The Java programming language provides safety and security guarantees such as type safety and its security architecture. They distinguish it from other mainstream programming languages like C and C++. In this work, we develop a machine-checked model of concurrent Java and the Java memory model in the proof assistant Isabelle/HOL and investigate the impact of concurrency on these guarantees. From the formal model, we show how to automatically obtain an executable, verified compiler to bytecode and a validated virtual machine. Modularisation is the key to get a tractable and usable model; we carefully partition the definitions and proofs into modules that capture the interactions between sequential parts, concurrency, and the memory model.
Andreas Lochbihler

A Machine-Checked, Type-Safe Model of Java Concurrency
Language, Virtual Machine, Memory Model, and Verified Compiler
A Machine-Checked, Type-Safe Model of Java Concurrency

Language, Virtual Machine, Memory Model, and Verified Compiler

by
Andreas Lochbihler
A Machine-Checked, Type-Safe Model of Java Concurrency Language, Virtual Machine, Memory Model, and Verified Compiler

zur Erlangung des akademischen Grads eines Doktors der Naturwissenschaften
der Fakultät für Informatik des Karlsruher Instituts für Technologie (KIT)
genehmigte Dissertation
von
Andreas Lochbihler
aus Memmingen

Erster Gutachter: Prof. Dr.-Ing. Gregor Snelting
Zweiter Gutachter: Prof. Tobias Nipkow, PhD
# Contents

1 Introduction 1
1.1 Java concurrency 2
1.2 Historical overview 7
1.3 Contributions 8
1.4 Isabelle/HOL 13
1.4.1 Notation 14
1.4.2 Locales 17
1.4.3 Induction and coinduction 19

2 Sequential JinjaThreads 23
2.1 Source code 23
2.1.1 Abstract syntax 24
2.1.2 Type system 29
2.1.3 Native methods 34
2.1.4 Well-formedness 35
2.1.5 Dynamic semantics 37
2.1.6 Type safety 42
2.2 The JinjaThreads virtual machine 43
2.2.1 The bytecode language 44
2.2.2 Semantics 46
2.2.3 Well-typings 50
2.2.4 Type safety 54
2.3 Comparison with Jinja, Bali, and $\mu$Java 55

3 Interleaving semantics 59
3.1 Framework for interleaving semantics 60
3.1.1 The multithreaded state 61
3.1.2 Thread actions 66
3.1.3 Interleaving semantics 75
3.1.4 Infrastructure for well-formedness constraints 78
3.2 Multithreading in JinjaThreads 82
3.2.1 Native methods for synchronisation 82
3.2.2 Source code ........................................... 91
3.2.3 Bytecode ............................................. 96
3.3 Deadlock and type safety ............................... 100
  3.3.1 Deadlock as a state property .................... 101
  3.3.2 Deadlock for threads .............................. 105
  3.3.3 Progress up to deadlock ......................... 109
  3.3.4 Type safety for source code .................... 112
  3.3.5 Type safety for bytecode ....................... 122
3.4 Related work .......................................... 127
  3.4.1 Formalisations of Java and Java bytecode ...... 127
  3.4.2 Type safety proofs and deadlocks ............... 129
  3.4.3 Large-scale programming language formalisations 130

4 Memory models ....................................... 131
  4.1 The heap as a module ............................... 132
    4.1.1 Abstract operations and their properties ..... 133
    4.1.2 Adaptations to semantics and proofs ......... 136
    4.1.3 Design considerations .......................... 140
  4.2 Sequential consistency ............................. 141
  4.3 Java memory model .................................. 142
    4.3.1 Informal explanation ............................ 143
    4.3.2 Formal definition ............................... 148
    4.3.3 The data race freedom guarantee ............. 164
    4.3.4 Consistency ..................................... 185
    4.3.5 Type safety .................................... 188
    4.3.6 Discussion ..................................... 192
  4.4 Related work ....................................... 201
    4.4.1 Memory models and data race freedom ........ 201
    4.4.2 Abstract heap modules ......................... 203
    4.4.3 Modular formalisations ......................... 204

5 Compiler ............................................. 205
  5.1 Semantic preservation via bisimulation .......... 208
    5.1.1 Semantic preservation .......................... 208
    5.1.2 Simulation properties ........................... 210
    5.1.3 Lifting simulations in the interleaving framework 218
    5.1.4 Semantic preservation for the Java memory model 225
  5.2 Explicit call stacks for source code ............ 226
    5.2.1 State and semantics ............................ 227

x
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2.2</td>
<td>Semantic equivalence</td>
<td>232</td>
</tr>
<tr>
<td>5.3</td>
<td>Register allocation</td>
<td>236</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Intermediate language $J_1$</td>
<td>236</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Compilation stage 1</td>
<td>242</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Preservation of well-formedness</td>
<td>243</td>
</tr>
<tr>
<td>5.3.4</td>
<td>Semantic preservation</td>
<td>244</td>
</tr>
<tr>
<td>5.4</td>
<td>Code generation</td>
<td>250</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Compilation stage 2</td>
<td>250</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Preservation of well-formedness</td>
<td>251</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Semantic preservation</td>
<td>252</td>
</tr>
<tr>
<td>5.5</td>
<td>Complete compiler</td>
<td>255</td>
</tr>
<tr>
<td>5.6</td>
<td>Discussion</td>
<td>257</td>
</tr>
<tr>
<td>5.7</td>
<td>Related work</td>
<td>259</td>
</tr>
<tr>
<td>6</td>
<td>JinjaThreads as a Java interpreter</td>
<td>261</td>
</tr>
<tr>
<td>6.1</td>
<td>Isabelle code extraction facilities</td>
<td>262</td>
</tr>
<tr>
<td>6.1.1</td>
<td>The code generator</td>
<td>263</td>
</tr>
<tr>
<td>6.1.2</td>
<td>The predicate compiler</td>
<td>265</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Data structures</td>
<td>266</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Locales and code extraction</td>
<td>267</td>
</tr>
<tr>
<td>6.2</td>
<td>Static semantics</td>
<td>268</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Generic well-formedness</td>
<td>268</td>
</tr>
<tr>
<td>6.2.2</td>
<td>The bytecode verifier</td>
<td>269</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Type inference for source code</td>
<td>271</td>
</tr>
<tr>
<td>6.3</td>
<td>Interpreter and virtual machine</td>
<td>272</td>
</tr>
<tr>
<td>6.3.1</td>
<td>The single-threaded semantics</td>
<td>272</td>
</tr>
<tr>
<td>6.3.2</td>
<td>Schedulers</td>
<td>274</td>
</tr>
<tr>
<td>6.3.3</td>
<td>Tabulation</td>
<td>276</td>
</tr>
<tr>
<td>6.3.4</td>
<td>Efficiency of the interpreter</td>
<td>277</td>
</tr>
<tr>
<td>6.4</td>
<td>Guidelines for executable formalisations</td>
<td>280</td>
</tr>
<tr>
<td>6.5</td>
<td>The translator Java2Jinja</td>
<td>282</td>
</tr>
<tr>
<td>6.5.1</td>
<td>The translation</td>
<td>284</td>
</tr>
<tr>
<td>6.5.2</td>
<td>Validation</td>
<td>287</td>
</tr>
<tr>
<td>6.6</td>
<td>Related Work</td>
<td>289</td>
</tr>
<tr>
<td>7</td>
<td>Discussion and Future Work</td>
<td>291</td>
</tr>
<tr>
<td>7.1</td>
<td>Efforts and rewards of a machine-checked formalisation</td>
<td>291</td>
</tr>
<tr>
<td>7.2</td>
<td>Experience: Working with Isabelle/HOL</td>
<td>294</td>
</tr>
<tr>
<td>7.3</td>
<td>From Java$^{light}$ to JinjaThreads</td>
<td>299</td>
</tr>
</tbody>
</table>
7.4 Comparison between Java and JinjaThreads 301
7.5 Future work 305

8 Conclusion 307

A Producer-consumer example 311

B Formal definitions 315
B.1 Declarations and lookup functions 315
B.2 Binary operators 317
B.3 Heap module implementations 319
  B.3.1 Sequential consistency 319
  B.3.2 The Java memory model 321
B.4 Native methods 322
  B.4.1 Signatures 322
  B.4.2 Semantics of method clone 323
  B.4.3 Semantics of native methods 324
B.4.4 Observability 327
B.5 Generic well-formedness 327
B.6 Source code 328
  B.6.1 Syntax 328
  B.6.2 Typing rules for expressions 328
  B.6.3 Definite Assignment 330
  B.6.4 Well-formedness 332
  B.6.5 Small-step semantics 332
  B.6.6 Observability 338
B.7 Bytecode 339
  B.7.1 Syntax 339
  B.7.2 Applicability and effect 339
  B.7.3 The virtual machine 343
  B.7.4 Observability 348
B.8 The Java memory model 349
B.9 The compiler 352
  B.9.1 Program compilation 352
  B.9.2 Compilation stage 1 352
  B.9.3 Compilation stage 2 353
  B.9.4 Preprocessor 357

List of Figures 361
<table>
<thead>
<tr>
<th>Contents</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>365</td>
</tr>
<tr>
<td>Bibliography</td>
<td>367</td>
</tr>
<tr>
<td>Index</td>
<td>389</td>
</tr>
</tbody>
</table>
Abstract

Klein and Nipkow’s formalisation Jinja [83] of a Java-like programming language was the first that unifies source code, bytecode, and a compiler, is executable, and has been shown type safe – with Isabelle/HOL [128] having mechanically checked all definitions and proofs. In this thesis, I extend Jinja to JinjaThreads with concurrency in the form of Java threads and the Java memory model (JMM). Moreover, I transfer the existing theorems of type safety and compiler correctness, and prove the important JMM guarantee that data-race free programs behave like under interleaving semantics. Furthermore, I present the first formally-verified compiler for multithreaded Java.

JinjaThreads splits in two dimensions. On the one hand, like in Jinja, the compiler connects source code with bytecode on the level of languages. On the other hand, the semantics spans across different layers ranging from the implementation of the shared memory via the formalisation of the languages to the interleaving of threads and the axiomatic JMM. JinjaThreads is more than the sum of its parts, because it is their integration in a unified model that permits to correctly capture their interaction and to make reliable statements about the theory of the Java programming language.

Jinja has simplified Java in many places for clarity. In contrast, JinjaThreads investigates concurrency as described in the Java language specification in detail. On the language level, JinjaThreads covers dynamic thread creation, synchronisation via locks and monitors, wait and notify, interruption, and joining on threads. To obtain a tractable model, I have structured JinjaThreads in modules which encapsulate language-independent parts and which source code and bytecode share. For example, the interleaving semantics is parametrised over the single-threaded semantics and responsible for managing the thread pool, locks, interrupts, wait sets and notifications. By instantiating the parameters, I directly obtain the semantics for source code and bytecode. This modularity allows to formally define deadlock caused by synchronisation, which the type safety proof has to account for.

The second aspect of concurrency is the JMM. In this thesis, I connect its axiomatic specification with an operational semantics of Java for the
Abstract

first time. The intuitive memory model sequential consistency interleaves the individual steps of the threads and makes changes to memory immediately visible for all threads. In comparison, the JMM allows more executions such that compilers and the virtual machine itself may optimise more aggressively. Here, I prove – across all layers of the semantics – that for the important class of data-race free programs, the JMM allows only those intuitive executions that sequential consistency also allows. It is this link to an operation semantics that allows to formally apply this guarantee to concrete programs. Conversely, I also prove that the JMM is consistent by showing that it allows all (interleaved) executions that sequential consistency allows – even for programs with data races. Regarding type safety of Java with the JMM, I show that it depends on how type information is managed at runtime.

The JinjaThreads compiler connects source code with bytecode; its verification shows that both fit together. In particular, the compiler addresses the interaction between synchronisation and exceptions. Non-termination, intermediate output, and non-determinism constitute the challenges for the verification. Here, modularity of the model directly translates into manageable proofs. For example, I completely resolve the non-determinism at the level of interleaving semantics – independent of the language. Unlike for the semantics and type safety, I was not able to adapt the verification proofs of Jinja, because they were conducted against the big-step semantics of Jinja source code, which cannot express interleaving adequately.

Since JinjaThreads is a definitional artefact, one must argue that it faithfully models Java. In this case, formal verification is not possible, because Java is not specified formally. Instead, using Isabelle’s code generator, I have automatically extracted from the formalisation an interpreter, compiler, and virtual machine in Standard ML. Using them, I have validated the semantics of source code and bytecode by running and compiling a test suite of Java programs, which a conversion tool translated to JinjaThreads abstract syntax. To achieve reasonable execution times, the interpreter and the virtual machine use verified efficient data structures and formalised schedulers to resolve the non-determinism. This way, they perform as good as other formalised virtual machines for Java.

This work demonstrates that today, it is possible to build tractable models of sizeable programming languages in a theorem prover. JinjaThreads now provides the basis for verifying the program analyses for information flow control that our group is developing. Machine support
has been crucial, because it reliably detects the impact of changes and extensions on other parts of the model.
Zusammenfassung


Zusammenfassung


Der in dieser Arbeit entwickelte Übersetzer verbindet Quell- und Bytecode; seine Verifikation zeigt, dass beide Sprachen zusammenpassen. Bei der Kompilation ist die Herausforderung das Zusammenspiel der Synchronisationsprimitive mit Ausnahmen, bei der Verifikation sind es nicht terminierende Programme, Ausgaben und Nichtdeterminismus. Hier zeigt sich, wie wesentlich die Modularität der Semantik für die Handhabbarkeit der Beweise ist; beispielsweise lässt sich Nichtdeterminismus auf der Ebene der verschränkten Ausführung vollkommen sprachunabhängig auflösen. Anders als bei Semantik und Typsicherheit ließen sich dabei die alten Jinja-Beweise nicht anpassen, da diese in Bezug auf die Gesamtschrittsemantik der Quellsprache geführt wurden, die Nebenläufigkeit nicht adäquat ausdrücken kann.

Da JinjaThreads ein definitorisches Artefakt des Modellierungsprozesses ist, muss begründet werden, dass JinjaThreads Java adäquat abbilden. Eine formale Verifikation ist hier nicht möglich, weil die Spezifikation von Java nicht formal ist. Stattdessen wurde mit Isabell's Codegenerator vollautomatisch aus der Formalisierung ein Interpreter, ein Übersetzer und eine virtuelle Maschine für JinjaThreads-Programme in Standard ML extrahiert. Damit wurde die Semantik von Quell- und Bytecode durch eine Testsuite von Java-Programmen validiert, nachdem diese ein Konvertierungswerkzeug in die abstrakte Syntax von Jinja-
Threads übersetzt hatte. Um akzepetable Ausführungszeiten zu erzielen, verwenden der Interpreter und die virtuelle Maschine verifizierte effiziente Datenstrukturen und eigens formalisierte Abwickler zur Auflösung des Nichtdeterminismus. Damit sind sie ähnlich schnell wie andere formalisierte virtuelle Maschinen für Java.

Diese Arbeit zeigt, dass heutzutage umfangreiche Programmiersprachen in handhabbarer Form in Theorembeweisern abgebildet werden können. So stellt JinjaThreads jetzt die semantische Grundlage für die Verifikation der am Lehrstuhl entwickelten Programmanalysen zur Informationsflusskontrolle. Umgekehrt wäre diese Arbeit wegen der Komplexität des Modells ohne Maschinenunterstützung unmöglich gewesen, da nur so Auswirkungen von Änderungen und Erweiterungen auf andere Teile des Modells zuverlässig festgestellt werden konnten.
Acknowledgements

First, I would like to thank my advisor Prof. Gregor Snelting for his support, advice and the opportunity to pursue my own ideas without pressure. I also thank Prof. Tobias Nipkow for reviewing this thesis.

I would also like to thank my former and current colleagues in the Isabelle group at Passau and Karlsruhe, Daniel Wasserrab and Denis Lohner, for the numerous discussions ranging from technical issues with Isabelle to possible directions to pursue. Whenever I ran into a problem or had a sketchy idea, they never denied me to scribble inintelligible symbols and formulae on their whiteboards, but always helped to arrange my ideas and sort things out. Further, I thank Martin Hecker for sharing with me his understanding of the Java memory model. Neither must I forget to mention the discussions with all the other members of the group, which helped to set my views in perspective. In particular, they are Matthias Braun, Sebastian Buchwald, Andreas Zwinkau, and Manuel Mohr from the compiler group, and Christian Hammer, Dennis Giffhorn, Jürgen Graf, and Martin Mohr, who develop VALSOFT/Joana.

I am also indebted to the Isabelle developers in Munich, for answering my questions on the Isabelle mailing list and on the phone. In particular, I thank Stephan Berghofer and Florian Haftmann for introducing me to Isabelle’s code generator at TPHOLs 2008 in Montreal. They and Lukas Bulwahn set me on track for code generation and always helped to fix or circumvent its limitations. Jasmin Blanchette and his tool Nitpick have saved me from trying to prove wrong lemmata. The cooperation with Peter Lammich set the ground for extracting efficient code from JinjaThreads.

Furthermore, I thank the students Jonas Thedering and Antonio Zea, who developed the converter Java2Jinja and solved all the annoyances of Eclipse by themselves.

Finally, I thank Wolfgang Pausch, Denis Lohner, Martin Hecker, and Claudia Reinert for reading preliminary drafts of this thesis. Their comments helped to make the presentation more intelligible and focused.

The work in Chapters 2, 3, and 6 has been partially funded by the Deutsche Forschungsgemeinschaft grants Sn11/10-1 and Sn11/10-2.
Introduction

The Java programming language provides safety and security guarantees for all programs, which distinguish it from other mainstream programming languages like C and C++. Two are particularly important: type safety and Java’s security architecture. Type safety expresses that “nothing bad”, e.g., a segmentation fault, will happen during execution. The security architecture permits to execute untrusted code safely in a sandbox, i.e., without access to critical system resources [54].

Another distinctive feature of Java is its built-in support for multithreading and its semantics for executing threads in parallel [56, §17]. Yet, while it is well-known that multithreading non-trivially interacts with type safety and Java’s security guarantees [56, 145], their combination has never been considered formally.

In this thesis, I build a machine-checked model of Java concurrency called JinjaThreads for both Java source code and bytecode, and investigate the effects of multithreading on type safety and Java’s security guarantees. Moreover, I formalise a compiler from source code to bytecode and prove it correct. As the starting point of this work, I have used Jinja, a sequential Java-like language with compiler and type-safety proofs by Klein and Nipkow [83].

This work originates in the Quis custodiet (QC) project [147]. Using the proof assistant Isabelle/HOL [128], QC aims at mechanically verifying program analyses for information flow control (IFC) [53, 64, 65] that are developed in the VALSOFT/Joana project [173]. In QC, JinjaThreads defines the programming language and semantics against which program analyses like Wasserrab’s formalisation of program slicing [175–177] are verified. In the long term, QC aims to build a verified, trusted prototype for analysing and executing security-critical Java programs.
Chapter 1. Introduction

1.1 Java concurrency

For this thesis to be self-contained, this section gives a quick tour of the concurrency features of Java 6. Since Java itself is widely used today, I do not explicitly introduce sequential Java, but refer unfamiliar readers to the Java language specification (JLS) [56].

Java concurrency revolves around threads, i.e., parallel strands of execution with shared memory. A program controls a thread through its associated object of (a subclass of) class *Thread*. To spawn a new thread, one allocates a new object of class *Thread* (or any subclass thereof) and invokes its `start` method. The new thread will then execute the `run` method of the object, in parallel with all other threads. Each thread must be spawned at most once, every further call to `start` raises an *IllegalThreadStateException* exception. The thread terminates when `run` terminates, either normally or abruptly due to an exception. The static method `currentThread` in class *Thread* returns the object associated with the executing thread.

Java offers four kinds of synchronisation between threads: locks, wait sets, joining and interrupts. The package `java.util.concurrent` in the Java API [76] builds sophisticated forms of synchronisation from these primitives and atomic compare-and-set operations.

Every object (and array) has an associated monitor with a lock and a wait set. Locks are mutually-exclusive, i.e., at most one thread can hold a lock at a time, but re-entrant, i.e., a thread can acquire a lock multiple times [56, §17.1]. For locking, Java uses synchronized blocks that take a reference to an object or array. A thread must acquire the object’s lock before it executes the block’s body, and releases the lock afterwards. If another thread already holds the lock, the executing thread must wait until the other thread has released it. Thus, synchronized blocks on the same object never execute in parallel. The method modifier `synchronized` is equivalent to wrapping the method’s body in a synchronized block on the `this` reference [56, §8.4.3.6]. Java bytecode has explicit instructions for locking (`monitorenter`) and unlocking (`monitorexit`) of monitors. The major difference to synchronized blocks is that they can be used in unstructured ways; if the executing thread does not hold the lock, `monitorexit` fails with an *IllegalMonitorStateException* exception.

To avoid busy waiting, a thread can suspend itself to the wait set of an object by calling the object’s method `wait`, which class *Object* declares [56, §17.8]. To enter the wait set, the thread must have locked the object’s
monitor and must not be interrupted; otherwise, an `IllegalMonitorStateException` exception or `InterruptedException`, respectively, is thrown. If successful, the call also releases the monitor’s lock completely. The thread remains in the wait set until (i) another thread interrupts or notifies it, or (ii) if `wait` is called with a time limit, the specified amount of time has elapsed, or (iii) it wakes up spuriously. After having been removed, the thread reacquires the lock on the monitor before its execution proceeds normally or, in case of interruption, by raising an `InterruptedException`. The methods `notify` and `notifyAll` remove one unspecified or all threads from the wait set of the call’s receiver object. Like for `wait`, the calling thread must hold the lock on the monitor. Thus, the notified thread continues its execution only after the notifying thread has released the lock.

When a thread calls `join` on another thread, it blocks until (i) the thread that the receiver object identifies has terminated, or (ii) another thread interrupts the joining thread, or (iii) an optionally-specified amount of time has elapsed. In the second case, the call raises an `InterruptedException`; otherwise, it returns normally.

Interruption [56, §17.8.3] provides asynchronous communication between threads. Calling the `interrupt` method of a thread sets its interrupt status. If the interrupted thread is waiting or joining, it aborts that, raises an `InterruptedException`, and clears its interrupt status. Otherwise, interruption has no immediate effect on the interrupted thread. Instead, class `Thread` implements two methods to observe the interrupt status. First, the static method `interrupt` returns and resets the interrupt status of the executing thread. Second, the method `interrupted` returns the interrupt status of the receiver object’s thread without changing it.

Apart from that, class `Thread` also declares the methods `yield` and `sleep` [56, §17.9]. They instruct the scheduler to prefer other threads and cease execution for the specified time, respectively. Since these are only recommendations to the scheduler, they cannot be used for synchronisation.

Figure 1.1 shows some examples of synchronisation in a program with three threads that are run in parallel. However, it is prone to various forms of deadlock caused by locking, waiting and joining. Note that interruption alone cannot lead to deadlocks, because it is asynchronous.

For a start, suppose that threads t1 and t2 first acquire the locks on the shared objects p and q, respectively. Then, all threads are in deadlock for the following reasons: t1 must acquire the lock on q, which t2 is
Figure 1.1: Three Java threads with different deadlock possibilities

holding, but t2 itself must acquire the lock on p that t1 is holding, i.e., they are waiting for each other cyclically. Moreover, t3 is waiting for either t2 terminating or itself being interrupted, but t2 cannot terminate and there is no other thread which could interrupt t3.

A slightly different situation arises when t1 executes first until it suspends itself to q’s wait set. In terms of locks, it has first acquired p’s, then q’s twice, and finally released both on q, but it is still holding the lock on p. Hence, when t2 starts to execute, it can acquire q’s lock, but not p’s. Consequently, all threads are again in deadlock: t1 waits to be removed from the wait set, but none of the other threads could do so, because t2 waits for t1 releasing p’s lock and t3 waits for t2 to terminate or some other thread interrupting itself. Note that spurious wake-ups do not matter in this case. If t1 wakes up spuriously, then it must reacquire its locks on q first, i.e., t1 and t2 again end up waiting for each other.

Now suppose that t2 starts and is the first to acquire both locks q and p. Then, the call to notify has no effect since q’s wait set is empty. Thus, it releases p and q (in that order) and terminates. Hence, t3’s call to join terminates normally. Suppose further that t3’s if condition evaluates to false, i.e., t1 is not interrupted and t3 terminates. When t1 subsequently enters q’s wait set, it is immediately deadlocked, because there is no thread to remove it from the wait set. Note that in this case, there is only a single thread in deadlock, which is not possible if deadlock is due to locks and joins only. In case of a spurious wakeup, t1 terminates normally.

Finally, consider the same scenario again, but let t3 interrupt t1. Under this schedule, no deadlock occurs. If t3 calls interrupt before t1 calls wait, the latter call does not suspend t1 to q’s wait set, but raises the InterruptedException immediately. Otherwise, the interrupt removes t1 from q’s wait set. Then, t1 first reacquires the locks on q before it raises
the exception. In either case, t1’s synchronized blocks correctly release all locks despite the exception.

Beyond threads and synchronisation, Java also specifies how shared memory behaves under concurrent accesses, which is known as the Java memory model (JMM) [56, §17.4]. Let me sketch the main ideas behind the JMM with Figure 1.2. The program on the left has two threads, each of which sets one of C’s static fields x and y and subsequently reads the other into a local variable. Figure 1.2b shows three schedules for the program and for each schedule, the final values stored in the threads’ local variables. There are three further schedules, but they result in the same assignments to i and j. All these schedules assume sequential consistency (SC) [93], which is the most intuitive memory model: There is a global notion of time, one thread executes at a time, and every write to a memory location immediately becomes visible to all threads. In particular, the result \( i = j = 0 \) is impossible under SC as the following argument shows. If it was possible, then l. 1 must execute after l. 4 and l. 3 after l. 2. Since l. 1 and l. 3 literally precede l. 2 and l. 4, respectively, one obtains the contradiction that l. 1 executes after l. 4 after l. 3 after l. 2 after l. 1.

For efficiency reasons, modern hardware implements memory models that are weaker than SC to allow for local caches and optimisations [3,165]. For example, if threads t1 and t2 execute on different processors, the writes in ll. 1 and 3 might still be queued in the processors’ write buffers, when the reads in ll. 2 and 4 execute. Thus, the reads retrieve the initial values for C.x and C.y, i.e., 0, from main memory, which results in \( i = j = 0 \). Similarly, compiler optimisations might reorder the independent statements in each thread. Then, \( i = j = 0 \) is possible.
Chapter 1. Introduction

```java
static Object data;
volatile static boolean done = false;

thread t1            thread t2
1: data = ...;
2: done = true;
3: while (!done) {}
4: ... = data;
```

Figure 1.3: Synchronisation and publication of data through a `volatile` field

for the transformed program even under SC. Therefore, a correct implementation of SC must take extra precautions and conservatively disable such optimisations in all code, because the code does not provide any clues when it should do so. To avoid the ensuing slow-down, the JMM relaxes SC and allows the outcome `i == j == 0` in the example.

Nevertheless, the JMM provides the intuitive SC semantics under additional assumptions – known as the data-race freedom (DRF) guarantee [4]. Two accesses to the same location conflict if (i) they originate from different threads, (ii) at least one is a write, and (iii) the location is not explicitly declared as `volatile`. A data race occurs if two conflicting accesses may happen concurrently, i.e., without synchronisation in between. If the program contains no data races, the JMM promises that it behaves like under SC. In other words: If a programmer protects all accesses to shared data via locks or declares the fields as `volatile`, she can forget about the JMM and assume interleaving semantics, i.e., SC.

In the above example, there are two data races: the write of `C.y` in l. 1 races with the read in l. 4 and similarly l. 2 and l. 3 for `C.x`, i.e., the DRF guarantee does not apply. To eliminate these data races, one can use the synchronisation mechanisms from above, e.g., wrapping every line in its own synchronized block on `C`’s class object.

Alternatively, one can declare `C`’s static fields `x` and `y` as `volatile`, because accesses to such fields never conflict.¹ Since these fields are marked, Java implementations know when to take appropriate measures. Thus, programmers can use volatile fields to implement their own

¹When a thread reads from a volatile field, it synchronises with all other threads that have written previously to that field. Hence, the reading thread can be sure that everything that should have happened in the other threads prior to their writes in fact has happened prior to its read. For the formal semantics, see §4.3.2.
synchronisation mechanisms like in the example in Figure 1.3. After thread \( t_1 \) has finished the construction of the data object to be passed, it releases thread \( t_2 \) from spinning in l. 3 by setting the volatile flag \( \text{done} \). Volatile semantics of the JMM guarantees that thread \( t_2 \) sees the correct data, even though data itself is not volatile, i.e., no precautions slow down accesses to data.

Java also gives semantics to programs with data races, which is the main cause for the technical complexity of the JMM. This is essential, since malicious code could otherwise exploit data races to break type safety and Java’s security architecture. This semantics, however, is weaker than SC in that it allows more behaviours. Still, it is too strong, because it does not allow as many compiler optimisations as desired [38, 115, 162]. Conversely, it is unclear whether it is strong enough to ensure type safety and the Java security guarantees.

### 1.2 Historical overview

In the mid-1990s, Nipkow’s group started their work on the Bali project [11, 84] which lead to a comprehensive model (called Java[^light]) of the JavaCard language, a sequential subset of Java. They formalised the type system and a big-step semantics with a proof of type safety, and an axiomatic Hoare-style semantics that is shown sound and relatively complete with respect to the big-step semantics [129, 135–138, 156, 157]. At the same time, they studied the interaction between Java source code and bytecode for a smaller subset that was named \( \mu \text{Java} \) [85]. This line of work [24, 79–81, 86, 87, 126, 130, 146, 167, 168] lead to formal models of the virtual machine (VM), of the bytecode verifier, and to a compiler from source code to bytecode. These are complemented by proofs of type safety for source code and bytecode, and preservation of type correctness and semantics for the compiler.

Both Bali and \( \mu \text{Java} \) only consider sequential Java, although multi-threading has been envisioned as future work from the start [129]. An important step towards this goal was Jinja by Klein and Nipkow [83], because they developed a small-step semantics for Jinja source code that they proved equivalent to the big-step semantics. Additionally, they redesigned the type safety proof to use the small-step semantics, and they considerably slimmed down Bali and \( \mu \text{Java} \).
Chapter 1. Introduction

Thus, when I started to work on a formal semantics for concurrent Java in 2007, the choice for the sequential semantics was obvious: Jinja. Isabelle/HOL as the proof assistant was set, because the Quis custodiet project used it already and the semantics should become part of it – and Jinja was the most complete semantics of Java in Isabelle/HOL that featured a small-step semantics. A small-step semantics is crucial, because big-step semantics cannot express interleaving semantics adequately.

Of course, there have already been other formalisations of concurrent Java source code or bytecode [104,20] in other provers (see §3.4.1 for an in-depth discussion), which could have been ported to Isabelle. However, none of them had been used in large proofs about the semantics. Hence, it was unclear whether they would be easy to use in verifying the program analyses of the QC project. In contrast, Jinja had evolved over ten years and the type safety proof and compiler verification demonstrate its usability. Thus, it seemed reasonable to extend Jinja with concurrency.

In retrospect, I have not regretted this choice. For validating the semantics, it would have been better if Jinja had included all the Java features that Bali and µJava had already covered. Hence, I have reintroduced arrays and the full set of binary operators – see §2.3 for a detailed comparison.

Adding multithreading to a sequential language is pervasive, because almost every definition and every proof needs to be adapted. Although I have tried to reuse in JinjaThreads as much as possible from Jinja, it is more the general ideas and concepts that have survived than their literal formulation in Isabelle/HOL. Hence, JinjaThreads is incompatible with Jinja, but every Jinja program can be trivially transformed into a JinjaThreads program.

1.3 Contributions

The technical contributions of this thesis are the following:

- a model of Java threads for source code and bytecode (Chapter 3);
- proofs of type safety with deadlocks (§3.3);
- modular single-threaded semantics shared between SC and the JMM (Chapter 4);
- a proof of the DRF guarantee (§4.3.3), consistency (§4.3.4), and type safety (§4.3.5) of the JMM;
1.3. Contributions

- an example that the JMM corrupts Java security in theory (§4.3.5);
- a verified compiler from source code to bytecode (Chapter 5);
- an efficient, executable interpreter, virtual machine, type checker, and compiler that are extracted automatically from the formal model (Chapter 6); and
- validating the model by compiling and running Java programs in a test harness (§6.5).

The complete model and all proofs are formalised in the proof assistant Isabelle/HOL and available online [106] in the Archive of Formal Proofs.²

My model JinjaThreads covers all concurrency features from the Java language specification [56] except

- the methods stop, destroy, suspend, and resume in class Thread, as they are deprecated;
- timing-related features like timed wait and Thread.sleep, because JinjaThreads does not model time;
- the compare-and-swap operations for the java.util.concurrent package, since these are vendor-specific extensions of Java; and
- spurious wake-ups, because the JLS discourages VMs to perform those and they would obscure deadlocks. Standard Java coding practice circumvents this; see §4.3.6 for details.

The concurrency features that JinjaThreads covers are embedded in (an extension of) Jinja [83] by Klein and Nipkow, which I introduce in Chapter 2. The sequential features include classes with objects, fields, and methods, inheritance with method overriding and dynamic dispatch, arrays, exception handling, assignments, local variables, and standard control structures. Like its predecessor, JinjaThreads omits some sequential features from Java to remain tractable, e.g., static and final fields and methods, visibility modifiers, interfaces, class initialisation, and garbage collection. §7.4 contains the complete list, and in §6.5.1, I discuss how some of them can be emulated.

Thus, JinjaThreads is the first machine-checked model that unifies multithreaded Java source code, bytecode, and a compiler. In particular, JinjaThreads subsumes all of Jinja except for the big-step semantics and the proof of equivalence to the small-step semantics.

²In this thesis, I describe version e7d44e610544 in the archive. It works with Isabelle development version 915af80f74b3.
Chapter 1. Introduction

Figure 1.4: Structure of JinjaThreads in comparison with Jinja’s

Figure 1.4 shows the resulting structure of JinjaThreads. New parts are set in bold, adapted ones normally, and dropped ones in grey with dotted lines. The source code part defines the syntax, the type system, and a small-step semantics. The bytecode part formalises bytecode instructions, a virtual machine for individual threads in two equivalent flavours (aggressive and defensive), and a bytecode verifier. Both parts share some general infrastructure, the interleaving semantics and the JMM formalisation. The compiler translates source code into bytecode in two stages.

I prove type safety using the standard approach by Wright and Felleisen [180]. Subject reduction, i.e., preservation of well-typedness, easily carries over from Jinja. However, potential deadlocks severely complicate the progress theorem, which shows that execution does not get stuck. In fact, formalisations of type soundness for concurrent programming languages typically leave out the progress theorem or their notion of deadlock is implicit in the theorem’s assumptions, e.g., [57, 73, 94, 166, 169]. This way, one cannot be sure that the theorem’s
1.3. Contributions

Figure 1.5: JinjaThreads stack of semantics

<table>
<thead>
<tr>
<th>layer</th>
<th>source code</th>
<th>bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Java memory model</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>complete interleavings</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>interleaved small-step</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>thread start &amp; finish events</td>
<td></td>
</tr>
<tr>
<td></td>
<td>statements</td>
<td>call stacks</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>exception handling</td>
</tr>
<tr>
<td></td>
<td>expressions</td>
<td>single instruction</td>
</tr>
<tr>
<td>3</td>
<td>native methods</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>heap operations</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**concurrent semantics**

**single-threaded semantics**

The notion coincides with the intuitive understanding of deadlock, especially because deadlock can arise in many different ways (§1.1). In contrast, I formalise deadlock semantically (§3.3) and then prove type safety with respect to this notion.

Furthermore, JinjaThreads advances the state of the art in modelling concurrency. Previous formal semantics for multithreaded Java source code or bytecode [9, 14, 15, 20, 48, 70, 104, 166] stopped at interleaving semantics, i.e., sequential consistency. On the contrary, I formally connect the Java programming language with the Java memory model for the first time. Nevertheless, JinjaThreads models sequential consistency, too.

Here, separation of concerns and sharing of definitions and proofs are crucial to obtain a tractable model – not only between source code and bytecode, but also between the different memory models. To disentangle sequential aspects, the concurrency features, and the memory model from one another, I have built the semantics as a stack of seven layers (Figure 1.5). For example, to switch from source code to bytecode, one only needs to exchange layer 3, which defines the semantics of the language primitives. Analogously, the type safety proof at level 3 holds for both memory models, because they differ only in layers 1, 4, 6, and 7.

Furthermore, I have identified several previously unknown corner cases that the JMM misses and show how to deal with them. Moreover, I prove that the JMM indeed provides the DRF guarantee. Previous
proofs \([8,69]\) made assumptions about the sequential semantics, this work shows that these assumptions were justified. Regarding the other two promises of the JMM, namely type safety and Java’s security guarantees, the answers are less positive. Only a weak form of type safety holds, which excludes allocation of objects, i.e., the JMM allows Java programs to access unallocated objects (albeit in a type-correct fashion); and the JMM compromises Java’s security guarantees.

JinjaThreads also extends Jinja’s non-optimising compiler to handle the synchronisation primitives, and proves that it preserves semantics, well-typedness, and data race freedom of programs. Preservation of well-typedness is a straightforward extension of Jinja’s proofs, but semantic preservation requires a completely different approach, because Jinja used the big-step semantics, which no longer exists. In particular, verification must deal with the non-determinism of concurrency and different granularity of atomic operations. Using a bisimulation approach, I obtain a stronger correctness statement than Klein and Nipkow for Jinja, which also covers non-terminating executions. Again, JinjaThreads’s modular structure ensures that the result holds for both SC and the JMM. Thus, this is the first verified compiler for Java threads.

The various proofs about the semantics and the compiler demonstrate that JinjaThreads is indeed a tractable model, albeit large, and that today’s prover technology can handle such large models. Nevertheless, one must also make sure that it faithfully abstracts reality, i.e., Java. However, JinjaThreads’ size is beyond the point up to which good common sense suffices to convince oneself. Therefore, I have undertaken the effort to validate the model by executing smallish Java programs in both the source code and bytecode semantics. To that end, I have used Isabelle’s code generator to generate code for all definitions in grey boxes in Figure 1.4. Chapter 6 discusses the necessary steps and what the pitfalls were. This way, I have automatically extracted an executable well-formedness checker, interpreter, virtual machine and compiler for JinjaThreads programs from the Isabelle formalisation.

To make the vast supply of Java programs available for experimenting and testing with the semantics, I have developed together with the students Jonas Thedering and Antonio Zea the (unverified) conversion tool Java2Jinja\(^3\) as a plugin to the Eclipse IDE. It converts Java class

\(^3\)Java2Jinja is available for download at http://pp.info.uni-karlsruhe.de/projects/quis-custodiet/Java2Jinja/
declarations into JinjaThreads abstract syntax and provides a front-end to the well-formedness checker, interpreter and VM. Validation was not in vain, it discovered a bug in JinjaThreads’ implementation of the division and modulus operators (§6.5.2).

The size of the formalisation also poses a challenge for presentation. To keep the presentation intelligible, Chapter 2 starts with the sequential subset of JinjaThreads and omits everything that is related to multithreading. Then, I extend this subset with Java concurrency (Chapter 3) and the memory models (Chapter 4). This also demonstrates how Jinja evolved to JinjaThreads and what adaptations to the sequential semantics were necessary. Since I show most definitions only in excerpts or informally and change some of them multiple times, I have included the complete formal definition of the languages and semantics for source code and bytecode in Appendix B. Most of the proofs are only sketched or omitted completely, but they can be found in [106] with all the gory details of a machine-checked formalisation.

## 1.4 Isabelle/HOL

I have used the theorem prover Isabelle with higher-order logic (HOL) [128] as meta-language to formalise this work. Isabelle is able to check formalised definitions for being type-correct in the meta-language and formalised proofs for correctness. Although Isabelle offers sophisticated tools for proof automation, users must still decompose proofs into many small steps and guide the proof search. Yet, being an interactive proof assistant, Isabelle also supports the user in devising a formalised proof. For example, it correctly generates all non-trivial inductive cases for her and solves the trivial ones automatically. Conversely, Isabelle does not accept proofs of the form “analogous to . . . ” or “without loss of generality, . . . ” In such a case, the user must either repeat the proof or generalise it such that it works for all relevant cases. Thus, constructing elegant formal proofs still remains a business for experts. I have omitted most proofs in the presentation and only sketched the line of argument. Since I have written most proofs in the human-readable language Isar [25,179], the interested reader may consult the formalisation sources [106] for full details.

Despite Isabelle having formally checked all lemmas and theorems of this thesis, typing errors may have slipped in during typesetting.
Chapter 1. Introduction

Although Isabelle can in principle typeset definitions and theorems automatically to rule out such mistakes, I have transcribed all formulae in this thesis from the formalisation manually for two reasons. First, complex locale hierarchies (locales are Isabelle’s module system, §1.4.2) confuse Isabelle’s pretty-printer such that it loses track of pretty-printing syntax and outputs all fixed parameters, i.e., its output becomes unintelligible. Second, the presentation simplifies the formalisation in a few places for the sake of readability. For example, it glosses over some technical details such as trivial type coercions (see Footnotes 29 and 30). Chapters 2 and 3 present the definitions without the generalisations that later chapters add, although there is only one set of formal definitions with all extensions and generalisations. Consequently, I show how to adapt the simplified presentations in the later chapters. Appendix B contains the unsimplified definitions with all extensions.

1.4.1 Notation

The meta-language HOL mostly uses standard mathematical notation. This section introduces further notation and in particular some basic data types and operations on them.

Implication in Isabelle/HOL is written \( \to \) or \( \Rightarrow \) and associates to the right. Since the latter form stems from Isabelle’s environment for natural deduction, it separates the assumptions in proof rules, but cannot occur inside other HOL formulae. I abbreviate multiple assumptions by enclosing them in \([\ ]\) and \([\ ]\) with the separator “;”. Displayed implications are often printed as inference rules. For example, modus ponens is written \( P \to Q \to P \Rightarrow Q \) or \( [P \to Q; P] \Rightarrow Q \) or

\[
\begin{align*}
\frac{P \to Q}{P} & \quad P \\
& \quad Q
\end{align*}
\]

Biimplication \( P \iff Q \) is shorthand for \( P \to Q \) and \( Q \to P \).

The set of HOL types includes the basic types of truth values, natural numbers, integers and 32 bit machine words, which are called bool, nat, int, and word32, respectively. The space of total functions is denoted by \( \Rightarrow \). Type variables are written ‘a, ‘b, etc. \( t :: \tau \) means that the HOL term \( t \) has HOL type \( \tau \). To distinguish variables from defined constants, I typeset variables in italics (e.g., \( x, y, f \)) and defined names slantedly (e.g., \( x, y, f \)).
Pairs come with two projection functions \( \text{fst} :: \{a \times b \Rightarrow a \} \) and \( \text{snd} :: \{a \times b \Rightarrow b\} \). Tuples are identified with pairs nested to the right, i.e., \((a, b, c)\) is identical to \((a, (b, c))\) and \(a \times b \times c \) to \(a \times (b \times c)\). Dually, \(a + b\) denotes the disjoint sum of \(a\) and \(b\); the injections are \(\text{Inl} :: a \Rightarrow a + b\) and \(\text{Inr} :: b \Rightarrow a + b\). Records are tuples with labelled components, e.g., \([x = 1, y = 2]\).

Sets (type \(\{a\ \text{set} \})\) are isomorphic to predicates (type \(a \Rightarrow \text{bool}\)) with bijections \(\_ \in \_\) and \(\{x | x \_\_\}\), where \(x\) denotes the bound variable and can be omitted if equal to the preceding term, e.g., \(y \in \{x. P x\} \leftrightarrow P y\). \(\text{UNIV} :: \{a\ \text{set} \}\) is the set of all elements of type \(a\), \(\emptyset\) denotes the empty set. The image operator \(\text{f} \cdot A\) applies \(f\) to all elements of \(A\), i.e., \(\text{f} \cdot A = \{ f a | a \in A\}\). For example, \(\text{f} \cdot \text{UNIV}\) (written \(\text{range} f\)) denotes \(f\)'s range. The predicate finite on sets characterises all finite sets. The operator \(\cup :: \{a\ \text{set} \Rightarrow b\ \text{set} \Rightarrow (a + b)\ \text{set}\}\) denotes disjoint union on sets.

The definite description operator \(\exists x. P x\) is known as Russell’s \(\iota\)-operator. It denotes the unique \(x\) such that \(P x\) holds, provided exactly one exists. Hilbert’s \(\varepsilon\)-operator (indefinite description operator), written \(\varepsilon x. P x\), denotes one (fixed, but underspecified) \(x\) such that \(P x\) holds, provided \(P\) is satisfiable at all. Otherwise, both operators are unspecified.

Lists (type \(\{a\ \text{list} \}\)) come with the empty list \([\ ]\) and the infix constructor \(\cdot\) for consing. Variable names ending in “\(s\)” usually stand for lists. The function append, written \(\@\) as infix operator, concatenates two lists, \(|xs|\) denotes the length of \(xs\), and set converts lists into sets. If \(i < |xs|\), \(xs[i]\) denotes the \(i\)-th element of \(xs\), and \(xs[i := x]\) replaces the \(i\)-th element of \(xs\) with \(x\). Further standard operations on lists are available: hd \(xs\) returns \((\text{tl} \ x s\) removes) the first element of \((\text{from})\) \(xs\), and take \(n\) \(xs\) returns \((\text{drop} \ n \ x s\) removes) the first \(n\) elements of \((\text{from})\) \(xs\); replicate \(n\) \(x\) constructs the list \([x, x, \ldots, x]\) of length \(n\); rev \(xs\) reverses \(xs\); map \(f\) \(xs\) applies the function \(f\) to all elements of the list \(xs\); filter \(P \ x s\) with syntax \([x \leftarrow xs. P x]\) retains only elements from \(xs\) that fulfill \(P\); zip \(xs\) \(ys\) combines \(xs\) and \(ys\) elementwise into a list of pairs; foldl \(f \ a \ x s\) and foldr \(f \ x s\ a\) reduce the list \(xs\) with the binary operator \(f\) and start value \(a\), associating to the left and right, respectively; concat = foldl append \([\ ]\) concatenates a list of lists; distinct \(xs\) checks whether the elements in \(xs\) occur only once.

\begin{verbatim}
datatype 'a option = None | Some 'a
\end{verbatim}

adopts a new element None to \('a\), all existing elements in type \('a\) are also in \('a\ \text{option}\), but prefixed by Some. For succinctness, I write \([a]\)
for Some \( a \). For example, \( \text{bool option} \) consists of the three values \( \text{None} \), \( \text{[True]} \), and \( \text{[False]} \). The underspecified inverse \( \text{the} \) of \( \text{Some} \) satisfies \( \text{the} \) \( \lfloor x \rfloor = x \). Variables whose name ends in “o” usually have \( \text{option} \) type. \( \text{Option.map} \ f \) maps \( \text{None} \) to \( \text{None} \) and \( \lfloor x \rfloor \) to \( \lfloor f \ x \rfloor \).

Case distinctions on datatypes use guard-like syntax. For example, \( \text{case xo of None ⇒ a | \lfloor x \rfloor ⇒ f} \) pattern-matches on \( xo \). If \( xo \) is \( \text{None} \), it returns \( a \); if \( xo \) is \( \lfloor x \rfloor \), the result is \( f \) where \( f \) may refer to \( x \).

Function update is defined as follows: Let \( f :: \ 'a \Rightarrow \ 'b \), \( a :: \ 'a \), and \( b :: \ 'b \). Then, \( f(a := b) = \lambda x. \text{if } x = a \text{ then } b \text{ else } f \ x \).

As all functions in HOL are total, partial functions are modelled as functions of type \( 'a \Rightarrow \ 'b \text{ option} \) where \( \text{None} \) represents undefinedness and \( f \ x = \lfloor y \rfloor \) means that \( f \) maps \( x \) to \( y \). I abbreviate \( 'a \Rightarrow \ 'b \text{ option} \) by \( 'a \Rightarrow \ 'b \) and call such functions maps. The notation \( f(x \mapsto y) \) is shorthand for \( f(x := \lfloor y \rfloor) \), and it extends to lists: \( f(xs \mapsto ys) \) means \( f(x_0 \mapsto y_0) \ldots (x_i \mapsto y_i) \) where \( i \) is the minimum of \( |xs| - 1 \) and \( |ys| - 1 \). Multiple updates like \( f(x \mapsto y)(xs \mapsto ys) \) can be written as \( f(x \mapsto y, xs \mapsto ys) \). The everywhere undefined map \( \lambda_. \text{None} \) is written \( \text{empty} \). Updates of \( \text{empty} \) are written \( [x \mapsto y], [xs \mapsto ys], \) etc. The domain of \( f \) (written \( \text{dom} \ f \)) is the set of points at which \( f \) is defined, \( \text{ran} \ f \) denotes the range of \( f \). Function \( \text{map-of} \) turns an association list, i.e., a list of pairs, into a map:

\[
\text{map-of} \ [\ ] = \text{empty} \\
\text{map-of} \ ((x, y) \cdot xs) = (\text{map-of} \ xs)(x \mapsto y)
\]

The order \( m_1 \preceq_m m_2 \) on maps denotes that \( m_2 \)'s domain contains \( m_1 \)'s and \( m_1 \) and \( m_2 \) are equal on \( \text{dom} \ m_1 \). I say that \( m_1 \) is a restriction of \( m_2 \) and \( m_2 \) an extension of \( m_1 \).

\( \text{FinFuns} \ [109] \) (type \( 'a \Rightarrow_{f} \ 'b \)) are functions that are almost everywhere constant, which Isabelle’s code generator implements as associative lists (§6.1.3). For this thesis, one may treat them just like ordinary functions except for some notation. Function application and update are written \( f_f x \) and \( f(x :=_f y) \), respectively; \( K^f y \) denotes the FinFun that maps everything to \( y \). The decoration with subscripts \( f \) extends to other update notation for maps, e.g., \( f(x \mapsto_f y) \) denotes \( f(x :=_f \lfloor y \rfloor) \) and \( [x \mapsto_f y] \) abbreviates \( (K^f \text{None})(x :=_f \lfloor y \rfloor) \).
1.4.2 Locales

Locales [12] are Isabelle’s approach to modularisation. A locale declaration defines the signature of a context, which consists of the locale parameters with fixed types (fixes) and the assumptions about the parameters (assumes). It also defines a predicate with the locale’s name that collects all assumptions.

For example, the following locale monoid declares a module monoid whose parameters, i.e., abstract operations, are the binary operator ⊙ and the neutral element e. It has three assumptions: assoc states associativity and neutral states that e is left and right neutral – free variables in the assumptions (such as a, b, and c) are implicitly universally quantified. Note that the type of the elements of the monoid is the type variable 'a rather than an opaque type. This way, module implementations can instantiate 'a as needed.

```
locale monoid = fixes ⊙ :: 'a ⇒ 'a ⇒ 'a and e :: 'a
  assumes assoc: (a ⊙ b) ⊙ c = a ⊙ (b ⊙ c)
  and neutral: a ⊙ e = a e ⊙ a = a
```

A locale context collects declarations such as theorems and definitions, which may depend on the parameters and assumptions. If a definition or proof uses the module, but does not need to look at one concrete implementation, it goes in such a locale context. For example:

```
fun (in monoid) pow :: 'a ⇒ nat ⇒ 'a where
  pow x 0 = e
  | pow x (n + 1) = x ⊙ pow x n

lemma (in monoid) pow-plus: pow x (n + m) = pow x n ⊙ pow x m
```

Locale interpretations correspond to module implementations – they instantiate the parameters of a locale and discharge the assumptions. This specialises all collected declarations to the given parameters, optionally adding a name prefix. For example,

```
interpretation list: monoid @ [] ⟨⟨proof⟩⟩
```

interprets the monoid for lists. This yields the function list.pow and the lemma list.pow-plus, in which the type 'b list instantiates the type variable
'a for the monoid elements and @ and [] replace \( \odot \) and \( e \), respectively, i.e.,

\[
\text{list.pow } x \ 0 = [] \quad \text{list.pow } x \ (n + 1) = x @ \text{list.pow } x \ n
\]

\[
\text{list.pow } x \ (n + m) = \text{list.pow } x \ n @ \text{list.pow } x \ m
\]

To use or extend a module, locales can inherit from other locales where parameters may be renamed, specialised and names prefixed as necessary. From the module point of view, the inherited locale exports all the declarations it has collected. Here is an example for monoid homomorphisms:

```
locale monoid-hom = $m_1 \colon \text{monoid } \odot_1 \ e_1 + m_2 \colon \text{monoid } \odot_2 \ e_2$
  for $\odot_1 :: 'a \Rightarrow 'a$ and $e_1 :: 'a$ and $\odot_2 :: 'b \Rightarrow 'b$ and $e_2 :: 'b +$
  fixes $h :: 'a \Rightarrow 'b$
  assumes $h
eq_1 = e_2$ and $h(a \odot_1 b) = (h\ a) \odot_2 (h\ b)$

lemma (in monoid-hom) hom-pow: $h\ (m_1.pow \ x \ n) = m_2.pow \ (h \ x) \ n$
```

The locale `monoid-hom` imports the locale `monoid` twice and redeclares the inherited parameters in the for clause. To avoid ambiguities, references to the first (second) are prefixed with \( m_1 \) (\( m_2 \)) and the parameters renamed to \( \odot_1 \) and \( e_1 \) (\( \odot_2 \) and \( e_2 \)), respectively. Additionally, it fixes another parameter \( h \) and assumes that it is a monoid homomorphism. The lemma demonstrates how imported declarations are referenced.

JinjaThreads heavily uses locales to define interfaces between the different layers in the stack of semantics such that the development on higher levels can be reused by different implementations of the lower levels. For example (Chapter 3), the locale for interleaving semantics fixes a parameter for the single-threaded semantics. Then, source code and bytecode reuse the definitions and theorems for the interleaving semantics by interpretation. From the module point of view, source code and bytecode implement the module for single-threaded semantics on which the interleaving semantics builds.

To overcome incompatibilities between the code generator and the locale infrastructure, which I explain in §6.1.4, I have separated the declaration of parameters from the assumptions by splitting the locales whenever executability is a concern. Locales with names ending in `-base` only fix the parameters, the others inherit from them and add the assumptions.
1.4.3 Induction and coinduction

JinjaThreads heavily uses inductive and coinductive definitions, e.g., for the type system, small-step semantics, and complete interleavings. Since coinductive definitions and coinduction are less known than inductive ones, I contrast the former with the latter. Moreover, I briefly show how Isabelle’s automation supports such definitions and proofs.

Consider, for example, the following introduction rules of the reflexive and transitive closure (RTC) $r^{**}$ of a binary relation $r$.

\[
\begin{align*}
\text{r}^{**} a a \quad & \quad r^{**} a b \quad r b c \\
\end{align*}
\]

When one interprets them as an inductive definition, the binary predicate $r^{**}$ is defined as follows:

1. $r^{**} a a$ holds for any $a$, and
2. $r^{**} a c$ holds if $r^{**} a b$ already holds for some $b$ such that $r b c$, and
3. $r^{**}$ holds for nothing else – or, equivalently, $r^{**}$ is the strongest predicate satisfying 1 and 2.

The last clause is characteristic for inductive definitions. In particular, $r^{**}$ holds for concrete $x$ and $y$ only if one can prove $r^{**} x y$ in finitely many steps using the rules of (1.1).

Now, consider the coinductive definition of $r^{*}$ with the same introduction rules as for $r^{**}$:

\[
\begin{align*}
\text{refl} : & \quad r^{**} a a \\
\text{step} : & \quad r^{**} a b \quad r b c \\
\end{align*}
\]

This definition characterises $r^{*}$ as follows:

1. $r^{*} a a$ holds for any $a$, and
2. $r^{*} a c$ holds if $r^{*} a b$ holds for some $b$ such that $r b c$, and
3. $r^{*} a b$ is the weakest predicate with the property that whenever it holds, this can be justified by 1 or 2.\footnote{I use double horizontal bars for coinductive definitions to distinguish them from inductive ones.}

\footnote{Pierce calls this property “‘self-justifying’: every assertion in it [the predicate considered as a set] is justified by other assertions that are also in it” [143, §21.1].}
Chapter 1. Introduction

Figure 1.6: Example to illustrate the difference between inductive and coinductive definitions

Again, the last clause is characteristic for coinductive definitions. In particular, \( r_{**} \) does not hold for concrete \( x \) and \( y \) only if one can prove \( \neg r_{**} x y \) in finitely many steps.

To appreciate the difference between inductive and coinductive definitions, consider Figure 1.6. Let \( r \) be the binary predicate on a type with four elements \( a, b, c, \) and \( d \) as shown as a graph in Figure 1.6a, i.e., only \( r a b, r b d, \) and \( r c c \) hold. The inductive RTC \( r^{**} \) adds the missing loops at \( a, b, \) and \( d, \) and the transitive edge from \( a \) to \( d \) (Figure 1.6b). Figure 1.6c shows the coinductive RTC. It includes everything from the inductive RTC, but adds edges from \( c \) to every other element, too – even though \( c \) is disconnected from the rest of the graph. These edges are justified, because the loop at \( c \) permits to apply \( \text{step} \) infinitely often. Hence, one cannot prove that these edges do not belong to \( r_{**} \). When defining predicates coinductively, one must be careful to avoid such infinite recursion when it is not intended.

Isabelle/HOL provides a package to automate inductive and coinductive definitions [142] of predicates and sets. Given the introduction rules for a (co)inductive predicate, it computes the corresponding functional \( F \) over predicates and defines the predicate as the least (greatest) fixed point \( \text{lfp} \ F \) (\( \text{gfp} \ F \)) of \( F \). Next, the package proves monotonicity of \( F \) to ensure that the fixed point exists. Unless the premises involve user-defined operators that take as a parameter the predicate to be defined, this is completely automatic. Then, the package derives from the definition the specified introduction rules, a rule for case analysis, and a (co)induction rule.

The functional \( F \) for the introduction rules in (1.1) and (1.2) is the same, namely

\[
\lambda P \ x_1 \ x_2. \ (\exists a. \ x_1 = a \land x_2 = a) \lor (\exists a \ b \ c. \ x_1 = a \land x_2 = c \land P \ a \ b \land r \ b \ c)
\]
where the parameter $P$ abstracts the predicates $r^{**}$ and $r^{**}$ to be defined. Since $F$ is automatically proven monotone, $r^{**}$ is defined as $lfp \, F$ and $r^{**}$ as $gfp \, F$.

The induction rule (1.3) stems from the fact that $lfp \, F$ is the intersection of all $F$-closed binary predicates, where $P$ is $F$-closed iff $F(P) \, x \, y$ implies $P \, x \, y$. Hence, to prove that some property $P$ holds for all points $x$ and $y$ with $r^{**} \, x \, y$, it suffices to prove that $P$ is $F$-closed.

\[
\begin{align*}
r^{**} \, x \, y & \quad \forall a. \, P \, a \, a \\
& \quad \forall a \, b \, c. \, r^{**} \, a \, b \rightarrow P \, a \, b \rightarrow r \, b \, c \rightarrow P \, a \, c
\end{align*}
\] (1.3)

Dually to (1.3), the package derives the coinduction rule (1.4) for $r^{**}$ from the fact that $gfp \, F$ is the union of all $F$-consistent binary predicates, where $P$ is $F$-consistent iff $P \, x \, y$ implies $F(P) \, x \, y$.

\[
\begin{align*}
P \, x \, y & \quad \forall x_1 \, x_2. \, P \, x_1 \, x_2 \rightarrow \\
& \quad (\exists a. \, x_1 = a \land x_2 = a) \land \\
& \quad (\exists a \, b \, c. \, x_1 = a \land x_2 = c \land (P \, a \, b \lor r^{**} \, a \, b) \land r \, b \, c)
\end{align*}
\] (1.4)

Duality of (1.3) and (1.4) extends to how they are used in proofs: Induction (1.3) serves to prove properties $P$ of all elements of $r^{**}$; it eliminates the premise $r^{**} \, x \, y$ towards the goal $P \, x \, y$. Conversely, the coinduction rule (1.4) has the coinductive predicate $r^{**}$ in the conclusion. Hence, it can only establish that $r^{**}$ contains all elements that satisfy the ($F$-consistent) property $P$, but it is useless in deriving properties of all elements of $r^{**}$.

Sometimes, it is convenient to strengthen the coinduction rule such that one may delay to show the consistency requirement for $P$. To that end, $P$ is replaced by a family $Q \, k \, x \, y$ of predicates indexed over a well-founded relation $R$ (written $wf \, R$), where $k$ measures how often one may delay to show the consistency requirement. For $r^{**}$, the strengthened coinduction rule is

\[
\begin{align*}
wf \, R & \quad Q \, k \, x \, y \\
& \quad \forall k \, x_1 \, x_2. \, Q \, k \, x_1 \, x_2 \rightarrow \\
& \quad (\exists k'. \, (k', k) \in R \land Q \, k' \, x_1 \, x_2) \land \\
& \quad (\exists a. \, x_1 = a \land x_2 = a) \land \\
& \quad (\exists a \, b \, c. \, x_1 = a \land x_2 = c \land ((\exists k'. \, Q \, k' \, a \, b) \lor r^{**} \, a \, b) \land r \, b \, c)
\end{align*}
\]
Chapter 1. Introduction

The additional case $\exists k'. (k', k) \in R \land Q k' x_1 x_2$ in the last premise permits to defer showing consistency if one can descend in $R$ from $k$ to $k'$ without changing the elements. After one step in the consistency proof, one may freely choose the new index $k'$ in the last case. For examples of such strengthened coinduction rules, see Lemmata 4.16 and 5.2.

Proof. Isabelle does not automatically prove the strengthened coinduction rule, i.e., the user has to prove it manually for every coinductive definition for which she needs it. From the last premise, one proves by well-founded induction that the last premise of (1.4) holds for $P$ instantiated with $\lambda x y. \exists k. Q k x y$. The claim $r^{* *}_x y$ then follows directly with (1.4).

(Co-)Induction is the main proof principle for (co-)datatypes, too. Isabelle’s datatype package [26] defines algebraic datatypes. The package automatically derives the induction rule and a combinator for primitive recursion, which destructs the term towards a result. For example, the type of finite lists ‘a list is defined by

$$\textbf{datatype} \ 'a \text{ list } = [\ ] | 'a \cdot 'a \text{ list}$$

Again, coalgebraic datatypes (codatatypes) are dual: Users must construct them manually, e.g., [110, 142], and the corecursion operator constructs them. Possibly infinite lists (HOL type ‘a llist), e.g., have the corecursion operator

$$\text{llist-corec} a f = (\text{case } f a \text{ of } \text{None } \Rightarrow [\ ] \mid (x, a') \Rightarrow x \cdot \text{llist-corec} a' f)$$

It can be used, e.g., to define the iteration of a function $f$:

$$[b, f b, f (f b), f (f (f b)), \ldots ] = \text{llist-corec} b (\lambda a. [(a, f a)])$$

For a good introduction to corecursive definitions in Isabelle, see [142] by Paulson. The coinduction rule is used to prove such terms equal.
Bernard of Chartres used to say that we are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size.

John of Salisbury, *Metalogicon*

# Sequential JinjaThreads

In this chapter, I present the abstract syntax, type system, and semantics of sequential JinjaThreads including a proof of type safety, for both source code (§2.1) and bytecode (§2.2). In Chapters 3 and 4, I extend the sequential parts to Java concurrency. JinjaThreads also comes with a compiler and bytecode verifier, whose presentation I defer to Chapters 5 and 6, respectively, since this chapter focuses on the languages, type systems, and semantics.

As JinjaThreads builds on Jinja, I have taken the vast bulk of sequential features from Jinja, but made some adaptations and extensions. Klein and Nipkow have presented most of these features in detail [83], so I do not treat the sequential constructs in depth, here. Rather, this chapter introduces the main ideas and notation relevant for the rest of this thesis. Consequently, I present most definitions only in excerpts or informally. Appendix B contains the complete formal definitions, which include all generalisations of the later chapters. In §2.3, I compare sequential JinjaThreads to its predecessors Jinja, Bali, and μJava, and discuss the relevant changes.

## 2.1 Source code

This section describes the sequential part of JinjaThreads source code called $J$. First, I define the abstract syntax, covering values and types, expressions and statements, and program declarations with lookup operations on them (§2.1.1). The static semantics of the language is completed by the type system (§2.1.2) and well-formedness constraints (§2.1.4). In §2.1.5, I sketch the small-step semantics, focussing on native
methods. Finally, the type safety proof (§2.1.6) shows that static and
dynamic semantics fit together.

### 2.1.1 Abstract syntax

In this section, I present the abstract syntax for JinjaThreads source code,
bottom up. I start off with values and types, continue with expressions
and statements, and conclude with program declarations and lookup
functions on them. Abstract syntax falls into a generic part and one
specific to source code. This way, bytecode can reuse the generic parts
(see §2.2.1). JinjaThreads only defines an abstract syntax, but no concrete
input syntax. In §6.5, I discuss how to translate the concrete syntax of
Java into JinjaThreads abstract syntax.

There are HOL types $\text{cname}$ for class names, $\text{mname}$ for method
names, and $\text{vname}$ for variable names and field names. To make
the semantics executable (Chapter 6), all of them are isomorphic to strings
– as opposed to leaving them unspecified. For example, the special
reference $\text{this}$ to the current object is modelled as a local variable with
name “this”. In the sequel, I use the following variable conventions: $C$, $D$
for class names, $M$ for method names, $V$ for variable names, and $F$ for
field names. $v$ stands for values, $T$ for JinjaThreads types, $a$ for addresses
and $e$ for expressions.

#### Values and types

There are five kinds of JinjaThreads values (of HOL type $\text{val}$, see Fig-
ure 2.1): a dummy value $\text{Unit}$, booleans $\text{Bool } b$ where $b :: \text{bool}$, 32-bit
integers $\text{Intg } i$ where $i :: \text{word32}$, the null reference $\text{Null}$, and references
$\text{Addr } a$ where $a :: \text{addr}$. For the moment, I treat the type of addresses
$\text{addr}$ as an opaque type, the memory models in Chapter 4 add more
structure.

JinjaThreads types (of HOL type $\text{ty}$, see Figure 2.1) are the type
$\text{Void}$ for $\text{Unit}$, primitive types $\text{Boolean}$ and $\text{Integer}$, and three kinds of
reference types, namely $\text{NT}$ for the null reference, $\text{Class } C$ for classes,
and $\text{Array } T$ for arrays with element type $T$. $T[]$ is shorthand for $\text{Array } T$.
The predicate $\text{is-ref}\text{T}$ on types $\text{ty}$ tests for reference types.

The map $\text{typeof}$ statically assigns types to values, see Figure 2.1 for
the definition. References ($\text{Addr } a$) are statically not typable, because
2.1. Source code

**datatype** val = Unit | Bool bool | Intg word32 | Null | Addr addr

**datatype** ty = Void | Boolean | Integer | NT | Class cname | Array ty

```plaintext
typeof Unit      = ⌊ Void ⌋
typeof (Bool b)  = ⌊ Boolean ⌋
typeof Null      = ⌊ NT ⌋
typeof (Intg i)  = ⌊ Integer ⌋
typeof (Addr a)  = None
```

Figure 2.1: JinjaThreads values and types

their type may depend on the object or array that gets allocated at run time.

Expressions

JinjaThreads source code J is an imperative language where everything is an expression (HOL type expr) with a return value: statements are modelled as expressions that return Unit. Table 2.1 shows the sequential subset of expressions that JinjaThreads supports. It extends Jinja’s source code language by the following expressions (marked with ∗): (i) test on run time types with instanceof and (ii) array creation, array cell access and update, and reading the length of an array. Expressions whose syntax has been generalised are marked with ‡: casts are now possible between arbitrary types, not only classes; and local variable blocks carry an optional explicit initialisation vo :: val option. Figure 2.2 introduces abbreviations for frequent expressions.

The binary operator `bop` ranges over the binary operators (HOL type bop) known from Java: `==`, `!=`, `<`, `<=`, `>`, `>=`, `+`, `−`, `*`, `/`, `%`, `&`, `|`, `^`, `<<`, `>>`, and `>>>`. The two other binary operators in Java, short-circuit conjunction `&&` and disjunction `||`, are represented by the conditional operator:

```plaintext
e1 «&&» e2 = if (e1) e2 else false
e1 «||» e2  = if (e1) true else e2
```

To implement static binding of fields, field access and assignment are annotated with the class D that declares the field. Jinja provides a preprocessor [83, §2.6] that computes these annotations and converts unqualified field accesses into `Var this.F{D}`, where D is the class being processed. JinjaThreads extends this preprocessor (see Appendix B.9.4
### Table 2.1: Sequential JinjaThreads expressions

<table>
<thead>
<tr>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>new C</code></td>
<td>allocation of an object of class C</td>
</tr>
<tr>
<td><code>new T[e]</code></td>
<td>* allocation of an array with element type T</td>
</tr>
<tr>
<td><code>Cast T e</code></td>
<td>‡ checked cast of e to type T</td>
</tr>
<tr>
<td><code>e instanceof T</code></td>
<td>* check if e ≠ Null is assignment-compatible to T</td>
</tr>
<tr>
<td><code>Val v</code></td>
<td>literal value</td>
</tr>
<tr>
<td><code>e₁ «bop» e₂</code></td>
<td>binary operator</td>
</tr>
<tr>
<td><code>Var V</code></td>
<td>local variable access</td>
</tr>
<tr>
<td><code>V := e</code></td>
<td>assignment to local variable</td>
</tr>
<tr>
<td><code>e₁[e₂]</code></td>
<td>* array cell access</td>
</tr>
<tr>
<td><code>e₁[e₂] := e₃</code></td>
<td>* array cell assignment</td>
</tr>
<tr>
<td><code>e.length</code></td>
<td>* array length</td>
</tr>
<tr>
<td><code>e.F{D}</code></td>
<td>field access</td>
</tr>
<tr>
<td><code>e₁.F{D} := e₂</code></td>
<td>field assignment</td>
</tr>
<tr>
<td><code>e.M(es)</code></td>
<td>method call</td>
</tr>
<tr>
<td><code>{V : T = vo; e}</code></td>
<td>‡ local variable declaration with opt. initial value</td>
</tr>
<tr>
<td><code>e₁; ; e₂</code></td>
<td>sequential composition</td>
</tr>
<tr>
<td><code>if (e₁) e₂ else e₃</code></td>
<td>conditional</td>
</tr>
<tr>
<td><code>while (e₁) e₂</code></td>
<td>while loop</td>
</tr>
<tr>
<td><code>throw e</code></td>
<td>exception throwing</td>
</tr>
<tr>
<td><code>try e₁ catch(C V) e₂</code></td>
<td>exception handling</td>
</tr>
</tbody>
</table>

*Figure 2.2: Abbreviations for common expressions*

null = Val Null  unit = Val Unit
true = Val (Bool True)  false = Val (Bool False)
addr a = Val (Addr a)  Throw a = throw (addr a)
{V : T; e} = {V : T = None; e}

for the definition) to handle (i) the super qualifier for fields and (ii) array lengths like fields. Regarding the latter, the JLS specifies [56, §6.4.5] that array lengths be stored in a final field `length`. However, JinjaThreads provides the special syntax `e.length` for accessing array lengths, because Java bytecode and the Java memory model (JMM) treat array lengths specially anyway. Therefore, the preprocessor replaces the unannotated
2.1. Source code


datatype 'm prog = Program 'm cdecl list

type_synonym 'm cdecl = cname × 'm class

type_synonym 'm class = cname × fdecl list × 'm option mdecl list

record fdecl = vname × ty × fmod

record fmod = volatile :: bool

record fdecl = 'm mdecl = mname × ty list × ty × 'm

type_synonym J-mb = vname list × expr

type_synonym J-prog = J-mb prog

Figure 2.3: Type definitions for program declarations

access e.length[] by e.length if e's type is an array. This correctly implements that array length hides the field length of class Object if it declares such.

Program declaration

Figure 2.3 shows the type definitions for program declarations. A program declaration (of type 'm prog, variable convention P)\(^6\) is a list of class declarations (type 'm cdecl), each of which consists of the class name and the class itself. The class (type 'm class) declares its direct superclass, its fields and methods. A field declaration (type fdecl) is a tuple of field name, type and field modifiers. For field modifiers, JinjaThreads features only volatile. A method declaration (type 'm mdecl) consists of the method name, a list of the parameters' types, the return type, and the method body. Class declarations specialize 'm to 'm option to allow for optional method bodies. If the method body is None (written Native), only the signature of a native method is declared.

The method body is left as a type parameter 'm such that all JinjaThreads languages reuse this generic format for declarations. For the source code language, a method body consists of the list of formal parameter names and the expression itself (type J-mb). Then, a source code program has type J-prog, which plugs in J-mb for 'm in 'm prog.

---

\(^6\)In JinjaThreads, 'm prog is a type of its own (with injection Program) rather than an abbreviation like the other types for declarations and 'm prog in Jinja, because this permits data refinement of programs in Isabelle’s code generator (see §6.3.3). This is only of technical interest since 'm prog and 'm cdecl list are isomorphic.
Chapter 2. Sequential JinjaThreads

Although JinjaThreads requires certain system classes to work properly, it does not distinguish between them and user-defined classes. System classes are like ordinary classes except that JinjaThreads specifies their class names and every proper program declaration must define them. For the sequential part, the following system classes are needed: Object, Throwable, and the system exceptions

\[ \text{sys-xcpts} = \]
\[ [\text{NullPointerException, ClassCastException, ArithmeticException, OutOfMemory, ArrayIndexOutOfBoundsException, ArrayStore, NegativeArraySize}] \]  

(2.1)
The class Object is the root of the class hierarchy and Throwable the root of all exception and error classes that can be thrown. The names for the system exceptions are self-explanatory, their Java counterparts reside in the java.lang package.

Lookup functions

For most parts of JinjaThreads, the exact representation of programs is irrelevant. Instead, they access declaration information via the following lookup functions:

- **class** \( P \ C \) extracts the class with name \( C \) from \( P \).
- **is-class** \( P \ C \) predicates that \( P \) declares a class with name \( C \),
- \( P \vdash C \prec D \) denotes that \( C \) is a direct subclass of \( D \) in \( P \).
- \( P \vdash C \text{ sees } M:Ts\rightarrow T = \text{meth} \text{ in } D \) means that class \( C \) sees a method named \( M \) implemented in class \( D \), taking method overriding into account. \( Ts \) is the list of parameter types, \( T \) the return type, and \( \text{meth} :: \text{\'m option} \) the optional method body. For JinjaThreads source code, \( \text{meth} \) is either Native or of the form \([(pns, body)]\), i.e. the formal parameter names \( pns \) and the expression \( body \).
- \( P \vdash C \text{ has } F:T (fm) \text{ in } D \) denotes that in \( P \), the superclass \( D \) of \( C \) declares the field \( F \) with type \( T \) and field modifiers \( fm \).
- \( P \vdash C \text{ sees } F:T (fm) \text{ in } D \) is like \( P \vdash C \text{ has } F:T (fm) \text{ in } D \) except that \( F \) must also be visible in \( C \), i.e. there is no intervening declaration of \( F \) in the subclass hierarchy between \( C \) and \( D \) that hides the field \( F \) in \( D \).
2.1. Source code

The following example (in an imaginary concrete syntax) from [83] illustrates method overriding and field hiding:

```java
class B extends A {field F:TB
    method M:TBs→T1 = mB}
class C extends B {field F:TC
    method M:TCs→T2 = mC}
```

We have both $P \vdash C$ has $F$: $TC$ in $C$ and $P \vdash C$ has $F$: $TB$ ($fm$) in $B$, where $fm$ is $\{\text{volatile} = \text{False}\}$. In contrast, we have $P \vdash C$ sees $F$: $TC$ ($fm$) in $C$, but not $P \vdash C$ sees $F$: $TB$ ($fm$) in $B$, because the declaration of $F$ in $C$ hides the one in $B$.

Method overriding only considers names, not the declared types. Consequently, JinjaThreads does not allow method overloading. In the above example, we have $P \vdash B$ sees $M$: $TBs→T1 = [mB]$ in $B$ and $P \vdash C$ sees $M$: $TCs→T2 = [mC]$ in $C$, but not $P \vdash C$ sees $M$: $TBs→T1 = [mB]$ in $B$, because $M$’s definition in $C$ overrides the one from $B$. Moreover, the lookup functions need not take into account the effects of access modifiers like private and protected, because JinjaThreads omits those.

I do not present the formal definitions for these lookup functions here, because I have taken them almost unchanged from Jinja [83]. They can be found in Appendix B.1.

2.1.2 Type system

The base type of a type $T$ is $T$ with all Array constructors stripped off:

$$\text{base-type } T = (\text{case } T \text{ of } T'[] \Rightarrow \text{base-type } T' | _\emptyset \Rightarrow T)$$

Types are valid (predicate is-type) iff all classes they refer to exist in the program and, in case of array types, their base type is not $NT$ (see Figure 2.4). Array types with element type $NT$ are an artefact of the formalisation, they do not occur in Java programs as there is no syntax for the type $NT$. Treating them as invalid types also avoids other problems, e.g., infinite ascending chains in the subtype relation (see below). types $P$ denotes the set of all valid types for $P$.

Subtyping

In this section, I present subtyping on JinjaThreads types. The subclass relation $P \vdash _\emptyset \leq^* _\emptyset$ is the reflexive and transitive closure of the direct
is-type \( P \) Void = True \hspace{1cm} is-type \( P \) Boolean = True

is-type \( P \) Integer = True \hspace{1cm} is-type \( P \) \( NT \) = True

is-type \( P \) (Class C) = is-class \( P \) C

is-type \( P \) (\( T[] \)) = is-type \( P \) \( T \land \text{base-type } T \neq NT \)

types \( P \) = \{ \( T. \) is-type \( P \) \( T \) \}

**Figure 2.4:** Valid types of a program

\[ \leq_{REFL}: P \vdash T \leq T \hspace{1cm} \leq_{SUBCLS}: \frac{P \vdash C \leq^{*} D}{P \vdash \text{Class } C \leq \text{Class } D} \]

\[ \leq_{NULL}: P \vdash NT \leq \text{Class } C \hspace{1cm} \leq_{NULL}[]: P \vdash NT \leq T[] \]

\[ \leq_{OBJECT}: P \vdash T[] \leq \text{Class } Object \hspace{1cm} \leq_{ARRAY}: \frac{P \vdash T \leq T'}{P \vdash T[] \leq T'[]} \]

**Figure 2.5:** The subtype relation

subclass relation \( P \vdash _{-} \prec^{1} _{-} \). This induces a subtype relation \( P \vdash _{-} \leq _{-} \) on JinjaThreads types; Figure 2.5 shows the definition. The point-wise extension of \( \leq \) to lists of types is written \([\leq]\).

The rules for subtyping follow Java, e.g., the array type constructor \( _[] \) is covariant (\( \leq_{ARRAY} \)). Subtyping is reflexive, transitive, and, if the subclass hierarchy is acyclic, antisymmetric.

The subtype relation has infinite chains, both decreasing and ascending ones. For example, \( \leq_{ARRAY} \) gives in combination with \( \leq_{OBJECT} \) and \( \leq_{NULL}[] \):

\[ P \vdash \ldots \leq \text{Class } Object[][] \leq \text{Class } Object[] \leq \text{Class } Object \]

\[ P \vdash NT \leq NT[] \leq NT[][] \leq NT[][][] \leq \ldots \]

However, every infinite ascending chain, as I show in §6.2.2, must contain invalid types. The bytecode verifier (§6.2.2) cannot deal with infinite ascending chains, which has been the motivation for disallowing arrays with \( NT \) as element type.

Figure 2.6 shows the Hasse diagram for the subtype relation for the following program declaration. Invalid types have been omitted.
class Object {}
class A extends Object {}
class B extends Object {}
class C extends B {}

The left-hand side illustrates how the subclass relation is replicated for every number of array dimensions. The right-hand side shows that arrays of primitive types are subtypes of arrays of Object with one dimension less.
Chapter 2. Sequential JinjaThreads

**Figure 2.7:** Selected typing rules for JinjaThreads source code

The type system for J is modelled as type judgements of the form
\[ P, E \vdash e :: T \] where the environment \( E \) (of type \( vname \mapsto ty \), abbreviated as \( env \)) assigns types to local variables. \( P, E \vdash es [::] Ts' \) extends \( P, E \vdash _[..]_ :: _\) pointwise to lists of expressions and types. Figure 2.7 shows a subset of the rules that exhibit the key features, Appendix B.6.2 lists the full set.

Analogous to the abstract syntax, where \( e_1 \langle bop \rangle e_2 \) treats all 17 binary operators uniformly, \( WTbinop \) applies to all binary operators – the judgement \( P \vdash T_1 \langle bop \rangle T_2 :: T \) captures the individual differences. It denotes that the binary operator \( bop \) takes arguments of types \( T_1 \) and \( T_2 \) and the result is of type \( T \). Figure 2.8 shows some representative rules – the others are similar (Appendix B.2). \( WT/ \) for integer division is typical for integer operators. The equality operator requires the arguments’
2.1. Source code

\[
\begin{align*}
\text{WT/}: & \quad P \vdash \text{Integer «/» Integer :: Integer} \\
\text{WT==}: & \quad P \vdash T_1 \leq T_2 \lor P \vdash T_2 \leq T_1 \\
\text{WT&INT}: & \quad P \vdash \text{Integer «&» Integer :: Integer} \\
\text{WT&BOOL}: & \quad P \vdash \text{Boolean «&» Boolean :: Boolean}
\end{align*}
\]

Figure 2.8: Typing rules for the binary operators /, ==, and &

types to be comparable in the subtype relation (WT==), which imposes in particular equal types for primitive types. Overloaded operators like & and | have one rule for each case (WT&INT and WT&BOOL). Unlike in Java, + only operates on Integers, not on String, because Strings are not primitive in JinjaThreads.

Array cell assignment (WTaass), like any other assignment, requires that the type of the right-hand side is a subtype of the left-hand side’s.

Rule WTCALL deals with method calls. Since programs must explicitly declare native methods, it uniformly handles calls to native and non-native ones. It uses the partial function \text{class-of } T that returns the least class that is a supertype of \( T \) if \( T \neq NT \). It is defined as

\[
\text{class-of (Class } C) = [C] \quad \text{class-of (}\{\} \text{) } = [\text{Object}] \quad (2.2)
\]

and None for all other types. Hence, if the type \( T \) of the receiver is an array, method lookup starts in \( C = \text{Object} \). Thus, all arrays inherit the methods from class Object.\footnote{The JLS [56, §6.4.5] requires that arrays inherit all methods from class Object except for \text{clone}. Arrays (type \( T[] \)) must override the \text{clone} from Object to (i) increase its visibility from protected to public, (ii) remove the checked exception \text{CloneNotSupportedException} from the signature, and (iii) specialize the return type from \text{Object} to \( T[] \). As JinjaThreads neither models visibility nor checked exceptions, (i) and (ii) are irrelevant. However, JinjaThreads does not follow (iii) – which only the third edition of the JLS has introduced – because this special case would unnecessarily complicate definitions and proofs. Hence, JinjaThreads programs must explicitly cast the return value. For example, int[] a; a = new int[1].clone(); becomes [a : Integer[]; a := Cast (Integer[]) (new Integer[Val (Intg 1)].clone([]))] with an explicit cast.}

In WTCOND for conditionals, \( P \vdash \text{lub}(T_1, T_2) = T \) denotes that \( T \) is a least upper bound (lub) for \( T_1 \) and \( T_2 \) w.r.t. subtyping. Note that
lubs need not be unique at this point because $P$ might be an ill-formed program declaration with cycles in the class hierarchy.

In rules WT\text{throw} and WT\text{try} for exception throwing and handling, the constraint $P \vdash C \leq^* \text{Throwable}$ enforces that only subclasses of \text{Throwable} may be used as exceptions. WT\text{try} also illustrates the environment for local variables: The catch block implicitly declares the local variable $V$ of type \text{Class C} for the exception reference, which is modelled as the update of $V$ in $E$ in the premise for $e_2$. The constraint $P \vdash C \leq^* \text{Throwable}$ also ensures that the catch block refers to an existing class $C$ because \text{Throwable} is a system class which all well-formed programs must declare.

### 2.1.3 Native methods

Some methods that the standard Java API specifies cannot be implemented in Java syntax, e.g., \text{clone} and \text{hashcode} in class \text{Object}. If it were for the sequential part only, it would not be worth the effort to model native methods in JinjaThreads. However, calls to native methods implement most of Java’s concurrency features, e.g., \text{wait} in \text{Object} and \text{start} in \text{Thread}. Therefore, I added the infrastructure for including native methods in JinjaThreads.

A program must explicitly declare the native methods it uses, with \text{Native} as method body (see §2.1.1). This allows the typing rules and method overriding to treat native methods like normal methods, see WT\text{call} for an example. However, the semantics only provides a fixed set of native methods, because native methods by definition cannot be implemented in JinjaThreads syntax and thus must be hard-wired. To that end, the judgement $C.M(Ts) :: T_r$ expresses that the semantics provides a native method $M$ for class $C$ with parameter types $Ts$ and return type $T_r$. Well-formedness (§2.1.4) ensures that the semantics implements all methods that a program declares as native.

The sequential part defines only the two native methods \text{clone} and \text{hashcode} as follows, but I add more in §3.2.1:

\begin{equation}
\begin{align*}
\text{Object.clone([]} &:: \text{Class Object} \\
\text{Object.hashcode(>[])} &:: \text{Integer}
\end{align*}
\end{equation}

I defer the semantics of these native methods to the semantics section (§2.1.5).
2.1.4 Well-formedness

For most proofs, JinjaThreads programs must be well-formed. The well-formedness criteria fall in two categories: First, generic constraints such as acyclicity of the class hierarchy are independent of the concrete language. Second, language-specific constraints like well-typedness depend on the type of method bodies. To factor out the latter, the well-formedness test for programs takes a well-formedness test for method bodies as a parameter, ranged over by \( wf-mb: \)

\[
\text{type_synonym } m \text{ wf-mdecl-test } = m \text{ prog } \Rightarrow \text{cname } \Rightarrow m \text{ mdecl } \Rightarrow \text{bool}
\]

Generic well-formedness

The generic well-formedness predicates are shown in Figure 2.9. The function \( \text{classes} \), which is the inverse to \( \text{Program} \), returns the list of class declarations of a program. Hence, \( \map{\text{fst}}{\text{classes } P} \) computes the list of declared class names.

A program \( P \) is well-formed iff the system classes are as required (denoted by \( \text{wf-syscls } P \)), every class is declared only once, and all class declarations are well-formed. For the system classes, JinjaThreads requires class declarations for \( \text{Object} \) and \( \text{Throwable} \) and the latter must be a superclass of all system exceptions. This implicitly ensures that \( P \) declares all system exceptions.

A declaration of class \( C \) is well-formed (predicate \( \text{wf-cdecl} \)) iff all field and method declarations are well-formed (predicates \( \text{wf-fdecl} \) and \( \text{wf-mdecl} \)), no field or method is declared twice, and if \( C \) is not \( \text{Object} \), its direct superclass \( D \) exists and is no subclass of \( C \) (which rules out cycles in the subclass relation), and method overriding is contravariant in the parameters and covariant in the return type.

The predicate \( \text{wf-overriding} \) checks the last constraint. The overriding method may have less specific parameter types and a more specific return type. Note that method overriding considers only method names, not signatures, i.e., there is no overloading.

The predicate \( \text{wf-mdecl} \) checks that declared types are valid. For native methods, it checks that the semantics provides this method; for normal methods, it applies the language-specific well-formedness checker.
Chapter 2. Sequential JinjaThreads

\[
\text{wf-prog} :: \text{'m} \text{wf-mdecl-test} \Rightarrow \text{'m} \text{prog} \Rightarrow \text{bool} \\
\text{wf-prog} w f m d P \leftarrow \text{wf-syscls} P \land \text{distinct} (\text{map} \text{fst} (\text{classes} P)) \land \\
(\forall c d \in \text{set} (\text{classes} P). \text{wf-cdecl} w f m d P c d)
\]

\[
\text{wf-syscls} :: \text{'m} \text{prog} \Rightarrow \text{bool} \\
\text{wf-syscls} P \leftarrow \text{is-class} P \text{Object} \land \text{is-class} P \text{Throwable} \land \\
(\forall C \in \text{set sys-xcpts}. \text{P} \vdash C \preceq \ast \text{Throwable})
\]

\[
\text{wf-cdecl} :: \text{'m} \text{wf-mdecl-test} \Rightarrow \text{'m} \text{prog} \Rightarrow \text{'m} \text{cdecl} \Rightarrow \text{bool} \\
\text{wf-cdecl} w f m d P (C, D, f s, m s) \leftarrow \\
(\forall f d \in \text{set} f s. \text{wf-fdecl} P f d) \land \text{distinct} (\text{map} \text{fst} f s) \land \\
(\forall m d \in \text{set} m s. \text{wf-mdecl} w f m d P C m d) \land \text{distinct} (\text{map} \text{fst} m s) \land \\
(C \not= \text{Object} \Rightarrow \text{is-class} P D \land \neg \text{P} \vdash D \preceq \ast C \land \\
(\forall m d \in \text{set} m s. \text{wf-overriding} P D m d))
\]

\[
\text{wf-fdecl} :: \text{'m} \text{prog} \Rightarrow \text{cdecl} \Rightarrow \text{bool} \\
\text{wf-fdecl} P (F, T, f m) \leftarrow \text{is-type} P T
\]

\[
\text{wf-overriding} :: \text{'m} \text{prog} \Rightarrow \text{cname} \Rightarrow \text{'m} \text{cdecl} \Rightarrow \text{bool} \\
\text{wf-overriding} P D (M, Ts, T r, m) \leftarrow (\forall D' T s' T' r m'. \\
\text{P} \vdash D \text{sees} M: T s' \rightarrow T' r = m' \text{ in} D' \rightarrow \text{P} \vdash T s' \preceq \text{T} s \land \text{P} \vdash T r \preceq T' r)
\]

\[
\text{wf-mdecl} :: \text{'m} \text{wf-mdecl-test} \Rightarrow \text{'m} \text{option} \text{wf-mdecl-test} \\
\text{wf-mdecl} w f m d P C (M, Ts, T r, m) \leftarrow \\
\text{set} Ts \subseteq \text{types} P \land \text{is-type} P T r \land \\
(\text{case} m \text{of Native} \Rightarrow C: M(T s) :: T r \\
| [\text{mb}] \Rightarrow \text{wf-md} P C (M, Ts, T r, mb))
\]

**Figure 2.9:** Generic well-formedness constraints

\[
\text{wf-J-mdecl} :: \text{J-mb} \text{wf-mdecl-test} \\
\text{wf-J-mdecl} P C (M, Ts, T r, (pns, body)) \leftarrow \\
|Ts| = |pns| \land \text{distinct} pns \land \text{this} \notin \text{set} pns \land \\
(\exists T. P, [\text{this} \mapsto \text{Class} C, pns [\mapsto] T s] \vdash \text{body} :: T \land \text{P} \vdash T \preceq T r) \land \\
D \text{body} [\{ \text{this} \} \cup \text{set} pns]
\]

**Figure 2.10:** Well-formedness for JinjaThreads source code
2.1. Source code

type_synonym  J-state = heap × locals

well-formed source code

Let me now turn to well-formedness for J programs (Figure 2.10). To that end, \( \text{wf-prog} \) gets instantiated with the checker \( \text{wf-J-mdecl} \) for J method declarations, abbreviated as \( \text{wf-J-prog} \). I have taken the definition unchanged from Jinja [83]. It checks for

parameter names They must be equally many as parameter types, pairwise different and different from the this pointer.

typability The method body must be typable with a subtype of the declared return type.

definite assignment Definite assignment requires that during evaluation, whenever a local variable is read, it must have been assigned to before. The function \( D e A \) syntactically checks if \( e \) accesses only initialised variables if run from a state in which all variables in \( A \) have already been initialised. I have taken this test unchanged from Jinja with straightforward adaptations for the language extensions. Hence, I do not present the details here, Appendix B.6.3 shows the formal definition. For how it works, see [83].

2.1.5 Dynamic semantics

Having described the static semantics of J, I now turn to the dynamic semantics of the sequential subset.

State

The state (type J-state) is a pair of a heap (of type heap) for objects and arrays and a store for local variables (of type locals), see Figure 2.11. For
now, the heap maps addresses to heap entries, which can either be objects or arrays. Objects Obj C fs store their class name C and a field table fs which maps pairs (F, D) to values. It is essential that the table’s keys include the class D that declares the field F, because an object may have multiple fields of the same name. Arrays Arr T fs cs have an element type T, a field table fs, and a list of cells cs each of which contains a value. The length of the cell list determines the array length. In Java, arrays inherit all fields of Object [56, §6.4.5], so arrays need the field table for them. The declaring class is irrelevant in this field table, but I keep it for uniformity. For notation, h ranges over heaps, xs over local variables, and s over states. The functions hp and lcl extract the heap and the store from a state, respectively.

To simplify the semantics, JinjaThreads allocates one object for each system exception on the initial heap start-heap at start-up time. The function addr-of-sys-xcpt C returns the address of the preallocated object for the system exception C. As the exact definition is not relevant at the moment, I defer it to §4.1.

Statically, addresses have no type (cf. Figure 2.1). Dynamically, type information for an address (HOL type hty) consists of the class name or – if it is an array – of the type of its elements and its length. The accessor functions ty-of and array-length-of extract the type and array length.

\[
\text{datatype } hty = \text{ClassT } cname | \text{ArrayT } ty \text{ nat}
\]
\[
\begin{align*}
\text{ty-of} (\text{ClassT } C) &= \text{Class } C \\
\text{ty-of} (\text{ArrayT } T \text{ n}) &= T[] \\
\text{array-length-of} (\text{ArrayT } T \text{ n}) &= n
\end{align*}
\]

The function typeof-addr :: heap ⇒ addr ↦ hty computes the type information from the heap:

\[
\text{typeof-addr } h \ a = (\text{case } h \ a \ of [\text{Obj } C \_] ⇒ [\text{ClassT } C] \\
| [\text{Arr } T \_ \ cs] ⇒ [\text{ArrayT } T | cs|] \\
| \text{None} ⇒ \text{None})
\]

Then, typeof\_h extends typeof to addresses accordingly:

\[
\text{typeof}_h \ v = (\text{case } v \ of \text{Addr } a ⇒ \text{Option.map ty-of (typeof-addr } h \ a) \\
| _ ⇒ \text{typeof } v)
\]


\[\text{In Chapter 4, I will revisit and change the heap model to accommodate different notions of concurrency.}\]
2.1. Source code

\[\text{Clone:}\]
\[
\begin{align*}
\text{new-Addr } h = & \lfloor a' \rfloor \\
\implies & h' = h(a' := h a) \\
\end{align*}
\]
\[\frac{P \vdash \langle a.\text{clone}([],) h \rangle \to_{\text{nc}} \langle \text{Ret-Val } a', h' \rangle}{\text{new-Addr } h = \text{None}}\]

\[\text{CloneF:}\]
\[
\frac{P \vdash \langle a.\text{clone}([],) h \rangle \to_{\text{nc}} \langle \text{Ret-sys-xcpt OutOfMemory}, h \rangle}{\text{new-Addr } h = \text{None}}\]

\[\text{NCHashc:}\]
\[
\frac{P \vdash \langle a.\text{hashcode}([],) h \rangle \to_{\text{nc}} \langle \text{Ret-Val } (\text{Intg } (\text{hash-addr } a)), h \rangle}{\text{new-Addr } h = \text{None}}\]

Figure 2.12: Semantics of native method calls in the sequential part

Native methods

I now turn to the semantics for native methods. The sequential language defines only two, namely hashcode and clone (2.3), but I add more in §3.2.1 for Java concurrency. Native method calls may either return a value \(v\) (denoted Ret-Val \(v\)) or the address \(a\) of an exception (written Ret-Xcp \(a\)). I abbreviate Ret-Xcp (addr-of-sys-xcpt \(C\)) as Ret-sys-xcpt \(C\).

Native method calls execute in one step: \(P \vdash \langle a.M(vs), h \rangle \to_{\text{nc}} \langle vx, h' \rangle\) denotes that calling \(M\) on address \(a\) with parameters \(vs\) and the heap \(h\) returns a value or exception address \(vx\) and the updated heap \(h'\). Figure 2.12 shows the definition. The partial function new-Addr returns a fresh address for a given heap, if there is one, and None otherwise. Clone copies the object at \(a\) to the fresh address \(a'\) in one go, CloneF raises the OutOfMemory exception if all addresses are already in use. NCHashc computes the object’s hash using the function hash-addr :: addr \(\Rightarrow\) word32. Again, the concrete definitions for new-Addr and hash-addr are irrelevant until Chapter 4.

Binary operators

The evaluation of binary operators is defined via a partial function binop :: bop \(\Rightarrow\) val \(\Rightarrow\) val \(\rightarrow\) bop-ret, which returns either the evaluated value or the address of an exception.

\[
\text{type_synonym } \text{bop-ret} = \text{val} + \text{addr}
\]

In case of type mismatches, it returns None to denote undefinedness. Essentially, it unpacks the values and applies the appropriate operator on bool or word32. Figure 2.13 shows the definitions for the binary
Chapter 2. Sequential JinjaThreads

operators `/`, `==`, and `&`, see Appendix B.2 for the others. Division `/` (and analogously the remainder operation `%`) tests whether the divisor is 0, and if so, returns the preallocated `ArithmeticException`. Contrary to the typing rules in Figure 2.8, the equality test `==` omits any (unnecessary) type checks, infeasible cases are mapped to `[Inl (Bool False)]`. Overloaded operators like `&` are defined for more than one combination of value constructors. `binop `/`, `binop ==`, and `binop &` return `None` for all value combinations not shown in Figure 2.13.

Small-step semantics

The core semantics for \( J \) is a small-step semantics written \( P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \). I say that the expression \( e \) reduces in state \( s \) to \( e' \) and state \( s' \). It is a standard small-step semantics with rules for subexpression reduction and exception propagation. An expression is final, i.e., fully reduced, when it is either a value `Val v` or a thrown exception `Throw a`. For example, Figure 2.14 shows all rules for reducing method calls, which demonstrate the main ideas. For an extensive discussion, see [83]. The formal definition, with all extensions due to concurrency, can be found in Appendix B.6.5.

Rules `Robj` and `Rparam` reduce the subexpressions of the call – \( P \vdash \langle \_, \_ \rangle \rightarrow \langle \_, \_ \rangle \) lifts \( P \vdash \langle \_, \_ \rangle \rightarrow \langle \_, \_ \rangle \) to parameter lists of expressions. If one of the subexpressions throws an exception, rules `RobjX` and `RparamX` propagate it. For the other expressions, similar rules exist for subexpressions that need to be evaluated. These rules also determine the order of evaluation: The object is evaluated before the parameters because `Rparam` and `RparamX` are only applicable when the object is already a value.
2.1. Source code

\[ P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \]
\[ P \vdash \langle e.M(es), s \rangle \rightarrow \langle e'.M(es), s' \rangle \]

\[ P \vdash \langle es, s \rangle [\rightarrow] \langle es', s' \rangle \]
\[ P \vdash \langle Val v.M(es), s \rangle \rightarrow \langle Val v.M(es'), s' \rangle \]

\[ P \vdash \langle Throw a.M(es), s \rangle \rightarrow \langle Throw a, s \rangle \]
\[ P \vdash \langle Throw a.M(map Val vs @ Throw a \cdot es), s \rangle \rightarrow \langle Throw a, s \rangle \]
\[ P \vdash \langle null.M(map Val vs), s \rangle \rightarrow \langle THROW NullPointer, s \rangle \]

\[ P \vdash \langle addr a.M(map Val vs), s \rangle \rightarrow \langle blocks (this \cdot pns) (Class D \cdot Ts) (Addr a \cdot vs) body, s \rangle \]
\[ P \vdash \langle addr a.M(map Val vs), (h, xs) \rangle \rightarrow \langle e', (h', xs) \rangle \]

**Figure 2.14:** Small-step semantics for reducing method calls

If the receiver evaluates to the null pointer `null`, the preallocated `NullPointer` exception is thrown (R\text{CallN}), where `THROW C` abbreviates `Throw (addr-of-sys-xcpt C)`. `map Val vs` expresses that the parameter list has completely reduced to a list of values.

R\text{Call} and R\text{Native} are the main rules for calling a normal and native method, respectively. If the called method `M` for the receiver is not native, R\text{Call} looks up the method definition in `P` and inlines the method body; the function `blocks Vs Ts vs e` surrounds `e` with local variable blocks for variables names `Vs` with types `Ts` and initial values `vs`. Dynamic inlining avoids the need for modelling the call stack explicitly; the local variable blocks ensure static binding for the `this` pointer and parameter names. Conversely, R\text{Native} dispatches the call to the semantics for native methods. The function `native-Ret2J` injects the result `vx` into
Chapter 2. Sequential JinjaThreads

expressions as follows: `Val v` for return values `Ret-Val v` and `Throw a` for exception addresses `Ret-Xcp a`. Similar to the partial function `class-of` on `ty`, the total function `class-of'` on `hty` determines the class at which method lookup starts, i.e., `Object` for `ArrayT _ _` and `C` for `ClassT C`.

The semantics is strict in the sense that it gets stuck when values and types are not as expected, e.g., if the called method does not exist or requires a different number of parameters. The progress theorem (Theorem 2.1) shows that the type system rules out such cases.

Typically, the semantics of a program results from chaining together the individual steps of the semantics. However, as I aim for concurrency, I stop with the small-step semantics for now. Chapter 3 completes the semantics.

### 2.1.6 Type safety

In this section, I sketch the type safety proof for the sequential part of J. It takes the traditional form of two theorems, progress and preservation [180]. Progress expresses that any well-typed expression can reduce unless it is final. Preservation means that well-typed expressions reduce only to well-typed expressions whose type may only become more specific in the subtype relation. The proofs require the following invariants:

Conformance expresses that semantic objects conform to their syntactic description. A value `v` conforms to a type `T` (written `P, h ⊢ v :≤ T`) iff `v`’s dynamic type is a subtype of `T`. This conformance notion naturally extends to list of values (written `P, h ⊢ vs :≤ Ts`), stores and environments (written `P, h ⊢ xs :≤ E`), objects, arrays, heaps (written `P ⊢ h√`), and states (written `P, E ⊢ s√`), see Appendix B.3 for the formal definitions. Conformance satisfies two essential properties: First, values read from a conformant state always conform to their declared type. Second, state updates preserve state conformance if the new values conform to the locations’ types.

The typing rules from §2.1.2 are too strong to be invariant under reductions. For example, they rule out literal addresses in expressions, which arise naturally during reduction. To make them well-typed, the run-time type system [46] `P, E, h ⊢ e : T` takes the heap into account and relaxes various preconditions that are not invariant. For example, the subtyping condition in `WTAAss` may be violated during evaluation because `e_1`’s element type may become more specific thanks to covariant
subtyping on arrays. The semantics throws an `ArrayStore` exception in that case. The details have been discussed at length elsewhere [46, 83], so I do not repeat them here.

Now, progress holds under suitable conditions:

**Theorem 2.1** (Progress). If \( \text{wf-J-prog} \ P \text{ and } P, E, h \vdash e : T \) and \( D e \lceil \text{dom } xs \rceil \) and \( P \vdash h \checkmark \) and \( \neg \text{final } e \), then \( P \vdash \langle e, (h, xs) \rangle \rightarrow \langle e', s' \rangle \) for some \( e' \) and \( s' \).

**Proof.** The proof proceeds by induction on \( P, E, h \vdash e : T \). Well-formedness ensures that method calls have the correct number of parameters; definite assignment checks that local variables are initialised when read; and heap conformance prevents field access from getting stuck if the accessed field has no value. \( \square \)

The subject reduction theorem expresses that reductions preserve well-typedness. It is shown by induction on \( P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \).

**Theorem 2.2** (Subject reduction). If \( \text{wf-J-prog} \ P \text{ and } P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \) and \( P, E, h p s \vdash e : T \) and \( P, E \vdash s \checkmark \), then there exists a \( T' \) such that \( P, E, h p s' \vdash e' : T' \) and \( P \vdash T' \leq T \).

Preservation also requires the following preservation lemmata for all other invariants, i.e. definite assignment and conformance. Again, their proofs are by induction on the reduction.

**Lemma 2.1** (Preservation of definite assignment). If \( \text{wf-J-prog} \ P \text{ and } P \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \) and \( D e \lceil \text{dom } (lcl s) \rceil \), then \( D e' \lceil \text{dom } (lcl s') \rceil \).

**Lemma 2.2** (Preservation of conformance). Let \( \text{wf-J-prog} \ P \text{ and } P \vdash \langle e, (h, xs) \rangle \rightarrow \langle e', (h', xs') \rangle \) and \( P, E, h \vdash e : T \). If \( P \vdash h \checkmark \), then \( P \vdash h' \checkmark \). If \( P, h \vdash xs (\leq) E \), then \( P, h' \vdash xs' (\leq) E \).

### 2.2 The JinjaThreads virtual machine

This section describes the JinjaThreads virtual machine language (§2.2.1), its operational semantics (§2.2.2), well-formedness (§2.2.3) and the type safety proof (§2.2.4). The bytecode language and the JinjaThreads virtual machine (VM) model Java bytecode and the Java VM according to the Java Virtual Machine Specification (JVMS) [103].
type_synonym jvm-method = nat × nat × instr list × ex-table

type_synonym jvm-prog = jvm-prog prog

type_synonym ex-table = ex-entry list

type_synonym ex-entry = pc × pc × cname option × pc × nat

Figure 2.15: Method bodies in bytecode

2.2.1 The bytecode language

The bytecode language JVM reuses many concepts from source code. For program declarations, I only have to specify the type of method bodies to be plugged in for \( \text{\textasciitilde}m \). Everything else remains unchanged, in particular, the lookup functions, subtyping and generic well-formedness.

A method body \((msl, mxl, ins, xt)\) in the bytecode language (of type \textit{jvm-method}, see Figure 2.15) consists of an instruction list \(\text{ins}\), an exception table \(\text{xt}\), the maximum stack length \(\text{msl}\), and the number \(\text{mxl}\) of required registers, not counting the \textit{this} pointer and parameters. Hence, a JVM program instantiates the type variable \(\text{\textasciitilde}m\) with \textit{jvm-method}. The lookup functions \textit{instrs-of} \(P\ C\ M\) and \textit{ex-table-of} \(P\ C\ M\) extract the instruction list and exception table for method \(M\) in class \(C\) from \(P\).

The exception table \(\text{xt}\) is a list of exception table entries \((f, t, Co, pc, d)\) where \(Co\) is either some class name \(\lfloor C \rfloor\) or the special constant \(\text{\specialconstant{Any}} = \text{\textit{None}}\). The exception handler starting at the index \(pc\) in \(\text{ins}\) expects \(d\) elements on the stack. It handles exceptions that are raised by instructions in the interval from \(f\) inclusive to \(t\) exclusive. If \(Co\) is a class name \(\lfloor C \rfloor\), it handles only those that are a subclass of \(C\); if \(Co\) is \(\text{\specialconstant{Any}}\), it handles all.

\(\text{\specialconstant{Any}}\) might seem redundant because all exceptions must be subclasses of \textit{Throwable}, i.e., \(\lfloor \textit{Throwable} \rfloor\) could replace \(\text{\specialconstant{Any}}\). However, I include \(\text{\specialconstant{Any}}\) for two reasons: First, the Java Virtual Machine specification (JVMS) [103, §4.7.3] also specifies such a “catch-all” value, which is meant for compiling \textit{finally} blocks. Second, \(\lfloor \textit{Throwable} \rfloor\) and \(\text{\specialconstant{Any}}\) are interchangeable only if one can prove that all raised exceptions are subclasses of \textit{Throwable}, which requires a type safety proof. Thanks to \(\text{\specialconstant{Any}}\), the compiler verification (Chapter 5) can avoid the subject reduction and preservation proofs for the intermediate language by not relying on such invariants.

Table 2.2 shows the instruction set of the sequential JinjaThreads VM (HOL type \textit{instr}). If not provided explicitly, operands are taken
### 2.2. The JinjaThreads virtual machine

<table>
<thead>
<tr>
<th>instruction</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load $i$</td>
<td>load from register $i$</td>
</tr>
<tr>
<td>Store $i$</td>
<td>store into register $i$</td>
</tr>
<tr>
<td>Push $v$</td>
<td>push literal value $v$ on stack</td>
</tr>
<tr>
<td>Pop</td>
<td>pop value from stack</td>
</tr>
<tr>
<td>Dup</td>
<td>* duplicate top value on stack</td>
</tr>
<tr>
<td>Swap</td>
<td>* swap top elements on stack</td>
</tr>
<tr>
<td>BinOp $bop$</td>
<td>‡ apply binary operator $bop$</td>
</tr>
<tr>
<td>New $C$</td>
<td>create object of class $C$</td>
</tr>
<tr>
<td>NewArray $T$</td>
<td>* create array with element type $T$</td>
</tr>
<tr>
<td>ALoad</td>
<td>* fetch array cell</td>
</tr>
<tr>
<td>AStore</td>
<td>* set array cell</td>
</tr>
<tr>
<td>ALength</td>
<td>* get length of array</td>
</tr>
<tr>
<td>Getfield $F$ $C$</td>
<td>fetch field $F$ declared in class $C$</td>
</tr>
<tr>
<td>Putfield $F$ $C$</td>
<td>set field $F$ declared in class $C$</td>
</tr>
<tr>
<td>Checkcast $T$</td>
<td>‡ ensure that value conforms to type $T$</td>
</tr>
<tr>
<td>Instanceof $T$</td>
<td>* check if object is of type $T$</td>
</tr>
<tr>
<td>Invoke $M$ $n$</td>
<td>invoke method $M$ with $n$ parameters</td>
</tr>
<tr>
<td>Return</td>
<td>return from method</td>
</tr>
<tr>
<td>Goto $i$</td>
<td>relative jump</td>
</tr>
<tr>
<td>IfFalse $i$</td>
<td>branch if top of stack is Bool False</td>
</tr>
<tr>
<td>ThrowExc</td>
<td>throw top of stack as exception</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>instruction</th>
<th>description</th>
</tr>
</thead>
</table>
|                | from the stack and results pushed onto the stack. §3.2.3 will add the instructions for multithreading. Similar to Table 2.1, ‡ marks instructions which have been generalised with respect to Jinja. Now, casts between arbitrary types are possible, and the two instructions for binary operators in Jinja have been combined into one that covers all 17 operators from JinjaThreads. The instructions that Jinja does not feature are marked with *. Klein [79] has already modelled arrays and the instructions Dup and Swap for stack manipulation for µJava. Instanceof is a variation of Checkcast that returns a Boolean rather than throwing an exception if it fails.

In comparison to Java bytecode instructions, JinjaThreads unifies instructions that only differ on their operand types (e.g., aLoad and

---

Table 2.2: Instructions of the sequential virtual machine
Chapter 2. Sequential JinjaThreads

\begin{verbatim}
type_synonym jvm-state = addr option × heap × frame list

type_synonym frame = opstack × registers × cname × mname × pc

type_synonym opstack = val list

type_synonym registers = val list

type_synonym pc = nat

Figure 2.16: Type definitions for the sequential VM state space
\end{verbatim}

\texttt{i1oad}) in polymorphic ones (e.g., \texttt{Load}), but the instructions have not been simplified conceptually. Moreover, a few instructions for stack and register manipulation (e.g., \texttt{dup2}, \texttt{iinc}) have been omitted, but they can be simulated by existing ones or could be added easily. Neither does JinjaThreads include any instructions for omitted types such as \texttt{byte} and \texttt{float} nor advanced control flow instructions like \texttt{tableswitch} and \texttt{jsr} for subroutines.

2.2.2 Semantics

As with source code, I only present the VM up to executing single instructions and defer complete executions to §3.2.3. The model of the sequential JinjaThreads virtual machine (VM) covers its state space and the semantics of the bytecode language.

The state space

The state space is taken from the Jinja VM [83], Figure 2.16 shows the relevant type definitions: The state \((xcp, h, frs)\) of type \texttt{jvm-state} consists of an exception flag \texttt{xcp} ([\(a\]) corresponds to \texttt{Throw a} in \texttt{J} and \texttt{None} denotes none), a heap \texttt{h} (the same as in \texttt{J}) and a stack of call frames.

Each method executes in its own call frame (type \texttt{frame}). A call frame \((stk, loc, C, M, pc)\) contains the operand stack \texttt{stk}, an array \texttt{loc} of registers for the \texttt{this} pointer, the parameters, and local variables, the class name \texttt{C}, the method name \texttt{M}, and the program counter \texttt{pc}. Although \texttt{registers} are modelled as lists, their length does not change during execution. In contrast, the size of the operand stack does change, but the maximum size is statically known.

