UNIVERSITY OF OSLO
Department of Informatics

Deadlock checking by a behavioral effect system for lock handling

March 2011

Research Report No. 404

Ka I Pun, Martin Steffen, and Volker Stolz

ISBN 82-7368-366-4
ISSN 0806-30360806

March 2011

Deadlock checking by a behavioral effect system for lock handling[⋆]

March 2011

Ka I Pun[1], Martin Steffen[1], and Volker Stolz[1,2]

[1] University of Oslo, Norway
[2] United Nations University—Intl. Inst. for Software Technology, Macao

Abstract. Deadlocks are a common error in programs with lock-based concurrency and are hard to avoid or even to detect. One way for deadlock prevention is to statically analyze the program code to spot sources of potential deadlocks. Often static approaches try to confirm that the lock-taking adheres to a given order, or, better, to infer that such an order exists. Such an order precludes situations of cyclic waiting for each other's resources, which constitute a deadlock.
In contrast, we do not enforce or infer an explicit order on locks. Instead we use a *behavioral* type and effect system that, in a first stage, checks the behavior of each thread or process against the declared behavior, which captures potential interaction of the thread with the locks. In a second step on a global level, the state space of the behavior is explored to detect potential deadlocks. We define a notion of *deadlock-sensitive simulation* to prove the soundness of the abstraction inherent in the behavioral description. Soundness of the effect system is proven by subject reduction, formulated such that it captures deadlock-sensitive simulation. To render the state-space finite, we show two further abstractions of the behavior sound, namely restricting the upper bound on re-entrant lock counters, and similarly by abstracting the (in general context-free) behavioral effect into a coarser, tail-recursive description. We prove our analysis sound using a simple, concurrent calculus with re-entrant locks.

1 Introduction

Deadlock is a well-known problem for concurrent programs, where multiple processes share access to mutually exclusive resources. According to Coffman [11], there are four necessary conditions for a deadlock to occur, namely, mutual exclusion, no-preemption, wait-for condition, and *circular wait*. The first three are typically programming language specific; whether or not a deadlock occurs in one particular run of one particular program depends on whether the running program reaches a configuration, in which a number of processes wait for resources held by the others in a circular chain. Whenever concurrent activities attempt to acquire more than one lock, there is a potential for deadlocks. Since the actual occurrence of a deadlock depends on the actual scheduling

[⋆] Partly funded by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (`http://www.hats-project.eu`) and the ARV grant of the Macao Science and Technology Development Fund.

at run-time, deadlocks may occur only intermittently, making them difficult to debug. Preventing deadlocks at compile time altogether, on the other hand, must necessarily over-approximate the actual executions of the program, as the question of whether a program may deadlock or not is undecidable. The over-approximation may report spurious deadlocks, i.e., deadlocks reported on the abstract level do not reflect actual deadlocks on the concrete execution.

Apart from using run-time monitoring for deadlock detection, a number of static methods to assure deadlock freedom have been proposed [6,1,15,38]. In this paper, we detect potential deadlocks *statically* by capturing the lock interaction of processes by a behavioral type and *effect* system [33,2]. While type systems assure proper use of values, effect systems capture phenomena, which occur during evaluation, such as exceptions, side-effects, resource usage, etc. Expressive effects can deal with *behavior* of a program, which is important for concurrent or parallel programs. We, in particular, use a behavioral effect system to detect potential *deadlocks* in a setting with re-entrant locks. Locks are commonly used among processes to ensure mutual access to shared resources in concurrent programming. The effect system characterizes the behavior of a concurrent program in terms of sequences of lock interactions among parallel threads. By executing the abstraction of the actual behavior, we detect cycles of processes waiting for shared locks as a symptom of a deadlock.

In this article, we use the well-known characterization of cyclic wait to detect deadlocks on an abstract model of the program behavior. The effect system focuses on primitives for lock manipulations and primitives for creating threads and locks. The analysis of the abstract behavior must consider different interleavings of the threads, which quickly leads to an explosion of the state space. On top of that, typically the number of threads and locks is potentially unbounded, leading to an infinite state space. To keep the state space finite, we limit ourselves to a finite amount of resources (threads and locks). While this rules out two major sources of infinity in the behavior, we still need to tackle infinite executions through recursion. We bound non-tail recursive function calls, and put an upper limit on lock counters, which keep track of how often a re-entrant thread has locked a resource. This gives an upper limit on the state space size at the cost of further approximation. The results are formalized for a core calculus supporting functions, multi-threading concurrency, and re-entrant locks.

In Section 2, we present the syntax and semantics of our calculus. A type- and effect system that checks a concrete program, and produces the finite description of the abstract behavior of the program is presented in Section 3. We show the correctness of the abstraction into a *finite* state space in Section 4, and conclude in Section 5.

2 A calculus for lock-based concurrency

Before defining syntax and operational semantics of our calculus, we illustrate deadlocks in a simple example using the Java language. Concentrating on the core aspects of concurrency and lock handling, the calculus later will be not introduce objects and classes; instead we base our study on a calculus based on threads, functions, and locks. The syntax will be given in Section 2.1, the operational semantics in Section 2.2, and a characterization of deadlocks as cyclic wait in Section 2.3.

We motivate our analysis with a slightly abridged textbook example from the Java tutorials [23] on concurrency.

Listing 1.1. Java concurrency example

```
class Friend {
        ...
        public synchronized void bow(Friend bower) {
        System.out.format("%s: %s has bowed to me!%n",
                    this.name, bower.getName());
            bower.bowBack(this);
        }
        public synchronized void bowBack(Friend bower) {
            System.out.format("%s: %s has bowed back to me!%n",
                    this.name, bower.getName());
       }

    public static void main(String[] args) {
        final Friend alphonse = new Friend("Alphonse");
        final Friend gaston = new Friend("Gaston");
        new Thread(new Runnable() {
            public void run() { alphonse.bow(gaston); }
        }).start();
        new Thread(new Runnable() {
            public void run() { gaston.bow(alphonse); }
        }).start();
    }
}
```

At run-time, we may observe the following deadlock: each of the two threads proceeds into its respective synchronized `bow`-method, locking one object in one thread each; `alphonse` will be locked by the first thread, and `gaston` respectively locked by the second one. The next instruction, `bowBack` is then invoked on the partner *with the current object locked*. `Alphonse` holds "his" lock, and attempts to acquire `gaston`'s lock for the `bowBack`. As `gaston` holds his own lock, `alphonse` is suspended until that lock is released. The converse is happening for `gaston`, who keeps his lock held, and is waiting for `alphonse` lock: each one is waiting for a lock that its partner holds, a "deadly embrace".

2.1 Syntax

The abstract syntax for a small concurrent calculus with functions, thread creation, and re-entrant locks is given in Table 1. A program P consists of a parallel composition of processes $p\langle t\rangle$, where p identifies the process and t is a thread, i.e., the code being executed. The empty program is denoted as $\emptyset$. As usual, we assume $\parallel$ to be associative and commutative, with $\emptyset$ as neutral element. As for the code we distinguish threads t and expressions e, where t basically is a sequential composition of expressions. The terminated thread is denoted by `stop`. Values are denoted by v, and `let` $x{:}T = e$ `in` t represents the sequential composition of e followed by t, where the eventual result of e, i.e., once evaluated to a value, is bound to the local variable x. Expressions, as said, are given by e, and threads are among possible expressions. Further expressions are function application $e_1\ e_2$, conditionals, and the spawning of a new thread, written `spawn` t. The last three expressions deal with lock handling: `new L` creates a new lock (initially free) and gives a reference to it (the `L` may be seen as a class for locks), and furthermore

v. `lock` and v. `unlock` acquires and releases a lock, respectively. Values, i.e., evaluated expressions, are variables, lock references, and function abstractions, where we use `fun` $f{:}T_1.x{:}T_2.t$ for recursive function definitions.

$P ::= \emptyset \mid p\langle t\rangle \mid P \parallel P$	program
$t ::=$ `stop`	stopped thread
$\mid v$	value
$\mid$ `let` $x{:}T = e$ `in` t	local variables and sequ. composition
$e ::= t$	thread
$\mid v\ v$	application
$\mid$ `if` e `then` e `else` e	conditional
$\mid$ `spawn` t	spawning a thread
$\mid$ `new L`	lock creation
$\mid v.$ `lock`	acquiring a lock
$\mid v.$ `unlock`	releasing a lock
$v ::= x$	variable
$\mid l$	lock reference
$\mid$ `fn` $x{:}T.t$	function abstraction
$\mid$ `fun` $f{:}T.x{:}T.t$	recursive function abstraction

Table 1. Abstract syntax

In our calculus we focus on the concurrency aspects and locks. The introductory example can be encoded by making the implicit locking through `synchronized` explicit, and use a lock per object. In addition, we assume the natural extension of function declarations to multiple arguments and we elide types for better readability:

```
let bowBack = fn (this , bower) . this.lock; /* skip */ this.unlock in
  let bow = fn (this , bower) . this.lock; bowBack (bower, this) in
    let alphonse = new L in
      let gaston = new L in
        spawn (bow (alphonse ,gaston)); spawn (bow (gaston ,alphonse))
```

2.2 Semantics

The small-step operational semantics given below is straightforward, where we distinguish between local and global steps (cf. Tables 2 and 3). The local level deals with execution steps of one single thread, where the steps specify reduction steps in the following form:

$$t \to t' \,. \tag{1}$$

Rule R-RED is the basic evaluation step, replacing in the continuation thread t the local variable by the value v. Rule R-LET restructures a nested let-construct. As the let-construct generalizes sequential composition, the rule expresses associativity of that construct. Ignoring the local variable definition, it corresponds to transforming $(e_1;t_1);t_2$ into $e_1;(t_1;t_2)$. Together with the other rule, which performs a case distinction of the first basic expression in a let construct, that assures a deterministic left-to-right evaluation within each thread. The two R-IF-rules cover the two branches of the conditional and the R-APP-rules deals with function application (of non-recursive, resp. recursive functions).

$\texttt{let}\, x{:}T = v\ \texttt{in}\, t \rightarrow t[v/x]$	R-RED
$\texttt{let}\, x_2{:}T_2 = (\texttt{let}\, x_1{:}T_1 = e_1\ \texttt{in}\, t_1)\ \texttt{in}\, t_2 \rightarrow \texttt{let}\, x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\, x_2{:}T_2 = t_1\ \texttt{in}\, t_2)$	R-LET
$\texttt{let}\, x{:}T = \texttt{if}\ \texttt{true}\ \texttt{then}\, e_1\ \texttt{else}\, e_1\ \texttt{in}\, t \rightarrow \texttt{let}\, x{:}T = e_1\ \texttt{in}\, t$	R-IF$_1$
$\texttt{let}\, x{:}T = \texttt{if}\ \texttt{false}\ \texttt{then}\, e_1\ \texttt{else}\, e_2\ \texttt{in}\, t \rightarrow \texttt{let}\, x{:}T = e_2\ \texttt{in}\, t$	R-IF$_2$
$\texttt{let}\, x{:}T = (\texttt{fn}\, x'{:}T'.t')\ v\ \texttt{in}\, t \rightarrow \texttt{let}\, x{:}T = t'[v/x']\ \texttt{in}\, t$	R-APP$_1$
$\texttt{let}\, x{:}T = (\texttt{fun}\, f.x'.t')\ v\ \texttt{in}\, t \rightarrow \texttt{let}\, x{:}T = t'[v/x'][\texttt{fun}\, f.x'.t'/f]\ \texttt{in}\, t$	R-APP$_2$

Table 2. Local steps

The global steps are given in Table 3, formalizing transitions of configurations of the form $\sigma \vdash P$, i.e., the steps are of the form

$$\sigma \vdash P \rightarrow \sigma' \vdash P' \ , \tag{2}$$

where P is a program, i.e., the parallel composition of a finite number of threads running in parallel, and σ contains the *locks*, i.e., it is a finite mapping from lock identifiers to the status of each lock (which can be either free or taken by a thread where a natural number indicates how often a thread as acquired the lock, modelling re-entrance). A thread-local step is lifted to the global level by R-LIFT. Rule R-PAR specifies that the steps of a program consist of the steps of the individual threads, sharing σ. Executing the spawn-expression creates a new thread with a new identity which runs in parallel with the parent thread (cf. rule R-SPAWN). A new lock is created by `new L` (cf. rule R-NEWL) which allocates a fresh lock reference in the heap. Initially, the lock is free. A lock l is acquired by executing $l.$`lock`. There are two situations where that command does not block, namely the lock is free or it is already held by the requesting process p. The heap update $\sigma + l_p$ is defined as follows: If $\sigma(l) = \mathit{free}$, then $\sigma + l_p = \sigma[l \mapsto p(1)]$ and if $\sigma(l) = p(n)$, then $\sigma + l_p = \sigma[l \mapsto p(n+1)]$. Dually $\sigma - l_p$ is defined as follows: if $\sigma(l) = p(n+1)$, then $\sigma - l_p = \sigma[l \mapsto p(n)]$, and if $\sigma(l) = p(1)$, then $\sigma - l_p = \sigma[l \mapsto \mathit{free}]$. Unlocking works correspondingly, i.e., it sets the lock as being free resp. decreases the lock count by one (cf. rule R-UNLOCK). In the premise of the rules it is checked that the thread performing the unlocking actually holds the lock.

$$\frac{t_1 \to t_2}{\sigma \vdash p\langle t_1\rangle \to \sigma \vdash p\langle t_2\rangle}\ \text{R-LIFT} \qquad \frac{\sigma \vdash P_1 \to \sigma' \vdash P_1'}{\sigma \vdash P_1 \parallel P_2 \to \sigma' \vdash P_1' \parallel P_2}\ \text{R-PAR}$$

$$\sigma \vdash p_1\langle \texttt{let}\ x{:}T = \texttt{spawn}\ t_2\ \texttt{in}\ t_1\rangle \to \sigma \vdash p_1\langle \texttt{let}\ x{:}T = p_2\ \texttt{in}\ t_1\rangle \parallel p_2\langle t_2\rangle \qquad \text{R-SPAWN}$$

$$\frac{\sigma' = \sigma[l \mapsto \mathit{free}] \qquad l \text{ is fresh}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{new}\ L\ \texttt{in}\ t\rangle \to \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-NEWL}$$

$$\frac{\sigma(l) = \mathit{free} \vee \sigma(l) = p(n) \qquad \sigma' = \sigma + l_p}{\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\,\texttt{lock}\ \texttt{in}\ t\rangle \to \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-LOCK}$$

$$\frac{\sigma(l) = p(n) \qquad \sigma' = \sigma - l_p}{\sigma \vdash p\langle \texttt{let}\ x{:}T = l.\,\texttt{unlock}\ \texttt{in}\ t\rangle \to \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-UNLOCK}$$

Table 3. Global steps

2.3 Deadlocks

We can now characterize formally our deadlock criterion. First, we define what it means for a thread to be *waiting for* a lock, and then for a *program* to be deadlocked. See also [30] or [21] for an early discussion of different definitions of deadlock. Being deadlocked is a *global* property of a system in that it concerns more than one process. In our setting with re-entrant locks, a process cannot deadlock "on itself", and therefore at least two processes must be involved in a deadlock.

Later, to relate the operational behavior with its abstract behavioral description and to show correctness, it will be helpful to *label* the transitions of the operational semantics appropriately. Most importantly, lock-manipulating steps are labelled indicating which *lock* is being taken resp. released and by which *process*. We will discuss the exact nature of the labels in Section 3. For now, it is sufficient to consider only the label for process p taking lock l, written $\xrightarrow{p\langle \mathtt{L}^l\mathtt{lock}\rangle}$.

Definition 1 (Waiting for a lock). *Given a configuration $\sigma \vdash P$, a process p* waits *for a lock l in $\sigma \vdash P$, written as waits$(\sigma \vdash P, p, l)$, if it is not the case that $\sigma \vdash P \xrightarrow{p\langle \mathtt{L}^l\mathtt{lock}\rangle}$, and furthermore there exists a σ' s.t. $\sigma' \vdash P \xrightarrow{p\langle \mathtt{L}^l\mathtt{lock}\rangle} \sigma'' \vdash P'$.*

This indicates that process p is waiting for lock l to become available. Note that this does not yet indicate a deadlock, as the lock may be released by the process holding it. A configuration $\sigma \vdash P$ is deadlocked if it contains a number of processes each is holding a lock and trying to acquire the lock of the next process in a cyclic manner.

Definition 2 (Deadlock). *A configuration $\sigma \vdash P$* is deadlocked *if $\sigma(l_i) = p_i(n_i)$ and furthermore waits$(\sigma \vdash P, p_i, l_{i+_k 1})$ (where $k \geq 2$ and for all $0 \leq i \leq k-1$). The $+_k$ is meant as addition modulo k. A configuration $\sigma \vdash P$* contains a deadlock*, if, starting from $\sigma \vdash P$, a deadlocked configuration is reachable; otherwise the configuration is deadlock free.*

3 Type and effect system

In this section, we present the type and effect system, which is used to capture the behavior of a program. The behavior can then be executed using the abstract operational semantics. We show that each deadlock in the concrete behavior is preserved in the abstract behavior.

3.1 Annotations, effects, and types

The behavioral effects later capture lock interactions of a program. To specify which locks are meant statically, we label the program points of lock creations appropriately. We use π for program points, and annotations are given as sets of program points:

$$r ::= \{\pi\} \mid r \cup r \mid \emptyset \qquad \text{annotations} \tag{3}$$

We use this annotation to augment the syntax of Table 1 to keep track of locks, so all lock creation expressions `new L` are augmented to

$$\mathtt{new}_{\pi}\ \mathtt{L}\ . \tag{4}$$

For a given program, the annotations π are assumed unique. That assumption does not influence the soundness of the analysis, but the analysis gets more precise by not confusing different program points. The annotation does not influence the semantics (apart from the fact that we will label the transition relation of the operational semantics later as well).

The grammar for the types is given in Table 4. The underlying types are standard. We assume as basic types booleans and integers (`Bool` and `Int`). As far as the effects are concerned, two points are important. First, in the type system the type `L` for a lock must remember the potential places where the lock is created. Therefore, the effect type for lock references is written $\mathtt{L}^r$. The type of a thread is stated as `Thread`.

$$T ::= \mathtt{Bool} \mid \mathtt{Int} \mid T \xrightarrow{\varphi} T \mid \mathtt{L}^r \mid \mathtt{Thread}$$

Table 4. Types

The types of Table 4 carry two kinds of extra annotations on top of the underlying types, namely the annotation r on the lock types and *effects* φ as annotation on the functional types. Types of an expression describe the domain of values to which the expression eventually evaluates if it terminates. Effects in contrast are used to describe "phenomena" that happens during that evaluation. In our case, we capture interaction with locks, and in particular which locks are accessed during the execution and in which order: This means the effects capture *behavioral* information related to lock handling.

The type and effect judgments on the local level look as follows

$$\Gamma \vdash e : T :: \varphi\ , \tag{5}$$

meaning that expression e has type T and effect φ[3]. The contexts Γ contain type information for variables and lock references and are of the form $v_1{:}T_1, \ldots, v_n{:}T_n$, where the values v_i are either variables or lock references. We silently assume that all variables and references in Γ are different, and that the order does not matter. Thus, a context Γ is equivalently also seen as finite mappings and we use $dom(\Gamma)$ to refer to the domain of that mapping and $\Gamma(x)$ and $\Gamma(l)$ to look up the type remembered in Γ for x resp. for l. Furthermore, $\Gamma, v{:}T$ is the extension of Γ where we assume that v does not occur in Γ. Note that Γ does not bind an *effect* to variables resp. references. Effect information, however, is indirectly contained in the context, as functional types carry behavior information for the latent effect of functions in the type.

