UNIVERSITY OF OSLO
Department of Informatics

Lock-Polymorphic Behaviour Inference for Deadlock Checking

September 2013

Research Report No. 436

Ka I Pun, Martin Steffen, and Volker Stolz

ISBN 82-7368-398-2
ISSN 0806-3036

September 2013

Lock-Polymorphic Behaviour Inference for Deadlock Checking

Ka I Pun, Martin Steffen, and Volker Stolz

University of Oslo, Department of Informatics

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. The analysis is context-sensitive and locks are summarized based on their creation-site. 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 behavior of a program, in particular that deadlocks in the program are preserved in the effects.

1 Introduction

Deadlocks are a common problem for concurrent programs with shared resources. According to the classic characterization from [4], a deadlocked state is marked by a number of processes forming a cycle where each process, unwilling to release its own resource, is waiting on the resource held by its neighbor. The inherent non-determinism make deadlocks, as other errors in the presence of concurrency, hard to detect and to reproduce. We present a static analysis using behavioral 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 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 approach presented in this paper works in two stages: in a first stage, an effect-type system uses a static behavioral abstraction of the codes' behavior, concentrating on the lock interactions. To analyze the consequences on the global level, in particular for detecting deadlocks, the combined individual abstract thread behaviors are explored in the second stage.

Two challenges need to be tackled to make such a framework applicable in practice. For the first stage on the thread local level, the static analysis must be able to *derive* the abstract behavior, 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 behavior needs to over-approximate the concrete one which means, concrete and abstract description 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, 1) allowing recursion, supports 2) dynamic thread creation, 3) dynamic lock creation, and 4) with re-entrant locks, the lock counters are unbounded. Our approach offers sound abstractions for the mentioned sources of unboundedness, except that we do not have an abstraction usable for deadlock detection in the presence of dynamic thread creation.

We shortly present in a non-technical manner the ideas behind the abstraction.

1.1 Effect inference on the thread local level

As mentioned, in the first stage of the analysis, the analysis uses a behavioral type and effect system to over-approximate the lock-interactions of a single thread. To force the user to annotate the program with the expected behavior in the form of effects is impractical, so the type and especially the behavior should be inferred automatically. Effect inference, including inferring behavioral effects, has been studied earlier and applied to various settings, including obtaining static over-approximations of behavior for concurrent languages by Amtoft, Nielson and Nielson [2]. We apply effect inference to deadlock detection and as is standard (cf. e.g. [12,17,2]), the inference system is constraint-based, where the constraints in particular express an approximate order between behaviors.

Besides being able to infer the behavior, it is important that the static approximation is as precise as possible. Since our calculus supports higher-order functions, it is thus 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 [6,5] and this has been used for effect-inference in various settings, e.g., in the works mentioned above.

Deadlock checking in our earlier work [14] was not polymorphic (and we did not address effect inference). The extension in this paper leads to an increase in precision wrt. checking for deadlocks, as illustrated by the small example below, where the two lock creation statements are labeled by π_1 and π_2:

```
let l1 = new^π1 L in let l2 = new^π2 L in
let f = fn x:L . ( x.lock; x.lock )
in spawn(f(l1)); f(l2)
```

The main thread, after creating two locks and defining function f, spawns a thread, and afterward, 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 [14] determines the potential origin of locks by data-flow analysis. When analyzing 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, $\mathrm{L}^{\{\pi_1,\pi_2\}}$, 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" [7] 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 analyzes the example as deadlock-free.

1.2 Deadlock preserving abstractions on the global level

Lock abstraction For dynamic data allocation, a standard abstraction is to *summarize* all data allocated at a given program point into one abstract representation. In the presence of loops or recursion, the abstracting function mapping concrete locks to their abstract representation necessarily is non-injective. For concrete, ordinary programs it is clear that identifying locks may change the behavior of the program. What makes identification of locks in general tricky, and here in particular connection with deadlocks, is that, on the one hand. it leads to *less* steps, in that lock-protected critical sections may become larger, and on the other hand to *more* steps at the same time, in that deadlocks may disappear when identifying locks. That this form of summarizing lock abstraction is problematic when analyzing properties of concurrent programs has been observed elsewhere as well, cf. e.g. Kidd et al. in [10].

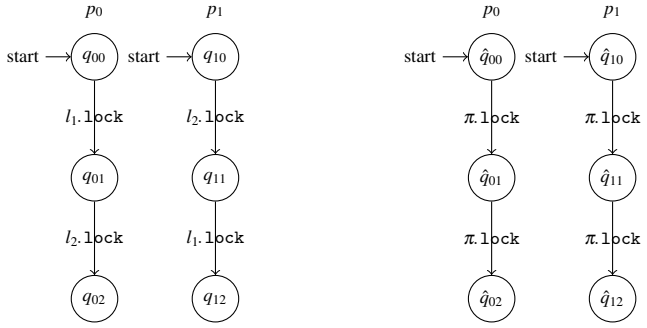
For a sound abstraction for deadlock detection when identifying locks in the described way, one faces thus the following dilemma: a) the abstract level, using the abstract locks, need to show at least the behavior of the concrete level, i.e., we expect they are related by a form of simulation. On the other hand, to preserve not only the possibility of doing steps, but also *deadlocks,* the opposite must hold sometimes: a) 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 e.g., those freshly created, which may be taken.

Consequently, abstract locks loose their "mutual exclusion" capacity: where a concrete heap is a mapping which associates to each lock references the number of times *at most* one process is holding it, an abstract heap $\hat{\sigma}$ then records how many times an abstract lock π is held by the various processes, e.g. three times by one process and two times by another. The corresponding natural number of the abstractly represent 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 basically has to capture all possibilities of waiting on one of the corresponding concrete locks (see Definition 8 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 behavior be preserved as in standard simulation, also the possibility of waiting on a lock and ultimately possibility of deadlock needs to be preserved. For this we introduce the notion of *deadlock sensitive* simulation (see Definition 11). The definition is analogous to the

one from [14]. However, it takes into account now that the analysis is polymorphic and the definition is no longer based on an direct operational interpretation of the behavior of the effects. Instead it is based on the behavioral 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 behavior, 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 abstraction, summarizing concrete locks into an abstract one, improves on our earlier analysis [14], which could therefore deal only with a static number of locks.



(a) Concrete (b) Abstract

Fig. 1: Lock abstraction

Counter abstraction and further behavior 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 behavior represent abstractly the *stack* of function calls for that thread. With sequential composition as construct for abstract behavioral effects allows to represent non-tail-recursive behavior, corresponding to the context-free call-and-return behavior of the underlying program. To curb that source of infinity, we allow to replace the behavior 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 arbitrary, chaotic behavior. 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.

