# Polymorphic behavioural lock effects for deadlock checking

Ka I Pun, Martin Steffen, Volker Stolz
Department of Informatics, University of Oslo, Norway

Deadlocks are a common problem for concurrent programs, in particular where multiple threads are accessing shared mutually exclusive resources synchronized by locks. 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 exists 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 [4].

One common way to prevent deadlocks is to statically ensure that such cycles on locks or resources in general can never occur. This can be achieved by arranging shared resources in some partial order and enforcing that the resources are accessed in accordance with that order. This idea has, e.g., been formalized in a type-theoretic setting in the form of deadlock types [3]. The static system presented in [3] supports also type *inference* (and besides deadlocks, prevents race conditions, as well). Deadlock types are also used in [1], but not for static deadlock prevention, but for improving the efficiency for deadlock avoidance at run-time.

In contrast, we use a behavioural type and effect system [2, 6] to capture lock interaction and use that behavioural description to explore an abstraction of the state space to detect potential deadlocks. The effects of our system express the relevant behaviour of a concurrent program with regard to reentrant 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. In our previous work [7], we define a type and effect system formalizing the sketched approach. The system presented there has two important restrictions: first of all, it is *explicitly* typed, which forces the user to declare functions by specifying its expected lock behaviour (in terms of the function's effect). Putting that burden on programmers is clearly unwelcome. Secondly, based on sub-effecting as the only form of polymorphism, the formalization suffers from a lack of precision and therefore reports more spurious deadlocks than necessary. To improve the precision, we propose in this paper a lock-polymorphic extension of that work, which addresses the two mentioned weaknesses. The formulation also can serve as the specification for type and effect inference system.

## Parametric lock effects

We use a behavioural type and effect system to capture the interaction of *shared locks*. Characterizing the behaviour of each thread in a program as sequences of lock interactions allows detecting the symptom of deadlocks, i.e. waiting for shared locks in a cyclic chain.

The grammar of the effects which we use to abstractly represent the behaviour of a simple concurrent calculus with reentrant locks is presented in Fig. 1. To track which locks are actually handled in the interactions, we annotate each lock with the corresponding program point $\pi$ of its creation to specify which lock is referring to at static time. As we focus on detecting deadlocks due to shared locks, we improve the precision by introducing location variables, $\rho$, representing lock locations. Effects can be either global in a program, or local in one single thread. For the effect construct, $\|$ represents multiple threads running in parallel globally, while semicolon represents sequential composition and + a choice among effects. The behaviour of function abstraction and recursive function is parametrized by the location variable $\rho$. The behaviour of lock handling: creating, locking and releasing a lock, is represented by $\nu \mathrm{L}^{r}, \mathrm{L}^{r}.\texttt{lock}$, and $\mathrm{L}^{r}.\texttt{unlock}$, respectively.

$$
\begin{array}{rcll}
\Phi & ::= & \mathbf{0} \mid p\langle\varphi\rangle \mid \Phi \parallel \Phi & \text{effects (global)} \\
\varphi & ::= & \varepsilon \mid \varphi;\varphi \mid \varphi+\varphi \mid ee(\vec{r}) \mid \alpha & \text{effects (local)} \\
ee & ::= & X \mid \lambda\vec{\rho}.\varphi \mid recX(\vec{\rho}).\varphi & \text{parametric behavior} \\
a & ::= & \texttt{spawn}\ \varphi \mid \nu \mathrm{L}^r \mid \mathrm{L}^r.\texttt{lock} \mid \mathrm{L}^r.\texttt{unlock} & \text{labels/basic effects} \\
\alpha & ::= & a \mid \tau & \text{transition labels} \\
r & ::= & \pi \mid \rho & \text{location annotations}
\end{array}
$$

Figure 1: Types and effects

## A behavioural type and effect system

The type and effect system uses judgments of the form $\Gamma \vdash e : T :: \varphi$, which is read as: under the environment $\Gamma$, expression $e$ has type $T$ and effect $\varphi$. Three typical rules of system are sketched in Fig. 2. They deal with thread creation as well as interaction with an existing lock, where $\mathrm{L}^r$ represents a lock which is created at $r$, for $r$ is either a program point or a location variable.

$$\frac{\Gamma \vdash e : T :: \varphi}{\Gamma \vdash \texttt{spawn}\ e : \texttt{Thread} :: \texttt{spawn}\ \varphi}\ \text{TE-SPAWN}$$

$$\frac{\Gamma \vdash v : \mathrm{L}^r :: \varphi}{\Gamma \vdash v.\,\texttt{lock} : \mathrm{L}^r :: \varphi;\mathrm{L}^r.\,\texttt{lock}}\ \text{TE-LOCK}$$

$$\frac{\Gamma \vdash v : \mathrm{L}^r :: \varphi}{\Gamma \vdash v.\,\texttt{unlock} : \mathrm{L}^r :: \varphi;\mathrm{L}^r.\texttt{unlock}}\ \text{TE-UNLOCK}$$

Figure 2: Type and Effect System

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

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

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

Figure 3: Operational semantics for effects

The effect system describes the behaviour of a program in terms of sequences of lock interactions among parallel processes. We detect deadlocks by executing the abstraction of the actual behaviour and spotting processes waiting for shared locks in a circular chain. The analysis of the abstract behaviour easily leads to state space explosion as different interleavings of the threads must be considered. Three rules of the operational semantics for effects corresponding to the typing rules in Fig. 2 are sketched in Fig. 3. The notation $\xrightarrow{p\langle\varphi\rangle}$ means that a step with effect $\varphi$ of a thread $p$ is executed. To tackle infinite executions through recursion which may lead to an infinite reachable state space, we place an upper bound on lock counters which are used to keep track of how often a re-entrant lock has been taken by the same thread. In addition, we bound non-tail recursive function calls by putting a similar limit on the recursion depth; for details, see [7]. Beyond that chosen limit, the behaviour is over-approximated by arbitrary, chaotic behaviour. The state space of this abstraction is finite and therefore allows exhaustive search for deadlocks. We furthermore define the notion of *deadlock and termination sensitive simulation* [5] to show that the behaviour of a program has been correctly captured in the abstraction.

## Current Research Results

With the proposed specification of type and effect system, we can automatically check for deadlocks in the five dining philosophers in around 2.5 minutes with 82269 states. Our approach correctly detects the deadlock situation in the original program without reporting any false positives. Also, our approach correctly certifies the amended version of the dining philosophers where one of the philosophers will always pick the right fork first as safe. We prove the correctness of the abstraction with regard to this

simulation of the original program, i.e., if an abstraction is deadlock free, then its original program must be deadlock free, but not vice versa: a deadlock in the abstraction, as an over-approximation, does not necessarily exist in the concrete program.

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

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

[5] R. Milner. An algebraic definition of simulation between programs. In *Proceedings of the Second International Joint Conference on Artificial Intelligence*, pages 481–489. William Kaufmann, 1971.

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

[7] K. I. Pun, M. Steffen, and V. Stolz. Deadlock checking by a behavioral effect system for lock handling. Technical report 404, University of Oslo, Dept. of Informatics, Mar. 2011.