Effect-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 summarised 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 behaviour 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 [4], a deadlocked state is marked by a number of processes, which forms a cycle where each process is unwilling to release its own resource, and is waiting on the resource held by its neighbour. The inherent non-determinism makes deadlocks hard to detect and to reproduce. We present a static analysis using behavioural effects to detect deadlocks in a higher-order concurrent calculus. Deadlock freedom, an important safety property for concurrent programs, is a thread-global property, i.e., two or more processes form a deadlock. The presented approach works in two stages: in a first stage, an effect-type system uses a static behavioural abstraction of the codes' behaviour, concentrating on the lock interactions. To detect potential deadlocks on the global level, the combined individual abstract thread behaviours are explored in the second stage.

Two challenges need to be tackled to make the approach applicable in practice. For the first stage on the thread local level, the static analysis must be able to *derive* the abstract behaviour, not just check compliance of the code with a user-provided description. This is the problem of type and effect *inference* or reconstruction. As usual, the abstract behaviour needs to over-approximate the concrete one, i.e., concrete and abstract descriptions are connected by some *simulation* relation: everything the concrete system does, the abstract one can do as well. For the second stage, exploring the (abstract) state space on the global level, obtaining *finite* abstractions is crucial. In our setting, there are four principal sources of infinity: the calculus allows 1) recursion, supports 2) dynamic thread creation, as well as 3) dynamic lock creation, and 4) unbounded lock counters for re-entrant locks. Our approach offers sound abstractions for the mentioned sources of unboundedness, except for dynamic thread creation. We first shortly present in a non-technical manner the ideas behind the abstractions before giving the formal theory.

* Partly funded by the EU projects FP7-610582 (ENVISAGE) and FP7-612985 (UPSCALE).

1.1 Effect inference on the thread local level

In the first stage of the analysis, a behavioural type and effect system is used to over-approximate the lock-interactions of a single thread. To force the user to annotate the program with the expected behaviour in the form of effects is impractical, so the type and especially the behaviour should be inferred automatically. Effect inference, including inferring behavioural effects, has been studied earlier and applied to various settings, including obtaining static over-approximations of behaviour for concurrent languages by Amtoft et al. [2]. We apply effect inference to deadlock detection and as is standard (cf. e.g., [11,16,2]), the inference system is constraint-based, where the constraints in particular express an approximate order between behaviours. Besides being able to infer the behaviour, it is important that the static approximation is as precise as possible. For that it is important that the analysis may distinguish different instances of a function body depending on their calling context, i.e., the analysis should be *polymorphic* or *context-sensitive*. This can be seen as an extension of let-polymorphism to effects and using constraints. The effect reconstruction resembles the known type-inference algorithm for let-polymorphism by Damas and Milner [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 [13] 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 x1 = new^π1 L in let x2 = new^π2 L in
let f = fn x:L . ( x.lock; x.lock )
in spawn(f(x2)); f(x1)
```

The main thread, after creating two locks and defining function f, spawns a thread, and 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 [13] determines the potential origin of locks by data-flow analysis. The analysis cannot distinguish the two instances of f (the analysis is context-*insensitive*), and therefore forces that the type of the formal parameter is, at best, $\mathtt{L}^{\{\pi_1,\pi_2\}}$. 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

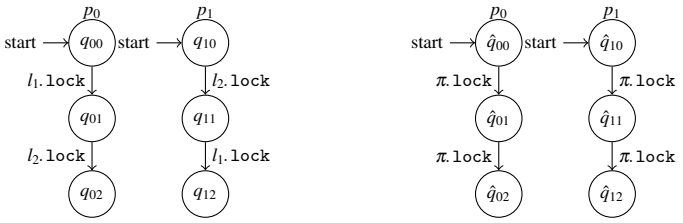
Lock abstraction For dynamic data allocation, a standard abstraction is to *summarize* all data allocated at a given program point into one abstract representation. In the presence of loops or recursion, the abstracting function mapping concrete locks to their abstract representation is necessarily non-injective. For concrete, ordinary programs it is clear that identifying locks may change the behaviour of the program. Identification of locks is in general tricky (and here in particular in connection with deadlocks): on

the one hand it leads to *less* steps, in that lock-protected critical sections may become larger. On the other hand it may lead to *more* steps at the same time, and deadlocks may disappear when identifying locks. This form of summarizing lock abstraction is problematic when analyzing properties of concurrent programs, and has been observed elsewhere as well, cf. e.g., Kidd et al. in [9].