To summarize, compared to [14], the paper makes the following contributions: 1) the effect analysis is generalized to a context-sensitive formulation, using constraint, 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 behavioral type system is presented in Section 3, which also includes the soundness result in the form of subject reduction. The conclusion in Section 4 discusses related and future work. Additional technical material, lemmas and proofs can be found in the appendix (Appendix A).

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 as threads t and expressions e, where t is either a value v or a sequential composition written as $\texttt{let}\ x{:}T = e\ \texttt{in}\ t$, where the let-construct binds the local variable x in t. Expressions include function application, conditionals, and thread creation. 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.

$P ::= \emptyset \mid p\langle t\rangle \mid P \,\|\, P$		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.
$v ::= x \mid l \mid \texttt{true} \mid \texttt{false} \mid \texttt{fn}\ x{:}T.t \mid \texttt{fun}\ f{:}T.x{:}T.t$		values

Table 1: Abstract syntax

Semantics

The small-step operational semantics, presented next, distinguishes between local and global steps (cf. Table 2 and 3). 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 $e_1;(t_1;t_2)$. Together with the rest of the rules, which perform a case distinction on the first basic expression in a let construct, that assures a deterministic left-to-right evaluation within each thread. The two R-IF-rules cover the two branches of the conditional and the R-APP-rules deals with function application, of non-recursive, resp., recursive functions.

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

Global configurations are of the form $\sigma \vdash P$ where P is a program and the heap σ is a finite mapping from lock identifiers 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. For the analysis later, we allow ourselves also to write $\sigma(l,p) = n+1$ if $\sigma(l) = p(n+1)$ and $\sigma(l,p) = 0$ otherwise.

The global steps are given as transitions between global configurations. It will be handy later to assume the transitions appropriately labeled (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).

The notion of (resource) deadlock used in the analysis is rather standard, where a number of processes waiting for each other's locks in a cyclic manner constitute a deadlock (see also [14]). In our setting with re-entrant locks, a process cannot deadlock "on itself".

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

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

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

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

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

Table 3: Global steps

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

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

3 Type system

Next we present an effect type system to derive behavioral information which is 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 behavior. The representation extends our earlier system [14] by making the analysis *context-sensitive* and furthermore by supporting effect type *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 behavior inference, see for instance Amtoft, Nielson, and Nielson [2] and type-based flow analysis, see e.g. Mossin [12]. Unlike the presentation in [14], 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 [6,5,9,11], 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 Table 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 is a corresponding variable. Types includes basic types 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 are the flow information about the potential places where the lock is created. This information will be reconstructed, and the user uses types without annotations (the "underlying" types) in the program. We write T as meta-variables for the underlying types, and $\hat{T}$ and its syntactic variants for the annotated types, as given in the grammar.

Whereas the type of an expression captures the results of the computations of the expression if it terminates, the effect captures the *behavior* 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 effect is denoted by ε, representing behavior without lock operations. Sequential composition is represented by $\varphi_1;\varphi_2$. The choice between two effects $\varphi_1+\varphi_2$ and recursive effects $recX.\varphi$ are 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

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

Table 4: Types and type schemes

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

Table 5: Effects

also, that recursion is not polymorphic. Labels a capture the three basic effects: spawning a new process with behavior φ is represented by $\texttt{spawn}\,\varphi$. The basic effects $r.\texttt{lock}$ and $r.\texttt{unlock}$ respectively capture lock manipulations, acquiring and releasing a lock, where r is referring to the possible point of creation. Silent transitions are represented by τ. Note that lock-creation has not effect and will be represented by a τ-transition.

Constraints C finally are finite sets of in-equations of the form $\rho \sqsupseteq r$ or of $X \sqsupseteq \varphi$, where ρ is, as mentioned, a flow variable and X an effect or behavior variable. 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. The type and effect system presented in this paper uses a constraint-based flow analysis as proposed by Mossin [12] for lock information. Likewise, the effects captured as a sequence of behavior is formulated using constraints.

3.2 Type inference

Next we present a type inference algorithm which derives types and effects and generating corresponding constraints (see Table 7 below). It is formulated in a rule-based manner, with judgments of the following form

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

meaning that under the context Γ and the constraints C, expression e has type $\hat{T}$ and effect φ; The system will be 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 the grammar of Table 1. In contrast, e contains no flow or effect annotations; those are derived by the algorithmic type system. It would be straightforward to omit the underlying types and have them reconstructed as well, using standard type inference à la Hindley/Milner/Damas [6,5,9]. For simplicity, we concentrate on the type annotations and the effect part.

The intended meaning of the judgment from (1) is that, relative to a given typing context Γ and for the given 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 behavioral 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. Solutions to a constraint set C are ground substitutions θ, assigning label sets to flow variables ρ and effect variables. We write $\theta \models C$ if θ is a solution to C.

Ultimately, one is interested in the minimal solution of the constraints, as it provides the most precise flow and effect 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 "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 7, 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 3). As an invariant, the type system makes sure that lock types are always of the form L^{ρ}, i.e., using flow *variables* and similarly that only variables X are used for the latent effects for function types.

Definition 3 (Constraint generation). *The judgment $\hat{T}_1 \leq \hat{T}_2 \vdash C$ (read as "requiring $\hat{T}_1 \leq \hat{T}_2$ generates the constraints C") is inductively given as follows:*

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

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

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 $\hat{T}_1 \vee \hat{T}_2 \vdash \hat{T}; C$, 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 (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 6. The operation $\wedge$ is defined dually.*

The rules for the type and effect system then are given in Table 7. 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)

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

$$\frac{\hat{T}_1' \wedge \hat{T}_1'' = \hat{T};C_1 \qquad \hat{T}_2' \vee \hat{T}_2'' = \hat{T}';C_2 \qquad X_1 \sqcup X_2 = X;C_3}{\hat{T}_1' \xrightarrow{X_1} \hat{T}_2' \vee \hat{T}_1'' \xrightarrow{X_2} \hat{T}_2'' = \hat{T}_1 \xrightarrow{X} \hat{T}_2;C_1,C_2,C_3}\ \text{LT-ARROW} \qquad \frac{X\ \mathit{fresh} \qquad C = \{\varphi_1 \sqsubseteq X, \varphi_2 \sqsubseteq X\}}{\varphi_1 \sqcup \varphi_2 = X;C}\ \text{LE-EFF}$$

Table 6: Least upper bound

is of the type L^{ρ}, has an empty effect and the 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 evaluated and therefore have no effect. As usual, the type of the argument need to be a subtype of the input type of the function, and corresponding constraints C_3 are generated by $\hat{T}_2' \leq \hat{T}_2 \vdash C_3$. 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 6 from Definition 4). The `let`-construct (cf. TA-LET) for the sequential composition has as 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 first approximation, given $\hat{T}_1$ of e_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 [2]). The close-operation $\mathit{close}(\Gamma,\varphi,C,\hat{T})$ first computes the set of all "relevant" free variables in a type $\hat{T}$ and the constraint C by the operation $\mathit{close}_{\uparrow\downarrow}(\hat{T}_1,C_1)$ (cf. Definition 5(2)). Among the set of free variables, those are free in the context or in the effect and the corresponding downward closure (cf. Definition 5(1)) are non-generalizable and are excluded.