$\Phi ::= \mathbf{0} \mid p\langle\varphi\rangle \mid \Phi \parallel \Phi$	effects (global)
$\varphi ::= \varepsilon \mid X \mid \varphi;\varphi \mid \varphi + \varphi \mid rec X.\varphi \mid \alpha$	effects (local)
$a ::= \texttt{spawn}\ \varphi \mid \nu \mathrm{L}^r \mid \mathrm{L}^r\texttt{.lock} \mid \mathrm{L}^r\texttt{.unlock}$	labels/basic effects
$\alpha ::= a \mid \tau$	transition labels

Table 5. Effects

The grammar for the effects is given in Table 5. As for processes, we distinguish between a (thread-)local level φ and a global level Φ. The empty effect is written ε, representing behavior without interaction of locks. Recursive behavior is captured by $rec X.\varphi$, where the recursion operator binds variable X in φ. Sequential composition of φ_1 followed by φ_2 resp. non-deterministic choice between φ_1 and φ_2 are written $\varphi_1;\varphi_2$, resp. $\varphi_1 + \varphi_2$. Basic effects are captured by labels a, which can be one of four different forms: The effect `spawn` φ means that a new process with behavior φ is created, and $\nu\mathrm{L}^r$ indicates that a new lock is created at one of the program points in r. The effects L^r`.lock` and L^r`.unlock` describe the effect of acquiring a lock and releasing a lock, respectively, where again r denotes the potential places of creation. τ is used later to label silent transitions.

Example 1. Consider the following piece of code:

Listing 1.2. Deadlock

```
let x : L^π1 = new_π1 L in
let y : L^π2 = new_π2 L in
  spawn (y.lock;x.lock;stop);  x.lock;y.lock;stop
```

We use the semicolon as a shorthand for sequential composition as before instead of a let-construct. The example shows that after two locks have been created at two different locations π_1 and π_2, a new process is spawned such that both processes are running in parallel, sharing the two locks. These two processes try to take the two locks in reverse order. The situation right after spawning the second thread is depicted in Figure 1: the states correspond to the relevant control locations of each process, and the transitions

[3] In the abstract syntax, expressions e comprise threads t.

indicate the corresponding locking statements. When we consider possible interleavings of execution steps, we note that a deadlock occurs when both processes reach their respective intermediate state p_{11}/p_{21}: both will have acquired one lock, and are waiting on the opposite lock (see Figure 2). Not all interleavings are feasible: p_{11}/p_{22}, p_{12}/p_{22} are "shadowed" by the deadlock, and thus not reachable.

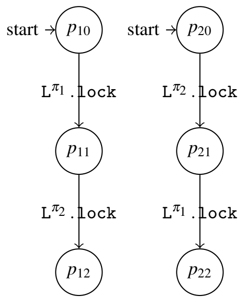

Fig. 1. Deadlock

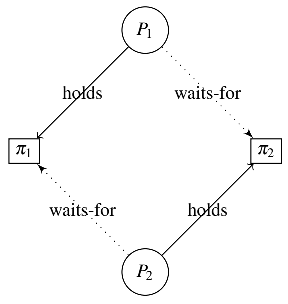

Fig. 2. Wait-for graph

□

3.2 Type system

The rules for the type and effect system for expressions, i.e., on the thread local level, are given in Table 6. The type of a variable is looked up from the typing context Γ and its effect is empty (cf. rule TE-VAR). Likewise, empty is the effect for lock references and thread references (cf. rules TE-LREF and TE-PREF). As a general rule, all values, especially abstractions, have no effect, as they cannot be evaluated any further. The terminated thread `stop` has an empty effect (cf. rule TE-STOP). In rule TE-IF, the two branches need to agree on a common type—see also the rule of subsumption—and the effect of a conditional is the non-deterministic choice between the effects of the two branches. Abstractions are values and consequently their effect is empty (cf. the TE-ABS rules). The effect of the body of the function, checked in the premise of the rule, is kept as annotation, i.e., as *latent* effect, on the arrow type of the abstraction in the conclusion of the rule. In the rule TE-APP, the effect of an application consists of the sequential composition of the effects of the function followed by the effect of the argument, followed by the effect of the function body, noted as annotation on the arrow of the function type, if one assumes a call-by-value evaluation from left to right. In our representation where we assume that the function as well are argument in an application are already evaluated (cf. the syntax of Table 1), it is assured that the effect of both abstraction and argument are empty and the overall effect consists of the latent effect of the function body only. The effect of the let-construct is expressed in rule TE-LET by sequencing effects of e and that of the body of the expression. Rule TE-SPAWN deals with the generation of a new thread executing the expression e. The type of this construct is `Thread`, while the effect is written as `spawn` φ which represents the behavior of the spawned thread. Rule TE-NEWL deals with the creation of a new lock, i.e., an "instance" of "class" `L`. In the annotated syntax, the creation expression is labelled by a (unique) program point π (cf. equation (4)). This point is remembered *both* in the type of that expression as well as in its effect. The type of a lock creation is $\mathtt{L}^{\pi}$ (which is a short-hand for $\mathtt{L}^{\{\pi\}}$). As for the effects, the expression has exactly *one* effect, namely the creation of a lock (at the indicated region r), is written as $\nu\mathtt{L}^{r}$ in the grammar of Table 5. As here we explicitly know the point π of creation, the effect is is more precisely $\nu\mathtt{L}^{\pi}$ (or $\nu\mathtt{L}^{\{\pi\}}$). Rules TE-LOCK and TE-UNLOCK for locking and unlocking an existing lock which has created at the indicated potential program points r. Both constructs are of the same type, namely $\mathtt{L}^{r}$; whereas the effects are $\mathtt{L}^{r}\mathtt{.lock}$ and $\mathtt{L}^{r}\mathtt{.unlock}$, respectively. The final one is the rule of subsumption. The corresponding sub-typing and sub-effecting relations are defined in Section 3.3.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T :: \varepsilon}\ \text{TE-VAR} \qquad \frac{}{\Gamma \vdash l^{\pi} : \mathtt{L}^{\pi} :: \varepsilon}\ \text{TE-LREF} \qquad \frac{\Gamma(p) = \mathtt{Thread}}{\Gamma \vdash p : \mathtt{Thread} :: \varepsilon}\ \text{TE-PREF}$$

$$\frac{}{\Gamma \vdash \mathtt{stop} : T :: \varepsilon}\ \text{TE-STOP} \qquad \frac{\Gamma \vdash v : \mathtt{Bool} \qquad \Gamma \vdash e_1 : T :: \varphi_1 \qquad \Gamma \vdash e_2 : T :: \varphi_2}{\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : T :: (\varphi_1 + \varphi_2)}\ \text{TE-IF}$$

$$\frac{\Gamma, x : T_1 \vdash e : T_2 :: \varphi}{\Gamma \vdash \mathtt{fn}\ x : T_1.e : T_1 \rightarrow^{\varphi} T_2 :: \varepsilon}\ \text{TE-ABS}_1 \qquad \frac{\Gamma, f{:}T_1 \rightarrow^{\varphi} T_2,\ x{:}T_1 \vdash t : T_2 :: \varphi}{\Gamma \vdash \mathtt{fun}\ f{:}T_1 \rightarrow^{\varphi} T_2.x{:}T_1.t : T_1 \rightarrow^{\varphi} T_2 :: \varepsilon}\ \text{TE-ABS}_2$$

$$\frac{\Gamma \vdash e_1 : T_2 \rightarrow^{\varphi} T_1 :: \varphi_1 \qquad \Gamma \vdash e_2 : T_2 :: \varphi_2}{\Gamma \vdash e_1\ e_2 : T_1 :: \varphi_1;\varphi_2;\varphi}\ \text{TE-APP} \qquad \frac{\Gamma \vdash e_1 : T_1 :: \varphi_1 \qquad \Gamma, x{:}T_1 \vdash e_2 : T_2 :: \varphi_2}{\Gamma \vdash \mathtt{let}\ x : T_1 = e_1\ \mathtt{in}\ e_2 : T_2 :: \varphi_1;\varphi_2}\ \text{TE-LET}$$

$$\frac{\Gamma \vdash e : T :: \varphi}{\Gamma \vdash \mathtt{spawn}\ e : \mathtt{Thread} :: \mathtt{spawn}\ \varphi}\ \text{TE-SPAWN} \qquad \Gamma \vdash \mathtt{new}_{\pi}\ \mathtt{L} : \mathtt{L}^{\pi} :: \nu\mathtt{L}^{\pi} \quad \text{TE-NEWL}$$

$$\frac{\Gamma \vdash v : \mathtt{L}^{r} :: \varphi}{\Gamma \vdash v.\ \mathtt{lock} : \mathtt{L}^{r} :: \varphi; \mathtt{L}^{r}\ .\ \mathtt{lock}}\ \text{TE-LOCK} \qquad \frac{\Gamma \vdash v : \mathtt{L}^{r} :: \varphi}{\Gamma \vdash v.\ \mathtt{unlock} : \mathtt{L}^{r} :: \varphi; \mathtt{L}^{r}\ .\mathtt{unlock}}\ \text{TE-UNLOCK}$$

$$\frac{\Gamma \vdash e : T' :: \varphi' \qquad T' \leq T \qquad \varphi' \leq \varphi}{\Gamma \vdash e : T :: \varphi}\ \text{TE-SUB}$$

Table 6. Type and effect checking (local)

Typing for the global level is shown in Table 7. An empty program, which does not have any effect, is well-typed *ok* defined by the rule TE-EMPTY. The rule TE-THREAD says a process p is well-typed if the thread t run by the process is also well-typed. Concurrent programs are well-typed if each one of them is so.

Example 2. We show the derivation of the behavior of Example 1 with the type and effect system we presented above. In the derivation, we leave out typing part except when needed (which is in using TE-LOCK) and concentrate on the effect part. In the derivation, let t abbreviate the code of Listing 1.2, and t_0 be $\texttt{spawn}\, t_1;t_1'$ where $t_1 \triangleq y.\,\texttt{lock};x.\,\texttt{lock};\texttt{stop}$ and $t_1' \triangleq x.\,\texttt{lock};y.\,\texttt{lock};\texttt{stop}$.

$$\frac{\Gamma_0 \vdash \texttt{new}_{\pi_1}\!:\!\texttt{L}^{\pi_1} :: \nu\texttt{L}^{\pi_1} \qquad \Gamma_0 \vdash \texttt{new}_{\pi_2}\!:\!\texttt{L}^{\pi_2} :: \nu\texttt{L}^{\pi_2} \qquad \Gamma_1 \vdash t_0 :: \varphi_0}{\Gamma_0 \vdash t :: \nu\texttt{L}^{\pi_1};\nu\texttt{L}^{\pi_2};\varphi_0}$$

Starting with the empty context Γ_0, the context Γ_1 is in the following form:

$$\Gamma_1 = x\colon \texttt{L}^{\pi_1}, y\colon \texttt{L}^{\pi_2}$$

We abbreviate the effect of t_1 as $\varphi_1 \triangleq \texttt{L}^{\pi_2}.\texttt{lock};\texttt{L}^{\pi_1}.\texttt{lock}$. We capture the effect of t_1' analogously except that the locks are taken in a reverse order, written as $\varphi_1' \triangleq \texttt{L}^{\pi_1}.\texttt{lock};\texttt{L}^{\pi_2}.\texttt{lock}$:

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma(y) = \texttt{L}^{\pi_2}}{\Gamma_1 \vdash y : \texttt{L}^{\pi_2} :: \varepsilon}}{\Gamma_1 \vdash y.\,\texttt{lock} :: \texttt{L}^{\pi_2}.\texttt{lock}} \qquad \dfrac{\dfrac{\Gamma(x) = \texttt{L}^{\pi_1}}{\Gamma_1 \vdash x : \texttt{L}^{\pi_1} :: \varepsilon}}{\Gamma_1 \vdash x.\,\texttt{lock} :: \texttt{L}^{\pi_1}.\texttt{lock}}}{\Gamma_1 \vdash t_1 :: \texttt{L}^{\pi_2}.\texttt{lock};\texttt{L}^{\pi_1}.\texttt{lock}} \qquad \Gamma_1 \vdash \texttt{stop} :: \varepsilon}{\Gamma_1 \vdash \texttt{spawn}\, t_1 :: \texttt{spawn}\, \varphi_1} \qquad \dfrac{\vdots}{\Gamma_1 \vdash t_1' :: \varphi_1'}}{\Gamma_1 \vdash t_0 :: \varphi_0}$$

Followings are the summary of abbreviations used in the derivation:

$$\begin{array}{rcl}
t & = & \texttt{let}\, x\colon \texttt{L}^{\pi_1} = \texttt{new}_{\pi_1}\texttt{L}\ \texttt{in}\ (\texttt{let}\, y\colon \texttt{L}^{\pi_2} = \texttt{new}_{\pi_2}\texttt{L}\ \texttt{in}\, t_0) \\
t_0 & = & \texttt{spawn}\ (t_1);t_1' \\
t_1 & = & y.\,\texttt{lock};x.\,\texttt{lock};\texttt{stop} \\
t_1' & = & x.\,\texttt{lock};y.\,\texttt{lock};\texttt{stop} \\
\hline
\Gamma_0 & = & () \\
\Gamma_1 & = & \Gamma_0, x\colon \texttt{L}^{\pi_1}, y\colon \texttt{L}^{\pi_2} \\
\hline
\varphi_0 & = & \texttt{spawn}\ \varphi_1;\varphi_1' \\
\varphi_1 & = & \texttt{L}^{\pi_2}.\texttt{lock};\texttt{L}^{\pi_1}.\texttt{lock} \\
\varphi_1' & = & \texttt{L}^{\pi_1}.\texttt{lock};\texttt{L}^{\pi_2}.\texttt{lock}
\end{array}$$

The overall effect is of Listing 1.2 is

$$t :: \nu\texttt{L}^{\pi_1};\nu\texttt{L}^{\pi_2};\texttt{spawn}\ (\texttt{L}^{\pi_2}.\texttt{lock};\texttt{L}^{\pi_1}.\texttt{lock});\texttt{L}^{\pi_1}.\texttt{lock};\texttt{L}^{\pi_2}.\texttt{lock}, \tag{6}$$

capturing the structure of the control flow in the concrete program. □

$$\frac{}{\vdash \emptyset : ok :: \varepsilon}\ \text{TE-EMPTY} \qquad \frac{() \vdash t : T :: \varphi}{\vdash p\langle t\rangle : ok :: p\langle\varphi\rangle}\ \text{TE-THREAD} \qquad \frac{\vdash P_1 : ok :: \Phi_1 \qquad \vdash P_2 : ok :: \Phi_2}{\vdash P_1 \parallel P_2 : ok :: \Phi_1 \parallel \Phi_2}\ \text{TE-PAR}$$

Table 7. Type and effect checking (global)

3.3 Ordering behavior

The behavior describes possible traces of an expression, over-approximating the actual behavior. There is a notion of *order* on such traces, with the usual intention that if an expression is approximated by behavior φ_1, and $\varphi_1 \leq \varphi_2$, then also φ_2 is a safe approximation of the expression. The order is called *sub-effecting* and is formalized in Table 9. Underlying the order on effects is an order on types, or, even more basic, the order on the sets r of annotations. For locks, the sets r contain potential program points where the lock may have been created. Thus, the smaller that set, the more precise the analysis, and a larger set still remains safe. That induces order $\leq$ on *types* ("subtyping") as given in the rules of Table 8. The order is reflexive by rule S-REFL. Rule S-ARROW acts, as usual, contra-variant on the left-hand side and co-variant on the right. As far as the annotation on the arrow is concerned, it is handled *co*-variantly. Finally, the subset order on annotation sets is lifted to lock types in rule S-LOCK: the larger the set of potential locations, the less information the type carries. It is straightforward to prove transitivity of subtyping.

$$T \leq T \quad \textsc{S-Refl} \qquad \frac{T_1' \leq T_1 \qquad T_2 \leq T_2' \qquad \varphi \leq \varphi'}{T_1 \rightarrow^{\varphi} T_2 \leq T_1' \rightarrow^{\varphi'} T_2'}\ \textsc{S-Arrow} \qquad \frac{r \subseteq r'}{\mathrm{L}^{r} \leq \mathrm{L}^{r'}}\ \textsc{S-Lock}$$

Table 8. Subtyping

Table 9 specifies an ordering on behavior, i.e., sub-effecting. That relation is not intended to capture the deadlock-sensitive simulation between configurations which we will study later; it is used for the formulation of the type and effect system. The relation $\leq$ is reflexive and transitive by rules SE-REFL and SE-TRANS. Rules SE-LOCK and SE-UNLOCK indicate that taking/releasing a lock from a lock-set may be approximated by choosing from a wider set of locks, in a similar manner as for S-LOCK. SE-CHOICE$_1$ expresses order of a behavior and a choice of that behavior and another behavior. The order of a behavior and itself followed by a sequence of behaviors is stated in SE-PREFIX. SE-CHOICE$_2$ allows us to widen the argument of a choice. Rules SE-SEQ, SE-SPAWN, and SE-REC describe the same equivalence for sequencing, spawning, recursion. Unfolding a recursion is represented by EE-REC. EE-UNIT and EE-ASSOC$_S$ express unit and associativity of a sequential operator. EE-CHOICE describes the equivalence of a behavior and the choice between the same behavior itself. The distributivity of sequencing respect to choice is stated by EE-DISTR. EE-COMM and EE-ASSOC$_C$ shows the commutativity and associativity of a choice.

3.4 Semantics of the behavior

Next we define the reduction steps of abstract behavior. In contrast to the operational semantics on the concrete level, σ is now a finite mapping from each lock *location* π to its corresponding status. The corresponding rules are given in Table 10. To formulate

$$rec X.\varphi \equiv \varphi[rec X.\varphi/X] \quad \textsc{EE-Rec}$$

$$\varepsilon;\varphi \equiv \varphi \quad \textsc{EE-Unit} \qquad \varphi_1;(\varphi_2;\varphi_3) \equiv (\varphi_1;\varphi_2);\varphi_3 \quad \textsc{EE-Assoc}_S$$

$$\varphi + \varphi \equiv \varphi \quad \textsc{EE-Choice} \qquad (\varphi_1 + \varphi_2);\varphi_3 \equiv \varphi_1;\varphi_3 + \varphi_2;\varphi_3 \quad \textsc{EE-Distr}$$

$$\varphi_1 + \varphi_2 \equiv \varphi_2 + \varphi_1 \quad \textsc{EE-Comm} \qquad \varphi_1 + (\varphi_2 + \varphi_3) \equiv (\varphi_1 + \varphi_2) + \varphi_3 \quad \textsc{EE-Assoc}_C$$

$$\frac{\varphi_1 \equiv \varphi_2}{\varphi_1 \le \varphi_2}\ \textsc{SE-Refl} \qquad \frac{\varphi_1 \le \varphi_2 \qquad \varphi_2 \le \varphi_3}{\varphi_1 \le \varphi_3}\ \textsc{SE-Trans}$$

$$\frac{r_1 \subseteq r_2}{L^{r_1}\texttt{.lock} \le L^{r_2}\texttt{.lock}}\ \textsc{SE-Lock} \qquad \frac{r_1 \subseteq r_2}{L^{r_1}\texttt{.unlock} \le L^{r_2}\texttt{.unlock}}\ \textsc{SE-Unlock}$$

$$\varphi_1 \le \varphi_1 + \varphi_2 \quad \textsc{SE-Choice}_1 \qquad \varphi_1 \le \varphi_1;\varphi_2 \quad \textsc{SE-Prefix}$$

$$\frac{\varphi_1 \le \varphi_1' \qquad \varphi_2 \le \varphi_2'}{\varphi_1 + \varphi_2 \le \varphi_1' + \varphi_2'}\ \textsc{SE-Choice}_2 \qquad \frac{\varphi_1 \le \varphi_1' \qquad \varphi_2 \le \varphi_2'}{\varphi_1;\varphi_2 \le \varphi_1';\varphi_2'}\ \textsc{SE-Seq} \qquad \frac{\varphi_1 \le \varphi_2}{\texttt{spawn}\ \varphi_1 \le \texttt{spawn}\ \varphi_2}\ \textsc{SE-Spawn}$$

$$\frac{\varphi_1 \le \varphi_2}{rec X.\varphi_1 \le rec X.\varphi_2}\ \textsc{SE-Rec}$$

Table 9. Subeffecting

later the connection between the concrete steps of the program and the abstract steps of the effects, we decorate both the old reduction relation in Tables 2 and 3 and the new one with the relevant lock interaction. This proceeds in the same manner as we already annotated the reduction for locking, which was needed to formalize a deadlock. This labelling does not change the operational behavior and is needed only for formulating the correctness result in a clean manner.