To obtain 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, needs to show at least the behaviour of the concrete level, i.e., we expect that 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: b) a concrete program waiting on a lock and unable to make a step thereby, must imply an analogous situation on the abstract level, lest we should miss deadlocks. Let's write $l, l_1, l_2, \ldots$ for concrete lock references and $\pi, \pi', \ldots$ for program points of lock creation, i.e., abstract locks. To satisfy a): when a concrete program takes a lock, the abstract one must be able to "take" the corresponding abstract lock, say π. A consequence of a) is that taking an abstract lock is always enabled. That is consistent with the abstraction as described where the abstract lock π confuses an arbitrary number of concrete locks including, e.g., those freshly created, which may be taken.

Thus, abstract locks lose their "mutual exclusion" capacity: whereas a concrete heap is a mapping which associates to each lock reference the number of times that *at most one* process is holding it, an abstract heap $\hat{\sigma}$ records how many times an abstract lock π is held by the various processes, e.g., thrice by one process and twice by another. The corresponding natural number abstractly represents the *sum* of the lock values of all concrete locks (per process). Without ever blocking, the abstraction leads to more possible steps, but to cater for b), the abstraction still needs to appropriately define, given an abstract heap and an abstract lock π, when a process waits on the abstract lock, as this may indicate a deadlock. The definition has to capture all possibilities of waiting on one of the corresponding concrete locks (see Definition 6 later). The sketched intuitions to obtain a sound abstract summary representation for locks and correspondingly for heaps lead also to a corresponding refinement of "over-approximation" in terms of simulation: not only must the a) positive behaviour be preserved as in standard simulation, but also the b) possibility of waiting on a lock and ultimately the possibility of deadlock needs to be preserved. For this we introduce the notion of *deadlock sensitive* simulation (see Definition 9). The definition is analogous to the one from [13]. However, it takes into account now that the analysis is polymorphic and the definition is no longer based on a direct operational interpretation of the behaviour of the effects. Instead it is based on the behavioural constraints used in the inference systems.

The points discussed are illustrated in Fig. 1, where the left diagram Fig. 1a depicts two threads running in parallel and trying to take two concrete locks, l_1 and l_2 while Fig. 1b illustrates an abstraction of the left one where the two concrete locks are summarized by the abstract lock π (typically because being created at the same program point). The concrete program obviously may run into a deadlock by reaching commonly the states q_{01} and q_{11}, where the first process is waiting on l_2 and the second process on l_1. With the abstraction sketched above, the abstract behaviour, having reached the corresponding states $\hat{q}_{01}$ and $\hat{q}_{11}$, can proceed (in two steps) to the common states $\hat{q}_{02}$ and $\hat{q}_{12}$,



(a) Concrete (b) Abstract

Fig. 1: Lock abstraction

reaching an abstract heap where the abstract lock π is "held" twice 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 our earlier analysis [13], which could therefore deal only with a static number of locks.

Counter abstraction and further behaviour abstraction Two remaining causes of an infinite state space are the values of lock counters, which may grow unboundedly, and the fact that for each thread, the effect behaviour abstractly represents the *stack* of function calls for that thread. Sequential composition as construct for abstract behavioural effects allows to represent non-tail-recursive behaviour (corresponding to the context-free call-and-return behaviour of the underlying program). To curb that source of infinity, we allow for replacing the behaviour by a tail-recursive over-approximation. The precision of the approximation can be adapted in choosing the depth of calls after which the call-structure collapses into an arbitrary, chaotic behaviour. A finite abstraction for the lock-counters is achieved similarly by imposing an upper bound on the considered lock counter, beyond which the locks behave non-deterministically. Again, for both abstractions it is crucial, that the abstraction preserves also deadlocks, which we capture again using the notion of deadlock-sensitive simulation. These two abstractions have been formulated and proven in the non-context-sensitive setting of [13].