Definition 5 (Closure).

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

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

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

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

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

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

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

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

Table 7: Algorithmic effect inference

1. *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 $fv(\varphi) \subseteq \vec{Y}$ 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.*
2. *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.*

Definition 6 (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 $p_1\langle\varphi_1;C_1\rangle \parallel \ldots \parallel p_n\langle\varphi_n;C_n\rangle$.

3.3 Semantics of the behavior

Next we are going to define the transition relation on the abstract behavior by using 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 [2]. The a is one of the labels from Table 5 and do not include τ-labels. The relation $C \vdash \varphi_1 \sqsubseteq \varphi_2$ is defined in Table 8.

Definition 7. *The transition relation between configurations of the form $C;\hat{\sigma} \vdash \Phi$ is given inductively by the rules of Table 9, 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 labeled with one of the labels in Table 5, and correspondingly capture the three possible steps we describe in the behavior, namely creating a new process with a given behavior, locking and unlocking. Analogous to the corresponding case in the concrete semantics, rule RE-SPAWN covers the spawning 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

$\varepsilon;\varphi \equiv \varphi$ EE-UNIT $\qquad \varphi_1;(\varphi_2;\varphi_3) \equiv (\varphi_1;\varphi_2);\varphi_3$ EE-ASSOC

$C,\varphi \sqsubseteq X \vdash \varphi \sqsubseteq X$ S-AX$_E$ $\qquad \dfrac{\varphi_1 \equiv \varphi_2}{C \vdash \varphi_1 \sqsubseteq \varphi_2}$ S-REFL$_E$ $\qquad C, r \subseteq \rho \vdash r \subseteq \rho$ S-AX$_L$ $\qquad C \vdash r \subseteq r$ S-REFL$_L$

$\dfrac{C \vdash r_1 \subseteq r_2 \qquad C \vdash r_2 \subseteq r_3}{C \vdash r_1 \subseteq r_3}$ S-TRANS$_L$ $\qquad \dfrac{C \vdash \varphi_1 \sqsubseteq \varphi_2 \qquad C \vdash \varphi_2 \sqsubseteq \varphi_3}{C \vdash \varphi_1 \sqsubseteq \varphi_3}$ S-TRANS$_E$ $\qquad \dfrac{C \vdash \varphi_1 \sqsubseteq \varphi_1' \qquad C \vdash \varphi_2 \sqsubseteq \varphi_2'}{C \vdash \varphi_1;\varphi_2 \sqsubseteq \varphi_1';\varphi_2'}$ S-SEQ

$\dfrac{C \vdash \varphi_1 \sqsubseteq \varphi_2}{C \vdash \mathtt{spawn}\ \varphi_1 \sqsubseteq \mathtt{spawn}\ \varphi_2}$ S-SPAWN $\qquad \dfrac{C \vdash r_1 \subseteq r_2}{C \vdash r_1.\mathtt{lock} \sqsubseteq r_2.\mathtt{lock}}$ S-LOCK $\qquad \dfrac{C \vdash r_1 \subseteq r_2}{C \vdash r_1.\mathtt{unlock} \sqsubseteq r_2.\mathtt{unlock}}$ S-UNLOCK

Table 8: Orders on behaviors

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

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

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

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

Table 9: Global transitions

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 behavior of concrete locks. The ensure that the abstraction preserves deadlocks requires to adapt the definition of what it means that a (abstract) behavior waits on a lock (cf. Definition 1 for concrete programs and heaps).

Definition 8 (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 $waits_{\sqsubseteq}(C;\hat{\sigma} \vdash \Phi, p, \rho)$, if $C \vdash \varphi \xrightarrow{\rho.\texttt{lock}}_{\sqsubseteq} \varphi'$ but $\hat{\sigma}(\rho,q) \geq 1$ for some $q \neq p$.*

Definition 9 (Deadlock). *A configuration $C;\hat{\sigma} \vdash \Phi$* is deadlocked *if $\hat{\sigma}(\rho_i,p_i) \geq 1$ and furthermore $waits(C;\hat{\sigma} \vdash \Phi, p_i, \rho_{i+_k 1})$ (where $k \geq 2$ and for all $0 \leq i \leq k-1$). The $+_k$ is meant as addition modulo k. A configuration $C;\hat{\sigma} \vdash \Phi$* contains a deadlock*, if, starting from $C;\hat{\sigma} \vdash \Phi$, a deadlocked configuration is reachable; otherwise the configuration is* deadlock free.

3.4 Soundness

A crucial part for proving the soundness of the algorithm wrt. the semantics is preservation of well-typedness under reduction. This includes to check that the operational interpretation of the program is over-approximated by the effect given by the type system. This proof of over-approximation means establishing a simulation relation between a program and its effect, where 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 10 (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 11 (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 $C_1;\hat{\sigma}_1 \vdash \Phi_1 \stackrel{p\langle a\rangle}{\Longrightarrow}_\sqsubseteq C_1;\hat{\sigma}'_1 \vdash \Phi'_1$, then $C_2;\hat{\sigma}_2 \vdash \Phi_2 \stackrel{p\langle a\rangle}{\Longrightarrow}_\sqsubseteq C_2;\hat{\sigma}'_2 \vdash \Phi'_2$ 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 $waits_\sqsubseteq((C_2;\hat{\sigma}_2 \vdash \Phi_2), p, \theta(\rho))$.*

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 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.

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

Fig. 2: Deadlock sensitive simulation $\lesssim^D_\sqsubseteq$

The notation $\stackrel{p\langle a\rangle}{\Longrightarrow}$ is a weak transition relation which is 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 $C;\hat{\sigma}_1 \vdash \Phi_1 \lesssim^D_\sqsubseteq C;\hat{\sigma}_2 \vdash \Phi_2$, the property of deadlock freedom is straightforwardly carried from the more abstract behavior to the concrete one (cf. Lemma 1).

Lemma 1 (Preservation of deadlock freedom). *Assume $C;\hat{\sigma}_1 \vdash \Phi_1 \lesssim^D_\sqsubseteq C;\hat{\sigma}_2 \vdash \Phi_2$. If $C;\hat{\sigma}_2 \vdash \Phi_2$ is deadlock free, then so is $C;\hat{\sigma}_1 \vdash \Phi_1$.*

The next lemma shows the compositionality of the deadlock simulation relation wrt. parallel composition.

