Effect-Polymorphic Behaviour Inference for Deadlock Checking[☆,†]

Ka I Pun[a], Martin Steffen[a], Volker Stolz[a,b]

[a]*Dept. of Informatics, University of Oslo, Norway*
[b]*Bergen University College, Norway*

Abstract

We present a constraint-based effect inference algorithm for deadlock checking. The static analysis is developed for a concurrent calculus with higher-order functions and dynamic lock creation, where the locks are summarised based on their creation-site. The analysis is context-sensitive and the resulting effects can be checked for deadlocks using state space exploration. We use a specific deadlock-sensitive simulation relation to show that the effects soundly over-approximate the behaviour of a program, in particular that deadlocks in the program are preserved in the effects.

Keywords: deadlock detection, simulation, type and effect system, concurrency, formal method

1. Introduction

Deadlocks are a common problem for concurrent programs with shared resources. According to the classic characterization from [11], a deadlocked state is marked by a number of processes, which forms a cycle where each process is unwilling to release its own resource, and is waiting on the resource held by

[☆]Partly funded by the EU projects FP7-610582 ENVISAGE: Engineering Virtualized Services (`http://www.envisage-project.eu`) and FP7-612985 UPSCALE: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (`http://www.upscale-project.eu`).

[†]This article is a full version of the extended abstract presented at the 25th Nordic Workshop on Programming Theory, NWPT 2013, in Tallinn.

Email addresses: `violet@ifi.uio.no` (Ka I Pun), `msteffen@ifi.uio.no` (Martin Steffen), `stolz@ifi.uio.no` (Volker Stolz)

its neighbour. The inherent non-determinism makes deadlocks, as other errors in the presence of concurrency, hard to detect and to reproduce. We present a static analysis using behavioural effects to detect deadlocks in a higher-order concurrent calculus. Deadlock freedom, an important safety property for concurrent programs, is a thread-global property, i.e., the blame for a deadlock in a defective program cannot be put on a single thread, it is two or more processes that share the responsibility; the somewhat atypical situation, where a process forms a deadlock with itself, cannot occur in our setting, as we assume re-entrant locks. The presented approach works in two stages. The first stage, which is the focus of this paper, corresponds to *model extraction*: an effect-type system uses a static behavioural abstraction of the codes' behaviour, concentrating on the lock interactions. To analyse the consequences on the global level, in particular for detecting deadlocks, the combined individual abstract thread behaviours are explored in the second stage.

Two challenges need to be tackled to make the approach applicable in practice. For the first stage on the thread local level, the model extraction, the static analysis must be able to *derive* the abstract behaviour, not just check compliance of the code with a user-provided description. This is the problem of type and effect *inference* or reconstruction. As usual, the abstract behaviour needs to over-approximate the concrete one, i.e., concrete and abstract descriptions are connected by some *simulation* relation: everything the concrete system does, the abstract one can do as well (modulo some abstraction function relating the concrete and abstract states). For the second stage, exploring the (abstract) state space on the global level, obtaining *finite* abstractions is crucial. In our setting, there are four principal sources of infinity: the calculus allows 1) recursion, supports 2) dynamic thread creation, as well as 3) dynamic lock creation, and 4) with re-entrant locks, where the lock counters are unbounded. To allow static checking, appropriated abstractions, especially to tame the unbounded size of the mentioned dynamic aspects of the language. Our paper focuses on the model extraction in the first stage and how to infer the behavioural model and the role of polymorphism. This model extraction stage includes dealing with dynamic lock creation, as well. The model exploration in the second stage is covered in our previous work [43], which offers sound abstractions for lock counters and for recursion (but not for dynamic thread creation). See also the concluding remarks for a further discussion of how the earlier results carry over. Next, we shortly present in a non-technical manner the ideas behind the abstractions before giving the formal theory.

1.1. Effect inference on the thread local level

In the first stage of the analysis, a behavioural type and effect system is used to over-approximate the lock-interactions of a single thread. To force the user to annotate the program with the expected behaviour in the form of effects is impractical, so the type and especially the behaviour should be inferred automatically. Effect inference, including inferring behavioural effects, has been studied earlier and applied to various settings, including obtaining static over-approximations of behaviour for concurrent languages in the monograph by Amtoft, Nielson, and Nielson [5]. See also the shorter accounts in [38, 39]. We apply effect inference to deadlock detection and as is standard (cf. e.g., [36, 50, 5]), the inference system is constraint-based, where the constraints in particular express an approximate order between behaviours. Besides being able to infer the behaviour, it is important that the static approximation is as precise as possible. For that it is important that the analysis may distinguish different instances of a function body depending on their calling context, i.e., the analysis should be *polymorphic* or *context-sensitive.* This can be seen as an extension of let-polymorphism to effects and using constraints. The effect reconstruction resembles the known type-inference algorithm for let-polymorphism by Damas and Milner [14, 13] and this has been used for effect-inference in various settings, e.g., in the works mentioned above.

Deadlock checking in our earlier work [43] was not polymorphic (and we did not address effect inference). The extension in this paper leads to an increase in precision with respect to checking for deadlocks, as illustrated by the small example below, where the two lock creation statements are labelled by π_1 and π_2:

Listing 1: Deadlock analysis and polymorphism

```
let x1 = new^π1 L in let x2 = new^π2 L in
let f = fn x:L . ( x.lock; x.lock )
in spawn(f(x2)); f(x1)
```

The main thread, after creating two locks and defining function f, spawns a thread, and afterwards, the main thread and the child thread run in parallel, each one executing an instance of f with different actual lock parameters. In a setting with re-entrant locks, the program is obviously deadlock-free. Part of the type system of [43] determines the potential origin of locks by data-flow analysis. When analysing the body of the function definition, the analysis cannot distinguish the two instances of f (the analysis is context-*insensitive*). This inability to distinguish the two call sites —the "context"— forces that the type of the formal parameter is, at best, $\mathtt{L}^{\{\pi_1,\pi_2\}}$, which means that the lock-argument of the function is potentially created at either point. Based on that approximate information, a

deadlock looks possible through a "deadly embrace" [16] where one thread takes first lock π_1 and then π_2, and the other thread takes them in reverse order, i.e., the analysis would report a (spurious) deadlock. The context-sensitive analysis presented here correctly analyses the example as deadlock-free.

1.2. Deadlock preserving abstractions on the global level

Lock abstraction

A standard abstraction for dynamically allocated data is to *summarize* all data allocated at a given program point into one abstract representation. We apply this idea to dynamically allocated locks. In general, this mapping from concrete data items, here locks, to their abstract representation is non-injective. For concrete, ordinary programs it is clear that identifying locks may change the behaviour of the program. Identification of locks is in general tricky (and here in particular in connection with deadlocks): on the one hand, comparing the operational behaviour of the programs, identifying locks may lead to *less* execution steps, in that lock-protected critical sections may become larger. On the other hand it may lead to *more* steps at the same time, as deadlocks may disappear when identifying (re-entrant) locks. This form of summarizing lock abstraction is problematic when analysing properties of concurrent programs, and has been observed elsewhere as well, cf. e.g., Kidd et al. in [30].

For a sound abstraction when identifying locks, one faces the following dilemma: a) the abstract level needs to exhibit at least the behaviour of the concrete level, i.e., we expect that concrete and abstract levels are related by a form of simulation. On the other hand, to preserve *deadlocks,* the following condition must hold: b) a concrete program waiting on a lock and unable to make a step thereby, must imply an analogous situation on the abstract level, lest we should miss deadlocks. Let's write $l, l_1, l_2, \ldots$ for concrete lock references and $\pi, \pi', \ldots$ for program points of lock creation, i.e., abstract locks. To satisfy a): when a concrete program takes a lock, the abstract one must be able to "take" the corresponding abstract lock, say π. A consequence of a) is that taking an abstract lock is always enabled. That is consistent with the abstraction as described where the abstract lock π confuses an arbitrary number of concrete locks including, for example those freshly created, which may be taken.

Thus, abstract locks lose their "mutual exclusion" capacity: whereas a concrete heap is a mapping which associates to each lock reference the number of times that *at most one* process is holding it, an abstract heap $\hat{\sigma}$ records how many times an abstract lock π is held by the various processes, e.g., thrice by one process and twice by another. The corresponding natural number abstractly represents the

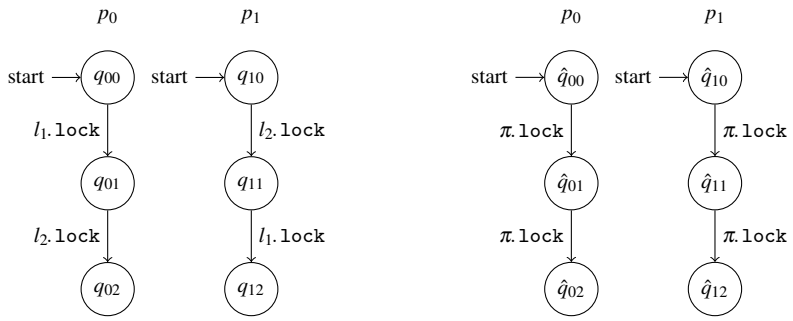
sum of the lock values of all concrete locks (per process). Without ever blocking, the abstraction leads to more possible steps, but to cater for b), the abstraction still needs to appropriately define, given an abstract heap and an abstract lock π, when a process waits on the abstract lock, as this may indicate a deadlock. The definition has to capture all possibilities of waiting on one of the corresponding concrete locks (see Definition 4.19 later). The sketched intuitions to obtain a sound abstract summary representation for locks and correspondingly for heaps lead also to a corresponding refinement of "over-approximation" in terms of simulation: not only must the a) positive behaviour be preserved as in standard simulation, but also the b) possibility of waiting on a lock and ultimately the possibility of deadlock needs to be preserved. For this we introduce the notion of *deadlock sensitive* simulation (see Definition 4.24). The definition is analogous to the one from [43]. However, it takes into account now that the analysis is polymorphic and the definition is no longer based on a direct operational interpretation of the behaviour of the effects. Instead it is based on the behavioural constraints used in the inference systems.

The points discussed are illustrated in Fig. 1, where the left diagram Fig. 1a depicts two threads running in parallel and trying to take two concrete locks, l_1 and l_2 while Fig. 1b illustrates an abstraction of the left one where the two concrete locks are summarized by the abstract lock π (typically because being created at the same program point). The concrete program obviously may run into a deadlock by reaching commonly the states q_{01} and q_{11}, where the first process is waiting on l_2 and the second process on l_1. With the abstraction sketched above, the abstract behaviour, having reached the corresponding states $\hat{q}_{01}$ and $\hat{q}_{11}$, can proceed (in two steps) to the common states $\hat{q}_{02}$ and $\hat{q}_{12}$, reaching an abstract heap where the abstract lock π is "held" two times by each process. In the state $\hat{q}_{01}$ and $\hat{q}_{11}$, however, the analysis will correctly detect that, with the given lock abstraction, the first process *may* actually wait on π, resp. on one of its concretizations, and dually for the second process, thereby detecting the deadly embrace.

Allowing this form of lock abstraction, summarizing concrete locks into an abstract one, improves our earlier analysis [43], which could therefore deal only with a static number of locks.

Counter abstraction and further behaviour abstraction

Two remaining causes of an infinite state space are the values of lock counters, which may grow unboundedly, and the fact that for each thread, the effect behaviour abstractly represents the *stack* of function calls for that thread. Sequential composition as construct for abstract behavioural effects allows to represent non-tail-recursive behaviour (corresponding to the context-free call-and-return be-



(a) Concrete (b) Abstract

Figure 1: Lock abstraction

haviour of the underlying program). To curb that source of infinity, we allow for replacing the behaviour by a tail-recursive over-approximation. The precision of the approximation can be adapted in choosing the depth of calls after which the call-structure collapses into an arbitrary, chaotic behaviour. A finite abstraction for the lock-counters is achieved similarly by imposing an upper bound on the considered lock counter, beyond which the locks behave non-deterministically. Again, for both abstractions it is crucial, that the abstraction preserves also deadlocks, which we capture again using the notion of deadlock-sensitive simulation. These two abstractions have been formulated and proven in the non-context-sensitive setting of [43].

It is indeed straightforward to carry over that part of the deadlock checking to the polymorphic analysis presented here. The polymorphic treatment here is a result from the fact that the analysis is compositional and is able to yield the most general type and effect per function. The model-exploration on the abstracted behaviour in the second stage does not work on *open* systems, but the abstract behaviour of all threads must be present. Consequently, the model exploration does not have to deal with generic, polymorphic behaviour, but with concrete instances, only, which allows to carry over the results and the abstractions developed on [43] unchanged. Thus, also the prototype implementation for the state-exploration part in the monomorphic setting of [43] carries over.

To summarize, compared to [43], the paper makes the following contributions: 1) the effect analysis is generalized to a context-sensitive formulation, using constraints, for which we provide 2) an inference algorithm. Finally, 3) we allow summarizing multiple concrete locks into abstract ones, while still preserving deadlocks.

The rest of the paper is organized as follows. After presenting syntax and semantics of the concurrent calculus in Section 2, the specification of the behavioural type system is presented in Section 3. We then convert the type system into an algorithm in Section 4, which also includes the soundness result in the form of subject reduction. The conclusion in Section 5 discusses related and future work.

2. Calculus

This section presents the syntax and semantics for our calculus with higher-order functions for lock-based concurrency. The abstract syntax is given in Table 1 (the types T will be covered in more detail in Section 3). A program P consists of processes $p\langle t\rangle$ running in parallel, where p is a process identifier and t is a thread, i.e., the code being executed. The empty program is represented by $\emptyset$. We assume, as usual, parallel composition $\|$ to be associative and commutative. The code is categorized into threads t and expressions e. A thread t is either a value v, where values includes the truth values, the unit value, leaving further values such as integers etc. unspecified, as they are irrelevant to the analysis. A thread $\texttt{let}\ x{:}T = e\ \texttt{in}\ t$ represents the sequential composition of first e followed by t, where the let-construct binds the local variable x in t. We use the sequential composition operator ; as abbreviation when the let-bound variable x does not occur free in t. Expressions include function applications and conditionals. Threads are created with the expression $\texttt{spawn}\ t$. For lock manipulation, `new L` yields the reference to a newly created lock (initially free), and the operations $v.\,\texttt{lock}$ and $v.\,\texttt{unlock}$ deal with acquiring and releasing a lock. Values which are evaluated expressions are variables, lock references, and function abstractions, where $\texttt{fun}\ f{:}T.x{:}T.t$ represents recursive function definitions.

P	$::=$	$\emptyset \mid p\langle t\rangle \mid P \| P$	program
t	$::=$	$v \mid \texttt{let}\ x{:}T = e\ \texttt{in}\ t$	thread
e	$::=$	$t \mid v\ v \mid \texttt{if}\ v\ \texttt{then}\ e\ \texttt{else}\ e \mid \texttt{spawn}\ t \mid \texttt{new L}$	
	$\mid$	$v.\,\texttt{lock} \mid v.\,\texttt{unlock}$	expr.
v	$::=$	$x \mid l \mid () \mid \texttt{true} \mid \texttt{false} \mid \texttt{fn}\ x{:}T.t \mid \texttt{fun}\ f{:}T.x{:}T.t$	values

Table 1: Abstract syntax

Semantics

The small-step operational semantics, presented next, distinguishes between local and global steps (cf. Tables 2 and 3). The local steps are straightforward. Rule R-RED is the basic evaluation step, replacing in the continuation thread t the local variable by the value v (where $[v/x]$ is understood as capture-avoiding substitution). Rule R-LET restructures a nested let-construct. As the let-construct generalizes sequential composition, the rule expresses associativity of that construct. Thus it corresponds to transforming $(e_1;t_1);t_2$ into $e_1;(t_1;t_2)$. Note that the grammar insists that, e.g., in an application, both the function and the arguments are values, analogously when acquiring a lock, etc. This form of representation is known as *a-normal form* [18]. Together with the rest of the rules, which perform a case distinction on the first non-let expression (e.g., `spawn`, `new L`, etc.) in a let construct, that ensures 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. Global configurations are of the form $\sigma \vdash P$ where P is a program and

---

$\texttt{let}\ x{:}T = v\ \texttt{in}\ t \to t[v/x]$ R-RED

$\texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2 \to \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2)$ R-LET

$\texttt{let}\ x{:}T = \texttt{if}\ \texttt{true}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t \to \texttt{let}\ x{:}T = e_1\ \texttt{in}\ t$ R-IF$_1$

$\texttt{let}\ x{:}T = \texttt{if}\ \texttt{false}\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t \to \texttt{let}\ x{:}T = e_2\ \texttt{in}\ t$ R-IF$_2$

$\texttt{let}\ x{:}T = (\texttt{fn}\ x'{:}T'.t')\ v\ \texttt{in}\ t \to \texttt{let}\ x{:}T = t'[v/x']\ \texttt{in}\ t$ R-APP$_1$

$\texttt{let}\ x{:}T = (\texttt{fun}\ f{:}T_1.x'{:}T_2.t')\ v\ \texttt{in}\ t \to \texttt{let}\ x{:}T = t'[v/x'][\texttt{fun}\ f{:}T_1.x'{:}T_2.t'/f]\ \texttt{in}\ t$ R-APP$_2$

Table 2: Local steps

the heap σ is a finite mapping from lock identifiers, denoted as $\sigma(l)$, to the status of each lock, which can be either *free* or a tuple indicating the number of times a lock has been taken by a thread, written as $p(n)$. When relating the concrete heap with its abstraction later, we allow ourselves also to write $\sigma(l,p) = n+1$ if $\sigma(l) = p(n+1)$ (indicating the pair of process identifier p and lock count n) and $\sigma(l,p) = 0$, otherwise. Note that for certain situations, $\sigma +_p l$ and $\sigma -_p l$ are undefined. The premises of R-LOCK and R-UNLOCK, however, prevent that the definitions are used in these undefined situations. Note that as a consequence, there is no rule covering an attempt by a process to unlock a lock the process does

not currently own, meaning that the process "block" in that case. In a real programming language, a more realistic behaviour in the unlocking case would be to throw an exception instead. Our results concerning detecting deadlocks are not affected by this choice of behaviour, as the definition of deadlock later will not count such a process as deadlocked.

The global steps are given as transitions between global configurations. It will be handy later to assume the transitions appropriately labelled (cf. Table 3). Thread-local transition steps are lifted to the global level by rule R-LIFT. A global step is a thread-local step made by one of the individual threads sharing the same σ (cf. rule R-PAR). R-SPAWN creates a new thread with a fresh identity running in parallel with the parent thread. All the identities are unique at the global level. Creating a new lock, which is initially free, allocates a fresh lock reference l in the heap (cf. rule R-NEWL). The locking step (cf. rule R-LOCK) takes a lock when it is either free or already being held by the requesting process. To update the heap, we define: If $\sigma(l) = \mathit{free}$, then $\sigma +_p l = \sigma[l \mapsto p(1)]$ and if $\sigma(l) = p(n)$, then $\sigma +_p l = \sigma[l \mapsto p(n+1)]$. Dually $\sigma -_p l$ is defined as follows: if $\sigma(l) = p(n+1)$, then $\sigma -_p l = \sigma[l \mapsto p(n)]$, and if $\sigma(l) = p(1)$, then $\sigma -_p l = \sigma[l \mapsto \mathit{free}]$. Unlocking works correspondingly, i.e., it sets the lock as being free resp. decreases the lock count by one (cf. rule R-UNLOCK).

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

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

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

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

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

Table 3: Global steps

To later relate the operational behaviour to its behavioural abstraction, we *label* the transition of the operational semantics appropriately. In particular, steps for lock manipulations are labelled to indicate which *process* has taken or released which *lock*. For instance, the labelled transition step $\xrightarrow{p\langle l\mathtt{lock}\rangle}$ means that a process p takes a lock labelled l. We discuss further details about the labels in Section 3.1 and the labelled transition steps in Section 4.2.

Before defining the notion of deadlock, we first characterize the situation in which one thread in a program attempts to acquire a lock which is not available as follows:

Definition 2.1 (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 $\sigma \vdash P \xrightarrow{p\langle l\mathtt{lock}\rangle}$, and if furthermore there exists a σ' s.t. $\sigma' \vdash P \xrightarrow{p\langle l\mathtt{lock}\rangle} \sigma'' \vdash P'$.*

The notion of (resource) deadlock used is rather standard, where a number of processes waiting for each other's locks in a cyclic manner constitute a deadlock (see also [43]). In our setting with re-entrant locks, a process cannot deadlock "on itself".

Definition 2.2 (Deadlock)**.** *A configuration $\sigma \vdash P$* is deadlocked *if $\sigma(l_i) = p_i(n_i)$ and furthermore waits$(\sigma \vdash P, p_i, l_{i+_k 1})$, for all $0 \leq i \leq k-1$ (and where $k \geq 2$ for some k). The $+_k$ represents addition modulo k. A configuration $\sigma \vdash P$* contains a deadlock*, if, starting from $\sigma \vdash P$, a deadlocked configuration is reachable; otherwise, it is deadlock free.*

3. Type system

Next we present an effect type system to derive behavioural information which can be used, in a second step, to detect potential deadlocks. The type system derives flow information about which locks may be used at various points in the program. Additionally, it derives an abstract, i.e., approximate representation of the code's behaviour. The representation extends our earlier system [43] by making the analysis *context-sensitive* and furthermore by supporting type and effect *inference*, both important from a practical point of view. Being context-sensitive, making the effect system polymorphic, increases the precision of the analysis. Furthermore, inference removes the burden from the programmer to annotate the program appropriately to allow checking for potential deadlock. These extensions follow standard techniques for behaviour inference, see for instance Amtoft, Nielson, and Nielson [5] and type-based flow analysis, see e.g., Mossin [36]. Unlike

the presentation in [43], and following the mentioned standard techniques, the system here makes use of explicit *constraints*. Type systems are, most commonly, formulated in a syntax-directed manner, i.e., analyzing the program code in a divide-and-conquer manner. That obviously results in an efficient analysis of the code. However, a syntax-directed formulation of the deduction rules of the type system, which forces to analyze the code following the syntactic structure of the program, may have disadvantages as well. Using constraints in a type system *decouples* the syntax-directed phase of the analysis, which collects the constraints, from the task of actually *solving* the constraints. Formulations of type systems without relying on constraints can be seen as solving the underlying constraints "on-the-fly", while recurring through the structure of the code. For illustration: in connection with (conventional) unification-based type inference, instead of integrating unification into the rule system, as is often done for instance in presentations of the most well-known type-inference algorithm of Hindley-Milner-Damas [14, 13, 27, 35], one may collect the need to unify types as a set of unification constraints left to be solved later.

3.1. Types, effects, and constraints

The analysis performs a data flow analysis to track the usage of locks. For that purpose, the lock creation statements are equipped with labels, writing $\texttt{new}^{\pi}\ \texttt{L}$, where π is taken from a countably infinite set of labels. As usual, the labels π are assumed unique in a given program. The grammar for annotations, types, and effects is given in Tables 4 and 5. As said, the annotation π is used to label program points where locks are created, r denotes sets of πs with ρ representing corresponding variables. Types include basic types, represented by B, such as the unit type `Unit`, booleans, integers, etc., functional types with latent *effect* φ, and lock types $\texttt{L}^{r}$ where the annotation r captures the flow information about the potential places at which the lock is created. This information will be reconstructed, and the user writes types without annotations (the "underlying" types) in the program. We write T (and its syntactic variants) as meta-variables for the underlying types, and $\hat{T}$ (and its syntactic variants) for the annotated types, as given in the grammar. The universally quantified types, represented by $\hat{S}$, capture functions which are polymorphic in locations and effects.

Whereas the type of an expression captures the results of the computations of the expression if it terminates, the effect captures the *behaviour* during the computations. For the deadlock analysis, we capture the lock interactions as effects, i.e., which locks are accessed during execution and in which order. The effects (cf. Table 5) are split between a (thread-) local level φ and a global level Φ. The empty

$Y ::= \rho \mid X$		type-level variables
$r ::= \rho \mid \{\pi\} \mid r \sqcup r$		lock/label sets
$\hat{T} ::= B \mid \mathtt{L}^r \mid \hat{T} \xrightarrow{\varphi} \hat{T}$		types
$\hat{S} ::= \forall \vec{Y}{:}C.\, \hat{T} \xrightarrow{\varphi} \hat{T} \mid \hat{T}$		type schemes
$C ::= \emptyset \mid \rho \sqsupseteq r, C \mid X \sqsupseteq \varphi, C$		simple constraints

Table 4: Types and type schemes

effect is denoted by ε, representing behaviour without lock operations. Sequential composition is represented by $\varphi_1;\varphi_2$. The choice between two effects $\varphi_1 + \varphi_2$, as well as recursive effects $rec X.\varphi$, is actually not generated by the algorithm; they would show up when solving the constraints generated by the algorithm. We included their syntax for completeness. Note also that recursion is not polymorphic. Labels a capture the three basic effects: spawning a new process with behaviour φ is represented by `spawn` φ, while $r.$`lock` and $r.$`unlock` respectively capture lock manipulations, acquiring and releasing a lock, where r refers to the possible points of creation. Silent transitions are represented by τ. Lock-creation has no corresponding effect, as newly created locks are initially free, i.e., with a lock-count of 0. On the abstract level, locks are summarized by the *sum* of all locks created at a given point. Hence, lock creation will be represented by a τ-transition.

Simple constraints C finally are finite sets of in-equations of the form $\rho \sqsupseteq r$ or

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

Table 5: Effects

of $X \sqsupseteq \varphi$, where ρ is, as mentioned, a flow variable and X an effect or behaviour variable. The constraints are called simple as their form is restricted in that on the "larger" side of the inequalities, only a variable is allowed. Later, we also will make use of a more general form of constraints, which consists of "sets" of constraints of the forms $r_1 \sqsupseteq r_2$ and $\varphi_1 \sqsupseteq \varphi_2$. We also use the meta-variable C for

those more general constraints. Whether or not the constraints at hand are simple should be clear from the context, and type schemes always use *simple* constraints, only, as given in the grammar. To allow polymorphism we use type schemes $\hat{S}$, i.e., prefix-quantified types of the form $\forall\vec{Y}{:}C.\ \hat{T}$, where Y are variables ρ or X. The qualifying constraints C in the type scheme impose restrictions on the bound variables. Note that $\hat{T}$ can only be a *function type* $\hat{T} \xrightarrow{\varphi} \hat{T}$ in a type scheme $\hat{S}$. The formal system presented in this paper uses a constraint-based flow analysis as proposed by Mossin [36] for lock information. Likewise, the effects captured as a sequence of behaviour are formulated using constraints.

3.2. Type and effect system

This section presents the specification of the type and effect system used to derive information about the lock usage. The typing part contains the flow information about locks, which keeps track of the use of locks with respect to their point of creation. In addition to the flow information, the *effect* part captures an abstraction of the behaviour of a program. The static analysis is a standard constraint-based analysis (see e.g., [50, 5]). To enhance the precision of the analysis, the type and effect analysis is *context-sensitive* with the support of universally *polymorphic* types. As is standard when capturing a form of subtyping in the presence of universal polymorphism, type schemes do not quantify unrestrictedly over type-level variables, but are of an extended form where the universally quantified variables range over all instances satisfying a given constraint (as part of the type scheme).

Besides constraints over the quantified variables, also the judgments of the type and effect system are formulated using constraints and are of the following form

$$C;\Gamma \vdash e : \hat{S} :: \varphi \tag{1}$$

where C are simple constraints and with the intended meaning that given the constraint C and the context Γ, expression e is of type $\hat{S}$ and has effect φ. As the types and the effects contain variables ρ and X, the judgment is interpreted relative to solutions of the constraint set C. The rules of the type and effect system to derive such judgments are presented in Table 6.