Each transition is labelled with one of the labels from Table 5, which capture the four possible visible steps we describe in the behavior: creating a lock, locking and unlocking, and finally creating a new process with a given behavior. Besides that, τ represents an internal, invisible step. We introduce additionally $\surd$ as label on a transition to indicate termination. It is intended as decorations for steps, only, not to be part of a behavior φ. As for programs, we distinguish between local and global behavior. On the global level, the identity p of the process is relevant, and the corresponding transitions are labelled by $p\langle a\rangle$ resp. $p\langle\alpha\rangle$ instead of a, resp. α to indicate which process does the step. In abuse of notation, we use a and α also to mark global steps, when not interested in the identity of p. The formalization of the labelled operational steps of behaviors is straightforward. The behavior is determined up to $\equiv$-equivalence and parallel components run in an interleaving manner (rules RE-MOD and RE-PAR). Sequential composition is given in rule RE-SEQ; not that for $\varepsilon;\varphi$, the empty effect can be discarded by $\varepsilon;\varphi \equiv \varphi$ from EE-UNIT. A thread which has terminated does as last action a $\surd$-step indicating termination.

$$\frac{\varphi_1 \equiv \varphi_1' \quad \sigma \vdash p\langle \varphi_1' \rangle \xrightarrow{\alpha} \sigma \vdash p\langle \varphi_2 \rangle}{\sigma \vdash p\langle \varphi_1 \rangle \xrightarrow{\alpha} \sigma \vdash p\langle \varphi_2 \rangle}\ \text{RE-MOD} \qquad \frac{\sigma \vdash \Phi_1 \xrightarrow{\alpha} \sigma' \vdash \Phi_1'}{\sigma \vdash \Phi_1 \parallel \Phi_2 \xrightarrow{\alpha} \sigma' \vdash \Phi_1' \parallel \Phi_2}\ \text{RE-PAR}$$

$$\frac{\sigma \vdash p\langle \varphi_1 \rangle \xrightarrow{\alpha} \sigma' \vdash p\langle \varphi_1' \rangle \qquad \alpha \neq \surd}{\sigma \vdash \langle \varphi_1 ; \varphi_2 \rangle \xrightarrow{\alpha} \sigma' \vdash p\langle \varphi_1' ; \varphi_2 \rangle}\ \text{RE-SEQ} \qquad \sigma \vdash p\langle \varepsilon \rangle \xrightarrow{p\langle \surd \rangle} \sigma \vdash \mathbf{0} \quad \text{RE-TICK}$$

$$\sigma \vdash p\langle \varphi_1 + \varphi_2 \rangle \xrightarrow{p\langle \tau \rangle} \sigma \vdash p\langle \varphi_1 \rangle \qquad \text{RE-CHOICE}$$

$$\sigma \vdash p_1\langle (\texttt{spawn}\ \varphi); \varphi' \rangle \xrightarrow{p\langle \texttt{spawn}\ \varphi \rangle} \sigma \vdash p_1\langle \varphi' \rangle \parallel p_2\langle \varphi \rangle \qquad \text{RE-SPAWN}$$

$$\frac{\sigma(\pi) = \bot \qquad \sigma' = \sigma[\pi \mapsto \mathit{free}]}{\sigma \vdash p\langle \nu \mathtt{L}^{\pi} \rangle \xrightarrow{p\langle \nu \mathtt{L}^{\pi} \rangle} \sigma' \vdash p\langle \varepsilon \rangle}\ \text{RE-NEWL}$$

$$\frac{\pi \in r}{\sigma \vdash p\langle \mathtt{L}^{r}\,.\,\texttt{lock} \rangle \xrightarrow{p\langle \tau \rangle} \sigma \vdash p\langle \mathtt{L}^{\pi}\,.\,\texttt{lock} \rangle}\ \text{RE-LOCK}_1 \qquad \frac{\sigma(\pi) = \mathit{free} \vee \sigma(\pi) = p(n) \qquad \sigma' = \sigma + \pi_p}{\sigma \vdash p\langle \mathtt{L}^{\pi}\,.\,\texttt{lock} \rangle \xrightarrow{p\langle \mathtt{L}^{\pi}.\texttt{lock} \rangle} \sigma' \vdash p\langle \varepsilon \rangle}\ \text{RE-LOCK}_2$$

$$\frac{\pi \in r}{\sigma \vdash p\langle \mathtt{L}^{r}\,.\,\texttt{unlock} \rangle \xrightarrow{p\langle \tau \rangle} \sigma \vdash p\langle \mathtt{L}^{\pi}\,.\,\texttt{unlock} \rangle}\ \text{RE-UNLOCK}_1 \qquad \frac{\sigma(\pi) = p(n) \quad n > 1 \quad \sigma' = \sigma - \pi_p}{\sigma \vdash p\langle \mathtt{L}^{\pi}\,.\,\texttt{unlock} \rangle \xrightarrow{p\langle \mathtt{L}^{\pi}.\texttt{unlock} \rangle} \sigma' \vdash p\langle \varepsilon \rangle}\ \text{RE-UNLOCK}_2$$

Table 10. Operational semantics for effects

A point concerning non-deterministic choice defined by rule RE-CHOICE deserves mentioning : to take the choice "costs" a τ-step. Since the effects are meant to over-approximate concrete program behavior especially wrt. deadlocking, it is important that the $+$-operator corresponds to an *internal* choice. The rule RE-SPAWN creates a new activity and works basically analogously to the thread creation at concrete level. The next five rules deal with effects concerning lock handling. Rule R-NEWL covers lock creation, captured by the effect $\nu\mathtt{L}^{\pi}$. The effect is caused by $\texttt{new}_{\pi}\ \mathtt{L}$ on the concrete level (cf. rule TE-NEWL from the type system), which means also on the abstract level, it is always *one* specific program point, and not a set r, where a lock is *created.* Unlike in the semantics on concrete level, not a new or fresh lock reference is created, but one statically fixed location π is used. The premise of RE-NEWL requires the location π has not been used previously for allocating a lock. By our restriction of not allowing lock creation in recursion, there is difference between the reduction rule R-NEWL and rule RE-NEWL: on the concrete level, the lock allocation step is *always* possible as there is an unbounded amount of fresh lock references available; whereas the effect $\nu\mathtt{L}^{\pi}$ in RE-NEWL here is not always executable.

The effect of taking a lock, $\mathtt{L}^{r}\,.\,\texttt{lock}$, is handled by the two RE-LOCK-rules. The abstraction may include uncertainty about at which location the lock in question comes from originally, i.e., r in general will be a *set* of candidate locations. Hence the lock-manipulating steps involve a non-deterministic choice which lock is affected. For the same reason that $+$ was interpreted as internal choice, the lock manipulation is done in two steps: first the choice of locks is specialized by picking one π from r by a τ-step (cf. RE-LOCK$_1$) and only afterwards the lock is taken in a second step with RE-LOCK$_2$. That means the choice which lock is actually attempted to be taken is made independent from the availability of the lock. The alternative formalization in one rule combining RE-LOCK$_1$ and RE-LOCK$_2$ into one atomic step would be unsound: a deadlock in the program may be missed in the abstract behavior description. Unlocking works dually. The notations $\sigma + \pi_p$ and $\sigma - \pi_p$ are used analogously (with π instead of l) as for the concrete heap. Note also the there is no specific rule for recursion; a recursive behavior can be unrolled by EE-REC from Table 9. The definition of simulation will later relate more concrete and more abstract effects, but also a program with its effect.

As for the semantics on the level of programs: as mentioned shortly earlier when characterizing processes waiting on a lock (Definition 1), the transitions of configurations $\sigma \vdash P$ are labelled, as well. So the steps of Tables 2 and 3 are considered labelled accordingly in the following. We assume further that the locks l are labelled by the point of creation, i.e., are of the form l^π. Due to our restriction on lock creation, that labelling is well-defined. So for instance, a lock-taking step of a lock l^π is of the form $\sigma \vdash P \xrightarrow{p\langle \mathtt{L}^\pi\mathtt{.lock}\rangle} \sigma' \vdash P'$, etc.

Example 3. To detect potential deadlock in our example, we execute the effect obtained in equation (6) of Example 2 in a process starting from the empty heap, using the abstract operational semantics from Table 10. We show the configuration consisting of the heap and parallel processes for the particular interleaving which ends up in the deadlocked configuration.

$$
\begin{array}{r}
[] \vdash p_1\langle \nu\mathtt{L}^{\pi_1};\nu\mathtt{L}^{\pi_2};\mathtt{spawn}\,(\mathtt{L}^{\pi_2}\,\mathtt{.lock};\mathtt{L}^{\pi_1}\,\mathtt{.lock});\mathtt{L}^{\pi_1}\,\mathtt{.lock};\mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle \xrightarrow{p_1\langle \nu\mathtt{L}^{\pi_1}\rangle} \\
[\pi_1 \mapsto \mathit{free}] \vdash p_1\langle \varepsilon;\nu\mathtt{L}^{\pi_2};\mathtt{spawn}\,(\mathtt{L}^{\pi_2}\,\mathtt{.lock};\mathtt{L}^{\pi_1}\,\mathtt{.lock});\mathtt{L}^{\pi_1}\,\mathtt{.lock};\mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle \xrightarrow{p_1\langle \nu\mathtt{L}^{\pi_2}\rangle} \\
[\pi_1 \mapsto \mathit{free}][\pi_2 \mapsto \mathit{free}] \vdash p_1\langle \varepsilon;\mathtt{spawn}\,(\mathtt{L}^{\pi_2}\,\mathtt{.lock};\mathtt{L}^{\pi_1}\,\mathtt{.lock});\mathtt{L}^{\pi_1}\,\mathtt{.lock};\mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle \xrightarrow{p_1\langle \mathtt{spawn}\,(\mathtt{L}^{\pi_2}\mathtt{.lock};\mathtt{L}^{\pi_1}\mathtt{.lock})\rangle} \\
[\pi_1 \mapsto \mathit{free}][\pi_2 \mapsto \mathit{free}] \vdash p_2\langle \mathtt{L}^{\pi_2}\,\mathtt{.lock};\mathtt{L}^{\pi_1}\,\mathtt{.lock}\rangle \parallel p_1\langle \mathtt{L}^{\pi_1}\,\mathtt{.lock};\mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle \xrightarrow{p_2\langle \mathtt{L}^{\pi_2}\mathtt{.lock}\rangle} \\
[\pi_1 \mapsto \mathit{free}][\pi_2 \mapsto p_2(1)] \vdash p_2\langle \mathtt{L}^{\pi_1}\,\mathtt{.lock}\rangle \parallel p_1\langle \mathtt{L}^{\pi_1}\,\mathtt{.lock};\mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle \xrightarrow{p_1\langle \mathtt{L}^{\pi_1}\mathtt{.lock}\rangle} \\
[\pi_1 \mapsto p_1(1)][\pi_2 \mapsto p_2(1)] \vdash p_2\langle \mathtt{L}^{\pi_1}\,\mathtt{.lock}\rangle \parallel p_1\langle \mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle
\end{array}
$$

The processes p_1 and p_2 reach the configuration with $\sigma = [\pi_1 \mapsto p_1(1)][\pi_2 \mapsto p_2(1)]$, $\sigma \vdash p_2\langle \mathtt{L}^{\pi_1}\,\mathtt{.lock}\rangle \parallel p_1\langle \mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle$, for which we have $\mathit{waits}(\sigma \vdash p_1\langle \mathtt{L}^{\pi_2}\,\mathtt{.lock}\rangle \parallel \ldots, p_1, \pi_2)$ and $\mathit{waits}(\sigma \vdash p_2\langle \mathtt{L}^{\pi_1}\,\mathtt{.lock}\rangle \parallel \ldots, p_2, \pi_1)$, satisfying our condition for a circular wait (cf. Definition 2). $\square$

In the following, we are going to use a few more examples to present the calculation of behavior with type and effect system.

Example 4 (Conditionals). Consider the following piece of code:

Listing 1.3. Conditional

```
let x = (if   b
          then new_π1
          else new_π2 )
in
    x.lock
```

The condition b is assumed to be a value of boolean type. This gives rise to the following derivation, where t represents the shown code fragment and $\Gamma_2 = \Gamma_1, x\colon \mathtt{L}^{\pi_1,\pi_2}$. To cover the two branches of the conditions, we must weaken the respective types $\mathtt{L}^{\pi_1}$ and $\mathtt{L}^{\pi_2}$ to a common $\mathtt{L}^{\pi_1,\pi_2}$ using subsumption.

$$\dfrac{\Gamma_1 \vdash b : \mathtt{Bool} \qquad \dfrac{\Gamma_1 \vdash \mathtt{new}_{\pi_1} : \mathtt{L}^{\pi_1} :: \nu\mathtt{L}^{\pi_1}}{\Gamma_1 \vdash \mathtt{new}_{\pi_1} : \mathtt{L}^{\pi_1,\pi_2} :: \nu\mathtt{L}^{\pi_1}} \qquad \dfrac{\Gamma_1 \vdash \mathtt{new}_{\pi_2} : \mathtt{L}^{\pi_2} :: \nu\mathtt{L}^{\pi_2}}{\Gamma_1 \vdash \mathtt{new}_{\pi_2} : \mathtt{L}^{\pi_1,\pi_2} :: \nu\mathtt{L}^{\pi_2}}}{\Gamma_1 \vdash \mathtt{if}\ c\ \mathtt{then}\ \mathtt{new}_{\pi_1}\ \mathtt{else}\ \mathtt{new}_{\pi_2} : \mathtt{L}^{\pi_1,\pi_2} :: \nu\mathtt{L}^{\pi_1} + \nu\mathtt{L}^{\pi_2}} \qquad \dfrac{\dfrac{\Gamma_2(x) : \mathtt{L}^{\pi_1,\pi_2}}{\Gamma_2 \vdash x : \mathtt{L}^{\pi_1,\pi_2} :: \varepsilon}}{\Gamma_2 \vdash x.\,\mathtt{lock} : \mathtt{L}^{\pi_1,\pi_2} :: \mathtt{L}^{\pi_1,\pi_2}.\mathtt{lock}}$$
$$\overline{\qquad \Gamma_1 \vdash t : \mathtt{L}^{\pi_1,\pi_2} :: (\nu\mathtt{L}^{\pi_1} + \nu\mathtt{L}^{\pi_2});\mathtt{L}^{\pi_1,\pi_2}.\mathtt{lock} \qquad}$$

Consider alternatively the following code:

Listing 1.4. Conditional

```
(if b then   let x = new_{\pi_1}  in x.lock
     else   let x = new_{\pi_2}  in x.lock)
```

This leads to the following derivation, where t' represents the above code and $t_1 \triangleq \mathtt{let}\ x = \mathtt{new}_{\pi_1}\ \mathtt{in}\ x.\,\mathtt{lock}$ and $t_2 \triangleq \mathtt{let}\ x = \mathtt{new}_{\pi_2}\ \mathtt{in}\ x.\,\mathtt{lock}$. Again, we have to use subsumption to reconcile the lock type for the two branches of the conditional.

$$\dfrac{\Gamma_1 \vdash b : \mathtt{Bool} \qquad \dfrac{\dfrac{\Gamma_1 \vdash \mathtt{new}_{\pi_1} : \mathtt{L}^{\pi_1} :: \nu\mathtt{L}^{\pi_1} \qquad \Gamma_2 \vdash x_1.\,\mathtt{lock} : \mathtt{L}^{\pi_1} :: \mathtt{L}^{\pi_1}.\mathtt{lock}}{\Gamma_1 \vdash t_1 : \mathtt{L}^{\pi_1} :: \nu\mathtt{L}^{\pi_1};\mathtt{L}^{\pi_1}.\mathtt{lock}}}{\Gamma_1 \vdash t_1 : \mathtt{L}^{\pi_1,\pi_2} :: \nu\mathtt{L}^{\pi_1};\mathtt{L}^{\pi_1}.\mathtt{lock}} \qquad \dfrac{\dfrac{\dots}{\Gamma_1 \vdash t_2 : \mathtt{L}^{\pi_2} :: \nu\mathtt{L}^{\pi_2};\mathtt{L}^{\pi_2}.\mathtt{lock}}}{\Gamma_1 \vdash t_2 : \mathtt{L}^{\pi_1,\pi_2} :: \nu\mathtt{L}^{\pi_2};\mathtt{L}^{\pi_2}.\mathtt{lock}}}{\Gamma_1 \vdash t' : \mathtt{L}^{\pi_1,\pi_2} :: \nu\mathtt{L}^{\pi_1};\mathtt{L}^{\pi_1}.\mathtt{lock} + \nu\mathtt{L}^{\pi_2};\mathtt{L}^{\pi_2}.\mathtt{lock}}$$

The two effects

$$(\nu\mathtt{L}^{\pi_1} + \nu\mathtt{L}^{\pi_2});\mathtt{L}^{\pi_1,\pi_2}.\mathtt{lock} \quad \text{vs} \quad \nu\mathtt{L}^{\pi_1};\mathtt{L}^{\pi_1}.\mathtt{lock} + \nu\mathtt{L}^{\pi_2};\mathtt{L}^{\pi_2}.\mathtt{lock} \tag{7}$$

reflect the different "branching structure" of the two programs. Clearly, in the second alternative of Listing 1.4, more information about which lock is actually used in the lock-operation is available. □

Example 5 (Behavior checking). This example revisits the previous Example 1, this time using functional abstraction. So instead of the code of Listing 1.2, consider the following:

Listing 1.5. Deadlock

```
let f : L^{\pi_2} × L^{\pi_1} →^{\varphi'} _ = fn (z_1:L^{\pi_2}, z_2:L^{\pi_1}). z_1.lock;z_2.lock;stop
in
   let x : L^{\pi_1} = new_{\pi_1} L in
   let y : L^{\pi_2} = new_{\pi_2} L in
   spawn (f (y,x)); x.lock;y.lock;stop
```

Note that the behavior-annotated functional type $\mathtt{L}^{\pi_2} \times \mathtt{L}^{\pi_1} \xrightarrow{\varphi'} \_$ is used to give a type to the variable f, representing the functional abstraction. We use _ as "wild card" represent any type. As for the behavior, $\varphi \triangleq \mathtt{L}^{\pi_1}.\mathtt{lock};\mathtt{L}^{\pi_2}.\mathtt{lock}$ and $\varphi' \triangleq \mathtt{L}^{\pi_2}.\mathtt{lock};\mathtt{L}^{\pi_1}.\mathtt{lock}$.

