Ein maschinengeprüftes, typsicheres Modell der Nebenläufigkeit in Java: Sprachdefinition, virtuelle Maschine, Speichermodell und verifizierter Compiler*

Andreas Lochbihler

Institut für Informationssicherheit, ETH Zürich
andreas.lochbihler@inf.ethz.ch

Abstract: Charakteristisch für die Programmiersprache Java sind sowohl ihre Sicherheitsgarantien wie beispielsweise Typsicherheit und die Sicherheitsarchitektur als auch die direkte Unterstützung von Threads. In der hier vorgestellten Dissertation [Loc12b] wird ein maschinengeprüftes Modell von nebenläufigem Java einschließlich des Java-Speichermodells entwickelt und die Auswirkungen der Nebenläufigkeit auf diese Garantien untersucht. Aus dem formalen Modell wurde automatisch ein ausführbarer Interpreter, Übersetzer und eine virtuelle Maschine einschließlich eines Bytecode-Verifizierers generiert, mit dem das Modell empirisch gegen Java-Benchmarks validiert wurde.

1 Einführung

Typsicherheit und die Sicherheitsarchitektur sind wichtige Garantien der Programmiersprache Java. Sie erlauben es, Code aus nicht vertrauenswürdigen Quellen in einer Sandbox ohne die Gefahr, dass sich das ausführende System mit Schadcode infiziert, auszuführen. Ein weiteres Kennzeichen Javas sind Threads und das Java-Speichermodell (JMM) zur nebenläufigen Ausführung. Verglichen mit dem intuitiven Modell für sequenzielle Konsistenz (SC), bei der die Thread-Einzelschritte verschränkt ausgeführt werden und Speicherveränderungen sofort für alle Threads sichtbar werden, erlaubt das JMM mehr Ausführungen, um aggressive Optimierungen bei der Übersetzung nach Bytecode und in der virtuellen Maschine (VM) selbst zu ermöglichen. Deswegen wirkt sich Nebenläufigkeit auch auf die Typsicherheit und Sicherheitsarchitektur aus, wie seit langem durch Beispiele bekannt ist [GJSB05, Pug00].

Die hier zusammengefasste Dissertation [Loc12b] untersucht diese Auswirkungen nun zum ersten Mal formal. Dazu wird ein formales Modell, genannt JinjaThreads, der Nebenläufigkeit durch Java-Threads einschließlich des JMMs konstruiert und im Detail analysiert. Grundlage des sequenziellen Teils ist Jinja [KN06], das eine sequenzielle Java-ähnliche Programmiersprache mit Quellcode, Bytecode und einem Übersetzer im Beweis-

*Englischer Titel der Dissertation: "A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model and Verified Compiler"

assistent Isabelle/HOL [NPW02] formalisiert und maschinengeprüft als typsicher nachweist. Die Arbeit erweitert dieses Modell um Nebenläufigkeit durch Java-Threads (§2) und das JMM (§3) und überträgt die bisherigen Theoreme zur Typsicherheit und Korrektheit des Übersetzers (§4). Es zeigte sich, dass wegen neu entdeckter Schwachstellen des JMMs Typsicherheit eine Nicht-Standard-Modellierung erfordert und theoretisch sogar die Sicherheitsarchitektur ausgehebelt werden könnte. Zudem beweist die Arbeit die für Programmierer und Programmanalysen wichtige Garantie des JMMs, dass wettlauffreie Programme sich wie bei verschränkter Ausführung, d.h. sequenziell konsistent, verhalten. Insgesamt entstanden so der erste verifizierte Übersetzer für nebenläufiges Java und die erste formale Verbindung zwischen einer operationalen Java-Semantik und dem axiomatisch spezifizierten JMM.

Alle Definitionen und Beweise wurden von Isabelle/HOL geprüft und sind im Archive of Formal Proofs veröffentlicht [Loc07]. Damit zeigt diese Arbeit, dass heutzutage umfangreiche Programmiersprachen in handhabbarer Form in Theorembeweisern abgebildet werden können. Umgekehrt wäre diese Arbeit wegen der Komplexität des Modells ohne Maschinenunterstützung unmöglich gewesen, da nur so alle Auswirkungen einer Änderung oder Erweiterung auf andere Teile des Modells zuverlässig festgestellt werden konnten. Außerdem erlaubte die Verwendung von Isabelle, das formale Modell auszuführen und damit zu validieren (§5). Dazu wurde mit Isabelles Codegenerator automatisch aus der Formalisierung ein Interpreter, ein Übersetzer und eine virtuelle Maschine in Standard ML extrahiert. Durch eine Testsuite von Java-Programmen wurde damit das Modell von Quell- und Bytecode validiert.