A variable has no effect and its type (scheme) corresponds to its declaration in the context Γ (cf. T-VAR). Similarly, lock references have no effect. As a general observation, values have no effect as they cannot be evaluated further. Also, lock

$$\frac{\Gamma(x) = \hat{S}}{C;\Gamma \vdash x : \hat{S} :: \varepsilon}\ \text{T-VAR} \qquad \frac{}{C;\Gamma \vdash l^{\rho} : \mathrm{L}^{\rho} :: \varepsilon}\ \text{T-LREF} \qquad \frac{C \vdash \rho \sqsupseteq \{\pi\}}{C;\Gamma \vdash \texttt{new}^{\pi}\ \mathrm{L}\colon\ \mathrm{L}^{\rho} :: \varepsilon}\ \text{T-NEWL}$$

$$\frac{\lfloor \hat{T}_1 \rfloor = T_1 \qquad C;\Gamma, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: \varphi}{C;\Gamma \vdash \texttt{fn}\ x{:}T_1.e : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \varepsilon}\ \text{T-ABS}_1$$

$$\frac{\lfloor \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 \rfloor = T_1 \to T_2 \qquad C;\Gamma, f{:}\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: \varphi}{C;\Gamma \vdash \texttt{fun}\ f{:}T_1 \to T_2.x{:}T_1.e : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \varepsilon}\ \text{T-ABS}_2$$

$$\frac{C;\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon \qquad C;\Gamma \vdash v_2 : \hat{T}_2 :: \varepsilon}{C;\Gamma \vdash v_1\ v_2 : \hat{T}_1 :: \varphi}\ \text{T-APP}$$

$$\frac{C;\Gamma \vdash v : \texttt{Bool} :: \varepsilon \quad C;\Gamma \vdash e_1 : \hat{T} :: \varphi \quad C;\Gamma \vdash e_2 : \hat{T} :: \varphi}{C;\Gamma \vdash \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \varphi}\ \text{T-COND}$$

$$\frac{C;\Gamma \vdash e_1 : \hat{S}_1 :: \varphi_1 \qquad \lfloor \hat{S}_1 \rfloor = T_1 \qquad C;\Gamma, x{:}\hat{S}_1 \vdash e_2 : \hat{T}_2 :: \varphi_2}{C;\Gamma \vdash \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \varphi_1;\varphi_2}\ \text{T-LET}$$

$$\frac{C;\Gamma \vdash t : \hat{T} :: \varphi \qquad C \vdash X \sqsupseteq \texttt{spawn}\ \varphi}{C;\Gamma \vdash \texttt{spawn}\ t : \texttt{Unit} :: X}\ \text{T-SPAWN}$$

$$\frac{C;\Gamma \vdash v : \mathrm{L}^{\rho} :: \varepsilon \qquad C \vdash X \sqsupseteq \rho.\texttt{lock}}{C;\Gamma \vdash v.\ \texttt{lock}\colon \mathrm{L}^{\rho} :: X}\ \text{T-LOCK} \qquad \frac{C;\Gamma \vdash v : \mathrm{L}^{\rho} :: \varepsilon \qquad C \vdash X \sqsupseteq \rho.\texttt{unlock}}{C;\Gamma \vdash v.\ \texttt{unlock}\colon \mathrm{L}^{\rho} :: X}\ \text{T-UNLOCK}$$

$$\frac{C_1, C_2;\Gamma \vdash e : \hat{T} :: \varphi \qquad \vec{Y} \text{ not free in } \Gamma, C_1, \varphi \qquad \forall \vec{Y}{:}C_2.\hat{T} \text{ solvable from } C_1 \qquad \forall \vec{Y}{:}C_2.\hat{T} \vdash wf}{C_1;\Gamma \vdash e : \forall \vec{Y}{:}C_2.\hat{T} :: \varphi}\ \text{T-GEN}$$

$$\frac{C_1;\Gamma \vdash e : \forall \vec{Y}{:}C_2.\hat{T} :: \varphi \qquad \forall \vec{Y}{:}C_2.\hat{T} \text{ solvable from } C_1 \text{ by } \theta}{C_1;\Gamma \vdash e : \theta\hat{T} :: \varphi}\ \text{T-INST}$$

$$\frac{C;\Gamma \vdash e : \hat{T} :: \varphi \qquad C \vdash \hat{T} \le \hat{T}', \varphi \sqsubseteq \varphi'}{C;\Gamma \vdash e : \hat{T}' :: \varphi'}\ \text{T-SUB}$$

Table 6: Type and effect system (specification)

creation in T-NEWL has no effect.[1] As for the flow, the constraint $C \vdash \rho \sqsupseteq \{\pi\}$ in the premise ensures that the annotation ρ of the lock type contains π, which labels the point of creation of the lock. In anticipation of the subject reduction proof later, the conclusion of the rule can be understood to document our pick of the flow variable ρ. Likewise, we will soon record the effect that has been derived for a spawned thread in annotation to `spawn`.

For function abstraction (cf. T-ABS$_1$), the type of the formal parameters is declared as underlying type T, i.e., without any annotation. The declaration is then annotated as $\hat{T}$ and remembered in the context as the binding $x{:}\hat{T}$ where $T = \lfloor \hat{T} \rfloor$. The operator $\lfloor\ \rfloor$ removes all the annotations (including quantifications of type schemes) and returns the corresponding un-annotated type. The premise checks the function body e with the extended context. Rule T-ABS$_2$ for recursive functions works analogously. For function application in rule T-APP, both function and argument have no effect as they are values. The effect of the function application is the latent effect of the function body. The treatment of conditionals (cf. T-COND) is standard: both types and effects of the two branches have to agree with each other. The effect of the let-construct in rule T-LET is the sequential composition of the effect of e_1 and e_2. As for the type, the derived (annotated) type scheme $\hat{S}_1$ of e_1 must be compatible with the declared underlying type T_1. The type is remembered and is used to extend the context to check expression e_2. The overall type of the let-construct is the type of e_2. For spawning a thread, the premise of T-SPAWN checks the well-typedness and the effect of the thread being spawned. The constraint $C \vdash X \sqsupseteq \texttt{spawn}\ \varphi$ ensures that the effect variable X is an upper bound of the effect of spawning a thread with effect φ. The two rules (cf. T-LOCK and T-UNLOCK) dealing with locking and unlocking work similarly by using a constraint in the premise to guarantee the effect $\rho.\texttt{lock}$ resp. $\rho.\texttt{unlock}$ is contained in the effect variable X.

There are three non-syntax directed rules in the system, generalization, instantiation, and subsumption. For the latter, we need the following definition:

Definition 3.1 (Subtyping and subeffecting)**.** *Let C be a set of* simple *constraints. Then the subtyping relation $C \vdash \hat{T}_1 \leq \hat{T}_2$ is defined in Table 7. Furthermore, the relations $C \vdash \varphi_1 \sqsubseteq \varphi_2$ and $C \vdash r_1 \sqsubseteq r_2$ are given in Table 8. Note that the single constraints on the right-hand sides of the $\vdash$ are* not *necessarily simple constraints.*

We generalize $C_1 \vdash \varphi_1 \sqsubseteq \varphi_2$ and $C_1 \vdash r_1 \sqsubseteq r_2$ to $C_1 \vdash C_2$ as follows, where C_1

[1] That is different from the account in [43]. That paper uses a different abstraction for the locks, disallowing that different concrete lock references could be abstracted into one abstract one.

is still a simple constraint set: $C_1 \vdash C_2$ *if* $C_1 \vdash r_1 \sqsubseteq r_2$ *for all* $r_1 \sqsubseteq r_2$ *from* C_2 *and* $C_1 \vdash \varphi_1 \sqsubseteq \varphi_2$ *for all* $\varphi_1 \sqsubseteq \varphi_2$ *from* C_2.

In the following we adopt the convention that for $C_1 \vdash C_2$, it's required that $fv(C_2) \subseteq fv(C_2)$, analogously for the other judgments. This can always been assumed by adding C_1 variables and trivial constraints (for instance, adding $X \sqsubseteq X$).

$$C \vdash \hat{T} \leq \hat{T} \quad \text{S-REFL} \qquad \frac{C \vdash \hat{T}_1 \leq \hat{T}_2 \qquad C \vdash \hat{T}_2 \leq \hat{T}_3}{C \vdash \hat{T}_1 \leq \hat{T}_3}\ \text{S-TRANS} \qquad \frac{C \vdash r_1 \sqsubseteq r_2}{C \vdash \mathrm{L}^{r_1} \leq \mathrm{L}^{r_2}}\ \text{S-LOCK}$$

$$\frac{C \vdash \hat{T}_1' \leq \hat{T}_1 \qquad C \vdash \hat{T}_2 \leq \hat{T}_2' \qquad C \vdash \varphi \sqsubseteq \varphi'}{C \vdash \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 \leq \hat{T}_1' \xrightarrow{\varphi'} \hat{T}_2'}\ \text{S-ARROW}$$

Table 7: Subtyping

Substitutions are defined, as usual, as finite mappings from variables to terms, where the variables here are type-level variables ρ and X and the terms are label sets r resp. effects φ (cf. Table 4 and 5). We say a substitution θ solves or satisfies a constraint set C, written $\theta \models C$, if $\emptyset \vdash \theta C$.

For the type system, we will put some general well-formedness restrictions on the form of allowed type schemes, basically restricting which sets of variables can be quantified over. Remember that constraints $c \in C$ need to be of the form $\rho \sqsupseteq r$ or $X \sqsupseteq \varphi$, i.e., for upper bounds, *only* variables are allowed. Correspondingly, in a type scheme $\forall \vec{Y}{:}C.\hat{T}$ and for a constraint for instance of the form $X \sqsupseteq \varphi \in C$, if a variable Y_1 occurring free in φ is bound by $\forall \vec{Y}$, then also its direct upper bound X needs to be bound (analogously for constraints concerning ρ-variables). In other words, the set of variables used in the quantification needs to be *upward* closed, as defined in Definition 3.2(1). The dual definition for downward closure is given in 3.2(2) while the upward-downward closure is given in 3.2(3).

Definition 3.2 (Closure)**.** 1. *A set of variables* $\vec{Y}$ *is* upward closed *wrt. C, if the following implication holds: if* $Y \in \vec{Y}$ *and* $Y \in fv(\varphi)$ *or* $Y \in fv(r)$ *for a constraint* $\varphi \sqsubseteq Y' \in C$ *resp.* $r \sqsubseteq Y' \in C$*, then also* $Y' \in \vec{Y}$*. The* upward closure *of a set of variables* $\vec{Y}$ *wrt. a constraint set C (written* $close_{\uparrow}(\vec{Y}, C)$*) is the smallest set* $\vec{Y}'$ *s.t.* $\vec{Y}' \supseteq \vec{Y}$ *and* $\vec{Y}'$ *is upward closed wrt. C.*

2. *A set of variables* $\vec{Y}$ *is* downward closed *wrt. C if the following implication holds: if* $Y \in \vec{Y}$*, and a constraint* $\varphi \sqsubseteq Y \in C$ *or* $r \sqsubseteq Y \in C$*, then also* $fv(\varphi) \subseteq$

$C, r \sqsubseteq \rho \vdash r \sqsubseteq \rho$ S-Ax$_L$ $\quad$ $C \vdash r \sqsubseteq r$ S-Refl$_L$ $\quad$ $\dfrac{C \vdash r_1 \sqsubseteq r_2 \quad C \vdash r_2 \sqsubseteq r_3}{C \vdash r_1 \sqsubseteq r_3}$ S-Trans$_L$

$C \vdash r_1 \sqsubseteq r_1 \sqcup r_2$ S-LUB^1_L $\quad$ $\dfrac{C \vdash r_1 \sqsubseteq r \quad C \vdash r_2 \sqsubseteq r}{C \vdash r_1 \sqcup r_2 \sqsubseteq r}$ S-LUB^2_L

$C \vdash r_1 \sqcup r_2 \sqsubseteq r_2 \sqcup r_1$ S-Comm$_L$ $\quad$ $C \vdash r_1 \sqcup (r_2 \sqcup r_3) \sqsubseteq (r_1 \sqcup r_2) \sqcup r_3$ S-Assoc$_L$

$\varepsilon; \varphi \equiv \varphi$ EE-Unit $\quad$ $\varphi_1; (\varphi_2; \varphi_3) \equiv (\varphi_1; \varphi_2); \varphi_3$ EE-Assoc

$C, \varphi \sqsubseteq X \vdash \varphi \sqsubseteq X$ S-Ax$_E$ $\quad$ $\dfrac{\varphi_1 \equiv \varphi_2}{C \vdash \varphi_1 \sqsubseteq \varphi_2}$ S-Refl$_E$

$\dfrac{C \vdash \varphi_1 \sqsubseteq \varphi_2 \quad C \vdash \varphi_2 \sqsubseteq \varphi_3}{C \vdash \varphi_1 \sqsubseteq \varphi_3}$ S-Trans$_E$

$\dfrac{C \vdash \varphi_1 \sqsubseteq \varphi_1' \quad C \vdash \varphi_2 \sqsubseteq \varphi_2'}{C \vdash \varphi_1; \varphi_2 \sqsubseteq \varphi_1'; \varphi_2'}$ S-Seq $\quad$ $\dfrac{C \vdash \varphi_1 \sqsubseteq \varphi_2}{C \vdash \texttt{spawn}\ \varphi_1 \sqsubseteq \texttt{spawn}\ \varphi_2}$ S-Spawn

$\dfrac{C \vdash r_1 \sqsubseteq r_2}{C \vdash r_1.\texttt{lock} \sqsubseteq r_2.\texttt{lock}}$ S-Lock $\quad$ $\dfrac{C \vdash r_1 \sqsubseteq r_2}{C \vdash r_1.\texttt{unlock} \sqsubseteq r_2.\texttt{unlock}}$ S-Unlock

Table 8: Orders on behaviours

$\vec{Y}$ *resp.* $fv(r) \subseteq \vec{Y}$. *The* downward closure *of a set of variables* $\vec{Y}$ *wrt. a constraint set* C *(written* $close_{\downarrow}(\vec{Y}, C)$*) is the smallest set* $\vec{Y}'$ *s.t.* $\vec{Y}' \supseteq \vec{Y}$ *and* $\vec{Y}'$ *is downward closed wrt.* C.

3. *A set of variables* $\vec{Y}$ *is* upward-downward closed *wrt.* C *if the following implication holds: if* $Y \in \vec{Y}$*, and* $c \in C$ *with* $Y \in fv(c)$ *and* $Y' \in fv(c)$*, then also* $Y' \in \vec{Y}$. *The* upward-downward closure *of a set of variables* $\vec{Y}$ *wrt. a constraint set* C *(written* $close_{\uparrow\downarrow}(\vec{Y}, C)$*) is the smallest set* $\vec{Y}'$ *s.t.* $\vec{Y}' \supseteq \vec{Y}$ *and* $\vec{Y}'$ *is upward-downward closed wrt.* C.

Besides the mentioned closure condition on the set of quantified variables, each constraint used in the type scheme should contain at least one quantified variable (otherwise there would be no need to put the corresponding condition

into the qualifying constraints, the condition may equally well be captured by the global constraints). Finally, at least one of the quantified variables should actually occur in the type and all quantified variables should actually occur in the qualifying constraints.

Definition 3.3 (Well-formedness)**.** *A type scheme $\forall\vec{Y}{:}C.\hat{T}$ is well-formed if the following holds*

1. *$\vec{Y}$ is upward closed wrt. C.*
2. *if $c \in C$, then $fv(c) \cap \vec{Y} \neq \emptyset$.*
3. *$\emptyset \neq (\vec{Y} \cap fv(\hat{T}))$ and $\vec{Y} \subseteq fv(C)$.*

The generic instance relation between two type schemes is defined as follows:

Definition 3.4 (Generic instance)**.** *A type scheme $\forall\vec{Y_1}{:}C_1.\hat{T}_1$ is a generic instance of $\forall\vec{Y_2}{:}C_2.\hat{T}_2$ wrt. a constraint C, written as $C \vdash \forall\vec{Y_1}{:}C_1.\hat{T}_1 \lesssim^g \forall\vec{Y_2}{:}C_2.\hat{T}_2$, iff there exists a substitution θ where $dom(\theta) \subseteq \vec{Y_2}$ such that*

1. *$C, C_1 \vdash \theta C_2$,*
2. *$C, C_1 \vdash \theta\hat{T}_2 \leq \hat{T}_1$, and*
3. *No Y in $\vec{Y_1}$ is free in $\forall\vec{Y_2}{:}C_2.\hat{T}_2$.*

The two dual rules T-GEN and T-INST, which are not syntax-directed, introduce resp. eliminate type schemes. While both rules require the type scheme to be solvable from the constraint set C_1 (see Definition 3.5) in the conclusion, the generalization rule also ensures that the resulting type is well-formed.

Definition 3.5 (Solvable)**.** *A type scheme $\forall\vec{Y}{:}C_1.\hat{T}$ is solvable from C_2 by substitution θ, if $dom(\theta) \subseteq \vec{Y}$ and $C_2 \vdash \theta C_1$. The type scheme is called solvable from C_2 if there exists a substitution that solves it.*

The typing rules from Table 6 on page 14 are thread-local. To later detect deadlocks, which are global phenomena as they involve more than one thread, we need also well-typedness of a global system. The corresponding rules are given in Table 9 on the next page, straightforwardly lifting the results of the local analysis to the global level. In particular, a global program is well-typed as soon as all its threads are. In abuse of notation, we use Φ to abbreviate $p_1\langle\varphi_1; C_1\rangle \parallel \ldots \parallel p_n\langle\varphi_n; C_n\rangle$.

$$\frac{C;\emptyset \vdash t : T :: \varphi}{\vdash p\langle t\rangle : ok :: p\langle \varphi; C\rangle}\ \text{T-THREAD} \qquad \frac{\vdash P_1 : ok :: \Phi_1 \qquad \vdash P_2 : ok :: \Phi_2}{\vdash P_1 \parallel P_2 : ok :: \Phi_1 \parallel \Phi_2}\ \text{T-PAR}$$

Table 9: Type and effect system (global)

4. Type inference

Next we develop an algorithm for the type system in Section 3.2, which derives types and effects and generates corresponding constraints (see Table 11 below). It is formulated in a rule-based manner, with judgments of the form:

$$\Gamma \vdash e : \hat{T} :: \varphi; C\ . \tag{2}$$

The system is syntax-directed, i.e., algorithmic, where Γ and e are considered as "input", and the annotated type $\hat{T}$, the effect φ, and the set of constraints C as "output". Concentrating on the flow information and the effect part, expressions e, in particular the let-expression and function definitions, are type-annotated with the *underlying* types, as given in Table 4. In contrast, e contains no flow or effect annotations; those are derived by the algorithmic type system. It would be straightforward to have the underlying types reconstructed as well, using standard type inference à la Hindley/Milner/Damas [14, 13, 27]. For simplicity, we focus on the type annotations and the effect part. The intended meaning of the typing judgment in Equation (2) is that, relative to a typing context Γ and for expression e: if evaluating e terminates, the corresponding values are elements of the domain represented by $\hat{T}$ (more precisely by the underlying type T). For locks, the flow annotation over-approximates the point of lock creation, and finally, φ over-approximates the lock-interactions while evaluating e. As usual, the behavioural over-approximation is a form of simulation. For our purpose, we will define a particular, deadlock-sensitive form of simulation. These intended over-approximations are understood relative to the generated constraints C, i.e., *all* solutions of C give rise to a sound over-approximation in the mentioned sense.

Ultimately, one is interested in the minimal solution of the constraints, as it provides the most precise information. Solving the constraints is done after the algorithmic type system, but to allow for the most precise solution afterward, each rule should generate the most general constraint set, i.e., the one which allows the maximal set of solutions. This is achieved using *fresh* variables for each additional constraint. In the system below, new constraints are generated from requesting

that types are in a "subtype" relationship. Without subtyping on the underlying types, e.g., stipulating relationships between basic types such as $\texttt{Int} \leq \texttt{Real}$, "subtyping" here concerns the flow annotations on the lock types and the latent effects on function types. Instead of requesting that, for instance in rule TA-APP in Table 11, the argument of a function of type $\hat{T}_2 \xrightarrow{\varphi} \hat{T}_1$ is of a subtype $\hat{T}_2'$ of $\hat{T}_2$, i.e., instead of requiring $\hat{T}_2' \leq \hat{T}_2$ in that situation, the corresponding rule will generate new constraints in requiring the subtype relationship to hold (see Definition 4.1). As an invariant, the type system makes sure that lock types are always of the form $\texttt{L}^\rho$, i.e., using flow *variables* and similarly that only variables X are used for the latent effects for function types.

Definition 4.1 (Constraint generation)**.** *The judgment $\hat{T}_1 \leq \hat{T}_2 \vdash_a C$ (read as "requiring $\hat{T}_1 \leq \hat{T}_2$ generates the constraints C") is inductively given as follows:*

$$B \leq B \vdash_a \emptyset \quad \text{C-BASIC} \qquad \texttt{L}^{\rho_1} \leq \texttt{L}^{\rho_2} \vdash_a \{\rho_1 \sqsubseteq \rho_2\} \quad \text{C-LOCK}$$

$$\frac{\hat{T}_1' \leq \hat{T}_1 \vdash_a C_1 \quad \hat{T}_2 \leq \hat{T}_2' \vdash_a C_2 \quad C_3 = \{X \sqsubseteq X'\}}{\hat{T}_1 \xrightarrow{X} \hat{T}_2 \leq \hat{T}_1' \xrightarrow{X'} \hat{T}_2' \vdash_a C_1, C_2, C_3} \text{C-ARROW}$$

In the presence of subtyping/sub-effecting, the overall type of a conditional needs to be an upper bound on the types/effects of the two branches (resp. the least upper bound in case of a minimal solution). To generate the most general constraints, fresh variables are used for the result type. This is captured in the following definition. Note that given $\hat{T}$ by $\hat{T}_1 \vee \hat{T}_2 \vdash_a \hat{T}; C$, type $\hat{T}$ in itself does not represent the least upper bound of $\hat{T}_1$ and $\hat{T}_2$. The use of fresh variables assures, however, that the minimal solution of the generated constraints makes $\hat{T}$ into the least upper bound.

Definition 4.2 (Least upper bound)**.** *The partial operation $\vee$ on annotated types (and in abuse of notation, on effects), giving back a set of constraints plus a type (resp. an effect) is inductively given by the rules of Table 10. The operation $\wedge$ is defined dually.*

The rules for the type and effect system then are given in Table 11. A variable has no effect and its type (scheme) is looked up from the context Γ. The constraints C that may occur in the type scheme, are given back as constraints of the variable x, replacing the $\forall$-bound variables $\vec{Y}$ in C by fresh ones. Lock creation at point π (cf. TA-NEWL) is of the type $\texttt{L}^\rho$, has an empty effect and the

$$\frac{B_1 = B_2}{B_1 \vee B_2 = B_1;\emptyset}\ \text{LT-BASIC} \qquad \frac{\rho\ \mathit{fresh} \quad \mathrm{L}^{\rho_1} \leq \mathrm{L}^{\rho} \vdash_a C_1 \quad \mathrm{L}^{\rho_2} \leq \mathrm{L}^{\rho} \vdash_a C_2}{\mathrm{L}^{\rho_1} \vee \mathrm{L}^{\rho_2} = \mathrm{L}^{\rho};C_1,C_2}\ \text{LT-LOCK}$$

$$\frac{\hat{T}_1' \wedge \hat{T}_1'' = \hat{T}_1;C_1 \qquad \hat{T}_2' \vee \hat{T}_2'' = \hat{T}_2;C_2 \qquad X_1 \sqcup X_2 = X;C_3}{\hat{T}_1' \xrightarrow{X_1} \hat{T}_2' \vee \hat{T}_1'' \xrightarrow{X_2} \hat{T}_2'' = \hat{T}_1 \xrightarrow{X} \hat{T}_2;C_1,C_2,C_3}\ \text{LT-ARROW}$$

$$\frac{X\ \mathit{fresh} \qquad C = \{\varphi_1 \sqsubseteq X, \varphi_2 \sqsubseteq X\}}{\varphi_1 \sqcup \varphi_2 = X;C}\ \text{LE-EFF}$$

Table 10: Constraints for least upper bound

generated constraint requires $\rho \sqsupseteq \{\pi\}$, using a fresh ρ. As values, abstractions have no effect (cf. TA-ABS rules) and again, fresh variables are appropriately used. In rule TA-ABS$_1$, the latent effect of the result type is represented by X under the generated constraint $X \sqsupseteq \varphi$, where φ is the effect of the function body checked in the premise. The context in the premise is extended by $x{:}\lceil T \rceil_a$, where the operation $\lceil T \rceil_a$ annotates all occurrences of lock types `L` with fresh variables and introduces fresh effect variables for the latent effects. Rule TA-ABS$_2$ for recursive functions works analogously, with an additional constraint generated by requiring $\hat{T}_2 \geq \hat{T}_2'$, where $\hat{T}_2'$ is the type of function body e checked in the premise. For applications (cf. TA-APP), both the function and the arguments are values and therefore have no effect. As usual, the type of the argument needs to be a subtype of the input type of the function, and corresponding constraints C_3 are generated by $\hat{T}_2' \leq \hat{T}_2 \vdash_a C_3$. For the overall effect, again a fresh effect variable is used which is connected with the latent effect of the function by the additional constraint $X \sqsupseteq \varphi$. For conditionals, rule TA-COND ensures both the resulting type and the effect are upper bounds of the types resp. effects of the two branches by generating two additional constraints C and C' (cf. Table 10 from Definition 4.2). The `let`-construct (cf. TA-LET) for the sequential composition has an effect $\varphi_1;\varphi_2$. To support context-sensitivity (corresponding to let-polymorphism), the let-rule is where the generalization over the type-level variables happens, i.e., the introduction of $\forall$-quantified types in the binding for x when extending Γ. In the first approximation, given e_1 is of $\hat{T}_1$, variables which do not occur free in Γ can be generalized over to obtain $\hat{S}_1$. To make the rule deterministic and to use the most "polymorphic" representation for $\hat{S}$, which is necessary for an algorithmic formulation, $\hat{S}$ quantifies over the maximal number of variables for which such

generalization is sound. In the setting here, the quantification affects not type variables, but only flow variables ρ and effect variables X. As those variables are connected by the $\sqsubseteq$-relations in the constraints, also variables which do not literally occur in Γ and φ may indirectly affect the variables which do and thus cannot be generalized over either (see Amtoft, Nielson, and Nielson [5]). The close-operation $close(\Gamma, \varphi, C, \hat{T})$ first computes the set of all "relevant" free variables in a type $\hat{T}$ and the constraint C by the operation $close_{\uparrow\downarrow}(fv(\hat{T}_1), C_1)$(cf. Definition 3.2(3)). Among the set of free variables, those that are free in the context or in the

$$\frac{\Gamma(x) = \forall \vec{Y}{:}C.\hat{T} \qquad \vec{Y}' \textit{ fresh} \qquad \theta = [\vec{Y}'/\vec{Y}]}{\Gamma \vdash x : \theta\hat{T} :: \varepsilon; \theta C}\ \text{TA-VAR} \qquad \frac{\rho \textit{ fresh}}{\Gamma \vdash \texttt{new}^{\pi}\ \texttt{L} : \texttt{L}^{\rho} :: \varepsilon; \rho \sqsupseteq \{\pi\}}\ \text{TA-NEWL}$$

$$\frac{\hat{T}_1 = \lceil T_1 \rceil_a \qquad \Gamma, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: \varphi; C \qquad X \textit{ fresh}}{\Gamma \vdash \texttt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \varepsilon; C, X \sqsupseteq \varphi}\ \text{TA-ABS}_1$$

$$\frac{\lceil T_1 \to T_2 \rceil_a = \hat{T}_1 \xrightarrow{X} \hat{T}_2 \qquad \Gamma, f{:}\hat{T}_1 \xrightarrow{X} \hat{T}_2, x{:}\hat{T}_1 \vdash e : \hat{T}_2' :: \varphi; C_1 \qquad \hat{T}_2' \leq \hat{T}_2 \vdash_a C_2}{\Gamma \vdash \texttt{fun}\, f{:}T_1 \to T_2, x{:}T_1.e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \varepsilon; C_1, C_2, X \sqsupseteq \varphi}\ \text{TA-ABS}_2$$

$$\frac{\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon; C_1 \qquad \Gamma \vdash v_2 : \hat{T}_2' :: \varepsilon; C_2 \qquad \hat{T}_2' \leq \hat{T}_2 \vdash_a C_3 \qquad X \textit{ fresh}}{\Gamma \vdash v_1\ v_2 : \hat{T}_1 :: X; C_1, C_2, C_3, X \sqsupseteq \varphi}\ \text{TA-APP}$$

$$\frac{\begin{array}{c}\lfloor \hat{T} \rfloor = \lfloor \hat{T}_1 \rfloor = \lfloor \hat{T}_2 \rfloor \qquad \hat{T}_1 \vee \hat{T}_2 = \hat{T}; C \qquad \varphi_1 \sqcup \varphi_2 = X; C' \\ \Gamma \vdash v : \texttt{Bool} :: \varepsilon; C_0 \quad \Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1; C_1 \quad \Gamma \vdash e_2 : \hat{T}_2 :: \varphi_2; C_2\end{array}}{\Gamma \vdash \texttt{if}\, v\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \hat{T} :: X; C_0, C_1, C_2, C, C'}\ \text{TA-COND}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1; C_1 \qquad \lfloor \hat{T}_1 \rfloor = T_1 \\ \hat{S}_1 = close(\Gamma, \varphi_1, C_1, \hat{T}_1) \qquad \Gamma, x{:}\hat{S}_1 \vdash e_2 : \hat{T}_2 :: \varphi_2; C_2\end{array}}{\Gamma \vdash \texttt{let}\, x{:}T_1 = e_1\, \texttt{in}\, e_2 : \hat{T}_2 :: \varphi_1; \varphi_2; C_1, C_2}\ \text{TA-LET}$$

$$\frac{\Gamma \vdash t : \hat{T} :: \varphi; C \qquad X \textit{ fresh}}{\Gamma \vdash \texttt{spawn}\, t : \texttt{Unit} :: X; C, X \sqsupseteq \texttt{spawn}\, \varphi}\ \text{TA-SPAWN}$$

$$\frac{\Gamma \vdash v : \texttt{L}^{\rho} :: \varepsilon; C \quad X \textit{ fresh}}{\Gamma \vdash v.\,\texttt{lock} : \texttt{L}^{\rho} :: X; C, X \sqsupseteq \rho.\,\texttt{lock}}\ \text{TA-LOCK} \qquad \frac{\Gamma \vdash v : \texttt{L}^{\rho} :: \varepsilon; C \quad X \textit{ fresh}}{\Gamma \vdash v.\,\texttt{unlock} : \texttt{L}^{\rho} :: X; C, X \sqsupseteq \rho.\,\texttt{unlock}}\ \text{TA-UNLOCK}$$

Table 11: Algorithmic effect inference

effect, as well as the corresponding downward closure (cf. Definition 3.2(2)), are non-generalizable and are excluded.

