Journal of Automated Reasoning manuscript No.
(will be inserted by the editor)

Mechanising a type-safe model of multithreaded Java with a verified compiler

Andreas Lochbihler

Received: date / Accepted: date

Abstract This article presents JinjaThreads, a unified, type-safe model of multithreaded Java source code and bytecode formalised in the proof assistant Isabelle/HOL. The semantics strictly separates sequential aspects from multithreading features like locks, forks and joins, interrupts, and the wait-notify mechanism. This separation yields an interleaving framework and a notion of deadlocks that are independent of the language, and makes the type safety proofs modular. JinjaThreads's non-optimising compiler translates source code into bytecode. Its correctness proof guarantees that the generated bytecode exhibits exactly the same observable behaviours as the source code, even for infinite executions and under the Java memory model. The semantics and the compiler are executable.

JinjaThreads builds on and reuses the Java formalisations Jinja, Bali, μJava, and Java^{light} by Nipkow's group. Being the result of more than fifteen years of studying Java in Isabelle/HOL, it constitutes a large and long-lasting case study. It shows that fairly standard formalisation techniques scale well and highlights the challenges, benefits, and drawbacks of formalisation reuse.

Keywords Java · concurrency · type safety · compiler verification · operational semantics

1 Introduction

The formal analysis of the Java programming language started soon after it had been released in 1995. The Bali project, e.g., lead to a comprehensive model Java^{light} of the JavaCard language, a sequential subset of Java [59, 62–65, 74, 75]. Using Isabelle/HOL, Nipkow's group formalised the type system and a big-step semantics with a proof of type safety, and

This article extends two papers presented at FOOL 2008 [46] and ESOP 2010 [47] and combines the results with a formalisation of the Java memory model [48,50]. It is a condensed and updated version of parts of the author's PhD thesis [49]. Most of this work was done while the author was at the University of Passau and at the Karlsruhe Institute of Technology in Germany. The Deutsche Forschungsgemeinschaft supported the author with grants Sn11/10-1 and Sn11/10-2.

Andreas Lochbihler
Institut für Informationssicherheit, Departement für Informatik, ETH Zürich
Universitätstrasse 6, 8092 Zürich, Switzerland
Tel.: +41-44-6328470
Fax: +41-44-6321172
E-mail: andreas.lochbihler@inf.ethz.ch

an axiomatic Hoare-style semantics that they proved sound and relatively complete. At the same time, they studied the interaction between Java source code and bytecode for a smaller subset that was named μJava. This line of work [8, 28–30, 34, 35, 58, 60, 67, 80, 81] lead to formal models of the Java 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. Later, Jinja by Klein and Nipkow [32] extended the core parts of μJava with a small-step semantics for source code, which they proved equivalent to the big-step semantics. Java^{light}, μJava, and Jinja were considered milestones in formalised semantics [83, 85].

All these models consider only sequential Java, although multithreading has been envisioned as future work from the start [59]. In this article, I extend Jinja with concurrency in the form of Java threads and the Java memory model (JMM). My new model JinjaThreads covers all of Jinja: the source code language (except for the big-step semantics), the bytecode language, the type safety proofs, and the compiler with its correctness statements.

In detail, I present a type-safe model of Java threads for source code and bytecode (Section 2). It includes synchronisation via monitors and volatile fields, the wait-notify mechanism, forks and joins, interrupts, and spurious wake-ups (Section 1.1 reviews these synchronisation mechanisms). Thus, it covers all concurrency features from the Java language specification 8 [21] except the following: (i) the deprecated methods `stop`, `destroy`, `suspend`, and `resume` in class `Thread`; (ii) timing-related features like timed `wait` and `Thread.sleep` because JinjaThreads does not model time; and (iii) low-level memory operations from the `java.util.concurrent` package that lack a specification in terms of the Java memory model. In particular, my model includes an atomic compare-and-set operation on volatile fields that suffices to implement most of the lock-free algorithms and synchronisation primitives from the `java.util.concurrent` package.

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 Java features to remain tractable, e.g., static and final fields and methods, visibility modifiers, interfaces, generics, class initialisation, and garbage collection. I develop a semantics for the concurrency features that is independent of the sequential parts and the Java memory model. This way, I can reuse the same multithreading semantics for both source code and bytecode and conduct proofs about concurrency without being overwhelmed by the details of the sequential parts. For example, type safety for source code and bytecode (Section 3) follows from lemmas about the single-threaded semantics by a generic argument. Deadlocks in particular are captured and dealt with abstractly in the proofs.

Two memory models specify the shared heap: sequential consistency and the JMM (Section 4). I only sketch them as they have been presented in detail in a companion article [50].