Lemma 2 (Composition). *Assume* $C;\hat{\sigma}_1 \vdash p\langle\varphi_1\rangle \lesssim^D_{\sqsubseteq} C;\hat{\sigma}_2 \vdash p\langle\varphi_2\rangle$*, then* $C;\hat{\sigma}_1 \vdash \Phi \parallel p\langle\varphi_1\rangle \lesssim^D_{\sqsubseteq} C;\hat{\sigma}_2 \vdash \Phi \parallel p\langle\varphi_2\rangle$.

To show the soundness of algorithmic type and effect inference, the proof is formulated as a form of 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. As for the effects, the type system derives the formal behavioral description, i.e. over-approximation, for the future behavior of a program; one hence cannot expect to the effect is preserved for each reduction step. Furthermore, many steps which are irrelevant to the behavior of lock manipulations, i.e., internal steps, are ignored. Thus, we relate the behavior of the program and the behavior of the effects via a deadlock-sensitive simulation relation.

Lemma 3 (Subject reduction). *Let* $\Gamma \vdash p\langle t\rangle :: p\langle\varphi;C\rangle$*, and* $\sigma_1 \equiv_{\hat{\theta}} \hat{\sigma}_2$*. Assume furthermore* $\theta \models C$.

1. $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1' \vdash p\langle t'\rangle$*, then* $\Gamma \vdash p\langle t'\rangle :: p\langle\varphi';C'\rangle$ *with* $C \vdash \theta' C'$ *for some* θ'*, and furthermore* $C \vdash \varphi \sqsupseteq \theta'\varphi'$*, and* $\sigma_1' \equiv_{\hat{\theta}} \hat{\sigma}_2$.
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* $C;\hat{\sigma}_2 \vdash p\langle\varphi\rangle \overset{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq} C;\hat{\sigma}_2' \vdash p\langle\varphi'\rangle$, $\Gamma \vdash p\langle t'\rangle :: p\langle\varphi'';C'\rangle$ *with* $C \vdash \theta' C'$*, and furthermore* $C \vdash \varphi' \sqsupseteq \theta'\varphi''$ *and* $\sigma_1' \equiv_{\hat{\theta}} \hat{\sigma}_2'$.

   (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* $\Gamma \vdash p\langle t''\rangle :: p\langle\varphi''';C''\rangle$ *with* $C \vdash \theta'' C''$ *and* $C \vdash \varphi'' \sqsupseteq \theta''\varphi'''$*, and furthermore* $\Gamma \vdash p'\langle t'\rangle :: p'\langle\varphi'''';C'\rangle$ *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* $waits_{\sqsubseteq}(C;\hat{\sigma}_2 \vdash p\langle\varphi\rangle, p, \hat{\theta}l)$.

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

Corollary 1. *Given* $\sigma_1 \equiv_\theta \hat{\sigma}_2$ *and* $\Gamma \vdash p\langle t\rangle :: p\langle\varphi;C\rangle$*, then* $\sigma_1 \vdash p\langle t\rangle \lesssim^D_{\sqsubseteq} C;\hat{\sigma}_2 \vdash p\langle\varphi\rangle$.

4 Conclusion

We have presented a constraint-based type and effect inference algorithm for deadlock checking. It infers a behavioral description of a thread's behavior 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 [14] for deadlock detection using behavior abstraction. Another important extension of that work is to enhance the precision by making the analysis context-sensitive and furthermore to support effect inference ([14] in contrast required the programmer to provide the behavior 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 deadlock checking have been investigated and implemented for various languages. 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 [4], can occur. For instance, Boyapati, Lee, and Rinard [3] prevent deadlocks by introducing deadlock types and imposing an order among these. The paper also covers type inference and polymorphism wrt. the lock levels. Instead of preventing deadlocks statically, Agarwal, Wang, and Stoller [1] use deadlock types to improve the efficiency for run-time checking with a static type system. The type inference algorithms by Suenaga, and Vasconcelos et al. [16,18] assure deadlock freedom in a well-typed program with a strict partial order on lock acquisition. Similar to our approach, Naik et al. [13] 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 checking for deadlocks, the approach by Kidd et al. [10] 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.

Future work As mentioned in the introduction, there are four principal sources of infinity in the state-space obtained by the effect inference system. For the unboundedness of dynamic lock creation, we presented an appropriate sound abstraction. We expect that the techniques for dealing with the unboundedness of lock counters and of the call stack can be straightforwardly carried over from the non-context-sensitive setting of [14], as sketched in Section 1.2. All mentioned abstractions are compatible with our notion of deadlock-sensitive simulation in the sense that being more abstract —identifying more locks, choosing a smaller bound on the lock counters or on the allowed stack depth— leads to a larger behavior wrt. our notion of simulation. This allows an incremental approach, starting from a coarse-grained abstraction, which may be refined in case of spurious deadlocks. To find sound abstractions for process creation as the last source of infinity seems more challenging and a naive approach by simply summarizing processes by their point of creation is certainly not enough.

We have developed a prototype implementation of the state-exploration part in the monomorphic setting of [14]. We plan to adapt the implementation to the more general setting and to incorporate with an implementation of the type inference system.

A Appendix

A.1 Annotated semantics and type system

The transitions of the operational semantics shown in Table 2 on page 6 for local steps and 3 on page 7 for global steps are properly labeled for deadlock checking. While the transitions for the local ones are annotated with $p\langle\tau\rangle$, the annotated semantics for the global steps are presented in Table 10 on the following page.

The specification of our type and effect inference algorithm (cf. Table 7 on page 12) is presented in Table 11 on page 32, while the global type system is shown in Table 12 on page 32.

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

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

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

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

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

Table 10: Global steps

B Additional technical lemmas

In the following, we collect some additional lemmas, needed in particular for *subject reduction*.

B.1 Miscellaneous

Lemma 4. *Wlog:* $C_1 \vdash C_2$ *implies* $fv(C_1) \supseteq fv(C_2)$.

Proof. Straightforward.

Lemma 5. *If* $dom(\theta) \subseteq fv(C)$*, then* $ran(\theta) \subseteq fv(\theta C)$.

Proof. Straightforward. □

Lemma 6. $C_1 \vdash \theta C_2$ *and* $dom(\theta) \subseteq fv(C_2)$*, then* $ran(\theta) \subseteq fv(C_1)$.

Proof. A direct consequence of Lemmas 4 and 5. □

Note that $fv(\theta C_2) \subseteq C_2$ does not hold.