To summarize, compared to [13], the paper makes the following contributions: 1) the effect analysis is generalized to a context-sensitive formulation, using constraints, for which we provide 2) an inference algorithm. Finally, 3) we allow summarizing multiple concrete locks into abstract ones, while still preserving deadlocks.

The rest of the paper is organized as follows. After presenting syntax and semantics of the concurrent calculus in Section 2, the behavioural 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.

2 Calculus

This section presents the syntax and semantics for our calculus. 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. A thread 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 applications and conditionals. Threads are created with the expression $\texttt{spawn}\ t$. For lock manipulation, `new L` yields the reference to a newly created lock (initially free), and the operations $v.\ \texttt{lock}$ and $v.\ \texttt{unlock}$ deal with acquiring and releasing a lock. Values which are evaluated expressions are variables, lock references, and function abstractions, where $\texttt{fun}\ f{:}T.x{:}T.t$ represents recursive function definitions.

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

Table 1: Abstract syntax

Semantics

The small-step operational semantics, presented next, distinguishes between local and global steps (cf. Table 2). The local steps are straightforward and therefore left out here. 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 $\sigma(l) = p(n+1)$ (indicating the pair of process identifier p and lock count n) 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 2). 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).

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

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

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

$$\frac{\sigma \vdash P_1 \to \sigma' \vdash P_1'}{\sigma \vdash P_1 \parallel P_2 \to \sigma' \vdash P_1' \parallel P_2}\ \text{R-PAR} \qquad \frac{\sigma' = \sigma[l \mapsto \mathit{free}] \qquad l \text{ is fresh}}{\sigma \vdash p\langle \texttt{let}\ x{:}T = \texttt{new L}\ \texttt{in}\ t\rangle \to \sigma' \vdash p\langle \texttt{let}\ x{:}T = l\ \texttt{in}\ t\rangle}\ \text{R-NEWL}$$

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

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

Table 2: 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 if furthermore there exists a σ' s.t. $\sigma' \vdash P \xrightarrow{p\langle l\,\texttt{lock}\rangle} \sigma'' \vdash P'$.*

The notion of (resource) deadlock used is rather standard, where a number of processes waiting for each other's locks in a cyclic manner constitute a deadlock (see also [13]). In our setting with re-entrant locks, a process cannot deadlock "on itself".

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

3 Type system

Next we present an effect type system to derive behavioural information which 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 behaviour. The representation extends our earlier system [13] by making the analysis *context-sensitive* and furthermore by supporting type and effect *inference*, both important from a practical point of view. Being context-sensitive, making the effect system polymorphic, increases the precision of the analysis. Furthermore, inference removes the burden from the programmer to annotate the program appropriately to allow checking for potential deadlock. These extensions follow standard techniques for behaviour inference, see for instance Amtoft et al. [2] and type-based flow analysis, see e.g., Mossin [11]. 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.

3.1 Types, effects, and constraints

The analysis performs a data flow analysis to track the usage of locks. For that purpose, the lock creation statements are equipped with labels, writing $\texttt{new}^{\pi}\ \texttt{L}$, where π is taken from a countably infinite set of labels. As usual, the labels π are assumed unique in a given program. The grammar for annotations, types, and effects is given in Tables 3 and 4. We use r to denote sets of πs with ρ representing corresponding variables. Types include basic types, represented by B, such as the unit type `Unit`, booleans, integers, etc., functional types with latent *effect* φ, and lock types $\texttt{L}^r$ where the annotation r captures the flow information about the potential places where the lock is created. This information will be reconstructed, and the user writes types without annotations (the "underlying" types) in the program. We write T (and its syntactic variants) as meta-variables for the underlying types, and $\hat{T}$ (and its syntactic variants) for the annotated types, as given in the grammar. The universally quantified types, represented by $\hat{S}$, capture functions which are polymorphic in locations and effects.