A state is final iff the stack of call frames is empty.
2.2. The JinjaThreads virtual machine

exec-instr (Invoke $M'$ $n$) $P$ $h$ stk loc $C$ $M$ pc frs =
(let ps = rev (take $n$ stk); $r = stk[n]$; Addr $a = r$; [$hT] = typeof-addr $h$ $a$
  in if $r = \text{Null}$ then
    \{ ([addr-of-sys-xcpt $\text{NullPointer}$]$h$, (stk, loc, $C$, $M$, pc) $\cdot$ frs) \}
  else let $(D, Ts, T_r, m) = \text{method} P$ (class-of' $hT$) $M'$
    in case $m$ of Native $\Rightarrow$
      \{ native-Ret2jvm $n$ $h'$ stk loc $C$ $M$ pc frs vx $\mid$ vx $h'$.
      $(vx, h') \in \text{exec-native} P$ a $M'$ ps $h$ \}
    | \{ $(msl, mxl, ins, xt)] \Rightarrow$
      let fr' = (\[], $r \cdot$ ps @ replicate mxl undefined-Val, $D, M'$, 0)
    in \{ (None, $h$, fr' $\cdot$ (stk, loc, $C$, $M$, pc) $\cdot$ frs) \})

Figure 2.17: Single-step semantics of the Invoke instruction

The aggressive VM

I now turn to the semantics of JinjaThreads bytecode. Following the Jinja VM, the JinjaThreads VM is defined in a functional style.

The function exec-instr :: instr $\Rightarrow$ jvm-prog $\Rightarrow$ heap $\Rightarrow$ opstack $\Rightarrow$
registers $\Rightarrow$ cname $\Rightarrow$ mname $\Rightarrow$ pc $\Rightarrow$ frame list $\Rightarrow$ jvm-state set defines
the semantics of a single instruction. Given the instruction, the program,
the heap, and the curried non-empty call stack, it produces a non-
empty set of successor states by pattern-matching on the instruction.
The complete definition for exec-instr can be found in Appendix B.7.3,
which includes the adaptations for concurrency. Although exec-instr
is deterministic for sequential JinjaThreads, it returns a set of successor
states, because this will be convenient later for multithreading (§3.2.3)
and the Java memory model (Chapter 4).

As an example, Figure 2.17 shows the definition for Invoke. If the
receiver $r$ is the null pointer, it throws a $\text{NullPointer}$ exception; the old
frame and call stack is reassembled such that exception handling can find
the right handler. If not, let $a$ denote the receiver address, $hT$ its dynamic
type information and $m$ the method body that method lookup returns.
If the call is native, i.e., $m = \text{Native}$, exec-instr delegates the call to the
functional reimplementation exec-native of $P \vdash \langle \_ \_ \langle \_ \_ \_ \rangle, \_ \_ \rangle \Rightarrow_{nc} \langle \_ \_ \_ \rangle$.
Similarly to native-Ret2J, the function native-Ret2jvm assembles the
new state from the result. Otherwise, exec-instr prepares the new call
frame $fr'$ according to the method body $(msl, mxl, ins, xt)$: Initially, the
Chapter 2. Sequential JinjaThreads

\[
\begin{align*}
\text{exec } P (xcp, h, []) &= \emptyset \\
\text{exec } P (\text{None}, h, (stk, loc, C, M, pc) \cdot frs) &= \\
&\quad \text{exec-instr (instrs-of } P C M [pc] P h stk loc C M pc frs \\
\text{exec } P ([a], h, fr \cdot frs) &= \{ \text{xcpt-step } P a h fr frs \}
\end{align*}
\]

**Figure 2.18:** Combining normal execution and exception handling in the VM

operand stack is empty; the registers contain the receiver (i.e., `this` pointer) and the parameters `ps` (which are in reverse order on the stack); `replicate mxl undefined-Val` fills the remaining registers with the dummy value `undefined-Val`.

Being functional rather than relational, `exec-instr` uses functional lookup operations like `method` whose specifications rely on Russell’s \(\iota\)-operator. For example,

```latex
method P C M = \iota(D, Ts, Tr, \text{meth}) \quad P \vdash C \text{ sees } M:Ts \rightarrow Tr = \text{meth in } D
```

If there is no such method, `method P C M` is unspecified.

This leads to an aggressive VM: As can be seen in Figure 2.17, `exec-instr` assumes that there are always sufficiently many operands of the right types on the stack, all methods and fields exist, etc. If not, the result is unspecified. The type safety proof (§2.2.4) shows that these cases cannot occur for well-formed programs (to be defined in §2.2.3).

The function `exec :: jvm-prog \Rightarrow jvm-state \Rightarrow jvm-state set incorporates exception handling in the semantics (see Figure 2.18). The VM halts if the call stack is empty. If no exception is flagged, `exec` executes the next instruction via `exec-instr`. Otherwise, `xcpt-step` (definition in Appendix B.7.3) tries to find an exception handler in the top-most call frame `fr` that matches the flagged exception at address `a`. If one is found, the operand stack is trimmed to the size specified in the exception table entry, `a` is pushed on the operand stack, and the program counter is set to the start of the handler. Otherwise, it pops `fr` and rethrows `a` at the `Invoke` instruction of the previous call frame.

For convenience, there also is a relational view on `exec`:

\[
s' \in \text{exec } P s \\
P \vdash s \rightarrow_{\text{jvm}} s'
\]
2.2. The JinjaThreads virtual machine

check-instr \((\text{Invoke } M' \ n) \ P \ h \ stk \ loc \ C \ M \ pc \ f \ rs = \)
\((n < |stk| \land \text{is-Ref } stk_{[n]} \land \)
\((stk_{[n]} \neq \text{Null} \implies \)
\((\text{let } ps = \text{rev (take } n \ stk); \ Addr \ a = stk_{[n]}; \ [hT] = \text{typeof-addr } h \ a; \)
\(C' = \text{class-of' } hT; (D, Ts, T_r, \text{meth}) = \text{method } P \ C' \ M'
\text{in } \text{typeof-addr } h \ a \neq \text{None } \land P \vdash C' \text{ has } M' \land \)
\(P, h \vdash ps [\leq] Ts \land (\text{meth } = \text{Native } \implies D.M'(Ts) :: T_r))))\)

Figure 2.19: Defensive checks for the \text{Invoke} instruction

The defensive VM

JinjaThreads also features a defensive VM that introduces additional type and sanity checks at run time. If they are violated, the defensive VM raises a type error. The function execd adds these checks on top of the aggressive VM exec.

datatype 'a type-error = TypeError | Normal 'a

execd P s = if check P s then Normal (exec P s) else TypeError

The function check checks that the class and method in the top call frame exist and that the program counter and stack size are valid. Moreover, if an exception is flagged, it must be the address of an object on the heap and, if an exception handler in the current method matches, the stack must have at least as many elements as the handler expects. Otherwise, if no exception is flagged, check calls check-instr (with identical parameters as exec-instr) to check instruction-specific conditions. For example, Figure 2.19 shows the run-time checks for the Invoke instruction. It clearly parallels the definition of exec-instr in Figure 2.17. \text{is-Ref } v predicates that \(v\) is \(\text{Null}\) or some \(\text{Addr } a\), and \(P \vdash C \text{ has } M\) abbreviates \(\exists Ts T_r \text{meth } D. P \vdash C \text{ sees } M:Ts \to T_r = \text{meth in } D\). When a native method is called, i.e., \(m = \text{Native}\), check-instr checks that the semantics does implement the native method.

Again, there is also a relational view on the defensive VM:

\[
\begin{align*}
\neg \text{check } P s & \quad \text{check } P s \quad s' \in \text{exec } P s \\
\quad \quad P \vdash \text{Normal } s \to_{jvmd} \text{TypeError} & \quad \quad P \vdash \text{Normal } s \to_{jvmd} \text{Normal } s' \quad (2.5)
\end{align*}
\]


2.2.3 Well-typings

When executing bytecode, the JinjaThreads VM relies on the following assumptions: There are as many operands as needed and of the right types; registers are initialised before being read; the operand stack stays within the declared limit; the declared register number is correct; and the program counter always points to a valid instruction.

To prevent violations of these assumptions during execution, the bytecode must satisfy certain type constraints, similar to the typing rules for source code. Other than source code, bytecode does not declare the types of the registers and the stack elements. Hence, JinjaThreads models type information separately. This yields an abstract interpretation of the bytecode semantics with values being abstracted to types, similar to bytecode verification (§6.2.2).

Well-typings in JinjaThreads do not di
differ from Jinja in any essential way. Here, I therefore only sketch the main ideas and introduce the notation relevant for this thesis. For details, see [83].

Typings

A state type \( \tau \) characterises a set of run-time states for one instruction by giving type information for the operand stack and registers. \( \tau = \text{None} \) denotes that control flow cannot reach the instruction. Otherwise, say \( \tau = [(ST,LT)] \), \( ST :: ty \ list \) gives the types for the elements on the operand stack and \( LT \) the types for the register contents. The elements of \( LT \) are either \( \text{Err} \) or \( \text{OK} \ T \) for some type \( T :: ty \). \( \text{Err} \) denotes that a register is unusable and its type is unknown, e.g., if it has not been initialised yet. For example, \( [(\text{}), \text{[OK (Class C), Err]}] \) denotes that the stack is empty and there are two registers, register 0 holds a reference to an object of a subclass of \( C \) (or the \text{Null} pointer), the second is unusable.

A method type \( \tau s \) is a list of state types, one for each instruction. A program typing for \( P \) is a function \( \Phi \) such that \( \Phi C M \) is the method type for every method \( M \) in every class \( C \) of \( P \).

A state type \( \tau \) for an instruction \( i \) is a state well-typing iff \( i \) can execute safely in any state that \( \tau \) characterises and \( \tau \) is consistent with the successor instruction’s state type (to be explained below). A method type \( \tau s \) is a method well-typing iff each of its state types is a state well-typing for its instruction.
Abstract interpretation of the semantics

Applicability \( \text{app} \) takes the checks of the defensive VM to the type level – in a state well-typing, it defines when the instruction can execute safely. It is split into \( \text{app}_i \) for normal and \( \text{xcept-app} \) for exceptional execution. Similar to \( \text{check-instr} \), the function \( \text{app}_i \) checks the instruction-specific preconditions. For example, Figure 2.20 shows the definition for the \text{Invoke} instruction. It parallels \( \text{check-instr} \) from Figure 2.19 – except for the check that the semantics implements native methods because well-formedness ensures this already.

For exceptions, \( \text{is-relevant-class} \) \( i \ P \ C \) predicates that handlers for class \( C \) might match some exception that instruction \( i \) raises. Using this specification, \( \text{xcept-app} \) statically approximates all handlers that might match at run time. Among others, it checks that they do not expect more stack elements than are currently on the stack.

Effect (function \( \text{eff} \)) simulates \text{exec} on the type level. Since \( \text{app} \) has already checked the instructions preconditions, \( \text{eff} \)’s specification is aggressive. It returns a list of pairs, each of which consists of the successor program counter and the changed state type. Like \( \text{app} \), \( \text{eff} \) decomposes into \( \text{eff}_i \) for normal executions and \( \text{xcept-eff} \) for jumping to exception handlers. Note how \( \text{eff}_i \) closely follows \text{exec-instr}, e.g., by comparing Figure 2.20 to Figure 2.17. Here, \( \text{eff}_i \) does not treat the case when the receiver is \text{Null}, because \text{exec-instr} always raises an exception in that case. Note how \( \text{eff}_i \) uses the return types in method signatures as summary information for the call. In contrast to the semantics, it deals with normal and native methods uniformly, because looking up the method to retrieve the signature works uniformly.
Like `xcpt-app`, `xcpt-eff` determines via `is-relevant-class` all potential exception handlers and adds them as successors with the stack typing trimmed to the specified height.

Consistency formalises that the effects are consistent with the state types of the successor instructions. To that end, Jinja first lifts the subtype relation to state types, written \( P \vdash \tau \leq \tau' \). OK elements are ordered according to the subtype relation, \( \text{Err} \) becomes the new top element. For register types \( LT \), this relation is extended pointwise to lists. For the stack typing \( ST \), the relation is \( P \vdash _\_ [\leq] _\_ \). The least state type is \( \text{None} \); on \( (ST, LT) \), the order is componentwise. Now, consistency requires that the state type \( \tau \) of the successor \( i' \) is greater than or equal to the state type that \( \text{eff} \) computes.

**An example well-typing**

For example, Figure 2.21 shows a method in class \( C \) that takes a single Integer parameter and returns an \( Object \). The left column specifies the instruction list and all of the right-hand side is a well-typing. A state type is one single row of the table, it describes the situation before executing the instruction in the same row. Initially, the stack is empty and the registers contain the \( \text{this} \) pointer and the parameter. It loads the \( \text{this} \) pointer and calls the native method \( \text{clone} \) on it, which clones the \( \text{this} \) pointer. Note how the receiver type \( \text{Class C} \) on the stack is replaced with the declared return type \( \text{Class Object} \) of \( \text{clone} \). Then, it stores the returned reference to the clone in register 1, reusing the former parameter register. Finally, it loads \( \text{this} \) again and returns it. The \( \text{Return} \) instruction is also the handler for the \( \text{OutOfMemory} \) exception that the call to \( \text{clone} \) may raise. In that case, the method returns the caught exception.

Control flow merges at \( \text{Return} \): Normal execution linearly runs from the first to the last instruction. In the exceptional case, \( \text{Return} \) executes directly after \( \text{Invoke} \). Hence, the normal effect of the second \( \text{Load} \ 0 \) and the exceptional effect of \( \text{Invoke} \) for the specified handler must both be \( \leq' \)-smaller than or equal to the state type for \( \text{Return} \). They are

\[
[ \text{Class C}, [\text{OK (Class C)}, \text{OK (Class Object)}] ] \\
\text{and} \quad [ [\text{Class OutOfMemory}], [\text{OK (Class C)}, \text{OK Integer}] ]
\]

respectively. For the stack, \( \text{Object} \) is a superclass of \( C \) and \( \text{OutOfMemory} \), so this is fine. In the register 1, \( \text{Integer} \) and \( \text{Class Object} \) have no common supertype, so it must be marked unusable, i.e., \( \text{Err} \).
2.2. The JinjaThreads virtual machine

<table>
<thead>
<tr>
<th>instruction list</th>
<th>stack</th>
<th>registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load 0</td>
<td>[(</td>
<td>[], [OK (Class C), OK Integer] )]</td>
</tr>
<tr>
<td>Invoke clone 0</td>
<td>[(</td>
<td>Class C, OK (Class C), OK Integer) ]</td>
</tr>
<tr>
<td>Store 1</td>
<td>([ [Class Object, OK (Class C), OK Integer] ] )</td>
<td></td>
</tr>
<tr>
<td>Load 0</td>
<td>[</td>
<td>[], [OK (Class C), OK (Class Object)]) ]</td>
</tr>
<tr>
<td>Return</td>
<td>[</td>
<td>([Class Object], [OK (Class C), Err]] )</td>
</tr>
</tbody>
</table>

exception table: 

[1, 2, [OutOfMemory], 4, 0]

maximum stack length: 1

local registers: 0

Figure 2.21: Example of a method well-typing

Well-formedness

Method well-typings are only method-local. Rather than checking applicability and effect across method boundaries do they take the method signatures as summaries for method calls. Among others, well-formedness ensures that these summaries are indeed correct.

A method declaration \((M, Ts, T, [(msl, mxl, ins, xt)])\) in class \(D\) is well-typed with respect to \(Ts\) iff

(i) \(Ts\) is a well-typing for the method,

(ii) all state types in \(Ts\) contain only valid types, and respect the maximum stack length \(msl\) and the fixed number \(mxl\) of registers, and

(iii) \(Ts\) satisfies the start condition, i.e., it is non-empty and the first state type \(Ts[0]\) is \(\leq\)'-greater than the abstraction of the initial state of the call frame. Remember from Figure 2.17 that in the initial state, the operand stack is empty and the registers contain the this pointer and parameters, the local variables are undefined. Its abstraction is \([([], OK (Class D) \cdot map OK Ts @ replicate mxl Err)]\).

Note that without the start condition, the typing \(replicate |ins| None\) would be a well-typing for any method declaration with non-empty instruction sequence, but then, the method may never be called, because the first instruction is marked unreachable (None).
Chapter 2. Sequential JinjaThreads

A program typing $\Phi$ is a (program) well-typing for $P$ iff every method $M$ in every class $C$ of $P$ is well-typed with respect to $\Phi C M$. A JVM program $P$ is well-formed (written $wf$-jvm-prog $P$) iff it satisfies the generic well-formedness constraints (Figure 2.9) and there is a program well-typing $\Phi$ for it.

This definition of well-formedness is not constructive, because it does not specify how to obtain the well-typing from the bytecode program. To that end, JinjaThreads contains a bytecode verifier (§6.2.2). It rephrases the abstract interpretation as a data flow analysis problem and computes a well-typing with Kildall’s algorithm [78].

2.2.4 Type safety

Type safety for bytecode means that the assumptions of the VM are always met at run time. This is expressed most clearly as: The defensive VM never raises a type error when it executes a well-formed program $P$. Let $\Phi$ be the well-typing for $P$. Naturally, the proof requires a state invariant called conformance, written $P, \Phi \vdash s \sqrt{\cdot}$.

Conformance requires that the state type correctly abstracts the run-time state $(xcp, h, f rs)$, i.e.,

(i) if $xcp = [a]$ flags the exception at address $a$, $a$ designates an object on the heap $h$ whose class $C$ is a subclass of $Throwable$ and conforms to the exception specification in $xcpt$-app for the current instruction in the top-most call frame,

(ii) the heap conforms, i.e., $P \vdash h \sqrt{\cdot}$,

(iii) for all call frames $(stk, loc, C, M, pc)$ in $f rs$, $C$ declares $M$, $pc$ points to a reachable instruction and the contents of the operand stack $stk$ and registers $loc$ conform to the type that the well-typing $(\Phi C M)[pc]$ specifies, and

(iv) all call frames except for the top-most one are halted at an Invoke instruction that calls a method whose static summary information, i.e. parameter types and return type, is compatible with the call frame above.

For a well-formed program and correct state, one can now show type safety:
2.3. Comparison with Jinja, Bali, and µJava

**Theorem 2.3** (Type safety). Let $\Phi$ be a well-typing for $P$ and $P, \Phi \vdash s \sqrt{.}$ Then, the following statements hold:

(a) The defensive VM does not raise a type error:

\[ \neg P \vdash Normal s \rightarrow_{jvmd} TypeError \]

(b) Aggressive and defensive VM agree:

\[ P \vdash s \rightarrow_{jvm} s' \text{ iff } P \vdash Normal s \rightarrow_{jvmd} Normal s'. \]

(c) Conformance is preserved: If $P \vdash s \rightarrow_{jvm} s'$, then $P, \Phi \vdash s' \sqrt{.}$.

The proof for (a) and (c) proceeds by case distinction on the current instruction. (b) directly follows from (a) by construction of the defensive VM. (c) shows that $app$ and $eff$ are indeed abstract interpretations of the semantics.

### 2.3 Comparison with Jinja, Bali, and µJava

In this section, I compare the sequential part of JinjaThreads with its predecessors Jinja [83], Bali [11, 84, 135], and µJava [79, 85]. A goal in extending Jinja with concurrency was to reuse as much as possible from Jinja. Hence, I built on many of the fundamental concepts, although textually, almost every definition has undergone some adaptations. Here, I only present the conceptual changes.

Every valid Jinja program can be trivially converted into a valid JinjaThreads program; one merely needs to adapt it to the generalisations of expressions, instructions, and declarations of native methods and additional system exceptions. Beyond that, sequential JinjaThreads extends Jinja mainly in three respects: binary operators, arrays, and native methods.

#### Binary operators

Jinja and µJava support only the binary operators for equality $==$ and integer addition $+$. In [84], Bali covers the full range of binary operators in Java – except for string concatenation, but syntactically distinguishes the overloaded operators $\&$, $|$, and $\hat{\vee}$. Binary operators always return a value,
e.g., division by 0 yields 0 instead of throwing an `ArithmeticException`. Moreover, Bali declares, but does not correctly implement the operators `&`, `|`, and `>>` that manipulate integers bit-wise.

JinjaThreads covers the same set of operators as Bali. While JinjaThreads’ predecessors use unbounded integers (type `int`), I changed them in JinjaThreads to 32-bit machine integers (type `word32`) from Isabelle’s word library. Hence, I can draw on the functions from the Isabelle word library for correctly implementing the bit-wise operators – except for `/` and `%`. The latter are implemented manually, because the word library only defines division and remainder for unsigned words. Thus, the arithmetic operations follow the Java behaviour even in case of overflows. However, this comes at the expense of introducing various coercions between `word32`, `nat`, and `int` for array accesses. In bytecode, `BinOp` takes the operator as a parameter, thereby unifying the various Java bytecode instructions, which are available as abbreviations.

Like Bali, JinjaThreads defines typing rules for binary operators (Figure 2.8) and the semantics (Figure 2.13) to encapsulate binary operators from the rest of the language. Unlike Bali, JinjaThreads generalises these such that a binary operator may fail with an exception, e.g., division by 0.

Arrays

Both Bali and Klein’s extension of `μJava` bytecode [79] model arrays, but they have been omitted in Jinja. JinjaThreads reintroduces arrays. Everything is standard except for the set `types P` of valid types. Bali allows arrays with `NT` as element type; infinite ascending chains do not matter as it does not model an executable bytecode verifier. Klein limits the maximum number of array dimensions such that `types P` is finite. The class file format for Java bytecode limits array dimensions to 255 [103, §4.8.1]. There is no such bound for Java source code, but the maximum number of array dimensions can be computed statically for any fixed program. Instead of formalising and verifying such a computation, I leave the number of array dimensions unbounded in JinjaThreads.

Arrays entail the following adaptations to Jinja: First, additional conversions with `class-of` and `class-of’` permit to uniformly reuse the lookup functions, which are defined only for class names, e.g., in method calls (Figure 2.7, `WTcall`; Figure 2.14, `Rcall`; Figures 2.17 and 2.19). This way, `Object` may declare fields and methods freely, which all array
types inherit [56, §6.4.5]. Bali and $\mu$Java disallow fields in class Object, because their heap model cannot handle fields in arrays [79, §6.2]. The JLS neither specifies any field in Object, nor does it forbid such. The implementation of Object in the Java standard library does not declare any fields, but user-defined implementations of Object may do so.

Second, JinjaThreads generalises casts to arbitrary types, not only classes as in Jinja and $\mu$Java. At the same time, I also introduce an instanceof operator which allows to test on reference types without resorting to exceptions.

Third, as all thrown exceptions must be objects (and not arrays), the type safety proof requires that Throwable replace Object as the root of the exception hierarchy (like in Bali). For example, WTTHROW and WTTry (Figure 2.7) strengthen the check is-class $P C$ with $P \vdash C \leq^* \text{Throwable}$. In contrast, Klein’s extension of $\mu$Java permits to throw arrays as exceptions.

Native methods

Native methods are new in JinjaThreads, none of its predecessors has included them. Although the semantics only implements two native methods, namely hashcode and clone in sequential JinjaThreads, native methods will be crucial for multithreading.

Unlike Java, calls to clone in Object omit the test whether the receiver object implements the Cloneable interface, because JinjaThreads does not model interfaces. Instead, clone always returns a copy of the receiver if there is enough memory left. Also, JinjaThreads does not specialize the return type of clone for arrays (see Footnote 7 on page 33).

Other changes

Apart from the above extensions, sequential JinjaThreads also changes the following aspects of Jinja: For source code, JinjaThreads only defines a small-step semantics, but no big-step semantics, because a big-step semantics cannot adequately express the interleaving of threads. Jinja defined both a big-step and small-step semantics and proved them equivalent. Bali and $\mu$Java source code define only a big-step semantics.

Moreover, following a change in the JLS from the second [55] to the third edition [56], the typing rule for the conditional operator $?:$ (in JinjaThreads written as if $(e) e_1 \text{ else } e_2$) has changed: According to the second edition [55, §15.25], which Jinja follows, the subtype relation
must relate \( e_1 \)'s type \( T_1 \) with \( e_2 \)'s type \( T_2 \), i.e., \( P \vdash T_1 \leq T_2 \) or \( P \vdash T_2 \leq T_1 \); then, the conditional's type is the larger one of \( T_1 \) and \( T_2 \). In the new edition [56, §15.25], \( T_1 \) and \( T_2 \) only need to have a lub, which is also the type of the conditional, cf. WT\text{cond} (Figure 2.7). Although this change may seem negligible, it affects code extraction (§6.2.3) non-trivially.

Further, a local variable block \( \{ V : T = \lfloor v \rfloor ; e \} \) in source code contains an optional initialisation value \( v \), in which execution also keeps track of \( V \)'s value. Jinja omits the \( = \lfloor v \rfloor \), but emulates it as \( \{ V : T ; V := \text{Val } v ; ; e \} \), i.e., an uninitialised block whose body starts with assignment to the variable. Adding the explicit initialisation simplifies the semantics and proofs, e.g., progress (Theorem 2.1) no longer requires a custom induction rule.

In bytecode, exception handling has changed a little: Following the JVMS [103, §4.7.3], exception tables may now contain the special value \text{Any} to handle all exceptions that are raised at the guarded instructions (cf. §2.2.1). This increase in expressiveness simplifies the compiler verification (Chapter 5).

When an instruction raises an exception, the Jinja VM immediately transfers control to the right exception handler, which may require to scan and pop an arbitrary large part of the call stack. In contrast, the JinjaThreads VM allows raised, but yet unhandled exceptions as intermediate states, and searches the exception handlers of one call frame at a time (cf. Figure 2.18). This simplifies some proofs, in particular the compiler verification (cf. Chapter 5), but, in turn, conformance must be strengthened to account for the new states (condition (i) in the conformance definition, §2.2.4, is new).

More importantly, with this approach, the defensive VM can easily check that the operand stack contains sufficiently many values, before it trims it to the length that the exception handler expects. In Jinja, this check is missing.
Multithreading is just one damn thing after, before, or simultaneous with another.

Andrei Alexandrescu

3

Interleaving semantics

In this chapter, I describe how to add Java-style multithreading to the semantics from Chapter 2 and prove type safety. A precursor to this work has been published in [107].

Multithreading in JinjaThreads comprises dynamic thread creation, synchronisation via monitors, the wait-notify mechanism, and thread interruption. To separate the semantics of these mechanisms and their implications for type safety from the low-level details of allowed compiler and run-time optimisations, I defer the Java Memory Model [56, §17.4] to Chapter 4. Instead, I use interleaving semantics with a single shared memory through which threads interact. JinjaThreads omits all timing-related features such as Thread.sleep(long) and Object.wait(long) because JinjaThreads does not model time. Neither does it include the deprecated methods stop, suspend, and resume in Thread, nor the library extensions in java.util.concurrent which are not an integral part of Java.

The challenges in this chapter are the following:

language-independent interleaving Since both source code and bytecode must implement the above multithreading features, they best share as much as possible of the formalisation. To that end, I first develop a generic framework for interleaving semantics (§3.1), which is parametrised by the semantics of individual threads.

reuse of the sequential semantics and proofs As the framework manages the multithreaded state and implements the multithreading features, I reuse the sequential semantics from Chapter 2 with as little adaptations as possible (§3.2.2 and §3.2.3). Consequently, I am able to reuse the type safety theorems of sequential JinjaThreads when
Chapter 3. Interleaving semantics

<table>
<thead>
<tr>
<th>layer</th>
<th>source code</th>
<th>bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>interleaved small-step</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>statements &amp; expressions</td>
<td>call stacks &amp; exception handling</td>
</tr>
<tr>
<td>2</td>
<td>native methods</td>
<td>single instruction</td>
</tr>
</tbody>
</table>

**Figure 3.1:** Stack of semantics with interleaving

proving type safety for multithreaded JinjaThreads. Here again, I show how to share as much as possible between source code and bytecode. The stack of semantics in Figure 3.1 shows how source code and bytecode share native methods and the interleaving semantics. As it can be seen at the layer numbering, some layers are still missing. I add them in Chapter 4 on memory models.

**characterisation of deadlocks** Progress is non-trivial when threads can deadlock. Therefore, §3.3 formalises deadlock and refines progress such that it allows for deadlock. This again is designed to be independent of the language.

### 3.1 Framework for interleaving semantics

The framework for interleaving semantics manages the interleaving of individual threads. It separates the single-thread semantics from the burdens of multithreading: It manages the multithreaded state (§3.1.1), i.e., the locks, wait sets, the thread pool, and interrupts. At the same time, it isolates the local states of the threads from each other.

Interaction between individual threads and the interleaving semantics only happens through designated actions called thread actions (cf. §3.1.2). They allow the framework to be oblivious of thread-local states such that I can use it for both source and bytecode.

Apart from the interleaving semantics (§3.1.3), the framework also provides the infrastructure to lift predicates and invariants on states of single threads to multithreaded states (§3.1.4).
3.1. Framework for interleaving semantics

3.1.1 The multithreaded state

The multithreaded state consists of five components: the locks status, the thread pool, the shared heap, the wait sets, and the interrupts. In this section, I present each of them first before I assemble the multithreaded state.

Locks

Java-style locks are mutually exclusive and re-entrant, i.e., at most one thread may hold the lock at one time, but it can acquire it multiple times. Hence, a lock (of type ‘t lock) stores which thread holds it, and if so, how many times; ‘t is the type variable for thread IDs. A natural choice for ‘t lock is the HOL type (‘t × nat) option. Then, None denotes that the lock is not held by any thread, and [(t, n)] means that the thread with ID t has acquired the lock n + 1 times. Instead of working directly with this type, I introduce the following operations on ‘t lock; Figure 3.2 shows their implementations.

\[
\begin{align*}
\text{has-locks} &\quad L \ t & \text{number of times } t \text{ has acquired } L \\
\text{may-lock} &\quad L \ t & \text{test whether } t \text{ may lock } L \\
\text{lock-lock} &\quad L \ t & \text{acquire } L \text{ for } t \text{ once} \\
\text{unlock-lock} &\quad L & \text{release } L \text{ once} \\
\text{acquire-lock} &\quad L \ t \ n & \text{acquire } L \text{ for } t \text{ } n \text{ times} \\
\text{release-lock} &\quad L \ t & \text{completely release } L \text{ if } t \text{ holds } L
\end{align*}
\]

In the following, has-lock L t is short-hand for has-locks L t > 0.

Note that some operations on locks should be partial. For example, lock-lock [(t’, n)] t only makes sense if t’ = t. However, since HOL is a logic of total functions, I must define lock-lock for t’ ≠ t, too. For simplicity, lock-lock ignores t and increments t’’s lock counter in that case, but any other implementation would be fine, too.\(^9\) This choice allows to remove some preconditions from certain lemmata. For instance, if has-lock L t, then also has-lock (acquire-lock L t’ n) t. Other implementations would require the precondition t’ = t or, equivalently, may-lock L t’.

The status of all locks is modelled as a function from lock identifiers (type variable ‘l) to locks. Anticipating code generation in Chapter 6, this

\(^9\)I could also have left lock-lock unspecified for that case using Isabelle’s specification command. However, underspecification is detrimental to code generation (see §6.4).
Chapter 3. Interleaving semantics

has-locks = None \( t = 0 \)

\[
\text{has-locks } [(t', n)] t = (\text{if } t = t' \text{ then } n + 1 \text{ else } 0)
\]

may-lock = None \( t = \text{True} \)

\[
\text{may-lock } [(t', n)] t = (t = t')
\]

lock-lock = None \( t = \lfloor (t, 0) \rfloor \)

\[
\text{lock-lock } [(t', n)] t = \lfloor (t', n + 1) \rfloor
\]

unlock-lock = None

\[
\text{unlock-lock } [(t, n)] t = (\text{if } n = 0 \text{ then } \text{None else } [(t, n - 1)])
\]

acquire-lock \( L \) \( t = 0 \)

\[
\text{acquire-lock } L \ t = 0 \text{ (should be } L \text{)}
\]

acquire-lock \( L \) \( t = n + 1 \)

\[
\text{acquire-lock } (\text{lock-lock } L \ t) t n
\]

release-lock = None

\[
\text{release-lock } [(t', n)] t = (\text{if } t' = t \text{ then } \text{None else } [(t', n)])
\]

**Figure 3.2:** Implementation of lock operations

is actually not a function, but a FinFun (see §1.4.1). Thus, the status of all locks formally has type \( l \Rightarrow t \text{ lock} \), written \((l, t) \text{ locks}\). For this chapter, the almost constant nature of FinFun’s is irrelevant, and the reader may think of them as ordinary functions – except for the omnipresent “\( l \)” in the notation.

**Thread pool**

The multithreaded semantics manages the threads and their local states in a thread pool. A thread pool \( ts \) is a map from thread IDs to thread-local states (type variable \( x \)) of the following type.

\[
\text{type_synonym } (l, t, x) \text{ tpool } = l \rightarrow (x \times l \text{ tr-locks})
\]

Free thread IDs are mapped to None. If \( ts \ t = [(x, ln)] \), then \( t \) identifies the thread whose local state is \( x \) (of type \( x \)). For example, for the source code semantics, \( x \) stores the current expression and the store of local variables, i.e., \( x \) gets instantiated with \( \text{expr} \times \text{locals} \). In bytecode, \( x \) consists of the exception flag and the call stack.

The thread pool also stores for every thread a multiset \( ln \) of temporarily released locks, which I also model as a FinFun:

\[
\text{type_synonym } l \text{ tr-locks } = l \Rightarrow l \text{ nat}
\]
For example, when a Java thread suspends itself to a wait set, it temporarily releases the lock on the associated monitor. Upon removal from the wait set, it must reacquire the lock before it can continue (see §3.2.1 for the details of the mechanism). Recall the notation for FinFuns from §1.4.1: $K^f 0$ denotes the empty multiset, and $(K^f 0)(l := l_2)$, e.g., is the multiset which only contains $l$ twice.

The multithreaded semantics keeps track of which locks have been thus released and how many times the thread had acquired them before. Moreover, the thread continues to execute only after it has reacquired its temporarily released locks. Since the multithreaded semantics implements all this (§3.1.3), the single-threaded semantics need neither remember how many times the lock had been acquired, nor reacquire it explicitly afterwards.

**Wait sets**

Java offers wait sets as an alternative to busy waiting. Threads may suspend themselves to the wait set of a monitor where they remain until another thread notifies or interrupts them.

JinjaThreads does not model wait sets explicitly. Rather, it stores for every thread its wait set status:

\[
\text{type_synonym } (\mathbf{\acute{w}}, \mathbf{t}) \text{ wait-sets } = \mathbf{t} \rightarrow \mathbf{\acute{w}} \text{ wait-set-status}
\]

where $\mathbf{\acute{w}}$ wait-set-status consists of the values $\text{InWS} \mathbf{w}$, $\text{WS-Notified}$, and $\text{WS-WokenUp}$, and $\mathbf{\acute{w}}$ represents the type of wait set identifiers. The wait set of a monitor $\mathbf{w} :: \mathbf{\acute{w}}$ contains all threads whose wait status is $[\text{InWS} \mathbf{w}]$. The predicate $\text{waiting} \mathbf{w}$ tests whether the wait set status $\mathbf{w} :: \mathbf{\acute{w}}$ wait-set-status option is of the form $[\text{InWS} \mathbf{w}]$, i.e., the associated thread is in a wait set.

Figure 3.3 shows the different values for a thread’s wait set status and their transitions as an automaton. Initially, after the thread has been spawned, its wait set status is $\text{None}$, i.e., the thread is not in any wait set. Normal execution takes place in that state (dotted arrow). The thread can suspend itself to a wait set with ID $\mathbf{w}$ (status $[\text{InWS} \mathbf{w}]$). When another thread notifies or wakes up the thread (dashed lines), the latter’s status changes to $[\text{WS-Notified}]$ or $[\text{WS-WokenUp}]$, respectively. From either of them, it leaves the wait set cycle and returns to the normal state $\text{None}$ by processing the notification or wake-up, respectively. If the thread has
temporarily released some locks when it suspended itself to the wait set, it must reacquire them in states $\lfloor WS-Notified \rfloor$ and $\lfloor WS-WokenUp \rfloor$ (dotted arrow). Following the JLS [56, §17.8], reacquisition precedes processing the removal from the wait set, i.e., notification or wake-up, although the order is semantically irrelevant.

This only illustrates the purpose of the different wait set status. I define the semantics and the interaction with locks formally in §3.1.3.

**Interrupts**

Java threads interrupt each other by invoking the `interrupt` method in class `Thread`. When running normally, the interrupted thread must actively query its interrupt status to take notice of the interrupt. However, if it is in a wait set at the time of interruption, it is removed from the wait set. When it is joining on a thread, it aborts joining. In both cases, it raises an `InterruptedException`.

The interleaving semantics stores the pending interrupts in a set of thread IDs, i.e., a thread $t$ is interrupted iff $t$ is in this set.

```
type_synonym 't intrs = 't set
```

**The multithreaded state**

A multithreaded state $(ls, (ts, h), ws, is)$ consists of locks $ls$, a thread pool $ts$, the shared memory $h$, the wait set status $ws$, and the interrupts $is$. 
### 3.1. Framework for interleaving semantics

```
thread t1
  synchronized (f) {
    synchronized (g) {
      synchronized (f) {
        g.wait();
      }
    }
  }
```

```
thread t2
  synchronized (g) {
    t1.interrupt();
  }
```

**Figure 3.4:** Two threads with locks, wait sets, and interrupts

The multithreaded semantics leaves the shared memory $h :: 'h$ uninterpreted, it just passes $h$ between the threads. The projection functions `locks`, `thr`, `shr`, `wset`, and `intrs` return the locks, the thread pool, the shared memory, the wait sets, and the pending interrupts of a state, respectively.

For example, consider Figure 3.4 with two threads $t_1$ and $t_2$. Suppose $t_1$ executes first until it enters the wait set of monitor $g$ and then $t_2$ executes until it has interrupted $t_1$. Suppose further that $f$ references an object at address $f$, and similarly for $g$. Then, this state is represented by the tuple $(ls, (ts, h), ws, is)$ where

- $ls = [f \mapsto_f (t_1, 1), g \mapsto_f (t_2, 0)]$, i.e., threads $t_1$ and $t_2$ hold the locks $f$ and $g$ twice and once, respectively; all other locks are free.

- $ts = [t_1 \mapsto (\ldots, (K^f 0) (g :=_f 1)), t_2 \mapsto (\ldots, K^f 0)]$, i.e., the thread pool stores the thread-local states (omitted) and the temporarily released locks. Thread $t_1$ has temporarily released the lock on $g$ which it had held once before. Thread $t_2$ has not temporarily released any locks.

- $h = [f \mapsto \text{Obj } \ldots, g \mapsto \text{Obj } \ldots, t_1 \mapsto \text{Obj Thread } \ldots, t_2 \mapsto \text{Obj Thread } \ldots]$, i.e., the shared heap contains the objects referenced by $f$ and $g$, and the objects associated with threads $t_1$ and $t_2$.

- $ws = [t_1 \mapsto \text{WS-WokenUp}]$, i.e., an interrupt has removed thread $t_1$ from $g$’s wait set.

- $is = \{ t_1 \}$, i.e., only thread $t_1$ has a pending interrupt.
3.1.2 Thread actions

The framework manages the multithreaded state, but the single-threaded semantics needs to query and manipulate that state. To keep them separated, the latter must not directly access that state. Instead, it uses a thread action to spawn a new thread, to acquire a lock, to wake another thread, etc., which provides a clear and restricted interface to access and update the multithreaded state. Thus, I can decompose proofs about the multithreaded semantics to the level of threads, because interaction between threads can only happen via thread actions (and the shared memory).

Reductions in the single-threaded semantics carry a thread action as label, i.e., I will adapt the sequential semantics from Chapter 2 accordingly (§3.2). When the interleaving semantics picks a reduction, it changes the multithreaded state according to the thread action. Note that these actions are the only means of “communication” between the two levels. Since this is unidirectional, the interleaving semantics can transfer information to the single-threaded semantics only by picking one reduction that the latter offers. Hence, the single-threaded semantics must anticipate in its reductions all possible answers it is willing to accept from the interleaving semantics.

Syntax of thread actions

Thread actions are composed of multiple basic thread actions (BTA). The framework implements 17 different BTAs, which can be split in five groups. Figure 3.5 shows the type definitions.

Locking  *Lock* acquires a lock once for the current thread *t* – no other thread may hold the lock. *Unlock* releases it once, provided *t* is holding it. *Release* temporarily releases the lock if *t* holds it, and has no effect otherwise. *UnlockFail* tests whether *t* does not hold a lock, i.e., unlocking it would fail. There is no counterpart to *UnlockFail* for testing whether a thread holds a lock. This can be simulated by [*Unlock*, *Lock*], because the interleaving semantics picks a reduction only if the current state satisfies the precondition of all BTAs of the thread action.
3.1. Framework for interleaving semantics

datatype lock-act = Lock | Unlock | UnlockFail | Release
datatype (′t, ′x, ′h) new-thread-act = Spawn ′t ′x ′h | ThreadEx ′t bool
datatype ′t condition-act = Join ′t | Yield
datatype (′t, ′w) wait-set-act =
    Suspend ′w | Notify ′w | NotifyAll ′w | WakeUp ′t | Notified | WokenUp
datatype ′t intr-act = Intr ′t | ClearIntr ′t | IsIntrd ′t bool

Figure 3.5: Type definitions for basic thread actions

Thread creation  Spawn t x h spawns a new thread with ID t and initial
    local state x. There must not yet be a thread with ID t. Later (§3.1.4),
    it will be convenient to remember the shared heap h at spawn time.
    ThreadEx t b tests whether there is a thread with ID t in the thread
    pool, where b :: bool denotes the result.

Condition actions  Join t joins on thread t, i.e., t must have terminated
    before.  Yield causes the current thread to pause temporarily and
    allow other threads to execute.  Yield is only relevant for schedulers
    (§6.3.2).

Wait sets  Suspend w inserts the current thread into the wait set w, any
    previous assignment is lost.  Notify w (NotifyAll w) wakes up one
    (all) of the threads in the wait set w, their wait set status becomes
    ⌊WS-Notified⌋ (see Figure 3.3).  If w is empty, no thread is woken up.
    WakeUp t changes t’s wait set status to ⌊WS-WokenUp⌋, if it has been
    in a wait set before. Otherwise, nothing happens.

    Notified and WokenUp label reductions that process the notification
    and wake-up for the thread that has been notified or woken up.

Interruption  Intr t adds t to the set of interrupted threads; ClearIntr t
    removes it.  IsIntrd t b tests whether the set of interrupted threads
    contains t, i.e., whether t has a pending interrupt; b :: bool denotes
    the result.

To remember the lock on which BTAs for locking operate, they are
    arranged in a FinFun las :: ′l lock-acts such that the BTAs for lock l are
    listed in lasf l. In summary, a thread action consists of a FinFun for the
    BTAs for locks and one list for each of the other groups.
type_synonym 'l lock-acts = 'l ⇒_f lock-act list

type_synonym (('l, 't, 'x, 'h, 'w, 'o) thread-action =
  'l lock-acts × ('t, 'x, 'h) new-thread-act list × 't condition-act list ×
  ('t, 'w) wait-set-act list × 't intr-act list × 'o list)

Since thread actions are used as labels for the reductions, they include
a sixth component of type 'o list for further extensions (§4.3.2). The
projection functions ⟨ta⟩_l, ⟨ta⟩_t, ⟨ta⟩_c, ⟨ta⟩_w, ⟨ta⟩_i, and ⟨ta⟩_o extract from
the thread action ta the BTAs for locks, thread creation, conditions, wait
sets, interrupts, and the extension part, respectively.

All BTAs of a thread action are executed in a single reduction; if there
is at least one BTA in the thread action of a reduction whose precondition
is not met, the interleaving semantics does not select the reduction.
Thus, a single-thread semantics can affect and query multiple parts of
the multithreaded state in one step by composing BTAs. Since every
BTA (except Release) affects only one part of the state, the interleaving
semantics remains flexible and the proofs simple.

For example, in Java, a call to the wait method must test that the
thread t has not been interrupted and that it holds the lock l associated
with the receiver object, release the latter, and suspend t to the associated
wait set w. This can be expressed by the following thread action:

((K^f [])(l :=_f [Unlock, Lock, Release]),
 [], [], [Suspend w], [IsIntrd t False], ...)

As this notation is cumbersome to read and write, I use a list-like notation
for thread actions, with lock identifiers added to lock BTAs. The Isabelle
parser and pretty printer are set up such that they automatically convert
it into the corresponding thread action. Hence, the above thread action
is written as

 Unlock→l, Lock→l, Release→l, Suspend w, IsIntrd t False, ...

Let’s now examine how the thread action achieves its goal. First,
Unlock→l, Lock→l checks that the current thread holds l without effect-
tively changing the lock status. Then, Release→l releases l and Suspend w
adds the thread to the wait set. IsIntrd t False tests that the thread t has
no pending interrupt.

Note that the order of BTAs of the same group (and lock identifier)
is important. For example, (Unlock→l, Unlock→l) does not alter the lock

68
3.1. Framework for interleaving semantics

\[
\text{upd-}L :: \cdot \text{lock} \Rightarrow \cdot \Rightarrow \text{lock-act} \Rightarrow \cdot \text{lock}
\]

\[
\text{upd-}L \; L \; t \; \text{Lock} = \text{lock-lock} \; L \; t
\]

\[
\text{upd-}L \; L \; t \; \text{Unlock} = \text{unlock-lock} \; L
\]

\[
\text{upd-}L \; L \; t \; \text{UnlockFail} = L
\]

\[
\text{upd-}L \; L \; t \; \text{Release} = \text{release-lock} \; L \; t
\]

\[
\text{upd-}Ls :: \cdot \text{lock} \Rightarrow \cdot \Rightarrow \text{lock-act list} \Rightarrow \cdot \text{lock}
\]

\[
\text{upd-}Ls \; L \; t \; [] = L
\]

\[
\text{upd-}Ls \; L \; t \; (\text{la} \cdot \text{las}) = \text{upd-}Ls \; (\text{upd-}L \; L \; t \; \text{la}) \; t \; \text{las}
\]

\[
\text{upd-locks} :: (l, \cdot t) \; \text{locks} \Rightarrow \cdot \Rightarrow (l, \cdot t) \; \text{lock-acts} \Rightarrow (l, \cdot t) \; \text{locks}
\]

\[
(\text{upd-locks} \; l s \; t \; \text{las})_l = \text{upd-}Ls \; (l s_l) \; t \; (l s_{l_l})
\]

**Figure 3.6:** Update functions of the lock status for lock BTAs

state either, but checks that no other thread holds \(l\). Conversely, BTAs of different groups are unordered, even though the \(\langle \ldots \rangle\) notation might conjure up the illusion of a total ordering.

**Semantics of thread actions**

The semantics for BTAs follows their division in groups. For each group, there are update functions for the affected parts of the multithreaded states and predicates to check the preconditions.

**Lock actions** For lock BTAs, Figure 3.6 shows the update functions of the lock status. The function \(\text{upd-}L\) maps lock BTAs to the operations on locks from §3.1.1. Note that \(\text{UnlockFail}\) does not change the lock because this BTA only queries the lock status. The functions \(\text{upd-}Ls\) and \(\text{upd-locks}\) lift this function to lists of lock BTAs for a single lock and to FinFuns for all locks, respectively.

The preconditions for the lock actions are shown in Figure 3.7. These functions exactly follow the pattern for the update functions: \(\text{ok-}L\) translates single BTAs to operations on locks, \(\text{ok-}Ls\) lifts \(\text{ok-}L\) to lists, and \(\text{ok-locks}\) checks all lists in the lock FinFun of a thread action simultaneously. Note how \(\text{ok-}Ls \; L \; t \; (\text{la} \cdot \text{las})\) updates the lock \(L\) such that checking the remaining BTAs \(\text{las}\) takes the effect of the first BTA \(\text{la}\) on \(L\) into account.

Since the locks that a thread has temporarily released are stored separately from the lock status, there are update functions for that part,
Chapter 3. Interleaving semantics

\[
ok-L :: 't lock \Rightarrow 't \Rightarrow \text{lock-act} \Rightarrow \text{bool}
\]
\[
ok-L L t \text{ Lock} = \text{may-lock} L t
\]
\[
ok-L L t \text{ Unlock} = \text{has-lock} L t
\]
\[
ok-L L t \text{ UnlockFail} = \lnot\text{has-lock} L t
\]
\[
ok-L L t \text{ Release} = \text{True}
\]

\[
ok-Ls :: 't lock \Rightarrow 't \Rightarrow \text{lock-act list} \Rightarrow \text{bool}
\]
\[
ok-Ls L t [] = \text{True}
\]
\[
ok-Ls L t (la \cdot las) = ok-L L t la \land ok-Ls (\text{upd-L} L t la) t las
\]

\[
ok-locks :: ('l', 't) \text{ locks} \Rightarrow 't \Rightarrow 'l \text{ lock-acts} \Rightarrow \text{bool}
\]
\[
ok-locks ls t las = (\forall l. ok-Ls (ls_f l) t (las_f l))
\]

**Figure 3.7:** Preconditions for lock BTAs

\[
upd-trl :: \text{nat} \Rightarrow 't lock \Rightarrow 't \Rightarrow \text{lock-act} \Rightarrow \text{nat}
\]
\[
upd-trl n L t \text{ Release} = n + \text{has-locks} L t
\]
\[
 upd-trl n L t_ = n
\]

\[
 upd-trls :: \text{nat} \Rightarrow 't lock \Rightarrow 't \Rightarrow \text{lock-act list} \Rightarrow \text{nat}
\]
\[
 upd-trls n L t [] = n
\]
\[
 upd-trls n L t (la \cdot las) = \text{upd-trls} (\text{upd-trl} n L t la) (\text{upd-L} L t la) t las
\]

\[
 upd-TRL :: 'l \text{ tr-locks} \Rightarrow ('l', 't) \text{ locks} \Rightarrow 't \Rightarrow 'l \text{ lock-acts} \Rightarrow 'l \text{ tr-locks}
\]
\[
(\text{upd-TRL} ln ls t las)_f l = \text{upd-trls} (ln_f l) (ls_f l) t (las_f l)
\]

**Figure 3.8:** Update functions for temporarily released locks

too (Figure 3.8), which follow the same pattern. Remember that for a fixed lock ID, the state of the temporarily released locks is just the number of times the thread had held it.

**Thread creation actions** For thread creation BTAs, the functions \textit{upd-thr} and \textit{upd-thrs} update the thread pool (Figure 3.9). Spawned threads are stored under their thread ID with the initial state given in the BTA and no temporarily released locks. The predicates \textit{ok-thr} and \textit{ok-thrs} check the preconditions (Figure 3.10), i.e., \textit{t} is a free thread ID for \textit{Spawn t x m}, and \textit{b} in \textit{ThreadEx t b} expresses whether \textit{t} is not a free thread ID. Thread IDs are free iff they are not in the domain of the thread pool map.
3.1. Framework for interleaving semantics

upd-thr :: (l, t', x) tpool ⇒ (t, x, h) new-thread-act ⇒ (l, t', x) tpool
upd-thr ts (Spawn t x m) = ts(t ↦ (x, K^0))
upd-thr ts (ThreadEx t b) = ts

upd-thrs ::
(l, t', x) tpool ⇒ (t', x, h) new-thread-act list ⇒ (l, t', x) tpool
upd-thrs ts [] = ts
upd-thrs ts (nt ⋆ nts) = upd-thrs (upd-thr ts nt) nts

Figure 3.9: Update functions for thread creation BTAs

free-thread-id :: (l, t', x) tpool ⇒ t ⇒ bool
free-thread-id ts t = (ts t = None)

ok-thr :: (l, t', x) tpool ⇒ (t', x, h) new-thread-act ⇒ bool
ok-thr ts (Spawn t x m) = free-thread-id ts t
ok-thr ts (ThreadEx t b) = (b ≠ free-thread-id ts t)

ok-thrs :: (l, t', x) tpool ⇒ (t', x, h) new-thread-act list ⇒ bool
ok-thrs ts [] = True
ok-thrs ts (nt ⋆ nts) = ok-thr ts nt ∧ ok-thrs (upd-thr ts nt) nts

Figure 3.10: Preconditions for thread creation BTAs

Note that Spawn t x m has the same precondition as ThreadEx t False, which seems redundant at first sight. However, this redundancy simplifies the theorems and proofs because Spawn BTAs cannot override existing threads. Hence, I do not have to formalise that a thread action tests whether a thread t exists before it spawns t.

Condition actions Condition actions do not affect the multithreaded state. Hence, there are no update functions, but only predicates for the preconditions, see Figure 3.11. A thread t successfully joins on the thread t' iff

• t' has not yet been started, i.e., thr s t = None, or

• t' is not the executing thread itself, t' has been fully evaluated, not temporarily released any locks and is not in any wait set.
locale final-thread = fixes final :: 'x ⇒ bool

ok-cond :: (′l,′t,′x,′h,′w) state ⇒ 't ⇒ 't condition-act ⇒ bool
ok-cond s t (Join t′) = 
  case thr s t′ of None ⇒ True
  ∣ [(x, ln)] ⇒ t ≠ t′ ∧ final x ∧ ln = K^f 0 ∧ wset s t′ = None
ok-cond s t Yield = True

ok-conds :: (′l,′t,′x,′h,′w) state ⇒ 't ⇒ 't condition-act list ⇒ bool
ok-conds s t cas = (∀ca ∈ set cas. ok-cond s t ca)

Figure 3.11: Predicates for condition actions

The predicate final on the thread-local state determines if t′ has been
fully evaluated. For modularity, final is an implicit parameter to ok-cond
which source code and bytecode will instantiate (§3.2.2 and §3.2.3). In
Isabelle, I use the locale declaration to hide the final parameter.
The predicate ok-conds lifts ok-cond to lists cas. As there is no update
function, it merely conjoins the preconditions of all actions in cas.

Wait set actions For updating the wait sets, I define the relation t ⊩
ws = wa ⇒ ws′ where t denotes the executing thread, ws and ws′ the
original and successor wait sets, and wa the wait set action to be executed.
Unlike the other update functions, it is a relation, because Notify w
non-deterministically picks one thread t′ from the wait set.10 The rules
in Figure 3.12 implement the wait set automaton from Figure 3.3. For
Notify w, the update relation removes one arbitrary thread t′ from w if
there is any. In contrast, WakeUp t′ is deterministic as it removes t′ from
any wait set it has been suspended to. The BTAs Notified and WokenUp
reset t′s wait set status to None for normal execution.

As before, I lift _ ⊩ _ = _ ⇒ _ to lists of BTAs. To that end, I define
the reflexive, transitive closure r*** :: 'a ⇒ 'b list ⇒ 'a ⇒ bool of a ternary
relation r :: 'a ⇒ 'b ⇒ 'a ⇒ bool as

\[ r*** a [] a \]  \[ r*** a bs a' r a' b a'' \]  \[ r*** a (bs @ [b]) a'' \]

10I model this as a non-deterministic relation rather than an underspecified function,
using, e.g., Hilbert’s choice operator, because the code generator cannot deal with this kind
of underspecification (§6.4).
3.1. Framework for interleaving semantics

\[
t \vdash ws = \text{Suspend} w \Rightarrow ws(t \mapsto \text{InWS} w)
\]

\[
ws t' = [\text{InWS} w] \quad \forall t. \ ws t' \neq [\text{InWS} w]
\]

\[
t \vdash ws = \text{Notify} w \Rightarrow ws(t' \mapsto \text{WS-Notified})
\]

\[
\forall w. \ ws t' \neq [\text{InWS} w]
\]

\[
t \vdash ws = \text{NotifyAll} w \Rightarrow
\lambda t. \ if \ ws t = [\text{InWS} w] \ then \ [\text{WS-Notified}] \ else \ ws t
\]

\[
ws t' = [\text{InWS} w] \quad \forall t. \ ws t' \neq [\text{InWS} w]
\]

\[
t \vdash ws = \text{WakeUp} t' \Rightarrow
\text{ws}(t' \mapsto \text{WS-WokenUp})
\]

\[
\forall w. \ ws t' \neq [\text{InWS} w]
\]

\[
t \vdash ws = \text{Notified} \Rightarrow ws(t := \text{None}) \quad t \vdash ws = \text{WokenUp} \Rightarrow ws(t := \text{None})
\]

Figure 3.12: Update relation for wait sets

Then, \( t \vdash_\_ [=\Rightarrow] \_ \) is the reflexive, transitive closure of \( t \vdash_\_ [=] \Rightarrow \_), i.e., the former folds the latter over a list of wait set actions.

Unlike the other BTAs, the ones for wait sets do not have individual preconditions on the state. Rather, \([\text{Notified}]\) and \([\text{WokenUp}]\) characterise reductions of \( t \) that are meant to process notifications and interruptions. The predicate \( \text{ok-wsets} \) formalises that such reductions require the wait set status to be \([\text{WS-Notified}]\) and \([\text{WS-WokenUp}]\), respectively, and all other reductions require the wait set status to be \( \text{None} \).

\[
\text{ok-wsets} \ ws t \ was =
\quad if \ \text{Notified} \in \text{set} \ was \ then \ ws t = [\text{WS-Notified}]
\quad else if \ \text{WokenUp} \in \text{set} \ was \ then \ ws t = [\text{WS-WokenUp}]
\quad else \ ws t = \text{None}
\]

In particular, the rules for \( \_ \vdash \_ [=\Rightarrow] \_ \) and \( \_ \vdash \_ [\text{Notified}] \Rightarrow \_ \) and \( \_ \vdash \_ [\text{WokenUp}] \Rightarrow \_ \) in Figure 3.12 are applicable unconditionally.

**Interrupt actions** Figure 3.13 defines the update functions \( \text{upd-int} \) and \( \text{upd-ints} \) and predicates \( \text{ok-intr} \) and \( \text{ok-intrs} \) for interrupt actions. The interleaving semantics merely manages the set of interrupts, but it attaches no specific behaviour to them, nor does it impose any form of
access control. Note that Intr and ClearIntr have no preconditions, they can change the interrupts of any thread, even non-existing ones. IsIntrd tests for pending interrupts similar to ThreadEx for thread existence.

As all BTAs in a thread action must be checked and executed in one step, it is straight-forward to implement a compare-and-swap operation on the interrupt status: For example, \((\text{IsIntrd } t \ True, \text{ClearIntr } t)\) checks that \(t\) is interrupted and clears it atomically.

**Thread actions** Now, I combine all the update functions and predicates for basic thread actions into a single one for thread actions. ok-ta checks the preconditions of all five groups, and upd-ta combines the update functions, except for upd-TRL, which the interleaving semantics will apply directly when it updates the thread-local state.

\[
\text{ok-ta} :: (l,t,x,h,w) \text{ state} \Rightarrow (l,t,x,h,w,v) \text{ thread-action} \Rightarrow \text{bool}
\]
\[
\text{ok-ta} \ s \ t \ t a = \text{ok-locks (locks} s) \ t \langle t a \rangle _1 \land \text{ok-thrs (thr} s) \ t \langle t a \rangle _t \land
\]
\[
\text{ok-conds} \ s \ t \ t \langle t a \rangle _c \land \text{ok-wsets (wset} s) \ t \ t \langle t a \rangle _w \land \text{ok-intrs (intrs} s) \ t \ t \langle t a \rangle _i
\]

Figure 3.13: Update functions and predicates for interruption BTAs
Design considerations

I have designed the basic thread actions such that each depends on and affects only one part of the state, if possible. This also guided the partitioning of the multithreaded state into five components. I store locks separately from threads because all threads need to access them. The locks that a thread has released temporarily, however, are local to the thread and thus stored together with its local state in the thread pool. As Notify and WakeUp modify the wait set status of other threads, the wait set status is split off the thread pool in a separate map. The interrupt status of a thread is like a global variable that all threads can access and modify. Thus, it could also have been part of the shared heap, which the single-threaded semantics manages. However, I model it explicitly in the interleaving semantics because interruption is relevant for deadlocks (see §3.3).

The multithreaded state must satisfy simultaneously the preconditions of all BTAs of which a thread action is composed. This is a powerful means to the single thread semantics to express preconditions for certain reductions. In fact, it is more expressive than what I need for Java threads. For example, no thread simultaneously joins on a thread and spawns a new thread. Thus, I am free to define the semantics of combining BTAs such that proofs are as modular and simple as possible. For example, the preconditions of the different parts of a thread action are checked independently on the original state (3.1).

3.1.3 Interleaving semantics

In this section, I put everything together that I have defined so far to obtain the interleaving semantics in the framework. It takes the single-threaded semantics as a parameter \( r :: (l', t', x, h', w', o) \) semantics which source and bytecode will instantiate accordingly (§3.2).
Chapter 3. Interleaving semantics

N

\[ \text{thr } s \ t = \lfloor (x, K^f_0) \rfloor \quad t \vdash (x, \text{shr } s) - ta \rightarrow (x', h') \]

ok-ta \ s \ t \ ta \quad \text{upd-ta } s \ t \ ta \ (ls', (ts', h), ws', is')

\[ s' = (ls', (ts' (t \mapsto (x', \text{upd-TRL } ls \ t (K^f_0) (ta_1)), h'), ws', is')) \]

\[ \text{NORMAL } \quad s - t: ta \rightarrow s' \]

Acquire

\[ \text{thr } s \ t = \lfloor (x, ln) \rfloor \quad ln \neq K^f_0 \]

\[ \neg \text{waiting } (wset \ s \ t) \quad \text{may-acquire-TRL } (\text{locks } s) \ t \ ln \]

\[ s - t: (K^f [], [], [], [], [], \text{acq-events } ln) \rightarrow \text{upd-acq } s \ t \ x \ ln \]

Figure 3.14: Reductions in the interleaving semantics

\textbf{type_synonym} \ ((l, t', x', h', w', o)) \ \text{semantics} =

\[ t' \Rightarrow x' \times h' \Rightarrow (l, t', x', h', w', o) \ \text{thread-action} \Rightarrow x' \times h \Rightarrow \text{bool} \]

In Isabelle, the locale \textit{multithreaded-base} fixes the parameters \textit{final} (inherited from \textit{final-thread}), \textit{r}, and \textit{acq-events}:

\textbf{locale} \textit{multithreaded-base} = \textit{final-thread} +

fixes \textit{r} :: (l, t', x', h', w', o) \ \text{semantics} \quad (_ \vdash _ - _ \rightarrow _) \quad (3.2)

and \textit{acq-events} :: l \ tr-locks \Rightarrow o \ \text{list}

A single-step reduction \textit{rt} (x, h) \ \textit{ta} (x', h') is written \textit{t} \vdash (x, h) - \textit{ta} \rightarrow (x', h'). It denotes that the thread with ID \textit{t} can atomically reduce in the thread-local state \textit{x} with shared heap \textit{h} to the thread-local state \textit{x'} with the new heap \textit{h'} with thread action \textit{ta}.

The last parameter \textit{acq-events} of the locale produces the label (i.e., the sixth component of a thread action) for when a thread reacquires its temporarily released locks. Although these labels only become relevant in Chapter 4, I include them in the formalisation now such that I do not have to adapt the definitions later.

Figure 3.14 shows the definition of the atomic steps of the interleaving semantics \textit{redT} (with syntax \textit{s} - t: \textit{ta} \rightarrow \textit{s'}). The reductions of the interleaving semantics carry the ID \textit{t} of the executing thread and the executed thread action \textit{ta} as label.

\textbf{NORMAL} injects the atomic steps of the threads into the interleaving semantics. Given a thread \textit{t} with local state \textit{x} and without any temporarily released locks (\(K^f_0\)), if \textit{t} can reduce with the shared heap \textit{shr} \ \textit{s} to \textit{x'} and \textit{h'} with thread action \textit{ta}, the interleaving semantics tests whether
3.1. Framework for interleaving semantics

\(ta\)'s preconditions are met and, if so, applies \(ta\)'s effects to the state. In this new state, it updates \(t\)'s local state to \(x'\) and its temporarily released locks, and the shared heap to \(h'\), which yields the successor state.

The other rule \textsc{Acquire} reacquires the locks \(ln\) that a thread has temporarily released. Suppose \(t\) is a thread which has temporarily released some locks, i.e., \(ln \neq K^f 0\), and is not in a wait set, i.e., \(\neg \text{waiting} (wset s t)\). The predicate \(\text{may-acquire-TRL} :: (l,t) \text{ locks} \Rightarrow t \Rightarrow l \text{ tr-locks} \Rightarrow \text{bool}\) tests whether \(t\) may acquire all of its temporarily released locks.

\[
\text{may-acquire-TRL} \ l s t ln = (\forall l. ln_f l > 0 \rightarrow \text{may-lock} \ l s t l)
\]

If so, \(\text{acquire-TRL} :: (l,t) \text{ locks} \Rightarrow t \Rightarrow l \text{ tr-locks} \Rightarrow (l,t) \text{ locks}\) updates the lock state accordingly.

\[
(\text{acquire-TRL} \ l s t ln) l = \text{acquire-lock} (ls f l) t (ln f l)
\]

This produces the multithreaded state

\[
(\text{acquire-TRL} (\text{locks} s) t ln, ((\text{thr} s)(t \mapsto (x, K^f 0))), shr s, wset s, intrs s)
\]

which I abbreviate as \(\text{ upd-acq} \ s t x ln\). Note that \(t\)'s multiset of temporarily released locks is now empty. This reduction step carries as label the thread ID and a thread action without any BTAs, but the third parameter \(acq-events\) of the locale determines the last component of the thread action – I will define it in §4.3.2.

Reductions in the reflexive and transitive closure \(\text{redT}^{***}\) of the interleaving semantics are written as \(s \xrightarrow{ttas} * s'\). The list \(ttas\), which has the type \((t \times (l,t,'x,'n,'w,'o) \text{ thread-action}) \text{ list}\), collects all the labels, i.e., thread ID and thread action, of the individual steps.

A thread is final iff its local state is \textit{final}, its multiset of temporarily released locks is empty, and its wait set status is \textit{None}, i.e., neither is it in a wait set, nor has it been removed from one without having processed the removal. \(\text{final-threads} \ s\) denotes the set of all final threads in the multithreaded state \(s\). \(s\) itself is final (written \(m\text{final} \ s\)) iff all threads are final.

\[
thr s t = [ (x, K^f 0) ] \quad \text{final} x \quad \text{wset} s t = \text{None}
\]

\[
t \in \text{final-threads} \ s
\]

\[
m\text{final} \ s = (\text{dom} (\text{thr} s) \subseteq \text{final-threads} \ s)
\]
Chapter 3. Interleaving semantics

The locale *multithreaded-base* imposes no assumptions on its parameters, but for the interleaving semantics to work correctly, the single-threaded semantics must be well-behaved. The locale *multithreaded* collects the most fundamental constraints.

\[
\text{locale multithreaded = multithreaded-base +}
\]

\[
\text{assumes final-no-red: } [t \vdash (x, h) \rightarrow (x', h')]; \text{ final } x \Rightarrow False
\]

and Spawn-heap:

\[
[t \vdash (x, h) \rightarrow (x', h')]; \text{ Spawn } t'' x'' h'' \in \text{ set } \langle ta \rangle_t \Rightarrow h'' = h'
\]

*final-no-red* expresses that *final* states are final, i.e., they cannot reduce any further. For proofs, it is convenient to remember the shared heap at the time when a thread is spawned (see §3.1.4 and §5.1.3). The assumption *Spawn-heap* imposes that the shared heap in the BTAs indeed remembers the shared heap at spawn time.

*final-no-red* also implies that \( m_{\text{final}} \) states are final:

\[\text{Lemma 3.1. If } m_{\text{final}} s, \text{ then there are no } t, ta, s' \text{ such that } s \rightarrow t:ta \rightarrow s'.\]

### 3.1.4 Infrastructure for well-formedness constraints

Many theorems about the single-threaded semantics impose constraints on the state. For example, subject reduction (Theorem 2.2) requires conformance of the state, and progress (Theorem 2.1) requires definite assignment and heap conformance. For type safety, one must show that the semantics preserves these constraints (Lemmata 2.2 and 2.1 for state conformance and definite assignment, respectively). I now define the machinery to transfer such constraints and their preservation lemmata to the multithreaded semantics at little cost.

**Thread-local predicates**

Suppose that the predicate \( Q :: 't \Rightarrow 'x \Rightarrow 'h \Rightarrow \text{bool} \) denotes such a constraint. The operator \( \uparrow \_ \_ \uparrow \) lifts \( Q \) to a predicate of type \( ('t, 't, 'x) tpool \Rightarrow 'h \Rightarrow \text{bool} \) on the thread pool and shared heap such that \( \uparrow Q \uparrow \) imposes \( Q \) on all threads in the thread pool.

\[
\uparrow Q \uparrow ts h = (\forall t. \text{ case } ts t \text{ of } \text{None } \Rightarrow \text{True } || (x, ln) ] \Rightarrow Q t x h)
\]

In the case of definite assignment, e.g., \( Q = (\lambda t (e, xs) h. D e [\text{dom } xs]) \), since the definite assignment check \( D \) (§2.1.4) only depends on the
locale lifting-wf = multithreaded +
fixes Q :: 't ⇒ 'x ⇒ 'h ⇒ bool
assumes preserves-red: \[ t \vdash (x,h) \rightarrow (x',h'); Q t x h \] ⇒ Q t' x' h'
and preserves-Spawn:
\[ t \vdash (x,h) \rightarrow (x',h'); Q t x h; \text{Spawn } t'' x'' h' \in \text{set } \langle \text{ta} \rangle_t \] ⇒ Q t'' x'' h'
and preserves-other:
\[ t \vdash (x,h) \rightarrow (x',h'); Q t x h; Q t'' x'' h' \] ⇒ Q t'' x'' h'

Figure 3.15: Definition of locale lifting-wf

thread-local state, which has type \( \text{expr} \times \text{locals} \) in source code. Then, \( \uparrow Q \uparrow ts h \) denotes that all threads in the thread pool \( ts \) assign to all local variables before they are used provided that all variables in the thread’s local store have already been initialised.

To transfer the preservation lemma to the multithreaded state, I define the locale lifting-wf (see Figure 3.15). It fixes the constraint \( Q \) and assumes that

1. single-thread reductions preserves \( Q \) (preserves-red),

2. \( Q \) holds for new threads at the time of creation (preserves-Spawn), and

3. \( Q \) is preserved even if another thread, which also satisfies \( Q \), changes the shared heap in a reduction (preserves-other).

Under these assumptions, the interleaving semantics preserves \( \uparrow Q \uparrow \).

Lemma 3.2 (Preservation for \( \uparrow \_ \uparrow \)). Let \( \uparrow Q \uparrow (\text{thr } s) (\text{shr } s') \). If \( s \rightarrow t: \text{ta} \rightarrow s' \) or \( s \rightarrow t: \text{tas} \rightarrow s', \) then \( \uparrow Q \uparrow (\text{thr } s') (\text{shr } s') \), too.

Reconsider the definite assignment example. Lemma 2.1 discharges the first assumption of lifting-wf. As a spawned thread executes the run method of the associated Thread object, the well-formedness constraints for source code (§2.1.4) ensure preserves-Spawn. The last assumption preserves-other is vacuous for \( D \) because \( D \) does not depend on the heap.
Chapter 3. Interleaving semantics

Thread-local predicates with additional data

Some predicates on the thread level also need additional data, which is thread-specific, but invariant, e.g., a typing environment for the local store. I model such extra invariant data as maps from thread IDs to some type $'i$. Now, let $Q :: 'i \Rightarrow 'x \Rightarrow 'h \Rightarrow bool$ also include the invariant data. The operator $\lceil Q \rceil$ lifts $Q$ to thread pools similar to $\lceil \_ \_ \_ \rceil$.

$\lceil Q \rceil I ts h =
\forall t. \text{case } t \text{ of None } \Rightarrow True | [(x, ln)] \Rightarrow \exists i. I t = [i] \land Q i t x h$

where $I :: 't \rightarrow 'i$ is a map to invariant data. Such a map $I$ is well-formed with respect to the thread pool $ts$ (written $ts \vdash i I$) iff their domains are equal.

In the case of state conformance from §2.1.6, for example, $I$ stores the typing environment $E$ for the thread’s local store. Hence, the constraint is $\lambda E t (e, xs) h. P, E \vdash (h, xs) \checkmark$.

Let $I(nts \Rightarrow Q)$ denote the extension of $I$ with invariant data for threads spawned in $nts$. For all $\text{Spawn } t x h \in \text{set } nts, I(nts \Rightarrow Q)$ updates $I$ at $t$ to $\varepsilon i. Q i t x h$.

$I([], \Rightarrow Q) = I$
$I(Spawn t x h \cdot nts \Rightarrow Q) = (I(t \mapsto \varepsilon i. Q i t x h))(nts \Rightarrow Q)$
$I(ThreadEx t b \cdot nts \Rightarrow Q) = I(nts \Rightarrow Q)$

(3.3)

For labels $ttas$ of the reflexive and transitive closure $\_ \_ \_ \Rightarrow^* \_ \_ \_$, let $I(ttas [\Rightarrow] Q)$ denote the extension

$I(\text{concat } (\text{map } (\lambda (t, ta). \langle ta \rangle_t) ttas) \Rightarrow Q)$.

The term $\text{map } (\lambda (t, ta). \langle ta \rangle_t) ttas$ extracts all thread creation BTAs from $ttas$, $\text{concat}$ combines them in one list. The extension preserves well-formedness of maps to invariant data.

**Lemma 3.3.** Suppose $\text{thr } s \vdash i I$. If $s \rightarrow t:ta \Rightarrow s’, \text{then } \text{thr } s’ \vdash i I(\langle ta \rangle_t \Rightarrow Q)$. If $s \rightarrow ttas \Rightarrow^* s’, \text{then } \text{thr } s’ \vdash i I(ttas [\Rightarrow] Q)$.

Equation 3.3 shows why it is necessary to remember the shared heap in $\text{Spawn}$ actions. $\_ \_ (\Rightarrow Q)$ must know the heap at spawn time to choose the right invariant data, because it may depend on the heap at
creation time. Since the transitive, reflexive closure discards the heaps of intermediate steps, I must store it in the thread actions.\footnote{Alternatively, one might try to choose the invariant data for spawned heaps w.r.t. to some underspecified intermediate heap. In that case, transitivity of \(\_ \_ \_ \rightarrow^* \_\_\) does not carry over to extensions because the possible intermediate heaps may change. But with the above approach, \(I(nts \ @ \ nts' \rightarrow^* Q) = I(nts \rightarrow^* Q)(nts' \rightarrow^* Q)\) holds.}

Similarly to \textit{lifting-wf}, the locale \textit{lifting-inv} collects the assumptions for lifting the preservation theorems. The main difference is that \textit{lifting-inv} existentially quantifies over the invariant data for spawned threads. In fact, \textit{lifting-wf} is just the special case of \textit{lifting-inv} with the constraint instantiated to \(\lambda_\_Q\).

\begin{verbatim}
locale lifting-inv = multithreaded +
  fixes Q :: '_ \Rightarrow 't \Rightarrow 'x \Rightarrow 'h \Rightarrow bool
  assumes \(\|t \vdash (x,h) \rightarrow^* (x',h')\; Q \; i \; t \; x \; h\]\ \(\Rightarrow\) \(Q \; i \; t \; x' \; h'\)
  and \(\|t \vdash (x,h) \rightarrow^* (x',h')\; Q \; i \; t \; x \; h\; \text{Spawn} \; t'' \; x'' \; h'' \in \text{set} \langle ta \rangle_t\]\ \(\Rightarrow\) \(\exists i''. \; Q \; i'' \; t'' \; x'' \; h''\)
  and \(\|t \vdash (x,h) \rightarrow^* (x',h')\; Q \; i \; t \; x \; h\; Q \; i'' \; t'' \; x'' \; h''\]\ \(\Rightarrow\) \(Q \; i'' \; t'' \; x'' \; h''\)
\end{verbatim}

Analogous to Lemma 3.2, the next lemma shows that these assumptions are sufficient for the interleaving semantics preserving \(\uparrow \Box Q \uparrow\).

\textbf{Lemma 3.4 (Preservation for \(\uparrow \Box \_\_\_\))}. Suppose that \(\uparrow \Box I (\text{thr} \; s) \; (\text{shr} \; s)\).

\begin{enumerate}
  \item If \(s \rightarrow^* t : \text{ta} \rightarrow s'\), then \(\uparrow \Box I(\langle \text{ta} \rangle_t \rightarrow^* Q) \; (\text{thr} \; s') \; (\text{shr} \; s')\).
  \item If \(s \rightarrow^* \text{ttas} \rightarrow^* s'\), then \(\uparrow \Box I(\text{ttas} \rightarrow^* Q) \; (\text{thr} \; s') \; (\text{shr} \; s')\).
\end{enumerate}

\textbf{General invariants}

The above machinery works only for constraints that depend on the thread-local state and the shared heap only, but some lemmata need to impose constraints on the complete state. I have not developed any setup for such cases, but deal with them on a per-case basis. Nevertheless, I define a predicate \textit{invariant} to succinctly express preservation for arbitrary ternary relations \(r\) and sets of states \(\text{invar}\), i.e., that \(\text{invar}\) is \(r\)-closed:

\[\text{invariant} \; r \; \text{invar} = (\forall a \; b \; a'. \; a \in \text{invar} \rightarrow r \; a \; b \; a' \rightarrow a' \in \text{invar})\]

\textbf{Lemma 3.5}. Let \(a \in \text{invar}\) and \textbf{invariant} \(r \; \text{invar}\). If \(r \; a \; b \; a'\) or \(r^* \; a \; b \; a'\), then \(a' \in \text{invar}\).
3.2 Multithreading in JinjaThreads

The framework for interleaving semantics from the previous section leaves the single-threaded semantics abstract. In this section, I define the single-threaded semantics for source code (§3.2.2) and bytecode (§3.2.3), which are adaptations and extensions of the sequential semantics from §2.1.5 and §2.2.2, respectively. They describe the full behaviour of a single thread, including all synchronisation and communication with other threads, e.g., via thread actions. Hence, I present how to implement Java concurrency in terms of thread actions. The multithreaded semantics of source code and bytecode specialises the framework for interleaving semantics with the respective single-threaded semantics. Since Java provides most synchronisation as native methods, I define them first (§3.2.1) such that source code and bytecode can share their implementation.

3.2.1 Native methods for synchronisation

This section concentrates on the multithreading features that Java provides via method calls. I implement them as native methods, reusing the formalism for native methods in sequential JinjaThreads from §2.1.3 and §2.1.5.

Signatures I add to the signatures for native methods from (2.3) the signatures shown in Figure 3.16. The first group of four are the methods of Thread for (i) spawning (start), (ii) joining on (join), (iii) interrupting (interrupt), and (iv) testing for interruption (isInterrupted) of a thread.

The second group of methods consists of static methods in class Thread in Java. Since JinjaThreads lacks static methods, I moved them to class Object such that they can be called from every method via the this pointer. They all operate on the current thread: currentThread returns the associated object, interrupted checks and clears the interrupt status, and yield advises the scheduler to schedule other threads if possible.

The last group declares Object’s methods for the wait-notify-mechanism.

---

12To ensure that this change does not affect method lookup, the method names currentThread, interrupted, and yield internally use characters that do not occur in normal Java method names.
3.2. Multithreading in JinjaThreads

<table>
<thead>
<tr>
<th>Method</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Thread.start()</code></td>
<td>Void</td>
</tr>
<tr>
<td><code>Thread.join()</code></td>
<td>Void</td>
</tr>
<tr>
<td><code>Thread.interrupt()</code></td>
<td>Void</td>
</tr>
<tr>
<td><code>Thread.isInterrupted()</code></td>
<td>Boolean</td>
</tr>
<tr>
<td><code>Object.currentThread()</code></td>
<td>Class Thread</td>
</tr>
<tr>
<td><code>Object.interrupted()</code></td>
<td>Boolean</td>
</tr>
<tr>
<td><code>Object.yield()</code></td>
<td>Void</td>
</tr>
<tr>
<td><code>Object.wait()</code></td>
<td>Void</td>
</tr>
<tr>
<td><code>Object.notify()</code></td>
<td>Void</td>
</tr>
<tr>
<td><code>Object.notifyAll()</code></td>
<td>Void</td>
</tr>
</tbody>
</table>

Figure 3.16: Signatures of native methods for Java concurrency

Recording thread interaction  Before I present the semantics for the new native methods, I define the sixth component (of type ‘o list in the interleaving semantics) of thread actions, although they will become relevant only in Chapter 4. During this chapter, they may be safely ignored. I introduce them now to include them in the semantics rules such that I do not have to change them later on.

The JinjaThreads semantics instantiate ‘o to the HOL type event, which records the heap operations and synchronisation events of a thread; Table 3.1 describes the elements. Although some of the synchronisation events are similar to BTAs, I keep them separate such that thread interactions for implementing correct interleaving are clearly distinguished from synchronisation that is relevant for the memory model. For example, $SUnlock$ represents the effects of releasing a lock on the shared memory; these are independent of how often a thread actually releases the lock using Unlock or Release BTAs. For a detailed discussion, see §4.3.1 and §4.3.2.

Well-formedness  Concurrency needs three additional system exceptions. Hence, I append `IllegalThreadState`, `IllegalMonitorState`, and `InterruptedException` to `sys-xcpts` from Equation 2.1. Consequently, generic well-formedness requires that every program declares them as subclasses of `Throwable` (cf. Figure 2.9). Moreover, $wf-syscls$ $P$ now also requires that $P$ declare the class `Thread`, and $wf-cdecl$ demands that
Chapter 3. Interleaving semantics

Table 3.1: Events record memory and synchronisation operations of a thread

<table>
<thead>
<tr>
<th>event</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>heap</td>
<td></td>
</tr>
<tr>
<td>Read $a \ a l \ v$</td>
<td>read $v$ from member $a l$ of address $a$</td>
</tr>
<tr>
<td>Write $a \ a l \ v$</td>
<td>write $v$ to member $a l$ of address $a$</td>
</tr>
<tr>
<td>Allocate $a \ h T$</td>
<td>allocate address $a$ for type information $h T$</td>
</tr>
<tr>
<td>New-Obj $a \ C$</td>
<td>abbreviation for Allocate $a$ (Class$T \ C$)</td>
</tr>
<tr>
<td>New-Arr $a \ T \ n$</td>
<td>abbreviation for Allocate $a$ (Array$T \ T \ n$)</td>
</tr>
<tr>
<td>synchronisation</td>
<td></td>
</tr>
<tr>
<td>TStart $t$</td>
<td>spawn thread $t$</td>
</tr>
<tr>
<td>TJoin $t$</td>
<td>join thread $t$</td>
</tr>
<tr>
<td>SLock $a$</td>
<td>acquire lock $a$ at least once</td>
</tr>
<tr>
<td>SUnlock $a$</td>
<td>release lock $a$ at least once</td>
</tr>
<tr>
<td>TIntr $t$</td>
<td>interrupt thread $t$</td>
</tr>
<tr>
<td>TIntrd $t$</td>
<td>determine that the thread $t$ has been interrupted</td>
</tr>
<tr>
<td>Extern $a \ M \ vs \ v$</td>
<td>call to external method $M$ on $a$ with parameters $vs$ and return value $v$, e.g., for printing</td>
</tr>
</tbody>
</table>

Thread declare a run method without parameters and with return type Void, i.e.,

$$C = \text{Thread} \rightarrow (\exists m. (\text{run}, [], \text{Void}, m) \in \text{set } ms)$$

is added conjunctively to wf-cdecl $wf-md P (C, D, fs, ms)$ in Figure 2.9.

Semantics To implement the new native methods, I extend the semantics for native methods $P, t \vdash \langle a.\mathcal{M}(vs), h \rangle \rightarrow_{\text{nc}} \langle va, h' \rangle$. Note that the semantics now takes the thread ID $t$ of the executing thread as an additional parameter and carries a thread action $ta$ as label. The former rules from Figure 2.12 are adapted accordingly. They also take the thread ID, but ignore it; their thread action label only records the heap interaction (see §4.1.2 for details), i.e., it is of the form $\langle \ldots \rangle$ where $\ldots$ is a list of events.

Now, I must also specify how source code and bytecode instantiate type variables in the interleaving semantics. Addresses (HOL type $addr$) identify locks (type variable $'l$) and wait sets ($'w$), because every object has one monitor and one wait set. The heap becomes the shared memory ($'h$). For thread IDs ($'t$), I use the opaque type thread-id with the operations $a2t :: addr \Rightarrow \text{thread-id}$ and $t2a :: \text{thread-id} \Rightarrow addr$ for
now – later, thread IDs and addresses will be the same (§4.2, §4.3.2). $a2t$ converts addresses of objects of (subclasses of) Thread into their associated thread ID, and $t2a$ is $a2t$’s left inverse on addresses at which such objects may be allocated.\textsuperscript{13} I defer instantiating the thread-local state ($\prime x$) because it differs between source code and bytecode.

The JLS specifies the native methods from Figure 3.16 only incompletely or not at all. Hence, I relied on the Java API \cite{JavaAPI} and, in case the JLS and API are ambiguous, on test runs of the Java HotSpot VM, version 1.6.0_22. The latter cases are marked as such.

Figure 3.17 shows the rules for the native methods of Thread, all of which carry the preconditions $\text{typeof-addr } h \ a = \lfloor \text{ClassT } C \rfloor, P \vdash C \leq^* \text{Thread}$, and – except for $\text{STARTFail}$ and $\text{JOINIntr} – t' = a2t \ a$. I have omitted them for conciseness.

Rule $\text{START}$ spawns a new thread which is associated with the receiver object. The new thread is to execute the run method of the receiver object. Since both source code and bytecode build on the semantics of native methods, $\text{START}$ cannot include the concrete initial state of the new thread in the $\text{Spawn}$ BTA, because their state representations differ. Instead, it specifies to execute (without parameters) the run method that class $C$ sees with receiver object $a$. The semantics for source code and bytecode has to convert this into the state representation as required. If the thread has already been started, $\text{STARTFail}$ raises an $\text{IllegalThreadState}$ exception.

The join method waits for the receiver thread to terminate, so $\text{JOIN}$ includes the basic thread action $\text{Join } t'$ where $t' = a2t \ a$. However, the API specifies that $\text{join}$ first has to test whether the current thread $t$ has not been interrupted. Otherwise, it raises an $\text{InterruptedException}$ ($\text{JOINIntr}$). Note that the interleaving semantics picks the reductions $\text{JOIN}$ or $\text{JOINIntr}$ only if their thread action’s precondition is satisfied. In particular, if $t'$ is not $\text{final}$ and $t$ is not interrupted, the call to $\text{join}$ gets stuck until either $t'$ terminates or $t$ gets interrupted.

Although the implementation of class Thread in Sun’s and Oracle’s JDK SE 6 declares (and has always declared in previous versions) the methods $\text{start}$ and $\text{join}$ as synchronized, neither the JLS nor the API require this. Hence, none of the rules for $\text{start}$ and $\text{join}$ includes Lock and Unlock actions, because such synchronisation would erroneously hide data races, which the Java memory model in §4.3 is about.

\textsuperscript{13}This restriction may seem overly complicated, because any object may be allocated at any address in the current heap model (see §2.1.5). However, it is relevant for more elaborated ones, e.g., when addresses store the thread’s ID that allocated it.
Chapter 3. Interleaving semantics

\[ \text{Start: } P, t \vdash \langle a.\text{start}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val Unit}, h \rangle \]

\[ \text{StartFail: } P, t \vdash \langle a.\text{start}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-sys-xcpt IllegalThreadState}, h \rangle \]

\[ \text{Join: } P, t \vdash \langle a.\text{join}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val Unit}, h \rangle \]

\[ \text{JoinIntr: } P, t \vdash \langle a.\text{join}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-sys-xcpt InterruptedException}, h \rangle \]

\[ \text{Intr: } P, t \vdash \langle a.\text{interrupt}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val Unit}, h \rangle \]

\[ \text{IntrInex: } P, t \vdash \langle a.\text{interrupt}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val Unit}, h \rangle \]

\[ \text{isIntrdT: } P, t \vdash \langle a.\text{isInterrupted}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val (Bool True)}, h \rangle \]

\[ \text{isIntrdF: } P, t \vdash \langle a.\text{isInterrupted}([], h) \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val (Bool False)}, h \rangle \]

**Figure 3.17:** Semantics of native methods of class Thread. All rules additionally have the premises typeof-addr \( h \ a = [\text{ClassT} \ C] \), \( P \vdash C \leq^{*} \text{Thread} \), and – except for StartFail and JoinIntr – \( t' = a2t \ a \).

When a thread interrupts another thread \( t' \) through the interrupt method (Intr), \( t' \) is removed from any wait set (BTA WakeUp \( t' \)) and its interrupt status is set (BTA Intr \( t' \)), if \( t' \) already exists as a thread in the thread pool (BTA ThreadEx \( t' \ True \)). However, if \( t' \) is merely a Thread object which has not yet been started, the call to interrupt has no effect (IntrInex). The isInterrupted method returns the interrupt status of the receiver thread \( t' = a2t \ a \) (isIntrdT and isIntrdF).

Figure 3.18 shows the rules for static native methods of Thread. A call to currentThread returns the address of the object associated with the current thread (CurrTh). It uses the function \( t2a \) to convert the thread ID \( t \) into the address of the associated object. The interrupted method returns and clears the interrupt status of the executing thread (IntrdT
3.2. Multithreading in JinjaThreads

\[
\text{CurrTh: } P, t \vdash \langle a.\text{currentThread}(), h \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val (Addr (t2a t)), h} \rangle
\]

\[
\text{IntrdT: } P, t \vdash \langle a.\text{interrupted}(), h \rangle \rightarrow \langle \text{IsIntrd } t \text{ True, ClearIntr } t, T\text{Intrd } t \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val (Bool True), h} \rangle
\]

\[
\text{IntrDF: } P, t \vdash \langle a.\text{interrupted}(), h \rangle \rightarrow \langle \text{IsIntrd } t \text{ False} \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val (Bool False), h} \rangle
\]

\[
\text{Yield: } P, t \vdash \langle a.\text{yield}(), h \rangle \rightarrow \langle \text{Yield} \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val Unit, h} \rangle
\]

Figure 3.18: Static native methods of Thread, implemented as methods of Object

and IntrDF). Calling yield simply emits the BTA Yield (Yield), which tells the scheduler to schedule another thread.

The rules for wait, notify, and notifyAll are more complicated (Figure 3.19). I start with notify and notifyAll. A call to either of them first tests via Unlock→a, Lock→a whether the current thread has locked the receiver object’s monitor a, without changing a’s lock state. If so, it emits the BTA Notify a or NotifyAll a, respectively (N\text{tf} and N\text{tfAll}). Otherwise, UnlockFail→a checks that the thread does not hold the lock; rules N\text{tfFail} and N\text{tfAllFail} then raise an IllegalMonitorState exception.

I now explain the rules for wait from Figure 3.19 by going through Figure 3.20. It shows the individual steps to execute the call to wait as state transitions. Every state is defined by four relevant components which the legend at the bottom explains. The transitions are labelled by the rules that generate them – solid lines are t’s steps, dashed lines denote reductions of other threads. From the three initial states at the top, the transitions lead to what the call returns, i.e., Unit or a system exception. Note how the transitions from the initial state in the centre step through the wait set automaton from Figure 3.3.

If the thread does not hold the lock on a (top left state), WaitFail raises an IllegalMonitorState exception. If t does hold the lock, wait tests whether t has a pending interrupt.\(^1\) If so (top right state), WaitIntrd1 clears it and raises an InterruptedException.

\(^1\)Neither the JLS [56, Ch. 17.8] nor the Java API [76] specify whether wait first tests for interrupts or for the lock on the monitor. JinjaThreads follows the HotSpot VM, which tests for the lock state first.
Chapter 3. Interleaving semantics

Ntf: \[ P, t \vdash \langle a.\text{notify}([], h) \rangle \to^{\text{nc}} \langle \text{Unlock} \to a, \text{Lock} \to a, \text{Notify} a \rangle \to_{\text{nc}} \langle \text{Ret-Val Unit}, h \rangle \]

NtfFail: \[ P, t \vdash \langle a.\text{notify}([], h) \rangle \to^{\text{nc}} \langle \text{UnlockFail} \to a \rangle \to_{\text{nc}} \langle \text{Ret-sys-xcpt IllegalMonitorState}, h \rangle \]

NtfAll: \[ P, t \vdash \langle a.\text{notifyAll}([], h) \rangle \to^{\text{nc}} \langle \text{Unlock} \to a, \text{Lock} \to a, \text{NotifyAll} a \rangle \to_{\text{nc}} \langle \text{Ret-Val Unit}, h \rangle \]

NtfAllFail: \[ P, t \vdash \langle a.\text{notifyAll}([], h) \rangle \to^{\text{nc}} \langle \text{UnlockFail} \to a \rangle \to_{\text{nc}} \langle \text{Ret-sys-xcpt IllegalMonitorState}, h \rangle \]

WaitFail: \[ P, t \vdash \langle a.\text{wait}([], h) \rangle \to^{\text{nc}} \langle \text{UnlockFail} \to a \rangle \to_{\text{nc}} \langle \text{Ret-sys-xcpt IllegalMonitorState}, h \rangle \]

WaitIntrd1: \[ P, t \vdash \langle a.\text{wait}([], h) \rangle \to^{\text{nc}} \langle \text{Unlock} \to a, \text{Lock} \to a, \text{IsIntrd} t \to \text{True}, \text{ClearIntr} t, \text{TIntrd} t \rangle \to_{\text{nc}} \langle \text{Ret-sys-xcpt InterruptedException}, h \rangle \]

WaitIntrd2: \[ P, t \vdash \langle a.\text{wait}([], h) \rangle \to^{\text{nc}} \langle \text{Unlock} \to a, \text{Lock} \to a, \text{Release} \to a, \text{IsIntrd} t \to \text{False}, \text{SUnclock} a \rangle \to_{\text{nc}} \langle \text{Ret-Unchanged}, h \rangle \]

WaitNtf: \[ P, t \vdash \langle a.\text{wait}([], h) \rangle \to^{\text{nc}} \langle \text{Notified} \rangle \to_{\text{nc}} \langle \text{Ret-Val Unit}, h \rangle \]

WaitIntrd2: \[ P, t \vdash \langle a.\text{wait}([], h) \rangle \to^{\text{nc}} \langle \text{WokenUp}, \text{ClearIntr} t, \text{TIntrd} t \rangle \to_{\text{nc}} \langle \text{Ret-sys-xcpt InterruptedException}, h \rangle \]

Figure 3.19: Semantics of the native methods wait, notify, and notifyAll

Otherwise, the thread is in the top state in the centre, i.e., \( t \) holds the lock on \( a \), it has not temporarily released any locks, its wait set status is None and it is not interrupted. Then, \( \text{WAIT} \) suspends \( t \) to \( a \)'s wait set and temporarily releases all locks on \( a \). Whether \( t \) will be interrupted or notified determines whether \( \text{wait} \) should return normally or raise an InterruptedException. As this is still indeterminate at this point of time, I extend the return values for native method calls with the special token Ret-Unchanged, which \( \text{WAIT} \) returns. The semantics of Ret-Unchanged
3.2. Multithreading in JinjaThreads

Legend:
- reduction of current thread: \( \rightarrow \)
- reduction of other threads: \( \longrightarrow \)

lock state of \( a \) w.r.t. \( t \):
- \( \emptyset \) \( t \) does not hold \( a \)
- \( \odot \) \( t \) holds \( a \)

set of \( t \)'s temporarily released locks

\( t \)'s wait set status:
- None
- \[ InWS \ a \]
- [WS-Notified]
- [WS-WokenUp]

\( t \)'s interrupt status:
- \( \lambda \) \( t \) is interrupted
- \( \chi \) \( t \) is not interrupted
- \( \lambda, \chi \) either of both

Figure 3.20: Steps for \( t \) executing a call to \( \text{wait} \) on object \( a \)
is that the thread’s next step should be to call the same method with the same parameters once more. This allows to effectively split the call to \textit{wait} into two steps. Since \texttt{Wait} suspends \texttt{t} to the wait set \texttt{a}, the interleaving semantics expects \texttt{t} to process its removal from the wait set in its next reduction, which must be tagged with the BTA \texttt{Notified} and \texttt{WokenUp}, respectively.

However, the interleaving semantics only picks one of \texttt{t}’s reductions after it has been removed from the wait set and reacquired the locks on \texttt{a}. Hence, when another thread calls \texttt{notify} or \texttt{notifyAll} on \texttt{a}, or \texttt{interrupt} on \texttt{t}, \texttt{t} is removed from the wait set \texttt{a} (cf. Figure 3.3). Note that this determines the result of the call to \texttt{wait}. Now, \texttt{t}’s next step is to reacquire the locks on \texttt{a}. There is no need to add a rule for that to \texttt{t}’s semantics because \texttt{ACQUIRE} of the interleaving semantics takes care of this.

Next, the second call to \texttt{wait} must process \texttt{t}’s removal from the wait set, which \texttt{WaitNtfd} and \texttt{WaitIntrd2} implement. The latter also clears \texttt{t}’s interrupt status as required by the JLS [56, §17.8.1]. Note that another thread may interrupt \texttt{t} after \texttt{t} is has been notified, but before \texttt{WaitNtfd} processes \texttt{t}’s removal. In that case, the call to \texttt{wait} returns normally and \texttt{t}’s interrupt status is still set.

**Interaction of interruption and notification**  JinjaThreads follows the JLS on the interaction between interruption and notification. Suppose the thread \texttt{t} in a wait set \texttt{a} is notified and interrupted “simultaneously”. Then, it may either return normally with its interrupt status set, or raise an \texttt{InterruptedException}. In the second case, the notification must not be lost, i.e., all other threads in the wait set must return with an \texttt{InterruptedException} or one of them gets notified.

JinjaThreads meets this requirement as follows: Notifications use the BTA \texttt{Notify} or \texttt{NotifyAll}, interrupts use \texttt{WakeUp} (\texttt{Intr}). Hence, the interleaving semantics remembers how a thread has been removed in the wait set status [\texttt{WS-Notified}] and [\texttt{WS-WokenUp}], respectively (cf. Figures 3.3 and 3.20). This already determines whether \texttt{WaitNtfd} or \texttt{WaitIntrd2} will process the removal, i.e., whether \texttt{wait} returns normally or throws an \texttt{InterruptedException}. In particular, \texttt{WaitNtfd} and \texttt{WaitIntrd2} do not depend on the interrupt status at the time when they execute.

\footnote{If \texttt{a}’s wait set contains multiple threads, the interleaving semantics may pick a thread different from \texttt{t} to be removed upon the call to \texttt{notify}.}
Otherwise, if they did and the two wait set states [WS-Notified] and [WS-WokenUp] were merged, notifications could be lost. Suppose, for example, two threads $t_1$ and $t_2$ are in a wait set $w$ and another thread calls `notify` on $a$, which removes $t_1$. While $t_1$ waits to reacquire the locks on $a$, another thread interrupts $t_1$ (the InTR transitions from left to centre in Figure 3.20). In the above scenario, these states would be merged with the ones on the right. Hence, when $t_1$ processes its removal, its interrupt status has been set, i.e., it raises an `InterruptedException`. But now, the notification is lost, because $t_2$ remains in the wait set. This violates the above requirement.

The JLS also requires that any implementation must determine an order over the concurrent notification and interruption – which does not have to be consistent with other orderings – and behave accordingly [56, §17.8.1]. In JinjaThreads, the interleaving of threads automatically defines an order on notifications (BTA `Notify` and `NotifyAll`) and interrupts (BTA `WakeUp`) from different threads which is consistent with all other orderings. If a single thread could issue notifications and interrupts in one step, these would be ordered by their position in the thread action. For example, $(\langle \text{Notify } a, \text{WakeUp } t \rangle)$ orders the notification before the interrupt, and in $(\langle \text{WakeUp } t, \text{Notify } a \rangle)$, the interrupt precedes the notification.

### 3.2.2 Source code

In the previous section, I have implemented native methods that deal with most of Java concurrency. Now, I define the single-threaded semantics for concurrency and extend the source code language $J$ with `synchronized` statements. With this semantics, I instantiate the interleaving semantics from §3.1.3 to obtain $J$’s interleaving semantics.

**Adaptations to sequential JinjaThreads**

Similarly to the semantics for native methods in the previous section, the semantics $P, t \vdash \langle e, (h, xs) \rangle \xrightarrow{ta} \langle e', (h', xs') \rangle$ of single threads for $J$ adapts the small step semantics $P \vdash \langle e, (h, xs) \rangle \rightarrow \langle e', (h', xs') \rangle$. First, it takes the thread ID $t$ as a parameter. Second, every reduction is labelled by a thread action. None of the existing rules – except for $R_{\text{native}}$ for native methods – issues BTAs of their own. Instead, rules for subexpression reduction propagate the label, and all others carry the empty thread
action (\(\emptyset\)).\(^{16}\) For example, \(\text{ROBJ}\) and \(\text{RCALLN}\) now are as follows; the additions are highlighted in grey.

\[
\text{ROBJ:} \quad P, t \vdash \langle e, s \rangle -ta \rightarrow \langle e', s' \rangle \\
\]

\[
P, t \vdash \langle e.\text{M}(es), s \rangle -ta \rightarrow \langle e'.\text{M}(es), s' \rangle
\]

\[
\text{RCALLN:} \quad P, t \vdash \langle \text{null}.\text{M}(es), s \rangle -\emptyset \rightarrow \langle \text{THROW NullPointer}, s \rangle
\]

Beyond the above changes, I have to adapt the rule \(\text{RNATIVE}\) for native method calls to the changes in §3.2.1 as follows:

\[
\text{RNATIVE:} \quad P, t \vdash \langle \text{addr a.M(map Val vs)}, (h, xs) \rangle -ta' \rightarrow \langle e', (h', xs) \rangle
\]

Remember that \(\text{native-Ret2J}\) converts the returned value or exception into \(J\) syntax. To deal with the new return token \(\text{Ret-Unchanged}\), it now takes another parameter of type \(\text{expr}\) that it returns for \(\text{Ret-Unchanged}\) and ignores otherwise. Moreover, the local state of newly spawned threads in the thread action \(ta\) of the native call must be adapted to the state representation in \(J\). To that end, \(\text{native-TA2J} P ta\) applies the conversion function \(\text{native-BTA2J} P\) to all local states \((C, M, a)\) of all \(\text{Spawn} \; \text{BTAs}\) in \(ta\). \(\text{native-TA2J} P (C, M, a)\) looks up the parameter-less method \(M\) for class \(C\) and wraps its body in a block for the \(\text{this}\) pointer initialised to \(a\), similarly to the rule \(\text{RCALL}\) for method calls. It leaves the local store undefined for all variables.

\[
\text{native-BTA2J} P (C, M, a) = \\
\text{let } (D, _, _, m) = \text{method} P C M; \ [(_{\phantom{m}}, \text{body})] = m \\
\text{in } ((\{\text{this}: \text{Class} D = [\text{Addr} a]; \text{body}, \text{empty}))
\]

The synchronized statement

Java has only a single statement for concurrency, namely \text{synchronized}. Everything else is provided as native methods, see §3.2.1. Hence, I

---

\(^{16}\)Except for rules that access the heap, i.e., object and array allocation, field and array cell access. These rules record their accesses in the sixth component using the heap events from Table 3.1. As these only become relevant in Chapter 4, I discuss them there.
3.2. Multithreading in JinjaThreads

<table>
<thead>
<tr>
<th>expression</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>sync (e_1) e_2</code></td>
<td>lock monitor <code>e_1</code> for executing block <code>e_2</code></td>
</tr>
<tr>
<td><code>insync (a) e</code></td>
<td>execute block <code>e</code> while having locked monitor <code>a</code></td>
</tr>
</tbody>
</table>

Table 3.2: Expressions for synchronized statements

extend the abstract syntax of $J$ (see Table 2.1) with the expressions shown in Table 3.2.

`sync (e_1) e_2` models Java’s synchronized statements, as specified in [56, §14.19]. It acquires the monitor `e_1` first, then executes `e_2`, and finally releases the monitor. To remember that a synchronized statement has already acquired the monitor, I use the expression `insync (a) e` instead of `sync (addr a) e`, which is not part of the input language. JinjaThreads does not model synchronized methods explicitly, because they are syntactic sugar for ordinary methods with their whole body inside a synchronized statement.

There is only one typing rule $WT_{sync}$ for `sync (_, _)` statements.

\[
WT_{sync}: \quad \frac{\begin{align*}
& P, E \vdash e_1 :: T_1 \\
& \text{is-ref}\ T_1 \quad T_1 \neq NT \\
& P, E \vdash e_2 :: T
\end{align*}}{
P, E \vdash sync (e_1) e_2 :: T}
\]

The monitor expression $e_1$ must have a reference type, but not $NT$. The return type of `sync (e_1) e_2` is `e_2`’s type. Similar to `try _ catch (_, _) _`, `sync (e_1) e_2` generalises the synchronized statement to arbitrary expressions. This is necessary to treat synchronized methods – which may return a value – as syntactic sugar for `sync (this) _` around their body. As `insync (_, _)` must not occur in programs, there is no typing rule for it.

Figure 3.21 shows the new reduction rules for synchronized blocks. $R_{sync1}$ reduces the monitor subexpression. If the monitor subexpression becomes `null`, a `NullPointerException` exception is raised ($R_{syncN}$). If an exception is raised while reducing the monitor subexpression, $R_{syncX}$ propagates the same exception. If the monitor subexpression reduces to some monitor address $a$, the thread can only reduce further by acquiring the lock on $a$. In that case, $R_{lock}$ rewrites the `sync (addr a) e` expression to `insync (a) e` to remember that the lock has been granted. Then, $R_{sync2}$ executes the body. Once it has become a value or raised an
exception, \texttt{RUNLCK} and \texttt{RUNLCKX} unlock the monitor, and return the value or propagate the exception, respectively.

Note that it is not necessary to explicitly release (and later reacquire) the monitor \(a\) when the body \(e\) of \texttt{insync (a)} \(e\) calls \texttt{wait} on \(a\). The basic thread action \texttt{Release} in \texttt{Wait} and temporarily released locks in the interleaving semantics (cf. \texttt{ACQUIRE}) take care of this.

\section*{Interleaving semantics}

In §3.2.1, I have already specified how the type variables in the interleaving semantics get instantiated for JinjaThreads source code and bytecode. Now, I fill in the missing details for thread-local states and the parameters of the locales. For \(j\), the local state (type variable \(x\)) consists of an expression and the local store, i.e., \(\text{expr} \times \text{locals}\).

To instantiate the locale \texttt{multithreaded-base} (3.2), I need some glue to adjust the parameter grouping, which is the price for not changing the original Jinja small-step semantics to syntactically match the format...
of the framework for interleaving.\textsuperscript{17} \( J\text{-final} \) and \( J\text{-red} P \), defined below, instantiate the parameters \( \text{final} \) and \( r \). Since events are irrelevant for this chapter, I defer the definition of \( \text{acq-events} \) to Chapter 4.

\[
\begin{align*}
J\text{-final} &= (\lambda(e, xs). \text{final } e) \\
J\text{-red} P &= (\lambda t ((e, xs), h) ta ((e', xs'), h')).
\end{align*}
\]

By instantiating the locale, I also specialize the definitions in it. To distinguish them from the original definitions, I prefix them with \( J \), e.g., \( J\text{-redT} \) denotes \( \text{redT} \) with the above parameter instantiations. As \( J\text{-red} \) takes an additional parameter \( P \) for the program declaration, the specialised versions also take such a parameter if necessary, e.g., \( J\text{-redT} P \).

Analogous to §3.1.3, I write \( P \vdash s - t:ta \rightarrow s' \) for a single step in the interleaving semantics \( J\text{-redT} P \), and \( P \vdash s - ttas \rightarrow^* s' \) for the reflexive and transitive closure \( (J\text{-redT} P)^* \).

\textbf{Lemma 3.6.} \( J\text{-final} \) and \( J\text{-red} P \) are well-formed with respect to the interleaving semantics.

\textit{Proof.} I must show the assumptions of the locale \textit{multithreaded} (cf. §3.1.3). \( \text{final-no-red} \) follows by case analysis of the rules. \( \text{Spawn-heap} \) holds by induction on the small step semantics and case analysis on the semantics for native methods. \( \square \)

\textbf{Start state}

The start state \( J\text{-start} P C M \ vs \) for program \( P \) has exactly one thread \( \text{start-tID} \) with thread-local state

\[
(\text{blocks } (\text{this } \cdot \text{pns}) (\text{Class } D \cdot Ts) (\text{Null } \cdot \text{vs}) \text{body, empty})
\]

where \( (D, Ts, \_, [(\text{pns, body})]) = \text{method } P C M \). Hence, \( \text{start-tID} \) is about to execute the non-native method \( M \) in class \( C \) with parameters \( \text{vs} \).

\textsuperscript{17}Modelling the interleaving semantics after the single-threaded semantics would have been no better alternative. Then, the interleaving semantics would be cluttered with glue code for splitting and composing pairs, because the thread local state would consist of two parts that I would have to store together to avoid additional invariants.
Setting the this pointer to Null simulates a static method.\textsuperscript{18} The initial heap start-heap has preallocated a Thread object for start-tID and objects for all system exceptions. There are no locks held or temporarily released, all wait sets are empty and there are no interrupts. Formally:

\[
J\text{-}\text{start } P C M \text{ vs } = \\
(\text{let } (D, Ts, \_, m) = \text{method } P C M; \lfloor (\text{pns, body}) \rfloor = m \\
e = \text{blocks (this \cdot pns) (Class } D \cdot Ts) (\text{Null \cdot vs) body} \\
in (K^f \text{None, } ([\text{start-tID}\mapsto ((e, empty), K^f 0)], \text{start-heap, empty, } 0))
\]

A start state as specified by \(P, C, M,\) and vs is well-formed (written \(\text{wf-start } P C M \text{ vs}\)) iff \(C\) sees a non-native method \(M\) and the parameters vs conform to \(M\)'s parameter types in the start-heap.

### 3.2.3 Bytecode

In this section, I extend the single-threaded semantics for JinjaThreads bytecode to concurrency. As I use the interleaving semantics from §3.1 for bytecode, too, this section follows the same line as the previous for source code.

#### Adaptations to the sequential JinjaThreads VM

As the multithreading features are mostly hidden in native methods (§3.2.1), which source code and bytecode implement similarly, I adapt the JinjaThreads VM analogously to what I did for \(J\):

1. The semantics functions \(\text{exec-native, exec-instr,}\) and \(\text{exec}\) now take the thread ID as parameter and return a set of pairs of thread action and successor state rather than just a set of successor states.

2. \(\text{native-Ret2jvm}\) also handles the return token \(\text{Ret-Unchanged}\).

\textsuperscript{18}Calling Thread’s static native methods like \textit{yield} from the start method \(M\) becomes more complicated, because \texttt{Var this.yield([])} (as §3.2.1 suggests) raises a \texttt{NullPointerException} exception. Instead, they must be invoked on some other object. In practice, this is negligible since it only applies to the start method and the converter from Java to JinjaThreads (§6.5) wraps a method for bootstrapping around the actual main method. The same applies to the initial state for bytecode (§3.2.3).
3.2. Multithreading in JinjaThreads

exec-instr \( (\text{Invoke } M' \ n) \ P t h \ stk \ loc \ C \ M \ pc \ f r s = \)
(\text{let } ps = \text{rev } (\text{take } n \ stk); \ r = stk[n]; \ Addr \ a = r; \ [hT] = \text{typeof-addr } h \ a \in \text{if } r = \text{Null then} \)
\[
\{ (\emptyset, [\text{addr-of-sys-xcpt } \text{NullPointer}], h, (stk, loc, C, M, pc) \cdot f r s) \}
\]
else let \( (D, Ts, T_r, m) = \text{method } P (\text{class-of' } hT) \ M' \)
in case \( m \) of Native \( \Rightarrow \)
\[
\{ (\text{native-TA2jvm } P \ ta, \)
native-Ret2jvm \( n' \ stk \ loc \ C \ M \ pc \ f r s \ vx) | ta \ vx h'. \)
\( (ta, vx, h') \in \text{exec-native } P t a M' \ ps \ h \}
\]
\[\text{in } \{ (\emptyset, \text{None, } h, fr' \cdot (stk, loc, C, M, pc) \cdot f r s) \}\]

Figure 3.22: Adaptations to the single-step semantics for the VM

3. Analogous to \text{native-BTA2J}, the function \text{native-BTA2jvm} constructs the initial states of spawned threads. Then, the conversion function \text{native-TA2jvm} \( P \ ta \) applies \text{native-BTA2jvm} \( P \) to all spawned threads in \( ta \).

