, T-LET, and T-THREAD as follows:

$$\dfrac{\dfrac{\dfrac{\Delta_1, l{:}0 \vdash l}{\Delta_1, l{:}0 \vdash l :: \Delta_1, l{:}0 \& l}\,\text{T-V}\textsc{al}_1 \qquad (\Delta_1, l{:}0, x{:}0 \vdash t :: \Delta_2)[l/x] \quad FE(\Delta_1, l{:}0, \Delta_1,, l)}{\Delta_1, l{:}0 \vdash \texttt{let}\ x{:}\mathrm{L} = l\ \texttt{in}\ t :: \Delta_2[l/x]}\,\text{T-L}\textsc{et}}{\sigma' \vdash p\langle \texttt{let}\ x{:}\mathrm{L} = l\ \texttt{in}\ t\rangle :ok}\,\text{T-T}\textsc{hread} \tag{32}$$

Note that $\Delta_1, l{:}0 = \sigma' \downarrow_p$. Note the difference between the previous case of field look-up for fields containing a lock reference and the creation of a new lock here. In both cases, the premises of the typing rule is actually identical (cf. equations (15) and (31)). The difference is that for field look-up, the lock reference is present in $\Delta_1$ whereas for lock creation it is not, as it's freshly created in the step. Therefore, in the first case $(\Delta_1, x{:}0)[l/x]$ equals $\Delta_1$, whereas in the second case it equals $\Delta_1, l{:}0$. Note finally that $\Delta_2 \vdash \mathit{free}$ implies that also $\Delta_2[l/x] \vdash \mathit{free}$, (as the sum of two free locks ($x$ and $l$) is still free).

*Case:* R-LOCK$_1$: $\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\,\texttt{lock}\ \texttt{in}\ t\rangle \to \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle$, where $\sigma' \downarrow_p = \sigma \downarrow_p + l$. The well-typedness assumption gives a derivation as follows:

$$\dfrac{\Delta_1 = \sigma \downarrow_p \qquad \dfrac{\dfrac{\Delta_1' = \Delta_1 + l}{\Delta_1 \vdash l.\,\texttt{lock} :: \Delta_1' \& l} \qquad (\Delta_1', x{:}0 \vdash t :: \Delta_2)[l/x]}{\Delta_1 \vdash \texttt{let}\ x{:}T = l.\,\texttt{lock}\ \texttt{in}\ t :: \Delta_2[l/x]} \qquad \Delta_2[l/x] \vdash \mathit{free}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\,\texttt{lock}\ \texttt{in}\ t\rangle : ok}$$

From that, the result follows by T-THREAD, T-LET, and T-VAL$_1$.

$$\dfrac{\Delta_1' = \sigma' \downarrow_p \qquad \dfrac{\Delta_1' \vdash l :: \Delta_1' \& l \qquad (\Delta_1', x{:}0 \vdash t :: \Delta_2)[l/x]}{\Delta_1' \vdash \texttt{let}\ x{:}T = l\ \texttt{in}\ t :: \Delta_2[l/x]} \qquad \Delta_2[l/x] \vdash \mathit{free}}{\sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle : ok}$$

The cases for R-LOCK$_2$ and for unlocking work analogously.

*Case:* R-TRYLOCK$_1$: $\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{if}\ l.\,\texttt{trylock}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t\rangle \Longrightarrow' \vdash p\langle \texttt{let}\ x{:}T = e_1\ \texttt{in}\ t\rangle$ where $\sigma(l) = 0$ and $\sigma' \downarrow_p = \sigma \downarrow_p + l$. The assumption of well-typedness gives the following derivation:

$$\dfrac{\Delta_1 = \sigma \downarrow_p \qquad \dfrac{\dfrac{\Delta_1' = \Delta_1 + l \quad \Delta_1' \vdash e_1 :: \Delta_3 \& v \quad \Delta_1 \vdash e_2 :: \Delta_3 \& v}{\Delta_1 \vdash \texttt{if}\ l.\,\texttt{trylock}\ \texttt{then}\ e_1\ \texttt{else}\ e_2 :: \Delta_3 \& v} \qquad (\Delta_3, x{:}0 \vdash t :: \Delta_2)[v/x]}{\Delta_1 \vdash \texttt{let}\ x{:}T = \texttt{if}\ l.\,\texttt{trylock}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t :: \Delta_2[v/x]} \qquad \Delta_2[v/x] \vdash \mathit{free}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{if}\ l.\,\texttt{trylock}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t\rangle : ok}$$

The case then follows by T-THREAD and T-LET.

$$\dfrac{\Delta_1' = \sigma' \downarrow_p \qquad \dfrac{\Delta_1' \vdash e_1 :: \Delta_3 \& v \quad (\Delta_3, x{:}0 \vdash t :: \Delta_2)[v/x]}{\Delta_1' \vdash \mathtt{let}\, x{:}T = e_1 \;\mathtt{in}\, t :: \Delta_2[v/x]} \qquad \Delta_2[v/x] \vdash \mathit{free}}{\sigma' \vdash p\langle \mathtt{let}\, x{:}T = e_1 \;\mathtt{in}\, t\rangle : \mathit{ok}}$$

The cases for R-TRYLOCK$_2$ and R-TRYLOCK$_3$ work similarly. □

The next lemma states that a well-typed configuration does not exhibit an lock-error in the *next* step. Together with preservation of well-typedness under reduction, this property assures that for a program starting statically well-typed, *never* a lock error will occur.

**Lemma 18.** *Let $P = P' \parallel p\langle t\rangle$. If $\sigma \vdash P : ok$ then $\sigma \vdash P \not\Longrightarrow \sigma \vdash P' \parallel p\langle \mathtt{error}\rangle$.*

*Proof.* Let $\sigma \vdash P : ok$ and assume for a contradiction that $\sigma \vdash P \Longrightarrow \vdash P' \parallel p\langle \mathtt{error}$ $\rangle$. From the rules of the operational semantics it follows that $P = p\langle \mathtt{let}\, x{:}T = l.\,\mathtt{unlock}\;\mathtt{in}\, t'\rangle \parallel P'$ for some thread $t'$. Furthermore, either (1) the lock is currently held by a thread different from $p$ or (2) the lock is free (cf. rules R-ERROR$_1$ and R-ERROR$_2$).