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
$\hat{S}$	$::= \forall \vec{Y}{:}C.\ \hat{T} \xrightarrow{\varphi} \hat{T} \mid \hat{T}$	type schemes
C	$::= \emptyset \mid \rho \sqsupseteq r, C \mid X \sqsupseteq \varphi, C$	constraints

Table 3: Types and type schemes

Whereas the type of an expression captures the results of the computations of the expression if it terminates, the effect captures the *behaviour* during the computations. For the deadlock analysis, we capture the lock interactions as effects, i.e., which locks are accessed during execution and in which order. The effects (cf. Table 4) are split between a (thread-) local level φ and a global level Φ. The empty effect is denoted by ε, representing behaviour without lock operations. Sequential composition is represented by $\varphi_1;\varphi_2$. The choice between two effects $\varphi_1 + \varphi_2$, as well as recursive effects $recX.\varphi$, is actually not generated by the algorithm; they would show up when solving the constraints generated by the algorithm. We included their syntax for completeness. Note also that recursion is not polymorphic. Labels a capture the three basic effects: spawning a new process with behaviour φ is represented by $\texttt{spawn}\ \varphi$, while $r.\texttt{lock}$ and $r.\texttt{unlock}$ respectively capture lock manipulations, acquiring and releasing a lock, where r refers to the possible points of creation. Silent transitions are represented by τ. Lock-creation has no corresponding effect, as newly created locks are initially free, i.e., with a lock-count of 0. On the abstract level, locks are summarized by the *sum* of all locks created at given point. Hence lock creation 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

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

Table 4: Effects

ρ is, as mentioned, a flow variable and X an effect or behaviour 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 formal system presented in this paper uses a constraint-based flow analysis as proposed by Mossin [11] for lock information. Likewise, the effects captured as a sequence of behaviour are formulated using constraints.

3.2 Type inference

Next we present a type inference algorithm which derives types and effects and generates corresponding constraints (see Table 6 below). It is formulated in a rule-based manner, with judgments of the form: $\Gamma \vdash e : \hat{T} :: \varphi;C$. The system is syntax-directed, i.e., algorithmic, where Γ and e are considered as "input", and the annotated type $\hat{T}$, the effect φ, and the set of constraints C as "output". Concentrating on the flow information and the effect part, expressions e are type-annotated with the *underlying* types, as given in Table 3. In contrast, e contains no flow or effect annotations; those are derived by the algorithmic type system. It would be straightforward to have the underlying types reconstructed as well, using standard type inference à la Hindley/Milner/Damas [6,5,8]. For simplicity, we focus on the type annotations and the effect part. For locks, the flow annotation over-approximates the point of lock creation, and finally, φ over-approximates the lock-interactions while evaluating e. As usual, the behavioural over-approximation is a form of simulation. For our purpose, we will define a particular, deadlock-sensitive form of simulation. These intended over-approximations are understood relative to the generated constraints C, i.e., *all* solutions of C give rise to a sound over-approximation in the mentioned sense. Solutions to a constraint set C are ground substitutions θ, assigning label sets to flow variables ρ and effect variables X. 