\text{native-BTA2jvm} \( P (C, M, a) = \)
let \( (D, _, _, m) = \text{method } P C M; \ [(mxs, mxl, ins, xt)] = m \)
in \( \text{None, } [(\emptyset, \text{Addr } a \cdot \text{replicate } mxl \text{ undefined-Val}, D, M, 0)]) \))

Figure 3.22 shows the updated definitions from Figures 2.17 and 2.18. I highlighted the additions in grey, everything else remains unchanged. In the same way, I adapt the defensive VM \text{execd} \( P t s \) and the relational views (notation \( P, t s \vdash s \rightarrow_{jvm} s' \) and \( P, t s \vdash s \rightarrow_{jvmd} s' \), respectively) to include the thread ID and the thread action.
Table 3.3: Instructions for monitors

<table>
<thead>
<tr>
<th>instruction</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEnter</td>
<td>acquire lock on monitor</td>
</tr>
<tr>
<td>MExit</td>
<td>release lock on monitor</td>
</tr>
</tbody>
</table>

exec-instr MEnter P t h stk loc C M frs =
(let v · stk′ = stk; Addr a = v
  in if v = Null then
    { ⟨⟩, [addr-of-sys-xcpt NullPointer], h, (stk, loc, C, M, pc) · frs } |
  else { ⟨⟨Lock→a, SLock a⟩⟩, None, h, (stk′, loc, C, M, pc + 1) · frs ) |

exec-instr MExit P t h stk loc C M frs =
(let v · stk′ = stk; Addr a = v
  in if v = Null then
    { ⟨⟩, [addr-of-sys-xcpt NullPointer], h, (stk, loc, C, M, pc) · frs ) |
  else
    { ⟨⟨Unlock→a, SUnlock a⟩⟩, None, h, (stk′, loc, C, M, pc + 1) · frs ) |
        ⟨⟨UnlockFail→a⟩⟩, [addr-of-sys-xcpt IllegalMonitorState] , |
                     h, (stk, loc, C, M, pc) · frs ) |

Figure 3.23: Semantics of the instructions for monitors

Instructions for monitors

Table 3.3 lists the new instructions MEnter and MExit for monitors. They are the bytecode equivalent of sync (_ _) blocks; they lock and unlock, respectively, the monitor whose address is at the top of the stack. Figure 3.23 defines their semantics.

If the value v on top of the stack is Null, they both raise a NullPointer exception. Otherwise, MEnter acquires the lock with the thread action ⟨Lock→a, SLock a⟩ where a is the address that v denotes. Note that the source code semantics uses the same thread action (RLock). MExit unlocks the monitor a with the thread action ⟨Unlock→a, SUnlock a⟩ just like RUnlock. Unlike sync (_ _) _, locking and unlocking need not be structured in bytecode, i.e., MEnter and MExit need not come in pairs. Hence, MExit may also fail with an IllegalMonitorState exception if the current thread does not hold the monitor.
3.2. Multithreading in JinjaThreads

For the well-typings and the bytecode verifier, Figure 3.24 defines the abstract interpretation of MEnter and MExit on the level of types (see §2.2.3). For both instructions, applicability requires that the stack is non-empty and the top element is a reference type, and the effect is dropping the top element.

**Interleaving semantics**

Bytecode instantiates the interleaving semantics twice, once for the aggressive VM and once for the defensive. The type variables are the same as for J – except for the thread-local states. For both the aggressive and defensive VM, a thread-local state consists of the exception flag and the call stack. Such a thread-local state is jvm-final iff the call stack is empty. Then, jvm-exec P and jvm-execd P instantiate parameter r of the locale multithreaded-base for the aggressive and defensive VM, respectively. Like for J-red, they require some glue to adjust the parameter grouping.

\[
\text{jvm-final} = (\lambda (xcp, frs). \text{frs} = [])
\]

\[
\text{jvm-exec} P = (\lambda t ((xcp, frs), h) \text{ta} ((xcp', frs'), h')). \\
P, t \vdash (xcp, h, frs) \rightarrow_{\text{jvm}} (xcp', h', frs')
\]

\[
\text{jvm-execd} P = (\lambda t ((xcp, frs), h) \text{ta} ((xcp', frs'), h')). \\
P, t \vdash \text{Normal} (xcp, h, frs) \rightarrow_{\text{jvmd}} \text{Normal} (xcp', h', frs')
\]

Note that jvm-execd turns the defensive VM into a strict VM as it no longer raises TypeError, but just gets stuck. Nevertheless, I keep referring to it as the defensive VM in the following. The reason for dropping TypeError is that I can avoid duplications by using the same representation for thread-local states for both the aggressive and defensive VM.
For example, I can use the same state invariants for both VMs. In §3.3.5, I discuss how this simplification affects the type safety theorem.

For the definitions in the locale multithreaded-base, I use the prefixes jvm. and jvmd. to mark the versions of the aggressive and defensive VM, respectively. I write \( P \vdash s \rightarrow_{jvm} s' \) for \( jvm.redT P s (t, ta) s' \), \( P \vdash s \rightarrow_{jvmd} s' \) for \( jvmd.redT P s (t, ta) s' \), and \( P \vdash s \rightarrow^{*}_{jvm} s' \) (\( P \vdash s \rightarrow^{*}_{jvmd} s' \)) for the reflexive and transitive closure of \( jvm.redT P \) (jvmd.redT P).

**Lemma 3.7.** jvm-final with either jvm-exec \( P \) or jvm-execd \( P \) are well-formed with respect to the interleaving semantics.

*Proof.* I show the assumptions of the locale multithreaded for the parameter instantiations jvm-final, jvm-exec \( P \) and jvm-final, jvm-execd \( P \). final-no-red follows by unfolding the definitions and exec \( P t (xcp, h, []) \) being the empty set (see Figure 3.22). Spawn-heap holds by case analysis on emptiness of the call stack, the exception flag, the current instruction and the native method that is being called. \( \square \)

**Start state**

The initial state jvm-start \( P C M vs \) of the JVM is the same as for J, except for the thread-local state of start-tID, which is

\[
(\text{None}, [([], \text{Null} \cdot vs @ \text{replicate} mxl \text{ undefined-Val}, D, M, 0)])
\]

where \( (D, \_\_\_\_, [(mxs, mxl, ins, xt)]) = \text{method } P C M \), i.e., no exception is flagged and the VM is about to execute the first instruction of method \( M \) in \( D \). Like in J-start \( P C M vs \), no lock is held, all wait sets are empty, there are no interrupts, and start-heap has preallocated objects for start-tID and the system exceptions.

### 3.3 Deadlock and type safety

For sequential languages, type safety is typically expressed as the syntactic properties progress and preservation (see §2.1.6). Progress means that every well-formed and well-typed expression can be reduced unless it has already been fully evaluated, i.e., the semantics is not missing any reduction rules. Full evaluation is determined by a syntactic predicate.
3.3. Deadlock and type safety

final. Preservation requires that reductions preserve well-typedness and well-formedness. In §3.1.4, I have shown how to transfer preservation proofs from single threads to the interleaving semantics, i.e., they do not pose a problem. Deadlocks, however, can break the progress property.

Therefore, I formally define the concept of deadlock (§3.3.1, §3.3.2) and prove progress up to deadlock (§3.3.3) for the interleaving semantics. For the latter, I identify sufficient conditions on the single-threaded semantics. Hence, source code and bytecode can reuse the progress theorem by discharging these conditions (§3.3.4, §3.3.5).

Although progress typically identifies allowed stuck states syntactically, I formalise deadlock (and thus the allowed stuck states) semantically for two reasons. First, I want the formalisation to be shared between source code and bytecode, but any syntactic characterisation necessarily depends on the thread-local state representation. Second, deadlock typically involves several threads, i.e., a syntactic characterisation would have to examine all of them together. Hence, not only is there no easy syntactic characterisation, but any such would also break the abstraction of the interleaving semantics. Instead, I define deadlocks by looking at the possible thread actions of all threads. In principle, one could derive syntactic conditions from this definition by analysing the single-thread semantics, but I have not done so.

3.3.1 Deadlock as a state property

A system is said to be in deadlock iff all threads are waiting for something that will never occur. In operating systems, deadlock of processes has four preconditions on resource usage: mutual exclusion, hold and wait, circular waiting, and no preemption (see, e.g., [134]). In the interleaving semantics, there are four “resources” that a thread can wait for: acquiring a lock, termination of another thread, being removed from a wait set, and interruption. Since all of them are implemented as BTAs, I can formally define deadlock solely in terms of the reductions of a thread – independent of the concrete language.

Consider, for example, the schedules in Figure 3.25. On the left-hand side, in Figure 3.25a, thread $t_1$ acquires the lock $l_1$, then thread $t_2$ acquires the lock $l_2$. To continue, $t_2$ needs the lock $l_1$, too, so the $\text{Lock} \rightarrow l_1$ is postponed. However, $t_1$ requests the lock $l_2$, which $t_2$ is holding. Hence, both threads are in deadlock. In the centre (Figure 3.25b), $t_3$ waits for $t_4$’s termination, but $t_4$ suspends itself to the wait set $w$. Hence, both $t_3$
and $t_4$ are in deadlock, because there is no thread left to notify $t_4$. The right-hand side (Figure 3.25c) shows a similar example with interruption. $t_6$ waits for being interrupted, but $t_5$ terminates without doing so. Thus, $t_6$ is deadlocked.

In the interleaving semantics, things are a bit more tricky than in the above examples, because threads can atomically request any number of locks and join on other threads. Moreover, they can wait for different events non-deterministically. In Java, for example, the Thread. join() method either waits for the receiver thread to terminate or for the executing thread to be interrupted.

To get a hold on this, I first define for what a thread may wait in deadlock. The function $\text{waits} \; ta : (l + t + t) \; \text{set}$ extracts all the resources for which the executing thread may be waiting, i.e., all the locks it acquires, the threads it joins on, and the threads which must be interrupted. Formally:

$$\text{waits} \; ta =$$

$$\{ \ l. \ \text{Lock} \in \text{set} \; (\langle ta \rangle_L) \} \cup \{ \ t. \ \text{Join} \in \text{set} \; (\langle ta \rangle_c) \} \cup \text{intr-waits} \; (\langle ta \rangle_i)$$

where $\cup$ is disjoint union and $\text{intr-waits} \; (\langle ta \rangle_i)$ is the set of thread IDs $t$ for which $\langle ta \rangle_i$ contains an element $\text{IsIntrd} \; t \; \text{True}$ which is not preceeded by $\text{Intr} \; t$ or $\text{ClearIntr} \; t$. The last constraint removes interrupt checks whose result does not depend on the initial interrupt state, but is determined by the preceeding interrupt actions. Consider, for example,

$$\text{intr-waits} \; (\text{IsIntrd} \; t \; \text{True}, \text{ClearIntr} \; t) = \{ t \} \quad (3.5)$$

$$\text{intr-waits} \; (\text{Intr} \; t, \text{IsIntrd} \; t \; \text{True}) = \emptyset \quad (3.6)$$

The thread action in (3.5) tests whether $t$ has been interrupted and, if so, clears the interrupt status. Hence, it waits for $t$ being interrupted.
3.3. Deadlock and type safety

**MWlock:**
\[
\text{has-lock } ((\text{locks } s) t l) t' t' \neq t t' \in T \\
\text{must-wait } s t (\text{Inl } l) T
\]

**MWjoin:**
\[
\text{not-final-thread } s t' t' \in T \\
\text{must-wait } s t (\text{Inr } (\text{Inl } t')) T
\]

**MWintr:**
\[
\text{all-final-except } s T t' \notin \text{intrs } s \\
\text{must-wait } s t (\text{Inr } (\text{Inr } t')) T
\]

Figure 3.26: Thread \( t \) must wait for resource \( w \) indefinitely

In (3.6), however, the test \( \text{IsIntrd } t \text{ True} \) holds vacuously because \( \text{Intr } t \) sets the interrupt flag right before the test.

Note that I do not care about actions for unlocking, thread creation and thread existence, wait sets and not being interrupted for the following reasons:

**unlocking** Only a thread itself would be able to remedy the missing lock, not others.

**thread creation** In Java, spawning a thread either succeeds or raises an exception, i.e., it cannot deadlock.

**wait sets** A thread in a wait set cannot do anything to be removed. \( \text{ok-wsets} \) only distinguishes normal execution from processing the removal from a wait set. Waiting threads will be dealt with specifically.

**non-interruption** For the interleaving semantics, non-interruption is dual to interruption, i.e., I could treat both uniformly, but in Java, threads can only wait for being interrupted.

The predicate \( \text{must-wait } s t w T \) determines that in state \( s \), the thread \( t \) will wait indefinitely for resource \( w :: \ell ' + t' + t \) under the assumption that all threads in \( T \) are already deadlocked. Figure 3.26 shows the formal definition. The thread \( t \) must wait forever for the lock \( l \) (case \( w = \text{Inl } l \)), if \( l \) is held by another thread \( t' \) which is deadlocked (MWlock). The join on another thread \( t' \) fails forever (case \( w = \text{Inr } (\text{Inl } t') \)), if the thread \( t' \) is not final and already deadlocked (MWjoin). The predicate \( \text{not-final-thread } s t \) denotes that \( t \) exists, but is not final, i.e., \( t \in \text{dom } (\text{thr}s) \)
Chapter 3. Interleaving semantics

deadlock \ s = \\
(\forall x. \ thr \ s t = [ (x, K^f 0) ] \land \neg final \ x \land wset \ s t = None \\
\rightarrow t \vdash (x, shr \ s) \leftarrow \\
\forall W. t \vdash (x, shr \ s) W \leftarrow (\exists w \in W. \ must-wait \ s t \ w (dom (\ thr \ s)) \land \\
(\forall x ln. \ thr \ s t = [ (x, ln) ] \land ln \neq K^f 0 \land \neg waiting (wset \ s t) \\
\rightarrow \exists ln. ln I_1 > 0 \land must-wait \ s t (Inl I_1 \ (dom (\ thr \ s))) \land \\
(\forall t. \ thr \ s t = [ (x, K^f 0) ] \\
\rightarrow wset \ s t \neq [WS-Notified] \land wset \ s t \neq [WS-WokenUp])

Figure 3.27: Formal definition of deadlock

and \ t \notin final-threads \ s. Note that not-final-thread \ s t negates the precondition of Join (see Figure 3.11) except for \ t \neq t', which is irrelevant here. A thread waits indefinitely for \ t' \ being interrupted (case \ w = Inr (Inr t'), MWintr), if \ t' \ is not interrupted, but all non-deadlocked threads are final – expressed by the predicate all-final-except.

all-final-except \ s T = \{ t. not-final-thread \ s t \} \subseteq T

With these preparations, I can now formally define deadlock, see Figure 3.27. First, I introduce two abbreviations: \ t \vdash (x, h) \leftarrow denotes that \ t \ can reduce in the local state \ x \ and heap \ h \ with some thread action \ ta \ such that \ ta \ is not contradictory in itself, i.e., there is a multithreaded state \ s \ such that ok-ta \ s t ta. For example, \ (\{Lock \rightarrow l, UnlockFail \rightarrow l\}) \ and \ (\{Intr \ t, IsIntrd \ t False\}) \ are contradictory. The predicate \ t \vdash (x, h) W \leftarrow denotes that \ t \ can reduce in state \ (x, h) \ with a thread action \ ta \ such that \ W = waits ta. It abstracts \ t's reductions to the resources \ W \ it waits for.

Then, a multithreaded state \ s \ is in deadlock (written deadlock \ s) iff

(i) every non-final thread \ t \ that is ready to execute, say \ thr \ s t = [ (x, K^f 0) ] \ and wset \ s t = None, can reduce, and no matter how it might reduce, it must wait indefinitely, and

(ii) every thread with temporarily released locks that is not in a wait set must wait indefinitely on one of these, and

(iii) for every thread which has not temporarily released any locks, its wait set status is neither \ [WS-Notified] \ nor \ [WS-WokenUp].

The first condition is the default case: \ t \vdash (x, shr \ s) \leftarrow ensures that the thread is not just stuck, i.e., universal quantification on \ W \ does not
hold vacuously. For must-wait, I consider all threads as deadlocked (i.e., \( T = \text{dom} (\text{thr}s) \)), because the system as a whole is supposed to be in deadlock. Note that quantifying over all \( W \) with \( t \vdash (x, \text{shr}s) \ W \) allows a thread to non-deterministically wait for different “resources”. The second condition accounts for acquisition of temporarily released locks. Since it is the interleaving semantics that performs the acquisition (\textsc{Acquire}), but not the single threads, I need an extra case for that. The last condition ensures that there is no thread which has been removed from a wait set and has already reacquired the released locks, but has not yet processed the removal internally. Note that there is no condition for threads in wait sets, because they are automatically deadlocked when there is no thread to remove them.

For simplicity, I do not require that there is at least a thread which is not final. Hence, every \textit{mfinal} state is also in \textit{deadlock}.

**Lemma 3.8.** States in deadlock are stuck, i.e., if \( s \rightarrow_{t:ta} s' \), then \( \neg \text{deadlock} s \).

### 3.3.2 Deadlock for threads

The previous section defines when all threads are in deadlock. However, some threads may already be in deadlock while others keep running. In this section, I define deadlock for single threads and prove that this notion generalises deadlock as a state property.

Figure 3.28 defines the set \textit{deadlocked} of threads in deadlock coinductively. Note that coinductivity naturally captures that a thread is not deadlocked iff one can deduce in finitely many steps that it is not. Hence, finitely many steps (of other threads) suffice to allow the thread under consideration to continue. This definition considers the set of threads to be closed with respect to the outside, i.e., one cannot add a spinning thread to the system, which would “undeadlock” again all waiting threads.

Rules \textsc{Dactive} and \textsc{Dacquire} are very similar to conditions (i) and (ii) of deadlock, respectively. The only difference is the set of threads that must-wait takes: Instead of all threads, it now consists of deadlocked and final ones. The last rule \textsc{Dwait} expresses that a waiting thread is deadlocked if all other threads are either final or \textit{deadlocked} themselves (which includes waiting).\(^{19}\)

---

\(^{19}\)Remember that I do not model spurious wake-ups from wait sets. Otherwise, a waiting thread would never be deadlocked, because it could always be woken up spuriously.
Chapter 3. Interleaving semantics

$$\text{thr } st = \lfloor (x, Kf) \rfloor$$
$$\text{wset } st = \text{None}$$
$$\text{t} \vdash (x, \text{shr } s) \triangleright$$

**Dactive:**

$$\forall W. \text{t} \vdash (x, \text{shr } s) W \triangleright \exists w \in W. \text{must-wait } st w (\text{deadlocked } s \cup \text{final-threads } s)$$

$$t \in \text{deadlocked } s$$

**Dacquire:**

$$\text{thr } st = \lfloor (x, \ln) \rfloor$$

$$\neg \text{waiting } (\text{wset } st) \ln_f l > 0$$

$$\text{must-wait } st (\text{Inl } l) (\text{deadlocked } s \cup \text{final-threads } s)$$

$$t \in \text{deadlocked } s$$

**Dwait:**

$$\text{waiting } (\text{wset } st) \text{ all-final-except } s (\text{deadlocked } s)$$

$$t \in \text{deadlocked } s$$

**Figure 3.28:** Coinductive definition of the set of threads in deadlock

Rule **Dwait** completely differs from condition (iii): The latter disallows the wait set status $[\text{WS-Notified}]$ and $[\text{WS-WokenUp}]$, which is implicit in **deadlocked** because there is no introduction rule for these cases. Recall that **deadlock** imposes no constraints on waiting threads, since they are implicitly included there.

Remember from §1.4.3 that (co)inductives predicates are defined as the fixed point of the associated functional and that this functional must be monotone. For **deadlocked**, I therefore must show that **must-wait** and **all-final-except** are monotone, because **deadlocked** is passed as a parameter to them.

**Lemma 3.9** (Monotonicity of all-final-except and must-wait). Let $T \subseteq T'$. If all-final-except $s T$, then all-final-except $s T'$. If must-wait $s t w T$, then must-wait $s t w T'$.

Consider Figure 3.29 for an example of different deadlock situations. Suppose there are six threads which at the moment can reduce with the thread actions shown on the left-hand side. If there are multiple thread actions, then there is one reduction for each. Suppose further that no thread is waiting and that the $i$-th thread holds the lock $l_i$. The graph on the right-hand side shows which thread is waiting to obtain a lock held by another thread. Then, threads III and VI are waiting for each other without other reduction options. Clearly, both of them are deadlocked. Although I and II are also waiting for each other, they are not deadlocked.
at the moment: II has two more reduction options. Waiting on lock $l_3$ will be in vain, because III is deadlocked. However, IV is not waiting for any resource, hence II may still hope to obtain the lock $l_4$ some time. Hence, I is not in deadlock either, as II might release $l_2$ afterwards. Since thread actions must be executed atomically, we may not interleave the thread actions of I and II, i.e., first unlock both $l_1$ and $l_2$ and then lock $l_2$ and $l_1$ again. Note that V is waiting simultaneously for II and III, because $V \vdash \{ Inl l_2, Inl l_3 \} \not$. Since III is already in deadlock, so is V. Clearly, IV is not deadlocked. If its wait set status is $\text{None}$, the empty thread action $\text{LM}$ is possible. Otherwise, it is $\text{⌊WS-Notified⌋}$ or $\text{⌊WS-WokenUp⌋}$, but then IV is not deadlocked by definition. Now, suppose thread IV is in a wait set. Then, all threads are deadlocked, because every thread except IV is waiting for some other thread to release a lock, and the only thread that could be reduced (i.e., IV) is waiting for some other thread waking it up.

Next, I show that $\text{deadlocked}$ leads to the same characterisation of states in deadlock as $\text{deadlock}$. Let $\text{deadlock'}$ denote that all non-final threads are deadlocked, i.e.,

$$\text{deadlock'} = (\forall t. \text{not-final-thread $s t \rightarrow t \in \text{deadlocked s}})$$

**Theorem 3.1.** $\text{deadlock} \iff \text{deadlock'}$.

**Proof.** From left to right: Suppose $\text{deadlock}$ and $\text{not-final-thread $s t$}$. Then, $t \in \text{deadlocked s}$ by coinduction with the set of $\text{not-final-threads}$ as the coinduction invariant. In the coinduction step, I must show that any $t'$ with $\text{not-final-thread $s t'$}$ satisfies the assumptions of one of $\text{deadlocked'}$’s introduction rules – with $\text{deadlocked s}$ replaced by $\{ t. \text{not-final-thread $s t$} \} \cup \text{deadlocked s}$. $\text{Dactive}$ and $\text{Dacquire}$ follow from the assumption $\text{deadlock s}$, because they differ from their counterparts in the definition of $\text{deadlock}$ only in the sets of thread IDs passed to
must-wait. But they are equal in that case for the coinduction invariant, because

\[ \{ t. \text{not-final-thread} s t \} \cup \text{deadlocked} s \cup \text{final-threads} s = \text{dom} (\text{thr} s) \]

As all-final-except s (\{ t. \text{not-final-thread} s t \} \cup \text{deadlocked} s) holds vacuously, the case for Dwait is trivial.

For the other direction, suppose \( t' \in \text{deadlocked} s \) for all \( t' \) such that \text{not-final-thread} s t’. All three conditions of deadlock trivially hold for final threads t, so suppose \text{not-final-thread} s t. Hence, \( t \in \text{deadlocked} s \).

From this, conditions (i) and (ii) in the definition of deadlock directly follow by monotonicity of must-wait (Lemma 3.9). The last condition (iii) holds because deadlocked’s definition excludes threads with the forbidden wait set status. □

Given the thread-wise predicate about deadlock, I strengthen Lemma 3.8 to the following lemma, which is proven by case analysis on \( s \xrightarrow{t:ta} s' \) and \( t \in \text{deadlocked} s \). Lemma 3.8 then directly follows with Theorem 3.1.

**Lemma 3.10.** Every thread in deadlock is stuck, i.e., if \( s \xrightarrow{t:ta} s' \), then \( t \notin \text{deadlocked} s \).

Since I am considering now deadlocks of individual threads, execution may continue even if some threads are deadlocked. Still, threads in deadlock should remain deadlocked; otherwise, the deadlock definition would be flawed. However, deadlock preservation requires that the single-threaded semantics is well-behaved:

1. The changes of the shared heap by the executing threads must not deprive a deadlocked thread of all of its reduction options. Otherwise, it would be stuck and therefore no longer deadlocked, since deadlock explicitly excludes stuck threads.

2. Such changes must not enable new reduction options which would undeadock it, either.

Locale preserve-deadlocked collects these assumptions (Figure 3.30). It fixes a set \text{wf-states} of well-formed states only for which the preservation requirements have to hold. The first assumption expresses that the interleaving semantics redT preserves \text{wf-states}. The other two
locale preserve-deadlocked = multithreaded +
fixes wf-states :: ('l, 't, 'x, 'h, 'w) state set
assumes invariant redT wf-states
and \[ s \in \text{wf-states}; \ s \to t':ta \to s' \; \text{thr} \ s \ t = \lfloor (x, K^f 0) \rfloor; \ t \vdash (x, shr s') \rfloor \]
\[ \Rightarrow t \vdash (x, shr s') \rfloor \]
and \[ s \in \text{wf-states}; \ s \to t':ta \to s' \; \text{thr} \ s \ t = \lfloor (x, K^f 0) \rfloor; \ t \vdash (x, shr s') W' \rfloor \]
\[ \Rightarrow \exists W \subseteq W'. \ t \vdash (x, shr s') W \rfloor \]

Figure 3.30: Locale preserve-deadlocked collects the requirements for preservation of deadlock

assumptions express exactly the requirements from above. Note the covariance in the set \( W \) in the last assumption: Since \( t \vdash _- W \rfloor \) expresses that one of \( t \)'s reduction requires all resources in \( W \), changes in the heap may only increase \( W \). Under these assumptions, the following holds:

Lemma 3.11 (Preservation of deadlock). Let \( s \in \text{wf-states} \). If \( s \to t:ta \to s' \) or \( s \to t\text{tas} \to s' \), then deadlocked \( s \subseteq \text{deadlocked} s' \).

Proof. If \( s \to t:ta \to s' \), the proof proceeds by case analysis on \( \text{redT} \). In either case Normal and Acquire, suppose \( t' \in \text{deadlocked} s \). I show \( t' \in \text{deadlocked} s' \) by coinduction with \text{deadlocked} s as coinduction invariant. In the coinductive step, case analysis on \( t' \in \text{deadlocked} s \) yields the interesting cases Dactive and Dacquire; the case Dwait contradicts the assumption \( s \to t:ta \to s' \). In both former cases, the reduction from \( s \) to \( s' \) preserves must-wait. For Dactive, the assumptions of preserve-deadlocked relate \( t' \vdash (x,_-) \rfloor \) and \( t' \vdash (x,_-) -_ \rfloor \) between shr \( s \) and shr \( s' \), respectively.

The case \( s \to t\text{tas} \to s' \) follows from \( s \to t:ta \to s' \) by induction; the locale's first assumption reestablishes the induction invariant \( s \in \text{wf-states} \) in the inductive step. \( \square \)

3.3.3 Progress up to deadlock

Recall that I have formalised deadlocks to obtain a progress result for the interleaving semantics. In this section, I prove the following theorem and present the assumptions about \( _- \vdash _- -_- \to _- \) on which it depends.

Theorem 3.2 (Progress up to deadlock). Let \( s \in \text{wf-states} \). If \( \neg \text{deadlock} s \), then there are \( t, ta, \) and \( s' \) such that \( s \to t:ta \to s' \).
locale progress = multithreaded +
fixes wf-states :: (l, t, x, h, w) state set
assumes invariant redT wf-states
and wf-stateD: s ∈ wf-states → ok-locks-thr s ∧ ok-wset-final s
and progress: [s ∈ wf-states; thr s t = (x, K^f 0)] → ¬ final x
⇒ ∃ta x’ h’. t ⊢ (x, shr s) → (x’, h’)
and wf-ta:
[s ∈ wf-states; thr s t = (x, K^f 0)]; ¬ waiting (wset s t);
t ⊢ (x, shr s) → (x’, h’)
⇒ ∃s’. ok-ta’ s’ t ta
and Suspend-not-final:
[s ∈ wf-states; thr s t = (x, K^f 0)]; ¬ waiting (wset s t);
t ⊢ (x, shr s) → (x’, h’); Suspend w ∈ set ⟨ta⟩_w
⇒ ¬ final x’
and Wakeup-waits:
[s ∈ wf-states; thr s t = (x, K^f 0)]; t ⊢ (x, shr s) → (x’, h’);
Notified ∈ set ⟨ta⟩_w ∨ WokenUp ∈ set ⟨ta⟩_w
⇒ waits ta = 0

Figure 3.31: Definition of locale progress

Theorem 3.2 requires certain assumptions about the single-threaded semantics, which I collect in the locale progress, see Figure 3.31. Like preserve-deadlocked, it fixes a set wf-states of well-formed states that is closed under reductions. wf-stateD ensures that well-formed states satisfy two invariants: Only existing threads hold the locks (ok-locks-thr s) and all threads in dom (wset s) exist in thr s with their local state not being final (ok-wset-final s). Together with the other constraints, the latter invariant ensures that threads which have been removed from a wait set are able to process the removal.

The assumption progress expresses the usual progress condition for single threads: Every thread in any well-formed state whose local state is not final can reduce.

The remaining assumptions restrict the single-thread semantics such that irreducible multithreaded states are final or in deadlock. wf-ta
ensures that the thread action of any reduction is not contradictory in itself.

Similar to progress expressing that no reduction rule is missing for well-typed terms, \( \text{wf-red} \) formalises that the reduction rules are able to generate all thread actions that are needed. \( \text{ok-ta}' \) formalises that \( \text{ta}' \)'s conditions are met except for BTAs which are allowed to cause deadlock. It is like \( \text{ok-ta} \) (3.1) with the following modifications:\(^{20}\)

- \( \text{ok-Ls} \) stops checking the lock preconditions when it encounters the first \( \text{Lock} \) BTA for \( l \) that \( t \) cannot acquire – though it does enforce the preconditions of \( \text{Unlock} \) and \( \text{UnlockFail} \) prior to this \( \text{Lock} \) BTA. For example, take \( \text{ta} = \langle \text{Lock} \rightarrow l, \text{Unlock} \rightarrow l, \text{Unlock} \rightarrow l \rangle \). Then, \( \text{ok-ta}' \) requires that \( t \) already holds the lock \( l \) once, or that it cannot acquire the lock \( l \).\(^{21}\)

- \( \text{ok-intr} \) ignores conditions of BTAs of the form \( \text{IsIntrd} \_ \_ \text{True} \).

- \( \text{ok-cond} \ s \ t \ (\text{Join } t') \) is always \( \text{True} \).

Thus, \( \text{wf-red} \) requires that every thread which is ready to execute, say with thread action \( \text{ta} \), can reduce with thread action \( \text{ta}' \) such that either the current state \( s \) already meets \( \text{ta}' \)'s preconditions, or \( s \) meets them except for BTAs that are allowed to deadlock, but in the latter case, it must not add anything it is waiting for in \( \text{ta}' \) compared to \( \text{ta} \).

Assumption \( \text{Suspend-not-final} \) demands that a thread be not final after it has suspended itself to a wait set, i.e., it can later process its removal.

The last one \( \text{Wakeup-waits} \) requires that while processing the removal from a wait set, the thread does not execute BTAs which may cause deadlock. Although the interleaving semantics could deal with such BTAs, I disallow them, because they would complicate the deadlock formalisation and proofs unnecessarily, and neither source code nor bytecode semantics uses this.

Under these assumptions, Theorem 3.2 holds.

\(^{20}\)Technically, to implement the modifications, I have defined predicates \( \text{ok-Ls}' \), \( \text{ok-intr}' \), and \( \text{ok-cond}' \) that \( \text{ok-ta}' \) uses. As these modifications are obvious, I omit their presentation.

\(^{21}\)\( \text{ok-Ls} \) stops checking at \( \text{Locks} \) instead of skipping them (like \( \text{ok-intr} \) and \( \text{ok-cond} \) do for \( \text{IsIntrd} \_ \_ \text{True} \) and \( \text{Join} \_ \)) because \( \text{Lock} \) changes the lock status, i.e., checking the conditions of subsequent lock BTAs would be meaningless.
Chapter 3. Interleaving semantics

Proof of Theorem 3.2. As \( s \) is not in deadlock, by definition (Figure 3.27), there must be a thread, say \( \text{thr s t} = [(x, ln)] \), such that

(a) \( t \) is not waiting, \( ln = K^f 0 \), not final \( x \), and either not \( t \vdash (x, shr s) \) or there is a set \( W \) such that \( t \vdash (x, shr s) W \) and \( t \) need not wait for any \( w \in W \), or
(b) \( t \) is not waiting, \( ln \neq K^f 0 \), and it need not wait for any of the locks in \( ln \), or
(c) \( ln = K^f 0 \) and \( t \)'s wait set status is \([WS-Notified]\) or \([WS-WokenUp]\).

I show for each case that \( t \) can take a step next.

In case (a), progress postulates a reduction \( t \vdash (x, shr s) \rightarrow t \rightarrow (\_., \_) \). \( wf-ta \) ensures that the thread actions of any reduction is not contradictory in itself, in particular \( ta \) is not. Hence, \( t \vdash (x, shr s) \) by definition. With (a), let \( W \) be such that \( t \vdash (x, shr s) W \) and \( t \) need not wait for any \( w \in W \). Again by definition, there is a reduction \( t \vdash (x, shr s) \rightarrow ta' \rightarrow (\_., \_) \) with \( W = \text{waits ta'} \). By \( wf-red \), there is another reduction \( t \vdash (x, shr s) \rightarrow ta'' \rightarrow (x', h') \) such that either (i) \( \text{ok-ta s t ta''} \), or (ii) \( \text{ok-ta' s t ta''} \) and \( \text{waits ta''} \subseteq \text{waits ta'} \). In case (i), I am done by \textsc{Normal}, because \( \text{upd-ta} \) is a right-total relation. In case (ii), by choice of \( ta' \), all \( w \in \text{waits ta''} \subseteq \text{waits ta'} \) meet their precondition. Hence, \( \text{ok-ta' s t ta''} \) implies \( \text{ok-ta s t ta''} \), and I am back at case (i).

In case (b), the reduction directly follows with \textsc{Acquire}.

In case (c), \( t \) must process its removal from a wait set. By \( wf-stateD \), \( \text{ok-wset-final s} \), in particular not final \( x \). By the same argument as in case (a), there is a reduction \( t \vdash (x, shr s) \rightarrow ta'' \rightarrow (x', h') \) such that \( \text{ok-ta s t ta''} \) or \( \text{ok-ta' s t ta''} \). If \( \text{ok-ta' s t ta''} \), \( \langle ta'' \rangle_w \) contains \text{Notified} or \text{WokenUp} due to \( t \)'s wait set status. By \textsc{Wakeup-waits}, \( \text{waits ta''} = \emptyset \), i.e., \( \text{ok-ta' s t ta''} \) and \( \text{ok-ta s t ta''} \) coincide. Thus, \textsc{Normal} yields the desired reduction. \( \square \)

3.3.4 Type safety for source code

With the above preparations in place, I now prove type safety for \( J.redTP \) (Theorem 3.3). Thereby, I reuse the type safety proof for the sequential subset from §2.1.6.

Theorem 3.3 (Type safety). Let \( wf-J-prog P \) and \( wf-start P C M vs. \). If \( P \vdash J-start P C M vs \rightarrow ta \rightarrow^* s \) such that \( \neg P \vdash s -t': ta' \rightarrow s' \) for any \( t', ta', s' \), then for every thread \( t \) in \( s \), say \( \text{thr s t} = [(e, (x, ln))] \),
3.3. Deadlock and type safety

(i) if $e = Val\ v$, then $ln = K^f 0$ and $P, shr s \vdash v : \leq T$ where

$$(\text{start-ETs } P C M)(\text{ttas} [\rightsquigarrow] P, -, - \vdash - \checkmark) t = [(E, T)]$$

and start-ETs $P C M$ is the initial map $[\text{start-tID} \mapsto \text{(empty, } T_r)]$ where $T_r$ is $M$'s return type.

(ii) if $e = \text{Throw } a$, then $ln = K^f 0$ and typeof-addr $(shr s) a = [\text{ClassT } C]$ for some $C$ such that $P \vdash C \preceq^* \text{Throwable}$,

(iii) otherwise, $t \in J.\text{deadlocked } P s$.

In any case, $t$ has an associated Thread object at address $t2a\ t$ in $shr\ s$.

Let me first review the type safety statement. Suppose we run the non-native method $M$ of class $C$ with the correct number of parameters $v$s of the correct types, and this halts in state $s$. Then, all threads of $s$ either (i) have terminated normally with a return value $v$ which conforms to the return type of the thread’s initial method, which is $M$ for $t = \text{start-tID}$ and run otherwise, or (ii) have terminated abnormally with an exception $a$ which refers to an object of a subclass of Throwable, or (iii) are deadlocked. In particular, this also shows that synchronized blocks cannot get stuck because the thread does not hold the lock on the monitor. Note that type safety does not state anything about non-terminating program runs. These are uninteresting, because they are obviously not stuck, but do not return anything either.

In the remainder of this section, I develop the invariant necessary to ultimately prove Theorem 3.3 via progress and preservation. I always assume that $P$ is well-formed, i.e., $\text{wf-J-prog } P$.

For progress, I have already discussed in §3.3.3 that it does not suffice to show that every non-final thread can reduce in the single-threaded semantics – the interleaving semantics may not be able to execute the reduction, because the current state violates the thread action’s precondition. In locale $\text{progress}$ (Figure 3.31), I have collected sufficient conditions to lift a single-threaded progress property to the interleaving semantics (Theorem 3.2). Now, I show that $J\text{-red } P$ satisfies these conditions. Remember that the locale $\text{progress}$ restricts the assumptions about the single-threaded semantics to a set $\text{wf-states}$ of well-formed states that all reductions of the interleaving semantics must preserve. Hence, I first develop what the well-formedness constraints for $J$ are.
Interleaving semantics

The progress theorem for single threads (Theorem 2.1) requires that (i) the expression is type correct, (ii) it passes the definite assignment check, and (iii) the heap conforms. Subject reduction (Theorem 2.2) shows preservation for (i), but additionally requires that the local store conforms. Lemmata 2.1 and 2.2 show preservation of definite assignment and conformance, respectively.

First of all, I must adapt these proofs to the changes and extensions from the previous section. In particular, I add typing rules for \(\text{sync}(\_\_\_\_\_\_\_\_\_)\) blocks to the run-time type system (see Figure 3.32). The rule \(\text{WT}_{\text{TrtSync}}\) replaces \(\text{WT}_{\text{Sync}}\) for \(\text{sync}(e_1) e_2\) and drops the constraint \(T_1 \neq NT\), because \(e_1\) may evaluate to \textit{null} during reduction. There is now a rule \(\text{WT}_{\text{TrtInsync}}\) for \(\text{insync}(a) e\) blocks, because \(\text{Rlock}\) introduces them in reductions. It requires the monitor address to be allocated and the body to be run-time typable. With these rules, proving the cases for \(\text{sync}(\_\_\_\_\_\_\_\_)\) blocks in progress and subject reduction follows the standard pattern. Since \(\text{sync}(\_\_\_\_\_\_\_\_)\) blocks involve neither the local store nor the heap, preservation of definite assignment and conformance is not affected.

Moreover, subject reduction now additionally requires that the ID of the executing thread has an associated thread object, because calling \textit{currentThread} returns its address (\texttt{CurrTh}). Thread conformance \(P, h \vdash t \sqrt{i}\) captures this as it predicates that \textit{typeof-addr} \(h\) \(\langle t2a t \rangle = [\text{ClassTC}]\) for some \(C\) such that \(P \vdash C \preceq^* \text{Thread}\). Consequently, the subject reduction theorem now is as follows. Again, additions with respect to 2.2 are highlighted in grey.

\textbf{Theorem 3.4 (Subject reduction).} If \(\text{wf-J-prog} P\), and \(P, t \vdash \langle e, s \rangle \rightarrow t a \rightarrow \langle e', s' \rangle\), and \(P, E, hp s \vdash e : T\), and \(P, E \vdash s\sqrt{i}\), and \(P, hp s' \vdash t \sqrt{j}\), then there is a \(T'\) such that \(P, E, hp s' \vdash e' : T'\) and \(P \vdash T' \leq T\).
Naturally, \( wf\text{-}states \) must impose all these thread-local constraints. In §3.1.4, I have developed the infrastructure for lifting such predicates and preservation theorems to the interleaving semantics. For definite assignment, the lifted predicate is \( \uparrow \lambda t \ (e, xs) \ h \ D e \ [\text{dom} \ xs]^\uparrow \), written \( \uparrow D \uparrow \). Conformance and typability depend on a typing environment \( E \) and the initial type \( T \) of the expression, which do not change during reduction. Hence, I model them as invariant data in a combined predicate. Let \( P, (E, T), t \vdash (e, xs), h \ \sqrt{ } \) denote

\[
\exists T'. P, E, h \vdash e : T' \land P \vdash T' \leq T \land P, E \vdash (h, xs) \sqrt{ } \land P, h \vdash t \ \sqrt{ } (3.7)
\]

Then, \( \uparrow \uparrow \) lifts \( P, _, _ \vdash _, _ \) to multithreaded states using the lifting infrastructure from §3.1.4:

\[
P, ETs \vdash ts, h \uparrow \uparrow \uparrow = \uparrow P, _, _ \vdash _, _ \uparrow \uparrow \uparrow ETs ts h
\]

A heap \( h' \) extends \( h \) (written \( h \triangleleft h' \)) iff \( h' \) allocates at least all addresses that \( h \) does, and types and array lengths of entries in \( h \) are unchanged in \( h' \).\(^{22}\) Heap extension abstracts from changes to data in object fields and array cells during reductions:

**Lemma 3.12.** If \( P, t \vdash \langle e, (h, xs) \rangle -ta\rightarrow \langle e', (h', xs') \rangle \), then \( h \triangleleft h' \). If \( P \vdash s -t:ta\rightarrow s' \) or \( P \vdash s -ttas\rightarrow^* s' \), then \( shr s \triangleleft shr s' \).

As conformance and the run-time type system only rely on type information in the heap, they are monotone with respect to heap extensions:

**Lemma 3.13.** Let \( h \triangleleft h' \). If \( P, h \vdash v : \leq T \), then \( P, h' \vdash v : \leq T \). If \( P, E, h \vdash e : T \), then \( P, E, h' \vdash e : T \). If \( P, h \vdash t \ \sqrt{ }_I \), then \( P, h' \vdash t \ \sqrt{ }_I \).

**Lemma 3.14** (Preservation of definite assignment, typability and conformance). Suppose \( P \vdash s -t:ta\rightarrow s' \).

(i) If \( \uparrow D \uparrow (thr s) (shr s') \), then \( \uparrow D \uparrow (thr s') (shr s') \).

(ii) If \( P, ETs \vdash thr s, shr s \ \uparrow \uparrow \uparrow \), let \( ETs' = ETs(\langle ta \rangle_t \rightsquigarrow P, _, _ \vdash _, _ \uparrow \uparrow \uparrow \) \). Then \( P, ETs' \vdash thr s', shr s' \ \uparrow \uparrow \uparrow \).

**Proof.** By Lemmata 3.2 and 3.3, it suffices to discharge the assumptions of locales \( lifting\text{-}wf \) and \( lifting\text{-}inv \) from §3.1.4 for \( \uparrow D \uparrow \) and \( P, _, _ \vdash _, _ \uparrow \uparrow \uparrow \), respectively.

\( \uparrow D \uparrow \) and \( \langle ta \rangle_t \rightsquigarrow P, _, _ \vdash _, _ \uparrow \uparrow \uparrow \) formalised in Jinja [82], but is not described in [83].
Chapter 3. Interleaving semantics

The first assumption of each follows from the preservation theorems for single threads (Theorem 3.4 and Lemmata 2.1 and 2.2, adapted to concurrency) except for thread conformance. Thread conformance is preserved because it is monotone with respect to heap extensions (Lemma 3.13) and reductions only extend the heap (Lemma 3.12).

The second assumption requires that the initial state of spawned threads satisfy the conditions $D$ and (3.7), respectively. Thread conformance holds because $t2a$ is the left-inverse to $a2t$ for addresses of objects of subclasses of Thread. For the rest, native-BTA2J builds the initial state from the method body of the run method, which must exist by well-formedness. Hence, its declaration meets $J$’s method well-formedness constraints (Figure 2.10), from which the desired constraints follow.

The third assumption imposes that changes to the heap by other threads preserve the constraints. Since $D$ does not depend on the heap, this holds trivially for (i). For (ii), such changes are only heap extensions by Lemma 3.12 with respect to which conformance and typability are monotone (Lemma 3.13).

\[ \square \]

**Inter-thread well-formedness constraints**

Unfortunately, these constraints do not suffice to discharge the assumption $\text{wf-red}$ of progress. It demands that if there is a reduction, then there is always a feasible one – except for deadlocking reductions due to Lock, Join, and IsIntrd _ True. For most reductions with BTAs, there are other reductions with negated preconditions such that one of them is always feasible. For example, Start and StartFail complement each other, and so do IntrdT and IntrdF. However, if a thread’s local state does not conform to the multithreaded state, $\text{wf-red}$ may be violated in two cases:

1. The lock status assigns less locks to a thread than its $\text{insync \ (\_\_\_)}$ blocks remember. In that case, RunLck and RunLckX try to unlock a monitor that is not held, but there is no reduction with UnlockFail.

2. The wait set status is [$\text{WS-Notified}$] or [$\text{WS-WokenUp}$], but the next reduction is not a native call to wait. Hence, the semantics cannot issue thread actions with Notified or WokenUp, respectively.

For both cases, I introduce additional constraints that the reductions in $J.\text{redTP}$ preserve.

116
For case 1, I define a function $I :: expr \Rightarrow addr \Rightarrow nat$ such that $I e a$ counts the $insync (a)$ _ subexpressions in $e$ for any monitor address $a$. I write $has-I e$ if $I e$ is not 0 everywhere, i.e., $e$ contains at least one $insync (_)$ _ subexpression. Then, the invariant $lock-conf ls ts$ expresses that for all thread IDs $t$,

(i) if $t$ does not exist, it holds no locks, i.e., if $ts t = None$, then $\neg has-lock (ls t) a$ for all monitor addresses $a$, and

(ii) if $t$ does exist, its $insync (_)$ _ subexpressions exactly remember its locks, i.e., if $ts t = [((e, xs), ln)]$, then $I e a = has-locks (ls t) a + ln f a$ for all $a$. Note that $lock-conf$ must add $t$’s temporarily released locks ($ln f a$) to the locks $t$ actually holds ($has-locks (ls t) a$), because the $insync (a)$ _ blocks remain when a call to $wait$ on $a$ temporarily releases the locks on $a$.

However, preservation of $lock-conf$ requires another invariant. Consider, for example, RsyncN and suppose that $e$ has an $insync (a)$ _ subexpression. Then, $I (sync (null) e) a > 0$ and $I (THROW NullPointer) a = 0$, but $a$’s lock state does not change. Hence, if the original state satisfies $lock-conf$, the successor state will not. The problem here is that $e$ contains an $insync (a)$ _ block although execution has not yet reached it.

To disallow such cases, I define the predicate $ok-I e$ which ensures that $insync (_)$ _ subexpressions occur only in subexpressions in which the next reduction will take place. Figure 3.33 shows the definition. For expressions with subexpressions, the definition follows a common pattern; all subexpressions must satisfy $ok-I$, too, and if $has-I$ for any subexpression, then all subexpressions which are evaluated before must be a value. The predicate $is-Val e$ expresses that $e$ is of the form $Val v$. For example, if $has-I e_2$ in $e_1 \ «bob» e_2$, then $e_1$ must be a value, because $e_1$ is evaluated before $e_2$. Control expressions allow $insync (_)$ _ blocks only for the currently evaluated subexpression; for example, in $e_1; e_2$, only for $e_1$. The loop $while (e_1) e_2$ does not allow them in either $e_1$ or $e_2$ because the semantics immediately rewrites it to $if (e_1) e_2; \ while (e_1) e_2 \ else \ unit$, which would duplicate any $insync (_)$ _ subexpression. The auxiliary function $has-Is es$ checks whether $has-I e$ holds for some $e \in set es$.

**Lemma 3.15.** If $\neg has-I e$, then $ok-I e$.

*Proof*. By induction on $e$. \qed
Like with the other thread-local well-formedness conditions, I lift \( ok-I \) to multithreaded states, written \( \uparrow ok-I \uparrow \), and show preservation with the locale lifting-wf.

**Lemma 3.16** (Preservation of \( ok-I \) and \( \uparrow ok-I \uparrow \)).

(i) If \( P, t \vdash \langle e, s \rangle \rightarrow t a \rightarrow \langle e', s' \rangle \) and \( ok-I \ e \), then \( ok-I \ e' \).

(ii) If \( P \vdash s \rightarrow t a \rightarrow s' \) and \( \uparrow ok-I \uparrow (t r h s) \ (s h r s) \), then \( \uparrow ok-I \uparrow (t r h s') \ (s h r s') \).

**Proof.** Case (i) by induction on the semantics. Isabelle proves all cases automatically except for non-native method calls (RCALL). In that case,
3.3. Deadlock and type safety

the body body of the called method is well-typed because $P$ is well-formed. Well-typed expressions have no insync (\_) subexpressions because there is no typing rule for them. Hence, ok-$I$ body by Lemma 3.15.

Case (ii) by Lemma 3.2 and instantiating the locale lifting-wf (Figure 3.15). Case (i) discharges the first assumption, and the same argument as for $R$call in the proof of (i) shows the second assumption. The third is vacuous because ok-$I$ does not depend on the heap. □

Now, I am finally able to prove preservation of lock-conf.

Lemma 3.17 (Preservation of lock-conf). Let lock-conf (lock$s$s)(thr $s$s) and $\uparrow$ok-$I$ $\uparrow$ (thr $s$s) (shr $s$s). If $P \vdash s - t:ta \rightarrow s'$, then lock-conf (lock$s'$s')(thr $s'$s').

Proof. By case analysis on the interleaving semantics. In case Acquire, $t$’s local state remains unchanged and therefore $I$, too. Moreover, upd-acq correctly acquires all of $t$’s temporarily released locks, because no other thread holds any of them by Acquire’s last assumption. Since the temporarily released locks $ln$ are reset to $K^f 0$, the sum of $ln_f a$ and has-locks ((lock$s$s)$(a)$) $t$ remains the same for all locks $a$. Moreover, other threads are not affected at all.

In case Normal, I have $P, t + \langle e, (shr s, xs) \rangle - ta \rightarrow \langle e', (shr s', xs') \rangle$ where $thr s t = [((e, xs), K^f 0)]$ and $thr s' t = [((e', xs'), ln')]$ for some $ln'$. Hence, ok-$I$ $e$. Induction on the small-step semantics (and case analysis for native method calls) shows that $I e' a$ is equal to $I e a$ plus the number of Lock→$a$ and less the number of Unlock→$a$ BTAs in $\langle ta \rangle_{lf} a$. But this change is exactly how upd-locks and upd-TRL change the sum of $t$’s locks and temporarily released locks on $a$, respectively (by induction on $\langle ta \rangle_{lf} a$ with the assumption ok-Ls ((lock$s$s)$(a)$) $t$ $\langle (ta)_{lf} a \rangle$). Hence, lock conformance holds for $t$ in $s'$. As all lock actions’ preconditions are satisfied, the reduction does not affect locks and local states of threads other than $t$, so lock conformance holds for them in $s'$, too. Having inspected all existing threads, I am left with newly spawned threads $t'$. Condition (1) in lock-conf’s definition ensures that $t'$ does not hold any lock. By the same argument as in the proof of Lemma 3.16, $t'$’s initial expression $e$ satisfies $\neg$has-$I$ $e$, because it is the body of the run method. Since $t'$ has not temporarily released any locks by construction (cf. Figure 3.9), lock conformance also holds for $t'$ in $s'$. □

I now turn to the second way in which $J$-red may violate wf-red. In principle, I could pursue the same path as for lock conformance and
require that whenever a thread’s wait set status is not None, its next reduction will be a call to the native method wait, and show preservation. However, formalising the native-call-to-wait invariant is tedious and preservation proofs are no easier. Instead, I define an invariant that is independent of the local state and that I can reuse for the bytecode type safety proof in §3.3.5.

Given a set I of well-formed multithreaded states, ok-Suspend I restricts I to states in which the local states of all threads with wait set status other than None have resulted from a former reduction whose thread action contains a Suspend BTA. Formally (in locale multithreaded-base):

\[
\text{ok-Suspend } I = \{s. s \in I \land \\
(\forall t \in \text{dom (wset } s). \exists s_0 \in I. \exists s_1 \in I. \exists t\mathbb{t}as x_0 ta x w ln ln'. \ s_0 -t:ta \to s_1 \land s_1 -t\mathbb{t}as \to^* s \land thr s_0 \ t = [ (x_0, K^f 0)] \land \\
t \vdash (x_0, shr s_0) -ta \to (x, shr s_1) \land \text{Suspend } w \in \text{set } \langle ta \rangle_w \land \\
ok-ta s_0 t ta \land \text{thr } s_1 \ t = [ (x, ln)] \land \text{thr } s t = [(x, ln')] \} \}
\]

Then, ok-Suspend preserves preservation of invariants by definition.

**Lemma 3.18 (Preservation of ok-Suspend).** If invariant redT I, then invariant redT (ok-Suspend I).

**Type safety**

Finally, I define the set J-wf-states P of well-formed states for the type safety proof as

\[
J.\text{ok-Suspend } P \{s. \exists ETs. P, ETs \vdash thr s, shr s \uparrow \sqrt{\Gamma} \land \uparrow \mathcal{D} \uparrow (thr s) (shr s) \land \\
\uparrow \text{ok-I} \uparrow (thr s) (shr s) \land \text{lock-conf (locks } s) (thr s) \}
\]


**Proof.** I proof the assumptions of locale progress (Figure 3.31) as follows. Invariance of J-wf-states follows from Lemmata 3.14, 3.16, 3.17, and 3.18. The well-formedness condition ok-locks-thr directly follows from lock-conf because this is just case (i) in lock-conf’s definition. ok-wset-final follows from J.ok-Suspend and the assumption Suspend-not-final, which I discharge below. progress is just the progress theorem.
2.1, adapted for concurrency. Inductions on the small-step semantics show the assumptions $wf-ta$, $Suspend-not-final$, and $Wakeup-waits$.

Now, only $wf-red$ remains to be shown. By $s \in J-wf-states P$, if $t$’s wait set status is not $None$, $t$’s last reduction must have issued a $Suspend$ BTA in a state with a heap which the current heap $shr s$ extends. Induction on this reduction shows that $t$ can now reduce with basic thread actions $\langle Notified \rangle$ and $\langle WokenUp \rangle$ as necessary.

So suppose $wset s t = None$. Without loss of generality, assume $\neg ok-ta’ s t ta$. Proof by induction over the small-step semantics. The interesting cases are $RunLck$ and $Rnative$.

In case $RunLck$, $I (insync (a) _) a > 0$, i.e., by lock conformance, $t$ holds the lock $a$. Thus, unlocking $a$ is possible, i.e., $ok-ta’ s t ta$ holds.

In case $Rnative$, the proof proceeds by case analysis of the semantics for native methods. For each case with a non-trivial thread action, one must manually provide the alternative reduction. I present $Join$ as an example: From $\neg ok-ta’ s t ta$, I obtain $t \in intrs s$, because $ok-ta’$ does not check the precondition of $Join$ to allow for deadlocks. Hence, $JoinIntr$ is possible. □

**Corollary 3.1.** Let $s \in J-wf-states P$ be not in deadlock. Then, there are $t$, $ta$, and $s’$ such that $P \vdash s -t:ta\rightarrow s’$.

**Proof.** This is Theorem 3.2 with Lemma 3.19 discharging the locale assumptions. □

The initial state $J-start P C M vs$ satisfies all these well-formedness constraints.

**Lemma 3.20.** If $wf-J-prog P$ and $wf-start P C M vs$, then $J-start P C M vs \in J-wf-states P$.

Finally, I am able to prove type safety.

**Proof of Theorem 3.3.** By Lemma 3.20, $J-start P C M vs \in J-wf-states P$. Since $J-wf-states P$ is closed under reductions, $s \in J-wf-states P$, too. Hence, $s \in J.deadlock P$ by Corollary 3.1, which subsumes all $mfinal$ states. If $e$ is $final$, cases (i) and (ii) follow from $s$ being well-formed. Otherwise, Theorem 3.1 yields case (iii). The associated thread object exists because $s \in J-wf-states P$ implies thread conformance. □
Preservation of deadlocks

In §3.3.2, the locale \textit{preserve-deadlocked} established conditions on the single-threaded semantics such that threads in deadlock remain in deadlock, even when other threads keep executing. Now, I show that \textit{J-red} \( P \) satisfies these conditions with \textit{J-wf-states} \( P \) as the set of well-formed states.

\textbf{Lemma 3.21.} \textit{J-red} \( P \) meets the assumptions of locale \textit{preserve-deadlocked} (Figure 3.30), i.e., \textit{preserve-deadlocked \( J \)-final} (\textit{J-red} \( P \)) (\textit{J-wf-states} \( P \)).

\textit{Proof.} The first assumption is identical to the first in \textit{progress} and therefore holds by the same argument as in Lemma 3.19. The assumptions \textit{progress} and \textit{wf-ta} of \textit{progress} imply preservation of \( P, t \vdash _t \) (second assumption in \textit{preserve-deadlocked}). The third assumption follows from the next lemma, because \( P, s \vdash _t \) \( ta \rightarrow s' \) implies \( shr s \triangleright shr s' \). \( \square \)

\textbf{Lemma 3.22.} Suppose \( P, t \vdash \langle e, (h', xs) \rangle - ta \rightarrow \langle e', s' \rangle \). If \( P, E, h + e : T \) for some \( h \) such that \( h \leq h' \), \( P, h \vdash h \), and \( P, h \vdash t \), then there exist \( ta', e'', s'' \) such that \( P, t \vdash \langle e, (h, xs) \rangle - ta' \rightarrow \langle e'', s'' \rangle \) and \( \text{waits } ta' = \text{waits } ta \).

\textit{Proof.} By induction on the small step semantics. Well-typedness of \( e \) in \( h \) and heap conformance ensure that field access and method lookup still succeed, heap extension ensures that the same methods (in particular native ones) are called. \( \square \)

Preservation of deadlock now follows from Lemmata 3.11 and 3.21.

\textbf{Theorem 3.5} (Preservation of deadlock). Suppose \( s \in \textit{J-wf-states} \( P \). If \( P, s - t:ta \rightarrow s' \) or \( P, s - ttas \rightarrow s' \), then \textit{J.deadlocked} \( P \) \( s \subseteq \textit{J.deadlocked} \( P \) \( s' \).

\subsection{3.3.5 Type safety for bytecode}

In this section, I show type safety for well-typed bytecode. The approach is the same as for source code (§3.3.4), namely (i) identify necessary additional invariants, and (ii) prove the assumptions of locale \textit{progress}. Instead of presenting the steps in detail once more, I focus on the similarities with and differences from source code. In this section, I always assume that \( \Phi \) is a well-typing for \( P \).

\textbf{Theorem 3.6} (Type safety). Let \( P \) be well-formed with well-typing \( \Phi \) and the start state \textit{jvm-start} \( P \ C M \) \( vs \) be well-formed.
3.3. Deadlock and type safety

(a) The aggressive VM runs until all threads have terminated or are deadlocked. Formally:
If \( P \vdash P, C M \vdash \text{start} P C M \vdash t\text{tas} \rightarrow^{*}_{jvm} s \) such that \( \neg P \vdash s - t\text{:tas} \rightarrow^{*}_{jvm} s' \) for any \( t, ta, s' \), then for every thread \( t \) in \( s \), say \( \text{thr} t = \lfloor (xcp, frs), ln \rfloor \),

(i) \( P, \Phi \vdash t : (xcp, shr s, frs) \sqrt{ } \), and
(ii) if \( frs \neq \[] \) or \( ln \neq K \), then \( t \in jvm\text{.deadlocked} P s \).

(b) Aggressive and defensive VM commute. Formally:
\( P \vdash jvm\text{-start} P C M \vdash t\text{tas} \rightarrow^{*}_{jvm} s \) iff
\( P \vdash jvm\text{-start} P C M \vdash t\text{tas} \rightarrow^{*}_{jvmd} s \).

Compare this statement of type safety to the one for the sequential VM (Theorem 2.3). Theorem 2.3(a) showed that the sequential defensive VM never raises a type error. However, the multithreaded VM \( jvm\text{-execd} \) cannot raise such type errors by construction because of the following drawbacks of modelling type errors: On the one hand, I could have adjoined \( \text{TypeError} \) to the thread-local state of the defensive VM, but as I have argued in §3.2.3, I then cannot reuse the proof invariants for the defensive and aggressive VM. On the other hand, a single thread raising a type error could halt the whole VM with a type error. However, this does not fit the structure of the interleaving semantics because a single thread cannot abort the execution of other threads.

Hence, I cannot express absence of type errors directly, but I show progress (Lemma 3.27) instead. This is equivalent to the absence of type errors for individual steps, because type errors and normal reductions exclude each other in the defensive VM, see (2.5). Yet, Theorem 3.6 is slightly weaker than 2.3 for type errors, because the former does not exclude the case in which one thread is stuck at a type error and another thread runs for ever.

Parts 3.6(a)(i) and 3.6(b) express the remaining two parts of Theorem 2.3, where \( P, \Phi \vdash _{-} \sqrt{ } \) strengthens \( P, \Phi \vdash _{-} \sqrt{ } \) to include thread conformance (see below).

Let me now turn to the proof of Theorem 3.6. Both the theorem and its proof are in structure similar to type safety for source code (Theorem 3.3). First, I describe the necessary well-formedness constraints \( jvm\text{-wf-states} P \Phi \) on the multithreaded states. Then, I prove that \( jvm\text{.redT} P \) and \( jvmd\text{.redT} P \) preserve them and the start state \( jvm\text{-start} P C M \vdash \text{vs} \) satisfies them, and that they are sufficient to discharge
the assumptions of locale progress. The actual proof of Theorem 3.6 follows the same line as the one for Theorem 3.3, so I do not repeat it here.

Well-formedness constraints

Just like \( J\text{-wf-states} \) \( P \), \( jvm\text{-wf-states} \) \( P \) \( \Phi \) includes all well-formedness constraints of the sequential type safety proof, i.e., state conformance. Due to the new native method \( \text{currentThread} \), single-threaded preservation now additionally requires thread conformance. Hence, I strengthen bytecode conformance to include thread conformance

\[
P, \Phi \vdash t : s \sqrt{=} P, \Phi \vdash s \sqrt{=} P, \text{snd} s \vdash t \sqrt{i}
\]

and lift it to thread pools with the infrastructure from §3.1.4.

\[
P, \Phi \vdash (ts, h) \uparrow \downarrow = \uparrow \land t : (xcp, frs) h. P, \Phi \vdash t : (xcp, h, frs) \sqrt{=} ts h
\]

Then, I define \( jvm\text{-wf-states} \) \( P \) \( \Phi \) as

\[
jvm.ok-Suspend P \{ s. P, \Phi \vdash (thr s, shr s) \uparrow \downarrow \wedge \text{ok-locks-thr} (locks s) (thr s) \}
\]

Let me compare \( jvm\text{-wf-states} \) with \( J\text{-wf-states} \). Using the language-independent invariant transformer \( ok\text{-Suspend} \), they both ensure that wait set status and thread-local states agree. Note that it is irrelevant whether \( jvm\text{-wf-states} \) uses \( jvm.ok\text{-Suspend} \) or \( jvmd.ok\text{-Suspend} \). They both yield the same set because aggressive and defensive VM commute for conformant states (see Lemma 3.23 below). Also, they both include the thread-local well-formedness constraints lifted to multithreaded states with \( \uparrow \_ \_ \_ \) and \( \uparrow \_ \_ \_ \).

In contrast, \( J\text{-wf-states} \) and \( jvm\text{-wf-states} \) differ on the inter-thread well-formedness constraints. The latter requires only \( ok\text{-locks-thr} \), because locale progress does so. In particular, no constraints on the locks are necessary, because unlike \( \text{sync} (\_\_\_\_) \_ \) blocks, unlocking a monitor with \( M\text{Exit} \) may fail (see Figure 3.23).

Proof of type safety

Now, I present the key lemmata for proving the JinjaThreads VM type safe.
Lemma 3.23. Aggressive and defensive VM commute.

(i) Let $P, \Phi \vdash t:s$ $\sqrt{\text{.}}$. Then

$P, t \vdash \text{Normal } s \rightarrow_{\text{jvm}} \text{ Normal } s$ iff $P, t \vdash s \rightarrow_{\text{jvm}} s'$.

(ii) Let $P, \Phi \vdash (\text{thr } s, \text{shr } s) \rightarrow^{\sqrt{\text{.}}^{\text{.}}\sqrt{\text{.}}}$. Then

$P \vdash s \rightarrow_{\text{jvm}} s'$ iff $P \vdash s \rightarrow_{\text{jvm}} s'$.

Proof. Case (i) adapts Theorem 2.3 (b) to include thread ID and thread actions. I extend the proof to cover the new native methods and monitor instructions.

For (ii), either direction proceeds by case analysis of the reduction. Case ACQUIRE is trivial because it does not depend on the single-threaded semantics. Case NORMAL follows from (i).

The next lemma shows that the VM does not get stuck. Together with Lemma 3.23, it strengthens absence of type errors (Theorem 2.3 (a)) in that the defensive VM may not get stuck.

Lemma 3.24 (Progress). Let $P, \Phi \vdash t: (\text{xcp }, h, \text{frs } s) \sqrt{\text{.}}$ and frs $\neq []$. Then, there are $ta$ and $s'$ such that $P, t \vdash (\text{xcp }, h, \text{frs } s) \rightarrow_{\text{jvm}} s'$.

Next, I turn to the preservation lemmata.

Lemma 3.25 (Preservation of conformance). If $P, \Phi \vdash (\text{thr } s, \text{shr } s) \rightarrow^{\sqrt{\text{.}}}^{\sqrt{\text{.}}}$ and $P \vdash s \rightarrow_{\text{jvm}} s'$, then $P, \Phi \vdash (\text{thr } s', \text{shr } s') \rightarrow^{\sqrt{\text{.}}}^{\sqrt{\text{.}}}$.

Proof. It suffices to discharge the assumptions of locale lifting-wf from §3.1.4. Single-threaded preservation (assumption 1) needs to extend the sequential type safety proof (Theorem 2.3 (c)) for the new native methods and monitor instructions.

The argument for spawned threads (assumption 2) follows the one for non-native method calls. By construction, the initial state of a new thread is just one call frame for the parameter-less run method with program counter 0. Condition (iii) of method well-typings ensures that the frame conforms to the well-typing.

Preservation for another thread $t$ changing the heap (assumption 3) falls in two parts. $t$’s reduction itself preserves heap conformance (condition (ii) in the definition of $P, \Phi \vdash _{\sqrt{\text{.}}}$) and extends the heap. As all other conformance conditions (exceptions, call frames, thread) are monotone with respect to heap extensions, conformance is preserved. □
Lemma 3.26 (Preservation of jvm-wf-states). \( P \vdash _\rightarrow \) jvm \( \Leftrightarrow \) and \( P \vdash _\rightarrow \) jvmd \( \Leftrightarrow \) preserve jvm-wf-states \( P \Phi \).

Proof. Note that redT preserves ok-locks-thr independent of the single-threaded semantics (proof by case analysis). Then, for the aggressive VM, preservation follows from preservation of conformance (Lemma 3.25) and preservation of ok-Suspend (Lemma 3.18).

By Lemma 3.23, Lemma 3.25 also holds for the defensive VM. As noted above, I would have obtained the same set if jvm-wf-states’s used jvmd.ok-Suspend. Therefore, preservation for the defensive VM follows with Lemma 3.18. □

Lemma 3.27. The aggressive and the defensive VM satisfy the assumption of progress for well-formed states jvm-wf-states \( P \Phi \).

Proof. I only sketch the proof for the defensive VM. For the aggressive VM, I reuse the results for the defensive VM and Lemma 3.23. Invariance of jvm-wf-states \( P \Phi \) follows from Lemma 3.26. Assumption wf-stateD holds because jvm-wf-states \( P \Phi \) explicitly requires ok-locks-thr \( s \) and jvm.ok-Suspend \( P \) enforces ok-wset-final \( s \) like for \( J \). Lemma 3.24 discharges progress. Assumptions wf-red, Suspend-not-final and Wakeup-waits are shown by case analysis whether an exception is flagged and which instruction executes next. For wf-red, other than in Theorem 3.3 for RUNLCK, the case for MExit is straightforward because MExit may fail with an IllegalMonitorState exception. □

Lemma 3.28. The initial state is well-formed, i.e., if wf-start \( P C M \ vs \), then jvm-start \( P C M \ vs \in \) jvm-wf-states \( P \Phi \).

Deadlock preservation

Preservation of deadlocks for the JinjaThreads VM is analogous to \( J \), Lemma 3.29 corresponds to Lemma 3.21. The proof for the defensive VM is analogous, too, where Lemma 3.30 replaces Lemma 3.22.

Lemma 3.29. For well-formed states jvm-wf-states \( P \Phi \), exec \( P \) and execd \( P \) satisfy the assumptions of locale preserve-deadlocked.

Lemma 3.30. Let \( P, t \vdash Normal \ (xcp, h', f rs) -ta\rightarrow jvmd Normal s' \) and \( P \vdash h' \sqrt{.} \). If \( P, \Phi \vdash t: (xcp, h, f rs) \sqrt{.} \) for some \( h \) such that \( h \leq h' \), then there exist \( ta' \) and \( s'' \) such that \( P, t \vdash Normal \ (xcp, h, f rs) -ta'\rightarrow jvmd Normal s'' \) and \( waits \ ta' \subseteq \) waits \( ta \).
3.4. Related work

Hence, preservation of deadlock again follows with Lemma 3.11. Since aggressive and defensive VM commute, preservation also extends to the aggressive VM.

**Theorem 3.7** (Preservation of deadlock). Suppose \( s \in jvm-wf-states \) \( P \Phi \).

\( (i) \) If \( P \vdash s \xrightarrow{-\text{t:ta}}_{jvmd} s' \) or \( P \vdash s \xrightarrow{-\text{ttas}}_{jvmd}^* s' \), then \( jvmd.deadlocked \ P \ s \subseteq jvmd.deadlocked \ P \ s' \).

\( (ii) \) If \( P \vdash s \xrightarrow{-\text{t:ta}}_{jvm} s' \) or \( P \vdash s \xrightarrow{-\text{ttas}}_{jvm}^* s' \), then \( jvm.deadlocked \ P \ s \subseteq jvm.deadlocked \ P \ s' \).

3.4 Related work

3.4.1 Formalisations of Java and Java bytecode

Formalisations of (aspects of) sequential Java and Java bytecode abound in the literature, many of which study Java features that Jinja and JinjaThreads omit (cf. §7.4). Hartel and Moreau [67] provide a good overview, Alves-Foss [7] has collected many early works. Most closely related to JinjaThreads are its predecessors Jinja, Bali and \( \mu \)Java for sequential Java, see §2.3 for a detailed comparison.

Most formalisations cover either only Java source code or only Java bytecode. One notable exception is the semantics by Stärk et al. [166] for a subset of Java source code and bytecode in terms of abstract state machines, for which they prove subject reduction. Recently, Grunwald et al. [59] extended it to Java generics. However, they use neither machine support for the semantics nor for checking their proofs.

For concurrent Java, AtomicJava [51] by Flanagan et al. models most Java source code features except inheritance and exception handling. They use it to show that their non-standard type system ensures atomicity.

There are much more formalisations for multithreaded Java bytecode. First, Liu and Moore [104] report on an executable model M6 of the KVM, a JVM implementation for embedded devices, in ACL2, which covers all aspects of the Connected Limited Device Configuration (CLDC) specification [41]. Like JinjaThreads, their VM semantics implements native methods from the CLDC standard library, in particular to deal with reflection, class loading and concurrency. Since they model only the JVM, the implementations for the native methods manipulate the
VM state directly. In contrast, JinjaThreads encapsulates the semantics (and signatures) of native method calls such that both source code and bytecode can reuse it. They aim for verifying small Java programs [105, 122] and JVM implementations with respect to the JVMS. Thus, they do not define a type system for bytecode nor prove type safety. The M6 models bytecode much closer to the JVMS than JinjaThreads does, e.g., the M6 explicitly models the constant pool and string literals. Like its predecessor, JinjaThreads abstracts from such technical details.

Second, Bicolano [121] serves as the basis for the proof carrying code infrastructure in the Mobius project [13]. It provides a comprehensive model for CLDC except for concurrency and class loading in Coq, which includes the class file format. In BicolanoMT, Huisman and Petri [70] extend Bicolano with interleaving concurrency. By using the extension framework by Czarnik and Schubert [42], they do not need to change the sequential Bicolano semantics at all. In contrast, JinjaThreads adds the semantics of MEnter and MExit instructions to the single-threaded semantics and clutters all rules with thread actions. Conversely, JinjaThreads uses the same interleaving semantics for source code and bytecode whereas the BicolanoMT extension is tightly tied to the bytecode language.

Third, Belblidia and Debbabi present a formal small-step semantics for multithreaded Java bytecode [20]. Like JinjaThreads, they have a semantics for threads in isolation and a second layer which manages the threads and receives basic thread actions, which they call labels, from them. In contrast to the JinjaThreads interleaving framework, at most one basic thread action can be issued at a time. Their single-threaded semantics already takes care of the locks, which are stored in the shared memory, i.e., they only have actions for creating, killing, blocking, and notifying threads. Yet, neither do they model the wait-notify mechanism, nor thread interruption; the second layer uses the block and notify actions to keep track of which threads are ready for execution. Like the M6 and Bicolano, they only give the semantics, but no type system and no proofs. In JinjaThreads, the framework semantics manages the complete multithreaded state which includes the locks, wait sets, and interrupts. This isolates them from one another and alleviates individual threads from the burdens of multithreading. By allowing multiple basic thread actions in a single reduction step, single threads can combine basic thread actions as building blocks for more complex behaviour, while each basic thread action still has simple semantics.
3.4. Related work

Forth, JavaFAN by Farzan et al. [48,47] is a formal analyser for Java source and bytecode in Maude, see §6.6 for details. Although they provide formal semantics for both source code and bytecode, these are unconnected.

Apart from the ones mentioned above, I know of two other large-scale formalisations of sequential Java bytecode. First, Barthe et al. [14,15] built a JVM similar to the sequential JinjaThreads VM in Coq, which covers the JavaCard platform. They show type safety for type-correct bytecode similar to Theorem 2.3.

Second, Atkey [9] has developed an executable JVM in Coq, which encodes the checks of the defensive VM as dependent types. He argues that this simplifies large-scale proofs against the JVM because they no longer need to show that these checks cannot fail.

3.4.2 Type safety proofs and deadlocks

There are only few type safety proofs for multithreaded languages like Java. However, often only subject reduction is shown, which eliminates the need to deal with deadlocks.

In [58], Grossman reports on type-based data race detection for multithreaded Cyclone – a type safe variant of C. In the type safety proof, he shows the progress property that no well-typed thread can get “badly stuck”. A thread is badly stuck iff it either is final and still holds some locks or it would not be able to reduce any further even if it could acquire an arbitrary additional lock. Together with subject reduction, type safety follows, i.e., all threads reachable from a well-typed thread via reductions are not badly stuck. Like deadlocked in JinjaThreads, badly stuck is defined semantically. In general, being deadlocked is stronger than being badly stuck because the latter does not involve the aspect of circular waiting.

Goto et al. [57] also prove type safety for a multithreaded calculus with a weak memory model. Their progress statement only applies to a thread if it is not about to execute a synchronisation statement, which is a crude syntactic approximation of deadlock. Progress with this restriction no longer guarantees that the semantics is not missing any rule. For instance, it holds even for an operational semantics without any rules for synchronized statements.

There are also approaches to prevent potential deadlocks through type systems, which also must formalise deadlock. For example, Suenaga
and Kobayashi [169] propose a calculus with thread creation, interrupts and synchronisation via structured locking. They assign to each syntactic occurrence of a lock a unique level tag. Their type system remembers bounds on the level of acquired locks in effect labels and ensures deadlock freedom by requiring that locks must be acquired in ascending order, which breaks the circular waiting condition. Their deadlock formalisation is purely syntactic: A set of threads is in deadlock iff every “reducible” subexpression of the thread’s expressions is a synchronisation statement which has to acquire a lock which is already held. For JinjaThreads, such a syntactic characterisation could be obtained from the semantic definition of deadlock, but I have not done so. However, this cannot express that some threads are in deadlock while others are still active like deadlocked does. This unnecessarily weakens their type safety statement.

### 3.4.3 Large-scale programming language formalisations

Beyond Java, there are several other large-scale formalisations of programming languages in proof assistants. Foster and Vytiniotis [52] have formalised the core of FeatherweightJava [71] in Isabelle/HOL. Later, Delaware et al. [45] formalised FeatherweightJava with various extensions in Coq.

There are also substantial formalisations of C and C++, which are predecessors to Java. Based on Jinja, Wasserrab et al. [178] have formalised multiple inheritance in C++ and proven it type-safe. Ramananandro et al. [148, 149] have translated this semantics to Coq and extended it with concrete memory layouts and object construction and destruction. Norrish [131–133] has formalised large parts of the C and C++ language in HOL. Krebbers and Wiedijk [90] are working on formalising the full C99 standard. They aim to correctly capture the low-level intricacies of C while keeping the semantics relatively easy to use. Schirmer [158, 159] developed the SIMPL framework for sequential imperative languages in Isabelle/HOL and embeds a subset of C. As he concentrates on verifying programs, he devises an operational and axiomatic semantics where the latter is sound and complete with respect to the former.

Compiler verifications also come with formalised semantics, see §5.7 for a discussion.
Zappa Nardelli et al. demand that a specification for a multiprocessor or programming-language memory model “be integrated with the semantics of the rest of the system (describing the behaviour of the processor instructions or of the phrases of the programming language). Memory models are typically presented in isolation, and this makes it all too easy to gloss over important details.” [181] Hitherto, this criticism has applied to Java and the JMM, too [8,38,69]. In this chapter, I address this problem by linking the JinjaThreads semantics from Chapter 3 with the JMM. Moreover, I study whether this connection sustains the DRF guarantee, type safety, and the Java security architecture.

To that end, I parametrise the semantics from Chapter 3 over the memory model (MM) first (§4.1). Thus, the definitions and theorems from Chapter 3 are in fact the special case for sequential consistency (§4.2). The main contributions of this chapter then revolve around the JMM:

- a formal link between multithreaded Java and the JMM (§4.3.2),
- proofs of the DRF guarantee (§4.3.3) and consistency of the JMM (§4.3.4), and
- a proof that the JMM provides only a notion of type safety weaker than sequential consistency and an example that the JMM compromises Java’s security guarantees (§4.3.5).

Technically, the challenges in this chapter are the following:

**abstract over the memory model** Chapters 2 and 3 define the semantics in terms of the concrete heap representation from §2.1.5. To support different memory models, I abstract over the implementation. To that end, I identify a set of kernel operations for shared memory together
with their properties and adapt the definitions and proofs. These operations are flexible enough to cover sequential consistency and the JMM.

**link JinjaThreads with the JMM** Implementing the kernel operations does not suffice for the JMM, because its axiomatic rules decide a posteriori whether an execution is allowed. Therefore, I connect the operational semantics for source code and bytecode with the JMM by associating statements and instructions with their JMM events, i.e., the events from Table 3.1 that I asked the reader to ignore during Chapter 3.

**prove the DRF guarantee and consistency of the JMM** The JMM and the DRF guarantee have been formalised before [8, 69], but unconnected to a single-threaded semantics. To bridge the gap, I identify the assumptions of the DRF proof and show that the single-threaded semantics satisfy them. Although these assumptions are intuitive, discharging them surprisingly requires a subject reduction proof for non-speculative executions. Initialisations turn out to be the main complication. In particular, I construct sequentially consistent executions for a given prefix.

For consistency, I show that the JMM allows all sequentially consistent executions of all well-formed programs and, therefore, every well-formed program has at least one legal behaviour. To my knowledge, this is the first consistency proof for the JMM.

### 4.1 The heap as a module

So far, I have only sketched how JinjaThreads models the heap in §2.1.5. In fact, JinjaThreads models the heap as a module with abstract operations and provides implementations for sequential consistency and the Java memory model. In this section, I introduce the module and its operations (§4.1.1) and show how the semantics and proofs use it (§4.1.2). In §4.1.3, I discuss some design decisions for the module.

In Chapters 2 and 3, addresses and thread IDs have been taken from the opaque types `addr` and `thread-id`, respectively. The JMM and code generation (§6.3.1) require more concrete types. Since Isabelle does not support type refinement, these types are actually type variables `'addr`
4.1. The heap as a module

and ’t in JinjaThreads. Hence, all previous definitions that depend on addresses or thread IDs are in fact polymorphic. For example, the real types for values (Figure 2.1) and J programs (Figure 2.3) are ’addr val and ’addr J-prog, respectively. From now on, I write such types with these type parameters.

4.1.1 Abstract operations and their properties

In this section, I define the module’s interface and present the properties of the abstract operations. As one would expect from a module’s interface, the type variable ’heap generalises the type heap for heaps such that the concrete implementations in §4.2 and §4.3 can instantiate ’heap as needed.

In Figure 4.1, the locale declaration heap-base fixes the abstract heap operations. Locale heap specifies how they affect type information and array lengths that the heap stores. I now explain the intended meaning of the individual operations.

The functions a2t and t2a convert between addresses and thread IDs; I have already introduced them in §3.2.1. Since addresses and thread IDs are now type variables, I must fix the conversion functions as locale parameters, because different MMs might implement them differently. Assumption a2t-inverse states that t2a is the left-inverse of a2t on addresses at which objects of subclasses of Thread may be allocated. It ensures that the native method currentThread correctly returns the address of the current thread (Currect).

The parameter typeof-addr replaces the constant defined in Equation (2.4). From a heap, it extracts type information for addresses, which also includes the length of arrays.

The other operations manipulate the heap. The constant empty-heap denotes the heap which has no objects allocated. The operation alloc allocates a new object of the given class or an array of the given element type and size, respectively. It returns the updated heap h’, and the

---

23 The locale heap both fixes a parameter P and imposes assumptions, which deviates from the general rule of separating these steps (§1.4.2). However, heap only needs P to know about valid types and subtyping. Actually, heap-base should fix P, but this is impossible, because the compiler verification in Chapter 5 refers to definitions in heap-base for the source code and the compiled program at the same time. Although these definitions are provably equal, the locale mechanism and the HOL type system do not allow to conveniently combine them. As all definitions go into heap-base and theorems are not affected, heap fixes P and the declarations in heap-base explicitly have P as an additional parameter when necessary.
**datatype** \( \text{loc} = \text{Field vname cname} \mid \text{Cell nat} \)

**locale** \( \text{heap-base} = \)

\[
\text{fixes } a2t :: 'addr \Rightarrow 't \text{ and } t2a :: 't \Rightarrow 'addr
\]

\[
\text{and } \text{typeof-addr} :: 'heap \Rightarrow 'addr \rightarrow hty
\]

\[
\text{and } \text{empty-heap} :: 'heap
\]

\[
\text{and } \text{alloc} :: 'heap \Rightarrow hty \Rightarrow 'heap \times ('addr \text{ option})
\]

\[
\text{and } \text{read} :: 'heap \Rightarrow 'addr \Rightarrow loc \Rightarrow 'addr val \Rightarrow \text{bool}
\]

\[
\text{and } \text{write} :: 'heap \Rightarrow 'addr \Rightarrow loc \Rightarrow 'addr val \Rightarrow 'heap \Rightarrow \text{bool}
\]

**locale** \( \text{heap} = \text{heap-base} + \text{fixes } P :: 'm \text{ prog} \)

\[
\text{assumes } a2t^{-\text{inverse}}:
\]

\[
\begin{align*}
\llbracket \text{typeof-addr } h \ a = [\text{ClassT } C] \rrbracket; \ P \vdash C \leq^* \text{ Thread} & \implies t2a (a2t a) = a \\
\llbracket \text{alloc } h \ hT = (h', [a]) ; \ \text{is-htype } P hT \rrbracket & \implies \text{typeof-addr } h' \ a = |hT|
\end{align*}
\]

\[
\text{and } \text{alloc-type}:
\]

\[
\begin{align*}
\llbracket \text{alloc } h \ hT = (h', [a]) ; \ \text{is-htype } P hT \rrbracket & \implies \text{typeof-addr } h' \ a = |hT|
\end{align*}
\]

\[
\text{and } \text{alloc-hext} : \text{alloc } h \ hT = (h', ao) \implies h \leq h'
\]

\[
\text{and } \text{write-hext} : \text{write } h \ a \ al \ v \ h' \implies h \leq h'
\]

**Figure 4.1:** Locales \( \text{heap-base} \) and \( \text{heap} \) declare the heap module’s interface

allocated address \([a]\) – or \(\text{None}\) if the allocation fails, e.g., due to insufficient memory. If the allocation succeeds, \(a\)’s type information in \(h'\) must be correct – provided that the allocated type is valid \((\text{alloc-type})\); \(\text{is-htype } P hT\) abbreviates \(\text{is-type } P (\text{ty-of } hT)\).

The predicates \(\text{read}\) and \(\text{write}\) model access to the heap. The member \(al :: \text{loc}\) specifies which field \((al = \text{Field } F D)\) or array cell \((al = \text{Cell } n)\) of an address should be accessed. An address \(a\) and a member \(al\) identify a location \((a, al)\). Given the current heap \(h\) and a location \((a, al)\), \(\text{read } h \ a \ al\) models (as a predicate) the set of values that memory may return. Similarly, writing to a location in the heap \((\text{write } h \ a \ al \ v \ h')\) non-deterministically updates the heap. Allocations and write access must be implemented such that they extend the heap \((h \leq h')\), i.e., type information grows monotonically.

\[
h \leq h' = \text{typeof-addr } h \subseteq_m \text{typeof-addr } h'
\]

Type safety requires further assumptions (Figure 4.2); in particular, the values that memory returns must conform to the type. This requires the
locale conf-base = heap-base +
  fixes hconf :: 'heap ⇒ bool and P :: 'm prog

locale conf = conf-base + heap +
  assumes hconf empty-heap
  and \[[\text{alloc } h hT = (h', ao); hconf h; is-htype P hT]\] \implies hconf h'
  and \[[\text{write } h a al v h'; hconf h; P, h + a-ul : T; P, h + v :\leq T]\] \implies hconf h'
  and \[[\text{typeof-addr } h a = [hT]; hconf h]\] \implies is-htype P hT

locale conf-progress = conf +
  assumes \[[hconf h; P, h + a-ul : T]\] \implies \exists v. \text{read } h a al v
  and \[[hconf h; P, h + a-ul : T; P, h + v :\leq T]\] \implies \exists h'. \text{write } h a al v h'

locale conf-read = conf +
  assumes \[[\text{read } h a al v; hconf h; P, h + a-ul : T]\] \implies P, h + v :\leq T

locale typesafe = conf-progress + conf-read

Figure 4.2: Locales for heap conformance

notion of the type of a location. Let \(P, h + a-ul : T\) denote that the location \((a, al)\) is supposed to store values that conform to type \(T\). Formally:

\[
\text{typeof-addr } h a = [hT] \quad P + \text{class-of' } hT \text{ has } F : T (fm) \text{ in } D
\]

\[
P, h + a \cdot \text{Field } F D : T
\]

\[
\text{typeof-addr } h a = [\text{ArrayT } T n'] \quad n < n'
\]

\[
P, h + a \cdot \text{Cell } n : T
\]

Now, consider the locales in Figure 4.2 in detail. Locale conf-base fixes the predicate hconf for heap conformance as parameter, which abstracts heap conformance \(\_ + \_\sqrt{\_}\) from §2.1.6. Locale conf assumes that the empty heap conforms and all heap-manipulating operations preserve heap conformance – if only valid types are allocated and type-conforming values written. Moreover, heap conformance must ensure that the dynamic types of all addresses are valid, too.

Following the division in progress and preservation, the actual assumptions for type safety are split in two groups. Locale conf-progress
Chapter 4. Memory models

collects the progress assumptions, i.e., for a conformant heap and valid member of an address, we must always be able to read some value from that member and write any type-conforming value to it. Conversely, subject reduction requires that reading from memory always returns type-conforming values, expressed in locale \textit{conf-read}. Locale \textit{typesafe} combines all these assumptions. This completes the specification of the heap module.

4.1.2 Adaptations to semantics and proofs

Before I discuss the design considerations in §4.1.3, I present how to adapt the definitions, theorems, and proofs in Chapter 3 to the heap module.

The language definitions are not affected at all, because the heap module only affects the dynamic semantics. Neither is the framework for interleaving semantics, since it is oblivious of the heap representation. The instantiations themselves for source code and bytecode differ slightly, as the type variable for the heap \( h \) is specialised to the type variable \( \text{'heap} \) instead of the concrete type \( \text{heap} \).

The abstract heap necessitates many syntactic adaptations in the specification of and proofs about the single-threaded semantics. Although locales are invaluable in syntactically hiding the heap operations, some details still change. For example, heap conformance no longer depends on the program \( P \) inside the locale, because \( P \) is one of its parameters and it would be nonsensical to clutter the formalisation with unnecessary applications of heap conformance to \( P \). Accordingly, its syntax changes from \( P \vdash h \checkmark \) to \( h\text{conf} \). Nevertheless, the definitions and theorems as presented in Chapter 3 are still machine-checked – to obtain them, one merely instantiates the generalised versions from this chapter with sequential consistency as defined in §4.2. I omit to list these generalisations as they are straightforward.

The definitions from Chapter 3 that depend on the heap are distributed over the locales \textit{heap-base} and \textit{conf-base} as follows:\textsuperscript{24} Definitions that involve heap conformance go into \textit{conf-base}, whereas \textit{heap-base} collects

\textsuperscript{24}Language-specific proof invariants (such as run-time well-typedness and bytecode conformance) demand specialised types for program declarations (such as \('addr J-prog\) and \('addr jvm-prog\), respectively, instead of \('m prog\)). Since the type variable \('m\) of the locale parameter \( P \) cannot be instantiated inside the locale due to HOL restrictions, I replicate the locale hierarchy from Figures 4.1 and 4.2 and constrain \('m\) as necessary. As this is only a technical issue, I do not distinguish these copies in the presentation.
4.1. The heap as a module

<table>
<thead>
<tr>
<th>theorem</th>
<th>description</th>
<th>locale</th>
</tr>
</thead>
<tbody>
<tr>
<td>L 3.6</td>
<td>assumptions of <em>multithreaded</em></td>
<td>heap-base</td>
</tr>
<tr>
<td>T 3.3</td>
<td>type safety</td>
<td>typesafe</td>
</tr>
<tr>
<td>T 3.4</td>
<td>subject reduction</td>
<td>conf-read</td>
</tr>
<tr>
<td>L 3.12</td>
<td>heap extension</td>
<td>heap</td>
</tr>
<tr>
<td>L 3.13</td>
<td>heap extension monotonicity</td>
<td>heap</td>
</tr>
<tr>
<td>L 3.14(i)</td>
<td>preservation of definite assignment</td>
<td>heap-base</td>
</tr>
<tr>
<td>L 3.14(ii)</td>
<td>preservation of conformance</td>
<td>conf-read</td>
</tr>
<tr>
<td>L 3.16</td>
<td>preservation of <em>ok</em>-I</td>
<td>heap-base</td>
</tr>
<tr>
<td>L 3.17</td>
<td>preservation of lock conformance</td>
<td>heap-base</td>
</tr>
<tr>
<td>L 3.19</td>
<td>assumptions of <em>progress</em></td>
<td>typesafe</td>
</tr>
<tr>
<td>C 3.1</td>
<td>multithreaded progress</td>
<td>typesafe</td>
</tr>
<tr>
<td>L 3.20</td>
<td>well-formedness of the initial state</td>
<td>conf</td>
</tr>
<tr>
<td>L 3.21</td>
<td>assumptions of <em>preserve-deadlocked</em></td>
<td>typesafe</td>
</tr>
<tr>
<td>L 3.22</td>
<td>preservation of reductions</td>
<td>conf-progress</td>
</tr>
<tr>
<td>T 3.5</td>
<td>deadlock preservation</td>
<td>typesafe</td>
</tr>
<tr>
<td>L 3.7</td>
<td>assumptions of <em>multithreaded</em></td>
<td>heap-base</td>
</tr>
<tr>
<td>T 3.6</td>
<td>type safety</td>
<td>typesafe</td>
</tr>
<tr>
<td>L 3.23</td>
<td>aggressive &amp; defensive VM commute</td>
<td>conf-base + heap</td>
</tr>
<tr>
<td>L 3.24</td>
<td>single-threaded progress</td>
<td>conf-progress</td>
</tr>
<tr>
<td>L 3.25</td>
<td>preservation of conformance</td>
<td>conf-read</td>
</tr>
<tr>
<td>L 3.26</td>
<td>preservation of <em>jvm-wf-states</em></td>
<td>conf-read</td>
</tr>
<tr>
<td>L 3.27</td>
<td>assumptions of <em>progress</em></td>
<td>typesafe</td>
</tr>
<tr>
<td>L 3.28</td>
<td>well-formedness of the initial state</td>
<td>conf</td>
</tr>
<tr>
<td>L 3.29</td>
<td>assumptions of <em>preserve-deadlocked</em></td>
<td>typesafe</td>
</tr>
<tr>
<td>L 3.30</td>
<td>preservation of execution options</td>
<td>conf-progress</td>
</tr>
<tr>
<td>T 3.7</td>
<td>deadlock preservation</td>
<td>typesafe</td>
</tr>
</tbody>
</table>

"a(Locale progress assumes preservation of *wf-states*, which requires *conf-read*. Therefore, L 3.19 and L 3.27 are in the stronger locale typesafe. For the progress assumptions *progress* and *wf-red*, the assumptions of locale *conf-progress* suffice.

"b(L 3.23 requires Φ ⊢ t:sv, which is defined in *conf-base* because it imposes heap conformance. However, the proof does not depend on heap conformance, but only requires the heap assumptions. Therefore, it lives in a locale of its own that combines heap with *conf-base*."

**Table 4.1:** Distribution of lemmata (L), theorems (T), and corollaries (C) from §3.2, §3.3.4, and §3.3.5 over the locales from Figures 4.1 and 4.2
the others that only depend on the heap and its operations. The latter are the vast majority and include all semantics definitions. The former consist only of proof invariants; they no longer take the program $P$ as an explicit parameter, because the locale already fixes it. For example, bytecode conformance is now written as $\Phi \vdash t:s\sqrt{\_}$ instead of $P, \Phi \vdash t:s\sqrt{\_}$.

For the theorems, the distribution is more complicated. Table 4.1 assigns the lemmata and theorems from §3.2, §3.3.4, and §3.3.5 to the locales according to the following rules: Theorems whose preconditions do not depend on heap conformance are provable under the assumptions of locale `heap`. Locale `heap-base` collects those that do not depend on the heap at all. Single-reduction preservation lemmata that require heap conformance go into `conf` – except for subject reduction, which requires `conf-read`. Single-reduction progress theorems are provable in `conf-progress`. Combinations of subject reduction and progress such as the type safety theorems live in `typesafe`. Accordingly, the proofs of these lemmata and theorems are adapted to use the assumptions of the respective locales.

Beyond these easy changes in syntax, there are also the following non-trivial adaptations. First, I construct the start heap `start-heap` with the preallocated `Thread` object and system exceptions from the primitive heap operations as follows: The constant `start-data :: 'heap × 'addr list × bool` preallocates the start heap using `alloc`, Figure 4.3 shows the definition. It consists of the start heap `start-heap`, the list of preallocated addresses `start-addrs` and the flag `start-ok` to indicate whether all allocations have succeeded. Then, the start thread’s ID `start-tID` is the thread ID associated with the preallocated `Thread` object, and `addr-of-sys-xcpt C` retrieves the right address for preallocated system exceptions from `start-addrs`. For the start state being well-formed, all allocations must have succeeded.

Second, the semantics rules for accessing members of objects and arrays now use the abstract heap operations. Figures 4.4 and 4.5 show the semantics of reading and updating a field of an object or array for source code and bytecode, respectively. Note that the thread action records the reading and writing. The semantics for array cells are similar and can be found in the Appendices B.6.5 and B.7.3.

Third, I must change the implementations for the native methods `hashcode` and `clone` (Figure 2.12). Note that the other native methods (Figure 3.16) do not manipulate the heap and can thus be left untouched (Figures 3.17, 3.18, and 3.19). For `hashcode`, I use Isabelle’s type classes to safely overload the function `hash-addr`. 

138
4.1. The heap as a module

\[
\text{start-data} = (\text{let init } (h, as, b) C = \begin{cases} \text{if } b \text{ then} \\
\text{let } (h', ao) = \text{alloc } h \text{ (ClassT } C) \\
\text{in case } ao \text{ of None } \Rightarrow (h', as, \text{False}) \\
\mid [a] \Rightarrow (h', as @ [a], \text{True}) \\
\text{else } (h, as, \text{False}) \\
\text{in foldl init (empty-heap, [], True) (Thread ∙ sys-xtcpts)})
\]

\[
\text{(start-heap, start-addrs, start-ok) = start-data}
\]

\[
\text{start-tID} = a2t (\text{hd start-addrs})
\]

\[
\text{addr-of-sys-xcpt} = \text{the } \circ \text{map-of (zip (Thread ∙ sys-xtcpts) start-addrs)}
\]

\[
\text{wf-start } P C M \ vs \leftrightarrow \\
(\exists Ts T \text{ meth } D. P ⊢ C \text{ sees } M:T s→ T = [\text{meth}] \text{ in } D \land \\
P, \text{start-heap} ⊢ vs \left[ \leq \right]Ts \land \text{start-ok})
\]

**Figure 4.3:** The bootstrap process constructs the start heap

\[
\begin{align*}
\text{R_{FACC}}: & \quad \quad \text{read } h a (\text{Field } F D) v \\
& \quad P, t ⊢ \langle \text{addr } a.F[D], (h, xs) \rangle \\
& \quad \rightarrow \langle \text{Read } a (\text{Field } F D) v \rangle \rightarrow \langle \text{Val } v, (h, xs) \rangle
\end{align*}
\]

\[
\begin{align*}
\text{R_{FASS}}: & \quad \quad \text{write } h a (\text{Field } F D) v h' \\
& \quad P, t ⊢ \langle \text{addr } a.F[D] := \text{Val } v, (h, xs) \rangle \\
& \quad \rightarrow \langle \text{Write } a (\text{Field } F D) v \rangle \rightarrow \langle \text{unit}, (h', xs) \rangle
\end{align*}
\]

**Figure 4.4:** Semantics of field access and field assignment

The former implementation of clone copies the complete object to a fresh address (Clone) in one step. This is no longer feasible, because the heap operations allow to access only one member at a time. Therefore, I define a copy operation copy-mems \( a a' als h obs h' \) that copies one by one a list of members \( als \) from address \( a \) and heap \( h \) to address \( a' \). The resulting heap is \( h' \). Like for ordinary field access, \( obs \) records the memory accesses. Then, based on the receiver’s type, the rules for clone (i) allocate a new object or array according to the receiver’s type, (ii) compute the list of members to copy from the program declaration, and (iii) copy them using copy-mems. The lengthy formal definition can be found in Appendix B.4.2.
exec-instr (Getfield F D) P t h stk loc C M pc frs =
(let v · stk' = stk; Addr a = v
in if v = Null then
  { (∥), [addr-of-sys-xcpt NullPointer], h, (stk, loc, C, M, pc) · frs}
else
  { (∥ (Read a (Field F D) v'), None, h, (v' · stk', loc, C, M, pc + 1) · frs)
    | v'. read h a (Field F D) v' }
)

exec-instr (Putfield F D) P t h stk loc C M pc frs =
(let v · r · stk' = stk; Addr a = r
in if r = Null then
  { (∥), [addr-of-sys-xcpt NullPointer], h, (stk, loc, C, M, pc) · frs}
else
  { (∥ (Write a (Field F D) v)), None, h', (stk', loc, C, M, pc + 1) · frs)
    | h'. write h a (Field F D) v h' }
)

Figure 4.5: Semantics of Getfield and Putfield

4.1.3 Design considerations

Like JinjaThreads as a whole reuses as many concepts of Jinja as possible, the heap module attempts to naturally abstract from the concrete heap representation. Jinja has already defined abstractions over the heap representation such as typeof_h; the heap module carries this one step further. First, it defines a kernel set of operations that permit to completely separate the semantics from the heap implementation. Then, it re-implements the existing abstractions in terms of these operations. When this was not possible, e.g., for heap conformance, I turned the abstraction into a parameter of the model.

It was more difficult to identify usable assumptions about the operations on which the proofs rest. To do so, I started to convert representation-dependent pieces of the main theorems for source code (progress, preservation, subject reduction) to use the heap module. Thereby, I examined the facts known in the proof context, which led me to the above assumptions. On the level of values, they are deliberately weak to allow for different memory models. For example, allocation must only return some address which has the correct type in the updated heap. However, there is no constraint that the address must be fresh or that it
has not been typable in the original heap. Similarly, reading from and writing to the heap need only be defined for conformant heaps.

In contrast to values, JinjaThreads assumes that heap operations never mess up type information, i.e., the heap is only ever extended and typable addresses never change their type. Some proofs, e.g., the compiler verification in Chapter 5, rely on this even for non-conforming heaps. Therefore, the assumptions in heap must hold for any heap, not just conforming ones. Such a distinction can also be found in the specification of the JMM: The JMM guarantees only weak semantics for values, but requires sequential consistency for types.

The locale hierarchy mirrors JinjaThreads’ proof structure. Distributing the assumptions over different locales has two advantages over combining them in a single one. First, as Table 4.1 shows, I can identify which assumptions a theorem really depends on. Second, one of the implementations for the JMM (§4.3.2) does not satisfy the locale conf-read – and therefore neither typesafe. Thanks to the modularity, I can still use all theorems in conf-progress and its ancestor theories for proofs about the JMM.

Finally, let me motivate why reading and writing are non-deterministic, but allocation is not. Since the JMM abstracts from implementation details like caches in processors and optimisations in compilers, non-deterministic reading models their effects. Writing is also non-deterministic for reasons of symmetry and because other MM formalisations [34] use non-deterministic writing instead of reading. However, I model allocation as a function such that bootstrapping can functionally construct the start heap from the empty heap (see Figure 4.3). If allocation was non-deterministic, too, I could not have fully specified the initial heap, which would complicate code generation. Moreover, a functional specification does not need progress assumptions like in locale conf-progress. Nevertheless, there are no fundamental obstacles to non-deterministic allocation and it would have elegantly solved other issues, see §4.3.6.

4.2 Sequential consistency

Under sequential consistency [93], a multithreaded program behaves as if every thread executes at a time and all threads immediately see all the writes of a thread. Interleaving semantics as presented in Chap-
Chapter 4. Memory models

Chapter 4. Memory models

ter 3 captures sequential consistency. In this section, I introduce the implementation of the heap module from §4.1 for sequential consistency.

To distinguish types and definitions from the abstract ones, I prefix their names and decorate the notation with “sc”, i.e., sc-heap for heap, sc-read for read, _ _ ⊢sc _ ≤ _ for _ _ ⊢ _ ≤ _ , etc. I use the same scheme to refer to definitions in the locales heap-base and conf-base that the interpretation specialises.

For the purpose of code generation (§6.3.1), I use natural numbers as addresses. Then, the heap representation is defined as in Figure 2.11 and sc-typeof-addr like typeof-addr in Equation 2.4. The implementation of the abstract operations follows Jinja’s heap model, I have only extracted them from different parts of the semantics. The interesting operations are reading, writing and heap conformance: sc-read h a al v holds iff in the heap h, member al of address a stores the value v. Similarly, sc-write h a al v h’ requires that a is allocated in h and h’ is h where a’s member al is updated to v. The heap h conforms (written P ⊢sc h √) iff for all allocated objects and arrays, their type T is valid, the field map stores type-conforming values for all fields that P declares for T , and – for arrays – all array cells store type-conforming values. The formal definitions of all operations can be found in Appendix B.3.1.

Theorem 4.1. Sequential consistency satisfies all assumptions of the heap module for any input program.

```
typesafe sc-a2t sc-t2a sc-typeof-addr sc-empty-heap
(sc-allocate P) sc-read sc-write (P ⊢sc _ √) P
```

Corollary 4.1. JinjaThreads source code and bytecode are type safe under sequential consistency.

4.3 Java memory model

In this section, I first motivate the JMM and explain its central concepts (§4.3.1), in more detail than in §1.1. Next, I formalise the JMM and connect it to the JinjaThreads semantics (§4.3.2). Then, I prove modularly that it provides the DRF guarantee (§4.3.3) and is consistent (§4.3.4). Further, I show that JMM only provides a weaker form of type safety and compromises security (§4.3.5). Finally, I discuss the formalisation
and the proofs (§4.3.6). A preliminary version of this section has been published in [112].

4.3.1 Informal explanation

In §1.1, I have briefly motivated the need for the JMM already. Now, I delve deeper into how the JMM works.

Since the JMM must ensure that compilers can implement Java on a variety of hardware with different MMs efficiently, it reduces concrete thread operations to the events from Table 3.1, which are called inter-thread actions in JMM terminology:

- reading (Read) from, writing (Write) to and initialising (Allocate) locations on the heap – with the abbreviations New-Obj and New-Arr,
- locking (SLock) and unlocking (SUnlock) a monitor,
- interrupting (TIntr) a thread and observing that it has been interrupted (TIntrd),
- spawning (TStart) of and joining (TJoin) on a thread, and
- external actions (Extern) – for I/O, for example.

Additionally, the JMM requires that events TInit and TFinish mark the start and termination of a thread, respectively.

This way, the JMM is independent from syntax and implementation techniques. It nevertheless gets a global view on how a given program works algorithmically and on how its threads interact, and uses this to determine the set of legal behaviours. Note that the JMM falls in two parts: First, the JMM provides the strong model of sequential consistency for allocation, type information, and array lengths. In JinjaThreads, the JMM heap module implementations deal with them. Second, axiomatic rules determine a posteriori whether a given execution is an allowed (legal) behaviour of a given program. I model this part as additional layers on top of the interleaving semantics.

25Technically, the JMM defines an initialisation action that initialises only a single location. JinjaThreads uses one event per memory allocation that initialises all members of the allocated address. This way, allocation events keep track of allocated addresses whereas JMM initialisation actions would not if the allocated object or array contains no members, e.g., an array of length 0. Note that the special treatment of allocations in the JMM (see below) ensures that this deviation does not matter semantically.

26The JMM does not mention the events TIntr and TIntrd for thread interruption although its specification requires them.
Chapter 4. Memory models

Figure 4.6: Happens-before provides a notion of time relative to a given event \(\alpha\). If \(\alpha\) is a Read, it may see Write events in the grey area.

In such an execution, the events of a single thread are totally ordered by the sequence in which they would occur according to the single-threaded semantics, the so-called program order (notation \(\leq_{po}\)). Being consistent with this total order, the happens-before order \(\leq_{hb}\) provides a notion of time relative to a given event. As Figure 4.6 illustrates, it partitions the other events of the execution into three groups: those that must have happened before it, those that must happen after it, and those that may happen concurrently. Synchronisation events, which are all events except for external actions and reads from and writes to non-volatile locations, introduce happens-before relationships between events of different threads. Hence, whenever \(\alpha\)’s thread cannot deduce – using only allowed means of synchronisation – that an event \(\beta\) of another thread must have happened before or will happen after \(\alpha\), then \(\alpha\) and \(\beta\) may happen concurrently. This permits compilers and hardware to freely reorder independent statements of a thread without synchronisation in between.

Since the JMM is independent from a concrete language and sequential semantics, it is custom to write examples such as in Figure 4.7a in a simple imperative language rather than to obfuscate the point by irrelevant Java details. In this language, thread-local variables start with “r”, e.g., \(r1, r2\), whereas \(x, y\), etc. denote shared locations. In examples, vertical rules separate the threads, and the thread in column \(i\) has ID \(t_i\). Above the threads, the initial values of shared locations and any necessary declarations are given. In Figure 4.15 below, I show how to translate the above example to Java.

Sequential consistency does not allow the result \(r1 == 2\) and \(r2 == 1\) for Figure 4.7a, but the JMM does, because multi-core processors allow it, too. From a single-threaded perspective, l. 1 and l. 2 in the thread on the left are independent. Hence, one core might execute the write in l. 2 while l. 1 is still waiting for the memory controller to answer its request for \(x\). While such an early execution is transparent to sequential
4.3. Java memory model

Initially: \( x = y = 0 \);

1: \( r_1 = x; \) \\
2: \( y = 1; \) \\
3: \( r_2 = y; \) \\
4: \( x = 2; \)

(a)

1: \((t_1, \text{Read } x)\) \\
2: \((t_1, \text{Write } y)\) \\
3: \((t_2, \text{Read } y)\) \\
4: \((t_2, \text{Write } x)\)

(b)

Figure 4.7: Example program with data races [115, Fig. 1] (a) and its JMM execution for the result \( r_1 == 2, r_2 == 1 \) (b)

Initially: \( x = y = 0 \);

1: \( r_1 = x; \) \\
2: \( y = r_1; \) \\
3: \( r_2 = y; \) \\
4: \( x = r_2; \)

1: \((t_1, \text{Read } x)\) \\
2: \((t_1, \text{Write } y)\) \\
3: \((t_2, \text{Read } y)\) \\
4: \((t_2, \text{Write } x)\)

Figure 4.8: Example of the value 42 appearing out of thin air [115, Fig. 2]

In programs, the thread on the right can pierce the veil. In this situation, the second core reads 1 for \( y \) in l. 3 from their shared cache and then writes to \( x \). If memory has not yet returned \( x \)'s value, the first core uses the new value 2 instead – and the unexpected result becomes real.

Figure 4.7b shows how executions are depicted. The threads are abstracted to events – labelled with the thread ID – and orders. Solid arrows represent program order, transitive relationships are not shown. Dotted arrows used in later examples denote synchronisation (synchronises-with relationships, see §4.3.2 for the formal definition). The dashed arrows denote the flow of values from writes to reads; an execution assigns to each read event the write event it sees. The JMM requires that a read sees a write that happens before or may happen concurrently (grey area in Figure 4.6), i.e., the write must not happen after the read.

The execution shown in Figure 4.7b results in \( r_1 == 2 \) and \( r_2 == 1 \). As there is no synchronisation, happens-before coincides with program order. Hence, l. 1 and l. 2 may happen “concurrently” with l. 3 and l. 4. Therefore, l. 1 and l. 3 are allowed to see the writes from l. 4 and l. 2, respectively. In particular, the thread on the left is not allowed to deduce that l. 3 must have already executed from the fact that l. 1 reads the value 2 from l. 4, because there is no synchronisation involved.

If only happens-before constrains visibility of write events, values may appear out of thin air. Consider, e.g., the program and its execution
in Figure 4.8, adapted from [115, Fig. 2]. The reads in ll. 1 and 3 may see the writes in ll. 4 and 2, respectively, as they may happen concurrently. If both writes write 42, both reads may read 42, although the program cannot normally produce 42. Hence, 42 appears out of thin air, as the happens-before constraints do not forbid this.

For type safety and security guarantees, it is vital that values do not appear out of thin air [145]. Otherwise, malicious code could exploit this to forge a pointer to an object to which it must not gain access or which it can then access in a type-unsafe fashion. To preclude this, the JMM adds a causality condition: Reads that see concurrent writes must be committed, i.e., there must be a justifying execution that writes the same value, but the read event sees a write that happens before it. This causality condition distinguishes the JMM from memory models of other languages like ADA, where concurrent reads and writes immediately result in the behaviour being undefined. In the above example, causality forbids $r_1 == 42$ because no execution can produce the value 42 without having both reads see the concurrent writes. The important thing to note is that at the basis of any sequence of justifying executions, there is one in which all reads see writes that happen before them.

This is where initialisations come into play. The JMM assumes that all locations are initialised to their default value at the start of the execution. By definition, these initialisations happen before any other event. Thus, there is always at least one suitable write that happens before a given read, which ensures that a basis for justifying executions exists. Remember that program-order is consistent with happens-before. Hence, initialisations precede all other events of the same thread in program order, too, although they correspond to allocations that occur in the midst of an execution. Figure 4.9b shows an example. Thread $t_1$ generates the events as shown from top to bottom, but the allocation in l. 3 precedes all others in program order and happens before order.

The requirement that a heap location is conceptually initialised at the start (instead of when it is allocated) has been one of the main complications in the proofs – which previous formalisations have omitted [8, 69]. Since initialisation events originate from dynamic allocation, I must consider complete executions, which may be infinite, instead of finite prefixes – at least for the single-threaded semantics. Consider, e.g., the program and one of its executions in Figure 4.9. Remember from above that the initialisation for the field $f$ of the object created in l. 3 at location $a$ happens before all other events, although the single-
class A { field f:int; } initially: x = y = null;

1: r1 = x;
2: if (r1 != null) r2 = r1.f;
3: r3 = new A();
4: y = r3;
5: r4 = y;
6: x = r4;

(a)

(b)

Figure 4.9: Program with an execution in which the read in l. 2 sees the initialisation from l. 3, which occurs later in the program text.

threaded semantics executes it after ll. 1 and 2. Now take the prefix of this execution up to l. 2. If (t1, New-Obj A A) was not part of the prefix, the prefix would be ill-formed because l. 2 sees no write. Hence, prefixes must include the initialisation events. As the single-threaded semantics produces initialisation events only at allocations, I must run the program to completion, because it is undecidable at intermediate states whether all initialisation events have been collected. Thus, the formalisation must deal with infinite executions.