Definition 4.3 (Closure)**.** *The* closure $close(\Gamma, \varphi, C, \hat{T})$ *of a type* $\hat{T}$ *wrt. a context* Γ*, an effect* φ*, and constraints* C *is given as type scheme* $\forall \vec{Y}{:}C'.\hat{T}$*, where* $\vec{Y} = close_{\uparrow\downarrow}(fv(\hat{T}), C) \setminus close_{\downarrow}(fv(\Gamma, \varphi), C)$ *and* C' *is the largest set where* $C' \subseteq C$ *with for all constraints* $c \in C'$*,* $fv(c) \cap \vec{Y} \neq \emptyset$*.*

The spawn expression is of unit type (cf. TA-SPAWN) and again a fresh variable is used in the generated constraint. Finally, rules TA-LOCK and TA-UNLOCK deal with locking and unlocking an existing lock created at the potential program points indicated by ρ. Both expressions have the same type $\mathtt{L}^{\rho}$, while the effects are $\rho.\,\mathtt{lock}$ and $\rho.\,\mathtt{unlock}$.

The type and effect system works on the thread local level. The definition for the global level is straightforward. If all the processes are well-typed, so is the corresponding global program. A process p is well-typed, denoted as $\vdash p\langle t \rangle :: p\langle \varphi; C \rangle$, if $\vdash t : \hat{T} :: \varphi; C$. In abuse of notation, we use Φ to abbreviate $p_1\langle \varphi_1; C_1 \rangle \parallel \ldots \parallel p_n\langle \varphi_n; C_n \rangle$.

Example 4.4. *This example revisits the motivating example in Listing 1, and illustrates the corresponding derivation by using the algorithmic type and effect system in Table 11. As mentioned, we use* $t_1; t_2$ *as a shorthand for* $\mathtt{let}\ x{:}T = t_1\ \mathtt{in}\ t_2$*, where* x *does not occur free in* t_2*. In the derivation, let* t *abbreviate the code in Listing 1, and* t_0 *be* $t_1; t_2$ *where* $t_1 \triangleq \mathtt{fn}\ x{:}\,\mathtt{L}\ .(x.\,\mathtt{lock}; x.\,\mathtt{lock})$ *and* $t_2 \triangleq \mathtt{spawn}\ (f(x_2)); f(x_1)$*. The derivation starts with an empty context* Γ_0*:*

$$\dfrac{\dfrac{\rho_1\ \textit{fresh}}{\Gamma_0 \vdash \mathtt{new}^{\pi_1}\ \mathtt{L} : \mathtt{L}^{\rho_1} :: \varepsilon; C^{\rho_1}} \qquad \dfrac{\rho_2\ \textit{fresh}}{\Gamma_0 \vdash \mathtt{new}^{\pi_2}\ \mathtt{L} : \mathtt{L}^{\rho_2} :: \varepsilon; C^{\rho_2}} \qquad \dfrac{\vdots}{\Gamma_1 \vdash t_0 : \hat{T}_0 :: \varphi_0; C_0}}{\Gamma_0 \vdash t : \hat{T} :: \varphi; C}$$

where $C^{\rho_1} = \rho_1 \sqsupseteq \{\pi_1\}$ *and* $C^{\rho_2} = \rho_2 \sqsupseteq \{\pi_2\}$*. The derivation continues as shown in the third subgoal as above with the extended context* Γ_1 *which is of the form:*

$$\Gamma_1 = x_1{:}\mathtt{L}^{\rho_1}, x_2{:}\mathtt{L}^{\rho_2}\ .$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1, x{:}\mathrm{L}^{\rho} \vdash x : \mathrm{L}^{\rho} :: \varepsilon;\emptyset \quad X_{f_1}\ \textit{fresh}}{\Gamma_1, x{:}\mathrm{L}^{\rho} \vdash x.\,\texttt{lock}{:}\ \mathrm{L}^{\rho} :: X_{f_1};X_{f_1} \sqsupseteq \rho.\,\texttt{lock}} \quad \dfrac{\vdots}{\Gamma_1, x{:}\mathrm{L}^{\rho} \vdash x.\,\texttt{lock} :\ \ldots}}{\Gamma_1, x{:}\mathrm{L}^{\rho} \vdash x.\,\texttt{lock};x.\,\texttt{lock}{:}\ \mathrm{L}^{\rho} :: (X_{f_1};X_{f_2});\ X_{f_1} \sqsupseteq \rho\,.\texttt{lock}, X_{f_2} \sqsupseteq \rho.\,\texttt{lock}} \quad \rho, X_f\ \textit{fresh}}{\Gamma_1 \vdash \texttt{fn}\, x{:}\,\mathrm{L}\,.(x.\,\texttt{lock};x.\,\texttt{lock}) : \mathrm{L}^{\rho} \xrightarrow{X_f} \mathrm{L}^{\rho} :: \varepsilon;C_f} \quad \dfrac{\vdots}{\Gamma_2 \vdash t_2 : \hat{T}_1 :: \varphi_1;C_1}}{\Gamma_1 \vdash t_0 : \hat{T}_0 :: \varphi_0;C_0}$$

The derivation of the second locking expression is analogous, where the effect is captured by another fresh effect variable X_{f_2} with the constraint $X_{f_2} \sqsupseteq \rho.\,\texttt{lock}$. We abbreviate the effect of the function body as $\varphi_f = X_{f_1};X_{f_2}$, and the accumulated constraints of the function abstraction as $C_f = X_{f_1} \sqsupseteq \rho.\,\texttt{lock}, X_{f_2} \sqsupseteq \rho.\,\texttt{lock}, X_f \sqsupseteq X_{f_1};X_{f_2}$. The typing context is then extended with a binding for the variable f, which corresponds to the function abstraction above, in the following form:

$$\Gamma_2 = \Gamma_1, f{:}\hat{S}_f\ ,$$

where $\hat{S}_f = \forall \rho, X_f, X_{f_1}, X_{f_2}{:}C_f.\mathrm{L}^{\rho} \xrightarrow{X_f} \mathrm{L}^{\rho}$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\textit{cf. eq (3)}\ \ \Gamma_2(f) = \hat{S}_f}{\Gamma_2 \vdash f : \mathrm{L}^{\tilde{\rho}} \xrightarrow{\tilde{X}_f} \mathrm{L}^{\tilde{\rho}} :: \varepsilon;\tilde{\theta}_2 C_f} \quad \dfrac{\Gamma_2(x_2) = \mathrm{L}^{\rho_2}}{\Gamma_2 \vdash x_2 : \mathrm{L}^{\rho_2} :: \varepsilon;\emptyset} \quad \mathrm{L}^{\tilde{\rho}} \geq \mathrm{L}^{\rho_2} \vdash_a \tilde{\rho} \sqsupseteq \rho_2}{\Gamma_2 \vdash f(x_2) : \mathrm{L}^{\tilde{\rho}} :: X_2;C_2} \quad \dfrac{\vdots}{\Gamma \vdash f(x_1) : \mathrm{L}^{\tilde{\rho}'} :: X_1;C_1'}}{\Gamma_2 \vdash \texttt{spawn}\ (f(x_2)) : \texttt{Unit} :: X_3;C_2, X_3 \sqsupseteq \texttt{spawn}\ X_2}}{\Gamma_2 \vdash \texttt{spawn}\ (f(x_2)); f(x_1) : \hat{T}_1 :: \varphi_1;C_1}$$

where X_2 and X_3 are fresh. In addition,

$$\tilde{\rho}, \tilde{X}_f, \tilde{X}_{f_1}, \tilde{X}_{f_2}\ \textit{fresh}, \quad \textit{and} \quad \tilde{\theta}_2 = [\tilde{\rho}, \tilde{X}_f, \tilde{X}_{f_1}, \tilde{X}_{f_2}/\rho, X_f, X_{f_1}, X_{f_2}]\ . \tag{3}$$

Furthermore, we abbreviate the constraints generated for the function application $f(x_2)$ as $C_2 = \tilde{\theta}_2 C_f, \tilde{\rho} \sqsupseteq \rho_2, X_2 \sqsupseteq \tilde{X}_f$. The type, effect and contraints for $f(x_1)$ are derived analogously. With

$$\tilde{\rho}', \tilde{X}_f', \tilde{X}_{f_1}', \tilde{X}_{f_2}'\ \textit{fresh}, \quad \textit{and} \quad \tilde{\theta}_1 = [\tilde{\rho}', \tilde{X}_f', \tilde{X}_{f_1}', \tilde{X}_{f_2}'/\rho, X_f, X_{f_1}, X_{f_2}]\ , \tag{4}$$

we summarise the constraints generated throughout the derivation in the following table:

$$\begin{array}{rcl} C & = & C^{\rho_1}, C^{\rho_2}, C_0 \\ C^{\rho_1} & = & \rho_1 \sqsupseteq \{\pi_1\} \\ C^{\rho_2} & = & \rho_2 \sqsupseteq \{\pi_2\} \\ C_0 & = & C_f, C_1 \\ C_f & = & X_{f_1} \sqsupseteq \rho.\,\texttt{lock}, X_{f_2} \sqsupseteq \rho.\,\texttt{lock}, X_f \sqsupseteq X_{f_1}; X_{f_2} \\ C_1 & = & C_2, X_3 \sqsupseteq \texttt{spawn}\, X_2, C_1' \\ C_1' & = & \tilde{\theta}_1 C_f, \tilde{\rho}' \sqsupseteq \rho_1, X_1 \sqsupseteq \tilde{X}_f' \\ & = & \tilde{X}_{f_1}' \sqsupseteq \tilde{\rho}'.\,\texttt{lock}, \tilde{X}_{f_2}' \sqsupseteq \tilde{\rho}'.\,\texttt{lock}, \tilde{X}_f' \sqsupseteq \tilde{X}_{f_1}'; \tilde{X}_{f_2}', \tilde{\rho}' \sqsupseteq \rho_1, X_1 \sqsupseteq \tilde{X}_f' \\ C_2 & = & \tilde{\theta}_2 C_f, \tilde{\rho} \sqsupseteq \rho_2, X_2 \sqsupseteq \tilde{X}_f \\ & = & \tilde{X}_{f_1} \sqsupseteq \tilde{\rho}.\,\texttt{lock}, \tilde{X}_{f_2} \sqsupseteq \tilde{\rho}.\,\texttt{lock}, \tilde{X}_f \sqsupseteq \tilde{X}_{f_1}; \tilde{X}_{f_2}, \tilde{\rho} \sqsupseteq \rho_2, X_2 \sqsupseteq \tilde{X}_f \end{array} \tag{5}$$

The overall type, effect, and constraints of Listing 1 is

$$t : \mathtt{L}^{\tilde{\rho}'} :: X_3; X_1,\ C\ . \tag{6}$$

With the minimal solution of the constraints C, the type $\mathtt{L}^{\tilde{\rho}'}$ *and effect* $X_3; X_1$ *can be interpreted as follows:*

$$t : \mathtt{L}^{\{\pi_1\}} :: \texttt{spawn}\ (\{\pi_2\}.\texttt{lock}; \{\pi_2\}.\texttt{lock}); \{\pi_1\}.\texttt{lock}; \{\pi_1\}.\texttt{lock} \tag{7}$$

which reflects the structure of the flow information in the concrete program. □

4.1. Equivalence of the two formulations

Before we show that the static analysis is correct with respect to the operational semantics, that is, it correctly over-approximates the behaviour of a program at runtime, we prove that the two alternative formulations of the analysis are equivalent in terms of soundness and completeness. We use $\vdash_s$ to denote the judgments and derivations of the formulation in Section 3.2 and $\vdash_a$ for the ones which generate constraints. As usual, the formulation of soundness is straightforward. The algorithmic formulation ($\vdash_a$) is sound with respect to the specification ($\vdash_s$) if everything derivable in $\vdash_a$ has a valid derivation in $\vdash_s$. Before proving the equivalence, the following lemma relates constraint checking with constraint generation.

Lemma 4.5 (Constraint generation)**.**

1. (a) $\hat{T}_1 \leq \hat{T}_2 \vdash_a C$, *then* $C \vdash \hat{T}_1 \leq \hat{T}_2$.

(b) $\varphi \sqsubseteq X \vdash_a C$, *then* $C \vdash \varphi \sqsubseteq X$.

2. (a) *If* $C \vdash \theta\hat{T}_1 \leq \theta\hat{T}_2$, *then* $\hat{T}_1 \leq \hat{T}_2 \vdash_a C'$ *with* $C \vdash \theta C'$.
   (b) *If* $C \vdash \theta\varphi \sqsubseteq \theta X$, *then* $\varphi \sqsubseteq X \vdash_a C'$ *with* $C \vdash \theta C'$.

Proof. Part 1 is straightforward. For part 2a, since C are a set of simple constraints, C is always consistent. Therefore, $C \vdash \theta\hat{T}_1 \leq \theta\hat{T}_2$ implies $\hat{T}_1 \leq \hat{T}_2 \vdash_a C'$ for some constraint set C. $C \vdash \theta C'$ is then followed by induction on $\hat{T}_1 \leq \hat{T}_2 \vdash_a C'$. Part 2b works analogously. □

Theorem 4.6 (Soundness)**.** *Given* $\Gamma \vdash_a t : \hat{T} :: \varphi; C$, *then* $C; \Gamma \vdash_s t : \hat{T} :: \varphi$.

Proof. By straightforward induction on the derivation by the rules in Table 11.

Case: TA-Var
We are given $\Gamma \vdash_a x : \theta\hat{T} :: \varepsilon; \theta C$ where $\Gamma(x) = \forall\vec{Y}{:}C.\hat{T}$ and $\theta = [\vec{Y}'/\vec{Y}]$ for some fresh variables $\vec{Y}'$. The case follows by T-VAR and T-INST.

Case: TA-NewL
We are given in this case that $\Gamma \vdash_a \texttt{new}^{\pi}\ \texttt{L} : \texttt{L}^{\rho} :: \varepsilon; \rho \sqsupseteq \{\pi\}$ with ρ is fresh. The case follows immediately from T-NEWL.

Case: TA-Abs$_1$
We are given

$$\frac{\hat{T}_1 = \lceil T_1 \rceil_a \qquad \Gamma, x{:}\hat{T}_1 \vdash_a e : \hat{T}_2 :: \varphi; C \qquad X \textit{ fresh}}{\Gamma \vdash_a \texttt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \varepsilon; C'}$$

where $C' = C, X \sqsupseteq \varphi$. By induction on the second premise, we get $C; \Gamma, x{:}\hat{T}_1 \vdash_s e : \hat{T}_2 :: \varphi$. Then, we strengthen the constraint set from C to C'. This together with $\lfloor \lceil T_1 \rceil \rfloor = T_1$, and by T-SUB and T-ABS$_1$, the case follows:

$$\dfrac{\dfrac{\dfrac{C'; \Gamma, x{:}\hat{T}_1 \vdash_s e : \hat{T}_2 :: \varphi \qquad C' \vdash X \sqsupseteq \varphi}{C'; \Gamma, x{:}\hat{T}_1 \vdash_s e : \hat{T}_2 :: X}\ \text{T-SUB} \qquad \lfloor \hat{T}_1 \rfloor = T_1}{C'; \Gamma \vdash_s \texttt{fn}\, x{:}T_1.e : \hat{T}_1 \xrightarrow{X} \hat{T}_2 :: \varepsilon}\ \text{T-ABS}_1$$

The case for TA-ABS$_2$ works analogously.

Case: TA-App
In this case, we assume

$$\frac{\Gamma \vdash_a v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon; C_1 \qquad \Gamma \vdash_a v_2 : \hat{T}_2' :: \varepsilon; C_2 \qquad \hat{T}_2' \leq \hat{T}_2 \vdash_a C_3 \qquad X \textit{ fresh}}{\Gamma \vdash_a v_1\ v_2 : \hat{T}_1 :: X; C} \text{ TA-APP}$$

where $C = C_1, C_2, C_3, X \sqsupseteq \varphi$. Induction on the first premise and strengthening the constraint set from C_1 to C gives $C; \Gamma \vdash_s v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon;$. Similarly, induction on the second premise and strengthening the constraint set from C_2 to C yields $C; \Gamma \vdash_s v_2 : \hat{T}_2' :: \varepsilon$. The case then follows by T-SUB and T-APP:

$$\frac{\dfrac{C; \Gamma \vdash_s v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon \quad C \vdash X \sqsupseteq \varphi}{C; \Gamma \vdash_s v_1 : \hat{T}_2 \xrightarrow{X} \hat{T}_1 :: \varepsilon} \text{ T-SUB} \qquad \dfrac{C; \Gamma \vdash_s v_2 : \hat{T}_2' :: \varepsilon \quad C \vdash \hat{T}_2 \geq \hat{T}_2'}{C; \Gamma \vdash_s v_2 : \hat{T}_2 :: \varepsilon} \text{ T-SUB}}{C; \Gamma \vdash_s v_1\ v_2 : \hat{T}_1 :: X}$$

Case: TA-Cond
By assumption, we are given

$$\frac{\begin{array}{c} \lfloor \hat{T} \rfloor = \lfloor \hat{T}_1 \rfloor = \lfloor \hat{T}_2 \rfloor \qquad \hat{T}; C_3 = \hat{T}_1 \vee \hat{T}_2 \qquad X; C_4 = \varphi_1 \sqcup \varphi_2 \\ \Gamma \vdash_a v : \texttt{Bool} :: \varepsilon; C_0 \quad \Gamma \vdash_a e_1 : \hat{T}_1 :: \varphi_1; C_1 \quad \Gamma \vdash_a e_2 : \hat{T}_2 :: \varphi_2; C_2 \end{array}}{\Gamma \vdash_a \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: X; C_0, C_1, C_2, C_3, C_4}$$

where $C = C_0, C_1, C_2, C_3, C_4$. By induction and also strengthening the constraint sets to C, we get

$$C; \Gamma \vdash_s v : \texttt{Bool} :: \varepsilon, \quad C; \Gamma \vdash_s e_1 : \hat{T}_1 :: \varphi_1 \quad \text{and} \quad C; \Gamma \vdash_s e_2 : \hat{T}_2 :: \varphi_2\ . \tag{8}$$

Since $C \vdash \hat{T} \geq \hat{T}_1$ and $C \vdash \hat{T} \geq \hat{T}_2$ as well as $C \vdash \varphi \sqsupseteq \varphi_1$ and $C \vdash \varphi \sqsupseteq \varphi_2$ (cf. Definition 4.2), the case is concluded by T-SUB and T-COND:

$$\frac{C; \Gamma \vdash_s\ v : \texttt{Bool} :: \varepsilon, \quad C; \Gamma \vdash_s\ e_1 : \hat{T} :: \varphi \qquad C; \Gamma \vdash_s\ e_2 : \hat{T} :: \varphi}{C; \Gamma \vdash_s \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \varphi}$$

Case: TA-Let
We are given in this case

$$\frac{\begin{array}{c} \hat{S}_1 = \mathit{close}(\Gamma, \varphi_1, C_1, \hat{T}_1) \qquad \lfloor \hat{T}_1 \rfloor = T_1 \\ \Gamma \vdash_a e_1 : \hat{T}_1 :: \varphi_1; C_1 \qquad \Gamma, x{:}\hat{S}_1 \vdash_a e_2 : \hat{T}_2 :: \varphi_2; C_2 \end{array}}{\Gamma \vdash_a \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \varphi_1; \varphi_2; C}$$

where $C = C_1, C_2$. Induction on the left lower premise, we get $C_1; \Gamma \vdash_s e_1 : \hat{T}_1 :: \varphi_1$, which by T-GEN further implies $C_1; \Gamma \vdash_s e_1 : \hat{S}_1 :: \varphi_1$. The constraint set is strengthened from C_1 to C. This together with the induction on the right lower premise and strengthening the constraint set to C, we conclude the case by T-LET:

$$\frac{C; \Gamma \vdash_s e_1 : \hat{S}_1 :: \varphi_1 \qquad C; \Gamma, x{:}\hat{S}_1 \vdash_s e_2 : \hat{T}_2 :: \varphi_2}{\Gamma \vdash_s \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \varphi_1; \varphi_2; C}$$

Case: TA-Spawn
Straightforward.

Case: TA-Lock
By assumption, we have

$$\frac{\Gamma \vdash_a v : \mathrm{L}^{\rho} :: \varepsilon; C \quad X \textit{ fresh}}{\Gamma \vdash_a v.\ \texttt{lock}\colon \mathrm{L}^{\rho} :: X; C'}$$

where $C' = C, X \sqsupseteq \rho.\texttt{lock}$. Induction on the premise and strengthening the constraint set from C to C' yields $C'; \Gamma \vdash_s v : \mathrm{L}^{\rho} :: \varepsilon$. We conclude the case by T-LOCK and T-SUB:

$$\frac{\dfrac{C'; \Gamma \vdash_s v : \mathrm{L}^{\rho} :: \varepsilon}{C'; \Gamma \vdash_s v.\ \texttt{lock}\colon \mathrm{L}^{\rho} :: \rho.\texttt{lock}}\ \text{T-LOCK} \qquad C' \vdash X \sqsupseteq \rho.\texttt{lock}}{C'; \Gamma \vdash_s v.\ \texttt{lock}\colon \mathrm{L}^{\rho} :: X}$$

The unlocking case is analogous. □

Completeness is, in general, the opposite of soundness, and the formulation is more involved. While constraints in $\vdash_s$ are given as assumption, $\vdash_a$ generates as little constraints as possible. Thus, one cannot expect that both formulations use the same set of constraints because the ones generated by $\vdash_a$ are weaker and less restrictive than those assumed in $\vdash_s$. Similar relationships also apply to the types and effects. To prove $\vdash_a$ is complete with respect to $\vdash_s$, we first tackle the sources of non-determinism in the specification in Section 3.2, namely, instantiation, generalisation, and weakening the result by subsumption. As a first step, we find a variant of the specification, where the use of the mentioned non-syntax-directed rules is restricted to specific points in a derivation; derivations adhering to that more disciplined use of the rules are called *normalized.* As for generalization/specialization: a normalized derivation uses instantiation as "early" as possible and

generalization as "late" as possible: Instantiation is done only directly after an application of rule T-VAR, i.e., when looking up a variable from the typing context, and generalization is used *only* preceding an application of T-LET, i.e., before extending the typing context with a variable. The normalized system of the type system in Table 6 on page 14 is defined in Table 12. We use $\vdash_n$ to distinguish the judgments and derivations from the other two formulations ($\vdash_s$ and $\vdash_a$). We write $\Gamma_1 \lesssim_\theta \Gamma_2$ for $\Gamma_1 = \theta\Gamma_2$.

$$\frac{\Gamma(x) = \forall\vec{Y}{:}C'.\hat{T} \qquad \forall\vec{Y}{:}C'.\hat{T} \text{ solvable from } C \text{ by } \theta}{C;\Gamma \vdash x : \theta\hat{T} :: \varepsilon}\ \text{T-VAR} \qquad \frac{}{C;\Gamma \vdash l^\rho : \mathtt{L}^\rho :: \varepsilon}\ \text{T-LREF}$$

$$\frac{C \vdash \rho \sqsupseteq \{\pi\}}{C;\Gamma \vdash \mathtt{new}^\pi\ \mathtt{L}\ L : \mathtt{L}^\rho :: \varepsilon}\ \text{T-NEWL} \qquad \frac{\lfloor\hat{T}_1\rfloor = T_1 \qquad C;\Gamma, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: \varphi}{C;\Gamma \vdash \mathtt{fn}\ x{:}T_1.e : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \varepsilon}\ \text{T-ABS}_1$$

$$\frac{\lfloor\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2\rfloor = T_1 \to T_2 \qquad C;\Gamma, f{:}\hat{T}_1 \xrightarrow{\varphi} \hat{T}_2, x{:}\hat{T}_1 \vdash e : \hat{T}_2 :: \varphi}{C;\Gamma \vdash \mathtt{fun}\ f{:}T_1 \to T_2, x{:}T_1.e : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \varepsilon}\ \text{T-ABS}_2$$

$$\frac{C;\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon \qquad C;\Gamma \vdash v_2 : \hat{T}_2' :: \varepsilon \qquad C \vdash \hat{T}_2 \geq \hat{T}_2'}{C;\Gamma \vdash v_1\ v_2 : \hat{T}_1 :: \varphi}\ \text{T-APP}$$

$$\frac{\begin{array}{c} C \vdash \hat{T} \geq \hat{T}_1 \quad C \vdash \hat{T} \geq \hat{T}_2 \quad C \vdash \varphi \sqsupseteq \varphi_1 \quad C \vdash \varphi \sqsupseteq \varphi_2 \\ C;\Gamma \vdash v : \mathtt{Bool} :: \varepsilon \quad C;\Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1 \quad C;\Gamma \vdash e_2 : \hat{T}_2 :: \varphi_2 \end{array}}{C;\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \hat{T} :: \varphi}\ \text{T-COND}$$

$$\frac{\begin{array}{l} \vec{Y} \text{ not free in } \Gamma, C_1, \varphi_1 \qquad \forall\vec{Y}{:}C_2.\hat{T}_1 \text{ solvable from } C_1 \qquad \forall\vec{Y}{:}C_2.\hat{T}_1 \vdash wf \\ C_1, C_2;\Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1 \qquad \lfloor\hat{T}_1\rfloor = T_1 \qquad C_1;\Gamma, x{:}\forall\vec{Y}{:}C_2.\hat{T}_1 \vdash e_2 : \hat{T}_2 :: \varphi_2 \end{array}}{C_1;\Gamma \vdash \mathtt{let}\ x{:}T_1 = e_1\ \mathtt{in}\ e_2 : \hat{T}_2 :: \varphi_1;\varphi_2}\ \text{T-LET}$$

$$\frac{C;\Gamma \vdash t : \hat{T} :: \varphi \qquad C \vdash X \sqsupseteq \mathtt{spawn}\ \varphi}{C;\Gamma \vdash \mathtt{spawn}\ t : \mathtt{Unit} :: X}\ \text{T-SPAWN}$$

$$\frac{C;\Gamma \vdash v : \mathtt{L}^\rho :: \varepsilon \qquad C \vdash X \sqsupseteq \rho.\mathtt{lock}}{C;\Gamma \vdash v.\,\mathtt{lock} : \mathtt{L}^\rho :: X}\ \text{T-LOCK} \qquad \frac{C;\Gamma \vdash v : \mathtt{L}^\rho :: \varepsilon \qquad C \vdash X \sqsupseteq \rho.\mathtt{unlock}}{C;\Gamma \vdash v.\,\mathtt{unlock} : \mathtt{L}^\rho :: X}\ \text{T-UNLOCK}$$

Table 12: Type and effect system (syntax-directed)

In the following, we collect some additional lemmas, needed in particular for

the proof of *completeness* and later *subject reduction*.