To be well-typed, i.e., for the judgment $\sigma \vdash p\langle \mathtt{let}\, x{:}T = l.\,\mathtt{unlock}\;\mathtt{in}\, t'\rangle \parallel P' :$ *ok* to be derivable, it is easy to see (by inverting T-PAR and T-THREAD) that the derivation must contain $\Delta_1 \vdash \mathtt{let}\, x{:}T = l.\,\mathtt{unlock}\;\mathtt{in}\, t' : T' :: \Delta_2$ as sub-goal, where the lock-context $\Delta_1$ is given as the local projection of $\sigma$ onto $p$, i.e., $\Delta_1 = \sigma \downarrow_p$. By the definition of projection (cf. Definition 5), both case (1) and (2) give that $\Delta(l) = 0$. This is a contradiction, as the premise of T-UNLOCK requires that the lock is taken with an $n \geq 1$. □

The next result captures one of the two aspects of correct lock handling, namely that never a lock is improperly unlocked.

**Theorem 19** (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 $\bullet \vdash P_0 \longrightarrow^* \sigma' \vdash P \parallel p\langle \mathtt{error}\rangle$.*

*Proof.* A direct consequence of subject reduction and Lemma 18. Note that subject reduction preserves well-typedness *only* under steps which are no error steps. □

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 $P = P' \parallel p\langle \mathtt{stop}\rangle$ 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 20** (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.*

*Proof.* A consequence of subject reduction. Note that Lemma 16 preserves the property for the post-context, that all locks are free. $\square$

## 6. Exception handling

In this section, we equip our language with exception handling constructs and extend our type and effect system accordingly. In the presentation so far, there has been one situation which constitutes an exceptional situation, namely the improper use of an unlocking statement. In the operational rules, such a lock error reduces a thread to the `error` state (cf. the two R-ERROR-rules of Table 3). Basically, that corresponds to throwing an exception without catching it.

We start by adding syntax for exception throwing and handling. As in Java, the construct for handling exceptions, in its general form, consists of three parts or blocks: The try-part harnesses the code which may raise an exception, one catch-branch is executed if it matches an exception raised in the try-block. The catch-clauses work like a case construct in that *at most* one case-branch is executed and which one (if any) is decided on a first-match policy. Especially, if an exception is thrown in one of the catch-clauses, it cannot be fielded in a subsequent catch-clause of the same try-catch-finally expression. The trailing finally-clause is unconditionally executed, i.e., independent of whether or not an exception is raised and/or caught in the try- and the catch-clauses.

We extend the abstract syntax from Table 1 by extending the expressions $e$ as given in Table 6. For the threads $t$, the `error`-thread is replaced by $\texttt{error}(E)$ which represents abnormal termination by a thrown exception $E$. We also slightly extended the *types* as given in Table 1 by adding the type `Top`. The type is used for technical reasons and on the run-time syntax only, i.e., it is not available at the user-level.

$$
\begin{array}{rcl}
t & ::= & \texttt{error}(E) \\
e & ::= & \texttt{throw}\ E \mid \texttt{try}\ e\ cb\ \texttt{finally}\ e \\
cb & ::= & \varepsilon \mid \texttt{catch}\ E > e;\ cb
\end{array}
$$

Table 6: Abstract syntax, exceptions

Concentrating on relevant cases of the control flow, we simplified the language as compared to Java, while still keeping different situations as far as the control flow is concerned. We omitted inheritance and thus subtyping from the calculus. The different exceptions are represented by $E$, where $E ::= \mathtt{E}_{unlock} \mid \mathtt{E}_1 \mid \mathtt{E}_2 \mid \ldots$. We assume one specific exception $\mathtt{E}_{unlock}$ representing lock errors and which we will prove that it is never thrown. In Java, the finally block is optional. In our abstract syntax, a try-construct always has a finally-clause, but a missing one can be represented by an "empty" finally-expression. The try-catch-finally construct consists therefore of three parts: the try clause, followed by a finite list of catch-branches, called *cb* in the abstract syntax for exceptions, and a trailing finally-clause.

The operational behavior is specified in Table 7. Rule R-THROW throws an exception, here represented by $\mathtt{error}(E)$. Evaluating a thrown exception $E$, i.e., evaluating $\mathtt{error}(E)$, without being caught ignores the rest of the thread (cf. rule R-ERROR). The two R-TRY-rules evaluate the try-clause. The first rule simply does one step in evaluating the try-expression as part of the larger try-expression. In rule R-TRY$_2$, the corresponding expression is evaluated to a value $v$ (which is a normal value, not a thrown exception $\mathtt{error}(E)$) and the evaluation continues with the finally clause.