2 Thread-Modellierung und verschränkte Ausführungen

Der erste Schritt zum nebenläufigen Modell von Java erweitert das sequenzielle Jinja um Java-Threads. Auf Sprachebene deckt JinjaThreads dynamische, unbeschränkte Thread-Erzeugung, Synchronisation über Monitore, Warten auf Benachrichtigung sowie Unterbrechung und Beitreten von Threads ab. JinjaThreads deckt somit alle Konstrukte für Nebenläufigkeit aus der Sprachspezifikation [GJSB05] außer den missbilligten oder zeitabhängigen Methoden und den Vergleichen-und-Tauschen-Operationen aus dem `java.util.concurrent`-Paket ab. Syntaktisch sind dafür nur geringe Anpassungen der sequenziellen Jinja-Sprache nötig (`synchronized`-Blöcke und `volatile`-Deklarationen), da die anderen Primitive als Aufrufe nativer Methoden realisiert sind.

Dagegen ändert sich die operationale Semantik grundlegend, da Synchronisationsprimitive implementiert und die Threads verschränkt ausgeführt müssen. Damit das Modell handhabbar bleibt, ist es wesentlich, möglichst viele Teile wiederzuverwenden, sowohl von Jinja als auch zwischen Quell- und Bytecode. Dazu wird die Semantik in sieben Schichten zerlegt (Abb. 1), von denen vorerst nur Nr. 2, 3 und 5 relevant sind. Da sowohl Quell- als auch Bytecode die gleichen Nebenläufigkeitskonstrukte benutzen, implementiert ein Transitionssystem $s -t{:}\alpha s\rightarrow s'$ diese sprachunabhängig in Schicht 5; es verschränkt die atomaren Schritte einer als Parameter gegebenen Einzelthreadsemantik (ETS) $t \vdash (x, h) -\alpha s\rightarrow (x', h')$ (t bezeichnet den Thread, x den thread-lokalen Zustand und h

Abbildung 1: Schichtenmodell der Semantik

den gemeinsamen Speicher). Insbesondere übernimmt es vollständig die Verwaltung des globalen Systemzustands s, d.h., der Threads, Sperren, Wartemengen und Unterbrechungen.

Entsprechend kann sich die ETS auf die Umsetzung der Syntax in atomare Berechnungsschritte konzentrieren (Schicht 3). Für Quellcode kann dafür die strukturelle operationelle Semantik (SOS) aus Jinja mit nur kleinen Anpassungen wiederverwendet werden, gleiches gilt für den funktionalen Stil der VM. Dies gilt nicht nur für die Defintionen, sondern auch für die deutlich aufwändigeren Beweise zur Typsicherheit (s.u.). Die Ausführung der Synchronisationsprimitive wird dabei an Schicht 5 delegiert, in dem die Transitionen mit entsprechenden abstrakten Aktionsanweisungen αs versehen werden, beispielsweise fordert *Lock*$\rightarrow l$ eine Sperre auf dem Monitor l an und *Unlock*$\rightarrow l$ gibt sie wieder frei. Für native Methoden werden diese Anweisungen bereits in Schicht 2 generiert und von Schicht 3 lediglich weitergeleitet.

Diese Aktionsanweisungen ermöglichen es, dass die Schichten 3 und 5 jeweils ohne Kenntnis der Implementierung der anderen Schicht definiert werden können. Die vollständige Semantik von Quellcode oder Bytecode erhält man einfach durch Instanziierung des ETS-Parameters. Dadurch ist automatisch sichergestellt, dass die lokalen Zustände der Threads gegen Zugriffe aus anderen Threads geschützt sind, und umgekehrt wird verhindert, dass die ETS direkt den Systemzustand manipulieren kann. Manchmal erfordert dies zwar eine aufwändigere Modellierung einzelner Aspekte. Beispielsweise gibt es in der SOS zwei Transitionen für das Starten eines Threads – eine für den erfolgreichen Start und eine mit einer Ausnahme, falls der Thread schon zuvor gestartet worden war. Dadurch wird die ETS nichtdeterministisch, und erst Schicht 5 entscheidet anhand der Aktionsanweisungen, welche davon im aktuellen Systemzustand die richtige ist. Jedoch stellt diese Trennung sicher, dass Aussagen und Beweise auf einem geeigneten Abstraktionsniveau formuliert bzw. geführt werden können, da alle relevanten Interaktionen zwischen den Threads durch die Aktionsanweisungen erfasst werden. Dies reduziert die Komplexität der Beweise und fördert die Wiederverwendung, wie am Beweis der Typsicherheit gut zu sehen ist:

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

	Thread t1	Thread t2
1:	`C.y = 1;`	`C.x = 1;` :3
2:	`int r1 = C.x;`	`int r2 = C.y;` :4

Abbildung 2: Beispiel-Programm mit zwei Threads

Theorem 1 (Typsicherheit)**.** *Ausgehend von einem korrekten JinjaThreads-Programm und passenden Startzustand gilt für alle über* $\_ -\_:\_\rightarrow \_$ *erreichbaren Zustände* s*, in denen die Ausführung endet (d.h., es gibt keine Transition* $s -\_:\_\rightarrow \_$*), dass jeder Thread* t *in* s *entweder verklemmt oder vollständig ausgewertet (d.h. ein Wert, dessen Typ kompatibel zum deklarierten Typ von* t *ist, oder eine unbehandelte Ausnahme) ist.*

Verklemmungen verkomplizieren die Typsicherheitsaussage, da selbst eine typkorrekte Ausführung eventuell nicht mehr vollständig auswertet. Deswegen muss obige Aussage Verklemmungen explizit berücksichtigen, aber dafür benötigt man eine Formalisierung von Verklemmungen. Da diese nur durch Synchronisation zwischen Threads entstehen können, erlauben die Aktionsanweisungen eine exakte semantische Charakterisierung mittels eines größten Fixpunkts, die genauer als die üblichen syntaktischen Approximationen ist und damit die Typsicherheitsaussage verstärkt. Außerdem lässt sich abstrakt in Schicht 5 für alle ETS zeigen, dass eine Ausführung nicht zu Ende ist, solange auch nur ein Thread weder verklemmt noch vollständig ausgewertet ist – vorausgesetzt, dass Typsicherheit bereits für die ETS gilt und nur erfüllbare Aktionsanweisungen verwendet werden. Dank dieser Aussage lassen sich also Jinjas Typsicherheitsbeweise für Quell- und Bytecode für den nebenläufigen Fall einfach wiederverwenden und die beiden neuen Typsicherheitsbeweise können den Verklemmungsfall einfach und auf die gleiche Weise abhandeln.

3 Das Java-Speichermodell

Bis jetzt wurde noch nicht auf die Modellierung des allen Threads gemeinsamen Speichers eingegangen, in dem alle Objekte und Arrays liegen. JinjaThreads modelliert diesen Teil des Systemzustands (Variablekonvention h für Halde in den ETSen) in zwei austauschbaren Modulen mit gleicher Schnittstelle (Schicht 1 in Abb. 1). Zum einen führt die in der Literatur übliche (endliche) Abbildung von Adressen auf die gespeicherten Daten zusammen mit der verschränkten Ausführung der Threads zur gewohnten SC, bei der jede Veränderung des Speicher durch einen Thread sofort für alle anderen Threads sichtbar wird. Beispielsweise kann das Paar der lokalen Variablen `r1` und `r2` der beiden Threads in Abb. 2 am Ende der Ausführung nur die Wertkombinationen $(0, 1)$, $(1, 0)$ und $(1, 1)$ enthalten, aber nicht $(0, 0)$.