Lemma 4.7. *If $dom(\theta) \subseteq fv(C)$, then $fv(ran(\theta)) \subseteq fv(\theta C)$.*

Proof. Straightforward. □

Lemma 4.8. *Assume $C_1 \vdash \theta C_2$ and $\vec{Y} \cap fv(C_1) = \emptyset$, and furthermore $dom(\theta) \subseteq fv(C_2)$ and $dom(\theta) \subseteq \vec{Y}$. Then*

1. *$dom(\theta) \cap fv(C_1) = \emptyset$.*
2. *$dom(\theta) \cap fv(ran(\theta)) = \emptyset$.*
3. *If $\vec{Y} \subseteq fv(C_2)$, then $dom(\theta) = \vec{Y}$.*
4. *$\vec{Y} = fv(C_2) \setminus fv(C_1)$.*

Proof. For part 1: the conditions $dom(\theta) \subseteq \vec{Y}$ and $\vec{Y} \cap fv(C_1) = \emptyset$ immediately imply the result. For part 2: The condition $C_1 \vdash \theta C_2$ implies by convention

$$fv(C_1) \supseteq fv(\theta C_2) \; . \tag{9}$$

We have by assumption that $dom(\theta) \subseteq C_2$ and thus Lemma 4.7 gives $ran(\theta) \subseteq fv(\theta C_2)$. Together with equation (9), this means $ran(\theta) \subseteq fv(C_1)$. Thus the result follows with part 1. For part 3, the inclusion $dom(\theta) \subseteq \vec{Y}$ is given as assumption. For the opposite direction, assume for a contradiction there exists a variable $Y' \in \vec{Y}$ but $Y' \notin dom(\theta)$. The assumption $\vec{Y} \subseteq fv(C_2)$ implies $Y' \in fv(C_2)$. Since $Y' \notin dom(\theta)$, this implies $Y' \in fv(\theta C_2)$, as well. Equation (9) implies that also $Y' \in fv(C_1)$. This contradicts the condition that $fv(C_1) \cap \vec{Y} = \emptyset$. For part 4: that $\vec{Y} \subseteq fv(C_2) \setminus fv(C_1)$ follows from part 3 and the fact that $dom(\theta) \subseteq C_2$. For the opposite direction, assume for a contradiction that there exists a $Y' \in fv(C_2) \setminus fv(C_1)$ and $Y' \notin \vec{Y}$. Since $Y' \in fv(C_2)$ but $Y' \notin dom(\theta)$, $Y' \in fv(\theta C_2) \subseteq C_1$, which contradicts above assumption. □

The following lemmas (about occurrence of free variables in connection with sub-effecting and upward closure) will be needed in the proof of completeness as well as subject reduction later in the paper.

Lemma 4.9. *Given $C \vdash \varphi_2 \sqsupseteq \varphi_1$ and let $\vec{Y}$ be upward closed wrt. C. If $y_1 \in \vec{Y}$ and $y_1 \in fv(\varphi_1)$, then $y_2 \in fv(\varphi_2)$ for some $y_2 \in \vec{Y}$.*

Proof. Straightforward. □

Lemma 4.10. *Given two constraint sets C and C' and a set of variables $\vec{Y}$ with $\vec{Y} \cap fv(C) = \emptyset$. If $\vec{Y}$ is upward closed wrt. C', then it is upward closed wrt. C, C' as well.*

Proof. Straightforward. □

Lemma 4.11. *Assume $C, C' \vdash \varphi_2 \sqsupseteq \varphi_1$ and further $\vec{Y}$ is upward closed wrt. C' and $\vec{Y} \cap fv(C) = \emptyset$. If $fv(\varphi_2) \cap \vec{Y} = \emptyset$, then $\vec{Y} \cap fv(\varphi_1) = \emptyset$.*

Proof. Immediate by Lemma 4.9 and 4.10. □

Lemma 4.12. *If $C, C' \vdash \varphi_2 \sqsupseteq \varphi_1$, $C \vdash \theta C'$, and $dom(\theta) \cap fv(C, \varphi_1, \varphi_2) = \emptyset$, then $C \vdash \varphi_2 \sqsupseteq \varphi_1$.*

Proof. Straightforward. □

Lemma 4.13. *Assume $C_1, C_1' \vdash C_2$, and furthermore $C_1 \vdash \theta C_1'$ for some substitution θ with $dom(\theta) \cap fv(C_1, C_2) = \emptyset$. Then $C_1 \vdash C_2$.*

Proof. Straightforward. □

Lemma 4.14. *If $C, C_2 \vdash \theta_1 C_1$ and $C \vdash \theta_2 C_2$ for some substitutions θ_1 and θ_2, where $dom(\theta_2) \cap fv(C) = \emptyset$, then $C \vdash \theta C_1$, for some substitution θ.*

Proof. Applying θ_2 to the first assumption gives $\theta_2(C, C_2) \vdash \theta_2\theta_1 C_1$, i.e., $\theta_2 C, \theta_2 C_2 \vdash \theta_2\theta_1 C_1$. Since $dom(\theta_2) \cap fv(C) = \emptyset$, this further gives $C, \theta_2 C_2 \vdash \theta_2\theta_1 C_1$. The assumption $C \vdash \theta_2 C_2$ thus implies with Lemma 4.13 $C \vdash \theta_2\theta_1 C_1$, as required. □

Lemma 4.15 (Characterization of subtypes)**.** *If $C \vdash \hat{T} \leq \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2$, then $\hat{T} = \hat{T}_1' \xrightarrow{\varphi'} \hat{T}_2'$ with $C \vdash \hat{T}_1 \leq \hat{T}_1'$, $C \vdash \hat{T}_2' \leq \hat{T}_2$, and $C \vdash \varphi' \sqsubseteq \varphi$.*

Proof. The $C \vdash \hat{T} \leq \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2$ is given by a derivation of the corresponding rules from Table 7. The proof follows then by straightforward induction on the derivation: the case for S-REFL is immediate, the one for S-ARROW follows by inspection of the premises of the S-ARROW rule, and S-TRANS follows by induction, using transitivity of the $\leq$-relation on types and on abstract states. □

Lemma 4.16 (Weakening (type schemes))**.** *Assume $C; \Gamma, x{:}\hat{S}_1 \vdash_n e : \hat{T}_2 :: \varphi$ and $C_1 \vdash \hat{S}_1 \lesssim^g \hat{S}_1'$. Then, $C; \Gamma, x{:}\hat{S}_1' \vdash_n e : \hat{T}_2 :: \varphi$.*

Proof. By induction on the derivation in the specification of Table 12. □

The next theorem formulates completeness of the algorithm with respect to the normalized system: everything that is derivable in $\vdash_n$ is derivable algorithmically in $\vdash_a$. To be precise, the algorithm in general derives more specific type with respect to subtyping, infers more precise effects, and generates weaker constraints; thus, ultimately derives a "more general" judgment.

Theorem 4.17 (Completeness)**.** *Assume* $\Gamma \lesssim_\theta \Gamma'$*, and* $C;\Gamma \vdash_n t : \hat{T} :: \varphi$*, then* $\Gamma \vdash_a t : \hat{T}' :: \varphi';C'$ *such that*

1. $C \vdash \theta' C'$,
2. $C \vdash \theta' \hat{T}' \leq \hat{T}$, *and*
3. $C \vdash \theta' \varphi' \sqsubseteq \varphi$,

where $\theta' = \theta, \theta''$ *for some* θ''.

Proof. Case: $e = x$
We are given

$$\frac{\Gamma(x) = \forall \vec{Y}{:}\tilde{C}.\hat{T} \qquad \forall \vec{Y}{:}\tilde{C}.\hat{T} \text{ solvable from } C \text{ by } \tilde{\theta}}{C;\Gamma \vdash_n x : \tilde{\theta}\hat{T} :: \varepsilon} \tag{10}$$

The assumption $\Gamma \lesssim_\theta \Gamma'$ implies $\Gamma'(x) = \forall \vec{Y}{:}\tilde{C}'.\hat{T}'$ for some $\tilde{C}'$ and $\hat{T}'$ where $\tilde{C} = \theta\tilde{C}'$ and $\hat{T} = \theta\hat{T}'$.

$$\frac{\Gamma'(x) = \forall \vec{Y}{:}\tilde{C}'.\hat{T}' \qquad \tilde{\theta}' = [\vec{Y}'/\vec{Y}] \qquad \vec{Y}' \text{ fresh}}{\Gamma \vdash_a x : \tilde{\theta}'\hat{T}' :: \varepsilon; \tilde{\theta}'\tilde{C}'} \text{ TA-VAR}$$

By definition 3.5, the premise $\forall \vec{Y}{:}\tilde{C}.\hat{T}$ solvable from C by $\tilde{\theta}$ in equation (10) implies that $dom(\tilde{\theta}) \subseteq \vec{Y}$ and $C \vdash \theta\tilde{C}$. Since $\tilde{\theta}\tilde{C} = \tilde{\theta}\theta\tilde{C}' = \tilde{\theta}\theta\tilde{\theta}'^{-1}\tilde{\theta}'C'$ [2], letting $\theta' = \tilde{\theta}\theta\tilde{\theta}'^{-1}$ gives $C \vdash \theta'\tilde{\theta}'\tilde{C}'$, as required. Futhermore, $\theta'\tilde{\theta}'\hat{T}' = \tilde{\theta}\theta\tilde{\theta}'^{-1}\tilde{\theta}'\hat{T}' = \tilde{\theta}\theta\hat{T}' = \tilde{\theta}\hat{T}$ and hence, by reflexivity $C \vdash \theta'\tilde{\theta}'\hat{T}' \leq \tilde{\theta}\hat{T}$, as required. It is immediate for the effect part.

Case: $e = \texttt{new}^\pi\ \texttt{L}$
We are given $C;\Gamma \vdash_n \texttt{new}^\pi\ \texttt{L}\colon\ \texttt{L}^\rho :: \varepsilon$ where $C \vdash \rho \sqsupseteq \{\pi\}$. In the algorithm,

$$\frac{\rho' \textit{ fresh}}{\Gamma \vdash_a \texttt{new}^\pi\ \texttt{L} : \texttt{L}^{\rho'} :: \varepsilon; \rho' \sqsupseteq \{\pi\}}$$

and by setting $\theta' = \theta, [\rho/\rho']$, the case is immediate, using reflexivity. The case for references works similarly.

[2] Note that the left-inverse of $\tilde{\theta}'$ exists because $\tilde{\theta}'$ is an injective mapping.

Case: $\texttt{fn}\, x{:}T_1.e'$
We are given

$$\frac{\lfloor \hat{T}_1 \rfloor = T_1 \qquad C;\Gamma,x{:}\hat{T}_1 \vdash_n e' : \hat{T}_2 :: \varphi}{C;\Gamma \vdash_n \texttt{fn}\, x{:}T_1.e' : \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2 :: \varepsilon}$$

and furthermore, $\Gamma \lesssim_\theta \Gamma'$. Let $\hat{T}_1' = \lceil \hat{T}_1 \rceil_a$, i.e., $\hat{T}_1$ is T_1 annotated with fresh variables. Thus, $\hat{T}_1 = \theta_1 \hat{T}_1'$ where $dom(\theta_1) = fv(\hat{T}_1')$. We then let $\tilde{\theta} = \theta, \theta_1$ and hence

$$\Gamma,x{:}\hat{T}_1 \lesssim_{\tilde{\theta}} \Gamma',x{:}\hat{T}_1' \,. \tag{11}$$

By induction we get $\Gamma',x{:}\hat{T}_1' \vdash_a e' : \hat{T}_2' :: \varphi';C'$, where in addition

$$C \vdash \tilde{\theta}'C', \quad C \vdash \tilde{\theta}'\hat{T}_2' \leq \hat{T}_2, \quad C \vdash \tilde{\theta}'\varphi' \sqsubseteq \varphi \quad \text{and} \quad \tilde{\theta}' = \tilde{\theta}, \tilde{\theta}'' \tag{12}$$

for some $\tilde{\theta}''$. Using TA-$\textsc{Abs}_1$ gives

$$\frac{\lceil \hat{T}_1 \rceil_a = \hat{T}_1' \quad \Gamma',x{:}\hat{T}_1' \vdash_a e' : \hat{T}_2' :: \varphi';C' \quad X \textit{ fresh}}{\Gamma' \vdash_a \texttt{fn}\, x{:}T_1.e' : \hat{T}_1' \xrightarrow{X} \hat{T}_2' :: \varepsilon;C', \varphi' \sqsubseteq X}$$

For the constraint condition of the completeness formulation, let $\theta' = \tilde{\theta}', [\varphi/X]$, and the induction (cf. equation (12)) gives $C \vdash \tilde{\theta}'\varphi' \sqsubseteq \varphi$, which implies $C \vdash \theta'\varphi' \sqsubseteq \theta'X$. This together with $C \vdash \tilde{\theta}'C'$ from the induction in equation (12) gives $C \vdash \theta'(C', \varphi' \sqsubseteq X)$, as required. For the typing part, we have $\theta'\hat{T}_1' = \tilde{\theta}'\hat{T}_1' = \theta_1\hat{T}_1' = \hat{T}_1$. By reflexivity, we have $C \vdash \theta'\hat{T}_1' \geq \hat{T}_1$. Similarly, we have $\theta'\hat{T}_2' = \tilde{\theta}'\hat{T}_2'$. Given by induction that $C \vdash \tilde{\theta}'\hat{T}_2' \leq \hat{T}_2$ in equation (12), we get $C \vdash \theta'\hat{T}_2' \leq \hat{T}_2$. Finally, for the latent effect, we have $\theta'X = \varphi$ which implies $C \vdash \theta'X \sqsubseteq \varphi$ by reflexivity. We conclude the case by having:

$$\frac{C \vdash \theta'\hat{T}_1' \geq \hat{T}_1 \qquad C \vdash \theta'\hat{T}_2' \leq \hat{T}_2 \qquad C \vdash \theta'X \sqsubseteq \varphi}{C \vdash \theta'(\hat{T}_1' \xrightarrow{X} \hat{T}_2') \leq \hat{T}_1 \xrightarrow{\varphi} \hat{T}_2}$$

The case for recursive function works analogously.

Case: $e = v_1 v_2$
By assumption, we are given in this case

$$\frac{C;\Gamma \vdash_n v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon \qquad C;\Gamma \vdash_n v_2 : \hat{T}_2' :: \varepsilon \qquad C \vdash \hat{T}_2' \leq \hat{T}_2}{C;\Gamma \vdash_n v_1\ v_2 : \hat{T}_1 :: \varphi} \tag{13}$$

where $\Gamma \lesssim_\theta \Gamma'$. Induction on v_1 yields $\Gamma' \vdash_a v_1 : \hat{T}' :: \varphi';C_1'$ where

$$C \vdash \theta_1'C_1', \quad C \vdash \theta_1'\hat{T}' \leq \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1, \quad \theta_1' = \theta, \theta_1'' \tag{14}$$

for some θ_1''. By the characterization of subtyping from Lemma 4.15, $\theta_1'\hat{T}' = \tilde{\hat{T}}_2 \xrightarrow{\tilde{\varphi}} \tilde{\hat{T}}_1 = \theta_1'\hat{T}_2'' \xrightarrow{\theta_1'\varphi''} \theta_1'\hat{T}_1''$ where

$$C \vdash \hat{T}_2 \leq \theta_1'\hat{T}_2'', \quad C \vdash \theta_1'\hat{T}_1'' \leq \hat{T}_1, \quad C \vdash \theta_1'\varphi'' \sqsubseteq \varphi\ . \tag{15}$$

Induction on v_2 gives $\Gamma' \vdash_a v_2 : \hat{T}_2''' :: \varepsilon; C_2'$ where

$$C \vdash \theta_2' C_2', \quad C \vdash \theta_2'\hat{T}_2''' \leq \hat{T}_2', \quad \theta_2' = \theta, \theta_2'' \tag{16}$$

for some θ_2''. Wlog. $dom(\theta_1'') \cap dom(\theta_2'') = \emptyset$. Let $\tilde{\theta} = \theta, \theta_1'', \theta_2''$, then by transitvity, the first judgment of equation (15), the second judgment of equation (16) and the last premise of equation (13) yields $C \vdash \theta_2'\hat{T}_2''' \leq \theta_1'\hat{T}_2''$, which implies

$$C \vdash \tilde{\theta}\hat{T}_2''' \leq \tilde{\theta}\hat{T}_2''\ . \tag{17}$$

By Lemma 4.5, this means

$$\tilde{\theta}\hat{T}_2''' \leq \tilde{\theta}\hat{T}_2'' \vdash C_3' \quad \text{with} \quad C \vdash \tilde{\theta}C_3'\ . \tag{18}$$

Applying TA-APP gives

$$\frac{\begin{array}{c}\tilde{\theta}\hat{T}_2''' \leq \tilde{\theta}\hat{T}_2'' \vdash_a C_3' \qquad X \text{ fresh} \\ \Gamma' \vdash_a v_1 : \hat{T}_2'' \xrightarrow{\varphi''} \hat{T}_1'' :: \varphi'; C_1' \qquad \Gamma' \vdash_a v_2 : \hat{T}_2''' :: \varepsilon; C_2'\end{array}}{\Gamma' \vdash_a v_1 v_2 : \hat{T}_1'' :: X; C'} \tag{19}$$

where $C' = C_1', C_2', C_3', \varphi'' \sqsubseteq X$. Setting $\tilde{\theta}' = \tilde{\theta}, [\varphi/X]$, together with the last judgment in equation (15) give $C \vdash \tilde{\theta}'\varphi'' \sqsubseteq \tilde{\theta}'X$. The condition concerning the constraints C can be summed up as follows:

$$C \vdash \tilde{\theta}'C_1', \quad C \vdash \tilde{\theta}'C_2' \quad C \vdash \tilde{\theta}'C_3', \quad \text{and} \quad C \vdash \tilde{\theta}'(\varphi'' \sqsubseteq X) \tag{20}$$

which implies $C \vdash \tilde{\theta}'C'$, as required in part 1. For part 2, $C \vdash \tilde{\theta}\hat{T}_1'' \leq \hat{T}_1$ follows from the second judgment of equation (15). Finally, for part 3, we conclude by reflexivity and the definition of $\tilde{\theta}', [\varphi/X]$, which gives $C \vdash \tilde{\theta}'X \sqsubseteq \varphi$.

Case: $e = \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$
In this case, we are given

$$\frac{\begin{array}{c}C \vdash \hat{T} \geq \hat{T}_1 \quad C \vdash \hat{T} \geq \hat{T}_2 \quad C \vdash \varphi \sqsupseteq \varphi_1 \quad C \vdash \varphi \sqsupseteq \varphi_2 \\ C; \Gamma \vdash_n v : \texttt{Bool} :: \varepsilon \quad C; \Gamma \vdash_n e_1 : \hat{T}_1 :: \varphi_1 \quad C; \Gamma \vdash_n e_2 : \hat{T}_2 :: \varphi_2\end{array}}{C; \Gamma \vdash_n \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T} :: \varphi} \tag{21}$$

and furthermore $\Gamma \lesssim_\theta \Gamma'$. Induction on the subterm v gives $\Gamma' \vdash_a v : \texttt{Bool} :: \varepsilon; C'_0$, where

$$C \vdash \theta'_0 C'_0, \quad C \vdash \theta_0 \texttt{Bool} \leq \texttt{Bool}, \quad C \vdash \theta_0 \varepsilon \sqsubseteq \varepsilon, \quad \text{and} \quad \theta'_0 = \theta, \theta''_0 \tag{22}$$

for some substitution θ''_0. Induction the second subterm gives $\Gamma' \vdash_a e_1 : \hat{T}'_1 :: \varphi'_1; C'_1$ where

$$C \vdash \theta'_1 C'_1, \quad C \vdash \theta'_1 \hat{T}'_1 \leq \hat{T}_1, \quad C \vdash \theta'_1 \varphi'_1 \sqsubseteq \varphi_1, \quad \text{and} \quad \theta'_1 = \theta, \theta''_1 \tag{23}$$

for some substitution θ''_1. By transitivity, the second resp. the third judgment of equation (23), and the first resp. the third premise of equation (21) give

$$C \vdash \theta'_1 \hat{T}'_1 \leq \hat{T}, \quad \text{resp.} \quad C \vdash \theta'_1 \varphi'_1 \sqsubseteq \varphi\ . \tag{24}$$

Similarly, induction on the last subterm e_2 gives $\Gamma' \vdash_a e_2 : \hat{T}'_2 :: \varphi'_2; C'_2$, where

$$C \vdash \theta'_2 C'_2, \quad C \vdash \theta'_2 \hat{T}'_2 \leq \hat{T}_2, \quad C \vdash \theta'_2 \varphi'_2 \sqsubseteq \varphi_2, \quad \text{and} \quad \theta'_2 = \theta, \theta''_2 \tag{25}$$

for some substitution θ''_2. By transitivity, the seond resp. the third judgment of equation (25), and the second resp. the forth premise of equation (21) give

$$C \vdash \theta'_2 \hat{T}'_2 \leq \hat{T}, \quad \text{resp.} \quad C \vdash \theta'_2 \varphi'_2 \sqsubseteq \varphi\ . \tag{26}$$

By rule TA-COND, we get

$$\frac{\begin{array}{ccc} \lfloor \hat{T} \rfloor = \lfloor \hat{T}'_1 \rfloor = \lfloor \hat{T}'_2 \rfloor & \hat{T}'_1 \vee \hat{T}'_2 = \hat{T}'; C'_3 & \varphi'_1 \sqcup \varphi'_2 = X'; C'_4 \\ \Gamma' \vdash_a v : \texttt{Bool} :: \varepsilon; C'_0 & \Gamma' \vdash_a e_1 : \hat{T}'_1 :: \varphi'_1; C'_1 & \Gamma' \vdash_a e_2 : \hat{T}'_2 :: \varphi'_2; C'_2 \end{array}}{\Gamma \vdash_a \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2 : \hat{T}' :: X'; C'}$$

where $C' = C'_0, C'_1, C'_2, C'_3, C'_4$ with $\hat{T}'_1 \leq \hat{T}', \hat{T}'_2 \leq \hat{T}' \vdash C'_3$ and $\varphi'_1 \sqsubseteq X', \varphi'_2 \sqsubseteq X' \vdash C'_4$. Wlog. the domains of the substitutions θ''_0, θ''_1, and θ''_2 are pairwise disjoint. By the definition of $\vee$ on types, $\hat{T}'$ is annotated with *fresh* variables, and hence $(dom(\theta) \cup dom(\theta''_0) \cup dom(\theta''_1) \cup dom(\theta''_2)) \cap fv(\hat{T}') = \emptyset$. Furthermore, $\hat{T} = \tilde{\theta}' \hat{T}'_1$ where $dom(\tilde{\theta}') = fv(\hat{T}')$. Similarly, by the definition of $\sqcup$, X' is a fresh variable. Hence, $(dom(\theta) \cup dom(\theta''_0) \cup dom(\theta''_1) \cup dom(\theta''_2)) \cap fv(X') = \emptyset$. We further know that $\varphi = \tilde{\theta}'' X'$ where $dom(\tilde{\theta}'') = fv(X')$. Now setting $\theta' = \theta, \theta''_0, \theta''_1, \theta''_2, \tilde{\theta}', \tilde{\theta}''$, the first judgment of equation (24) resp. equation (26) implies

$$C \vdash \theta' \hat{T}_1 \leq \theta' \hat{T}' \quad \text{resp.} \quad C \vdash \theta' \hat{T}'_2 \leq \theta' \hat{T}' \tag{27}$$

By Lemma 4.5, that means

$$\hat{T}_1' \leq \hat{T}', \hat{T}_2' \leq \hat{T}' \vdash C_3' \quad \text{with} \quad C \vdash \theta' C_3' \tag{28}$$

Similarly, the second judgment of equation (24) resp. equation (26) implies

$$C \vdash \theta' \varphi_1' \sqsubseteq \theta' X' \quad \text{resp.} \quad C \vdash \theta' \varphi_2' \sqsubseteq \theta' X' \tag{29}$$

which implies again by Lemma 4.5 that

$$\varphi_1' \sqsubseteq X', \varphi_2' \sqsubseteq X' \vdash C_4' \quad \text{with} \quad C \vdash \theta' C_4' \,. \tag{30}$$

Furthermore, the first judgments of the equations (22), (23) and (24) gives

$$C \vdash \theta' C_0', \quad C \vdash \theta' C_1' \quad \text{and} \quad C \vdash \theta' C_2' \,. \tag{31}$$

Hence, equations (28), (30) and (31) imply $C \vdash \theta' C'$, as required for part 1. For part 2 resp. 3, $C \vdash \theta' \hat{T}' \leq \hat{T}$ by induction resp. $C \vdash \theta' X' \sqsubseteq \varphi$ by reflexivity, as required.

