# Behaviour Inference for Deadlock Checking

Ka I Pun and Martin Steffen and Volker Stolz
Department of Informatics
University of Oslo
Oslo, Norway
{violet,msteffen,stolz}@ifi.uio.no

***Abstract*—This paper extends our behavioural type and effect system for detecting dealocks in [18] by polymorphism and formalizing type inference (with respect to lock types). Our inference is defined for a simple concurrent, first-order language. From the inferred effects, after suitable abstractions to keep the state space finite, we either obtain the verdict that the program will not deadlock, or that it may deadlock. We show soundness and completeness of the type inference.**

## I. Introduction

Deadlocks are a well-known problem for concurrent programs with shared resources. As the scheduling at run-time affects the occurrence of a deadlock, deadlocks may only occur occasionally, and therefore are difficult to detect. Whether or not a deadlock occurs in a specific run in a particular program mainly depends on if the running program encounters a number of processes forming a circular chain, where each process waits for shared resources held by the others [5]. Here, we focus on a type system designed to derive suitable abstract information to analyze *deadlocks of explicit, reentrant or binary locks*, like we might find e.g., underlying Java's `synchronized` mechanism.

Deadlock *prevention* (as opposed to deadlock avoidance) statically ensures that deadlock do not occur, typically by preventing cyclic waits by enforcing an order on lock acquisitions. This idea has, e.g., been formalized in a type-theoretic setting in the form of deadlock types [4]. That system also supports type *inference* (and besides deadlocks, prevents race conditions, as well). Deadlock types are also used in [1], but not for static deadlock prevention, but for improving the efficiency for deadlock avoidance at run-time.

In contrast, the approach in [18] uses a behavioural type and effect system [2], [16] to capture lock interaction of an explicitly-typed input program, and to use that behavioural description to explore an abstraction of the state space to detect potential deadlocks. An effect system commonly captures phenomena that may occur *during* evaluation, for instance exceptions. Expressive effects, which can represent the behaviour of a program, are important for concurrent programs. In particular, the effects of our system express the relevant behaviour of a concurrent program with regard to re-entrant locks. To detect potential deadlocks, we execute the abstraction of the actual behaviour to spot cyclic waiting for shared locks among parallel threads in the program.

Compared to our earlier work [18], an algorithmic behavioural type and effect inference system [8], [7] is proposed to provide polymorphism for first-order programming, and to enhance user-friendliness and usefulness: the most general type and the most specific abstract behaviour of an implicitly-typed input program are reconstructed. The algorithmic inference is proven to be both sound and complete with respect to the specification of the type system, while the abstraction of the behaviour is shown to preserve potential deadlocks in the original program, i.e., if the abstraction is deadlock free, then the corresponding program is also deadlock free, but not vice versa.

### *Overview*

As said, this technical paper extends the previous one [18] by "type-inference" or "type-reconstruction". In doing so we concentrate on effect-part, i.e., we ignore the underlying, standard typing part when dealing with the inference problem; that part is standard and would only notationally complicate the derivation rules without really adding to the result.

Effect reconstruction is practically motivated. For deadlock detection we use a behavioural effect type system that captures sequences of lock interactions of the program. This abstraction captured in the behavioural effects then can be used in a second stage to potentially detect deadlocks. The type and effect system is based on explicit typing, i.e., the user is required to in particular specify the expected locking behaviour when introducing variables; it is preferable not to burden the programmer with this but to *infer* that behaviour. Type inference or type reconstruction, also for effects, is a known problem and orthogonal to the problem of the proposed method of deadlock checking, which we develop in detail in [18] and which is based on a special form of simulation relation (called deadlock-sensitive simulation). Therefore, we omitted effect inference in [18] and provide details of an effect type inference algorithms in this paper.

Technically, and as usual, the key to allow effective type reconstruction is the addition of type level variables. Concentrating on the effects, it basically means effect variables, here for locks. That is not only a technical means to enable algorithmic type reconstruction (using unification), but renders also the language more expressive by allowing (universally) polymorphic functions. In our development later, we allow universally polymorphic functions, polymorphic lets, but do not cover polymorphic recursion.

As a starting point, i.e., as a specification for the algorithmic inference, we use the previous type system with the following three changes. First, of course, the type system now uses implicit typing, i.e., we switch from Curry-style typing to Church-style. Secondly, we simplify the type system in that we disallow "lock types" to denote *sets* of (abstract) locks which as a consequence does away with *subtyping* as far

as lock types are concerned. This restriction is compensated by the fact that now we allow lock polymorphic functions. Another way of understanding the change is that we replaced subtype polymorphism which we used in [18] by universal polymorphism as far as lock arguments are concerned. This restriction is also in parts technically motivated: it allows to separate the treatment of sub-type polymorphism from the treatment of universal polymorphism (see also below where we describe the development slightly more detailed). Note that here we still support subtyping as "imported" from the sub-effecting relations to the effect-annotated function types. Finally, we restrict the development to first-order functions. This restriction is technically motivated. In principle, as far as type inference is concerned, higher-order functions are unproblematic of course, after all, type inference as most commonly used today was developed in the context of the $\lambda$-calculus/functional languages. In our setting with behavioural effects for deadlock checking, we restrict to the first-order case not because of the treatment of type level variables, but because of *sub-effecting*. In the presence of sub-effecting, an algorithmic version of the type system requires to be able to calculate the *minimal* effect and, as a consequence, be able to calculate the least upper bound of, for instance, two effects. In the language for behavioural effects, which can be understood to specify traces or sequences of lock interactions, this least upper bound of two effects $\varphi_1$ and $\varphi_2$ is represented as the non-deterministic choice $\varphi_1 + \varphi_2$. Due to the contra/co-variant typing of functions, the question of a least upper bound dualizes to finding the greatest lower bound when dealing with sub-effecting on contra-variant positions. It is, however, unclear whether one could/should have a construction which corresponds to the dual for $+$ and which is useful in finding deadlocks.