The technical complexity of the JMM makes it very hard to decide what the legal behaviours of a program are. Numerous subtleties, some of which the examples in this chapter illustrate, and the fact that legality can only be decided for whole programs complicate this further. Consequently, one cannot expect the average Java programmer to reason...
about the JMM. The DRF guarantee relieves the average programmer from this burden. If all thread communication is explicit in the program, Java maintains the illusion of sequential consistency. I prove that the JMM does provide the DRF guarantee in §4.3.3.

### 4.3.2 Formal definition

The JMM formalisation extends the stack of semantics from Figure 3.1 as shown in Figure 4.10; there are four new layers (shaded). In the single-threaded semantics, the JMM implements the heap module in two different ways (layer 1) and inserts the events for thread start and finish (layer 4). On top of the stack, I define complete interleavings of a program (layer 6) from which the axiomatic JMM (layer 7) selects the legal executions. The JMM also introduces a native method for printing on layer 2, but does not affect the sequential semantics (layer 3) or the interleaving semantics themselves (layer 5). The rest of this section traverses the stack from bottom to top and presents the new layers and changes, respectively.

#### The heap module: JMM implementation 1

An implementation of the heap module must define operations for values and types, but the weak semantics of the JMM only applies to values.
Recall that checked type casts, virtual method calls, and reading the length of an array are not part of the events. According to the JLS, reading types and array lengths must always return the correct data [56, §17.4.5].

Hence, type information also needs a “memory model”.

In this implementation, I adopt sequential consistency for types, i.e., allocation determines the type information of the allocated address and distributes this information immediately to all threads. Hence, on the level of types, this approach is the same as in §4.2. Although being straightforward, it suffers from some drawbacks that I discuss below.

Technically, the heap $h$ reduces to a map from addresses to their type information, i.e., $h :: ':addr \rightarrow hty$. Then, an address is fresh in the heap $h$ iff $h$ contains no type information for it. Since type information grows monotonically (as JinjaThreads does not model garbage collection), a fresh address has not been allocated before and will never be allocated once more afterwards.

With this heap implementation in mind, the module implementation is then straightforward: Thread IDs are identified with addresses, the empty heap is the empty map $empty$, allocation picks an arbitrary fresh address, and $jmm$-typeof-addr is map lookup. Any value can be read at any time, i.e., $jmm$-read is constantly True; writing does not change the heap. The heap conforms iff all allocated addresses have valid types. The formal definitions can be found in Appendix B.3.2.

Like “sc” for sequential consistency, I prefix or decorate the declarations in the heap locales with “jmm” to refer to their JMM instantiation.

**Lemma 4.1.** The JMM heap implementation 1 satisfies the assumptions of the locale conf-progress and its ancestors.

$\text{conf-progress jmm-a2t jmm-t2a jmm-typeof-addr jmm-empty-heap}$

$jmm$-allocate $jmm$-read $jmm$-write $(P \vDash_{jmm} \sqrt{\_}) P$

Although this heap implementation is simple and straightforward, it does not satisfy the assumption of locale conf-read, because $jmm$-read allows to read arbitrary values, even non-type-conforming ones. Hence, subject reduction fails. The DRF guarantee shows that correctly synchronised programs, i.e., without data races, are type safe (§4.3.3), but correct synchronisation is undecidable. Programs with data races, however,

---

27 Although the JLS specifies that every array has a final field length [56, §6.4.5] that stores its length, the JMM treats array lengths specially [56, §17.4.5].
may have type-unsafe executions of two kinds. Both exploit that the type of an address is only determined upon allocation, but not when it is first used.

First, when the type of an address is not yet known, the source code semantics or defensive VM get stuck whereas the aggressive VM behaves in an undefined way. For example, the JMM allows that l. 1 in Figure 4.11 reads the address $a$ of the object allocated in l. 3 via the detour of the second thread, because an optimising compiler might move l. 3 before l. 1. However, the semantics does not anticipate such optimisations, but executes the program as it is. Hence, when l. 2 executes the call to $f$ on $a$, the source code semantics gets stuck, because $a$’s type is still undefined (i.e., $typeof-addr \, h \, a = None$). So does the defensive VM, and the aggressive VM calls an unspecified method.

Second, reading an address before its type is determined may also compromise subject reduction. The program in Figure 4.12 has a legal execution where $x$, $y$ and $r2$ reference an $Integer$ object, although the program is type correct if it declares them of type $String$. For a detailed derivation of this execution, see §4.3.5. This is an artefact of the JMM as I am not aware of any optimisation that would produce such a result. In this example, the problem is again that l. 3 reads an address $a$ that has not yet been allocated. Then, l. 4 non-deterministically chooses the type of the object to allocate – note that $b$ can be either $false$ (initial value) or $true$ (assignment in l. 9). Since the object has already been stored in $r2$ of type $String$, the second alternative results in a non-conforming state, i.e., type safe is broken.

A temptingly simple measure would be to restrict reading such that only allocated addresses may be read from memory. However, the semantics then would miss some legal JMM behaviours, because this restriction prohibits reordering with memory allocations. For example, reconsider the program in Figure 4.11 with l. 2 removed. The JMM
4.3. Java memory model

|   | 1: r1 = y; | 2: x = r1; | 3: r2 = x; | 4: if (b) | 5: r3 = new Integer(); | 6: else | 7: r2 = new String(); | 8: y = r2; | 9: b = true; |
|---|---|---|---|---|---|---|---|---|
|   | initially: b = false; x = y = null; |

Figure 4.12: A program with a legal execution where r2 of type String references an Integer object

allows r1 == y at the end of the modified program, because compilers are allowed to reorder the independent statements in ll. 1 and 3. However, the semantics could not produce this result because the read in l. 1 always happens before the allocation in l. 3, i.e., it could not return the address to be allocated.

The heap module: JMM implementation 2

To address the above type safety problems, I define the following alternative implementation of the heap module – see Appendix B.3.2 for the formal definitions. It is motivated by the insight that the core of the above problems is that the type of an address is only determined upon allocation, but not when it is first used. This model exposes the communication channel of type information between threads to the memory model while the previous one hides it (see below for an example).

Now, it is the address and not the heap that stores type information for the address. Hence, an address consists of its type information (HOL type hty) and a sequence number to distinguish objects of the same type.

```
datatype addr = Address hty nat
```

The type and array length of an address is the information stored in the address – provided that it refers to a valid type. In particular, type information for every address is available from the start. Hence, the

---

28This restriction is necessary for the last assumption of locale conf. Since I want typeof-addr to not depend on the heap, typeof-addr may only assign types to addresses that refer to a valid type. Note that this is possible as alloc-type of locale heap is restricted to valid types.
programs in Figures 4.11 and 4.12 are unproblematic, because with this implementation,

(i) the defensive checks only ensure type correctness, but no longer test on the referenced object having been allocated, and
(ii) the allocations in ll. 5 and 7 always return different addresses.

Now that I have stripped type information off the shared heap, it only needs to remember which addresses are fresh for allocation. Hence, the heap (type \textit{jmm}'-heap) maps type information (type \textit{hty}) to the number of objects of that type that have already been allocated. Allocation then merely increments that number and uses the former value as sequence number for the new address. All heaps conform. Moreover, I also restrict the read operation to require that the value must be type-conforming. This way, this implementation satisfies all assumptions of the heap module. To distinguish it from the previous implementation, I call this implementation JMM' and use “jmm'” as prefix and decoration of the operations and derived definitions.

\textbf{Theorem 4.2.} The JMM heap implementation 2 satisfies all assumptions of the heap module for any input program.

\[\begin{align*}
\text{typesafe jmm}'-a2t & \quad \text{jmm}'-t2a \quad \text{jmm}'-\text{typeof-addr} \quad \text{jmm}'-\text{empty-heap} \\
\text{jmm}'-\text{read} & \quad \text{jmm}'-\text{write} \quad \text{jmm}'-\text{hconf} \quad P
\end{align*}\]

The JMM' heap implementation suffers from two disadvantages compared to the first one. First, since type information partitions the address space, each read or write of an address value not only transfers a pointer value as on standard hardware, but simultaneously does so for the complete run-time type information of the object it references. From an implementation point of view, this is unrealistic.

Second, restricting the read operation to type correct values effectively makes the heap implementation type safe “by definition.” Unlike for JMM heap implementation 1, I conjecture that this restriction does not exclude any legal JMM behaviour. The causality constraints should ensure that an execution which reads non-type-conforming values has no sequence of justifying executions. I have not attempted a formal proof yet, but if this is shown in the future, then the restriction may be dropped.

Let me now demonstrate how this heap implementation exposes the hidden communication channel of type information. Consider the
4.3. Java memory model

Initially:
\[ x = 0; \quad y = \text{null}; \]

1: \( r_1 = x; \)
2: \( r_2 = (r_1 == 0 ? \text{new } A() : \text{new } B()); \)
3: \( y = r_2; \)
4: \( x = 1; \)
5: \( r_3 = y; \)
6: \( r_4 = r_3.f(); \)

Figure 4.13: Example for implicit communication via type information

The program in Figure 4.13. Suppose that classes \( A \) and \( B \) inherit from some class \( C \) which declares a method \( f() \). Under implementation 1, the allocation in l. 2 returns the same address value, no matter whether \( A \) or \( B \) is allocated. Still, dynamic dispatch at l. 6 tells the thread on the right about the left thread’s local variable \( r_1 \). However, from the point of view of events, the thread on the right only reads an address (in fact the same value in both cases), but behaves differently. Heap implementation 2 allocates \( A \)’s objects at different addresses than \( B \)’s. Hence, the value that l. 5 reads completely determines the call target in l. 6. Analogously, threads can communicate through array lengths instead of types, see Figure 4.30 for an example. This is why implementation 2 allocates arrays of different lengths at distinct addresses.

External actions as observable behaviour

For the JMM, an observable behaviour of a program execution consists of a subset of the external actions that is consistent with happens before order and synchronisation order [56, §17.4.9]. This models what the user or other programs may observe without looking at the internal state of the JVM. However, the single-threaded semantics for source code and bytecode do not yet generate external actions. To remedy this, I add a native method \( \text{print} \) to class \( \text{Object} \) that generates intermediate output.

\[
\text{Object.print([Integer]) :: Void}
\]

\[
P, t \vdash \langle a.\text{print}(\text{vs}), h \rangle \rightarrow [\text{Extern a print vs Unit}] \rightarrow_{nc} \langle \text{Ret-Val unit}, h \rangle \quad (4.1)
\]

to attach observable behaviour to the examples of this chapter, one assumes that all variables are output at the end of the program.

Intermediate output is only relevant for constraining the semantics of non-terminating executions, because in case of termination, one
could augment the final state with observations of intermediate states. However, there is no such final state in non-terminating ones.

**Associating statements and instructions with events**

The JMM determines which values a read may return by assigning writes to reads. Synchronisation can restrict (and even enlarge [2, 115]) this set of possible values. For that, the JinjaThreads semantics produce events that record memory accesses and synchronisation. Now, I return to the semantics of native methods and source code from §3.2.1 and §3.2.2, respectively, to explain the events that they generate. The bytecode instructions generate the same events as their source code equivalents.

First, reconsider the native methods of class `Thread` from Figure 3.17. The generated events follow exactly the JMM specification [56]. Successfully spawning a thread generates the appropriate `TStart` event. However, no event occurs when spawning fails, because only successful spawns synchronise with the first action of the started thread [56, §17.4.4].

Similarly, interrupting a thread (Intr) synchronises with the observation that `t` has been interrupted (JoinIntr, isIntrdT, IntrdT, WaitIntrd1, WaitIntrd2), i.e., Intr produces the event `TIntr t`, the latter rules generate `TIntrd t`. Asymmetrically, clearing `t`'s interrupt status (JoinIntr, IntrdT, WaitIntrd1, WaitIntrd2), does not synchronise with the observation that `t` is not interrupted (isIntrdF, IntrdT, Wait), i.e., no event records that. Also, interrupting a thread that has not yet been started has no effect (IntrInex).

A call to `wait` unlocks the monitor and reacquires the locks after notification. Hence, `Wait` (Figure 3.19) generates the event `SUnlock`. Remember that the interleaving semantics already ensures mutual exclusion via thread actions as explained in Chapter 3. Since `SUnlock` only serves to identify synchronisation points for the memory model, one event `SUnlock` suffices, although `Wait` may release the monitor multiple times.

When the thread reacquires the lock, `Acquire` (Figure 3.14) generates the corresponding event `SLock` via the locale parameter `acq-events`, which I have left unspecified in the presentation so far. Source code and bytecode both instantiate `acq-events` such that `acq-events ln` computes a list of `SLock` events for all temporarily released locks in `ln`.

Only the `clone` method generates multiple events in a single reduction (see Appendix B.4.2), namely an allocation for the cloned object or array and `Read` and `Write` for all members that it copies.
The source code semantics generates three kinds of events: First, `sync (_, _)` produces `SLock` and `SUnlock` events (`RLOCK`, `RUNLCK`, and `RUNLCKX` in Figure 3.21). Second, access to members of an object are recorded in `Read` and `Write` events, see, e.g., `RFACC` and `RFASS` in Figure 4.4. Third, memory allocation via `new C` and `new T[_]` generates events `New-Obj _C` and `NewArray _T _`, respectively.

**Thread start and finish events**

Remember that the JMM requires that events mark the start and termination of every thread (notation `TInit` and `TFinish`, respectively). In JinjaThreads, layer 4 on the stack of semantics implements this generically as a semantics transformer in the context `multithreaded-base` from §3.1.3. Remember that $r :: (l, t, x, 'h, 'w, 'o)$ semantics with notation $t \vdash (x, h) \rightarrow ta \rightarrow (x', h')$ is the single-threaded semantics and the predicate $final :: t \Rightarrow bool$ characterises final states. The transformer pairs every thread’s local state (type `x`) with its status (HOL type `status` with elements `Pre-Start`, `Running`, and `Finished`) and extends $r$ to $sf.r$ with notation $t \vdash ((s, x), h) \rightarrow ta \rightarrow sf ((s', x'), h')$ as follows:

**SFinit:**

$$t \vdash ((Pre-Start, x), h) \rightarrow TInit \rightarrow sf ((Running, x), h)$$

**SFrun:**

$$t \vdash (x, h) \rightarrow ta \rightarrow (x', h') \rightarrow sf ((Running, x'), h')$$

**SFfinish:**

$$final \ x \ t \vdash ((Running, x), h) \rightarrow TFinish \rightarrow sf ((Finished, x), h)$$

Threads are initially marked as `Pre-Start`. They must transition to `Running` – which generates the start event `TInit` (SFinit) – before they can execute according to $r$ (SFrun). The conversion function `pair-Pre-Start ta` pairs the initial states of spawned threads in $ta$ with `Pre-Start` such that their first reduction generates the event `TInit` via SFinit.\(^{29}\) After the

\(^{29}\)Technically, the type of events `event` and its abstraction `o` do not contain the events `TInit` and `TFinish`. Instead, I extend `o` to `o sf` where

```
datatype o sf = Event o | TInit | TFinish
```

In the presentation, I identify `o` and `o sf` and omit the coercion `Event`. In particular, `pair-Pre-Start ta` implicitly injects events in $\langle ta \rangle_o$ to `o sf` and $sf.acq-events = map Event o acq-events$ replaces $acq-events$. 

155
thread has terminated as predicated by \textit{final}, $\text{SF_{FINISH}}$ adds the $T_{Finish}$ event.

Analogously, $\text{sf.final}$ replaces $\text{final}$ where

$$\text{sf.final} \ (s, x) \iff s = \text{Finished} \land \text{final} \ x$$

I will use the prefix “\textit{sf.”} to refer to semantics that have been transformed this way, e.g., $\text{sf.J-red}$ and $\text{J.sf.redT}$.

\textbf{Lemma 4.2.} If $r$ and $\text{final}$ satisfy the well-formedness conditions of the interleaving semantics (locale \textit{multithreaded}), so do $\text{sf.r}$ and $\text{sf.final}$.

\begin{align*}
\text{multithreaded final } r \ acq-events \\
\text{multithreaded } sf.final \ sf.r \ sf.acq-events
\end{align*}

\textbf{Complete interleavings}

The JMM suggests to execute all threads in isolation and to determine a posteriori the legal executions in terms of their traces of events. Unfortunately, the events from §4.3.1 are insufficient to correctly implement the JLS, because the JLS (and the Java API) introduce other communication channels between threads. For example, consider the program in Figure 4.14 in which two threads race for spawning the same thread. Suppose both reads in ll. 3 and 5 see the write at ll.2. Then, either ll. 4 or ll. 6 must throw an \textit{IllegalThreadState} exception, but not both. Hence, both ll. 4 and ll. 6 must be allowed to fail in some executions. Thus, the two right-most threads may just start, read the address of the \textit{Thread} object (then fail with the exception, but no event records that), and then finish. Hence, if each thread were run in isolation, they both would be allowed to fail, too. Since this contradicts the specification of the \textit{start} method, there is a covert communication channel.

For the \textit{start} method, the JMM specifies synchronisation only between a successful call and the first action of the spawned thread [56, §17.4.4]. A JVM implementation might add additional synchronisation, but my semantics must not, since such synchronisation might eliminate data races from programs, i.e., it could wrongly certify programs with data races as DRF. Therefore, the JinjaThreads semantics generates no synchronisation events when \textit{start} fails (\textit{STARTFAIL}).

Hence, threads cannot execute in isolation, as the JMM suggests. Instead, I employ the interleaving semantics from §3.1 to compute
4.3. Java memory model

initially: \( x = \text{null} \);  
\[
1: r1 = \text{new Thread();}
2: x = r1;
3: r2 = x;
4: r2.start();
5: r3 = x;
6: r3.start();
\]

**Figure 4.14:** Thread spawn as an implicit communication channel

Their interleavings. This way, I reuse the infrastructure for mutual exclusion for locks and managing the monitor wait sets, notifications and interrupts. The single-threaded semantics for source code and bytecode are \( sf.J\text{-red} P \) and \( sf.jvm\text{-exec} P \), respectively, i.e., the semantics from Chapter 3 augmented with thread start and finish actions from layer 4.

A complete interleaving \( E \) is a possibly infinite list\(^{30} \) of pairs of thread ID and event. The relation \( s \downarrow E \) characterises all complete interleavings \( E \) that start in the state \( s \), which I define as

\[
s \downarrow E \iff (\exists \text{ttas} \cdot s \downarrow \text{ttas} \land E = \text{concat} (\text{map events} \ \text{ttas})) \tag{4.2}
\]

where \( \text{events} (t, \text{ta}) = \text{map} (\lambda e. (t, e)) \langle \text{ta} \rangle \) pairs all events in \( \text{ta} \) with the thread ID \( t \), and \( s \downarrow \text{ttas} \) collects the list of thread actions labelled with the thread ID as follows:

\[
\begin{align*}
\text{STOP} & \quad \quad \text{STEP} \\
\frac{s \implies s \downarrow \text{ttas}}{s \downarrow []} & \quad \quad \frac{s \downarrow \langle t, \text{ta} \rangle \cdot \text{ttas}}{s \implies s' \implies s' \downarrow \text{ttas}}
\end{align*}
\]

where \( _\implies \) characterises stuck states in the interleaving semantics.

Note that the detour via a list of thread actions is necessary. If I defined \( s \downarrow E \) directly with the above coinductive rules \( \text{STOP} \) and \( \text{STEP} \) (i.e., prepending \( \text{events} (t, \text{ta}) \) to \( E \) instead of consing), I could derive every trace \( E \) for a state \( s \) that can perform an infinite sequence of reductions without events, i.e., \( \langle \text{ta} \rangle = [] \), because it would be impossible to prove that \( E \) was not a trace (see also the example on coinductive definitions in §1.4.3). My approach works fine since \( (t, \text{ta}) \cdot E \) is productive and concatenating the infinite list of empty lists yields \( [] \).

\(^{30}\)Isabelle’s default type for lists \( \text{’a list} \) models only finite lists. Therefore, I have developed a theory of possibly infinite lists (HOL type \( \text{’a llist} \)) \([110]\) based on the codatatype construction by Paulson \([142]\). For clarity of presentation, I use the same notation for operations on \( \text{’a list} \) and \( \text{’a llist} \) (e.g., \( _\cdot @ _ \) and \( \text{concat} \) and omit the coercions between the two types, although Isabelle distinguishes the former and requires the latter.
The start state (Figure 4.3) has preallocated start-tlD’s Thread object and the system exceptions. Hence, I prefix the complete interleavings with the appropriate initialisation events start-events:

\[
\text{start-events} = \\
\text{(start-tlD, TInit)} \cdot \text{map (λ(C, a). (start-tlD, New-Obj a C))} \\
\text{zip (Thread \cdot sys-xtpts) start-addrs)
\]

For the JMM, a program P always comes with a fixed start state, i.e., let start-state refer to either J-start P C M vs or jvm-start P C M vs. Then, the JMM identifies a program with the set \( \mathcal{E} \) of complete interleavings that start in start-state, prefixed with start-events. Formally:

\[
\mathcal{E} = \{ \text{start-events} @ E \mid E. \text{start-state} \downarrow E \}
\]

\( \mathcal{E} \) contains many ill-formed executions, because read operations may read arbitrary values, even not type-conforming ones under heap implementation 1 that no write operation of the program can ever produce. Since such executions have no write-seen function, the rules on layer 7 discard them.

Let me now present an example of \( \mathcal{E} \) in detail. Figure 4.15 shows a Java implementation of the example in Figure 4.7. There is a bootstrapping thread \( t_0 \) that creates and spawns the two threads \( t_1 \) and \( t_2 \) whose run methods contain the code from the example. Since JinjaThreads does not model static fields, the shared locations \( x \) and \( y \) are represented by the fields of a container class \( C \).

All complete interleavings start with the following events that represent \( t_0 \)’s main method up to the first call to start, abbreviated as as:

\[
\text{start-events @} \\
[ (t_0, \text{New-Obj } a_0 \ C), \\
(t_0, \text{New-Obj } a_1 \ T1), (t_0, \text{Write } a_1 (\text{Field } c \ T1) a_0), \\
(t_0, \text{New-Obj } a_2 \ T2), (t_0, \text{Write } a_2 (\text{Field } c \ T2) a_0), (t_0, \text{TStart (a2t a1)})]
\]

i.e., \( t_0 \) allocates the objects for the container and the two threads at locations \( a_0 \) to \( a_2 \) and executes \( T1 \)’s and \( T2 \)’s constructors. Remember that the allocations initialise the fields with default values, i.e., 0 for \( x \) and \( y \) declared in \( C \), and Null for \( c \) declared in \( T1 \) and \( T2 \).

\(^{31}\)Since start-events already contains start-tlD’s TInit, I assume that the initial thread-local state for start-tlD is paired with Running.
class T0 {
    public static void main(String[] args) {
        C c = new C();
        Thread t1 = new T1(c);
        Thread t2 = new T2(c);
        t1.start();
        t2.start();
    }
}

class T1 extends Thread {
    C c;
    T1(C c) { this.c = c; }
    public void run() { C c = this.c; int r1 = c.x; c.y = 1; }
}

class T2 extends Thread {
    C c;
    T2(C c) { this.c = c; }
    public void run() { C c = this.c; int r2 = c.y; c.x = 2; }
}

class C { int x, y; }

Figure 4.15: Java implementation for the example in Figure 4.7

Thread $t_1$ has three structurally different traces depending on the kind of value that reading $\texttt{this}.c$ stores in the local variable $c$, namely:

1. $[\text{TInit}, \text{Read } a_1 (\text{Field } c \ T1) (\text{Addr } a), \text{Read } a (\text{Field } x \ C) v, \text{Write } a (\text{Field } y \ C) 1, \text{TFinish}]$,
2. $[\text{TInit}, \text{Read } a_1 (\text{Field } c \ T1) \text{ Null}, \text{TFinish}]$, and
3. $[\text{TInit}, \text{Read } a_1 (\text{Field } c \ T1) w]$,

where $a$ is an arbitrary address, $v$ is an arbitrary value, and $w$ is either $\text{Unit}$ or an integer or boolean value. In the first form, the address $a$ from the $\text{Read}$ is then used to access the fields $x$ and $y$ of the referenced container. In the second, $t_1$ reads $\text{Null}$, so the subsequent field access raises the preallocated NullPointer exception and the thread immediately terminates. In the last case, $w$ is type-incorrect, so the semantics gets stuck upon the next field access, i.e., there is no $\text{TFinish}$. Under implementation 2, the last case is impossible, and $v$ must be an integer value if $a = a_0$, 159
because it forbids reads to return type-incorrect values. Thread $t_2$ has the same traces with $x$ and $y$ exchanged.

The complete interleavings in $E$ for this program all start with $a$s and then interleave the threads. Of these, the JMM considers only those as well-formed that originate from the first form with $a = a_0$, $v \in \{\text{Intg 0, Intg 2}\}$ for $t_1$ and $v \in \{\text{Intg 0, Intg 1}\}$ for $t_2$. In particular, the unexpected behaviour from Figure 4.7 is well-formed. In terms of the JMM, all these interleavings collapse to four well-formed executions as shown in Figure 4.16 (where I have omitted the bootstrap events for clarity – except for $t_0$’s initial event that is relevant for determining what happens before what). The write-seen arrows are labelled with conditions for which they apply. All well-formed executions are legal in this example.
4.3. Java memory model

Definition of the Java memory model

Now, I formally derive the orders of the JMM from a complete interleaving $E \in \mathcal{E}$. As this core of the JMM has been formalised before [8, 69], I keep this section brief; for the intuition behind the concepts, see [8, 56, 69, 115].

Since an event can occur multiple times in $E$, I use the index in $E$ to assign a unique identifier to an event, i.e., $\mathcal{A}_E = \{ \alpha. \alpha < |E| \}$ denotes the set of events for $E$. In the following, I identify an event with its index, i.e., I write $\alpha$ instead of $E[\alpha]$ when it is clear from the context. A read event is an event of the form $(t, \text{Read } a al v)$, it reads from location $(a, al)$ the value $v$; $\mathcal{R}_E$ denotes the set of read events of $E$. A write event is either a write $(t, \text{Write } a al v)$ or an initialisation $(t, \text{Allocate } a hT)$, $\mathcal{W}_E$ denotes the set of write events in $E$. A write event $\alpha \in \mathcal{W}_E$ writes to location $(a, al)$ (is a write to $(a, al)$) iff $\alpha = (\_, \text{Write } a al \_)$, or $\alpha = (\_, \text{Allocate } a hT)$ and $al$ is a member of $hT$, i.e., (i) if $al = \text{Field } F D$, then $P \vdash \text{class-of' } hT \text{ has } F: \_ (\_) \text{ in } D$, and (ii) if $al = \text{Cell } n$, then $hT = \text{ArrayT } \_ n'$ such that $n < n'$. I say that $\alpha$ accesses location $(a, al)$ iff $\alpha$ is a read or write event that reads from or writes to $(a, al)$, respectively. $\text{locs } P \ E \alpha$ denotes the set of locations that $\alpha \in \mathcal{R}_E \cup \mathcal{W}_E$ accesses; $\text{locs } P \ E \alpha = \emptyset$ for $\alpha \in \mathcal{A}_E - (\mathcal{R}_E \cup \mathcal{W}_E)$.

For $\alpha \in \mathcal{W}_E$, value-written $P \ E \alpha (a, al)$ denotes the value that $\alpha$ writes to location $(a, al)$ — allocation events write default values ($\text{Intg } 0$, $\text{Bool } False$, and $\text{Null}$, respectively) for all allocated members; normal writes $\text{Write}$ store the value written themselves; it is undefined if $\alpha$ does not write to $(a, al)$.

A member $al$ is volatile (written $\text{is-volatile } P al$) iff $al = \text{Field } F D$ and $P$ declares field $F$ in $D$ as volatile. A read or write $\alpha$ is volatile iff $\alpha$ reads from or writes to a volatile member of a location. Note that array cells are never volatile by definition [56, §8.3.1.4].

A complete interleaving $E$ already provides the induced total order $\leq^E = \leq|_{\mathcal{A}_E}$ over $\mathcal{A}_E$, where $R|_A$ restricts the binary relation $R$ to elements from $A$ and $\leq$ is the standard order on natural numbers.

Since the JMM requires initialisation events (i.e. $\text{Allocate}$) to be ordered before the threads’ initial events, I introduce the (total) execution order $\leq_{eo}^E$ on $\mathcal{A}_E$:

$$\alpha \leq_{eo}^E \alpha' \iff (\text{if } \text{init}_E \alpha \text{ then } \neg \text{init}_E \alpha' \vee \alpha \leq^E \alpha' \text{ else } \neg \text{init}_E \alpha' \land \alpha \leq^E \alpha')$$

where $\text{init}_E \alpha$ predicates that $\alpha$ is an initialisation event in $E$, i.e., $E[\alpha] = (\_, \text{Allocate } \_ \_)$. 161
Chapter 4. Memory models

Figure 4.17: Release-acquire pairs

The program order $\leq_{po}^E$ restricts $\leq_{eo}^E$ to events of the same thread. The synchronisation order $\leq_{so}^{p,E}$ restricts $\leq_{eo}^E$ to synchronisation events. Synchronisation events are the initialisation events ($Allocate$), reads from and writes to volatile locations, locking ($SLock$) and unlocking ($SUnlock$), thread spawns ($TStart$) and joins ($TJoin$), thread start ($TInit$) and finish events ($TFinish$), and the interruption events $TIntr$ and $TIntrd$. The synchronises-with order $\leq_{sw}^{p,E}$ restricts $\leq_{so}^{p,E}$ to release-acquire pairs of events. $(\alpha, \alpha')$ is a release-acquire pair (notation $\alpha \Rightarrow_{sw} \alpha'$, definition in Figure 4.17) iff

(i) $\alpha$ unlocks a monitor and $\alpha'$ locks the same monitor,
(ii) $\alpha$ writes to a location that $\alpha'$ reads,$^{32}$
(iii) $\alpha$ spawns a thread whose start action is $\alpha'$,
(iv) $\alpha$ is the finish event of the thread on which $\alpha'$ joins,
(v) $\alpha$ is an initialisation event and $\alpha'$ is a thread start event, or
(vi) $\alpha$ interrupts a thread $t$ and $\alpha'$ observes that $t$ has been interrupted.

The happens-before order $\leq_{hb}^{p,E}$ is the transitive closure of $\leq_{po}^E$ and $\leq_{sw}^{p,E}$.

An execution $(E, ws)$ consists of a complete interleaving $E$ and a write-seen function $ws$ that assigns to every read event in $A_E$ the write event it sees. This yields the JMM notion of an execution [56, §17.4.6] as $(E, A_E, \leq_{po}^E, \leq_{so}^{p,E}, ws, value-written \in P E, \leq_{sw}^{p,E}, \leq_{hb}^{p,E})$.

An execution is well-formed (written $P \vdash (E, ws) \checkmark$) iff every thread has a thread start event that $\leq_{E}$-precedes its other events except for initialisation events (denoted $ok-init E$) and for all read events $\alpha \in R_E$ to some location $(a, al)$.

$^{32}$I do not need to restrict writes and reads to volatiles explicitly like the JMM does [56, §17.4.4], because the synchronisation order already imposes this.
4.3. Java memory model

W1 \( ws \alpha \) writes to \((a, al)\), i.e., \( ws \alpha \in W_{E} \) and \((a, al) \in \text{locs } P E (ws \alpha) \),

W2 \( \alpha \) reads the value value-written \( E (ws \alpha) (a, al) \),

W3 \( \alpha \not\leq_{\text{hb}}^{P,E} ws \alpha \),

W4 for all write events \( \beta \) to \((a, al)\), if \( ws \alpha \leq_{\text{hb}}^{P,E} \beta \leq_{\text{hb}}^{P,E} \alpha \), then \( \beta = ws \alpha \), and

W5 if \( \alpha \) is a volatile read, then \( \alpha \not\leq_{\text{so}}^{P,E} ws \alpha \) and for all write actions \( \beta \) to \((a, al)\), if \( ws \alpha \leq_{\text{so}}^{P,E} \beta \leq_{\text{so}}^{P,E} \alpha \), then \( \beta = ws \alpha \).

\( E \) is well-formed iff \( P \vdash (E, ws) \sqrt{1} \) for some \( ws \).

These conditions correspond to the JMM well-formedness conditions 1 (each read sees a write to the same location), 4 (\( \leq_{\text{hb}}^{P,E} \) consistency) and 5 (\( \leq_{\text{so}}^{P,E} \) consistency for volatiles) in [56, §17.4.7]. \((E, ws)\) meets conditions 2 (\( \leq_{\text{hb}}^{P,E} \) is a partial order) and 3 (intra-thread consistency) by construction.

The JMM constrains \( \leq_{\text{so}}^{P,E} \) to be an \( \omega \)-order for well-formed executions. As Aspinall and Ševčík [8] already noted, in an infinitely running program, infinitely many allocation events for volatile fields synchronise with thread start events, which violates this constraint, i.e., the JMM would allow no behaviour at all. To remedy this, I drop this constraint. Note that \( \leq^{E} \) is of order at most \( \omega \) by construction, hence \( \leq_{\text{so}}^{P,E} \) is of order at most \( \omega + \omega \) by definition.

Lemma 4.3. If \( E \in \mathcal{E} \), then ok-init \( E \).

A legal execution is a well-formed execution \((E, ws)\) that is justified by a sequence of justifying executions \((E_i, ws_i, C_i, \varphi_i)\), where \( C_i \) are the sets of committed events and the event renaming functions \( \varphi_i \) inject the committed events of \( E_i \) into \( E \)'s events. The definition in Figure 4.18 uses the following notation: inj-on \( f A \) expresses that the function \( f \) is injective on the set \( A \); \( \alpha \varphi_i^{-1}(R) \alpha' \) iff \( \varphi_i \alpha R \varphi_i \alpha' \) for a binary relation \( R \); \( \varphi_{i+1}^{-1}(\alpha) \) abbreviates \( \varepsilon \alpha' \). \( \alpha' \in C_{i+1} \wedge \varphi_{i+1} \alpha' = \alpha \); \( (t, e) \simeq (t', e') \) iff \( t = t' \) and the events \( e \) and \( e' \) are identical except for the values they write or read.

The constraints are exactly the JMM legality conditions 1 to 7 and 9 [56, §17.4.8] with explicit renaming of events plus basic requirements for commit sequences and event renamings. I omit condition 8 for two reasons: First, it relies on the transitive reduction of \( \leq_{\text{hb}}^{P,E_i} \), which need
Chapter 4. Memory models

\( P \vdash (E, ws) \) justified-by \((E_i, ws_i, C_i, \varphi_i)_i \) \iff

\( (\forall i. P \vdash (E_i, ws_i)) \land \)

\( (\forall i. \text{inj-on } \varphi_i \mathcal{A}_{E_i} \land (\forall \alpha \in C_i. E_i[\alpha] \simeq E_{(\varphi_i \alpha)}) \land (\forall i. C_i \subseteq \mathcal{A}_{E_i}) \land \)

\( C_0 = \emptyset \land (\forall i. \varphi_i \land C_i \subseteq \varphi_{i+1} \land \mathcal{A}_E = \bigcup \varphi_i \land C_i \land \)

\( (\forall i. \leq_{P,E_i} C_i = \varphi_i^{-1}(\leq_{P,E_i}) \land (\forall i. \leq_{P,E_i} C_i = \varphi_i^{-1}(\leq_{P,E_i})) \land \)

\( (\forall i. \forall \alpha \in W_i \cap C_i. \forall (a, al) \in \text{locs } P \ E (\varphi_i \alpha). \)

value-written \( P \ E \ a \ (a, al) = \text{value-written } P \ E \ (\varphi_i \alpha) \ (a, al) \land \)

\( (\forall i. \forall \alpha \in R_{E_i} \cap C_i. \varphi_i+1 (ws_{i+1} \ (\varphi_i^{-1} \ (\varphi_i \alpha))) = ws \ (\varphi_i \alpha) \land \)

\( (\forall i. \forall \alpha \in R_{E_{i+1}}. \varphi_{i+1} \alpha \in \varphi_i \land C_i \lor ws_{i+1} \alpha \leq_{hb} P_{E_{i+1}} \alpha) \land \)

\( (\forall i. \forall \alpha \in R_{E_{i+1}} \cap C_{i+1}. \)

\( \varphi_i \alpha \in \varphi_i \land C_i \lor \{ \varphi_i+1 (ws_{i+1} \alpha), ws \ (\varphi_i+1 \alpha) \} \subseteq \varphi_i \land C_i \land \)

\( (\forall i. \forall \alpha \in A_{E_i}. \forall \alpha' \in C_i. \forall a M \ vs \ v. \ E_{i[\alpha]} = \text{Extern } a M \ vs \ v \rightarrow \alpha \leq_{hb} P_{E_{i+1}} \alpha' \rightarrow \alpha \in C_i) \)

\( P, E \vdash (E, ws) \) legal \iff \( E \in \mathcal{E} \land P \vdash (E, ws) \land \)

\( (\exists (E_i, ws_i, C_i, \varphi_i)_i. P \vdash (E, ws) \) justified-by \((E_i, ws_i, C_i, \varphi_i)_i \land \)

\( (\forall i. E_i \in \mathcal{E})) \)

Figure 4.18: Legality constraints for the justification \((E_i, ws_i, C_i, \varphi_i)_i\) for the execution \((E, ws)\)

not exists for infinite executions. Second, Torlak et al. [171] have shown that it is irrelevant for all JMM test cases.

I am not going to explain the constraints in detail, as others have done so already [8, 69]. As §4.3.1 explains, they serve to ban values appearing out of thin air, but §4.3.5 shows that they fail to do so.

4.3.3 The data race freedom guarantee

Remember that the JMM promises that correctly synchronised programs behave as if they were executed under sequential consistency. In this section, I recapitulate the definitions and identify the assumptions of this guarantee. Then, I show that source code and bytecode indeed satisfy these assumptions.

The proof of the DRF guarantee extends over all layers of the semantics stack (Figure 4.10). Hence, the challenge consists of adequately decomposing the proof and distributing it over the layers such that each
proof is as abstract as possible. This way, I prove the DRF guarantee for both JMM heap implementations and for both source code and bytecode (Theorem 4.6) almost simultaneously. To that end, I extend the locales for the heap module (§4.1) and the multithreaded semantics (§3.1.3) with additional operations and assumptions. It is crucial that these assumptions respect the abstraction of the layer, i.e., they only refer to notions of the layer or below, but not above. For example, assumptions about the heap implementation must not mention JMM executions.

Therefore, this section focuses on what these assumptions are and how to formalise them. The transition from the global behaviour (executions and complete interleavings) to the individual steps of the small-step semantics is the most difficult one, because it must translate global notions into state invariants. Sometimes, this is not directly possible: for example, levels 5 and below generalise the happens-before order to the execution order, because happens-before is hard to express as a state invariant.

To derive and motivate the low-level assumptions from those on higher levels, the presentation starts with the proofs on the JMM level and then descends the stack of semantics, similar to backward-style reasoning.

### The DRF guarantee

In this section, I formally state the DRF guarantee and prove it. Two events of an execution are conflicting if they are read or write events to the same location with at least one being a write event. Two conflicting events constitute a data race if they are not ordered by happens-before, i.e., may happen concurrently.

An execution \((E, ws)\) is sequentially consistent (SC) ifff every read event \(\alpha \in R_E\) sees the most recent write event, i.e. \(ws \alpha \leq_{eo} E \alpha\), and \(\beta \leq_{eo} ws \alpha\) or \(\alpha \leq_{eo} E \beta\) for all write events \(\beta\) to the location that \(\alpha\) reads from.\(^{33}\)

A program is correctly synchronised (data race free) ifff none of its SC executions contains a data race. Formally: Whenever \(E \in \mathcal{E}, P \vdash (E, ws)\).

\(^{33}\)The JMM only requires that \(\leq_{po}^E\) is extended to a total order over all events of all threads to determine most recent writes [56, §17.4.3]. Aspinall and Ševčík [8] showed that, to respect mutual exclusion of locks, the total order must also extend \(\leq_{po}^{PE}\). My execution order \(\leq_{eo}^E\) extends both by construction.
and \((E, ws)\) is SC, then \(\alpha \leq_{\text{hb}}^{P, E} \alpha'\) or \(\alpha' \leq_{\text{hb}}^{P, E} \alpha\) for all conflicting events \(\alpha, \alpha' \in \mathcal{A}_E\).

For the DRF guarantee, it is important that only SC executions must not contain a data race. Otherwise, it would fail its purpose because the programmer would have to understand the whole JMM to see whether her program is correctly synchronised and the DRF guarantee applies to it.

My proof of the DRF guarantee (Theorem 4.3) adapts the others' [8, 69, 115] to deal with memory allocation and initialisations (see §4.3.6 for a discussion). The key idea in all of them is that in a DRF program, a well-formed execution \((E, ws)\) is SC if every read sees a write that happens before it (Lemma 4.4) – then, the legality constraints ensure that all legal executions are SC.

**Lemma 4.4 (DRF lemma [8, Lemma 2]).** Let \(E\) be correctly synchronised and \(E \in \mathcal{E}\) such that \(P \vdash (E, ws)\)\(\checkmark\). If \(ws \alpha \leq_{\text{hb}}^{P, E} \alpha\) for every read \(\alpha\) in \(\mathcal{R}_E\), then \((E, ws)\) is sequentially consistent.

To exploit correct synchronisation in a proof of this lemma by contradiction (see below), one first obtains a SC execution \((E', ws')\) from \((E, ws)\) as follows: \(E'\) starts like \(E\) until the first non-SC read \(\alpha\) in \(E\) and continues sequentially consistently from there on. Then, it suffices to find a data race between \(\alpha, ws \alpha\), and \(ws' \alpha\) in \(E'\). For the latter, I use Lemma 4.6 (see below) to transfer happens-before relationships between \(E\) and \(E'\) on their common prefix. The proof therefore rests on two assumptions on the set of complete interleavings \(\mathcal{E}\):

\begin{itemize}
  \item **D1** For every sequentially-consistent prefix of a well-formed execution \((E, ws)\) with \(E \in \mathcal{E}\), there is a complete interleaving \(E' \in \mathcal{E}\) with the same prefix and a write seen-function \(ws'\) such that \(P \vdash (E', ws')\)\(\checkmark\) and \((E', ws')\) is SC. If \(E\) immediately continues with a read after the prefix, \(E'\) also continues with a read from the same location.
  \item **D2** Every execution \(E \in \mathcal{E}\) initialises every location at most once.
\end{itemize}

The first assumption ensures that \(E'\) as required in the proof of Lemma 4.4 does exist, the second is a standard well-formedness condition. Below, I prove that source code and bytecode satisfy these.

From the DRF lemma, the DRF guarantee (Theorem 4.3) follows. I omit its proof since it closely follows [8, Theorem 1].

**Theorem 4.3 (DRF guarantee).** If the program \(P\) is correctly synchronised and \((E, ws)\) a legal execution, then \((E, ws)\) is sequentially consistent.
To prove Lemma 4.4, I need two more lemmata about happens-before:

**Lemma 4.5.** Let ok-init \( E \) and \( \alpha, \alpha' \in \mathcal{R}_E \) such that \( \text{init}_E \alpha \) and \( \neg \text{init}_E \alpha' \). Then \( \alpha \leq_{\text{hb}}^{P,E} \alpha' \).

**Proof.** Let \( t \) be the initial event \( T\text{Init} \) of \( \alpha' \)'s thread. By definition, \( \alpha \leq_{\text{sw}}^{P,E} t \leq_{\text{po}}^{P,E} \alpha' \).

**Lemma 4.6** (happens-before prefix lemma). Let \( E \) and \( E' \) be two complete interleavings such that their first \( n \) events differ only in the values read or written, and let \( \alpha, \alpha' < n \). If ok-init \( E' \) and \( \alpha \leq_{\text{hb}}^{P,E} \alpha' \), then \( \alpha \leq_{\text{hb}}^{P,E'} \alpha' \).

**Proof.** By induction on the transitive closure of \( \alpha \leq_{\text{hb}}^{P,E} \alpha' \). In the base case, \( \alpha \leq_{\text{po}}^{E} \alpha' \) or \( \alpha \leq_{\text{sw}}^{P,E} \alpha' \). By unfolding the definitions, \( \alpha \leq_{\text{po}}^{E'} \alpha' \) or \( \alpha \leq_{\text{sw}}^{P,E'} \alpha' \) follows. Hence, \( \alpha \leq_{\text{hb}}^{P,E'} \alpha' \).

In the induction step, I may assume \( \alpha, \alpha'' < n \), and \( \alpha \leq_{\text{hb}}^{P,E} \alpha' \), and \( \alpha' \leq_{\text{po}}^{E} \alpha'' \) or \( \alpha' \leq_{\text{sw}}^{P,E} \alpha'' \), and the induction hypothesis if \( \alpha' < n \), then \( \alpha \leq_{\text{hb}}^{P,E} \alpha' \). I must show that \( \alpha \leq_{\text{hb}}^{P,E'} \alpha'' \).

If \( \neg \text{init}_E \alpha' \) or \( \text{init}_E \alpha'' \), then \( \alpha' \leq_{\text{po}}^{E} \alpha'' \) by definition of \( \leq_{\text{eo}}^{E} \), because \( \alpha' \leq_{\text{eo}}^{E} \alpha'' \) follows from either \( \alpha' \leq_{\text{po}}^{E} \alpha'' \) or \( \alpha' \leq_{\text{sw}}^{E} \alpha'' \). Since \( \alpha'' < n \), also \( \alpha' \leq_{\text{hb}}^{P,E'} \alpha'' \) or \( \alpha' \leq_{\text{sw}}^{P,E'} \alpha'' \) follow from \( \alpha' \leq_{\text{po}}^{E} \alpha'' \) or \( \alpha' \leq_{\text{sw}}^{P,E} \alpha'' \) as in the base case. Therefore, \( \alpha \leq_{\text{hb}}^{P,E'} \alpha'' \).

Otherwise, I have \( \text{init}_E \alpha' \) and \( \neg \text{init}_E \alpha'' \). Then, \( \text{init}_E \alpha \) follows from \( \text{init}_E \alpha' \) by induction on \( \alpha \leq_{\text{hb}}^{P,E} \alpha' \). Since \( \alpha, \alpha'' < n \) and \( E' \)'s \( \alpha \) actions only differ in the values read or written, \( \text{init}_E \alpha \) and \( \neg \text{init}_E \alpha'' \), too. Hence \( \alpha \leq_{\text{hb}}^{P,E'} \alpha'' \) by Lemma 4.5. \( \square \)

Now, I am ready to prove the DRF lemma.

**Proof of Lemma 4.4.** By contradiction. Suppose that \( (E, ws) \) is not SC. Note that \( \leq_{\text{eo}}^{E} \) is well-founded by construction. Let \( \alpha \in \mathcal{R}_E \) be the \( \leq_{\text{eo}}^{E} \) minimal read event from some location \( (a, al) \) such that \( ws \alpha \) is not the most recent write for \( \alpha \) in \( E \). By assumption, \( ws \alpha \leq_{\text{hb}}^{P,E} \alpha \). Moreover, there is a write event \( \beta \in \mathcal{W}_E \) to \( (a, al) \) such that \( \beta \not\leq_{\text{hb}}^{P,E} ws \alpha, \alpha \not\leq_{\text{hb}}^{P,E} \beta \), and \( \beta \leq_{\text{eo}}^{E} \alpha \) otherwise, \( ws \alpha \) would be the most recent write for \( \alpha \).
Then, \(\neg \text{init}_E \beta\), as otherwise \(\neg \text{init}_E (\text{ws} \alpha)\), because \(E\) initialises every location at most once, and therefore \(\beta \leq_{\text{hb}}^{P,E} \text{ws} \alpha\) by Lemma 4.5. I show that \(\text{ws} \alpha \leq_{\text{hb}}^{P,E} \beta\), which violates the happens-before consistency condition W4 of \(\text{ws}\) being well-formed. If \(\text{init}_E (\text{ws} \alpha)\), then \(\text{ws} \alpha \leq_{\text{hb}}^{P,E} \beta\) by Lemma 4.5. So, suppose \(\neg \text{init}_E (\text{ws} \alpha)\). With \(\neg \text{init}_E \beta\), both \(\beta\) and \(\text{ws} \alpha\) occur before \(\alpha\) in \(E\), i.e., \(\beta, \text{ws} \alpha \leq_{E} \alpha\). By requirement D1, I obtain a well-formed execution \(\langle E', \text{ws}' \rangle\) that starts with \(E\) up to \(\alpha\) and continues SC, with \(\alpha\) being a read from \((a, al)\) in \(E'\). As \(\beta, \text{ws} \alpha \in \mathcal{A}_{E'}\), are conflicting and \(\langle E', \text{ws}' \rangle\) is SC, \(\beta \leq_{\text{hb}}^{P,E'} \text{ws} \alpha\) or \(\text{ws} \alpha \leq_{\text{hb}}^{P,E'} \beta\). By Lemma 4.6, \(\beta \leq_{\text{hb}}^{P,E} \text{ws} \alpha\) or \(\text{ws} \alpha \leq_{\text{hb}}^{P,E} \beta\), but \(\beta \not\leq_{\text{hb}}^{P,E} \text{ws} \alpha\). □

Note that Lemma 4.6 (and Lemma 4.4 and Theorem 4.3 which build on it) requires that all initialisation events synchronise with thread start events, i.e., they are synchronisation events. For example, consider the program and two of its well-formed interleavings in Figure 4.19. Both interleavings share the prefix \([(t_1, \text{TInit}), (t_1, \text{initialise} x), (t_2, \text{TInit})]\). If only initialisations for volatile locations synchronised with thread start events (as Manson suggested [114]), \((t_1, \text{initialise} x)\) would not synchronize with \((t_2, \text{TInit})\), i.e., there would not be such a dotted arrow in Figure 4.19. For \(E_1\), we would still get \((t_1, \text{initialise} x) \leq_{\text{hb}}^{P,E_1} (t_2, \text{TInit})\), because

\[(t_1, \text{initialise} x) \leq_{E_1}^{E_1} (t_1, \text{New-Obj} \ a \ C) \leq_{\text{sw}}^{P,E_1} (t_2, \text{TInit})\]

and \((t_1, \text{New-Obj} \ a \ C)\) writes (initialises) the volatile member \(\text{Field} v C\). However, for \(E_2\), not \((t_1, \text{initialise} x) \leq_{\text{hb}}^{P,E_2} (t_2, \text{TInit})\).

As all initialisation events synchronise with \(\text{TInit}\) events, I may subsume initialisation events in a single allocation event. Otherwise, I would have had to separate the event for initialising ordinary members from the one for volatiles. This would have complicated the model and therefore the proofs as well.

At most one initialisation

Above, I have shown the DRF guarantee under two assumptions on the set \(E\) of complete interleavings. In the remainder of this section, I discharge them for source code and bytecode by descending the stack of semantics (Figure 4.10) and adapting the assumptions. They act like an
class C {
    volatile int v;
}

\[ t_1 \text{ initialises } x = \emptyset; \]
\[ \begin{array}{ll}
1: & r1 = x; \\
2: & \text{if } (r1 == \emptyset) \\
3: & r2 = \text{new } C(); \\
4: & x = 1;
\end{array} \]

\( E_1: [(t_1, T\text{Init}), (t_1, \text{initialise } x), (t_2, T\text{Init}), (t_1, \text{Read } x 0),
(t_1, \text{New-Obj } a \text{ C}), (t_2, \text{Write } x 1), (t_1, T\text{Finish}), (t_2, T\text{Finish})] \)

\( E_2: [(t_1, T\text{Init}), (t_1, \text{initialise } x), (t_2, T\text{Init}), (t_2, \text{Write } x 1),
(t_1, \text{Read } x 1), (t_1, T\text{Finish}), (t_2, T\text{Finish})] \)

\textbf{Figure 4.19}: Two well-formed complete interleavings for the program at the top
interface between the levels and ensure that I can share the proofs for all layers that source code and bytecode share.

To help the reader follow the proofs, Figure 4.20 shows how the assumptions D1 and D2 evolve and assigns them to the level of the semantics. The arrows are stylised implications, i.e., the assumption at its source discharges the one at the target. Where multiple arrows point to one assumption, the conjunction of the sources imply the target. For example, assumptions D4 and D3 together discharge D1. The assumptions ka-type, m-allocated, and allocated extend the heap module by new operations; hence, I have phrased them as separate locales instead of additional assumptions. Layer 4 is not shown, because layer 4 is irrelevant for the assumptions that level 5 makes. Layer 3 and 2 are the core of the semantics where the crucial parts of the assumptions are discharged. Their assumptions about the heap module are listed at layer 1.

I start with assumption D2 that every execution initialises a location at most once. Remember that allocation events initialise locations. When an allocation returns an address, it was fresh before, but afterwards, it is allocated, i.e., not fresh. Hence, it suffices to prove that the semantics correctly keeps track of all memory allocations in the events.

To discharge D2 on the level of interleaving semantics, I extend the locale multithreaded with a parameter allocated :: 'h ⇒ 'addr set that returns the set of allocated addresses (locale m-allocated in Figure 4.21). Its specification requires that for every single-thread reduction $t ⊢ (x, h) →_t a (x', h')$
(i) the set of allocated addresses never shrinks, i.e., $allocated h \subseteq allocated h'$,

(ii) $\langle ta \rangle_o$ contains an allocation event for address $a$ iff $a \in allocated h'$ and $a \notin allocated h$, and

(iii) $\langle ta \rangle_o$ contains at most one allocation event per address.

Thus, the concept of allocated addresses reduces the global property of at most one initialisation to a property of single reductions.

To prove that start-events correctly records the allocations of the start-heap, I relate allocated also with the abstract heap operations. This way, the proofs about start-events and that the single-threaded semantics satisfy the assumptions of m-allocated do not depend on the concrete heap implementation. Locale allocated, from which m-allocated inherits, formalises these additional assumptions (Figure 4.21):

(i) the empty heap empty-heap has no allocated addresses,

(ii) if successful, allocation updates the heap $h$ to $h'$ such that the returned address $a$ is allocated in $h'$, but not in $h$, and the other addresses' allocation status remains unchanged, and

(iii) writing to a location and failed allocations leave the set of allocated addresses unchanged.\(^{34}\)

The JMM heap implementations instantiate allocated with

$$\text{dom} \circ jmm\text{-typeof-addr} \quad \text{and} \quad \lambda h. \{ \text{Address } hT \ n | hT \ n. \ n < h \ hT \},$$

respectively. Both meet the assumptions of allocated.

Under these assumptions, it is routine to show that start-events exactly records the allocations for the start heap start-heap and that the sequential semantics of source code and bytecode meet m-allocated's assumptions. This concludes the proof of D2.

**Lemma 4.7 (Assumption D2).** In locale m-allocated, every execution $E \in E$ initialises every location at most once.

---

\(^{34}\)I do not need any assumption for read because reading cannot change the heap, i.e., it does not affect allocated by construction.
locale allocated-base = heap-base + fixes allocated :: 'heap ⇒ 'addr set

locale allocated = allocated-base + heap +
assumes allocated empty-heap = ∅
and alloc h hT = (h', [a'])
implies allocated h' = {a'} ∪ allocated h ∧ a' ∉ allocated h
and alloc h hT = (h', None) ⇒ allocated h' = allocated h
and write h a al v h' =⇒ allocated h' = allocated h

locale m-allocated = allocated + mthr: multithreaded +
assumes t ⊢ (x, h) −ta→ (x', h') ⇒ allocated h ⊆ allocated h'
and [[t ⊢ (x, h) −ta→ (x', h')]; Allocate a hT ∈ set ⟨ta⟩₀]
⇒ a ∈ allocated h' ∧ a ∉ allocated h
and [[t ⊢ (x, h) −ta→ (x', h'); a ∈ allocated h'; a ∉ allocated h]]
⇒ ∃hT. Allocate a hT ∈ set ⟨ta⟩₀
and [[t ⊢ (x, h) −ta→ (x', h'); ⟨ta⟩₀[i] = Allocate a hT; i < |⟨ta⟩₀|];
⟨ta⟩₀[i] = Allocate a hT'; j < |⟨ta⟩₀|] ⇒ i = j

Figure 4.21: Locales formalising the set of allocated addresses

Sequential consistency coinductively

For the DRF guarantee, assumption D1 remains to be shown. However, the JMM definition of SC is not amenable to the coinductive definition of ↓ as it relies on the notions of write-seen function and most recent write, which are only defined for complete interleavings. Therefore, I introduce a coinductive version of SC and prove that it adequately models SC.

A snapshot of a sequentially consistent heap (snapshot heap) H is a finite map from locations to values. The function mrw P h α updates the snapshot heap H if α is a write or initialisation action, else leaves H unchanged. The function mrws P folds mrw P over lists of events. An event list as is sequentially consistent (SC') for the snapshot heap H (denoted P, H ⊢ as sc) iff

\[ P, H \vdash [] \text{sc} \]
\[ P, \text{mrw} P H \alpha \vdash \text{as} \text{sc} \]
\[ \alpha = \text{Read} a al v \implies H (a, al) = [v] \]
\[ P, H \vdash \alpha \cdot \text{as} \text{sc} \]
4.3. Java memory model

i.e., the empty list is $SC'$ for all snapshot heaps, and $\alpha \cdot \alpha$ is $SC'$ for $H$ iff $\alpha$ is $SC'$ for the updated snapshot heap $mrw P H \alpha$ and if $\alpha$ reads the value $v$ from a location $(a, al)$, then the snapshot heap $H$ must store $v$ at $(a, al)$.

The next theorem shows that $P, empty \vdash \sqrt{sc}$ and sequential consistency are equivalent under the following assumption:

D3 Initialisations precede reads in $E$. If $\alpha \in R_E$ reads from some location $(a, al)$, then there is a write event $\beta \in W_E$ such that $\beta \leq E \alpha$, $\text{init}_E \beta$, and $(a, al) \in \text{locs } P E \beta$.

Thus, I can use coinduction to show an execution being $SC$.

**Theorem 4.4.** Let ok-init $E$.

(i) If $E$ initialises every location at most once (assumption D2) and $P, empty \vdash E \sqrt{sc}$, then there is a $ws$ such that $P \vdash (E, ws) \sqrt{sc}$ and $(E, ws)$ is $SC$.

(ii) If initialisations precede reads in $E$ (assumption D3) and $P \vdash (E, ws) \sqrt{sc}$ and $(E, ws)$ is $SC$, then $P, empty \vdash E \sqrt{sc}$.

**Proof.** (i): Set $ws \alpha$ to be the most recent write for $\alpha \in R_E$ to location $(a, al)$. $P, empty \vdash E \sqrt{sc}$ ensures that there is a write event for every read, D2 guarantees the existence of the most recent one. Then, $(E, ws)$ is $SC$ by definition.

For $P \vdash (E, ws) \sqrt{sc}$, only condition W2 of well-formedness, i.e., $\alpha$ reads value-written $E (ws \alpha) (a, al)$, is interesting. Let $as$ be the prefix of $E$ up to $\alpha$. From $P, empty \vdash E \sqrt{sc}$, I obtain that $mrws P empty as (a, al) = [v]$ and $\alpha$ reads the value $v$. Since $ws \alpha$ is the most recent write for $\alpha$ in $E$, assumption D2 and $P, empty \vdash E \sqrt{sc}$ ensure that $ws \alpha < \alpha$. Hence, value-written $E (ws \alpha) (a, al) = v$ holds.

(ii): Suppose $P \vdash (E, ws) \sqrt{sc}$ and $(E, ws)$ is $SC$. Let $\alpha \in R_E$ read $v$ from location $(a, al)$, and let $as$ denote the prefix of $E$ up to $\alpha$. Since initialisations precede reads in $E$, the most recent write $ws \alpha$ precedes $\alpha$, i.e. $ws \alpha \leq E \alpha$. Well-formedness condition W2 of $P \vdash (E, ws) \sqrt{sc}$ yields that $v = value-written E (ws \alpha) (a, al)$. Since $ws \alpha$ is the most recent write for $\alpha$ and $ws \alpha \leq E \alpha$, I also have $mrws P empty as (a, al) = [v]$. As this holds for all reads $\alpha$, $P, empty \vdash E \sqrt{sc}$ follows by coinduction. □

---

Assumption D2 is necessary. For example, suppose that $E$ initialises some location $(a, al)$ infinitely often, but there is no Write $a al _$. Then, a read of the default value in $E$ from $(a, al)$ would be $SC'$, but not $SC$, because none of the initialisations is most recent.
Corollary 4.2. Let unique initialisations precede reads (assumptions D2 and D3) and ok-init E. Then, $P, \text{empty} \vdash E \sqrt{sc}$ iff there is a $ws$ such that $P \vdash (E, ws) \sqrt{sc}$ and $(E, ws)$ is SC.

This equivalence holds only if the initialisation of any location $(a, al)$ occurs before the first read from $(a, al)$ in the complete interleaving. For example, the complete interleaving

$$E = [(t, TInit), (t, \text{Read } a \ (\text{Cell } 0 \ 0), (t, \text{New-Arr } a \ \text{Integer } 1)]$$

is SC for $ws = 2$, but not SC', i.e. $\neg P, \text{empty} \vdash E \sqrt{sc}$. The problem is real: Figure 4.9 shows a (non-SC) execution of a type-correct program that violates assumption D3: The initialisation of $(a, \text{Field } F D)$ in l. 3 occurs after the read in l. 2. Thus, in order to exploit this equivalence, I show that initialisations precede reads in SC' prefixes of a complete interleaving:

D3' If a complete interleaving $E \in \mathcal{E}$ has an SC' prefix $\alpha$ followed by a read from $(a, al)$, as initialises $(a, al)$.

Lemma 4.8. Assumption D3' implies D3 for all $E \in \mathcal{E}$ that are SC'.

Initialisations precede reads

Next, I tackle D3' by decomposing it into smaller assumptions that no longer refer to complete interlings, but only to single reductions in the small-step semantics – similar to what I did for assumption D2 above. At the same time, I prove it for a more general class of prefixes such that I can reuse this assumption when proving consistency in §4.3.4.

A heap record $\mathcal{H}$ is a function from locations to sets of values – it records all values that have been written to a location. Similar to mrw, if $\alpha$ is a write or initialisation event, the function $\text{uhr } P \mathcal{H} \alpha$ adds the written value(s) to the heap record $\mathcal{H}$, else it leaves $\mathcal{H}$ unchanged. The function $\text{uhrs } P$ folds $\text{uhr } P$ over lists of events. An event list $\alpha$ is non-speculative with respect to the heap record $\mathcal{H}$ (denoted $P, \mathcal{H} \vdash \alpha \sqrt{ns}$) iff for any read event $\alpha$ in $\alpha$ from any location $(a, al)$, $\alpha$ reads a value that has been written to $(a, al)$ in $\alpha$ before $\alpha$ or that has already been in $\mathcal{H} (a, al)$. Formally:

$$P, \mathcal{H} \vdash [] \sqrt{ns}$$

$$P, \text{uhr } P \mathcal{H} \alpha \vdash \alpha \sqrt{ns} \quad \alpha = \text{Read } a al \ v \implies v \in \mathcal{H} (a, al)$$

$$P, \mathcal{H} \vdash \alpha \cdot \alpha \sqrt{ns}$$
A prefix of a complete interleaving is non-speculative if its list of events is non-speculative with respect to the empty heap record $\lambda_\emptyset$.

A snapshot heap $H$ fits to a heap record $\mathcal{H}$ iff whenever $H$ stores a value for a location $(a, al)$, then $\mathcal{H}(a, al)$ contains that value.

**Lemma 4.9.** If $\alpha_s$ is SC' for $H$ and $H$ fits to $\mathcal{H}$, then $\alpha_s$ is non-speculative with respect to $\mathcal{H}$.

**Corollary 4.3.** An SC' prefix of a complete interleaving is non-speculative.

This corollary shows that it suffices to prove the following assumption D3'' instead of D3'.

D3’’ If a complete interleaving $E \in \mathcal{E}$ has a non-speculative prefix $\alpha_s$ followed by a read from $(a, al)$, $\alpha_s$ initialises $(a, al)$.

To discharge D3’’, it suffices to show that (i) the program cannot make up addresses and (ii) it accesses only the declared fields of objects and cells within the bounds of the array. Then, since the prefix does not speculate, the read can only access an existing member of an address that has been allocated before. Since allocation initialises all fields and array cells, the member therefore must have been initialised.

For (i), I introduce the concept of known addresses. Let $ka :: t \Rightarrow x \Rightarrow addr$ set be another locale parameter which returns the set of addresses that a given thread stores in its local state. A thread learns an address $a$ in an event list $\alpha_s$ iff $\alpha_s$ contains a read event $\text{Read }_a(\text{Addr }a)$ or an allocation $\text{Allocate }_a$, i.e., it either reads the address from some location or allocates it. The function $\text{learns }\alpha_s$ computes the set of learnt addresses from $\alpha_s$. The single-threaded semantics does not invent addresses iff $ka$ satisfies the assumptions of locale $ka$ in Figure 4.22.

In detail, after any reduction step, a thread may only know addresses which it has known before or learnt in this step. Also, a spawned thread may only know those addresses that the spawning thread knows.\(^{36}\) Conversely, known addresses restrict the heap interactions of a thread as follows: It must only read from members of known addresses and whenever it writes an address to a location, it must know the address. Note the asymmetry between reads and writes. It suffices to restrict the

\(^{36}\)I could relax this assumption such that the spawned thread may also know addresses that the spawning thread is learning in the reduction. However, this is irrelevant for JinjaThreads and the above form is simpler.
locale ka-base = multithreaded-base + fixes ka :: 't ⇒ 'x ⇒ 'addr set

locale ka = m-allocated + ka-base +
asumes t ⊢ (x,h) → (x',h') \implies ka t x' ⊆ ka t x \cup learns ⟨ta⟩_o
and \[t ⊢ (x,h) → (x',h'); Spawn t' x'' h'' ∈ set ⟨ta⟩_t\]
⇒ ka t' x'' ⊆ ka t x
and \[t ⊢ (x,h) → (x',h'); Read a al v ∈ set ⟨ta⟩_o\] ⇒ a ∈ ka t x
and \[t ⊢ (x,h) → (x',h'); ⟨ta⟩_o[i] = Write a al (Addr a'); i < |⟨ta⟩_o|\]
⇒ a' ∈ ka t x \cup learns (take i ⟨ta⟩_o)

Figure 4.22: Locale ka formalises that threads do not invent addresses

reads to members at known addresses, but not the writes, because it cannot read from such locations. Moreover, ka allows to write an address that has just been learnt, but not to read from one of its members. The clone implementation requires the former because it copies (i.e., reads and writes) all locations of the object, which may contain arbitrary (unknown) addresses as values. Technically, it would be fine to immediately read from a learnt address, but this would unnecessarily complicate the proofs.

The concept of known addresses naturally extends to multithreaded states and the interleaving semantics. A state s knows the addresses \(\bigcup_{t ∈ \text{dom}(\text{thr}s)} \text{ka t (fst (the (thr s t)))}\), written as \(\text{kas s}\).

**Lemma 4.10.** Let \(s \rightleftharpoons t:ta → s'\) in locale ka. Then \(\text{kas s'} \subseteq \text{kas s} \cup \text{learns} \langle ta \rangle_o\). If \(\text{Read a al v} ∈ \text{set} \langle ta \rangle_o\), then \(a ∈ \text{kas s}\).

Let the recorded addresses \(\text{addr s} \mathcal{H}\) be the set of all addresses in the heap record \(\mathcal{H}\), i.e., \(\text{addr s} \mathcal{H} = \{ a. \exists a' \text{ al. Addr a} ∈ \mathcal{H} (a,a') \}\). Then, the interleaving semantics preserves the invariant that all known or recorded addresses are allocated for non-speculative executions. The proof goes by case analysis of the reduction and induction on the prefixes of \(\langle ta \rangle_o\). It requires that \(\text{acq-events only generates synchronisation events}\).

**Lemma 4.11.** Let \(s \rightleftharpoons t:ta → s'\) in locale ka such that \(P, \mathcal{H} ⊢ ⟨ta⟩_o \sqrt{\text{ns}}\). If \(\text{kas s} \cup \text{addr s} \mathcal{H} \subseteq \text{allocated} (\text{shr s})\), then

\[\text{kas s'} \cup \text{addr (uhrs P \mathcal{H} ⟨ta⟩_o)} \subseteq \text{allocated (shr s')}\]

Now, I can prove that non-speculative prefixes allocate addresses before they read from their locations, which is part (i) of proving D3″.
locale ka-type = ka +

fixes Q :: 't ⇒ 'x ⇒ 'heap ⇒ bool

assumes t ⊢ (x, h) −ta→ (x′, h′) ⇒ h ≤ h′

and \[s −t:ta→ s′; \uparrow Q \uparrow (\text{thr } s) (\text{shr } s); P ⊢ \mathcal{H} ::≤ \text{shr } s; P, \mathcal{H} ⊢ \langle ta\rangle_o \mathcal{V}_{ns}\] 
\[⇒ \uparrow Q \uparrow (\text{thr } s′) (\text{shr } s′) \land P ⊢ \text{hrs } P \mathcal{H} \langle ta\rangle_o ::≤ \text{shr } s′\]

and \[t ⊢ (x, h) −ta→ (x′, h′); Q t x h; \text{Read } a \ a l v ∈ \text{set } \langle ta\rangle_o\] 
\[⇒ \text{typeof-addr } h′ a = [hT]\]

Figure 4.23: Locale ka-type combines known addresses with type information

Let start-\mathcal{H} denote the start heap record \text{hrs } P (\lambda _. \emptyset) \text{start-events}, and define \text{events'} (t, ta) = \langle ta\rangle_o to extract events from a reduction label.

Lemma 4.12. In locale ka, let start-state \text{−ttas}→* s and s −t:ta→ s′ with \text{Read } a \ a l v ∈ \text{set } \langle ta\rangle_o such that

\[P, \lambda _. \emptyset ⊢ \text{start-events} @ \text{concat } (\text{map } \text{events'} \text{ttas}) \mathcal{V}_{ns}.\]

Suppose kas start-state ⊆ allocated start-heap. Then, either

(i) \text{a is preallocated, i.e., Allocate } a \ hT ∈ \text{set start-events for some } hT, or

(ii) some reduction has allocated a, i.e., there are t′, ta′, and hT such that (t′, ta′) ∈ set ttas and Allocate a hT ∈ set \langle ta\rangle_o.

Proof. If a ∈ allocated start-heap, then (i) holds by construction of start-heap. So suppose a ∉ allocated start-heap. Let as abbreviate concat (map \text{events'} ttas). Since allocations write default values, which are never addresses, addr start-\mathcal{H} = \emptyset by definition of start-events. Therefore, kas s ∪ addr (\text{hrs } P start-\mathcal{H} as) ⊆ allocated (shr s) by Lemma 4.11 and induction on start-state \text{−ttas}→* s. In particular, I have a ∈ allocated (shr s), because a ∈ kas s by Lemma 4.10. Since a ∉ allocated start-heap, induction on start-state \text{−ttas}→* s yields (ii) using the assumptions of locale m-allocated. □
Chapter 4. Memory models

and heap extension. Conformance of a heap record $\mathcal{H}$ with respect to a heap $h$, written $P \vdash \mathcal{H} :: \leq h$, denotes that all values in $\mathcal{H}$ conform to the location’s type, i.e.,

$$P \vdash \mathcal{H} :: \leq h \iff (\forall al. \forall v \in \mathcal{H} (a,al). \exists T. P, h \vdash a \cdot al : T \land P, h \vdash v : \leq T)$$

Locale $ka$-type in Figure 4.23 formally connects the three notions, where the new parameter $Q$ abstracts over language-specific conformance conditions:

- Type information grows monotonically.
- Reductions with non-speculative events preserve conformance of states and heap records.
- Any reduction that starts in a conforming state reads only locations that have a type.
- After an allocation event, the address’ type information agrees with the event’s.

For source code, $Q$ is instantiated to the invariant for subject reduction (Theorem 3.4) from Equation 3.7, i.e., $Q t (e, xs) h = \exists E T. P, (E, T), t \vdash (e, xs), h \sqrt{\_}$. For bytecode, $Q$ is bytecode conformance from Equation 3.8, i.e., $Q t (xcp, frs) h = P, \Phi \vdash t : (xcp, h, frs) \sqrt{\_}$.

The locale parameter $ka$ is implemented as follows. For source code, $J.ka t (e, xs)$ consists of the addresses that (i) occur in $e$ or ran $xs$, or (ii) are preallocated (set start-addrs), or (iii) $t2a t$ returns. The latter two are necessary, because JinjaThreads preallocates the system exceptions – for example, throw null reduces to THROW NullPointer, but addr-of-sys-xtcp NullPointer need not be known before and is not learnt – and currentThread returns the address of $t$’s Thread object. Accordingly, in bytecode, $jvm.ka$ includes the exception flag when set, all addresses in local registers and the operand stack in any call frame, and the preallocated addresses and $t2a t$. Then, source code and bytecode satisfy the assumptions of locale $ka$-type for well-formed programs.\(^{37}\)

\(^{37}\)For bytecode, this proof assumes that undefined-Val is no address, because this value initialises the registers of call frames (see Figure 3.22). Theoretically, I could have defined $jvm.ka$ such that it ignores inaccessible registers in call frames as determined by the bytecode verifier. Then, this assumption would not be needed. However, this would severely complicate the proofs in two ways. First, since bytecode satisfies the assumptions
The next lemma shows that locale $ka$-type implies assumption D3’’. Hence, initialisations precede reads in non-speculative prefixes of complete interleavings and Theorem 4.4 and Corollary 4.2 are applicable.

**Lemma 4.13** (Assumption D3’’). In locale $ka$-type, let $P$ be well-formed, $\uparrow Q \uparrow$ (the start-state) start-heap, $kas$ start-state $\subseteq$ allocated start-heap, and $P \vdash$ start-$\mathcal{H}$ :: start-heap. If $E \in \mathcal{E}$, $E[i] = \text{Read } a \text{ al } v$ with $i < |E|$, and $P, \lambda_. \emptyset \vdash$ take $i$ (map snd $E$) $\forall_{ns}$, then there is a $j < i$ such that $E[j]$ initialises $(a, a)$.

**Proof.** Since $E \in \mathcal{E}$, there is $ttas$ such that $\text{start-state} \downarrow ttas$ and $E = \text{start-events} @ \text{concat} (\text{map events} ttas)$. Since $i < |E|$, I can split $ttas$ such that $ttas = ttas' @ (t, ta) \cdot ttas''$, and $\text{Read } a \text{ al } v \in \text{set} (ta)_o$, and $|\text{concat} (\text{map events} ttas')| < i$. Then, there are $s$ and $s'$ such that $\text{start-state} \downarrow ttas' \rightarrow s$ and $s \rightarrow t : ta \rightarrow s'$. Since non-speculative prefixes of executions preserve conformance, $(a, a)$ is typable in $shr s$, i.e., $P, shr s \vdash a : T$ for some $T$. It suffices to show that there is an event Allocate $a \ hT$ in $\text{start-events}$ or $ttas'$ such that $\text{typeof-addr} (shr s) a = [hT]$, because $P, shr s \vdash a : T$ then implies that Allocate $a \ hT$ initialises $(a, a)$.

By Lemma 4.12, the address $a$ has been allocated before, i.e., either in $\text{start-events}$ or in $ttas'$. If Allocate $a \ hT \in \text{set} \ \text{start-events}$, then $\text{typeof-addr} \ \text{start-heap} a = [hT]$, and therefore $\text{typeof-addr} (shr s) a = [hT]$ since $\text{start-heap} \subseteq shr s$. Otherwise, $ttas' = ttas' @ (t', ta') \cdot ttas''$ for some $ttas'$, $t'$, and $ttas''$ such that Allocate $a hT \in \text{set} (ta')_o$ for some $hT$. Hence, there are states $s'$ and $s''$ such that $\text{start-state} \downarrow ttas' \rightarrow s'$, $s' \rightarrow t' : ta' \rightarrow s''$, and $s'' 
\rightarrow ttas'' \rightarrow s$. By assumption of $ka$-type, I obtain $\text{typeof-addr} (shr s'') = [hT]$ and therefore $\text{typeof-addr} (shr s) a = [hT]$ since $shr s'' \subseteq shr s$.

**Sequentially consistent completions**

Now, only assumption D1 remains to be shown, i.e., sequentially consistent prefixes of well-formed executions can be completed sequentially consistently. To that end, I construct a sequentially consistent completion of locale $ka$ only for conformant states, $ka$ must already fix the conformance predicate $Q$ from $ka$-type and restrict its assumptions to conformant states. Consequently, all proofs in $ka$ are burdened with conformance. Second, discharging the assumptions of $ka$ for bytecode requires to exploit conformance. As can be seen in the bytecode type safety proof of Jinja and JinjaThreads, good automation is hard to achieve for bytecode conformance.
Chapter 4. Memory models

$scc s H$ that starts with a multithreaded state $s$ and a snapshot heap $H$. I define $scc$ by corecursion as follows:

$$scc s H = \begin{cases} \text{if } \exists t \; ta \; s -t:ta \rightarrow s' \text{ then let } (t, ta, s') = \varepsilon(t, ta, s') \land P, H \vdash \langle ta \rangle_o \forall_{sc} \in (t, ta) \cdot scc s' (\text{mrws } P H \langle ta \rangle_o) \text{ else } [] \end{cases}$$

In order to prove anything about $scc s H$, I must make sure that the predicate to the $\varepsilon$-operator is satisfiable for all reachable configurations. Hence, I presume for now the following:

D4 The interleaving semantics satisfies the cut-and-update property for the start state $\text{start-state}$ and the start snapshot heap $\text{start-H} = \text{mrws } P \text{ empty start-events}$.

The cut-and-update property (C&U) for $s$ and $H$ (denoted $C&U s H$) denotes the following. Let the state $s'$ be reachable from $s$ via an $\text{SC'}$ prefix of a complete interleaving, say $s -ttas \rightarrow^* s'$ such that $P, H \vdash \langle \text{map events'} ttas \rangle \forall_{sc}$, and let $H'$ denote the updated snapshot heap $\text{mrws } P H \langle \text{map events'} ttas \rangle$. Then, for every reduction $s' -t':ta' \rightarrow s''$ from $s'$, there are $ta''$, $s'''$ such that (i) $s' -t':ta'' \rightarrow s'''$, (ii) $P, H' \vdash \langle ta'' \rangle_o \forall_{sc}$, and (iii) $P, H' \vdash \langle ta' \rangle_o \approx \langle ta'' \rangle_o$ (to be explained in a moment).

Conditions (i) and (ii) predicate that all reachable, non-stuck states can reduce with events $\langle ta'' \rangle_o$ that are $\text{SC'}$ w.r.t. the current snapshot heap $H'$; they suffice to prove that $scc$ does compute an $\text{SC'}$ interleaving (Lemma 4.15). In condition (iii), $P, H' \vdash as \approx as'$ denotes that two event lists $as$ and $as'$ consist of the same events up to the first $\text{SC'}$ inconsistent read in $as$ (if any) and $as'$ continues with a read from the same location. With condition (iii), given a complete interleaving that is $\text{SC'}$ up to a read $\alpha$, I can cut the interleaving after $\alpha$, replace $\alpha$ with a read of the most recent value, and continue the interleaving $\text{SC'}$.

Lemma 4.14 (Preservation of C&U). If $C&U s H$, $s -t:ta \rightarrow s'$, and $P, H \vdash \langle ta \rangle_o \forall_{sc}$, then $C&U s' (\text{mrws } P H \langle ta \rangle_o)$.

Proof. This holds by definition of $C&U$ because every state that is reachable via $\text{SC'}$ executions from $s'$ is also reachable via $\text{SC'}$ executions from $s$ by prefixing the $\text{SC'}$ reduction $s -t:ta \rightarrow s'$.
4.3. Java memory model

Under assumption D4, scc computes an SC’ execution (Lemma 4.15). By the equivalence of SC and SC’ (Theorem 4.4), I then discharge the main assumption of the DRF proof (Theorem 4.5).

Lemma 4.15.
If $C\&U s H$, then $s \downarrow scc s H$ and $P, H \vdash concat (map events’ (scc s H)) \sqrt{sc}$.

Theorem 4.5 (SC completion). Let $E \in \mathcal{E}$, $P \vdash (E, ws) \sqrt{sc}$, $(E, ws)$ be SC up to a read event $(t, \text{Read } a \text{ al } v)$, say $E = E_1 \circ (t, \text{Read } a \text{ al } v) \cdot E_2$ with $ws \alpha$ being the most recent write for all reads $\alpha \in \mathcal{A}_{E_1}$. Then, there are $E_3$, $v'$, and $ws'$ such that $E^* := E_1 \circ (t, \text{Read } a \text{ al } v') \cdot E_3 \in \mathcal{E}$, $P \vdash (E^*, ws') \sqrt{sc}$, and $(E^*, ws')$ is SC.

Proof of Lemma 4.15. I show $s \downarrow scc s H$ by coinduction with $C\&U s H$ as the coinduction invariant. If $s$ is stuck, then $scc s H = []$ and I am done by $\text{Stop}$. Otherwise, conditions (i) and (ii) of C&U ensure that the predicate to Hilbert’s choice is satisfiable. Hence, it does pick an SC’ reduction step $s \leftarrow t: t \rightarrow s'$ and updates $H$ to $H' := \text{mrws } P H \langle ta \rangle_0$. Note how this mimics $\text{Step}$. Since SC’ reductions preserve C&U (Lemma 4.14), and the reduction is SC’, C&U $s$’ $H'$ holds, too. This concludes the coinductive step.

For $P, H \vdash concat (map events’ (scc s H)) \sqrt{sc}$, the standard coinduction rule is too weak because concat is unproductive for any number of consecutive reductions without events. Hence, I derive a custom coinduction rule for $\_ \_ \vdash \_ \_ \sqrt{sc}$ (Lemma 4.16 below), which allows to defer the next step if one decreases in a well-founded relation. Taking as measure the length of the maximal prefix of $scc s H$ for which $\langle ta \rangle_0$ consists of empty lists, I show $P, H \vdash concat (map events’ (scc s H)) \sqrt{sc}$ with the invariant C&U $s$ H like above.

Lemma 4.16 (Strong coinduction rule for $\_ \_ \vdash \_ \_ \sqrt{sc}$). Let $(R_q)_q$ be a family of sets over snapshot heaps and lists of events indexed over a type with a well-founded order $\prec$. Suppose that for all $q$ and $(H, as) \in R_q$, either $as = []$, or there is a $q'$ with $q' \prec q$ and $(H, as') \in R_{q'}$, or $as$ can be split in $as'$ and $as''$ such that $as' \neq []$ and $P, H \vdash as' \sqrt{sc}$ and if $as'$ is finite, there is a $q'$ such that $(\text{mrws } P H \langle as', as'' \rangle) \in R_{q'}$ or $P, \text{mrws } P H as' \vdash as'' \sqrt{sc}$. If $(H^*, as^*) \in R_{q^*}$ for some $q^*$, then $P, H^* \vdash as^* \sqrt{sc}$.

Proof. By well-founded induction on $\prec$, I prove that the union $R$ of all $R_q$ satisfies the following: For all $(H, as) \in R$, either $as = []$, or $as$ can
be split in \( \alpha' \) and \( \alpha'' \) such that \( \alpha' \neq [] \) and \( P, H \vdash \alpha' \sqrt{sc} \) and if \( \alpha' \) is finite, \((\text{mrws } P H \alpha', \alpha'') \in R \) or \( P, \text{mrws } P H \alpha' \vdash \alpha'' \sqrt{sc} \). This proof follows the same pattern as for the strong coinduction rule in §1.4.3.

Then, I show \( P, H^* \vdash \alpha^* \sqrt{sc} \) by coinduction on \( P, \_ \vdash \_ \sqrt{sc} \) with coin-
duction invariant \((H^*, \alpha^*) \in R\). □

Proof of Theorem 4.5. Construct \( E_3 \) as follows: First, identify the reduction \( s \xrightarrow{-t} t'a \) that generates \( (t, \text{Read } a al v) \). Let \( E_1' \) be the prefix of \( E \) up

to events \( (t, t'a) \) exclusively, which is also a prefix of \( E_1 \). Since all

reads in \( E_1 \) (and thus \( E_1' \)) see the most recent write, \( E_1' \) is SC' by The-

orem 4.4. Since C&U holds for the start state and the start snapshot heap and SC' reductions preserve C&U (Lemma 4.14), C&U holds for \( s \) and \( H_1 = \text{mrws } P \text{ empty } E_1' \), too. Hence, by C&U, there are \( t'a' \) and \( s'' \) such that \( s \xrightarrow{-t} t'a' \rightarrow s'' \), \( P, H_1 \vdash \langle t'a \rangle_0 \sqrt{sc} \), and \( P, H_1 \vdash \langle t'a \rangle_0 \approx \langle t'a' \rangle_0 \).

From the latter, I know that \( \langle t'a \rangle_0 \) and \( \langle t'a' \rangle_0 \) are the same up to the read \( \text{Read } a al v \) in \( \langle t'a \rangle_0 \) (exclusively), which is \( \text{Read } a al v' \) in \( t'a' \) for the SC'-correct value \( v' \). Now, choose \( E_3 \) to be the rest of \( \langle t'a \rangle_0 \) followed by

\( \text{concat } (\text{map events } (\text{sc } s (\text{mrws } P H_1 \langle t'a \rangle_0))) \).

With Lemma 4.15, I get that \( E^* \) is SC' and \( E^* \in \mathcal{E} \). Theorem 4.4 yields the required \( ws' \). □

Corollary 4.4. Every well-formed program has a well-formed, sequentially consistent execution.

Proof. Set \( E = \text{start-events } \oplus \text{concat } (\text{map events } (\text{sc } s H)) \). Then, \( E \in \mathcal{E} \) and \( P, \text{empty } \vdash E \sqrt{sc} \) by Lemma 4.15 and definition of \( \mathcal{E} \) and \text{start-events}.

By Theorem 4.4, there is a \( ws \) such that \( P \vdash (E, ws) \sqrt{sc} \) and \( (E, ws) \) is SC. □

Cut and update

For the DRF guarantee, it remains to show that the interleaving semantics satisfies C&U for the start state (assumption D4). Similar to initialisations preceding reads, I generalise this property to non-speculative prefixes and reading any previously written value, not only the most recent one. Thus, I can reuse the proof for consistency in §4.3.4.

Formally, the sequential semantics has the generalised cut-and-update property (gC&U) for a state \( s \) and heap record \( \mathcal{H} \) iff for all states \( s' \) reachable from \( s \) in the interleaving semantics with non-speculative events \( \alpha s \) and any reduction \( t \vdash (x, \text{shr } s') \rightarrow (x', h') \) of any thread \( t \) in \( s' \) with \( \text{ok-ta} s' t \), whenever \( \langle t \rangle_0[i] = \text{Read } a al v \) for some \( i < |\langle t \rangle_0| \) such
that \( P, \text{uhrs } P \mathcal{H} \vdash \text{take } i \langle t \rangle_0 \sqrt{\text{ns}} \) and for any value \( v' \in \text{uhrs } P \mathcal{H} (as @ take i \langle t \rangle_0) (a, al) \), there is a reduction \( t \vdash (x, shr s') \to (x'', h'') \) such that \( \alpha@take i \langle t \rangle_0 \), \( i < |\langle t \rangle_0| \leq |\langle t \rangle_0|, \langle t \rangle_0[i] = \text{Read } a al v' \), and \( \text{take } i \langle t \rangle_0 = \text{take } i \langle t \rangle_0 \).

Intuitively, gC&U allows to cut a complete interleaving at any read event in its non-speculative prefix and replace it with a read from the same location that reads any value which has previously been written to that location. This might seem overly complicated, but I cannot require that the updated read reads any arbitrary value, because I want the DRF guarantee to hold for both JMM implementations of the abstract heap model. In particular, the second allows to read only type-correct values. Since non-speculative executions preserve conformance, gC&U holds for any abstract heap model that always allows to read all type-correct values, see Lemma 4.18 below. If I were able to drop this conformance restriction on the read operation in JMM heap implementation 2, I could simply require that every read in every state can be cut and arbitrarily updated, which would considerably simplify the proofs.

**Lemma 4.17 (gC&U implies C&U).** If \( P \) is well-formed, the single-threaded semantics satisfies gC&U for start-state and start-\( \mathcal{H} \), and kas start-state \( \subseteq \) allocated start-heap, and \( \uparrow Q \uparrow (\text{thr start-state}) \) start-heap, then it also satisfies C&U for start-state and start-\( \mathcal{H} \).

**Proof.** Suppose \( \text{start-state } t \vdash \text{take } s' \) such that \( P, \text{start-} \mathcal{H} \vdash \langle t \rangle_0 \sqrt{\text{sc}} \). Let \( as = \text{concat } (\text{map events' } t) \) and \( H = \text{mrws } P \text{ start-} \mathcal{H} \text{ as} \). I must show that there are \( t \text{'s } ta' \text{'s } s'' \) such that \( s \vdash t \vdash \langle t \rangle_0 \sqrt{\text{sc}} \), and \( P, H \vdash \langle t \rangle_0 \approx \langle t \rangle_0 \).

If \( s \vdash t \vdash s' \) originates from ACQUIRE, \( \langle t \rangle_0 \) contains no read by definition of acq-events. Hence, \( ta \text{ and } s' \) themselves serve as witnesses.

Otherwise, \( t \vdash (x, shr s) \to (x', shr s') \) for some thread in \( s \) by NORMAL. By Lemma 4.9, \( P, \text{start-} \mathcal{H} \vdash as \sqrt{\text{ns}} \), i.e., gC&U allows to cut and update \( t \text{'s } ta \text{'s } i \) iteratively as follows: Start with \( ta' = ta \) and consider the first event in \( \langle t \rangle_0 \). If it is an event reading not the most recently written value (according to the snapshot heap \( H \)), change the reduction to the most recently written value using gC&U, then continue with the new reduction for the next event. Otherwise, update the snapshot heap \( H \) for the event and consider the next event. This process terminates after at most \( |\langle t \rangle_0| \) iterations because gC&U bounds the length of the replacement events \( \langle t \rangle_0 \) to that length. The reduction thus obtained serves as witness.
Chapter 4. Memory models

The key step in the iteration is to show that $H$ stores a most recently written value at all. I show similar to Lemma 4.13 that $start-state \rightarrow\text{ttas} \rightarrow^* s$ initialises the location. This ensures that $H$ does store some value $v$ for the location and – since $H$ fits to $\mathcal{H} = uhrs P start-H as - H$ has recorded $v$, too. Hence, gC&U ensures that I can cut and update the reduction as described. □

Now, it remains to show that both source code and bytecode satisfy gC&U, i.e.,

D4’ The interleaving semantics satisfies gC&U for $start-state$ and $start-H$.

Although D4’ is tedious to prove for the layers 2 to 5, from the clone method to the source code semantics and defensive VM, these proofs do not pose any interesting challenges. To abstract over both JMM heap implementations, they make the following assumption about the implementation, which both implementations satisfy by definition:

D5 The heap implementation allows to read any type-correct value, i.e., whenever $hconf h$ and $P, h \vdash a:al : T$ and $P, h \vdash v : \leq T$, then $\text{read } h a al v$.

This assumption is stronger than progress because it allows to read any type-correct value. Since gC&U involves only non-speculative prefixes of executions, locale $ka$-type ensures that these preserve conformance of the heap and the heap record. Therefore, any value that gC&U requires to be read conforms to its type. Thus, I am finally able to conclude that the DRF guarantee holds for source code and bytecode.

Lemma 4.18. In locale $ka$-type and under assumption D5, the sequential semantics for source code (bytecode) satisfies gC&U for the start state $J$-start $P C M vs (jvm$-start $P C M vs)$ and $start-H$ provided that (i) $P$ is well-formed, (ii) $wf$-start $P C M vs$, and (iii) the parameters $vs$ only refer to preallocated addresses, if at all.

Theorem 4.6. The DRF guarantee holds for source code and bytecode.

If the start state satisfies the conditions of Lemma 4.18 and $P$ is correctly synchronised, then every legal execution is SC.
4.3.4 Consistency

In the previous section, I have shown that the JMM allows solely sequentially consistent behaviour for correctly synchronised programs. This proves that the JMM is strong enough to disallow certain undesired behaviours. Conversely, consistency requires that the JMM be not too strong in that for some programs it does not allow any behaviour at all. In this section, I prove that the JMM does assign some legal behaviour to every *well-formed* program, not only to correctly synchronised ones. In particular, I show that any sequentially consistent execution is legal. This is not trivial because in programs with data races, the most recent write for a read need not happen before it. Hence, these data races must be justified.

**Theorem 4.7.** Every well-formed source code and bytecode program has a sequentially consistent execution. Every sequentially consistent execution is legal.

Like in the previous section, I have identified assumptions on the interfaces between different layers of the semantics such that I can conduct the proofs as abstractly as possible. In fact, this section only relies on the properties of the single-threaded semantics from the previous section. All theorems are on the level of the interleaving semantics or on higher ones. Like in §4.3.3, I start at the JMM level with assumptions about complete interleavings and then discharge these assumptions in the levels below.

**At the JMM level, the assumptions are now**

C1 For every sequentially consistent prefix of a well-formed execution $(E, ws)$ with $E \in \mathcal{E}$, there is a complete interleaving $E' \in \mathcal{E}$ with the same prefix and a write seen function $ws'$ such that (i) $P \vdash (E', ws') \sqrt{\bot}$, (ii) for all read actions $r \in \mathcal{A}_{E'}$, if $r$ is in the prefix, then $ws' r = ws r$ else $ws' r \leq_{hb}^{P,E} r$, and (iii) if $E$ continues with an event $\alpha$ directly after the prefix, $E'$ continues with the same $\alpha$, except that if $\alpha$ is a read, it may read a different value.

C2 If a well-formed execution has an SC prefix $\alpha s$ followed by a read from $(a,al)$, $\alpha s$ initialises $(a,al)$.

D2 Every execution initialises every location at most once.
Assumption C1 expresses that I can cut any execution after an SC prefix and continue such that every read in the continuation sees a write that happens before. The second assumption C2 is similar to D3’ with SC’ replaced by SC. Assumptions C2 and D2 ensure that for well-formed executions with an SC prefix followed by a read α, (i) a most recent write α’ exists for α with α’ < α, and (ii) if a write α∗ happens before α, then α∗ < α, too.

**Theorem 4.8.** Under assumptions C1, C2, and D2, every SC execution is legal.

**Proof.** Let $E \in \mathcal{E}$ such that $P \vdash (E, ws) \checkmark$ and $(E, ws)$ is SC. I must justify $(E, ws)$ by a justifying sequence $(E_i, ws_i, C_i, \varphi_i)_i$. For $i \leq |E|$, choose some $(E_i, ws_i)$ with the following properties:

- $E_i \in \mathcal{E}$
- $P \vdash (E_i, ws_i) \checkmark$
- $i \leq |E_i|$
- $\text{take } (i-1) E = \text{take } (i-1) E_i$
- Suppose $i > 0$. If $E_{[i-1]} = \text{Read } a \text{ al } v$, then $E_{i[i-1]} = \text{Read } a \text{ al } v'$ for some $v'$, else $E_{[i-1]} = E_{i[i-1]}$
- For all read events $j \in \mathcal{A}_{E_i}$, if $j < i-1$ then $ws_i j = ws j$, else $ws_i j \leq_{\text{hb}}^{P,E_i} j$

Assumption C1 ensures that such $(E_i, ws_i)$ exist. Set $C_i = \{ j. j < i \}$, i.e., $(E_i, ws_i)$ commits the first $i$ events.

For $i > |E|$, set $E_i = E$ and $ws_i = ws$ and $C_i = \mathcal{A}_E$. Then, the sequence $(E_i, ws_i, C_i, \varphi_i)_i$ justifies $(E, ws)$ where all renamings $\varphi_i$ are the identity.

To illustrate how $(E_i, ws_i, C_i, \varphi_i)_i$ justifies reads which see writes that do not happen before, consider the following program where the write in l. 1 races with the read in l. 2.

\[
\begin{align*}
\text{initially: } x &= 0; \\
1: & \quad x = 1; \\
2: & \quad r = x;
\end{align*}
\]

(P)

Figure 4.24 shows the executions for the two complete interleavings where the thread $t_1$ on the left executes before the one on the right ($t_2$). I wish to justify the SC execution shown in Figure 4.24a. Suppose that we are about to commit the read event, i.e., $i = 5$. Figure 4.24b shows $(E_4, ws_4)$ where the grey area contains all committed events.
4.3. Java memory model

The JMM justification rules allow the write-seen function (dashed arrows) to change only for reads that the previous justifying execution has committed for the first time. Since \((E_5, ws_5)\) commits the read, it must still see the allocation as there are no other writes that happen before the read. Thus, Figure 4.24b also shows \((E_5, ws_5)\). In the next step, \((E_6, ws_6)\) may change the read such that it sees the write, i.e., Figure 4.24a shows \((E_6, ws_6)\). At the same time, it also commits the last event \((t_2, TFinish)\).

This offset explains why the specification of \((E_i, ws_i)\) mostly refers to \(i - 1\). However, one cannot shift the whole sequence by one because the JMM requires that \((E_0, ws_0)\) has not yet committed any events.

The proof that \((E_i, ws_i)\) justifies \((E, ws)\) is tedious and largely uninteresting, except for the case where the \(i - 1\)-th event in \(E\) reads from a write \(w\) that does not happen before. In that case, \((E_i, ws_i)\) changes the write from \(ws_{i-1} \) to \(ws_i\). The legality conditions require that \((E_i, ws_i)\) has already committed both of them, i.e., \(ws_{i-1} < i\) and \(ws_i < i\). As noted above, assumptions C2 and D2 ensure this, because \(ws_i\) is the most recent write and \(ws_{i-1} \) happens before \(i\).

\[\square\]

**Corollary 4.5.** Under assumptions D2, D3’, D4, C1, and C2, every program has a legal execution.

**Proof.** By Corollary 4.4, it has a well-formed SC execution. By Theorem 4.8, this execution is legal. \[\square\]
Next, I show that source code and bytecode satisfy assumptions C1 and C2. Note that the latter is equivalent to D3’ by Theorem 4.4, i.e., Lemma 4.13 discharges it.

Assumption C1 is structured similarly to D1. Thus, I construct a witness execution by corecursion similar to scc, but choose $ta$ such that reads in $⟨ta⟩o$ see writes that happen before them. Since assumptions C2 and D2 ensure that such writes precede the reads in the execution, the prefix up to the read is non-speculative and thus gC&U ensures that such a witness exists. Here, the crucial step is to show that such a write exists. Note that the initialisation exists by assumption D3'' and happens before the read by Lemma 4.5. Then, the $≤_e_o$-maximal write to the location that happens before the read serves as witness. Since the proof structure is similar to sequentially consistent completions (Lemmata 4.14 and 4.15, Theorem 4.5), I omit the details.

4.3.5 Type safety

For the JMM heap implementations, I have claimed in §4.3.2 that only the second leads to a type-safe language. In this section, I substantiate this claim and discuss the relation between type safety and values appearing out of thin air.

First, note that for correctly synchronised programs, the DRF guarantee applies (§4.3.3). Hence, all of their executions are sequentially consistent, and analogous to the type safety proof of sequential consistency, one can show type safety for them. However, the JMM assigns semantics to all Java programs such that type safety and Java’s security promises hold unconditionally.

Now, reconsider the program in Figure 4.12, which Figure 4.25a repeats. Figures 4.25b to Figure 4.25e show the justifying executions for the type unsafe execution in Figure 4.25e under the JMM heap implementation 1. The shaded areas contain the committed events. The trick is to justify the address of the Integer object allocated in l. 5 as an out-of-thin-air value for the data races on x and y. These races have the same pattern as in Figure 4.8, where the JMM is sufficiently strong to disallow out-of-thin-air values. However, in Figure 4.25a, this cycle occurs only if the then branch (l. 5) executes. The justification first executes the else branch (l. 7) until both data races on x and y are committed (Figures 4.25b to 4.25d). Then, the branches are switched (Figure 4.25e) and the address $a$ keeps being passed between the two
data races as an out-of-thin-air value. The then branch could then do almost anything – in the example, it allocates an Integer object. The allocation strategy of the JMM heap implementation 1 uses the same address as the allocation in l. 7 has used in previous executions. Hence, the locations x and y of type String now point to an Integer object, which is type-unsafe.

Note that this problem is not specific to the allocation strategy of the first heap implementation. A similar example can be conceived for any given strategy that allows to allocate objects of different types at the same address. JMM heap implementation 2 circumvents this issue, because Integer and String objects have distinct address spaces.

However, tying addresses to their dynamic type information only treats the symptoms, not the cause. First, the type safety statement under the JMM heap implementation 2 is weaker than for sequential consistency from §4.2, because jmm'-typeof-addr no longer encodes which addresses are allocated. For example, bytecode type safety (Theorem 3.6) states that if a thread terminates with a raised exception a, then typeof-addr h a = \([ClassT\ C]\) such that \(P \vdash C \subseteq^* \text{Throwable}\) in the final heap h. Under sequential consistency, this expresses that a refers to an allocated object. Although this theorem literally holds for any legal execution of the JMM with heap implementation 2, too, typeof-addr h a = \([ClassT\ C]\) no longer implies that a has been allocated during the execution. For example, Figure 4.26 shows a slight modification of Figure 4.25a. A justification analogous to Figure 4.25 allows the execution where r1, r2, x, and y all contain the address that the allocation in l. 5 would return, but without executing l. 5. Thus, this model assumes that all objects already “exist” at the start of the execution, allocation merely picks an “unused” one.

Second, another variation of this program (Figure 4.27) shows how to break the out-of-thin-air guarantee of the JMM. It changes ll. 5 and 7 such that they allocate an array of the same type and length. Then, no heap implementation would be able to disallow x and y pointing to the object allocated in l. 5. Although the out-of-thin-air guarantee is not clearly defined, such a forged pointer could be used to break Java’s security features, which rely on this guarantee. Imagine l. 5 allocates the char array that is to store the contents of a String object. By using malicious code like this, an attacker could obtain a reference to modify the string’s contents, but Java’s security features “depend upon Strings being perceived as truly immutable” [56, §17.5].
initially: $b = \text{false}$; $x = y = \text{null}$;

1: $r_1 = y$
2: $x = r_1$
3: $r_2 = x$
4: if ($b$)
5: $r_3 = \text{new Integer()}$
6: else
7: $r_2 = \text{new String()}$
8: $y = r_2$
9: $b = \text{true}$

(a) Program (repeated from Figure 4.12)

(b) Initial execution in which every write happens before the read that sees it. Commit the writes to and read from location $y$.

(c) Change the read of $y$ in l. 1 to see the write in l. 8, commit the writes to and read from location $x$.

**Figure 4.25:** Justifying executions for the type-unsafe one of the program in (a)
(d) Change the read of \( x \) in l. 3 to see the write in l. 2, commit the writes to and read from location \( b \).

(e) Change the read of \( b \) in l. 4 to see the write in l. 9, which switches the if’s branches such that l. 5 allocates an \texttt{Integer} instead of a \texttt{String}; commit all remaining events.

**Figure 4.25:** Justifying executions for the type-unsafe one of the program in (a)
initially: \( b = \text{false}; x = y = \text{null}; \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( r1 = y; )</td>
<td>2: ( x = r1; )</td>
<td>3: ( r2 = x; )</td>
</tr>
<tr>
<td>4: ( \text{if} \ (\neg b) )</td>
<td>5: ( r2 = \text{new Exception}(); )</td>
<td>6: ( y = r2; )</td>
</tr>
<tr>
<td>7: ( \text{throw} \ r2; )</td>
<td>8: ( b = \text{true}; )</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.26:** A program that the JMM allows to terminate with a raised, but unallocated exception

initially: \( b = \text{false}; x = y = \text{null}; \)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1: ( r1 = y; )</td>
<td>2: ( x = r1; )</td>
<td>3: ( r2 = x; )</td>
</tr>
<tr>
<td>4: ( \text{if} \ (b) )</td>
<td>5: ( r3 = \text{new char}[2]; )</td>
<td>6: ( \text{else} )</td>
</tr>
<tr>
<td>7: ( r2 = \text{new char}[2]; )</td>
<td>8: ( y = r2; )</td>
<td>9: ( b = \text{true}; )</td>
</tr>
</tbody>
</table>

**Figure 4.27:** Variation of Figure 4.25a

This example shows that the out-of-thin-air guarantee is too weak to support the Java’s security architecture [54]. However, this is only a theoretical example, because I do not know of any optimisation in a compiler, a JVM, nor in hardware that could lead to such behaviour. Hence, this should be considered a deficiency of the JMM specification.

### 4.3.6 Discussion

**The JMM formalisation**

**Lessons learnt.** The formalisation in §4.3.2 shows how to connect a Java semantics with the JMM, which has been missing in the literature [8,38,69]. The main insight is that event traces of isolated threads do not suffice to obey the JLS and Java API. Figures 4.13 and 4.14 present examples of hidden communication channels in Java that the JMM inter-thread actions do not capture – although the examples only use Java features that the JMM mentions. In my model, the basic thread actions are the only communication channel between the interleaving semantics and the different threads. With the JMM heap implementation 2, the shared state
remembers only what addresses are fresh. Hence, BTAs suffice to expose all communication channels for the features that JinjaThreads models.\(^\text{38}\)

Most obviously, the JMM misses events for thread interrupts. It predicates that \texttt{Thread.interrupt} “synchronises-with the point where any other thread […] determines that [the thread] has been interrupted” [56, §17.4.4], but there are no designated events for neither thread interruption nor “that point”. Hence, I have added the synchronisation events \texttt{TIntr} and \texttt{TIntrd}. Their duals for non-interruption \texttt{ClearIntr} and \texttt{IsIntrd _False} only exist as BTAs for the interleaving semantics. Similarly, the API of class \texttt{Thread} requires the existence of BTAs to query a thread’s state, e.g., \texttt{ThreadEx}. Thus, a comprehensive model of Java concurrency has to include such BTAs. Previous JMM formalisations [8, 34, 69] did without these, because they omitted interruption and wait sets, but a realistic formalisation like JinjaThreads cannot.

The interesting question was which of these new BTAs should become events that participate in synchronisation and happens-before order. I follow the original JMM in that only \texttt{TIntr} synchronises with \texttt{TIntrd}. In particular, BTAs do not occur in complete interleavings, and therefore, they do not synchronise with any event and need not be committed or justified. Hence, they do not affect the writes that a read may see. I consider this sensible, because I have found it very hard to construct programs that can exploit such additional synchronisation to avoid data races. Typically, other schedules exhibit races in such programs.

Figure 4.28 shows an exception to this. The read in l. 6 executes only if the left thread has spawned \texttt{x} before l. 4 executes, which happens after the write to \texttt{y} in l. 1. Yet, l. 1 does not happen before l. 6 according to the JMM, because l. 4 does not generate any synchronisation event. Hence, the program is not correctly synchronised according to my definitions.

However, there are alternative definitions for data races that do not depend on the happens-before order to approximate time. More intuitively, two conflicting events race iff in some complete interleaving, they are adjacent, i.e., no event occurs between them, and the location is

\(^{38}\)There is still a covert channel via \texttt{hashcode}, because JinjaThreads implements \texttt{hashcode} in terms of the overloaded function \texttt{hash-addr :: `addr ⇒ int}. Hence, a thread can infer how many objects of a given type have already been allocated globally by allocating a new one and taking its hash code. The easiest way to close this channel is to decouple \texttt{hashcode}'s implementation from the concrete addresses; Liu and Moore [104], e.g., implement \texttt{hashcode} such that it always returns 0. Note that this covert channel exists in commercial JVMs, too, and poses problems for security analyses [123].
Chapter 4. Memory models

initially: \texttt{x = new Thread(); y = 0}

1: \texttt{y = 1;}
2: \texttt{x.start();}
3: \texttt{try {}
4: \texttt{x.start();}
5: \texttt{} catch (IllegalThreadStateException e) {
6: \texttt{r = y;}
7: \texttt{}}

Figure 4.28: The race on \texttt{Thread.start} does not eliminate the data race on \texttt{y}

not marked volatile. For simple models of happens-before, both definitions are equivalent [32], but not for Java with implicit communication channels between threads.

The program in Figure 4.28 would be correctly synchronised under the alternative definition, because there is no sequentially consistent interleaving with adjacent conflicting events. I argue that it is correct to not consider this program as correctly synchronised, because thread spawns are a degenerate form of synchronisation, and compilers should not need to respect such forms. In [32], Boehm and Adve have a similar problem with \texttt{trylock} in C++. They restore the equivalence by allowing \texttt{trylock} to fail spuriously. Analogously, one could tweak the \texttt{start} method to fail spuriously, but this would violate the semantics (see the example in Figure 4.14).

Under JMM heap implementation 1, I interleave the execution to obtain sequential consistency for types. This solves the problem of finding a fresh address for memory allocation, as the shared memory stores which addresses are fresh. However, complete interleavings introduce a global notion of time, which typical implementations in concurrent hardware do not provide.

The JMM heap implementation 2 actually does not need to broadcast type information at all, because it partitions the address space by type and array length like in [75], i.e, an address carries complete type information. Still, the interleaving needs to communicate (via shared state) which addresses are fresh. With an allocation operation that non-deterministically returns any address of the correct type, and with the BTAs as additional events, one could eliminate the complete interleaving; new rules at the JMM layer then would ensure that every addresses is allocated at most once. I have not followed this path because I would have had to reimplement all the management facilities that the
interleaving framework already provides. Moreover, the proofs via state invariants and preservation theorems rely on the global notion of time, although the JMM order relations do not when the constrain the possible writes for a read.

Faithfulness of the semantics Aspinall and Ševčík [8] suggested to weaken legality to enable more optimisation without sacrificing the DRF guarantee. Since my proof on the JMM level follows theirs, it also works for their weaker notion of legality. I have not formally checked that JinjaThreads validates all JMM test cases by Pugh et. al. [144]. Torlak et. al. [171] have shown that the original JMM does not validate test cases 19 and 20, but the fix by Aspinall and Ševčík [8] does. Since none of the test cases uses dynamic allocation, spawning nor interruption of threads, nor \texttt{wait} and \texttt{notify}, my formalisation should perform equivalent to the original JMM. With the fix by Aspinall and Ševčík, my formalisation should also validate test cases 19 and 20.

Alas, there are two corner cases where JinjaThreads does not model all allowed behaviours. Remember from Chapter 3 that I do not model spurious wake-ups, which the JLS permits. However, this opens a communication channel between a call to \texttt{notify} and \texttt{wait} that can erroneously make programs correctly synchronised. Figure 4.29 shows an example. There are only two conflicting events, namely the write in l. 1 conflicts with the read in l. 8. If spurious wake-ups are impossible, the following argument shows that l. 1 always happens before l. 8, i.e., there is no data race. The read in l. 8 executes only if the right-hand thread wakes up from the call to \texttt{wait} in l. 6. In the absence of spurious wake-ups, only \texttt{notify} in l. 3 can cause this. Hence, the reacquisition of the monitor \texttt{m} when \texttt{wait} returns happens after the unlock in l. 4, which itself happens after l. 1 by program order. However, suppose the thread on the right wakes up spuriously in l. 6. Then, the execution where the thread on the right runs before the one on the left is sequentially consistent, but the conflicting ll. 1 and 8 are not related by happens-before, i.e., the program is incorrectly synchronised. Note that “the Java coding practice of using \texttt{wait} only within loops that terminate only when some logical condition that the thread is waiting for holds” [56, §17.8.1] eliminates this problem. If this advice is followed, spurious wake-ups obscure deadlocks, because they replace deadlocks by busy waiting (see Figure 7.2 in §7.4 for such a potential deadlock in Java’s class initialisation procedure [56, §17.4.2]).
initially: \( m = \text{new Object(); } x = 0; \)

1: \( x = 1; \)
2: \( \text{synchronized (m) \{ } \)
3: \( m.\text{notify(); } \)
4: \( \} \)
5: \( \text{synchronized (m) \{ } \)
6: \( m.\text{wait(); } \)
7: \( \} \)
8: \( r = x; \)

**Figure 4.29:** Incorrectly synchronised program due to spurious wake-ups

The second corner case arises from inadequate atomicity of execution steps. The interleaving semantics executes every step of the single-threaded semantics atomically, which includes calls to native methods. For \textit{clone}, this assumption is unrealistic. Sun and OpenJDK JVMs allow threads to concurrently manipulate the object that is being cloned. To see this, I ran the following test. One thread clones an object with 1000 volatile fields of type \textit{int} while another changes the first and the last field. Volatility ensures that the program is correctly synchronised and the DRF guarantee applies. Then, cloning is observed as not atomic if the first field of the clone stores the original value, but not the last field. Depending on the hardware and JVM, I observed non-atomicity between 270 and 2.5k times out of 10M. Hence, JinjaThreads’s \textit{clone} implementation should be replaced by a better one.

**Technical changes to the JMM** First, for the DRF guarantee, \textit{all} initialisation events must be synchronisation events, not only those for volatile locations, which follows Aspinall and Ševčík [8]. In contrast to them, I do not need a special initialisation thread (which might run infinitely in the case of an infinite execution), but assign to initialisation events the thread’s ID which created the object. This change is relevant for the final field semantics extension to the JMM, which requires to know which thread created which object [56, §17.5.1].