$$\sigma \vdash \mathtt{let}\, x{:}T = \mathtt{throw}\ E\ \mathtt{in}\, t \rightarrow \sigma \vdash \mathtt{let}\, x{:}T = \mathtt{error}(E)\ \mathtt{in}\, t \quad \text{R-THROW}$$

$$\sigma \vdash \mathtt{let}\, x{:}T = \mathtt{error}(E)\ \mathtt{in}\ t \rightarrow \sigma \vdash \mathtt{error}(E) \quad \text{R-ERROR}$$

$$\frac{\sigma \vdash e_1 \rightarrow \sigma' \vdash e_1'}{\sigma \vdash \mathtt{try}\ e_1\ cb\ \mathtt{finally}\ e_2 \rightarrow \sigma' \vdash \mathtt{try}\ e_1'\ cb\ \mathtt{finally}\ e_3} \ \text{R-TRY}_1$$

$$\sigma \vdash \mathtt{try}\ v\ cb\ \mathtt{finally}\ e_2 \rightarrow \sigma \vdash e_2 \quad \text{R-TRY}_2$$

$$\sigma \vdash \mathtt{try}\ \mathtt{error}(E)\ \mathtt{catch}\ E > e_1;\ cb\ \mathtt{finally}\ e_2 \rightarrow \sigma \vdash \mathtt{try}\ e_1\ \mathtt{finally}\ e_2 \quad \text{R-CATCH}$$

$$\frac{E \neq E_1}{\sigma \vdash \mathtt{try}\ \mathtt{error}(E)\ \mathtt{catch}\ E_1 > e_1;\ cb\ \mathtt{finally}\ e_2 \rightarrow \sigma \vdash \mathtt{try}\, \mathtt{error}(E)\, cb\, \mathtt{finally}\ e_2} \ \text{R-NEXTCATCH}$$

$$\sigma \vdash \mathtt{try}\ \mathtt{error}(E)\ \mathtt{finally}\ e \rightarrow \sigma \vdash \mathtt{let}\, x'{:}\,\mathtt{Top}{=}\, e\ \mathtt{in}\ \mathtt{error}(E) \quad \text{R-NOCATCH}$$

Table 7: Exception handling

Catching an exception is formalized in rule R-CATCH where the evaluation continues with the expression $e_1$ of the catch clause. As mentioned above, if $e_1$ throws another exception during its evaluation it will *not* be caught again, at least not by the try-construct whose steps we describe in the rule, but perhaps by an enclosing one. This means, after the step, the catch-clauses have disappeared. Rule R-NEXTCATCH formalizes the first-match policy: if the first branch does not match, it is discarded and the remaining branches are checked. Rule R-NOCATCH deals with the situation where a thrown exception is not caught. In this situation, the control flow continues with evaluating the finally clause $e$. Note also that the error in rule R-NOCATCH can originally come from a try clause or from a catch clause, as rule R-CATCH transforms a try-catch-finally expression into try-finally form, where the orignal catch-expresssion can raise (another) exception. If evaluating the finally clause does not raise an exception itself, the original exception needs to be propagated. This is specified in R-NOCATCH by the expression $\texttt{let}\,x'\!:\texttt{Top}{=}\,e\;\texttt{in}\;\texttt{error}(E)$ (which corresponds to $e;\texttt{error}(E)$, as $x'$ does not occur in $\texttt{error}(E)$). If, however, $e$ throws an (uncaught) exception itself, the trailing $\texttt{error}(E)$ is ignored. Note that since we have to formulate the reduction rule independant of the type of $e$, we use `Top` to give a type to $x$ which subsumes whatever type $e$ is.

With the language extended by a throw-expression, we can reformulate the erroneous lock handling steps from Table 3 as follows:

$$\frac{\sigma(l) = p'(n) \qquad p \neq p'}{\sigma \vdash p\langle \texttt{let}\, x : T = l.\,\texttt{unlock in}\, t\rangle \Longrightarrow\vdash p\langle \texttt{let}\, x : T = \texttt{throw}\, \mathrm{E}_{unlock}\, \texttt{in}\, t\rangle}\ \text{R-ERROR}_1$$

$$\frac{\sigma(l) = 0}{\sigma \vdash p\langle \texttt{let}\, x : T = l.\,\texttt{unlock in}\, t\rangle \Longrightarrow\vdash p\langle \texttt{let}\, x : T = \texttt{throw}\, \mathrm{E}_{unlock}\, \texttt{in}\, t\rangle}\ \text{R-ERROR}_2$$

*Static analysis*

Apart from adding type rules to deal with the new constructs of throwing and catching exceptions, the type and effect system needs to be extended in general to express the possibility of exceptions being thrown. This is expressed by introducing (another) effect, basically the "set" of potential exceptions raised (and not caught) during the execution of an expression or thread. With this extra information, the judgment will take the following form

$$\sigma;\Gamma;\Delta_1 \vdash e : T :: \Delta_2, \Omega, [\&v] \tag{33}$$

where $\Omega$ is the mentioned effect capturing the potential exceptions. Assuming lock counters as given by the precondition $\Delta_1$ before executing $e$, the purpose of the analysis is to keep track over the lock counters. Therefore, the information about *which* exceptions are thrown is not enough to prevent lock errors. We additionally need information about the different lock status at the different control points where the exceptions are thrown in case $e$ is exited abnormally, i.e., by an exception. Therefore, the effect $\Omega$ is of the following form

$$\Omega ::= \emptyset \mid \Omega, E(\Delta) \tag{34}$$

where the $\Delta$ is a lock context, i.e., of the form $\vec{v}{:}\vec{n}$. For $\Omega$, we assume that each exception $E$ occurs at most once in $\Omega$ and that the order of $E$'s in $\Omega$ is irrelevant (i.e., the comma-separator is assumed to be associative and commutative). So the efffect $\Omega$ in the judgment of Equation 33 specifies: if $\mathtt{E}(\vec{v}{:}\vec{n}) \in \Omega$, then $e$ potentially thows exception $\mathtt{E}$, and at *all* program points where it may be thrown, the status of the lock counters is described by $\vec{v}{:}\vec{n}$.

