Science of Computer Programming 112 (2015) 227–260

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Cooperative types for controlling thread interference in Java

Jaeheon Yi [a,1], Tim Disney [a], Stephen N. Freund [b,*], Cormac Flanagan [a]

[a] *UC Santa Cruz, Santa Cruz, CA, USA*
[b] *Williams College, Williamstown, MA, USA*

ARTICLE INFO

Article history:
Received 20 June 2014
Received in revised form 3 August 2015
Accepted 4 August 2015
Available online 15 August 2015

Keywords:
Cooperability
Concurrency
Type systems

ABSTRACT

Multithreaded programs are notoriously prone to unintended interference between concurrent threads. To address this problem, we argue that yield annotations in the source code should document all thread interference, and we present a type system for verifying the absence of undocumented interference in Java programs. Under this type system, well-typed programs behave as if context switches occur only at yield annotations. Thus, well-typed programs can be understood using intuitive sequential reasoning, except where yield annotations remind the programmer to account for thread interference.

Experimental results show that yield annotations describe thread interference more precisely than prior techniques based on method-level atomicity specifications. In particular, yield annotations reduce the number of interference points one must reason about by an order of magnitude. The type system is also more precise than prior methods targeting race freedom, and yield annotations highlight all known concurrency defects in our benchmarks.

The type system reasons about program behavior modularly via method-level specifications. To alleviate the programmer burden of writing these specifications, we extend our system to support method specification inference, which makes our technique more practical for large code bases.

1. Controlling interference

Developing reliable multithreaded software is challenging due to the potential for nondeterministic interference between concurrent threads. Programmers use synchronization idioms such as locks to control thread interference but unfortunately do not typically document where interference may still occur in the source code. Consequently, every programming task (such as a feature extension, code review, etc.) may require the programmer to manually reconstruct the actual interference points by analyzing the synchronization idioms in the source code. This approach is problematic since manual reconstruction of interference points is tedious and error-prone. Moreover, interference points are fairly sparse in practice, and consequently programmers often simply assume that code is free of interference — a shortcut that may result in scheduler-dependent defects, which are notoriously difficult to detect or eliminate.

We propose to use yield annotations to document the actual interference points in the source code, thereby avoiding the need to manually infer interference points during program maintenance. Moreover, yield annotations enable us to decompose the hard problem of multithreaded program correctness into two simpler correctness requirements:

[*] Corresponding author at: 47 Lab Campus Drive, Williams College, Williamstown, MA 01267, USA. Tel.: +413 597 4260.
E-mail address: freund@cs.williams.edu (S.N. Freund).
[1] Current address: Google, Mountain View, CA, USA.

http://dx.doi.org/10.1016/j.scico.2015.08.004
0167-6423/© 2015 Elsevier B.V. All rights reserved.

```
class TSP {
  final Object lock;
  volatile int shortestPathLength;

  compound void searchFrom(Path path) {
    if (path.length >= this..shortestPathLength)
      return;

    if (path.isComplete()) {
      ..synchronized (this.lock) {
        if (path.length < this.shortestPathLength)
          this.shortestPathLength = path.length;
      }
    } else {
      for (Path c: path.children())
        this..searchFrom#(c);
    }
  }
}
```

Fig. 1. Traveling Salesperson Algorithm.

Cooperative-preemptive equivalence: Does the program exhibit the same behavior under a *cooperative scheduler* (that performs context switches only at yield annotations) as it would under a traditional *preemptive scheduler* (that performs context switches at arbitrary program points)?

Cooperative correctness: Is the program correct when run under a cooperative scheduler?

In this paper, we present a static type and effect system for Java that verifies the correctness property of cooperative-preemptive equivalence, which indicates that all thread interference is documented with yield annotations. The type system checks that the instructions of each thread consist of a sequence of *transactions* separated by yield annotations, where each transaction is serializable according to Lipton's theory of reduction [33]. Consequently, any preemptive execution of a well-typed program is guaranteed to behave *as if* executing under a cooperative scheduler, where context switches happen only at explicit yield annotations.

Cooperative scheduling provides an appealing concurrency semantics with the following properties:

1. *Thread interference is always highlighted with yields.* Yield annotations remind the programmer to allow for the effects of interleaved concurrent threads. In addition, unintended interference does not go unnoticed because our type system will reject programs with missing yields.
2. *Sequential reasoning is correct by default for code fragments with no yield annotations.* A programmer may safely ignore concurrency when reasoning about yield-free fragments, which can drastically simplify correctness arguments. Moreover, any automatic program analysis for sequential code may be applied to those regions, including, for example, the many verification techniques based on axiomatic semantics, such as ESC/Java [20], SLAM [5], Blast [6], separation logic [40,46], and many others. In contrast, previous systems based on identifying atomic methods provide little assistance in reasoning about non-atomic code fragments since they do not highlight exactly where in those fragments interference may occur [18,25,54].

A previous user study showed that these properties provide a distinct benefit to the programmer [49]. Specifically, the presence of yield annotations produces a statistically significant improvement in the ability of programmers to identify concurrency defects during code inspection.

We have developed a prototype implementation, called the Java Cooperability Checker (JCC), of our type system for verifying cooperative-preemptive equivalence. Experimental results on a range of Java benchmarks show that JCC requires yield annotations on only about 16 interference points (IPs) per KLOC. In comparison, previous non-interference specifications generate significantly more interference points: 134 IP/KLOC using atomic methods specifications; 50 IP/KLOC when reasoning about data races; and 25 IP/KLOC when reasoning about both data races and atomic methods.

1.1. Example

To illustrate the benefits of explicit yield annotations, consider the traveling salesperson algorithm shown in Fig. 1. The `TSP` class contains a method `searchFrom` that recursively searches through all extensions of a particular `path`, aborting the search whenever `path.length` becomes greater than `shortestPathLength` (the length of the shortest complete path found so far). As we describe below, the "`..`" notation indicates yield points, and the `compound` and `#` annotations mark the declaration and call of a method containing yield points, respectively.