Verifying a Compiler Optimization for Multi-Threaded Java

Bernhard Reus, Alexander Knapp, Pietro Cenciarelli, and Martin Wirsing

Ludwig-Maximilians-Universität München
{reus, knapp, cenciare, wirsing}@informatik.uni-muenchen.de

Abstract. The specification for the object-oriented concurrent language Java [3] is rather loose with respect to the interaction of shared memory and the local working memories of different threads. This allows maximal freedom in the language implementation. Such freedom is reflected in the semantics provided in [2], where threads-memory interaction is formalized in terms of structures called event spaces. Two kinds of memories are described in the Java specification: a “normal” memory and a more liberal one, where values can sometimes be stored even before they are produced as results of computation. Here we compare two structural operational semantics of a sublanguage of Java modelling the two types of memory. The two semantics share the same set of operational rules but put different requirements (expressed as first order theories) on the notion of event space. We prove a result which is informally stated in [3]: the two semantics coincide for properly synchronized programs. This shows the applicability of a new technique for combining structural operational semantics and first order specification of process behaviour.

1 Introduction

A concurrent program consists of multiple tasks that are or behave as if they were executed all at the same time. Such tasks can be implemented using threads (short for “threads of execution”), which are sequences of instructions that run independently within the encompassing program. The object-oriented language Java supports thread programming (see e.g. [1], [4]).

Java threads share a common memory, but keep working copies of shared variables in private working memories. It is only when leaving a synchronized block that must a thread copy the content of its working memory in the main memory. However, possible implementations of the run-time system may choose to update the value of a variable in the main memory as soon as a thread makes an assignment to its working copy of that variable. The Java language specification [3] leaves freedom to the implementation in that respect.

A particular implementation technique is also discussed in [3], where a value can be stored by a thread in the main memory before such value is produced by the computation. This is called a prescient store action [3, §17.8]. The only restriction is that between the prescient store and the matching assignment nothing “bad” happens, e.g. no other thread reads illegitimately the prescient value. Consider the following code example.
\[ o.b = 2; \]
\[ \text{for}(o.a = 1; o.a < 10; o.a = o.a + 1) \]
\[ o.b = o.a + o.b; \]

where \( o \) is (a reference to) an object with two attributes \( a \) and \( b \) of type int. When executed the value of \( o.a \) can only be stored after \( o.a \) has been assigned a new value, i.e. after it was incremented. If prescient store operations are permitted, then it would be also legal to store the value 10 for \( o.a \) in advance, i.e. before the loop is entered, excluding thereby that any other thread can load \( o.a \) before the end of the loop.

The rearrangement of store operations can be used to speed up programs when updating of variables is split into a thread action (called \( \text{Store} \)) and a memory action (\( \text{Write} \)). The global memory can concurrently provide the value of a pre-stored variable while a second thread waits for it. Re-grouping the store-operations might also optimize memory access itself.

In [2] we present a structural operational semantics (in the style of [5]) of a nontrivial sub-language of Java which includes dynamic creation of objects, blocks, and synchronization of threads. The notion of \( \text{event space} \) is introduced in that paper to formalize the communication protocol between shared memory and threads. Event spaces correspond roughly to \( \text{configurations} \) in Winskel's \( \text{event structures} \) [6], which are used for denotational semantics of concurrent languages.

Here we exploit the flexibility of the approach proposed in [2], where the operational semantics is given parametrically in the notion of event space, and compare two language implementations which share the same set of operational rules. The implementations are obtained by imposing different requirements on event spaces, so that prescient stores are possible in one case and impossible in the other. Such requirements are expressed in simple first order clauses. In this framework we prove that prescient and nonprescient semantics coincide for properly synchronized programs, that is programs where any two threads are not allowed to write a variable into the global memory without synchronization (\( \text{race conditions} \) [4]). This property was only informally stated in [3, §17.8]. We also provide an example where prescient store actions for non-properly synchronized programs lead to inconsistent memory contents.

The contribution of the paper is twofold: on the one hand it provides welcome formal confirmation of the intuitive correctness of certain compiler optimization techniques; on the other hand it shows the applicability of an innovative technique for combining structural operational semantics and first order axiomatization of process behaviour.

The paper is organized as follows: Section 2 recapitulates the definition of event spaces from [2]. These are used in Section 3 for the SOS-rules. Next, the axiomatization of event spaces is changed (Section 4) in order to allow for prescient stores and prescient operational semantics is defined in Section 5. It is then proven, in Section 6, that for properly synchronized programs the extension is conservative w.r.t. the old semantics.