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 information. Solving the constraints is done after the algorithmic type system, but to allow for the most precise solution afterward, each rule should generate the most general constraint set, i.e., the one which allows the maximal set of solutions. This is achieved using *fresh* variables for each additional constraint. In the system below, new constraints are generated from requesting that types are in a "subtype" relationship. In our setting, "subtyping" concerns the flow annotations on the lock types and the latent effects on function types. For instance in rule TA-APP in Table 6, 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:*

$$B \leq B \vdash \emptyset \quad \text{C-Basic} \qquad \mathrm{L}^{\rho_1} \leq \mathrm{L}^{\rho_2} \vdash \{\rho_1 \sqsubseteq \rho_2\} \quad \text{C-Lock} \qquad \frac{\hat{T}_1' \leq \hat{T}_1 \vdash C_1 \quad \hat{T}_2 \leq \hat{T}_2' \vdash C_2 \quad C_3 = \{X \sqsubseteq X'\}}{\hat{T}_1 \xrightarrow{X} \hat{T}_2 \leq \hat{T}_1' \xrightarrow{X'} \hat{T}_2' \vdash C_1, C_2, C_3} \text{ C-Arrow}$$

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 5. The operation $\wedge$ is defined dually.*

$$\frac{B_1 = B_2}{B_1 \vee B_2 = B_1; \emptyset} \text{ LT-Basic} \qquad \frac{\hat{T}_1' \wedge \hat{T}_1'' = \hat{T}_1; C_1 \qquad \hat{T}_2' \vee \hat{T}_2'' = \hat{T}_2; C_2 \qquad X_1 \sqcup X_2 = X; C_3}{\hat{T}_1' \xrightarrow{X_1} \hat{T}_2' \vee \hat{T}_1'' \xrightarrow{X_2} \hat{T}_2'' = \hat{T}_1 \xrightarrow{X} \hat{T}_2; C_1, C_2, C_3} \text{ LT-Arrow}$$

$$\frac{\rho \textit{ fresh} \quad \mathrm{L}^{\rho_1} \leq \mathrm{L}^{\rho} \vdash C_1 \quad \mathrm{L}^{\rho_2} \leq \mathrm{L}^{\rho} \vdash C_2}{\mathrm{L}^{\rho_1} \vee \mathrm{L}^{\rho_2} = \mathrm{L}^{\rho}; C_1, C_2} \text{ LT-Lock} \qquad \frac{X \textit{ fresh} \qquad C = \{\varphi_1 \sqsubseteq X, \varphi_2 \sqsubseteq X\}}{\varphi_1 \sqcup \varphi_2 = X; C} \text{ LE-Eff}$$

Table 5: Least upper bound

The rules for the type and effect system then are given in Table 6. A variable has no effect and its type (scheme) is looked up from the context Γ. The constraints C that may occur in the type scheme, are given back as constraints of the variable x, replacing the $\forall$-bound variables $\vec{Y}$ in C by fresh ones. Lock creation at point π (cf. TA-NewL) is of the type 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. 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 needs to be a subtype of the input type of the function, and corresponding

constraints C_3 are generated by $\hat{T}_2' \leq \hat{T}_2 \vdash C_3$. 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 (cf. Table 5). The `let`-construct (cf. TA-LET) for the sequential composition has an effect $\varphi_1;\varphi_2$. To support context-sensitivity (corresponding to let-polymorphism), the let-rule is where the generalization over the type-level variables happens. In the first approximation, given e_1 is of $\hat{T}_1$, variables which do not occur free in Γ can be generalized over to obtain $\hat{S}_1$, which quantifies over the maximal number of variables for which such generalization is sound. In the setting here, the quantification affects only flow variables ρ and effect variables X. The close-operation $close(\Gamma,\varphi,C,\hat{T})$ first computes the set of all "relevant" free variables in a type $\hat{T}$ and the constraint C by the operation $close_{\uparrow\downarrow}(fv(\hat{T}_1),C_1)$ which finds the upward and downward closure of the free variables in $\hat{T}_1$ wrt. C_1. Among the set of free variables, those that are free in the context or in the effect, as well as the corresponding downward closure, are non-generalizable and are excluded. (See also Amtoft et al. [2]). 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 $p_1\langle\varphi_1;C_1\rangle \parallel \ldots \parallel p_n\langle\varphi_n;C_n\rangle$.