The abstract effects we derive capture enough information to model dynamic thread creation, lock accesses, branching, function application, and assignments of values (e.g., of functions or locks) to variables. Note that the system we describe here only algorithmically infers (through syntax-driven rules) the abstract information—the deadlock checking itself is a separate phase and out of the scope of this article. The interested reader may refer to [18] for the deadlock checking through state space exploration, including two parametrizable abstractions to eliminate infinite behaviour. Nonetheless, we will here later establish the correctness of our inference with regard to this second phase of our analysis.

The main problem for an algorithmic treatment of the given specification of the type system is to overcome the inherent *non-determinism* of the typing rules, when interpreting them in a goal-directed manner. The two central problematic phenomena, which are treated by non-syntax directed rules, are the two forms of *polymorphism*: subtyping in the form of sub-effecting and universal polymorphism. The standard way to turn a type system into the presence of these forms of polymorphism is to let the algorithm calculate not non-deterministically an unnecessarily specific type, but to calculate, at each point, the "best possible" one in the sense of committing in the least possible way to any specific type. For that it is necessary to avoid *backtracking* which would yield at least an unpractical implementation, if not outright result in undecidability. Concerning sub-effecting, this amounts to determining the *minimal* type which in our setting is the type with the minimal effect. Concerning type variables and type schemes, the best possible type is the most general type, also known as principal type, from which all others can be obtained by substitution. As usual, *unification* is the mechanism to determine the most general type if it exists. To deal with both mechanisms, we adopt a layered approach. In a first step, we tackle the problem of sub-typing/sub-effecting which yields as an intermediate system. This system is equivalent to the specification but where the only non-syntax directed rules are the ones dealing with universal polymorphism. In a second step, we get rid of the remaining non-deterministic rules (generalization and instantiation) following standard techniques. For the formulation of soundness and in particular completeness, we follows the classic formulation of [7], which provided an inductive proof of completeness of algorithm $\mathscr{W}$ for Hindley/Milner/Damas kind of let-polymorphism (captured by type schemes).

In Section II, we introduce the syntax and semantics of our calculus. Section III presents a non-deterministic specification of a type- and effect system and the corresponding inference algorithm which reconstructs the type of an implicitly-typed concrete program, and captures the abstract behaviour of the program, and the conclusion is given in Section IV.

## II. Calculus

This section gives the syntax and semantics of the calculus with lock-based concurrency our analysis based on.

### A. Syntax

The abstract syntax for a concurrent calculus with functions, thread creation, and re-entrant locks is presented in Table I. A program $P$ consists of a parallel composition of processes $p\langle t\rangle$, where $p$ identifies the process and $t$ is a thread, i.e., the code being executed. The empty program is denoted as $\emptyset$. As usual, we assume $\|$ to be associative and commutative, with $\emptyset$ as neutral element. As for the code we distinguish threads $t$ and expressions $e$, where $t$ basically is a sequential composition of expressions. Values are denoted by $v$, and $\texttt{let}\, x{:}T = e\, \texttt{in}\, t$ represents the sequential composition of $e$ followed by $t$, where the eventual result of $e$, i.e., once evaluated to a value, is bound to the local variable $x$. Expressions, as said, are given by $e$, and threads are among possible expressions. Further expressions are function application $e_1\ e_2$, conditionals, and the spawning of a new thread, written $\texttt{spawn}\ t$. The last three expressions deal with lock handling: $\texttt{new}^{\pi}\ \texttt{L}$ creates a new lock (initially free) and gives a reference to it (the L may be seen as a class for locks), and furthermore $v.\texttt{lock}$ and $v.\texttt{unlock}$ acquires and releases a lock, respectively. Values, i.e., evaluated expressions, are variables, lock references, and function abstractions, where we use $\texttt{fun}\ f{:}T_1.x{:}\vec{T}_2.t$ for recursive function definitions. To keep track of lock creations, the corresponding expressions are annotated, where we assume an infinite reservoir of locations or labels $\pi$.

The types and the effects are given in Table II, resp. in Table V for the effects. Besides basic types for integers and booleans, the calculus supports types $\texttt{L}^r$ for lock references and function types $\vec{U} \xrightarrow{\varphi} U$, under the restriction to first-order functions. For type inference or type reconstruction, we concentrate on the part we are most interested in, namely the behaviour

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

TABLE I: Abstract syntax

part, in particular, the locations. To do so, we assume that the user provides the underlying types, i.e., without location and effect annotations and that they are not reconstructed by type inference. It would be straightforward with standard techniques, to incorporate conventional type reconstruction for the underlying types. For the current presentation, we omit that part not to clutter the presentation. So, the reconstruction just concerns the omitted locations for lock creation. In abuse of notation, we use $T$ resp. $U$ both for the annotated types from the grammar of Table I and the underlying types with the annotations stripped; which is meant to be clear from the context. Note further the simplification entailed by the restriction to first-order functions: in absence of higher-order functions, the only variables of function type are let-bound variables, but not formal parameters of functions. It means that the type reconstruction algorithm does not need to *guess* (using type-level variables) the effect annotation $\varphi$ on functional types $T_1 \xrightarrow{\varphi} T_2$; for let-bound variable, such a guess is not needed as the effect is *immediately* available upon analysis of the definition of the code bound to the variable.

Polymorphism for function definitions is captured by type-level variables. In our setting, the only type-level variables are location variables $\rho$ representing locations $\pi$. They may show up in the lock types $\mathtt{L}^{\rho}$ as well as in the effects $\varphi$ on the latent effects of functions. The universally quantified type (corresponding to type schemes) captures functions polymorphic in the locations. We abbreviate $\forall\rho_1.\ldots.\forall\rho_n.T$ by $\forall\vec{\rho}.T$.