Case: $e = \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2$
We are given

$$\frac{\begin{array}{ccc} \vec{Y} \text{ not free in } \Gamma, C_1, \varphi_1 & \forall \vec{Y}{:}C_2.\hat{T}_1 \text{ solvable from } C_1 & \forall \vec{Y}{:}C_2.\hat{T}_1 \vdash wf \\ C_1, C_2; \Gamma \vdash_n e_1 : \hat{T}_1 :: \varphi_1 & \lfloor \hat{T}_1 \rfloor = T_1 & C_1; \Gamma, x{:}\forall \vec{Y}{:}C_2.\hat{T}_1 \vdash_n e_2 : \hat{T}_2 :: \varphi_2 \end{array}}{C_1; \Gamma \vdash_n \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2 :: \varphi_1; \varphi_2} \tag{32}$$

and further $\Gamma \lesssim_\theta \Gamma'$. Induction on e_1 gives $\Gamma' \vdash_a e_1 : \hat{T}_1' :: \varphi_1'; C_1'$, where in addition

$$C_1, C_2 \vdash \theta_1 C_1', \quad C_1, C_2 \vdash \theta_1 \hat{T}_1' \leq \hat{T}_1, \quad C_1, C_2 \vdash \theta \varphi_1' \sqsubseteq \varphi_1 \quad \text{and} \quad \theta_1 = \theta, \theta_1' \tag{33}$$

for some substitution θ_1'. Now let $\hat{S}_1 = \forall \vec{Y}{:}C_2.\hat{T}_1$ and $\hat{S}_2 = close(\Gamma', \varphi_1', C_1', \hat{T}_1') = \forall \vec{Y}'{:}C_1''.\hat{T}_1'$. By the definitions 4.3, $C_1'' \subseteq C_1'$, the first judgment in equation (33) implies $C_1, C_2 \vdash \theta_1 C_1''$. Then, by definition 3.4 and equation (33), as well as the well-formedness definition 3.3, $C_1 \vdash \hat{S}_1 \lesssim^g \hat{S}_2$. The lower second premise of equation (32) can be weakend with Lemma 4.16 to

$$C_1; \Gamma, x{:}\hat{S}_2 \vdash_n e_2 : \hat{T}_2 :: \varphi_2 \tag{34}$$

The assumption $\Gamma \lesssim_\theta \Gamma'$ implies $\Gamma, x{:}\hat{S}_2 \lesssim_\theta \Gamma' \hat{S}_2$. Then, induction on the judgment in equation (34) gives $\Gamma', x{:}\hat{S}_2 \vdash_a e_2 : \hat{T}_2' :: \varphi_2'; C_2'$, where in addition

$$C_1 \vdash \theta_2 C_2', \quad C_1 \vdash \theta_2 \hat{T}_2' \leq \hat{T}_2, \quad C_1 \vdash \theta_2 \varphi_2' \sqsubseteq \varphi_2 \quad \text{and} \quad \theta_2 = \theta, \theta_2' \tag{35}$$

for some substitution θ_2'. By TA-LET, we get, with $\hat{S}_2 = close(\Gamma', \varphi_1', C_1', \hat{T}_1')$, as described above,

$$\frac{\Gamma' \vdash_a e_1 : \hat{T}_1' :: \varphi_1'; C_1' \qquad \Gamma', x{:}\hat{S}_2 \vdash_a e_2 : \hat{T}_2' :: \varphi_2'; C_2'}{\Gamma' \vdash_a \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : \hat{T}_2' :: \varphi_1'; \varphi_2'; C'}$$

where $C' = C_1', C_2'$. By definition 3.5, the second upper judgment in equation (35) implies $C_1 \vdash \tilde{\theta} C_2$ where $dom(\tilde{\theta}) \subseteq \vec{Y}$. The first judgment of the same equation further implies

$$dom(\tilde{\theta}) \cap fv(C_1, \varphi_1) = \emptyset \tag{36}$$

This together with the first judgment in equation (33) gives by Lemma 4.14 that $C_1 \vdash \tilde{\theta}' C_1'$ with $\tilde{\theta}' = \tilde{\theta}, \theta_1$. Wlog. the domains of the substitutions θ_1' and θ_2' are pairwise disjoint. We then set $\theta' = \tilde{\theta}, \theta, \theta_1', \theta_2'$, and $C_1 \vdash \tilde{\theta}' C_1'$ gives $C_1 \vdash \theta' C_1'$ resp. $C_1 \vdash \theta_2 C_2'$ in equation (35) gives $C_1 \vdash \theta' C_2'$ which yields $C_1 \vdash \theta' C'$, as required for part 1. For the typing part, we have $C_1 \vdash \theta' \hat{T}_2' \leq \hat{T}_2$ by induction. For the effect part, the second judgment in equation (33) resp. equation (35) gives

$$C_1, C_2 \vdash \theta' \varphi_1' \sqsubseteq \varphi_1 \quad \text{resp.} \quad C_1 \vdash \theta' \varphi_2' \sqsubseteq \varphi_2\ . \tag{37}$$

By the well-formedness of $\forall \vec{Y}{:}C_2.\hat{T}_1$, $\vec{Y}$ is upward closed with respect to C_2. Then, by Lemma 4.11, equation (36) and the first judgment in equation (36) imply $dom(\tilde{\theta}) \cap fv(\theta' \varphi_1') = \emptyset$, where $dom(\tilde{\theta}) \subseteq \vec{Y}$. Since $C_1 \vdash \tilde{\theta} C_2$ and $dom(\tilde{\theta}) \cap fv(C_1, \varphi_1, \theta' \varphi_1') = \emptyset$, Lemma 4.12 implies $C_1 \vdash \theta' \varphi_1' \sqsubseteq \varphi_1$. This together with the second judgment in equation (37), and by S-SEQ in Table 8 yield $C \vdash \theta'(\varphi_1'; \varphi_2') \sqsubseteq \varphi_1; \varphi_2$, which concludes the case.

Case: $e = \texttt{spawn}\ e'$

In this case, we have by assumption

$$\frac{C; \Gamma \vdash_n t : \hat{T} :: \varphi \qquad C \vdash X \sqsupseteq \texttt{spawn}\ \varphi}{C; \Gamma \vdash_n \texttt{spawn}\ t^{\varphi} : \texttt{Unit} :: X} \tag{38}$$

and furthermore $\Gamma \lesssim_\theta \Gamma'$. By induction on e', $\Gamma' \vdash_a e' : \hat{T}' :: \varphi'; C'$ where in addition

$$C \vdash \theta' C', \quad C \vdash \theta' \hat{T}' \leq \hat{T}, \quad C \vdash \theta' \varphi' \sqsubseteq \varphi, \quad \text{and} \quad \theta' = \theta, \theta'' \tag{39}$$

for some substituion θ''. By applying TA-SPAWN we get

$$\frac{\Gamma \vdash_a t : \hat{T}' :: \varphi'; C' \qquad X'\ fresh}{\Gamma \vdash_a \texttt{spawn}\ t : \texttt{Unit} :: X'; C''}$$

where $C'' = C', X' \sqsupseteq \texttt{spawn}\ \varphi'$. By S-SPAWN in Table 8, and by the third judgment $C \vdash \theta'\varphi' \sqsubseteq \varphi$ in equation (39), we get $C \vdash \texttt{spawn}\ \theta'\varphi' \sqsubseteq \texttt{spawn}\ \varphi$. This together with the right premise in equation (38) implies by transitivity that $C \vdash \texttt{spawn}\ \theta'\varphi' \sqsubseteq X$. Setting $\tilde{\theta} = \theta', [X/X']$ gives $C \vdash \texttt{spawn}\ \tilde{\theta}\varphi' \sqsubseteq \tilde{\theta}X'$, which implies by Lemma 4.5 that $C \vdash \tilde{\theta}(\texttt{spawn}\ \varphi' \sqsubseteq X')$. The first judgment in equation (39) gives $C \vdash \tilde{\theta}C'$, and therefore implies $C \vdash \tilde{\theta}C''$, which concludes part 1 for the constraint set. The case follows by reflexivity for the typing part, $C \vdash \texttt{Unit} \leq \texttt{Unit}$, as well as for the effect part, $C \vdash \tilde{\theta}X' \sqsubseteq X$.

Case: $e = v.\,\texttt{lock}$
In this case, we have

$$\frac{C;\Gamma \vdash_n v : \mathrm{L}^{\rho} :: \varepsilon \qquad C \vdash X \sqsupseteq \rho.\,\texttt{lock}}{C;\Gamma \vdash_n v.\,\texttt{lock}\colon \mathrm{L}^{\rho} :: X} \tag{40}$$

and further $\Gamma \lesssim_{\theta} \Gamma'$. We get by induction on v that $\Gamma' \vdash_a v : \mathrm{L}^{\rho'} :: \varepsilon; C'$. In addition, we have

$$C \vdash \theta'C', \quad C \vdash \theta'\mathrm{L}^{\rho'} \leq \mathrm{L}^{\rho}, \quad C \vdash \theta'\varepsilon \sqsubseteq \varepsilon, \quad \text{and} \quad \theta' = \theta, \theta'' \tag{41}$$

for some substitution θ''. By rule TA-LOCK, we get

$$\frac{\Gamma' \vdash_a v : \mathrm{L}^{\rho'} :: \varepsilon; C' \qquad X' \textit{ fresh}}{\Gamma' \vdash_a v.\,\texttt{lock}\colon \mathrm{L}^{\rho'} :: X'; C''}$$

where $C'' = C', X' \sqsupseteq \rho'.\,\texttt{lock}$. By S-LOCK in Table 7, and the second judgment in equation (41) imply that $C \vdash \rho' \subseteq \rho$. Then, S-LOCK in Table 8 yields $C \vdash \rho'.\,\texttt{lock} \sqsubseteq \rho.\,\texttt{lock}$. The right premise in equation (40) gives by transitivity that $C \vdash \rho'.\texttt{lock} \sqsubseteq X$. Setting $\tilde{\theta} = \theta', [X/X']$ and by Lemma 4.5 imply $C \vdash \tilde{\theta}(\rho'.\texttt{lock} \sqsubseteq X')$, which gives $C \vdash \tilde{\theta}C''$, as required for the constraint part. The typing part follows by induction as $C \vdash \tilde{\theta}\mathrm{L}^{\rho'} \leq \mathrm{L}^{\rho}$, while the effect part follows by reflexivity as $C \vdash \tilde{\theta}X' \sqsubseteq X$, as required.

The unlocking case works analogously.

□

4.2. Annotated semantics

In Section 2, we briefly made use of labelled steps of the operational semantics, used as a technical way to define what it means to wait on a lock and to be deadlocked in terms of transitions (as opposed to refer to the syntactic form of a

program). We postponed the actual formalization of that labelling, which is done next, where the purpose of the labelled version of the semantics is to be able to *relate* the behaviour of the program to the type system. It is in particular needed to formulate and prove the correctness of the analysis with respect to the operational semantics. For one, all global steps need to be annotated by an appropriate label, not only the lock handling steps. Besides that, to be able to state the soundness of the flow analysis (as part of the general soundness), the locks themselves and the labels in the semantics must carry corresponding information. They are therefore augmented to carry the *flow variables* ρ.

The resulting annotated global steps are shown in Table 13 on the following page. Apart from the additional annotations, the rules are identical to the ones from Table 3 earlier. For the annotation of the steps with ρ-variables: The type system, run on the static code, uses flow variables ρ in its derivations. In particular, for type checking lock creation statements $\texttt{new}^{\pi}\ \texttt{L}$, rule T-NEWL checks whether the type is of the appropriate form $\texttt{L}^{\rho}$. We use the corresponding flow variable ρ to annotate the lock creation statement to $\texttt{new}^{\pi}_{\rho}\ \texttt{L}$ to remember which variable the static type system has used in T-NEWL. That variable is then remembered further as annotation on the freshly created lock reference l^{ρ}. The role of the type system for annotating the spawn-steps is similar: in rule T-SPAWN, checking `spawn`-expression, the effect φ is also assumed to be annotated in the syntax, which allows in the reduction rule R-SPAWN here to record φ as part of the reduction label. Note that since the constructs for spawning and for lock creation now are additionally labelled, we assume that the corresponding *typing* rules T-NEWL and T-SPAWN from Tables 6 and 12 are formulated for that augmented syntax accordingly.

4.3. Semantics of the behaviour

Next we are going to define the transition relation on the abstract behaviour with the effect-constraints. Given a constraint set C where $C \vdash a; \varphi_2 \sqsubseteq \varphi_1$, we interpret it as φ_1 may first perform an a-step before executing φ_2. See also [5]. The a is one of the labels from Table 5 which do not include the τ-label.

Definition 4.18. *The transition relation between configurations of the form* $C; \hat{\sigma} \vdash \Phi$ *is given inductively by the rules of Table 14, where we write* $C \vdash \varphi_1 \stackrel{a}{\Rightarrow}_{\sqsubseteq} \varphi_2$ *for* $C \vdash a; \varphi_2 \sqsubseteq \varphi_1$. *The* $\hat{\sigma}$ *represents an* abstract *heap, which is a finite mapping from a flow variable* ρ *and a process identity* p *to a natural number.*

Each transition is labelled with one of the labels in Table 5, and correspondingly captures the three possible steps we describe in the behaviour, namely creating

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

$$\frac{p_2 \text{ fresh}}{\sigma \vdash p_1\langle \mathtt{let}\ x{:}T = \mathtt{spawn}\ t_2^{\varphi}\ \mathtt{in}\ t_1\rangle \xrightarrow{p_1\langle\mathtt{spawn}(\varphi)\rangle} \sigma \vdash p_1\langle \mathtt{let}\ x{:}T = ()\ \mathtt{in}\ t_1\rangle \parallel p_2\langle t_2\rangle}\ \text{R-SPAWN}$$

$$\frac{\sigma' = \sigma[l^{\rho} \mapsto \mathit{free}] \qquad l \text{ is fresh}}{\sigma \vdash p\langle \mathtt{let}\ x{:}T = \mathtt{new}_{\rho}^{\pi}\ \mathrm{L}\ \mathtt{in}\ t\rangle \xrightarrow{p\langle\tau\rangle} \sigma' \vdash p\langle \mathtt{let}\ x{:}T = l^{\rho}\ \mathtt{in}\ t\rangle}\ \text{R-NEWL}$$

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

$$\frac{\sigma(l^{\rho}) = p(n) \qquad \sigma' = \sigma -_p l^{\rho}}{\sigma \vdash p\langle \mathtt{let}\ x{:}T = l^{\rho}.\,\mathtt{unlock}\ \mathtt{in}\ t\rangle \xrightarrow{p\langle l^{\rho}.\mathtt{unlock}\rangle} \sigma' \vdash p\langle \mathtt{let}\ x{:}T = l^{\rho}\ \mathtt{in}\ t\rangle}\ \text{R-UNLOCK}$$

Table 13: Global steps (annotated)

a new process with a given behaviour, locking and unlocking. Analogous to the corresponding case in the concrete semantics, rule RE-SPAWN covers the creation of a new (abstract) thread and leaves the abstract heap unchanged. Taking a lock is specified by rule RE-LOCK which increases the corresponding lock count by one. Unlocking works similarly by decreasing the lock count by one (cf. RE-UNLOCK), where the second premise makes sure the lock count stays non-negative. The transitions of a global effect Φ consist of the transitions of the individual thread (cf. RE-PAR). As stipulated by rule RE-LOCK, the step to take an abstract lock is always enabled, which is in obvious contrast to the behaviour of concrete locks. To ensure that the abstraction preserves deadlocks requires to adapt the definition of what it means that an abstract behaviour waits on a lock (cf. also Definition 2.1 for concrete programs and heaps).

Definition 4.19 (Waiting for a lock ($\Rightarrow_{\sqsubseteq}$))**.** *Given a configuration $C;\hat{\sigma} \vdash \Phi$ where $\Phi = \Phi' \parallel p\langle\varphi\rangle$, a process p* waits *for a lock ρ in $\hat{\sigma} \vdash \Phi$, written as $\mathit{waits}_{\sqsubseteq}(C;\hat{\sigma} \vdash \Phi, p, \rho)$, if $C \vdash \varphi \xrightarrow{\rho.\mathtt{lock}}_{\sqsubseteq} \varphi'$ but $\hat{\sigma}(\rho, q) \geq 1$ for some $q \neq p$.*

Definition 4.20 (Deadlock)**.** *A configuration $C;\hat{\sigma} \vdash \Phi$* is deadlocked *if $\hat{\sigma}(\rho_i, p_i) \geq 1$ and furthermore $\mathit{waits}(C;\hat{\sigma} \vdash \Phi, p_i, \rho_{i+_k 1})$ (where $k \geq 2$ and for all $0 \leq i \leq$*

$$\frac{C;\hat{\sigma} \vdash \Phi_1 \overset{a}{\Rightarrow}_{\sqsubseteq} C;\hat{\sigma}' \vdash \Phi_1'}{C;\hat{\sigma} \vdash \Phi_1 \parallel \Phi_2 \overset{a}{\Rightarrow}_{\sqsubseteq} C;\hat{\sigma}' \vdash \Phi_1' \parallel \Phi_2} \text{ RE-PAR}$$

$$\frac{C \vdash \varphi \xrightarrow{\texttt{spawn}(\varphi'')}_{\sqsubseteq} \varphi'}{C;\hat{\sigma} \vdash p_1\langle\varphi\rangle \xrightarrow{p_1\langle\texttt{spawn}(\varphi'')\rangle}_{\sqsubseteq} C;\hat{\sigma} \vdash p_1\langle\varphi'\rangle \parallel p_2\langle\varphi''\rangle} \text{ RE-SPAWN}$$

$$\frac{C \vdash \varphi \xrightarrow{\rho.\texttt{lock}}_{\sqsubseteq} \varphi' \quad \hat{\sigma}'(\rho,p) = \hat{\sigma}(\rho,p)+1}{C;\hat{\sigma} \vdash p\langle\varphi\rangle \xrightarrow{p\langle\rho.\texttt{lock}\rangle}_{\sqsubseteq} C;\hat{\sigma}' \vdash p\langle\varphi'\rangle} \text{ RE-LOCK}$$

$$\frac{C \vdash \varphi \xrightarrow{\rho.\texttt{unlock}}_{\sqsubseteq} \varphi' \quad \hat{\sigma}(\rho,p) \geq 1 \quad \hat{\sigma}'(\rho,p) = \hat{\sigma}(\rho,p)-1}{C;\hat{\sigma} \vdash p\langle\varphi\rangle \xrightarrow{p\langle\rho.\texttt{unlock}\rangle}_{\sqsubseteq} C;\hat{\sigma}' \vdash p\langle\varphi'\rangle} \text{ RE-UNLOCK}$$

Table 14: Global transitions

$k-1$*). The* $+_k$ *is meant as addition modulo k. A configuration* $C;\hat{\sigma} \vdash \Phi$ contains a deadlock*, if, starting from* $C;\hat{\sigma} \vdash \Phi$*, a deadlocked configuration is reachable; otherwise it is* deadlock free.

Example 4.21. *To detect potential deadlocks in the program in Listing 1, we explore the abstract behaviour* $\varphi = X_3;X_1$ *obtained in equation (6) of Example 4.4 in a process starting from an empty heap, and the constraints C generated during the derivation of the program in equation (6). The execution uses the transition relation on the abstract behaviour defined in Table 14, and Definition 4.18, as well as the orders of behaviours in Table 8. We show the execution of the abstract behaviour for a particular interleaving of the parallel processes,* p_1 *and* p_2*, as follows:*

$$\begin{aligned}
C,[] \vdash p_1\langle X_3;X_1\rangle &\xrightarrow{p_1\langle\texttt{spawn}(X_2)\rangle}_{\sqsubseteq} \\
C,[] \vdash p_1\langle X_1\rangle \parallel p_2\langle X_2\rangle &\xrightarrow{p_1\langle\tilde{\rho}'.\texttt{lock}\rangle}_{\sqsubseteq} \\
C,[\tilde{\rho}' \mapsto p_1(1)] \vdash p_1\langle \tilde{X}'_{f_2}\rangle \parallel p_2\langle X_2\rangle &\xrightarrow{p_2\langle\tilde{\rho}.\texttt{lock}\rangle}_{\sqsubseteq} \\
C,[\tilde{\rho}' \mapsto p_1(1)][\tilde{\rho} \mapsto p_2(1)] \vdash p_1\langle \tilde{X}'_{f_2}\rangle \parallel p_2\langle \tilde{X}_{f_2}\rangle &\xrightarrow{p_1\langle\tilde{\rho}'.\texttt{lock}\rangle}_{\sqsubseteq} \\
C,[\tilde{\rho}' \mapsto p_1(2)][\tilde{\rho} \mapsto p_2(1)] \vdash p_1\langle \varepsilon\rangle \parallel p_2\langle \tilde{X}_{f_2}\rangle &\xrightarrow{p_2\langle\tilde{\rho}.\texttt{lock}\rangle}_{\sqsubseteq} \\
C,[\tilde{\rho}' \mapsto p_1(2)][\tilde{\rho} \mapsto p_2(2)] \vdash p_1\langle \varepsilon\rangle \parallel p_2\langle \varepsilon\rangle &
\end{aligned}$$

Note that with the given constraints, the lock-manipulating steps involve a non-deterministic choice between which abstract lock is affected. For instance, in the

first locking step of the above execution, the relevant constraints for this locking step are (cf. equation (5) of Example 4.4):

$$X_1 \sqsupseteq \tilde{X}'_f,\ \tilde{X}'_f \sqsupseteq \tilde{X}'_{f_1};\tilde{X}'_{f_2},\ \tilde{X}'_{f_1} \sqsupseteq \tilde{\rho}'.\,\texttt{lock},\ \tilde{\rho}' \sqsupseteq \rho_1,\ \rho_1 \sqsupseteq \{\pi_1\}\ .$$

The mentioned locking step can proceed by non-deterministically taking one of the abstract locks, namely $\tilde{\rho}'$ and $\tilde{\rho}_1$.[3] *It is analogous for the other three locking steps.*

The abstract heap $\hat{\sigma} = [\tilde{\rho}' \mapsto p_1(2)][\tilde{\rho} \mapsto p_2(2)]$ at the end of the above execution does not satisfy the condition of deadlock (cf. Definitions 4.19 and 4.20). The executions for all the other possible process interleavings and non-deterministic choices of abstract locks in this example can be done similarly. In this case, no reachable configuration contains a deadlock, and therefore the program code in Listing 1 is deadlock free. □

Example 4.22. *This example illustrates deadlock detection for a program in which locks are created dynamically. In particular, the mapping from concrete locks to their abstract representation is non-injective, i.e., one abstract lock can represent one or more concrete locks. Consider the following piece of code:*

Listing 2: Dynamic lock creation and deadlock analysis

```
let g = fn () . new^π L in
let f = fn (z1:L, z2:L) . ( z1.lock; z2.lock ) in
let x1 = g() in let x2 = g()
in spawn(f(x2, x1)); f(x1, x2)
```

The example shows that after creating two locks by calling the function g twice, a new process is spawned such that both processes are running in parallel. The two processes respectively call the function f to acquire the two locks, but in a reverse order (see Figure 1a). The above program exhibits the well-known "deadly embrace". On the abstract level, both locks are abstracted to the location π, which means that the two processes are taking the same abstract locks (see Figure 1b), but different concrete locks. The analysis starts with the derivation of the code in Listing 2, which is abbreviated with t. The derivation is captured in equation (42).

$$\frac{(\textit{cf. eq (43)}) \qquad (\textit{cf. eq (44)}) \qquad (\textit{cf. eq (45)}) \qquad (\textit{cf. eq (46)}) \qquad (\textit{cf. eq (48)})}{\Gamma_0 \vdash t : \hat{T} :: \varphi;C} \tag{42}$$

[3] Process p_1 cannot take the lock represented by $\{\pi_1\}$ because the operational semantics for the behaviours only works on variables.

With an empty context Γ_0, the derivation starts by analysing the function g which creates a lock at location π.

$$\dfrac{\dfrac{\rho \textit{ fresh}}{\Gamma_0 \vdash \texttt{new}^{\pi}\,\mathrm{L}\colon \mathrm{L}^{\rho} :: \varepsilon, \rho \sqsupseteq \{\pi\}} \qquad X_g \textit{ fresh}}{\Gamma_0 \vdash \texttt{fn}\,().\texttt{new}^{\pi}\,\mathrm{L}\colon \_ \xrightarrow{X_g} \mathrm{L}^{\rho} :: \varepsilon; C_g} \tag{43}$$

where $C_g = \rho \sqsupseteq \{\pi\}, X_g \sqsupseteq \varepsilon$. We ignore the constraint $X_g \sqsupseteq \varepsilon$ later in the example to keep the presentation clean. The derivation continues with an extended context

$$\Gamma_1 = \Gamma_0, g{:}\forall\rho{:}C_{g\_} \xrightarrow{X_g} \mathrm{L}^{\rho}\ .$$

$$\dfrac{\rho_1, \rho_2, X_f \textit{ fresh} \qquad \dfrac{\dfrac{\Gamma_1, z_1{:}\mathrm{L}^{\rho_1}, z_2{:}\mathrm{L}^{\rho_2} \vdash z_1 : \mathrm{L}^{\rho_1} :: \varepsilon; \emptyset \quad X_{f_1} \textit{ fresh}}{\Gamma_1, z_1{:}\mathrm{L}^{\rho_1}, z_2{:}\mathrm{L}^{\rho_2} \vdash z_1.\,\texttt{lock}\colon \mathrm{L}^{\rho_1} :: X_{f_1}, X_{f_1} \sqsupseteq \rho_1.\texttt{lock}} \qquad \dfrac{\vdots}{\Gamma_1, z_1{:}\mathrm{L}^{\rho_1}, z_2{:}\mathrm{L}^{\rho_2} \vdash z_2.\,\texttt{lock}\colon \ldots}}{\Gamma_1, z_1{:}\mathrm{L}^{\rho_1}, z_2{:}\mathrm{L}^{\rho_2} \vdash z_1.\,\texttt{lock}; z_2.\,\texttt{lock}\colon \mathrm{L}^{\rho_2} :: X_{f_1}; X_{f_2}, X_{f_1} \sqsupseteq \rho_1.\texttt{lock}, X_{f_2} \sqsupseteq \rho_2.\texttt{lock}}}{\Gamma_1 \vdash \texttt{fn}\,(z_1, z_2).(z_1.\,\texttt{lock}; z_2.\,\texttt{lock}) : \mathrm{L}^{\rho_1} \times \mathrm{L}^{\rho_2} \xrightarrow{X_f} \mathrm{L}^{\rho_2} :: \varepsilon, C_f} \tag{44}$$

The derivation of the second locking expression is similar, where the effect is captured by another fresh effect variable X_{f_2} with the constraint $X_{f_2} \sqsupseteq \rho_2.\texttt{lock}$. We abbreviate the effect of the function body as $\varphi_f = X_{f_1}; X_{f_2}$, and the accumulated constraints of the function abstraction as $C_f = X_{f_1} \sqsupseteq \rho_1.\texttt{lock}, X_{f_2} \sqsupseteq \rho_2.\texttt{lock}, X_f \sqsupseteq X_{f_1}; X_{f_2}$. The typing context is then extended with a binding for the variable f, which corresponds to the function abstraction above, in the following form:

$$\Gamma_2 = \Gamma_1, f{:}\hat{S}_f\ ,$$

where $\hat{S}_f = \forall\rho_1, \rho_2, X_f, X_{f_1}, X_{f_2}{:}C_f.\mathrm{L}^{\rho_1} \times \mathrm{L}^{\rho_2} \xrightarrow{X_f} \mathrm{L}^{\rho_2}$.

$$\dfrac{\dfrac{\dfrac{\rho_1' \textit{ fresh} \quad \theta_{\rho_1'} = [\rho_1'/\rho] \quad \Gamma_2(g) = \forall\rho{:}C_{g\_} \xrightarrow{X_g} \mathrm{L}^{\rho}}{\Gamma_2 \vdash g : \_ \xrightarrow{X_g} \mathrm{L}^{\rho_1'} :: \varepsilon, \theta_{\rho_1'} C_g} \qquad \Gamma_2 \vdash () : \_ :: \varepsilon; \emptyset \quad X_{g_1} \textit{ fresh}}{\Gamma_2 \vdash g() : \mathrm{L}^{\rho_2'} :: X_{g_1}; X_{g_1} \sqsupseteq X_g} \qquad \dfrac{\vdots}{\Gamma_2 \vdash g() : \ldots}}{\Gamma_2 \vdash g(); g() : \mathrm{L}^{\rho_2'} :: X_{g_1}; X_{g_2}; C_g'} \tag{45}$$