Das JMM erlaubt hingegen alle vier Kombinationen, da heutige Prozessoren nur ein schwächeres Speichermodell bieten. Wenn beispielsweise die beiden Threads aus Abb. 2 auf verschiedenen Kernen mit getrennten Caches ausgeführt werden, können die geschriebenen Werte aus den Zeilen 1 und 3 noch in dem jeweiligen Cache feststecken, wenn bereits die nachfolgenden Leseoperationen der Zeilen 2 und 4 ausgeführt werden, die folglich beide

```
class A {
  void f() {}
}
A x = null, y = null;
```

Thread t1	Thread t2
1: `r1 = x;`	4: `r2 = y;`
2: `if (r1 != null) r1.f();`	5: `x = r2;`
3: `y = new A();`	

Abbildung 3: Ein Programm, bei dem eine Methode vor der Erzeugung des Zielobjekts aufgerufen werden kann.

den initialen Wert 0 aus dem Hauptspeicher lesen. Um SC zu garantieren, müsste der Compiler bzw. die VM zwischen den Schreib- und Lesebefehlen jeweils Synchronisationscode einfügen, der die Programmausführung über Gebühr verlangsamen würde.

Um auch komplexere Optimierungen bei der Übersetzung von Java-Programmen zu erlauben, kann es im JMM sogar passieren, dass Werte aus dem Speicher gelesen werden, die erst später zum ersten Mal erzeugt werden. Beispielsweise darf in Abb. 3 der Übersetzer Zeile 3 in Thread `t1` vor die Zeilen 1 und 2 vorziehen, so dass das neu erstellte Objekt Ziel des Methodenaufrufs in Zeile 2 sein kann (Ablauf 3, 4, 5, 1, 2); in der Semantik wird der Thread aber wie angegeben ausgeführt, also Zeile 2 immer vor 3. Um solche Effekte trotzdem zu erlauben, verwaltet das Schicht-1-Modul für das JMM nur den Allokationsstatus der Adressen und Typinformationen über die zu speicherden Werte, aber nicht die Werte selbst. Dies übernehmen drei weitere Semantik-Schichten: Nr. 4 generiert neue Ereignisse, die den Start und das Ende eines Threads markieren; Nr. 6 generiert die Menge aller möglichen Traces durch das Transitionssystem von Schicht 5, die alle Speicher- und Synchronisationsoperationen (einschließlich der Ereignisse aus Nr. 4) aufzeichnen; Nr. 7 wählt aus diesen Kandidaten die erlaubten Ausführungen aus. Die Auswahlkriterien sind der Kern des JMM, erst sie stellen sicher, dass die gelesenen und geschriebenen Werte der Speicherzellen zusammen passen. Um sogenannte Werte aus dem Nichts zu verhindern, reicht es jedoch nicht aus, einen einzelnen Trace zu betrachten; die Kriterien benötigen die Menge aller Kandidaten-Traces, um deren Teile untereinander in Beziehung zu setzen. Außerdem müssen auch unendliche Traces betrachtet werden, da das JMM unterstellt, dass aller Speicher bereits am Anfang einer Ausführung initialisiert wird – selbst wenn das entsprechende Objekt erst später zur Laufzeit erzeugt wird.

Um die Komplexität des Speichermodells einem durchschnittlichen Java-Programmierer nicht zumuten zu müssen, garantiert das JMM, dass korrekt synchronisierte Programme sich wie unter SC verhalten. Korrekte Synchronisation bedeutet dabei, dass keine SC-Ausführung einen Datenwettlauf enthalten darf, was immer durch ausreichende Synchronisation erreicht werden kann.

Theorem 2 (DRF-Garantie)**.** *Ist ein Programm korrekt synchronisiert, dann sind alle Ausführungen unter dem JMM nicht von den SC-Ausführungen unterscheidbar.*

Der Beweis dieses Theorems ersteckt sich über alle Schichten der Semantik, insbesondere verlangt er, dass ein sequenziell konsistentes Präfix eines Trace in sequenziell konsistenter Weise fortgesetzt werden kann. Da gezeigt wird, dass die Sonderbehandlung von Speicherinitialisierungen bei sequenziell konsistenten Präfixen ohne Bedeutung ist, lässt sich die Fortsetzung korekursiv konstruieren.

Frühere Formalisierungen der Auswahlkriterien und DRF-Garantie [AŠ07, HP07] hatten keine Verbindung mit einer operationalen Java-Semantik und konnten so die Komplikationen der Speicherinitialisierung ignorieren. Ebenso konnten sie die an die Trace-Menge gestellten Annahmen nicht nachweisen. Erst durch die neu geschaffene Verbindung mit der operationalen Semantik lässt sich diese Garantie also für konkrete Programme formal nutzen.