The effects of Table V form a behavioural abstraction of the behaviour of the concurrent programs wrt. the lock usage. The behaviour language can be seen as a small process algebra, for which certain algebraic laws will hold and for which we will give a semantics in the form of operational rules as well. We give the definition of effects $\varphi$ later in Section III.

The set of free and bound variables of a type is defined as usual, where $\forall$ acts as binder; bound variables are considered up-to renaming. We write $fv(T)$ for the set of free variables in $T$. Substitutions, with typical element $\theta$, are mappings from (location) variables $\rho$ to location annotations $r$. We write $\theta T$ for the application of $\theta$ to a type $T$, replacing all free variables of $T$ according to $\theta$, with renaming of bound variables, if necessary. The domain $dom(\theta)$ of $\theta$ is defined as the set of all variables where $\theta(\rho) \neq \rho$. Concrete substitutions we write

| | | | |
|---|---|---|---|
| $U$ | ::= | $\mathtt{Bool} \mid \mathtt{Int} \mid \mathtt{Thread} \mid \mathtt{L}^{r}$ | basic types |
| $T$ | ::= | $U \mid \vec{U} \xrightarrow{\varphi} U$ | types |
| $S$ | ::= | $T \mid \forall\rho.S$ | type schemes |
| $r$ | ::= | $\rho \mid \pi$ | location annotations |

TABLE II: Types and type schemes

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

TABLE III: Local steps

in the form $[r_1/\rho_1]\ldots[r_k/\rho_k]$ or $[\vec{r}/\vec{\rho}]$.

### B. Semantics

The small-step operational semantics given below is straightforward, distinguishing between local and global steps (cf. Tables III and IV). The local level deals with execution steps of one single thread, specifying reduction steps of the form $t \to t'$. 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 other rule, which performs a case distinction of the first basic expression in a let construct, that assures a deterministic left-to-right evaluation within each thread. The two R-IF-rules cover the two branches of the conditional and the R-APP-rules deals with function application (of non-recursive, resp. recursive functions).

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

$$\sigma \vdash P \to \sigma' \vdash P' \ , \tag{1}$$

where $P$ is a program, i.e., the parallel composition of a finite number of threads running in parallel, and $\sigma$ contains the *locks*, i.e., it is a finite mapping from lock identifiers to the status of each lock (which can be either free or taken by a thread where a natural number indicates how often a thread has acquired the lock, modelling re-entrance). A thread-local step is lifted to the global level by R-LIFT. Rule R-PAR specifies that the steps of a program consist of the steps of the individual threads, sharing $\sigma$. Executing the spawn-expression creates a new thread with a fresh identity which runs in parallel with the parent thread (cf. rule R-SPAWN). Globally, the process identifiers are unique; for $P_1$ and $P_2$ to be composed in parallel, the $\parallel$-operator requires $dom(P_1)$ and $dom(P_2)$ to be disjoint, which assures global uniqueness. A new lock is created by `newL` (cf. rule R-NEWL) which allocates a fresh lock reference in the heap. Initially, the lock is free. A lock $l$ is acquired by executing $l.\mathtt{lock}$. There are two situations where that command does not block, namely the lock is free or it is already held by the requesting process $p$. The heap update $\sigma +_p l$ is defined as follows: If $\sigma(l) = \mathit{free}$, then $\sigma +_p l = \sigma[l \mapsto p(1)]$ and if $\sigma(l) = p(n)$, then $\sigma +_p l = \sigma[l \mapsto p(n+1)]$. The notation $p(n)$ indicates that the pair of process identifier $p$ and lock count $n$. 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 premise check that the thread performing the unlocking actually holds the lock.

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

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

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

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

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

TABLE IV: Global steps

## III. TYPE SYSTEM

The type and effect system for expressions is given in Table VIII. The judgments for expressions are of the form

$$\Gamma \vdash e : T :: \varphi \tag{2}$$

where the typing contexts $\Gamma = x_1{:}T_1, \ldots, x_n{:}T_n$ associate (annotated) types to variables. All variables $x_i$ are assumed different, so $\Gamma$ can be seen as a finite mapping; we write $\Gamma(x)$ for the type of $x$ in $\Gamma$ and $dom(\Gamma)$ for the set of variable typed in $\Gamma$.

### A. Effects

In specifying the syntax, we postponed the exact definition of the effects, which played no role in formulating the operational semantics. We do that next (see Table V), before we can present the rules of the type and effect system in Section III-C.

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

TABLE V: Effects

The effects are split between a global level $\Phi$ and a (thread-) local level $\varphi$. The empty effect $\varepsilon$ represents behaviour without lock operations. Sequential composition and nondeterministic choice are represented by $\varphi_1;\varphi_2$ and $\varphi_1+\varphi_2$, respectively. Recursive behaviour is introduced through $\mathit{rec}X.\varphi$, where $\mathit{rec}X$ binds the recursion variable in $\varphi$. Recursion *is not polymorphic*, i.e., location variables in the effect depend entirely on the types that introduce them.

Labels $a$ capture four basic effects: $\texttt{spawn}\,\varphi$ represents the effect of creating a new process with behaviour $\varphi$, while $\nu\mathrm{L}^r$ means the effect of creating a new lock at program point $r$. The effects of lock manipulations are captured by $r.\texttt{lock}$ and $r.\texttt{unlock}$, meaning acquiring a lock and releasing a lock, respectively, where $r$ is again referring to the point of creation. $\tau$ is used to label silent transitions.

For instance, the effect

$$\nu\mathrm{L}^{\pi};\ \mathit{rec}X.(\pi.\mathit{lock};\ \texttt{spawn}\,\varphi;\ \pi.\mathit{unlock};\ X)$$

represents the behaviour of a thread which first creates a lock at location $\pi$ and then enters a recursive definition which takes a lock created at $\pi$, spawns a new thread with some behavioural effect $\varphi$, releases the lock, and invokes itself (the recursive behaviour) again.