The type and effect for the second function application $g()$ can be derived similarly: ρ_2' is a fresh variable by instantiating the type of g with a substitution $\theta_{\rho_2'} = [\rho_2'/\rho]$. A constraint $\rho_2' \sqsupseteq \{\pi\}$ is generated during the instantiation by applying $\theta_{\rho_2'}$ to C_g. The effect is captured by a fresh variable X_{g_2} with constraints $X_{g_2} \sqsupseteq X_g$ generated. For the overall generated constraints for the sequential composition, we abbreviate $C_g' = \rho_1' \sqsupseteq \{\pi\}, X_{g_1} \sqsupseteq X_g, \rho_2' \sqsupseteq \{\pi\}, X_{g_2} \sqsupseteq X_g$. The derivation proceeds with the extended context

$$\Gamma_3 = \Gamma_2, x_1{:}\mathrm{L}^{\rho_1'}, x_2{:}\mathrm{L}^{\rho_2'} \ .$$

$$\dfrac{\dfrac{\dfrac{(\textit{cf. eq (47)})\ \ \Gamma_3(f) = \hat{S}_f}{\Gamma_3 \vdash f : \mathrm{L}^{\tilde{\rho}_1} \times \mathrm{L}^{\tilde{\rho}_2} \xrightarrow{\tilde{X}_f} \mathrm{L}^{\tilde{\rho}_2} :: \varepsilon;\ \theta_{f_1} C_f} \qquad \dfrac{\Gamma_3(x_2) = \mathrm{L}^{\rho_2'} \quad \Gamma_3(x_1) = \mathrm{L}^{\rho_1'}}{\Gamma_3 \vdash x_2, x_1 : \mathrm{L}^{\rho_2'} \times \mathrm{L}^{\rho_1'} :: \varepsilon;\ \emptyset} \qquad \mathrm{L}^{\tilde{\rho}_1} \times \mathrm{L}^{\tilde{\rho}_2} \geq \mathrm{L}^{\rho_2'} \times \mathrm{L}^{\rho_1'} \vdash C_1'}{\Gamma_3 \vdash f(x_2, x_1) : \mathrm{L}^{\tilde{\rho}_2} :: X_1;\ \theta_{f_1} C_f,\ C_1'}}{\Gamma_3 \vdash \texttt{spawn}\ (f(x_2, x_1)) : \texttt{Unit} :: X_2; C_1} \tag{46}$$

where X_1 and X_2 are fresh, and

$$C_1' = \tilde{\rho}_1 \sqsupseteq \rho_2',\ \tilde{\rho}_2 \sqsupseteq \rho_1',\ X_1 \sqsupseteq \tilde{X}_f \ , \quad \textit{and} \quad C_1 = \theta_{f_1} C_f, C_1', X_2 \sqsupseteq \texttt{spawn}\, X_1 \ .$$

Furthermore,

$$\tilde{\rho}_1, \tilde{\rho}_2, \tilde{X}_f, \tilde{X}_{f_1}, \tilde{X}_{f_2}\ \textit{fresh}, \ \textit{and} \quad \theta_{f_1} = [\tilde{\rho}_1, \tilde{\rho}_2, \tilde{X}_f, \tilde{X}_{f_1}, \tilde{X}_{f_2} / \rho_2', \rho_1', X_f, X_{f_1}, X_{f_2}] \tag{47}$$

The derivation of the function application $f(x_1, x_2)$ is performed analogously.

$$\dfrac{\dfrac{\dfrac{(\textit{cf. eq (49)})\ \ \Gamma_3(f) = \hat{S}_f}{\Gamma_3 \vdash f : \mathrm{L}^{\tilde{\rho}_1'} \times \mathrm{L}^{\tilde{\rho}_2'} \xrightarrow{\tilde{X}_f'} \mathrm{L}^{\tilde{\rho}_2'} :: \varepsilon;\ \theta_{f_2} C_f} \qquad \dfrac{\Gamma_3(x_1) = \mathrm{L}^{\rho_1'} \quad \Gamma_3(x_2) = \mathrm{L}^{\rho_2'}}{\Gamma_3 \vdash x_1, x_2 : \mathrm{L}^{\rho_1'} \times \mathrm{L}^{\rho_2'} :: \varepsilon; \emptyset} \qquad \mathrm{L}^{\tilde{\rho}_1'} \times \mathrm{L}^{\tilde{\rho}_2'} \geq \mathrm{L}^{\rho_1'} \times \mathrm{L}^{\rho_2'} \vdash C_2'}{\Gamma_3 \vdash f(x_1, x_2) : \mathrm{L}^{\tilde{\rho}_2'} :: X_3;\ C_2} \tag{48}$$

where X is fresh, and

$$C_2' = \tilde{\rho}_1' \sqsupseteq \rho_1',\ \tilde{\rho}_2' \sqsupseteq \rho_2',\ X_3 \sqsupseteq \tilde{X}_f' \ , \quad \textit{and} \quad C_2 = \theta_{f_2} C_f,\ C_2' \ .$$

Furthermore,

$$\tilde{\rho}_1', \tilde{\rho}_2', \tilde{X}_f', \tilde{X}_{f_1}', \tilde{X}_{f_2}'\ \textit{fresh}, \ \textit{and} \quad \theta_{f_2} = [\tilde{\rho}_1', \tilde{\rho}_2', \tilde{X}_f', \tilde{X}_{f_1}', \tilde{X}_{f_2}' / \rho_1', \rho_2', X_f, X_{f_1}, X_{f_2}] \tag{49}$$

The overall type, effect, and constraints of Listing 2 is

$$t : \mathtt{L}^{\tilde{\rho}'_2} :: X_{g_1}; X_{g_2}; X_2; X_3; C \tag{50}$$

We summarise the constraints C which are generated throughout the derivation in the following table:

$$\begin{array}{rcl}
C & = & C_g, C_f, C'_g, C_1, C_2 \\
C_g & = & \rho \sqsupseteq \{\pi\}, X_g \sqsupseteq \varepsilon \\
C_f & = & X_{f_1} \sqsupseteq \rho_1.\mathtt{lock}, X_{f_2} \sqsupseteq \rho_2.\mathtt{lock}, X_f \sqsupseteq X_{f_1}; X_{f_2} \\
C'_g & = & \rho'_1 \sqsupseteq \{\pi\}, X_{g_1} \sqsupseteq X_g, \rho'_2 \sqsupseteq \{\pi\}, X_{g_2} \sqsupseteq X_g \\
C_1 & = & \theta_{f_1} C_f, C'_1, X_2 \sqsupseteq \mathtt{spawn}\, X_1 \\
 & = & \tilde{X}_{f_1} \sqsupseteq \tilde{\rho}_1.\mathtt{lock}, \tilde{X}_{f_2} \sqsupseteq \tilde{\rho}_2.\mathtt{lock}, \tilde{X}_f \sqsupseteq \tilde{X}_{f_1}; \tilde{X}_{f_2}, C'_1, X_2 \sqsupseteq \mathtt{spawn}\, X_1 \\
C'_1 & = & \tilde{\rho}_1 \sqsupseteq \rho'_2,\ \tilde{\rho}_2 \sqsupseteq \rho'_1,\ X_1 \sqsupseteq \tilde{X}_f \\
C_2 & = & \theta_{f_2} C_f, C'_2 \\
 & = & \tilde{X}'_{f_1} \sqsupseteq \tilde{\rho}'_1.\mathtt{lock}, \tilde{X}'_{f_2} \sqsupseteq \tilde{\rho}'_2.\mathtt{lock}, \tilde{X}'_f \sqsupseteq \tilde{X}'_{f_1}; \tilde{X}'_{f_2}, C'_2 \\
C'_2 & = & \tilde{\rho}'_1 \sqsupseteq \rho'_1, \tilde{\rho}'_2 \sqsupseteq \rho'_2, X_3 \sqsupseteq \tilde{X}'_f
\end{array} \tag{51}$$

With the minimal solution of the constraints C, the type $\mathtt{L}^{\tilde{\rho}'_2}$ *and the effect* $X_2; X_3$ *in equation (50) can be interpreted as follows:*

$$t : \mathtt{L}^{\pi} :: \mathtt{spawn}\,(\{\pi\}.\,\mathtt{lock};\ \{\pi\}.\,\mathtt{lock});\ \{\pi\}.\,\mathtt{lock};\ \{\pi\}.\,\mathtt{lock}\ . \tag{52}$$

To detect potential deadlocks in the program in Listing 2, we explore the abstract behaviour $\varphi = X_{g_1}; X_{g_2}; X_2; X_3$ *obtained earlier in equation (50) in a process starting from an empty abstract heap, and the constraints C generated during the derivation of the program in equation (51). The execution uses the transition relation on the abstract behaviour defined in Table 14, and Definition 4.18, as well as the orders of behaviours in Table 8. According to the constraints in equation (51), we see that* $X_{g_1} \sqsupseteq X_g \sqsupseteq \varepsilon$ *resp.* $X_{g_2} \sqsupseteq X_g \sqsupseteq \varepsilon$*. We therefore concentrate on the effect* $X_2; X_3$ *for the execution. With a particular interleaving, and a particular choice of abstract lock with respect to the aforementioned constraints C, we show that a deadlocked configuration is reachable from the initial configuration.*

$$\begin{array}{rl}
C, [] \vdash p_1\langle X_2; X_3\rangle & \xrightarrow{p_1\langle \mathtt{spawn}(X_1)\rangle}_{\sqsubseteq} \\
C, [] \vdash p_1\langle X_3\rangle \parallel p_2\langle X_1\rangle & \xrightarrow{p_1\langle \rho'_1.\mathtt{lock}\rangle}_{\sqsubseteq} \\
C, [\rho'_1 \mapsto p_1(1)] \vdash p_1\langle \tilde{X}'_{f_2}\rangle \parallel p_2\langle X_1\rangle & \xrightarrow{p_2\langle \rho'_2.\mathtt{lock}\rangle}_{\sqsubseteq} \\
C, [\rho'_1 \mapsto p_1(1)][\rho'_2 \mapsto p_2(1)] \vdash p_1\langle \tilde{X}'_{f_2}\rangle \parallel p_2\langle \tilde{X}_{f_2}\rangle &
\end{array}$$

Note that although the abstract locks ρ_1' and ρ_2' are respectively held by process p_1 and p_2, it does not block process p_1 and p_2 from taking the abstract lock ρ_2' respectively ρ_1' (cf. rule RE-LOCK *in Table 14). The configuration at the end of the above execution, for which we have $C \vdash \tilde{X}'_{f_2} \xrightarrow{\rho_2'.\mathtt{lock}}_{\sqsubseteq} \varepsilon$ and $C \vdash \tilde{X}_{f_2} \xrightarrow{\rho_1'.\mathtt{lock}}_{\sqsubseteq} \varepsilon$ for processes p_1 and p_2, respectively, implies by definition 4.19 that both processes are waiting for a lock which is being held by each of their neighbour. That is, $waits_{\sqsubseteq}(C; \hat{\sigma} \vdash p_1\langle \tilde{X}'_{f_2} \rangle \parallel \ldots, p_1, \rho_2')$ and $waits_{\sqsubseteq}(C; \hat{\sigma} \vdash \ldots \parallel p_2\langle \tilde{X}_{f_2} \rangle, p_2, \rho_1')$, which satisfies the condition for being a deadlocked configuration (cf. Definition 4.20).* □

4.4. Soundness

A crucial part for soundness of the algorithm with respect to the semantics is preservation of well-typedness under reduction. This includes to check that the operational semantics of the program is over-approximated by the effect given by the type system, i.e., establishing a simulation relation between a program and its effect; in our setting, this relation has to be sensitive to deadlocks. Defining the simulation relation requires to relate concrete heaps with abstract ones where concrete locks are summarized by their point of creation. See the discussion in Section 1.2 for rationale behind the abstraction function and the design of the corresponding deadlock sensitive simulation.

Definition 4.23 (Wait-sensitive heap abstraction)**.** *Given a concrete and an abstract heap σ_1 and $\hat{\sigma}_2$, and a mapping Θ from the lock references of σ_1 to the abstract locks of $\hat{\sigma}_2$, $\hat{\sigma}_2$ is a* wait-sensitive heap abstraction *of σ_1 wrt. Θ, written $\sigma_1 \leq_\Theta \hat{\sigma}_2$, if $\sum_{l \in \{l' \mid \Theta l' = \rho\}} \sigma_1(l,p) \leq \hat{\sigma}_2(\rho,p)$, for all p and ρ. The definition is used analogously for comparing two abstract heaps. In the special case of mapping between the concrete and an abstract heap, we write $\equiv_\Theta$ if the sum of the counters of the concrete locks coincides with the count of the abstract lock.*

Definition 4.24 (Deadlock sensitive simulation $\lesssim^D_\sqsubseteq$)**.** *Assume a heap-mapping Θ and a corresponding wait-sensitive abstraction $\leq_\Theta$. A binary relation R between configurations is a* deadlock sensitive simulation relation *(or just simulation for short) if the following holds. Assume $C_1; \hat{\sigma}_1 \vdash \Phi_1$ R $C_2; \hat{\sigma}_2 \vdash \Phi_2$ with $\hat{\sigma}_1 \leq_\Theta \hat{\sigma}_2$. Then:*

1. *If $C_1; \hat{\sigma}_1 \vdash \Phi_1 \xrightarrow{p\langle a \rangle}_\sqsubseteq C_1; \hat{\sigma}_1' \vdash \Phi_1'$, then $C_2; \hat{\sigma}_2 \vdash \Phi_2 \xrightarrow{p\langle \Theta a \rangle}_\sqsubseteq C_2; \hat{\sigma}_2' \vdash \Phi_2'$ for some $C_2; \hat{\sigma}_2' \vdash \Phi_2'$ with $\hat{\sigma}_1' \leq_\Theta \hat{\sigma}_2'$ and $C_1; \hat{\sigma}_1' \vdash \Phi_1'$ R $C_2; \hat{\sigma}_2' \vdash \Phi_2'$.*
2. *If $waits_\sqsubseteq((C_1; \hat{\sigma}_1 \vdash \Phi_1), p, \rho)$, then $waits_\sqsubseteq((C_2; \hat{\sigma}_2 \vdash \Phi_2), p, \Theta(\rho))$.*

Again, given a heap-mapping Θ*, configuration* $C_1;\hat{\sigma}_1 \vdash \Phi_1$ *is* simulated by $C_2;\hat{\sigma}_2 \vdash \Phi_2$ *(written* $C_1;\hat{\sigma}_1 \vdash \Phi_1 \lesssim^D_{\sqsubseteq} C_2;\hat{\sigma}_2 \vdash \Phi_2$*), if there exists a deadlock sensitive simulation for* Θ *s.t.* $C_1;\hat{\sigma}_1 \vdash \Phi_1 \; R \; C_2;\hat{\sigma}_2 \vdash \Phi_2$.

The definition is used analogously for simulations between program and effect configurations, i.e., for $\sigma_1 \vdash P \lesssim^D_{\sqsubseteq} C;\hat{\sigma}_2 \vdash \Phi$*. In that case, the transition relation* $\stackrel{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq}$ *is replaced by* $\stackrel{p\langle a\rangle}{\Longrightarrow}$ *for the program configurations.*

$$\begin{array}{ccc} C_1;\hat{\sigma}_1 \vdash \Phi_1 & \stackrel{p\langle a\rangle}{\Longrightarrow_{\sqsubseteq}} & C_1;\hat{\sigma}_1' \vdash \Phi_1' \\ \Big| & & \Big| \\ R & & R \\ \Big| & & \Big| \\ C_2;\hat{\sigma}_2 \vdash \Phi_2 & \stackrel{p\langle \Theta a\rangle}{\Longrightarrow_{\sqsubseteq}} & C_2;\hat{\sigma}_2' \vdash \Phi_2' \end{array}$$

Figure 2: Deadlock sensitive simulation $\lesssim^D_{\sqsubseteq}$

The notation $\stackrel{p\langle a\rangle}{\Longrightarrow}$ is used for weak transitions, defined as $\stackrel{p\langle\tau\rangle}{\longrightarrow}{}^* \stackrel{p\langle a\rangle}{\longrightarrow}$. This relation captures the internal steps which are ignored when relating two transition systems by simulation. It is obvious that the binary relation $\lesssim^D_{\sqsubseteq}$ is itself a deadlock simulation. The relation is transitive and reflexive. Thus, if $C_1;\hat{\sigma}_1 \vdash \Phi_1 \lesssim^D_{\sqsubseteq} C_2;\hat{\sigma}_2 \vdash \Phi_2$, the property of deadlock freedom is straightforwardly carried over from the more abstract behaviour to the concrete one (cf. Lemma 4.25).

Lemma 4.25 (Preservation of deadlock freedom)**.** *Assume* $C_1;\hat{\sigma}_1 \vdash \Phi_1 \lesssim^D_{\sqsubseteq} C_2;\hat{\sigma}_2 \vdash \Phi_2$*. If* $C_2;\hat{\sigma}_2 \vdash \Phi_2$ *is deadlock free, then so is* $C_1;\hat{\sigma}_1 \vdash \Phi_1$.

Proof. We prove contra-positively that if the configuration $C_1;\hat{\sigma}_1 \vdash \Phi_1$ contains a deadlock, so does $C_2;\hat{\sigma}_2 \vdash \Phi_2$. Assume that $C_1;\hat{\sigma}_1 \vdash \Phi_1 \stackrel{s}{\Rightarrow}_{\sqsubseteq} C_1;\hat{\sigma}_1' \vdash \Phi_1'$ such that $C_1;\hat{\sigma}_1' \vdash \Phi_1'$ is deadlocked, where s denotes a sequence of labels. By assumption, there exists a simulation relation s.t. $C_1;\hat{\sigma}_1 \vdash \Phi_1 \; R \; C_2;\hat{\sigma}_2 \vdash \Phi_2$. This implies by Definition 4.24 that $C_2;\hat{\sigma}_2 \vdash \Phi_2 \stackrel{s}{\Rightarrow}_{\sqsubseteq} C_2;\hat{\sigma}_2' \vdash \Phi_2'$ s.t.

$$C_1;\hat{\sigma}_1' \vdash \Phi_1' \; R \; C_2;\hat{\sigma}_2' \vdash \Phi_2' \quad \text{and} \quad \hat{\sigma}_1' \leq_\Theta \hat{\sigma}_2' \,, \tag{53}$$

where Θ is a heap-mapping. The configuration $C_1;\hat{\sigma}_1' \vdash \Phi_1'$ is deadlocked means that $\hat{\sigma}_1'(\rho_i, p_i) \geq 1$ and $waits_{\sqsubseteq}((C_1;\hat{\sigma}_1' \vdash \Phi_1'), p_i, \rho_{i+_k 1})$ for some $k \geq 1$. Being in a simulation relation implies with part 2 of Definition 4.24 that $waits_{\sqsubseteq}((C_2;\hat{\sigma}_2' \vdash \Phi_2'), p_i, \Theta(\rho_{i+_k 1}))$. Furthermore, $\hat{\sigma}_1' \leq_\Theta \hat{\sigma}_2'$ in Equation (53) gives $\hat{\sigma}_1'(\Theta(\rho_i), p_i) \geq 1$. Hence, $C_2;\hat{\sigma}_2' \vdash \Phi_2'$ is deadlocked and $C_2;\hat{\sigma}_2 \vdash \Phi_2$ contains a deadlock. □

The next lemma shows compositionality of $\lesssim^D_{\sqsubseteq}$ with respect to parallel composition.

Lemma 4.26 (Compositionality)**.** *Assume* $C;\hat{\sigma}_1 \vdash p\langle\varphi_1\rangle \lesssim^D_{\sqsubseteq} C;\hat{\sigma}_2 \vdash p\langle\varphi_2\rangle$ *for a given heap-mapping* Θ*, then* $C;\hat{\sigma}_1 \vdash \Phi \parallel p\langle\varphi_1\rangle \lesssim^D_{\sqsubseteq} C;\hat{\sigma}_2 \vdash \Theta\Phi \parallel p\langle\varphi_2\rangle$ *(for the same* Θ*)*

Proof. Straightforward. □

The soundness proof for the algorithmic type and effect inference is formulated as a *subject reduction* result such that it captures the deadlock-sensitive simulation. The part for the preservation of typing under substitution is fairly standard and therefore omitted here. For the effects, the system derives the formal behavioural description, i.e., over-approximation, for a program's future behaviour; one hence cannot expect the effect being preserved by reduction. Thus, we relate the behaviour of the program and the behaviour of the effects via a deadlock-sensitive simulation relation.

The proof of subject reduction is best not done using the algorithmic formulation, i.e., on the derivation rules from Table 11. On the other hand, also the original specification from Section 3.2 is problematic for performing the proof. The reason for that lies in the presence of the three non-syntax-directed rules: sub-typing/sub-effecting on the one hand and generalization and instantiation on the other. The rules of the operational semantics are syntax-directed which means that the syntactic structure of an expression or thread determines which step can be taken: the reduction strategy is deterministic (for a single thread). In contrast, the type system does not determine the typing rule for a given syntactic structure which, connecting the type derivations with the operational semantics in the subject reduction proof, necessitates considering different combinations of rules justifying a given typing judgment. To avoid that complication, subject reduction is done for the normalized system presented in Table 12 on page 29.

For the basic step of β-reduction in the proof of subject reduction, one needs preservation of typing under substitution. Since the proof of subject reduction uses the normalized system and since the typing context may associate type *schemes* to variables whereas expressions can carry only types), the formulation of the substitution lemma is slightly more involved (see Lemma 4.28 below). The next lemma is helpful for the substitution lemma in the crucial case for variables.

Lemma 4.27. *Assume* $C_1, C_2; \Gamma \vdash t : \hat{T} :: \varphi$*. If* $C_2 \vdash \theta C_1$ *with* $dom(\theta) \cap fv(\Gamma, C_2, \varphi) = \emptyset$*, then* $C_2; \Gamma \vdash t : \theta\hat{T} :: \varphi$*.*

Proof. Straightforward. □

Lemma 4.28 (Substitution)**.** *Assume* $C_2;\Gamma,x{:}\forall\vec{Y}{:}C_1.\hat{T}_1 \vdash_n t : \hat{T}_2 :: \varphi$ *and* $C_1,C_2;\Gamma \vdash_n v : \hat{T}_1$*. If further* $C_2 \vdash \theta C_1$ *where* $dom(\theta) = \vec{Y}$*, then* $C_2;\Gamma \vdash_n t[v/x] : \theta\hat{T}_2 :: \varphi$*.*

Proof. Straightforward, with the help of Lemma 4.27. □

The following substitution Lemma 4.29 resp., Corollary 4.30 for the normalized system, is more general than the previous one in that the value being substituted is allowed to be a subtype of the variable; on the other hand, we can restrict our attention to types, not type schemes.

Lemma 4.29 (Substitution)**.** *Assume* $C;\Gamma,x{:}\hat{T}_1 \vdash t : \hat{T}_2 :: \varphi$ *and* $C;\Gamma \vdash v : \hat{T}_1'$*. If further* $C \vdash \hat{T}_1' \leq \hat{T}_1$*, then* $C;\Gamma \vdash t[v/x] : \hat{T}_2 :: \varphi$*..*

Proof. Straightforward. □

Corollary 4.30 (Substitution)**.** *Assume* $C;\Gamma,x{:}\hat{T}_1 \vdash t : \hat{T}_2 :: \varphi$ *and* $C;\Gamma \vdash_n v : \hat{T}_1'$*. If further* $C \vdash \hat{T}_1' \leq \hat{T}_1$*, then* $C;\Gamma \vdash t[v/x] : \hat{T}_2' :: \varphi'$ *where* $C \vdash \hat{T}_2' \leq \hat{T}_2$ *and* $C \vdash \varphi' \sqsubseteq \varphi$*, for some* $\hat{T}_2'$ *and* φ'*.*

Proof. A direct consequence of Lemma 4.29 plus soundness and completeness of the normalized system. □

The following is a simple property connecting subtyping and generic instances.

Lemma 4.31. *If* $C,C' \vdash \hat{T}_1 \leq \hat{T}_2$*, then* $C \vdash \forall\vec{Y}{:}C'.\hat{T}_1 \gtrsim^g \forall\vec{Y}{:}C'.\hat{T}_2$*.*

Proof. An easy consequence of Definition 3.4 of generic instance, choosing $C_1 = C_2$ and $\theta = id$. □

The following lemma expresses that typing is preserved when a typing assumption in the context is "strengthened" by using larger type with respect to the $\lesssim^g$-order (thereby weakening the judgment). The lemma is formulated for the specification of the type system. Corollary 4.33, the formulation as needed in the proof of subject reduction, is an easy consequence.

Lemma 4.32 (Weakening (type schemes))**.** *Assume* $C;\Gamma,x{:}\hat{S}_1 \vdash e : \hat{S}_2 :: \varphi$ *and* $C \vdash \hat{S}_1 \lesssim^g \hat{S}_1'$*. Then,* $C;\Gamma,x{:}\hat{S}_1' \vdash e : \hat{S}_2 :: \varphi$*.*

Proof. Straightforward. □

Corollary 4.33 (Weakening (type schemes))**.** *Assume* $C;\Gamma,x{:}\hat{S}_1 \vdash_n e : \hat{T}_2 :: \varphi$ *and* $C \vdash \hat{S}_1 \lesssim^g \hat{S}'_1$*. Then,* $C;\Gamma,x{:}\hat{S}'_1 \vdash_n e : \hat{T}'_2 :: \varphi'$ *where* $C \vdash \hat{T}_2 \geq \hat{T}'_2$ *and* $C \vdash \varphi \sqsupseteq \varphi'$*.*

Proof. A direct consequence of Lemma 4.32. □

Next the proof of subject reduction, which is done for the normalized system.

Lemma 4.34 (Subject reduction for the normalized system)**.** *Let* $\Gamma \vdash_n p\langle t\rangle :: p\langle\varphi;C\rangle$*, and* $\sigma_1 \equiv_\Theta \hat{\sigma}_2$*. Assume furthermore* $\theta \models C$*, where* Θ *is a ground substitution.*

1. $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma'_1 \vdash p\langle t'\rangle$*, then* $\Gamma \vdash_n p\langle t'\rangle :: p\langle\varphi';C\rangle$ *with* $C \vdash \varphi \sqsupseteq \varphi'$*, and* $\sigma'_1 \equiv_\Theta \hat{\sigma}_2$*.*
2. (a) $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle a\rangle} \sigma'_1 \vdash p\langle t'\rangle$ *where* $a \neq \texttt{spawn}\ \varphi''$*, then* $C;\hat{\sigma}_2 \vdash p\langle\varphi\rangle \overset{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq} C;\hat{\sigma}'_2 \vdash p\langle\varphi'\rangle$*,* $\Gamma \vdash_n p\langle t'\rangle :: p\langle\varphi'';C\rangle$*, and furthermore* $C \vdash \varphi' \equiv \varphi''$ *and* $\sigma'_1 \equiv_\Theta \hat{\sigma}'_2$*.*

   (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* $C;\hat{\sigma}_2 \vdash p\langle\varphi\rangle \overset{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq} C;\hat{\sigma}_2 \vdash p\langle\varphi''\rangle \parallel p'\langle\varphi'\rangle$ *and such that* $\Gamma \vdash_n p\langle t''\rangle :: p\langle\varphi''';C\rangle$ *where* $C \vdash \varphi'' \equiv \varphi'''$*, and* $\Gamma \vdash_n p'\langle t'\rangle :: p'\langle\varphi';C\rangle$*.*
3. *If* $waits(\sigma_1 \vdash p\langle t\rangle, p, l^\rho)$*, then* $waits_\sqsubseteq(\hat{\sigma}_2 \vdash p\langle\varphi\rangle, p, \rho)$*.*

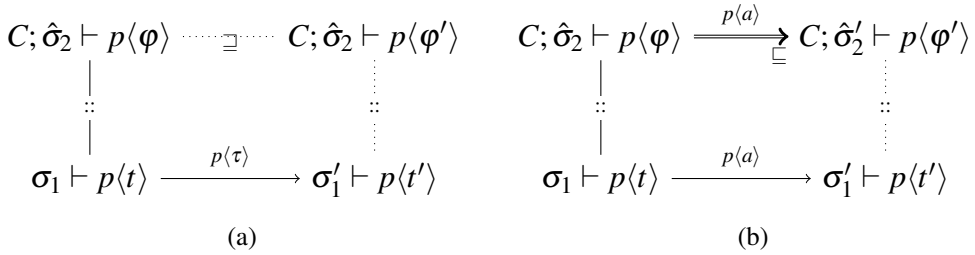


Figure 3: Subject reduction

Proof. We are given $\Gamma \vdash_n p\langle t\rangle :: p\langle\varphi;C\rangle$. 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 reduction steps of Table 2, σ_1 remains unchanged. We proceed by case distinction on the rules for the local transition steps from Table 2 (together with R-LIFT from Table 3).