Second, happens-before for the \textit{wait} method arises not only from the associated unlock and lock events [56, §17.4.5], but calling \textit{interrupt} on the waiting thread synchronises with throwing the \textit{InterruptedException}, too. When a thread in a wait set is both interrupted and notified, the JinjaThreads semantics always respects happens-before, although the JLS does not require this [56, §17.8.1] – see the discussion in §3.2.1.

Third, I do not model thread divergence events. The JMM introduces them to “model how a thread may cause all other threads to stall and
fail to make progress” [56, §17.4.2]. My construction achieves the same via the coinductive definition of \( _\downarrow_ \) and concatenation of event lists in Equation 4.2.

Finally, JinjaThreads models neither final fields nor class initialisation nor finalisation. Hence, I do not model that part of the JMM, either [56, §17.5, §12.4.2, §12.6.1.1] – see §7.4 for how this could be included.

The data race freedom guarantee

Insights The DRF guarantee for Java (§4.3.3) has been formalised before [8, 69] – in fact, I employ the same key ideas for the proof on the JMM level. The novel aspects are that

- JinjaThreads’ JMM formalisation covers dynamic allocation with explicit allocation events and infinite executions, and
- I identify the assumptions of the DRF guarantee on the single-threaded semantics and discharge them for source code and bytecode.

The key insights are the following:

1. The new events for interruption and different kinds of synchronisation do not affect the DRF proof. This suggests that other means of synchronisation that JinjaThreads does not cover, e.g. atomics in java.util.concurrent, do not affect it either.

2. One must find better ways to handle initialisations, as the JMM way severely complicates the proofs. One option is to omit initialisations completely. Instead, a read is allowed to not see any write event if no write to that location has happened before and the value read is the default value. The default value is uniquely defined when types partition the address space as explained above.

3. The equivalence of SC and SC’ (Corollary 4.2) shows that the treatment of initialisations is irrelevant for the DRF guarantee. Hence, one is not constrained when searching for better ways.

Insight 3 a posteriori justifies Aspinall’s and Ševčík’s simpler approach of considering finite prefixes for the purpose of formalising the DRF guarantee [8]. However, it is still insufficient when dealing with the full
JMM. For example, the JMM allows the execution in Figure 4.9, but not some of its prefixes.

Similarly, my DRF proof shows that it would be safe to globally restrict read operations to type-conforming values – for correctly synchronised programs. Subject reduction and preservation proofs would become significantly easier. However, it would disallow some legal executions of programs with data races such as Figure 4.9.

Technical considerations My proof of Theorem 4.3 differs from [8, 69, 115] mainly in the proof of the DRF Lemma 4.4. I adapt the others’ in two respects to deal with explicit initialisations.

First, the others topologically sort \( \leq_{\text{po}} [115] \) or \( \leq_{\text{hb}}^{\text{P,E}} [8, 69] \) first to obtain \( \leq_{\text{eo}}^{E} \), and then take \( \{ \beta. \beta \leq_{\text{eo}}^{E} \alpha \} \) as the prefix for the sequentially consistent execution. I omit the sorting and use the induced total order \( \leq^{E} \) (rather than \( \leq_{\text{eo}}^{E} \)), which does not move initialisation events to the program start.

Second, Manson et al. [115] and Huisman and Petri [69] require a sequentially consistent completion \( E’ \); so do I. However, the former ignore that different initialisation events in the suffix might change the \( \leq_{\text{hb}}^{\text{P,E}} \) relation on the prefix. The latter note this problem, but add an axiom that \( \leq_{\text{hb}}^{\text{P,E}} \) remain unchanged. I circumvent the issue by using \( \leq^{E} \) instead of \( \leq_{\text{eo}}^{E} \). Hence, \( \leq_{\text{hb}}^{\text{P,E}} \) on the prefix becomes independent of later initialisations (Lemma 4.6). Aspinall and Ševčík [8] completely avoid it by restricting their model to finite prefixes of executions.

Initialisations also complicate the construction of sequentially consistent completions. I failed to construct them directly, as due to the special treatment of initialisations, ill-formed programs might not have such, see the example below. Hence, I would need appropriate constraints that the semantics preserves, but the JMM notion of execution is unsuitable for preservation proofs. Instead, I proved that sequential consistency with respect to happens-before is the same as for interleaving semantics – if initialisations do not interfere (Corollary 4.2).39 Being operational, interleaving semantics is much more amenable to reduction invariants and their preservation proofs than the JMM. While it is still challenging

39Interestingly, Batty et al. [16, §4] found that initialisations of atomics cause problems in the DRF proof for C++11, too.
4.3. Java memory model

<table>
<thead>
<tr>
<th>initially: x = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: print a.length;</td>
</tr>
<tr>
<td>2: x = 1;</td>
</tr>
<tr>
<td>3: r1 = x;</td>
</tr>
<tr>
<td>4: new int[r1];</td>
</tr>
</tbody>
</table>

(a)

(b)

**Figure 4.30:** An ill-formed program (a) and its execution (b) with a sequentially consistent prefix (grey area) followed by a read (l. 3) that cannot be cut, updated, and completed sequentially consistently.

To show properties about scc, most proofs follow the well-known pattern of preservation.

The program in Figure 4.30a demonstrates that ill-formed programs can have sequentially consistent prefixes of executions which cannot be cut, updated, and completed sequentially consistently.\(^{40}\) Note that the program is ill-formed only because it literally contains an address. However, such a program could well occur as an intermediate state while executing a well-formed program.

In the execution in Figure 4.30b, the read in l. 3 sees the write from l. 2, but the most recent write would be the initialisation of location x. Suppose that l. 3 is scheduled after l. 1, but before l. 2, as indicated by the grey area. Then, the prefix up to l. 1 is sequentially consistent, but has no sequentially consistent completion when l. 3 executes next. If l. 3 is updated to read the initial value 0, then l. 4 allocates an array of length

\(^{40}\)For simplicity, this example abstracts from the type checks that cause the semantics to get stuck.
0 at address \( a \), but line 1 has already output \( a \)’s array length as 1. This violates the JLS that array lengths are always correct [56, §17.4.5]. Note that this problem only arises for JMM heap implementation 1, because implementation 2 uses different addresses for arrays of different lengths.

In this example, the problem is that \( t_1 \) literally contains the address \( a \) that the allocation of the other thread \( t_2 \) returns. The proof of the DRF guarantee relies on the fact that a thread only knows an address if it has allocated it itself or it has read it from memory.

**Consistency, type safety and out-of-thin-air values**

To the best of my knowledge, consistency of the JMM has never been proven formally before. Although the proof itself is unsurprising, consistency is not obvious, because legality is a collection of axiomatic constraints. In particular, it only succeeds because my formalisation omits legality constraint 8 (§4.3.2). Aspinall and Ševčík [8] have already noted this inconsistency. Consistency and the DRF guarantee show the following:

(i) Every program has some legal behaviour (Corollary 4.5), i.e., the JMM constraints are not contradictory.

(ii) The JMM is indeed weaker than sequential consistency, because it allows all SC executions. Together with the introductory example in Figure 4.7, the JMM is strictly weaker.

(iii) For correctly synchronised programs, the JMM is *equivalent* to sequential consistency.

Banning values that appear out of thin air has been an important concern during the last decade – Pugh [145] first noticed the need to ban such values and Manson et al. [115] expand on the issue. The recent standard C++11 bans out of thin air values, too, although informally [72, §29.3.10]. Nevertheless, it is still unclear what actually constitutes a value appearing out of thin air and no formal definition has been found to date. However, one can narrow down this notion from its motivation, namely Java’s type safety and security promises.

Ševčík [160, 161] proves a weak form of out-of-thin-air guarantee: If a program has no means of constructing a value, such a value will never appear in any legal execution. For example, if a program never allocates an object of a class \( C \), then no legal execution may contain a pointer.
to a C object. But this guarantee is too weak to ensure Java’s security promises and type safety, as my examples in §4.3.5 shows. Partitioning the address space by run-time type information rescues type safety, but is nothing more than a quick fix. A real solution is still missing.

4.4 Related work

4.4.1 Memory models and data race freedom

A lot of work has been devoted to hardware MMs, see [3] for an overview. Here, I focus on programming language MMs, which are looser than hardware MMs (and therefore harder to design), because they should be efficiently implementable on various hardware and allow as many compiler optimisations as possible, but nevertheless should be defined unambiguously.

Huisman and Petri [69] have formalised the JMM and the proof of the DRF guarantee in Coq. They have already noted that initialisations break the proof, but added an axiom to avoid the problem. They set out at the abstract level of threads in isolation, without connection to an operational semantics.

Aspinall and Ševčík [8] have formalised parts of the JMM relevant for the DRF guarantee and proved the latter in Isabelle/HOL — which I have found very helpful in extending the DRF guarantee proof. Since they omit dynamic allocation, they need to consider only finite prefixes of executions. This simplifies their proofs considerably, as they do not need to assume that sequentially consistent completion of executions exist. They do not provide an intra-thread semantics, either. Instead, they model a program as an unspecified predicate that checks whether a trace of memory accesses and synchronisation operations represents a valid execution of the thread. This does not suffice to model the hidden communication channels between threads that the JLS specifies (see the examples in Figures 4.13 and 4.14).

For a kernel language, Cenciarelli et al. [38] define an interleaving small-step semantics that generates configuration structures of events which an axiomatic theory constrains. On paper, they show that they only generate behaviours that the JMM allows, but it is unknown whether they produce every allowed behaviour and whether their model satisfies the DRF guarantee.
Chapter 4. Memory models

Torlak et al. [171] developed a model checker for axiomatic memory models. Using whole-program analysis, they derive JMM executions from small Java programs that are restricted to a small (finite) number of heap locations and finite state; loops are unrolled. Thus, their algorithm can compute all inter-thread actions and memory allocations in advance. They focus on checking small test cases rather than providing a full semantics and proofs.

Jagadeesan et al. [75] define an operational semantics for weak MMs with speculative computations similar to the JMM. Instead of validating executions a posteriori, their semantics explicitly encodes permitted reorderings and speculation. Yet, their model is neither machine-checked nor comparable to the JMM for programs with data races and synchronisation. Although they claim that their model bans values out of thin air, the modified example from §4.3.5 also exhibits the out of thin air value in their semantics.

Boyland [34] formalises in Twelf a semantics for a simple language with allocation, synchronisation, volatiles, thread spawns and joins, which may raise an error upon a data race. He shows that a program never raises such errors if it is data-race free in the JMM sense. For programs with data races, the semantics misses many behaviours that the JMM allows, e.g., reorderings as in Figures 4.7 and 4.9, whereas my semantics deals with the full JMM.

The standard C++11 [72] considers programs with data races ill-formed and assigns undefined semantics to them, but offers finer shades of synchronisation than Java. Boehm and Adve [32] describe the MM and prove the DRF guarantee for programs which use only strong synchronisation primitives. They show that such programs are characterised more intuitively as never having conflicting events adjacent in any interleaving. For the JMM, this equivalence does not hold since threads can communicate without introducing happens-before relationships (§4.3.6). Batty et al. [16, 17] have formalised the MM with a focus on rigorously defining the semantics, and proved correct some compilation schemes for synchronisation primitives to assembly code.

Ševčík et al. [163] have verified the CompCert compiler backend with respect to the formal MM for x86 processors by Sewell et al. [164], which is the first formal correctness proof for an optimising compiler backend with respect to a weak MM. They expose the x86-TSO model in a C-like programming language, which is considerably stronger than the JMM and also provides a DRF guarantee.
Various type systems exist to statically ensure that programs are data race free, i.e., the DRF guarantee applies. Flanagan and Abadi [49] came up with an object calculus and a type system with dependent types to ensure that data races in accessing object members cannot occur. Object members are annotated with locks’ names, the type system ensures that accessing a member is only possible if the specified lock is held by the thread. An appropriate subject reduction theorem shows soundness. Flanagan and Freund [50] translated this calculus to full Java bytecode and implemented it in the rccjava tool. In [58], Grossman extends their approaches to multithreaded Cyclone, which is a type safe variant of C (see §3.4.2).

### 4.4.2 Abstract heap modules

As part of the CompCERT project, Leroy and Blazy [100] have formalised a heap model for a subset of sequential C. Similar to my approach, they use Coq’s module system [40] to separate the axiomatic specification from a concrete implementation. Their module signatures consists of the four operations `alloc`, `free`, `load`, and `store`, and two predicates `_ ⊨ _` and `B` on validity of block references and their bounds, respectively. Their JinjaThreads counterparts are `alloc`, `read`, `write`, and `allocated`, respectively. Since JinjaThreads has no garbage collection, `free` has no counterpart. And `B` is remotely similar to the information that `typeof-addr` and `P`, `_ ⊨ _ : _` encode. The four operations are all partial functions, i.e., implementations must be deterministic. Although JinjaThreads has similar operations, the assumptions are different. Leroy and Blazy’s memory module aims to simplify the formal verification of program transformations. Hence, their assumptions mostly describe the interplay between allocations, loads and stores on the level of values. In contrast, most assumptions in JinjaThreads deal with type information. They also implement one concrete representation that distantly resembles Jinja’s heap representation, which has been the basis for §4.2.

Ramananandro et al. [148] verify object layout algorithms for C++ multiple inheritance. They use a concrete low-level memory implementation and abstract over the concrete object layout in terms of a module. Its operations compute offsets to the `this` pointer for fields and casts and must satisfy 26 constraints. They prove that two layout algorithms satisfy these constraints and that an implementation of a high-level heap model similar to one by Wasserrab et al. [178] based on these operations
is sound. JinjaThreads has no low-level implementation of the heap, but the heap module is sufficiently abstract that such an implementation could be shown correct. Then, it would be interesting to show that the low-level implementation is correct with respect to the current high-level implementation.

4.4.3 Modular formalisations

Other theorem provers provide facilities for modular reasoning similar to Isabelle, e.g., Coq [40] and PVS [141]. Since modular developments are consequently ubiquitous, I only list works with a special emphasis on modularity and programming languages. Wasserrab and Lohner [176] decompose the definition and verification of program slicing and checking for non-interference into locales. Like the heap module and the single-threaded semantics in JinjaThreads, they use different interpretations of the locales to easily share definitions and proofs.

Delaware et al. [45] take modularisation to another level. They formalised FeatherweightJava [71] with various language extensions in Coq in the style of product lines. Extensions are modelled as features that can be plugged together to obtain different languages. To that end, they equip all definitions with variation points, i.e., parameters of the modules. Like in JinjaThreads, assumptions about the parameters separate the modules such that the proofs are module-local and theorems compose. While JinjaThreads employs this concept only for the semantics stack, Delaware et al. focus on syntactic language extensions. It would be interesting to see whether their rigorous approach scales to JinjaThreads to separate the extensions of Chapters 3 and 4 from the previous ones.
5

Compiler

JinjaThreads’ compiler from source code to bytecode bridges the gap between the two languages; its correctness proof shows that both fit together. More precisely, in this chapter, I extend Jinja’s non-optimising compiler [83, §5] to JinjaThreads and prove that it is

- **type-preserving** It compiles well-formed source code into well-formed bytecode (Theorem 5.14).

- **semantics-preserving** If the source program terminates or diverges or deadlocks, then so does the compiled program and vice versa (Theorem 5.17). In any case, the intermediate output is the same and all terminated threads have terminated in the same way (normally or by throwing a certain exception). In particular, source code and compiled code have the same set of legal executions under the JMM (Theorem 5.18).

- **DRF-preserving** The compiled program is correctly synchronised iff the source code is (Corollary 5.4).

The proof for type preservation follows Jinja’s, because JinjaThreads only extends the Jinja language. In contrast, concurrency changes the semantics drastically and, therefore, pervades the proofs, too. The challenges of semantic preservation are the following:

- **compiling synchronized blocks** is nontrivial in three ways: First, the source code semantics remembers in a `sync (__)` block the monitor address whereas bytecode must cache it in a local register. Second,

---

41 A preliminary version of this chapter has been published in [111].
unlike the bytecode instructions for monitors \texttt{MEnter} and \texttt{MExit}, synchronized blocks enforce structured locking of monitors, i.e., unlocking never fails in source code. Hence, the compiler verification must prove that \texttt{MExit} never fails in compiled programs. Third, the monitor must also be unlocked when an exception abruptly terminates the block. To that end, the compiler adds an exception handler to ensure this.

**small-step semantics** As JinjaThreads only defines a small-step semantics, I cannot verify the compiler against the big-step semantics as Jinja does. This is much harder, because the verification must deal with intermediate states and incomparable granularity of atomic steps. Moreover, on the state representation level, the VM uses explicit call stacks, which would be implicit in the big-step semantics. In contrast, the small-step semantics dynamically inlines method calls, i.e., there is no call stack at all.

**interleavings, deadlocks and nontermination** While sequential programs are typically deterministic, there are many ways in which threads can interleave. The compiler verification must address this, because nondeterminism precludes the standard approach of modelling nontermination as “no behaviour”, and deadlocks are a new form of behaviour. To separate these concerns from sequential challenges, I prove that the compiler preserves the behaviour of single threads that is observable to other threads. Since this includes synchronisation and heap access, I conveniently deal with interleavings and deadlock on the abstract level of the interleaving framework, which both source code and bytecode instantiate.

The JMM plays only a minor role in the verification, because the compiler does not optimise. Consequently, DRF preservation follows easily from semantic preservation, because it preserves the set of complete interleavings (§5.1.4).

To deal with nondeterminism, I follow a bisimulation approach (§5.1). As notion of bisimulation, I use delay bisimulation with explicit divergence, which is sufficiently strong to preserve deadlocks and nontermination and to support thread-local reasoning. Bisimulation also addresses the granularity of atomic steps, because it allows the compiler to introduce or eliminate internal computation steps that other threads cannot observe.
Figure 5.1: Structure of the compiler (top row) and its verification. Each column corresponds to a semantics or bisimulations ($\approx$), the rows represent different levels ($m = \text{multithreaded}$, $t = \text{single-threaded}$, $e = \text{expressions}$). The grey areas group the semantics by the language that they belong to.

JinjaThreads’ compiler $J2JVM$ operates in two stages: The first stage $\text{comp}P_1$ allocates local variables to registers ($\S 5.3$), the second $\text{comp}P_2$ generates the bytecode instructions ($\S 5.4$). Figure 5.1 shows its structure and the different semantics that the verification involves.

Although there is just one intermediate language $J_1$, the verification spans five different semantics: For source code, I first ($\S 5.2$) develop a small-step semantics $J_0$ that makes call stacks explicit like in bytecode and prove it bisimilar to the source code semantics from $\S 3.2.2$. To deal with the difficulty that $\text{MExit}$ can fail, there are two semantics for the intermediate language: $J'_1$ allows $\text{sync (}_ -$ blocks to fail upon unlocking, whereas $J''_1$ does not. For bytecode, I choose the defensive VM because it gets stuck in ill-formed states. The aggressive VM would carry on with undefined behaviour, which the source code semantics cannot simulate.\footnote{Sometimes, the defensive VM gets stuck earlier than the source code semantics. To avoid problems in the simulation proof, I take a detour via a semi-aggressive VM $\text{exec-meth}$ and exploit that the defensive and aggressive VM commute (Theorem 3.6). See $\S 5.4.3$ for...}
The main verification effort is on the level of expressions and statements (last row in Figure 5.1), which contains all execution steps of a single thread except for calls to and returns from non-native methods. The bisimulation relations on this level are marked with “e”.

The next group of semantics lifts the expression level semantics to call stacks and adds method calls and returns. This level (marked with “t”) corresponds to the single-threaded semantics $J$-red and $jvm$-execd for source code and bytecode, respectively. Finally, the multithreaded semantics models the full behaviour for multithreaded programs. In all languages, this is the interleaving semantics instantiated with the appropriate single-threaded semantics. The legality constraints of the JMM, which reside even higher in the stack of semantics, are not affected because the compiler preserves the set of complete interleavings (§5.1.4).

### 5.1 Semantic preservation via bisimulation

In this section, I define semantic preservation (§5.1.1), introduce delay bisimulations with explicit divergence as proof tool (§5.1.2), and show how preservation for single threads extends to the interleaving semantics (§5.1.3) and the JMM (§5.1.4).

#### 5.1.1 Semantic preservation

Semantic preservation aims to show that semantic properties established on the source code also hold for the target code and vice versa. Such properties or specifications, e.g., a safety property like no null pointer exceptions, are typically modelled as predicates on the traces of observable behaviour, i.e., the sequence of observable steps of a program execution, or on the sets of possible traces (for nondeterministic programs). Thus, a correct compiler $\text{Comp}$ must ensure that the (sets of) traces of the source program $P$ and of the compiled program $\text{Comp} P$ are equal.

Formally, $\text{Comp}$ preserves the semantics of $P$ iff the following holds: Let $s_1$ and $s_2$ be the initial states for $P$ and $\text{Comp} P$, respectively. For every execution of $P$ that starts in $s_1$ and terminates in $s'_1$, there must be an execution of $\text{Comp} P$ from $s_2$ to $s'_2$ such that both the executions’
traces and the observable data in \( s'_1 \) and \( s'_2 \) (such as the result values or exceptional termination) are the same. For every infinite execution of \( P \) that starts in \( s_1 \), \( \text{Comp} \ P \) has an infinite execution with the same trace that starts in \( s_2 \). Conversely, every execution of \( \text{Comp} \ P \) from \( s_2 \) must be matched that way by one of \( P \) from \( s_1 \).

From a specification point of view, only the second direction is essential. The interleaving semantics and the memory model only specify the set of allowed behaviours, of which the implementation \( \text{Comp} \ P \) may pick any (non-empty) subset. Still, I show both directions for two reasons.

The main reason is that semantics properties on sets of traces such as possibilistic security properties [118] require both directions. The compiled code must not miss any observable nondeterministic choice, neither may it introduce additional observable behaviour. Some atomic high-level statements are translated into a sequence of simple instructions, which allow more interleavings. A correct compiler must ensure that these new interleavings do not lead to new behaviours. Conversely, some constructs (like exception handling) are atomic in the compiled code, but require many steps in the source code semantics. Although the compiled code has consequently less interleavings, no observable behaviour must be missed. Being part of the Quis Custodiet project [147], JinjaThreads serves as the semantics basis for verifying concurrent information flow control algorithms, e.g., for possibilistic non-interference [53]. Hence, to transfer such results between source code and bytecode, it is essential that verification covers both directions.

On the technical level, a second reason is that the interleaving framework defines deadlocks in terms of the semantics (§3.3) whose preservation requires both directions, too.

Regarding schedulers, semantic preservation is possibilistic: The source and compiled program may have different behaviour under a fixed scheduler whose strategy depends on unobservable steps. Under a round-robin scheduler, e.g., the number of unobservable steps between two observable ones influences the interleaving. Since a compiler changes this number, source code and bytecode may have different behaviours under this scheduler. In this sense, semantic preservation means: If there is a scheduler for \( P \) such that \( s_1 \) produces trace \( t \) and either terminates in \( s'_1 \) or runs infinitely, then there is also a scheduler for \( \text{Comp} \ P \) such that \( s_2 \) produces trace \( t \) and either ends in \( s'_2 \) or runs infinitely, respectively.
5.1.2 Simulation properties

For semantic preservation, I must show trace equivalence for the source code and the compiled code. To do this, it is standard to show bisimilarity. The latter implies trace equivalence and can be shown by inspecting individual steps of execution instead of whole program executions. For the verification, I have chosen delay bisimilarity [1,119] augmented with explicit divergence [27], because multithreaded states are delay bisimilar with explicit divergence if each of their threads is so. As this notion is transitive, I can decompose the compiler into smaller transformations and verify each on its own. Transitivity ensures that the overall compiler is correct, too.

In this setting, programs define labelled transition systems (LTS) whose states are the program states and whose labels constitute the observable behaviour. I write \( s \overset{t}{\rightarrow} s' \) for a single transition (move), i.e., execution step in the small-step semantics, from state \( s \) to state \( s' \) with transition label \( tl \). Both the semantics \( t \vdash \_ \rightarrow \_ \) of an individual thread \( t \) and the interleaving semantics \( \_ \rightarrow \_ \rightarrow \_ \) fit into this format. A predicate \( \tau\text{-move} ~ s ~ tl ~ s' \) determines whether the transition \( s \overset{tl}{\rightarrow} s' \) is unobservable to the outside world, i.e., other threads for the single-threaded semantics and other processes and the user for the multithreaded semantics. Such transitions are called silent or \( \tau\)-moves. Since their labels are irrelevant, I don’t keep track of them and write \( s \overset{\_}{\rightarrow} s' \) for \( \exists tl. ~ s \overset{tl}{\rightarrow} s' \land \tau\text{-move} ~ s ~ tl ~ s' \). Moreover, \( \_ \overset{\tau}{\rightarrow}^+ \_ \) denotes the transitive closure of \( \_ \overset{\tau}{\rightarrow} \_ \) and \( \_ \overset{\tau}{\rightarrow}^* \_ \) the reflexive and transitive closure. A state \( s \) can diverge (denoted \( s \overset{\_}{\rightarrow} \infty \)) iff an infinite sequence of \( \tau\)-moves starts in \( s \). A visible move \( s \overset{tl}{\rightarrow} s' \) consists of a finite sequence of \( \tau\)-moves followed by an observable transition, i.e., \( s \overset{tl}{\rightarrow} s' \) abbreviates \( \exists s''. ~ s \overset{\_}{\rightarrow}^* s'' \land s'' \overset{tl}{\rightarrow} s' \land \neg \tau\text{-move} ~ s'' ~ tl ~ s' \).

In this chapter, I often have states, labels, reductions, and the like for two or more programs and semantics. To keep the notation simple and clear, I will usually index variables, arrows, etc. with numbers to assign them to one of them, i.e., \( x_1, s_1, t \vdash \_ \rightarrow \_ \rightarrow 1 \_ \_ \) etc. for the first, \( x_2, s_2, t \vdash \_ \rightarrow \_ \rightarrow 2 \_ \_ \) etc. for the second and so on.

A delay bisimulation (with explicit divergence) consists of two binary relations \( \approx \) and \( \sim \) on states and transition labels, respectively, that satisfy the simulation diagrams in Figure 5.2:
5.1. Semantic preservation via bisimulation

![Simulation diagrams for delay bisimulations with explicit divergence. Solid lines denote assumptions, dashed lines conclusions.](image)

(a) An observable move is simulated by a visible move such that \(\approx\) relates the resulting states and \(\sim\) relates the transition labels.

(b) A \(\tau\)-move is simulated by a finite (possibly empty) sequence of \(\tau\)-moves such that \(\approx\) relates the resulting states.

(c) \(\approx\) relates only states of which either both or none can diverge.

Locale \(\text{dbisim-div}\) in Figure 5.3 formalises this notion.

Two programs, i.e., transition systems, are (delay) bisimilar (with explicit divergence) iff there exists a delay bisimulation with explicit divergence for them that relates their start states. A special case of delay bisimulation is strong bisimulation [120] where every move is simulated by exactly one move. When \(\sim\) is obvious from the context, I sometimes omit it and refer to \(\approx\) as a delay bisimulation.

Note that condition (b) does not imply condition (c) because of the classic infinite stuttering problem. Infinitely many \(\tau\)-moves may be simulated by no move at all.

Figure 5.4 shows two LTSs with states \(\{s_1, s'_1, s''_1\}\) and \(\{s_2, s'_2, s''_2\}\), respectively, and a delay bisimulation with explicit divergence \((\approx, \sim)\) between them. The upper LTS with start state \(s_1\) can delay arbitrarily long the decision whether to diverge or to produce the observable transition with label \(tl_1\), whereas the lower LTS, whose start state is \(s_2\), must decide immediately. Nevertheless, they are delay bisimilar with
type_synonym ('s', 't') lts = 's ⇒ 't ⇒ 's ⇒ bool

locale dbisim-base =
  fixes →₁ :: ('s₁', 'tl₁') lts and →₂ :: ('s₂', 'tl₂') lts
  and ≈ :: ('s₁', 's₂') bisim and ~ :: ('tl₁', 'tl₂') bisim
  and τ-move₁ :: ('s₁', 'tl₁') lts and τ-move₂ :: ('s₂', 'tl₂') lts

locale dbisim-div = dbisim-base +
  assumes simulation₁: [...]
  and simulation₂: [...]
  and simulation-τ₁: [...]
  and simulation-τ₂: [...]
  and bisim-diverge: [...]

locale dbisim-final = dbisim-base +
  fixes final₁ :: 's₁ ⇒ bool and final₂ :: 's₂ ⇒ bool
  assumes final₁-simulation:
  and final₂-simulation:

Figure 5.3: Locale dbisim-div formalises the notion of delay bisimulations with explicit divergence; locale dbisim-final defines preservation of final states

Figure 5.4: Example of a delay bisimulation with explicit divergence that is not a well-founded delay bisimulation
5.1. Semantic preservation via bisimulation

explicit divergence because divergence (Figure 5.2c) is a trace property and thus independent of the visited states.

A delay bisimulation \( (\approx, \sim) \) preserves final states iff whenever one of the related states is final, then the other can reach a final state via \( \tau \)-moves. Locale \( \text{dbisim-final} \) in Figure 5.3 formalises this notion, Figure 5.5 shows the simulation diagrams. Note that a delay bisimulation with explicit divergence preserves final states if finality coincides with being stuck. However, in general, not all stuck states are final. For example, deadlocked states in the multithreaded semantics are stuck, but not final – and so may type-incorrect states. Preservation of final states ensures that compiled programs whose source code terminates in a final state also terminate in a final state, and vice versa.

**Lemma 5.1 (Transitivity of delay bisimulations).** Let \( (\approx_1, \sim_1) \) and \( (\approx_2, \sim_2) \) be delay bisimulations with explicit divergence. Then, their composition \( (\approx_1 \circ \approx_2, \sim_1 \circ \sim_2) \) (denoted \( (\approx_1, \sim_1) \circ_B (\approx_2, \sim_2) \)) is also a delay bisimulation – where \( \circ \) denotes relational composition, i.e., \( x R \circ S z \) for binary relations \( R \) and \( S \) iff there is a \( y \) such that \( x R y \) and \( y S z \).

If both \( (\approx_1, \sim_1) \) and \( (\approx_2, \sim_2) \) preserve final states, so does their composition.

**Proof.** Aceto et al. [1] showed this for delay bisimulations without explicit divergence, i.e., the simulation diagrams in (a) and (b) of Figure 5.2. The simulation diagrams in (c) are straightforward to prove.

For preservation of final states, suppose \( s_1 \approx_1 \circ \approx_2 s_3 \) and \( \text{final}_1 s_1 \). Hence, \( s_1 \approx_1 s_2 \) and \( s_2 \approx_2 s_3 \) for some \( s_2 \). By assumption, there is a \( s'_2 \) with \( s_2 \xrightarrow{\tau} s'_2 \) and \( s' \approx_1 s'_2 \) and \( \text{final}_2 s'_2 \). By induction on \( s_2 \xrightarrow{\tau} s'_2 \), obtain \( s'_3 \) such that \( s'_2 \approx s'_3 \) and \( s_3 \xrightarrow{\text{simulation-}\tau} s'_3 \), using \( \text{simulation-}\tau \) in the inductive step. Since \( s'_2 \) is \( \text{final}_2 \), there is an \( s''_3 \) with \( s'_3 \xrightarrow{\tau} s''_3 \) and \( s'_2 \approx s''_3 \) and \( \text{final}_3 s''_3 \). Then, \( s_3 \xrightarrow{\text{simulation-}\tau} s''_3 \) and \( s_1 \approx_1 \circ_B \approx_2 s''_3 \) and \( \text{final}_3 s''_3 \) by transitivity and definition, which
Chapter 5. Compiler

Figure 5.6: Simulation diagrams for well-founded delay bisimulations

concludes this direction. The other direction that starts from final $s_3$ follows by symmetry.$^{43}$

Explicit divergence violates the approach of inspecting individual steps of execution, because divergence consists of infinitely many steps. Hence, it is difficult to prove delay bisimilarity with explicit divergence directly. Instead, I adapt Leroy’s notion of star simulation [97] as follows:

Let $\prec_1$ and $\prec_2$ be two well-founded binary relations on states. $(\approx, \sim)$ is a well-founded delay bisimulation iff it satisfies the simulation diagrams in Figure 5.6:

(a) Observable moves are simulated as in delay bisimulations with explicit divergence.

(b) A $\tau$-move $s_i \xrightarrow{\tau} s'_i$ ($i \in \{1, 2\}$) is either simulated by a finite non-empty sequence of $\tau$-moves, or by no move at all. In the latter case, the $\tau$-move being simulated must descend in $\prec_i$, i.e., $s'_i \prec_i s_i$.

Since $\prec_1$ and $\prec_2$ are well-founded, i.e., there are no infinitely decreasing chains, the infinite stuttering problem cannot occur. Proving well-founded delay bisimulation is easier than delay bisimulation with explicit divergence because all assumptions only involve a single transition.

Proofs by symmetry in this chapter are not only a matter of presentation. I appeal to symmetry also in the Isabelle formalisation. This way, I had to write detailed proofs for only one direction.

\footnote{Proofs by symmetry in this chapter are not only a matter of presentation. I appeal to symmetry also in the Isabelle formalisation. This way, I had to write detailed proofs for only one direction.}
5.1. Semantic preservation via bisimulation

The next lemma shows that well-founded delay bisimulation is at least as strong as delay bisimulation with explicit divergence, i.e., I can use the former whenever I need the latter.

**Lemma 5.2.** Let \( (\approx, \sim) \) be a well-founded delay bisimulation for \( <_1 \) and \( <_2 \). Then, \( (\approx, \sim) \) is a delay bisimulation with explicit divergence.

**Proof.** Since Figure 5.2a and Figure 5.6a are identical and Figure 5.6b trivially implies Figure 5.2b, only the simulation diagrams for divergence (Figure 5.2c) are interesting. The latter directly follow by coinduction on \( s_i \xrightarrow{\tau} j \xrightarrow{\infty} (i \in \{1, 2\}) \) with \( s_1 \approx s_2 \) and \( s_{3-i} \xrightarrow{\tau} j \xrightarrow{\infty} \) as coinduction invariant using the strengthened coinduction rule (see §1.4.3) with \( <_{3-i} \) as well-founded order. \( \Box \)

The converse does not hold, as Figure 5.4 shows.\(^{44} \) The relations \( \approx \) and \( \sim \) as shown form a delay bisimulation with explicit divergence and relate the start states \( s_1 \) and \( s_2 \). However, there is no well-founded delay bisimulation \( (\approx', \sim') \) that relates states \( s_1 \) and \( s_2 \), because \( s_2 \) cannot simulate the \( \tau \)-move \( s_1 \xrightarrow{\tau} s_1 \) according to Figure 5.6b. Clearly, \( \approx' \) cannot relate \( s_1 \) and \( s'_2 \), because \( s_1 \) can produce the observable label \( tl_1 \) and \( s'_2 \) cannot. This excludes the possibility on the left of Figure 5.6b. However, the right one is not feasible, either, as \( s_1 <_1 s_1 \) would violate well-foundedness of \( <_1 \).

The advantage of delay bisimulation with explicit divergence over well-founded delay bisimulation is that only the former is closed under composition (Lemma 5.1), but not the latter. Figure 5.7 shows three LTSs (solid, dashed, and dotted arrows) and two well-founded delay bisimulations (dashed and dotted lines) whose composition is no well-founded delay bisimulation, because there is no suitable well-founded relation \( <'_1 \) for the solid LTS. Suppose \( <'_1 \) were such. Then, for all \( i > 0 \), \( u_i \) can only simulate the \( \tau \)-move \( s_{2i-1} \xrightarrow{\tau} s_{2i} \) by staying at \( u_i \), hence \( s_{2i} <'_1 s_{2i-1} \) by Figure 5.6b. But for all \( j > 0 \), \( u'_{j+1} \) is related to \( s_{2j} \) whose \( \tau \)-move to \( s_{2j+1} \) it can only simulate by staying at \( u'_{j+1} \). Hence, \( s_{2j+1} <'_1 s_{2j} \) by Figure 5.6b. Therefore, \( s_{i+1} <'_1 s_i \) for all \( i \), i.e., \( <'_1 \) contains an infinite descending chain, which contradicts well-foundedness.

This example demonstrates only that well-founded delay bisimulations do not compose. It does not rule out that well-founded delay

\(^{44} \)This example was found by Nitpick [30], a counter example generator for Isabelle/HOL, after several failed attempts of mine to prove equivalence (see §7.2).
Figure 5.7: Three transition systems (solid, dashed, and dotted arrows) and two well-founded delay bisimulations (dashed and dotted lines) whose composition is no well-founded delay bisimulation for the solid and dotted transition systems. All transitions are $\tau$-moves.

bisimilarity is transitive, because there may be delay bisimilarity relations other than the composition $\circ_B$ for which a well-founded relation exists. I have not attempted to prove this, because directly composing bisimulation relations helps in breaking down the ultimate correctness result.

For this chapter, an execution $\xi$ from a state $s$ consists of the labels of a possibly infinite sequence of observable moves and – if these are only finitely many – the terminal state $[s]$ or the special symbol $\text{None}$ (written $\infty$) for divergence. Formally, it is a possibly infinite list of labels where the constructor for the empty list carries a symbol, i.e., elements of the following codatatype:

\[
\text{codatatype} \quad \langle a, b \rangle \text{tlist} = \bot_b | a \cdot \langle a, b \rangle \text{tlist}
\]

\[
\text{type_synonym} \quad \langle s, tl \rangle \text{execution} = \langle tl, \text{option} \rangle \text{tlist}
\]

\[45\] Since Isabelle/HOL has no built-in codatatype support, I construct this type with the quotient package [77] as the quotient of $\langle a \text{ list} \times b \rangle$ through the equivalence relation \{ (($xs, b), (ys, b')$). $xs = ys \land (\text{Ifinite } ys \longrightarrow b = b')$ \} where \text{Ifinite} characterises finite lists.
5.1. Semantic preservation via bisimulation

\[
\begin{align*}
\downarrow \text{terminate} : & \quad s \xrightarrow{\tau} \ast s', \quad \forall tl s'', -s' \xrightarrow{tl} s'' \quad \implies \quad s \downarrow [s'] \\
\downarrow \text{diverge} : & \quad s \xrightarrow{\tau} \infty \\
\downarrow \text{step} : & \quad s \xrightarrow{tl} s', \quad s' \downarrow \xi \quad \implies \quad s \downarrow tl \cdot \xi
\end{align*}
\]

**Figure 5.8:** Executions as traces of observable moves

Like for possibly infinite lists (see Footnote 30), I do not distinguish syntactically such lists from ordinary lists, except for the additional symbol of the empty list constructor. An execution \( \xi \) terminates in the state \( s \) iff \( \xi \) is finite and the empty list constructor in \( \xi \) carries the symbol \([s] \).

The predicate \( s \downarrow \xi \) characterises all executions \( \xi \) that start in \( s \) (Figure 5.8). If \( s \) can reach via \( \tau \)-moves a stuck state \( s' \), then an execution terminates in \( s' \) (\( \downarrow \text{terminate} \)). If an infinite sequence of \( \tau \)-moves starts in \( s \), then an execution diverges (\( \downarrow \text{diverge} \)). If \( s \) can do a visible move with transition label \( tl \) to some state \( s' \), then \( s' \)'s executions include \( s' \)'s prepended with \( tl \) (\( \downarrow \text{step} \)).

Let \([\approx, \sim]\) denote the point-wise extension of \((\approx, \sim)\) to executions, i.e., \( \sim \) holds point-wise for all list elements and \( \approx \) for the terminal states, if any. In delay bisimilar transition systems with explicit divergence, related states have bisimilar executions, i.e., delay bisimulations imply semantic preservation.

**Theorem 5.1** (Semantic preservation). Let \((\approx, \sim)\) be a delay bisimulation with explicit divergence for the LTSs \((\rightarrow_1, \tau\text{-move}_1)\) and \((\rightarrow_2, \tau\text{-move}_2)\) and \( s_1 \approx s_2 \). Then, the following holds:

(i) Whenever \( s_1 \downarrow_1 \xi_1 \), then \( s_2 \downarrow_2 \xi_2 \) for some \( \xi_2 \) such that \( \xi_1 \mid [\approx, \sim] \xi_2 \).

(ii) Whenever \( s_2 \downarrow_2 \xi_2 \), then \( s_1 \downarrow_1 \xi_1 \) for some \( \xi_1 \) such that \( \xi_1 \mid [\approx, \sim] \xi_2 \).

**Proof sketch.** It suffices to prove (i) because (ii) follows from (i) by symmetry. Since \( \xi_2 \) is existentially quantified in (i), I must first construct it explicitly by corecursion from \( s_1 \) and \( \xi_1 \) before showing \( s_2 \downarrow_2 \xi_2 \) and \( \xi_1 \mid [\approx, \sim] \xi_2 \) by coinduction. However, \( \xi_1 \) is only a trace of transition labels without the intermediate states. Since trace properties are strictly weaker than bisimulation properties, it is too weak to be used as coinduction invariant.
Therefore, I define a variant $\ll' \downarrow$ of $\ll \downarrow$ where $\ll' \text{STEP}$ not only prepends the transition label, but also remembers the intermediate state, i.e.,

$$\ll' \text{STEP}: \frac{s \xrightarrow{t} s' \quad s' \ll' \xi}{s \ll' (tl, s') \cdot \xi}$$

Coinduction (with invariant $s \ll \xi_1$) shows that there is a $\xi'_1$ such that $s \ll' \xi'_1$ and $\xi_1$ is the projection of $\xi'_1$ on the transition labels. Construct $\xi'_2 = \text{simulate} s_2 \xi'_1$ by corecursion as follows:

\[
\text{simulate } s_2 \ll s_2' = \begin{cases} 
\ll' s_2' \text{ s.t. } (\forall tl_2 s''_2'. \neg s''_2' \xrightarrow{tl_2} s'_2) \wedge s'_1 \approx s'_2 \text{ in } \ll s_2'
\end{cases}
\]

\[
\text{simulate } s_2 \ll s_2' = \begin{cases} 
\ll' s_2' \text{ s.t. } (\forall tl_2 s''_2'. \neg s''_2' \xrightarrow{tl_2} s'_2) \wedge s'_1 \approx s'_2 \text{ in } \ll s_2'
\end{cases}
\]

Then, I prove by coinduction with invariant $s_1 \approx s_2$ and $s_1 \ll' \xi'_1$ that $s \ll' \xi'_2$ and that $\xi'_1$ and $\xi'_2$ are related point-wise by $\approx$ and $\sim$. Hence, choose $\xi'_2$ to be the projection of $\xi'_2$ to transition labels. □

**Theorem 5.2** (Preservation of final states). Let $(\approx, \sim)$ preserve final states. Suppose $s_1 \ll_1 \xi_1$ and $s_2 \ll_2 \xi_2$ such that $\xi_1 [\approx, \sim] \xi_2$. Then, $\xi_1$ terminates in a final$_1$-state iff $\xi_2$ terminates in a final$_2$-state.

### 5.1.3 Lifting simulations in the interleaving framework

The delay bisimulations for showing semantic preservation always relate multithreaded states. As I use the framework for interleaving semantics at all compilation stages, I uniformly lift delay bisimulations for single threads to multithreaded states. Thus, to show delay bisimilarity on the multithreaded level, it suffices to show delay bisimilarity for single threads plus some constraints that the lifting imposes.

A reduction in the multithreaded semantics is a $\tau$-move (predicated by $m\tau$-move) iff it originates from a $\tau$-move of the single-threaded semantics via Normal. In particular, reacquisition of temporarily released locks
5.1. Semantic preservation via bisimulation

(rule Acquire) is no $\tau$-move, because another thread can observe the lock acquisition by no longer being able to acquire the lock.

First, I lift a relation on thread-local states and the shared heap to multithreaded states $s_1$ and $s_2$ as follows: Let $t \vdash (x_1, h_1) \approx (x_2, h_2)$ denote a bisimulation relation for thread $t$ and $x_1 \approx x_2$ be a relation of thread-local states for threads in wait sets. This induces the following relation $s_1 \approx m s_2$:

(i) Locks, wait sets and interrupts are equal in $s_1$ and $s_2$, i.e., $\text{locks } s_1 = \text{locks } s_2$, $\text{wset } s_1 = \text{wset } s_2$, and $\text{intrs } s_1 = \text{intrs } s_2$.

(ii) All threads in $s_1$ also exist in $s_2$ and vice versa, i.e., $\text{dom } (\text{thr } s_1) = \text{dom } (\text{thr } s_2)$.

(iii) For every thread in $s_1$ and $s_2$, say $\text{thr } s_1 t = \lfloor (x_1, \text{ln } 1) \rfloor$ and $\text{thr } s_2 t = \lfloor (x_2, \text{ln } 2) \rfloor$, the temporarily released locks are the same ($\text{ln } 1 = \text{ln } 2$), the thread-local states related ($t \vdash (x_1, \text{shr } s_1) \approx (x_2, \text{shr } s_2)$), and if $t$’s wait set status is not None, $x_1 \approx x_2$.

(iv) All waiting threads exist, i.e., $\text{dom } (\text{wset } s_1) \subseteq \text{dom } (\text{thr } s_1)$.

(v) There are only finitely many threads, i.e., $\text{finite } (\text{dom } (\text{thr } s_1))$.

Let me now briefly review the constraints. Since threads can observe the status of locks, wait sets and interrupts and the existence of threads, equality in (i) and (ii) ensures that $\approx_m$-related states are indistinguishable to any of their threads. Constraint (iii) imposes the thread-local bisimulation on all thread states. The last condition $x_1 \approx x_2$ imposes stronger simulation properties on threads in wait sets, because the interleaving semantics does not allow a thread to execute $\tau$-moves between its removal from the wait set and the (observable) reduction of processing the removal – with BTAs Notified and WokenUp, respectively. Constraint (iv) ensures that spawned threads are not in a wait set, i.e., their thread-local states need not satisfy the stronger $\approx_w$ relation. Finally, the last constraint (v) ensures that $\approx_m$ preserves divergence.

To see that (v) is necessary, consider, e.g., two infinite pools of threads. In one of them, each thread only does a single $\tau$-move $x_1 \xrightarrow{\tau} x'_1$ before it terminates. In the other, all threads have terminated in state $x_2$. For the (well-founded) delay bisimulation between single threads that relates both $x_1$ and $x'_1$ with $x_2$, constraints (i) to (iv) are satisfied, but the first
type_synonym (l, t, x, h, w, o) \tau\text{-moves} = \hspace{2cm} \\
'x \times h \Rightarrow (l, t, x, h, w, o) \text{ thread-action} \Rightarrow 'x \times h \Rightarrow \text{bool}

locale \tau\text{-multithreaded} = \text{multithreaded} + \hspace{2cm} \\
\text{fixes} \ \tau\text{-move} :: (l, t, x, h, w, o) \tau\text{-moves} \\
\text{assumes} \ \tau\text{-ta} : \tau\text{-move} (x, h) \text{ ta} (x', h') \Longrightarrow \text{ta} = (\emptyset) \\
\text{and} \ \tau\text{-heap} : \texttt{[}[t \vdash (x, h) \text{ ta} (x', h')]; \tau\text{-move} (x, h) \text{ ta} (x', h')]\texttt{]} \Longrightarrow h' = h

Figure 5.9: Locale \tau\text{-multithreaded} enforces that \tau\text{-moves are unobservable

thread pool can diverge (by executing one thread at a time), whereas the second is stuck. Hence, without constraint (v), \approx_m would not be a delay bisimulation with explicit divergence.

The bisimulation \_ \vdash _ \approx _ for single threads also yields the relation on the thread actions as transition labels: \_ \vdash ta_1 \approx ta_2 denotes that ta_1 and ta_2 are equal except for the parameters x_1, h_1 and x_2, h_2 to Spawn t BTAs which must satisfy \_ \vdash (x_1, h_1) \approx (x_2, h_2), i.e., ta_1 and ta_2 may only differ in the initial states of spawned threads, which must be bisimilar.\footnote{This is the second place (besides lifting thread-local well-formedness conditions in §3.1.4) where storing the current heap in the BTA Spawn simplifies the definition, because \_ \vdash _ \approx _ relates pairs of thread-local states and the current heaps.}

The above definition for \approx_m is sensible. If \_ \vdash _ \approx _ \_ \vdash _ \approx _ is a delay bisimulation with explicit divergence, then so is \approx_m. However, this requires additional assumptions that the communication channels through thread actions are respected; Figures 5.9 and 5.10 collect them.

Most importantly, a thread must not be able to observe the \tau\text{-moves of other threads. To that end, I require that \tau\text{-moves neither execute any BTAs, nor change the shared heap, which the locale \tau\text{-multithreaded formalises in Figure 5.9.}

Second, since bisimilarity of threads comprises the shared heap, I require that heap changes preserve \_ \vdash _ \approx _ \_ i.e., the heap changes by one thread must not break bisimilarity of other threads. Assumptionheap-change-preserve of locale m-dbisim-div captures this formally: Let \_ \vdash (x_1, h_1) \approx (x_2, h_2) be two bisimilar states with visible moves to (x'_1, h'_1) and (x'_2, h'_2), respectively, such that \_ \vdash (x'_1, h'_1) \approx (x'_2, h'_2), i.e., the visible moves simulate each other. Then, for any thread \_t', whenever \_t' \vdash (x'_1, h_1) \approx (x'_2, h_2) holds for the old heaps h_1 and h_2, \_t' \vdash (x'_1, h'_1) \approx (x'_2, h'_2) must still hold for the updated heaps h'_1 and h'_2.
5.1. Semantic preservation via bisimulation

locale m-dbisim-div =  
  r1 : τ-multithreaded final1 r1 acq-events τ-move1 +  
  r2 : τ-multithreaded final2 r2 acq-events τ-move2 +  
  fixes _ ⊢ _ ≈_ :: 't ⇒ (x₁ × h₁, x₂ × h₂) bisim and ≈_ :: (x₁, x₂) bisim  
  assumes dbisim-div (r₁ t) (r₂ t) (t ⊢ _ ≈ _) (t ⊢ _ ≈ _) τ-move₁ τ-move₂  
  and dbisim-final (r₁ t) (r₂ t) (t ⊢ _ ≈ _) (t ⊢ _ ≈ _) τ-move₁ τ-move₂  
  (λ(x₁,h₁). final₁ x₁) (λ(x₂,h₂). final₂ x₂)  
and heap-change-preserve:  
  [t ⊢ (x₁,h₁) ≈ (x₂,h₂); t ⊢ (x₁',h₁') ≈ (x₂',h₂') ; t tl₁ ~ tl₂;  
  t ⊢ (x₁,h₁) ⃗[tl₁]₁ (x₁',h₁'); t ⊢ (x₂,h₂) ⃗[tl₂]₂ (x₂',h₂');  
  t' ⊢ (x₁',h₁) ≈ (x₂',h₂)] ⇒ t' ⊢ (x₁',h₁') ≈ (x₂',h₂')  
and ≈₁:  
  [t ⊢ (x₁,h₁) ≈ (x₂,h₂); t ⊢ (x₁',h₁') ≈ (x₂',h₂') ; t tl₁ ~ tl₂;  
  t ⊢ (x₁,h₁) ⃗[tl₁]₁ (x₁',h₁'); t ⊢ (x₂,h₂) ⃗[tl₂]₂ (x₂',h₂');  
  Suspend w ∈ set(ta₁)ₘ; Suspend w ∈ set(ta₂)ₘ]  
  ⇒ (x₁',h₁') ≈ (x₂',h₂')  
and simulation-≈₁:  
  [t ⊢ (x₁,h₁) ≈ (x₂,h₂); x₁ ≈ x₂; t ⊢ (x₁,h₁) -ta₁→₁ (x₁',h₁');  
  Notified ∈ set ⟨ta₁⟩ₘ ∨ WokenUp ∈ set ⟨ta₁⟩ₘ]  
  ⇒ ∃ta₂ x₂' h₂'. t ⊢ (x₂,h₂) -ta₂→₂ (x₂',h₂') ∧  
  t ⊢ (x₁',h₁) ≈ (x₂',h₂') ∧ t ta₁ ~ ta₂  
and simulation-≈₂:  
  [t ⊢ (x₁,h₁) ≈ (x₂,h₂); x₁ ≈ x₂; t ⊢ (x₂,h₂) -ta₂→₂ (x₂',h₂');  
  Notified ∈ set ⟨ta₂⟩ₘ ∨ WokenUp ∈ set ⟨ta₂⟩ₘ]  
  ⇒ ∃ta₁ x₁' h₁'. t ⊢ (x₁,h₁) -ta₁→₁ (x₁',h₁') ∧  
  t ⊢ (x₁',h₁) ≈ (x₂',h₂') ∧ t ta₁ ~ ta₂  
and Ex-final-inv: (∃x₁. final₁ x₁) ↔ (∃x₂. final₂ x₂)

Figure 5.10: Locale m-dbisim-div collects the necessary assumptions for ≈ₘ  
being a delay bisimulation with explicit divergence
Third, \( t \vdash _\_ \approx _\_ \) must preserve final states for all \( t \). This ensures that if a thread \( t \) in state \( s_1 \) successfully joins on another thread \( t' \) in one state (recall that successfullness is determined by \( t'' \)'s local state being final), then any \( \approx_m \)-bisimilar state \( s_2 \) can reach via \( \tau \)-moves a bisimilar state \( s'_2 \) in which \( t'' \)'s local state is also final, i.e., \( t' \)'s join succeeds in \( s'_2 \), too.

Forth, the wait-notify mechanism requires that when a thread has been removed from its wait set, its very next step processes the removal. Therefore, \( \approx_m \) enforces that threads in wait sets are related in \( \approx_w \). The next three assumptions connect \( \approx_w \) with the semantics. Assumption \( \approx_I \) expresses that whenever a thread suspends itself to a wait set, \( \approx_w \) must relate the resulting states. Moreover (assumptions \( \text{simulation-}\approx_1 \) and \( \text{simulation-}\approx_2 \)), \( \approx_w \) is a “one-step” strong bisimulation for processing the removal from the wait set, i.e., whenever one of the related states can process a removal, so can the other (without any intervening \( \tau \)-moves), but only \( t \vdash _\_ \approx _\_ \) needs to relate resulting states.

Last, for technical reasons, \( m\text{-dbisim-div} \) requires that final states exist for either both single-threaded semantics or none. I discuss this assumption in more detail in Footnote 47 when I prove preservation for deadlocks (Theorem 5.4).

Under these assumptions, \( \approx_m \) is a delay bisimulation with explicit divergence that preserves final states.

**Theorem 5.3.** Under the assumptions of locale \( m\text{-dbisim-div} \), \( (\approx_m, \sim_m) \) is a delay bisimulation for \( r1.redT \) and \( r2.redT \) that preserves final states. Formally:

\[
\begin{align*}
(i) \quad & \text{dbisim-div} \ r1.redT \ r2.redT \ \approx_m \sim_m \ r1.m\tau\text{-move} \ r2.m\tau\text{-move} \\
(ii) \quad & \text{dbisim-final} \ r1.redT \ r2.redT \ \approx_m \sim_m \ r1.m\tau\text{-move} \ r2.m\tau\text{-move} \\
& \quad \quad \quad \quad \quad \quad \quad \quad \quad / \quad \quad \quad \quad r1.mfinal \ r2.mfinal
\end{align*}
\]

**Proof.** I show the simulation diagrams from Figures 5.2 and 5.5 for (i) and (ii), respectively. As before, it suffices to show just one direction of each diagram, because the other follows by symmetry.

For an observable move \( s_1 \rightarrow t:ta \rightarrow_1 s'_1 \) and \( \approx_m \)-bisimilar state \( s_2 \) (Figure 5.2a), if the move originates from \textsc{Acquire}, \( s_2 \) can directly simulate it because \( \approx_m \) ensures that the lock states of and \( t' \)'s temporarily released locks and wait set status in \( s_1 \) and \( s_2 \) are equal.

Otherwise (\textsc{Normal}), it originates from some observable move of \( t' \)'s semantics, say \( t \vdash (x_1, shr s_1) -ta \rightarrow_1 (x'_1, shr s'_1) \). Since \( s_1 \approx_m s_2 \), \( t \) exists in \( s_2 \) with local state \( x_2 \) such that \( t \vdash (x_1, shr s_1) \approx (x_2, shr s_2) \).
and – if t’s wait set status is not None – \( x_1 \approx x_2 \). If t’s wait set status is None, bisimilarity of the single threads yields a visible move \( t \vdash (x_2, \text{shr}~s_2) \xrightarrow{ta_2} (x'_2, h'_2) \) such that \( t \vdash (x'_1, \text{shr}~s'_1) \approx (x'_2, h'_2) \) and \( t \vdash ta_1 \sim ta_2 \). Unfortunately, this does not directly translate into a visible move of the interleaving semantics, because \( ta_2 \)'s preconditions need not be met in \( s_2 \). In particular, if \( ta_2 \) joins on a thread \( t' \), i.e., \( \text{Join}~t' \in \langle ta_2 \rangle_c \), \( t' \) may be final in \( s_1 \), but need not be final in \( s_2 \). However, since \( t' \vdash _\vDash \approx _\vDash \) preserves final states, \( t' \) can silently reduce to a final state.

Hence, the simulating visible move consists of

- t’s \( \tau \)-moves,
- the silent reductions to final states of all threads \( ta_2 \) joins on, and
- t’s observable move.

Proving that \( \approx_m \) relates the resulting states \( s'_1 \) and \( s'_2 \) falls in five parts. First, the locks, wait sets, interrupts and domains of the thread pool are equal because \( \tau \)-moves do not change them and the observable moves have identical thread actions except for the initial states of spawned threads. Second, t’s thread-local states \( x'_1 \) and \( x'_2 \) are related as required – assumption \( w \approx I \) establishes \( x'_1 \approx x'_2 \) if \( \text{wset}~s'_1 t \neq \text{None} \), which can only occur if \( ta_1 \) and \( ta_2 \) contain a \( \text{Suspend} \) BTA. Third, \( t \vdash ta_1 \sim ta_2 \) guarantees that local states of newly spawned threads are related – \( \text{dom}(\text{wset}~s_1) \subseteq \text{dom}(\text{thr}~s_1) \) ensures that their wait set status is None, i.e., their local states need not be related in \( \approx_w \). Forth, thanks to assumption \( \text{heap-change-preserve} \), all other threads remain bisimilar. Fifth, finiteness of the thread pool is preserved because only finitely many threads can be spawned in a single step. The case for \( \text{wset}~s_1 t \neq \text{None} \) is analogous except that assumption \( \text{simulation-} \approx_{\vDash} \) yields the simulating move and \( t \) does no \( \tau \)-moves before the observable move.

Simulating a \( \tau \)-move (Figure 5.2b) is easy. It must originate from a \( \tau \)-move in the single-threaded semantics, so there is a simulating sequence of \( \tau \)-moves. Since none of them generates any thread action, \( \text{NORMAL} \) naturally injects them into the interleaving semantics.

To prove preservation of divergence, note that \( \approx_m \) ensures that there are only finitely many threads. Induction on this finite set yields that the interleaving semantics can only diverge if one of its threads can diverge. Conversely, it obviously can diverge if one of the threads can (by coinduction). Putting these arguments together, \( \approx_m \) preserves divergence.
For preservation of final states, remember that all threads in an \textit{mfinal} state are final. So suppose \( r1.m\text{final} s_1 \) and \( s_1 \approx_m s_2 \). By induction on the (finite) set of threads in \( s_1 \), there is a state \( s'_2 \) such that \( s_2 \overset{\tau}{\rightarrow} s'_2 \) and \( s_1 \approx_m s'_2 \) and \( r2.m\text{final} s'_2 \). In the inductive step, the next thread \( t \) can silently reduce to an appropriate final state because \( t \vdash _\approx \_ \) preserves final states. I combine them with the silent moves from the induction hypothesis to obtain the desired reductions. \( \square \)

The proof of Theorem 5.3 motivates why I chose delay bisimulation instead of more common notions of bisimulation like weak bisimulation by Milner [120]. Milner allows observable moves to be simulated by a finite sequence of \( \tau \)-moves, the observable move, and another finite sequence of \( \tau \)-moves. However, there is no analogue of Theorem 5.3 for weak bisimulations, because the observable transition might suspend itself to a wait set, after which the interleaving semantics does not allow any additional \( \tau \)-moves.

\textbf{Theorem 5.4} (Preservation of deadlocks). Under the assumptions of locale \( \text{m-dbisim-div} \), \( \approx_m \) preserves deadlocks. Let \( s_1 \approx_m s_2 \). If \( t \in r1.\text{deadlocked} s_1 \), then \( s_2 \) can reduce via \( \tau \)-moves to some \( s'_2 \) such that \( s_1 \approx_m s'_2 \) and \( t \in r2.\text{deadlocked} s'_2 \). Conversely, if \( t \in r2.\text{deadlocked} s_2 \), then \( s_1 \) can reduce via \( \tau \)-moves to some \( s'_1 \) such that \( s'_1 \approx_m s_2 \) and \( t \in r1.\text{deadlocked} s'_1 \).

\textit{Proof.} By symmetry, it suffices to prove only one direction. So suppose \( t \in r1.\text{deadlocked} s_1 \). Since \( _\approx \vdash _\approx \approx \_ \) preserves final states, there is an \( s'_1 \) which is reachable via \( \tau \)-moves from \( s_2 \) such that \( s_1 \approx_m s'_2 \) and all final threads in \( s_1 \) are final in \( s'_2 \), too (by induction on the finite set of threads). Since \( _\approx \vdash _\approx \approx \_ \) preserves divergence, a similar induction shows that there is an \( s'_2 \) which is reachable via \( \tau \)-moves from \( s'_2 \) such that \( s_1 \approx_m s'_2 \) and all threads in \( s_1 \) that cannot do any \( \tau \)-move cannot do any in \( s'_2 \) either.

Then, I prove \( t \in r2.\text{deadlocked} s'_2 \) by coinduction with invariant \( t \in r1.\text{deadlocked} s_1 \). In the coinductive step, I show by case analysis of \( t \in r1.\text{deadlocked} s_1 \) that each case suffices to prove the corresponding case for \( t \in r2.\text{deadlocked} s'_2 \).

For \( \text{Dactive} \), this follows from the simulation for observable moves. Since \( t \) is deadlocked in \( s'_1 \), it cannot do any \( \tau \)-moves and, therefore, neither can it in \( s'_2 \). Hence, the visible move that simulates a move of \( t \) can only consist of the observable move, but no \( \tau \)-moves. Therefore, \( \approx_m \)
preserves \( t \vdash (\_ \text{shr} s) \_ \tau \) and \( t \vdash (\_ \text{shr} s) \_ \_ \).

Since final threads in \( s_1 \) are final in \( s_2 \), too, and locks and wait sets are equal, must-wait is preserved, too. This concludes this case.

For the cases Dacquire and Dwait, preservation of must-wait and all-final-except is straightforward by the choice of \( s_2' \). Hence, they follow easily. \( \square \)

### 5.1.4 Semantic preservation for the Java memory model

Concerning semantic preservation for the JMM, the JLS only specifies that “an implementation is free to produce any code it likes, as long as all resulting executions of a program produce a result that can be predicted by the memory model” [56, §17.4], i.e., it takes the specification point of view (§5.1.1).

My bisimulation approach for semantic preservation ensures that both the source program and the compiled code have the same set of traces (Theorem 5.1). Since legality of executions only depends on the set of traces, a semantics-preserving compiler is correct for the JMM, too. However, the JMM and semantic preservation refer to different definitions of traces. The former does not distinguish between observable and silent moves, but includes all of them, whereas the latter only consists of observable moves. Fortunately, since \( \tau \)-moves produce no events by assumption \( \tau \text{-ta} \) in Figure 5.9, concatenating all events as in Equation 4.2 yields the same set of complete interleavings.

**Lemma 5.3.** \( s \Downarrow E \text{ iff } E \text{ is of the form } \text{concat} \left( \text{map events } E' \right) \text{ such that } s \Downarrow \Downarrow E' \).

**Corollary 5.1.** Two programs that are delay bisimilar with explicit divergence have the same set of legal executions.

Since the JMM formalisation adds thread start and finish events to the single-threaded semantics (layer 4 in Figure 4.10) by a semantics

---

\[47\text{Remember that } t \vdash (x, h) \tau \text{ expresses that } t \text{ with local state } x \text{ can reduce with a thread action } la \text{ which is not contradictory in itself. This is where the technical assumption Ex-final-inv from Figure 5.10 becomes relevant for preservation. It excludes the case in which some } x_1 \text{ satisfies final}_1 \text{, but final}_2 \text{ is unsatisfiable. Hence, } \_ \vdash \_ \approx \_ \text{ allows Join BTAs in the underlying execution, but } \_ \vdash \_ \_ \text{ does not, which compromises preservation. Note that the other assumptions do not exclude this case. Only preservation of final states involves final, but if } x_1 \text{ is unreachable, } \_ \vdash \_ \_ \approx \_ \text{ need not relate } x_1 \text{ to any } x_2 \text{, i.e., preservation of final states would be trivially satisfied.} \]
Chapter 5. Compiler

transformer, I lift τ-moves and delay bisimulations in the same way. For the former, sf.τ-move adapts τ-move to the changes in the thread-local states, but adds no additional τ-moves (assumption τ-ta in Figure 5.9). In particular, the reductions for start and final events are observable, because (i) other threads can observe through Join BTAs that the thread has terminated, and (ii) the interleaving framework does not allow events for τ-moves.

\[ \tau\text{-move} (x, h) \tau\text{a} (x', h') \]

\[ sf.\tau\text{-move} ((\text{Running}, x), h) (\text{pair-Pre-Start ta}) ((\text{Running}, x'), h') \]

The lifted bisimulation \( t \vdash _{-} \approx_{sf} _{-} \) is defined as follows:

\[ t \vdash ((s_1, x_1), h_1) \approx_{sf} ((s_2, x_2), h_2) \leftrightarrow \]

\[ s_1 = s_2 \land t \vdash (x_1, h_1) \approx (x_2, h_2) \land \]

\[ (s_2 = \text{Finished} \rightarrow \text{final}_1 x_1 \land \text{final}_2 x_2) \]

Similarly, \( w \approx_{sf} \) transforms \( w \) to multithreaded bisimulations just like \( t \vdash _{-} \approx_{sf} _{-} \) and \( w \approx_{sf} \).

**Lemma 5.4.**

\[ m\text{-dbisim-div final}_1 r_1 \text{ acq-events } \tau\text{-move}_1 \]

\[ final_2 r_2 \text{ acq-events } \tau\text{-move}_2 \left( _{-} \vdash _{-} \approx _{-} \right) \approx \]

\[ m\text{-dbisim-div sf.final}_1 sf.r_1 sf.\text{acq-events } \tau\text{-move}_1 \]

\[ sf.final_2 sf.r_2 sf.\text{acq-events } \tau\text{-move}_2 \left( _{-} \vdash _{-} \approx_{sf} _{-} \right) \approx_{sf} \]

### 5.2 Explicit call stacks for source code

The verification of both compiler stages requires an explicit notion of call stacks in the states themselves. The small-step semantics for source code, however, dynamically inlines method calls (RCALL). Hence, I first
define an alternative state representation and small-step semantics \( J_0 \) with explicit call stacks for the source code language (§5.2.1). Note that this does not affect the program declarations, i.e., the input language itself. Second, after having defined the observable transitions, I prove that the semantics with call stacks is delay bisimilar to the semantics without call stacks (§5.2.2).

### 5.2.1 State and semantics

On the call-stack level, local stores are irrelevant for the semantics; any free variable can be bound by an additional block. The rule for method calls \( \text{Rcall} \) and the start state \( J\text{-start} \) already do so for the \( \text{this} \) pointer and the parameters. Hence, the thread-local state in \( J_0 \) consists only of a (non-empty) list of expressions, one for each method on the call stack. To encode non-emptiness in the HOL type, I model the local state as a pair of the head and tail of the call stack.

There are now three levels for the semantics. The expression level deals with the execution of expressions, i.e. method bodies. The call-stack level lifts the semantics for expressions to call stacks and handles calls and returns. This is also the semantics for a single thread. The interleaving semantics lifts this to concurrent programs as before.

To separate method calls and returns from the rest of the semantics, I introduce two auxiliary functions. First, the partial function \( \text{call } e \) returns \([(a, M, vs)]\) whenever \( e \) is about to call the method \( M \) on address \( a \) with parameter values \( vs \), and it is None in all other cases. I say that \( e \) pauses at the call \((a, M, vs)\) iff \( \text{call } e = [(a, M, vs)] \), and \( e \) pauses at a call iff \( \text{call } e \neq \text{None} \). Figure 5.11 shows the definition for \( \text{call} \) where \( \text{is-addr } e \) predicates that \( e \) is of the form \( \text{addr } a \) and \( \text{is-Vals } es \) denotes that \( es \) consists only of values, i.e., \( es = \text{map } \text{Val } vs \) for some \( vs \). Note that \( \text{call} \) traverses the abstract syntax tree in exactly the same way as the small-step semantics \( \_ \_ + \langle \_ \_ \rangle \rightarrow \langle \_ \_ \rangle \) does. When it finds a call of the form \( \text{addr } a.M(\text{map } \text{Val } vs) \), it returns \([(a, M, vs)]\); otherwise, it descends into the subexpression that the small-step semantics evaluates next if there is any, else it returns None.

The second function \( \text{inline } e_0 \ e \) follows the same recursion pattern (Figure 5.12). If \( e \) pauses at a call, \( \text{inline } e_0 \ e \) replaces this call with \( e_0 \). Otherwise, \( \text{inline } e_0 \ e \) leaves \( e \) unchanged. Hence, \( \text{inline} \) mimicks the small-step semantics’ dynamic inlining of method calls.
call (new C) = None

\( call (new T[e]) = \text{call } e \) 

\( call (e \text{ instanceof } T) = \text{call } e \) 

\( call (\text{Cast } T e) = \text{call } e \) 

\( call (\text{Val } v) = \text{None} \) 

\( call (e \bullet \text{bop} \bullet e_2) = (\text{if is-Val } e_1 \text{ then call } e_2 \text{ else call } e_1) \) 

\( call (\text{Var } V) = \text{None} \) 

\( call (V := e) = \text{call } e \) 

\( call (e_1[e_2]) = (\text{if is-Val } e_1 \text{ then call } e_2 \text{ else call } e_1) \) 

\( call (\text{Cast } T e_1[e_2] := e_3) = \) 

\( (\text{if is-Val } e_1 \text{ then (if is-Val } e_2 \text{ then call } e_3 \text{ else call } e_2) \text{ else call } e_1) \) 

\( call (e.\text{length}) = \text{call } e \) 

\( call (e.F[D]) = \text{call } e \) 

\( call (e_1.F[D] := e_2) = (\text{if is-Val } e_1 \text{ then call } e_2 \text{ else call } e_1) \) 

\( call (e.M(es)) = \) 

\( (\text{if is-Val } e \text{ then } \text{if is-Val } e_1 \text{ and is-addr } e \) 

\( \text{then } [(ua.e = \text{addr } a, M, vs. es = \text{map } \text{Val } vs)] \text{ else calls } es) \) 

\( \text{else call } e) \) 

\( call \{V: T = vo; e\} = \text{call } e \) 

\( call (e_1; e_2) = \text{call } e_1 \) 

\( call (\text{if } (e) e_1 \text{ else } e_2) = \text{call } e \) 

\( call (\text{while } (e_1) e_2) = \text{None} \) 

\( call (\text{throw } e) = \text{call } e \) 

\( call (\text{try } e_1 \text{ catch}(C V) e_2) = \text{call } e_1 \) 

\( call (\text{sync } (e_1) e_2) = \text{call } e_1 \) 

\( call (\text{insync } (a) e) = \text{call } e \) 

\( \text{calls } [] = \text{None} \) 

\( \text{calls } (e \cdot es) = (\text{if is-Val } e \text{ then call } e \text{ else calls } es) \) 

\( \text{no-call } P h e = \) 

\( (\forall a M \text{ vs. call } e = [a,M,vs]) \rightarrow \) 

\( (\exists hT Ts T_r D. \text{ typeof-addr } h \text{ a = } [hT] \land \) 

\( P \vdash \text{class-of'}hT \text{ sees } M:Ts \rightarrow T_r = \text{Native in } D)) \)

**Figure 5.11:** The partial function \textit{call} computes which method \( e \) is about to call, if any; \textit{no-call} \( P h e \) predicates that \( e \) is not about to call a non-native method.
5.2. Explicit call stacks for source code

inline \( e_0 \) (new C) = new C
inline \( e_0 \) (new T[\( e \)]) = new T[inline \( e_0 \) \( e \)]
inline \( e_0 \) (\( e \) instanceof T) = (inline \( e_0 \) \( e \)) instanceof T
inline \( e_0 \) (Cast T \( e \)) = Cast t (inline \( e_0 \) \( e \))
inline \( e_0 \) (Val \( v \)) = Val \( v \)
inline \( e_0 \) (\( e_1 \) «bop» \( e_2 \)) =
  (if is-Val \( e_1 \) then \( e_1 \) «bop» (inline \( e_0 \) \( e_2 \)) else (inline \( e_0 \) \( e_1 \) «bop» \( e_2 \))
inline \( e_0 \) (Var \( V \)) = Var \( V \)
inline \( e_0 \) (\( V := e \)) = \( V := \) inline \( e_0 \) \( e \)
inline \( e_0 \) (\( e_1[\{e_2\}] \)) =
  (if is-Val \( e_1 \) then \( e_1[\{\text{inline } e_0 \} e_2] \) else (inline \( e_0 \) \( e_1 \)[\( e_2 \])
inline \( e_0 \) (\( e_1[\{e_2\}] := e_3 \)) =
  (if is-Val \( e_1 \) then
    (if is-Val \( e_2 \) then \( e_1[\{e_2\}] := \) inline \( e_0 \) \( e_3 \) else \( e_1[\{\text{inline } e_0 \} e_2] := e_3 \))
  else (inline \( e_0 \) \( e_1 \)[\( e_2 \)]) := e_3)
inline \( e_0 \) (\( e.F[D] \)) = (inline \( e_0 \) \( e \)).\( F[D] \)
inline \( e_0 \) (\( e_1.F[D] := e_2 \)) =
  (if is-Val \( e_1 \) then \( e_1.F[D] := \) inline \( e_0 \) \( e_2 \) else (inline \( e_0 \) \( e_1 \)).\( F[D] := e_2 \))
inline \( e_0 \) (\( e.M(\{e\}) \)) =
  (if is-Val \( e \) then
    (if is-\( \{\text{Vals } e \} \land \text{is-addr } e \) then \( e_0 \) else \( e.M(\{\text{inline } e_0 \} \{\text{es} \}))
    else (\text{inline } e_0 \) \( e \).\( M(\{\text{es} \))
inline \( e_0 \) (\( V : T = v_0 ; e \)) = \{\( V : T = v_0 ; \) inline \( e_0 \) \( e \)
inline \( e_0 \) (\( e_1 ; : e_2 \)) = (\text{inline } e_0 \) \( e_1 \); \( ; e_2 \)
inline \( e_0 \) (\( (e) e_1 \) else \( e_2 \)) = if (\text{inline } e_0 \) \( e \) \( e_1 \) else \( e_2 \)
inline \( e_0 \) (\( \text{while } (e_1) e_2 \)) = while (\( e_1 \) \( e_2 \)
inline \( e_0 \) (\( \text{throw } e \)) = throw (\text{inline } e_0 \) \( e \)
inline \( e_0 \) (\( \text{try } e_1 \) catch(C V) \( e_2 \)) = try \text{inline } e_0 \) \( e_1 \) catch(C V) \( e_2 \)
inline \( e_0 \) (\( \text{sync } (e_1) e_2 \)) = sync (\text{inline } e_0 \) \( e_1 \) \( e_2 \)
inline \( e_0 \) (\( \text{insync } (a) e \)) = insync (\( a \) \( \text{inline } e_0 \) \( e \)
\text{inlines } e_0 [\[] = []
\text{inlines } e_0 (e \cdot \text{es}) =
  (if is-Val \( e \) then \( \text{inline } e_0 \) \( e \) \cdot \text{es} \) else \( e \cdot (\text{inlines } e_0 \) \( \text{es} \))

**Figure 5.12:** The function \( \text{inline } e_0 \) \( e \) replaces the call that \( e \) is about to execute with \( e_0 \), if any, and leaves \( e \) unchanged otherwise.
Chapter 5. Compiler

\[
P, t \vdash \langle e, (h, \text{empty}) \rangle \rightarrow_{ta} \langle e', (h', xs') \rangle \quad \text{no-call } P \ h \ e
\]

\[
P, t \vdash \langle e, h \rangle \rightarrow_{ta} e \langle e', h' \rangle
\]

\[
P, t \vdash \langle (e, es), h \rangle \rightarrow_{ta} ^{t} \langle (e', es), h' \rangle
\]

\[
call \ e = [(a, M, vs)] \quad \text{typeof-addr } h \ a = [hT]
\]

\[
\begin{align*}
P \vdash \text{class-of} & \ hT \ \text{sees } M:Ts \rightarrow Tr = [(pns, body)] \ \text{in } D \\
& \quad [vs] = |pns| \quad |Ts| = |pns|
\end{align*}
\]

\[
P, t \vdash \langle (e, es), h \rangle \rightarrow_{ta} e \langle (blocks (this\cdot pns) (Class D\cdot Ts) (Addr a\cdot vs) body, e\cdot es), h \rangle
\]

\[
\begin{align*}
\text{final } e
\quad & \quad P, t \vdash \langle (e, e' \cdot es), h \rangle \rightarrow_{ta} e \langle (inline e \ e', es), h \rangle
\end{align*}
\]

\textbf{Figure 5.13:} Single-threaded source code semantics with explicit call stacks

The expression level semantics \(\text{red}_0\) (notation \(P, t \vdash \langle e, h \rangle \rightarrow_{ta} ^{e} \langle e', h' \rangle\)) is the same as in \(J\) except for calling, see \(\text{RED}_0\) in Figure 5.13.\(^{48}\) To avoid redundancies, I do not define a new small-step semantics, but use the predicate \(\text{no-call } P \ h \ e\) to filter out all reductions due to \(\text{RCALL}\) from \(P, t \vdash \langle e, (h, xs) \rangle \rightarrow_{ta} \langle e', (h', xs') \rangle\). As Figure 5.11 shows, \(\text{no-call } P \ h \ e\) holds iff whenever \(e\) pauses at a call, then the called method must be native. Note that \(\text{red}_0\) discards the new local store \(xs'\) after the reduction – well-formedness ensures that \(xs'\) is always \text{empty} (Corollary 5.2).

Figure 5.13 also shows the small-step semantics \(\text{J}_0\)-\(\text{red}\) for the call-stack level (notation \(P, t \vdash \langle (e, es), h \rangle \rightarrow_{ta} ^{t} \langle (e', es'), h' \rangle\)). It consists of all reductions of the expression level semantics \(\text{red}_0\) for the top of the call stack (\(\text{R}_0\text{RED}\)). Additionally, it reintroduces the reductions for method calls that \(\text{red}_0\) has filtered out (\(\text{R}_0\text{CALL}\)). Rather than dynamically inlining

\(^{48}\)Technically, I must also adjust the thread-local state in \text{Spawn} actions in \(ta\). Formally, 
\[
d_1 \vdash \langle \ldots \rangle \rightarrow \langle \ldots \rangle
\]
is parametrised over \text{native-TA2J} \ P. The definition of \(\text{red}_0\) passes \text{native-TA2J} \ P instead of \text{native-TA2J} \ P, where \text{native-TA2J} \ P converts all local states \((C, M, a)\) of \text{Spawn} BTAs to

\[
\text{let } (D, _\ldots, m) = \text{method } P \ C \ M; \ [(\ldots \text{body})] = m \ \text{in } ((\text{this}: \ Class \ C = [Addr \ a]; body), [])
\]

In the remainder of this chapter, I omit this technical detail.
the method body, $R_0$ push the called method’s body on top of the call stack and leaves the caller’s expression unchanged. When a method returns, i.e., its expression is final, $R_0$ replaces the call in the caller’s expression with the return value or exception using inline. This assumes that every expression on the stack except the top one pauses at the method invocation. $R_0$ has no counterpart in $_\_\_ \vdash \langle _\_\_ \rangle \rightarrow \langle _\_\_ \rangle$, because dynamic inlining turns returns into no-ops.

A state $(e, es)$ in $J_0$ is final, written $final_0 (e, es)$, iff $final e$ and $es = \[]$. The start state $J_0$-start $P C M$ has one thread $start-tID$ with local state

$$(blocks (this \cdot pns) (Class D \cdot Ts) (Null \cdot vs) body, [])$$

where $(D, Ts, _\_ [(pns, body)]) = method P C M$.

Analogous to $J$, the multithreaded semantics $J_0$.redT for $J_0$ is the interleaving semantics for the parameter instantiations $final_0$ and $J_0$-red.

I consider the following operations as observable moves: memory allocation, calls to native methods other than hashcode and currentThread, access and assignment to fields and array cells, reading array lengths, and synchronisation. In particular, since thread spawns, joining, interruption and the wait-notify mechanism are implemented as native methods, all of them are observable. Conversely, all control flow constructs, including exception throwing and handling, and local variable manipulation are only relevant to the thread that executes them, so these generate only $\tau$-moves.

For simplicity of the formalisation, I define observability in terms of the state being reduced rather than the reduction itself. Hence, either all reductions of a thread in one thread-local state and heap are observable or none of them. As a consequence, the set of observable reductions is larger than necessary. For example, the array cell access $addr a[Val (Intg i)]$ returns either the $i$-th array cell’s content or fails with an ArrayIndexOutOfBoundsException exception. Thus, $J$ and $J_0$ also treat throwing the ArrayIndexOutOfBoundsException exception as an observable move. Fortunately, a larger set of observable moves only strengthens the correctness result.

Formally, I define a predicate $\tau$-move :: $'m prog \Rightarrow 'heap \Rightarrow 'addr expr \Rightarrow bool$ that identifies states in which $\tau$-moves originate. Its definition can be found in Appendix B.6.6. Then, $J$-$\tau$-move and $J_0$-$\tau$-move determine the $\tau$-moves for $J$ and $J_0$, respectively, where

$$J$-$\tau$-move $P ((e, xs), h) ta _\_ \leftrightarrow \tau$-move $P h e \land ta = ()$$

$$J_0$-$\tau$-move $P ((e_0, es_0), h_0) ta _\_ \leftrightarrow (\tau$-move $P h_0 e_0 \lor final e_0) \land ta = ()$$

231
**Lemma 5.5.** $J$ and $J_0$ satisfy the assumptions of locale $\tau$-multithreaded.

$\tau$-multithreaded $J$-final $(J$-red $P)$ acq-events $J$-$\tau$-move

$\tau$-multithreaded final$_0$ $(J_0$-red $P)$ acq-events $J_0$-$\tau$-move

### 5.2.2 Semantic equivalence

Now, I show that $J_0$ is equivalent to $J$ in the sense that a program is delay bisimilar to itself under the two semantics. Thus, I can verify the compiler against $J_0$ instead of $J$.

A variable $V$ is free in the expression $e$ (written $V \in \text{fv} \ e$) iff $e$ contains a subexpression $\text{Var} \ V$ that is not contained in a local-variable or catch block that declares $V$. The expression $e$ is closed iff it contains no free variables, i.e., $\text{fv} \ e = \emptyset$. A call stack of expressions $(e_0, es_0)$ is well-formed (notation $\text{wf}_0 (e_0, es_0)$) iff $e_0$ is closed, and all expressions in $es_0$ are closed and pause at a call. Formally:

$$
\text{wf}_0 (e_0, es_0) \iff \text{fv} \ e_0 = \emptyset \land (\forall e \in \text{set} \ es_0. \text{fv} \ e = \emptyset \land \text{call} \ e \neq \text{None})
$$

Closedness rules out references to global variables.\(^{49}\) Hence, it is irrelevant that $J_0$ executes method bodies in a local store that contains only $\text{this}$ and the parameters $(R_{0\text{call}})$ while inlining method calls with blocks for $\text{this}$ and the parameters in $J$ adds them to the local store of the caller $(R_{\text{call}})$.

Let $\text{collapse } (e_0, es_0)$ abbreviate the expression to which inlining collapses the call stack, i.e., $\text{collapse } (e_0, es_0) = \text{foldl inline} e_0 es_0$. Then, the delay bisimulation relation $\approx_0$ for single threads relates the $J$ state $((e, xs), h)$ with $((e_0, es_0), h_0)$ iff $xs$ is empty, the heaps are the same, $(e_0, es_0)$ is well-formed, and $(e_0, es_0)$ collapses to $e$. Formally:

$$
\text{wf}_0 (e_0, es_0) \quad \text{-collapse } (e_0, es_0) (\emptyset, h) \approx_0 ((e_0, es_0), h)
$$

\(^{49}\)Already in Jinja, closedness was crucial for the small-step semantics and its equivalence to the big-step semantics [83, §2.3.2, §2.4.1, §2.5]. Klein and Nipkow write: “we can only get away with this simple rule for method calls [for the small-step semantics] because there are no global variables in Java. Otherwise one could unfold a method body that refers to some global variable into a context that declares a local variable of the same name, which would essentially amount to dynamic variable binding.” [83, §2.3.2].
Since both J.redT and J₀.redT are instances of the interleaving semantics, I lift \( \approx \) to multithreaded states as described in §5.1.3. Let \( \approx_{0} \) and \( \sim \) denote the corresponding instance of \( \approx \) and \( \sim \) where \( (e, xs) \approx (e_0, es_0) \) iff \( \sim \) final \( e_0 \).

**Theorem 5.5.** If \( \text{wf-J-prog} \ P \), then \( \langle \approx_{0}, \sim \rangle \) is a delay bisimulation with explicit divergence for J-red \( P \) t and J₀-red \( P \) t.

The proof requires a number of lemmata first.

**Lemma 5.6.** If \( \text{wf-J-prog} \ P \) and \( P, t \vdash \langle e, (h, xs) \rangle \rightarrow_{ta} \langle e', (h', xs') \rangle \), then \( \text{fv} e' \subseteq \text{fv} e \) and \( \text{dom} xs' \subseteq \text{dom} xs \cup \text{fv} e \).

**Proof.** By induction on \( P, t \vdash \langle e, (h, xs) \rangle \rightarrow_{ta} \langle e', (h', xs') \rangle \). Method call (R_CALL) is the interesting case. Since the call sees the method, say \( P \vdash C \) sees \( M:Ts \rightarrow T_r \) in \( D \), and \( P \) is well-formed, \( P, E \vdash \text{body} :: T \) for \( E = [\text{this} \mapsto \text{Class C}, \text{pns} [\rightarrow] Ts] \) and some \( T \) (see Figure 2.10). Induction on \( P, E \vdash \text{body} :: T \) yields \( \text{fv body} \subseteq \text{dom} E \). Hence, blocks \( (\text{this-pns}) \) (Class \( D \cdot Ts \)) (Addr \( a \cdot vs \)) body is closed. \( \square \)

**Corollary 5.2.** If \( \text{wf-J-prog} \ P \) and \( P, t \vdash \langle e, (h, \text{empty}) \rangle \rightarrow_{ta} \langle e', (h', xs') \rangle \) and \( e \) is closed, then \( xs' = \text{empty} \).

**Lemma 5.7.** If \( \text{wf-J-prog} \ P \), then \( \text{J₀-red} \ P \) t preserves well-formedness.

**Proof.** Let \( P, t \vdash \langle (e_0, es_0), h \rangle \rightarrow_{ta} \langle (e_0', es_0'), h' \rangle \) and \( \text{wf}_0 \ (e_0, es_0) \). I show \( \text{wf}_0 (e_0', es_0') \) by case analysis of the reduction.

For R₀RED, Lemma 5.6 applies. For R₀CALL, the same argument as for method calls in Lemma 5.6 yields closedness of the new call frame. For R₀RET, \( \text{fv} (\text{inline} e_0 e) \subseteq \text{fv} e_0 \cup \text{fv} e \) (provable by induction on \( e \)) yields \( \text{fv} (\text{inline} e_0 e) = \emptyset \). \( \square \)

A call stack \( (e_0, es_0) \) is normalised iff \( \sim \) final \( e_0 \) or \( es_0 = [] \), i.e., rule R₀RET is not applicable. The next lemma shows that \( J_0 \) can silently normalise any call stack; it is provable by induction on \( es_0 \).

**Lemma 5.8.** For every call stack \( (e_0, es_0) \), there is a normalised call stack \( (e'_0, es'_0) \) such that \( \text{collapse} (e_0, es_0) = \text{collapse} (e'_0, es'_0) \) and \( (e_0, es_0) \) silently reduces to \( (e'_0, es'_0) \).
A normalised call stack \((e_0, es_0)\) simulates reductions of the collapsed call stack \(\text{collapse}\ (e_0, es_0)\) directly, i.e., without any additional \(\tau\)-moves.

**Lemma 5.9.** Let \((e_0, es_0)\) be well-formed and normalised.

(i) If \(P, t \vdash \langle e_0, (h, \text{empty})\rangle \rightarrow \langle e'_0, (h', \text{empty})\rangle\), then 
\(P, t \vdash \langle \text{collapse} \ (e_0, es_0), (h, \text{empty})\rangle \rightarrow \langle \text{collapse} \ (e'_0, es_0), (h', \text{empty})\rangle\)

(ii) If \(P, t \vdash \langle \text{collapse} \ (e_0, es_0), (h, \text{empty})\rangle \rightarrow \langle e', (h', \text{empty})\rangle\), then 
\(e'\) is of the form \(\text{collapse} \ (e'_0, es_0)\) and \(P, t \vdash \langle e_0, (h, \text{empty})\rangle \rightarrow \langle e'_0, (h', \text{empty})\rangle\).

**Proof.** Note that inlining the top of the call stack preserves well-formedness. Without loss of generality, \(-\text{final} \ e_0 -\) otherwise, \(es_0 = []\) by normalisation and the lemma holds trivially. Then, each direction follows by induction on \(es_0\) from the following generalised one-step versions for arbitrary \(e\) with call \(e \neq \text{None}\):

(i) If \(P, t \vdash \langle e_0, (h, \text{empty})\rangle \rightarrow \langle e'_0, (h', \text{empty})\rangle\), then 
\(P, t \vdash \langle \text{inline} \ e_0 \ e, (h, xs)\rangle \rightarrow \langle \text{inline} \ e'_0 \ e, (h', xs)\rangle\).

(ii) If \(P, t \vdash \langle \text{inline} \ e_0 \ e, (h, xs)\rangle \rightarrow \langle e', (h', xs')\rangle\), then \(xs = xs'\) and 
\(e'\) is of the form \(e' = \text{inline} \ e'_0 \ e\) and \(P, t \vdash \langle e_0, (h, \text{empty})\rangle \rightarrow \langle e'_0, (h', \text{empty})\rangle\).

Both of them are proved by induction on \(e\). The only interesting case is when \(e\) is the call that \(\text{inline}\) replaces with \(e_0\). Then, the local store in \(e_0\)'s reduction must change from \(xs\) to \(\text{empty}\) or vice versa. This follows from the next two easy lemmata (provable by induction) that Jinja uses to prove the big-step and small-step semantics equivalent [83, §2.5]:

(i) If \(P, t \vdash \langle e, (h, xs)\rangle \rightarrow \langle e', (h', xs')\rangle\), then 
\(P, t \vdash \langle e, (h, xs_0 ++ xs)\rangle \rightarrow \langle e', (h', xs_0 ++ xs')\rangle\).

(ii) If \(P, t \vdash \langle e, (h, xs)\rangle \rightarrow \langle e', (h', xs')\rangle\) and \(fv \ e \subseteq W\), then 
\(P, t \vdash \langle e, (h, xs \upharpoonright W)\rangle \rightarrow \langle e', (h', xs' \upharpoonright W)\rangle\),

where \(f \upharpoonright A\) restricts the map \(f\) to \(A\), i.e., \(\lambda x. \text{if } x \in A \text{ then } f \ x \text{ else None}\), and \(f ++ g\) abbreviates \(\lambda x. \text{case } g \ x \text{ of None } \Rightarrow f \ x | [y] \Rightarrow [y]\). □

The next lemma shows that when \(e\) pauses at a call, \(J\)'s next reduction is to replace the call by the method body.
Lemma 5.10. Let \( \text{call } e = [(a, M, vs)] \) and \( \text{typeof-addr } h \ a = [hT] \) and \( P \vdash \text{class-of'} hT \) sees \( M: Ts \to Tr = [(pns, body)] \) in \( D \) and \( \text{blks} = \text{blocks (this.pns) (Class } D \cdot Ts \) (Addr } a \cdot vs \) body.

(i) If \( P, t \vdash \langle e, (h, xs) \rangle \rightarrow \text{ta} \rightarrow \langle e', (h', xs') \rangle \), then \( e' = \text{inline } \text{blks } e \).

(ii) If \( |vs| = |pns| \) and \( |Ts| = |pns| \), then \( P, t \vdash \langle e, (h, xs) \rangle \rightarrow \text{ta} \rightarrow \langle \text{inline } \text{blks } e, (h', xs') \rangle \).

Proof of Theorem 5.5. By Lemma 5.2, it suffices to find two measures \( \prec_1 \) and \( \prec_2 \) for which \( (\approx_0, \sim_0) \) is a well-founded delay bisimulation. Choose \( \_ \prec_1 \_ \) and \( ((e_0, es_0), h_0) \prec_2 ((e'_0, es'_0), h'_0) \) iff \( |es_0| < |es'_0| \), i.e., only returns \( R_0 \text{ret} \) from method calls (when the call stack shrinks) need not have a counterpart in \( J \).

For the simulation diagrams from Figure 5.6, I distinguish three cases:

1. Calls of non-native methods. For \( J_0 \) simulating \( J \), it normalises the call stack first (Lemma 5.8). Then, Lemma 5.9 shows that the top call frame of the normalised call stack can reduce using the call, and Lemma 5.10 decomposes the resulting expression as necessary for the simulation with \( R_0 \text{call} \). For the other direction, Lemma 5.10 shows that the expression in the top call frame could also inline the call and so can the collapsed call stack by Lemma 5.9.

2. Returns from a method call (\( R_0 \text{ret} \)) are a no-op in \( J \), but the call stack length decreases.

3. Otherwise, it is an expression-level reduction, for which \( J \) and \( J_0 \) use the same semantics; Lemma 5.9 shows that collapsing the call stack does not change the semantics. As with calls of non-native methods, \( J_0 \) first normalises the call stack.

In all cases, well-formedness of the new call stack holds by preservation (Lemma 5.7). □

Lemma 5.11. \( (\approx_0, \sim_0) \) preserves final states.

Theorem 5.6. \( (\approx_m, \sim_0) \) is a delay bisimulation with explicit divergence for \( J.\text{redT } P \) and \( J_0.\text{redT } P \) that preserves final states.

Proof. By Theorem 5.3, it suffices to discharge the assumptions of locale \( m\text{-dbisim-div} \). Theorem 5.5 and Lemma 5.11 discharge the inherited
locales. Since \( t \approx 0 \) does not depend on the heap, heap-change-preserve holds trivially. For \( w \approx I \), case analysis and induction show that whenever \( J_0\text{-}red \) generates a Suspend BTA, the top call frame is not final. Since \( w \approx \) guarantees that the \( J_0 \) call stack is normalised, simulation-\( w \approx 1 \) follows easily, because normalised call stacks simulate \( J \)'s reductions without any additional \( \tau \)-moves. Similarly, simulation-\( w \approx 2 \) holds because \( J \) simulates observable moves of \( J_0 \) without \( \tau \)-moves. Finally, Ex-final-inv holds trivially. \( \square \)

Finally, I show that the bisimulation relation \( m \approx 0 \) contains the well-formed start state.

Lemma 5.12. If \( \text{wf-J-prog} P \) and \( \text{wf-start} P C M \) vs, then \( J\text{-}start P C M \) vs \( m \approx 0 J_0\text{-}start P C M \) vs.

5.3 Register allocation

The first stage of the compiler replaces variable names in expressions by indices into an array of registers. In this section, I present the intermediate language \( J_1 \) with syntax, well-formedness, and semantics (§5.3.1), the first compilation stage (§5.3.2), and the proof of its correctness (§5.3.3 and §5.3.4).

5.3.1 Intermediate language \( J_1 \)

Syntax

The intermediate language \( J_1 \) retains the expressions from source code, but stores local variable values in an array of registers – like bytecode does. Hence, local variables in \( J_1 \) are no longer identified by their name, but by an index in the array. \( J_1 \) extends Jinja’s intermediate language [83, §5.1] analogous to what \( J \) does in source code, of which I have already described the details in §2.3 and §3.2.2.

To avoid duplication, JinjaThreads parametrises the type of expressions \( (a, b, \text{addr}) \text{exp} \) not only over the type of addresses \( \text{addr} \), but also over the variable names \( a \) and an annotation type \( b \) for sync (\( _\_ \)) \_ and insync (\( _\_ \)) \_ blocks. \( \text{addr exp} \) abbreviates \( (\text{vname}, \text{unit}, \text{addr}) \text{exp} \) where \( \text{unit} \) is the HOL type of only one element (\( \_ \)). Expressions in \( J_1 \) are of type
(nat, nat, addr) exp (denoted 'addr expr1), i.e., variable names are natural numbers and synci (__) _ and insynci (__) _ blocks are now annotated with i :: nat. Following the JVMS [103, §7.14], the variable i will be used in bytecode to store the monitor address between the MEnter and MExit instructions that implement the monitor locking and unlocking.

The type of J1 programs 'addr J1-prog is 'addr expr1 prog. Note that methods no longer declare parameter names because they have been replaced by numbers.

Well-formedness

J1 requires a very specific layout of the registers, which compP1 ensures. Register 0 holds the this pointer, the parameters occupy registers 1 to n where n is the number of parameters. Then, the local variables follow according to the nesting depth: If a block {i: T = vo; e} is nested in k local variable or catch blocks or bodies of synci (__) _, then i = 1 + n + k; and similarly for try e catch(C i) e' and synci (e) e'. For example,

\[
\text{try e1 catch(C 3) (sync4 ([4: T1 = vo1; e2]) [5: T2 = vo2; e3])}
\]

is fine for a method with two parameters, but {3: T1 = vo1; {5: T2 = vo2; e}}, {4: T = vo; e}, and {3: T1 = vo1; {2: T2 = vo2; e}} are not. Klein and Nipkow call this layout “an inverse de Bruijn numbering scheme” [83, §5.1.1]. The predicate Be n in Figure 5.14 enforces it where n denotes the starting number for the outermost blocks.

The typing rules for J1 are almost identical to J. In P, E ⊢_1 e :: T, the program P has type 'addr J1-prog and the environment E for local variables now is a list of types where the i-th element corresponds to variable i. The rule WT1SYNC for synci (__) _ blocks demonstrates all relevant changes:

\[
\begin{align*}
\text{WT1SYNC:} & \quad \frac{P, E \vdash_1 e_1 :: T_1 \quad T_1 \neq NT \quad P, E @ [\text{Class Object}] \vdash_1 e_2 :: T}{P, E \vdash_1 \text{sync}_j (e_1) e_2 :: T} \\
\end{align*}
\]

P, E ⊢_1 _ :: _ implicitly relies on the numbering scheme with |E| as start index, as WT1SYNC ignores the annotation variable j. Instead, it extends the environment for the monitor variable with the type Class Object.

---

50Technically, they are annotated in J, too, namely with (). Hence, sync (e) e' actually abbreviates sync1 (e) e' and similarly for insync (a) e.
Since the compiler introduces these monitor variables, no expression should access them. The predicate $S e$ ensures this by checking that $i \not\in \text{fv } e'$ for all subexpressions of $e$ of the form $\text{sync}_i (\_ ) e'$ or $\text{insync}_i (a) e'$.\textsuperscript{51}

The well-formedness conditions specific to $J_1$ are similar to $J$'s (Figure 2.10). The constraints on parameter names have been dropped and the numbering scheme and no access to monitor variables are required. \{ ..n \} denotes the set of natural numbers from 0 to $n$ inclusive.

\textsuperscript{51}In the definition of $fv$, I pretended that it is defined on $\text{'addr expr}$, but in reality it is defined on ($a', b', \text{'addr}$) expr. Hence, it works on $\text{'addr expr}_1$, too. The same applies to final, $D$, inline, and ok-$I$. 

---

<table>
<thead>
<tr>
<th>$B$ (new C) $i$</th>
<th>$\iff$ True</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B$ (new T[e]) $i$</td>
<td>$\iff B e i$</td>
</tr>
<tr>
<td>$B$ (e instanceof T) $i$</td>
<td>$\iff B e i$</td>
</tr>
<tr>
<td>$B$ (Cast T e) $i$</td>
<td>$\iff B e i$</td>
</tr>
<tr>
<td>$B$ (Val v) $i$</td>
<td>$\iff$ True</td>
</tr>
<tr>
<td>$B$ (new T[e]) $i$</td>
<td>$\iff B e_i$</td>
</tr>
<tr>
<td>$B$ (e1 «bop» e2) $i$</td>
<td>$\iff B e_1 i \land B e_2 i$</td>
</tr>
<tr>
<td>$B$ (Var j) $i$</td>
<td>$\iff$ True</td>
</tr>
<tr>
<td>$B$ (V B e) $i$</td>
<td>$\iff B e_i$</td>
</tr>
<tr>
<td>$B$ (e[ ][e]) $i$</td>
<td>$\iff B e_1 i \land B e_2 i$</td>
</tr>
<tr>
<td>$B$ (e1[e2] := e3) $i$</td>
<td>$\iff B e_1 i \land B e_2 i \land B e_3 i$</td>
</tr>
<tr>
<td>$B$ (e.length) $i$</td>
<td>$\iff B e_i$</td>
</tr>
<tr>
<td>$B$ (e.F[D]) $i$</td>
<td>$\iff B e_i$</td>
</tr>
<tr>
<td>$B$ (e1.F[D] := e2) $i$</td>
<td>$\iff B e_1 i \land B e_2 i$</td>
</tr>
<tr>
<td>$B$ (e.M(es)) $i$</td>
<td>$\iff B e_i \land Bs es i$</td>
</tr>
<tr>
<td>$B$ {j: T = vo; e} $i$</td>
<td>$\iff i = j \land B e (i + 1)$</td>
</tr>
<tr>
<td>$B$ (e1;; e2) $i$</td>
<td>$\iff B e_1 i \land B e_2 i$</td>
</tr>
<tr>
<td>$B$ (if (e) e1 else e2) $i$</td>
<td>$\iff B e_i \land B e_1 i \land B e_2 i$</td>
</tr>
<tr>
<td>$B$ (while (e1) e2) $i$</td>
<td>$\iff B e_1 i \land B e_2 i$</td>
</tr>
<tr>
<td>$B$ (throw e) $i$</td>
<td>$\iff B e_i$</td>
</tr>
<tr>
<td>$B$ (try e1 catch(C j) e2) $i$</td>
<td>$\iff B e_1 i \land i = j \land B e_2 (i + 1)$</td>
</tr>
<tr>
<td>$B$ (sync_j (e1) e2) $i$</td>
<td>$\iff B e_1 i \land i = j \land B e_2 (i + 1)$</td>
</tr>
<tr>
<td>$B$ (insync_j (a) e) $i$</td>
<td>$\iff i = j \land B e (i + 1)$</td>
</tr>
<tr>
<td>$Bs [] i$</td>
<td>$\iff$ True</td>
</tr>
<tr>
<td>$Bs (e \cdot es) i$</td>
<td>$\iff B e_i \land Bs es i$</td>
</tr>
</tbody>
</table>

**Figure 5.14:** Definition of $B
5.3. Register allocation

\[
\text{wf-} J_1 \text{-mdecl} \quad P \quad C \quad (M,T_s,T_r,\text{body}) = \\
(\exists T. \quad P, \text{Class} C \cdot \quad T_s \vdash_1 \text{body} :: T \quad \wedge \quad P \vdash T \leq T_r) \wedge \\
\mathcal{D} \text{body} \setminus \{..|T_s|\} \wedge \mathcal{B} \text{body} \setminus (|T_s| + 1) \wedge \mathcal{S} \text{body}
\]

\[
\text{wf-} J_1 \text{-prog} = \text{wf-prog} \quad \text{wf-} J_1 \text{-mdecl}
\]

Semantics

The state space of \( J_1 \) is already close to bytecode. The thread-local state is a list of call frames each of which consists of an expression and a fixed-size array of registers for the local variables.

On the expression level, the small-step semantics \( \text{red}_1 \) is now of the form \( \text{fail}, P, t \vdash \langle e, (h,xs) \rangle \quad \text{\texttt{ta} \rightarrow}^e_1 \langle e', (h',xs') \rangle \) where \( P :: \text{\texttt{\textsc{addr} \ J_1 \text{-prog}}} \) and \( xs, xs' :: \text{\texttt{addr \ val \ list}} \). The new parameter \( \text{fail} :: \text{bool} \) determines whether unlocking a monitor may fail.

The main difference between \( J_0 \) and \( J_1 \) is that \( J_1 \) handles local variables and synchronisation like bytecode.

In particular, \( \text{sync}_i \ (\text{\texttt{addr} \ a}) \ e \) stores the monitor address \( a \) in the local variable \( i \) upon locking the monitor. Accordingly, when unlocking the monitor, \( \text{insync}_i \ (a) \ e \) ignores \( a \), but retrieves the monitor address from register \( i \). Figure 5.15 shows the rules for synchronisation. Note that they explicitly check for the bounds of the register array. Compared to Figure 3.21, there are two new pairs of rules for unlocking: First, \( R_1 \text{unlckN} \) and \( R_1 \text{unlckXN} \) raise a \texttt{NullPointerException} exception if the register \( xs[i] \) for the monitor stores the \texttt{Null} pointer instead of an address. Second, \( R_1 \text{unlckF} \) and \( R_1 \text{unlckXF} \) allow unlocking to non-deterministically fail with an \texttt{IllegalMonitorState} exception like \texttt{MExit} (see Figure 3.23). The switch \( \text{fail} \) must be set to \texttt{True} to activate the latter.

\[
\text{fail}, P, t \vdash \langle e, (h,xs) \rangle \quad \text{\texttt{ta} \rightarrow}^e_1 \langle e', (h',xs') \rangle \quad \text{differs from} \quad P, t \vdash \langle \_, \_ \rangle \quad \text{\texttt{\texttt{\texttt{-\rightarrow}}}}_1 \quad \langle \_, \_ \rangle \quad \text{in two further respects. First, the small-step semantics treats local variable blocks with initialisation, say} \quad \{i: \quad T = [v]; \quad e\}, \quad \text{like} \quad \{i: \quad T = \texttt{None}; \quad i := \texttt{Val} v; \quad e\} \quad \text{and completely ignores uninitialised blocks. This ensures that} \quad J_1 \quad \text{and bytecode treat local variables identically. Second, there is no rule for calling non-native methods.}
\]

Instead, the semantics \( J_1 \text{-red} \) for single threads (notation \( \text{fail}, P, t \vdash \langle ((e,xs),exs),h \rangle \text{\texttt{\texttt{\texttt{-\rightarrow}}}}_1 \langle ((e',xs'),exs'),h' \rangle \) takes care of method calls and returns similar to \( J_0 \text{-red} \) – see Figure 5.16 for the definition. The conversion function \( \text{native-TA2J}_1 \) for thread actions is defined analogously to \( \text{native-TA2J} \) and \( \text{native-TA2J}_0 \). \( \text{call}_1 \) differs from \( \text{call} \) only for
Chapter 5. Compiler

Figure 5.15: Reduction rules for synchronized blocks in $J_1$
5.3. Register allocation

R1 red:

R1 call:

R1 ret:

f ail, P, t ` he, (h, xs)i −ta→e1 he0 , (h0 , xs0 )i
f ail, P, t ` h((e, xs), exs), hi −native-TA2J1 P ta→t1
h((e0 , xs0 ), exs), h0 i


call1 e = (a, M, vs)
typeof-addr
 h a= bhTc
0
P ` class-of hT sees M:Ts→T = body in D
|vs| = |Ts|
e0 = blocks1 0 (Class D · Ts) body
0
xs = Addr a · vs @ replicate (max-vars body) undefined-Val
f ail, P, t ` h((e, xs), exs), hi −LM→t1 h((e0 , xs0 ), (e, xs) · exs), hi
final e
f ail, P, t ` h((e, xs), (e0 , xs0 ) · exs), hi −LM→t1
h((inline e e0 , xs0 ), exs), hi
Figure 5.16: Reduction rules for call stacks in J1

blocks. Initialised blocks never pause at a call, since the semantics first
“uninitialises” them.
call1 ({i : T = vo; e}) = (if vo = None then call1 e else None)
Note that R1 call initialises the registers of the new call frame just like
exec-instr does in Figure 3.22. The function max-vars computes the
maximum depth of nested local variables including the variables for
sync_ (_) _ blocks. Analogous to blocks, blocks1 n Ts body wraps body
in uninitialised blocks for local variables n to n + |ts| − 1 with types Ts.
Setting f ail to False or True yields two different semantics of J1 , to
which I refer as J#1 and J01 , respectively; J1 refers to both. Similarly,
I sometimes omit the f ail parameter from the semantics and instead
decorate them with 0 or #, e.g, J#1 -red and _, _ ` h_, _i −_→t10 h_, _i.
Like for source code, the multithreaded semantics J#1 .redT and J01 .redT
are the interleaving semantics instantiated with J#1 -red and J01 -red, respectively. A J1 thread is final, written J1 -final ((e, xs), exs), iff final e and exs =
[]. The start state J1 -start P C M vs has one thread start-tID with local state

((blocks1 0 (Class D · Ts) (Null · vs) body,
Null · vs @ replicate (max-vars body)undefined-Val), [])


where (D, Ts, _, body ) = method P C M.
241


\[
\begin{align*}
\text{compE}_1 (\text{Var} \ V) &= \text{Var} (\text{index} \ Vs \ V) \\
\text{compE}_1 \ Vs \ [V: T = vo; e] &= \{ |Vs| : T = vo; \text{compE}_1 (Vs @ [V]) e \} \\
\text{compE}_1 \ Vs (\text{sync} \ (e_1) e_2) &= \\
\quad \text{sync}_{|Vs|} (\text{compE}_1 \ Vs \ e_1) (\text{compE}_1 (Vs @ [\text{fresh-vname} Vs]) e_2) \\
\text{compE}_1 \ Vs (\text{insync} \ (a) e) &= \\
\quad \text{insync}_{|Vs|} (a) (\text{compE}_1 (Vs @ [\text{fresh-vname} Vs]) e)
\end{align*}
\]

Figure 5.17: Register allocation \(\text{compE}_1\) for local variables, blocks, and synchronisation

On the level of single threads, \(J^\#_1\) and \(J'_1\) are not bisimilar, because unlocking a monitor can non-deterministically fail in \(J'_1\), but not in \(J^\#_1\). Below, I will use \(J^\#_1\) for proving the first compiler stage correct and \(J'_1\) for the second. Then, I will show that under suitable conditions, \(J^\#_1\) and \(J'_1\) coincide on the multithreaded level.

To identify \(\tau\)-moves, \(J_1\) defines a predicate \(\tau\)-move\(_1\) similar to \(\tau\)-move. Like for \text{call} vs. \text{call}\(_1\), the only difference is that \text{red}\(_1\) may silently “uninitialise” initialised blocks first.

\[
\tau\text{-move}_1 \ P \ h \ {i : T = vo; e} \iff vo \neq \text{None} \lor \tau\text{-move}_1 \ P \ h \ e \lor \text{final} \ e
\]

\(J_1\)-\(\tau\)-move lifts \(\tau\)-move\(_1\) to call stacks:

\[
J_1\text{-}\tau\text{-move} \ P \ ((e, xs), exs), h \ \text{ta} \_ \iff (\tau\text{-move}_1 \ P \ h \ e \lor \text{final} \ e) \land \text{ta} = (\|)
\]

Lemma 5.13. \(J^\#_1\) and \(J'_1\) satisfy the assumptions of locale \(\tau\text{-multithreaded}\).

\[
\tau\text{-multithreaded} \ J_1\text{-final} (J^\#_1\text{-red} \ P) \ \text{acq-events} \ J_1\text{-}\tau\text{-move}
\]

\[
\tau\text{-multithreaded} \ J_1\text{-final} (J'_1\text{-red} \ P) \ \text{acq-events} \ J_1\text{-}\tau\text{-move}
\]

### 5.3.2 Compilation stage 1

Jinja already contains a compiler \(\text{compE}_1\) from 'addr expr' to 'addr expr\(_1\)', i.e., for method bodies. It assigns registers to variables [83, §5.2] in the following order: first the \(\text{this}\) pointer, then the method parameters, and finally local variables ordered by block nesting level. While traversing the expression, \(\text{compE}_1\) keeps track of the list of variables \(Vs\) declared on
the path from the root of the expression to the current subexpression and replaces variables \( V \) by their index in \( Vs \) (written \( \text{index} Vs V \)), i.e., the position of the last occurrence of \( V \) in \( Vs \). Figure 5.17 shows an excerpt of its definition, the full definition can be found in Appendix B.9.2.