### B. Sub-effecting

The behaviour of an effect expression describes its possible traces to over-approximate the actual behaviour. The effects are *ordered* and the corresponding *sub-effect* relation is formalized in Table VII. Sub-effecting, i.e., the order on effects, leads to an order $\sqsubseteq$ on types, in particular on function types.

*Definition 3.1 (Subtyping and sub-effecting):* The binary relations $\equiv$ (equivalence) and $\sqsubseteq$ (sub-effecting) on effects are given inductively by the rules of Table VII. In abuse of notation, the *subtyping* relation types, relative to a given context $\Gamma$, is written $S_1 \leq S_2$, as well and given in Table VI. Furthermore, we define $\vee$ by induction on types: $S \xrightarrow{\varphi_1} T \vee S \xrightarrow{\varphi_2} T$ is defined as $S \xrightarrow{\varphi_1+\varphi_2} T$. Else $\forall\rho.S \vee \forall\rho.T = \forall\rho.S \vee T$. Else $T \vee T$ equals $T$ and $\vee$ is undefined otherwise.

Sequential composition is associative, with $\varepsilon$ as neutral element (cf. rules E-UNIT and EE-ASSOC$_s$). Nondeterministic choice is commutative, associative, and distributes over sequential composition (cf. rule EE-COMM, EE-ASSOC$_c$, and EE-DISTR). Idempotence of choice is described by EE-CHOICE. Sub-effecting is reflexive (modulo $\equiv$) and transitive. Note that sequential composition and choice are "monotone" wrt. $\sqsubseteq$ (by SE-CHOICE$_2$ and SE-SEQ) and the premise of SE-CHOICE$_1$ makes sure that all variables occurring free in the effect are covered by the context (for the other sub-effecting rules, that is assured by induction). Monotonicity of the spawn-construct and recursion is covered by SE-SPAWN and SE-REC. As for subtyping: Rule SE-REFL expresses the reflexivity of the order. The order of two effects is lifted to function types and universally quantified types in rule S-ARROW and S-ALL. Transitivity of subtyping is straightforward.

$$\Gamma \vdash T \leq T \quad \text{S-REFL} \qquad \frac{\Gamma \vdash \varphi \sqsubseteq \varphi'}{\Gamma \vdash T_1 \xrightarrow{\varphi} T_2 \leq T_1 \xrightarrow{\varphi'} T_2}\ \text{S-ARROW}$$

$$\frac{\Gamma \vdash S_1 \leq S_2}{\Gamma \vdash \forall\rho.S_1 \leq \forall\rho.S_2}\ \text{S-ALL}$$

TABLE VI: Subtyping

### C. Typing rules

The rules of the type system are given in Table VIII. Variables, thread names, and lock references are all values and thus have no effect (cf. rules T-VAR, T-PREF, and T-LREF). Both branches of a conditional must agree on their

$rec X.\varphi \equiv [rec X.\varphi/X]\varphi$ EE-REC

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

$\varphi+\varphi \equiv \varphi$ EE-CHOICE $\quad (\varphi_1+\varphi_2);\varphi_3 \equiv \varphi_1;\varphi_3+\varphi_2;\varphi_3$ EE-DISTR

$\varphi_1+\varphi_2 \equiv \varphi_2+\varphi_1$ EE-COMM $\quad \varphi_1+(\varphi_2+\varphi_3) \equiv (\varphi_1+\varphi_2)+\varphi_3$ EE-ASSOC$_C$

$$\frac{\varphi_1 \equiv \varphi_2}{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_2}\ \text{SE-REFL} \qquad \frac{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_2 \quad \Gamma \vdash \varphi_2 \sqsubseteq \varphi_3}{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_3}\ \text{SE-TRANS}$$

$$\frac{fv(\varphi_2) \subseteq fv_T(\Gamma)}{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_1+\varphi_2}\ \text{SE-CHOICE}_1 \qquad \frac{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_1' \quad \Gamma \vdash \varphi_2 \sqsubseteq \varphi_2'}{\Gamma \vdash \varphi_1+\varphi_2 \sqsubseteq \varphi_1'+\varphi_2'}\ \text{SE-CHOICE}_2$$

$$\frac{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_1' \quad \Gamma \vdash \varphi_2 \sqsubseteq \varphi_2'}{\Gamma \vdash \varphi_1;\varphi_2 \sqsubseteq \varphi_1';\varphi_2'}\ \text{SE-SEQ} \qquad \frac{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_2}{\Gamma \vdash \texttt{spawn}\ \varphi_1 \sqsubseteq \texttt{spawn}\ \varphi_2}\ \text{SE-SPAWN}$$

$$\frac{\Gamma \vdash \varphi_1 \sqsubseteq \varphi_2}{\Gamma \vdash rec X.\varphi_1 \sqsubseteq rec X.\varphi_2}\ \text{SE-REC}$$