$$\dfrac{\dfrac{\dfrac{\Gamma_3 \vdash z_1.\,\texttt{lock}::\mathrm{L}^{\pi_2}\,.\texttt{lock} \quad \Gamma_3 \vdash z_1.\,\texttt{lock}::\mathrm{L}^{\pi_1}\,.\texttt{lock}}{\Gamma_3 \vdash t_f : \_ :: \varphi'}}{\Gamma_0 \vdash \texttt{fn}\,(z_1\!: \mathrm{L}^{\pi_2}, z_2\!: \mathrm{L}^{\pi_1}).t_f : (\mathrm{L}^{\pi_2} \times \mathrm{L}^{\pi_1}) \xrightarrow{\varphi'} \_ :: \varepsilon} \quad \dfrac{\Gamma_1 \vdash \texttt{new}_{\pi_i} :: \nu\mathrm{L}^{\pi_i} \quad \dfrac{(9)}{\Gamma_2 \vdash t_1 :: \texttt{spawn}\ \varphi';\varphi}}{\Gamma_1 \vdash t_1' : \_ :: \nu\mathrm{L}^{\pi_1};\nu\mathrm{L}^{\pi_2};\texttt{spawn}\ \varphi';\varphi}}{\Gamma_0 \vdash t_0 : \_ :: \varphi_0} \qquad (8)$$

The right-hand sub-tree for t_1'' continues with extending the context Γ_1 two times by the bindings for x and y, i.e., $\Gamma_2 = \Gamma_1, x : \mathrm{L}^{\pi_1}, y : \mathrm{L}^{\pi_2}$, and the derivation continues as follows:

$$\dfrac{\dfrac{(10)}{\Gamma_2 \vdash \texttt{spawn}\,(f(y,x)) :: \texttt{spawn}\ \varphi'} \quad \dfrac{\Gamma_2 \vdash x.\,\texttt{lock}::\mathrm{L}^{\pi_1}\,.\texttt{lock} \quad \Gamma_2 \vdash y.\,\texttt{lock}::\mathrm{L}^{\pi_2}\,.\texttt{lock}}{\Gamma_2 \vdash t :: \varphi}}{\Gamma_2 \vdash t_1 :: \texttt{spawn}\ \varphi';\varphi} \qquad (9)$$

Further, the left-hand subtree continues as follows:

$$\dfrac{\dfrac{\dfrac{\Gamma_2(f) = \mathrm{L}^{\pi_2} \times \mathrm{L}^{\pi_1} \xrightarrow{\varphi'} \_}{\Gamma_2 \vdash f : \mathrm{L}^{\pi_2} \times \mathrm{L}^{\pi_1} \xrightarrow{\varphi'} \_ :: \varepsilon}\ \text{TE-VAR} \quad \Gamma_2 \vdash y : \mathrm{L}^{\pi_2} :: \varepsilon \quad \Gamma_2 \vdash x : \mathrm{L}^{\pi_1} :: \varepsilon}{\Gamma_2 \vdash f(y,x) :: \varphi'}\ \text{TE-APP}}{\Gamma_2 \vdash \texttt{spawn}\,(f(y,x)) :: \texttt{spawn}\ \varphi'}\ \text{TE-SPAWN} \qquad (10)$$

The derivation trees uses the following abbreviations:

$$\begin{array}{rl}
t &= x.\,\texttt{lock};y.\,\texttt{lock};\texttt{stop} \\
t_1 &= \texttt{spawn}\,(f(y,x));t \\
t_1' &= \texttt{let}\,x\!: \mathrm{L}^{\pi_1} = \texttt{new}_{\pi_1}\mathrm{L}\ \texttt{in}\ (\texttt{let}\,y\!: \mathrm{L}^{\pi_2} = \texttt{new}_{\pi_2}\mathrm{L}\ \texttt{in}\ t_1) \\
t_f &= z_1.\,\texttt{lock};z_2.\,\texttt{lock};\texttt{stop} \\
t_0 &= \texttt{let}\,f : \mathrm{L}^{\pi_2} \times \mathrm{L}^{\pi_1} \xrightarrow{\varphi'} \_ = \texttt{fn}\,(z_1\!: \mathrm{L}^{\pi_2}, z_2\ \mathrm{L}^{\pi_1}).t_f\ \texttt{in}\ t_1' \\
\hline
\Gamma_3 &= \Gamma_0, z_1 : L^{\pi_1}, z_2 : L^{\pi_2} \\
\Gamma_2 &= \Gamma_1, x : \mathrm{L}^{\pi_1}, y : \mathrm{L}^{\pi_2} \\
\Gamma_1 &= \Gamma_0, f : \mathrm{L}^{\pi_1} \times \mathrm{L}^{\pi_1} \xrightarrow{\varphi'} \_ \\
\Gamma_0 &= () \\
\hline
\varphi_0 &= \nu\mathrm{L}^{\pi_1};\nu\mathrm{L}^{\pi_2};(\texttt{spawn}\ \varphi');\varphi \\
\varphi' &= \mathrm{L}^{\pi_2}\,.\texttt{lock};\mathrm{L}^{\pi_1}\,.\texttt{lock} \\
\varphi &= \mathrm{L}^{\pi_1}\,.\texttt{lock};\mathrm{L}^{\pi_2}\,.\texttt{lock}
\end{array} \qquad (11)$$

□

Example 6 (Behavior checking). The example revisits the previous Example 5, this time however, using the function f not once, but twice:

Listing 1.6. Behavior checking

```
let f = fn (z1,z2). z1.lock;z2.lock;stop
in
   let x  = new_π1 L in
   let y  = new_π2 L in
   spawn (f (y,x)); f (x,y); stop
```

The code does not contain the type and effect annotations. In the code of Listing 1.5, the function is used exactly once, namely as $f(y,x)$, where the lock in y is created at π_2 and the one of x created at π_2, hence the type of f had been $\mathtt{L}^{\pi_2} \times \mathtt{L}^{\pi_1} \xrightarrow{\varphi'} \_$. Now the function f is used two times, namely in $f(y,x)$ as before and further in $f(x,y)$. This means, the input types of the function must allow locks from both locations π_1 and π_2. Furthermore, the behavior φ' is now of the form $\mathtt{L}^{\pi_1,\pi_2}\mathtt{.lock};\mathtt{L}^{\pi_1,\pi_2}\mathtt{.lock}$. This gives the following code:

Listing 1.7. Behavior checking

```
let f :$L^{\pi_1,\pi_2} \times L^{\pi_1,\pi_2} \xrightarrow{\varphi'} \_$ = fn ($z_1$:$L^{\pi_1,\pi_2}$,$z_2$:$L^{\pi_1,\pi_2}$). $z_1$.lock;$z_2$.lock;stop
in
   let x :$L^{\pi_1}$ = new$_{\pi_1}$ L in
   let y :$L^{\pi_2}$ = new$_{\pi_2}$ L in
   spawn (f(x,y)); f(y,x);stop
```

That loss of precision is the price to pay of having a monomorphic type system.[4] □

3.5 Deadlock-sensitive simulation

Next we prove that the type and effect systems formalizes our intention in that the effect of a well-typed program *over-approximates* the actual behavior. Both the meaning of the effects and the meaning of the program are specified operationally. The proof of correctness relates therefore the operational interpretation on the concrete level of the program with that of the abstract level of behavior.

To do so we start by defining an appropriate notion of simulation [31], a definition which allows to transfer deadlock freedom from the simulating processes to the ones being simulated. The definition relates the behavior of two configurations, and as part of the definition, the corresponding heaps need to be appropriately related. As we will abstract lock counters, the heaps containing the lock counter cannot be requested to be identical: they operate on distinct domains (lock references on the concrete side, and locations on the abstract side). The following definition relates two heaps as equivalent (modulo renaming the locks involved), if they behave the same wrt. when a threads waits on a lock or not. The definition will be used in the deadlock-sensitive simulation relation afterwards.

Definition 3. *Given two heaps σ_1 and σ_2. A* heap-mapping *θ is a bijection between $dom(\sigma_1)$ and $dom(\sigma_2)$ such that $\sigma_1(l) = \sigma_2(\theta(l))$. Two heaps σ_1 and σ_2 are* wait-equivalent*, written $\sigma_2 \equiv \sigma_1$, if $dom(\sigma_1) = dom(\sigma_2)$, and furthermore $\sigma_1(l) = free$ iff $\sigma_2(l) = free$, and $\sigma_1(l) = p(n_1)$ iff $\sigma_2(l) = p(n_2)$. We use $\sigma_1 \equiv_\theta \sigma_2$ for $\sigma_1 \equiv \sigma_2'$ where the locks of σ_2' renamed according to θ. We will later use the definition analogously for locations π instead of lock references l.*

The definition of simulation is standard, except that we need to add that the simulating behavior cannot do everything the partner can do, as well, i.e., to preserve the

[4] To be precise, the type and effect system is not monomorphic, as it supports subtyping and sub-effecting. However, it does not support universal polymorphism.

ability to do labelled steps. In addition, the simulating partner must also be able to go into a *waiting* state, if its partner does and furthermore, the same preservation for termination. The latter condition about termination is not needed to prove preservation of deadlocks via simulation; the additional condition will be relevant later when using a *compositional* argument for deadlock preservation, namely when showing preservation of simulation in the context of sequential composition. As customary, internal steps, when relating two transition systems via simulation, do not count. To capture that, we start define a "weak" notion of transition, ignoring leading τ-steps (where $p\langle\tau\rangle$-labels count as silent). So the weak transition relation $\xRightarrow{p\langle a\rangle}$ is defined as $\xrightarrow{p\langle\tau\rangle}{}^{*}\xrightarrow{p\langle a\rangle}$ (the p is meant to be the same). Formalization and axiomatization of termination has been studied widely in the context of process algebra (see for instance [3]), especially for ACP. Termination is also relevant when formalizing *action refinement* since replacing a single action by more than one requires to consider sequential composition of actions, not just action prefixing. Respective notions of equivalence have been studied, cf. e.g. [37], collecting a number of results in the context of event structures, a well-known *true concurrency* model of concurrency. The paper includes equivalence notions preserved by action refinement/sequential composition such as history-preserving bisimulation.

Definition 4 (Deadlock and termination sensitive simulation $\lesssim^{DT}$ / $\lesssim^{D}$). *Assume a heap-mapping θ and a corresponding equivalence $\equiv_\theta$ (for which we simply write $\equiv$ in the following). A binary relation R between configurations is a* deadlock and termination sensitive simulation relation *(or just simulation for short) if the following holds. Assume $\sigma_1 \vdash \Phi_1 \; R \; \sigma_2 \vdash \Phi_2$ with $\sigma_1 \equiv \sigma_2$. Then:*

1. *If $\sigma_1 \vdash \Phi_1 \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash \Phi_1'$, then $\sigma_2 \vdash \Phi_2 \xrightarrow{p\langle\tau\rangle} \sigma_2 \vdash \Phi_2'$ or $\sigma_2 \vdash \Phi_2' = \sigma_2 \vdash \Phi_2$ s.t. $\sigma_1' \vdash \Phi_1' R \; \sigma_2 \vdash \Phi_2'$.*
2. *If $\sigma_1 \vdash \Phi_1 \xrightarrow{p\langle a\rangle} \sigma_1' \vdash \Phi_1'$, then $\sigma_2 \vdash \Phi_2 \xRightarrow{p\langle a\rangle} \sigma_2' \vdash \Phi_2'$ for some $\sigma_2' \vdash \Phi_2'$ with $\sigma_1' \equiv \sigma_2'$ and $\sigma_1' \vdash \Phi_1' R \; \sigma_2' \vdash \Phi_2'$.*
3. *If $waits((\sigma_1 \vdash \Phi_1), p, l)$, then $(\sigma_2 \vdash \Phi_2) \xrightarrow{p\langle\tau^*\rangle} \sigma_2' \vdash \Phi_2'$ for some $\sigma_2' \vdash \Phi_2'$ where $waits((\sigma_2' \vdash \Phi_2'), p, \theta(l))$.*
4. *If $\sigma_1 \vdash \Phi_1 \xrightarrow{p\langle\surd\rangle} \sigma_1 \vdash \Phi_1'$, then $\sigma_2 \vdash \Phi_2 \xrightarrow{p\langle\surd\rangle} \sigma_2 \vdash \Phi_2'$ and $\sigma_1 \vdash \Phi_1' \; R \; \sigma_2 \vdash \Phi_2'$.*

The configuration $\sigma_1 \vdash \Phi_1$ is simulated by *configuration $\sigma_2 \vdash \Phi_2$ (written $\sigma_1 \vdash \Phi_1 \lesssim^{DT} \sigma_2 \vdash \Phi_2$), if there exists a deadlock and termination sensitive simulation s.t. $\sigma_1 \vdash \Phi_1 \; R \; \sigma_2 \vdash \Phi_2$. Without part 4 of the definition, we call it* deadlock-sensitive *simulation, and write $\lesssim^{D}$ for the corresponding relation.*

Figure 3 illustrates the definition, where part 3 of the definition of deadlock sensitive simulation, where the negated transition is a graphical representation of the definition of waiting on a lock (cf. Definition 1).

It is straightforward to see that the binary relation $\lesssim^{DT}$ is itself a deadlock and termination sensitive simulation, and furthermore that it is reflexive and transitive.

The simulation relation from Definition 4 allows straightforwardly to carry over the property of deadlock freedom from the more abstract behavior to the more concrete. For the preservation result, termination sensitivity is not needed.

$$\begin{tikzcd}
\sigma_1 \vdash \Phi_1 \arrow[r, dash, "R"] \arrow[d, "p\langle\tau\rangle"] & \sigma_2 \vdash \Phi_2 \arrow[d, dotted, "=/p\langle\tau\rangle"] \\
\sigma_1 \vdash \Phi_1' \arrow[r, dotted, dash, "R"] & \sigma_2 \vdash \Phi_2'
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
\sigma_1 \vdash \Phi_1 \arrow[r, dash, "R"] \arrow[d, "p\langle a\rangle"] & \sigma_2 \vdash \Phi_2 \arrow[d, dotted, Rightarrow, "p\langle a\rangle"] \\
\sigma_1' \vdash \Phi_1' \arrow[r, dotted, dash, "R"] & \sigma_2' \vdash \Phi_2'
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
\sigma_1 \vdash \Phi_1 \arrow[r, dash, "R"] \arrow[d, "\neg p\langle \mathtt{L}^{\pi}\mathtt{.lock}\rangle"] & \sigma_2 \vdash \Phi_2 \arrow[d, dotted, Rightarrow, "\neg p\langle \mathtt{L}^{\pi}\mathtt{.lock}\rangle"] \\
\sigma_1' \vdash \Phi_1' & \sigma_2' \vdash \Phi_2'
\end{tikzcd}$$

(c)

$$\begin{tikzcd}
\sigma_1 \vdash \Phi_1 \arrow[r, dash, "R"] \arrow[d, "p\langle\surd\rangle"] & \sigma_2 \vdash \Phi_2 \arrow[d, dotted, "p\langle\surd\rangle"] \\
\sigma_1 \vdash \Phi_1' \arrow[r, dotted, dash, "R"] & \sigma_2 \vdash \Phi_2'
\end{tikzcd}$$

(d)

Fig. 3. DT-Simulation

Lemma 1 (Preservation of deadlock freedom). *Assume* $\sigma_1 \vdash \Phi_1 \lesssim^D \sigma_2 \vdash \Phi_2$. *If* $\sigma_2 \vdash \Phi_2$ *is deadlock free, then so is* $\sigma_1 \vdash \Phi_1$.

Proof. For a trace of labels s, let $\overset{s}{\Rightarrow}$ denote the corresponding sequence of labelled weak transition steps. We prove contra-positively that if $\sigma_1 \vdash \Phi_1$ contains a deadlock, then also $\sigma_2 \vdash \Phi_2$. So assume that $\sigma_1 \vdash \Phi_1 \overset{s}{\Rightarrow} \sigma_1' \vdash \Phi_1'$ such that $\sigma_1' \vdash \Phi_1'$ is deadlocked. By assumption, there exists a simulation relation s.t. $\sigma_1 \vdash \Phi_1 \; R \; \sigma_2 \vdash \Phi_2$. This implies that $\sigma_2 \vdash \Phi_2 \overset{s}{\Rightarrow} \sigma_2' \vdash \Phi_2'$ s.t.

$$\sigma_1' \vdash \Phi_1' \; R \; \sigma_2' \vdash \Phi_2' \; . \tag{12}$$

Being deadlocked means for the configuration $\sigma_1' \vdash \Phi_1'$ that $\sigma_1'(\pi_i) = p_i(n_i)$ and $waits(\sigma_1' \vdash \Phi_1', p_i, \pi_{i+_k 1})$ for some $k \geq 1$. Being in simulation relation implies with part 3 of Definition 4, that also $waits(\sigma_2' \vdash \Phi_2', p_i, \pi_{i+_k 1})$. Hence, $\sigma_2' \vdash \Phi_2'$ is deadlocked and thus $\sigma_2 \vdash \Phi_2$ contains a deadlock. □

The next two lemmas are concerned with “compositionality” as far as the simulation relation is concerned. It is crucial to use the stronger notion $\lesssim^{DT}$ of simulation that respects termination, and not $\lesssim^D$. In particular, in the presence of sequential composition, $\lesssim^D$ is *not preserved:* given $\sigma \vdash p\langle\varphi_1\rangle \lesssim^D \sigma \vdash p\langle\varphi_2\rangle$ does not imply $\sigma \vdash p\langle\varphi_1;\varphi\rangle \lesssim^D \sigma \vdash p\langle\varphi_2;\varphi\rangle$ (see rule S-SEQ$_2$ in Table 11), namely in situations where φ_1 terminates but φ_2 does not. A simple example illustrating that point is $\varphi_1 = rec X.\tau;X + \varepsilon$ and $\varphi_2 = rec X.\tau;X$.

Lemma 2 (Composition). *The implications formalized as rules in Table 11 are valid.*

Proof. Case: S-CHOICE:
In this case of sequential composition, we are given $\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle$ and we need to prove that also $\sigma \vdash p\langle\varphi_1 + \varphi\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2 + \varphi\rangle$. From the assumption $\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle$ we get that $\sigma \vdash p\langle\varphi_1\rangle \; R' \; \sigma \vdash p\langle\varphi_2\rangle$ for some deadlock and termination sensitive simulation relation R'.

Define the binary relation R between two configurations as follows: It contains the pair of initial configurations, i.e., $\sigma \vdash p\langle\varphi_1 + \varphi\rangle \; R \; \sigma \vdash p\langle\varphi_2 + \varphi\rangle$, furthermore it is defined as R' on the left subtrees of the initial configurations $\sigma \vdash p\langle\varphi_1 + \varphi\rangle$ and $\sigma \vdash p\langle\varphi_2 + \varphi\rangle$, and the identity in the right sub-trees of the initial configurations.

Due to non-determinstic choice, there are two possible post-configurations of $\sigma \vdash p\langle\varphi_1 + \varphi\rangle$, both reachable by a τ-step. One step is $\sigma \vdash p\langle\varphi_1 + \varphi\rangle \xrightarrow{p\langle\tau\rangle} \sigma \vdash p\langle\varphi_1\rangle$. For this case, $\sigma \vdash p\langle\varphi_2 + \varphi\rangle \xrightarrow{p\langle\tau\rangle} \sigma \vdash p\langle\varphi_2\rangle$, where $\sigma \vdash p\langle\varphi_1\rangle \; R' \; \sigma \vdash p\langle\varphi_2\rangle$ by assumption, which implies $\sigma \vdash p\langle\varphi_1\rangle \; R \; \sigma \vdash p\langle\varphi_2\rangle$, by definition of R.