Ein wesentlicher Grund für die Komplexität des JMMs ist, dass es auch die Semantik von Programmen mit Datenwettläufen definiert. Dies ist notwendig, um die Typsicherheit und Sicherheitsarchitektur Javas zu garantieren. Deswegen beziehen sich die weiteren Untersuchungen dieses Abschnitts explizit auch auf Programme mit Wettläufen.

Zuerst wird gezeigt, dass das JMM jede SC-Ausführung erlaubt. Damit ergibt sich folgendes Verhältnis zwischen SC und dem JMM: Für wettlauffreie Programme sind die gleichen Ausführungen beobachtbar, bei Wettläufen erlaubt das JMM mehr Ausführungen als SC. Da immer eine SC-Ausführung existiert, folgt unmittelbar, dass das JMM in sich widerspruchsfrei ist – dies ist wegen der Komplexität und axiomatischen Art der Auswahlkriterien nicht offensichtlich.

Theorem 3 (Widerspruchsfreiheit)**.** *Das JMM erlaubt jede SC-Ausführung. Somit hat jedes Programm mindestens eine erlaubte Ausführung.*

Zum Thema Typsicherheit wird anhand neuer Beispielprogramme gezeigt, dass es zu Laufzeitproblemen kommen kann, wenn in Schicht 1 Typinformationen wie in der Literatur üblich in den angelegten Objekten und Arrays gespeichert werden. Beispielsweise kann in Abb. 3 das Ziel des Methodenaufrufs in Zeile 2 nicht bestimmt werden, wenn das Objekt an der referenzierten Adresse noch nicht angelegt worden ist. Dazu kann es wie oben beschriebenen durch Optimierungen während der Übersetzung kommen, die das JMM zwar erlaubt, die aber in der Semantik nicht vorweg genommen werden.

Außerdem lassen sich mit Datenwettläufen einer Variablen vom Typ T Referenzen auf Objekte eines beliebigen anderen Typs unterschieben, beispielsweise einem `String`-Zeiger eine Referenz auf ein `Integer`-Objekt; die Folge sind Zugriffsfehler, die Typsicherheit ist also verletzt. Obwohl die Auswahlkriterien eigentlich genau dies verhindern sollen, lässt sich mit Speicherallokationen ein subtiles Beispiel konstruieren, für das das JMM solche Ausführungen doch erlaubt. Als Lösung wird vorgeschlagen, den Adressraum statisch nach den zu speichernden Werten zu partitionieren und die Menge der Traces auf typ-korrekte Leseoperationen einzuschränken.[1] Dadurch unterscheidet sich eine Referenz auf ein `String`-Objekt *syntaktisch* von einer auf ein `Integer`-Objekt, so dass sie beim Vergleich verschiedener Traces in den Auswahlkriterien nicht mehr verwechselt werden können. Dann überträgt sich Typsicherheit (Thm. 1) auf das JMM, da beide obigen Szenarien nicht mehr möglich sind. Beispielsweise ist die Methodenaufrufzielauflösung unabhängig davon, ob das Zielobjekt bereits alloziiert wurde, da die Referenz selbst die für die Zielauswahl nötigen Informationen enthält.

[1] Durch die Einschränkung auf typkorrekte Leseoperationen wird das JMM „per Konstruktion“ typsicher. In [Loc12a] zeige ich, dass diese Einschränkung unnötig ist, da die Auswahlkriterien selbst nur Traces mit typkorrekten Leseoperationen erlauben – somit ist das JMM wirklich typsicher. Diese Einschränkung hilft bei der Standard-Modellierung in Schicht 1 nicht, da dann erlaubte Ausführungen, wie das Beispiel aus Abb. 3 zeigt, ausgeschlossen würden.

Eine leichte Abwandlung des zweiten Beispiels führt zu einem Angriff auf die Sicherheitsarchitektur Javas, die auf die Unveränderbarkeit von `String`s vertraut. Mittels Datenwettläufen lassen sich Referenzen fälschen, um so Zugriff auf die internen Datenstrukturen der `String`-Klasse zu erhalten. Die Einschränkungen bei Typsicherheit helfen hier nicht weiter, weil die gefälschte Referenz vom richtigen Typ ist und sich nicht syntaktisch von der richtigen unterscheidet. Allerdings handelt es sich hier nur um einen hypothetischen Angriff, da das JMM nur das erlaubte Verhalten in realen VMs einschränkt – mir ist bisher keine Optimierung bekannt ist, die zu dem Verhalten im Beispiel führen kann. Trotzdem zeigen diese Beispiele eine grundsätzliche Schwachstelle in der Spezifikation des JMMs auf, die bei einer künftigen Überarbeitung angegangen werden sollte.