TABLE VII: Sub-effecting

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

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

$$\frac{\Gamma \vdash e_1 : S_1 :: \varphi_1 \quad \lfloor S_1 \rfloor = T_1 \quad \Gamma, x{:}S_1 \vdash e_2 : T_2 :: \varphi_2}{\Gamma \vdash \texttt{let}\ x{:}T_1 = e_1\ \texttt{in}\ e_2 : T_2 :: \varphi_1;\varphi_2}\ \text{T-LET}$$

$$\frac{\lfloor \vec{T} \rfloor = \vec{T_1} \quad \Gamma, \vec{x}{:}\vec{T} \vdash e : T_2 :: \varphi}{\Gamma \vdash \texttt{fn}\ \vec{x}{:}\vec{T_1}.e : \vec{T} \xrightarrow{\varphi} T_2 :: \varepsilon}\ \text{T-ABS}$$

$$\frac{\lfloor \vec{T_1'} \rfloor = \vec{T_1} \quad \lfloor T_2' \rfloor = T_2 \quad \Gamma, f{:}\vec{T_1'} \xrightarrow{X} T_2', \vec{x}{:}\vec{T_1'} \vdash e : T_2' :: \varphi}{\Gamma \vdash \texttt{fun}\ f{:}\vec{T_1} \to T_2.\vec{x}{:}\vec{T_1}.e : \vec{T_1'} \xrightarrow{rec X.\varphi} T_2' :: \varepsilon}\ \text{T-ABS}_{rec}$$

$$\frac{\Gamma \vdash v_1 : \vec{T_2} \xrightarrow{\varphi} T :: \varepsilon \quad \Gamma \vdash \vec{v_2} : \vec{T_2} :: \varepsilon}{\Gamma \vdash v_1\ \vec{v_2} : T :: \varphi}\ \text{T-APP}$$

$$\frac{\Gamma \vdash e : S' :: \varphi' \quad \Gamma \vdash S' \leq S \quad \Gamma \vdash \varphi' \sqsubseteq \varphi}{\Gamma \vdash e : S :: \varphi}\ \text{T-SUB}$$

$$\frac{\Gamma \vdash e : S :: \varphi \quad \rho \notin fv_T(\Gamma) \quad \rho \in fv_T(S)}{\Gamma \vdash e : \forall \rho.S :: \varphi}\ \text{T-GEN}$$

$$\frac{\Gamma \vdash e : \forall \rho.S :: \varphi \quad \rho = dom(\theta)}{\Gamma \vdash e : \theta S :: \varphi}\ \text{T-INST}$$

TABLE VIII: Type and effect system

type and their effect (cf. rule T-COND). The let-construct generalizes sequential composition and its effect $\varphi_1;\varphi_2$ is the sequential composition of the effects of the constituent parts (cf. rule T-LET). Note further that the type $T_1$ given by the user for the variable $x$ is from the underlying types, i.e., without annotations, whereas the result of analyzing $e_1$ may be annotated (with locations and effects). So the user-given $T_1$ must correspond to the annotated type scheme $S_1$ with all annotations stripped, written $\lfloor S_1 \rfloor$. Note that the operator $\lfloor \_ \rfloor$ applies also to quantified types by stripping the quantifier and the related bound variables. The analysis of the body $e_2$ of the let-construct continues assuming the annotated type scheme $S_1$ for the local variable, not the underlying, declared type. Abstractions are considered as values and therefore have empty effect (cf. rule T-ABS). Similar to the let-construct, the types in $\vec{T_1}$ given by the user for the variables $\vec{x}$ are without annotations, and must correspond to the annotated $\vec{T}$. The analysis of the function body is based on the assumption of the annotated type $\vec{T}$ for the input parameters. The effect of the body in the premise is the *latent* effect of the overall function, and is annotated on the function type of the abstraction in the conclusion of the rule. Note that for the recursive function (cf. rule T-ABS$_{rec}$), the function type $\vec{T_1} \to T_2$ given by the user for the function $f$ corresponds to the annotated type $\vec{T_1}' \to T_2'$ whose latent effect is guessed as the recursive effect variable $X$ in the context of the premise. The overall type of the recursive function is the function type annotated with the recursive effect of the function body $rec X.\varphi$. In the rule T-APP, the function as well as the arguments in an application are considered as values and have empty effect. Therefore, the overall effect is the latent effect of the function body.

The spawn-statement is of type `Thread`, provided the spawned expression is well-typed (cf. rule T-SPAWN) and the effect just expresses that the effect $\varphi$ of the body $t$ is executed by a new thread. The treatment of creating a new lock at location $\pi$ is straightforward: the expression is of (annotated) type $\mathrm{L}^\pi$ and of effect $\nu\mathrm{L}^\pi$. The non-syntax-directed T-SUB is a standard rule of subsumption, allowing to relax types and effects (cf. also Definition 3.1). The two remaining, dual rules of generalization and specialization or instantiation introduce, resp. eliminate polymorphic types (cf. rules T-GEN and T-INST). As usual, to introduce a universally-quantified type, the typing of the corresponding expression must not depend (via $\Gamma$) on the variable (here $\rho$) being quantified over. We also assume that we only quantify over variables actually occurring in the type. The function $fv_T(\_)$ in the premise of the generalization rule returns the free variables which occur in the types, but *not* in latent effects.

### D. Algorithmic formulation

Next we turn the type and effect system of Section III into an algorithm. The reason, why the type system in itself is non-algorithmic are the non-syntax-directed rules of Table VIII which are the rules dealing with *polymorphism* of the type system. The system supports two forms of polymorphism, subtype polymorphism (or rather sub-effecting) captured by the subsumption rule T-SUB and universal polymorphism (in the form of let-polymorphism for location variables) captured by the generalization and instantiation rules T-GEN and T-INST. To obtain an algorithm, those rules need to be replaced by syntax-directed counter-parts; for subtype polymorphism, the algorithm must calculate the most specific type, i.e., minimal type wrt. the subtype/sub-effecting order. For the universal polymorphism, the most general type needs to be determined. As usual, unification is the key for that.

The definition of unification is standard, ignoring the latent effects for the arrow types in the unification. Instead of unifying the latent effects $\varphi_1$ and $\varphi_2$ of two arrow types, the choice $\varphi_1+\varphi_2$ will be used in the algorithmic rule for handling conditionals, over-approximating both $\varphi_1$ and $\varphi_2$. Other consumers of unification (application and abstraction) do not have to consider arrow types due to our restriction to first order.

We write $T = T_1 \wedge_\theta T_2$ if $\theta$ is an mgu of $T_1$ and $T_2$ and $T = \theta T_2 (= \theta T_2)$ (cf. also [17] for details).