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

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

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

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

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

$$\frac{\rho \textit{ fresh}}{\Gamma \vdash\ \mathtt{new}^\pi\ \mathtt{L} :\ \mathtt{L}^\rho :: \varepsilon;\rho \sqsupseteq \{\pi\}}\ \text{TA-NEWL}$$

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

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

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

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

Table 6: Algorithmic effect inference

3.3 Semantics of the behaviour

Next we are going to define the transition relation on the abstract behaviour with the effect-constraints. Given a constraint set C, we interpret $C \vdash a;\varphi_2 \sqsubseteq \varphi_1$ as φ_1 may first perform an a-step before executing φ_2, where a is one of the labels from Table 4 which do not include the τ-label. See also [2]. The relation $C \vdash \varphi_1 \sqsubseteq \varphi_2$ is defined in Table 7.

Definition 5. *The transition relation between configurations of the form $C;\hat{\sigma} \vdash \Phi$ is given inductively by the rules of Table 8, 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 correspondingly captures the three possible steps we describe in the behaviour, namely creating a new process with a given behaviour, locking and unlocking. Analogous to the corresponding case in the concrete semantics, rule RE-SPAWN covers the creation of a new (abstract) thread and leaves the abstract heap unchanged. Taking a lock increases the corresponding lock count by one (cf. RE-LOCK). Unlocking works similarly by decreasing the lock count by one (cf. RE-UNLOCK), where the second premise makes sure the lock count stays non-negative. The transitions of a global effect Φ consist of the transitions of the individual thread (cf. RE-PAR). As stipulated by rule RE-LOCK, the step to take an abstract lock is always enabled, which is in obvious contrast to the behaviour of concrete locks. To ensure that the abstraction preserves deadlocks requires to adapt the definition of what it means that an abstract behaviour waits on a lock (cf. also Definition 1 for concrete programs and heaps).

Definition 6 (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 $C \vdash \varphi \stackrel{\rho.\mathtt{lock}}{\Longrightarrow}_{\sqsubseteq} \varphi'$ but $\hat{\sigma}(\rho, q) \geq 1$ for some $q \neq p$.*

Definition 7 (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 it is* deadlock free.

$$\varepsilon;\varphi \equiv \varphi \quad \text{EE-UNIT} \qquad \varphi_1;(\varphi_2;\varphi_3) \equiv (\varphi_1;\varphi_2);\varphi_3 \quad \text{EE-ASSOC} \qquad \frac{C \vdash \varphi_1 \sqsubseteq \varphi_1' \qquad C \vdash \varphi_2 \sqsubseteq \varphi_2'}{C \vdash \varphi_1;\varphi_2 \sqsubseteq \varphi_1';\varphi_2'}\ \text{S-SEQ}$$

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

$$\frac{C \vdash r_1 \subseteq r_2 \qquad C \vdash r_2 \subseteq r_3}{C \vdash r_1 \subseteq r_3}\ \text{S-TRANS}_L \qquad \frac{C \vdash \varphi_1 \sqsubseteq \varphi_2 \qquad C \vdash \varphi_2 \sqsubseteq \varphi_3}{C \vdash \varphi_1 \sqsubseteq \varphi_3}\ \text{S-TRANS}_E$$

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