The rules for the type and effect system are given in Table 8. The rules are used in addition to the rules from Section 4. Having introduced `Top` as additional type, we add furthermore variants of the rules T-LET, T-FIELD, and T-NEWL, i.e., the local rules dealing with the let-construct. The variants T-LET-TOP, T-FIELD-TOP, and T-NEWL-TOP correspond to the original versions except that the type of the let-bound variable is required to be `Top`. Since we have slightly generalized the syntax of a thrown exception from `error` to $\mathtt{error}(E)$, the corresponding rule T-ERROR is adapted accordingly. Furthermore, to avoid hanging locks, the old spawn rule T-SPAWN from Table 4 which required that in the post-condition $\Delta'$ all locks are free is extended now with an additional premise requiring that also for all post-conditions in uncaught exceptions, the locks are free. I.e., T-SPAWN now has $\Omega \vdash \mathit{free}$ as additional premise, where the assertion is defined as: Given $\Omega = E_1(\Delta_1), \ldots, E_k(\Delta_k)$, then $\Omega \vdash \mathit{free}$ if $\Delta_i \vdash \mathit{free}$ for all $i$.

The treatment for throwing an exception, resp. a thrown exception is straightforward (cf. rules T-THROW and T-ERROR). As the control flow never reaches the point directly *after* the $\mathtt{throw}\,E$-expression resp. the error-expression, it can be given any type $T$. For the same reason, the lock context as post-condition for normal termination is irrelevant, so the rules specify $\Delta'$ as an arbitrary post-context. As far as the exception-effects are concerned: clearly, exception $E$ is (being) thrown, and therefore included in $\Omega$. To record the current lock-counters, $\Omega$ is of the form $E(\Delta)$.

The analysis of the try-catch-finally construct is done in rule T-TCF. The treatment of the underlying types is straightforward: the try-clause $e$ must be well-typed, and the type of the finally block $e'$ is the type of the overall expression. More complex is to keep track of the lock counters and the thrown exceptions, as throwing an exception leaves the ordinary left-to-right control-flow. Basically, we need to cover the following control-flows between the different parts of the try-catch-finally expression. See also Figure 2, which sketches the control-flow for the expression $\mathtt{try}\ e\ \mathtt{catch}\ E_1 > e_1; \ldots; \mathtt{catch}\, E_k > e_k\ \mathtt{finally}\, e'$.

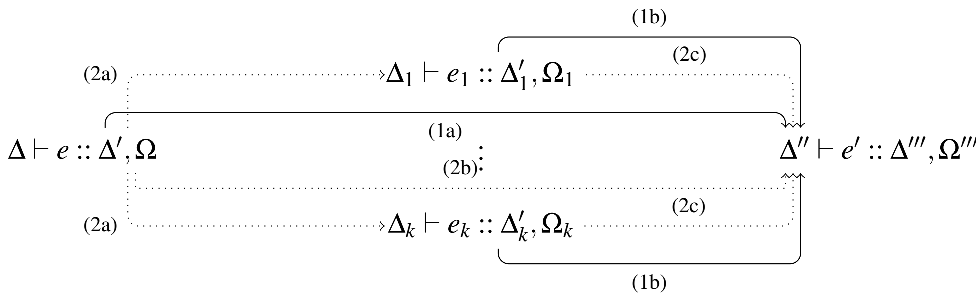



Figure 2: Control-flow for try-catch-finally blocks

$$\frac{}{\sigma;\Gamma;\Delta \vdash \texttt{error}(E)\ : T :: \Delta', E(\Delta)}\ \textsc{T-Error} \qquad \frac{}{\sigma;\Gamma;\Delta \vdash \texttt{throw}\ E : T :: \Delta', E(\Delta)}\ \textsc{T-Throw}$$

$$\frac{\begin{array}{c} \sigma;\Gamma;\Delta \vdash e : T :: \Delta', \Omega \qquad \Delta' = \Delta'' \\ \sigma;\Gamma;\Delta_i \vdash e_i : T :: \Delta'_i, \Omega_i \qquad \forall i \in \{1,\ldots,k\}.E_i \in \Omega \Rightarrow \Delta'_i = \Delta'' \quad \Omega(E_i) = \Delta_i \quad FE(\Delta'', \Delta''', v') \\ \lfloor \Omega \setminus \Omega_{caught} \rfloor = \Delta'' \quad \forall i \in \{1,\ldots,k\}.E_i \in \Omega \Rightarrow \lfloor \Omega_i \rfloor = \Delta'' \quad \sigma;\Gamma;\Delta'' \vdash e' : T' :: \Delta''', \Omega'''\ [\&v'] \\ \textit{where} \quad cb = \texttt{catch}\, E_1 > e_1; \ldots; \texttt{catch}\, E_k > e_k \quad \Omega_{caught} = E_1(\Delta_1), \ldots, E_k(\Delta_k) \end{array}}{\sigma;\Gamma;\Delta \vdash \texttt{try}\ e\ cb\ \texttt{finally}\ e' : T' :: \Delta''', (\Omega''' + ((\Omega \setminus \Omega_{caught})[\_ \mapsto \Delta''']) + \sum_{i \mid E_i \in \Omega} \Omega'_i[\_ \mapsto \Delta''']) \ [\&v']}\ \text{T-TCF}$$

$$\frac{\sigma;\Gamma;\Delta \vdash e : T :: \Delta;\Omega[\&v] \qquad \Omega \leq \Omega'}{\sigma;\Gamma;\Delta \vdash e : T :: \Delta;\Omega'[\&v]}\ \textsc{T-Sub}$$

Table 8: Type and effect system (exceptions)

1. *Non-exceptional control flow:* (solid lines)
   (a) from the post-context of the try-block to the pre-context of the finally clause.
   (b) from the post-contexts of all branches to the pre-context of the finally clause.
2. *Exceptional control flow:* (dotted lines)
   (a) from the post-context of the try-clause to one of the catch clauses, in case of a caught exception.
   (b) from the post-context of the try-clause to the pre-context of the finally clause, in case such a thrown exception falls through.
   (c) From the post-contexts of the catch-clauses to the pre-context of the finally clause, in case a catch-clause throws an exception itself.