Lemma 7. *Assume* $C_1 \vdash \theta C_2$ *and* $\vec{Y} \notin fv(C_1)$ *and* $dom(\theta) \subseteq \vec{Y}$*. Then*

1. $dom(\theta) \cap fv(C_1) = \emptyset$.
2. $dom(\theta) \cap ran(\theta) = \emptyset$.
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 $C_1 \vdash \theta C_2$ gives with Lemma 4 that

$$fv(C_1) \supseteq fv(\theta C_2) \; . \tag{2}$$

Wlog. we can assume that $dom(\theta) \subseteq C_2$ and thus Lemma 5 gives $ran(\theta) \subseteq fv(\theta C_2)$. Together with equation (2), 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 (2) 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. □

B.2 Subject reduction

Next we prove subject reduction, connecting the type and effect system with the operational semantics. The proof of subject reduction is best not done using the algorithmic formulation, i.e., on the derivation rules from Table 7. On the other hand, also the original specification from Section A.1 is problematic for performing the proof. The reason for that lies in the presence of non-syntax-directed rules, of which there are three: subtyping/sub-effeting on the one hand and generalization and instantiation on the other. The rules of 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 fix 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 judgement. To avoid that complication, subject reduction is done for a more disciplined system which works without those three mentioned rules.

The compromise system can be considered as a variant of the specification, where the use of the mentioned non-syntax-directed rules is restricted to specific points in a deriviation; 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 an 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.

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, 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, in the following sense.

Definition 12 (Upward closure). *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.*

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 13 (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 14 (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. *$C, C_1 \vdash \theta\hat{T}_2 \leq \hat{T}_1$*
3. *No y in $\vec{Y_1}$ is free in $\forall\vec{Y_2}{:}C_2.\hat{T}_2$.*

For generalization and instantiation, we need the following definition (cf. [2]).

Definition 15 (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 such that it solves it.*

The normalized system of the type system in Table 11 on page 32 is defined in Table 13 on page 33.

The following lemmas (about occurrence of free variables in connection with subeffecting and upward closure) will be needed in the proof of subject reduction.

Lemma 8. *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 9. *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 10. *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 8 and 9. □

Lemma 11. *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 12. *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 13. *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 $dom(\theta_2) \cap fv(C)$, 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 12 $C \vdash \theta_2\theta_1 C_1$, as required. □

Lemma 14 (Substitution). *If $C; \Gamma, x{:}\hat{S}_1 \vdash t : \hat{S}_2 :: \varphi$ and $C; \Gamma \vdash v : \hat{S}_1$, then $C; \Gamma \vdash t[v/x] : \hat{S}_2 :: \varphi$.*
Proof. □

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 16 below). The next lemma is helpful for the substitution lemma in the crucial case for variables.

Lemma 15. *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 16 (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 15. □

The following substitution Lemma 17 resp., the Corollary 2 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 17 (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 2 (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 17 plus soundness and completeness of the normalized system. □

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

Lemma 18. *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 14 of generic instance, choosing $C_1 = C_2$ and $\theta = id$. □

The following lemma expressed that typing is preserved when a typing assumption in the context is "strengthened" by using larger type wrt. the $\lesssim^g$-order (thereby weakening the judgment). The lemma is formulated for the specification of the type system. Corollary 3, the formulation as needed in the proof of subject reduction, is an easy consequence.

Lemma 19 (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 3 (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 19. □

Remark 1 (Subject reduction). This formulation of subject reduction differs from earlier formulation, in particular the one from the article [14] which pursued the same methodology but without tackling *inference* and without *context sensitivity.*

A general difference between the old setting and the new setting is that effects can now contain *variables,* whereas in the earlier setting, all effects where *concrete.* As a consequence, effects (with variables) have no direct *operational* interpretation in the form of a standard SOS semantics. Instead, the subeffect relation $\sqsubseteq$ (which is defined on effects containing variables and relative to constraints) is used as basis for the behavior of effects (leading to $\stackrel{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq}$). Attempts to simply reuse the old subject reduction proof an transfer it to the new definition failed. Therefore it seemed necessary to re-do subject reduction, now based on the new $\stackrel{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq}$-relation. ...

In [14], we could prove $\varphi = \varphi'$ since we had subsumption. □

Lemma 20 (Subject reduction (effects)). *Let $\Gamma \vdash p\langle t\rangle :: p\langle\varphi;C\rangle$, and $\sigma_1 \equiv_{\hat{\theta}} \sigma_2$. Assume furthermore $\theta \models C$.*

1. *$\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1' \vdash p\langle t'\rangle$, then $\Gamma \vdash p\langle t'\rangle :: p\langle\varphi';C\rangle$ with $C \vdash \varphi \sqsupseteq \varphi'$, and $\sigma_1' \equiv_{\hat{\theta}} \sigma_2$.*
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 $C;\sigma_2 \vdash p\langle\varphi\rangle \stackrel{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq} C;\sigma_2' \vdash p\langle\varphi'\rangle$, $\Gamma \vdash p\langle t'\rangle :: p\langle\varphi'';C\rangle$, and furthermore $C \vdash \varphi' \equiv \varphi''$ and $\sigma_1' \equiv_{\hat{\theta}} \sigma_2'$.*

   (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 $C;\sigma_2 \vdash p\langle\varphi\rangle \stackrel{p\langle a\rangle}{\Longrightarrow}_{\sqsubseteq} C;\sigma_2 \vdash p\langle\varphi''\rangle \parallel p'\langle\varphi'\rangle$ and such that $\Gamma \vdash p\langle t''\rangle :: p\langle\varphi''';C\rangle$ where $C \vdash \varphi'' \equiv \varphi'''$, and $\Gamma \vdash p'\langle t'\rangle :: p'\langle\varphi';C\rangle$.
3. *If waits$(\sigma_1 \vdash p\langle t\rangle, p, l^\rho)$, then waits$_\sqsubseteq(\sigma_2 \vdash p\langle\varphi\rangle, p, \rho)$.*

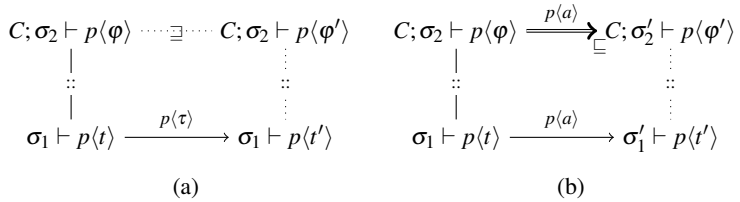


Fig. 3: Subject reduction

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

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

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$_2$ from Table 13 on page 33). By preservation of typing under substitution (Lemma 16) we get from the last premise from above that $C_2;\Gamma \vdash t[v/x] : \theta\hat{T}_2 :: \varphi_2$, and thus

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

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.