Case: R-RED: $\sigma_1 \vdash p\langle \texttt{let}\ x{:}T = v\ \texttt{in}\ t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash p\langle t[v/x]\rangle$
By well-typedness, we are given $\Gamma \vdash_n p\langle \texttt{let}\ x{:}T = v\ \texttt{in}\ t\rangle :: p\langle\varphi;C\rangle$, so inverting rules T-THREAD and T-LET gives:

$$\dfrac{\dfrac{C_1,C_2;\Gamma \vdash_n v : \hat{T}_1 :: \varphi_1 \quad \lfloor \hat{T}_1 \rfloor = T_1 \quad \vec{Y} \text{ not free in } C_2,\Gamma,\varphi_1 \quad C_2;\Gamma,x{:}\forall\vec{Y}{:}C_1.\hat{T}_1 \vdash_n t : \hat{T}_2 :: \varphi_2}{C_2;\Gamma \vdash_n \texttt{let}\ x{:}T_1 = v\ \texttt{in}\ t :: \varphi_1;\varphi_2}}{\Gamma \vdash_n p\langle \texttt{let}\ x{:}T_1 = v\ \texttt{in}\ t\rangle :: p\langle\varphi_1;\varphi_2;C_2\rangle}$$

with $\varphi = \varphi_1;\varphi_2$ and where furthermore $C_2 \vdash \theta C_1$ with $dom(\theta) \subseteq \vec{Y}$ and $dom(\theta) \cap fv(\Gamma,C_2,\varphi_1) = \emptyset$. For the effect of the value v, we have $\varphi_1 = \varepsilon$ (cf. the corresponding rules for values T-VAR, T-LREF, T-ABS$_1$, and T-ABS$_2$ for the normalized system from Table 12 on page 29). By preservation of typing under substitution (Lemma 4.28) we get from the last premise from above that $C_2;\Gamma \vdash_n t[v/x] : \hat{T}_2 :: \varphi_2$, and thus

$$\dfrac{C_2;\Gamma \vdash_n t[v/x] : \hat{T}_2 :: \varphi_2}{\Gamma \vdash_n p\langle t[v/x]\rangle :: p\langle\varphi_2;C_2\rangle}\ \text{T-THREAD}$$

where by rule EE-UNIT, $\varphi_1;\varphi_2 = \varepsilon;\varphi_2 \equiv \varphi_2$, and by rule SE-REFL, $C \vdash \varphi_1;\varphi_2 \sqsupseteq \varphi_2$, as required. Since the heaps σ_1 and $\hat{\sigma}_2$ do not change, the relation $\equiv_\Theta$ relates the heaps after the steps, as well.

Case: R-LET: $\sigma_1 \vdash p\langle \texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash p\langle \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2)\rangle$
We are given that $\Gamma \vdash_n p\langle \texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2\rangle :: p\langle(\varphi_1;\varphi_2);\varphi_3;C\rangle$.
Then, by inverting rules T-THREAD, and T-LET twice, we get:

$$\dfrac{\dfrac{\dfrac{C_1,C_2,C;\Gamma \vdash_n e_1 : \hat{T}_1 :: \varphi_1 \quad C_2,C;\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1 \vdash t_1 : \hat{T}_2 :: \varphi_2}{C_2,C;\Gamma \vdash_n \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1 : \hat{T}_2 :: \varphi_1;\varphi_2} \qquad C;\Gamma,x_2{:}\forall\vec{Y}_2{:}C_2.\hat{T}_2 \vdash_n t_2 : \hat{T}_3 :: \varphi_3}{C;\Gamma \vdash_n \texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2 : \hat{T}_3 :: (\varphi_1;\varphi_2);\varphi_3}}{\Gamma \vdash_n p\langle \texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2\rangle :: p\langle(\varphi_1;\varphi_2);\varphi_3;C\rangle} \tag{54}$$

We have further that

$$\vec{Y}_1 \notin fv(C_2,C,\Gamma,\varphi_1) \quad dom(\theta_1) \subseteq \vec{Y}_1 \quad C_2,C \vdash \theta_1 C_1 \tag{55}$$

$$\vec{Y}_2 \notin fv(C,\Gamma,\varphi_1;\varphi_2) \quad dom(\theta_2) \subseteq \vec{Y}_2 \quad C \vdash \theta_2 C_2 \tag{56}$$

Consider now the following derivation tree (using two times T-LET and T-THREAD)

and the additional premises from equations (58) and (59):

$$\dfrac{\dfrac{C_1,C;\Gamma\vdash_n e_1:\hat{T}_1::\varphi_1 \qquad \dfrac{C_2,C;\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1\vdash_n t_1:\hat{T}_2::\varphi_2 \quad C;\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1,x_2{:}\forall\vec{Y}_2{:}C_2.\hat{T}_2\vdash_n t_2:\hat{T}_3::\varphi_3}{C;\Gamma,x{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1\vdash_n \texttt{let}\ x_2{:}T_2=t_1\ \texttt{in}\ t_2:\hat{T}_3::\varphi_2;\varphi_3}}{C;\Gamma\vdash_n \texttt{let}\ x_1{:}T_1=e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2=t_1\ \texttt{in}\ t_2):\hat{T}_3::\varphi_1;(\varphi_2;\varphi_3)}}{\Gamma\vdash_n p\langle\texttt{let}\ x_1{:}T_1=e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2=t_1\ \texttt{in}\ t_2)\rangle::p\langle\varphi_1;(\varphi_2;\varphi_3);C\rangle} \tag{57}$$

$$\vec{Y}_1\notin fv(C,\Gamma,\varphi_1)\quad dom(\theta_1')\subseteq\vec{Y}_1\quad C\vdash\theta_1'C_1 \tag{58}$$

$$\vec{Y}_2\notin fv(C,\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1,\varphi_2)\quad dom(\theta_2')\subseteq\vec{Y}_2\quad C\vdash\theta_2'C_2 \tag{59}$$

They can be justified as follows.

The first condition of (58) directly follows from the corresponding one from (55). The above conditions (55) imply with Lemma 4.8(1), $dom(\theta_1)\cap fv(C_2,C)=\emptyset$, i.e., in particular $dom(\theta_1)\cap fv(C)=\emptyset$. Thus, with Lemma 4.14, the rightmost conditions from (55) and from (56) imply $C\vdash\theta_1'C_1$. As for equation (59): The first condition follow from the first one of (56) wlog., the remaining two are unchanged from (56).

The leftmost $C_1,C_2,C;\Gamma\vdash_n\ e_1:\hat{T}_1::\varphi_1$ leaf in the tree from (54) can be strengthened to

$$C_1,C;\Gamma\vdash_n\ e_1:\theta_2\hat{T}_1::\varphi_1 \tag{60}$$

with Lemma 4.27. Note that the conditions $dom(\theta_2)\cap fv(C,C_1,\Gamma,\hat{T},\varphi_1)=\emptyset$ and $C_1,C\vdash\theta_2C_2$ hold as well (the $dom(\theta_2)\cap fv(C_1,\hat{T}_1)=\emptyset$ holds wlog.). Thus, the judgment of equation (60) corresponds to the left-most subgoal of the tree in (57) The second sub-goal of (57) is directly covered by the second subgoal of (54). The third sub-goal of (57) follows from the third one of (54) by weakening with respect to the typing context. Thus, we conclude by EE-ASSOC$_S$ and SE-REFL that $C\vdash(\varphi_1;\varphi_2);\varphi_3\sqsupseteq\varphi_1;(\varphi_2;\varphi_3)$.

Case: R-IF$_1$: $\sigma_1\vdash p\langle\texttt{let}\ x{:}T=\texttt{if true then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t\rangle\xrightarrow{p\langle\tau\rangle}\sigma_1\vdash p\langle\texttt{let}\ x{:}T=e_1\ \texttt{in}\ t\rangle$

From the well-typedness assumption and inverting rules T-THREAD, T-LET, and

T-IF$_1$, we get:

$$\dfrac{\dfrac{\dfrac{C,C' \vdash \hat{T}' \geq \hat{T}_1 \quad C,C' \vdash \hat{T}' \geq \hat{T}_2 \quad C,C' \vdash \varphi' \sqsupseteq \varphi_1 \quad C,C' \vdash \varphi' \sqsupseteq \varphi_2 \quad C,C';\Gamma \vdash_n \texttt{true} : \texttt{Bool} :: \varepsilon \quad C,C';\Gamma \vdash_n e_1 : \hat{T}_1 :: \varphi_1 \quad C,C';\Gamma \vdash_n e_2 : \hat{T}_2 :: \varphi_2}{C,C';\Gamma \vdash_n \texttt{let } x{:}T = \texttt{if true then } e_1 \texttt{ else } e_2 : \hat{T}' :: \varphi' \quad C;\Gamma,x{:}\forall\vec{Y}'{:}C'.\hat{T}' \vdash_n t : \hat{T} :: \varphi}}{C;\Gamma \vdash_n \texttt{let } x{:}T = \texttt{if true then } e_1 \texttt{ else } e_2 \texttt{ in } t : \hat{T} :: \varphi';\varphi}}{\Gamma \vdash_n p\langle \texttt{let } x{:}T = \texttt{if true then } e_1 \texttt{ else } e_2 \texttt{ in } t\rangle :: p\langle \varphi';\varphi;C\rangle}$$

where $\vec{Y}' \notin fv(C,\Gamma,\varphi'), C \vdash \theta C'$ and $dom(\theta) \subseteq \vec{Y}'$ for some θ. Lemma 4.31 gives $C \vdash \forall\vec{Y}'{:}C'.\hat{T}' \lesssim^g \forall\vec{Y}'{:}C'.\hat{T}_1$, and further by Corollary 4.33, the right-most subgoal can be weakened to $C;\Gamma,x{:}\forall\vec{Y}'{:}C'.\hat{T}_1 \vdash_n t : \hat{T}'' :: \varphi''$ for some $\hat{T}''$ and φ'', where $C \vdash \hat{T}'' \leq \hat{T}$ and $C \vdash \varphi'' \sqsubseteq \varphi$.

Then, we get by applying T-LET and T-THREAD that

$$\dfrac{\dfrac{C,C';\Gamma \vdash_n e_1 : \hat{T}_1 :: \varphi_1 \quad C;\Gamma,x{:}\forall\vec{Y}'{:}C'.\hat{T}_1 \vdash_n t : \hat{T}'' :: \varphi''}{C;\Gamma \vdash_n \texttt{let } x{:}T = e_1 \texttt{ in } t : \hat{T}'' :: \varphi_1;\varphi''}}{\Gamma \vdash_n p\langle \texttt{let } x{:}T = e_1 \texttt{ in } t\rangle :: p\langle \varphi_1;\varphi'';C\rangle}$$

We are given $\vec{Y}' \cap fv(C,\Gamma,\varphi') = \emptyset$ and well-formedness of $\forall\vec{Y}'{:}C'.\hat{T}'$ implies $\vec{Y}'$ is upward closed with respect to C'. Both together with $C,C' \vdash \varphi' \sqsupseteq \varphi_1$ give by Lemma 4.11 that $\vec{Y}' \cap fv(\varphi_1) = \emptyset$. Thus, by Lemma 4.12 implies $C \vdash \varphi' \sqsupseteq \varphi_1$. This together with $C \vdash \varphi \sqsupseteq \varphi''$ implies by SE-SEQ $C \vdash \varphi';\varphi \sqsupseteq \varphi_1;\varphi''$, which concludes the case.

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

Case: R-APP$_1$: $\sigma_1 \vdash p\langle \texttt{let } x_2{:}T_2 = (\texttt{fn } x_1{:}T_1.t_1)\ v \texttt{ in } t_2\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1 \vdash p\langle \texttt{let } x_2{:}T_2 = t_1[v/x_1] \texttt{ in } t_2\rangle$

From the well-typedness assumption and inverting rules T-THREAD, T-LET, T-APP and T-ABS$_1$, we get:

$$\dfrac{\dfrac{\dfrac{\dfrac{\lfloor \hat{T}_1 \rfloor = T_1 \quad C,C';\Gamma,x_1{:}\hat{T}_1 \vdash_n t_1 : \hat{T}_1' :: \varphi_1}{C,C';\Gamma \vdash_n \texttt{fn } x_1{:}T_1.t_1 : \hat{T}_1 \xrightarrow{\varphi_1} \hat{T}_1' :: \varepsilon} \quad C,C';\Gamma \vdash_n v : \hat{T}_1'' :: \varepsilon \quad C,C' \vdash \hat{T}_1 \geq \hat{T}_1''}{C,C';\Gamma \vdash_n (\texttt{fn } x_1{:}T_1.t_1)\ v : \hat{T}_1' :: \varphi_1} \quad C;\Gamma,x_2{:}\forall\vec{Y}{:}C'.\hat{T}_1' \vdash_n t_2 : \hat{T}_2 :: \varphi_2}{C;\Gamma \vdash_n \texttt{let } x_2{:}T_2 = (\texttt{fn } x_1{:}T_1.t_1)\ v \texttt{ in } t_2 : \hat{T}_2 :: \varphi_1;\varphi_2}}{\Gamma \vdash_n p\langle \texttt{let } x_2{:}T_2 = (\texttt{fn } x_1{:}T_1.t_1)\ v \texttt{ in } t_2\rangle :: p\langle \varphi_1;\varphi_2;C\rangle} \tag{61}$$

where $\vec{Y} \notin fv(C,\Gamma,\varphi'), C \vdash \theta C'$ and $dom(\theta) \subseteq \vec{Y}$ for some θ. Using the substitution lemma (Corollary 4.30) on the left-most subgoal gives $C,C';\Gamma \vdash_n t_1[v/x_1] : \hat{T}_1''' :: \varphi_1'$ where

$$C,C' \vdash \hat{T}_1''' \leq \hat{T}_1' \tag{62}$$

and $C,C' \vdash \varphi_1' \sqsubseteq \varphi_1$. Equation (62) implies with Lemma 4.31 $\forall\vec{Y}{:}C'.\hat{T}_1''' \gtrsim^g \forall\vec{Y}{:}C'.\hat{T}_1'$. Thus by using weakening (Corollary 4.33) on the right-most subgoal of (61) we get $C;\Gamma,x_2{:}\forall\vec{Y}{:}C'.\hat{T}_1''' \vdash_n t_2 : \hat{T}_2' :: \varphi_2'$ with $C \vdash \hat{T}_2' \leq \hat{T}_2$ and $C \vdash \varphi_2' \sqsubseteq \varphi_2$. Thus we can derive

$$\dfrac{\dfrac{C,C';\Gamma \vdash_n t_1[v/x_1] : \hat{T}_1''' :: \varphi_1' \quad C;\Gamma,x_2{:}\forall\vec{Y}{:}C'.\hat{T}_1''' \vdash_n t_2 : \hat{T}_2' :: \varphi_2'}{C;\Gamma \vdash_n \mathtt{let}\ x_2{:}T = t_1[v/x_1]\ \mathtt{in}\ t_2 : \hat{T}_2' :: \varphi_1';\varphi_2'}}{\Gamma \vdash_n p\langle \mathtt{let}\ x_2{:}T = t_1[v/x_1]\ \mathtt{in}\ t_2\rangle :: p\langle \varphi_1';\varphi_2';C\rangle}$$

Therefore, we conclude the case observing that $C \vdash \varphi_1;\varphi_2 \sqsupseteq \varphi_1';\varphi_2'$ (by SE-SEQ).

Case: R-APP$_2$: $\sigma_1 \vdash p\langle \mathtt{let}\ x_2{:}T_2 = (\mathtt{fun}\ f{:}T_f.x_1{:}T_1.t_1)v\ \mathtt{in}\ t_2\rangle \xrightarrow{p\langle\tau\rangle}$
$\sigma_1 \vdash p\langle \mathtt{let}\ x_2{:}T_2 = t_1[v/x_1][\mathtt{fun}\ f{:}T_f.x_1{:}T_1.t_1/f]\ \mathtt{in}\ t_2\rangle$
From the well-typedness assumption and inverting rules T-THREAD, T-LET, T-APP and T-ABS$_2$, we get:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\lfloor \hat{T}_f \rfloor = T_f \quad \hat{T}_f = \hat{T}_1 \xrightarrow{\varphi_1} \hat{T}_1'}{C;C';\Gamma,x_1{:}\hat{T}_1,f{:}\hat{T}_f \vdash_n t_1 : \hat{T}_1' :: \varphi_1}}{C,C';\Gamma \vdash_n \mathtt{fun}\ f{:}T_f.x_1{:}T_1.t_1 : \hat{T}_1 \xrightarrow{\varphi_1} \hat{T}_1' :: \varepsilon} \quad C,C';\Gamma \vdash_n v : \hat{T}_1'' :: \varepsilon \quad C,C' \vdash \hat{T}_1 \geq \hat{T}_1''}{C,C';\Gamma \vdash_n (\mathtt{fun}\ f{:}T_f.x_1{:}T_1.t_1)\ v : \hat{T}_1' :: \varphi_1} \quad C;\Gamma,x_2{:}\forall\vec{Y}{:}C'.\hat{T}_1' \vdash_n t_2 : \hat{T}_2 :: \varphi_2}{C;\Gamma \vdash_n \mathtt{let}\ x_2{:}T_2 = (\mathtt{fun}\ f{:}T_f.x_1{:}T_1.t_1)\ v\ \mathtt{in}\ t_2 : \hat{T}_2 :: \varphi_1;\varphi_2}}{\Gamma \vdash_n p\langle \mathtt{let}\ x_2{:}T_2 = (\mathtt{fun}\ f{:}T_f.x_1{:}T_1.t_1)\ v\ \mathtt{in}\ t_2\rangle :: p\langle \varphi_1;\varphi_2;C\rangle} \tag{63}$$

where $\vec{Y} \notin fv(C,\Gamma,\varphi'), C \vdash \theta C'$ and $dom(\theta) \subseteq \vec{Y}$ for some θ. Using two times the substitution lemma (Corollary 4.30) on the left-most subgoal gives $C,C';\Gamma \vdash_n t_1[v/x_1][\mathtt{fun}\ f{:}T_f.x_1{:}T_1.t_1/f] : \hat{T}_1''' :: \varphi_1'$ where

$$C,C' \vdash \hat{T}_1''' \leq \hat{T}_1' \tag{64}$$

and $C,C' \vdash \varphi_1' \sqsubseteq \varphi_1$. Equation (64) implies with Lemma 4.31 $\forall\vec{Y}{:}C'.\hat{T}_1''' \gtrsim^g \forall\vec{Y}{:}C'.\hat{T}_1'$. Thus by using weakening (Corollary 4.33) on the right-most subgoal of (63) we get $C;\Gamma,x_2{:}\forall\vec{Y}{:}C'.\hat{T}_1''' \vdash_n t_2 : \hat{T}_2' :: \varphi_2'$ with $C \vdash \hat{T}_2' \leq \hat{T}_2$ and $C \vdash \varphi_2' \sqsubseteq \varphi_2$.

Therefore we can derive:

$$\dfrac{\dfrac{C,C';\Gamma \vdash_n t_1[v/x_1][\texttt{fun}\ f{:}T_f.x_1{:}T_1.t_1/f] : \hat{T}_1''' :: \varphi_1' \quad C;\Gamma,x_2{:}\forall\vec{Y}{:}C'.\hat{T}_1' \vdash_n t_2 : \hat{T}_2' :: \varphi_2'}{C;\Gamma \vdash_n \texttt{let}\ x_2{:}T_2 = t_1[v/x_1][\texttt{fun}\ f{:}T_f.x_1{:}T_1.t_1/f]\ \texttt{in}\ t_2 : \hat{T}_2' :: \varphi_1';\varphi_2'}}{\Gamma \vdash_n p\langle \texttt{let}\ x_2{:}T_2 = t_1[v/x_1][\texttt{fun}\ f{:}T_f.x_1{:}T_1.t_1/f]\ \texttt{in}\ t_2\rangle :: p\langle \varphi_1';\varphi_2';C\rangle}$$

Thus, by SE-SEQ, $C \vdash \varphi_1;\varphi_2 \sqsupseteq \varphi_1';\varphi_2'$, as required.

Case: R-NEWL: $\sigma_1 \vdash p\langle \texttt{let}\ x{:}T = \ \texttt{new}^{\pi}_{\rho} L\ \texttt{in}\ t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1' \vdash p\langle \texttt{let}\ x{:}T = l^{\rho}\ \texttt{in}\ t\rangle$ where $\sigma_1' = \sigma_1[l^{\rho} \mapsto \mathit{free}]$ for a fresh l. By the well-typedness assumption and inverting T-THREAD, T-LET, and T-NEWL we get

$$\dfrac{\dfrac{\dfrac{C,C' \vdash \rho \sqsupseteq \{\pi\}}{C,C';\Gamma \vdash_n \ \texttt{new}^{\pi}_{\rho} L : \mathrm{L}^{\rho} :: \varepsilon} \quad C;\Gamma,x{:}\forall\vec{Y}{:}C'.\mathrm{L}^{\rho} \vdash_n t : \hat{T} :: \varphi}{C;\Gamma \vdash_n \texttt{let}\ x{:}T = \ \texttt{new}^{\pi}_{\rho} L\ \texttt{in}\ t : \hat{T} :: \varepsilon;\varphi}}{\Gamma \vdash_n p\langle \texttt{let}\ x{:}T = \ \texttt{new}^{\pi}_{\rho} L\ \texttt{in}\ t\rangle :: p\langle \varepsilon;\varphi;C\rangle}$$

where $\vec{Y} \notin fv(C,\Gamma), C \vdash \theta C'$, and $dom(\theta) \subseteq \vec{Y}$ for some θ. Using T-LET and T-THREAD gives:

$$\dfrac{\dfrac{C,C';\Gamma \vdash_n l^{\rho} : \mathrm{L}^{\rho} :: \varepsilon \quad C;\Gamma,x{:}\forall\vec{Y}{:}C'.\mathrm{L}^{\rho} \vdash_n t : \hat{T} :: \varphi}{C;\Gamma \vdash \texttt{let}\ x{:}T = l^{\rho}\ \texttt{in}\ t : \hat{T} :: \varepsilon;\varphi}}{\Gamma \vdash_n p\langle \texttt{let}\ x{:}T = l^{\rho}\ \texttt{in}\ t\rangle :: p\langle \varepsilon;\varphi;C\rangle}$$

Then, by SE-REFL, $C \vdash \varepsilon;\varphi \sqsupseteq \varepsilon;\varphi$. Finally, $\sigma_1 \equiv_{\Theta} \sigma_2$ before the step implies that also $\sigma_1' \equiv_{\Theta} \sigma_2$ after the step, as the new lock is free initially.

Case: R-LOCK: $\sigma_1 \vdash p\langle \texttt{let}\ x{:}T = l^{\rho}.\,\texttt{lock}\ \texttt{in}\ t\rangle \xrightarrow{p\langle l^{\rho}.\texttt{lock}\rangle} \sigma_1' \vdash p\langle \texttt{let}\ x{:}T = l^{\rho}\ \texttt{in}\ t\rangle$
where $\sigma_1(l^{\rho}) = \mathit{free}$ or $\sigma_1(l^{\rho}) = p(n)$ and $\sigma_1' = \sigma_1 +_p l^{\rho}$. By the well-typedness assumption and by inverting rules T-THREAD, T-LET, T-LOCK, and T-LREF, we get:

$$\dfrac{\dfrac{\dfrac{C,C';\Gamma \vdash_n l^{\rho} : \mathrm{L}^{\rho} :: \varepsilon \quad C,C' \vdash X \sqsupseteq \rho.\texttt{lock}}{C;C' \vdash_n l^{\rho}.\,\texttt{lock}{:}\ \mathrm{L}^{\rho} :: X} \quad C;\Gamma,x{:}\forall\vec{Y}{:}C'.\mathrm{L}^{\rho} \vdash_n t : \hat{T} :: \varphi}{C;\Gamma \vdash_n \texttt{let}\ x{:}T = l^{\rho}.\,\texttt{lock}\ \texttt{in}\ t : \hat{T} :: X;\varphi}}{\Gamma \vdash_n p\langle \texttt{let}\ x{:}T = l^{\rho}.\,\texttt{lock}\ \texttt{in}\ t\rangle :: p\langle X;\varphi;C\rangle}$$

where $\vec{Y} \notin fv(C,\Gamma,X), C \vdash \theta C'$, and $dom(\theta) \subseteq \vec{Y}$ for some θ. By rules T-LREF, T-LET and T-THREAD, we can derive

$$\frac{\dfrac{C,C';\Gamma \vdash_n l^\rho : \mathrm{L}^\rho :: \varepsilon \quad C;\Gamma,x{:}\forall\vec{Y}{:}C'.\mathrm{L}^\rho \vdash_n t : \hat{T} :: \varphi}{C;\Gamma \vdash_n \mathtt{let}\ x{:}T = l^\rho\ \mathtt{in}\ t : \hat{T} :: \varepsilon;\varphi}}{\Gamma \vdash_n p\langle \mathtt{let}\ x{:}T = l^\rho\ \mathtt{in}\ t\rangle :: p\langle\varepsilon;\varphi;C\rangle}$$

We are given $\vec{Y} \cap fv(C,X) = \emptyset$ and well-formedness of $\forall\vec{Y}{:}C'.\mathrm{L}^\rho$ implies $\vec{Y}$ is upward closed with respect to C'. Both together with $C,C' \vdash X \sqsupseteq \rho.\,\mathtt{lock}$ give by Lemma 4.11 that also $\vec{Y} \cap fv(\rho.\,\mathtt{lock}) = \emptyset$. Thus, all conditions of Lemma 4.12 are satified, yielding $C \vdash X \sqsupseteq \rho.\,\mathtt{lock}$. Then, by SE-SEQ $C \vdash X;\varphi \sqsupseteq \rho.\,\mathtt{lock};\varphi$, i.e., $C \vdash X;\varphi \xRightarrow{\rho.\mathtt{lock}}_{\sqsubseteq} \varphi$. Then, by RE-LOCK we get

$$C;\sigma_2 \vdash p\langle X;\varphi\rangle \xRightarrow{p\langle\rho.\mathtt{lock}\rangle}_{\sqsubseteq} C;\sigma_2' \vdash p\langle\varphi\rangle \tag{65}$$

where $\sigma'(\rho,p) = \sigma(\rho,p)+1$. Thus, the assumption $\sigma_1 \equiv_\Theta \sigma_2$ before the step implies $\sigma_1' \equiv_\Theta \sigma_2'$ after the step. Finally, by EE-UNIT, $C \vdash \varepsilon;\varphi \equiv \varphi$, as required.

The case for R-UNLOCK works analogously.

Part 2b deals with spawn-steps.

