it – Information Technology 2014; 56(2): 82–86 — DOI 10.1515/itit-2013-1037 DE GRUYTER OLDENBOURG

Distinguished Dissertation

Andreas Lochbihler*

Analysing Java's safety guarantees under concurrency

Abstract: Two features distinguish Java from other main-stream programming languages like C and C++: its built-in support for concurrency and safety guarantees such as type safety or safe execution in a sandbox. In this work, we build a formal, unified model of Java concurrency, validate it empirically, and analyse it with respect to the safety guarantees using a proof assistant. We show that type safety and Java's data race freedom guarantee hold. Our analysis, however, revealed a weakness in the Java security architecture, because the Java memory model theoretically allows pointer forgery. As a result, this work clarifies the specification of the Java memory model.

Keywords: ACM CCS → Software and its engineering → Software notations and tools → Formal language definitions → Semantics, ACM CCS → Theory of computation → Semantics and reasoning → Program semantics → Operational semantics, ACM CCS → Computing methodologies → Concurrent computing methodologies → Concurrent programming languages, Java threads, operational semantics, type safety, data race freedom, compiler verification.

***Corresponding Author: Andreas Lochbihler,**
Institute of Information Security, ETH Zurich,
e-mail: andreas.lochbihler@inf.ethz.ch

1 Introduction

The programming language Java provides two important guarantees: type safety and sandboxing. Numerous applications build on this fundament; we mention just three: First, sandboxing allows one to execute code from untrusted sources without running the risk of infecting the host system with malicious code. Second, compilers can optimise programs better and the Java virtual machine (JVM) can execute programs faster. For example, when a field of an object is accessed, type safety ensures that the field always exists, so the JVM does not have to search for the field at run-time – as is common in untyped scripting languages. Third, tools for static analysis, debugging and verification of Java programs rely on type safety to, e.g., internally compute abstract representations such as control flow graphs and determine the possible targets of a method call.

Threads and the Java memory model (JMM) are another characteristic feature of the programming language. Java was the first main-stream programming language to embrace shared-memory concurrency with a well-defined semantics. Many programmers intuitively expect that the steps of individual threads interleave, which is known as sequential consistency (SC). But this hinders optimisations both in compilers and JVMs, as the following excerpt of a Java program illustrates:

```
class C { static int x = 0, y = 0; }
```

thread 1	thread 2
1: `C.y = 1;`	3: `C.x = 1;`
2: `int r1 = C.x;`	4: `int r2 = C.y;`

If we execute both threads under interleaving semantics, the pair of local variables `r1` and `r2` may store only the values $(0, 1)$, $(1, 0)$, or $(1, 1)$ at the end, but not $(0, 0)$. Now suppose that the two threads run on different cores that cache their writes to memory in their own buffers, i.e., on a standard x86 multi-core processor. Then, the writes to memory in lines 1 and 3 may still be stuck in the buffers when the subsequent reads from memory in lines 2 and 4 execute. In this case, they both return the initial value 0 from main memory, i.e., both `r1` and `r2` are 0. To enforce SC, the compiler or the JVM would therefore have to insert synchronisation instructions like barriers or fences be-

The dissertation is entitled "A Machine-Checked, Type-Safe Model of Java Concurrency – Language, Virtual Machine, Memory Model and Verified Compiler". The examiners were Prof. Dr.-Ing. G. Snelting (Karlsruhe Institute of Technology) and Prof. T. Nipkow, PhD (TU Munich). The dissertation has been recommended to the GI-Dissertation Award 2012 by the Karlsruhe Institute of Technology.

tween the writes and the reads. Moreover, standard compiler optimisations like common subexpression elimination have similar effects, so they would have to be restricted or even disabled.

To avoid the ensuing slowdown, the JMM relaxes the semantics and allows *more* results than SC such as (0, 0) in the example. Consequently, concurrency nontrivially interacts with safety guarantees like type safety and sandboxing, which is well-known by examples [11]. Nevertheless, programmers may forget about the JMM and assume sequential consistency if their programs are correctly synchronised. This promise by the JMM is commonly called the data race freedom (DRF) guarantee. In the above example, it suffices to declare `x` and `y` as `volatile`. Then, the compiler and JVM are warned that these variables are used concurrently, and they can take the necessary precautions.