The other case is that $\sigma_1 \vdash p\langle\varphi_1 + \varphi\rangle \xrightarrow{p\langle\tau\rangle} \sigma \vdash p\langle\varphi\rangle$. Also, the second configuration can do the corresponding τ-step, i.e., $\sigma \vdash p\langle\varphi_2 + \varphi\rangle \xrightarrow{p\langle\tau\rangle} \sigma \vdash p\langle\varphi\rangle$. The two post-configurations are identical, i.e., they are related by the identity: $\sigma \vdash p\langle\varphi\rangle \; Id \; \sigma \vdash p\langle\varphi\rangle$, which implies $\sigma \vdash p\langle\varphi\rangle \; R \; \sigma \vdash p\langle\varphi\rangle$, by definition of R. Note that we do not consider the remaining conditions of Definition 4 as $\sigma \vdash p\langle\varphi_1 + \varphi\rangle$ takes only a τ-step for the non-deterministic choice. To sum up: R is a deadlock and termination sensitive simulation relating the initial configurations $\sigma \vdash p\langle\varphi_1 + \varphi\rangle$ and $\sigma \vdash p\langle\varphi_2 + \varphi\rangle$. Hence, $\sigma \vdash p\langle\varphi_1 + \varphi\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2 + \varphi\rangle$ (by definition of $\lesssim^{DT}$), as required.

Case: S-SEQ$_1$:
In this case we are given $\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle$ and we need to prove that $\sigma \vdash p\langle\varphi;\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi;\varphi_2\rangle$. From the assumption $\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle$ we get that $\sigma \vdash p\langle\varphi_1\rangle \; R' \; \sigma \vdash p\langle\varphi_2\rangle$ for some deadlock and termination sensitive simulation relation R'.

Define the binary relation R between two configurations as follows: It contains the pair of initial configurations, i.e., $\sigma \vdash p\langle\varphi;\varphi_1\rangle \; R \; \sigma \vdash p\langle\varphi;\varphi_2\rangle$, and it is defined as the identity of the prefix in the two initial configurations, that is, $\sigma \vdash p\langle\varphi\rangle \; R \; \sigma \vdash p\langle\varphi\rangle$; furthermore, it is defined as R' on the postfix of the two configurations. It is straightforward to see that R is indeed a deadlock termination sensitive simulation relation.

Case: S-SEQ$_2$:
In this case we are also given $\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle$ and we need to prove that $\sigma \vdash p\langle\varphi_1;\varphi\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2;\varphi\rangle$. From the assumption $\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle$ we get that $\sigma \vdash p\langle\varphi_1\rangle \; R' \; \sigma \vdash p\langle\varphi_2\rangle$ for some deadlock and termination sensitive simulation relation R'.

We define the binary relation R between two configurations as follows: It contains the pair of initial configurations, i.e., $\sigma \vdash p\langle\varphi_1;\varphi\rangle \; R \; \sigma \vdash p\langle\varphi_2;\varphi\rangle$ and defined as R' on the prefix in the two initial configurations; further it is defined as identity for the postfix φ of the configurations; furthermore, the heap-parts are related by $\equiv$.

The first three conditions of deadlock and termination sensitive simulation hold straightforwardly for R. For condition (4), dealing with termination, the assumption $\sigma \vdash p\langle\varphi_1\rangle \; R' \; \sigma \vdash p\langle\varphi_2\rangle$ guarantees that if φ_1 terminates in one state, φ_2 may also terminate in an equivalent state. Therefore, the simulation can continue with the identical remaining effect φ such that $\sigma \vdash p\langle\varphi\rangle \; R \; \sigma' \vdash p\langle\varphi\rangle$ where $\sigma \equiv \sigma'$. Thus, R is a deadlock and termination sensitive simulation relation, and therefore $\sigma \vdash p\langle\varphi_1;\varphi\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2;\varphi\rangle$, as required.

Case: S-SPAWN
For the rule of spawning an effect, we are given $\sigma \vdash p'\langle\varphi_1\rangle \lesssim^{DT} \sigma_1 \vdash p'\langle\varphi_2\rangle$ and we have to prove that $\sigma \vdash p\langle\mathtt{spawn}\ \varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\mathtt{spawn}\ \varphi_2\rangle$. From the assumption $\sigma \vdash p'\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p'\langle\varphi_2\rangle$ we get that $\sigma \vdash p'\langle\varphi_1\rangle\ R'\ \sigma \vdash p'\langle\varphi_2\rangle$ for some deadlock and termination sensitive simulation relation R'.

We define the binary relation R between two configurations as follows: It contains the pair of initial configurations, i.e., $\sigma \vdash p\langle\mathtt{spawn}\ \varphi_1\rangle\ R\ \sigma \vdash p\langle\mathtt{spawn}\ \varphi_2\rangle$, and it is defined as R' between the two configurations given in the assumption, that is, $\sigma \vdash p'\langle\varphi_1\rangle\ R'\ \sigma \vdash p'\langle\varphi_2\rangle$; and is further defined as the identity for the empty effects such that $\sigma \vdash p\langle\varepsilon\rangle\ R\ \sigma \vdash p\langle\varepsilon\rangle$.

Since $\sigma \vdash p\langle\mathtt{spawn}\ \varphi_1\rangle$ can only proceed with a transition step with label $\mathtt{spawn}\ \varphi_1$, we just only have to consider part 2 of Definition 4. For part 2, the label a is $\mathtt{spawn}\ \varphi_1$ and therefore $\sigma \vdash p\langle\mathtt{spawn}\ \varphi_1\rangle \xrightarrow{p\langle\mathtt{spawn}\varphi_1\rangle} \sigma \vdash p\langle\varepsilon\rangle \parallel p'\langle\varphi_1\rangle$ by RE-SPAWN in Table 10. Similarly, the other configuration will also do a spawning step, that is, $\sigma \vdash p\langle\mathtt{spawn}\ \varphi_2\rangle \xrightarrow{p\langle\mathtt{spawn}\varphi_2\rangle} \sigma \vdash p\langle\varepsilon\rangle \parallel p'\langle\varphi_2\rangle$. By the assumption and the definition of R, it is straightforward that $\sigma \vdash p\langle\varepsilon\rangle \parallel p'\langle\varphi_1\rangle\ R\ \sigma \vdash p\langle\varepsilon\rangle \parallel p'\langle\varphi_2\rangle$ holds, which implies that R is a deadlock termination sensitive simulation relation. Hence, $\sigma \vdash p\langle\mathtt{spawn}\ \varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\mathtt{spawn}\ \varphi_2\rangle$.

Case: S-PAR:
Straightforward. □

The next lemma is a straightforward consequence, showing that $\lesssim^{DT}$ behaves covariantly or monotonely when replacing behavior within a context. We ignore the situation when replacing inside a recursion; the reason simply is that later we do not need to consider that case.

Lemma 3 (Context). *If $\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle$, then $\sigma' \vdash p\langle\varphi[\varphi_1]\rangle \lesssim^{DT} \sigma' \vdash p\langle\varphi[\varphi_2]\rangle$, where the holes $[]$ in $\varphi[]$ do not occur inside a recursion.*

Proof. By straightforward induction on the structure of $\varphi[]$, and with the help of Lemma 2. The base case $\varphi[] = []$ where the context is empty is immediate. For the case of choice, we are given $\varphi[] = \varphi_l[] + \varphi_r[]$. By induction, we get $\sigma \vdash \varphi_l[\varphi_1] \lesssim^{DT} \sigma \vdash \varphi_l[\varphi_2]$ and $\sigma \vdash \varphi_r[\varphi_1] \lesssim^{DT} \sigma \vdash \varphi_r[\varphi_2]$. Thus, the case follows by S-CHOICE of Lemma 2 and transitivity. The remaining cases work analogously. Note that we do not need to consider the case where $\varphi[] = rec X.\varphi'[]$. □

3.6 Subject reduction as simulation

In the type/effect based formalization, the proof of deadlock-sensitive simulation can be formulated in the form of a *subject reduction* result. As the static analysis is conceptually split into a (standard) typing part and the behavioral effect part, we split the preservation result also into these two aspects. We start with the typing part. The first lemma, preservation of typing under substitution, is standard.

Lemma 4 (Substitution). *If $\Gamma, x{:}T_1 \vdash t : T_2$ and $\Gamma \vdash v : T_1$, then $\Gamma \vdash t[v/x] : T_2$.*

$$\frac{\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle}{\sigma \vdash p\langle\varphi_1 + \varphi\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2 + \varphi\rangle}\ \textsc{S-Choice}$$

$$\frac{\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle}{\sigma \vdash p\langle\varphi;\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi;\varphi_2\rangle}\ \textsc{S-Seq}_1 \qquad \frac{\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle}{\sigma \vdash p\langle\varphi_1;\varphi\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2;\varphi\rangle}\ \textsc{S-Seq}_2$$

$$\frac{\sigma \vdash p'\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p'\langle\varphi_2\rangle}{\sigma \vdash p\langle\texttt{spawn}\,\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\texttt{spawn}\,\varphi_2\rangle}\ \textsc{S-Spawn}$$

$$\frac{\sigma \vdash p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash p\langle\varphi_2\rangle}{\sigma \vdash \Phi \parallel p\langle\varphi_1\rangle \lesssim^{DT} \sigma \vdash \Phi \parallel p\langle\varphi_2\rangle}\ \textsc{S-Par}$$

Table 11. Pre-congruence properties of $\lesssim^{DT}$

Proof. By straightforward induction on the typing derivation, generalizing the property of the lemma slightly: If $\Gamma_1, x{:}T_1, \Gamma_2 \vdash t : T_2$ and $\Gamma_1 \vdash v : T_1$, then $\Gamma_1, \Gamma_2 \vdash t[v/x] : T_2$. $\Box$

Lemma 5 (Subject reduction, local steps (types)). *If $\Gamma \vdash e : T$ and $e \xrightarrow{\tau} e'$, then $\Gamma \vdash e' : T$.*

Proof. Assume that e is well-typed, more precisely, $\Gamma \vdash e : T$, and assume further that e does a local transition step, i.e., $e \xrightarrow{\tau} e'$. Proceed by a case distinction on the rules for the transition step.

Case: R-Red: $\texttt{let}\,x{:}T_1 = v\;\texttt{in}\,t \xrightarrow{\tau} t[v/x]$
By the well-typedness assumption for t, we have $\Gamma \vdash \texttt{let}\,x{:}T_1 = v\;\texttt{in}\,t : T$. Inverting rule TE-Let gives :

$$\frac{\Gamma \vdash \texttt{let}\,v : T_1 \qquad \Gamma, x{:}T \vdash t : T}{\Gamma \vdash \texttt{let}\,x{:}T_1 = v\;\texttt{in}\,t : T}\ \textsc{TE-Let}$$

By preservation of typing under substitution (Lemma 4), $\Gamma \vdash t[v/x] : T$, which concludes the case.

Case: R-Let: $\texttt{let}\;x_2{:}T_2 = (\texttt{let}\;x_1{:}T_1 = e_1\;\texttt{in}\;t_1)\;\texttt{in}\;t_2 \xrightarrow{\tau} \texttt{let}\;x_1{:}T_1 = e_1\;\texttt{in}\;(\texttt{let}\;x_2{:}T_2 = t_1\;\texttt{in}\,t_2)$

By assumption we are given $\texttt{let}\,x_2{:}T_2 = (\texttt{let}\,x_1{:}T_1 = e_1\;\texttt{in}\,t_1)\;\texttt{in}\,t_2 : T$. Inverting the type rule TE-Let two times gives:

$$\frac{\dfrac{\Gamma \vdash e_1 : T_1 \qquad \Gamma, x_1{:}T_1 \vdash t_1 : T_2}{\Gamma \vdash \texttt{let}\,x_1{:}T_1 = e_1\;\texttt{in}\,t_1 : T_2}\ \textsc{TE-Let} \qquad \Gamma, x_2{:}T_2 \vdash t_2 : T}{\Gamma \vdash \texttt{let}\,x_2{:}T_2 = (\texttt{let}\,x_1{:}T_1 = e_1\;\texttt{in}\,t_1)\;\texttt{in}\,t_2 : T}\ \textsc{TE-Let}$$

Using weakening on the right-most subgoal $\Gamma, x_2{:}T_2 \vdash t_2 : T$ gives $\Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t_2 : T$. Therefore we can conclude with using two times TE-LET again:

$$\dfrac{\Gamma \vdash e_1 : T_1 \qquad \dfrac{\Gamma, x_1{:}T_1 \vdash t_1{:}T_2 \qquad \Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t_2 : T}{\Gamma, x_1{:}T_1 \vdash \texttt{let}\, x_2{:}T_2 = t_1 \;\texttt{in}\, t_2 : T}\,\text{TE-LET}}{\Gamma \vdash \texttt{let}\, x_1{:}T_1 = e_1 \;\texttt{in}\, (\texttt{let}\, x_2{:}T_2 = t_1 \;\texttt{in}\, t_2) : T}\,\text{TE-LET}$$

Case: R-IF$_1$: $\texttt{let}\, x{:}T = \texttt{if true then}\, e_1 \;\texttt{else}\, e_2 \;\texttt{in}\, t \xrightarrow{\tau} \texttt{let}\, x{:}T = e_1 \;\texttt{in}\, e$
We are given that $\texttt{let}\, x{:}T = \texttt{if true then}\, e_1 \;\texttt{else}\, e_2 \;\texttt{in}\, e{:}T$, so inverting rule TE-LET followed by rule TE-IF gives:

$$\dfrac{\dfrac{\Gamma \vdash \texttt{true:Bool} \qquad \Gamma \vdash e_1 : T \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if true then}\, e_1 \;\texttt{else}\, e_2 : T}\,\text{TE-IF}_1 \qquad \Gamma, x{:}T \vdash t : T}{\texttt{let}\, x{:}T = \texttt{if true then}\, e_1 \;\texttt{else}\, e_2 \;\texttt{in}\, t{:}T}\,\text{TE-LET}$$

We conclude our proof by using TE-LET which gives:

$$\dfrac{\Gamma \vdash e_1 : T \qquad \Gamma, x{:}T \vdash t : T}{\Gamma \vdash \texttt{let}\, x{:}T = e_1 \;\texttt{in}\, t : T}\,\text{TE-LET}$$

The case for R-IF$_2$ works analogously.

Case: R-APP$_1$: $\texttt{let}\, x{:}T_1 = (\texttt{fn}\, x'{:}T'.t')\; v \;\texttt{in}\, t \xrightarrow{\tau} \texttt{let}\, x{:}T_1 = t'[v/x'] \;\texttt{in}\, t$
By assumption, $\Gamma \vdash \texttt{let}\, x{:}T_1 = (\texttt{fn}\, x'{:}T'.t')\; v \;\texttt{in}\, t : T$. By inverting rules TE-LET, TE-APP, and T-ABS$_1$, we get:

$$\dfrac{\dfrac{\dfrac{\Gamma, x'{:}T' \vdash t' : T_1}{\Gamma \vdash \texttt{fn}\, x'{:}T'.t' : T' \to T_1}\,\text{TE-ABS}_1 \qquad \Gamma \vdash v : T'}{\Gamma \vdash (\texttt{fn}\, x'{:}T'.t')\; v : T_1}\,\text{TE-APP} \qquad \Gamma, x{:}T_1 \vdash t : T}{\Gamma \vdash \texttt{let}\, x{:}T_1 = (\texttt{fn}\, x'{:}T'.t')\; v \;\texttt{in}\, t : T}\,\text{TE-LET}$$

By preservation of typing by substitution from Lemma 4 on the left-most subgoal, we get $\Gamma \vdash t'[v/x'] : T_1$, thus by TE-LET:

$$\dfrac{\Gamma \vdash t'[v/x'] : T_1 \qquad \Gamma, x{:}T_1 \vdash t : T}{\Gamma \vdash \texttt{let}\, x{:}T_1 = t'[v/x'] \;\texttt{in}\, t : T}\,\text{TE-LET}$$

which concludes the case. The case for R-APP$_2$, dealing with the application of a recursive function works similarly. □

Lemma 6 (Subject reduction, global steps (types)). *If $\Gamma \vdash P : ok$ and $\sigma \vdash P \to \sigma' \vdash P'$, then $\Gamma \vdash P' : ok$ (where $\to$ is meant as an arbitrary step, independent of the label).*

Proof. Assume that P is well-typed, more precisely, $\Gamma \vdash P : ok$, and assume further that P does a global transition step, i.e., $\sigma \vdash P \to \sigma' \vdash P'$. Proceed by induction on the derivation for the reduction step.

Case: R-Lift: $\sigma \vdash p\langle t_1\rangle \to p\langle t_2\rangle$
where $t_1 \to t_2$. By assumption, $\vdash p\langle t_1\rangle : ok$. By inverting TE-Thread, $\vdash t_1 : T$ for some type T. By the previous subject reduction result for local reduction steps (Lemma 5), $\Gamma \vdash t_2 : T$. Thus the result follows by rule TE-Thread:

$$\dfrac{\Gamma \vdash t_2 : T}{\Gamma \vdash p\langle t_2\rangle : ok}$$

Case: R-Par: $\sigma \vdash P_1 \parallel P_2 \to \sigma' \vdash P_1' \parallel P_2$
where $\sigma \vdash P_1 \to \sigma' \vdash P_1$. From the well-typedness assumption, we get by inverting TE-Par:

$$\dfrac{\Gamma \vdash P_1 : ok \qquad \Gamma \vdash P_2 : ok}{\Gamma \vdash P_1 \parallel P_2 : ok}\ \textsc{TE-Par}$$

By induction on $\sigma \vdash P_1 \to \sigma' \vdash P_1'$, it gives $\Gamma \vdash P_1' : ok$ for the thread after the step, and hence by rule T-Par

$$\dfrac{\Gamma \vdash P_1' : ok \qquad \Gamma \vdash P_2 : ok}{\Gamma \vdash P_1' \parallel P_2 : ok}\ \textsc{TE-Par}$$

as required.