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

$$\dfrac{\dfrac{\dfrac{C_1,C_2,C;\Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1 \quad C_2,C;\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1 \vdash t_1 : \hat{T}_2 :: \varphi_2}{C_2,C;\Gamma \vdash \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1 : \hat{T}_2 :: \varphi_1;\varphi_2} \qquad C;\Gamma,x_2{:}\forall\vec{Y}_2{:}C_2.\hat{T}_2 \vdash t_2 : \hat{T}_3 :: \varphi_3}{C;\Gamma \vdash \texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2 : \hat{T}_3 :: (\varphi_1;\varphi_2);\varphi_3}}{\Gamma \vdash p\langle \texttt{let}\ x_2{:}T_2 = (\texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ t_1)\ \texttt{in}\ t_2\rangle :: p\langle(\varphi_1;\varphi_2);\varphi_3;C\rangle} \tag{3}$$

We have further that

$$\vec{Y}_1 \notin fv(C_2,C,\Gamma,\varphi_1) \quad dom(\theta_1) \subseteq \vec{Y}_1 \quad C_2,C \vdash \theta_1 C_1 \tag{4}$$
$$\vec{Y}_2 \notin fv(C,\Gamma,\varphi_1;\varphi_2) \quad dom(\theta_2) \subseteq \vec{Y}_2 \quad C \vdash \theta_2 C_2 \tag{5}$$

Consider now the following derivation tree (using two times T-LET and T-THREAD) and the additional premises from equations (7) and (8).

$$\dfrac{\dfrac{C_1,C;\Gamma \vdash e_1 : \hat{T}_1 :: \varphi_1 \qquad \dfrac{C_2,C;\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1 \vdash t_1 : \hat{T}_2 :: \varphi_2 \quad C;\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1,x_2{:}\forall\vec{Y}_2{:}C_2.\hat{T}_2 \vdash t_2 : \hat{T}_3 :: \varphi_3}{C;\Gamma,x{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1 \vdash \texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2 : \hat{T}_3 :: \varphi_2;\varphi_3}}{C;\Gamma \vdash \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2) : \hat{T}_3 :: \varphi_1;(\varphi_2;\varphi_3)}}{\Gamma \vdash p\langle \texttt{let}\ x_1{:}T_1 = e_1\ \texttt{in}\ (\texttt{let}\ x_2{:}T_2 = t_1\ \texttt{in}\ t_2)\rangle :: p\langle\varphi_1;(\varphi_2;\varphi_3);C\rangle} \tag{6}$$

$$\vec{Y}_1 \notin fv(C,\Gamma,\varphi_1) \quad dom(\theta_1') \subseteq \vec{Y}_1 \quad C \vdash \theta_1' C_1 \tag{7}$$
$$\vec{Y}_2 \notin fv(C,\Gamma,x_1{:}\forall\vec{Y}_1{:}C_1.\hat{T}_1,\varphi_2) \quad dom(\theta_2') \subseteq \vec{Y}_2 \quad C \vdash \theta_2' C_2 \tag{8}$$

They can be justified as follows.

The first condition of (7) directly follows from the corresponding one from (4). The above conditions (4) imply with Lemma 7(1), $dom(\theta_1) \cap fv(C_2, C) = \emptyset$, i.e., in particular $dom(\theta_1) \cap fv(C) = \emptyset$. Thus, with Lemma 13, the rightmost conditions from (4) and from (5) imply $C \vdash \theta_1' C_1$. As for equation (8): The first condition follow from the first one of (5) wlog., the remaining two are unchanged from (5).

The leftmost $C_1, C_2, C; \Gamma \vdash\ e_1 : \hat{T}_1 :: \varphi_1$ leaf in the tree from (3) can be strengthened to the

$$C_1, C; \Gamma \vdash\ e_1 : \theta_2 \hat{T}_1 :: \varphi_1 \tag{9}$$

using Lemma 15. Note that the conditions $C_1, C \vdash \theta_2 C_2$ and $dom(\theta_2) \cap fv(C, C_1, \Gamma, \hat{T}, \varphi_1) = \emptyset$ hold as well (the $dom(\theta_2) \cap fv(C_1, \hat{T}_1)$ holds wlog.). Thus, the judgment of equation (9) corresponds to the left-most subgoal of the tree in (6) The second sub-goal of (6) is directly covered by the second subgoal of (3). The third sub-goal of (6) follows from the third one of (3) by weakening wrt. 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$_1$: $p\langle \texttt{let}\ x{:}T = \texttt{if true then}\ e_1\ \texttt{else}\ e_2\ \texttt{in}\ t\rangle \xrightarrow{p\langle\tau\rangle} p\langle \texttt{let}\ x{:}T = e_1\ \texttt{in}\ t\rangle$
From the well-typedness assumption and inverting rules T-THREAD, T-LET, and T-IF$_1$, we get:

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

where $\vec{Y}' \notin fv(C, \Gamma, \varphi'), C \vdash \theta C'$ and $dom(\theta) \subseteq \vec{Y}'$ for some θ. Lemma 18 gives $C \vdash \forall \vec{Y}'{:}C'.\hat{T}' \lesssim^g \forall \vec{Y}'{:}C'.\hat{T}_1$, and further by Corollary 3, the right-most subgoal can be weakened to $C; \Gamma, x{:}\forall \vec{Y}'{:}C'.\hat{T}_1 \vdash t : \hat{T}'' :: \varphi''$ 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

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

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 wrt. to C'. Both together with $C, C' \vdash \varphi' \sqsupseteq \varphi_1$ give by Lemma 10 that $\vec{Y}' \cap fv(\varphi_1) = \emptyset$. Thus, by Lemma 11 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$_1$: $p\langle \texttt{let}\ x_2{:}T_2 = (\texttt{fn}\ x_1{:}T_1.t_1)\ v\ \texttt{in}\ t_2\rangle \xrightarrow{p\langle\tau\rangle} p\langle \texttt{let}\ x_2{:}T_2 = t_1[v/x_1]\ \texttt{in}\ t_2\rangle$
From the well-typedness assumption and inverting rules T-THREAD, T-LET, T-APP

and T-ABS$_1$, we get:

$$\dfrac{\dfrac{\dfrac{\lfloor \hat{T}_1 \rfloor = T_1 \quad C,C';\Gamma,x_1{:}\hat{T}_1 \vdash t_1 : \hat{T}_1' :: \varphi_1}{C,C';\Gamma \vdash \mathtt{fn}\, x_1{:}T_1.t_1 : \hat{T}_1 \xrightarrow{\varphi_1} \hat{T}_1' :: \varepsilon} \quad C,C';\Gamma \vdash v : \hat{T}_1'' :: \varepsilon \quad C,C' \vdash \hat{T}_1 \geq \hat{T}_1''}{C,C';\Gamma \vdash (\mathtt{fn}\, x_1{:}T_1.t_1)\ v : \hat{T}_1' :: \varphi_1} \quad C;\Gamma,x_2{:}\forall \vec{Y}{:}C'.\hat{T}_1' \vdash t_2 : \hat{T}_2 :: \varphi_2}{C;\Gamma \vdash \mathtt{let}\, x_2{:}T_2 = (\mathtt{fn}\, x_1{:}T_1.t_1)\ v\ \mathtt{in}\, t_2 : \hat{T}_2 :: \varphi_1;\varphi_2}}{\Gamma \vdash p\langle \mathtt{let}\, x_2{:}T_2 = (\mathtt{fn}\, x_1{:}T_1.t_1)\ v\ \mathtt{in}\, t_2\rangle :: p\langle \varphi_1;\varphi_2;C\rangle} \tag{10}$$