As mentioned, the generalization rule T-GEN allows to introduce universal quantification over a variable given the variable is not mentioned in the typing context. For the algorithm, the following closure operation is needed, which generalizes over all variables, to which T-GEN applies.

*Definition 3.2 (Closure):* The closure of a type $T$ with respect to context $\Gamma$ is given as $close_\Gamma(T) = \forall \vec{\rho}.T$, where $\vec{\rho} = fv(T) \setminus fv(\Gamma)$.

*Definition 3.3 (Instantiation with fresh variables):* Assume $\Gamma \vdash e : T :: \varphi$. We define a fresh instance of $S$ (wrt. $\Gamma$) by replacing all universally quantified variables of $S$ by fresh ones. I.e., for $S = \forall \vec{\rho}.T$, then $INST_\Gamma(S) = \theta T$, where $\theta = [\vec{\rho}/\vec{\rho}']$ where $\vec{\rho}'$ are fresh variables.

The judgements of the type reconstruction algorithm

$$\Gamma \vdash_A e : T :: \varphi, \theta \tag{3}$$

are interpreted as follows: under the typing context $\Gamma$, expression $e$ is of type $T$ and with effect $\varphi$, under a substitution $\theta$. Substitutions $\theta$ are finite and partial mappings from location variables $\rho$ to location annotations $r$. We assume that the type $T$ and effect $\varphi$ already have the substitution $\theta$ applied. As for the bindings in the typing context, variables are bound to *type schemes* (as in the type system). The rules for the algorithm are given in Table IX.

The type of a variable is looked up from the context $\Gamma$ and a variable introduces no constraints, i.e., the substitution is the identity *id*. As all values, variables have no effect (cf. rule TA-VAR. Similarly the treatment of thread names and lock references in rules TA-THREAD and TA-LREF). Note that $S$, as fetched from the typing environment, may be a type scheme, whereas the instance $T$ is a type (cf. Definition 3.3). As values, also abstractions have no effect (cf. rule TA-ABS). The types $\vec{T}_1$ given by the user are without annotations. The operator $\lceil\_\rceil_A$ used in the premise of the rule annotates $T_1$ with *fresh* variables, and the function body is checked in the premise under the assumption of the thus annotated type $\vec{T}_1'$ of the input parameters $\vec{x}$. The latent effect of the function is the effect of the function body. The substitution obtained from analyzing the function body is propagated in the conclusion, with the fresh variables from the user-provided type removed, as they are local to the body. Recursive function definitions in TA-ABS$_{rec}$ use an additional fresh recursion variable $X$ in place for the latent effect of a recursive invocation, which is bound when entering the recursion. Note that we do not allow polymorphic recursion.

For applications (cf. rule TA-APP), the function and the arguments are already evaluated, and therefore both abstraction and argument have empty effect and identity substitution. The type of the abstraction is a function type annotated with the latent effect. We use unification to ensure that the type of the argument is equal to the argument type of the abstraction. The overall type and effect of the application are respectively the output type and the latent effect of the abstraction which are specified by the unification substitution. The definition of conditionals (cf. rule TA-COND) first ensures the types of the two branches are equal modulo latent effects with unification. The overall type and effect are the least upper bound of the two types resp. two effects from the two branches. The sequential composition of two expressions is constructed by the rule TA-LET. The closure of the type of the `let`-bound variable is added to the context to give polymorphism. The overall effect $\varphi_1; \varphi_2$ is the sequential composition of the expressions. The remaining rules TA-SPAWN, TA-NEWL, TA-LOCK and TA-UNLOCK are straightforward.

$$\frac{\Gamma(x) = S}{\Gamma \vdash_A x : INST_\Gamma(S) :: \varepsilon, id}\ \text{TA-VAR}$$

$$\frac{}{\Gamma \vdash_A p : \texttt{Thread} :: \varepsilon, id}\ \text{TA-PREF} \qquad \frac{}{\Gamma \vdash_A l^\pi : \mathrm{L}^\pi :: \varepsilon, id}\ \text{TA-LREF}$$

$$\frac{\vec{T}_1' = \lceil \vec{T}_1 \rceil_A \qquad \Gamma, \vec{x} : \vec{T}_1' \vdash_A e : T_2 :: \varphi, \theta \qquad \vec{\rho} = fv(\vec{T}_1')}{\Gamma \vdash_A \texttt{fn}\, \vec{x} : \vec{T}_1.e : (\theta \vec{T}_1') \xrightarrow{\varphi} T_2 :: \varepsilon, \theta \setminus \vec{\rho}}\ \text{TA-ABS}$$

$$\frac{\vec{T}_1' = \lceil \vec{T}_1 \rceil_A \quad T_2' = \lceil T_2 \rceil_A \quad \Gamma, f : \vec{T}_1' \xrightarrow{X} T_2', x : \vec{T}_1' \vdash_A e : T :: \varphi, \theta \quad X\ fresh \quad \lfloor T \rfloor = T_2 \quad \vec{\rho} = fv(\vec{T}_1' \to T_2') \quad \theta_1 = \mathscr{U}(T, \theta T_2')}{\Gamma \vdash_A \texttt{fun}\, f : \vec{T}_1 \to T_2.x : \vec{T}_1.e : (\theta_1 \theta \vec{T}_1') \xrightarrow{\theta_1 (rec\, X.\varphi)} (\theta_1 \theta T_2') :: \varepsilon, \theta_1 \circ \theta \setminus \vec{\rho}}\ \text{TA-ABS}_{rec}$$

$$\frac{\Gamma \vdash_A v_1 : \vec{T}_2' \xrightarrow{\varphi} T' :: \varepsilon, id \qquad \Gamma \vdash_A v_2 : \vec{T}_2, \varepsilon, id \qquad \theta = \mathscr{U}(\vec{T}_2', \vec{T}_2)}{\Gamma \vdash_A v_1\ v_2 : \theta T' :: \theta\varphi, \theta}\ \text{TA-APP}$$