4 Der Übersetzer

JinjaThreads formalisiert auch einen nicht-optimierenden Compiler *J2JVM*, der Quellcode-Programme in Bytecode übersetzt. Er verbindet – wie schon in Jinja – die beiden Sprachen zu einem einheitlichen Modell. Bei der Kompilation sind `synchronized`-Blöcke am schwierigsten, da die Blockstruktur des Quellcodes im Bytecode nicht mehr vorhanden ist. Dies betrifft die Registerallokation, da ein weiteres Register die Monitoradresse während der Ausführung des Blocks zwischenspeichern muss, und die Ausnahmebehandlung, da die Sperre auch bei einer ausgelösten Ausnahme durch generierten Behandlungscode freigegeben werden muss.

Das folgende Theorem fasst die Korrektheitsaussagen der Übersetzerverifikation zusammen:

Theorem 4 (Korrektheit des Übersetzers)**.** *Sei P ein wohlgeformtes JinjaThreads-Programm. Dann gilt folgendes:*

(i) Der Bytecode-Verifizierer akzeptiert das übersetzte Programm J2JVM P.

(ii) P und J2JVM P haben das gleiche Verhalten, d.h., jeder (erlaubte) Trace von P ist auch ein (erlaubter) Trace von J2JVM P und umgekehrt.

(iii) P ist korrekt synchronisiert genau dann, wenn J2JVM P es ist.

Während der erste Punkt lediglich eine Erweiterung des Beweises aus Jinja benötigt, lässt sich der alte Korrektheitsbeweis zum Verhalten nicht übertragen, da dieser auf der nicht mehr vorhandenen Big-Step-Semantik aufbaute. Stattdessen muss der Beweis nun gegen die SOS geführt werden. Der Bisimulationsbeweis wird dadurch wesentlich aufwändiger, da auch alle Zwischenzustände der Berechnungen zueinander in Beziehung gesetzt werden müssen und manche Konstrukte im Quell-, andere im Bytecode mehr Berechnungsschritte erfordern. Bisimulation stellt unter anderem auch sicher, dass der Übersetzer keine neuen Verhalten einführt. Beispielsweise ist zu zeigen, dass die generierte Entsperrinstruktion am Ende eines `synchronized`-Blocks niemals mit einer `NullPointerException` oder `IllegalMonitorStateException` fehlschlägt, da dies die Blockstruktur im Quellcode ausschließt.

Im Beweis zeigt sich wieder, wie wesentlich die Modularität der Semantik für die Handhabbarkeit ist: Da Quell- und Bytecode die gleiche Semantik für Nebenläufigkeit verwenden, genügt es, die Bisimulation für einen einzelnen Thread zu zeigen. Da die Interaktionen mit anderen Threads in den Aktionsanweisungen gekapselt werden, folgt daraus, dass auch die Kombination der Threads bisimilar ist. Somit braucht man sich im eigentlichen Beweis nicht um die Aktionen anderer Threads und den durch die verschränkte Ausführung erzeugten Nichtdeterminismus kümmern. Da der übliche Begriff der schwachen Bisimulation in diesem Fall für die Komposition nicht ausreicht, wurde eigens der Begriff der Verzögerungsbisimulation mit expliziter Divergenz entwickelt. Divergenz bezeichnet hier die Nichttermination eines Threads, der dabei aber keine beobachtbaren Aktionen ausführt. Der Einbezug von Divergenz stellt sicher, dass der Übersetzer die Terminationseigenschaften des Programms erhält und keine neuen Verklemmungen einführt.