The typing judgments distinguish, as far as the post-contexts for lock-counters are concerned, between the non-exceptional post-context $\Delta$ and the exceptional one $\Omega$. The rule T-TCF must connect the pre-and post-contexts appropriately, as just discribed informally. The first premise of T-TCF handles the expression $e$ of the try-block where $\Delta'$ contains the context for the lock-counters if the try-block is exited normally, and $\Omega$ the corresponding information (per exception) for the exceptional termination of $e$. The second premise $\Delta' = \Delta''$ covers case (1a). The next two premises deal with the analysis of the catch-branches. For case (1b), each ordinary post-contexts $\Delta'_i$ for each branch must coincide with the pre-condition $\Delta''$ of the finally-clause, i.e., we require $\Delta'_i = \Delta''$. This, however, is necessary only for those catch-branches, which *may* be executed at all, i.e., for which the try-block may throw a corresponding exception. That information is contained in the exceptional post-condition $\Omega$ of the try-expression $e$ (from the first premise). Therefore, $\Delta'_i = \Delta''$ is required only for those $i$'s where the exception $E_i$ occurs in $\Omega$. Case (2a) is directly covered by the next premise $\Omega(E_i) = \Delta_i$ (where $\Omega(E_i)$ is meant as the lock context of exception $E_i$ as given in $\Omega$). To connect the exception post-context $\Omega$ of the try-block with the pre-context of the finally-block in case (2b), we need to determine all potential exceptions from $\Omega$ which are *not* caught. The context $\Omega_{caught}$ in T-TCF contains all caught exceptions, and the "difference" $\Omega \setminus \Omega_{caught}$ the ones that fall through. Since the finally-block is entered *irrespective* of which exception is actually thrown, we need to strip off that information from $\Omega \setminus \Omega_{caught}$. The case (2c) covered by the premise $\lfloor \Omega_i \rfloor = \Delta''$ requires that all exceptions raised in catch-blocks must agree on the lock-counters before enterring the finally-block. See Definition 21 for the corresponding context relations. The premise $FE(\Delta'', \Delta''', v')$ assures that $e$ does not create locks which are left with a balance $> 0$ after $e$ except potentially $v'$ (cf. the definition of $FE$ in Section 4).

**Definition 21** (Operations and order relation on exceptional lock contexts)**.** *Given an exceptional lock context $\Omega$. If $\Omega = \bullet$, $\lfloor \Omega \rfloor = \bullet$, if $\Omega = E_1(\Delta), \ldots E_k(\Delta)$ for a $k \geq 1$, then the context $\lfloor \Omega \rfloor = \Delta$. Otherwise, $\lfloor \Omega \rfloor$ is undefined. The* difference $\Omega_1 \setminus \Omega_2$ *of two contexts $\Omega_1$ and $\Omega_2$ is given as follows: let $\Omega = \Omega_1 \setminus \Omega_2$, then $\Omega(E) = \Omega_1(E)$ if $\Omega_2(E)$ is undefined, and $\Omega(E)$ is undefined otherwise. The* sum $\Omega = \Omega_1 + \Omega_2$ *is defined as: $\Omega(E) = \Omega_1(E)$ if $\Omega_2(E)$ is undefined, else $\Omega(E) = \Omega_2(E)$ if $\Omega_1(E)$ is undefined; if $\Omega_1(E) = \Omega_2(E)$, then $\Omega(E) = \Omega_1(E)$ $(= \Omega_2(E))$. $\Omega(E)$ is undefined if $\Omega_1(E) \neq \Omega_2(E)$. We write $\Omega_1 \leq \Omega_2$, if $\Omega_2 = \Omega_1, E_1(\Delta_1), \ldots, E_k(\Delta_k)$ where $k \geq 0$. Assuming $E \in \Omega$, the updated context $\Omega' = \Omega[E \mapsto \Delta]$ is defined as: $\Omega'(E') = \Omega(E')$ for all $E' \neq E$ and as $\Omega'(E) = \Delta$. The context $\Omega[\_ \mapsto \Delta]$ is $\Omega$, where $\Omega[E \mapsto \Delta]$ is applied for* all $E \in \Omega$.

Note that $\Omega_1 \leq \Omega_1 + \Omega_2$. Since the exceptional lock context $\Omega$ describe the set of potential exceptions, they are naturally ordered and a typing of an expression can be relaxed via subsumption.

Now we neeed to extend the preservation of well-typedness (Lemma 10 and 16) to deal with the new rules.

**Lemma 22** (Subject reduction)**.** *If* $\sigma;\Gamma \vdash e_1 : T$ *and* $\sigma \vdash e_1 \to \sigma' \vdash e_2$*, then* $\sigma';\Gamma \vdash e_2 : T$.

*Proof.* Proceed by induction on the derivation of the reduction step.

*Case:* R-THROW: $\sigma \vdash \texttt{let}\ x{:}T = \texttt{throw}\, E\ \texttt{in}\, t \to \sigma \vdash \texttt{let}\ x{:}T = \texttt{error}(E)\ \texttt{in}\, t$
Immediate, since $\texttt{throw}\, E$ as well as $\texttt{error}(E)$ can be of arbitrary type.

*Case:* R-ERROR: $\sigma \vdash \texttt{let}\ x{:}T = \texttt{error}(E)\ \texttt{in}\, t \to \sigma \vdash \texttt{error}(E)$
Immediate by rule T-ERROR.

*Case:* R-TRY$_1$: $\sigma \vdash \texttt{try}\, e_1\ cb\ \texttt{finally}\, e_2 \to \sigma' \vdash \texttt{try}\, e_1'\ cb\ \texttt{finally}\, e_2$
where $\sigma \vdash e_1 \to \sigma' \vdash e_1'$. The case follows by straightforward induction.

*Case:* R-TRY$_2$: $\sigma \vdash \texttt{try}\, v\ cb\ \texttt{finally}\, e_2 \to \sigma' \vdash\ e_2$
From $\sigma \vdash \texttt{try}\, v\ cb\ \texttt{finally}\, e_2 : T$ and inverting T-TCF, we get $\sigma;\Gamma \vdash e_2 : T$, as required.