2 A validated model of concurrent Java

But does Java with the JMM indeed provide these guarantees: the DRF guarantee, type safety, and safe execution in a sandbox? We will answer this question in Sections 3 and 4. However, we first build a single formal model of concurrent Java that includes source code and bytecode and a compiler.

2.1 Modelling Java

Mechanisation

When we design our model, we must strike a balance between comprehensiveness and ease of use. On the one hand, it should cover all relevant interactions between the different features of the language, and thus be as comprehensive as possible; in particular, we must not palliate the ugly corners. On the other hand, we need a model that we can analyse. However, concurrent Java is a complex language, and our model will therefore be large and complex, too. Thus, we decided to use machine support and formalise all our definitions and proofs in the proof assistant Isabelle/HOL [10]. Hence, Isabelle checks all our definitions and ensures that our reasoning is correct and that we do not miss any corner cases. This is especially important whenever we extend or adapt our model.

Our mechanised model JinjaThreads includes classes with objects, fields, and methods, inheritance with method overriding and dynamic dispatch, arrays, exception handling, assignments, local variables, binary operators and standard control structures. Moreover, it covers all concurrency primitives of the Java language specification [2] except for the deprecated and time-dependent ones and the compare-and-swap operations from `java.util.concurrent`.

Modular structure

To obtain a usable model, we followed a modular design. The semantics consists of three layers: the single-threaded semantics, the semantics of the concurrency primitives, and the Java memory model. The sequential part builds on Jinja by Klein and Nipkow [4], who formalised a sequential Java-like programming language including source code, bytecode and a compiler in Isabelle. At its core, an operational semantics translates program syntax into atomic execution steps of single threads. The second layer implements the concurrency primitives and interleaves the atomic steps from the layer below. Thus, we obtain a labelled transition system where the labels record all the interactions between the threads; every path in it yields an execution candidate. The top layer consists of the JMM. It specifies – using axiomatic criteria – which candidates are allowed executions.

This structure separates the sequential issues from the concurrency aspects. Thus, it clarifies our model, aids reuse of common parts, and simplifies our proofs. Nevertheless, it rigorously links Java and the JMM, which has been sorely missing in the literature [1, 3]. Although most language features have already been studied in isolation, we have found many new intricate interactions between them. We will discuss an example in Section 4.

Verified compiler

JinjaThreads' compiler translates source code programs into bytecode. This connection completes the unified model of Java and shows that both languages fit together. We have also formally verified that the compiler preserves both static and dynamic semantics of the programs [6].

Theorem 1 (**Compiler correctness**). *Given a well-typed source code program as input, the compiler produces bytecode with the same semantics that the bytecode verifier accepts.*

It is worth discussing what semantics means. In classical compiler verification for sequential languages, one shows that if the source code program terminates, then the compiled program terminates, too, and computes the same value [5]. In the concurrent setting, however, programs may terminate normally under some schedules, but diverge or deadlock under others. Moreover, we are also in-

terested in interactive programs that produce intermediate output or run forever. Therefore, our dynamic semantics explicitly models non-termination, deadlocks, and intermediate output in addition to the usual output upon termination. Consequently, our compiler preserves all these kinds of behaviour by Theorem 1.

When we prove this theorem, arranging the semantics in layers pays off. As our implementation of the concurrency primitives and the memory model does not depend on the language, it suffices to prove that the individual threads of the compiled program are bisimilar. Then, a generic argument shows that the compiled program as a whole is bisimilar to the source program, too. Thus, the actual verification can focus on sequential correctness and ignore concurrency issues.

2.2 Validation

Next, we should convince ourselves that we have faithfully modelled Java. Formal verification is not possible, because the Java language specification itself is not formal. But mechanisation offers an elegant solution: we have compiled our model into an executable prototype using Isabelle's code generator. Thus, we can type-check, compile, and run a test suite of Java programs and check whether the test output matches the specified result.