Table 7: Orders on behaviours

$$\frac{C \vdash \varphi \xrightarrow{\rho.\mathtt{lock}}_{\sqsubseteq} \varphi' \quad \hat{\sigma}'(\rho,p) = \hat{\sigma}(\rho,p)+1}{C;\hat{\sigma} \vdash p\langle\varphi\rangle \xrightarrow{p\langle\rho.\mathtt{lock}\rangle}_{\sqsubseteq} C;\hat{\sigma}' \vdash p\langle\varphi'\rangle}\ \text{RE-LOCK} \qquad \frac{C \vdash \varphi \xrightarrow{\mathtt{spawn}(\varphi'')}_{\sqsubseteq} \varphi'}{C;\hat{\sigma} \vdash p_1\langle\varphi\rangle \xrightarrow{p_1\langle\mathtt{spawn}(\varphi'')\rangle}_{\sqsubseteq} C;\hat{\sigma} \vdash p_1\langle\varphi'\rangle \parallel p_2\langle\varphi''\rangle}\ \text{RE-SPAWN}$$

$$\frac{C \vdash \varphi \xrightarrow{\rho.\mathtt{unlock}}_{\sqsubseteq} \varphi' \quad \hat{\sigma}(\rho,p) \geq 1 \quad \hat{\sigma}'(\rho,p) = \hat{\sigma}(\rho,p)-1}{C;\hat{\sigma} \vdash p\langle\varphi\rangle \xrightarrow{p\langle\rho.\mathtt{unlock}\rangle}_{\sqsubseteq} C;\hat{\sigma}' \vdash p\langle\varphi'\rangle}\ \text{RE-UNLOCK} \qquad \frac{C;\hat{\sigma} \vdash \Phi_1 \overset{a}{\Rightarrow}_{\sqsubseteq} C;\hat{\sigma}' \vdash \Phi_1'}{C;\hat{\sigma} \vdash \Phi_1 \parallel \Phi_2 \overset{a}{\Rightarrow}_{\sqsubseteq} C;\hat{\sigma}' \vdash \Phi_1' \parallel \Phi_2}\ \text{RE-PAR}$$

Table 8: Global transitions

3.4 Soundness

A crucial part for soundness of the algorithm wrt. the semantics is preservation of well-typedness under reduction. This includes to check that the operational semantics of the program is over-approximated by the effect given by the type system captured by a simulation relation; 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.

Definition 8 (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 9 (Deadlock sensitive simulation $\lesssim^D_\sqsubseteq$). *Assume a heap-mapping θ and a corresponding wait-sensitive abstraction $\leq_\theta$. A binary relation R between configurations is a* deadlock sensitive simulation relation *(or just simulation for short) if the following holds. Assume $C_1;\hat{\sigma}_1 \vdash \Phi_1\ R\ C_2;\hat{\sigma}_2 \vdash \Phi_2$ with $\hat{\sigma}_1 \leq_\theta \hat{\sigma}_2$. Then:*

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

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 $\xrightarrow{p\langle a\rangle}_\sqsubseteq$ is replaced by $\xrightarrow{p\langle a\rangle}$ for the program configurations.

The notation $\xrightarrow{p\langle a\rangle}$ is used for weak transitions, defined as $\xrightarrow{p\langle\tau\rangle}{}^* \xrightarrow{p\langle a\rangle}$. This relation captures the internal steps which are ignored when relating two transition systems by simulation. It is obvious that the binary relation $\lesssim^D_\sqsubseteq$ is itself a deadlock simulation. The relation is transitive and reflexive. Thus, if $C_1;\hat{\sigma}_1 \vdash \Phi_1 \lesssim^D_\sqsubseteq C_2;\hat{\sigma}_2 \vdash \Phi_2$, the property of

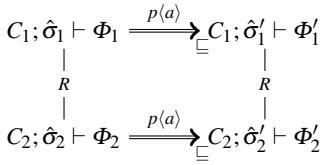


Fig. 2: Deadlock sensitive simulation $\lesssim^D_{\sqsubseteq}$

deadlock freedom is straightforwardly carried over from the more abstract behaviour to the concrete one (cf. Lemma 1).

Lemma 1 (Preservation of deadlock freedom). *Assume* $C_1;\hat{\sigma}_1 \vdash \Phi_1 \lesssim^D_{\sqsubseteq} C_2;\hat{\sigma}_2 \vdash \Phi_2$. *If* $C_2;\hat{\sigma}_2 \vdash \Phi_2$ *is deadlock free, then so is* $C_1;\hat{\sigma}_1 \vdash \Phi_1$.