Case: R-Spawn: $\sigma \vdash p_1\langle \texttt{let}\, x{:}T = \texttt{spawn}\, t_2\ \texttt{in}\, t_1\rangle \to \sigma \vdash p_1\langle \texttt{let}\, x{:}T = p_2\ \texttt{in}\, t_1\rangle \parallel p_2\langle t_2\rangle$
By well-typedness and inverting rules TE-Thread, TE-Let, and TE-Spawn, we obtain:

$$\dfrac{\dfrac{\dfrac{T = \texttt{Thread} \qquad \Gamma \vdash t_2 : T_2}{\Gamma \vdash \texttt{spawn}\, t_2 : T}\ \textsc{TE-Spawn} \qquad \Gamma, x{:}T \vdash t_1 : T_1}{\Gamma \vdash \texttt{let}\, x{:}T = \texttt{spawn}\, t_2\ \texttt{in}\, t_1 : T_1}\ \textsc{TE-Let}}{\Gamma \vdash p_1\langle \texttt{let}\, x{:}T = \texttt{spawn}\, t_2\ \texttt{in}\, t_1\rangle : ok}\ \textsc{TE-Thread}$$

Thus we can conclude with rules TE-RefP, TE-Let, TE-Thread and TE-Par:

$$\dfrac{\dfrac{\dfrac{\dfrac{T = \texttt{Thread}}{\Gamma \vdash p_2 : T}\ \textsc{TE-RefP} \qquad \Gamma, x{:}T \vdash t_1 : T_1}{\Gamma \vdash \texttt{let}\, x{:}T = p_2\ \texttt{in}\, t_1 : T_1}\ \textsc{TE-Let}}{\Gamma \vdash p_1\langle \texttt{let}\, x{:}T = p_2\ \texttt{in}\, t_1\rangle : ok} \qquad \dfrac{\Gamma \vdash t_2 : T_2}{\Gamma \vdash p_2\langle t_2\rangle : ok}}{\Gamma \vdash p_1\langle \texttt{let}\, x{:}T = p_2\ \texttt{in}\, t_1\rangle \parallel p_2\langle t_2\rangle : ok}\ \textsc{TE-Par}$$

Case: R-NewL: $\sigma \vdash p\langle \texttt{let}\, x{:}T_1 = \texttt{new}_\pi\, L\ \texttt{in}\, t\rangle \to \sigma' \vdash p\langle \texttt{let}\, x{:}T_1 = l^\pi\ \texttt{in}\, t\rangle$
where $\sigma' = \sigma[l \mapsto \mathit{free}]$ for a fresh l. By assumption, the thread before the step is well-typed, i.e., $\vdash p\langle \texttt{let}\ x{:}T_1 = \texttt{new}_\pi\ L\ \texttt{in}\ e\rangle : ok$. Inverting TE-Thread, TE-Let and TE-NewL gives

$$\dfrac{\dfrac{\dfrac{T_1 = \mathrm{L}^\pi}{\Gamma \vdash \texttt{new}_\pi\, L : T_1}\ \textsc{TE-NewL} \qquad \Gamma, x{:}T_1 \vdash t : T}{\Gamma \vdash \texttt{let}\, x{:}T_1 = \texttt{new}_\pi\, L\ \texttt{in}\, t : T}\ \textsc{TE-Let}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T_1 = \texttt{new}_\pi\, L\ \texttt{in}\, t\rangle : ok}\ \textsc{TE-Thread}$$

The type T_1 is L^π by TE-NEWL. Thus we can conclude:

$$\dfrac{\dfrac{\Gamma \vdash l^\pi : T_1 \qquad T_1 = \mathrm{L}^\pi \qquad \Gamma, x{:}T_1 \vdash e : T}{\Gamma \vdash \mathtt{let}\, x{:}T_1 = l^\pi\ \mathtt{in}\, t : T}\ \textsc{TE-Let}}{\Gamma \vdash p\langle \mathtt{let}\, x{:}T_1 = l^\pi\ \mathtt{in}\, t\rangle : ok}\ \textsc{TE-Thread}$$

Case: R-LOCK: $\sigma \vdash p\langle \mathtt{let}\, x{:}T_1 = l^\pi.\, \mathtt{lock}\ \mathtt{in}\, t\rangle \rightarrow p\langle \mathtt{let}\, x{:}T_1 = l^\pi\ \mathtt{in}\, t\rangle$
where $\sigma(l^\pi) = \mathit{free}$, or $\sigma' = \sigma[l^\pi \mapsto p(n)]$, i.e., l^π is taken by process p n times. By assuming well-typedness and inverting rules TE-THREAD, TE-LET, TE-LOCK and TE-LREF gives

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash l^\pi : \mathrm{L}^\pi}\ \textsc{TE-LRef} \qquad T_1 = \mathrm{L}^\pi}{\Gamma \vdash l^\pi.\, \mathtt{lock} : T_1}\ \textsc{TE-Lock} \qquad \Gamma, x{:}T_1 \vdash t : T}{\Gamma \vdash \mathtt{let}\, x{:}T_1 = l^\pi.\, \mathtt{lock}\ \mathtt{in}\, t : T}\ \textsc{TE-Let}}{\Gamma \vdash p\langle \mathtt{let}\, x{:}T_1 = l^\pi.\, \mathtt{lock}\ \mathtt{in}\, t\rangle : ok}\ \textsc{TE-Thread}$$

Thus we can conclude:

$$\dfrac{\dfrac{\Gamma \vdash l^\pi : T_1 \qquad \Gamma, x{:}T_1 \vdash t : T}{\Gamma \vdash \mathtt{let}\, x{:}T = l\ \mathtt{in}\, t : T}\ \textsc{TE-Let}}{\Gamma \vdash p\langle \mathtt{let}\, x{:}T_1 = l^\pi\ \mathtt{in}\, t\rangle : ok}\ \textsc{TE-Thread}$$

The cases for R-UNLOCK case work similarly. □

Next we treat subject reduction for the effects. As the behavioral effects describe (an over-approximation of) the future behavior of a program, we cannot expect that doing a reduction step *preserves* the effect in general. For instance, if the behavior is described by a behavior φ of the form $\mathrm{L}^r.\,\mathtt{lock};\varphi'$, then if the program actually does the corresponding lock-taking step, its behavior should be described by φ' afterwards. Clearly, not all steps the program does "count" in that way because some of the steps are irrelevant for the behavior dealing with locks. Technically, the behavior of the program and the behavior of the effects are related by a deadlock-sensitive simulation (see Corollary 1 later).

We start again with a simple substitution lemma, this time for effects.

Lemma 7 (Substitution (effects)). *If $\Gamma, x{:}T \vdash t :: \varphi$, then $\Gamma \vdash t[v/x] :: \varphi$.*

Proof. Straightforward, using the fact that v is a value and therefore has the empty effect (as has the variable x it replaces). □

Lemma 8 (Subject reduction (effects)). *Let the heap mapping θ map concrete locks l^π to their locations π. Note that due to the restrictions in our setting this mapping is a bijection, as required. Let $\Gamma \vdash p\langle t\rangle :: p\langle\varphi\rangle$, and furthermore $\sigma_1 \equiv \sigma_2$.*

1. *$\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash p\langle t'\rangle$, then $\Gamma \vdash p\langle t'\rangle :: p\langle\varphi\rangle$.*

2. (a) $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle a\rangle} \sigma_1' \vdash p\langle t'\rangle$ *and* $a \neq \texttt{spawn}\,\varphi''$, *then* $\sigma_2 \vdash p\langle\varphi\rangle \overset{p\langle a\rangle}{\Longrightarrow} \sigma_2' \vdash p\langle\varphi'\rangle$ *and* $\Gamma \vdash p\langle t'\rangle :: p\langle\varphi'\rangle$.

   (b) $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle a\rangle} \sigma_1 \vdash p\langle t''\rangle \parallel p'\langle t'\rangle$ *where* $a = \texttt{spawn}\,\varphi'$, *then* $\sigma_2 \vdash p\langle\varphi\rangle \overset{p\langle a\rangle}{\Longrightarrow} \sigma_2' \vdash p\langle\varphi''\rangle \parallel p'\langle\varphi'\rangle$ *such that* $\Gamma \vdash p\langle t''\rangle :: p\langle\varphi''\rangle$ *and* $\Gamma \vdash p'\langle t'\rangle :: p'\langle\varphi'\rangle$.

3. *If* $waits(\sigma_1 \vdash p\langle t\rangle, p, l^\pi)$, *then* $\sigma_2 \vdash p\langle\varphi\rangle \xrightarrow{p\langle\tau^*\rangle} \sigma_2 \vdash p\langle\varphi'\rangle$ *s.t.* $waits(\sigma_2 \vdash p\langle\varphi'\rangle, p, \pi)$.

Proof. We are given $\Gamma \vdash p\langle t\rangle :: p\langle\varphi\rangle$. In part 1, furthermore $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash p\langle t'\rangle$. In case of steps justified by the rules for local steps of Table 2, σ_1 remains unchanged. Furthermore, by inverting R-LIFT, we have $t \xrightarrow{\tau} t'$, and it suffices to show that $\Gamma \vdash t :: \varphi$ implies $\Gamma \vdash t' :: \varphi$. We proceed by case distinction on the rules for the local transition steps from Table 2.

Case: R-RED: $p\langle \texttt{let}\, x{:}T = v \,\texttt{in}\, t\rangle \xrightarrow{p\langle\tau\rangle} p\langle t[v/x]\rangle$
By well-typedness, we are given $\Gamma \vdash p\langle \texttt{let}\, x{:}T = v \,\texttt{in}\, t\rangle :: p\langle\varphi\rangle$, so inverting rule TE-THREAD and TE-LET gives:

$$\dfrac{\dfrac{\Gamma \vdash v :: \varphi_1 \qquad \Gamma, x{:}T \vdash t :: \varphi_2}{\Gamma \vdash \texttt{let}\, x{:}T = v \,\texttt{in}\, t :: \varphi_1;\varphi_2}\ \text{TE-LET}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = v \,\texttt{in}\, t\rangle :: p\langle\varphi_1;\varphi_2\rangle}\ \text{TE-THREAD}$$

where $\varphi = \varphi_1;\varphi_2$. The effect of a value v is empty, i.e., $\varphi_1 = \varepsilon$ (cf. the corresponding rules for values TE-VAR, TE-LREF, TE-ABS$_1$, and TE-ABS$_2$ from Table 6). Therefore, the overall effect of the thread is $\varepsilon;\varphi_2$. By preservation of typing under substitution (Lemma 7) we get from the second premise of the above rule that $\Gamma \vdash t[v/x] :: \varphi_2$. By SE-UNIT, $\varepsilon;\varphi_2 \equiv \varphi_2$ that implies $\varepsilon;\varphi_2 \geq \varphi_2$ with SE-REFL, from which the case follows by subsumption and TE-THREAD:

$$\dfrac{\dfrac{\Gamma \vdash t[v/x] :: \varphi_2 \qquad \varphi_2 \leq \varepsilon;\varphi_2}{\Gamma \vdash t[v/x] :: \varepsilon;\varphi_2}\ \text{TE-SUB}}{\Gamma \vdash p\langle t[v/x]\rangle :: p\langle\varepsilon;\varphi_2\rangle}\ \text{TE-THREAD}$$

Case: R-LET: $p\langle \texttt{let}\, x_2{:}T_2 = (\texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, t_1) \,\texttt{in}\, t_2\rangle \xrightarrow{p\langle\tau\rangle} p\langle \texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, (\texttt{let}\, x_2{:}T_2 = t_1 \,\texttt{in}\, t_2)\rangle$
By assumption, we are given $\Gamma \vdash p\langle \texttt{let}\, x_2{:}T_2 = (\texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, t_1) \,\texttt{in}\, t_2\rangle :: p\langle\varphi\rangle$. Inverting the type rules TE-THREAD, followed by TE-LET twice gives:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash e_1 :: \varphi_1 \qquad \Gamma, x_1{:}T_1 \vdash t_1 :: \varphi_2}{\Gamma \vdash \texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, t_1 :: \varphi_1;\varphi_2} \qquad \Gamma, x_2{:}T_2 \vdash t_2 :: \varphi_3}{\Gamma \vdash \texttt{let}\, x_2{:}T_2 = (\texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, t_1) \,\texttt{in}\, t_2 :: (\varphi_1;\varphi_2);\varphi_3}}{\Gamma \vdash p\langle \texttt{let}\, x_2{:}T_2 = (\texttt{let}\, x_1{:}T_1 = e_1 \,\texttt{in}\, t_1) \,\texttt{in}\, t_2\rangle :: p\langle(\varphi_1;\varphi_2);\varphi_3\rangle}\ \text{TE-THREAD}$$

for $\varphi = (\varphi_1;\varphi_2);\varphi_3$. Using weakening on the right-most subgoal $\Gamma, x_2{:}T_2 \vdash t_2 :: \varphi_3$, it gives $\Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t_2 :: \varphi_3$. Therefore we can conclude by using two times TE-LET plus one application of subsumption and TE-THREAD:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash e_1 :: \varphi_1 \qquad \cfrac{\Gamma, x_1{:}T_1 \vdash t_1 :: \varphi_2 \qquad \Gamma, x_1{:}T_1, x_2{:}T_2 \vdash t_2 :: \varphi_3}{\Gamma, x_1{:}T_1 \vdash \texttt{let}\, x_2{:}T_2 = t_1 \;\texttt{in}\; t_2 :: \varphi_2;\varphi_3}}{\Gamma \vdash \texttt{let}\, x_1{:}T_1 = e_1 \;\texttt{in}\; (\texttt{let}\, x_2{:}T_2 = t_1 \;\texttt{in}\; t_2) :: \varphi_1;(\varphi_2;\varphi_3)}}{\Gamma \vdash \texttt{let}\, x_1{:}T_1 = e_1 \;\texttt{in}\; (\texttt{let}\, x_2{:}T_2 = t_1 \;\texttt{in}\; t_2) :: (\varphi_1;\varphi_2);\varphi_3}\;\text{TE-SUB}}{\Gamma \vdash p\langle \texttt{let}\, x_1{:}T_1 = e_1 \;\texttt{in}\; (\texttt{let}\, x_2{:}T_2 = t_1 \;\texttt{in}\; t_2)\rangle :: p\langle(\varphi_1;\varphi_2);\varphi_3\rangle}\;\text{TE-THREAD}$$

where $\varphi_1;(\varphi_2;\varphi_3)$ is equivalent to $(\varphi_1;\varphi_2);\varphi_3$, which further implies $\varphi_1;(\varphi_2;\varphi_3) \leq (\varphi_1;\varphi_2);\varphi_3$, which is used in the subsumption step.

Case: R-IF$_1$: $p\langle \texttt{let}\, x{:}T = \texttt{if true then}\; e_1 \;\texttt{else}\; e_2 \;\texttt{in}\; e\rangle \xrightarrow{p\langle\tau\rangle} p\langle \texttt{let}\, x{:}T = e_1 \;\texttt{in}\; e\rangle$
Assuming that $\Gamma \vdash p\langle \texttt{let}\, x{:}T = \texttt{if true then}\; e_1 \;\texttt{else}\; e_2 \;\texttt{in}\; e\rangle :: p\langle\varphi\rangle$, inverting rules TE-THREAD, TE-LET and TE-IF gives:

$$\cfrac{\cfrac{\cfrac{\Gamma \vdash \texttt{true} :: \varepsilon \qquad \Gamma \vdash e_1 :: \varphi_1 \qquad \Gamma \vdash e_2 :: \varphi_2}{\Gamma \vdash \texttt{if true then}\; e_1 \;\texttt{else}\; e_2 :: \varphi_1 + \varphi_2}\;\text{TE-IF} \qquad \Gamma, x{:}T \vdash e :: \varphi_e}{\Gamma \vdash \texttt{let}\, x{:}T = \texttt{if true then}\; e_1 \;\texttt{else}\; e_2 \;\texttt{in}\; e :: (\varphi_1 + \varphi_2);\varphi_e}\;\text{TE-LET}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = \texttt{if true then}\; e_1 \;\texttt{else}\; e_2 \;\texttt{in}\; e\rangle :: p\langle(\varphi_1 + \varphi_2);\varphi_e\rangle}\;\text{TE-THREAD}$$

where $\varphi = (\varphi_1 + \varphi_2);\varphi_e$ is equivalent to $\varphi_1;\varphi_e + \varphi_2;\varphi_e$ by SE-DISTR in Table 9. By applying rule TE-LET, we get:

$$\frac{\Gamma \vdash e_1 :: \varphi_1 \qquad \Gamma, x{:}T \vdash e :: \varphi_e}{\Gamma \vdash \texttt{let}\, x{:}T = e_1 \;\texttt{in}\; e :: \varphi_1;\varphi_e}\;\text{TE-LET}$$

and $\varphi_1;\varphi_e \leq \varphi$. By rule TE-SUB and TE-THREAD, we get $p\langle \texttt{let}\, x{:}T = e_1 \;\texttt{in}\; e\rangle :: p\langle\varphi\rangle$ which proves the case. The case for R-IF$_2$ works analogously.

Case: R-APP$_1$: $p\langle \texttt{let}\, x{:}T = (\texttt{fn}\, x'{:}T'.t')\; v \;\texttt{in}\; t\rangle \xrightarrow{p\langle\tau\rangle} p\langle \texttt{let}\, x{:}T = t'[v/x'] \;\texttt{in}\; t\rangle$
We are given $\Gamma \vdash p\langle \texttt{let}\, x{:}T = (\texttt{fn}\, x'{:}T'.t')\; v \;\texttt{in}\; t\rangle :: p\langle\varphi\rangle$. Hence inverting rules TE-THREAD, TE-LET, TE-APP, and TE-ABS$_1$ gives:

$$\cfrac{\cfrac{\cfrac{\cfrac{\Gamma, x'{:}T' \vdash t' : T :: \varphi_1}{\Gamma \vdash \texttt{fn}\, x'{:}T'.t' : T' \to^{\varphi_1} T :: \varepsilon}\;\text{TE-ABS}_1 \qquad \Gamma \vdash v :: \varepsilon}{\Gamma \vdash (\texttt{fn}\, x'{:}T'.t')\; v :: \varphi_1}\;\text{TE-APP} \qquad \Gamma, x{:}T \vdash t :: \varphi_t}{\Gamma \vdash \texttt{let}\, x{:}T = (\texttt{fn}\, x'{:}T'.t')\; v \;\texttt{in}\; t :: \varphi_1;\varphi_t}\;\text{TE-LET}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = (\texttt{fn}\, x'{:}T'.t')\; v \;\texttt{in}\; t\rangle :: p\langle\varphi_1;\varphi_t\rangle}$$

By the substitution from Lemma 7 on the left-most subgoal, we get $\Gamma \vdash t'[v/x'] :: \varphi_1$ and hence by TE-LET which gives:

$$\frac{\Gamma \vdash t'[v/x'] :: \varphi_1 \qquad \Gamma, x{:}T \vdash t :: \varphi_t}{\Gamma \vdash \texttt{let}\, x{:}T = t'[v/x'] \;\texttt{in}\; t :: \varphi_1;\varphi_t}\;\text{TE-LET}$$

We conclude the case by TE-THREAD.

Case: R-APP$_2$: $p\langle \mathtt{let}\, x{:}T = (\mathtt{fun}\, f{:}T'.x'{:}T_1.t')\ v\ \mathtt{in}\, t\rangle \xrightarrow{p\langle\tau\rangle} p\langle \mathtt{let}\, x{:}T = t'[v/x'][\mathtt{fun}\, f.x'.t'/f]\ \mathtt{in}\, t\rangle$
We are given $\Gamma \vdash p\langle \mathtt{let}\, x{:}T = (\mathtt{fun}\, f{:}T'.x'{:}T_1'.t')\ v\ \mathtt{in}\, t\rangle :: p\langle\varphi\rangle$, so inverting rules TE-THREAD, TE-LET, TE-APP, and TE-ABS$_2$ gives (where $\varphi = \varphi_1;\varphi_2$):

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma, x'{:}T_1, f{:}T_1 \to^{\varphi_1} T \vdash t' : T :: \varphi_1 \quad T' = T_1 \to^{\varphi_1} T_2 \quad T = T_2}{\Gamma \vdash \mathtt{fun}\, f{:}T'.x'{:}T_1.t' : T_1 \to^{\varphi_1} T_2 :: \varepsilon} \qquad \Gamma \vdash v :: \varepsilon}{\dfrac{\Gamma \vdash (\mathtt{fun}\, f{:}T'.x'{:}T_1.t')\ v :: \varepsilon;\varepsilon;\varphi_1}{\Gamma \vdash (\mathtt{fun}\, f{:}T'.x'{:}T_1.t')\ v :: \varphi_1}} \qquad \Gamma, x{:}T \vdash t :: \varphi_2}{\Gamma \vdash \mathtt{let}\, x{:}T = (\mathtt{fun}\, f{:}T'.x'{:}T_1.t')\ v\ \mathtt{in}\, t :: \varphi_1;\varphi_2}}{\Gamma \vdash p\langle \mathtt{let}\, x{:}T = (\mathtt{fun}\, f{:}T'.x'{:}T_1.t')\ v\ \mathtt{in}\, t\rangle :: p\langle\varphi_1;\varphi_2\rangle}$$

Using two times the substitution Lemma 7 on the left-most subgoal, we get $\Gamma \vdash t'[v/x'] :: \varphi_1$ and hence by TE-LET and TE-THREAD

$$\dfrac{\dfrac{\Gamma \vdash t'[v/x'][\mathtt{fun}\, f{:}T'.x'{:}T_1.t'/f] :: \varphi_1 \qquad \Gamma, x{:}T_2 \vdash t :: \varphi_2}{\Gamma \vdash \mathtt{let}\, x{:}T_2 = t'[v/x'][\mathtt{fun}\, f{:}T'.x'{:}T_1.t'/f]\ \mathtt{in}\, t :: \varphi_1;\varphi_2}\ \text{TE-LET}}{\Gamma \vdash p\langle \mathtt{let}\, x{:}T_2 = t'[v/x'][\mathtt{fun}\, f{:}T'.x'{:}T_1.t'/f]\ \mathtt{in}\, t\rangle :: p\langle\varphi_1;\varphi_2\rangle}$$

which concludes the case.

