Journal of Logical and Algebraic Methods in Programming 83 (2014) 400–426

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Deadlock checking by data race detection [☆]

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

Dept. of Informatics, University of Oslo, Norway

ARTICLE INFO

Article history:
Received 8 March 2013
Received in revised form 1 July 2014
Accepted 24 July 2014
Available online 13 August 2014

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

ABSTRACT

Deadlocks are a common problem in programs with lock-based concurrency and are hard to avoid or even to detect. One way for deadlock prevention is to statically analyse the program code to spot sources of potential deadlocks.
We reduce the problem of deadlock checking to data race checking, another prominent concurrency-related error for which good (static) checking tools exist. The transformation uses a type and effect-based static analysis, which analyses the data flow in connection with lock handling to find out control-points which are potentially part of a deadlock. These control-points are instrumented appropriately with additional shared variables, i.e., race variables injected for the purpose of the race analysis. To avoid overly many false positives for deadlock cycles of length longer than two, the instrumentation is refined by adding "gate locks". The type and effect system, and the transformation are formally given. We prove our analysis sound using a simple, concurrent calculus with re-entrant locks.

1. Introduction

Concurrent programs are notoriously hard to get right and at least two factors contribute to this fact: Correctness properties of a parallel program are often global in nature, i.e., result from the correct interplay and cooperation of multiple processes. Hence also violations are non-local, i.e., they cannot typically be attributed to a single line of code. Secondly, the non-deterministic nature of concurrent executions makes concurrency-related errors hard to detect and to reproduce. Since typically the number of different interleavings is astronomical or infinite, testing will in general not exhaustively cover all behaviour and errors may remain undetected until the software is in use.

Arguably the two most important and most investigated classes of concurrency errors are *data races* [6] and *deadlocks* [13]. A data race is the simultaneous, unprotected access to mutable shared data with at least one write access. A deadlock occurs when a number of processes are unable to proceed, when waiting cyclically for each other's non-shareable resources without releasing one's own [11]. Deadlocks and races constitute equally pernicious, but complementary hazards: locks offer protection against races by ensuring mutually exclusive access, but may lead to deadlocks, especially using fine-grained locking, or are at least detrimental to the performance of the program by decreasing the degree of parallelism. Despite that, both share some commonalities, too: a race, respectively a deadlock, manifests itself in the execution of a concurrent program, when two processes (for a race) resp. two or more processes (for a deadlock) reach respective control-points that

[☆] 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).

[*] 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.2014.07.003
2352-2208/© 2014 Elsevier Inc. All rights reserved.

Table 1
Abstract syntax.

P	$::=$	$\emptyset \mid p\langle t\rangle \mid P \parallel P$	program
t	$::=$	v	value
	$\mid$	$\texttt{let}\, x{:}T = e\, \texttt{in}\, t$	local variables and sequ. composition
e	$::=$	t	thread
	$\mid$	$v\ v$	application
	$\mid$	$\texttt{if}\, v\, \texttt{then}\, e\, \texttt{else}\, e$	conditional
	$\mid$	$\texttt{spawn}\, t$	spawning a thread
	$\mid$	$\texttt{new L}$	lock creation
	$\mid$	$v.\texttt{lock}$	acquiring a lock
	$\mid$	$v.\texttt{unlock}$	releasing a lock
v	$::=$	x	variable
	$\mid$	l^r	lock reference
	$\mid$	$\texttt{true} \mid \texttt{false}$	truth values
	$\mid$	$\texttt{fn}\, x{:}T.t$	function abstraction
	$\mid$	$\texttt{fun}\, f{:}T.x{:}T.t$	recursive function abstraction

when reached *simultaneously*, constitute an unfortunate interaction: in case of a race, a read–write or write–write conflict on a shared variable, in case of a deadlock, running jointly into a cyclic wait.

In this paper, we define a static analysis for multi-threaded programs which allows reducing the problem of deadlock checking to race condition checking. Our target language has explicit locks, i.e. we address *non-block structured* locking, and we can certify programs as safe which cannot be certified by approaches that use a static lock order (see Section 7 on related work).

The analysis consists of two phases. The first phase statically calculates information about lock usages per thread through a type system. Since deadlocks are a global phenomenon, i.e., involving more than one thread, the derived information is used in the second phase to instrument the program with additional variables to signal a race at control points that potentially are involved in a deadlock. The formal type and effect system for lock information in the first phase uses a constraint based flow analysis as proposed by Mossin [23]. The effects, using the flow information, capture in an approximate manner how often different locks are being held and is likewise formulated using constraints. This information roughly corresponds to the notion of lock-sets in that at each point in the program, the analysis gives approximate information which locks are held. In the presence of re-entrant locks, an upper bound on how many times the locks are being held is given, which corresponds to a “may”-over-approximation. In contrast, the notion of lock-sets as used in many race-freedom analyses, represents sets of locks which are necessarily held, which dually corresponds to a “must”-approximation.

Despite the fact that races, in contrast to deadlocks, are binary global concurrency errors in the sense that only two processes are involved, the instrumentation is not restricted to deadlock cycles of length two. To avoid raising too many spurious alarms when dealing with cycles of length larger than 2, the transformation adds additional gate locks to check possible interleavings to a race (deadlock) pairwise.

Our approach widens the applicability of freely available state-of-the-art static race checkers: *Goblint* [32] for the C language, which is not designed to do any deadlock checking, will report appropriate data races from programs instrumented through our transformation, and thus becomes a deadlock checker as well. *Chord* [24] for Java only analyses deadlocks of length two for Java’s `synchronized` construct, but not explicit locks from `java.util.concurrent`, yet through our instrumentation reports corresponding races for longer cycles *and* for deadlocks involving explicit locks.

The remainder of the paper is organized as follows. Section 2 presents syntax and operational semantics of the calculus. Section 3 afterwards provides the specification of the data flow analysis in the form of a (constraint-based) effect system, whose algorithmic solution is formalized in Section 4. The obtained information is used in Sections 5 and 6 to instrument the program with race variables and additional locks. The sections also prove the soundness of the transformation. We conclude in Section 7 discussing related and future work.

2. Calculus

In this section we present the syntax and (operational) semantics for our calculus, formalizing a simple, concurrent language with dynamic thread creation and higher-order functions. Locks can be created dynamically, they are re-entrant and support non-lexical use of locking and unlocking. The abstract syntax is given in Table 1. 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 $\parallel$ 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 count among expressions. Further expressions are function application, conditionals, and the spawning of a new thread, written $\texttt{spawn}\, t$. The last three expressions deal with lock handling: `new L` creates a new lock (initially free) and returns a reference to it (the `L` may be seen as a class for locks), and furthermore $v.\texttt{lock}$ and $v.\texttt{unlock}$ acquire and release a lock, respectively. Values, i.e., evaluated expressions, are variables, lock refer-