Journal of Logical and Algebraic Methods in Programming ••• (••••) •••–•••

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Effect-polymorphic behaviour inference for deadlock checking[☆,☆☆]

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

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

A R T I C L E   I N F O

Article history:
Received 29 March 2014
Received in revised form 20 May 2016
Accepted 20 May 2016
Available online xxxx

Keywords:
Deadlock detection
Simulation
Type and effect system
Concurrency
Formal method

A B S T R A C T

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

1. Introduction

Deadlocks are a common problem for concurrent programs with shared resources. According to the classic characterization from [11], a deadlocked state is marked by a number of processes, which forms a cycle where each process is unwilling to release its own resource, and is waiting on the resource held by its neighbour. The inherent non-determinism makes deadlocks, as other errors in the presence of concurrency, hard to detect and to reproduce. We present a static analysis using behavioural effects to detect deadlocks in a higher-order concurrent calculus. Deadlock freedom, an important safety property for concurrent programs, is a thread-global property, i.e., the blame for a deadlock in a defective program cannot be put on a single thread, it is two or more processes that share the responsibility; the somewhat atypical situation, where a process forms a deadlock with itself, cannot occur in our setting, as we assume re-entrant locks. The presented approach works in two stages. The first stage, which is the focus of this paper, corresponds to *model extraction*: an effect-type system uses a static behavioural abstraction of the codes' behaviour, concentrating on the lock interactions. To analyse the consequences on the global level, in particular for detecting deadlocks, the combined individual abstract thread behaviours are explored in the second stage.

Two challenges need to be tackled to make the approach applicable in practice. For the first stage on the thread local level, the model extraction, the static analysis must be able to *derive* the abstract behaviour, not just check compliance of the code with a user-provided description. This is the problem of type and effect *inference* or reconstruction. As usual, the abstract behaviour needs to over-approximate the concrete one, i.e., concrete and abstract descriptions are connected by

☆ Partly funded by the EU projects FP7-610582 Envisage: Engineering Virtualized Services (http://www.envisage-project.eu) and FP7-612985 UpScale: From Inherent Concurrency to Massive Parallelism through Type-based Optimizations (http://www.upscale-project.eu).
☆☆ This article is a full version of the extended abstract presented at the 25th Nordic Workshop on Programming Theory, NWPT 2013, in Tallinn.
* Corresponding author.
E-mail addresses: violet@ifi.uio.no (K.I Pun), msteffen@ifi.uio.no (M. Steffen), stolz@ifi.uio.no (V. Stolz).

http://dx.doi.org/10.1016/j.jlamp.2016.05.003
2352-2208/© 2016 Elsevier Inc. All rights reserved.

some *simulation* relation: everything the concrete system does, the abstract one can do as well (modulo some abstraction function relating the concrete and abstract states). For the second stage, exploring the (abstract) state space on the global level, obtaining *finite* abstractions is crucial. In our setting, there are four principal sources of infinity: the calculus allows 1) recursion, supports 2) dynamic thread creation, as well as 3) dynamic lock creation, and 4) with re-entrant locks, where the lock counters are unbounded. To allow static checking, appropriated abstractions, especially to tame the unbounded size of the mentioned dynamic aspects of the language. Our paper focuses on the model extraction in the first stage and how to infer the behavioural model and the role of polymorphism. This model extraction stage includes dealing with dynamic lock creation, as well. The model exploration in the second stage is covered in our previous work [43], which offers sound abstractions for lock counters and for recursion (but not for dynamic thread creation). See also the concluding remarks for a further discussion of how the earlier results carry over. Next, we shortly present in a non-technical manner the ideas behind the abstractions before giving the formal theory.

1.1. Effect inference on the thread local level

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

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

```
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)
```

Listing 1. Deadlock analysis and polymorphism.

The main thread, after creating two locks and defining function f, spawns a thread, and afterwards, the main thread and the child thread run in parallel, each one executing an instance of f with different actual lock parameters. In a setting with re-entrant locks, the program is obviously deadlock-free. Part of the type system of [43] determines the potential origin of locks by data-flow analysis. When analysing the body of the function definition, the analysis cannot distinguish the two instances of f (the analysis is context-*insensitive*). This inability to distinguish the two call sites — the "context" — forces that the type of the formal parameter is, at best, $\mathtt{L}^{\{\pi_1,\pi_2\}}$, which means that the lock-argument of the function is potentially created at either point. Based on that approximate information, a deadlock looks possible through a "deadly embrace" [16] where one thread takes first lock π_1 and then π_2, and the other thread takes them in reverse order, i.e., the analysis would report a (spurious) deadlock. The context-sensitive analysis presented here correctly analyses the example as deadlock-free.

1.2. Deadlock preserving abstractions on the global level

1.2.1. Lock abstraction

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

For a sound abstraction when identifying locks, one faces the following dilemma: a) the abstract level needs to exhibit at least the behaviour of the concrete level, i.e., we expect that concrete and abstract levels are related by a form of simulation.