*Case:* R-CATCH: $\sigma \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{catch}\, E > e_1;cb\ \texttt{finally}\, e_2 \to \sigma \vdash \texttt{try}\ e_1\, \texttt{finally}\, e_2$
The well-typedness assumption and inverting T-TCF gives $\sigma;\Gamma \vdash e_1 : T$ and $\sigma;\Gamma \vdash e_2 : T'$, so the result follows by T-TCF.

*Case:* R-NEXTCATCH: $\sigma \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{catch}\, E_1 > e_1;cb\ \texttt{finally}\, e_2 \to \sigma \vdash \texttt{try}\ \texttt{error}(E)\ cb\ \texttt{finally}\, e_2$ where $E \neq E_1$.
The case is immediate.

*Case:* R-NOCATCH: $\sigma \vdash \texttt{try}\, \texttt{error}(E)\, \texttt{finally}\, e \to \texttt{let}\ x'{:}\, \texttt{Top}{=}\, e\ \texttt{in}\ \texttt{error}(E)$
We are given $\sigma;\Gamma \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{finally}\, e : T'$ for some type $T'$. Inverting rule T-TCF $\sigma;\Gamma \vdash e : T'$. Assuming that $e$ does not equal $\texttt{new}\,\texttt{L}$ or $v.f$, the result follows by rule T-LET-TOP and T-ERROR

$$\frac{\sigma;\Gamma \vdash e :\ T' \qquad \sigma;\Gamma, x'{:}\ \texttt{Top} \vdash \texttt{error}(E) : T'}{\sigma;\Gamma \vdash \texttt{let}\ x'{:}\ \texttt{Top}{=}\ e\ \texttt{in}\, \texttt{error}(E) : T'}\ \text{T-LET-TOP}$$

which concludes the case. The cases where $e$ is a lock creation or a field access are treated by the rule T-NEWL-TOP and T-FIELD-TOP correspondingly. □

The next lemma is an extension of the local subject reduction Lemma 16 to the setting with exceptions.

**Lemma 23** (Subject reduction)**.** *Assume* $\sigma \vdash ok$ *(i.e.,* $\sigma$ *is well-formed) and* $t$ *is well-typed with* $\sigma$*, i.e.,* $\Gamma;\sigma \vdash t : T$ *where* $\Gamma$ *is empty and for some type* $T$*. Assume further* $\Delta_1 \vdash t :: \Delta_2, \Omega_2 \& v$ *where* $\Delta_1 = \sigma \downarrow_p$ *for a thread identifier* $p$*. If* $\sigma \vdash t \to \sigma' \vdash t'$*, then* $\Delta_1' \vdash t' :: \Delta_2', \Omega_2 \& v$*, with* $\Delta_1' = \sigma' \downarrow_p$ *and with* $\Delta_2' = \Delta_2$.

*Proof.* Proceed by induction on the derivation of typing judgment.

*Case:* R-THROW: $\sigma \vdash \texttt{let}\ x{:}T = \texttt{throw}\ E\ \texttt{in}\ t \rightarrow \sigma \vdash \texttt{let}\ x{:}T = \ \texttt{error}(E)\ \texttt{in}\ t$
Straightforward, as the type rules for $\texttt{throw}\ E$ and $\texttt{error}(E)$ are identical.

*Case:* R-TRY$_1$: $\sigma \vdash \texttt{try}\ e_1 cb\ \texttt{finally}\ e_2 \rightarrow \sigma' \vdash \texttt{try}\ e'_1 cb\ \texttt{finally}\ e_2$
where $\sigma \vdash e_1 \rightarrow \sigma' \vdash e'_1$. We are given $\Delta_1 \vdash \texttt{try}\ e_1 cb\ \texttt{finally}\ e_2 :: \Delta_3, \Omega_3 \& v$. Inverting instances of subsumption and rule T-TCF gives $\Delta_1 \vdash e_1 :: \Delta_2, \Omega'$ for some $\Omega' \leq \Omega$. By induction we get $\Delta'_1 \vdash e'_1 :: \Delta'_2, \Omega'$ where $\Delta'_1 = \sigma' \downarrow_p$ and $\Delta'_2 = \Delta_2$. By subsumption, also $\Delta'_1 \vdash e'_1 :: \Delta'_2, \Omega$ and hence the result follows by rule T-TCF (omitting unchanged premises):

$$\frac{\Delta'_1 \vdash e'_1 :: \Delta_2, \Omega \qquad \ldots \qquad \Delta_2 \vdash e_2 :: \Delta_3, \Omega_3 \& v}{\Delta'_1 \vdash \texttt{try}\ e'_1 cb\ \texttt{finally}\ e_2 :: \Delta_3, \Omega_3 \& v}$$

*Case:* R-TRY$_2$: $\sigma \vdash \texttt{try}\ v\ cb\ \texttt{finally}\ e_2 \rightarrow \sigma \vdash\ e_2$
The well-typedness assumption $\Delta_1 \vdash \texttt{try}\ v\ cb\ \texttt{finally}\ e_2 :: \Delta_2, \Omega :: v$ and inverting instances of subsumption and T-TCF gives

$$\Delta'' \vdash e_2 : \Delta_2, \Omega''' \& v' \tag{35}$$

as judgment for $e_2$ and furthermore for the value $v$, that $\Delta_1 \vdash v :: \Delta_1$. Furthermore, $\Omega = \Omega''' + \tilde{\Omega}$ for some $\tilde{\Omega}$. By the second premise of T-TCF, $\Delta_1 = \Delta''$. The judgment (35) this equals $\Delta_1 \vdash e_2 :: \Delta_2, \Omega''' \& v$, whence the required $\Delta_1 \vdash e_2 :: \Delta_2, \Omega''' + \tilde{\Omega} \& v$ follows by subsumption, using the fact $\Omega''' \leq \Omega''' + \tilde{\Omega}$.