5 JinjaThreads als Java-Interpreter

Angesichts des Umfangs und der Komplexität des Modells lässt nicht mehr einfach sicherstellen, dass es Java adäquat abbildet. Eine formale Verifikation ist hier nicht möglich, weil die Spezifikation von Java nicht formal ist. Stattdessen wurden mit Isabelles Codegenerator automatisch aus der Formalisierung ausführbare Implementierungen in Standard ML extrahiert. Dies liefert einen Interpreter, Übersetzer und eine virtuelle Maschine samt Bytecode-Verifizierer. Der Codegenerator generiert aus bewiesenen Gleichungen der Formalisierung ein funktionales Programm, dessen Ausführungschritte je einer Termersetzung in der Logik entsprechen. Dadurch gelten automatisch alle bewiesenen Theoreme auch für das Ergebnis der Berechnung, da die Schritte direkt in der Logik simuliert werden können. Das bedeutet insbesondere, dass Interpreter und VM typsicher und der Übersetzer verifiziert sind.

Nun kann das Modell validiert werden, indem Java-Programme geprüft, übersetzt, ausgeführt und das Ergebnis mit spezifizierten Erwartungen verglichen wird. Zuvor müssen aber die Programme in JinjaThreads' abstrakter Syntax übertragen werden, auf der alle drei arbeiten. Dazu wurde das Konvertierungstool Java2Jinja (`http://pp.ipd.kit.edu/projects/quis-custodiet/Java2Jinja/`) als Eclipse-Plugin entwickelt, das zusätzlich nicht unterstützte Aspekte wie beispielsweise innere Klassen und Generics, sofern möglich, emuliert. Außerdem bietet es eine Benutzerschnittstelle für die Übersetzung und Ausführung einzelner Java-Programme mit den generierten Komponenten sowie automatisierte Läufe ganzer Testsuites an.

JinjaThreads wurde mit Hinblick auf einfache Beweise und ohne Effizienzgedanken entwickelt. Deswegen sind drei Anpassungen nötig, um akzeptable Ausführungszeiten zu erreichen: (i) Ein eigenes Modul implementiert Schicht 1 mit effizienten, verifizierten Rot-Schwarz-Bäumen als sequenziell konsistenten Objektspeicher. (ii) Zum Programmstart werden Lookup-Tabellen für Deklarationen erzeugt, die dann während der Prüfung und Ausführung des Programms statt wiederholter Neu-Berechnungen verwendet werden. (iii) Ein Abwickler bestimmt den nächsten auszuführenden Thread, so dass nur eine statt aller Verschränkungen berechnet wird. Alle Anpassungen wurden korrekt bewiesen,

wobei neben der Flexibilität des Code-Generators die Modularität der Formalisierung entscheidend waren. Auf diese Weise wird der generierte Code in der Effizienz vergleichbar mit anderen Java Formalisierungen, die von Anfang an auf eine effiziente Ausführung hin entwickelt wurden. Gegenüber diesen bietet JinjaThreads den Vorteil, dass der Compiler verifiziert und Interpreter und VM typsicher bewiesen wurden. Für die Verarbeitung von 100 000 Objekten in einem klassischen Erzeuger-Verbraucher-Beispiel braucht die VM beispielsweise 63 s.

Die so erzielte Effizienz genügte, um 189 Testprogramme mit Java2Jinja und JinjaThreads zu bearbeiten, darunter 24 OpenJDK-Regressionstests. Bei der Validierung wurde lediglich ein Fehler in der Implementierung der Divisions- und Modulo-Operatoren für negative Ganzzahlen gefunden und korrigiert. Somit lässt sich sagen, dass JinjaThreads ein getreues Modell von nebenläufigem Java ist.

6 Zusammenfassung und Ausblick

JinjaThreads gehört mit 84 000 Zeilen Formalisierung, 848 Definitionen und 3 857 Lemmata zu den größten Anwendungen des Theorembeweisers Isabelle/HOL. Zum Vergleich, der sequenzielle Vorgänger Jinja begnügte sich mit 17 kLoC. Dem stehen aber auch der erweiterte Sprachumfang und eine detailgetreue Modellierung der Nebenläufigkeit einschließlich des JMMs entgegen. Auch hier wird wieder sichtbar, wie wichtig eine modulare Struktur ist: Beispielsweise entfallen von 13 kLoC für das JMM 10 kLoC auf die Formalisierung, der Nachweis der sprachspezifischen Annahmen des DRF-Beweises braucht nur 3 kLoC für Quell- und Bytecode zusammen.