For part 2a, we are given $\sigma \vdash p\langle t\rangle \xrightarrow{p\langle a\rangle} \sigma' \vdash p\langle t'\rangle$.

Case: R-NEWL: $\sigma_1 \vdash p\langle \mathtt{let}\, x{:}T = \mathtt{new}_\pi\, L\ \mathtt{in}\, t\rangle \xrightarrow{p\langle \nu L^\pi\rangle} \sigma_1' \vdash p\langle \mathtt{let}\, x{:}T = l^\pi\ \mathtt{in}\, t\rangle$ where $\sigma_1' = \sigma_1[l^\pi \mapsto \mathit{free}]$ for a fresh l. By assumption, $\Gamma \vdash p\langle \mathtt{let}\, x{:}T = \mathtt{new}_\pi\, L\ \mathtt{in}\, t\rangle :: p\langle\varphi\rangle$. By inverting rules TE-THREAD and TE-LET, we get:

$$\dfrac{\dfrac{\Gamma \vdash \mathtt{new}_\pi\, L :: \nu L^\pi \qquad \Gamma, x{:}T \vdash t :: \varphi_t}{\Gamma \vdash \mathtt{let}\, x{:}T = \mathtt{new}_\pi\, L\ \mathtt{in}\, t :: \nu L^\pi;\varphi_t}\ \text{TE-LET}}{\Gamma \vdash p\langle \mathtt{let}\, x{:}T = \mathtt{new}_\pi\, L\ \mathtt{in}\, t\rangle :: p\langle\varphi\rangle}\ \text{TE-THREAD}$$

and $\varphi = \nu L^\pi;\varphi_t$. Using rules TE-LET and TE-THREAD again gives:

$$\dfrac{\dfrac{\Gamma \vdash l^\pi :: \varepsilon \qquad \Gamma, x{:}T \vdash t :: \varphi_t}{\Gamma \vdash \mathtt{let}\, x{:}T = l^\pi\ \mathtt{in}\, t :: \varphi_t}\ \text{TE-LET}}{\Gamma \vdash p\langle \mathtt{let}\, x{:}T = l^\pi\ \mathtt{in}\, t\rangle :: p\langle\varphi_t\rangle}\ \text{TE-THREAD}$$

By the assumption that $\sigma_2(\pi)$ is undefined, the case is concluded by rule RE-NEWL such that $\sigma_2 \vdash p\langle \nu L^\pi;\varphi_t\rangle \xrightarrow{\nu L^\pi} \sigma_2' \vdash p\langle\varphi_t\rangle$ where $\sigma_2' = \sigma_2[\pi \mapsto \mathit{free}]$.

Case: R-LOCK: $\sigma_1 \vdash p\langle \mathtt{let}\, x{:}T = l^\pi.\ \mathtt{lock}\ \mathtt{in}\, t\rangle \xrightarrow{p\langle L^\pi \mathtt{lock}\rangle} \sigma_1' \vdash p\langle \mathtt{let}\, x{:}T = l^\pi\ \mathtt{in}\, t\rangle$ where $\sigma_1(l^\pi) = \mathit{free}$ or $\sigma_1(l^\pi) = p(n)$, and $\sigma_1' = \sigma_1 + l_p$. Given that $\Gamma \vdash p\langle \mathtt{let}\, x{:}T = l^\pi.\ \mathtt{lock}\ \mathtt{in}\, t\rangle :: p\langle\varphi\rangle$, inverting rule TE-THREAD and followed by rules TE-LET and

TE-LOCK, we get:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash l^{\pi} :: \varepsilon}{\Gamma \vdash l^{\pi}.\,\texttt{lock}{::}\texttt{L}^{\pi}.\,\texttt{lock}}\ \text{TE-LOCK} \qquad \Gamma, x{:}T \vdash t :: \varphi_t}{\Gamma \vdash \texttt{let}\, x{:}T = l^{\pi}.\,\texttt{lock}\ \texttt{in}\, t :: \texttt{L}^{\pi}.\,\texttt{lock};\varphi_t}\ \text{TE-LET}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = l^{\pi}.\,\texttt{lock}\ \texttt{in}\, t\rangle :: p\langle \texttt{L}^{\pi}.\,\texttt{lock};\varphi_t\rangle}\ \text{TE-THREAD}$$

and φ is $\texttt{L}^{\pi}.\,\texttt{lock};\varphi_t$ where $\texttt{L}^{\{\pi\}}.\,\texttt{lock}$ is written as $\texttt{L}^{\pi}.\,\texttt{lock}$ for short. By applying TE-LET and TE-THREAD, we get:

$$\dfrac{\dfrac{\Gamma, x{:}T \vdash t :: \varphi_t}{\Gamma \vdash \texttt{let}\, x{:}T = l^{\pi}\ \texttt{in}\, t :: \varphi_t}\ \text{TE-LET}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = l^{\pi}\ \texttt{in}\, t\rangle :: p\langle \varphi_t\rangle}\ \text{TE-THREAD}$$

Then, we conclude the case by RE-LOCK$_1$ and RE-SEQ

$$\sigma_2 \vdash p\langle \texttt{L}^{\{\pi\}}.\,\texttt{lock};\varphi_t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_2 \vdash p\langle \texttt{L}^{\pi}.\,\texttt{lock};\varphi_t\rangle \xrightarrow{p\langle \texttt{L}^{\pi}\texttt{.lock}\rangle} \sigma_2' \vdash p\langle \varphi_t\rangle$$

for $\sigma_2' = \sigma_2 + \pi_p$. The case for R-UNLOCK works analogously.

For spawn-steps in part 2b, we are given $\sigma \vdash p\langle t\rangle \xrightarrow{p\langle a\rangle} \sigma \vdash p\langle t''\rangle \parallel p'\langle t'\rangle$ where $a = p\langle \texttt{spawn}\ \varphi'\rangle$.

Case: R-SPAWN: $\sigma_1 \vdash p\langle \texttt{let}\, x{:}T = \texttt{spawn}\, t'\ \texttt{in}\, t''\rangle \xrightarrow{p\langle \texttt{spawn}\ \varphi'\rangle} \sigma \vdash p\langle \texttt{let}\, x{:}T = p'\ \texttt{in}\ t''\rangle \parallel p'\langle t'\rangle$
Given the well-typedness assumption $\Gamma \vdash p\langle \texttt{let}\, x{:}T = \texttt{spawn}\, t'\ \texttt{in}\, t''\rangle :: p\langle\varphi\rangle$, inverting rules TE-THREAD, TE-LET, and TE-SPAWN gives:

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash t' :: \varphi'}{\Gamma \vdash \texttt{spawn}\, t' :: \texttt{spawn}\ \varphi';}\ \text{TE-SPAWN} \qquad \Gamma, x{:}T \vdash t'' :: \varphi''}{\Gamma \vdash \texttt{let}\, x{:}T = \texttt{spawn}\, t'\ \texttt{in}\, t'' :: \texttt{spawn}\ \varphi';\varphi''}\ \text{TE-LET}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = \texttt{spawn}\, t'\ \texttt{in}\, t''\rangle :: p\langle \texttt{spawn}\ \varphi';\varphi''\rangle}\ \text{TE-THREAD}$$

where $\varphi = \texttt{spawn}\ \varphi';\varphi''$. Using rule TE-LET, TE-THREAD, and TE-PAR gives:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma \vdash p' :: \varepsilon \quad \Gamma, x{:}T \vdash t'' :: \varphi''}{\Gamma \vdash \texttt{let}\, x{:}T = p'\ \texttt{in}\, t'' :: \varepsilon;\varphi''}\ \text{TE-LET}}{\Gamma \vdash \texttt{let}\, x{:}T = p'\ \texttt{in}\, t'' :: \varphi''}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = p'\ \texttt{in}\, t''\rangle :: p\langle\varphi''\rangle}\ \text{TE-THREAD} \qquad \dfrac{\Gamma \vdash t' :: \varphi'}{\Gamma \vdash p'\langle t'\rangle :: p'\langle\varphi'\rangle}\ \text{TE-THREAD}}{\Gamma \vdash p\langle \texttt{let}\, x{:}T = p'\ \texttt{in}\, t''\rangle \parallel p'\langle t'\rangle :: p\langle\varphi''\rangle \parallel p'\langle\varphi'\rangle}$$

From that the case follows by rule RE-SPAWN: $\sigma_2 \vdash p\langle \texttt{spawn}\ \varphi';\varphi''\rangle \xrightarrow{p\langle \texttt{spawn}\ \varphi'\rangle} \sigma_2 \vdash p\langle\varphi''\rangle \parallel p'\langle\varphi'\rangle$.

For part 3, we are given $waits(\sigma_1 \vdash p\langle t\rangle, p, l^\pi)$, i.e., by Definition 1 it is not the case that $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle \mathtt{L}^l\mathtt{lock}\rangle}$ but $\sigma_1' \vdash p\langle t\rangle \xrightarrow{p\langle \mathtt{L}^l\mathtt{lock}\rangle}$ for some heap σ_1'. This implies that the thread t is of the form $\mathtt{let}\, x{:}T = l^\pi.\,\mathtt{lock}\;\mathtt{in}\, t'$ and we are given more specifically that $\sigma_1 \vdash p\langle \mathtt{let}\, x{:}T = l^\pi.\,\mathtt{lock}\;\mathtt{in}\, t'\rangle \not\xrightarrow{p\langle \mathtt{L}^l\mathtt{lock}\rangle}$. The well-typedness assumption $\Gamma \vdash p\langle \mathtt{let}$ $x{:}T = l^\pi.\,\mathtt{lock}\;\mathtt{in}\, t'\rangle :: p\langle\varphi\rangle$ gives that φ is of the form $\mathtt{L}^r.\mathtt{lock};\varphi_t$ and φ_t is the effect of thread t' and where $\pi \in r$. Then $\sigma_2 \vdash p\langle \mathtt{L}^r.\mathtt{lock};\varphi_t\rangle$ is first reduced to $\sigma_2 \vdash p\langle \mathtt{L}^\pi.\mathtt{lock};\varphi_t\rangle$ by rule RE-LOCK$_1$ with a τ-step. The execution continues with RE-LOCK$_2$. Since $\sigma_1 \equiv \sigma_2$, we get $\sigma_2 \vdash p\langle \mathtt{L}^r.\mathtt{lock};\varphi_t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_2 \vdash p\langle \mathtt{L}^\pi.\mathtt{lock};\varphi_t\rangle \not\xrightarrow{p\langle \mathtt{L}^\pi\mathtt{lock}\rangle}$. Since for any heap σ_2' where the lock π is free, $\sigma_2' \vdash p\langle \mathtt{L}^\pi.\mathtt{lock};\varphi_t\rangle \xrightarrow{p\langle \mathtt{L}^\pi.\mathtt{lock}\rangle}$, we have $waits(\sigma_2 \vdash p\langle \mathtt{L}^\pi.\mathtt{lock};\varphi_t\rangle, p, \pi)$, which concludes the case. $\square$

An easy consequence is that well-typedness relation between a program and its effect is a deadlock-preserving simulation:

Corollary 1. *Given $\sigma_1 \equiv \sigma_2$ and $\Gamma \vdash p\langle t\rangle :: p\langle\varphi\rangle$, then $\sigma_1 \vdash p\langle t\rangle \lesssim^D \sigma_2 \vdash p\langle\varphi\rangle$.*

4 Two finite abstractions

In this section, we describe finite abstractions on the effects so that we can *effectively* check for potential deadlocks on the abstract level. The two sources of infinity we tackle are the unboundedness of the lock counters and the unboundedness of the "control stack" of recursive behavior descriptions. The reason for the latter is that the syntax of the behaviors includes sequential composition of behaviors, which allows to capture non-tail-recursive $recX.\varphi$. In the next section, we collapse the lock counters into a finite over-approximation, and in Section 4.2, we show how to transform the behavior representation into a tail-recursive one, necessarily loosing again precision. For both abstractions, we prove that the abstracted system simulates the concrete one, via the deadlock-sensitive relation $\lesssim^D$, i.e., the abstractions are sound.

4.1 Lock counters abstraction

The unbounded lock counters are the first source of infinity. We over-approximate the behavior by collapsing (for a given lock) all lock counts over a threshold M into one. That abstraction naturally introduces non-determinism. The lock-counters in rules RE-LOCK$_2$ and RE-UNLOCK$_2$ increases resp. decreases by one. We used the two *functions* $\sigma + \pi_p$ and $\sigma - \pi_p$ for that. With M as upper bound functions are then changed as follows. Let $\sigma' = \sigma + \pi_p$. Now, if $\sigma(\pi) < M$, then $\sigma' = \sigma[\pi \mapsto \sigma(\pi)+1]$. If $\sigma(\pi) = M$, then $\sigma' = \sigma$. The corresponding decreasing operation $\sigma - \pi_p$ now generalized to a *relation*, i.e., $\sigma - \pi_p$ is given as a *set* as follows: If $\sigma(\pi) = M$, then $\sigma - \pi_p = \{\sigma, \sigma[\pi \mapsto \sigma(\pi)-1]\}$. If $\sigma(\pi) < M$, then $\sigma - \pi = \{\sigma[\pi \mapsto \sigma(\pi)-1]\}$. To reflect the non-determinism, the premise $\sigma' = \sigma - \pi$ of rule RE-UNLOCK$_2$ needs to be generalized to $\sigma' \in \sigma - \pi$. The value of M is from the range $\{1, \ldots, \infty\}$, where ∞ means that the counter is unbounded. To be able to distinguish a free lock from a lock which is taken, the lowest value for the upper bound M we consider is 1.

The next lemma expresses an easy fact about the $\equiv$-relation, in particular that changing to an equivalent heap does not change the fact whether a process is waiting on a lock or not.

Lemma 9. *Assume $\sigma_1 \equiv \sigma_2$ with $\theta = id$.*

1. *If $waits(\sigma_1 \vdash \Phi, p, \pi)$, then $waits(\sigma_2 \vdash \Phi, p, \pi)$.*
2. *If $\sigma_1 \vdash \Phi \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash \Phi'$, then $\sigma_2 \vdash \Phi \xrightarrow{p\langle\tau\rangle} \sigma_2 \vdash \Phi'$.*

Proof. Immediate. $\square$

Lemma 10 (Lock counter abstraction). *Given a configuration $\sigma \vdash \Phi$, and let further denote $\sigma_1 \vdash^{n_1} \Phi$ and $\sigma_2 \vdash^{n_2} \Phi$ the corresponding configurations under the lock-counter abstraction. If $n_1 \geq n_2$, then $\sigma_1 \vdash^{n_1} \Phi \lesssim^D \sigma_2 \vdash^{n_2} \Phi$ (where n_1 and n_2 are $\in \{1, \dots, n, \dots, \infty\}$).*

Proof. We prove more specifically that $\sigma_1 \vdash^{n+1} \Phi \lesssim^D \sigma_1 \vdash^n \Phi$. We omit the case where $n_1 = \infty$, which works analogously. The result follows by transitivity and reflexivity.

Define the binary relation R between configurations as follows: $\sigma_1 \vdash \Phi_1 \; R \; \sigma_2 \vdash \Phi_2$ if $\Phi_1 = \Phi_2$ and $\sigma_1 = \sigma^{n+1}$ and $\sigma_2 = \sigma^n$; in abuse of notation, we write $\sigma_1 \; R \; \sigma_2$ also for the heap-part of the definition. Note further that for the heap-part, $\sigma_1 \; R \; \sigma_2$ implies $\sigma_1 \equiv \sigma_2$.

Obviously, the start configuration is in that relation.

Case: $\sigma_1 \vdash^{n+1} \Phi \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash^{n+1} \Phi'$
By Lemma 9(2), $\sigma_2 \vdash^n \Phi \xrightarrow{p\langle\tau\rangle} \sigma_2 \vdash^n \Phi'$. Case 3 of Definition 4 is covered by Lemma 9(1).

Case: If $\sigma_1 \vdash^{n+1} \Phi \xrightarrow{p\langle a\rangle} \sigma_1' \vdash^{n+1} \Phi'$
The only interesting cases are the ones for locking and unlocking. In the following we elide mentioning p from the operation $\sigma + \pi_p$.

Subcase: `L`$^\pi$`.lock`
In this case $\sigma_1' = \sigma_1 + \pi$, i.e., $\sigma_1'(\pi) = \sigma_1'(\pi) +_{n+1} 1$ (where $+_{n+1}$ is addition modulo the upper bound $n+1$). For $\sigma_2 \vdash^n \Phi$, we have that $\sigma_2 \vdash^n \Phi \xrightarrow{p\langle \mathtt{L}^\pi\mathtt{lock}\rangle} \sigma_2' \vdash^n \Phi'$, where $\sigma_2'(\pi) = \sigma_2(\pi) +_n 1$. Thus $\sigma_1' \; R \; \sigma_2'$. Part 3 of Definition 4 follows straightforwardly from the definition of R, in particular since R implies $\equiv$.

Subcase: `L`$^\pi$`.unlock`
It is straightforward to check analogously that R satisfies the conditions for a simulation relation also for unlocking. For an unlocking step, we can distinguish two cases for the post-configurations of $\sigma_1 \vdash^{n+1} \Phi$. If $\sigma_1(\pi) < n+1$, then the step is deterministic, i.e., $\sigma_1' \in \{\sigma_1[\pi \mapsto \sigma_1(\pi) - 1]\}$. In this case, there exists a transition $\sigma_2 \vdash^n \Phi \xrightarrow{p\langle \mathtt{L}^\pi\mathtt{.unlock}\rangle} \sigma_2' \vdash^n \Phi'$ where $\sigma_2'(\pi) = \sigma_2(\pi) - 1$, and hence $\sigma_1' \; R \; \sigma_2'$. Otherwise, if $\sigma_1(\pi) = n+1$, then $\sigma_1' \in \{\sigma_1, \sigma_1[\pi \mapsto \sigma_1(\pi) - 1]\}$. For this case, we choose $\sigma_2' = \sigma_2$, thus $\sigma_1' \; R \; \sigma_2'$. Note that condition 3 of Definition 4 is trivially satisfied as unlocking does not wait for a lock. $\square$

4.2 Tail-recursive behavior representation

A second source of infinity in the state space is recursion: the behavior contains *non*-tail-recursive descriptions. We deal with it in the same way as we did for the lock-counters: we allow a certain recursion depth —the choice of cut-off does not matter— after which the behavior is over-approximated. Just as with the variable upper limit on the lock count, we use a similar adjustable limit for the stack depth. Once the recursion limit is reached, the behavior becomes chaotic, i.e., it over-approximates all behavior. In the following, we make use of the fact that our deadlock analysis is limited to programs that *do not* recursively create new resources. For instance, in the definition of the chaotic behavior, we exclude lock creation and thread creation labels. So Ω randomly takes and releases locks, does internals steps, or terminates at arbitrary points. Note that Ω is tail-recursive.

Definition 5 (Chaotic behavior Ω). *Given a set of locations r, we define*

$$\Omega(r) = rec X.\varepsilon + X + \mathtt{L}^r\mathtt{.lock};X + \mathtt{L}^r\mathtt{.unlock};X \ .$$

We write Ω, if r is clear from the context.