*Case:* R-CATCH: $\sigma \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{catch}\ E > e_1; cb\ \ \texttt{finally}\ e_2 \rightarrow \sigma \vdash \texttt{try}\ e_1\ \texttt{finally}\ e_2$
The assumption $\Delta_1 \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{catch}\ E > e_1; cb\ \texttt{finally}\ e_2 :: \Delta_2, \Omega :: v$ and inverting instances of subsumption and T-TCF gives

$$\frac{\begin{array}{c}\sigma, \Delta_1 \vdash \texttt{error}(E) :: \Delta', \Omega \qquad \Omega(E) = \Delta_1 \qquad \sigma; \Delta_1 \vdash e_1 :: \Delta'_1, \Omega_1 \\ \Delta'_1 = \Delta'' \qquad \sigma; \Delta'' \vdash e_2 :: \Delta''', \Omega''' \& v \qquad \ldots\end{array}}{\sigma; \Delta_1 \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{catch}\ E > e_1; cb\ \ \texttt{finally}\ e_2 :: \Delta'''; \Omega''' \& v}$$

where $\Omega''' \leq \Omega$. Note that $\Delta'_1 = \Delta''$ is required by the premises of T-TCF since $E \in \Omega$. The result follows then by rule T-TCF and subsumption.

*Case:* R-NEXTCATCH: $\sigma \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{catch}\ E_1 > e_1; cb\ \texttt{finally}\ e_2 \rightarrow \sigma \vdash \texttt{try}\ \texttt{error}(E)\ cb\ \ \texttt{finally}\ e_2$ where $E \neq E_1$.
The case is immediate: the premises for rule T-TCF for the expression after the step is a subset of the premises for the expression before the step (the premise for $e_1$ is missing).

*Case:* R-NoCatch: $\sigma \vdash$`try error`$(E)$ `finally` $e \to \sigma \vdash$`let` $x'$: `Top`$=$ $e$ `in` `error`$(E)$
We are given $\Delta_1 \vdash$ `try error`$(E)$ `finally` $e :: \Delta''', \Omega[\&v]$. Inverting instances of subsumption and rule T-TCF gives

$$\frac{\Delta_1 \vdash \texttt{error}(E) :: \Delta_1, E(\Delta_1) \quad \Delta_1 = \Delta'' \quad \Delta'' \vdash e :: \ \Delta''',\ \Omega''' \ \&v \quad FE(\Delta'', \Delta''', v)}{\Delta_1 \vdash \texttt{try}\ \texttt{error}(E)\ \texttt{finally}\ e :: \Delta''', (\Omega''' + E(\Delta_1)[E \mapsto \Delta'''])}$$

where $(\Omega''' + E(\Delta_1)[E \mapsto \Delta''']) \leq \Omega$. Note that because the whole expression results in at least one exception, not in a normal termination, the evaluation will not produce a result value. Hence the $v$ from the assumption is absent. If $e \notin \{\texttt{new}\,\texttt{L}, v.f\}$, rules T-Let-Top and T-Error yield:

$$\frac{\Delta_1 \vdash e :: \Delta''',\ \Omega''' \ \&v \quad \Delta'''' = \Delta''', x{:}0 \quad \begin{array}{c} FE(\Delta_1, \Delta''', v) \\ (\Delta'''' \vdash \texttt{error}(E) :: \Delta'''',\ E(\Delta''''))[v/x'] \end{array}}{\Delta_1 \vdash \texttt{let}\ x'{:}\ \texttt{Top}{=}\ e\ \texttt{in}\ \texttt{error}(E) ::\ \Delta''''[v/x'],\ (\Omega''' + E(\Delta''''[v/x']))}$$

Note that $x'$ does not occur in `error`$(E)$, and that $(\Delta''', x'{:}0)[v/x'] = \Delta'''$. The result then follows by subsumption. ☐

We subject reduction in place, the two Lemmas 19 and 20 carry over to the setting with exceptions. The following lemma corresponds to Lemma 18, stating that a well-typed program does not immediately go into an "erroneous" state. The lemma expresses that in guaranteeing that never a lock-exception is thrown. If we would have taken over the formulation of Lemma 18 unchanged (apart from replacing `error` by `error`$(\texttt{E}_{unlock})$), Lemma 24 would explain a slightly weaker property, namely that lock exceptions may be thrown, but the program will not end with an uncaught lock exception. In the lemma, the notation $t[t_1]$ stands for thread $t$ containing an occurrence of $t_1$, and by $t[t_1] \to t[t_2]$ means, that the occurence of $t_1$ in $t$ is the redex in the reduction step.

**Lemma 24.** *Let* $P = P' \parallel p\langle t\rangle$. *If* $\sigma \vdash P : ok$ *then* $\sigma \vdash P' \parallel p\langle t[\texttt{let}\ x{:}T = l.\ \texttt{unlock}\ \texttt{in}\ t']\rangle \not\Longrightarrow \sigma \vdash P' \parallel p\langle t[\texttt{let}\ x{:}T = \texttt{throw}\ \texttt{E}_{unlock}\ \texttt{in}\ t']\rangle$

*Proof.* Analogous to the proof of Lemma 18. ☐

**Theorem 25** (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* $\bullet \vdash P_0 \longrightarrow^* \sigma' \vdash P' \parallel p\langle t[\texttt{let}\ x{:}T = \texttt{throw}\ \texttt{E}_{unlock}\ \texttt{in}\ t']\rangle$.

*Proof.* A direct consequence of subject reduction and Lemma 24. ☐

**Theorem 26** (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.*

*Proof.* A consequence of subject reduction. □

## 7. 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 [20], 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 [19] for a more thorough discussion of the differences. Compared to the earlier conference contribution [17], the formal treatment here covers exceptional behavior, the full formal description of the type and effect system, and the proofs.