JinjaThreads ist heute das umfassendste formale Modell von nebenläufigem Java einschließlich des JMMs. Entstanden ist es im Rahmen des Projekts Quis Custodiet (`http://pp.ipd.kit.edu/projects/quis-custodiet/`). Dort stellt es die semantische Grundlage für die Verifikation von Verfahren zur Informationsflusskontrolle (IFC) in nebenläufigen Java-Programmen. Dazu ist ein solches umfassendes und vereinheitlichtes Modell unabdingbar, da erst die Verbindung der verschiedenen Teile deren Interaktion korrekt erfasst und es erlaubt, die belastbaren Aussagen zur Theorie der Programmiersprache Java formal zu nutzen. Beispielsweise verlassen sich die IFC-Verfahren auf Typsicherheit und die DRF-Garantie; deren Verifikation kann also nur gelingen, da diese schon in dieser Arbeit nachgewiesen wurden. In diesem Kontext sind auch die ausführbaren Teile von Interesse, da sie sicherstellen, dass die durch die IFC-Verfahren nachgewiesenen Eigenschaften auch zur Laufzeit wirklich gelten.

Die aufgezeigten Probleme bei der Typsicherheit und Sicherheitsarchitektur werfen die Frage auf, wie diese Schwachstelle des JMMs behoben werden kann. In beiden Fällen sind Werte, die aus dem Nichts auftauchen, Ursache des Problems; der vorgestelle Ansatz rettet zwar die Typsicherheit, löst das Problem aber nicht abschließend. Zusammen mit den bereits bekannten Schwächen des JMMs deutet dies darauf hin, dass eine weitere Revision nötig ist. Es ist aber offen, wie eine überzeugende Antwort aussehen wird. Das Konzept des „Werts aus dem Nichts“ jedenfalls scheint als Kriterium zu unklar und allgemein zu sein –

wären die Annahmen der Sicherheitsarchitektur präzise formuliert, könnte man diese für das nächste JMM nachweisen.

Literatur

[AŠ07] David Aspinall und Jaroslav Ševčík. Formalising Java's Data-Race-Free Guarantee. In Klaus Schneider und Jens Brandt, Hrsgg., *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, Bd. 4732 von *LNCS*, Seiten 22–37. Springer, 2007.

[GJSB05] James Gosling, Bill Joy, Guy Steele und Gilad Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[HP07] Marieke Huisman und Gustavo Petri. The Java Memory Model: a Formal Explanation. In *Verification and Analysis of Multi-threaded Java-like Programs (VAMP 2007)*, Bericht ICIS-R07021, Seiten 81–96. University of Nijmegen, 2007.

[KN06] Gerwin Klein und Tobias Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[Loc07] Andreas Lochbihler. Jinja with Threads. *Archive of Formal Proofs*, 2007. `http://afp.sf.net/devel-entries/JinjaThreads.shtml`, Formal proof development.

[Loc12a] Andreas Lochbihler. The Java Memory Model is Type Safe. Bericht 2012-23, Karlsruhe Reports in Informatics, 2012.

[Loc12b] Andreas Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency : Language, Virtual Machine, Memory Model and Verified Compiler*. Dissertation, Fakultät für Informatik, Karlsruher Institut für Technologie, 2012.

[NPW02] Tobias Nipkow, Lawrence C. Paulson und Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, Bd. 2283 von *LNCS*. Springer, 2002.

[Pug00] William Pugh. The Java memory model is fatally flawed. *Concurrency: Practice and Experience*, 12:445–455, 2000.

Andreas Lochbihler, geboren 1981, studierte von 2001 bis 2006 Informatik mit Schwerpunkt mathematische Modellierung an der Universität Passau und während eines Auslandsjahrs an der University of Edinburgh. Nach dem Diplom-Abschluss mit Auszeichnung arbeitete er als wissenschaftlicher Mitarbeiter: bis März 2008 am Lehrstuhl für Softwaresysteme an der Universität Passau und danach bis 2012 am Lehrstuhl für Programmierparadigmen am Karlsruher Institut für Technologie. 2012 wurde er mit seiner in dieser Zeit verfassten Dissertation über die Semantik der Nebenläufigkeit in der Programmiersprache Java *summa cum laude* promoviert. Seit 2013 forscht er als Postdoktorierender am Institut für Informationssicherheit der ETH Zürich über die Sicherheit von Protokollen und entwickelt Ansätze zur Generierung fehlerfreier Protokoll-Implementierungen.