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

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

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

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

1. $\sigma_1 \vdash p\langle t\rangle \xrightarrow{p\langle\tau\rangle} \sigma_1' \vdash p\langle t'\rangle$, *then* $\Gamma \vdash p\langle t'\rangle :: p\langle\varphi';C'\rangle$ *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* $C \vdash \theta' C'$, *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 behavioural description of a thread's behaviour concerning its lock interactions which then is used to explore the abstract state space to detect potential deadlocks. The static analysis is developed for a concurrent calculus with higher-order functions and dynamic lock creation. Covering lock creation by an appropriate abstraction extends our earlier work [13] for deadlock detection using behaviour abstraction. Another important extension of that work is to enhance the precision by making the analysis context-sensitive and furthermore to support effect inference ([13] in contrast required the programmer to provide the behaviour annotations manually). The analysis is shown sound, i.e., the abstraction preserves deadlocks of the program. Formally that is captured by an appropriate notion of simulation ("deadlock-sensitive simulation").

Related work Deadlocks are a well-known problem in concurrent programming and a vast number of techniques for statically and dynamically detecting deadlocks have been investigated. 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 et al. [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. Likewise, the type inference algorithms by Suenaga [15] and Vasconcelos et al. [17] assure deadlock freedom in a well-typed program with a strict partial order on lock acquisition. In contrast, our approach will certify two processes as safe if they take locks in orders 1-2-3 and 1-3-2, even though no fixed global order exists. Agarwal et al. [1] use above deadlock types to improve the efficiency for run-time checking with a static type system, by introducing runtime checks only for those locks where the inferred deadlock type indicates potential for deadlocks. Similar to our approach, Naik et al. [12] detect potential deadlocks with a model-checking approach by abstracting threads and locks by their allocation sites. The approach is neither sound nor complete. Kobayashi [10] presents a constraint-based type inference algorithm for detecting communication deadlocks in the π-calculus. In contrast to our system, he attaches abstract usage information onto channels, not processes. Cyclic dependencies there indicate potential deadlocks. Further differences are that channel-based communication does not have to consider reentrance, and the lack of functions avoids having to consider polymorphism and higher order. Instead of checking for deadlocks, the approach by Kidd et al. [9] 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, 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 [13], as sketched in Section 1.2. All mentioned abstractions are compatible with our notion of deadlock-sensitive simulation in 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 behaviour 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 [13]. We plan to adapt the implementation to the more general setting and to extend it with implementing type inference.

For lack of space, all proofs have been omitted here. Further details can found in an extended version of this work (cf. the technical report [14]).

References

1. R. Agarwal, L. Wang, and S. Stoller. Detecting potential deadlocks with state analysis and run-time monitoring. In *Haifa Verification Conf. '05*, volume 3875 of *LNCS*. Springer, 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 *OOPSLA '02 (Seattle, USA)*. ACM, 2002. In *SIGPLAN Notices*.
4. E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2), 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 POPL (Albuquerque, NM)*. ACM, 1982.
7. E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, TU Eindhoven, 1965.
8. J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the AMS*, 146, 1969.
9. N. Kidd, T. W. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6), 2011.
10. N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of CONCUR 2006*, volume 4137 of *LNCS*. Springer, 2006.
11. C. Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.
12. M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *31st International Conference on Software Engineering (ICSE 09)*. IEEE, 2009.
13. K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. *J. of Logic and Algebraic Programming*, 81(3), 2012.
14. K. I. Pun, M. Steffen, and V. Stolz. Lock-polymorphic behaviour inference for deadlock checking. Tech. report 436, UiO, IFI, 2013. Submitted for journal publication.
15. K. Suenaga. Type-based deadlock-freedom verification for non-block-structured lock primitives and mutable references. In *APLAS 2008*, volume 5356 of *LNCS*. Springer, 2008.
16. J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *J. of Functional Programming*, 2(3), 1992.
17. V. Vasconcelos, F. Martins, and T. Cogumbreiro. Type inference for deadlock detection in a multithreaded polymorphic typed assembly language. In *PLACES'09*, volume 17 of *EPTCS*, 2009.