For \( \text{sync}\ (e_1)\ e_2 \) blocks, \( \text{compE}_1 \) reserves the register \( |Vs| \) to hold the monitor address. To shift the registers in \( e_2 \) by 1, it appends a fresh variable name \( \text{fresh-vname} Vs \) to \( Vs \). Freshness (i.e., \( \text{fresh-vname} Vs \notin \text{set}Vs \)) ensures that it does not hide any variables in surrounding blocks.

Jinja defines an operator \( \text{compP} \) to lift compilation at the level of expressions to whole programs [83, §5.4]. I have straightforwardly adapted it to JinjaThreads programs (see Appendix B.9.1). The compiler from source code to intermediate language

\[
\text{compP}_1 = \text{compP} (\lambda C\ M\ Ts\ T\ (\text{pns},\text{body})\ \text{compE}_1 (\text{this} \cdot \text{pns})\ \text{body})
\]

applies \( \text{compE}_1 \) to all method bodies.

For example, consider the following method declaration in Java, whose body is \( ([f], \text{sync}\ (\text{Var} f)\ (\text{Var this.m}())) \) in abstract syntax:

\[
\text{int}\ \text{foo}(\text{Object}\ f)\ {\{}\ \text{synchronized}(f)\ {\{}\ \text{return}\ \text{this.m}();\ {\}};\ {\}}\ {\}}
\]

This compiles to \( \text{sync}_2\ (\text{Var} 1)\ (\text{Var 0.m}()) \).

### 5.3.3 Preservation of well-formedness

Jinja’s proof of \( \text{compP}_1 \) generating well-formed programs sets the ground for JinjaThreads’. Extending it is straightforward except for two aspects:

First, JinjaThreads additionally requires that registers for monitors be not accessed, i.e., condition \( S\ \text{body} \). The next lemma (provable by induction) shows that \( \text{compE}_1 \) ensures this. The interesting cases \( \text{sync}\ (e_1)\ e_2 \) and \( \text{insync}\ (a)\ e \) rely on \( \text{fresh-vname} Vs \) being fresh for \( Vs \).

**Lemma 5.14.** If \( \text{fv}e \subseteq \text{set}Vs \), then \( S (\text{compE}_1\ Vs\ e) \).

Second, preservation of well-typedness requires the stronger induction on the structure of expressions instead of usual induction on the derivation of the typing judgement:

**Lemma 5.15** ([83, Lem. 5.5.]). If \( \text{wf-prog}\ \text{wf-md}\ P\ and\ P, [Vs \rightarrow] Ts \vdash e :: T \) and \( |Ts| = |Vs| \), then \( \text{compP}_1\ P, Ts \vdash_1 \text{compE}_1\ Vs\ e :: T \).
Proof. By induction on \(e\). The interesting new case is \(\text{sync} (e_1) e_2\). From \(P, [\text{Vs} \mapsto \text{Ts}] \vdash \text{sync} (e_1) e_2 :: T\), there is a \(T_1 \neq \text{NT}\) such that \(\text{is-ref} T_1\), \(P, [\text{Vs} \mapsto \text{Ts}] \vdash e_1 :: T_1\), and \(P, [\text{Vs} \mapsto \text{Ts}] \vdash e_2 :: T\) by rule inversion. By induction hypothesis, \(\text{comp} P_1 P, \text{Ts} \vdash_1 \text{comp} E_1 \text{Vs} e_1 :: T_1\). For \(e_2\), it does not suffice to apply the induction hypothesis directly, because this would give \(\text{comp} P_1 P, \text{Ts} \vdash_1 \text{comp} E_1 \text{Vs} e_2 :: T\), instead of

\[
\text{comp} P_1 P, \text{Ts} \vdash [\text{Class Object}] \vdash_1 \text{comp} E_1 (\text{Vs} @ [\text{fresh-vname} \text{Vs}]) e_2 :: T
\]

as required by \(\text{WT}_{1\text{SYNC}}\). This is also the reason why the standard induction rule for \(P, [\text{Vs} \mapsto \text{Ts}] \vdash e :: T\), which Jinja uses, is too weak for this proof. Instead, since \(\text{fresh-vname} \text{Vs}\) is fresh,

\[
P, [\text{Vs} @ [\text{fresh-vname} \text{Vs}] \mapsto_\text{Ts} \vdash [\text{Class Object}]] \vdash e_2 :: T
\]

follows from \([P, E \vdash e :: T; E \subseteq_m E'] \implies P, E' \vdash e :: T\) (provable by rule induction) – and the induction hypothesis applies. \(\square\)

The remaining language-specific well-formedness constraints hold like in Jinja. Hence, preservation of well-formedness follows.

Theorem 5.7. If \(\text{wf-J-prog} P\), then \(\text{wf-J}_1\text{-prog} (\text{comp} P_1 P)\).

5.3.4 Semantic preservation

Semantic preservation for the intermediate language falls in two parts. First, I prove that \(P\) and \(\text{comp} P_1 P\) are delay bisimilar under the semantics \(J_0\) and \(J^*_1\). Second, I show that unlocking a monitor in \(\text{comp} P_1 P\) never fails, i.e., \(J^*_1\) and \(J'_1\) give the same semantics to \(\text{comp} P_1 P\) in spite of \(J'_1\)'s additional reductions \(R_{1\text{UNLCKF}}\) and \(R_{1\text{UNLCKXF}}\).

Semantic preservation for \(\text{comp} E_1\)

\(J_0\) and \(J^*_1\) only differ in the treatment of local variables. Hence, the thread features and arrays that JinjaThreads adds to Jinja do not introduce anything essentially new for the verification. Still, extending the old correctness proof (which uses a big-step semantics) requires substantial changes:

(i) The delay bisimulation between \(J_0\) and \(J^*_1\) must now relate not only initial and final states, but also all intermediate states.
(ii) Since $J_0$ and $f^*_1$ consist of a stack of semantics, the delay bisimulation at one layer composes language-specific constraints and bisimulation relations from the level below (see Figure 5.1), and so do I compose the proofs.

(iii) I must now also show that the small-step reductions preserve the language-specific constraints that the bisimulation proof relies on. Although the simulations are now much finer and covers both directions, the key ideas for the correctness proof [83, §5.5] are still sufficient.

In detail, the bisimulation relation $e_0 \approx_1$ at the level of expressions is naturally the heart of the correctness proof, because the translation’s core is at this level. $t_0 \approx_1$ extends $e_0 \approx_1$ to call stacks; $m_0 \approx_1$ lifts $t_0 \approx_1$ to the interleaving semantics as described in §5.1.3. Hence, I want to prove that $0 \approx_1$ satisfies the assumptions of locale $m$-dbisim-div.

Most of these assumptions are simulation properties of the following form: Given two related states, if either can reduce in a given way, then the other can also reduce correspondingly such that the resulting states are related again. These properties can be derived from Theorem 5.8 (forward direction) and Theorem 5.9 (backward direction) in Figure 5.18, which I now discuss in detail.

Consider the assumptions of the theorems first. The central relation $Vs \vdash (e_0, xs_0) \approx (e_1, xs_1)$ fully encapsulates the relation between $(e_0, xs_0)$ and $(e_1, xs_1)$. The others, i.e., $fv e_0 \subseteq set Vs$ and $|Vs| + \text{max-vars } e_1 \leq |xs_1|$ and $D e_0 \vdash [dom xs_0]$, are only language-specific constraints that involve either of them. To improve proof automation, there are separate preservation lemmata for the latter. Consequently, only $Vs \vdash (e'_0, xs'_0) \approx (e'_1, xs'_1)$ appears in the conclusion. In detail, $Vs \vdash (e_0, xs_0) \approx (e_1, xs_1)$ predicates that

(a) the initialised local variables are the same, i.e., $xs_0 \subseteq_m [Vs [\mapsto] xs_1]$,
(b) $e_1$ adheres to the numbering scheme for variables, i.e., $B e_1 [|Vs|]$,
(c) for all insync$_i$ (a) _ subexpressions of $e_1$, $xs_1$ stores $Addr a$ in register $i$, and
(d) $e_0$ and $e_1$ are identical except for (i) variable names which are resolved according to the compilation scheme and (ii) local variable blocks where $xs_1$ may store the initialisation’s value of $e_0$ and the
Theorem 5.8. Let $\text{wf-J-prog } P$ and $Vs \vdash (e_0, xs_0) \approx (e_1, xs_1)$.
Let $fv e_0 \subseteq \text{set } Vs$ and $|Vs| + \text{max-vars } e_1 \leq |xs_1|$. Suppose that $P, t \vdash (e_0, (h, xs_0)) \rightarrow_{0}^{e} (e'_0, (h', xs'_0))$. Then, there are $ta_1, e'_1$, and $xs'_1$ such that $Vs \vdash (e'_0, xs'_0) \approx (e'_1, xs'_1)$ and the following hold:

(i) If $\tau\text{-move}_0 P h e_0$, then $h' = h$ and $ta_1 = (\emptyset)$ and $(e_1, (h, xs_1))$ reduces in $f^\#_1$ with at least one $\tau$-move to $(e'_1, (h', xs'_1))$.

(ii) If $\neg \tau\text{-move}_0 P h e_0$ and call $e_0 \neq \text{None}$ and call$_1 e_1 \neq \text{None}$, then $\neg \tau\text{-move}_1 (\text{compP}_1 P) h e_1$ and comp$_1 P, t \vdash (e_1, (h, xs_1)) \rightarrow_{0}^{e^\#} (e'_1, (h', xs'_1))$ and $ta_0$ is $0\simeq_1$-bisimilar to native-TA2$J_1 (\text{compP}_1 P) ta_1$.

(iii) Otherwise, there are $e''_1$ and $xs''_1$ such that $(e_1, (h, xs_1))$ reduces in $f^\#_1$ with (possibly no) $\tau$-moves to $(e''_1, (h, xs''_1))$ and $\neg \tau\text{-move}_1 (\text{compP}_1 P) h e''_1$ and comp$_1 P, t \vdash (e''_1, (h, xs''_1)) \rightarrow_{0}^{e^\#} (e'_1, (h', xs'_1))$ and $ta_0$ is $0\simeq_1$-bisimilar to native-TA2$J_1 (\text{compP}_1 P) ta_1$.

Theorem 5.9. Let $\text{wf-J-prog } P$ and $Vs \vdash (e_0, xs_0) \approx (e_1, xs_1)$.
Let $fv e_0 \subseteq \text{set } Vs$ and $|Vs| + \text{max-vars } e_1 \leq |xs_1|$ and $D e_0 [\text{dom } xs_0]$. Suppose that $\text{compP}_1 P, t \vdash (e_1, (h, xs_1)) \rightarrow_{0}^{e^\#} (e'_1, (h', xs'_1))$. Then, there are $ta_0, e'_0$, and $xs'_0$ such that $Vs \vdash (e'_0, xs'_0) \approx (e'_1, xs'_1)$ and the following hold:

(i) If $\tau\text{-move}_1 (\text{compP}_1 P) h e_1$, then $h' = h$ and $ta_0 = (\emptyset)$ and $(e_0, (h, xs_0))$ reduces in $f^\#_1$ with $\tau$-moves to $(e'_1, (h', xs'_1))$. If this involves no $\tau$-moves, then cnt-IB $e'_1 < \text{cnt-IB } e_1$.

(ii) If $\neg \tau\text{-move}_1 (\text{compP}_1 P) h e_1$ and call $e_0 \neq \text{None}$ and call$_1 e_1 \neq \text{None}$, then $\neg \tau\text{-move}_0 P h e_0$ and $P, t \vdash (e_0, (h, xs_0)) \rightarrow_{0}^{e} (e'_0, (h', xs'_0))$ and $ta_0$ is $0\simeq_1$-bisimilar to native-TA2$J_1 (\text{compP}_1 P) ta_1$.

(iii) Otherwise, there are $e''_0$ and $xs''_0$ such that $(e_0, (h, xs_0))$ reduces in $f^\#_1$ with (possibly no) $\tau$-moves to $(e''_0, (h, xs''_0))$ and $\neg \tau\text{-move}_0 P h e''_0$ and $P, t \vdash (e''_0, (h, xs''_0)) \rightarrow_{0}^{e} (e'_0, (h', xs'_0))$ and $ta_0$ is $0\simeq_1$-bisimilar to native-TA2$J_1 (\text{compP}_1 P) ta_1$.

Figure 5.18: Simulation theorems on the expression level for compilation stage 1
block is uninitialised in $e_1$. Such differences in initialisations may only occur in subexpressions that the semantics reduces next. Moreover, the other subexpressions must not contain $\text{insync\_} (\_ \_)$ blocks, i.e., $\neg \text{has\_I\_}$.

Note that I need not require (c) for $\text{sync\_} (\_ \_)$ expressions, because they have not yet stored the monitor address in the registers.

The language-specific constraints are similar to Jinja’s correctness proof. First, to ensure that register allocation succeeds, $fv e_0 \subseteq set V_s$ expresses that $V_s$ captures all free variables in $e$. Second, $|V_s| + \text{max-vars} e_1 \leq |x_{S_1}|$ guarantees that $x_{S_1}$ is large enough to hold all local variables during execution. The third constraint $\mathcal{D} e_0 \{\text{dom } x_{S_0}\}$ only appears in Theorem 5.9 and is new compared to Jinja. It ensures that $J_0$ does not get stuck when looking up a local variable in $x_{S_0}$. This is necessary because $J_1^\#$ does not distinguish initialised and uninitialised variables.

Now, turn to the conclusions. Case (i) corresponds to the $\tau$-move simulation diagrams for well-founded delay bisimulations in Figure 5.6b. Theorem 5.8 always proves the left column, i.e., $J_1^\#$ simulates every $\tau$-move in $J_0$ by at least one $\tau$-move. In contrast, $J_1$ deinitialises local variable blocks before it executes the block’s body (see §5.3.1), which has no counterpart in $J_0$. Hence, Theorem 5.9(i) allows $J_0$ to stall when the number of initialised blocks decreases – the measure $\text{cnt\_IB} e_1$ counts the initialised blocks in $e_1$.

Case (iii) corresponds to the visible moves simulating observable moves (Figure 5.6a). Case (ii) is the special case when both $e_0$ and $e_1$ pause at a call. In that case, no $\tau$-moves may precede the simulating move. Remember that $\text{simulation\_} \approx 1$ and $\text{simulation\_} \approx 2$ of locale $m\text{-dbisim\_} \div\text{div}$ require this for processing the removal from a wait set. Both cases require the thread actions to be bisimilar, i.e., identical except for thread-local states of spawned threads, which must be $\approx 1$-related. This is what well-formedness (premise $\text{wf\_J\_prog } P$) is necessary for. Bisimilarity (defined below) involves definite assignment and no free variables.

Both theorems are proven by induction on the derivation of the reduction. The only interesting cases are for local variables and synchronisation blocks, but Jinja’s notions of hidden and unmodified variables [83, §5.5] suffice for that. Again, it is essential that $\text{fresh\_vname } V_s$ is fresh.
Now, it is clear what the bisimulation relations should be. Remember that the expressions in $J_0$ call frames are closed, i.e., $Vs = []$ and $xs_0 = empty$. Hence,

$$e_0 \approx_1 (e_1, xs_1) \iff [\vdash (e_0, empty) \approx (e_1, xs_1) \land \text{fv } e_0 = \emptyset \land \mathcal{D} e_0 [\emptyset] \land \text{max-vars } e_1 \leq |xs_1|$$

Lifting to single threads is straightforward. The heaps must be the same, the call stacks must be $0\approx_1$-related pointwise, and all call frames except the top pause at a call.

$$((e_0, es_0), h_0) \approx_1 (((e_1, xs_1), exs_1), h_1) \iff h_0 = h_1 \land e_0 \approx_1 (e_1, xs_1) \land |es_0| = |exs_1| \land$$

$$(\forall (e'_0, (e'_1, xs'_1)) \in \text{set } (zip es_0 exs_1).$$

For $0\approx_1$, I take $\approx_m$ from §5.1.3 instantiated with $0\approx_1$ and $0\approx_1$, where

$$(e_0, es_0) \approx_1 ((e_1, xs_1), exs_1) \iff \text{call } e_0 \neq \text{None} \land \text{call } e_1 \neq \text{None},$$

i.e., threads in wait sets must pause at a call.

**Lemma 5.16.** Let $wf-J$-prog $P$. $0\approx_1$ is a delay bisimulation with explicit divergence for $J_0$-red $P$ $t$ and $J_1$-red $\text{comp } 1 P$ $t$.

**Proof.** Since $J_0$ and $J_1$ have similar call-stack semantics (Figures 5.13 and 5.16), it follows easily with Theorems 5.8 and 5.9 that $0\approx_1$ is a well-founded delay bisimulation with well-founded relations $<_0$ and $<_1$, where

$$((e'_0, es'_0), es_0) <_0 ((e_0, es_0), es_0) \iff \text{False}$$

$$((e'_1, xs'_1), exs'_1) <_1 ((e_1, xs_1), exs_1) \iff \text{cnt-IB } e'_1 < \text{cnt-IB } e_1$$

From this, the statement follows by Lemma 5.2. □

**Theorem 5.10.** Let $wf-J$-prog $P$. $\approx_m$ is a delay bisimulation with explicit divergence for $J_0$-red $T$ $P$ and $J_1$-red $\text{comp } 1 P$ $T$ that preserves final states.
5.3. Register allocation

Proof. It suffices to show that the assumptions of locale \(m\)-dbisim-div are met. Lemma 5.16 discharges the first. Preservation of final states and heap changes is trivial, because (i) finality is invariant under \(\approx\) and (ii) \(\approx\) only imposes equality on the heaps, but does not otherwise depend on it. \(\approx\) is provable by induction and case analysis, whereas \(\approx\) and \(\approx\) follow from case (ii) of Theorems 5.8 and 5.9.

\(\square\)

Lemma 5.17. If \(wf-J-prog\ P\) and \(wf-start\ P\ C M\ vs\), then

\[J-start\ P\ C M\ vs\ \approx\ J-start\ (compP\ P)\ C M\ vs\.

Equivalence of \(J\) and \(J\)

The compiler verification has to show that unlocking a monitor in compiled code never fails. The intermediate language is the right place for this, because its semantics already stores the monitor address in the registers like bytecode, but the syntax still enforces the structured locking discipline.

I prove that \(J\).redT\(P\) and \(J\).redT\(P\) are the same for a multithreaded state \(s_1\) in which the lock state agrees with the insync_(_)_subexpressions of the threads. Agreement (notation lock-conf\(f_1\)) is defined analogously to lock-conf in §3.3.4. In such a state, \(J\).redT\(P\) never picks R\(U\)n\(C\)k\(F\) nor R\(U\)n\(C\)k\(X\)F, because the precondition of the thread action \(\{UnlockFail\rightarrow a\}\) is violated. \(J\).redT\(P\) preserves lock-conf\(f_1\) under the following condition (notation \(\approx\)) that for every call frame \((e_1, xs_1)\) of every thread,

(i) for all subexpressions \(insync_i\ (a)\ e'_1\) of \(e_1, xs_1\) stores Addr\(a\) in register \(i\) and \(e'_1\) does not modify register \(i\) and

(ii) ok-I\(e_1\), i.e., all insync_(_)_subexpressions of \(e_1\) lie on one path from the root in \(e_1\)’s abstract syntax tree (see Figure 3.33).

Hence, I define \(\approx\) as follows:

\[s_1\approx_1 s'_1 \iff s_1 = s'_1 \land lock-conf_1 (locks s_1) (thr s_1) \land s_1 \sqrt{\}

\[249\]
Theorem 5.11. If \( \text{wf-J}_1 \text{-prog } P \), then \( 1 \equiv_1 \) is a strong bisimulation for \( J_1^\# \text{.red } T P \) and \( J'_1 \text{.red } T P \).

Corollary 5.3. If \( \text{wf-J}_1 \text{-prog } P \), then \( 1 \equiv_1 \) is a delay bisimulation with explicit divergence for \( J_1^\# \text{.red } T P \) and \( J'_1 \text{.red } T P \) that preserves final states.

Lemma 5.18. If \( \text{wf-J}_1 \text{-prog } P \) and \( \text{wf-start } P C M \text{ vs} \), then

\[
J_1 \text{-start } P C M \text{ vs } 1 \equiv_1 J_1 \text{-start } P C M \text{ vs}.
\]

5.4 Code generation

The first stage has already replaced variable names by register numbers. The second stage \( \text{compP}_2 \) now completes the translation in that it generates the bytecode instructions and exception tables for the expressions (§5.4.1). In this section, I show preservation of well-typedness (§5.4.2) and semantics (§5.4.3) for \( \text{compP}_2 \).

5.4.1 Compilation stage 2

The second stage of the compiler translates expressions into instruction lists (function \( \text{compE}_2 :: 'addr \text{ expr}_1 \Rightarrow 'addr \text{ instr list} \)) and exception tables (function \( \text{compxE}_2 :: 'addr \text{ expr}_1 \Rightarrow \text{pc} \Rightarrow \text{nat} \Rightarrow \text{ex-table} \)). \( \text{compP}_2 \) lifts \( \text{compE}_2 \) and \( \text{compxE}_2 \) to programs using \( \text{compP} \) and computes the maximum stack size \( \text{max-stack} \) and register size using \( \text{max-vars} \).

\[
\text{compP}_2 = \text{compP compMb}_2
\]

\[
\text{compMb}_2 C M Ts T body = \ (\text{let ins } = \text{compE}_2 body @ [\text{Return}]; \ xt = \text{compxE}_2 body 0 0 \ \text{in} \ (\text{max-stack body}, \text{max-vars body}, \text{ins}, \text{xt}))
\]

For JinjaThreads, I have extended Jinja’s \( \text{compE}_2 \) and \( \text{compxE}_2 \) to \( \text{sync}_i (e_1) e_2 \) expressions, on which I focus in this section. For details on the others, see [83, §5.3] and Appendix B.9.3.

The translation of a \( \text{sync}_i (e_1) e_2 \) expression to bytecode must ensure that the monitor is unlocked even if an unhandled exception occurs in \( e_2 \). An exception handler, which applies to all exceptions, needs to do this. Thus, the instructions for \( \text{sync}_i (e_1) e_2 \) are
5.4. Code generation

\( \text{compE}_2 \, e_1 @ [\text{Dup}, \text{Store } i, \text{MEnter}] @ \)
\( \text{compE}_2 \, e_2 @ [\text{Load } i, \text{MExit}, \text{Goto } 4] @ [\text{Load } i, \text{MExit}, \text{ThrowExc}] \)

First, the monitor expression \( e_1 \) is evaluated, its result on the stack duplicated and stored in register \( i \); \text{MEnter} locks the monitor. Then, the block \( e_2 \) is executed, the monitor address loaded back from register \( i \) and the monitor unlocked. \text{Goto } 4\) jumps to the instruction after the exception handler that follows. The handler also loads the monitor address, unlocks the monitor and rethrows the caught exception whose address is still on top of the stack.

Since the exception tables contain absolute program counters and stack depth, \( \text{compxE}_2 \) takes the current program counter \( pc \) and stack depth \( d \) as parameters. For \( \text{sync}_i (e_1) \, e_2 \), \( \text{compxE}_2 \) appends to the exception tables for \( e_1 \) and \( e_2 \) the entry \( (pc_1, pc_2, \text{Any}, pc_2 + 3, d) \) where \( pc_1 = pc + |\text{compxE}_2 \, e_1| + 3 \) and \( pc_2 = pc_1 + |\text{compxE}_2 \, e_2| \). Hence, the entry matches all exceptions that \( e_2 \)'s instructions raise. Since it is placed at the end, it does not take precedence over exception handlers in \( e_2 \).

Reconsider the compilation example in §5.3.2. \( \text{compE}_2 \) compiles the method body \( \text{sync}_2 (\text{Var } 1) (\text{Var } 0. m [[]]) \) to

\[
[\text{Load } 1, \text{Dup}, \text{Store } 2, \text{MEnter}, \text{Load } 0, \text{Invoke } m \, 0, \text{Load } 2, \text{MExit},
\text{Goto } 4, \text{Load } 2, \text{MExit}, \text{ThrowExc}, \text{Return}]
\]

with exception table \([(4, 6, \text{Any}, 9, 0)]\).

As I have already discussed in §2.2.1, \( \text{Any} \) is important for the verification, although \( [\text{Throwable}] \) would also do in theory. Since the latter only applies to subclasses of \( \text{Throwable} \), the bisimulation relation would have to ensure that only such exceptions are ever raised. This would pull in the complete type safety proof of the JVM and therefore severly complicate the proof.

5.4.2 Preservation of well-formedness

To show that \( \text{compP}_2 \) preserves well-formedness, Jinja defines a type compiler that computes a well-typing for the generated bytecode [83, §5.9]. Since JinjaThreads‘ extensions for arrays and synchronisation naturally fit in the compilation scheme, I only present the final theorem – see the formalisation [106] for the full details.

**Theorem 5.12.** If \( \text{wf-J}_1 \text{-prog } P \), then \( \text{wf-jvm-prog } P \).
5.4.3 Semantic preservation

The translation from the intermediate language to bytecode is the most complicated one. It flattens the tree structure of expressions to a linear list of instructions. Exception handlers are registered in exception tables. Synchronized blocks are implemented by MEnter and MExit instructions and an exception handler.

To show delay bisimilarity, I first must define which VM transitions are unobservable, i.e., \( \tau \)-moves. Exception handling and the following instructions generate only \( \tau \)-moves: Load, Store, Push, Pop, Dup, Swap, BinOp, Checkcast, Instanceof, Goto, IfFalse, ThrowExc, and Return. Additionally, Invoke generates a \( \tau \)-move only if the called method is non-native or one of the native methods hashcode or currentThread.

Like between \( J_0 \) and \( J_1^# \), the key to correctness is delay bisimilarity on the expression level, on which I focus in this section. Calling and returning from methods works similarly in \( J_1^# \) and the JVM, the laborious, but uninteresting proof lifts delay bisimilarity. The multithreaded level is the interleaving semantics in both semantics. Hence, I leverage Theorem 5.3 once more to show delay bisimilarity for \( J_1^# \) and the JVM.

For the expression level, I take a detour via two new bytecode semantics exec-meth and exec-methd that differ from the VM in when they get stuck (see Figure 5.1). A single step of execution is written \( (chk, P, ins, xt, t \vdash ((stk, loc, pc, xcp), h)) \rightarrow^{e}_{\text{jvm}} ((stk', loc', pc', xcp'), h') \): If the exception flag \( xcp \) is None, it denotes that the sanity check \( chk \) succeeds (see below), \( pc \) points to an instruction in \( ins \) and \((stk', loc', pc', xcp')\) describes a possible successor state of executing instruction \( ins_{[pc]} \) with stack \( stk \) and registers \( loc \) according to exec-instr; \( ta \) is the corresponding thread action. If the exception flag is set, it denotes that \( xt \) contains an exception handler at \( pc' \) that applies and no stack underflow occurs. Since I am at the expression level, the step must not change the length of the call stack, i.e., neither return from a method nor call a non-native method.

The parameter \( chk \) controls when the semantics gets stuck. For exec-meth, \( chk \) ensures that the stack does not underflow and that jumps only go to program counters between 0 and \( |pc| \) inclusive. Since it is stricter than the aggressive VM, steps in exec-meth are preserved when I enlarge the instruction list at either end and extend the stack at the bottom. The inductive cases in the simulation proof rely on this. But it is not as strict as exec-methd, where \( chk \) performs all checks of check-instr.
5.4. Code generation

Since exec-meth gets stuck only when red'₁ is also stuck, I use exec-meth for simulating red'₁'s reductions.

For the other direction, I use exec-methd, because it gets stuck more often than red'₁. For ThrowExc, e.g., check-instr requires that the thrown exception is a subclass of Throwable, but red'₁ does not. Hence, exec-methd cannot simulate red'₁ unless the bisimulation relation excludes such cases, e.g., by requiring bytecode conformance. But this would further complicate the proofs which are already tedious. Conversely, red'₁ cannot simulate exec-meth because the former gets stuck, e.g., when trying to access a field of an integer, but the latter carries on with undefined behaviour.

As the generated bytecode is well-formed (Theorem 5.12), exec-methd simulates exec-meth for conformant states and preserves bytecode conformance. Note that conformance does not complicate proofs any more at this level of abstraction, because I do not have to unfold its definition for individual instructions. This closes the circle of simulations. In principle, it should be possible to define chk such that the bytecode semantics gets stuck whenever red'₁ does. Since this appears to be very tedious, error-prone and sensitive to even small changes, I have not attempted to do so.

Note that this detour only affects the semantics, not the bisimulation relation ₁≡jvm. As before, ₁≡jvm consists of two parts, (i) a relation between J₁ call frames and JVM expression-level states and (ii) well-formedness conditions of the states that the semantics preserve individually. The relation P, e, h ⊢ (e₁, xs₁) ≈ (stk, loc, pc, xcp) relates a J₁ call frame (e₁, xs₁) (expression and local variables) to a JVM expression-level state (stk, loc, pc, xcp) for a heap h that is the same for both. P only defines the class hierarchy, whereas the expression e :: `addr expr₁ compiles to the instruction list ins = compE₂ e and xt = compxE₂ e 0 0. The inductive definition for ≈ mirrors the reduction rules of red'₁ and relates a partially evaluated expression e₁ with the corresponding stack stk and registers loc, and the instruction position pc in the compiled code.

Figure 5.19 shows some representative rules from the inductive definition. The single rule B₁ for all expressions exploits that the last instruction in a compiled expression always puts its result value on top of the stack. Unfortunately, this does not extend to exceptions, because bytecode does not propagate exceptions from subexpressions, but uses exception tables. Hence, ≈ contains separate exception propagation rules for every expression, similar to B₂. Still, it abstracts from computed
Figure 5.19: Example introduction rules for the relation $\approx$

values and addresses of thrown exceptions and only requires that they are the same in both states. Moreover, rules like $B_3$ for all subexpressions of all expressions embed bisimilar states for the subexpression into the context of the larger expression, thereby shifting the stack and instruction pointer as necessary. Finally, the definition contains a rule for every bytecode instruction and corresponding $J_1$ state. For example, $B_4$ relates the $J_1$ state which next acquires the monitor’s lock to the intermediate JVM state after executing the $\text{Store } i$ instruction that stores the monitor address. Although $J'_1$ and the JVM operate on the local variable array in the same way, the bisimulation relation must not require that $xs$ and $loc$ be equal, because they differ in such intermediate states like in $B_4$, which $R_1\text{lock}$ skips.

The bisimulation relation $\varepsilon_{jvm}$ specialises this relation to complete call frames as follows:

$$
P, h \vdash (e_1, xs_1) \varepsilon_{jvm} (xcp, stk, loc, C, M, pc)
$$
The bisimulation relation $1 \approx_{jvm}^t$ for single threads lifts $1 \approx_{jvm}^e$ to call stacks. Although great care is required to ensure that everything fits together, the construction and its verification just reuses the ideas from the first compilation stage. Hence, I conclude that the second stage is correct, too.

**Theorem 5.13** (Correctness of stage 2). In locale conf-read, suppose that $wf-J_1$-prog $P$. Then,

(i) $1 \approx_{jvm}^t$ is a delay bisimulation with explicit divergence for $J'_1$-red $P$ $t$ and the defensive VM $jvm$-execd $(compP_2 P)$ $t$.

(ii) $1 \approx_{jvm}^m$ is a delay bisimulation with explicit divergence for $J'_1$.red$T$ $P$ and $jvmd$.red$T$ $(compP_2 P)$ that preserves final states.

**Lemma 5.19.** If $wf-J_1$-prog $P$ and $wf$-start $P$ $C$ $M$ $vs$, then

$$J_1$-start $P$ $C$ $M$ $vs$ $1 \approx_{jvm}^m$ jvm-start $(compP_2 P)$ $C$ $M$ $vs$.

## 5.5 Complete compiler

In the previous sections, I have shown that all relations in Figure 5.1 are delay bisimulations with explicit divergence. Now, it remains to compose these results for the full compiler $J2JVM = compP_2 \circ compP_1$.

Preservation of well-formedness follows immediately from Theorems 5.7 and 5.12.

**Theorem 5.14.** If $wf-J$-prog $P$, then $wf$-jvm-prog $(J2JVM P)$.

For semantic preservation, let $(J \approx_{jvm}, J \sim_{jvm})$ be the composition of all multithreaded delay bisimulations, i.e.,

$$(J \approx_{jvm}, J \sim_{jvm}) = m \bowtie_B 0 \bowtie_B 1 \bowtie_B 1 \bowtie_B 1 \approx_{jvm}$$

Lemmata 5.12, 5.17, 5.18, and 5.19 show that $j \approx_{jvm}$ relates the start states. And Lemma 5.1 composes the bisimulation theorems 5.6, 5.10, 5.13 and Corollary 5.3 to obtain the main correctness theorem 5.16.

**Theorem 5.15.** If $wf-J$-prog $P$ and $wf$-start $P$ $C$ $M$ $vs$, then

$$J$-start $P$ $C$ $M$ $vs$ $j \approx_{jvm}$ jvm-start $(J2JVM P)$ $C$ $M$ $vs$.
Theorem 5.16. In locale conf-read, if wf-J-prog P, then \((J \approx_{jvm}, J \sim_{jvm})\) is a delay bisimulation with explicit divergence that preserves final states.

\[
\begin{align*}
\text{dbisim-div} & \; (J.\text{redT} \; P) \; (jvmd.\text{redT} \; (J2JVM \; P)) \; J \approx_{jvm} J \sim_{jvm} \\
\text{dbisim-final} & \; (J.\text{redT} \; P) \; (jvmd.\text{redT} \; (J2JVM \; P)) \; J \approx_{jvm} J \sim_{jvm} \\
& \; (J.\text{mτ-move} \; P) \; (jvm.\text{mτ-move} \; (J2JVM \; P)) \\
& \; J.\text{mfinal} \; jvm.\text{mfinal}
\end{align*}
\]

All proofs have been conducted independently of the heap module implementation in various locales. Since \(t \approx_{jvm}\) imposes bytecode conformance on the JVM state to show that exec-meth and exec-methd are equivalent, Theorem 5.16 holds for all heap implementations that satisfy conf-read. These are sequential consistency (§4.2, Theorem 4.1) and JMM heap implementation 2 (§4.3.2, Theorem 4.2).

Since \(J \approx_{jvm}\) decomposes into the relations for each stage, I now can break down correctness in terms of bisimilarity to concrete executions \(\xi\), i.e., trace equivalence, using the Theorems 5.1 and 5.2 about semantic preservation. As before, I decorate the semantics arrows with \(J\) and \(jvmd\) to refer to the definition with the parameters appropriately instantiated.

Theorem 5.17 (Correctness). Let wf-J-prog and wf-start \(P \; C \; M \; vs\).

(a) Let \(P \vdash J.\text{start} \; P \; C \; M \; vs \; \downarrow J \; \xi\). Then, there is a \(\xi'\) such that \(J2JVM \; P \vdash jvm.\text{start} \; (J2JVM \; P) \; C \; M \; vs \; \downarrow jvmd \; \xi'\) and \(\xi \; [J \approx_{jvm}, J \sim_{jvm}] \; \xi'\). In particular:

(i) If \(\xi\) terminates in \(s\) and \(J.\text{mfinal} \; s\), then \(\xi'\) terminates in the \(jvm\)-final state \(\text{mexception} \; s\).
(ii) If \(\xi\) deadlocks in state \(s\), then \(\xi'\) deadlocks in some \(s'\), too.
(iii) If \(\xi\) diverges or runs infinitely, so does \(\xi'\).

(b) Let \(J2JVM \; P \vdash jvm.\text{start} \; (J2JVM \; P) \; C \; M \; vs \; \downarrow jvmd \; \xi'\). Then there is a \(\xi\) such that \(P \vdash J.\text{start} \; P \; C \; M \; vs \; \downarrow J \; \xi\) and \(\xi \; [J \approx_{jvm}, J \sim_{jvm}] \; \xi'\). In particular:

(i) If \(\xi'\) terminates in \(s'\) and \(jvm.\text{mfinal} \; s'\), then \(\xi\) terminates in some \(s\) such that \(J.\text{mfinal} \; s\) and \(s' = \text{mexception} \; s\).
(ii) If \(\xi'\) deadlocks in state \(s'\), then \(\xi\) deadlocks in some \(s\), too.
(iii) If \(\xi'\) diverges or runs infinitely, so does \(\xi\).

In any case, \(\xi\) and \(\xi'\) produce the same sequence of events, i.e., map events \(\xi = \text{map events} \; \xi'\) – in particular, the same output.
5.6 Discussion

**Proof outline.** Case (a) states that every execution of the source code has a corresponding execution of the bytecode, and case (b) states the converse. The main statements directly follows from Theorems 5.1 and 5.2.

The subcases (i) to (iii) only specialise this statement to the concrete bisimulation. For a final J state s, the function mexception s extracts the correct exception flag for every thread in s, i.e., None for normal termination and \([a]\) if the exception at address a caused the abrupt termination.

Preservation of deadlocks in subcase (ii) does not follow directly, because I have defined deadlock in terms of the semantics of single threads, not the interleaved semantics that the bisimulation is about. However, Theorem 5.4 shows that \(m \approx 0\), \(m \approx 1\), and \(m \approx jvm\) preserve deadlocks. For \(m \approx 1\), it is easy to show that the semantics for \(J^\#_1\) and \(J'_1\) differ only in states that cannot be in deadlock. Hence, \(m \approx 1\) preserves deadlocks, too. Since s and s’ are stuck by construction of \(\xi\) and \(\xi'\), there are no \(\tau\)-moves that Theorem 5.4 would allow, i.e., s and s’ are both deadlocked.

While the above theorem correctly describes semantic preservation for sequential consistency, the JMM adds a few layers that are not yet reflected. First, the events for thread start and finish (layer 4) are missing. However, §5.1.4 explains how to extend the bisimulation relations \(m \approx 0\), \(m \approx 1\), and \(m \approx jvm\) to handle them (Lemma 5.4). It is also straightforward to show that I can extend \(m \approx 1\) accordingly. Hence, Theorem 5.16 has an analogue for the JMM stack of semantics. Second, the layers 6 and 7 for complete interleavings and legality of executions not yet covered. By Corollary 5.1, delay bisimilarity enforces that the complete interleavings of \(P\) and \(J^2\) are the same. Thus, so are the legal executions.

**Theorem 5.18.** Let \(wf-J-prog\) \(P\) and \(wf\)-start \(P\) \(C\) \(M\) vs. Then, \((E, ws)\) is a legal execution of \(P\) with start state \(J\)-start \(P\) \(C\) \(M\) vs iff \((E, ws)\) is a legal execution of \(J^2\) with start state \(jvm\)-start \((J^2)\) \(C\) \(M\) vs.

**Corollary 5.4.** Let \(wf-J-prog\) \(P\) and \(wf\)-start \(P\) \(C\) \(M\) vs. Then, \(P\) is correctly synchronised iff \(J^2\) is.

5.6 Discussion

Concurrency poses three challenges to verifying a compiler: (i) non-deterministic interleaving, (ii) different granularity of atomic operations
between source and bytecode, and (iii) memory models for optimisations. In this chapter, I have addressed (i) and (ii).

For nondeterminism, I use bisimulation instead of forward simulation, in which only the compiled program simulates the source program. For bisimulation, it does not suffice to merely show the other direction, but some subtleties arise: First, neither the source nor the compiled program may carry on if the other gets stuck, e.g., due to type errors. In JinjaThreads, the source code semantics is a small-step semantics, whereas the VM is an abstract state machine. Both naturally contain different type checks, only a full type system and type safety proof at every stage would ensure bisimilarity. By using both the aggressive and defensive VM in the simulation proofs, I only need a single type safety proof for bytecode which ensures that both VMs are equivalent for verified bytecode. Therefore, the compiler verification only holds for type-safe heap implementations such as JMM heap implementation 2. In particular, the proofs do not hold for JMM heap implementation 1.

Second, the bisimulation must relate all states that are reachable from either initial state. Ordinary simulations do not have to relate intermediate states in the compiled code which the source code skips. This substantially increases the size of the bisimulation relation and consequently the number of cases the simulation proofs have to consider.

For example, several statements such as sync (_) _ generate multiple instructions. Hence, a single observable step in the source code program is decomposed into a number of silent steps and one observable step in between. Conversely, exceptions slowly propagate up in source code whereas the VM directly jumps to the exception handler.

The interleaving semantics, which I use at all stages, allows to decompose the multithreaded case to single threads, where shared memory accesses and synchronisation must be observable. Hence, I do not have to worry about interleavings in the main correctness proofs themselves. However, this approach also restricts the allowed optimisations to thread-local ones, because observable behaviour must not be changed. Consequently, I cannot exploit the additional flexibility that the JMM allows. For J2JVM, this is irrelevant because it just follows the recommendations in the JVMS [103, Ch. 7] and does not optimise at all. In fact, even Sun’s javac compiler in Java 2 SE optimises only very little, but leaves this to the JIT compiler in the VM.

Classical compiler verifications only cover terminating executions, see, e.g., [83,166]. My correctness result also extends to nontermina-
tion and deadlock. However, the standard (bi-)simulation approach with $\tau$-moves cannot prove this, because infinitely many consecutive $\tau$-moves might be simulated by no moves at all, which is known as the infinite stuttering problem. Instead, my notion of bisimulation explicitly includes divergence, but is nevertheless transitive unlike other approaches.

5.7 Related work

Compiler verification in general has been an active research topic for more than 40 years; see [44] for an annotated bibliography. Ritti [150] and Wand [174] first used bisimulations for compiler verification for a simple, parallel functional language. They showed that running the compiled code on a virtual machine is weakly bisimilar to the source code’s denotational semantics, which ignores divergence.

Most closely related to JinjaThreads’ compiler is naturally Jinja’s [83, §5] which itself builds on Strecker’s [167] for $\mu$Java as part of the VerifyCard project. They handle subsets of sequential JinjaThreads and are verified with respect to the big-step semantics.

More remotely, JinjaThreads’ compiler is related to the one by Stärk et al. [166], which handles only sequential Java source code. As already pointed out in [83], they lack the formal rigour required for machine-checked proofs and their theorem does not imply preservation of non-terminating computations.

As for compiler verification for concurrent Java, Ševčík and Aspinall [162] report on verifying individual compiler optimisations w.r.t. the JMM. They show that (i) the JMM does not allow as many as intended by its designers for programs with data races, and (ii) all trace-preserving compilers are correct. However, their proofs are only on paper for a toy core language without almost all sequential Java features. Hecker [68] has verified two such transformations for the JMM in Isabelle/HOL.

Leroy’s CompCert project [97–99] has been the most remarkable landmark in mechanised compiler verification recently. He has verified a complete compilation tool chain from a subset of C source code to PowerPC assembly language in Coq. CompCert focuses on low-level details and language features such as memory layout, register allocation and instruction selection. JinjaThreads’s simulation diagrams for well-founded delay bisimulations are similar to CompCert’s, but the latter
only require the forward direction (the upper half of Figure 5.6) since CompCert’s target language is deterministic.

As part of the Verisoft project, Leinenbach [96] has verified a non-optimising compiler from C0, a subset of C, directly to DLX assembler in Isabelle/HOL. Like CompCert, he focuses on low-level details and only proves a weak simulation theorem for sequential executions, but not for the backward direction.

Compiler correctness for concurrent programs is an active research under the aspect of memory models. Ševčík et al. [163] have extended parts of CompCert’s backend to concurrency under the x86-TSO memory model (see §4.4.1). In [161], Ševčík studies what compiler optimisations are allowed under the DRF memory model, which provides only the DRF guarantee. Owens [140] and Batty et al. [16,17,154] verify the implementation of language primitives for synchronisation in assembly code for various hardware memory models. Lea [95] and McKenney and Silvera [117] have developed guidelines on how to implement the Java and C++ memory models in modern hardware. These works seem highly relevant for a potential future extension of JinjaThreads’ compiler to machine code.

One challenge in the compiler verification was to prove that unlocking a monitor in bytecode never fails. My proof rests on the syntactic restrictions for the intermediate language. For arbitrary bytecode, Iwama and Kobayashi [73] propose to tag every object with a usage label which specifies a policy on how this object may be locked. A type system, for which they also give a type inference algorithm, guarantees that method implementations respect the usage tags, and the subject reduction theorem ensures soundness. It would be interesting to see whether their type system accepts the bytecode which J2JVM generates – since this would require a considerable extension of their language, I have not attempted to do so. If this held, I could avoid having two semantics of the intermediate language.

To tackle the same problem of proper lock acquisition and release, Laneve [94] presents an operational semantics and a type system for a slightly larger subset of Java bytecode which includes both synchronisation and the wait-notify mechanism. It enforces the structured locking principle that is known from Java source code. Moreover, the soundness proof gives that well-typed programs are free of IllegalMonitorState exceptions, even for calls to wait and notify. Unfortunately, this makes his type system unusable for the verification of JinjaThreads’ compiler, because JinjaThreads allows calls to wait to fail like in Java.
JinjaThreads as a Java interpreter

Given the size and complexity of JinjaThreads, making sure that it faithfully models (a subset of) Java is non-trivial. Rushby [152] and Norrish [133] suggest three ways to address validation: (i) the social process of reviewing, publication, and reuse by others, (ii) challenging the specification by proving sanity theorems, and (iii) validation against a concrete implementation. JinjaThreads has gone all three ways to some extend. First, JinjaThreads continues the line of Bali, µJava, and Jinja. Hence, numerous publications in various venues [24, 79–87, 106, 107, 111–113, 126, 129, 130, 135–138, 146, 156–159, 167, 168] and its reuse in [37] support the claim of faithfulness. Type safety (Chapter 3) and in particular the compiler verification (Chapter 5) are excellent examples for the second. However, bugs may still hide in technical details that publications gloss over and in areas that sanity theorems fail to cover.

Therefore, this chapter takes the third route. Using Isabelle’s code generator, I obtain an executable interpreter, VM, and compiler from the formal definitions and validate the semantics by running Java test programs and comparing the results with Sun’s reference implementation. To make the vast supply of Java programs available for testing, I also developed a translator Java2Jinja from Java to JinjaThreads abstract syntax (§6.5). By running these test cases, I found a bug in the implementation of binary operators (§6.5.2), which all previous proofs were unable to reveal. Although validation through testing can never prove the absence of errors, I am now confident that JinjaThreads faithfully models the Java subset that I claim.

52 Excerpts of this chapter have been published in [113]. Although this is joint work with Lukas Bulwahn, this chapter presents only my work unless noted otherwise.
Another contribution beyond validation are the executable VM, bytecode verifier, and the compiler themselves, which I have automatically extracted from the formalisation. Under the assumption that Isabelle’s code generator is correct, the extracted compiler is verified and the VM is type safe. In general, everything provable about the formalisation also applies to them. This is particularly important in the context of Quis Custodiet, which uses JinjaThreads to verify algorithms for enforcing information flow policies. Running the certified Java programs under the extracted VM ensures that the program behaves indeed as predicted by the static program analysis.

Efficiency of the extracted code matters for both validation through testing and running non-trivial programs, but JinjaThreads focuses on proofs and modularity. Hence, non-negligible complications in specifications and proofs for the sake of direct executability and efficiency were out of the question. In particular, only the compiler is directly executable, since it is written as a functional program. In §6.2 and §6.3, I demonstrate how to combine Isabelle’s code extraction facilities (see §6.1 for an overview) such that (i) changes to the existing formalisation stay minimal and (ii) efficient implementations replace inefficient or even unexecutable ones. Empirical evidence shows that the VM is competitive in performance with another formalised JVM that has been designed for executability and efficiency from the start (§6.3.4).

From Bulwahn’s and my experience, we have distilled simple guidelines on how to develop future formalisations with executability in mind (§6.4). Thus, JinjaThreads as a case study demonstrates that extracting efficient code from large developments is feasible – even if executability is of little concern in their design.

### 6.1 Isabelle code extraction facilities

In its most basic form, code extraction in Isabelle\(^\text{53}\) converts functional programs in HOL into code in a functional programming language like Standard ML, Haskell, OCaml, and Scala. For example, Figure 6.1 shows

\(^{53}\text{In the recent Isabelle literature [61,62], the terms “code generation” and “generated code” refer to automatically obtaining (obtained) code in a functional programming language from formal specifications. To avoid ambiguities with the bytecode that JinjaThreads’s compiler outputs, I use the older term “extraction” [22]. This must not be confused with Coq’s extraction mechanism [102], which is based on the Curry-Howard isomorphism.}
6.1. Isabelle code extraction facilities

```isabelle
datatype tree = Leaf | Node tree int tree

fun preorder :: tree ⇒ int list where
  preorder Leaf = []
| preorder (Node l i r) = i · preorder l @ preorder r
```

(a) Isabelle declaration of binary trees and preorder traversal

```
datatype tree = Leaf | Node of tree * IntInf.int * tree;
fun preorder Leaf = []
| preorder (Node (l, i, r)) = i :: preorder l @ preorder r;
```

(b) Extracted ML code for preorder

```
data Tree = Leaf | Node Tree Integer Tree;
preorder :: Tree -> [Integer];
preorder Leaf = [];
preorder (Node l i r) = i : preorder l ++ preorder r;
```

(c) Extracted Haskell code for preorder

Figure 6.1: Preorder traversal for binary trees in Isabelle/HOL and the extracted code

the declaration of a binary tree datatype `tree` and a pre-order traversal `preorder` in Isabelle (a) and the extracted code in Standard ML (b) and Haskell (c). The extracted code directly mirrors the defining equations and uses the predefined types and operations for lists and integers.

It serves various purposes. It can be either interfaced and executed with other code (see the Java2Jinja tool in §6.5 for an example) or run in the theorem prover’s process to assist the user in proving theorems. Examples for the latter are Isabelle’s counter example generator Quickcheck [23,36] (§7.2), evaluation of ground terms [62], normalisation of HOL terms [5], and proof by normalisation [6].

6.1.1 The code generator

Isabelle’s code generator [62] turns a set of equational theorems into a functional program with the same equational rewrite system, see Figure 6.1 for an example. As it builds on equational logic, the translation guarantees partial correctness by construction, because one could simu-
late every execution step in the functional language by rewriting with the corresponding equational theorem in the logic. Thus, every theorem also holds for any terminating execution of the code. For example, the type safety proof shows that the defensive VM as defined in Isabelle never gets stuck at a type error for programs that the bytecode verifier (§6.2.2) accepts. Thus, we can be sure that running verified bytecode never crashes the extracted JVM because of a type error, either.\textsuperscript{54}

Since only equations matter, the user may easily refine programs and data without affecting her formalisation globally. Program refinement can separate code extraction issues from the rest of the formalisation. As any (executable) equational theorem suffices for code extraction, the user may \textit{locally} derive new (code) equations to use upon code extraction. Hence, existing definitions and proofs remain unaffected, which has been crucial for JinjaThreads.

For example, consider the definition of the prefix relation \textit{prefix} on lists in Figure 6.2. The code generator cannot use the original definition because it contains an existential quantifier which ranges over an infinite type. The user can prove the equations \textit{prefix-code} that pattern-match on the list constructors \texttt{[]} and \texttt{\_\_} such that the code generator uses them instead of the definition. Importantly, she does not touch the original definition, which might break some proofs.

For data refinement, she may replace constructors of a datatype by other constants and derive equations that pattern-match on these new (pseudo-)constructors. Neither need the new constructors be injective and pairwise disjoint, nor exhaust the type. Again, this is local as it affects only code extraction, but not the logical properties of the refined type. Thus, one cannot exploit inside the logic the type’s new structure for code extraction. For example, the lower half of Figure 6.2 defines the pseudo-constructor \textit{Lazy} for implementing lists lazily and states the appropriate code equation \textit{prefix-Lazy} for \textit{prefix}.

Only type constructors can be refined; some special types (such as \texttt{’a \rightarrow ’b for maps}) must first be wrapped in an (isomorphic) type of their

\textsuperscript{54}As equational logic has no notion of execution, the evaluation strategy of the target language still can cause abrupt or non-termination [22]. For example, the code for

\texttt{definition \textit{hd’} :: ’a list \rightarrow ’a where \textit{hd’} \texttt{xs} = (let \texttt{x = hd xs} in if \texttt{xs = []} then \texttt{None else [x])}

raises a \texttt{MATCH} exception in ML when \texttt{hd’} is applied to the empty list, because ML’s eager evaluation tries to compute \texttt{hd [\_]} although \texttt{x} is never used. In contrast, laziness in Haskell makes the definition work fine.
6.1. Isabelle code extraction facilities

**definition** prefix :: `'a list ⇒ `'a list ⇒ bool where
prefix xs ys ←→ (∃zs. ys = xs @ zs)

**lemma** prefix-code [code]:
prefix [] ys ←→ True
prefix (x · xs) [] ←→ False
prefix (x · xs) (y · ys) ←→ x = y ∧ prefix xs ys

**definition** Lazy :: (unit → `'a × `'a list) ⇒ `'a list where
Lazy xs = (case xs () of None ⇒ [] | [(x, xs')] ⇒ x · xs)

**lemma** prefix-Lazy [code]:
prefix (Lazy xs) (Lazy ys) =
  (case xs () of None ⇒ True
           | [(x, xs')] ⇒ (case ys () of None ⇒ False
                           | [(y, ys')] ⇒ x = y ∧ prefix xs' ys'))

**Figure 6.2:** Program and data refinement for the prefix predicate on lists

own (e.g., (‘a, ‘b) mapping). Data refinement allows to implement them as associative lists or red-black trees.

6.1.2 The predicate compiler

The predicate compiler [21] translates specifications of inductive predicates, i.e., the introduction rules, into executable equational theorems for Isabelle’s code generator. The translation is based on the notion of modes. A mode partitions the arguments into input and output. For a given predicate, the predicate compiler infers the set of possible modes such that all terms are ground during execution. Lazy sequences handle the non-determinism of inductive predicates. By default, the equations implement a Prolog-style depth-first execution strategy. Applying the predicate compiler to JinjaThreads’ inductive definitions has initiated the following improvements by Bulwahn [113] over its initial description [21]:

First, mode annotations restrict the generation of code equations to modes of interest. This is necessary because the set of modes is exponential in the number of arguments of a predicate. Therefore, the space and time consumption of the underlying mode inference algorithm grows exponentially in that number; for all applications prior to JinjaThreads, this has never posed a problem. In case of many
arguments (up to 15 in JinjaThreads), the plain construction of this set of modes burns up any available hardware resource. To sidestep this limitation, modes can now be declared and hence they are not inferred, but only checked to be consistent.

Second, he improved the compilation scheme: The previous one sequentially checked which of the introduction rules were applicable. Hence, the input values were repeatedly compared to the terms in the conclusion of each introduction rule by pattern matching. For large specifications, such as JinjaThreads’ semantics (contains 89 rules), this naïve compilation made execution virtually impossible due to the large number of rules. To obtain an efficient code expression, he modified the compilation scheme to partition the rules by patterns of the input values first and then only compose the matching rules – this resembles similar techniques in Prolog compilers, such as clause indexing and switch detection. I report on the performance improvements due to this modification in §6.3.4.

Third, the predicate compiler now offers non-intrusive program refinement, i.e. the user can declare alternative introduction rules. For an example, see §6.3.1.

Fourth, the predicate compiler was originally limited to the restricted syntactic form of introduction rules. Bulwahn added some preprocessing that transforms definitions in predicate logic to a set of introduction rules [36]. For example, the predicate compiler now directly processes the definition of type-safe method overriding `wf-overriding` in Figure 2.9.

### 6.1.3 Data structures

Typically, definitions in proof assistants use standard HOL types for data structures instead of concrete implementations, e.g., `′a →′b` instead of associative lists or red-black trees. This reduces clutter, because proofs need not deal with different representations of the map. Such maps occur in JinjaThreads source code as environment for typing judgements, the local store, and the heap in the SC implementation. For code extraction, however, this approach can lead to highly inefficient code or completely defy extraction (§6.3.4).

Isabelle’s standard library provides associative lists and red-black trees as backing implementations for maps and sets via data refinement (§6.1.1), i.e., the code can treat sets and maps as data. As part of my work, I have generalised this concept to FinFuns [108, 109]. A FinFun is
a function that is almost-everywhere constant. Via data refinement, it is
implemented as an associative list that additionally stores the function’s
value outside the domain of the associative list. For example, a finite map
would store the default value None, a finite set – modelled as a predicate
– False, and complements of finite sets True. Since the FinFun type ‘a ⇒ ‘b
contains only almost-everywhere constant functions, quantifiers and
function equality are decidable. JinjaThreads uses FinFuns for managing
the locks (§3.1.1) and storing lock BTAs in thread actions (§3.1.2).

Some operations remain unexecutable though, since data refinement
does not enrich the refined type’s structure. For example, iteration over
a finite set remains unexecutable when the result depends on the order
in which the iteration visits the elements. The Isabelle Collections Frame-
work (ICF) [91, 92] (joint work with Peter Lammich) advocates dealing
with refinement explicitly in the logic instead of hiding it in the code
generator. Locales abstractly specify the operations, concrete implemen-
tations interpret them. This allows for executing truly underspecified
functions. Additionally, the ICF provides more data structures (tries and
hashing) and a uniform interface for accessing them. JinjaThreads uses
the ICF for refining the non-deterministic interleaving semantics with a
scheduler.

6.1.4 Locales and code extraction

Locales and code extraction do not (yet) go well together. As code
extraction requires equational theorems in the (foundational) theory
context, equational theorems that reside in the context of a locale cannot
serve as code equations directly, but must be transferred into the theory
context. For a simple example, consider a locale L with one parameter
p, one assumption A p and one definition f = ... that depends on p. Let
g be a function in the theory context for which A (g z) holds for all z. I
want to generate code for f where p is instantiated to g z.

The Isabelle code generator tutorial [61] proposes interpretation and
definition: One instantiates p by g z and discharges the assumption with
A (g z), for arbitrary z. This yields the code equation L.f (g z) = ...,
which is ill-formed because the left-hand side applies f to the non-
constructor constant g. For code extraction, one must manually define a

\footnote{Similarly, the predicate compiler only operates on introduction rules in the theory
context, i.e. the same restrictions and solutions apply.}
new function $f'$ by $f'z = L.f(gz)$ and derive $f'z = \ldots$ as code equation. This approach is unsatisfactory for two reasons: It requires to manually re-define all dependent locale definitions in the theory context (and for each interpretation), and the interpretation must be unconditional, i.e. $A(gz)$ must hold for all $z$. In JinjaThreads, the latter is often violated, e.g. $gz$ satisfies $A$ only if $z$ is well-formed.

To overcome these deficiencies, my new approach splits the locale $L$ into two: $L$-base and $L^*$. $L$-base fixes the parameter $p$ and defines $f$; $L^*$ inherits from $L$-base, assumes $A p$, and contains the theorems and proofs from $L$. Since $L$-base makes no assumptions about $p$, the locale implementation exports the equation $f = \ldots$ in $L$-base as an unconditional equation $L$-base.$f p = \ldots$ in the theory context, which directly serves as code equation. For execution, I merely pass $gz$ to $L$-base.$f$. I use this scalable approach throughout JinjaThreads. Its drawback is that the existence of a model for $f$, as required for its definition, must not depend on $L$’s assumptions; e.g., the termination argument of a general recursive function must not require $L$’s assumptions. Many typical definitions (all in JinjaThreads) satisfy this restriction.

6.2 Static semantics

The static semantics fall into generic well-formedness constraints (Figure 2.9) and language-specific ones (Figure 2.10 and §2.2.3). In this section, I describe what is necessary to obtain an executable well-formedness checker for source code and bytecode.

6.2.1 Generic well-formedness

Generic well-formedness (Figure 2.9) poses little problems to code extraction. The direct subclass relation $P \vdash _{\leq_1} _$ and its reflexive transitive closure (RTC) $P \vdash _{\leq^*_} _$ and the subtype relation $P \vdash _{\leq} _$ are inductive predicates that the predicate compiler preprocesses. So does it for $wf$-overriding thanks to Bulwahn’s recent extensions in its preprocessor. Further, since all universal quantifiers are bound by lists, they pose no problem either. Hence, it is easy to extract code for $wf$-prog.

It is harder to see that the extracted code might not terminate for programs with cyclic class hierarchies. The subclass relation $P \vdash _{\leq^*_} _$ builds on the RTC and the prolog-style execution for RTC might loop
6.2. Static semantics

in the hierarchy’s cycle. Instead, I configure the code generator to use a tabled RTC by Berghofer [113, §2.3]. This ensures that querying \( P \vdash _\_ \preceq^* _\_ \) always terminates, i.e., \( \text{wf-prog} \) reliably detects cyclic class hierarchies.

Even less obvious, another source of non-termination hides in the order in which the various well-formedness constraints are executed. \( \text{wf-prog} \) calls \( \text{wf-cdecl} \) for all classes in \( P \) in the order in which they appear in \( P \) – and \( \text{wf-cdecl} \) checks that the current class is not part of a cycle in the class hierarchy (condition \( \neg P \vdash D \preceq^* C \)). Hence, when the other predicates of \( \text{wf-cdecl} \) are being executed, there still may be some undetected cycle in other parts of the hierarchy – in particular, superclasses may be part of that. But method lookup as used in \( \text{wf-overriding} \) recurses over the class hierarchy and – like the default setup for \( P \vdash _\_ \preceq^* _\_ \) – might not terminate. The simplest fix is to check acyclicity up-front. Hence, the code generator uses the following equation for \( \text{wf-prog} \):

\[
\text{wf-prog} \begin{array}{l}
\text{wf-md } P \iff \\
\text{acyclic-hierarchy } P \land \text{wf-syscls } P \land \\
\text{distinct } (\text{map } \text{fst } P) \land (\forall \text{cd } \in \text{set } (\text{classes } P). \text{wf-cdecl’ } \text{wf-md } P \text{ cd})
\end{array}
\]

\[
\text{acyclic-hierarchy } P \iff \\
(\forall (C, D, ms, fs) \in \text{set } (\text{classes } P). C \neq \text{Object } \rightarrow P \vdash D \preceq^* C)
\]

where \( \text{wf-cdecl’} \) differs from \( \text{wf-cdecl} \) only in that \( \text{wf-cdecl’} \) omits the acyclicity test \( \neg P \vdash D \preceq^* C \).

Both sources of non-termination have already flawed Jinja’s and \( \mu \text{Java} \)’s bytecode verifiers. They are generally hard to track down, because HOL cannot reflect the notion of computation that is inherent in the extracted code. Hence, only extensive testing as I have done for JinjaThreads with Quickcheck (§7.2) and Java2Jinja (§6.5) can provide some level of confidence.

6.2.2 The bytecode verifier

The bytecode verifier has to check statically that every execution of the program meets at run-time the assumptions that the VM makes (§2.2.3). To that end, the JVMS [103] specifies a verification procedure that interprets bytecode methods abstractly. Like Jinja, JinjaThreads models the verification as a data flow analysis (DFA) using Klein’s DFA framework [79]. The abstract values are state types, applicability \( \text{app} \) and effect \( \text{eff} \) implement the transfer functions (see §2.2.3). Hence, bytecode
verification is to compute a solution to the data flow problem, using the summary information for methods, i.e. parameter and return types, from the program declaration. Any solution of the data flow analysis yields a consistent method-well typing. Since JinjaThreads only extends Jinja’s bytecode verifier to the new instructions, I omit a detailed presentation and refer the interested reader to [83].

The DFA framework solves DFA problems using Kildall’s worklist algorithm [78] under the following prerequisites that ensure correctness and termination: First, there is no infinitely ascending chain of valid types, all transfer functions are monotone and preserve validity of types. Second, the least upper bound of two abstract values is computable and a valid type.

Only proving the ascending chain condition (ACC, notation acc) and computing least upper bounds (lub) are novel. In Jinja, the set of valid types is finite, i.e., acyclicity of the class hierarchy ensures the ACC. JinjaThreads, however, contains infinitely many types due to unbounded array dimensions. In fact, as described in §2.1.2, the subtype relation does contain infinitely ascending chains. The key to proving ACC and computing lubs is that single inheritance in JinjaThreads allows to organise valid reference types other than $NT$ in a tree rooted at $Class\ Object$ such that valid supertypes are exactly the transitive parents – see Figure 2.6 for the Hasse diagram. Hence, the distance from the root $Class\ Object$, i.e., the depth, is a measure on array and class types for subtyping. The function $inheritance\text{-}level\ P\ C$ recurses over $P$’s class hierarchy to determine the number of superclasses of $C$; the acyclicity check ensures that the recursion always terminates. Then, the function $subtype\text{-}measure$ computes the depth in the tree.

\[
\begin{align*}
subtype\text{-}measure\ P\ (Class\ C) & = inheritance\text{-}level\ P\ C \\
subtype\text{-}measure\ P\ (T[]) & = 1 + subtype\text{-}measure\ P\ T \\
subtype\text{-}measure\ P\ _ & = 0
\end{align*}
\]

\[
\text{inheritance\text{-}level\ P\ C} = \\
\quad (\text{if acyclic}\ (P\ P\ _ <^1 _) \land is\text{-}class\ P\ C \land C \neq Object \\
\quad \text{then let}\ [(D, _), _] = clas\ P\ C\ in\ inheritance\text{-}level\ P\ D + 1 \\
\quad \text{else} 0
\]

Hence, subtyping on valid class and array types has no infinitely ascending chain. From this, it follows easily that subtyping on valid types has no such chain either.
**Lemma 6.1.** If \( \text{wf-prog} \ w f - m d \ P \), then \( \text{acc} \ (\text{types} \ P) \ (P \vdash \_ \leq \_). \)

The above tree structure of the subtype relation also allows to extend Jinja’s iterative algorithm for computing lubs on class names to valid types. If both types \( T_1 \) and \( T_2 \) are class or array types, start with \( T \) at \( T_1 \) and keep ascending in the tree until \( T \) is a supertype of \( T_2 \). Then, \( T \) is the lub of \( T_1 \) and \( T_2 \). Otherwise, the lub does not depend on the program and therefore can be hardwired.

Using Klein’s DFA framework, I obtain an executable bytecode verifier to decide the bytecode-specific well-formedness constraints. Hence, \( \text{wf-jvm-prog} \) is executable.

In \( \mu \text{Java} \), Klein [79] bounds the number of array dimensions by 255 as the JMVS does [103] due to syntactic restrictions of the class file format. Thus, the type universe is finite and the ascending chain condition reduces to acyclicity. Moreover, Klein’s bytecode verifier computes sets of types instead of taking the least upper bound. Hence, he does not need to derive an executable lub implementation. The same would be possible for JinjaThreads, because one could compute a bound on the dimensions for every program. However, I conjecture that dealing with sets is less efficient than computing lubs. Moreover, type checking source code needs executable lubs anyway (§6.2.3).

### 6.2.3 Type inference for source code

Source code well-formedness (Figure 2.10) requires well-typedness of the method body. Hence, a well-formedness checker must include a type checker, but even type checking requires type inference. Consider, e.g., the typing judgement for assignment to array cells below, which repeats WT\_\_\_\_ from Figure 2.7.

\[
\begin{align*}
P, E \vdash e_1 :: T[] & \quad P, E \vdash e_2 :: \text{Integer} & \quad P, E \vdash e_3 :: T' & \quad P \vdash T' \leq T \\
P, E \vdash e_1[e_2] ::= e_3 :: \text{Void}
\end{align*}
\]

When the predicate compiler compiles \( \_ \_ \_ \vdash \_ \_ \_ :: \_ \_ \_ \) mode analysis must ensure that all variables are assigned to ground terms when the modes of the assumptions consider them as input. Type checking corresponds to the mode where all parameters are input, type inference assigns output to the expression type. Since \( P, E, e_1, e_2, \) and \( e_3 \) occur in input positions in the conclusion of WT\_\_\_\_ in either case, they pose no problem. However,
the first and third premise require \( T \) and \( T' \) to be ground for type checking. Hence, the predicate compiler can choose to either enumerate all pairs \((T', T)\) for which the subtype relation \( P \vdash T' \leq T \) holds, or infer \( e_1 \)'s and \( e_3 \)'s type and check for \( P \vdash T' \leq T \), or any combination thereof. Note that in case \( e_1[e_2] \coloneqq e_3 \) is type-incorrect, only the second option terminates always, because \textit{Class Object (NT)} has infinitely many subtypes (super- types). To force the predicate compiler choose right, mode annotations allow only mode “everything input” for the subtype relation.

For type inference, the rule for the conditional operator \(_ ? _ : _\) in Java requires to compute the lub of the types of the second and third argument [56, §15.25]. As the declarative definition of the type system uses the declarative lub definition (\( WT_{\text{COND}} \)), type inference (and thus type checking) is not executable. Therefore, I have parametrised \(_ \vdash _ : _\) by the lub operation. The standard version takes the declarative lub predicate from §2.1.2, the one for code extraction the executable lub implementation of the bytecode verifier (§6.2.2). Then, I prove that both versions agree on acyclic class hierarchies rooted at \textit{Object}, but I cannot refine the declarative definition because equality only holds under acyclicity. Since generic well-formedness ensures the precondition, \( wf-J-mdecl \) can use the executable version.

After these preparations, well-formedness for source code no longer poses any difficulties for code extraction. Note that all the setup is transparent to the existing formalisation.

## 6.3 Interpreter and virtual machine

In this section, I describe the obstacles on the way to an executable source code interpreter and an executable virtual machine, and how to overcome them.

### 6.3.1 The single-threaded semantics

**Memory model**

For code extraction, I only use sequential consistency as heap module implementation (§4.2), because the legality constraints of the JMM are axiomatic and thus not executable. As sketched in Figure 2.11, sequential consistency models the heap as a partial function from addresses (natural
6.3. Interpreter and virtual machine

\[
\text{Rparam}_{X} : \quad P, t \vdash \langle \text{Val} \, v.\, M(\text{map} \, \text{Val} \, vs \, @ \text{Throw} \, a \cdot es), s \rangle \\
- (\emptyset) \rightarrow \langle \text{Throw} \, a, s \rangle
\]

\[
\text{Rparam}_{X2} : \quad is-\text{Throws} \, es \\
P, t \vdash \langle \text{Val} \, v.\, M(es), s \rangle - (\emptyset) \rightarrow \langle \text{hd} \, (\text{dropWhile} \, is-\text{Val} \, es), s \rangle
\]

**Figure 6.3:** Original and alternative introduction rule of the small-step semantics

numbers) to objects and arrays. Allocation must find a fresh address, i.e., one not in the heap’s domain. Originally, this was defined via Hilbert’s underspecified (and thus not executable) \( \varepsilon \)-operator as in (6.1). For code extraction, I had to change \textit{new-Addr}'s specification to the least fresh address, replacing \( \varepsilon \) with \text{LEAST}. Then, I proved (6.2) and (6.3) to search for the least fresh address.

\[
\text{new-Addr} \, h \\
= \text{if} (\exists a. \, h \, a = \text{None}) \; \text{then} \lfloor \varepsilon a. \, h \, a = \text{None} \rfloor \; \text{else} \; \text{None} \quad (6.1)
\]

\[
\text{new-Addr} \, h = \text{gen-new-Addr} \, h \, 0 \quad (6.2)
\]

\[
\text{gen-new-Addr} \, h \, n \\
= \text{if} (h \, n = \text{None}) \; \text{then} \lfloor n \rfloor \; \text{else} \; \text{gen-new-Addr} \, h \, (n + 1) \quad (6.3)
\]

**Small-step semantics**

The small-step semantics \( P, t \vdash \langle e, (h, xs) \rangle \rightarrow^{ta} \langle e', (h', xs) \rangle \) is another inductive predicate. The predicate compiler processes 84 of 88 introduction rules automatically. For the others, I must provide alternative introduction rules via program refinement. Figure 6.3 shows the rule \text{Rparam}_{X} for propagating the exception raised while evaluating the parameters, which is representative for the four. This rule violates the desired mode for executing the semantics because its execution would require pattern-matching against the term \( \text{map} \, \text{Val} \, vs \, @ \text{Throw} \, a \cdot es \). The remedy is to declare the alternative introduction rule \text{Rparam}_{X2}: It replaces the problematic term by a fresh variable \( es \) and instead uses the guard \textit{is-Throws} \, es with the following inductive definition.

\[
is-\text{Throws} \, (\text{Throw} \, a \cdot es) \\
is-\text{Throws} \, es \\
is-\text{Throws} \, (\text{Val} \, v \cdot es)
\]

To obtain \text{Throw} \, a as the reduced expression, \text{dropWhile} \, is-\text{Val} removes \( es \)'s prefix of values until the head is the raised exception. To ensure
that the alternative introduction rules replace the definitional ones in a complete manner, I have to prove the elimination rule that corresponds to the new introduction rules, a technically difficult, but conceptionally straightforward task.

Mode annotations for executing the small-step semantics are crucial. The abstraction of the heap module in the locale heap-base adds seven parameters to the small-step semantics in the theory context, which consequently allows a monstrous number of modes.

Virtual machine

The single-threaded VM poses no significant challenges for code extraction, because it is written as a functional program. Unfortunately, the code generator does not yet support set comprehensions over predicates such as read and write. Hence, I adapt the equations for accessing the heap like in Figure 4.5 to directly use their functional implementation produced by the predicate compiler.

6.3.2 Schedulers

Executing the interleaving semantics from Chapter 3 poses three problems:

1. The thread pool and the wait sets are modelled as functions of type \_ \Rightarrow \_ \text{option}. Neither quantifying over these maps’ domains (e.g. to decide whether all threads have terminated) nor picking one of its elements (e.g. to remove an arbitrary thread from a wait set upon notification) are executable.

2. The state space of all possible interleavings is usually too large to be effectively enumerable. Therefore, one wants to pick one typical interleaving.

3. JinjaThreads programs that might not terminate should at least produce a prefix of the observable operations of such an infinite run.

To address the first, I previously [109] proposed to replace these maps with FinFuns, a generalisation of finite maps. Although quantification over the domain then becomes executable, it turned out that choosing
an underspecified element remains unexecutable. I therefore use them only for lock management (§3.1.1). For the thread pool and the wait sets, I instead follow the approach of the Isabelle Collections Framework [92]. In the refined multithreaded state, I replace the concrete functions with modules, i.e., type variables and abstract operations that I specify in two locales. Picking an arbitrary element remains underspecified, but this is now explicit inside the logic, not HOL’s metalogic. Before code extraction, I instantiate the locales with concrete data structure implementations like red-black trees and thus resolve the underspecification.

As to the second problem, I do not use the predicate compiler for _ _ → _ and _ _ → * , as it would produce a depth-first search that enumerates all possible interleavings. The first few interleavings would be such that one thread executes completely (or until it blocks), then the next thread executes completely, etc. Interesting interleavings would occur only very much later – or never at all, if one of the preceding ones did not terminate. Instead, I let a scheduler pick the next thread at each step.

Formally, a scheduler consists of two operations (that I specify abstractly in two locales again): The function schedule takes the scheduler’s state and the refined multithreaded state, and returns either a thread together with its next transition and the updated scheduler state, or None to denote that the execution has finished or is deadlocked. The other function wakeup chooses from a monitor’s wait set the thread to be notified. In terms of these two functions, I define a deterministic, executable step function that updates the multithreaded state just like the non-deterministic interleaving semantics does. To obtain a complete interleaving as a potentially infinite trace, I corecursively unfold this step function. Then, I formally prove that this in fact yields an interleaving.

I have instantiated this specification with two concrete schedulers: a round-robin scheduler and a random scheduler based on a pseudo-random number generator. The most intricate problem is how to obtain (as a function) the thread’s step from the (relational) small-step semantics, once the scheduler has decided which thread to execute. Fortunately, the semantics under SC is deterministic, if one considers only transitions whose preconditions are met by the current state. Thus, I use Bulwahn’s executable version of Russell’s definite descriptor [113].

Corecursive traces also solve the third problem. I instruct the code generator to implement possibly infinite lists lazily. For Haskell, this is the default; for the other target languages, data and program refinement provide an easy setup.
Chapter 6. JinjaThreads as a Java interpreter

1. **datatype** 'm prog = Program 'm cdecl list
2. **definition** prog-impl-invar P' c s f m = (c = Mapping (class (Program P')) ∧ ...)
3. **typedef** 'm prog-impl = { (P', c, s, f, m). prog-impl-invar P' c s f m }
4. **definition** ProgDecl = Program ◦ fst ◦ impl-of
5. **code_datatype** ProgDecl
6. **lemma** [code]: class (ProgDecl P) = lookup (fst (snd (impl-of P)))
7. **definition** tabulate P' = Abs-prog (P', tab-class P', tab-subcls P', ...)
8. **lemma** [code]: Program = ProgDecl ◦ tabulate
9. **fun** compP-impl f (P, c, s, f, m) =
   (let P' = map (compC f) P in (P', tab-class P', s, f, tab-method P'))
10. **definition** compP' f = Abs-prog ◦ compP-impl f ◦ impl-of
11. **lemma** [code]: compP f (ProgDecl P) = ProgDecl (compP' f P)

Figure 6.4: Tabulation for lookup functions and the subclass relation

### 6.3.3 Tabulation

An execution of a JinjaThreads program frequently checks type casts and performs method lookups. However, with the above setup, the semantics recomputes the subtype relation and lookup functions at every type cast and method call from scratch. Now, I show how to leverage program and data refinement to avoid such recomputations with only minimal changes to the formalisation itself. I precompute the subclass relation, field and method lookup (a standard technique for VMs) and store them in mappings – Isabelle’s special-purpose type for maps, see §6.1.1. Figure 6.4 sketches the necessary steps.

Remember that a JinjaThreads program declaration wraps the list of class declarations in a type of its own (l. 1 in Figure 6.4, repeated from Figure 2.3).

First, I define the type 'm prog-impl (l. 3); the morphisms impl-of and Abs-prog translate between the new type and the set of its elements. Apart from the original program declaration (as a list P'), the elements (P', c, s, f, m) consist of mappings from class names to (i) the class declaration (c), (ii) the set of its superclasses (s), and (iii) two mappings for field and method lookup with field and method names as keys (f and m). The invariant prog-impl-invar (l. 2) states that the mappings correctly tabulate
the lookup functions and subclass relation. Then, I define (l. 4) and declare (l. 5) the new constructor \( \text{ProgDecl} :: \text{'}m\text{ prog-impl} \Rightarrow \text{'}m\text{ prog} \) for data refinement, which (in the logic) only extracts the program declaration.

For the lookup functions, the subclass relation, and the associated constants that the predicate compiler has introduced, I next prove code equations that implement them via lookup in the respective mapping – see l. 6 for class declaration lookup. This program refinement suffices to avoid recomputing lookup functions and the subclass relation during execution.

However, the extracted code now expects the input program to come with the correctly precomputed mappings. Thus, I define \( \text{tabulate} \) (l. 7) and auxiliary functions (not shown) that tabulate the lookup functions and subclass relation in these mappings for a given list \( P' \) of class declarations. Finally, I implement the former constructor \( \text{Program} \) in terms of \( \text{tabulate} \) and \( \text{ProgDecl} \) (l. 8).

As the representation of programs has changed, I must also adapt the compiler. The function \( \text{compP-impl} \) is \( \text{compP} \)'s analogon on the representation of type \( \text{'}m\text{ prog-impl} \) (l. 9). First, it compiles all classes and then rebuilds the tabulations for \( \text{class} \) and \( \text{method} \). Since compilation does not change the subclass relation or field lookup, their tabulations need not be recomputed. \( \text{compP}' \) takes \( \text{compP-impl} \) to the new type \( \text{'}m\text{ prog-impl} \) (l. 10). Finally, I derive the new code equation for \( \text{compP} \) (l. 11).

### 6.3.4 Efficiency of the interpreter

Although I cannot expect the generated interpreter to be as efficient as an optimising JVM, to see whether it is suited to run small programs, I have evaluated it on a standard producer-buffer-consumer example; Appendix A contains the code. The producer thread allocates \( n \) objects and enqueues them in the ring buffer, which can store ten elements at the same time. Concurrently, the consumer thread dequeues \( n \) objects from the buffer. Table 6.1 lists the running times of the source code interpreter for different code generator setups. All tests ran on a Pentium DualCore E5300 2.6 GHz with 2GB RAM using MLton 20100608 and Ubuntu GNU/Linux 9.10. The figures are the average of four runs for each setup.

With the adaptations from §6.3.1 and §6.3.2 only, the code is unbearably slow (column 1). For \( n = 100 \), interpreting the program takes 38 min, i.e., 2,320.35 s. As the main bottleneck, I identified the naïve compilation scheme for the small-step semantics. Switching to the improved compilation scheme in the predicate compiler (§6.1.2) speeds up the
Chapter 6. JinjaThreads as a Java interpreter

<table>
<thead>
<tr>
<th>$n$</th>
<th>default setup</th>
<th>with indexing</th>
<th>almost strict</th>
<th>heap as RBT</th>
<th>tabulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>236.51</td>
<td>3.40</td>
<td>.12</td>
<td>.11</td>
<td>.09</td>
</tr>
<tr>
<td>100</td>
<td>2,320.35</td>
<td>31.03</td>
<td>1.71</td>
<td>.99</td>
<td>.86</td>
</tr>
<tr>
<td>1,000</td>
<td>—</td>
<td>—</td>
<td>579.76</td>
<td>9.84</td>
<td>8.67</td>
</tr>
<tr>
<td>10,000</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>91.77</td>
<td>81.73</td>
</tr>
<tr>
<td>100,000</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1,394.93</td>
<td>1,280.62</td>
</tr>
</tbody>
</table>

Table 6.1: Timing (in seconds) for running the producer-consumer example (Appendix A) on $n$ objects for different adjustments to the source code interpreter; — denotes timeout after 1 h

interpreter by two orders of magnitude (column 2). The definite descriptor $\iota$ that extracts the result configuration from the enumerations, strictly evaluates all branches. Hence, explicit laziness in the generated code is unnecessary. If I remove the most obvious constructions that enforce laziness from the code equations that were compiled under the improved scheme, a program run with $n = 100$ takes only 1.71 s (column 3).

As $n$ increases, another bottleneck shows up: memory allocation (cf. §6.3.1). Since the heap is modelled as a function and writes as function updates, i.e., closures, finding the next fresh address takes time quadratic in the number of previous allocations. Thus, interpreting the example program is quadratic in $n$ although the program itself only requires linearly many steps. To speed up allocation and read access, I implemented the heap module (§4.1.1) for sequential consistency a second time with efficient data structures from the ICF: The heap is now a red-black tree (RBT) with addresses as keys, the array cell list becomes an RBT with indices as keys, and tries implement the field tables. Combined with the other improvements, this already provides a decent interpreter (column 4): Run times grow linearly in $n$ as expected. $n = 100,000$ is an outlier because memory consumption exceeds the two 2 GB of physical memory and garbage collection of the ML runtime environment runs frequently.

Finally, I also added tabulation (§6.3.3), where the mappings are for simplicity implemented as associative lists. Surprisingly, the speed-up (less than 15%) is modest. The reason might be the tiny class hierarchy of the example program for which lookup functions terminate quickly.
6.3. Interpreter and virtual machine

<table>
<thead>
<tr>
<th>$n$</th>
<th>default setup</th>
<th>heap as RBT</th>
<th>tabulation</th>
<th>caching w/o tabulation</th>
<th>tabulation &amp; caching</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.00</td>
<td>.03</td>
<td>.01</td>
<td>.02</td>
<td>.01</td>
</tr>
<tr>
<td>100</td>
<td>1.81</td>
<td>.11</td>
<td>.07</td>
<td>.07</td>
<td>.06</td>
</tr>
<tr>
<td>1,000</td>
<td>23.68</td>
<td>1.07</td>
<td>.70</td>
<td>.61</td>
<td>.62</td>
</tr>
<tr>
<td>10,000</td>
<td>—</td>
<td>10.97</td>
<td>7.07</td>
<td>6.21</td>
<td>6.29</td>
</tr>
<tr>
<td>100,000</td>
<td>—</td>
<td>109.90</td>
<td>71.12</td>
<td>62.36</td>
<td>63.05</td>
</tr>
</tbody>
</table>

Table 6.2: Timing (in seconds) for running the producer-consumer example (Appendix A) on $n$ objects for different setups of the virtual machine; — denotes timeout after 1 h.

I also ran the tests with the code generated in Haskell (compiled with Glasgow Haskell Compiler 6.10.4) and OCaml (compiled to native code with OCaml 3.11.1). The Haskell code is about 50% slower than ML and the OCaml code takes between 2 and 5 times as much time as MLton. Nevertheless, the different adjustments to the interpreter affect the run times similarly to ML.

I have also compiled the example program with JinjaThreads’ compiler and run it in the virtual machine. Table 6.2 shows the timings for the VM – again with different setups. Although the VM is much faster than the source code interpreter, the VM is still relatively slow under the default setup (column 1). Like for source code, switching to red-black trees helps (column 2). Tabulation (column 3) has a much greater effect (35% speed-up) on the VM, because the VM performs method lookup at every step to retrieve the next instruction to execute. Therefore, I implemented a second VM that caches the instruction list and exception table in the call frames. With the heap as RBT, this gains another 12% in performance over tabulation (column 4). Surprisingly, combining tabulation and caching slightly slows the VM down (column 5). Since the performance loss is not a constant, it is unlikely that building the lookup tables is the reason. Rather, I suspect that the more complicated type representation for JinjaThreads programs impairs MLton’s optimiser, since the same effect is much stronger (20% slow-down) under PolyML 5.4.1, which optimises less aggressively.

Although the VM run-times are already decent (about one minute for 100,000 objects), the interpreter and VM are still far from a commercial VM: The Java HotSpot VM takes only 150 ms for 100,000 objects.
In [104], Lui and Moore test their JVM formalisation M6 in ACL2 on a simple parallel factorial algorithm. To compare my interpreter with theirs, I have converted the Java program to JinjaThreads with the Java2Jinja tool (§6.5). For computing 10! with five threads in parallel, the source code semantics takes 26.7 s and the VM just 0.2 s. The M6 takes 6.2 s when run in the ACL2 interpreter, version 2.7 with GNU CLISP 2.42. Hence, JinjaThreads and the M6 have comparable performance.

6.4 Guidelines for executable formalisations

From Bulwahn’s and my experience with JinjaThreads, we have distilled the following guidelines to easily obtain executable formalisations in Isabelle/HOL.

Avoid Hilbert’s $\varepsilon$-operator! Hilbert’s choice cannot express under-specification adequately as, in HOL’s model, its interpretation is fully specified. Partial correctness of the code generator guarantees that all evaluations in the functional language hold in every model. Thus, one cannot replace it by any implementing function that chooses one suitable value consistently and fixes the underspecified function to one concrete model. Instead, use one of the following options:

1. Change the definition to make the choice deterministic and implementable, e.g., always pick the least element.

2. Use locales for intra-logical underspecification and instantiate the choice operator to a concrete implementation by locale interpretation.

3. Switch to a relational description and prove the correctness for all values.

The first is least intrusive to the formalisation, but requires changes to the original specification. To execute the deterministic choice, one needs to run the predicate compiler on the choice property and use Bulwahn’s executable definite descriptor for predicates [113], or implement a suitable search algorithm via program refinement. I use the former in the scheduler (§6.3.2) and the latter for memory allocation (§6.3.1).
6.4. Guidelines for executable formalisations

The second is the most flexible option, but also tedious, as the locale does not automatically setup proof automation and lacks true polymorphism. I use this approach, e.g., to specify schedulers (§6.3.2). Care must be taken in combination with data refinement via the code generator, as the choice must not depend on the additional structure that the interpretation introduces.

The last option completely avoids underspecification, but relinquishes the functional implementation. For code extraction, one should either (i) apply the predicate compiler to obtain code that computes all possible implementations for the specification, or (ii) provide a functional implementation and show correctness (§6.3.2). For the latter, one must typically replace the involved types with others that have additional structure.

**Structure locales wisely!** Modular specifications, i.e., locales, and code extraction do not (yet) go well together (see §6.1.4). To combine them, one best adheres to the following discipline: One locale \texttt{Sig} fixes the parameters’ signatures and contains all definitions that depend on the parameters. Another locale \texttt{Spec} extends \texttt{Sig} and states the assumptions about the parameters; all proofs that depend on the assumptions go into \texttt{Spec}. For functions and inductive predicates of \texttt{Sig}, one feeds the equational theorems and introduction rules exported into the theory context to the code generator or predicate compiler, respectively. To obtain the (correctness) theorems, instantiate \texttt{Spec} and prove the assumptions. I have used this approach throughout JinjaThreads.

**Annotate predicates with modes!** Mode annotations for predicates instruct the predicate compiler to generate only modes of interest, not all modes that its mode analysis can infer. They provide three benefits. First, if the predicate has many parameters, analysing all modes can quickly become computationally intractable (cf. §6.3.1) – in this case, they are necessary. Second, they ease maintenance and debugging as they fail immediately after adjustments: If changes in the development disable a mode of interest, an error message indicates which clauses are to blame. Without annotations, the missing mode might remain undiscovered until much later, which then complicates correcting errors. Third, some not annotated, but inferable modes might lead to generation of slow or non-terminating functions. By disallowing them, the predicate compiler
cannot accidentally pick one of them when it compiles a subsequent predicate.

## 6.5 The translator Java2Jinja

Formalisations in the size of JinjaThreads require better validation than manually scrutinising the definitions and comparing them to the language specification. One way of validating the semantics is to execute test programs and check for the expected result. However, JinjaThreads’ syntax is different from and only a subset of Java’s.

Hence, to make the vast supply of Java programs available, Jonas Thedering, Antonio Zea, and I have developed the conversion tool Java2Jinja\(^56\) as a plugin to the Eclipse IDE with the following modes of operation. First, it translates Java class hierarchies into JinjaThreads abstract syntax, emulating many unsupported Java features. Second, it provides a frontend to the well-formedness checker (§6.2), interpreter (§6.3), and compiler (§5.5) that I have extracted from the formalisation using Isabelle’s code generator. Third, it can automatically run full test suites where command-line arguments and the expected output are provided as Java annotations in the programs.

Being implemented in Java and on top of the Eclipse Java Development Tools (JDT), the translator itself defies formal verification. The possibility for introducing errors is negligible for the subset of Java that JinjaThreads directly models, because the translations are minuscule. However, even small Java programs require some parts of the Java standard API which exceed this subset, e.g., class `String`. Therefore, Java2Jinja emulates these features with non-trivial, unverified transformations (§6.5.1), which possibly introduce errors. I have extensively tested Java2Jinja (§6.5.2), but this naturally cannot prove their absence. These threats could only be avoided if JinjaThreads was extended to support the missing features, which are mostly of sequential nature. However, this is beyond the scope of this work.

Figure 6.5 shows Java2Jinja’s user interface. When the user invokes the conversion tool, e.g., using the new button Java2Jinja in the tool bar, which the mouse cursor points to, she can choose which main method to execute and specify parameters as needed. Then, the translated

\(^{56}\)http://pp.info.uni-karlsruhe.de/projects/quis-custodiet/Java2Jinja/
program can be checked for well-formedness or executed in the extracted interpreter or VM – Java2Jinja prints the results in the console window (at the bottom).

Note that well-formedness checking is a sanity check for the translation being correct, not syntactic correctness of the Java source code. The JDT already catch such compiler errors. Nevertheless, JinjaThreads considers some well-formed Java programs to be ill-formed, because its definite assignment analysis (taken from Jinja) is less precise than the one that the JLS specifies [56, §16]. For example [56, §16], the following program passes Java’s definite assignment test, but not JinjaThreads’, because the latter’s analysis does not infer that the access to
k in the method call occurs only if both conditions are evaluated, i.e., k is initialised.

```java
int k; if (v > 0 && (k = ...) >= 0) { System.out.println(k); }
```

### 6.5.1 The translation

Although JinjaThreads syntax already covers a substantial part of Java, lots of syntactic sugar and some imperative control structures are omitted. Some of them could be easily added to the semantics, bloating the model and especially proofs. Others would require substantial re-engineering. Hence, Java2Jinja tries to emulate these as good as possible. In the following, I briefly describe the translations.

**Operators**  JinjaThreads misses the operator for string concatenation [56, §15.18.1], unary ones [56, §15.15], postfix ones [56, §15.14], and compound assignment ones [56, §15.26.2]. Java2Jinja replaces string concatenation with `StringBuilder` as allowed by the JLS [56, §15.18.1.2]. Unary operators are emulated via their binary counterparts, e.g., `-x` becomes `Val (Intg 0) «–» (Var x)`. Postfix operators are emulated using a fresh local variable that remembers the former value. For example, `y = x++;` becomes `y := {~tmpVar_0: Integer = None; ~tmpVar_0 := x;; x := Var ~tmpVar_0 «+» Val (Intg 1);} ; Var ~tmpVar_0}. Here, the translation exploits that Java identifiers never contain ~ and that local variable blocks in JinjaThreads may return a value. Compound assignments are desugared and fresh local variables store the computed parts of the left-hand side (e.g., `f()` in `a[ f() ] += x;` ) such that they are evaluated only once.

**Primitive types**  The primitive types `byte`, `short`, `char`, and `int` are all mapped to `Integer`. This is sound since all Java operators automatically promote these to `int`. There is no support for `long` or floating point values. Auto-(un)boxing of supported primitive types [56, §5.1.7, §5.1.8] is replaced by method calls of the appropriate classes, e.g., `Integer i = 0;` is translated like `Integer i = Integer.valueOf(0);`. Conversions between primitive types are either no-ops (widening [56, §5.1.2]) or extract the required bits (narrowing [56, §5.1.3]). For example, `(char) x` becomes `x & 65535`. 
Control flow structures  JinjaThreads only supports conditionals, while loops, and exceptions. Java’s other statements for locally transferring control (switch, do-while loops, and for loops) are implemented with conditionals and while loops. Non-local transfer of control (break, continue, return) are implemented by throwing an exception and catching it again at the target. For finally clauses, Java2Jinja follows the JVMS’ recommendation for compilation to bytecode [103, §7.13]. They are duplicated and inserted at the end of the try block and in an exception handler that catches all raised exceptions.

Methods  Java2Jinja resolves overloaded methods and appends the method descriptor to the name [103, §4.3.3]. Lack of visibility constraints affects method resolution in that private methods should be resolved statically. Since JinjaThreads knows only dynamic dispatch, private methods are renamed uniquely such that dynamic and static dispatch agree. The same applies to invocations with super – if necessary, method signatures are duplicated and the dynamically dispatched version delegates to the statically dispatched. For example, in

```java
class A { void g() { h(); } private void h() { } }
class B extends A { void g() { super.g(); } void h() { } }
```

g in A must call h of A and super.g() must disable dynamic dispatch for g and call A’s method. Hence, h in A is renamed to A~h()V and there are two copies of A’s method g: (i) A~g()V is called by super.g();, its body calls A~h()V. (ii) g()V delegates to A~g()V, B overrides it.

Abstract methods throw an UnsupportedOperationException. Native methods are translated to native JinjaThreads methods. If JinjaThreads does not provide an implementation (§2.1.3 and §3.2.1), the program is ill-formed and execution gets stuck when such a method is called.

Constructors and initialisation  Like in Java bytecode, JinjaThreads allows objects to be allocated without executing their constructor. Java2Jinja inserts the constructor calls and default constructors as necessary. Instance initialisers [56, §8.6] and field initialisations [56, §8.3.2] are added to the constructors after the call of the super-constructor in the order they appear in the program text.
Static members and compile-time constants  Since JinjaThreads supports only non-static members, Java2Jinja introduces a singleton class that collects all static fields, methods and initialisers. Their names are prefixed by the fully qualified name of the declaring class to avoid ambiguities. Moreover, every object and array stores a reference to the singleton object in a field through which threads access these members. For simplicity, all static initialisers are executed before the main method of the program, which violates the official class initialisation procedure [56, §12.4]. Following the latter would require further substantial transformations of the program.

Nested classes  Java2Jinja provides limited support for nested classes in that it replaces them by ordinary top-level classes that store a reference to the enclosing instance if necessary. However, accessing fields and local variables of enclosing classes and methods is still experimental.

Generics  Type erasure removes all generics. Reflective type tokens such as \(<T> void f(Class<T> c)\) and multiple bounds are not supported.

Interfaces  Although interfaces are widely used in Java programs, they cannot be simulated in JinjaThreads and are therefore unsupported. In the special case where a class \(C\) inherits from \(Object\) and a single interface \(I\), Java2Jinja turns \(I\) into a class and changes the class hierarchy such that \(I\) extends \(Object\) and \(C\) extends \(I\). Since interfaces are another common source of ill-formedness of converted programs, Java2Jinja lists all problematic classes and interfaces during the translation as a warning message.

Standard API  Every Java program uses Java’s standard API, e.g., the \(\text{main}\) method takes an array of \(\text{Strings}\). To avoid that unnecessary parts are sucked in, Java2Jinja approximates the classes and their members that are actually needed and only translates these. Otherwise, irrelevant code would bloat the JinjaThreads code or possibly inhibit correct translation, because it might include unsupported Java code, e.g., heavy use of interfaces. It would also deteriorate performance, since (i) the well-formedness checker has to analyse more code, and (ii) initialisations before the \(\text{main}\) method starts take longer, because they allocate all string
6.5. The translator Java2Jinja

literals. The heuristics is sound except for initialisations. For example, Java2Jinja removes the field i from

```java
class A { int i = 1 / 0;
    public static void main(String[] args) { new A(); } } 
```

because it is not used. Hence, the translated program terminates normally whereas the Java program fails with an `ArithmeticException`.

Further, JinjaThreads also provides a customized version of the standard library classes derived from OpenJDK 6 [139]. Some native methods like `System.arraycopy` are implemented manually such that they are available in translated programs. Moreover, we have removed Unicode processing from `String` and `Character` and replaced the affected methods with those from CLDC [41] with ASCII support only. Unicode processing requires large character data tables that are initialised lazily through class initialisation in Java. As Java2Jinja initialises all classes before the start of the program, the translated program would eagerly initialise all of them, which consumes prohibitive amounts of memory and time.

6.5.2 Validation

I have tested the semantics and translation thoroughly by executing numerous test cases from different sources. Although most bugs were caused by the translation, I also found one in the semantics: For division and remainder, JinjaThreads used the default implementation that Isabelle’s Word library provides, but the JLS specifies them differently for negative divisors. Since both source code and bytecode use the very same operation, the compiler verification was not able to catch this bug.

In detail, I have used the following test cases:

- 59 hand-written programs for regression testing.

- Jacks [74], a suite of test cases designed to identify bugs in Java compilers. These tests identified only bugs in the translation. I use 136 cases out of 149 from the runtime section, the 13 remaining ones deal with binary compatibility, which does not apply to JinjaThreads. For 30 out of the 136, the translation fails or is incorrect. In detail, 11 cases inherently rely on unsupported features like native methods, interfaces, and floats; initialisation issues cause 4 failures; 13 cases exhibit bugs in the translation,
in particular for nested classes. The remaining 2 cases require behaviour that the interpreter does not produce, e.g., it crashes instead of raising an `OutOfMemory` exception.

- Tests for the `java.lang.*` classes from OpenJDK 6 [139]. Since they test OpenJDK’s implementation of the API, many rely on native methods or other Java features like Unicode that JinjaThreads does not implement. Hence, I found only 24 out of 184 test cases (13%) that Java2Jinja can correctly translate to JinjaThreads. One of these also exhibited the division and remainder bug mentioned above. Using JinjaThreads’ compiler to bytecode and its VM for execution, all but four test cases terminated within minutes. These four are randomised test cases with a large number of iterations and use `System.arraycopy` frequently, which is particularly slow in JinjaThreads.

I have also looked at OpenJDK tests for other packages. However, these typically use IO or the Java collections framework. JinjaThreads does not model the former and the latter heavily uses interfaces. Therefore, they cannot be used.

Validation by executing translated test programs checks the translator and the semantics at the same time. Since the translation itself is very complex, failed test cases must be examined manually to see whether the translation or the semantics is buggy. In all but one case, it was the translation’s fault. In principle, it is conceivable that bugs in the translation hide errors of the semantics. Since Java2Jinja directly translates the Java subset that JinjaThreads models, this possibility is negligible.

Therefore, I conclude that validation has been successful for the subset under examination. The numerous failures indicate that the model should cover more features such that less error-prone emulations are necessary (see §7.4 for more details). Although validation has tested the sequential semantics thoroughly, execution explores only a single schedule for a multithreaded program. Hence, this approach cannot prove that the semantics covers all allowed behaviours. For example, in §4.3.6, I have already mentioned two cases for which JinjaThreads does not model the full behaviour. Moreover, validation has not covered the JMM at all, because it is not executable. In conclusion, this is only a first step towards rigorous validation.
6.6 Related Work

Most closely related to JinjaThreads’ well-formedness checker and compiler is Berghofer’s and Strecker’s work [24] on extracting a well-formedness checker and compiler for JinjaThreads’ predecessor Bali. The former suffers from the very same termination problems that I have presented in §6.2, the latter is similar to JinjaThreads compiler except that it does not support exceptions. For an in-depth comparison, see §7.3.

Some formalisations of Java and the JVM in theorem provers are directly executable. First, the M6 by Lui and Moore [104, 105] models the JVM in ACL2 covering the CLDC specification; see §3.4.1 for details. Efficient execution was a major goal in its design. Thus, it is remarkable that JinjaThreads achieves similar performance although it focuses on simplifying proofs. Moore’s and Porter’s jvm2ac12 tool [122] translates Java bytecode into M6’s representation. Since the M6 models Java bytecode quite closely to the JVMS, jvm2ac12 in fact translates only concrete to abstract syntax. JinjaThreads is far from such a point.

Second, Farzan et al. [48, 47] report on a formal semantics for Java source code and a virtual machine for Java bytecode in Maude. Maude’s rewriting logic allows to execute programs in abstract syntax symbolically and analyse them using model-checking techniques. Their source code semantics covers about the same subset as JinjaThreads except for exceptions. For efficiency reasons, their semantics is based on continuations. This way, they bypass a major source of inefficiency from which JinjaThreads’ source code interpreter suffers. For example, recursive methods enclose the statement to execute in more and more blocks for this and the parameters, and execution has to traverse them at every step of execution. Hence, executing

```java
void recurse(int d) { if (d > 0) recurse(d - 1); }
```

in source code is quadratic in d. Continuation-based semantics keeps the next statement to execute at the outer-most level. Note that explicit call stacks in JinjaThreads bytecode also avoid the problem. Their VM follows a traditional style, but separates deterministic sequential instructions from non-deterministic multithreading to boost performance. They also implemented JavaFAN as a front-end to the semantics that takes Java source code or bytecode and converts it into abstract syntax. It is unclear whether JavaFAN tries to emulate missing features like Java2Jinja does. Moreover, they do not report on any attempt to validate their semantics.
or the translation. In an effort to cross-validate deduction rules of the Key system, Sasse [155] found and corrected several bugs in their source code semantics.

Third, Atkey [9] presents an executable JVM model in Coq. He concentrates on encoding defensive checks as dependent types, but does not provide any data about the efficiency.

The CCCP tool by Wasserrab [175] is similar to Java2Jinja. It translates C programs to CoreC++ [178], another descendant of Jinja, and executes them in the source code semantics. He also uses the tool for validating CoreC++.

Batty et al. [17] have implemented Cppmem to explore their formalisation of the C++ memory model in Isabelle/HOL. It consists of two parts. First, a (manually written) symbolic execution engine generates sets of potential executions for a given (small) C++ program. Second, a checker that Isabelle’s code generator has extracted from the formalisation prunes this set to the executions that the C++ memory model actually allows. The interpreter and VM that I have extracted implement sequential consistency; the axiomatic definition of the Java memory model is beyond the capabilities of Isabelle’s code generator.

Code extraction in general is a well-established business, in particular extraction of functional implementations. In [116], Marić presents a formally verified implementation of a SAT solver in Isabelle/HOL. In the CeTA project, Thiemann and Sternagel [170] generate a self-contained executable termination checker for term rewrite systems. The Flyspeck project uses code extraction to compute the set of tame graphs [18, 127]. All these formalisations were developed with executability in mind. Complications in proofs to obtain an efficiently executable implementation were willingly taken and handling them are large contributions of these projects.

Code extraction in Coq [102] has been used in various developments, notably the CompCert compiler [99] and the certificate checkers in the MOBIUS project [13]. Like in Isabelle, functional specifications pose no intrinsic problems. Although code extraction is in principle possible for any Coq specification, mathematical theories can lead to “a nightmare in term of extracted code efficiency and readability” [102]. Hence, Coq’s users, too, are facing the problem of how to extract (roughly) efficient code from specifications not aimed towards executability. ACL2 and PVS translate only functional implementations to Common Lisp.
Discussion and Future Work

After hundreds of pages filled with formal definitions and technical details, this chapter takes a step back. Now, I contrast the efforts with the rewards (§7.1), comment on my experience as an Isabelle user (§7.2) and on how the advances in Isabelle influenced the formalisation (§7.3), compare JinjaThreads with Java (§7.4), and identify directions for future work (§7.5).

7.1 Efforts and rewards of a machine-checked formalisation

Efforts Building a formal model of a programming language in a theorem prover is a major investment. I have been working on JinjaThreads for five years, but not full-time. Not counting developing foundational libraries for Isabelle [108, 110], I estimate the net amount of time at three person years. It is difficult to exactly break down the figure to the individual parts, since I have worked on some of them in parallel. I had the first working version of the interleaving semantics (as described in [107] without thread interruption) and the type safety proof for source code after about four months. Extending it to bytecode took less than one month. The abstract heap module and the JMM formalisation with the proofs required about one year. The compiler was most laborious with more than one year. Since I worked on code extraction occasionally during the whole development, I am not able to give a figure for that. The remaining time has gone into refactoring the formalisation and extending the JinjaThreads language.
Table 7.1 lists some figures that measure JinjaThreads’ size: the number of definitions (including (co-)recursive and (co-)inductive ones), lemmata (including theorems and corollaries), locales, and the lines of Isabelle code. It breaks down the total figures into the various parts of JinjaThreads (cf. Figure 1.4). For comparison, the last column shows the lines of code (LoC) of the respective part of Jinja.

Since JinjaThreads covers a larger subset of Java than Jinja, it is, of course, larger. Source code and bytecode have roughly doubled in size to account for concurrency and arrays. Shared infrastructure has even tripled, but one third stems from native methods (1318 LoC) and binary operators (585 LoC). Here, it can be seen that reusing the interleaving semantics in source code and bytecode pays off, although I suppose that two monolithic semantics would have required less than twice the 5 kLoC of the framework. The benefit of reuse becomes much clearer for the JMM and the proofs about it. 10 kLoC out of the 13 kLoC are needed to formalise the JMM and the language-independent parts of the DRF guarantee and consistency. Only 3 kLoC fall upon discharging the final assumptions of the locales about the single-threaded semantics – and these 3 kLoC distribute roughly equally over native methods, source
code, and bytecode. This shows that my design separates concurrency from single-threaded issues well.

The compiler verification offers a different picture – the transition from big-step to small-step semantics caused its size to explode. In detail, the bisimulation framework and lifting proofs (§5.1) require 4,237 LoC. Explicit call stacks in source code add another 2,068 LoC. However, the majority (18,231 LoC) is spent on the bisimulation relations and proofs for the individual compiler stages. In code generation, in particular, I have to deal with each inductive case manually. Specialised proof procedures might have been sensible here.

Code extraction refers to the extensions that were necessary to obtain the executable interpreter, VM, and compiler. In particular, it subsumes the formalisation of schedulers (§6.3.2). It is one order of magnitude larger than in Jinja, but offers another level of quality in two respects. First, Jinja used manual translations for unexecutable definitions like Hilbert’s choice, whereas JinjaThreads does without. Second, Jinja does not care about efficiency, whereas I have demonstrated that the JinjaThreads VM is usable in practice.

On a Pentium DualCore 2.50 GHz running Ubuntu 10.04, Isabelle in single-threaded mode takes 1:55 h and 17 GB of memory to process the JinjaThreads sources, recheck all proofs, and extract the executable parts.

**Rewards** The investment in the formalisation starts to pay off fairly soon in various ways.

First, we can be sure that the proofs are correct with respect to the formal definitions, and that the formal definitions are at least type-checked. In particular, a series of false claims and their subsequent disproof – as has happened for the JMM [115, 38, 162, 171] – is impossible. Type-checking alone supports the formalisation tremendously, because obvious mistakes are caught early on. In addition, Isabelle provides some tools for finding bugs automatically (see §7.2 for details).

Furthermore, machine-support is fundamental for extending and adapting such a formal model, since proofs can be replayed automatically. JinjaThreads has evolved from Jinja gradually, adding feature after feature. In such a step, Isabelle determines which proofs need to be extended or changed and – most importantly – which ones I need not reconsider. Extending a pen-and-paper model requires to revisit all theorems and
proofs, which is tedious and error-prone, because subtle interactions may be missed easily.

The ability to replay proofs was equally essential during the development of a single feature. After I had corrected a typo or conceptual error in a definition or theorem, Isabelle automatically rechecks the theorems that had already been adapted to the new feature. Often enough, such typos have erroneously rendered some proof cases trivial that in fact need some argument. A typical example is to inadvertently give two (different) variables the same name. The following typing rule for local variable blocks illustrates the problem:

\[
\begin{align*}
\text{is-type } & P \; T, \; P, E(V \mapsto T) \vdash e :: T' \\
\text{case } & vo \text{ of None } \Rightarrow \text{True } | [v] \Rightarrow \exists T''. \; \text{typeof } v = [T''] \land P \vdash T'' \leq T \\
\hline
P, E \vdash \{V: T = vo; e\} :: T'
\end{align*}
\]

Following the format and naming conventions of the other rules (see Appendix B.6.2), one is tempted to write \( T \) instead of \( T' \) in the shaded positions, but \( T \) already denotes the type of the declared variable. Obviously, the corresponding inductive case of the theorem

\[
\begin{align*}
[P, E \vdash e :: T; \; \text{ran } E \subseteq \text{types } P] \Rightarrow \text{is-type } P \; T
\end{align*}
\]

becomes trivial with the typo.

In conclusion, mechanisation

(i) supports the process of formalising,
(ii) guarantees the absence of errors, and
(iii) eases reuse in other projects, because definitions can be easily adapted when necessary and Isabelle checks that everything fits together properly.

7.2 Experience: Working with Isabelle/HOL

In this section, I comment on my experience of formalising JinjaThreads with Isabelle/HOL and which features were particularly useful and which would have been desirable.
7.2. Experience: Working with Isabelle/HOL

Essential Isabelle features

Essential Isabelle features for JinjaThreads are Isar, locales, and the high degree of built-in proof automation, on which I now comment in turn.

Almost all proofs in JinjaThreads are written in Isabelle’s declarative proof language Isar [179, 25], see Figure 7.1 for an example. Proofs in Isar style tend to be a bit longer than imperative proof scripts and usually take more time to write, but the extra effort is not spent in vain. Since Isar follows the mathematical language of informal reasoning, Isar proofs are intelligible to non-experts. Being declarative, they typically make the line of argument explicit. Moreover, Isar proofs are more robust to adaptations than imperative proof scripts or hand-coded proof procedures, because when the proofs are replayed, (i) they break exactly where manual intervention is necessary, and (ii) being declarative, they record the exact intermediate statement, i.e., one need not guess what the proof state should have been. Therefore, Isar proofs are easier to maintain, which is a major concern in large formalisations like JinjaThreads. Such adaptations of existing proofs are triggered by new Isabelle releases (seven of which happened during the development of JinjaThreads) and changes to the formalisation itself.

Locales have proven essential in structuring the formalisation, as can be seen in Chapters 3, 4, and 5. Unfortunately, locale contexts are purely dynamic, i.e., whenever one switches from one locale to another, Isabelle discards the old and constructs the new from scratch. Since this involves to replay each previous declaration of the new context including all inherited locales, entering a locale context can take a few minutes. In fact, when Isabelle processes JinjaThreads, the peaks in memory consumption occur when it rebuilds such locale contexts.

Thanks to Isabelle’s sophisticated procedures for proof automation, JinjaThreads does not need custom tactics, although theorem-specific tactics might drastically reduce some proof sizes, as Chlipala demonstrates in a case study in Coq [39]. Instead, I have focused on properly setting up proof automation, where simplification is the main workhorse. Unfortunately, Isar provides only little support for simplification steps, but favors natural deduction. Consequently, predicates should come with introduction and elimination rules to be used in Isar proofs and simplification rules for good automation. However, Isabelle’s means of specification used to provide either the former ((co)inductive) or the
latter (**fun**). Upon my suggestion, Bulwahn extended the **inductive** package such that one can a posteriori derive rewrite rules for (co-)inductive predicates and sets. Since this feature has been available, I have defined most predicates and sets (co-)inductively and had Isabelle automatically derive simplification rules.

Sledgehammer [29, 33] runs automated theorem provers to discover a proof of a theorem. It is particularly useful for proof obligations about notions from libraries that the formalisation builds on, e.g., 32-bit integers and coinductive natural numbers. In such a case, Sledgehammer only needs to combine the right bunch of lemmata that the library provides. Since it identifies the relevant ones, this saves the user from the time-consuming task of becoming an expert for each library she uses. It is less effective for solving goals about what one is currently developing, though, because the lemma base is still largely incomplete, and the new lemmata are rarely simple consequences of others.

