Lock-Polymorphic Behaviour Inference for Deadlock Checking

Ka I Pun, Martin Steffen and Volker Stolz

Department of Informatics, University of Oslo, Norway

Deadlocks are a common problem for concurrent programs with shared resources. According to the classic characterization from [2], a deadlocked state is marked by a number of processes forming a cycle where each process, unwilling to release its own resource, is waiting on the resource held by its neighbor. The inherent non-determinism make deadlocks, as other errors in the presence of concurrency, hard to detect and to reproduce. We present a static analysis using behavioral effects to detect deadlocks in a higher-order concurrent calculus. Deadlock freedom, an important safety property for concurrent programs, is a thread-global property, i.e., the blame for a deadlock in a defective program cannot be put on a single thread, it is two or more processes that share responsibility; the somewhat atypical situation, where a process forms a deadlock with itself, cannot occur in our setting, as we assume re-entrant locks. The approach presented in this paper works in two stages: in a first stage, an effect-type system uses a static behavioral abstraction of the codes' behavior, concentrating on the lock interactions. To analyze the consequences on the global level, in particular for detecting deadlocks, the combined individual abstract thread behaviors are explored in the second stage.

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

Effect inference on the thread local level

As mentioned, in the first stage of the analysis, the analysis uses a behavioral type and effect system to over-approximate the lock-interactions of a single thread. To force the user to annotate the program with the expected behavior in the form of effects is impractical, so the type and especially the behavior should be inferred automatically. Effect inference, including inferring behavioral effects, has been studied earlier and applied to various settings, including obtaining static over-approximations of behavior for concurrent languages by Amtoft, Nielson and Nielson [1]. We apply effect inference to deadlock detection and as is standard (cf. e.g. [8, 11, 1]), the inference system is constraint-based, where the constraints in particular express an approximate order between behaviors. Besides being able to infer the behavior, it is important that the static approximation is as precise as possible. Since our calculus supports

higher-order functions, it is thus important that the analysis may distinguish different instances of a function body depending on their calling context, i.e., the analysis should be *polymorphic* or *context-sensitive.* This can be seen as an extension of let-polymorphism to effects and using constraints. The effect reconstruction resembles the known type-inference algorithm for let-polymorphism by Damas and Milner [4, 3] and this has been used for effect-inference in various settings, e.g., in the works mentioned above.

Deadlock checking in our earlier work [9] 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 l$_1$ = new$^{\flab_1}$ L in let l$_2$ = new$^{\flab_2}$ L in
let f = fn x:L . ( x.lock; x.lock )
in spawn(f(l$_1$)); f(l$_2$)
```

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 [9] determines the potential origin of locks by data-flow analysis. When analyzing the body of the function definition, the analysis cannot distinguish the two instances of f (the analysis is context-*insensitive*). This inability to distinguish the two call sites —the "context"— forces that the type of the formal parameter is, at best, $\mathtt{L}^{\{\pi_1,\pi_2\}}$, 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" [5] 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.

Deadlock preserving abstractions on the global level

Lock abstraction For dynamic data allocation, a standard abstraction is to *summarize* all data allocated at a given program point into one abstract representation. In the presence of loops or recursion, the abstracting function mapping concrete locks to their abstract representation necessarily is non-injective. For concrete, ordinary programs it is clear that identifying locks may change the behavior of the program. What makes identification of locks in general tricky, and here in particular connection with deadlocks, is that, on the one hand. it leads to *less* steps, in that lock-protected critical sections may become larger, and on the other hand to *more* steps at the same time, in that deadlocks may disappear when identifying locks. That this form of summarizing lock abstraction is problematic when analyzing properties of concurrent programs has been observed elsewhere as well, cf. e.g. Kidd et al. in [7].

For a sound abstraction for deadlock detection when identifying locks in the described way, one faces thus the following dilemma: a) the abstract level, using the abstract locks, need to show at least the behavior of the concrete level, i.e., we expect they are related by a form of simulation. On the other hand, to preserve not only the possibility of doing steps, but also *deadlocks,* the opposite must hold sometimes: a) a concrete program waiting on a lock and unable to make a step thereby, must imply an analogous situation on the abstract level, lest we should miss deadlocks. Let's write $l, l_1, l_2, \ldots$ for concrete lock references and $\pi, \pi', \ldots$ for program points of lock creation, i.e., abstract locks. To satisfy a): when a concrete program takes a lock, the abstract one must be able to "take" the corresponding abstract lock, say π. A consequence of a) is that taking an abstract lock is always enabled. That is consistent with the

abstraction as described where the abstract lock π confuses an arbitrary number of concrete locks including e.g., those freshly created, which may be taken.

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

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

Compared to [9], the paper makes the following contributions: 1) the effect analysis is generalized to a context-sensitive formulation, using constraint, for which we provide 2) an inference algorithm. Finally, 3) we allow summarizing multiple concrete locks into abstract ones, while still preserving deadlocks. All technical materials, lemmas and proofs can be found in the technical report [10].

References

[1] T. Amtoft, H. R. Nielson, and F. Nielson. *Type and Effect Systems: Behaviours for Concurrency.* Imperial College Press, 1999.

[2] E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2), 1971.

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

[4] 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.

[5] E. W. Dijkstra. Cooperating sequential processes. Technical Report EWD-123, Technological University, Eindhoven, 1965. Reprinted in [6].

[6] F. Genyus. *Programming Languages.* Academic Press, 1968.

[7] N. Kidd, T. W. Reps, J. Dolby, and M. Vaziri. Finding concurrency-related bugs using random isolation. *STTT*, 13(6):495–518, 2011.

[8] C. Mossin. *Flow Analysis of Typed Higher-Order Programs.* PhD thesis, DIKU, University of Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.

[9] 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, 2012. A preliminary version was published as University of Oslo, Dept. of Computer Science Technical Report 404, March 2011.

[10] K. I. Pun, M. Steffen, and V. Stolz. Lock-polymorphic behaviour inference for deadlock checking. Technical report 436, University of Oslo, Dept. of Informatics, Sept. 2013. available electronically at `http://www.ifi.uio.no/~msteffen/download/13/lockpolymorphic-rep.pdf`.

[11] J.-P. Talpin and P. Jouvelot. Polymorphic Type, Region and Effect Inference. *Journal of Functional Programming*, 2(3):245–271, 1992.