$$\frac{\Gamma \vdash_A v : \texttt{Bool} :: \varepsilon, id \qquad \Gamma \vdash_A e_1 : T_1 :: \varphi_1, \theta_1 \qquad \theta_1 \Gamma \vdash_A e_2 : T_2 :: \varphi_2, \theta_2 \qquad \theta_3 = \mathscr{U}(\theta_2\ T_1, T_2)}{\Gamma \vdash_A \texttt{if}\, v\, \texttt{then}\, e_1\, \texttt{else}\, e_2 : \theta_3\theta_2 T_1 \vee \theta_3 T_2 :: \theta_3\theta_2\varphi_1 + \theta_3\varphi_2,\ \theta_3 \circ \theta_2 \circ \theta_1}\ \text{TA-COND}$$

$$\frac{S_1 = close_{\theta_1\Gamma}(T_1') \qquad \lfloor S_1 \rfloor = T_1 \qquad \Gamma \vdash_A e_1 : T_1' :: \varphi_1, \theta_1 \qquad \theta_1 \Gamma, x : S_1 \vdash_A e_2 : T_2 :: \varphi_2, \theta_2}{\Gamma \vdash_A \texttt{let}\, x : T_1 = e_1\, \texttt{in}\, e_2 : T_2 :: \theta_2\varphi_1; \varphi_2, \theta_2 \circ \theta_1}\ \text{TA-LET}$$

$$\frac{\Gamma \vdash_A e : T :: \varphi, \theta}{\Gamma \vdash_A \texttt{spawn}\, e : \texttt{Thread} :: \texttt{spawn}\, \varphi, \theta}\ \text{TA-SPAWN} \qquad \frac{}{\Gamma \vdash_A \texttt{new}_\pi\, L : \mathrm{L}^\pi :: \nu \mathrm{L}^\pi, id}\ \text{TA-NEWL}$$

$$\frac{\Gamma \vdash_A v : \mathrm{L}^r :: \varepsilon, id}{\Gamma \vdash_A v.\, \texttt{lock} : \mathrm{L}^r :: r.\texttt{lock}, id}\ \text{TA-LOCK} \qquad \frac{\Gamma \vdash_A v : \mathrm{L}^r :: \varepsilon, id}{\Gamma \vdash_A v.\, \texttt{unlock} : \mathrm{L}^r :: r.\texttt{unlock}, id}\ \text{TA-UNLOCK}$$

TABLE IX: Algorithmic effect inference

### E. Soundness and completeness

Next we prove soundness and completeness of the type reconstruction algorithm from Table IX wrt. its specification (cf. Table VIII). We start with a few technical properties, mainly about the subtyping/sub-effecting relation, substitutions, and instantiation, which are needed for establishing those results.

*1) Soundness:* As usual, soundness is straightforward, whereas completeness later requires a careful formulation of the relationship between specification and algorithm to allow an inductive proof.

*Lemma 3.4 (Soundness):* If $\Gamma \vdash_A e : T :: \varphi, \theta$, then $\theta\Gamma \vdash e : T :: \varphi$.

*Proof:* By straightforward induction on the derivation by the rules of Table IX. ∎

*2) Completeness:* For completeness in the inverse direction we need to prove in principle that all typing judgments derivable by the specification are found by the algorithm as well. As the algorithm is deterministic whereas the specification is not, not all typings are literally given by the algorithm. Instead, and as usual, the algorithm gives back a type and effect

that represents all possible typings from the specification. Such type and effect are considered to be the "best" wrt. subtyping/sub-effecting. Unification is used to synthesize the "best" type and effect in the sense of the most general. To express that order, we introduce the following definitions.

*Definition 3.5 (Instantiation and ordering):* We write $S_1 \lesssim_\theta S_2$ for $S_1 = \theta S_2$, and $S_1 \lesssim S_2$ if $S_1 \lesssim_\theta S_2$, for some $\theta$. In abuse of notation we use the same notation for the order on substitutions, i.e., $\theta_1 \lesssim_\theta \theta_2$ if $\theta_1 = \theta \circ \theta_2$ and $\theta_1 \lesssim \theta_2$, if $\theta_1 \lesssim_\theta \theta_2$, for some $\theta$. $\forall\vec{\rho}.S$ is a *generic* instance of $\forall\vec{\rho}'.S'$, written $\forall\vec{\rho}.S \lesssim^g \forall\vec{\rho}'.S'$, iff $S = [T_i/\rho_i']S'$ for some types $T_i$, and $\rho_j$ does not occur free in $\forall\vec{\rho}'.S'$.

*Definition 3.6 (Minimal typing):* $S$ and $\varphi$ are *minimal* for $\Gamma$ and $e$, if $\Gamma \vdash e : S :: \varphi$, and furthermore $\Gamma \vdash e : S' :: \varphi'$ implies $S \leq S'$ and $\varphi \sqsubseteq \varphi'$ (for all $S'$ and $\varphi'$).

*Theorem 3.7 (Completeness):* If $\Gamma \lesssim \Gamma'$ and $\Gamma \vdash e : S :: \varphi$, then

1) $\Gamma' \vdash_A e : T' :: \varphi', \theta'$
2) There exists a $\theta$ s.t.
   a) $\Gamma \lesssim_\theta \theta'\Gamma'$
   b) $S' \lesssim^g \theta\ close_{\theta'\Gamma'}(T')$ and $S' \leq S$
   c) $\theta'\varphi' \sqsubseteq \varphi$

   where $S'$ and $\theta'\varphi'$ are minimal for $\Gamma$ and $e$.