Case: R-SPAWN: $\sigma_1 \vdash p_1\langle \mathtt{let}\ x{:}T = \mathtt{spawn}\ t_2^{\varphi_2}\ \mathtt{in}\ t_1\rangle \xrightarrow{p_1\langle\mathtt{spawn}(\varphi_2)\rangle} \sigma_1 \vdash p_1\langle \mathtt{let}\ x{:}T = ()\ \mathtt{in}\ t_1\rangle \parallel p_2\langle t_2\rangle$

By the well-typedness assumption and inverting rules T-THREAD, T-LET, and T-SPAWN gives

$$\frac{\dfrac{\dfrac{C,C';\Gamma \vdash_n t_2 : \hat{T}_2 :: \varphi_2 \quad C,C' \vdash X \sqsupseteq \mathtt{spawn}\ \varphi_2}{C,C';\Gamma \vdash_n \mathtt{spawn}\ t_2^{\varphi_2} : \mathtt{Unit} :: X} \qquad C;\Gamma,x{:}\forall\vec{Y}{:}C'.\,\mathtt{Unit} \vdash_n t_1 : \hat{T}_1 :: \varphi_1}{C;\Gamma \vdash_n \mathtt{let}\ x{:}T = \mathtt{spawn}\ t_2^{\varphi_2}\ \mathtt{in}\ t_1 : \hat{T}_1 :: X;\varphi_1}}{\Gamma \vdash_n p_1\langle \mathtt{let}\ x{:}T = \mathtt{spawn}\ t_2^{\varphi_2}\ \mathtt{in}\ t_1\rangle :: p_1\langle X;\varphi_1;C\rangle}$$

where $\vec{Y} \notin fv(C,\Gamma,X), C \vdash \theta C'$, and $dom(\theta) \subseteq \vec{Y}$ for some θ. Applying rules T-LET and T-THREAD gives:

$$\frac{\dfrac{C,C';\Gamma \vdash_n p_2 : \mathtt{Unit} :: \varepsilon \quad C;\Gamma,x{:}\forall\vec{Y}{:}C'.\,\mathtt{Unit} \vdash_n t_1 : \hat{T}_1 :: \varphi_1}{C;\Gamma \vdash_n \mathtt{let}\ x{:}T = p_2\ \mathtt{in}\ t_1 : \hat{T}_1 :: \varepsilon;\varphi_1}}{\Gamma \vdash_n p_1\langle \mathtt{let}\ x{:}T = p_2\ \mathtt{in}\ t_1\rangle :: p_1\langle\varepsilon;\varphi_1;C\rangle}$$

By well-formedness, $\forall\vec{Y}{:}C'.\,\texttt{Unit}$ implies $\vec{Y}=\emptyset$ and $C'=\emptyset$. Therefore, $C,C'\vdash X\sqsupseteq\texttt{spawn}\,\varphi_2$ implies $C\vdash X\sqsupseteq\texttt{spawn}\,\varphi_2$. Then, by SE-SEQ $C\vdash X;\varphi_1\sqsupseteq\texttt{spawn}\,\varphi_2;\varphi_1$, i.e. $C\vdash X;\varphi_1\xrightarrow{\texttt{spawn}(\varphi_2)}_{\sqsubseteq}\varphi_1$. Hence we get by RE-SPAWN

$$C;\sigma\vdash p_1\langle X;\varphi_1\rangle\xrightarrow{p_1\langle\texttt{spawn}(\varphi_2)\rangle}_{\sqsubseteq}C;\sigma\vdash p_1\langle\varphi_1\rangle\parallel p_2\langle\varphi_2\rangle \tag{66}$$

By EE-UNIT, $C\vdash\varepsilon;\varphi_1\equiv\varphi_1$, as required.

Since $C'=\emptyset$, the left-most subgoal is written as $C;\Gamma\vdash_n t_2:\hat{T}_2::\varphi_2$. Then, we conclude the case by T-THREAD:

$$\frac{C;\Gamma\vdash_n t_2:\hat{T}_2::\varphi_2}{\Gamma\vdash_n p_2\langle t_2\rangle::p_2\langle\varphi_2\rangle}$$

For part 3, we are given $\mathit{waits}(\sigma_1\vdash p\langle t\rangle,p,l^\rho)$, i.e., by Definition 2.1, it is not the case that $\sigma_1\vdash p\langle t\rangle\xrightarrow{p\langle l^\rho\,\texttt{lock}\rangle}$ but $\sigma_1'\vdash p\langle t\rangle\xrightarrow{p\langle l^\rho\,\texttt{lock}\rangle}$ for some σ_1' which implies that for some process q,

$$\sigma_1(l^\rho)=q(n)\quad\text{with}\quad q\neq p\,. \tag{67}$$

The definition further implies that the thread t is of the form $\texttt{let}\;x{:}T=l^\rho.\;\texttt{lock}\;\texttt{in}\;t'$, and we are given more specifically that $\sigma_1\vdash p\langle\texttt{let}\;x{:}T=l^\rho.\;\texttt{lock}\;\texttt{in}\;t'\rangle\not\xrightarrow{p\langle l^\rho\,\texttt{lock}\rangle}$. The well-typedness assumption and inverting rules T-THREAD, T-LET and T-LOCK gives

$$\frac{\dfrac{\dfrac{C,C';\Gamma\vdash_n l^\rho:\mathrm{L}^\rho::\varepsilon\quad C,C'\vdash X\sqsupseteq\rho.\,\texttt{lock}}{C;C'\vdash_n l^\rho.\,\texttt{lock}:\mathrm{L}^\rho::X}\qquad C;\Gamma,x{:}\forall\vec{Y}{:}C'.\mathrm{L}^\rho\vdash_n t':\hat{T}::\varphi}{C;\Gamma\vdash_n\texttt{let}\;x{:}T=l^\rho.\;\texttt{lock}\;\texttt{in}\;t':\hat{T}::X;\varphi}}{\Gamma\vdash_n p\langle\texttt{let}\;x{:}T=l^\rho.\;\texttt{lock}\;\texttt{in}\;t'\rangle::p\langle X;\varphi;C\rangle}$$

where $\vec{Y}\notin\mathit{fv}(C,\Gamma,X),C\vdash\theta C'$ and $\mathit{dom}(\theta)\subseteq\vec{Y}$ for some θ.

We are given $\vec{Y}\cap\mathit{fv}(C,X)=\emptyset$, and furthermore, well-formedness of $\forall\vec{Y}{:}C'.\mathrm{L}^\rho$ implies $\vec{Y}$ is upward closed with respect to C'. Both together with $C,C'\vdash X\sqsupseteq\rho.\,\texttt{lock}$ give by Lemma 4.11 that also $\vec{Y}\cap\mathit{fv}(\rho.\,\texttt{lock})=\emptyset$. Thus, all conditions of Lemma 4.12 are satisfied, yielding $C\vdash X\sqsupseteq\rho.\,\texttt{lock}$. Then, by SE-SEQ $C\vdash X;\varphi\sqsupseteq\rho.\,\texttt{lock};\varphi$, i.e. $C\vdash X;\varphi\xrightarrow{\rho.\,\texttt{lock}}_{\sqsubseteq}\varphi$.

The assumption $\sigma_1\equiv_\Theta\sigma_2$ and equation (67) imply by the wait-equivalence definition (Definition 4.23) that $\sigma_2(\rho,q)\geq 1$. Thus, by Definition 4.19, we have $\mathit{waits}_\sqsubseteq(\sigma_2\vdash p\langle X;\varphi\rangle,p,\rho)$, as required. □

The well-typedness relation between a program and its effect straightforwardly implies a deadlock-preserving simulation:

Corollary 4.35. *Given* $\sigma_1 \equiv_\Theta \sigma_2$ *and* $\Gamma \vdash p\langle t\rangle :: p\langle \varphi; C\rangle$*, then* $\sigma_1 \vdash p\langle t\rangle \lesssim^D_\sqsubseteq C; \sigma_2 \vdash p\langle \varphi\rangle$.

Proof. The weak transition relation $\xRightarrow{p\langle a\rangle}$ is defined as $\xrightarrow{p\langle\tau\rangle}{}^* \xrightarrow{p\langle a\rangle}$. Thus the result follows from subject reduction by induction on the number of τ-steps. □

Lemma 4.36 (Subject reduction)**.** *Let* $\Gamma \vdash_a p\langle t\rangle :: p\langle \varphi; C\rangle$*,* $\sigma_1 \equiv_\Theta \hat{\sigma}_2$*, and* $\theta \models C$.

1. $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1' \vdash p\langle t'\rangle$*, then* $\Gamma \vdash_a p\langle t'\rangle :: p\langle \varphi'; C'\rangle$ *with* $C \vdash \theta' C'$ *for some* θ'*, and furthermore* $C \vdash \varphi \sqsupseteq \theta'\varphi'$*, and* $\sigma_1' \equiv_\Theta \hat{\sigma}_2$.
2. (a) $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle a\rangle} \sigma_1' \vdash p\langle t'\rangle$ *where* $a \neq \texttt{spawn}\ \varphi''$*, then* $C; \hat{\sigma}_2 \vdash p\langle \varphi\rangle \xRightarrow{p\langle a\rangle}_\sqsubseteq C; \hat{\sigma}_2' \vdash p\langle \varphi'\rangle$*,* $\Gamma \vdash_a p\langle t'\rangle :: p\langle \varphi''; C'\rangle$ *with* $C \vdash \theta' C'$*, furthermore* $C \vdash \varphi' \sqsupseteq \theta'\varphi''$ *and* $\sigma_1' \equiv_\Theta \hat{\sigma}_2'$.
   (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* $C; \hat{\sigma}_2 \vdash p\langle \varphi\rangle \xRightarrow{p\langle a\rangle}_\sqsubseteq C; \hat{\sigma}_2 \vdash p\langle \varphi''\rangle \parallel p'\langle \varphi'\rangle$ *and such that* $\Gamma \vdash_a p\langle t''\rangle :: p\langle \varphi'''; C''\rangle$ *with* $C \vdash \theta'' C''$ *and* $C \vdash \varphi'' \sqsupseteq \theta''\varphi'''$*, and furthermore* $\Gamma \vdash_a p'\langle t'\rangle :: p'\langle \varphi''''; C'\rangle$ *with* $C \vdash \theta' C'$ *and* $C \vdash \varphi' \sqsupseteq \theta'\varphi''''$.
3. *If* $waits(\sigma_1 \vdash p\langle t\rangle, p, l)$*, then* $waits_\sqsubseteq(C; \hat{\sigma}_2 \vdash p\langle \varphi\rangle, p, \hat{\theta} l)$.

Proof. We are given that $\Gamma \vdash_a p\langle t\rangle :: p\langle \varphi; C\rangle$, and furthermore in part 1, $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1' \vdash p\langle t'\rangle$. By soundness, we have $\Gamma \vdash_s p\langle t\rangle :: p\langle \varphi; C\rangle$. Then, we get by part 1 in Lemma 4.34 that

$$\Gamma \vdash_s p\langle t'\rangle :: p\langle \varphi''; C\rangle \quad \text{with} \quad C \vdash \varphi \sqsupseteq \varphi'', \quad \text{and} \quad \sigma_1' \equiv_{\hat{\theta}} \sigma_2\,. \tag{68}$$

By the first condition in equation (68) and the completeness (cf. Theorem 4.17), we get

$$\Gamma \vdash_a p\langle t'\rangle :: p\langle \varphi'; C'\rangle, \quad C \vdash \theta' C' \quad \text{and} \quad C \vdash \theta'\varphi' \sqsubseteq \varphi'' \tag{69}$$

The second inequality in equation (68) and the last one in equation (69) give by transitivity that $C \vdash \varphi \sqsupseteq \theta'\varphi'$. This together with $\Gamma \vdash_a p\langle t'\rangle :: p\langle \varphi'; C'\rangle$ and $C \vdash \theta' C'$ in equation (69), and $\sigma_1' \equiv_\Theta \sigma_2$ in equation (68) conclude part 1.

It is analogously for the other parts. □

The well-typedness relation between a program and its effect straightforwardly implies a deadlock-preserving simulation:

Corollary 4.37. *Given* $\sigma_1 \equiv_\Theta \hat{\sigma}_2$ *and* $\Gamma \vdash p\langle t \rangle :: p\langle \varphi; C \rangle$, *then* $\sigma_1 \vdash p\langle t \rangle \lesssim^D_\sqsubseteq C; \hat{\sigma}_2 \vdash p\langle \varphi \rangle$.

Proof. The result follows from soundness in Theorem 4.6, Corollary 4.35 and completeness in Theorem 4.17. □

5. Conclusion

We have presented a constraint-based type and effect inference algorithm for deadlock checking. It infers a behavioural description of a thread's behaviour concerning its lock interactions which then is used to explore the abstract state space to detect potential deadlocks. The static analysis is developed for a concurrent calculus with higher-order functions and dynamic lock creation. Covering lock creation by an appropriate abstraction extends our earlier work [43] for deadlock detection using behaviour abstraction. Another important extension of that work is to enhance the precision by making the analysis context-sensitive and furthermore to support effect inference ([43] in contrast required the programmer to provide the behaviour annotations manually). The analysis is shown sound, i.e., the abstraction preserves deadlocks of the program. Formally that is captured by an appropriate notion of simulation ("deadlock-sensitive simulation").

Related work. Deadlocks are a well-known problem in concurrent programming and a vast number of techniques for statically and dynamically detecting concurrency errors such as deadlocks and races have been investigated and implemented for various languages [47, 17, 26, 48, 52, 34]. One common way to prevent deadlocks is to arrange locks in a certain partial order such that no cyclic wait on locks/resources, which is one of the four necessary conditions for deadlocks [11], can occur. Cf. also [12] for an earlier comparison of different analysis techniques for deadlock detection, concentrating on model-checking for detecting deadlocks in Ada programs. For instance, Boyapati, Lee, and Rinard [9] prevent deadlocks by introducing deadlock types and imposing an order among these. The paper also covers type inference and polymorphism wrt. the lock levels. Likewise, the type inference algorithms by Suenaga [49] and Vasconcelos et al. [51] assure deadlock freedom in a well-typed program with a strict partial order on lock acquisition. Agarwal, Wang, and Stoller [4] use above deadlock types to improve the efficiency for run-time checking with a static type system, by introducing runtime checks only for those locks where the inferred deadlock type indicates potential

for deadlocks. Similar to our approach, Naik et al. [37] detect potential deadlocks with a model-checking approach by abstracting threads and locks by their allocation sites. The approach is neither sound nor complete.

Instead of preventing deadlocks caused by the attempt to acquire locks, static analyses, as well as type systems, are also studied to prevent deadlocks caused by communication over channels and message passings, where the dynamically changing communication structures complicates the analyses for preventing deadlocks [31, 32, 33, 22]. Kobayashi [33] presents a constraint-based type inference algorithm for detecting communication deadlocks in the π-calculus. In contrast to our system, he attaches abstract usage information onto channels, not processes. Cyclic dependencies there indicate potential deadlocks. Further differences are that channel-based communication does not have to consider reentrance, and the lack of functions avoids having to consider polymorphism and higher order. Giachino et al. [22] derives behavioural types with dependency informations out of programs in concurrent object language with asynchronous message passings, then translates the behavioural types into a finite state model, and verifies deadlock freedom by detecting circular dependency in the model. Further works dealing with deadlocks in concurrency models using active objects include [20, 21, 15]. Instead of checking for deadlocks, the approach by Kidd et al. [30] generates an abstraction of a program to check for data races in concurrent Java programs, by abstracting unlimited number of Java objects into a finite set of abstract ones whose locks are binary. Gordon, Ernst, and Grossman [23] introduce the concept of lock capabilities used in a type and effect system for static deadlock prevention. The lock capabilities, integrated into the type system, are flexible enough to not insist on a global lock order but allow more flexible acquisition orders, and that flexibility makes the method amenable to handle fine-grained locking disciplines. The deadlock prevention analysis is formulated as a type and effect system (as the work presented here), but does not use behavioral effects. Their contribution relies on the availability of an external must-alias analysis, whereas ours has a may-alias-like analysis built in. Their language supports method calls, but not higher-order. Locksmith, a well-known static tool for detecting concurrency errors, in particular races, for multithreaded C is described in [41, 40], and for a more recent overview of the tool and engineering aspects of the techniques employed, see [42]. Like the work here and, the techniques make use of context-sensitive flow analysis (inspired by the work of [45, 44] using contraints, but the work and tool also incorporates further analyses to increase the precision in particular when dealing with pointers and shared references.

Also *dynamic,* runtime analyses have been widely used for deadlock detec-

tion (and other concurrency errors like races), e.g. in [25, 26, 6, 7, 29]. They are typically based on monitoring at run-time patterns and orders of lock-acquisitions and other behaviour to spot (potentially) erroneous behaviour or even to take corrective action in such cases. Combinations of static and dynamic techniques have been used, for instance in [1], to reduce the overhead of the run-time monitoring. In contrast to deadlock prevention, deadlock *avoidance* means to evade a program runs into a deadlock during execution. Boudol [8] uses a type and effect system to derive statically information which helps the scheduler to navigate around potential deadlocks. Unlike this paper, which is about deadlock prevention, [8] is concerned deadlock *avoidance*. A further difference is that the lock-usage is block structured whereas here, locks can be created and used without that restriction. That work is extended by Gerakios, Papaspyrou, and Sagonas [19] to deal also with non-lexical use of locks. A different combination of static and dynamic techniques, there called type discovery, is proposed in [3, 2] and extended in [46], which investigate a run-time assisted type inference, where monitoring provides annotations assisting the type inference.

Future work. To keep the state space of model exploration in the second stage of our analysis finite, we impose a restriction on process creation. A natural extension to our work is to find sound abstractions for process creation, which seems more challenging and a naive approach by simply summarizing processes by their point of creation is certainly not enough.

The analysis in the paper focuses on concurrent programs with lock-based synchronization. It would be interesting to study whether similar approaches can be applied to other synchronization mechanisms, for instance, message passing. It is practically valuable to study how they can be adapted for object-oriented languages like Java [24]. One possible approach is to bridge the calculus in the paper to Java by translating its syntax to that of the calculus in this paper. An initial step to this could be first considering the syntax of Featherweight Java (FJ) [28].

Since the analysis is based on over-approximations of programs, a reported potential deadlock may not necessarily exist in the program at runtime. It would be useful to study different techniques to increase precision by identifying spurious deadlocks and refine the over-approximation accordingly. One technique for such a refinement is known as counterexample guided abstraction refinement (CEGAR) [10].

Acknowledgements

We are grateful to the anonymous reviewers for their very thorough reviews and for giving helpful and critical feedback.

References

[1] Agarwal, R., Bensalem, S., Farchi, E., Havelund, K., Nir-Buchbinder, Y., Stoller, S. D., Ur, S., Wang, L., 2010. Detection of deadlock potentials in multithreaded programs. IBM Journal of Research and Development 54 (5), 3:1–3:15.

[2] Agarwal, R., Sasturkar, A., Stoller, S. D., 2004. Type discovery for parameterized race-free Java. Tech. Rep. DAR-04-16, Dept. of Computer Science, SUNY, Stony Brook.

[3] Agarwal, R., Stoller, S. D., 2004. Type inference for parameterized race-free Java. In: Steffen, B., Levi, G. (Eds.), Proceedings of the 5th International Conference on Verification, Model Checking, and Abstract Interpretation, (VMCAI). Vol. 2937 of Lecture Notes in Computer Science. Springer, pp. 149–160.

[4] Agarwal, R., Wang, L., Stoller, S. D., 2006. Detecting potential deadlocks with static analysis and run-time monitoring. In: Ur, S., Bin, E., Wolfsthal, Y. (Eds.), Proceedings of the Haifa Verification Conference 2005. Vol. 3875 of Lecture Notes in Computer Science. Springer, pp. 191–207.

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

[6] Bensalem, S., Havelund, J., Havelund, K., Mounier, L., 2006. Confirmation of deadlock potentials detected by runtime analysis. In: Ur, S., Farchi, E. (Eds.), Proceedings of the 4th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. ACM, pp. 41–50.

[7] Bensalem, S., Havelund, K., 2006. Dynamic deadllock analysis of multi-threaded programs. In: Ur, S., Bin, E., Wolfsthal, Y. (Eds.), Proceedings of the Haifa Verification Conference 2005. Vol. 3875 of Lecture Notes in Computer Science. Springer, pp. 208–223.

[8] Boudol, G., 2009. A deadlock-free semantics for shared memory concurrency. In: Leucker, M., Morgan, C. C. (Eds.), Proceedings of the 6th International Colloquium on Theoretical Aspects of Computing (ICTAC). Vol. 5684 of Lecture Notes in Computer Science. Springer, pp. 140–154.

[9] Boyapati, C., Lee, R., Rinard, M., 2002. Ownership types for safe programming: Preventing data races and deadlocks. In: Proceedings of 17th ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA). ACM, pp. 211–230.

[10] Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H., 2000. Counterexample-guided abstraction refinement. In: Emerson, E. A., Sistla, A. P. (Eds.), Proceedings of the 12th International Conference on Computer-Aided Verification (CAV). Vol. 1855 of Lecture Notes in Computer Science. Springer, pp. 154–169.

[11] Coffman Jr., E. G., Elphick, M., Shoshani, A., 1971. System deadlocks. Computing Surveys 3 (2), 67–78.

[12] Corbett, J., 1996. Evaluating deadlock detection methods for concurrent software. IEEE Transactions on Software Engineering 22 (3), 161–180.

[13] Damas, L., 1985. Type assignment in programming languages. Ph.D. thesis, Laboratory for Foundations of Computer Science, University of Edinburgh.

[14] Damas, L., Milner, R., 1982. Principal type-schemes for functional programming languages. In: Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, pp. 207–212.

[15] de Boer, F. S., Bravetti, M., Grabe, I., Lee, M., Steffen, M., Zavattaro, G., 2013. A Petri net based analysis of deadlock for active objects and futures. In: Pasareanu, C. S., Salaün, G. (Eds.), Revised Selected Papers of the 9th International Workshop on Formal Aspects of Component Software (FACS 2012). Lecture Notes in Computer Science. Springer, pp. 110–127.

[16] Dijkstra, E. W., 1965. Cooperating sequential processes. Tech. Rep. EWD-123, Technological University, Eindhoven.

[17] Engler, D. R., Ashcraft, K., 2003. RacerX: Effective, static detection of race conditions and deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles. ACM, pp. 237–252.

[18] Flanagan, C., Sabry, A., Duba, B. F., Felleisen, M., 1993. The essence of compiling with continuations. In: Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, pp. 237–247.

[19] Gerakios, P., Papaspyrou, N., Sagonas, K., 2011. A type and effect system for deadlock avoidance in low-level languages. In: Proceedings of the 6th ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI). ACM, pp. 15–28.

[20] Giachino, E., Grazia, C. A., Laneve, C., Lienhardt, M., Wong, P. Y., 2013. Deadlock analysis of concurrent objects: Theory and practice. In: Johnsen, E. B., Petre, L. (Eds.), Proceedings of the 11th International Conference on Integrated Formal Methods (iFM). Vol. 7940 of Lecture Notes in Computer Science. Springer, pp. 394–411.

[21] Giachino, E., Laneve, C., 2011. Analysis of deadlocks in object groups. In: Bruni, R., Dingel, J. (Eds.), Formal Techniques for Distributed Systems (FMOODS-FORTE). Vol. 6722 of Lecture Notes in Computer Science. Springer, pp. 168–182.

[22] Giachino, E., Laneve, C., 2014. Deadlock detection in linear recursive programs. In: Bernardo, M., Damiani, F., Hähnle, R., Johnsen, E. B., Schaefer, I. (Eds.), Advanced Lectures from the 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM). Vol. 8483 of Lecture Notes in Computer Science. Springer, pp. 26–64.

[23] Gordon, S. C., Ernst, M. D., Grossman, D., 2012. Static lock capabilities for deadlock freedom. In: Proceedings of the 8th ACM SIGPLAN workshop on Types in language design and implementation (TLDI). ACM, pp. 67–78.

[24] Gosling, J., Joy, B., Steele, G. L., Bracha, G., 2000. The Java Language Specification, 2nd Edition. Addison-Wesley.

[25] Harrow, J., 2000. Runtime checking of multithreaded applications applications with Visual Threads. In: Havelund, K., Penix, J., Visser, W. (Eds.), Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. Springer, pp. 331–342.

[26] Havelund, K., 2000. Using runtime analysis to guide model checking of Java programs. In: Havelund, K., Penix, J., Visser, W. (Eds.), Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification. Springer, pp. 245–264.

[27] Hindley, J. R., 1969. The principal type-scheme of an object in combinatory logic. Transactions of the American Mathematical Society 146, 29–60.

[28] Igarashi, A., Pierce, B. C., Wadler, P., 1999. Featherweight Java: A minimal core calculus for Java and GJ. In: Proceedings of 14th ACM SIGPLAN Conference on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA). ACM, pp. 132–146.

[29] Joshi, P., Naik, M., Sen, K., Gay, D., 2010. An effective dynamic analysis for detecting generalized deadlocks. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE). ACM, pp. 327–336.

[30] Kidd, N., Reps, T. W., Dolby, J., Vaziri, M., 2011. Finding concurrency-related bugs using random isolation. Software Tools for Technology Transfer (STTT) 13 (6), 495–518.

[31] Kobayashi, N., 1998. A partially deadlock-free typed process calculus. ACM Transactions on Programming Languages and Systems 20 (2), 436–482.

[32] Kobayashi, N., 2005. Type-based information flow analysis for the π-calculus. Acta Informatica 42 (4-5), 291–347.

[33] Kobayashi, N., 2006. A new type system for deadlock-free processes. In: Baier, C., Hermanns, H. (Eds.), Procceedings of the 17th International Conference on Concurrency Theory (CONCUR). Vol. 4137 of Lecture Notes in Computer Science. Springer, pp. 233–247.

[34] Leino, K. R. M., Müller, P., 2009. A basis for verifying multi-threaded programs. In: Castagna, G. (Ed.), Proceedings of the 18th European Symposium on Programming Languages and Systems (ESOP). Vol. 5502 of Lecture Notes in Computer Science. Springer, pp. 378–393.

[35] Milner, R., 1978. A theory of type polymorphism in programming. Journal of Computer and System Sciences 17 (3), 348–375.

[36] Mossin, C., 1997. Flow analysis of typed higher-order programs. Ph.D. thesis, DIKU, University of Copenhagen, technical Report DIKU-TR-97/1.

[37] Naik, M., Park, C.-S., Sen, K., Gay, D., 2009. Effective static deadlock detection. In: Proceedings of the 31st International Conference on Software Engineering (ICSE). IEEE, pp. 386–396.

[38] Nielson, H. R., Nielson, F., 1999. Type and effect systems. In: Olderog, E.-R., Steffen, B. (Eds.), Correct System Design – Recent Insights and Advances. Vol. 1710 of Lecture Notes in Computer Science. Springer, pp. 114–136.

[39] Nielson, H. R., Nielson, F., Amtoft, T., 1997. Polymorphic subtyping for effect analysis: The static semantics. In: Dam, M. (Ed.), Proceedings of the 5th LOMAPS Workshop. Vol. 1192 of Lecture Notes in Computer Science. Springer, pp. 141–171.

[40] Pratikakis, P., 2008. Sound, precise, and efficient static races detection for multi-threaded programs. Ph.D. thesis, University of Maryland.

[41] Pratikakis, P., Foster, J. S., Hicks, M. W., 2006. LOCKSMITH: Context-sensitive correlation analysis for race detection. In: Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI). ACM, pp. 320–331.

[42] Pratikakis, P., Foster, J. S., Hicks, M. W., 2011. LOCKSMITH: Practical static race detection for C. ACM Transactions on Programming Languages and Systems 33 (1), 3:1–3:55.

[43] Pun, K. I., Steffen, M., Stolz, V., 2012. Deadlock checking by a behavioral effect system for lock handling. Journal of Logic and Algebraic Programming 81 (3), 331–354.

[44] Rehof, J., Fähndrich, M., 2001. Type-based flow analysis: From polymorphic subtyping to CFL-reachability. In: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, pp. 54–66.

[45] Reps, T., Horwitz, S., Sagiv, M., 1995. Precise interprocedural data-flow analysis via graph reachability. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL). ACM, pp. 49–61.

[46] Rose, J., Swamy, N., Hicks, M., 2005. Dynamic inference of polymorphic lock types. Science of Computer Programming 58 (3), 366–383.

[47] Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T., 1997. Eraser: A dynamic data race detector for multithreaded programs. ACM Transactions on Computer Systems 15 (4), 391–411.

[48] Sterling, N., 1993. Warlock: A static race analysis tool. In: USENIX Winter Technical Conference. USENIX Association, pp. 97–106.

[49] Suenaga, K., 2008. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In: Ramalingam, G. (Ed.), Proceedings of the 6th Asian Symposium Programming Languages and Systems (APLAS). Vol. 5356 of Lecture Notes in Computer Science. Springer, pp. 155–170.

[50] Talpin, J.-P., Jouvelot, P., 1992. Polymorphic type, region and effect inference. Journal of Functional Programming 2 (3), 245–271.

[51] Vasconcelos, V., Martins, F., Cogumbreiro, T., 2009. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In: Beresford,

A. R., Gay, S. J. (Eds.), Proceedings of the 2nd International Workshop on Programming Language Approaches to Concurrency and Communication-cEntric Software (PLACES). Vol. 17 of Electronic Proceedings in Theoretical Computer Science (EPTCS). pp. 95–109.

[52] Vojdani, V., Vene, V., 2007. Goblint: Path-sensitive data race analysis. In: Proceedings of the 10th Symposium on Programming Languages and Software Tools. Eötvös Lorand Univ., pp. 130–141.