**Counterexample generation**

To support the user in exploratory stages, Isabelle provides two counterexample generators Quickcheck [23] and Nitpick [28, 30]. When the user enters a theorem she wants to prove, they automatically try to refute it and present a counterexample, if they succeed.

Quickcheck converts the putative statement into ML code via Isabelle’s code generator, installs test data generators for the free variables, and executes the code. If the code returns *False*, the assignment to the free variables constitutes a counterexample. Quickcheck supported the development of JinjaThreads only little because it suffers from two drawbacks. First, the code generator must be able to process all definitions that the statement refers to. This restriction is severe, because early on, when the definitions still keep changing, most errors are to be found, but time is not spent on deriving executable equations for all constants unless one expects them to be in the final form. Moreover, proof invariants such as conformance involve universal or existential quantifiers, which cannot be executed. Second, until recently, it used to generate the test data randomly, which often misses the subtle cases. Now, Bulwahn changed Quickcheck to use exhaustive search [29, 36]. On the positive side, when it is applicable, Quickcheck is extremely fast. Even statements that involve complicated definitions are tested within seconds. In JinjaThreads, Quickcheck has spotted typos in definitions and statements, but none
of the conceptual bugs in definitions; exhaustive testing might have changed the latter.

Nitpick follows a different approach. It converts the statement and all definitions and axioms it depends on into first-order relational logic (FORL) and calls a relational model finder. This way, it handles non-executable constructs and underspecification without further ado, but is limited by the complexity of the generated formula. Moreover, its performance is sensitive to the FORL encoding. Although sensible defaults are provided, Blanchette et al. [31] showed that manually tuning Nitpick’s numerous parameters can boost performance.

In JinjaThreads, I applied Nitpick successfully for the bisimulations from §5.1, where it found the counterexample in Figure 5.4 after several attempts of mine to prove the equivalence of delay bisimulation with explicit divergence and well-founded delay bisimulations. However, Nitpick’s output overwhelmed the model finder for any theorem that involves program declarations or the semantics. Conversely, Nitpick helped a lot in developing the library of coinductive lists [110], because it is able to find unintuitive counterexamples with infinite lists, which are out of Quickcheck’s reach.

In conclusion, both Quickcheck and Nitpick work well for smallish developments, but they are of little help to identify intricate cases where things do not fit together in formalisations as large as JinjaThreads.

Suggestions for improvement

Despite the good support for developing formalisations, I found the following suggestions for further improvement.

Isabelle’s support for coinductive datatypes, corecursive definitions and proofs by coinduction is poor. Coinductive datatypes still must be constructed manually like Paulson did 15 years ago [142]. The proof method for coinductive proofs lacks many of the conveniences of its inductive counterpart. Consequently, coinductive Isar proofs typically contain four steps of boilerplate code (the highlighted parts of the proof57 in Figure 7.1). First, auxiliary variables for instantiated coinduction parameters are introduced (l. 3). Second, all assumptions are combined using HOL conjunction (instead of meta-conjunction that

57\( tmap \ f \ g \ xs \) is the map operator for terminated coinductive lists. It applies \( f \) to all elements of \( xs \) and \( g \) to the symbol of the \( [] \) constructor.
Isar proofs favor) and generalised using existential quantifiers (l. 4). Third, the goal is restated using the auxiliary variables (l. 5). After the subsequent application of the coinduction rule, every coinductive case unpacks the existential quantifiers and separates the assumptions again (l. 8). Thus, a five-line proof is bloated with six lines of boilerplate code.

Another neglected issue is support for refactorings, e.g., (i) moving and renaming definitions and theorems, (ii) promoting subproofs to lemmata, (iii) changing the order of parameters or pretty-printing syntax for a function or predicate, and (iv) (un-)currying functions and predicates. While simple renamings can be achieved relatively easily with standard command-line tools and their support of regular expressions, changing the order of parameters is particularly tricky, because Isar proofs often refer to variables in theorems and proofs by position. In total, I have spent at least two months on such simple refactorings for the most pressing issues, but much remains to be done in this respect. In particular, JinjaThreads in its current state lacks consistent naming of constants, variables and theorems and much of the pretty syntax in this thesis. Fortunately, Ruegenberg [151] has recently implemented the Levity prototype for renaming definitions and theorems and moving the latter. I hope that this work continues and am looking forward to when such tool support is available in Isabelle’s IDE.
7.3 From $\text{Java}^{\text{light}}$ to JinjaThreads

JinjaThreads is currently the result of 15 years of formalising Java in Isabelle/HOL, see §1.2 for the historic view. In this section, I look back to see what has made it into JinjaThreads and how new features of Isabelle offer new formalisation options.

Reuse is most obvious in Jinja’s declaration infrastructure, lookup functions and generic well-formedness, which can be traced back to early versions of Bali [129]. Although the definitions have changed textually over time, JinjaThreads still benefits from the good choice of proven lemmata and setup for proof automation. Similarly, the sequential VM follows the well-engineered structure from $\mu$Java and Jinja. JinjaThreads’s compiler inherits Jinja’s and merely extends it to the new statements for arrays and synchronisation, and the same applies to the proof of type-preservation. However, the proof of semantic correctness merely exploits the same abstract ideas, but completely differs in all other respects, because it is now carried out against the concurrent small-step semantics instead of the sequential big-step semantics. Bytecode and the bytecode verifier also descend directly from $\mu$Java.

In contrast, the small-step semantics for source code is comparatively young, as Bali only defines a preliminary one and $\mu$Java none at all. Thus, in retrospect, Nipkow’s and Oheimb’s investment in Bali [129] keeps paying off. Still, it has taken more than a decade to achieve their goal of extending Bali with Java concurrency.

Among the numerous improvements in Isabelle, the effect of three on the formalisation deserves mentioning. The most obvious is the Isar language [179, 25] of human-readable proof, which I have discussed already in §7.2. Some parts of JinjaThreads go back to early parts of Bali and $\mu$Java and are therefore still written as imperative proof scripts.

Second come more powerful packages that reduce clutter in definitions and proofs. In detail, Krauss’ packages for defining (partial) recursive functions [88,89] automate termination proofs or render them unnecessary and additionally derive custom rules for induction and case analysis. Even if the function is primitively recursive and therefore structural induction would suffice, proof automation often works better with custom rules, because the latter incorporates how other parameters change in recursive calls. With structural induction, proof automation often fails to discover these changes. For example, I use \texttt{fun} to define $\text{ok-thrs}$ from Figure 3.10. The auxiliary lemma $[\text{ok-thrs ts nts}; \text{Spawn t x h} \in \}$

299
set $nts\] \rightarrow ts t = None$ is proven automatically with the custom induction rule, but automation fails with structural induction on $nts$.\footnote{Alternatively, I could have defined $ok-thrs$ inductively, like many of the other predicates (see the discussion in \S7.2). $fun$’s induction rule is more versatile, because the induction rule from an inductive definition requires that the statement to prove contains $ok-thrs$ as a premise. When proving simplification rules for such predicates, I can use only the former directly.}

The third aspect concerns code extraction. Jinja and its predecessors used the first version of Isabelle’s code generator by Berghofer \cite{22}. This has now been replaced by Haftmann’s \cite{62} and Bulwahn’s predicate compiler \cite{21}. The new version handles type classes, data refinement, and data type invariants, and compiles predicates inside the logic. In \cite{92,109,113}, I have explored these new possibilities for code extraction. JinjaThreads exploits them in the following ways. It rarely uses type classes – a widely-used Isabelle feature for type-safe overloading \cite{63} – but naturally depends on them via the standard library. Data refinement and data type invariants \cite{60} are essential to a posteriori replace unexecutable or inefficient data structures by more efficient ones, e.g., tabulation (\S6.3.3).

Berghofer’s code generator was already able to handle inductive definitions, but it performed the (non-trivial) transformations outside the LCF inference kernel \cite{21}. The new predicate compiler computes a functional implementation for inductive definitions inside the logic and proves it correct. Apart from the increased confidence of correctness, all Isabelle tools can now be used to manipulate the functional interpreter. For example, JinjaThreads simplifies the code equations to improve performance (see \S6.3.4). Moreover, an implementation of Russell’s definite description operator using the functional interpreter \cite{113} (\S6.1.2) would require an extension of the code generator if compilation happened outside the logic.

In \cite{24}, Berghofer and Strecker report on the effort needed to extract $\mu$Java’s well-formedness checker and compiler. They had to replace manually existential quantifiers like in $wf-J-mdecl$’s definition (Figure 2.10) with new predicates. Bulwahn has added a preprocessor to transform definitions in predicate logic to a set of introduction rules \cite{113}, such that this step is no longer necessary. Still, it is not advisable to invoke the preprocessor on $wf-J-mdecl$, because it applies the transformation too aggressively. Not only does it introduce such an inductive predicate for the existentially quantified subformula, but also for all the other constants,
7.4. Comparison between Java and JinjaThreads

i.e., $|\_|$, distinct, set, $D$, and for the constants these depend on. To guide the preprocessor, I have promoted this subformula to a constant of its own, invoked it on the new constant, and replaced the subformula by the new constant in $wf$-$J$-$mdecl$'s code equation. Although this still requires manual intervention, it is a significant improvement over the state in 2003, because the user does not need to perform any of the conversions himself.

Apart from that, Berghofer and Strecker also face the problem that a definition is only (efficiently) executable under certain preconditions, but all calling contexts satisfy the precondition. In JinjaThreads, this concerns least upper bounds and the typing rules (§6.2.3). It is unfortunate that no satisfactory solution has been found to date. Since the code generator restricts invariants for data types to be first-order [60], they can only solve this problem in the most simple cases.

7.4 Comparison between Java and JinjaThreads

JinjaThreads is not Java – this becomes most obvious when one looks at all the transformations Java2Jinja needs to perform (§6.5.1) – but JinjaThreads does not aim for a comprehensive model, anyway. Nevertheless, JinjaThreads covers the full range of concurrency features from the JLS. In particular, most thread features are accessed through native method calls like in Java. It is faithful except for spurious wake-ups and atomicity of clone (§4.3.6).

However, it misses a few concurrency issues from the Java standard API. Most prominent are the java.util.concurrent package and Thread's methods isAlive, holdsLock, and getState. The former, a library of lock-free and thread-safe classes for synchronisation and exchanging data between threads, builds on an atomic compare-and-set (CAS) operation. In JinjaThreads, the CAS operation could be implemented directly as a native method that atomically reads and writes a volatile location.\(^{59}\) Similarly, one could add the three methods of class Thread, too. holdsLock could be implemented directly via the following thread actions: $(Unlock \rightarrow a, Lock \rightarrow a)$ when it returns True, and

\(^{59}\)The java.util.concurrent classes define further CAS methods with weaker ordering constraints than volatile semantics. To model them, one has to define appropriate synchronisation events and extend the JMM accordingly. Since the proofs about the memory model hardly depend on the kind of synchronisation, this should not pose any big problems.
(UnlockFail→a) for False. However, isAlive requires another BTA for querying that a thread has not yet terminated (the dual to Join). getState necessitates further BTAs, e.g., for querying whether a thread is waiting in a wait set or joining on another thread. For the latter, the current implementation for join no longer works. At present, if a (uninterrupted) thread \( t \) joins on thread \( t' \), but \( t' \) has not yet finished, then \( t \) is stuck at the call. In particular, the multithreaded state does not record the failed attempt, but \( t' \)'s state that getState returns changes from running to waiting. One could extend the interleaving semantics to record such failed attempts in the multithreaded state. However, getState’s documentation disallows to use it for synchronisation purposes in the JMM sense. Hence, the same issues arise as for synchronisation via thread spawning (Figure 4.28).

As for sequential Java, numerous features are still missing, the following lists them. From the concurrency point of view, class initialisation, final fields, and finalisation are particularly interesting for the following reasons:

1. Classes must be initialised at most once even if multiple threads trigger initialisation concurrently. Interesting points are whether the initialisation procedure in the JLS [56, §12.4.2] in fact achieves this, and how one can model the allowed deviations from it [56, §12.4.3]. The latter affects whether a program is correctly synchronised, because threads can implicitly synchronise through class initialisation.

For example, consider the two threads in Figure 7.2 with the classes and their static initialisers on the left. Class initialisation occurs in ll. 2 and 3, and both classes cause initialisation of the other. Note that the initialisation of B succeeds only when it is triggered by A’s initialisation, because in that case, B’s static initialiser reads the default value false for A.a as part of recursive class initialisation. Hence, l. 4 executes only if l. 2 executes before l. 3. There is only one potential data race, namely on x between l. 1 and l. 4 – accesses to the static members occurs only in static initialisers and therefore cannot participate in data races. Since initialisation involves locking and unlocking, l. 2 happens before (in the JMM sense) l. 3, and so does l. 1 before l. 4. Thus, there is no data race and the program is correctly synchronised. However, the JLS allows compilers and the JVM to remove unnecessary synchronisation for class initialisation [56, §12.4.3]. It is unclear whether this exception applies to the above
class A {
    static boolean t = B.b;
    static boolean a = true;
}

class B {
    static boolean b = true;
    static {
        if (A.a) throw new Error();
    }
}

Initially: \( x = 0 \)

1: \( x = 1 \);
2: new A();
3: new B();
4: \( r = x \);

**Figure 7.2:** Synchronisation through and potential deadlock during class initialisation

Example and whether the compiler must still ensure the semantics of synchronisation, i.e., whether ll. 3 and 4 must not be reordered.

Moreover, class initialisation according to the JLS [56, §12.4.2] may cause the program from Figure 7.2 to deadlock. If both threads simultaneously start to initialise A and B, respectively, each thread can end up in the other class’ monitor, waiting to be notified. Since there is no other thread to wake them up, they are deadlocked.\(^{60}\)

This is a known defect of the class initialisation procedure, but no better solution is known [35]. If class initialisation is revised again, one could then use a suitable extension of JinjaThreads to prove the absence of deadlocks.

Class initialisation would require non-trivial changes to the semantics and the compiler verification. Liu and Moore [104] model class initialisation by switching their VM M6 into a designated state and executing the static initialiser like a static method in its own call frame. I guess that a similar approach might work for JinjaThreads, too.

2. For final fields, the JMM provides stronger guarantees than for ordinary ones [56, §17.5]. Since final fields are closely tied to constructors, one first must extend JinjaThreads with constructors.\(^{61}\)

---

\(^{60}\) Even if either of the threads wakes up spuriously, step 2 of the initialisation procedure immediately suspends the thread again. Hence, they should be considered deadlocked in that case, too. However, spurious wake-ups would remove the threads from the waits again, i.e., both threads would keep spinning in step 2. This is an example of how spurious wake-ups can obscure deadlocks.

\(^{61}\) A good starting point would be Klein’s work on constructors in \(\mu\)Java bytecode [79].
Chapter 7. Discussion and Future Work

but it should then be straightforward to extend the JMM formalisation with final fields. However, final fields are designed to be used without synchronisation. In particular, writes to final fields need not happen before reads [56, §17.5.1] – otherwise, final fields would have synchronisation semantics like volatiles. Therefore, one must also revisit data-race freedom. Ideally, one would show that data races on final fields can only occur in programs that already contain another data race on some non-final field.62 This would strengthen the DRF guarantee, because one would not need to consider final fields when checking data race freedom. If this does not hold, however, one should nevertheless exclude final fields from data races, because their usage should not expel programs from the DRF guarantee. However, one then has to check whether the DRF guarantee still holds.

3. Finalisation executes in separate finalisation threads for which the JMM defines separate visibility restrictions [56, §12.6.1.1]. It would be interesting to study which coding idioms for finalisers in fact yield DRF programs.

Note that exceptions are the only kind of non-local transfer of control in JinjaThreads source code. From the compiler perspective, it would be interesting to also include break, continue, return, because they interact with sync (_) _ blocks non-trivially. If one of the former causes such a block to terminate abruptly, the lock must be released like in the exceptional case, but bytecode provides no dedicated support like an exception handler table for that. More generally, finally blocks – which are missing, too – can intercept such transfer of control, execute arbitrary code and even redirect the flow of control. While it is well-known how to model such constructs semantically, e.g. [99, 135], the compiler and its verification would need substantial adaptations.

62The informal argument goes like this: Only the constructor assigns to final fields of an object, i.e., there are exactly two writes to a final field: the initialisation with the default value and the constructor’s assignment, but they never conflict by definition. Since a conflicting read of the final field must occur in another thread, the thread must have learnt the object’s address before. Hence, both threads must synchronise to pass the address without a data race. If the thread adheres to the coding discipline of not writing “a reference to the object being constructed in a place where another thread can see it before the object’s constructor is finished” [56, §17.5], then the writes to the final field happen before the reads. It remains to be shown whether this informal argument can be made rigorous.
7.5. Future work

When one views JinjaThreads as an efficient, executable formalised interpreter and VM, the most pressing issues are static fields and methods, interfaces, the remaining primitive types, and primitive support for Strings. Java\textsuperscript{light} has already covered the first two [135], but neither \(\mu\text{Java}\) nor Jinja have. Reintroducing them would be tedious, but uninteresting. The main obstacle for primitive types is that Isabelle lacks a formalisation of floating points at present, but existing solutions in other theorem provers should be easily portable [43,66,153]. Busenius [37] has included Strings in his port of JinjaThreads to Coq.

Apart from the above, JinjaThreads also omits the following: method overloading, abstract classes and methods, \textit{throws} clauses, auto-boxing, generics, nested classes, packages, access modifiers, class loaders, reflection, and the statements assert and switch.

7.5 Future work

The mainly negative results on the JMM – the original set of JMM events is insufficient to implement the Java API (§4.3.2), type safety hinges on all objects being in existence from the start (§4.3.2), security guarantees are compromised (§4.3.5) – immediately raise the question for future research: How can we do better? Since the JMM does not allow optimisations as intended [8,38,115,162,171], either, another revision seems necessary. However, it is unclear what out-of-thin-air values should be and how to characterise them formally. Analogously, it is currently unknown how to formally state and prove the security guarantees that the JMM is supposed to provide. Future work must address these questions and investigate how possible solutions restrict the compiler beyond the DRF model.

Although I have shown how to connect syntax and single-threaded semantics with the JMM, the solution is not completely satisfying, because interleaving semantics still occurs as an intermediate layer. It could be eliminated with two radical changes. First, the shared heap is reduced to carry no information at all – the allocation operation of the heap module returns non-deterministically any address, even if it might already be in use. Then, every thread executes in isolation. Second, the multithreaded semantics connects to single threads via their set of possible traces of thread actions and events. It imposes the global synchronisation order, ensures mutual exclusion of locks and freshness of allocations, keeps
track of the wait sets and interrupts, etc. The JMM then matches reads and writes as before.

Since this amounts to a complete redesign, most proofs would need to be redone, because the preservation proofs rely on the global notion of time. However, such a change would free the way to re-introduce the (more intuitive) big-step semantics for single threads. It seems worthwhile to explore whether coinductive big-step operational semantics, which has recently become popular in the Coq community [101, 124, 125], offers an elegant approach for infinite executions for JinjaThreads, too.

The Quis custodiet project, in which JinjaThreads originates, suggests another direction for future work, namely extending JinjaThreads to full Java. Then, one could run safety-critical programs, which the verified IFC tool has certified, in an extracted VM such that the program behaviour definitely respects the semantics.

At the moment, the main show stoppers are interfaces, primitive types like `long` and `double`, and class initialisation. Interfaces only affect the type system, but not concurrency; since Bali models them, including them should not pose any deep problems. Neither would additional primitive types, as they are orthogonal to concurrency, but this would require to formalise floating-point arithmetic in Isabelle first. Class initialisation is a different matter, see the previous section for details.

The challenges in this scenario are (i) keeping the formal model tractable and (ii) tuning the extracted VM for efficiency. However, I would not expect any deep insights for Java as a programming language, since most features have already been studied in isolation.

Finally, if either direction is followed, validation of the model becomes even more important. My work in §6.5.2 is only a first step in this direction, since the scheduler produces only one single behaviour, and only for the sequentially consistent heap implementation. Hence, validation tests the sequential subset more thoroughly than multithreading. In particular, this approach cannot prove that some behaviour is erroneously disallowed due to the inherent non-determinism. To boost confidence in JinjaThreads as a model for Java, one can devise a tool that tests whether a given behaviour is allowed, or that enumerates all allowed behaviours of a program. Then, one could explore the full model, ideally interactively, albeit only for tiny programs due to the state space explosion. For sequential consistency, e.g., enumeration of all possible interleavings would be possible by using the predicate compiler for the interleaving semantics (see §6.3.2).
My mechanised model of multithreaded Java spans from a realistic subset of Java source code and bytecode plus compiler via operational semantics for statements and instructions to the axiomatic Java memory model. I have shown type safety, consistency of the JMM, and the DRF guarantee, and verified the compiler. It is executable and has been tested thoroughly.

In this work, I have focused on studying the effects of concurrency in a unified model rather than in isolation such that important details cannot be missed easily. Tractability quickly becomes a major concern, modularity is the key to push the limits. I have demonstrated how to disentangle sequential aspects, concurrency features, and the memory model from each other, and the proofs show that JinjaThreads is indeed a usable model despite being sizeable.

Beyond the structuring principle and unified model, I also contributed to the analysis of individual facets. First, I have given a precise and formal definition of deadlock in terms of the single-threaded semantics. Hence, the type safety statement is stronger than previous ones which over-approximate deadlocks syntactically. Second, the compiler verification was the first for multithreaded Java. Although the proof technique is much more difficult than for sequential languages, it can be reused to verify translations between different program representations. Although the compiler literature often considers such translations obviously correct, this work shows once more [99] that formally proving them correct is non-trivial. Third, I have bridged the gap between Java and the Java memory model, and identified several subtle cases that the JMM misses. Via this link, I also proved the DRF guarantee and consistency, and discovered that the JMM in theory breaks Java’s security architecture.
Chapter 8. Conclusion

The DRF proof is not limited to Java, because it is largely independent of the Java-specific legality conditions. In fact, the key lemma 4.4 plays a similar role in other DRF guarantee proofs, e.g., [4,32]. They all postulate sequentially consistent completions of prefixes, which I have constructed formally for the first time and for a realistic language. For Java, this surprisingly requires a full type-safety proof, but this need not be a restriction for other languages. C and C++, e.g., assign such type-unsafe programs undefined semantics and exclude them from the guarantee.

For the Quis custodiet project, the formalisation of bytecode and the executable (sequentially consistent) VM are probably the most relevant parts. QC’s analyses for information flow control operate on Java bytecode, so it is natural to verify them against the bytecode formalisation. The executable VM and the converter Java2Jinja are the first step towards a trusted environment for executing security-critical Java programs.

However, the analyses are sound only for sequential consistency, but not the JMM. Like for Java’s security architecture, out-of-thin-air values compromise their soundness, too. For example, standard points-to analysis as used in VALSOFT/Joana and QC determines that r2 and r3 never alias for the program in Figure 4.27, but the JMM allows aliasing. Since the IFC algorithms build on this information, they, too, become unsound for programs with data races, although Hammer [64] and Gifhorn [53] claim that VALSOFT/Joana correctly over-approximates the weak semantics of the JMM. This is just another evidence that the specification of the JMM is flawed.

Beyond Quis custodiet, JinjaThreads is already used in other contexts, too. Busenius [37] has ported a preliminary version to Coq and extended it with generics and wildcards. He uses it as a target language for the verification of Expi2Java [10], a code generator for cryptographic protocols. He chose JinjaThreads “over other formalized Java fragments because of its comprehensiveness.” [10, §6]. In the context of the CeTA project [170], Kochesser is using Java2Jinja and the compiler to convert Java programs into Jinja bytecode [personal communication], which is then fed to CeTA’s termination and complexity analyser.

In the future, JinjaThreads could be used, e.g., to cross-validate other formal semantics for (concurrent) Java. For example, Trentelman [172] has verified some proof rules of the KeY tool [19], a verification tool for JavaCard programs, against Jinja’s predecessor Bali. Continuing this effort, JinjaThreads could formally link different analyses tools like KeY with the IFC checker formalisation from the Quis custodiet project. In
such a setting, the compiler verification becomes especially valuable because it permits to transfer results freely between source code (KeY) and bytecode (QC).
Producer-consumer example

This producer-consumer example program in JinjaThreads source code syntax has been used for benchmarking the interpreter and VM (§6.3.4).

Example $n = \text{Program} \ (\text{SystemClasses}\@\ [\text{ThreadC}, \text{StringC}, \text{IntegerC}, \text{BufferC}, \text{ProducerC n, ConsumerC n, TestC}])$

ProducerC $n =$
("Producer", Thread,
 [("buffer", Class "Buffer", {volatile = False})],
 [(run, [], Void, []),
  {"i": Integer = [Intg 0];
   while (Var "i" «!=» Val (Intg (word-of-int n))))
   (Var "buffer"."put"(["temp": Class "Integer";
                        "temp" := new "Integer";;
                        Var "temp"."value"{} := "i";; Var "temp"]));
  "i" := Var i «+» Val (Intg 1))])

ConsumerC $n =$
("Consumer", Thread,
 [("buffer", Class "Buffer", {volatile = False})],
 [(run, [], Void, []),
  {"i": Integer = [Intg 0];
   while (Var "i" «!=» Val (Intg (word-of-int n)))
   {"o": Class Object;
    "o" := Var "buffer"."get"([[]);
    "i" := Var i «+» Val (Intg 1))})]
Appendix A. Producer-consumer example

BufferC =
(“Buffer”, Object,
[([“buffer”, Class Object[], {volatile = False}]),
(“front”, Integer, {volatile = False}),
(“back”, Integer, {volatile = False}),
(“size”, Integer, {volatile = False})],
[([“constructor”, [Integer], Void, [[]],
  “buffer” := new Class Object[Var “size”]bool;
  “front” := Val (Intg 0);;
  “back” := Val (Intg -1);;
  Var this.”size”[] := Val (Intg 0))],
(“empty”, [], Boolean, [[]],
  sync (Var this) (Var “size” == Val (Intg 0))],
(“full”, [], Boolean, [[]],
  sync (Var this) (Var “size” == Var “buffer”.length)],
(“get”, [], Class Object, [[]],
  sync (Var this) (while (Var this.”empty”([]))
    try Var this.wait([]) catch(InterruptedException “e”) unit;;
  “size” := Var “size” + Val (Intg 1);;
  [“result” : Class Object;
    “result” := Var “buffer”[Var “front”];;
    “front” := Var “front” + Val (Intg 1);;
    (if (Var “front” == Var “buffer”.length)
      “front” := Val (Intg 0)
    else unit);;
    Var this.notifyAll([]);
    Var “result”])]],
(“put”, [Class Object], Void, [[]],
  sync (Var this) (while (Var this.”full”([]))
    try Var this.wait([]) catch(InterruptedException “e”) unit;;
    “back” := Var “back” + Val (Intg 1);;
    (if (Var “back” == Var “buffer”.length)
      “back” := Val (Intg 0)
    else unit);;
    Var “buffer”[Var “back”] := Var “o”;
    “size” := Var “size” + Val (Intg 1);;
    Var this.notifyAll([])])]})
TestC =
("Test", Object, [],
 ["main", [Class "String"[]], Void, (\n {"b": Class "Buffer";
 "b" := new "Buffer";;
 Var "b"."constructor"([Val (Intg 10)]);;
 {"p": Class "Producer";
 Var "p" := new "Producer";;
 {"c": Class "Consumer";
 Var "c" := new "Consumer";;
 Var "c"."buffer"{} := Var "b";;
 Var "p"."buffer"{} := Var "b";;
 Var "c".start([]); Var "p".start([]));])

IntegerC =
("Integer", Object, [("value", Integer, {volatile = False})], [])

StringC = ("String", Object, [], [])

ThreadC =
(Thread, Object, [],
 [(run, [], Void, [(unit)]),
 (start, [], Void, Native), (join, [], Void, Native),
 (interrupt, [], Void, Native), (isInterrupted, [], Boolean, Native)])

SystemClasses =
[ObjectC, ThrowableC, NullPointerC, ClassCastC, OutOfMemoryC, 
 ArrayIndexOutOfBoundsException, ArrayStoreC, NegativeArraySizeC, 
 ArithmeticC, IllegalMonitorStateException, IllegalThreadStateException, 
 InterruptedException]

ObjectC =
(Object, "", [],
 [(wait, [], Void, Native), (notify, [], Void, Native),
 (notifyAll, [], Void, Native), (hashCode, [], Integer, Native),
 (clone, [], Class Object, Native), (print, [Integer], Void, Native),
 (currentThread, [], Class Thread, Native),
 (interrupted, [], Boolean, Native), (yield, [], Void, Native)])

ThrowableC = (Throwable, Object, [], [])
Appendix A. Producer-consumer example

NullPointerC = (NullPointer, Throwable, [], [])
ClassCastC = (ClassCast, Throwable, [], [])
OutOfMemoryC = (OutOfMemory, Throwable, [], [])
ArrayIndexOutOfBoundsC =
  (ArrayIndexOutOfBounds, Throwable, [], [])
ArrayStoreC = (ArrayStore, Throwable, [], [])
NegativeArraySizeC = (NegativeArraySize, Throwable, [], [])
ArithmeticC = (ArithmeticException, Throwable, [], [])
IllegalMonitorStateC = (IllegalMonitorState, Throwable, [], [])
IllegalThreadStateC = (IllegalThreadState, Throwable, [], [])
InterruptedC = (InterruptedException, Throwable, [], [])
Formal definitions

This appendix contains the formal definitions for JinjaThreads source code and bytecode syntax and semantics. I have not repeated the interleaving framework from §3.1, because §3.1 already contains all unabridged definitions.

B.1 Declarations and lookup functions

Type declarations

datatype \( 'addr \) val = Unit | Bool bool | Int sg word32 | Null | Addr 'addr

datatype ty = Void | Boolean | Integer | NT | Class cname |
    Array ty

datatype hty = ClassT cname | ArrayT ty nat

datatype 'm prog = Program 'm cdecl list

type_synonym 'm cdecl = cname × 'm class

type_synonym 'm class = cname × fdecl list × 'm option mdecl list

type_synonym fdecl = vname × ty × fmod

record fmod = volatile :: bool

type_synonym 'm mdecl = mname × ty list × ty × 'm

\[
\begin{align*}
\text{ty-of (ClassT C)} &= \text{Class C} \\
\text{ty-of (ArrayT T n)} &= T[] \\
\text{array-length-of (ArrayT T n)} &= n
\end{align*}
\]

Lookup functions

class declarations classes (Program P) = P

class declaration class (Program P) = map-of P

valid class name is-class P C = (class P C ≠ None)
Appendix B. Formal definitions

direct subclass relation
\[
\text{class } P \ C = \lfloor (\text{D, rest}) \rfloor \quad \text{C \neq Object}
\]
\[P \vdash C \prec 1 \ \text{D} \]

subclass relation
\[P \vdash C \preceq ^* \text{D} = (\lambda C. D. P \vdash C \prec 1 \text{D})^{**} \ C \ \text{D} \]

valid types
\[
is\text{-type } P \ \text{Void} = \text{True} \\
is\text{-type } P \ \text{Boolean} = \text{True} \\
is\text{-type } P \ \text{Integer} = \text{True} \\
is\text{-type } P \ (\text{Class C}) = \text{is-class } C \\
is\text{-type } P \ (T[]) = \text{is-type } P \ T \land \text{base-type } T \neq \text{NT}
\]

types \( P = \{T. \text{is-type } P \ T\} \)

is-htype \(P \ hT = \text{is-type } P\ (\text{ty-of } hT)\)

base type
\[
\text{base-type } T = (\text{case } T \text{ of } T'[^{\prime}] \Rightarrow \text{base-type } T' | _{_{\_}} \Rightarrow T)
\]

method lookup
\[
_{_{\_}} \vdash _{_{\_}} \ \text{sees-methods}_ {_{::}}:
\]
\[
\text{'m prog} \Rightarrow \text{cname} \Rightarrow (\text{mname} \rightarrow (\text{ty list} \times \text{ty} \times 'm) \times \text{cname}) \Rightarrow \text{bool}
\]
\[
\text{class } P \ \text{Object} = \lfloor (\text{D, fs, ms}) \rfloor \\
Mm = \text{Option.map } (\lambda m. (m, \text{Object})) \circ \text{map-of } ms
\]
\[P \vdash \text{Object sees-methods } Mm
\]
\[
\text{class } P \ C = \lfloor (\text{D, fs, ms}) \rfloor \quad \text{C \neq Object} \\
P \vdash D \ \text{sees-methods } Mm
\]
\[Mm' = Mm +\text{ (Option.map } (\lambda m. (m, C)) \circ \text{map-of } ms)
\]
\[P \vdash C \ \text{sees-methods } Mm'
\]

where
\[
\text{Option.map } f \ None = \text{None} \\
\text{Option.map } f \ [x] = [f \ x]
\]
\[
P \vdash C \ \text{sees } M:Ts\rightarrow T = \text{meth in } D = \\
(\exists Mm. P \vdash C \ \text{sees-methods } Mm \land Mm \ M = \lfloor((Ts, T, \text{meth}), D)\rfloor)
\]

method \( P \ C \ M = (\iota(D, Ts, T, \text{meth}). P \vdash C \ \text{sees } M:Ts\rightarrow T = \text{meth in } D)\)
field lookup

\[ P \vdash F \text{ has-fields } FDTs \colon:\]
\[ 'm \text{ prog } \Rightarrow \text{ cname } \Rightarrow ((vname \times \text{ cname}) \times (ty \times \text{ fmod})) \text{ list } \Rightarrow \text{ bool} \]

\[
\begin{align*}
\text{class } P \text{ Object } & = \lfloor (D, fs, ms) \rfloor \\
FDTs & = \text{ map } (\lambda (F, T, fm). ((F, \text{ Object}), (T, fm))) \text{ fs} \\
\]
\[ P \vdash \text{ Object has-fields } FDTs \]

\[
\begin{align*}
\text{class } P \text{ C } & = \lfloor (D, fs, ms) \rfloor \quad C \neq \text{ Object} \\
FDTs' & = \text{ map } (\lambda (F, T, fm). ((F, C), (T, fm))) \text{ fs } @ FDTs \\
\]
\[ P \vdash C \text{ has-fields } FDTs' \]

\[ P \vdash C \text{ has } F : T \text{ (fm) in } D = \]
\[ (\exists \text{ FDTs}. P \vdash C \text{ has-fields } FDTs \land \text{ map-of } FDTs \ (F, D) = \lfloor (T, fm) \rfloor) \]

\[ P \vdash C \text{ sees } F : T \text{ (fm) in } D = (\exists \text{ FDTs}. P \vdash C \text{ has-fields } FDTs \land \]
\[ \text{ map-of } (\lambda ((F, D), T fm). (F, (D, T fm))) \text{ FDTs} \text{ F} = \lfloor (D, T, fm) \rfloor) \]

\[ \text{fields } P \text{ C } = (\iota \text{ FDTs}. P \vdash C \text{ has-fields } FDTs) \]
\[ \text{field } P \text{ C F } = (\iota (D, T, fm). \ . P \vdash C \text{ sees } F : T \text{ (fm) in } D) \]
\[ \text{is-volatile } P \ (\text{Field } F \ D) = \text{ volatile } (\text{snd } (\text{snd } (\text{field } P \ D \ F))) \]
\[ \text{is-volatile } P \ (\text{Cell } n) = \text{ False} \]

B.2 Binary operators

datatype \[bop = == | != | < | <= | > | >= | + | - | * | / | \% | << | >> | >>> | & | | | ^ \]

Signatures

\[
\begin{align*}
P \vdash T_1 \leq T_2 \lor P \vdash T_2 \leq T_1 & \quad P \vdash T_1 \leq T_2 \land P \vdash T_2 \leq T_1 \\
P \vdash T_1 \ll T_2 & : \text{ Boolean} \quad P \vdash ! T_1 \ll T_2 & : \text{ Boolean} \\
P \vdash \text{ Integer} \ll \text{ Integer} & : \text{ Boolean} \quad P \vdash \text{ Integer} \ll T_2 & : \text{ Boolean} \\
P \vdash \text{ Integer} \gg T_2 & : \text{ Boolean} \quad P \vdash \text{ Integer} \gg \text{ Integer} & : \text{ Boolean} \\
P \vdash \text{ Integer} \ll T_2 & : \text{ Integer} \quad P \vdash \text{ Integer} \gg \text{ Integer} & : \text{ Integer} \\
P \vdash \text{ Integer} \ll \text{ Integer} & : \text{ Integer} \quad P \vdash \text{ Integer} \gg \text{ Integer} & : \text{ Integer} \\
P \vdash \text{ Integer} \ll \text{ Integer} & : \text{ Integer} \quad P \vdash \text{ Integer} \gg \text{ Integer} & : \text{ Integer} \\
\end{align*}
\]
Appendix B. Formal definitions

\[ P \vdash \text{Integer} \llcorner \% \lrcorner \text{Integer} :: \text{Integer} \quad P \vdash \text{Integer} \llcorner \llcorner \lrcorner \text{Integer} :: \text{Integer} \]

\[ P \vdash \text{Integer} \llcorner \gg \lrcorner \text{Integer} :: \text{Integer} \quad P \vdash \text{Integer} \llcorner \gg \gg \lrcorner \text{Integer} :: \text{Integer} \]

\[ P \vdash \text{Integer} \llcorner \& \lrcorner \text{Integer} :: \text{Integer} \quad P \vdash \text{Boolean} \llcorner \& \lrcorner \text{Boolean} :: \text{Boolean} \]

\[ P \vdash \text{Integer} \llcorner | \lrcorner \text{Integer} :: \text{Integer} \quad P \vdash \text{Boolean} \llcorner | \lrcorner \text{Boolean} :: \text{Boolean} \]

\[ P \vdash \text{Integer} \llcorner \prec \lrcorner \text{Integer} :: \text{Integer} \quad P \vdash \text{Boolean} \llcorner \prec \rhd \lrcorner \text{Boolean} :: \text{Boolean} \]

**Semantics**

\[
\begin{align*}
\text{binop} == & \quad v_1 \quad v_2 \quad [\text{Inl} (\text{Bool} (v_1 = v_2))] \\
\text{binop} != & \quad v_1 \quad v_2 \quad [\text{Inl} (\text{Bool} (v_1 \neq v_2))] \\
\text{binop} < & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) \quad [\text{Inl} (\text{Bool} (i_1 < i_2))] \\
\text{binop} <= & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) \quad [\text{Inl} (\text{Bool} (i_1 \leq i_2))] \\
\text{binop} > & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) \quad [\text{Inl} (\text{Bool} (i_2 < i_1))] \\
\text{binop} >= & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) \quad [\text{Inl} (\text{Bool} (i_2 \leq i_1))] \\
\text{binop} + & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) \quad [\text{Inl} (\text{Bool} (i_1 + i_2))] \\
\text{binop} - & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) \quad [\text{Inl} (\text{Bool} (i_1 - i_2))] \\
\text{binop} * & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) \quad [\text{Inl} (\text{Bool} (i_1 \times i_2))] \\
\text{binop} / & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) = \\
& \quad [\text{if } i_2 = 0 \text{ then Inr (addr-of-sys-xcpt ArithmeticException) \else Inl (Intg (i_1 sdiv i_2))}] \\
\text{binop} \% & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) = \\
& \quad [\text{if } i_2 = 0 \text{ then Inr (addr-of-sys-xcpt ArithmeticException) \else Inl (Intg (i_1 smod i_2))}] \\
\text{binop} \& & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) = [\text{Inl} (\text{Intg} (i_1 \text{ AND } i_2))] \\
\text{binop} \& & \quad (\text{Bool} b_1) \quad (\text{Bool} b_2) = [\text{Inl} (\text{Bool} (b_1 \wedge b_2))] \\
\text{binop} | & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) = [\text{Inl} (\text{Intg} (i_1 \text{ OR } i_2))] \\
\text{binop} | & \quad (\text{Bool} b_1) \quad (\text{Bool} b_2) = [\text{Inl} (\text{Bool} (b_1 \vee b_2))] \\
\text{binop} ^ & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) = [\text{Inl} (\text{Intg} (i_1 \text{ XOR } i_2))] \\
\text{binop} ^ & \quad (\text{Bool} b_1) \quad (\text{Bool} b_2) = [\text{Inl} (\text{Bool} (b_1 \neq b_2))] \\
\text{binop} \llcorner \llcorner & \quad (\text{Intg} i_1) \quad (\text{Intg} i_2) = [\text{Inl} (\text{Intg} (i_1 \llcorner \llcorner \text{Intg} (i_2 \text{ AND } 0x1f)))] \\
\text{binop} \llcorner \gg & \quad (\text{Intg i_1}) \quad (\text{Intg i_2}) = [\text{Inl} (\text{Intg} (i_1 \llcorner \gg \text{Intg} (i_2 \text{ AND } 0x1f)))] \\
\text{binop} \llcorner \gg \gg & \quad (\text{Intg i_1}) \quad (\text{Intg i_2}) = [\text{Inl} (\text{Intg} (i_1 \gg \gg \text{Intg} (i_2 \text{ AND } 0x1f)))] \\
\text{binop} _ & \quad _ & \quad _ & = \text{None}
\end{align*}
\]
B.3 Heap module implementations

Auxiliary functions on word32:

- \( \leq_s \) signed comparisons
- +, −, \( \ast \) addition, subtraction, and multiplication
- sdiv, smod signed division and remainder
- AND, OR, XOR bitwise “and”, “or”, and “xor”
- unat unsigned interpretation of word32 as nat
- <<, >>, >>> left shift, unsigned and signed right shift\(^{63}\)

B.3 Heap module implementations

default values

\[
\text{default-val :: } \ty \Rightarrow \text{ addr } \text{ val} \\
\text{default-val Void} = \text{ Unit} \quad \text{default-val NT} = \text{ Null} \\
\text{default-val Boolean} = \text{ Bool False} \quad \text{default-val (Class C)} = \text{ Null} \\
\text{default-val Integer} = \text{ Intg 0} \quad \text{default-val (T[\])} = \text{ Null}
\]

conformance

\[
P, h \vdash v : \leq T \iff (\exists T'. \text{typeof}_h v = [T'] \land P \vdash T' : \leq T) \\
P, h \vdash vs : [\leq] T s \iff |vs| = |Ts| \land (\forall (v, T) \in \text{set} (\text{zip} vs Ts), P, h \vdash v : \leq T) \\
P, h \vdash xs : (\leq) E \iff (\forall v. xs V = [v] \rightarrow (\exists E. E V = [T] \land P, h \vdash v : \leq T)) \\
P, E \vdash (h, xs) \sqrt{\vdash} \iff P, h \vdash xs : (\leq) E \land \text{hconf } h
\]

B.3.1 Sequential consistency

type of thread IDs (instantiates ‘t):

\[\text{type_synonym thread-id ‘t} = \text{ nat}\]

type of addresses (instantiates ‘addr):

\[\text{type_synonym addr} = \text{ nat}\]

type of the heap (instantiates ‘heap):

\[\text{type_synonym heap} = \text{ addr } \rightarrow \text{ heap-entry}\]

\[\text{datatype heap-entry} = \text{ Obj } \text{ cname } \text{ fields} | \text{ Arr } \ty \text{ fields } \text{ cells}\]

\[\text{type_synonym fields} = \text{ vname } \times \text{ cname } \rightarrow \text{ addr } \text{ val}\]

\[\text{type_synonym cells} = \text{ addr } \text{ val } \text{ list}\]

conversion between address and thread ID

\[\text{abbreviation sc-a2t} = (\lambda a. a)\]

\[\text{abbreviation sc-t2a} = (\lambda t. t)\]

\(^{63}\)Note that the meaning of >> and >>> from Isabelle’s word library is opposite to Java.
Appendix B. Formal definitions

implementation of abstract heap operations

\[
\begin{align*}
\text{sc-typeof-addr } h a &= (\text{case } h a \text{ of } [\text{Obj } C \_] \Rightarrow [\text{Class } T C] \\
&\quad | [\text{Arr } T \_ cs] \Rightarrow [\text{Array } T T \{cs\}] \\
&\quad | \text{None } \Rightarrow \text{None}) \\
\text{sc-empty-heap} &= \text{empty} \\
\text{sc-allocate } P h h T &= (\text{case new-Addr } h \text{ of } \text{None } \Rightarrow (h, \text{None}) \\
&\quad | [a] \Rightarrow (h(a \mapsto \text{blank } P h T), [a])) \\
\text{sc-read:}
\hline 
\frac{h a = [\text{Obj } C f s]}{\text{sc-read } h a (\text{Field } F D) v} & \quad (F, D) = [v] \\
\frac{h a = [\text{Arr } T f s cs]}{\text{sc-read } h a (\text{Field } F D) v} & \quad (F, D) = [v] \\
\frac{h a = [\text{Arr } T f s cs]}{\text{sc-read } h a (\text{Cell } n) cs[\_n]} & \quad n < |cs| \\
\text{sc-write:}
\hline 
\frac{h a = [\text{Obj } C f s]}{\text{sc-write } h a (\text{Field } F D) v h'} & \quad h' = h(a \mapsto \text{Obj } C ((F, D) \mapsto v))) \\
\frac{h a = [\text{Arr } T f s cs]}{\text{sc-write } h a (\text{Field } F D) v h'} & \quad h' = h(a \mapsto \text{Arr } T ((F, D) \mapsto v)) cs \\
\frac{h a = [\text{Arr } T f s cs]}{\text{sc-write } h a (\text{Cell } n) v h'} & \quad h' = h(a \mapsto \text{Arr } T f s (cs[\_n := v])) \\
\end{align*}
\]

auxiliary definitions

\[
\begin{align*}
\text{hash-addr} &= \text{word-of-int } \circ \text{int} \\
\text{new-Addr } h &= \text{(if } \exists a. h a = \text{None } \text{then } [\text{LEAST } a. h a = \text{None}] \text{ else } \text{None}) \\
\text{blank } P (\text{Class } T C) &= \text{Obj } C (\text{init-fields } (\text{fields } P C)) \\
\text{blank } P (\text{Array } T n) &= \text{Arr } T (\text{init-fields } (\text{fields } P \text{ Object }) (\text{replicate } n (\text{default-val } T))) \\
\text{init-fields} &= \text{map-of } \circ \text{map } (\lambda (F D, (T, f m)). (F D, \text{default-val } T)) \\
\end{align*}
\]

heap conformance

\[
\begin{align*}
P \vdash_{\text{sc}} h \sqrt{\_} &= (\forall h_0 \in \text{ran } h. P, h \vdash_{\text{sc}} h_0 \sqrt{\_}) \\
P, h \vdash_{\text{sc}} \text{Obj } C f s \sqrt{\_} &= \text{is-class } P C \land P, C \vdash_{\text{sc}} f s \sqrt{\_} \\
P, h \vdash_{\text{sc}} \text{Arr } T f s cs \sqrt{\_} &= \text{is-type } P (T\[]) \land P, \text{Object}, h \vdash_{\text{sc}} f s \sqrt{\_} \land (\forall v \in \text{set } cs. P, h \vdash_{\text{sc}} v : \leq T) \\
\end{align*}
\]

320
B.3. Heap module implementations

\[ P, C, h \vdash_{\text{sc}} \exists f \sqrt{f s} = \quad (\forall F T \ f m \ D. \ P \vdash C \ \text{has} \ F \ T \ (f m) \ \text{in} \ D \rightarrow \quad (\exists v. \ f s \ (F, D) = [v] \land P, h \vdash_{\text{sc}} v : \leq T)) \]

B.3.2 The Java memory model

Version 1: Type information grows

thread IDs and addresses are unified, but remain unspecified
type of the heap (instantiates \texttt{'heap}):

\texttt{type_synonym 'addr jmm-heap = 'addr \rightarrow hty}

conversion between address and thread ID

\texttt{abbreviation jmm-a2t = (\lambda a. a)}
\texttt{abbreviation jmm-t2a = (\lambda t. t)}

implementation of abstract heap operations

\texttt{jmm-typeof-addr h a} = \text{Option.map ty-of (h a)}
\texttt{jmm-empty-heap} = \text{empty}
\texttt{jmm-allocate h hT} =
\quad (\text{case jmm-new-Addr h of None ⇒ (h, None)} \quad | \quad [a] ⇒ (h(a \mapsto hT), [a]))
\texttt{jmm-read h a al v} = \text{True}
\texttt{jmm-write h a al v h'} = (h' = h)

auxiliary definitions

\texttt{jmm-new-Addr h} =
\quad (\text{if } \exists a. h a = \text{None then } [\varepsilon a. h a = \text{None}] \text{ else None})

heap conformance

\[ P \vdash_{\text{jmm}} h \sqrt{h} = (\forall hT \in \text{ran} \ h. \ \text{is-htype} \ P hT) \]

Version 2: Addresses store type information

type of addresses (instantiates \texttt{'addr}):

\texttt{datatype addr} = Address hty nat

type of thread IDs (instantiates \texttt{'t}):

\texttt{type_synonym thread-id = addr}

type of the heap (instantiates \texttt{'heap}):

\texttt{type_synonym jmm'-heap = hty \Rightarrow nat}
conversion between address and thread ID

abbreviation \(jmm'\-a2t\) \(= (\lambda a. a)\)
abbreviation \(jmm'\-t2a\) \(= (\lambda t. t)\)

implementation of abstract heap operations:

\[
jmm'\-typeof-addr \ P h \ (\text{Address} \ hT \ n) =
\begin{cases}
\lfloor hT \rfloor & \text{if } \text{is-htype} \ P \ hT \ \text{then} \ [hT] \ \text{else None}
\end{cases}
\]
\[
jmm'\-empty-heap \ = (\lambda hT. 0)
\]
\[
jmm'\-allocate \ h \ hT \ =
\begin{cases}
\begin{cases}
\text{let } n = h \ hT & \text{in}\ (h(hT := n + 1), [\text{Address} \ hT \ n])
\end{cases}
\end{cases}
\]
\[
jmm'\-read \ P \ h \ a \ al \ v \ = (\forall T. P, h \vdash_{jmm'} a \cdot al : T \rightarrow P, h \vdash_{jmm'} v : \leq T)
\]
\[
jmm'\-write \ h \ a \ al \ v \ h' \ = (h' = h)
\]

auxiliary definitions

\[
\text{hash-addr} \ (\text{Address} \ hT \ n) = \text{word-of-int} \ (\text{int} \ n)
\]

heap conformance

\[
jmm'\-hconf \ h \ = \text{True}
\]

B.4 Native methods

B.4.1 Signatures

\begin{align*}
\text{Thread.start}([\ ] & :: \text{Void} \\
\text{Thread.join}([\ ] & :: \text{Void} \\
\text{Thread.interrupt}([\ ] & :: \text{Void} \\
\text{Thread.isInterrupted}([\ ] & :: \text{Boolean} \\
\text{Object.wait}([\ ] & :: \text{Void} \\
\text{Object.notify}([\ ] & :: \text{Void} \\
\text{Object.notifyAll}([\ ] & :: \text{Void} \\
\text{Object.clone}([\ ] & :: \text{Class Object} \\
\text{Object.hashcode}([\ ] & :: \text{Integer} \\
\text{Object.currentThread}([\ ] & :: \text{Class Thread} \\
\text{Object.interrupted}([\ ] & :: \text{Boolean} \\
\text{Object.yield}([\ ] & :: \text{Void} \\
\text{Object.print}([[\text{Integer}]] & :: \text{Void}
\end{align*}
B.4. Native methods

B.4.2 Semantics of method clone

copying one member:

\[\text{copy-mem} :: 'addr \Rightarrow 'addr \Rightarrow 'loc \Rightarrow 'heap \Rightarrow ('addr, 't) \text{ event list} \Rightarrow 'heap \Rightarrow \text{bool}\]

\[
\begin{align*}
\text{read } h & \ a & \ v \quad \text{write } h & \ a' & \ v & \ h' \\
\text{copy-mem } a & \ a' & \ h \ & \ | & \ & \text{[Read } a & \ a' & \ v] \ h'
\end{align*}
\]

copying a list of members:

\[\text{copy-mems} :: 'addr \Rightarrow 'addr \Rightarrow 'loc \text{ list} \Rightarrow 'heap \Rightarrow ('addr, 't) \text{ event list} \Rightarrow 'heap \Rightarrow \text{bool}\]

\[
\begin{align*}
\text{copy-mems } a & \ a' & \ [] & \ h & \ [] & \ h \\
\text{copy-mem } a & \ a' & \ h & \ as & \ h' & \ | & \ & \text{copy-mems } a & \ a' & \ als & \ h & \ as' & \ h''
\end{align*}
\]

cloning an object or array:

\[\text{heap-clone} :: 'm \ prog \Rightarrow 'heap \Rightarrow 'addr \Rightarrow 'heap \Rightarrow (('addr, 't) \text{ event} \times 'addr) \text{ option} \Rightarrow \text{bool}\]

\[
\begin{align*}
\text{typeof-addr } h & \ a = [hT] & \text{alloc } h & \ hT = (h', \text{None}) \\
\text{heap-clone } P & \ h & \ a & \ h' & \ \text{None} \\
\text{typeof-addr } h & \ a = [\text{ClassT C}] & \text{alloc } h & \ (\text{ClassT C}) = (h, [a']) & \ P & \vdash \text{has-fields FDTs} \\
\text{copy-mems } a & \ a' & \ (\text{map } (\lambda ((F, D), _). \text{ Field } F D) \text{ FDTs}) & \ h' & \ \text{as' } h''
\end{align*}
\]

\[
\begin{align*}
\text{heap-clone } P & \ h & \ a & \ h'' & \ [(\text{New-Obj } a' C \cdot \text{as' } a')] \\
\text{typeof-addr } h & \ a = [\text{ArrayT T n}] & \text{alloc } h & \ (\text{ArrayT T n}) = (h', [a']) & \ P & \vdash \text{Object has-fields FDTs} \\
\text{als} & = \text{map } (\lambda ((F, D), _). \text{ Field } F D) \text{ FDTs } @ \text{map Cell } [0..n] \\
\text{copy-mems } a & \ a' & \ als & \ h' & \ \text{as' } h''
\end{align*}
\]

\[
\begin{align*}
\text{heap-clone } P & \ h & \ a & \ h'' & \ [(\text{New-Arr } a' T n \cdot \text{as' } a')]
\end{align*}
\]
B.4.3 Semantics of native methods

native methods of class **Thread** (Figure 3.17):

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread} \quad t' = a_2 t\ a
\]

\[
P, t \vdash \langle a.\text{start}([],) \rangle, h \rangle \rightarrow \langle \text{Spawn } t' (C, \text{run}, a) \ h, \text{TStart } t' \rangle \rightarrow_{nc} \langle \text{Ret-Val Unit}, h \rangle
\]

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread}
\]

\[
P, t \vdash \langle a.\text{start}([],) \rangle, h \rangle \rightarrow \langle \text{ThreadEx } (a_2 t\ a) \ True \rangle \rightarrow_{nc}
\]

\[
\langle \text{Ret-sys-xcpt IllegalThreadState}, h \rangle
\]

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread}
\]

\[
P, t \vdash \langle a.\text{join}([],) \rangle, h \rangle \rightarrow \langle \text{Join } t', \text{IsIntrd } t \ False, \text{TJoin } t' \rangle \rightarrow_{nc} \langle \text{Ret-Val Unit}, h \rangle
\]

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread}
\]

\[
P, t \vdash \langle a.\text{join}([],) \rangle, h \rangle \rightarrow \langle \text{IsIntrd } t \ True, \text{ClearIntr } t, \text{TIntrd } t \rangle \rightarrow_{nc}
\]

\[
\langle \text{Ret-sys-xcpt InterruptedException}, h \rangle
\]

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread}
\]

\[
P, t \vdash \langle a.\text{interrupt}([],) \rangle, h \rangle \rightarrow \langle \text{ThreadEx } t' \ True, \text{WakeUp } t', \text{Intr } t', \text{TIntr } t' \rangle \rightarrow_{nc}
\]

\[
\langle \text{Ret-Val Unit}, h \rangle
\]

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread}
\]

\[
P, t \vdash \langle a.\text{interrupt}([],) \rangle, h \rangle \rightarrow \langle \text{ThreadEx } t' \ False \rangle \rightarrow_{nc} \langle \text{Ret-Val Unit}, h \rangle
\]

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread}
\]

\[
P, t \vdash \langle a.\text{isInterrupted}([],) \rangle, h \rangle \rightarrow \langle \text{IsIntrd } t \ True, \text{TIntrd } t' \rangle \rightarrow_{nc}
\]

\[
\langle \text{Ret-Val (Bool True)}, h \rangle
\]

\[
\text{typeof-addr } h \ a = [\text{ClassT } C] \quad P \vdash C \leq^* \text{Thread}
\]

\[
P, t \vdash \langle a.\text{isInterrupted}([],) \rangle, h \rangle \rightarrow \langle \text{IsIntrd } t' \ False \rangle \rightarrow_{nc} \langle \text{Ret-Val (Bool False)}, h \rangle
\]

\[
\text{static native methods of class **Thread** as methods of class **Object** (Figure 3.18):}
\]

\[
P, t \vdash \langle a.\text{currentThread}([],) \rangle, h \rangle \rightarrow \langle \text{Ret-Val (Addr (t_2 a t))}, h \rangle
\]

\[
P, t \vdash \langle a.\text{interrupted}([],) \rangle, h \rangle \rightarrow \langle \text{IsIntrd } t \ True, \text{ClearIntr } t, \text{TIntrd } t \rangle \rightarrow_{nc}
\]

\[
\langle \text{Ret-Val (Bool True)}, h \rangle
\]
native methods of class Object (Figures 2.12 and 3.19 and Equation 4.1):

\[
P, t \vdash \langle \text{a.interrupted}(), h \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val} \ (\text{Bool} \ \text{False}), h \rangle
\]

\[
P, t \vdash \langle \text{a.yield}(), h \rangle \rightarrow_{\text{nc}} \langle \text{Ret-Val} \ \text{Unit}, h \rangle
\]
aggressive functional implementation of native methods:

\[
\text{exec-native} :: 'm \text{ prog} \Rightarrow 't \Rightarrow '\text{addr} \Rightarrow \text{ mname} \Rightarrow '\text{addr} \text{ val list} \Rightarrow '\text{heap} \\
\Rightarrow (('\text{addr}', 't', \text{cname} \times \text{mname} \times '\text{addr}', '\text{heap}', '\text{addr}', ('\text{addr}', 't') \text{ event}) \\
\text{thread-action} \times '\text{addr} \text{ native-ret} \times '\text{heap}) \text{ set}
\]

\[
\text{exec-native } P \ t \ a \ M \ vs \ h = \\
(\text{if } M = \text{ wait then} \\
\{(\text{Unlock} \to a, \text{Lock} \to a, \text{IsIntrd } t \ True, \text{ClearIntr } t, T\text{Intrd } t), \\
\text{Ret-sys-xcpt InterruptedException}, h), \\
(\text{Suspend } a, \text{Unlock} \to a, \text{Lock} \to a, \text{Release} \to a, \text{IsIntrd } t \ False, \text{SUnlock } a), \\
\text{Ret-Changed}, h), \\
(\text{UnlockFail} \to a), \text{Ret-sys-xcpt IllegalMonitorState}, h), \\
(\text{Notified}), \text{Ret-Val Unit}, h), \\
(\text{WokenUp}, \text{ClearIntr } t, T\text{Intrd } t), h, \text{Ret-sys-xcpt InterruptedException}) \} \\
\text{else if } M = \text{ notify then} \\
\{(\text{Notify } a, \text{Unlock} \to a, \text{Lock} \to a), \text{Ret-Val Unit}, h), \\
(\text{UnlockFail} \to a), \text{Ret-sys-xcpt IllegalMonitorState}, h) \} \\
\text{else if } M = \text{ notifyAll then} \\
\{(\text{NotifyAll } a, \text{Unlock} \to a, \text{Lock} \to a), \text{Ret-Val Unit}, h), \\
(\text{UnlockFail} \to a), \text{Ret-sys-xcpt IllegalMonitorState}, h) \} \\
\text{else if } M = \text{ clone then} \\
\{(K^\text{fr} [], [], [], [], [], \text{as}), \text{Ret-Val } a', h') | as a' h' . \text{heap-clone } P \ h a \ h' [(as, a')] \} \\
\cup \{(\text{}, \text{Ret-sys-xcpt OutOfMemory}, h') | h' . \text{heap-clone } P \ h a \ h' \text{ None} \} \\
\text{else if } M = \text{ hashcode then} \{(\text{}, \text{Ret-Val (Intg (hash-addr a))}, h) \} \\
\text{else if } M = \text{ print then} \{(\text{Extern } a \ M \ vs \ Unit}, \text{Ret-Val Unit}, h) \} \\
\text{else if } M = \text{ currentThread then} \{(\text{}, \text{ Ret-Val (Addr (a2t a))}, h) \} \\
\text{else if } M = \text{ interrupted then} \\
\{(\text{IsIntrd } t \ True, \text{ClearIntr } t, T\text{Intrd } t), \text{Ret-Val (Bool True)}, h), \\
(\text{IsIntrd } t \ False), \text{Ret-Val (Bool False)}, h) \} \\
\text{else if } M = \text{ yield then} \{(\text{Yield}}, \text{Ret-Val Unit}, h) \} \\
\text{else let } [hT] = \text{typo-of-addr } h \ a \\
in \text{if } P \vdash \text{ty-of } hT \leq \text{Class Thread then let } t_a = a2t a; \text{Class C = ty-of } hT \text{ in} \\
\text{if } M = \text{ start then} \\
\{(\text{Spawn } t_a (C, \text{run}, a) \ h, T\text{Start } t_a), \text{Ret-Val Unit}, h), \\
(\text{ThreadEx } t_a \ True), \text{Ret-sys-xcpt IllegalThreadState}, h) \} \\
\text{else if } M = \text{ join then} \\
\{(\text{Join } t_a, \text{IsIntrd } t \ False, T\text{Join } t_a), \text{Ret-Val Unit}, h), \\
(\text{IsIntrd } t \ True, \text{ClearIntr } t, T\text{Intrd } t), \\
\text{Ret-sys-xcpt InterruptedException}, h) \} \\
\text{else if } M = \text{ interrupt then} \\
\{(\text{ThreadEx } t_a \ True, \text{WakeUp } t_a, T\text{Intr } t_a), \text{Ret-Val Unit}, h), \\
(\text{ThreadEx } t_a \ False), \text{Ret-Val Unit}, h) \}
\]
else if $M = \text{isInterrupted}$ then
\[
\{(\langle \text{IsIntrd} t_a \text{ False}\rangle, \text{Ret-Val} (\text{Bool} \text{ False}), h) , \langle \langle \text{IsIntrd} t_a \text{ True}, \text{TIntrd} t_a\rangle, \text{Ret-Val} (\text{Bool} \text{ True}), h \rangle \}
\]
else \{ \{(\langle\rangle, \text{undefined}) \} \}
else \{ \{(\langle\rangle, \text{undefined}) \}) \}

B.4.4 Observability

unobservable native methods calls: $\tau_{\text{n}}: \text{cname} \Rightarrow \text{mname} \Rightarrow \text{bool}$

$\tau_{\text{n}}$ native Object hashcode
$\tau_{\text{n}}$ native Object currentThread

B.5 Generic well-formedness

sys-xcpts :: cname list
sys-xcpts = [NullPointer, ClassCast, ArithmeticException, OutOfMemory, ArrayIndexOutOfBoundsException, ArrayStore, NegativeArraySize, IllegalThreadState, IllegalMonitorState, InterruptedException]

wf-prog :: 'm wf-mdecl-test ⇒ 'm prog ⇒ bool
wf-prog $wf-md P$ $\leftarrow$ $wf-syscls P$ $\land$ $\text{distinct (map fst (classes P))}$ $\land$
\[
(\forall cd \in \text{set (classes P)}. \text{wf-cdecl} \, \text{wf-md} \, P \, cd) \]

wf-syscls :: 'm prog ⇒ bool
wf-syscls $P$ $\leftarrow$ $\text{is-class} P$ Object $\land$ $\text{is-class} P$ Throwable $\land$ $\text{is-class} P$ Thread $\land$
\[
(\forall C \in \text{set sys-xcpts}. P \vdash C \leq^* \text{Throwable}) \]

wf-cdecl :: 'm wf-mdecl-test ⇒ 'm prog ⇒ 'm cdecl ⇒ bool
wf-cdecl $P$ $\leftarrow$ $\text{wf-md} P$ $\langle C, D, f s, m s \rangle$ $\leftarrow$
\[
(\forall fd \in \text{set fs}. \text{wf-fdecl} P f d) \land \text{distinct (map fst fs)} \land
(\forall md \in \text{set ms}. \text{wf-mdecl} \text{wf-md} P C \, md) \land \text{distinct (map fst ms)} \land
(C \neq \text{Object} \rightarrow \text{is-class} P D \land P \vdash D \leq^* C \land
(\forall md \in \text{set ms}. \text{wf-overriding} P \, D \, md) \land
(C = \text{Thread} \rightarrow (\exists m. (\text{run}, [], \text{Void}, m) \in \text{set ms}))) \]

wf-fdecl :: 'm prog ⇒ fdecl ⇒ bool
wf-fdecl $P$ $\leftarrow$ $F, T, f m$ $\leftarrow$ $\text{is-type} P \, T$

wf-overriding :: 'm prog ⇒ cname ⇒ 'm mdecl ⇒ bool
wf-overriding $P \, D \langle M, Ts, T_r, m \rangle$ $\leftarrow$ $(\forall D' \, Ts' \, T'_r \, m')$
$P \vdash D \text{ sees } M:Ts' \rightarrow T'_r = m' \text{ in } D' \rightarrow P \vdash Ts' [\leq] Ts \land P \vdash T_r \leq T'_r)$
Appendix B. Formal definitions

\[ \text{wf-mdecl} :: \text{'m \_wf-mdecl-test} \Rightarrow \text{'m option \_wf-mdecl-test} \]
\[ \text{wf-mdecl} \text{\_wf-md} \text{\_P C} (\text{M, Ts, Tr, m}) \leftarrow \]
\[ \text{set Ts} \subseteq \text{types P} \land \text{is-type P} \text{ Tr} \land \]
\[ \begin{array}{l}
(\text{case m of Native} \Rightarrow \text{C.M(Ts)} :: \text{Tr} \\
| [\text{mb}] \Rightarrow \text{wf-md P C} (\text{M, Ts, Tr, mb}))
\end{array} \]

B.6 Source code

B.6.1 Syntax

datatype (B.6.1 Syntax)

\begin{align*}
\text{Cast ty ('a, 'b, 'addr) exp} &= \text{new cname | new ty[('a, 'b, 'addr) exp]} | \\
\text{Var 'a} &\quad \text{| 'a = 'addr val option; ('a, 'b, 'addr) exp}\]
\end{align*}

\[\text{type\_synonym } 'a \text{addr expr} = (\text{vname, unit, 'addr) exp} \]
\[\text{type\_synonym } 'a \text{addr J-mb} = \text{vname list} \times 'a \text{addr expr} \]
\[\text{type\_synonym } 'a \text{addr J-prog} = 'a \text{addr J-mb prog} \]

B.6.2 Typing rules for expressions

\[ \begin{array}{c}
\text{is-class P C} \quad \text{P, E \_ e :: Integer} \quad \text{is-type P (T[\_])} \\
\frac{\text{P, E \_ e :: Class C}}{\text{P, E \_ new C :: Class C}} \quad \frac{\text{P, E \_ new T[e] :: T[\_]}}{}
\end{array} \]
\[ \begin{array}{c}
\text{P, E \_ e :: T' \quad P + T \leq T' \lor P + T' \leq T} \quad \text{is-type P T} \\
\frac{\text{P, E \_ Cast T e :: T}}{}
\end{array} \]
\[ \begin{array}{c}
\text{P, E \_ e :: T' \quad P + T \leq T' \lor P + T' \leq T} \quad \text{is-type P T} \quad \text{is-ref T T} \\
\frac{\text{P, E \_ e instanceof T :: Boolean}}{}
\end{array} \]

328
### B.6. Source code

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td><code>typeof v = [T]</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ Val v :: T</code></td>
</tr>
<tr>
<td>E V</td>
<td><code>E V = [T]</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ Var V :: T</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: T_1</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_2 :: T_2</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ T_1 «bop» T_2 :: T</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e «bop» e_2 :: T</code></td>
</tr>
<tr>
<td></td>
<td><code>E V = [T]</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: T'</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ T' ≤ T</code></td>
</tr>
<tr>
<td></td>
<td><code>V ≠ this</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ V := e :: Void</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: T[]</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_1 :: T[]</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_2 :: T[]</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ T' ≤ T</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_1 := e_2 :: Void</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: T</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ class-of T = [C]</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ C sees F:T' (fm) in D</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e.F(D) :: T'</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: T[]</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_1 :: T_1</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ class-of T_1 = [C]</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ C sees F:T' (fm) in D</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_2 :: T_2</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ T_2 ≤ T'</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_1.F(D) := e_2 :: Void</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: T</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: T[]</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ C sees M:Ts→T_r = meth in D</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ Ts' [≤] T_s</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e.M(es) :: T_r</code></td>
</tr>
<tr>
<td></td>
<td><code>is-type P T</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E(V ⊸ T) ⊢ e :: T'</code></td>
</tr>
<tr>
<td></td>
<td>`case v of None ⇒ True</td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ {V: T = vο; e} :: T'</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_1 :: T_1</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_2 :: T_2</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_1;; e_2 :: T_2</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e :: Boolean</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_1 :: T_1</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ e_2 :: T_2</code></td>
</tr>
<tr>
<td></td>
<td><code>P ⊢ lub (T_1, T_2) = T</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ if (e) e_1 else e_2 :: T</code></td>
</tr>
<tr>
<td></td>
<td><code>P, E ⊢ while (e_1) e_2 :: Void</code></td>
</tr>
</tbody>
</table>
Appendix B. Formal definitions

\[
P, E \vdash e :: \text{Class } C \quad P \vdash C \leq^* \text{ Throwable} \\
P, E \vdash \text{throw } e :: \text{Void}
\]

\[
P, E \vdash e_1 :: T \quad P, E(V \mapsto \text{Class } C) \vdash e_2 :: T \quad P \vdash C \leq^* \text{ Throwable} \\
P, E \vdash \text{try } e_1 \ \text{catch} (C \ V) \ e_2 :: T
\]

\[
P, E \vdash e_1 :: T_1 \quad \text{is-ref } T_1 \quad T_1 \neq NT \quad P, E \vdash e_2 :: T_2 \\
P, E \vdash \text{sync} \ (e_1) \ e_2 :: T_2
\]

\[
P, E \vdash [] :: [] \\
P, E \vdash e :: T \quad P, E \vdash es :: Ts \\
P, E \vdash e \cdot es :: [T \cdot Ts]
\]

B.6.3 Definite Assignment

Hypersets: 

\[
\text{type} \ _\text{synonym} \ 'a \ \text{hyperset} = 'a \ \text{set option}
\]

\[
\begin{align*}
\text{None} \triangleleft B &= \text{None} \\
A \triangleleft \text{None} &= \text{None} \\
[A] \triangleleft [B] &= [A \cup B] \\
\text{None} \ominus a &= \text{None} \\
[A] \ominus a &= [A - \{a\}] \\
A \ominus B &= B \\
A \ominus \text{None} &= A \\
[A] \ominus [B] &= [A \cap B]
\end{align*}
\]

\[
\begin{align*}
\text{None} \ominus a &= \text{None} \\
A \ominus a &= [A - \{a\}] \\
a \in\in \text{None} &= \text{True} \\
a \in\in [A] &= a \in A
\end{align*}
\]

Definitely assigned variables:

\[
\begin{align*}
\mathcal{A} :: (a,'b,'addr) \ \text{exp} \Rightarrow 'a \ \text{hyperset} \\
\mathcal{A} (\text{new } C) &= [\emptyset] \\
\mathcal{A} (\text{new } T[e]) &= \mathcal{A} e \\
\mathcal{A} (e \ \text{instanceof } T) &= \mathcal{A} e \\
\mathcal{A} (\text{Cast } T e) &= \mathcal{A} e \\
\mathcal{A} (\text{Val } v) &= [\emptyset] \\
\mathcal{A} (V := e) &= \{V\} \cup \mathcal{A} e \\
\mathcal{A} (e_1[e_2]) &= \mathcal{A} e_1 \cup \mathcal{A} e_2 \\
\mathcal{A} (e_1[e_2] := e_3) &= \mathcal{A} e_1 \cup \mathcal{A} e_2 \cup \mathcal{A} e_3 \\
\mathcal{A} (e.\text{length}) &= \mathcal{A} e \\
\mathcal{A} (e.F[D]) &= \mathcal{A} e \\
\mathcal{A} (e_1.F[D] := e_2) &= \mathcal{A} e_1 \cup \mathcal{A} e_2 \\
\mathcal{A} (e.M(es)) &= \mathcal{A} e \cup \mathcal{A} es \\
\mathcal{A} [V : T = vo; e] &= \mathcal{A} e \ominus V \\
\mathcal{A} (e_1;; e_2) &= \mathcal{A} e_1 \cup \mathcal{A} e_2
\end{align*}
\]
\( A (\text{if } (e) \ e_1 \text{ else } e_2) = A e \sqcup (A e_1 \cap A e_2) \)
\( A (\text{while } (e_1) \ e_2) = A e_1 \)
\( A (\text{throw } e) = \text{None} \)
\( A (\text{try } e_1 \text{ catch } (C \ V) \ e_2) = A e_1 \cap (A e_2 \ominus V) \)
\( A (\text{sync}_b (e_1) \ e_2) = A e_1 \cup A e_2 \)
\( A (\text{insync}_b (a) \ e) = A e \)

\( A_s :: (a, b, \text{addr}) \ \text{exp list} \Rightarrow 'a \ \text{hyperset} \)
\( A_s [] = [\emptyset] \)
\( A_s (e \cdot es) = A e \sqcup A es \)

Definite assignment test:
\( D :: (a, b, \text{addr}) \ \text{exp} \Rightarrow 'a \ \text{hyperset} \Rightarrow \text{bool} \)
\( D (\text{new } C) A \leftrightarrow \text{True} \)
\( D (\text{new } T[e]) A \leftrightarrow D e A \)
\( D (\text{e instanceof } T) A \leftrightarrow D e A \)
\( D (\text{Cast } T \ e) A \leftrightarrow D e A \)
\( D (\text{Val } v) A \leftrightarrow \text{True} \)
\( D (e_1 \\text{bop } e_2) A \leftrightarrow D e_1 A \land D e_2 (A \sqcup A e_1) \)
\( D (\text{Var } V) A \leftrightarrow V \in e A \)
\( D (V := e) A \leftrightarrow D e A \)
\( D (e_1[e_2]) A \leftrightarrow D e_1 A \land D e_2 (A \sqcup A e_1) \)
\( D (e_1[e_2] := e_3) A \leftrightarrow (D e_1 \land D e_2 (A \sqcup A e_1) \land D e_3 (A \sqcup A e_1 \land A e_2)) \land D e A \)
\( D (e.\text{length}) A \leftrightarrow D e A \)
\( D (e.F[D]) A \leftrightarrow D e A \)
\( D (e_1.F[D] := e_2) A \leftrightarrow D e_1 A \land D e_2 (A \sqcup A e_1) \)
\( D (e.M(es)) A \leftrightarrow D e A \land D \ e s (A \sqcup A e) \)
\( D [V: T = \text{vo}; e] \leftrightarrow (\text{if vo = None then } D e (A \ominus V) \text{ else } D e (A \cup [V]))) \)
\( D (e_1; e_2) A \leftrightarrow D e_1 A \land D e_2 (A \cup A e_1) \)
\( D (\text{if } (e) \ e_1 \text{ else } e_2) A \leftrightarrow D e A \land D e_1 (A \cup A e) \land D e_2 (A \cup A e) \)
\( D (\text{while } (e_1) \ e_2) A \leftrightarrow D e_1 A \land D e_2 (A \cup A e_1) \)
\( D (\text{throw } e) A \leftrightarrow D e A \)
\( D (\text{try } e_1 \text{ catch } (C \ V) \ e_2) A \leftrightarrow D e_1 A \land D e_2 (A \cup [V]) \)
\( D (\text{sync}_b (e_1) \ e_2) A \leftrightarrow D e_1 A \land D e_2 (A \cup A e_1) \)
\( D (\text{insync}_b (a) \ e) A \leftrightarrow D e A \)

\( Ds :: (a, b, \text{addr}) \ \text{exp list} \Rightarrow 'a \ \text{hyperset} \Rightarrow \text{bool} \)
\( Ds [] \leftrightarrow \text{True} \)
\( Ds (e \cdot es) A \leftrightarrow D e A \land D \ e s (A \sqcup A e) \)
B.6.4 Well-formedness

\[ \text{wf-J-mdecl} :: 'addr J-mb \text{wf-mdecl-test} \]
\[ \text{wf-J-mdecl} P C (M, Ts, Tr, (pns, body)) \leftrightarrow \]
\[ |Ts| = |pns| \land \text{distinct pns} \land \text{this} \notin \text{set pns} \land \]
\[ (\exists T. P, \{\text{this}\mapsto \text{Class} C, pns [\mapsto] Ts \vdash \text{body} :: T \land P \vdash T \leq Tr) \land \]
\[ \mathcal{D} \text{body} \left[ \{ \text{this} \} \cup \text{set pns} \right] \]

\[ \text{wf-J-prog} = \text{wf-prog} \text{wf-J-mdecl} \]

B.6.5 Small-step semantics

Subexpression reduction rules

\[
P, t \vdash \langle e, s \rangle -ta\rightarrow \langle e', s' \rangle
\]

\[
P, t \vdash \langle \text{new} T[e], s \rangle -ta\rightarrow \langle \text{new} T[e'], s' \rangle
\]

\[
P, t \vdash \langle e, s \rangle -ta\rightarrow \langle e', s' \rangle
\]

\[
P, t \vdash \langle \text{Cast} T e, s \rangle -ta\rightarrow \langle \text{Cast} T e', s' \rangle
\]

\[
P, t \vdash \langle e \text{ instanceof} T, s \rangle -ta\rightarrow \langle e' \text{ instanceof} T, s' \rangle
\]

\[
P, t \vdash \langle e_1, s \rangle -ta\rightarrow \langle e'_1, s' \rangle
\]

\[
P, t \vdash \langle e_1 \text{ «bop} e_2, s \rangle -ta\rightarrow \langle e'_1 \text{ «bop} e_2, s' \rangle
\]

\[
P, t \vdash \langle e_2, s \rangle -ta\rightarrow \langle e'_2, s' \rangle
\]

\[
P, t \vdash \langle \text{Val} v_1 \text{ «bop} e_2, s \rangle -ta\rightarrow \langle \text{Val} v_1 \text{ «bop} e'_2, s' \rangle
\]

\[
P, t \vdash \langle e, s \rangle -ta\rightarrow \langle e', s' \rangle
\]

\[
P, t \vdash \langle e_1, s \rangle -ta\rightarrow \langle e'_1, s' \rangle
\]

\[
P, t \vdash \langle e_1[e_2], s \rangle -ta\rightarrow \langle e'_1[e_2], s' \rangle
\]

\[
P, t \vdash \langle e_2, s \rangle -ta\rightarrow \langle e'_2, s' \rangle
\]

\[
P, t \vdash \langle \text{Val} v_1[e_2], s \rangle -ta\rightarrow \langle \text{Val} v_1[e'_2], s' \rangle
\]

\[
P, t \vdash \langle e_1, s \rangle -ta\rightarrow \langle e'_1, s' \rangle
\]

\[
P, t \vdash \langle e_1[e_2] := e_3, s \rangle -ta\rightarrow \langle e'_1[e_2] := e_3, s' \rangle
\]
\[ P, t \vdash \langle e_2, s \rangle \rightarrow (\langle e'_2, s' \rangle) \]

\[ P, t \vdash \langle \text{Val} \ v_1[e_2] := e_3, s \rangle \rightarrow (\langle \text{Val} \ v_1[e'_2] := e_3, s' \rangle) \]

\[ P, t \vdash \langle e_3, s \rangle \rightarrow (\langle e'_3, s' \rangle) \]

\[ P, t \vdash \langle \text{Val} \ v_1[\text{Val} \ v_2] := e_3, s \rangle \rightarrow (\langle \text{Val} \ v_1[\text{Val} \ v_2] := e'_3, s' \rangle) \]

\[ P, t \vdash \langle e, s \rangle \rightarrow (\langle e', s' \rangle) \]

\[ P, t \vdash \langle e, F[D], s \rangle \rightarrow (\langle e', F[D], s' \rangle) \]

\[ P, t \vdash \langle e, \langle s \rangle \rightarrow (\langle e', s' \rangle) \]

\[ P, t \vdash \langle e, M(es), s \rangle \rightarrow (\langle e', M(es), s' \rangle) \]

\[ P, t \vdash \langle \text{Val} \ v.M(es), s \rangle \rightarrow (\langle \text{Val} \ v.M(es'), s' \rangle) \]

\[ P, t \vdash \langle e, \langle h, xs(V := vo) \rangle \rangle \rightarrow (\langle e', \langle h', xs' \rangle \rangle) \]

\[ P, t \vdash \langle \langle V : T = vo; e \rangle, \langle (h, xs) \rangle \rangle \rightarrow (\langle (V : T = xs' V; e'), \langle (h', xs'(V := xs V)) \rangle \rangle) \]

\[ P, t \vdash \langle e_1, s \rangle \rightarrow (\langle e'_1, s' \rangle) \]

\[ P, t \vdash \langle e_1; e_2, s \rangle \rightarrow (\langle e'_1; e_2, s' \rangle) \]

\[ P, t \vdash \langle if \ e_1 \ else \ e_2, s \rangle \rightarrow (\langle if \ e' \ e_1 \ else \ e_2, s' \rangle) \]

\[ P, t \vdash \langle e, s \rangle \rightarrow (\langle e', s' \rangle) \]

\[ P, t \vdash \langle \text{throw} \ e, s \rangle \rightarrow (\langle \text{throw} \ e', s' \rangle) \]

\[ P, t \vdash \langle \text{try} \ e_1 \ \text{catch} \ (C \ V) \ e_2, s \rangle \rightarrow (\langle \text{try} \ e'_1 \ \text{catch} \ (C \ V) \ e_2, s' \rangle) \]

\[ P, t \vdash \langle \text{sync} \ (e_1) \ e_2, s \rangle \rightarrow (\langle \text{sync} \ (e'_1) \ e_2, s' \rangle) \]
Appendix B. Formal definitions

\[ P, t \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \]
\[ P, t \vdash \langle \text{insync} (a) e, s \rangle \rightarrow \langle \text{insync} (a) e', s' \rangle \]
\[ P, t \vdash \langle e, s \rangle \rightarrow \langle e', s' \rangle \]
\[ P, t \vdash \langle e \cdot es, s \rangle \rightarrow \langle e' \cdot es', s' \rangle \]
\[ P, t \vdash \langle \text{Val} v \cdot es, s \rangle \rightarrow \langle \text{Val} v \cdot es', s' \rangle \]

Expression reduction rules

\[
\begin{align*}
\text{alloc } h \ (\text{ClassT } C) &= (h', [a]) \\
P, t \vdash \langle \text{new } C, (h, xs) \rangle \rightarrow \langle \text{New-Obj } a \ C \rightarrow \text{addr } a, (h', xs) \rangle
\end{align*}
\]

\[
\begin{align*}
\text{alloc } h \ (\text{ClassT } C) &= (h', \text{None}) \\
P, t \vdash \langle \text{new } C, (h, xs) \rangle \rightarrow \langle \text{THROW OutOfMemory}, (h', xs) \rangle
\end{align*}
\]

\[
\begin{align*}
i < s 0 \\
P, t \vdash \langle \text{new } [\text{Val } \text{Intg } i], s \rangle \rightarrow \langle \text{THROW NegativeArraySize}, s \rangle
\end{align*}
\]

\[
\begin{align*}
0 \leq s i \\
P, t \vdash \langle \text{new } [\text{Val } \text{Intg } i], (h, xs) \rangle \rightarrow \langle \text{THROW OutOfMemory}, (h', xs) \rangle
\end{align*}
\]

\[
\begin{align*}
\text{typeof}_{hp} v = [T'] \quad P \vdash T' \leq T \\
P, t \vdash \langle \text{Cast } T \ (\text{Val } v), s \rangle \rightarrow \langle \text{Val } v, s \rangle
\end{align*}
\]

\[
\begin{align*}
\text{typeof}_{hp} v = [T'] \quad \neg P \vdash T' \leq T \\
P, t \vdash \langle \text{Cast } T \ (\text{Val } v), s \rangle \rightarrow \langle \text{THROW ClassCast}, s \rangle
\end{align*}
\]

\[
\begin{align*}
\text{typeof}_{hp} v = [T'] \quad b \leftrightarrow v \neq \text{Null} \wedge P \vdash T' \leq T \\
P, t \vdash \langle \text{Val } v \ \text{instanceof} T, s \rangle \rightarrow \langle \text{Val } (\text{Bool } b), s \rangle
\end{align*}
\]

\[
\begin{align*}
\text{binop } bop \ v_1 \ v_2 = [\text{Inl } v] \\
P, t \vdash \langle \text{Val } v_1 \ \text{«}bop\text{» } \text{Val } v_2, s \rangle \rightarrow \langle \text{Val } v, s \rangle
\end{align*}
\]

\[
\begin{align*}
\text{binop } bop \ v_1 \ v_2 = [\text{Inr } a] \\
P, t \vdash \langle \text{Val } v_1 \ \text{«}bop\text{» } \text{Val } v_2, s \rangle \rightarrow \langle \text{Throw } a, s \rangle
\end{align*}
\]
\[
\text{slice } V = [v] \\
P, t \vdash \text{Val } v, (h, x) \to \text{unit } (h, x) \to (V \to v)) \\
P, t \vdash \text{null[Val } v, s \to \text{THROW NullPointer, } s) \\
\text{typeof-addr } (h s) a = [\text{Array } T n] \\
0 \leq s \quad \text{sint } i < \text{int } n \\
P, t \vdash \text{Val } v_1, \text{Val } v_2 \to \text{THROW ArrayIndexOutOfBounds, } s) \\
\text{typeof-addr } h a = [\text{Array } T n] \\
0 \leq s \quad \text{sint } i \geq \text{int } n \\
P, t \vdash \text{Val } v_1, \text{Val } v_2 \to \text{THROW NullPointer, } s) \\
\text{typeof-addr } (h s) a = [\text{Array } T n] \\
0 \leq s \quad \text{sint } i < \text{int } n \\
P, t \vdash \text{Val } v_1, \text{Val } v_2 \to \text{THROW ArrayIndexOutOfBounds, } s) \\
\text{typeof-addr } h a = [\text{Array } T n] \\
0 \leq s \quad \text{sint } i < \text{int } n \\
P, t \vdash \text{null } \text{null } v_3 \to \text{THROW NullPointer, } s) \\
\text{typeof-addr } (h s) a = [\text{Array } T n] \\
P, t \vdash \text{null } \text{null } v_2 \to \text{THROW NullPointer, } s) \\
\text{read } h a (\text{Field } F D) v \\
P, t \vdash \text{null } \text{null } v_2 \to \text{THROW NullPointer, } s)
Appendix B. Formal definitions

\[
\text{write } h a \ (\text{Field } F D) \ v_2 h' \\
\]
\[
P, t \vdash \langle addr . F[D] := Val v_2, (h, xs) \rangle -\langle \text{Write } a \ (\text{Field } F D) \ v_2 \rangle \rightarrow \langle \text{unit}, (h', xs) \rangle
\]
\[
P, t \vdash \langle \text{null}.M(es), s \rangle \rightarrow \langle \text{throw } \text{NullPointer}, s \rangle
\]
\[
\text{typeof-addr } (hp \ s) a = [hT] \quad P \vdash \text{class-of' } hT \text{ sees } M:T_s \rightarrow T_r = [(\text{pns}, body)] \text{ in } D
\]
\[
|\text{vs}| = |\text{pns}| \quad |T_s| = |\text{pns}|
\]
\[
P, t \vdash \langle \text{addr } a.M(\text{map } Val \ vs), s \rangle -\langle () \rightarrow \langle \text{blocks } \text{this} \cdot \text{pns} \rangle (\text{Class } D \cdot T_s) (\text{Addr } a \cdot vs) \text{ body}, s \rangle
\]
\[
\text{typeof-addr } h a = [hT] \quad P \vdash \text{class-of' } hT \text{ sees } M:T_s \rightarrow T_r = \text{Native} \text{ in } D
\]
\[
|\text{vs}| = |\text{pns}|
\]
\[
\text{ta' = native-TA2J } P \text{ ta} \quad e' = \text{native-Ret2J } (\text{addr } a.M(\text{map } Val \ vs)) \langle vs \rangle
\]
\[
P, t \vdash \langle \text{addr } a.M(\text{map } Val \ vs), (h, xs) \rangle -\text{ta' } \rightarrow \langle e', (h', xs) \rangle
\]
\[
P, t \vdash \langle V: T = v_0; \text{Val } v, s \rangle -\langle () \rightarrow \langle \text{Val } v, s \rangle
\]
\[
P, t \vdash \langle \text{if } (\text{true}) \ e_1 \ \text{else } e_2, s \rangle -\langle () \rightarrow \langle e_2, s \rangle
\]
\[
P, t \vdash \langle \text{if } (\text{false}) \ e_1 \ \text{else } e_2, s \rangle -\langle () \rightarrow \langle e_2, s \rangle
\]
\[
P, t \vdash \langle \text{while } (e_1) \ e_2, s \rangle -\langle () \rightarrow \langle (e_1) \ e_2; \ \text{while } (e_1) \ e_2 \ \text{else } \text{unit}, s \rangle
\]
\[
P, t \vdash \langle \text{throw } \text{null}, s \rangle -\langle () \rightarrow \langle \text{throw } \text{NullPointer}, s \rangle
\]
\[
P, t \vdash \langle \text{try } \text{Val } v \ \text{catch}(C \ V) \ e_2, s \rangle -\langle () \rightarrow \langle \text{Val } v, s \rangle
\]
\[
\text{typeof-addr } (hp \ s) a = [\text{Class}T \ D] \quad P \vdash D \leq^* C
\]
\[
P, t \vdash \langle \text{try } \text{Throw } a \ \text{catch}(C \ V) \ e_2, s \rangle -\langle () \rightarrow \langle (V: \text{Class } C = [a]; e_2), s \rangle
\]
\[
\text{typeof-addr } (hp \ s) a = [\text{Class}T \ D] \quad \neg P \vdash D \leq^* C
\]
\[
P, t \vdash \langle \text{try } \text{Throw } a \ \text{catch}(C \ V) \ e_2, s \rangle -\langle () \rightarrow \langle \text{Throw } a, s \rangle
\]
\[
P, t \vdash \langle \text{sync } \text{null} \ e_2, s \rangle -\langle () \rightarrow \langle \text{throw } \text{NullPointer}, s \rangle
\]
\[
P, t \vdash \langle \text{sync } \text{addr } a \ e_2, s \rangle -\langle \text{Lock} \rightarrow a, \text{SLock } a \rangle -\langle \text{insync } a \ e_2, s \rangle
\]
\[
P, t \vdash \langle \text{insync } a \ (\text{Val } v), s \rangle -\langle \text{Unlock} \rightarrow a, \text{SUnlock } a \rangle -\langle \text{Val } v, s \rangle
\]
Exception propagation rules

\[ P, t \vdash \langle \text{new } T[\text{Throw } a], s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Cast } T \ (\text{Throw } a), s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{(Throw } a) \text{ instanceof } T, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a \ bop \ e_2, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Val } v_1 \ bop \ \text{Throw } a, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle V := \text{Throw } a, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a[e_2], s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Val } v_1[\text{Throw } a], s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a[e_2] := e_3, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Val } v_1[\text{Val } v_2] := \text{Throw } a, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a.\text{length}, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a.F\{D\}, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a.F\{D\} := e_2, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Val } v_1.F\{D\} := \text{Throw } a, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a.M(es), s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Val } v.M(\text{map } \text{Val } vs @ \text{Throw } a \cdot es), s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle V : T = \text{vo}; \text{Throw } a, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{Throw } a; \ e_2, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{if } (\text{Throw } a) \ e_1 \ \text{else } e_2, s \rangle \sim \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{throw } (\text{Throw } a), s \rangle \sim \langle \text{Throw } a, s \rangle \]
Appendix B. Formal definitions

\[ P, t \vdash \langle \text{sync } (\text{Throw } a) \ e, s \rangle - \langle \text{Throw } a, s \rangle \]

\[ P, t \vdash \langle \text{insync } (a) \ (\text{Throw } a'), s \rangle - \langle \text{Unlock} \rightarrow a, \text{SUnlock} \rightarrow a \rangle \rightarrow \langle \text{Throw } a', s \rangle \]

auxiliary functions

\[ \text{blocks } [\cdot] [\cdot] [\cdot] [\cdot] e = e \]
\[ \text{blocks } (V \cdot Vs) (T \cdot Ts) (v \cdot vs) e = \{ V : T = [v] ; \text{blocks } Vs Ts vs e \} \]

B.6.6 Observability

\( \tau \text{-move} :: 'm \text{ prog } \Rightarrow ' \text{heap } \Rightarrow ' \text{addr } \text{expr } \Rightarrow \text{bool} \)
\( \tau \text{-move } P h \ (\text{new } C) \quad \Rightarrow \text{False} \)
\( \tau \text{-move } P h \ (\text{new } T[e]) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \)
\( \tau \text{-move } P h \ (\text{Cast } T e) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \vee \text{is-Val } e \)
\( \tau \text{-move } P h \ (e \text{ instanceof } T) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \vee \text{is-Val } e \)
\( \tau \text{-move } P h \ (V (v := e)) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \vee \text{is-Val } e \)
\( \tau \text{-move } P h \ (e_1 [e_2]) \quad \Rightarrow \tau \text{-move } P h e_1 \vee \text{is-Throw } e_1 \vee \text{is-Val } e_1 \)
\( \tau \text{-move } P h \ (e_1 [e_2] := e_3) \quad \Rightarrow \tau \text{-move } P h e_1 \vee \text{is-Throw } e_1 \vee \text{is-Val } e_1 \vee (\tau \text{-move } P h e_2 \vee \text{is-Throw } e_2 \vee \text{is-Val } e_2) \)
\( \tau \text{-move } P h \ (e.\text{length}) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \)
\( \tau \text{-move } P h \ (e.F[D]) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \)
\( \tau \text{-move } P h \ (e_1.F[D] := e_2) \quad \Rightarrow \tau \text{-move } P h e_1 \vee \text{is-Throw } e_1 \vee \text{is-Val } e_1 \vee (\tau \text{-move } P h e_2 \vee \text{is-Throw } e_2) \)
\( \tau \text{-move } P h \ (e.M(vs)) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \vee (\exists v : \text{Val } v \land (\tau \text{-moves } P h e s \vee \text{is-Throws } es \vee \text{is-Vals } es \land (v = \text{Null} \lor (\forall T C Ts T_r D. \text{typeof}_h v = |T| \rightarrow \text{class-of } T = |C| \rightarrow P + C \text{ sees } M:T s \rightarrow Tr = \text{Native in } D \rightarrow \tau \text{native } D M))) ) \)
\( \tau \text{-move } P h \ (V : T = \text{null} ; e) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \vee \text{is-Val } e \)
\( \tau \text{-move } P h \ (\text{sync } (e_1) e_2) \quad \Rightarrow \tau \text{-move } P h e_1 \vee \text{is-Throw } e_1 \)
\( \tau \text{-move } P h \ (\text{insync } (a) e) \quad \Rightarrow \tau \text{-move } P h e \)
\( \tau \text{-move } P h \ (e_1 ; e_2) \quad \Rightarrow \tau \text{-move } P h e_1 \vee \text{is-Throw } e_1 \vee \text{is-Val } e_1 \)
\( \tau \text{-move } P h \ (\text{if } (e_1) e_2 \text{ else } e_3) \quad \Rightarrow \tau \text{-move } P h e_1 \vee \text{is-Throw } e_1 \vee \text{is-Val } e_1 \)
\( \tau \text{-move } P h \ (\text{while } (e_1) e_2) \quad \Rightarrow \text{True} \)
\( \tau \text{-move } P h \ (\text{throw } e) \quad \Rightarrow \tau \text{-move } P h e \vee \text{is-Throw } e \vee e = \text{null} \)
\( \tau \text{-move } P h \ (\text{try } e_1 \text{ catch}(C V) e_2) \quad \Rightarrow \tau \text{-move } P h e_1 \vee \text{is-Throw } e_1 \vee \text{is-Val } e_1 \)
τ-moves :: 'm prog ⇒ 'heap ⇒ 'addr expr list ⇒ bool
τ-moves P h [] ←→ False
τ-moves P h e · es ←→ τ-move P h e ∨ is-Val e ∧ τ-moves P h es

J-τ-move P ((e, xs), h) ta_ ←→ τ-move P h e ∧ ta = (||)

auxiliary functions: is-Throw e ←→ (∃a. e = Throw a)

B.7 Bytecode

B.7.1 Syntax

Instructions

datatype 'addr instr = Load nat | Store nat | Push 'addr val | Pop | Dup | Swap | BinOp bop | New cname | NewArray ty | ALoad | AStore | ALength | Getfield vname cname | Putfield vname cname | Checkcast ty | Instanceof ty | Invoke mname nat | Return | Goto int | IfFalse int | ThrowExc | MEnter | MExit

Program declarations

type_synonym 'addr jvm-method = nat × nat × 'addr instr list × ex-table
type_synonym 'addr jvm-prog = 'addr jvm-method prog
type_synonym ex-table = ex-entry list
type_synonym ex-entry = pc × pc × cname option × pc × nat

B.7.2 Applicability and effect

datatype 'a err = Err | OK 'a
type_synonym ty_1 = ty list × ty err list

ok-val (OK T) = T
the-Array (T[]) = T

Effect

successor instructions under normal execution:
succs :: 'addr instr ⇒ ty_1 ⇒ pc ⇒ pc list
succs (Load i ) τ pc = [pc + 1]
succs (Store i ) τ pc = [pc + 1]
succs (Push v) τ pc = [pc + 1]
Appendix B. Formal definitions

sucses $\tau$ $pc = [pc + 1]$
sucses $\text{Dup}$ $\tau$ $pc = [pc + 1]$
sucses $\text{Swap}$ $\tau$ $pc = [pc + 1]$
sucses $\text{(BinOp } bop\text{)}$ $\tau$ $pc = [pc + 1]$
sucses $\text{(New C)}$ $\tau$ $pc = [pc + 1]$
sucses $\text{(NewArray } T\text{)}$ $\tau$ $pc = [pc + 1]$
sucses $\text{ALoad}$ $\tau$ $pc = (if (fst \tau)_{[1]} = NT \text{ then } [] \text{ else } [pc + 1])$
sucses $\text{AStore}$ $\tau$ $pc = (if (fst \tau)_{[2]} = NT \text{ then } [] \text{ else } [pc + 1])$
sucses $\text{ALength}$ $\tau$ $pc = (if (fst \tau)_{[0]} = NT \text{ then } [] \text{ else } [pc + 1])$
sucses $\text{(Getfield } F \text{ D)}$ $\tau$ $pc = [pc + 1]$
sucses $\text{(Putfield } F \text{ D)}$ $\tau$ $pc = [pc + 1]$
sucses $\text{(Checkcast } T\text{)}$ $\tau$ $pc = [pc + 1]$
sucses $\text{(Instanceof } T\text{)}$ $\tau$ $pc = [pc + 1]$
sucses $\text{(Invoke } M \text{ n)}$ $\tau$ $pc = (if (fst \tau)_{[n]} = NT \text{ then } [] \text{ else } [pc + 1])$
sucses $\text{Return}$ $\tau$ $pc = []$
sucses $\text{(Goto } i\text{)}$ $\tau$ $pc = [\text{nat (int } pc + i\text{)}]$
sucses $\text{(IfFalse } i\text{)}$ $\tau$ $pc = [pc + 1, \text{nat (int } pc + i\text{)}]$
sucses $\text{ThrowExc}$ $\tau$ $pc = []$
sucses $\text{MEnter}$ $\tau$ $pc = (if (fst \tau)_{[0]} = NT \text{ then } [] \text{ else } [pc + 1])$
sucses $\text{MExit}$ $\tau$ $pc = (if (fst \tau)_{[0]} = NT \text{ then } [] \text{ else } [pc + 1])$

**effect of instructions:**

eff_{\text{i}} :: \text{`addr instr \times m prog \times ty}_{i} \Rightarrow \text{ty}_{i}

eff_{\text{i}} (\text{Load } i, p, ST, LT) = ( ok-val LT[i] \cdot ST, LT)

eff_{\text{i}} (\text{Store } i, p, T \cdot ST, LT) = (ST, LT[i := \text{OK } T])

eff_{\text{i}} (\text{Push } v, p, ST, LT) = (\text{the (typeof } v) \cdot ST, LT)

eff_{\text{i}} (\text{Pop } v, p, T \cdot ST, LT) = (ST, LT)

eff_{\text{i}} (\text{Dup } p, T \cdot ST, LT) = (T \cdot T \cdot ST, LT)

eff_{\text{i}} (\text{Swap } p, T_{1} \cdot T_{2} \cdot ST, LT) = (T_{2} \cdot T_{1} \cdot ST, LT)

eff_{\text{i}} (\text{BinOp } bop, p, T_{1} \cdot T_{2} \cdot ST, LT) = (((T_{1} \cdot p + T_{1} \langle bop \rangle T_{2} :: T) \cdot ST, LT)

eff_{\text{i}} (\text{New } C, p, ST, LT) = (\text{Class } C \cdot ST, LT)

eff_{\text{i}} (\text{NewArray } T, p, T' \cdot ST, LT) = (T[] \cdot ST, LT)

eff_{\text{i}} (\text{ALoad } p, T_{1} \cdot T_{2} \cdot ST, LT) = (\text{the-Array } T_{2} \cdot ST, LT)

eff_{\text{i}} (\text{AStore } p, T_{1} \cdot T_{2} \cdot T_{3} \cdot ST, LT) = (ST, LT)

eff_{\text{i}} (\text{ALength } p, T \cdot ST, LT) = (\text{Integer } ST, LT)

eff_{\text{i}} (\text{Getfield } F \text{ D}, p, T \cdot ST, LT) = (\text{fst (snd (field } P \text{ D} F)) \cdot ST, LT)

eff_{\text{i}} (\text{Putfield } F \text{ D}, p, T_{1} \cdot T_{2} \cdot ST, LT) = (ST, LT)

eff_{\text{i}} (\text{Checkcast } T, p, T' \cdot ST, LT) = (T \cdot ST, LT)

eff_{\text{i}} (\text{Instanceof } T, p, T' \cdot ST, LT) = (\text{Boolean } ST, LT)
B.7. Bytecode

\[ \text{eff}_i (\text{Invoke} M n, P, ST, LT) = \]
\[ (\text{let } [C] = \text{class-of } ST[n]; (\_, \_, T_r, \_) = \text{method } P C M \]
\[ \text{in } (T_r \cdot \text{drop } (n + 1) ST, LT)) \]
\[ \text{eff}_i (\text{Goto} i, P, ST, LT) = (ST, LT) \]
\[ \text{eff}_i (\text{IfFalse} i, P, T \cdot ST, LT) = (ST, LT) \]
\[ \text{eff}_i (\text{MEnter} i, P, T \cdot ST, LT) = (ST, LT) \]
\[ \text{eff}_i (\text{MExit} i, P, T \cdot ST, LT) = (ST, LT) \]

relevant classes for exception handlers

binop-relevant-class :: bop \Rightarrow 'm prog \Rightarrow \text{cname} \Rightarrow \text{bool}

binop-relevant-class / P C \leftarrow P \vdash \text{ArithmeticException} \leq^* C

binop-relevant-class % P C \leftarrow P \vdash \text{ArithmeticException} \leq^* C

binop-relevant-class _ P C \leftarrow \text{False}

is-relevant-class :: 'addr instr \Rightarrow 'm prog \Rightarrow \text{cname} \Rightarrow \text{bool}

is-relevant-class (BinOp bop) P C \leftarrow \text{binop-relevant-class } P C

is-relevant-class (Getfield F D) P C \leftarrow P \vdash \text{NullPointer} \leq^* C

is-relevant-class (Putfield F D) P C \leftarrow P \vdash \text{NullPointer} \leq^* C

is-relevant-class (Checkcast T) P C \leftarrow P \vdash \text{ClassCast} \leq^* C

is-relevant-class (New C) P C \leftarrow P \vdash \text{OutOfMemory} \leq^* C

is-relevant-class ThrowExc P C \leftarrow \text{True}

is-relevant-class Invoke M n P C \leftarrow \text{True}

is-relevant-class (NewArray T) P C \leftarrow
P \vdash \text{OutOfMemory} \leq^* C \lor P \vdash \text{NegativeArraySize} \leq^* C

is-relevant-class ALoad P C \leftarrow P \vdash \text{ArrayIndexOutOfBoundsException} \leq^* C \lor P \vdash \text{NullPointer} \leq^* C

is-relevant-class astore P C \leftarrow P \vdash \text{ArrayIndexOutOfBoundsException} \leq^* C \lor
P \vdash \text{ArrayStore} \leq^* C \lor P \vdash \text{NullPointer} \leq^* C

is-relevant-class ALength P C \leftarrow P \vdash \text{NullPointer} \leq^* C

is-relevant-class MEnter P C \leftarrow
P \vdash \text{IllegalMonitorStateException} \leq^* C \lor P \vdash \text{NullPointer} \leq^* C

is-relevant-class MExit P C \leftarrow
P \vdash \text{IllegalMonitorStateException} \leq^* C \lor P \vdash \text{NullPointer} \leq^* C

is-relevant-class _ P C \leftarrow \text{False}

is-relevant-entry :: 'm prog \Rightarrow 'addr instr \Rightarrow \text{pc} \Rightarrow \text{ex-entry} \Rightarrow \text{bool}

is-relevant-entry P i pc (f, t, Co, h, d) \leftarrow
\text{(case } Co \text{ of Any } \Rightarrow \text{True} | [C] \Rightarrow \text{is-relevant-class } i P C) \land f \leq pc \land pc < t

relevant-entries :: 'm prog \Rightarrow 'addr instr \Rightarrow \text{pc} \Rightarrow \text{ex-table} \Rightarrow \text{ex-table}

relevant-entries P i pc = \text{filter (is-relevant-entry } P i pc)
Appendix B. Formal definitions

xcpt-class Any = Class Throwable
xcpt-class [C] = Class C

xcpt-eff :: 'addr instr ⇒ 'm prog ⇒ pc ⇒ tyi ⇒ ex-table ⇒ (pc × tyi option) list
xcpt-eff i P pc (ST, LT) xt =
  map (λ(f, t, Co, h, d). (hi ((xcpt-class Co ∘ drop (|ST| − d) ST, LT))))
    (relevant-entries P i pc)

norm-eff :: 'addr instr ⇒ 'm prog ⇒ pc ⇒ tyi ⇒ (pc × tyi option) list
norm-eff i P pc τ = map (λpc'. (pc' ∘ [eff i P i τ])) (succ i τ pc)

eff :: 'addr instr ⇒ 'm prog ⇒ pc ⇒ ex-table ⇒ tyi option ⇒ (pc × tyi option) list
eff i P pc xt None = []

Applicability

app_i :: 'addr instr × 'm prog × pc × nat × ty × tyi ⇒ bool
app_i (Load i, P, pc, msl, Tr, ST, LT)
  ⋆ι < |LT| ∧ LT[i] ≠ Err ∧ |ST| < msl
app_i (Store i, P, pc, msl, Tr, T ∗ ST, LT)
  ⋆i < |LT|
app_i (Push v, P, pc, msl, Tr, ST, LT)
  ⋆|ST| < msl ∧ typeof v ≠ None
app_i (Pop, P, pc, msl, Tr, T ∗ ST, LT)
  ⋆True
app_i (Dup, P, pc, msl, Tr, T ∗ ST, LT)
  ⋆|ST| + 1 < msl
app_i (Swap, P, pc, msl, Tr, T1 ∗ T2 ∗ ST, LT)
  ⋆True
app_i (BinOp hop, P, pc, msl, Tr, T1 ∗ T2 ∗ ST, LT)
  ⋆(∃T. P ⊢ T1 «hop» T2 :: T)
app_i (New C, P, pc, msl, Tr, ST, LT)
  ⋆is-class P C ∧ |ST| < msl
app_i (NewArray T, P, pc, msl, Tr, Integer ∗ ST, LT)
  ⋆is-type P (T[]) universe T
app_i (ALoad, P, pc, msl, Tr, Integer ∗ ST, LT)
  ⋆T ≠ NT → is-Array T
app_i (AStore, P, pc, msl, Tr, T1 ∗ T2 ∗ T3 ∗ ST, LT)
  ⋆T2 = Integer ∧ (T3 ≠ NT → is-Array T3)
app_i (ALength, P, pc, msl, Tr, T ∗ ST, LT)
  ⋆T = NT ∨ is-Array T
app_i (Getfield F C, P, pc, msl, Tr, T ∗ ST, LT)
  ⋆(∃f) f. m. P ⊢ C sees F:T (fm) in C ∧ P ⊢ T ≤ Class C
app_i (Putfield F C, P, pc, msl, Tr, T1 ∗ T2 ∗ ST, LT)
  ⋆(∃f) f. m. P ⊢ C sees F:T (fm) in C ∧ P ⊢ T2 ≤ Class C ∧ P ⊢ T1 ≤ Tf
app_i (Checkcast T, P, pc, msl, Tr, T' ∗ ST, LT)
  ⋆is-type P T
app_i (Instanceof T, P, pc, msl, Tr, T' ∗ ST, LT)
  ⋆is-type P T ∧ is-refTT'
app_i (Invoke M n, P, pc, msl, Tr, ST, LT)

  n < |ST| ∧ (ST[n] ≠ NT → (∃C D Ts T m. class-of ST[n] = [C] ∧ P ⊢ C sees M:T ∗ T = m in D ∧ P ⊢ rev (take n ST) [≤] Ts))
B.7. Bytecode

Semantics of instructions

exec-instr (Load n) P t h stk loc C M pc frs =
\{ (\langle \rangle, (None, h, (loc[n] \cdot stk, loc, C, M, pc + 1) \cdot frs)) \}\n
exec-instr (Store n) P t h stk loc C M pc frs =
\{ (\langle \rangle, (None, h, (tl stk, loc[n := hd stk], C, M, pc + 1) \cdot frs)) \}\n
exec-instr (Push v) P t h stk loc C M pc frs =
\{ (\langle \rangle, (None, h, (v \cdot stk, loc, C, M, pc + 1) \cdot frs)) \}\n
exec-instr Pop P t h stk loc C M pc frs =
\{ (\langle \rangle, (None, h, (tl stk, loc, C, M, pc + 1) \cdot frs)) \}\n
exec-instr Dup P t h stk loc C M pc frs =
\{ (\langle \rangle, (None, h, (hd stk \cdot stk, loc, C, M, pc + 1) \cdot frs)) \}\n
exec-instr Swap P t h stk loc C M pc frs =
\{ let v_1 \cdot v_2 \cdot stk' = stk in (\langle \rangle, (None, h, (v_2 \cdot v_1 \cdot stk', loc, C, M, pc + 1) \cdot frs)) \}\n
B.7.3 The virtual machine

Semantics of instructions
Appendix B. Formal definitions

exec-instr (BinOp \( bop \)) \( P \ t \ h \ stk \ loc \ C \ M \ pc \ f r s \) =
\begin{enumerate}
\item let \( v_2 \cdot v_1 \cdot stk' = stk \); \([va] = \text{binop} \ bop \ v_1 \ v_2\)
\item in case \( va \) of Inl \( v \) ⇒ (\( \emptyset \), (None, \( h \cdot (v \cdot stk', loc, C, M, pc + 1) \cdot f r s \)))
\item Inr \( v \) ⇒ (\( \emptyset \), ([a], \( h \cdot (stk, loc, C, M, pc) \cdot f r s \)))
\end{enumerate}

exec-instr (New \( C' \)) \( P \ t \ h \ stk \ loc \ C \ M \ pc \ f r s \) =
\begin{enumerate}
\item let \((h', ao) = \text{alloc} \ h \ (\text{ClassT} \ C')\)
\item in case \( ao \) of None ⇒ (\( \emptyset \), ([addr-of-sys-xcpt OutOfMemory], \( h' \cdot (stk, loc, C, M, pc) \cdot f r s \)))
\item [a] ⇒ (\( \emptyset \), ([New-Obj \ a \ C'], (None, \( h' \cdot (Addr \ a \cdot stk, loc, C, M, pc + 1) \cdot f r s \))))
\end{enumerate}

exec-instr (NewArray \( T \)) \( P \ t \ h \ stk \ loc \ C \ M \ pc \ f r s \) =
\begin{enumerate}
\item let \( \text{Intg} \ si \cdot stk' = stk \); \( i = \text{nat} \ (\text{sint} \ si)\)
\item in if \( si < s_0 \) then (\( \emptyset \), ([addr-of-sys-xcpt NegativeArraySize], \( h \cdot (stk, loc, C, M, pc) \cdot f r s \)))
\item else let \((h', ao) = \text{alloc} \ h \ (\text{ArrayT} \ T \ i)\)
\item in case \( ao \) of None ⇒ (\( \emptyset \), ([addr-of-sys-xcpt OutOfMemory], \( h' \cdot (stk, loc, C, M, pc) \cdot f r s \)))
\item [a] ⇒ (\( \emptyset \), ([New-Arr \ a \ T \ i], (None, \( h' \cdot (Addr \ a \cdot stk', loc, C, M, pc + 1) \cdot f r s \))))
\end{enumerate}

exec-instr ALoad \( P \ t \ h \ stk \ loc \ C \ M \ pc \ f r s \) =
\begin{enumerate}
\item let \( \text{Intg} \ i \cdot v \cdot stk' = stk \); \( Addr \ a = v \); \( idx = \text{nat} \ (\text{sint} \ i)\);
\item \([hT] = \text{typeof-addr} \ h \ a\); \( len = \text{array-length-of} \ hT\)
\item in if \( v = \text{Null} \) then (\( \emptyset \), ([addr-of-sys-xcpt NP], \( h \cdot (stk, loc, C, M, pc) \cdot f r s \)))
\item else if \( i < s \) ∨ \( \text{int} \ len \leq \text{sint} \ i \) then (\( \emptyset \), ([addr-of-sys-xcpt ArrayIndexOutOfRange], \( h \cdot (stk, loc, C, M, pc) \cdot f r s \)))
\item else (\( \emptyset \), ([Read \ a \ (Cell \ idx) \ v'], (None, \( (v' \cdot stk', loc, C, M, pc + 1) \cdot f r s \))) | \( v' \).
\item \( \text{read} \ h \ a \ (Cell \ idx) \ v' \))
\end{enumerate}
exec-instr $\text{AStore} \ P \ h \ stk \ loc \ C \ M \ pc \ f rs$