To that end, we developed the front-end Java2Jinja as an Eclipse plug-in (http://pp.ipd.kit.edu/projects/quiscustodiet/Java2Jinja/). First, it converts a Java program into JinjaThreads' abstract syntax; if necessary and possible, it emulates features such as inner classes and generics that our model does not support. Then, it compiles and runs the program using the generated prototype. For regression tests, Java2Jinja can also run a whole test suite automatically.

Validation requires that our prototype is reasonably efficient. Yet, we have designed the model for ease of proving, not fast execution. Therefore, we have developed means to gain efficiency without affecting the definitions and proofs [9]. This way, the prototype becomes as efficient as other Java formalisations that have been designed for efficient execution from the start. Over those, our interpreter, compiler, and JVM have the advantage of being verified: Isabelle's code generator ensures that every step of the execution corresponds to a term rewrite step in the formal model. Thus, all our theorems hold for the output of our prototype, as we could simulate its execution in Isabelle.

Overall, we empirically validated our model with 189 test programs, 24 of which are OpenJDK regression tests. We found only one bug in the implementation of the division and modulo operations for negative integers, and fixed it. Thus, we are confident that JinjaThreads faithfully models Java.

3 Interleaving semantics for racefree programs

Next, we analyse our formal model – by the validation, our results also apply to Java. We start with the DRF guarantee:

Theorem 2 (**DRF guarantee**). *If a program is correctly synchronised, then its executions cannot be distinguished from sequentially consistent executions.*

Let us examine this statement first. The DRF guarantee does not require that compilers and JVMs enforce SC for correctly synchronised programs; it suffices if the program can observe neither the optimisations nor the hardware buffers. Correct synchronisation significantly constrains these observation capabilities: it requires that no sequentially consistent execution contains a data race. And a data race occurs whenever two threads access the same non-`volatile` memory location without synchronisation in between and at least one of them writes. Therefore, a correctly synchronised program tells the compilers and JVM when and what other threads may observe, so they can aggressively optimise the rest of the program at their will. In summary: when a programmer ensures (e.g., by means of locks and `volatile` declarations) that there is no data race, then he can forget about the JMM and instead intuitively reason with interleaving semantics.

The DRF guarantee has been formalised previously [1, 3], but only at the axiomatic level of the JMM with implicit assumptions about the underlying language. In contrast, our proof bridges the gap between the JMM and Java: it formally shows that Java meets the assumptions of the JMM [7]. In fact, we have further strengthened our result: for correctly synchronised programs, SC and the JMM are equivalent.

4 Type safety and the security architecture

Other language specifications such as the C++ standard stop at correctly synchronised programs and assign undefined semantics to programs with data races. In contrast, Java's designers have gone further, because type safety and sandboxing demand well-defined semantics for *all*

Java programs. However, giving semantics to data races has made the JMM very complex.

For example, the JMM allows programs to read values from fields whose objects will be created only later, as the following excerpt illustrates:

```
class A { static A x, y; int f;
          int m() { return this.f; } }
```

thread 1	thread 2
1: `A r = A.x;`	4: `A q = A.y;`
2: `if (r != null) r.m();`	5: `A.x = q;`
3 : `A.y = new A();`	

Java compilers are allowed to move the allocation in line 3 before line 1. Consequently, the newly allocated object may be the receiver of the method call in line 2 (schedule 3, 4, 5, 1, 2). However, our semantics executes the program as is, i.e., line 2 always before line 3. Consequently, we have to resolve the call to method `m()` in line 2, but the receiver and its class are only determined after line 3. Moreover, the method `m` reads the field `f`, but its memory has not yet been allocated.

For the DRF guarantee, we were able to ignore some of these complications by showing that they are only observable via data races. In contrast, type safety has to deal with such reorderings of statements with object allocations. In our attempts to prove type safety, we identified several fixes to the JMM specification. For instance, the example demonstrates that references themselves must contain type information about the object they point to. In the end, we proved that Java and the JMM are indeed type safe.

Theorem 3 (**Type safety**). *Java with the fixes to JMM is type safe.*

Surprisingly, a weakness in the JMM specification allows pointer forgery. Thus, Java with the JMM is type safe and allows pointer forgery at the same time, which normally exclude each other. It is the encoding of types into references that makes this possible.