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 2) on the left-most subgoal gives $C,C';\Gamma \vdash t_1[v/x_1] : \hat{T}_1''' :: \varphi_1'$ where

$$C,C' \vdash \hat{T}_1''' \leq \hat{T}_1' \tag{11}$$

and $C,C' \vdash \varphi_1' \sqsubseteq \varphi_1$. Equation (11) implies with Lemma 18 $\forall \vec{Y}{:}C'.\hat{T}_1''' \gtrsim^g \forall \vec{Y}{:}C'.\hat{T}_1'$. Thus by using weakening (Corollary 3) on the right-most subgoal of (10) we get $C;\Gamma,x_2{:}\forall \vec{Y}{:}C'.\hat{T}_1''' \vdash t_2 : \hat{T}_2' :: \varphi_2'$ with $C \vdash \hat{T}_2' \leq \hat{T}_2$ and $C \vdash \varphi_2' \sqsubseteq \varphi_2$. Thus we can derive

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

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

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

$$\dfrac{\dfrac{\dfrac{\lfloor \hat{T}_f \rfloor = T_f \quad \hat{T}_f = \hat{T}_1 \xrightarrow{\varphi_1} \hat{T}_1' \quad C;C';\Gamma,x_1{:}\hat{T}_1,f{:}\hat{T}_f \vdash t_1 : \hat{T}_1' :: \varphi_1}{C,C';\Gamma \vdash \mathtt{fun}\, f{:}T_f.x_1{:}T_1.t_1 : \hat{T}_1 \xrightarrow{\varphi_1} \hat{T}_1' :: \varepsilon} \quad C,C';\Gamma \vdash v : \hat{T}_1'' :: \varepsilon \quad C,C' \vdash \hat{T}_1 \geq \hat{T}_1''}{C,C';\Gamma \vdash (\mathtt{fun}\, f{:}T_f.x_1{:}T_1.t_1)\ v : \hat{T}_1' :: \varphi_1} \quad C;\Gamma,x_2{:}\forall \vec{Y}{:}C'.\hat{T}_1' \vdash t_2 : \hat{T}_2 :: \varphi_2}{C;\Gamma \vdash \mathtt{let}\, x_2{:}T_2 = (\mathtt{fun}\, f{:}T_f.x_1{:}T_1.t_1)\ v\ \mathtt{in}\, t_2 : \hat{T}_2 :: \varphi_1;\varphi_2}}{\Gamma \vdash p\langle \mathtt{let}\, x_2{:}T_2 = (\mathtt{fun}\, f{:}T_f.x_1{:}T_1.t_1)\ v\ \mathtt{in}\, t_2\rangle :: p\langle \varphi_1;\varphi_2;C\rangle} \tag{12}$$

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 2) on the left-most subgoal gives $C,C';\Gamma \vdash t_1[v/x_1][\mathtt{fun}\, f{:}T_f.x_1{:}T_1.t_1/f] : \hat{T}_1''' :: \varphi_1'$ where

$$C,C' \vdash \hat{T}_1''' \leq \hat{T}_1' \tag{13}$$

and $C,C' \vdash \varphi_1' \sqsubseteq \varphi_1$. Equation (13) implies with Lemma 18 $\forall \vec{Y}{:}C'.\hat{T}_1''' \gtrsim^g \forall \vec{Y}{:}C'.\hat{T}_1'$. Thus by using weakening (Corollary 3) on the right-most subgoal of (12) we get $C;\Gamma,x_2{:}\forall \vec{Y}{:}C'.\hat{T}_1''' \vdash t_2 : \hat{T}_2' :: \varphi_2'$ with $C \vdash \hat{T}_2' \leq \hat{T}_2$ and $C \vdash \varphi_2' \sqsubseteq \varphi_2$. Therefore we can derive:

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

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

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

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:

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

Then, by SE-REFL, $C \vdash \varepsilon;\varphi \sqsupseteq \varepsilon;\varphi$. Finally, $\sigma_1 \equiv_{\hat{\theta}} \sigma_2$ before the step implies that also $\sigma_1' \equiv_{\hat{\theta}} \sigma_2$ 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:

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

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

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

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 wrt. C'. Both together with $C, C' \vdash X \sqsupseteq \rho.\texttt{lock}$ give by Lemma 10 that also $\vec{Y} \cap fv(\rho.\texttt{lock}) = \emptyset$. Thus, all conditions of Lemma 11 are satified, yielding $C \vdash X \sqsupseteq \rho.\texttt{lock}$. Then, by SE-SEQ $C \vdash X;\varphi \sqsupseteq \rho.\texttt{lock};\varphi$, i.e., $C \vdash X;\varphi \xrightarrow{\rho.\texttt{lock}}_{\sqsubseteq} \varphi$. Then, by RE-LOCK we get

$$C; \sigma_2 \vdash p\langle X;\varphi\rangle \xrightarrow{p\langle\rho.\texttt{lock}\rangle}_{\sqsubseteq} C; \sigma_2' \vdash p\langle\varphi\rangle \tag{14}$$

where $\sigma'(\rho,p) = \sigma(\rho,p)+1$. Thus, the assumption $\sigma_1 \equiv_{\hat{\theta}} \sigma_2$ before the step implies $\sigma_1' \equiv_{\hat{\theta}} \sigma_2'$ 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: $\sigma_1 \vdash p_1\langle \texttt{let}\ x{:}T = \texttt{spawn}\ t_2^{\varphi_2}\ \texttt{in}\ t_1\rangle \xrightarrow{p_1\langle \texttt{spawn}(\varphi_2)\rangle} \sigma_1 \vdash p_1\langle \texttt{let}\ x{:}T = () \ \texttt{in}\ t_1\rangle \parallel p_2\langle t_2\rangle$

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

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

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:

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

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

$$C;\sigma \vdash p_1\langle X;\varphi_1\rangle \xRightarrow{p_1\langle \texttt{spawn}(\varphi_2)\rangle}_{\sqsubseteq} C;\sigma \vdash p_1\langle \varphi_1\rangle \parallel p_2\langle \varphi_2\rangle \tag{15}$$

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

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

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

For part 3, we are given $waits(\sigma_1 \vdash p\langle t\rangle, p, l^\rho)$, i.e., by Definition 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,

$$\sigma_1(l^\rho) = q(n) \quad \text{with} \quad q \neq p\,. \tag{16}$$

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

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

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

We are given $\vec{Y} \cap fv(C,X) = \emptyset$, and furthermore, well-formedness of $\forall\vec{Y}{:}C'.\mathrm{L}^\rho$ implies $\vec{Y}$ is upward closed wrt. C'. Both together with $C,C' \vdash X \sqsupseteq \rho.\mathtt{lock}$ give by Lemma 10 that also $\vec{Y} \cap fv(\rho.\mathtt{lock}) = \emptyset$. Thus, all conditions of Lemma 11 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. $C \vdash X;\varphi \xRightarrow{\rho.\mathtt{lock}}_{\sqsubseteq} \varphi$.