$(\text{let } v_2 \cdot \text{Intg } i \cdot v_1 \cdot \text{stk}^\prime = \text{stk}; \ \text{Addr } a = v_1; \ \text{idx} = \text{nat} (\text{sint } i);$

$[\text{hT}] = \text{typeof-addr } h \ a; \ \text{len} = \text{array-length-of } hT; \ \text{T}[\] = \text{ty-of } hT;$

$[\text{T}^\prime] = \text{typeof}_b v_2; \ \text{in if } v_1 = \text{null then}$

$\{ (\text{null}, \ (\text{addr-of-sys-exception NullPointer} : h, \ (\text{stk}, \text{loc}, C, M, pc) \cdot f rs)) \} \}$

$\text{else if } i < s_1 \vee \text{int len} \leq \text{sint } i \text{ then}$

$\{ (\text{null}, \ (\text{addr-of-sys-exception ArrayIndexOutOfBoundsException},$

$h, \ (\text{stk}, \text{loc}, C, M, pc) \cdot f rs)) \} \}$

$\text{else if } P + T^\prime \leq T \text{ then}$

$\{ (\text{Write } a \ (\text{Cell } \text{idx} \ v_2), \ (\text{None}, h^\prime, \ (\text{stk}^\prime, \text{loc}, C, M, pc + 1) \cdot f rs)) \} \ | h'.$

$\text{write } h \ a \ (\text{Cell } \text{idx} \ v_2 \ h^\prime)$

$\text{else } \{ (\text{null}, \ (\text{addr-of-sys-exception ArrayStore}, h, \ (\text{stk}, \text{loc}, C, M, pc) \cdot f rs)) \} \}$

exec-instr $\text{ALength} \ P \ h \ stk \ loc \ C \ M \ pc \ f rs$

$(\text{let } v \cdot \text{stk}^\prime = \text{stk}; \ \text{Addr } a = v; \ \text{[hT]} = \text{typeof-addr } a;$

$\text{len} = \text{array-length-of } hT$

$\text{in if } v = \text{null then}$

$(\text{null}, \ (\text{addr-of-sys-exception NullPointer} : h, \ (\text{stk}, \text{loc}, C, M, pc) \cdot f rs)) \}$

$\text{else } (\text{None}, h, \ (\text{word-of-int} (\text{int len}) \cdot \text{stk}^\prime, \text{loc}, C, M, pc + 1) \cdot f rs)) \}$

exec-instr $(\text{Getfield } F \ D) \ P \ h \ stk \ loc \ C \ M \ pc \ f rs$

$(\text{let } v \cdot \text{stk}^\prime = \text{stk}; \ \text{Addr } a = v$

$\text{in if } v = \text{null then}$

$(\text{null}, \ (\text{addr-of-sys-exception NullPointer} : h, \ (\text{stk}, \text{loc}, C, M, pc) \cdot f rs)) \}$

$\text{else } \{ (\text{Read } a \ (\text{Field } F \ D \ v^\prime), \ (\text{None}, h, (v^\prime \cdot \text{stk}^\prime, \text{loc}, C, M, pc + 1) \cdot f rs)) \} \ | v^\prime.$

$\text{read } h \ a \ (\text{Field } F \ D \ v^\prime) \}$

exec-instr $(\text{Putfield } F \ D) \ P \ h \ stk \ loc \ C \ M \ pc \ f rs$

$(\text{let } v \cdot r \cdot \text{stk}^\prime = \text{stk}; \ \text{Addr } a = r$

$\text{in if } r = \text{null then}$

$(\text{null}, \ (\text{addr-of-sys-exception NullPointer} : h, \ (\text{stk}, \text{loc}, C, M, pc) \cdot f rs)) \}$

$\text{else } \{ (\text{Write } a \ (\text{Field } F \ D \ v), \ (\text{None}, h^\prime, (\text{stk}^\prime, \text{loc}, C, M, pc + 1) \cdot f rs)) \} \ | h'.$

$\text{write } h \ a \ (\text{Field } F \ D \ v \ h^\prime) \}$

exec-instr $(\text{Checkcast } T) \ P \ h \ stk \ loc \ C \ M \ pc \ f rs$

$(\text{let } [\text{T}^\prime] = \text{typeof}_b (\text{hd} \text{stk})$

$\text{in if } P + T^\prime \leq T \text{ then } (\text{None}, h, (\text{stk}, \text{loc}, C, M, pc + 1) \cdot f rs)$

$\text{else } (\text{null}, \ (\text{addr-of-sys-exception ClassCast} : h, \ (\text{stk}, \text{loc}, C, M, pc) \cdot f rs)) \}$
exec-instr (Instanceof T) P t h stk loc C M pc frs =
\{ (\emptyset), let v \cdot stk' = stk; [T'] = typeof_h v \\
  in (None, h, (Bool (v \neq \text{Null} \land P + T' \leq T) \cdot stk', loc, C, M, pc + 1) \cdot frs) \}\}

exec-instr (Invoke M' n) P t h stk loc C M pc frs =
(\let ps = \text{rev} (\text{take n stk}); r = stk[n]; Addr a = r; [hT] = typeof-addr h a \\
in if r = \text{Null} then \\
  \{ (\emptyset), ([\text{addr-of-sys-xcpt NullPointer}], h, (stk, loc, C, M, pc) \cdot frs) \}\}
else let (D, Ts, T, m) = method P (class-of' hT) M' \\
in case m of Native ⇒ \\
  \{ (native-TA2jvm P ta, native-Ret2jvm n h' stk loc C M pc frs vx) \\
  | ta vx h'. (ta, vx, h') \in \text{exec-native} P (a M') ps h \} \\
| [\{\\text{msl}, ml, ins, xt\}] ⇒ \\
  let fr' = ([\emptyset], r \cdot ps @ \text{replicate msl undefined-Val}, D, M', 0) \\
in \{ (\emptyset), (None, h, fr' \cdot (stk, loc, C, M, pc) \cdot frs) \}\}

exec-instr Return P t h stk loc C M pc frs =
\{ (\emptyset), if frs = [] then (None, h, []) \\
  else let v = hd stk; (stk', loc', C', M', pc') \cdot frs' = frs; \\
  (-, Ts, -, -) = \text{method P C M} \\
in (None, h, (v \cdot \text{drop} ([Ts] + 1) stk', loc', C', M', pc' + 1) \cdot frs')) \}\}

exec-instr (Goto i) P t h stk loc C M pc frs =
\{ (\emptyset), (None, h, (stk, loc, C, M, nat (int pc + i)) \cdot frs) \}\}

exec-instr (IfFalse i) P t h stk loc C M pc frs =
\{ (\emptyset), let pc' = if hd stk = \text{Bool False} then nat (int pc + i) else pc + 1 \\
in (None, h, (stk, loc, C, M, pc') \cdot frs) \}\}

exec-instr ThrowExc P t h stk loc C M pc frs =
\{ (\emptyset), let v \cdot _ = stk; Addr a = v; \\
xp = if v = \text{Null} then [\text{addr-of-sys-xcpt NullPointer}] else [a] \\
in (xp, h, (stk, loc, C, M, pc) \cdot frs) \}\}

exec-instr MEnter P t h stk loc C M frs =
(\let v \cdot stk' = stk; Addr a = v \\
in if v = \text{Null} then \\
  \{ (\emptyset), ([\text{addr-of-sys-xcpt NullPointer}], h, (stk, loc, C, M, pc) \cdot frs) \}\}
else \{ (\{\text{Lock} \rightarrow a, \text{SLock} a\}, (None, h, (stk', loc, C, M, pc + 1) \cdot frs)) \}\)
B.7. Bytecode

exec-instr MExit P t stk loc C M frs =
(let v · stk' = stk; Addr a = v
in if v = Null then
  { ((), ([addr-of-sys-xcpt NullPointer], h, (stk, loc, C, M, pc) · frs)) }
else*
  { ([Unlock→a, SUnlock a], (None, h, (stk', loc, C, M, pc + 1) · frs)),
    ([UnlockFail→a], ([addr-of-sys-xcpt IllegalMonitorState],
    h, (stk, loc, C, M, pc) · frs)) }

Exception handling

matches-ex-entry :: 'm prog ⇒ cname ⇒ pc ⇒ ex-entry ⇒ bool
matches-ex-entry P C pc (f, t, Co, pc', d) ←→
  f ≤ pc ∧ pc < t ∧ (case Co of Any ⇒ True | [C'] ⇒ P ⊢ C ≤* C')

match-ex-table :: 'm prog ⇒ cname ⇒ pc ⇒ ex-entry ⇒ (pc × nat) option
match-ex-table P C pc [] = None
match-ex-table P C pc (ex · xt) =
  (if matches-ex-entry P C pc ex then [snd (snd (snd ex))]
  else match-ex-table P C pc xt)

xcpt-step :: 'addr jvm-prog ⇒ 'addr ⇒ 'heap ⇒ 'addr frame ⇒ 'addr frame list
  ⇒ ('addr, 'heap) jvm-state
xcpt-step P a h (stk, loc, C, M, pc) frs =
  (case match-ex-table P (cname-of h a) pc (ex-table-of P C M) of
   None ⇒ ([a], h, frs)
   | [(pc', d)] ⇒ (None, h, (Addr a · drop (|stk| − d) stk, loc, C, M, pc') · frs))

auxiliary functions:
cname-of h a = (let [hT] = typeof-addr h a; Class C = ty-of hT in C)
ex-table-of P C M =
  (let (_ _ _ _ meth) = method P C M; [(_ _ _ _ xt)] = meth in xt)
instrs-of P C M =
  (let (_ _ _ _ meth) = method P C M; [(_ _ _ ins, _)] = meth in ins)

Single-threaded semantics

exec P t (xcp, h, []) = ∅
exec P t (None, h, (stk, loc, C, M, pc) · frs) =
  exec-instr (instrs-of P C M)[pc] P t h stk C M pc frs
exec P t([a], h, fr · frs) = { ((), xcpt-step P a h fr frs) }

347
Appendix B. Formal definitions

B.7.4 Observability

\[\tau_{-}\text{instr} :: \text{'m prog} \Rightarrow \text{'heap} \Rightarrow \text{'addr opstack} \Rightarrow \text{'addr instr} \Rightarrow \text{bool}\]

\[\tau_{-}\text{instr} \ P h stk (\text{Load } i) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk (\text{Store } i) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk (\text{Push } v) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk \text{ Pop} \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk \text{ Dup} \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk \text{ Swap} \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk (\text{BinOp } bop) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk (\text{New } C) \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk (\text{NewArray } T) \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk \text{ ALoad} \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk \text{ AStore} \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk \text{ ALength} \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk (\text{Getfield } F D) \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk (\text{Putfield } F D) \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk (\text{Checkcast } T) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk (\text{Instanceof } T) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk (\text{Invoke } M n) \ \iff \ n < |stk| \land\]

\[(stk_{[n]} = \text{Null}) \lor\]

\[(\forall T T s T r D. \ \text{typeof-addr } h (\text{the-Addr } stk_{[n]}) = [T] \rightarrow P+ \ \text{class-of' } T \text{ sees } M: T s \rightarrow T r = \text{Native in } D \rightarrow \tau_{\text{native }} D M)\]

\[\tau_{-}\text{instr} \ P h stk \text{ Return} \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk \text{ (Goto } i) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk \text{ (IfFalse } i) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk \text{ (ThrowExc} \) \ \iff \ True\]

\[\tau_{-}\text{instr} \ P h stk \text{ MEnter} \ \iff \ False\]

\[\tau_{-}\text{instr} \ P h stk \text{ MExit} \ \iff \ False\]

\[\tau_{\text{jvm}} :: \text{'addr jvm-prog} \Rightarrow (\text{'addr, 'heap}) \text{jvm-state} \Rightarrow \text{bool}\]

\[\tau_{\text{jvm}} \text{P} (xcp, h, []) = \text{False}\]

\[\tau_{\text{jvm}} \text{P} (xcp, h, (stk, loc, C, M, pc) \cdot f rs) =\]

\[\text{let } (\_, \_, \_ \text{meth} ) = \text{method P C M; } [\_ \_, \_ \text{ins, xt}] = \text{meth}\]

\[\text{in } \text{pc } < |\text{ins}| \land (xcp = \text{None} \rightarrow \tau_{-}\text{instr} \ P h stk stk_{[pc]})\]

\[\text{jvm-}\text{move P} ((xcp, f rs), h) \text{ ta } \_ \iff \tau_{\text{jvm}} \text{P} (xcp, h, f rs) \land \text{ta } = \emptyset\]

 auxiliary function: \text{the-Addr} (\text{Addr } a) = a
B.8 The Java memory model

**type_synonym** JMM-event = nat

**type_synonym** ('addr', 't) execution = ('t × ('addr', 't) event sf) llist

\[ \mathcal{A}_E = \{ \alpha. \alpha < |E| \} \]

\( \text{is-write} (\text{Allocate} \ a \ hT) \quad \text{is-write} (\text{Write} \ a \ al \ v) \)

\( \mathcal{W}_E = \{ \alpha. \alpha \in \mathcal{A}_E \land \text{is-write} (\text{snd} \ E_\alpha) \} \)

\( \mathcal{R}_E = \{ \alpha. \alpha \in \mathcal{A}_E \land \exists \ a \ al \ v. \text{snd} \ E_\alpha = \text{Read} \ a \ al \ v \} \)

\( \text{init}_E \alpha \leftarrow (\exists \ a \ hT. \text{snd} \ E_\alpha = \text{Allocate} \ a \ hT) \)

**Synchronisation events**

\[
\begin{array}{ll}
\text{is-volatile} \ P \ al & \text{is-volatile} \ P \ al \\
\text{sync-event} \ P (\text{Read} \ a \ al \ v) & \text{sync-event} \ P (\text{Write} \ a \ al \ v) \\
\text{sync-event} \ P (\text{Allocate} \ a \ hT) & \text{sync-event} \ P (\text{TStart} \ t) \\
\text{sync-event} \ P (\text{TJoin} \ t) & \text{sync-event} \ P (\text{SLock} \ a) \\
\text{sync-event} \ P (\text{SUnlock} \ a) & \text{sync-event} \ P (\text{TIntr} \ t) \\
\text{sync-event} \ P (\text{TIntrd} \ t) & \text{sync-event} \ P \ TInit & \text{sync-event} \ P \ TFinish
\end{array}
\]

\( \text{sync-events} :: \ 'm \ \text{prog} \Rightarrow (\text{addr}', 't) \ \text{execution} \Rightarrow \text{JMM-event} \ \text{set} \)

\( \text{sync-events} \ P \ E = \{ \alpha. \alpha \in \mathcal{A}_E \land \text{sync-event} \ P (\text{snd} \ E_\alpha) \} \)

**Accessed locations**

\( (a, al) \in \text{locs}' \ P (\text{Read} \ a \ al \ v) \quad (a, al) \in \text{locs}' \ P (\text{Write} \ a \ al \ v) \)

\( \frac{P \vdash \text{class-of} \ hT \ \text{has} \ F:T_f (f m) \ \text{in} \ D}{(a, \text{Field} \ F \ D) \in \text{locs}' \ P (\text{Allocate} \ a \ hT)} \)

\( \frac{n < n'}{(a, \text{Cell} \ n) \in \text{locs}' \ P (\text{New-Arr} \ a \ T \ n')} \)
Appendix B. Formal definitions

locs :: 'm prog ⇒ ('addr, 't) execution ⇒ ('addr × loc) set
locs P E α = locs' P (snd E[α])

loc-default :: 'm prog ⇒ hty ⇒ loc ⇒ 'addr val
loc-default P (ClassT C) (Field F D) =
  default-val (fst (the (map-of (fields P C) (F, D))))
lc-default P (ArrayT T n) (Field F D) =
  default-val (fst (the (map-of (fields P Object) (F, Object))))
lc-default P (ArrayT T n) (Cell n') = default-val T

value-written' P (Allocate a hT) al = locs-default P hT al
value-written' P (Write a al v) al' = (if al = al' then v else undefined)

value-written P E α (a, al) = value-written' P (snd E[α]) al

Orders

release-acquire pairs:

(t, SUnlock a) ⇝ sw (t', SLock a) (t, Write a al v) ⇝ sw (t', Read a al v')
(t, New-Obj a C) ⇝ sw (t', Read a al v) (t, TStart t') ⇝ sw (t', TInit)
(t, TFinish) ⇝ sw (t', TJoin t) (t, Allocate a hT) ⇝ sw (t', TInit)
(t, TIntr t'') ⇝ sw (t', TIntrd t'')

induced total order: \( \leq^E = \leq_{A_E} \)

execution order:

\( \alpha \leq_{eo} \alpha' \rightleftharpoons (if \ init_E \alpha \ then \ notinit_E \alpha' \ or \ \alpha \leq^E \alpha' \ else \ notinit_E \alpha' \ and \ \alpha \leq^E \alpha') \)

program order:

\( \alpha \leq_{po} \alpha' \rightleftharpoons \alpha \leq_{eo} \alpha' \ and \ \text{fst} E[\alpha] = \text{fst} E[\alpha'] \)

synchronisation order:

\( \alpha \leq_{so} \alpha' \rightleftharpoons \alpha \leq_{eo} \alpha' \ and \ \alpha \in \text{sync-events} \ P E \ and \ \alpha' \in \text{sync-events} \ P E \)

synchronises-with order:

\( \alpha \leq_{sw} \alpha' \rightleftharpoons \alpha \leq_{po} \alpha' \ and \ \alpha \leq_{so} \alpha' \ and \ \alpha \leq_{sw} E[\alpha'] \)

happens-before order:

\( \leq_{hb} = \lambda \alpha' . \alpha \leq_{po} \alpha' \ or \ \alpha \leq_{sw} \alpha' \)

Well-formed and legal executions

ok-init E \rightleftharpoons (\forall \alpha \in A_E. \ notinit_E \alpha \ rightleftharpoons (\exists \beta . \ \beta < \alpha \ and \ E[\beta] = (\text{fst} E[\alpha], TInit)))
B.8. The Java memory model

\[ P \vdash (E, ws) \sqrt{f} \iff \text{ok-init } E \land \]
\[ (\forall \alpha \in R_E. \forall a \in \text{lock } E \alpha = \text{Read } a \alpha \implies \]
\[ \text{value-written } P E \alpha (a, al) \in \text{locs } P E \alpha \land \]
\[ \text{most recent write: } a M \equiv v \land \]
\[ (\forall \beta \in W_E. (a, al) \in \text{locs } P E \beta \implies \]
\[ (ws \alpha \leq_{hb} P E \beta \land \text{is-volatile } P al \implies \alpha \leq_{so} P E \alpha \land \]
\[ P, E \vdash \alpha \rightsquigarrow_{mrw} \beta \]
\[ \text{sequentially-consistent } P (E, ws) \iff (\forall r \in R_E. P, E \vdash r \rightsquigarrow_{mrw} ws r) \]

allocate a hT \equiv \text{allocate a hT } \quad \text{read } a \alpha v \equiv \text{read } a \alpha v' \quad \text{write } a \alpha v \equiv \text{write } a \alpha v' \quad \text{tstart } t = \text{tstart } t \quad \text{tjoin } t = \text{tjoin } t \quad \text{unlock } a = \text{unlock } a \quad \text{tunlock } a = \text{tunlock } a \quad \text{extern } a M \equiv \text{extern } a M \equiv v \quad \text{tintr } t = \text{tintr } t \quad \text{tintrd } t = \text{tintrd } t \quad \text{tinit } = \text{tinit } \quad \text{tfinish } = \text{tfinish} \]

\[ P \vdash (E, ws) \square \iff \text{justified-by } (E_i, ws_i, C_i, \phi_i) \]
\[ (\forall i. P \vdash (E_i, ws_i) \sqrt{f}) \land \]
\[ (\forall i. \text{inj-on } \phi_i \mathcal{A}_{E_i} \land (\forall \alpha \in C_i, E_{i[a]} \equiv E_{i[\phi_i a]}) \land (\forall i. C_i \subseteq \mathcal{A}_{E_i}) \land \]
\[ C_0 = \emptyset \land (\forall i. \phi_i \mathcal{C}_i \subseteq \phi_{i+1} \mathcal{C}_{i+1} \land \mathcal{A}_E = \bigcup_i \phi_i \mathcal{C}_i \land \]
\[ (\forall i. \leq_{hb} P_{E_i} \mid \mathcal{C}_i = \phi_i^{-1} (\leq_{so} P_{E_i} \mid \mathcal{C}_i) \land (\forall i. \leq_{so} P_{E_i} \mid \mathcal{C}_i = \phi_i^{-1} (\leq_{so} P_{E_i} \mid \mathcal{C}_i) \land \]
\[ (\forall i. \forall \alpha \in W_{E_i} \cap C_i. \forall (a, al) \in \text{locs } P E \phi_i a). \]
\[ \text{value-written } P E a \alpha (a, al) = \text{value-written } P E (\phi_i a) (a, al) \land \]
\[ (\forall i. \forall \alpha \in R_{E_i} \cap C_i. \phi_{i+1} (ws_{i+1} (\phi_{i+1}^{-1} (\phi_i a))) = ws (\phi_i a)) \land \]
\[ (\forall i. \forall \alpha \in R_{E_{i+1}}. \phi_{i+1} \alpha \in \phi_i \mathcal{C}_i \land \text{ws}_{i+1} \alpha \leq_{hb} P_{E_{i+1}} \alpha) \land \]
\[ (\forall i. \forall \alpha \in R_{E_{i+1}} \cap C_{i+1}. \]
\[ \phi_{i+1} \alpha \in \phi_i \mathcal{C}_i \land \phi_{i+1} \mathcal{C}_i \land \text{ws}_{i+1} (\phi_{i+1} \alpha) \leq_{so} \phi_{i+1} \mathcal{C}_i \land \]
\[ (\forall i. \forall \alpha \in \mathcal{A}_{E_i}. \forall \alpha' \in C_i. \forall a M \equiv v \quad \text{e}_{i[a]} = \text{extern } a M v \implies \alpha \leq_{hb} P_{E_i} \alpha' \implies \alpha \in C_i) \]

351
Appendix B. Formal definitions

\[ P, E \vdash (E, ws) \text{ legal } \iff E \in \mathcal{E} \land P \vdash (E, ws) \sqrt{\land} \\land \exists (E_i, ws_i, C_i, \phi_i). P \vdash (E, ws) \text{ justified-by } (E_i, ws_i, C_i, \phi_i) \land (\forall i. E_i \in \mathcal{E}) \]

\section{The compiler}

\textbf{complete compiler}

\[ J2JVM = \text{compP}_2 \circ \text{compP}_1 \]

\textbf{auxiliary functions}

\begin{align*}
\text{index } [] y &= 0 \\
\text{index } (x \cdot xs) y &= \begin{cases} 
  if x = y then if y \in \text{set xs} then \text{index xs } y + 1 else 0 else \text{index xs } y + 1 
\end{cases}
\end{align*}

\[ \text{fresh-vname } Vs = \text{concat } ("V" \cdot Vs) \]

\[ \text{max } x y = (if x \leq y then y else x) \]

\section{Program compilation}

\textbf{compilation of method declarations}

\[ \text{compM} :: (\text{mname} \Rightarrow \text{ty list} \Rightarrow \text{ty} \Rightarrow \text{t}_1 \Rightarrow \text{t}_2) \]
\[ \Rightarrow \text{t}_1 \text{ option mdecl } \Rightarrow \text{t}_2 \text{ option mdecl} \]
\[ \text{compM } f (M, Ts, T, \text{meth}) = (M, Ts, T, \text{Option.map } (f M Ts T) \text{ meth}) \]

\textbf{compilation of class declarations}

\[ \text{compC} :: (\text{cname} \Rightarrow \text{mname} \Rightarrow \text{ty list} \Rightarrow \text{ty} \Rightarrow \text{t}_1 \Rightarrow \text{t}_2) \]
\[ \Rightarrow \text{t}_1 \text{ cdecl } \Rightarrow \text{t}_2 \text{ cdecl} \]
\[ \text{compC } f (C, D, fs, ms) = (C, D, fs, \text{map } (\text{compM } (f C))) \text{ ms} \]

\textbf{compilation of program declarations}

\[ \text{compP} :: (\text{cname} \Rightarrow \text{mname} \Rightarrow \text{ty list} \Rightarrow \text{ty} \Rightarrow \text{t}_1 \Rightarrow \text{t}_2) \]
\[ \Rightarrow \text{t}_1 \text{ prog } \Rightarrow \text{t}_2 \text{ prog} \]
\[ \text{compP } f (\text{Program } P) = \text{Program } (\text{map } (\text{compC } f) P) \]

\section{Compilation stage 1}

\textbf{compilation of expressions}

\[ \text{compE}_1 :: \text{vname list } \Rightarrow \text{'addr expr } \Rightarrow \text{'addr expr}_1 \]
\[ \text{compE}_1 Vs (\text{new } C) = \text{new } C \]
\[ \text{compE}_1 Vs (\text{new } T[e]) = \text{new } T[\text{compE}_1 Vs e] \]
\[ \text{compE}_1 Vs (\text{Cast } T e) = \text{Cast } T (\text{compE}_1 Vs e) \]
B.9. The compiler

compE₁ Vs (e instanceof T) = (compE₁ Vs e) instanceof T
compE₁ Vs (Val v) = Val v
compE₁ Vs (e₁ «bop» e₂) = (compE₁ Vs e₁) «bop» (compE₁ Vs e₂)
compE₁ Vs (Var V) = Var (index Vs V)
compE₁ Vs (V := e) = (index Vs V) := (compE₁ Vs e)
compE₁ Vs (e₁[e₂]) = (compE₁ Vs e₁)[compE₁ Vs e₂]
compE₁ Vs (e₁[e₂] := e₃) =
  (compE₁ Vs e₁)[compE₁ Vs e₂] := compE₁ Vs e₃
compE₁ Vs (e.length) = (compE₁ Vs e).length
compE₁ Vs (e.F[D]) = (compE₁ Vs e).F[D]
compE₁ Vs (e₁.F[D] := e₂) = (compE₁ Vs e₁).F[D] := compE₁ Vs e₂
compE₁ Vs (e.M(es)) = (compE₁ Vs e).M(map (compE₁ Vs) es)
compE₁ Vs [V: T = vo; e] = [|Vs|: T = vo; compE₁ (Vs @ [V]) e]
compE₁ Vs (e₁;; e₂) = compE₁ Vs e₁;; compE₁ Vs e₂
compE₁ Vs (if (e₁) e₂ else e₃) =
  if (compE₁ Vs e₁) compE₁ Vs e₂ else compE₁ Vs e₃
compE₁ Vs (while (e₁) e₂) = while (compE₁ Vs e₁) (compE₁ Vs e₂)
compE₁ Vs (throw e) = throw (compE₁ Vs e)
compE₁ Vs (try e₁ catch(C V) e₂) =
  try compE₁ Vs e₁ catch(C |Vs|) (compE₁ (Vs @ |Vs|) e₂)
compE₁ Vs (sync (e₁) e₂) =
  sync|Vs| (compE₁ Vs e₁) (compE₁ (Vs @ [fresh-vname Vs]) e₂)
compE₁ Vs (insync (a) e) =
  insync|Vs| (a) (compE₁ (Vs @ [fresh-vname Vs]) e)

compilation of programs
compP₁ = compP (λC M Ts T (pns, body). compE₁ (this · pns) body)

B.9.3 Compilation stage 2

compilation of expressions
compE₂ :: `addr expr₁ ⇒ `addr instr list
compE₂ (new C) = [New C]
compE₂ (new T[e]) = compE₂ e @ [NewArray T]
compE₂ (Cast T e) = compE₂ e @ [Checkcast T]
compE₂ (e instanceof T) = compE₂ e @ [Instanceof T]
compE₂ (Val v) = [Push v]
compE₂ (e₁ «bop» e₂) = compE₂ e₁ @ compE₂ e₂ @ [BinOp bop]
compE₂ (Var i) = [Load i]
compE₂ (i := e) = compE₂ e @ [Store i, Push Unit]
compE₂ (e₁[e₂]) = compE₂ e₁ @ compE₂ e₂ @ [ALoad]
Appendix B. Formal definitions

\[
\begin{align*}
\text{compE}_2 (e_1[e_2] & := e_3) = \\
\text{compE}_2 e_1 @ \text{compE}_2 e_2 @ \text{compE}_2 e_3 @ [\text{AStore}, \text{Push Unit}] \\
\text{compE}_2 (e.\text{length}) & = \text{compE}_2 e @ [\text{ALength}] \\
\text{compE}_2 (e.F[D]) & = \text{compE}_2 e @ [\text{GetField F D}] \\
\text{compE}_2 (e_1.F[D] := e_2) & = \\
\text{compE}_2 e_1 @ \text{compE}_2 e_2 @ [\text{PutField F D}, \text{Push Unit}] \\
\text{compE}_2 (e.M(es)) & = \text{compE}_2 e @ \text{compE}_2 es @ [\text{Invoke M es}] \\
\text{compE}_2 [i: T = \nu v; e] & = \\
\begin{cases}
\text{case } \nu v \text{ of None } \Rightarrow [] & | [v] \Rightarrow [\text{Push } v, \text{Store } i] @ \text{compE}_2 e \\
\text{compE}_2 (e_1 ; e_2) & = \text{compE}_2 e_1 @ [\text{Pop }] @ \text{compE}_2 e_2 \\
\text{compE}_2 (\text{if } (e_1) e_2 \text{ else } e_3) & = \\
(\text{let } \text{cnd } = \text{compE}_2 e_1; \text{thn } = \text{compE}_2 e_2; \text{els } = \text{compE}_2 e_3 \\
in \text{cnd } @ [\text{IfFalse } (\text{int } (|\text{thn}| + 2))] @ \text{thn } @ [\text{Goto } (\text{int } (|\text{els}| + 1))] @ \text{els}) \\
\text{compE}_2 (\text{while } (e_1) e_2) & = \\
(\text{let } \text{cnd } = \text{compE}_2 e_1; \text{bdy } = \text{compE}_2 e_2; \text{loop } = [\text{Goto } (\text{-}(\text{int } (|\text{bdy}| + |\text{cnd}| + 2)))] \\
in \text{cnd } @ [\text{IfFalse } (\text{int } (|\text{bdy}| + 3))] @ \text{bdy } @ [\text{Pop }] @ \text{loop } @ [\text{Push Unit}]) \\
\text{compE}_2 (\text{throw } e) & = \text{compE}_2 e @ [\text{ThrowExc}] \\
\text{compE}_2 (\text{try } e_1 \text{ catch } (C i) e_2) & = \\
(\text{let } \text{catch } = \text{compE}_2 e_2 \\
in \text{compE}_2 e_1 @ [\text{Goto } (\text{int } |\text{catch}| + 2), \text{Store } i] @ \text{catch}) \\
\text{compE}_2 (\text{sync}_i (e_1) e_2) & = \\
\text{compE}_2 e_1 @ [\text{Dup }, \text{Store } i, M\text{Enter }] @ \\
\text{compE}_2 e_2 @ [\text{Load } i, M\text{Exit}, \text{Goto } 4] @ [\text{Load } i, M\text{Exit}, \text{ThrowExc}] \\
\text{compE}_2 (\text{insync}_i (a) e) & = [\text{Goto } 1]^{64} \\
\text{compE}_2 :: \text{`addr expr}_1 \text{ list } \Rightarrow \text{`addr instr list} \\
\text{compE}_2 [] & = [] \\
\text{compE}_2 (e \cdot es) & = \text{compE}_2 e @ \text{compE}_2 es
\end{align*}
\]

exception tables

\[
\begin{align*}
\text{compx}_2 :: \text{`addr expr}_1 \Rightarrow \text{pc } \Rightarrow \text{nat } \Rightarrow \text{ex-table} \\
\text{compx}_2 (\text{new } C) & = \text{pc } d = [] \\
\text{compx}_2 (\text{new } T[e]) & = \text{pc } d = \text{compx}_2 e \text{ pc } d \\
\text{compx}_2 (\text{Cast } T e) & = \text{pc } d = \text{compx}_2 e \text{ pc } d \\
\text{compx}_2 (\text{e instanceof } T) & = \text{pc } d = \text{compx}_2 e \text{ pc } d \\
\text{compx}_2 (\text{Val } v) & = \text{pc } d = [] \\
\text{compx}_2 (e_1 \text{ `bop } e_2) & = \text{pc } d = \\
\text{compx}_2 e_1 \text{ pc } d @ \text{compx}_2 e_2 (\text{pc } + |\text{compx}_2 e_1|) (d + 1) \\
\text{compx}_2 (\text{Var } i) & = \text{pc } d = [] \\
\text{compx}_2 (i := e) & = \text{pc } d = \text{compx}_2 e \text{ pc } d
\end{align*}
\]

\[^{64}\text{Since insync } (\_\_\_\_\_\text{ }) \text{ is not part of the input syntax, compE}_2 \text{ outputs a no-op.}\]
B.9. The compiler

\[
\text{compxE}_2 (e_1[e_2]) \quad \text{pc} \ d = \\
\text{compxE}_2 e_1 \\text{pc} \ d @ \text{compxE}_2 e_2 (\text{pc} + |\text{compxE}_2 e_1|) \ (d + 1)
\]
\[
\text{compxE}_2 (e_1[e_2] := e_3) \quad \text{pc} \ d = \\
(\text{let} \ \text{pc}_1 = \text{pc} + |\text{compxE}_2 e_1|; \ \text{pc}_2 = \text{pc}_1 + |\text{compxE}_2 e_2| \\
in \text{compxE}_2 e_1 \ \text{pc} \ d @ \text{compxE}_2 e_2 \text{pc}_1 \ (d + 1) @ \text{compxE}_2 e_3 \text{pc}_2 \ (d + 2))
\]
\[
\text{compxE}_2 (e.\text{length}) \quad \text{pc} \ d = \text{compxE}_2 e \ \text{pc} \ d
\]
\[
\text{compxE}_2 (e.F[D]) \quad \text{pc} \ d = \text{compxE}_2 e \ \text{pc} \ d
\]
\[
\text{compxE}_2 (e.F[D] := e_2) \quad \text{pc} \ d = \\
\text{compxE}_2 e_1 \ \text{pc} \ d @ \text{compxE}_2 e_2 (\text{pc} + |\text{compxE}_2 e_1|) \ (d + 1)
\]
\[
\text{compxE}_2 (e.M(es)) \quad \text{pc} \ d = \\
\text{compxE}_2 e \ \text{pc} \ d @ \text{compxE}_2 es (\text{pc} + |\text{compxE}_2 e|) \ (d + 1)
\]
\[
\text{compxE}_2 ((i: T = vo; e)) \quad \text{pc} \ d = \\
\text{compxE}_2 e \ (\text{case} vo \ \text{of} \ \text{None} \Rightarrow \text{pc} | [v] \Rightarrow \text{pc} + 2) \ d
\]
\[
\text{compxE}_2 (e_1; e_2) \quad \text{pc} \ d = \\
\text{compxE}_2 e_1 \ \text{pc} \ d @ \text{compxE}_2 e_2 (\text{pc} + |\text{compxE}_2 e_1| + 1) \ d
\]
\[
\text{compxE}_2 (\text{if} (e_1) e_2 \ \text{else} e_3) \quad \text{pc} \ d = \\
(\text{let} \ \text{pc}_1 = \text{pc} + |\text{compxE}_2 e_1| + 1; \ \text{pc}_2 = \text{pc}_1 + |\text{compxE}_2 e_2| + 1 \\
in \text{compxE}_2 e_1 \ \text{pc} \ d @ \text{compxE}_2 e_2 \text{pc}_1 \ d @ \text{compxE}_2 e_3 \text{pc}_2 \ d)
\]
\[
\text{compxE}_2 (\text{while} (e_1) e_2) \quad \text{pc} \ d = \\
\text{compxE}_2 e_1 \ \text{pc} \ d @ \text{compxE}_2 e_2 (\text{pc} + |\text{compxE}_2 e_1| + 1) \ d
\]
\[
\text{compxE}_2 (\text{throw} e) \quad \text{pc} \ d = \text{compxE}_2 e \ \text{pc} \ d
\]
\[
\text{compxE}_2 (\text{try} e_1 \ \text{catch} (C \ i) e_2) \ \text{pc} \ d = \\
(\text{let} \ \text{pc}_1 = \text{pc} + |\text{compxE}_2 e_1| \\
in \text{compxE}_2 e_1 \ \text{pc} \ d @ \text{compxE}_2 e_2 (\text{pc}_1 + 2) \ d @ [(\text{pc}, \text{pc}_1, [C], \text{pc}_1 + 1, d)])
\]
\[
\text{compxE}_2 (\text{sync}_i (e_1) e_2) \quad \text{pc} \ d = \\
(\text{let} \ \text{pc}_1 = \text{pc} + |\text{compxE}_2 e_1| + 3; \ \text{pc}_2 = \text{pc}_1 + |\text{compxE}_2 e_2| \\
in \text{compxE}_2 e_1 \ \text{pc} \ d @ \text{compxE}_2 e_2 \text{pc}_1 \ d @ [(\text{pc}_1, \text{pc}_2, \text{Any}, \text{pc}_2 + 3, d)])
\]
\[
\text{compxE}_2 (\text{insync}_i (a) e) \quad \text{pc} \ d = []
\]
\[
\text{compxE}_2 :: 'addr \ expr_1 \ list \Rightarrow 'addr \ instr \ list
\]
\[
\text{compxE}_2 [] \quad \text{pc} \ d = []
\]
\[
\text{compxE}_2 (e \cdot es) \quad \text{pc} \ d = \\
\text{compxE}_2 e \ \text{pc} \ d @ \text{compxE}_2 es (\text{pc} + |\text{compxE}_2 e|) \ (d + 1)
\]

**register size**

\[
\text{max-vars :: 'addr \ expr_1 \Rightarrow nat}
\]
\[
\text{max-vars} (\text{new} C) = 0
\]
\[
\text{max-vars} (\text{new} T[e]) = \text{max-vars} e
\]
\[
\text{max-vars} (\text{Cast} T \ e) = \text{max-vars} e
\]
\[
\text{max-vars} (e \ \text{instanceof} T) = \text{max-vars} e
\]
\[
\text{max-vars} (\text{Val} v) = 0
\]
\[
\text{max-vars} (e_1 \ «\text{bop}» e_2) = \text{max} (\text{max-vars} e_1) (\text{max-vars} e_2)
\]
Appendix B. Formal definitions

\[
\begin{align*}
\text{max-vars} (\text{Var } i) &= 0 \\
\text{max-vars} (i := e) &= \text{max-vars } e \\
\text{max-vars} (e_1[e_2]) &= \max (\text{max-vars } e_1) (\text{max-vars } e_2) \\
\text{max-vars} (e_1[e_2] := e_3) &= \\
&= \max (\max (\text{max-vars } e_1) (\text{max-vars } e_2)) (\text{max-vars } e_3) \\
\text{max-vars} (\text{e.length}) &= \text{max-vars } e \\
\text{max-vars} (e.F[D]) &= \text{max-vars } e \\
\text{max-vars} (e_1.F[D] := e_2) &= \max (\text{max-vars } e_1) (\text{max-vars } e_2) \\
\text{max-vars} (e.M(es)) &= \max (\text{max-vars } e) (\text{max-vars } es) \\
\text{max-vars} ([i: T = vo; e]) &= \text{max-vars } e + 1 \\
\text{max-vars} (e_1; e_2) &= \max (\text{max-vars } e_1) (\text{max-vars } e_2) \\
\text{max-vars} (\text{if } (e_1) e_2 \text{ else } e_3) &= \\
&= \max (\max (\text{max-vars } e_1) (\text{max-vars } e_2)) (\text{max-vars } e_3) \\
\text{max-vars} (\text{while } (e_1) e_2) &= \max (\text{max-vars } e_1) (\text{max-vars } e_2) \\
\text{max-vars} (\text{throw } e) &= \text{max-vars } e \\
\text{max-vars} (\text{try } e_1 \text{ catch } (C i) e_2) &= \max (\text{max-vars } e_1) (\text{max-vars } e_2 + 1) \\
\text{max-vars} (\text{sync} (\text{e_1}) e_2) &= \max (\text{max-vars } e_1) (\text{max-vars } e_2 + 1) \\
\text{max-vars} (\text{insync} (\text{a}) e) &= \text{max-vars } e + 1 \\
\end{align*}
\]

\[
\begin{align*}
\text{max-vars} :: \cdot \text{addr expr}_1 \text{ list } &\Rightarrow \text{nat} \\
\text{max-vars} [] &= 0 \\
\text{max-vars} (e \cdot es) &= \max (\text{max-vars } e) (\text{max-vars } es)
\end{align*}
\]

**maximum stack size**

\[
\begin{align*}
\text{max-stack} :: \cdot \text{addr expr}_1 &\Rightarrow \text{nat} \\
\text{max-stack} (\text{new } C) &= 1 \\
\text{max-stack} (\text{new } T[e]) &= \text{max-stack } e \\
\text{max-stack} (\text{Cast } T e) &= \text{max-stack } e \\
\text{max-stack} (e \text{ instanceof } T) &= \text{max-stack } e \\
\text{max-stack} (\text{Val } v) &= 1 \\
\text{max-stack} (e_1 \text{ «hop» } e_2) &= \max (\text{max-stack } e_1) (\text{max-stack } e_2) + 1 \\
\text{max-stack} (\text{Var } i) &= 1 \\
\text{max-stack} (i := e) &= \text{max-stack } e \\
\text{max-stack} (e_1[e_2]) &= \max (\text{max-stack } e_1) (\text{max-stack } e_2 + 1) \\
\text{max-stack} (e_1[e_2] := e_3) &= \\
&= \max (\max (\text{max-stack } e_1) (\text{max-stack } e_2 + 1)) (\text{max-stack } e_3 + 2) \\
\text{max-stack} (\text{e.length}) &= \text{max-stack } e \\
\text{max-stack} (e.F[D]) &= \text{max-stack } e \\
\text{max-stack} (e_1.F[D] := e_2) &= \max (\text{max-stack } e_1) (\text{max-stack } e_2) + 1 \\
\text{max-stack} (e.M(es)) &= \max (\text{max-stack } e) (\text{max-stack } es) + 1 \\
\text{max-stack} ([i: T = vo; e]) &= \text{max-stack } e \\
\text{max-stack} (e_1; e_2) &= \max (\text{max-stack } e_1) (\text{max-stack } e_2)
\end{align*}
\]

356
B.9. The compiler

\[
\begin{align*}
\text{max-stack (if } (e_1) e_2 \text{ else } e_3) &= \text{max (max-stack } e_1 \text{) (max (max-stack } e_2 \text{) (max-stack } e_3)) \\
\text{max-stack (while } (e_1) e_2) &= \text{max (max-stack } e_1 \text{) (max-stack } e_2) \\
\text{max-stack (throw } e) &= \text{max-stack } e \\
\text{max-stack (try } e_1 \text{ catch(C i) } e_2) &= \text{max (max-stack } e_1 \text{) (max-stack } e_2) \\
\text{max-stack (sync\_i (a) } e_2) &= 1 \\
\text{max-stacks :: 'addr expr_1 list ⇒ nat} \\
\text{max-stacks []} &= 0 \\
\text{max-stacks (e · es)} &= \text{max (max-stack } e \text{) (max-stacks es)}
\end{align*}
\]

**program compilation**

\[
\text{compMb}_2 C M Ts T body = (let ins = \text{compE}_2 body @ [Return]; xt = \text{compxE}_2 body 0 0 in (max-stack body, max-vars body, ins, xt))
\]

\[
\text{compP}_2 = \text{compP} \text{ compMb}_2
\]

**B.9.4 Preprocessor**

**Annotation rules**

\[
\begin{align*}
E V &= \text{None} \\
V \neq \text{super} &\quad E this = [\text{Class } C] \quad P \vdash C \text{ sees } V:T (f m) \text{ in } D \\
&\quad P, E \vdash \text{Var } V \leadsto \text{Var } \text{this}.V[D] \\
E V &= [T] \quad V \neq \text{super} \\
&\quad P, E \vdash \text{Var } V \leadsto V \\
E V &= \text{None} \quad V \neq \text{super} \\
E this = [\text{Class } C] &\quad P \vdash C \text{ sees } V:T (f m) \text{ in } D \\
&\quad P, E \vdash e \leadsto e' \\
&\quad P, E \vdash V := e \leadsto \text{Var } \text{this}.V[D] := e' \\
E V &= [T] \quad V \neq \text{super} \\
&\quad P, E \vdash e \leadsto e' \\
&\quad P, E \vdash V := e \leadsto V := e' \\
\text{class-of } T &= [C] \\
&\quad P \vdash C \text{ sees } F:T (f m) \text{ in } D \\
&\quad \text{is-Array } T \leadsto F \neq \text{length} \\
&\quad P, E \vdash e.F[] \leadsto e'.F[D]
\end{align*}
\]
Appendix B. Formal definitions

\[
P, E \vdash e \rightarrow e' \quad P, E \vdash e' : T[n]
\]
\[
P, E \vdash e.\text{length}[n] \rightarrow e'.\text{length}
\]

\[
E \text{ this } = [\text{Class } C] \\
\text{class } C = \{ (D, f, s, m) \} \\
P \vdash D \text{ sees } F : T (f, m) \text{ in } D'
\]
\[
P, E \vdash \text{Var super.} F[1] \rightarrow (\text{Cast (Class } D) (\text{Var this}) \cdot F[D'])
\]
\[
P, E \vdash e_1 \rightarrow e_1' \\
P, E \vdash e_2 \rightarrow e_2' \\
P, E \vdash e_3 : T
\]
\[
\text{class of } T = [C] \\
P + C \text{ sees } F : T (f, m) \text{ in } D' \\
\text{is - Array } T \rightarrow F \neq \text{length}
\]
\[
P, E \vdash e_1.F[1] := e_2 \rightarrow e_1'.F[D] := e_2'
\]

\[
E \text{ this } = [\text{Class } C] \\
\text{class } C = \{ (D, f, s, m) \} \\
P \vdash D \text{ sees } F : T (f, m) \text{ in } D'
\]
\[
P, E \vdash \text{Var super.} F[1] := e \rightarrow (\text{Cast (Class } D) (\text{Var this}) \cdot F[D']) := e'
\]

auxiliary function: is - Array \( T \longleftrightarrow (\exists T'. T = T'[[]]) \)

Copying rules

\[
P, E \vdash \text{new } C \rightarrow \text{new } C
\]
\[
P, E \vdash e \rightarrow e' \\
P, E \vdash \text{new } T[e] \rightarrow \text{new } T[e']
\]
\[
P, E \vdash \text{Cast } T e \rightarrow \text{Cast } T e'
\]
\[
P, E \vdash e \rightarrow e' \\
P, E \vdash \text{instance of } T \rightarrow e' \text{ instance of } T
\]
\[
P, E \vdash \text{Val } v \rightarrow \text{Val } v
\]
\[
P, E \vdash e_1 \rightarrow e_1' \\
P, E \vdash e_2 \rightarrow e_2' \\
P, E \vdash \text{「bop」} e_2 \rightarrow \text{「bop」} e_2'
\]
\[
P, E \vdash e_1[1] \rightarrow e_1'[1]
\]
\[
P, E \vdash e_2 \rightarrow e_2' \\
P, E \vdash e_3 \rightarrow e_3' \\
P, E \vdash e_3[1] := e_3' \rightarrow e_3'[1] := e_3'
\]
\[
P, E \vdash e.\text{length} \rightarrow e'.\text{length}
\]
\[
P, E \vdash e \rightarrow e' \\
P, E \vdash \text{es} \rightarrow \text{es'}
\]
\[
P, E \vdash \text{es} \rightarrow \text{es'} \\
P, E \vdash e.\text{M} (\text{es}) \rightarrow e'.\text{M} (\text{es'})
\]
\[
P, E \vdash \text{sync} (e_1) e_2 \rightarrow \text{sync} (e_1') e_2'
\]

358
B.9. The compiler

\[
\begin{align*}
&P, E \vdash e_1 \leadsto e'_1 \quad P, E \vdash e_2 \leadsto e'_2 \\
&P, E \vdash e_1; \ e_2 \leadsto e'_1; \ e'_2
\end{align*}
\]

\[
\begin{align*}
&P, E \vdash e_1 \leadsto e'_1 \quad P, E \vdash e_2 \leadsto e'_2 \\
&P, E \vdash e_3 \leadsto e'_3
\end{align*}
\]

\[
\begin{align*}
&P, E \vdash \text{if } (e_1) \ e_2 \text{ else } e_3 \leadsto \text{if } (e'_1) \ e'_2 \text{ else } e'_3
\end{align*}
\]

\[
\begin{align*}
&P, E \vdash (e_1) \ e_2 \leadsto (e'_1) \ e'_2
\end{align*}
\]

\[
\begin{align*}
&P, E \vdash \text{throw } e \leadsto \text{throw } e'
\end{align*}
\]

\[
\begin{align*}
&P, E \vdash \text{try } e_1 \text{ catch } (C \ V) \ e_2 \leadsto \text{try } e'_1 \text{ catch } (C \ V) \ e'_2
\end{align*}
\]

\[
\begin{align*}
&P, E \vdash e \leadsto e'
\end{align*}
\]

\[
\begin{align*}
&P, E \vdash e \cdot es \leadsto e' \cdot es'
\end{align*}
\]

Lifting to programs

\[
\begin{align*}
\text{annotate } P \ E \ e &= (\text{if } \exists! e'. \ P, E \vdash e \leadsto e' \text{ then } u e'. \ P, E \vdash e \leadsto e' \text{ else } e)
\end{align*}
\]

\[
\begin{align*}
\text{annotate-Mb } P \ C \ M \ Ts \ T \ (pns, e) &= (pns, \text{annotate } P \ [\text{this} \mapsto \text{Class } C, pns \mapsto T] \ e)
\end{align*}
\]

\[
\begin{align*}
\text{annotate-Prog } P &= \text{compP} (\text{annotate-Mb } P) \ P
\end{align*}
\]
List of Figures

1.1 Three Java threads with different deadlock possibilities 4
1.2 Program with two threads and three of its sequentially consistent schedules 5
1.3 Synchronisation and publication of data through a volatile field 6
1.4 Structure of JinjaThreads in comparison with Jinja’s 10
1.5 JinjaThreads stack of semantics 11
1.6 Example to illustrate the difference between inductive and coinductive definitions 20

2.1 JinjaThreads values and types 25
2.2 Abbreviations for common expressions 26
2.3 Type definitions for program declarations 27
2.4 Valid types of a program 30
2.5 The subtype relation 30
2.6 Hasse diagram for the subtype relation 31
2.7 Selected typing rules for JinjaThreads source code 32
2.8 Typing rules for the binary operators /, ==, and & 33
2.9 Generic well-formedness constraints 36
2.10 Well-formedness for JinjaThreads source code 36
2.11 Type definitions for the sequential state 37
2.12 Semantics of native method calls in the sequential part 39
2.13 Evaluation for the binary operators /, ==, and & 40
2.14 Small-step semantics for reducing method calls 41
2.15 Method bodies in bytecode 44
2.16 Type definitions for the sequential VM state space 46
2.17 Single-step semantics of the Invoke instruction 47
2.18 Combining normal execution and exception handling in the VM 48
2.19 Defensive checks for the Invoke instruction 49
2.20 Applicability and effect for the Invoke instruction 51
2.21 Example of a method well-typing 53
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Stack of semantics with interleaving</td>
<td>60</td>
</tr>
<tr>
<td>3.2</td>
<td>Implementation of lock operations</td>
<td>62</td>
</tr>
<tr>
<td>3.3</td>
<td>Wait sets, notification, and interruption</td>
<td>64</td>
</tr>
<tr>
<td>3.4</td>
<td>Two threads with locks, wait sets, and interrupts</td>
<td>65</td>
</tr>
<tr>
<td>3.5</td>
<td>Type definitions for basic thread actions</td>
<td>67</td>
</tr>
<tr>
<td>3.6</td>
<td>Update functions of the lock status for lock BTAs</td>
<td>69</td>
</tr>
<tr>
<td>3.7</td>
<td>Preconditions for lock BTAs</td>
<td>70</td>
</tr>
<tr>
<td>3.8</td>
<td>Update functions for temporarily released locks</td>
<td>70</td>
</tr>
<tr>
<td>3.9</td>
<td>Update functions for thread creation BTAs</td>
<td>71</td>
</tr>
<tr>
<td>3.10</td>
<td>Preconditions for thread creation BTAs</td>
<td>71</td>
</tr>
<tr>
<td>3.11</td>
<td>Predicates for condition actions</td>
<td>72</td>
</tr>
<tr>
<td>3.12</td>
<td>Update relation for wait sets</td>
<td>73</td>
</tr>
<tr>
<td>3.13</td>
<td>Update functions and predicates for interruption BTAs</td>
<td>74</td>
</tr>
<tr>
<td>3.14</td>
<td>Reductions in the interleaving semantics</td>
<td>76</td>
</tr>
<tr>
<td>3.15</td>
<td>Definition of locale lifting-wf</td>
<td>79</td>
</tr>
<tr>
<td>3.16</td>
<td>Signatures of native methods for Java concurrency</td>
<td>83</td>
</tr>
<tr>
<td>3.17</td>
<td>Semantics of native methods of class Thread</td>
<td>86</td>
</tr>
<tr>
<td>3.18</td>
<td>Static native methods of Thread</td>
<td>87</td>
</tr>
<tr>
<td>3.19</td>
<td>Semantics of the native methods wait, notify, and notifyAll</td>
<td>88</td>
</tr>
<tr>
<td>3.20</td>
<td>Steps for t executing a call to wait on object a</td>
<td>89</td>
</tr>
<tr>
<td>3.21</td>
<td>Semantics of synchronized blocks</td>
<td>94</td>
</tr>
<tr>
<td>3.22</td>
<td>Adaptations to the single-step semantics for the VM</td>
<td>97</td>
</tr>
<tr>
<td>3.23</td>
<td>Semantics of the instructions for monitors</td>
<td>98</td>
</tr>
<tr>
<td>3.24</td>
<td>Applicability and effect for MEnter and MExit</td>
<td>99</td>
</tr>
<tr>
<td>3.25</td>
<td>Three schedules that lead to deadlock</td>
<td>102</td>
</tr>
<tr>
<td>3.26</td>
<td>Thread t must wait for resource w indefinitely</td>
<td>103</td>
</tr>
<tr>
<td>3.27</td>
<td>Formal definition of deadlock</td>
<td>104</td>
</tr>
<tr>
<td>3.28</td>
<td>Coinductive definition of the set of threads in deadlock</td>
<td>106</td>
</tr>
<tr>
<td>3.29</td>
<td>Example with deadlocked threads</td>
<td>107</td>
</tr>
<tr>
<td>3.30</td>
<td>Definition of locale preserve-deadlocked</td>
<td>109</td>
</tr>
<tr>
<td>3.31</td>
<td>Definition of locale progress</td>
<td>110</td>
</tr>
<tr>
<td>3.32</td>
<td>Run-time typing rules for synchronized blocks</td>
<td>114</td>
</tr>
<tr>
<td>3.33</td>
<td>Definition of ok-I</td>
<td>118</td>
</tr>
<tr>
<td>4.1</td>
<td>Definition of locales heap-base and heap</td>
<td>134</td>
</tr>
<tr>
<td>4.2</td>
<td>Locales for heap conformance</td>
<td>135</td>
</tr>
<tr>
<td>4.3</td>
<td>The bootstrap process constructs the start heap</td>
<td>139</td>
</tr>
<tr>
<td>4.4</td>
<td>Semantics of field access and field assignment</td>
<td>139</td>
</tr>
<tr>
<td>4.5</td>
<td>Semantics of Getfield and Putfield</td>
<td>140</td>
</tr>
</tbody>
</table>
4.6 Happens-before provides a notion of time relative to a
given event .............................................. 144
4.7 Example program with data races and its JMM execution 145
4.8 Example of the value 42 appearing out of thin air ........... 145
4.9 Program with an execution in which a read sees an initial-
isation which occurs later in the program text .................. 147
4.10 Stack of semantics with the JMM .......................... 148
4.11 Dynamic dispatch requires type information that is not
yet available .............................................. 150
4.12 Type-unsafe program ................................... 151
4.13 Implicit communication via type information ............... 153
4.14 Thread spawn as an implicit communication channel ...... 153
4.15 Java implementation for the example in Figure 4.7 ........ 157
4.16 Well-formed executions for the program in Figure 4.15 .... 160
4.17 Release-acquire pairs .................................. 162
4.18 Legality constraints ..................................... 164
4.19 Program and two well-formed complete interleavings .... 169
4.20 Assumptions of the DRF guarantee and their decomposi-
tion over the stack of semantics ............................. 170
4.21 Locales formalising the set of allocated addresses .......... 172
4.22 Locale ka formalises that threads do not invent addresses 176
4.23 Locale ka-type combines known addresses with type in-
formation .................................................... 177
4.24 Two executions of complete interleavings .................... 187
4.25 Justifying execution for the type-unsafe program .......... 190
4.26 A program that the JMM allows to terminate with a raised,
but unallocated exception .................................. 192
4.27 Variation of Figure 4.25a ................................ 192
4.28 The race on Thread.start does not eliminate the data
race on y ...................................................... 194
4.29 Incorrectly synchronised program due to spurious wake-ups 196
4.30 An ill-formed program and its execution with a sequen-
tially consistent prefix followed by a read that cannot be
Cut, updated, and completed sequentially consistently .... 199

5.1 Structure of the compiler and its verification ................. 207
5.2 Simulation diagrams for delay bisimulations with explicit
divergence ................................................. 211
5.3 Definition of locales dbisim-div and dbisim-final .......... 212
<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Example of a delay bisimulation with explicit divergence that is not a well-founded delay bisimulation</td>
<td>212</td>
</tr>
<tr>
<td>5.5</td>
<td>Simulation diagrams for preservation of final states</td>
<td>213</td>
</tr>
<tr>
<td>5.6</td>
<td>Simulation diagrams for well-founded delay bisimulations</td>
<td>214</td>
</tr>
<tr>
<td>5.7</td>
<td>Non-transitivity of well-founded delay bisimulations</td>
<td>216</td>
</tr>
<tr>
<td>5.8</td>
<td>Executions as traces of observable moves</td>
<td>217</td>
</tr>
<tr>
<td>5.9</td>
<td>Locale $\tau$-multithreaded enforces that $\tau$-moves are unobservable</td>
<td>220</td>
</tr>
<tr>
<td>5.10</td>
<td>Locale $m$-dbisim-div collects the necessary assumptions for $\approx_m$ being a delay bisimulation with explicit divergence</td>
<td>221</td>
</tr>
<tr>
<td>5.11</td>
<td>Definition of call</td>
<td>228</td>
</tr>
<tr>
<td>5.12</td>
<td>Definition of inline</td>
<td>229</td>
</tr>
<tr>
<td>5.13</td>
<td>Single-threaded source code semantics with explicit call stacks</td>
<td>230</td>
</tr>
<tr>
<td>5.14</td>
<td>Definition of $\mathcal{B}$</td>
<td>238</td>
</tr>
<tr>
<td>5.15</td>
<td>Reduction rules for synchronized blocks in $J_1$</td>
<td>240</td>
</tr>
<tr>
<td>5.16</td>
<td>Reduction rules for call stacks in $J_1$</td>
<td>241</td>
</tr>
<tr>
<td>5.17</td>
<td>Register allocation for local variables, blocks, and synchronisation</td>
<td>242</td>
</tr>
<tr>
<td>5.18</td>
<td>Simulation theorems on the expression level for compilation stage 1</td>
<td>246</td>
</tr>
<tr>
<td>5.19</td>
<td>Example introduction rules for the relation $\approx$</td>
<td>254</td>
</tr>
<tr>
<td>6.1</td>
<td>Preorder traversal for binary trees in Isabelle/HOL and the extracted code</td>
<td>263</td>
</tr>
<tr>
<td>6.2</td>
<td>Program and data refinement for prefix</td>
<td>265</td>
</tr>
<tr>
<td>6.3</td>
<td>Original and alternative introduction rule of the small-step semantics</td>
<td>273</td>
</tr>
<tr>
<td>6.4</td>
<td>Tabulation for lookup functions and the subclass relation</td>
<td>276</td>
</tr>
<tr>
<td>6.5</td>
<td>Java2Jinja in the Eclipse IDE</td>
<td>283</td>
</tr>
<tr>
<td>7.1</td>
<td>Isar proof example</td>
<td>298</td>
</tr>
<tr>
<td>7.2</td>
<td>Class initialisation example</td>
<td>303</td>
</tr>
</tbody>
</table>
## List of Tables

2.1 Sequential JinjaThreads expressions .......................... 26  
2.2 Instructions of the sequential virtual machine ............... 45  

3.1 Events record memory and synchronisation operations of a thread ................................................. 84  
3.2 Expressions for synchronized statements ........................ 93  
3.3 Instructions for monitors ......................................... 98  

4.1 Distribution of lemmata, theorems, and corollaries from §3.2, §3.3.4, and §3.3.5 over the locales from Figures 4.1 and 4.2 ................................................................. 137  

6.1 Timing for running the producer-consumer example in the source code interpreter ......................... 278  
6.2 Timing for running the producer-consumer example in the virtual machine .................................. 279  

7.1 Formalisation size of JinjaThreads ............................... 292


Principles and practices of declarative programming (PPDP 2011),

concurrency memory model. In Proceedings of the 2008 ACM
SIGPLAN conference on Programming language design and

[33] Sascha Böhme and Tobias Nipkow. Sledgehammer: Judgement
Day. In Jürgen Giesl and Reiner Hähnle, editors, Automated
Reasoning (IJCAR 2010), volume 6173 of Lecture Notes in Computer

[34] John Boyland. An operational semantics including “volatile” for
Formal Techniques for Java Programs 2008.

[35] Deadlock in class initialization specification, JLS 2nd ed. 12.4.2.
Java bug database, ID 4891511 http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4891511,
2008.

[36] Lukas Bulwahn. Smart test data generators via logic
programming. In John Gallagher and Michael Gelfond, editors,
Technical Communications of the 27th International Conference on
Logic Programming (ICLP 2011), volume 11 of Leibniz International
Proceedings in Informatics (LIPIcs), pages 139–150. Schloss Dagstuhl–
Leibniz-Zentrum für Informatik, 2011.

[37] Alex Busenius. Mechanized formalization of a transformation
from an extensible Spi calculus to Java. Master’s thesis,
Information Security and Cryptography Group, Department of
Computer Science, Saarland University, 2011.

[38] Pietro Cenciarelli, Alexander Knapp, and Eleonora Sibilio. The
Java memory model: Operationally, denotationally, axiomatically.
In Rocco De Nicola, editor, Programming Languages and Systems
(ESOP 2007), volume 4421 of Lecture Notes in Computer Science,

language. In Proceedings of the 37th annual ACM


[60] Florian Haftmann. Data refinement (raffinement) in Isabelle/HOL. This is a draft of an envisaged publication still to be elaborated which, applying the usual rules of academic confidentiality, can be inspected at http://www4.in.tum.de/~haftmann/pdf/data_refinement_haftmann.pdf.


[65] Christian Hammer and Gregor Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control

374


[71] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23:396–450, 2001.


[130] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. µJava: Embedding a programming language in a theorem prover. In Friedrich L. Bauer and Ralf Steinbrüggen, editors, Foundations of


[169] Kohei Suenaga and Naoki Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In Rocco


Index

Symbols

& strict and operator, 25, 317
&& shortcut and operator, 25
−₁(→) inverse function, 163
−₁(→₁) inverse image on relations, 163
→ divergence, 216
/ division operator, 25, 317
× multiply operator, 26, 328
∈ hyperset membership, 330
∈ set membership, 15
@ list append, 15
% remainder operator, 25, 317
∧ bitwise xor operator, 25, 317
image operator, 15
| strict or operator, 25, 317
| restrict binary relation, 161
⁻¹(→₁) well-formedness of maps to invariant data, 80
∩ intersection of hypersets, 330
∪ union of hypersets, 330
∪ disjoint union, 15
| shortcut or operator, 25
() element of type unit, 236
⟨⟩ empty thread action, 68
[ ] empty list, 15
[ ] empty terminated list, 216
[] Some, 15
[ ] list element by index, 15
[ ] array type, 24
[ ] array cell access, 26, 328
[ ] := array cell assignment, 26, 328
{ , , } set comprehension, 15
{ , : , } uninitialised variable block, 26
{ , = , } local variable block, 26, 328
⟨⟩c projection to condition BTAs, 68
⟨⟩i projection to interruption BTAs, 68
⟨⟩l projection to lock BTAs, 68
⟨⟩o projection to events, 68
⟨⟩t projection to thread creation BTAs, 68
⟨⟩w projection to wait set BTAs, 68
⟨⟩:= local variable assignment, 26, 328
⟨⟩:= list update by index, 15
⟨⟩↦→ map update, 16
⟨⟩↦→ list of map updates, 16
⟨⟩↦→ FinFun map update, 16
⟨⟩↦→ list of updates of empty, 16
⟨⟩↦→ update of empty, 16
⟨⟩↦→ update of Kf None, 16
⟨⟩←→ list filtering, 15
⟨⟩↦→ extension with invariant data, 80
⟨⟩↦→ extension with invariant data, 80
⟨⟩⊢→ annotation rule for preprocessor, 357–359
⟨⟩⊢→ list extension of ⟨⟩⊢→ 359
_⇒ release-acquire pair, 162, 350
_⇒ most recent write, 351
⇒ implication, 14
⇒ partial function, map, 16
⇔ biimplication, 14
\[\tau\] transition, 210
\[\tau\] unobservable transition, 210
\[\tau\] possibly empty sequence of \(\tau\)-moves, 210
\[\tau\] non-empty sequence of \(\tau\)-moves, 210
\[\tau\] divergence, 210
\[\tau\] visible move, 210
\[\tau\] interleaving semantics, 76
\[\tau\] reflexive transitive closure of interleaving semantics, 77
\[\tau\] single-threaded semantics, 76
\[\tau\] single-threaded semantics with start and finish events, 155
\[\tau\] source code interleaving semantics, 95
\[\tau\] sequential aggressive VM, 48
\[\tau\] multithreaded aggressive VM, 100
\[\tau\] multithreaded aggressive VM, 100
\[\tau\] sequential defensive VM, 49
\[\tau\] single-threaded defensive VM, 97
\[\tau\] multithreaded defensive VM, 100
\[\tau\] multithreaded defensive VM, 100
\[\tau\] sequential source code semantics, 40
\[\tau\] list extension of \([\_\_\_\_]\rightarrow [\_\_\_\_]\), 40
\[\tau\] single-threaded source code semantics, 91, 332–338
\[\tau\] \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_]\)
J0 expression semantics, 230
\[\tau\] \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_]\)
J0 call stack semantics, 230
\[\tau\] \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_]\)
J1 expression semantics, 239
\[\tau\] \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_]\)
J1 call stack semantics, 239
\[\tau\] \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_]\)
JvM sequential method-level VM, 252
\[\tau\] \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_]\)
function space, 14
\[\tau\] \[\_\_\_\_] \rightarrow \[\_\_\_\_] \rightarrow \[\_\_\_\_]\)
meta-implication, 14
\[\tau\] update wait sets with BTA, 72, 73
\[\tau\] update wait sets with BTAs, 73
\[\tau\] restrict map, 234
\[\tau\] definite assignment for multithreaded states, 115
\[\tau\] lifts ok-I to multithreaded states, 118
\[\tau\] lifting for thread-local predicates, 78
\[\tau\] lifting for thread-local predicates with invariant data, 80
\[\tau\] run interleaving semantics until stuck, 157
\[\tau\] complete interleaving, 157
\[\tau\] execution starts in a state, 217
\[\tau\] execution with intermediate states, 218
\[\tau\] diverge, 217
\[\tau\] step, 217
\[\tau\] terminate, 217
\[\tau\] list length, 15
\[\tau\] list extension of \([\_\_\_\_]\rightarrow [\_\_\_\_]\), 40
\[\tau\] single-threaded source code semantics, 91, 332–338
heap conformance, 42
well-formed JMM execution, 162, 351
conformance of states, 42, 319
sequential bytecode conformance, 54
typability and conformance, 115
single-threaded bytecode conformance, 124
single-threaded bytecode conformance, 138
multithreaded bytecode conformance, 124
typability and conformance of multithreaded states, 115
non-speculative event list, 174
sequential consistency coinductively, 172
thread conformance, 114
JMM heap conformance, 321
field table conformance, 321
sequentially consistent heap conformance, 142, 320
object conformance, 320
preservation invariant for \textit{lock-conf$_1$}, 249
less than operator, 25, 317
left shift operator, 25, 317
less than or equal operator, 25, 317
induced total order, 161, 350
$\leq_{\text{Array}}$, 30
$\leq_{\text{Object}}$, 30
$\leq_{\text{eo}}$ execution order, 161, 350
$\leq_{\text{hb}}$ happens-before order, 144, 162, 350
$\leq_{\text{null}}$, 30
$\leq_{\text{null[]}},$ 30
$\leq_{\text{po}}$ program order, 144, 162, 350
$\leq_{\text{refl}}$, 30
$\leq_{\text{so}}$ synchronisation order, 162, 350
$\leq_{\text{subcls}}$, 30
$\leq_{\text{sw}}$ synchronises-with order, 162
$\vdash _{\text{strict}}$ direct subclass relation, 28, 316
$\vdash _{\text{strict}'}$ subclass relation, 29, 316
$\vdash _{\text{strict}}$ subtype relation, 30
$\vdash _{\text{strict}'}$ subtyping on state types, 52
extension of $\vdash _{\text{strict}}$ to lists, 30
typability and conformance of values, 42, 319
typability and conformance of stores, 42, 319
typability and conformance of lists of values, 42, 319
conformance under sequential consistency, 142
conformance of heap record, 178
inequality operator, 25, 317
order on maps, 16
heap extension, 115, 134
greater than operator, 25, 317
greater than or equal operator, 25, 317
signed right shift operator, 25, 317
unsigned right shift operator, 25, 317
equality operator, 25, 317
label relation between J and J$_0$, 233
label relation between source code and bytecode, 255
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊳</td>
<td>bisimulation relation on thread actions, 220</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>non-stuck thread, 104</td>
</tr>
<tr>
<td>⊳ ⊳ ⊳</td>
<td>thread requires resources, 104</td>
</tr>
<tr>
<td>⊳</td>
<td>same event except for values read or written, 163, 351</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>single-threaded bisimulation relation, 219</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>equal up to the first SC-inconsistency, 180</td>
</tr>
<tr>
<td>⊳ ⊳ ⊳</td>
<td>single-thread bisimulation with start and finish events, 226</td>
</tr>
<tr>
<td>⊳ ⊳ ⊳</td>
<td>central relation for bisimulation between $J_0$ and $J'_1$, 245</td>
</tr>
<tr>
<td>⊳</td>
<td>extension of $(\approx, \sim)$ to executions, 217</td>
</tr>
<tr>
<td>⊳ ⊳ ⊳</td>
<td>relation between $J_1$ and bytecode call frames, 253</td>
</tr>
<tr>
<td>⊳</td>
<td>bisimulation for expressions between $J_1$ and $J'_1$, 248</td>
</tr>
<tr>
<td>⊳ ⊳ ⊳</td>
<td>multithreaded bisimulation between $J_1$ and $J'_1$, 248</td>
</tr>
<tr>
<td>⊳</td>
<td>bisimulation for call stacks between $J_0$ and $J'_1$, 248</td>
</tr>
<tr>
<td>⊳</td>
<td>relation for threads in wait sets between $J_0$ and $J'_1$, 248</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>multithreaded bisimulation between $J$ and $J_0$, 233</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>single-threaded bisimulation between $J$ and $J_0$, 232</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>bisimulation relation between $J'_1$ and $J'_1$, 249</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>bisimulation between $J'_1$ and bytecode call frames, 254</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>single-threaded bisimulation between $J'_1$ and bytecode, 255</td>
</tr>
<tr>
<td>$\approx_{jvm}$</td>
<td>bisimulation between source code and bytecode, 255</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>bisimulation relation for multithreaded states, 219</td>
</tr>
<tr>
<td>⊳</td>
<td>relation for threads in wait sets, 219</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>threads in wait sets with start and finish events, 226</td>
</tr>
<tr>
<td>⊳</td>
<td>list consing, 15</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>method call, 26, 328</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>field access, 26, 328</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>semantics, 139</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>field assignment, 26, 328</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>semantics, 139</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>type system, 237</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>type system for $J_1$, 237</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>multiplication operator, 25, 317</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>reflexive transitive closure, 19</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>reflexive transitive closure for ternary relations, 72</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>addition operator, 25, 317</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>sum type, 15</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>map overwriting, 234</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>subtraction operator, 25, 317</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>removal from hyperset, 330</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>relational composition, 213</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>composition of bisimulations, 213</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>empty set, 15</td>
</tr>
<tr>
<td>⊳ ⊳</td>
<td>has method, 49</td>
</tr>
</tbody>
</table>
_ ⊢ _ has :: ( _) in _
  field lookup, 28, 317
_ ⊢ _ has-fields_
  compute field table, 317
_ ⊢ _ justified-by _ justification sequence, 164, 351
_ ⊢ _ legal
  legal JMM execution, 164, 352
_ ⊢ _ length _
  array length, 26, 328
_ ⊢ lub ( _ , _ ) = _
  least upper bound, 33
_ ⊢ _ sees :: ( _) in _ field lookup
  with visibility, 28, 317
_ ⊢ _ sees :: → _ = _ in _
  method lookup, 28, 316
_ ⊢ _ sees-methods_
  compute method table, 316

A
_ A_ definitely assigned variables, 330–331
_ A_ set of events of a complete interleaving, 161, 349
a2t conversion from address to thread ID, 84, 133, 134
Abs-prog wrap program declaration and tabulation, 276
abstract syntax
bytecode, 44–45
source code, 25, 93
acc ascending chain condition, 270
access location, 161
acq-events events for reacquiring temporarily released locks, 76, 154
Acquire, 76
acquire-lock acquire lock multiple times, 61, 62
acquire-TRL re-acquire temporarily released locks, 77
acyclic-hierarchy acyclic class hierarchy, 269
Addr address value, 24, 315
addr address as expression, 26
  type of addresses, 24, 319, 321
  JMM implementation 2, 151
Address address constructor for JMM', 151, 321
address
allocated, 170
invent, 175
known, 175
addr-of-sys-xcpt address of preallocated system exception, 38, 139
ALength push array length, 45, 339
all-final-except all threads final except, 104
alloc allocation operation, 133
Allocate allocation event, 84, 143
allocated
  assumptions about allocated addresses, 172
  set of allocated addresses, 170
allocated addresses, 170
allocated-base locale for allocated addresses, 172
ALoad fetch array cell, 45, 339
annotate annotate expression, 359
annotate-Mb annotate method declaration, 359
annotate-Prog preprocessor to annotate program declarations, 359
Any catch all exceptions, 44
app applicability, 51, 343
append list concatenation, 15
appi applicability of instruction, 51, 342–343
  MEnter, 99
  MExit, 99
applicability, 51
ArithmeticException, 28
Arr array on the SC heap, 37, 38, 319
Array array type, 24, 315
ArrayIndexOutOfBoundsException array index out of bounds exception, 28
array-length-of extract array length from dynamic type information, 38, 315
ArrayStore array store exception, 28
ArrayT dynamic type information for arrays, 38, 315
A list extension of A, 331
ascending chaining condition, 270
AStore set into array cell, 45, 339

B
B correct block numbering, 238
B1, 254
B2, 254
B3, 254
B4, 254
base type, 29
base-type strips off Array, 29, 316
basic thread action, 66
semantics, 69
biimplication, 14
BinOp instruction for binary operators, 45, 339
binop semantics of binary operators, 39, 40, 318–319
binop-relevant-class possibly raised exceptions by binary operator, 341
(_, _) bisim type of bisimulation relations, 212
bisimilar
delay with explicit divergence, 211
bisimulation
delay with explicit divergence, 210
strong, 211
weak, 224
well-founded delay, 214
blank object with default values in fields, 320
blocks surround expression with initialised local variable blocks, 41, 338
blocks1 surround expression with uninitialised local variable blocks, 241
Bool boolean value, 24, 315
bool type of truth values, 14
Boolean type for booleans, 24, 315
bop type of binary operators, 25, 317
bop-ret type of binary operator results, 39
Bs list extension of B, 238

C
C&U cut-and-update property, 180
call called method, 228
call frame, 46
call stack
normalised, 233
call1 called method in J1, 239
calls list extension of call, 228
case _ of _ ⇒ _ | _ ⇒ _
case distinction, 16
Cast checked type cast, 26, 328
cdecl type of class declarations, 27, 315
Cell array cell member, 134
cells type of array cells, 37, 319
check checks of the defensive VM, 49
Checkcast checked type cast, 45, 339
check-instr instruction-specific defensive check, 49
Class class type, 24, 315
class, 27
class retrieve class declaration, 28, 315
_class type of classes, 27, 315
class declaration, 27
  well-formedness, 35
ClassCast class cast exception, 28
classes extract class declarations, 35, 315
class-of class at which lookup starts, 33
class-of' class at which lookup starts, 42
ClassT class name as dynamic type information, 38, 315
ClearIntr BTA to clear interrupt status, 67, 74
Clone, 39
clone method clone
  semantics, 39, 139, 323
  signature, 34
CloneF, 39
closed expression, 232
  under functional, 21
cname type of class names, 24
cname-of class name of object, 347
cnt-IB count initialised local variable blocks, 247
codatatype, 22
code extraction
  guidelines, 280
  locales, 267
code generator, 263
coinduction rule
  strengthened, 21
coinductive definition, 19–22
  monotonicity, 20
collapse collapse call stack by inlining, 232
committed events, 163
compC generic compiler for classes, 352
compE1 compiler stage 1 for expressions, 242, 352–353
compE2 compiler from expressions to instructions, 250, 353–354
compEs2 compiler from expression lists to instructions, 354
complete interleaving, 157
  example, 158–160
  non-speculative, 175
  set of events, 161
  well-formedness, 163
completion sequentially consistent, 179
compM generic compiler for methods, 352
compMb2 compiler stage 2 for method bodies, 250, 357
compP generic program compiler, 243, 352
compP' generic compiler for tabulated programs, 276
compP1 first compiler stage, 207, 243, 353
compP2 second compiler stage, 207, 250, 357
compP-impl generic compiler with tabulation, 276
compxE2 compiler from expressions to exception tables, 250, 354–355
compxEs2 compiler from expression lists to exception tables, 355
concat list flattening, 15
condition action
  semantics, 71–72
  syntax, 67
  condition-act
    type of a condition BTA, 67
conf heap conformance assumptions, 135
conf-base heap module with heap conformance, 135
conflict, 165
conformance, 42, 319
  bytecode, 54
heap record, 178
lock, 117
properties of heap, 135
thread, 114
conf-progress progress assumptions on heap module, 135
conf-read locale for reading conformation values, 135
consistency, 52
Java memory model, 185
under functional, 21
copy-mem
  copy one heap location, 323
copy-mems
  copy heap locations, 139, 323
correctly synchronised, 165
currentThread
  method currentThread
    semantics, 87
    signature, 83
CurrTh, 87
cut-and-update property, 180
generalised, 182

D
definite assignment check, 37, 331
Dacquire, 106
Dactive, 106
data race, 165
data race free, 165
data refinement, 264
datatype, 22
dbisim-base
  locale for bisimulations, 212
dbisim-div assumptions of delay bisimulations with explicit divergence, 212
dbisim-final assumptions for preservation of final states, 212
deadlock
  example, 101, 106
  state, 104
  thread, 105
deadlock state in deadlock, 104
deadlock' all threads in deadlock, 107
deadlocked thread in deadlock, 106
declaration, 27
default value, 161, 319
default-val default value, 319
definite description operator, 15
delay bisimilar
  with explicit divergence, 211
delay bisimulation
    well-founded, 214
    with explicit divergence, 210
    example, 211
disjoint sum, 15
distinct
  distinctness of list elements, 15
divergence, 210
dom domain, 16
DRF guarantee, 166, 184
drop suffix of list, 15
dropWhile drop list elements, 273
Ds list extension of $D$, 331
Dup duplicate top stack element, 45, 339
Dwait, 106

E
set of complete interleavings, 158
ε_ _ indefinite description operator, 15
code extraction, 280
eff effect, 51, 342
effi effect of instruction, 51, 340–341
MEnter, 99
MExit, 99
empty
everywhere undefined map, 16
empty-heap empty heap, 133
Index

env  type of environments, 32
environment, 32
Err  error type, 50, 339
_err  type of the error type, 339
Event  coercion into type _sf, 155
event, 143
  committed, 163
  conflicting, 165
  non-speculative list of, 174
  renaming, 163
  synchronisation, 162, 349
  thread finish, 155
  thread start, 155
event  type of events, 83
events  extract and pair events with
  thread ID, 157
events’  extract events, 177
exception flag, 46
exception handler, 44
exception table, 44
exception table entry, 44
exec  single-threaded aggressive VM, 48, 96, 97, 347
execd  single-threaded defensive
  VM, 49
exec-instr  semantics of instructions, 47, 96, 97, 343–347
exec-meth  semi-aggressive method-
  level VM, 207, 252
exec-methd  defensive method-level VM, 252
exec-native  aggressive implementa-
  tion of native methods, 47, 96,
  326–327
execution
  Java memory model, 162
  justifying, 146
  legal, 163, 352
  sequence of justifying, 146, 163
  well-formedness, 162
(_, _) execution
  type of a JMM execution, 349
type of executions, 216
execution order, 161
ex-entry  type of exception table
  entries, 44, 339
(_, _, _) exp
  type of expressions, 236, 328
_expr  type of source code expres-
  sions, 25, 236, 328
_expr1  type of J1 expressions, 237
ex-table  type of exception tables, 44, 339
ex-table-of  lookup exception table
  for method, 44, 347
extension
  heap, 115, 134
  map, 16
  with invariant data, 80
Extern  externally observable event,
  84, 143

F

_<f_  FinFun application, 16
false  False as expression, 26
fdecl  type of field declarations, 27,
  315
Field  field member, 134
field  functional field lookup, 317
field declaration, 27
field modifier, 27
field table, 38
fields  functional field table lookup, 317
type of field tables, 37, 319
filter  list filtering, 15
final
  intermediate language, 241
  J0, 231
  multithreaded state, 77
  preservation of states, 213
  thread, 77
  virtual machine, 46, 99
Index

final final expression, 40, 72, 76
final0 final J0 state, 231
final-thread
  locale for parameter final, 72
final-threads set of final threads, 77
FinFun almost everywhere constant function, 16, 266
Finished terminal thread status, 155
finite finiteness of sets, 15
fit to, 175
fmod type of field modifiers, 27, 315
fold left fold on lists, 15
foldr right fold on lists, 15
frame type of a call frame, 46
free variable, 232
free-thread-id free thread ID, 71
fresh-vname fresh variable name, 243, 352
fst first projection, 15
fv set of free variables, 232

G
gC&U generalised cut-and-update property, 182
gen-new-Addr search algorithm for fresh addresses, 273
Getfield load field, 45, 339
  semantics, 140
gfp greatest fixed point, 20
Goto jump instruction, 45, 339

H
happens-before order, 144, 162
hash-addr hash function on addresses, 39, 320, 322
hashcode method hashcode, 138
  semantics, 39
  signature, 34
has-I holds some lock, 117
has-Is list extension of has-I, 117
has-lock test whether lock is held, 61
has-locks number of times a lock is held, 61, 62
hconf heap conformance, 135
hd head of list, 15
heap
  type of heaps, 37, 319
  type-level assumptions of heap module, 134
heap conformance
  JMM implementation 1, 149
  JMM implementation 2, 152
  properties, 135
  sequential consistency, 142
heap extension, 115, 134
heap module, 133–136
  design considerations, 140–141
  JMM implementation 1, 148
  JMM implementation 2, 151
  operations, 133
  sequential consistency, 141
heap record, 174
  conformance, 178
heap-clone clone object or array, 323
heap-base locale of heap operations, 134
heap-entry type of object and array representations, 37, 319
Hilbert’s ε-operator, 15
hp project J state to heap, 38
hty type of dynamic type information, 38, 315
_ hyperset type of hypersets, 330

I
I number of times a lock is held, 117
i_. _ definite description operator, 15

398
if _else_ conditional, 26, 328
IfFalse conditional jump instruction, 45, 339
IllegalMonitorState illegal monitor state exception, 83
IllegalThreadState illegal thread state exception, 83
impl-of unpack program declaration with tabulation, 276
indefinite description operator, 15
index list index of last occurrence, 243, 352
induced total order, 161
inductive definition, 19–21
monotonicity, 20
infinite stuttering, 211
inheritance locale, 18
init predicate for initialisation events, 161, 349
init-fields initialise field table with default values, 320
initialisation of location, 146
inj-on injective on, 163
Inl left injection, 15
inline inline call, 227, 229
inlines list extension of inline, 229
Inr right injection, 15
InstanceOf subtype test, 45, 339
instanceof subtype test, 26, 328
instr type of instructions, 44, 339
instrs-of lookup instructions for method, 44, 347
instruction set, 44
insync _synchronized block under execution, 93, 328
intermediate language, 237
semantics, 94
syntax, 93
int integers, 14
Integer type for integers, 24, 315
interpretation locale, 17
interrupt, 64
interrupt method interrupt, 64
semantics, 86
signature, 83
interrupted method interrupted semantics, 87
signature, 83
InterruptedException, 83
interruption
BTA semantics, 73–74
BTA syntax, 67
interaction with wait-notify, 90–91
semantics, 86
Intg integer value, 24, 315
Intr BTA to set interrupt status, 67, 74
INTR, 86
_int-act type of an interrupt BTA, 67
INTRD, 87
INTRDF, 87
INTRDT, 87
INTRINEX, 86
intrs projection to pending interrupts, 65
_intrs type of pending interrupts, 64
intr-waits interrupts that a thread action waits for, 102
invariant closedness under a relation, 81
invent address, 175
Invoke method call instruction, 45, 339
semantics, 97
InWS wait set status “in wait set”, 63
Isabelle Collections Framework, 267
is-addr is of the form addr _, 227
is-class  predicate for declared class names, 28, 315
is-htype  valid dynamic type information, 134, 316
isInterrupted  method isInterrupted semantics, 86
signature, 83
IsIntrd  BTA to query interrupt status, 67, 74
IsIntrdT, 86
is-Ref  reference value or null pointer, 49
is-refT  predicate for reference types, 24
is-relevant-class  possibly raised exceptions, 51, 341
is-relevant-entry  possibly matching exception table entry, 341
is-Throw  thrown exception, 339
is-Throws  thrown exception in method parameters, 273
is-type  valid type, 29, 30, 316
is-Val  is a value, 117
is-Vals  is a list of values, 227
is-volatile  volatile member, 161, 317

J
J  JinjaThreads source code, 23
J.  prefix for source code instantiation, 95
J.ka  addresses a source code thread knows, 178
J.redT  source code interleaving semantics, 95
J0  source code semantics with explicit call stack, 227
J0.redT  multithreaded J0 semantics, 231
J0-red  J0 call stack semantics, 230
J0-start  J0 start state, 231
J0-τ-move  unobservable transition in J0, 231
J1  intermediate language, 236
J′  J1 with unlocking possibly failing, 241
J′_1.redT  multithreaded J′_1 semantics, 241
J′_1  J1 with unlocking never failing, 241
J′_1.redT  multithreaded J′_1 semantics, 241
J1-final  final J1 state, 241
-J1-prog  intermediate language program, 237
J1-red  J1 call stack semantics, 239
J1-start  J1 start state, 241
J1-τ-move  unobservable transition in J1, 242
J2JVM  compiler, 207, 255, 352
Java memory model
informal, 143
Java2Jinja, 282
J-final  final J state, 95
J-mb  type of J method bodies, 27, 328
jmm  prefix for JMM heap implementation 1, 149
JMM heap implementation 1, 148
JMM heap implementation 2, 151
jmm′  prefix for JMM heap implementation 2, 152
jmm′-a2t  conversion from address to thread ID, 322
jmm′-allocate  JMM′ allocation operation, 322
jmm′-empty-heap  empty JMM′ heap, 322
jmm′-hconf  JMM′ heap conformance, 322
jmm′-heap  type of JMM′ heap, 152, 321
jmm′-read  JMM′ read operation, 322
Index

jmm’-t2a  conversion from thread ID to address, 322
jmm’-typeof-addr  dynamic type information under JMM’, 322
jmm’-write  JMM’ write operation, 322
jmm-a2t  conversion from address to thread ID, 321
jmm-allocate  JMM allocation operation, 321
jmm-empty-heap  empty JMM heap, 321
JMM-event  type of identifiers of a JMM event, 349
_jmm-heap  type of JMM heap, 321
jmm-new-Addr  JMM operation for fresh addresses, 321
jmm-read  JMM read operation, 149, 321
jmm-t2a  conversion from thread ID to address, 321
jmm-typeof-addr  dynamic type information under JMM, 149, 321
jmm-write  JMM write operation, 321
Join  BTA to join on a thread, 67, 72
Join, 86
join  method join semantics, 86
signature, 83
JOININTR, 86
J-prog  type of J program declarations, 27, 328
J-red  source code single-threaded semantics, 95
J-start  initial J state, 95, 96
J-state  type of sequential source code states, 37
J-τ-move  unobservable transition in J, 231
justifying execution, 146
example, 188
sequence of, 146
justifying sequence example, 186
jvm.  prefix for aggressive VM instantiation, 100
jvm.ka  addresses a bytecode thread knows, 178
jvm.redT  multithreaded aggressive VM, 100
jvmd.  prefix for defensive VM instantiation, 100
jvmd.redT  multithreaded defensive VM, 100
jvm-exec  single-threaded aggressive VM, 99
jvm-execd  single-threaded defensive VM, 99
jvm-final  final VM state, 99
jvm-method  type of bytecode method bodies, 44, 339
jvm-prog  type of bytecode program declarations, 44, 339
jvm-state  type of single-threaded VM states, 46
jvm-τ-move  unobservable JVM transition, 348
jvm-wf-states  well-formed JVM states for type safety, 124
J-wf-states  well-formed J states for type safety, 120

K
ka  addresses a thread knows, 175
assumptions about ka, 176
ka-base  locale for operation known addresses, 176
kas  addresses known in a state, 176
Index

ka-type locale connecting allocation, known addresses and conformance, 177, 178
K\textsuperscript{f} constant FinFun, 16
known addresses, 175
multithreaded state, 176

L
labelled transistion system, 210
Lazy lazy list constructor, 265
lcl project J state to local store, 38
Leaf leaf of a binary tree, 263
learn
address, 175
learns learnt addresses, 175
least upper bound, 33
computation, 271
legal execution, 352
legality, 163
lfp least fixed point, 20
lifting-inv assumptions for lifting thread-local predicates with invariant data, 81
lifting-wf assumptions for lifting thread-local predicates, 79
list, 15
_list type constructor for lists, 15, 22
_llist type of possibly infinite lists, 157
llist-corec corecursion operator for lazy lists, 22
Load load from register, 45, 339
loc type of a member, 134
locale
code extraction, 267
declaration, 17
inheritance, 18
interpretation, 17
locals type of local store, 37
location, 134
loc-default default value for location, 350
Lock BTA to acquire a lock once, 66, 67, 69, 70
lock, 61
temporarily released, 62
_lock type of a lock, 61
lock conformance, 117
_lock-acts type of lock BTAs, 68
lock-conf lock conformance, 117
lock-conf\textsubscript{1} lock conformance for J\textsubscript{1}, 249
locking
BTA semantics, 69–70
BTA syntax, 66
lock-lock acquire lock once, 61, 62
locks projection to locks, 65
(_,_ ) locks type of a lock state, 62
locs set of accessed locations, 161, 350
locs' locations accessed by an event, 349
(_,_ ) lts type of labelled transition systems, 212
lub least upper bound, 33
computation, 271

M
m-allocated assumptions about allocated addresses, 170, 172
map, 16
map list-element-wise application, 15
map to invariant data, 80
map-of conversion from list to map, 16
matches-ex-entry matching exception table entry, 347
match-ex-table find matching handler in exception table, 347
max maximum operation, 352
max-stack maximum stack size, 250, 356–357
max-stacks
   list extension of max-stack, 357
max-vars  maximum nesting depth of local variables, 241, 355–356
max-vars
   list extension of max-vars, 356
may-acquire-TRL precondition for re-acquiring temporarily released locks, 77
may-lock   test whether lock may be acquired, 61, 62
m-dbisim-div locale for lifting bisimulations, 221
m-decl     type of method declarations, 27, 315
MEnter    instruction monitorenter, 98, 339
           applicability, 99
           effect, 99
           semantics, 98
method
   functional method lookup, 48
method body, 27
   bytecode, 44
   source code, 27
method declaration, 27
   native, 27
   well-formedness, 35
method type, 50
method well-typing, 50
mexception
   extract exception flags, 257
MExit     instruction monitorexit, 98, 339
           applicability, 99
           effect, 99
           semantics, 98
m-final    final multithreaded state, 77
m-name    type of method names, 24
mode, 265, 281
   mode annotation, 265
monotonicity
   (co)inductive definition, 20
move, 210
mrw       update snapshot heap, 172
mrws      list fold of mrw, 172
mr-move   silent reduction of the interleaving semantics, 218
multithreaded   well-formedness assumptions about single-threaded semantics, 78
multithreaded-base locale for interleaving semantics, 76
must-wait  wait for resource indefinitely, 103
mutually exclusive, 61
MWintr, 103
MWjoin, 103
MWlock, 103

N
nat    natural numbers, 14
Native   body of native method, 27
native method
   semantics, 39, 84, 138, 153, 324–327
   signatures, 34, 82, 153, 322
native-BTA2J convert thread creation BTA to J, 92
native-BTA2jvm convert thread creation BTA to JVM, 97
native-Ret2J convert native method result to J, 41, 92
native-Ret2jvm convert native method result to JVM, 47, 96
native-TA2J convert thread action to J, 92
native-TA2J0 convert thread action to J0, 230
native-TA2J1 convert thread action to J1, 239
native-TA2jvm convert thread action to JVM, 97
Index

NChashc, 39

NegativeArraySize  negative array size exception, 28

New  object allocation, 45, 339

new  object allocation, 26, 328

new_[_]  array allocation, 26, 328

new-Addr  fresh address, 39, 273, 320

New-Arr  abbreviation for

Allocate_ (ArrayT_ _), 84, 143

NewArray  array allocation, 45, 339

New-Obj  abbreviation for

Allocate_ (ClassT_ _), 84, 143

(_, _, _)  new-thread-act  type of a thread creation BTA, 67

no-call  no non-native method call, 228

Node  inner node of a binary tree, 263

non-speculative  complete interleaving, 175

  event list, 174

None, 15

Normal  normal state of the defensive VM, 49

NORMAL, 76

normalised call stack, 233

norm-eff  effect of normal execution, 342

not-final-thread  existing, non-final thread, 103

Notified  BTA to process notification, 67, 73

Notify  BTA to notify thread in wait set, 67, 73

notify  method notify

  semantics, 88

  signature, 83

NotifyAll  BTA to notify all threads in wait set, 67, 73

notifyAll  method notifyAll

  semantics, 88

  signature, 83

NT  type for null, 24, 315

Ntf, 88

NtfAll, 88

NtfAllFail, 88

NtfFail, 88

Null  null reference, 24, 315

null  null expression, 26

NullPointerException  null pointer exception, 28

O

Obj  object on the SC heap, 37, 38, 319

Object  root of class hierarchy, 28

observable behaviour

  bytecode, 252

  intermediate language, 242

  Java memory model, 153

  source code, 231

OK  usable register, 50, 339

ok-cond  precondition of condition BTA, 72

ok-conds  precondition of condition BTAs, 72

ok-I  insync (_, _) only in reducing branch, 117, 118

ok-init_  thread start events precede other events, 162, 350

ok-intr  precondition of interruption BTA, 73, 74

ok-intrs  precondition of interruption BTAs, 73, 74

ok-Js  list extension of ok-I, 118

ok-L  precondition of lock BTA, 69, 70

ok-locks  precondition of FinFun of lock BTAs, 69, 70

ok-locks-thr  only existing threads hold locks, 110

ok-Ls  precondition of lock BTAs, 69, 70
ok-Suspend  constraint on threads in wait sets, 120
ok-ta  precondition of a thread action, 74
ok-ta’ precondition of thread action except missing resources, 111
ok-thr  precondition of thread creation BTA, 70, 71
ok-thrs  precondition of thread creation BTAs, 70, 71
ok-val  inverse of OK, 339
ok-wset-final waiting threads exist and are not final, 110
ok-wsets  precondition for wait set BTAs, 73
_opstack  type of the operand stack, 46
_option, 15
Option.map  map operator for _option, 316
out-of-thin-air  guarantee, 189
value, 146
OutOfMemory  out of memory error, 28

P

pair, 15
pair-Pre-Start  thread action transformer for start and finish events, 155
pause at a call, 227
pc  type of the program counter, 46
Pop  pop from stack, 45, 339
predicate compiler, 265
prefix  prefix relation on lists, 265
preorder  pre-order traversal of binary tree, 263
preservation, 42
  of final states, 213
  semantics, 208
  stage 1, 244

stage 2, 252
under heap changes, 220
well-formedness
  compiler stage 1, 243
  compiler stage 2, 251
preserve-deadlocked  locale for deadlock preservation, 109
Pre-Start  initial thread status, 155
primitive type, 24
print  method print, 153
_prog  type of program declarations, 27, 315
_ProgDecl  extract declaration from tabulation, 276
_prog-impl  type of tabulated program declarations, 276
_prog-impl-invar  invariant for tabulation, 276
Program  constructor for program declarations, 27, 315
program declaration, 27
program order, 144, 162
program refinement, 264
program typing, 50
progress, 42
progress  locale for progress up to deadlock, 110
Push  push value onto stack, 45, 339
Putfield  store into field, 45, 339
  semantics, 140

R

_R_  set of read events, 161, 349
_r_ single-threaded semantics, 76
R_0call, 230
R_0red, 230
R_0ret, 230
R_1call, 241
R_1lock, 240
R_1red, 241
R_1ret, 241
Index

Rsync1, 240
Rsync2, 240
RsyncN, 240
RsyncX, 240
Runlck, 240
RunlckF, 240
RunlckN, 240
RunlckX, 240
RunlckXF, 240
RunlckXN, 240
ran map range, 16
random scheduler, 275
range range of a function, 15
Rcall, 41
RcallN, 41, 92
Read read event, 84, 143
read heap read operation, 134
read event, 161
set of, 161
read from location, 161
record, 15
Red0, 230
red0 \( J_0 \) expression semantics, 230
red1 \( J_1 \) expression semantics, 239
redT interleaving semantics, 76
re-entrant, 61
reference type, 24
refinement
data, 264
program, 264
reflexive transitive closure, 19
registers, 46
layout, 237
registers type of the registers, 46
Release BTA to temporarily release a lock, 66, 67, 69, 70
release-acquire pair, 162
release-lock
completely release lock, 61, 62
relevant-entries filter exception table for relevant entries, 341
replicate repetition as list, 15
resource, 101
restriction of a map, 16
Ret-sys-xcpt system exception raised by native method, 39
Ret-unchanged call native method again, 88
Return return from method, 45, 339
return type, 27
Ret-Val return value of native method, 39
Ret-Xcp exception raised by native method, 39
rev list reversal, 15
Rfacc, 139
Rfass, 139
Rlock, 94
Rnative, 41, 92
Robj, 41, 92
RobjX, 41
round-robin scheduler, 275
Rparam, 41
RparamX, 41, 273
RparamX2, 273
Rsync1, 94
Rsync2, 94
RsyncN, 94
RsyncX, 94
run method run
signature, 84
run-time type system, 42
synchronized, 114
Runlck, 94
RunlckX, 94
Running thread status running, 155

S
S no access to monitor registers, 238
SC sequential consistency, 165
sc prefix for sequential consistency, 142
SC’ sequential consistency coinductively, 172
sc-a2t conversion from address to thread ID, 319
sc-allocate allocation operation for SC heap, 320
scc sequentially consistent completion, 180
sc-empty-heap empty heap, 320
sc-heap sequentially consistent heap, 142
schedule scheduling function, 275
scheduler, 275
random, 275
round-robin, 275
sc-read sequentially consistent read operation, 142, 320
sc-t2a conversion from thread ID to address, 319
sc-typeof-addr sequentially consistent dynamic type information, 142, 320
sc-write sequentially consistent write operation, 142, 320
semantic preservation, 208
( _, _, _, _, _ ) semantics type of single-threaded semantics, 76
sequence of justifying executions, 163
sequential consistency coinductively, 172
heap module, 141
Java memory model, 165
sequentially consistent completion, 179
set, 15
set conversion from lists to sets, 15
_ set type constructor for sets, 15
_ sf type of start and finish events, 155
sf prefix for single-threaded semantics with start and finish events, 156
sf-acq-events coerced events for reacquiring temporarily released locks, 155
sf-final final state in transformed semantics, 156
sf.r single-threaded semantics with start and finish events, 155
sf.τ-move silent transition with start and finish events, 226
SFinish, 155
SFINIT, 155
SFrun, 155
shr projection to shared memory, 65
silent transition, 210
SLock lock acquisition event, 84, 143
snapshot heap, 172
initial, 180
snd second projection, 15
Some, 15
Spawn BTA to spawn new thread, 67, 71
Start, 86
start method start semantics, 86
signature, 83
start snapshot heap, 180
start state
bytecode, 100
intermediate language, 241
J₀, 231
source code, 95
well-formedness, 96, 138
start-addrs addresses of preallocated objects, 138
start-data constructs the start heap, 138, 139
start-ETS initial map to invariant data for type safety, 113
start-events events for bootstrapping, 158
StartFail, 86
start-H start snapshot heap, 180
Index

**start-heap**
initial heap, 38, 96, 138, 139

**start-ok** successful preallocation of objects, 138

**start-tID** thread ID of initial thread, 138, 139

**state**
multithreaded, 61, 64
source code, 37, 94
terminal, 216
virtual machine, 46

\((_,_,_,_,_)_state\) type of multithreaded states, 65

**state type**, 50

state well-typing, 50

**status** type of thread status, 155

**Store** store into register, 45, 339

strong bisimulation, 211
subclass relation, 29
subtype relation, 30

**subtype-measure**
measure on subtyping, 270

**succs** successor instructions under normal execution, 339–340

**SUnlock** lock release event, 84, 143

**Suspend**
BTA to suspend to wait set, 67

**Swap** swap top stack elements, 45, 339

**sync (\(_\)_)** synchronized block, 328

exception table, 251
instructions, 250
intermediate language, 237
semantics, 94
syntax, 93
typing rule, 93

**sync-event**
synchronisation event, 349

**sync-events** synchronisation events of an execution, 349

synchronisation correct, 165
synchronisation event, 162, 349
synchronisation order, 162
synchronises-with order, 162

**synchronized block**
semantics, 93–94
syntax, 93
typing rule, 93

**system class**, 28

**system classes**
well-formedness, 35

**system exceptions**, 28, 83

**sys-xcpts**
system exceptions, 28, 83, 327

T

t2a conversion from thread ID to address, 84, 133, 134

**tab-class** tabulate class lookup, 276

**tab-subcls**
tabulate subclass relation, 276

**tabulate**
tabulate lookup functions, 276

**take** prefix of list, 15
terminal state, 216

**TFinish** thread termination event, 143, 155

**the** inverse of Some, 16

**the-Addr** inverse of Addr, 348

**the-Array** inverse of Array, 339

**this** reference to the current object, 24

**thr** projection to thread pool, 65

**Thread** thread class, 84

thread action, 66
design considerations, 75
semantics, 74–75
syntax, 66–68

**thread conformance**, 114

**thread creation**
BTA semantics, 70–71
BTA syntax, 66
semantics, 85
thread finish event, 155
thread pool, 62
thread start event, 155
thread-local state
  intermediate language, 239  
  \(J_0\), 227
  source code, 94
  virtual machine, 99
\((\_\_\_\_\_\_)\) thread-action
  type of a thread action, 68
ThreadEx  BTA to test whether thread
  exists, 67, 71
thread-id
  type of thread IDs, 84, 319, 321
THROW
  thrown system exception, 41
Throw
  thrown exception, 26
throw
  exception throwing, 26, 328
Throwable
  root class in exception
  hierarchy, 28
ThrowExc
  throw exception, 45, 339
TInit
  initial thread event, 143, 155
\(\tau\)-instr
  unobservable instructions, 348
TIntr
  interruption event, 84, 143
TIntrd
  event for observed interruption, 84, 143
TJoin
  join event, 84, 143
\(\tau\)_jvm
  unobservable single-threaded
  JVM state, 348
tl
  tail of list, 15
\((\_\_\_\_\_\_)\) tllist
  type of terminated possibly
  infinite lists, 216
tmap
  map operator on \((\_\_\_\_\_\_)\) tllist, 297
\(\tau\)-move, 210
\(\tau\)-move
  unobservability of transition, 210
  unobservable transition in \(J\), 231, 338
\(\tau\)-move\(_1\)
  unobservable transition in
  \(J_1\), 242
\(\tau\)-moves
  list extension of \(\tau\)-move, 339
\((\_\_\_\_\_\_\_)\) \(\tau\)-moves
  type of \(\tau\)-move predicates, 220
\(\tau\)-multithreaded
  locale for \(\tau\)-moves, 220
\(\tau\)native
  unobservable native method
  call, 327
transition, 210
tree
  type of binary trees, 263
true
  \(True\) as expression, 26
try _ catch\((\_\_\_\_\_\_)\)
  exception handling, 26, 328
TStart
  thread spawn event, 84, 143
ty
  type of JinjaThreads types, 24, 315
\(\tau\)_i
  instruction typing, 339
\(\tau\)-of
  extract type from dynamic
  type information, 38, 315
type, 24
  primitive, 24
  reference, 24
  valid, 29
type checking, 271
type inference, 271
type judgement, 32
type safety
  bytecode, 122
  source code, 112
type system
  run-time, 42
  synchronized, 114>TypeError
  type error in the defensive VM, 49
_ type-error
  adds a type error state, 49
typeof
  dynamic type of a value, 38
typeof
  type of a value, 24
typeof-addr
  dynamic type information
  of an address, 38, 133
types, 14

types  set of valid types, 29, 316
typesafe  type safety assumptions on heap module, 135

U

uhr  update heap record, 174
uhrs  list fold of uhr, 174
undefined-Val  undefined value, 48
Unit  dummy value, 24, 315
unit  dummy expression, 26
unit  type with one element (), 236
UNIV  set of all elements, 15
Unlock  BTA to unlock a lock once, 66, 67, 69, 70
UnlockFail  BTA to test for not having locked a lock, 66, 67, 69, 70
unlock-lock  release lock once, 61, 62
unobservable transition, 210
upd-acq  update state when re-acquiring temporarily released locks, 77
upd-int  update interrupt status with BTA, 73, 74
upd-ints  update interrupt status with BTAs, 73, 74
upd-L  semantics of lock BTA, 69
upd-locks  semantics of FinFun of lock BTAs, 69
upd-Ls  semantics of lock BTAs, 69
upd-ta  execute thread action, 75
upd-thr  update thread pool with thread creation BTA, 70, 71
upd-thrs  update thread pool with thread creation BTAs, 70, 71
upd-trl  update single temporarily released lock, 70
upd-TRL  update FinFun of temporarily released locks, 70
upd-trl  update single temporarily released lock, 70

V

Val  literal value, 26, 328
val  type of values, 24, 315
valid type, 29
value, 24
default, 161
value-written  value written by an event of an execution, 161, 350
value-written'
value written by an event, 350
Var  local variable, 26, 328
virtual machine
aggressive, 48
defensive, 49
visible move, 210
vname  type of variable and field names, 24
Void  type for Unit, 24, 315
volatile
event, 161
member, 161
volatile  volatile modifier, 27, 315

W

W_  set of write events, 161, 349
WAIT, 88
wait  method wait
semsitics, 88
signature, 83
wait set, 63
BTA semantics, 72–73
BTA syntax, 67
wait set status, 63
wait-notify
interaction with interruption,
90–91
semantics, 87–90
WaitFail, 88
waiting  wait set status InWS_, 63
WaitIntrd1, 88
WaitIntrd2, 88
WaitNtfd, 88
waits resources that a thread action waits for, 102
(_ _) wait-set-act
  type of a wait set BTA, 67
(_ _) wait-sets type of wait sets, 63
  _ wait-set-status
    type of a wait set status, 63
WakeUp BTA to remove thread from wait set, 67, 73
wakeup pick thread to be notified, 275
weak bisimulation, 224
well-formedness
  bytecode, 54
class, 35
  complete interelaving, 163
execution, 162
  intermediate language, 238
J0 call stack, 232
  map to invariant data, 80
method, 35
  method overriding, 35
preservation for stage 1, 243
preservation for stage 2, 251
program, 35
  single-threaded semantics, 78
start state, 96, 138
system classes, 35
well-founded delay bisimulation, 214
well-typing, 50, 54
  method declaration, 53
wf0 well-formed J0 call stack, 232
wf-cdecl well-formed class declaration, 35, 36, 83, 327
wf-cdecl’ well-formed class declaration except for acyclicity check, 269
wf-fdecl well-formed field declaration, 35, 36, 327
wf-j1-mdecl well-formed J1 method declaration, 239
wf-j1-prog well-formed J1 program, 239
wf-J-mdecl well-formed source code method declaration, 36, 37, 332
wf-J-prog well-formed source code program, 36, 37, 332
wf-jvm-prog well-formedness for bytecode, 54
wf-mdecl well-formed method declaration, 35, 36, 328
  _ wf-mdecl-test type of well-formedness tests for method declarations, 35
wf-overriding well-formed method overriding, 35, 36, 327
wf-prog generic well-formedness, 36, 327
  code extraction, 269
wf-start well-formed start state, 96, 139
wf-states well-formed multithreaded states, 108
wf-syscls well-formedness of system classes, 35, 36, 83, 327
while (_ _) while loop, 26, 328
WokenUp BTA to process wake-up, 67, 73
word32 32-bit machine words, 14
Write write event, 84, 143
write heap write operation, 134
write event, 161
  set of, 161
write to location, 161
write-seen function, 162
wset projection to wait sets, 65
WS-Notified wait set status “notified”, 63
WS-WokenUp wait set status “woken-up”, 63
WT1sync, 237
WTAAss, 32
WTbinop, 32
Index

WTcall, 32
WTcond, 32
WTrtInsync, 114
WTrtSync, 114
WTsync, 93
WTthrow, 32
WTtry, 32

X
xcpt-app applicability for raised exceptions, 51, 343
xcpt-class convert exception handler class restriction to type, 342
xcpt-eff effect of exception handling, 51, 342
xcpt-step exception handling for the VM, 48, 347

Y
Yield scheduler BTA to prefer other threads, 67, 72
YIELD, 87
yield method yield semantics, 87
signature, 83

Z
zip combines lists elementwise, 15
Andreas Lochbihler
A MACHINE-CHECKED, TYPE-SAFE MODEL OF JAVA CONCURRENCY

The Java programming language provides safety and security guarantees such as type safety and its security architecture. They distinguish it from other mainstream programming languages like C and C++. In this work, we develop a machine-checked model of concurrent Java and the Java memory model in the proof assistant Isabelle/HOL and investigate the impact of concurrency on these guarantees. From the formal model, we show how to automatically obtain an executable, verified compiler to bytecode and a validated virtual machine. Modularisation is the key to get a tractable and usable model; we carefully partition the definitions and proofs into modules that capture the interactions between sequential parts, concurrency, and the memory model.