However, pointer forgery allows behaviours that break Java's security architecture, i.e., malicious code could exploit the pointer forgery to escape from the sandbox. Fortunately, these are only theoretical examples, because we do not know of any existing optimisation in a compiler, a JVM, or in hardware that could lead to such behaviours. This shows that the JMM really needs to be revised in some form; yet, it remains unclear whether a quick fix is possible at all, as our examples suggest that the flaw is inherent to the style of the JMM.

5 Summary

Today, JinjaThreads is the most comprehensive model of Java concurrency and the Java memory model. It covers a realistic subset of Java source code and bytecode, a verified compiler, and the Java memory model. The formalisation resulted in clarifications of and fixes to Java and the JMM. We showed that concurrent Java is type safe and provides the DRF guarantee, and we identified a flaw in the JMM that in principle breaks the Java security architecture.

Mechanisation has been crucial in this work for three reasons: First, we have built the model incrementally adding one feature at a time. Therefore, we had to frequently revise the models. Then, Isabelle can replay all the existing formalisation and automatically pinpoint which parts need to be adapted. Thus, one saves the work of re-examining all definitions and proofs. Second, our validation relied on mechanisation as we compiled the formalised model into an executable prototype. Third, JinjaThreads constitues the semantics foundation in the Quis Custodiet project (http://pp.ipd.kit.edu/projects/quis-custodiet/) which aims at formally verifying an infrastructure for information flow control. Machine support is essential to ensure that everything indeed fits together – in particular as the techniques are sound only for correctly synchronised programs, and therefore rely on type safety and the DRF guarante.

Despite its size and complexity, JinjaThreads does not yet cover full sequential Java. From the concurrency perspective, three features are particularly interesting: class initialisation, final fields, and garbage collection. We expect new insights and more surprises from a thorough analysis.

Received October 29, 2013; accepted November 5, 2013.

References

1. D. Aspinall and J. Ševčík. Formalising Java's data-race-free guarantee. In *TPHOLs 2007*, LNCS 4732, pp. 22–37. Springer, 2007.
2. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edn.*. Addison-Wesley, 2005.
3. M. Huisman and G. Petri. The Java Memory Model: a formal explanation. In *VAMP 2007*, technical report ICIS-R07021, pp. 81–96. University of Nijmegen, 2007.
4. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Sys.*, 28(4):619–695, 2006.
5. X. Leroy. A formally verified compiler back-end. *J. Autom. Reason.*, 43(4):363–446, 2009.
6. A. Lochbihler. Verifying a compiler for Java threads. In *ESOP 2010*, LNCS 6012, pp. 427–447. Springer, 2010.
7. A. Lochbihler. Java and the Java memory model – a unified, machine-checked formalisation. In *ESOP 2012*, LNCS 7211, pp. 497–517. Springer, 2012.
8. A. Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model and Verified Compiler*. PhD thesis, Department of Informatics, Karlsruhe Institute of Technology, 2012.
9. A. Lochbihler and L. Bulwahn. Animating the formalised semantics of a Java-like language. In *ITP 2011*, LNCS 6898, pp. 216–232. Springer, 2011.
10. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, Lecture Notes in Computer Science 2283. Springer, 2002.
11. W. Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12:445–455, 2000.

Dr. Andreas Lochbihler
Institute of Information Security,
ETH Zurich, Universitätstrasse 6, CH-8092 Zurich, Switzerland, Tel.: +41-44-6328470, Fax: +41-44-6321172
andreas.lochbihler@inf.ethz.ch

Andreas Lochbihler is working as a post-doctoral researcher in the information security group at ETH Zurich. His research focuses on deriving machine-checked implementations from protocol specifications such that the security properties of the models are preserved. Andreas graduated in computer science from the University of Passau in 2006, after having studied there and at the University of Edinburgh. He received his doctorate from the Karlsruhe Institute of Technology in 2012. Before joining ETH, he was a member of Gregor Snelting's groups at the University of Passau and the Karlsruhe Institute of Technology working on programming languages and static program analysis.