Lemma 11 (Ω is maximal wrt. $\lesssim^{DT}$). *Assume φ over a set of locations r, then $\sigma \vdash p\langle\varphi\rangle \lesssim^{DT} \sigma \vdash p\langle\Omega\rangle$.*

Proof. We define a simulation relation R between $\sigma \vdash p\langle\varphi\rangle$ and $\sigma \vdash p\langle\Omega\rangle$ as follows. The states of Ω are shown schematically in Figure 4. The initial one corresponds to the term Ω. The four τ^*-transitions (including the "self-loop") originating from the initial state are caused by resolving the choice (cf. rule RE-CHOICE). And the states s_1 and s_2 correspond to the expressions $\mathtt{L}^r\mathtt{.lock};\Omega$ and $\mathtt{L}^r\mathtt{.unlock};\Omega$. The labels l_i and u_j in the picture correspond to lock-manipulating labels $p\langle\mathtt{L}^{\pi_i}\mathtt{.lock}\rangle$ and $p\langle\mathtt{L}^{\pi_j}\mathtt{.unlock}\rangle$. Note that may take more than one τ step to go from, for example Ω to s_1. This is due to the fact that resolving a choice costs a τ step and that we consider behavior up-to $\equiv$.[5]

The simulation relation R then couples Ω and processes $p\langle\varphi\rangle$ in an obvious manner. A φ of the form ε is related to ε of Ω, if $\varphi = \mathtt{L}^r\mathtt{.lock};\varphi'$, then $\varphi\ R\ s_1$, analogously for unlocking. If $\varphi = \mathtt{L}^{\pi_i}\mathtt{.lock};\varphi'$, then φ is related to the corresponding state of Ω in front of the lock-taking step. In all other cases, φ is related via R to the initial state Ω. It is straightforward to see that R is indeed a deadlock and termination sensitive simulation between $p\langle\varphi\rangle$ and Ω. □

We define the *depth-k-unrolling* of an effect, where we substitute the recursive invocation by Ω at recursion depth k.

Definition 6 (Unrolling). *Given an effect $\varphi = rec X.\varphi'$ with locations r, we define the* depth-k-unrolling *φ^k inductively as follows:*

$$\begin{aligned} \varphi^0 &= \Omega(r) \\ \varphi^{n+1} &= \varphi'[\varphi^n/X] \ . \end{aligned}$$

[5] The τ^+ is slightly imprecise, the maximal number of τ is determined by the number of summands in Ω, i.e. ultimately, the number of locks in r . Important, however, is that that all transitions originating from Ω start with one silent step, i.e., the choice is internal.

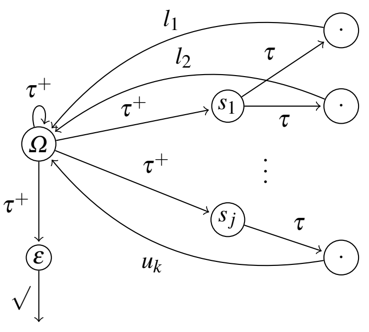

Fig. 4. Chaotic process Ω

The definition allows unrolling of a recursive behavior; we use it also to unroll a recursion inside a behavior expression. We write $\varphi_1[\varphi_2^n]$ for unrolling φ_2 inside the "context" φ_1. For simplicity we will assume that the position(s) $[]$ where we do that replacement in $\varphi_2[]$ do not occur inside a further recursion in the context. When later abstracting recursive behavior by unrolling, we can treat the recursions proceeding from the outer recursion to the inner ones.

Note further that unrolling to Ω is quite coarse and can easily be refined. A straightforward improvement in terms of precision while preserving soundness would be to split the set of locks into two sets, one for those that are used in locking effects, and one for those that are used in unlocking. These two sets can easily be derived from the effect being unrolled.

Lemma 12 (Behavior abstraction). *Given a configuration $\sigma \vdash \Phi$. Let Φ^m denote the m-unrolling of a specific occurrence of a recursion $recX.\varphi$ in Φ not occurring inside another recursion. If $m_1 \geq m_2$, then $\sigma \vdash \Phi^{m_1} \lesssim^{DT} \sigma \vdash \Phi^{m_2}$. The lemma holds identical for configurations based on the lock counter abstraction of Section 4.1.*

Proof. We prove specifically that $\sigma \vdash \Phi^{m+1} \lesssim^{DT} \sigma \vdash \Phi^m$ (where m is a natural number ≥ 0), and the result follows by transitivity and reflexivity. The case where $m_1 = \infty$ works similarly. So, Φ is of the form $\Phi[\varphi] = \Phi[recX.\ \varphi']$, where $\varphi = recX.\ \varphi'$ is the occurrence of the recursion being unrolled. By definition of unrolling it means that $\Phi^m = \Phi[\varphi^m]$ and analogously $\Phi^{m+1} = \Phi[\varphi^{m+1}]$. That further means for the form of Φ^m resp. of Φ^{m+1}

$$\Phi^m = \Psi[\Omega] \quad \text{and} \quad \Phi^{m+1} = \Psi[\varphi'[\Omega/X]] \tag{13}$$

for some $\Psi[]$. So the result follows by maximality of Ω from Lemma 11, and using the context Lemma 3 and Lemma 2 (for parallel composition). It is immediate to see that the required lemmas 11, 3, and Lemma 2 work identically under the assumption that some lock counters are abstracted. $\square$

We have shown already in the first part of the paper that the effect derived from type-checking preserves deadlocks. Next, we state the final theorem with regard to our contribution. It shows that we can conclude from the absence of a deadlock in effect checking with regard to lock-counter abstraction and behavior abstraction that the program is deadlock-free:

Theorem 1 (Soundness of the abstraction). *Given $\Gamma \vdash P : ok :: \Phi$ and two heaps $\sigma_1 \equiv \sigma_2$. Further, $\sigma_2' \vdash \Phi'$ is obtained by lock-counter resp. behavior abstraction of $\sigma_2 \vdash \Phi$. Then if $\sigma_2' \vdash \Phi'$ is deadlock free then so is $\sigma_1 \vdash P$.*

Proof. By Corollary 1, Lemmas 10 and 12, and with the help of transitivity of $\lesssim^D$. □

Theorem 2 (Finite abstractions). *The lock counter abstraction and behavior abstraction (when abstracting all locks and recursions) results in a finite state space.*

Proof. Under our restrictions —no lock and thread creations within recursions— the contribution of the heap to the state space is finite (due to lock counter abstraction). Further, the behavior abstraction renders the behavior Φ of the program into a bounded number of *tail-recursive* processes. Both together result in a finite reachable state space. □

5 Conclusion

We have presented a type and effect system that derives abstract behaviors from a core functional language with lock-based concurrency. Such an abstract behavior can be executed, and the resulting state space checked for deadlocks. The potentially infinite state space is abstracted in two ways: we place a user-definable upper bound on the lock-counters, and a similar limit on the recursion depth for non-tail-recursive function calls; beyond that chosen limit, the behavior is over-approximated by arbitrary behavior. This abstraction yields a finite state space that can be exhaustively checked for deadlocks.

We show soundness of the abstractions with regard to a deadlock- and termination-sensitive simulation of the original program, i.e., a program is deadlock free, if the abstraction is deadlock free. Using an over-approximation, the converse does not hold: a deadlock in the abstraction not necessarily represents a deadlock in the concrete program.

Being based on abstraction and state exploration, a reported deadlock on the abstract level can be mapped back to the original program by looking at the path labelled with the concurrent actions from the initial state to the deadlocked state. This may provide the user with intuition about whether he should refine the parameters for the abstraction. For example, the model could be easily augmented to indicate whether the lock-statement involved in a deadlock is the result of introducing an Ω. A natural extension of our work would thus be to counter-example guided abstraction refinement (CEGAR) [29,9].

Another straightforward increase in precision can be obtained through *type inference*: currently, explicit typing means that function declarations in our system need to be declared with the most general type. In the case of a parameter of a lock-type, this means that the corresponding region on the argument has to be declared as the union of the regions at the call-sites, leading to a loss of precision. As usual, with type inference and polymorphism, each invocation could be checked separately in the context of its caller. Obviously, effect inference would be welcome from a practical point of view that the effects could be automatically inferred.

For a practical application, not every program will fit our restriction of no recursive resource creation (threads/locks). Especially for programming languages that facilitate light-weight/“disposable” thread creation, such as Erlang or Concurrent Haskell, our analysis would be of limited use. In such cases, we may still be able to use our analysis to partition the problem into a part that can be statically tackled with our approach, and subject the remainder to a dynamic monitoring regime that will report locking-violations or warnings (see e.g. [35] below).

We plan to investigate how further static analysis techniques can help to eliminate doubts in such a more dynamic setting: if from our effect system we can tell that dynamically generated threads never share more than one lock, then we should be easily able to extend the range of acceptable input to our analysis.

Related work Deadlocks are a common problem in concurrent programming. Cyclic waiting has been identified early as one of four necessary conditions for a deadlock [11]. To tackle the problem of deadlocks, one traditionally istinguishes different approaches [14]: deadlock *prevention*, *avoidance*, *detection,* and *recovery*. A static type system like the one presented here would classify as deadlock prevention; avoidance, in contrast, refers to techniques that “dodge” looming deadlocks at run-time.

As said, a necessary condition for a deadlock to occur is the cyclic wait on (non-preemptive) resources, such as locks as in our case, but also waiting for channel communication (“communication deadlock”) and other resources may lead to deadlock. Therefore the most common way to prevent deadlocks is to statically make sure that such cycles on locks or resources in general can never occur. This can be done by arranging (classes of) locks in some partial order and enforcing that the locks are taken in accordance with that order. That old and straightforward idea has, for instance, be formalized in a type-theoretic setting in the form of deadlock types [6]. The static system presented in [6] supports also type *inference* (and besides deadlocks, prevents race conditions, as well). Deadlock types are also used in [1], but not for static deadlock prevention, but for improving the efficiency for deadlock avoidance at run-time.

Static analyses and type systems to prevent especially communication deadlocks have been studied for various process algebras, in particular for the π-calculus, where the dynamically changing communication structure makes preventing deadlock situations challenging [25,28,26,27]. Also for dynamically changing communication structures, [16] presents a type-based analysis for the prevention of deadlocks in a setting based on channel communication and message passing. The cause of deadlocks in the setting there is different, deadlocks are not caused by the attempt to acquire locks, but by communication over channels which may introduce wait cycles. One challenge there is that the communication topology may change dynamically. [22] propose a general framework for type system in the context of the π-calculus, which can be used to check deadlocks, live-locks, or race-freedom. Session types, a type-based abstract behavioral description of concrete behavior, typically for channel-based communication, have also been used for deadlock detection [7]. Model checking [10], i.e., the automatic state exploration (of a model) of a program has been used, as well, for deadlock detection. [12]

presents an empirical study comparing different model checkers and model checking techniques for detecting deadlocks (for Ada programs). To defuse the danger of cyclic wait, the above approaches rely on enforcing an order on *locks/resources,* respectively inferring that such an order exists. Ordering (classes of) locks is not the only way to break (potential) cycles. For the process algebra CSP, [34] propose to come up with a well-founded order attached to the *states* of the interacting processes in such a way, that if a process is waiting for another process, the value of the state of the waiting process is larger than the state of the process it waits for. The approach is a generalization (to networks of processes) of the *"variant"* proof method for establishing termination for loops.

Another notorious kind of error in shared variable concurrent programming are *race* conditions, i.e., the unprotected, simultaneous access to a shared resource. Whereas a deadlock may occur when communicating partners disagree on the *order* of lock-taking when simultaneously accessing more than one shared lock, a race results when the partners *fail* to take a lock before competing for a shared resource, or rather that the critical resource fails to be protected properly by a lock or other synchronization mechanism. The concurrency errors of deadlocks and of race conditions can be seen as related also in the following way: One may consider parts of the programs where lock interaction with conflicting orders may occur as "critical regions" where a potential "race" may occur. In the same way that, in a lock-based setting, races can be prevented by protecting the shared data, one can add additional locks ("gate locks") to protect pairs or sets of locks from potential deadlocks. Checking for potential race conditions has been widely studied, for instance using ownership types [5], fractional permissions and linear programming [36]. For static techniques assuring race freedom, it mostly amounts to check or infer that "enough" locks are held by a thread or process before accessing shared data. Such lock sets are used, e.g., in [18,19,20,32]. The type systems of [18,17], using singleton "lock types" as a restricted form of dependent types offer, in an extension, also protection against deadlocks. Often, the analyses are made more precise by combining them with alias analysis, or taking "ownership" concepts into account.

In our approach, we avoid the infinite state space caused by recursion, by approximating it by a tail-recursive approximation. Other approaches use language- or automata-theoretic decidability results to keep the stack-structure but achieve a finite-state representation nonetheless. For instance, [13] uses a specific class for push-down automata closed under products and for which reachability is decidable for deadlock checking for call-graph abstractions for multi-threaded Java programs. [24] gives a precise analysis for nested locking of binary locks for push-down systems, without dynamic lock- or thread creation.

[15] uses an unsound and incomplete static analysis of C programs to extract lock dependency constraints. They analyze the dependencies for circular waiting, and focus on reducing false positives. The tool has discovered numerous bugs in operating systems source code.

Our work puts an upper bound on the number of threads and the number of locks, as we disallow to spawn new activities resp. create new locks inside recursions. In contrast to our approach, [4] use *symbolic evaluation* to allow an unbounded number of task for deadlock detection for Ada programs by assigning symbolic task identifiers to each task at creation, though it is unclear that how dynamic shared resources are handled.

[35] in the context of run-time verification (or checking) observes lock chains in a (concrete) execution trace by means of a parametrized LTL formula and issues a warning if different lock-orders are observed which may potentially lead to a deadlock. Placeholders for e.g. thread and lock identifiers are bound by propositions for locking and unlocking. [8] speculatively execute Concurrent Haskell programs in the search for deadlocks, where all interleavings of concurrent threads are evaluated until the execution would have to commit to an I/O action with the environment (user). As their analysis takes concrete values into accounts, their analysis provides precise results *for a particular run* (input values), and only creates a bounded number of resources if the original program creates a bounded number of resources. They also discuss partial-order techniques for state space-reduction. A non-interactive Concurrent Haskell program with initial inputs will effectively be subjected to an explicit state space exploration modulo the above reduction.

References

1. R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with state analysis and run-time monitoring. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Proceedings of the Haifa Verification Conference 2005*, volume 3875 of *Lecture Notes in Computer Science*, pages 191–207. Springer-Verlag, 2006.
2. T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
3. J. Baeten and R. van Glabbeek. Merge and termination in process algebra. In K. V. Nori, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 287 of *Lecture Notes in Computer Science*, pages 153–172. Springer-Verlag, 1987. Also as CWI report CS-R8716.
4. J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic data flow analysis for detecting deadlocks in Ada tasking programs. In *Proceedings of the 5th Ada-Europe International Conference on Reliable Software Technologies*, volume 1845 of *Lecture Notes in Computer Science*, pages 225–237. Springer-Verlag, 2000.
5. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '02 (Seattle, USA)*. ACM, Nov. 2002. In *SIGPLAN Notices*.
6. C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation (San Diego, California)*. ACM, June 2003.
7. R. Bruni and L. G. Mezzina. Types and deadlock freedom on calculus of services and sessions. In J. Meseguer and G. Rosu, editors, *Proceedings of AMAST'08*, volume 5140 of *Lecture Notes in Computer Science*, pages 100–115. Springer-Verlag, 2008.
8. J. Christiansen and F. Huch. Searching for deadlocks while debugging concurrent Haskell programs. In C. Okasaki and K. Fisher, editors, *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 28–39. ACM, 2004.
9. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In E. A. Emerson and A. P. Sistla, editors, *Proceedings of CAV '00*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer-Verlag, 2000.
10. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

11. E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.
12. J. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, Mar. 1996.
13. F. S. de Boer and I. Grabe. Automated deadlock detection in synchronized reentrant multithreaded call-graphs. In J. van Leeuwen, A. Muscholl, D. Peleg, J. Pokorný, and B. Rumpe, editors, *SOFSEM 2010: Theory and Practice of Computer Science, 36th Conference on Current Trends in Theory and Practice of Computer Science, Spindleruv Mlýn, Czech Republic, January 23-29, 2010. Proceedings*, volume 5901 of *Lecture Notes in Computer Science*, pages 200–211. Springer-Verlag, 2010.
14. H. M. Deitel. *An Introduction to Operating Systems*. Addison-Wesley, revised 1st edition, 1984.
15. D. R. Engler and K. Ashcraft. Effective, static detection of race conditions and deadlocks: RacerX. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.
16. M. Fähndrich, S. K. Rajamani, and J. Rehof. Static deadlock prevention in dynamically configured communication network. In *Perspectives in Concurrency, Festschrift in Honor of P.S. Thiagarajan*, pages 128—156, 2008.
17. C. Flanagan and M. Abadi. Object types against races. In J. C. Baeten and S. Mauw, editors, *Proceedings of CONCUR '99*, volume 1664 of *Lecture Notes in Computer Science*, pages 288–303. Springer-Verlag, Aug. 1999.
18. C. Flanagan and M. Abadi. Types for safe locking. In S. Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 91–108. Springer, 1999.
19. C. Flanagan and S. Freund. Type-based race detection for Java. In *Proceedings of PLDI'00, ACM SIGPLAN Conference on ACM Conference on Programming Language Design and Implementation*, pages 219–232, 2000.
20. D. Grossman. Type-safe multithreading in Cyclone. In *TLDI'03: Types in Language Design and Implementation*, pages 13–25, 2003.
21. R. C. Holt. *On Deadlock in Computer Systems*. PhD thesis, Cornell University, Ithaca, NY, USA, 1971.
22. A. Igarashi and N. Kobayashi. A generic type system for the pi-calculus. In *Proceedings of POPL '01*, pages 128–141. ACM, Jan. 2001.
23. The Java tutorials: Concurrency. Available at `download.oracle.com` under `/javase/tutorial/essential/concurrency`, 2011.
24. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In K. Etessami and S. K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 505–518. Springer-Verlag, 2005.
25. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, 1998. An extended abstract previously appeared in the *Proceedings of LICS '97*, pages 128–139.
26. N. Kobayashi. Type-based information flow analysis for the π-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.
27. N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of CONCUR 2006*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer-Verlag, 2006.
28. N. Kobayashi, S. Saito, and E. Sumii. An implicitly-typed deadlock-free process calculus. In C. Palamidessi, editor, *Proceedings of CONCUR 2000*, volume 1877 of *Lecture Notes in Computer Science*, pages 489–503. Springer-Verlag, Aug. 2000.
29. R. Kurshan. *Computer-Aided Verification of Coordinating Processes, the automata theoretic approach*. Princeton Series in Computer Science. Princeton University Press, 1994.

30. G. N. Levine. Defining deadlock. *SIGOPS Oper. Syst. Rev.*, 37(1):54–64, 2003.
31. R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann, 1971.
32. M. Naik and A. Aiken. Conditional must not aliasing for static race detection. In *Proceedings of POPL '07*. ACM, Jan. 2007.
33. F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.
34. B. Roscoe and N. Dathi. The pursuit of deadlock freedom. *Information and Computation*, 75(3):289–327, Dec. 1987.
35. V. Stolz. Temporal assertions with parametrized propositions. *J. Log. Comput.*, 20(3):743–757, 2010.
36. 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.
37. R. van Glabbeek and U. Goltz. Refinement of actions and equivalence notions for concurrent systems. *Acta Informatica*, 37(4/5):229–327, 2001.
38. A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In A. P. Black, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages (ECOOP 2005)*, volume 3586 of *Lecture Notes in Computer Science*. Springer, 2005.