Deadlock checking by behavior inference for lock handling

September 27, 2010

Ka I Pun and Martin Steffen

Department of Informatics, University of Oslo, Norway

1 Motivation

Deadlock is a well-known problem of concurrent programs. There are 4 classical, necessary conditions for a deadlock in a program [3], namely mutual exclusion, no-preemption, the wait-for condition, and *circular wait*. All four condition must simultaneously hold for a deadlock, but to detect deadlock for particular programs, it is the last condition which matters, as the other three are typically language specific (and not specific for one program/run).

Apart from using run-time monitoring for deadlock detection, a number of statical methods to assure deadlock freedom have been proposed (cf. for instance [4,2,1,5,6]). In our work, we use type and especially effect systems capturing the interaction with locks to statically detect deadlocks.

2 A Type and Effect System for Deadlock Detection

In contrast to "traditional" type systems, which are used to classify values and assure proper use of those values, *effect* systems can capture phenomena that happens during evaluation (such as exceptions, side-effects, resource usage, ...). Particular expressive effects can deal with the "*behavior*" of a program during evaluation, i.e., taking the temporal ordering of the "phenomena" into account. That is especially needed for concurrent or parallel programs, where one is not so much interested in the eventual final value of the system, if any, but in the interactions with the environment.

Here, we apply the concept of a behavioral effect system to capture the interaction with *shared locks*. Having characterized the behavior of one thread (as a unit of concurrency) in terms of a sequences of lock interactions allows to detect the above-mentioned cycles as symptoms of deadlocks. Working on an abstraction of the actual behavior, the approach only allows to detect potential deadlocks.

The effects take roughly the following form

$$\varphi ::= \epsilon \mid \varphi_1; \varphi_2 \mid \varphi_1 + \varphi_2 \mid \varphi_1 \parallel \varphi_2 \mid \texttt{rec}\ \mu.\varphi \mid \texttt{spawn}\ \varphi \mid L^r \mid l^r.\ \texttt{lock} \mid l^r.\ \texttt{unlock}$$

for a underlying language with similar construct. In the above effect construct, semicolon represents sequential composition and + a choice. As the elementary interaction of a thread with a lock, $l.\ \texttt{lock}$ and $l.\ \texttt{unlock}$ represent

locking resp. releasing of a lock. To track which locks are actually handled in the interactions, the locks are annotated with the program points where they are created, and the r in the effects amount to a set of candidate locations.

The judgements of the type and effect system are given by

$$\Gamma \vdash e : T :: \varphi$$

meaning that expression e has type T and effect φ. The effect φ will be used for deadlock detection.

Four typical rules can be sketched as follows, dealing with thread and lock creation as well as interaction with an existing lock.

$$\frac{\Gamma \vdash e : T :: \varphi}{\Gamma \vdash \texttt{spawn}\ e : \texttt{thread}::\texttt{spawn}\ \varphi}\ \text{TE-Spawn} \qquad \Gamma \vdash \mathsf{new}_\pi\ L : L^\pi :: L^\pi\ \text{TE-NewL}$$

$$\frac{\Gamma \vdash v : \texttt{L}^r :: \varphi}{\Gamma \vdash v.\ \texttt{lock}: \texttt{L}^r :: v^r.\ \texttt{lock}:: \varphi}\ \text{TE-Lock} \qquad \frac{\Gamma \vdash v : \texttt{L}^r :: \varphi}{\Gamma \vdash v.\ \texttt{unlock}: \texttt{L}^r :: v^r.\ \texttt{unlock}:: \varphi}\ \text{TE-Unlock}$$

---

where L^r represents a lock which is created at r, where r is a set of program points. In particular, L^π (which is the same as $L^{\{\pi\}}$) states a lock is created at a program point π.

3 Results

We explore and formalize the idea for two different languages, a functional language with thread creation, and an object-oriented concurrent language based on active objects (Creol). We formalize operational semantics for the languages and a type and effect system for deadlock detection based on the ideas sketched above. We prove the soundness of our systems.

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. 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*.
3. E. G. Coffman Jr., M. Elphick, and A. Shoshani. System deadlocks. *Computing Surveys*, 3(2):67–78, June 1971.
4. F. S. de Boer and I. Grabe. Finite-state call-chain abstractions for deadlock detection in multithreaded object-oriented languages (extended abstract). In E. B. Johnsen, O. Owe, and G. Schneider, editors, *Proceedings of the 19th Nordic Workshop on Programming Theory (NWPT'07). Extended Abstracts. University of Oslo, Dept. of Computer Science, Technical Report 366*, Oct. 2007.

5. D. R. Engler and K. Ashcraft. Effective, static detection of race conditions and deadlocks: RacerX. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, pages 237–252, 2003.
6. A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages (ECOOP 2005)*, 2005.