The assumption $\sigma_1 \equiv_{\hat{\theta}} \sigma_2$ and equation (16) imply by the wait-equivalence definition (Definition 10) that $\sigma_2(\rho,q) \geq 1$. Thus, by Definition 8, we have $waits_{\sqsubseteq}(\sigma_2 \vdash p\langle X;\varphi\rangle, p, \rho)$, as required. □

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

Corollary 4. *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. □

B.3 Equivalence of the two formulations

Lemma 21 (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. □

Lemma 22 (Completeness). *Assume $C;\Gamma \vdash_s t : \hat{T} :: \varphi$, 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. *$C \vdash \theta'\varphi' \sqsubseteq \varphi$,*

for some θ'.

Proof. Similar to the proof in [15]. □

B.4 Additional proofs

Proof (Preservation of deadlock freedom, Lemma 1 on page 15). Similar to the proof of the corresponding lemma in [14]. □

Proof (Composition lemma 2 on page 16). Straightforward. □

Proof (Subject Reduction Lemma 3 on page 16). 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 20 that

$$\Gamma \vdash_s p\langle t'\rangle :: p\langle \varphi''; C\rangle \quad \text{with} \quad C \vdash \varphi \sqsupseteq \varphi'', \quad \text{and} \quad \sigma_1' \equiv_{\hat{\theta}} \sigma_2 \,. \tag{17}$$

By the first condition in equation (17) and the completeness (cf. Lemma 22), we get

$$\Gamma \vdash_a p\langle t'\rangle :: p\langle \varphi'; C'\rangle, \quad C \vdash \theta' C' \quad \text{and} \quad C \vdash \theta'\varphi' \sqsubseteq \varphi'' \tag{18}$$

The second inequality in equation (17) and the last one in equation (18) 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 (18), and $\sigma_1' \equiv_{\hat{\theta}} \sigma_2$ in equation (17) conclude part 1.

It is analogously for the other parts. □

Proof (Corollary 1 on page 16). The result follows from soundness in Lemma 21, Corollary 4 and commpleteness in Lemma 22. □

References

1. R. Agarwal, L. Wang, and S. D. Stoller. Detecting potential deadlocks with state analysis and run-time monitoring. In S. Ur, E. Bin, and Y. Wolfsthal, editors, *Proceedings of the Haifa Verification Conference 2005*, volume 3875 of *Lecture Notes in Computer Science*, pages 191–207. Springer-Verlag, 2006.
2. T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency*. Imperial College Press, 1999.
3. C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA) '02 (Seattle, USA)*. ACM, Nov. 2002. In *SIGPLAN Notices*.
4. E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.
5. L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1985. CST-33-85.
6. L. Damas and R. Milner. Principal type-schemes for functional programming languages. In *Ninth Annual Symposium on Principles of Programming Languages (POPL) (Albuquerque, NM)*, pages 207–212. ACM, January 1982.
7. E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [8].
8. F. Genyus. *Programming Languages*. Academic Press, 1968.
9. J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, Dec. 1969.
10. N. Kidd, T. W. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, 2011.
11. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, Dec. 1978.
12. C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.

13. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE09)*. IEEE, 2009.
14. K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. *Journal of Logic and Algebraic Programming*, 81(3):331–354, Mar. 2012. A preliminary version was published as University of Oslo, Dept. of Computer Science Technical Report 404, March 2011.
15. K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by data race detection. Technical report 421, University of Oslo, Dept. of Informatics, Oct. 2012.
16. K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In G. Ramalingam, editor, *APLAS 2008*, volume 5356 of *Lecture Notes in Computer Science*, pages 155–170. Springer-Verlag, 2008.
17. J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.
18. V. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In A. R. Beresford and S. J. Gay, editors, *Pre-Proceedings of the Workshop on Programming Language Approaches to Concurrent and Communication-Centric Software (PLACES 2009)*, volume 17 of *EPTCS*, pages 95–109, 2009.

$$\frac{\Gamma(x) = \hat{S}}{C;\Gamma \vdash x : \hat{S} :: \varepsilon}\ \text{T-VAR}$$

$$\frac{C \vdash \rho \sqsupseteq \{\pi\}}{C;\Gamma \vdash \ \mathtt{new}^{\pi}_{\rho}\, L :\ \mathrm{L}^{\rho} :: \varepsilon}\ \text{T-NEWL} \qquad \frac{}{C;\Gamma \vdash l^{\rho} : \mathrm{L}^{\rho} :: \varepsilon}\ \text{T-LREF}$$

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

$$\frac{C;\Gamma \vdash v_1 : \hat{T}_2 \xrightarrow{\varphi} \hat{T}_1 :: \varepsilon \qquad C;\Gamma \vdash v_2 : \hat{T}_2 :: \varepsilon}{C;\Gamma \vdash v_1\ v_2 : \hat{T}_1 :: \varphi}\ \text{T-APP} \qquad \frac{C;\Gamma \vdash v : \mathtt{Bool} :: \varepsilon \quad C;\Gamma \vdash e_1 : \hat{T} :: \varphi \quad C;\Gamma \vdash e_2 : \hat{T} :: \varphi}{C;\Gamma \vdash \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2 : \hat{T} :: \varphi}\ \text{TA-COND}$$

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

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

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

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

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

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

Table 11: Type and effect system

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

Table 12: Type and effect system (Global)

$$\frac{\Gamma(x) = \forall \vec{Y}{:}C'.\hat{T} \qquad \forall \vec{Y}{:}C'.\hat{T} \text{ solvable from } C \text{ by } \theta}{C;\Gamma \vdash x : \theta\hat{T} :: \varepsilon}\ \text{T-Var}$$

$$\frac{C \vdash \rho \sqsupseteq \{\pi\}}{C;\Gamma \vdash \mathtt{new}^{\pi}_{\rho}\, L : \mathrm{L}^{\rho} :: X}\ \text{T-NewL} \qquad \frac{}{C;\Gamma \vdash l^{\rho} : \mathrm{L}^{\rho} :: \varepsilon}\ \text{T-LRef}$$

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

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

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

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

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

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

Table 13: Type and effect system (syntax directed)