There are many type systems and formal analyses to catch already at 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 [11] 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, [11] considers type *inference* (or type reconstruction). Their language, a variant of the $\lambda$-calculus, however, is sequential. [25] 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. Unlike our language, [25]'s calculus does not cover exceptions. 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 avoid releasing free locks or locks not owned, and furthermore, to not leave any locks hanging. The language of [25] 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 [25] covers only binary locks. The same restriction applies to [26], which represents a type system assuring race-freedom. Gerakios et.al. [10] present a uniform treatment of region-based management and locks in a low-level language. A type and effect system guarantees the absence of memory access violations and data races in the presence of region aliasing. Re-entrant locks there are 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"). The paper, however, does not cover exceptional control flow. [9] 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 [5] uses a type-based approach to ensure safe use of resources (for C-programs). Furthermore the Rcc/Java type system tries to keep track of which locks are held (in an approximate manner), noting which field is guarded by which lock, and which locks must be held when calling a method. *Safe lock* analysis, supported e.g. by the Indus tool [22][13] as part of Bandera, is a static analysis that checks whether a lock is held indefinitely (in the context of multi-threaded Java). Laneve et. al. [4] [18] 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 byte-code 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. The type system covers, however, exceptional behavior. Extending [24], Iwama and Kobayashi [15] 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. Furthermore, they consider type *inference*, but unlike our system, they cannot deal with method calls, i.e., the system analyses method bodies in isolation.

Deviating from the standard evaluation, exceptional program behavior and (potential) exceptions are a common form of "effects" of a program, of methods, etc. In the context of Java, the operational semantics of exceptions has be formalized in various works: Based on a operational behavior and on a static type system, many works prove type soundness of a Java(-like calculi) in the presence of exceptions. Cf. e.g. [6] [7] [8] [2]. [23] present a type and effect system for a variant of *FJ* with exceptions, calculating *history effects*, i.e., describing the behavior of the program. The analysis there, however, does not consider finally-clauses, which also means it ignores the situation where a thrown but uncaught exception is "forgotten" by throwing another exception, namely one thrown in the finally-clause. [3] presents a semantical study for an effect analysis which keeps track of exceptions (and divergence) in a higher-order language. Conceptually close to the work presented here is the analysis in [14]: for a higher-order sequential calculus, the work provides a static type and effect system for *resource* analysis (and extending [11]). The language in particular features *exceptions* but neither supports concurrency nor mutable store, so aliasing or interference are no issues there.

## 8. Conclusion

We presented a static type and effect system to prevent certain errors for non-lexical lock handling as in Java 5 and considering exceptions. 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. Keeping track of the lock counters is further complicated by the non-local control flow caused by exceptions.

*Aliasing.* 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 (at least between different threads to synchronize shared data), one would expect that a static analysis on lock-usage relies on some form of alias analysis. Interestingly, aliasing can be elegantly dealt with in our system and 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 is 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. 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.

*Concurrency.* 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 5. 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 improper use of syntactical constructs for, e.g., opening/closing files, allocating/deallocating resources, with non-lexical scope. Furthermore we plan to implement the system for empirical results. The combination of our two type and effect systems, one for TFJ [20] 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 which helped to improve the paper.

## References

[1] T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.

[2] D. Ancona, G. Lagorio, and E. Zucca. A core calculus for Java exceptions. *SIGPLAN Notices*, 36:16–30, Oct. 2001.

[3] N. Benton and P. Buchlovsky. Semantics of an effect analysis for exceptions. In *Proceedings of the 2007 ACM SIGPLAN international workshop on Types in languages*

*design and implementation*, TLDI '07, pages 15–26, New York, NY, USA, 2007. ACM.

[4] 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.

[5] 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.

[6] S. Drossopoulou and S. Eisenbach. Describing the semantics of Java and proving type soundness. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *Lecture Notes in Computer Science State-of-the-Art-Survey*, pages 41–82. Springer-Verlag, 1999.

[7] S. Drossopoulou and T. Valkevych. Java exceptions throw no surprises. Technical report, Dept. of Computing, Imperial College of Science, London, 2000.

[8] S. Drossopoulou, T. Valkevych, and S. Eisenbach. Java type soundness revisited. Technical report, Dept. of Computing, Imperial College of Science, London, Mar. 2000.

[9] 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.

[10] 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.

[11] A. Igarashi and N. Kobayashi. Resource usage analysis. *ACM Transactions on Programming Languages and Systems*, 27(2):264–313, 2005.

[12] 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*.

[13] `http://indus.projects.cis.ksu.edu/`, 2010.

[14] F. Iwama, A. Igarashi, and N. Kobayashi. Resource usage analysis for a functional language with exceptions. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipumlation (PEPM)*, 2006.

[15] 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.

[16] S. Jagannathan, J. Vitek, A. Welc, and A. Hosking. A transactional object calculus. *Science of Computer Programming*, 57(2):164–186, Aug. 2005.

[17] E. B. Johnsen, T. Mai Thuong Tran, O. Owe, and M. Steffen. Safe locking for multi-threaded Java. In *Proceedings of FSEN 2011*, Lecture Notes in Computer Science. Springer-Verlag, Apr. 2011. 16 pages, accepted for publication. A longer version appeared as Technical Report 402, University of Oslo, Dept. of Computer Science, Jan. 2011.

[18] C. Laneve. A type system for JVM threads. *Theoretical Computer Science*, 290(1):741 – 778, 2003.

[19] 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.

[20] 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.

[21] S. Oaks and H. Wong. *Java Threads*. O'Reilly, Third edition, Sept. 2004.

[22] V. P. Ranganath and J. Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *International Journal of Software Tools and Technology Transfer*, 9(5):489–504, 2007.

[23] C. Skalka, S. Smith, and D. van Horn. A type and effect system for flexible abstract interpretation of java (extended abstract). *Electronic Notes in Theoretical Computer Science*, 131:111–124, 2005.

[24] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[25] 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.

[26] 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.