*Proof:* The proof can be developed in two levels. As an intermediate step, we first *remove* the subsumption rule from Table VIII and "build in" the weakening by subsumption into those rules where it is needed. We prove this intermediate system derives the minimal type and effect. The second step in the completeness proof shows that the algorithm infers the most general type and effect wrt. the intermediate system. The complete proof can be found in the research report [17]. ■

*3) Relation with deadlock checking:* Let us finish with a few words relating the effects that are inferred by the algorithm with potential deadlocks in the program: the abstract overall effect of a program needs to be further analysed to find potential deadlocks. To this end, we have developed an analysis based on state space exploration that symbolically executes the abstract program (cf. [18]). Since the state space of the abstract program is in general still infinite, the analysis allows the user to parametrize the bounds of lock counters and recursion depth. Any behaviour exceeding those bounds is soundly over-approximated.

To establish the validity of our claim that the inferred effects soundly capture potential deadlocks, we need to combine straightforward subject reduction between the specification of the type system and the operational semantics (augmented with suitable transition labels, cf. again [18]) with the soundness between algorithm and type system.

## IV. Conclusion

We have presented an algorithmic inference type- and effect system to reconstruct the type wrt. lock locations for an implicitly-typed program, and derives the abstract behaviour for a calculus supporting multi-threading concurrency, functions, and re-entrant locks. We have proven the algorithm is sound and complete with regard to a non-deterministic specification of the type system. The correctness of the abstraction with regard to preserving potential deadlocks which occur in the original program can be shown analogously to the one in [18] through the combination of subject reduction and our soundness result. The reverse does not hold, i.e., a deadlock in the abstraction does not necessarily exist in the concrete program, as the abstraction over-approximates the actual behaviour of the program.

### *Related work*

Deadlocks are a common problem in concurrent programming. Numerous techniques have been investigated to detect deadlocks at both compile-time and runtime, and implemented for different languages [9], [11], [14]. Various analysis techniques for detecting deadlocks in Ada programs have been evaluated earlier in [6].

The most common way to prevent deadlocks is to make sure that cyclic waiting on resources, locks in our case, does not occur. One typical approach is to impose a certain partial order on lock taking to avoid cyclic wait on locks. For instance, Boyapati et al. [3] formalise this technique in the form of deadlock types, which are ordered to enforce an order-consistent lock acquisition policy. The approach also allows type inference and polymorphism wrt. the lock levels to enhance precision of the analysis. Similarly, Suenaga [19] and Vasconcelos et al. [20] propose type inference algorithms which guarantee well-typed programs are deadlock free by imposing a strict partial order on lock taking. Deadlock types are also used in [1] to reduce the overhead of runtime checking. A static type system is developed in this approach to infer deadlock types. Only those locks which are identified are of such types will be checked at runtime. Instead of using a fixed global order on lock acquisition, our approach detects deadlocks by exploring the state space of the abstract behaviour of a program. Naik et al. [15] present a similar approach which abstracts threads and locks according to their allocation sites, and uses model-checking techniques to detect potential deadlocks. Their approach is neither sound nor complete.

Static analyses, as well as type systems, are also investigated to analyse deadlocks caused by communication over channels and message passings. For example, Kobayashi [12], [13] presents a deadlock-freedom analysis for $\pi$-calculus by using a constraint-based type inference algorithm to encode the abstract the usage information into channels. Giachino et al. [10] verify deadlock freedom for programs using asynchronous message passing, by detecting cyclic dependency in the model of programs. The model is derived from behavioural types which carry dependency information.

### *Future work*

The type and effect system presented in this paper derives abstract behaviour of all possible executions of a program to detect deadlocks. We consider to analyse other concurrency-related errors, e.g., race conditions and starvation, with this behavioural information. While we focus on concurrent programs with lock-based synchronization in this paper, it would be useful to study how our analysis can be applied to other communication models, for instance, transaction-based models and actor models. One practical extension is to apply our

analysis to object-oriented languages like Java. One approach is to translate the syntax of the languages into that of the calculus in this paper. A natural extension is to implement the type and effect inference system as well as the state exploration for the abstract behaviour. In addition, we plan to adapt the prototype implementation of the state-exploration part developed for the monomorphic setting in [18] to the more general setting presented in this paper.

For lack of space, all proofs have been omitted in the paper. Further technical details and proofs can be found in the accompanying research report [17].

*Acknowledgements* We thank Axel Simon for fruitful discussion and making available his copy of [7].

## 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] C. Boyapati, A. Salcianu, W. Beebee, and M. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation (PLDI) (San Diego, California)*. ACM, June 2003.

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

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

[7] L. Damas. *Type Assignment in Programming Languages*. PhD thesis, Laboratory for Foundations of Computer Science, University of Edinburgh, 1985. CST-33-85.

[8] 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.

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

[10] E. Giachino and C. Laneve. Deadlock detection in linear recursive programs. *CoRR*, abs/1310.7449, 2013.

[11] K. Havelund. Using runtime analysis to guide model checking of Java programs. In K. Havelund, J. Penix, and W. Visser, editors, *Proceedings of the Spin 2000 Workshop in Model Checking of Software*, 2000.

[12] N. Kobayashi. Type-based information flow analysis for the $\pi$-calculus. *Acta Informatica*, 42(4-5):291–347, 2005.

[13] N. Kobayashi. A new type system for deadlock-free processes. In *Proceedings of CONCUR 2006*, volume 4137 of *Lecture Notes in Computer Science*, pages 233–247. Springer-Verlag, 2006.

[14] P. Müller and R. K. Leino. A basis for verifying multi-threaded programs. In *Proceedings of Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2009*, volume 5502 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

[15] 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.

[16] F. Nielson, H.-R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[17] K. I. Pun, M. Steffen, and V. Stolz. Behaviour inference for deadlock checking. Technical report 416, University of Oslo, Dept. of Informatics, July 2012.

[18] 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.

[19] 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.

[20] 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.