Moreover, I have formally verified a (non-optimising) compiler that connects source code and bytecode (Section 5). To my knowledge, this is the first Java compiler that has been shown correct under the Java memory model. Even without optimisations, concurrency poses two challenges for compiler verification: non-deterministic interleaving and different granularity of atomic operations. I address non-deterministic interleaving using a bisimulation approach for sequential programs. Thanks to the separation of semantics, the correctness proof for the compiler reduces to a correctness proof for individual threads. As the observable behaviour of a thread includes all accesses to shared memory, method calls and synchronisation, I obtain a bisimulation for the multithreaded program from bisimulations for individual threads. Bisimulation also addresses the atomicity issue: unobservable steps may be decomposed into arbitrarily many unobservable steps, observable ones into

multiple unobservable ones followed by an observable one. Moreover, the bisimulation approach yields stronger correctness guarantees than Jinja's, even in the single-threaded case.

The complete model and all proofs are available online [45] in the Archive of Formal Proofs. In the presentation, I often omit the formal definitions of parts that are not directly relevant for concurrency. More details can be found in my PhD thesis [49].

Contributions. First, JinjaThreads is a *unified* model of Java concurrency, which is more than the sum of its parts. Indeed, JinjaThreads makes it possible to study how Java's sequential parts interact with the multithreading features. As I have strictly separated the single-threaded semantics from the multithreaded one, the formalisation itself makes these interactions explicit. The separation also applies to the theorems like type safety and compiler correctness. For example, I identify sufficient conditions under which the single-threaded statements extend to the multithreaded case and prove that source code and bytecode meet them.

Second, the separation yields a language-independent formalisation of multithreading with many synchronisation options. Thus, this framework could be reused for other languages that use similar forms of synchronisation. The sufficient conditions, too, are expressed independently of the language and therefore the extension proofs carry over. For type safety, in particular, the statement must account for the possibility of deadlocks that multithreading introduces. In my framework, I propose a precise semantic definition of deadlock for the type safety statement, which does again not depend on the language. This addresses a short-coming of typical type safety proofs for concurrent languages, which over-approximate deadlocks syntactically and therefore unnecessarily weaken the guarantees.

Third, for the compiler verification, I identify a notion of bisimulation that guarantees preservation of terminating, non-terminating and deadlocking behaviours. This gives stronger correctness guarantees for the compiler. For example, even in the sequential case, my proof guarantees that non-terminating programs get compiled to non-terminating programs, which μJava's and Jinja's verifications cannot.

Forth, JinjaThreads constitutes a large formalization case study; it is the culmination of more than fifteen years of Java formalisation efforts. It thus demonstrates that formalisation reuse is possible at a large scale and that fairly standard techniques scale well. I discuss the challenges, benefits, and drawbacks of formalisation reuse and illustrate them with examples from JinjaThreads (Section 6.2).

Relation to previous work. Earlier versions of the multithreading semantics and the type safety proof for the source language have been presented in [46]. This article additionally treats the bytecode language and extends the list of concurrency features by interrupts, joins, spurious wake-ups, volatile fields, atomic compare-and-set operations, and the JMM. Adding various kinds of synchronisation required a re-design of the multithreaded semantics to support a modular treatment of different synchronisation primitives and to simplify the proofs that the single-threaded semantics meet the requirements of the lifting theorems. For the JMM, I have separated the single-threaded semantics from the concrete heap model via an abstract heap interface.

A short version of the compiler verification has been published at ESOP [47]. This article extends the correctness guarantees to diverging executions, deadlocks, and to the JMM. The stronger guarantees require a stronger (bi-)simulation notion, as discussed in Section 5.2.

The formalisation of the JMM itself and its connection with JinjaThreads has been described in detail in [48,50] except for the compiler verification part and the compare-and-set operations. Here, I only sketch the connection to the JMM (Section 4). Instead, this article focuses on how Java threads are modelled in JinjaThreads, how type safety is proven, and how the compiler is verified.

Organisation. The technical sections 2, 3, and 5 present the semantics, the type safety proof, and the compiler verification, respectively, in great detail. Each of these section starts with a subsection that gives a high-level overview of the challenges and achievements of the section. These subsections can be read independently of the subsequent technical material such that readers can quickly get an idea of the most interesting aspects of each part. Section 4 sketches the two memory models for the shared heap. Formalisation choices, the pros and cons of formalisation reuse, and this project's motivations and benefits are presented in Section 6. Related work is discussed in Section 7.

1.1 Multithreading in Java

For this article to be self-contained, this section gives a quick tour of the concurrency features of Java 8. 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 *IllegalThreadState* 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 five kinds of synchronisation between threads: locks, wait sets, forks, joins, and interrupts. The package `java.util.concurrent` in the Java API [26] builds sophisticated forms of synchronisation from these primitives and atomic compare-and-set operations on volatile fields, which JinjaThreads also models (see below).

Every object (and array) has an associated monitor with a lock and a wait set. Locks are mutually-exclusive and re-entrant, i.e., at most one thread can hold a lock at a time, but the thread can acquire the lock multiple times [21, §17.1]. For locking, Java uses `synchronized` blocks that take a reference to an object or array. A thread acquires 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 [21, §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 *IllegalMonitorState* exception.

To avoid busy waiting, a thread can suspend itself to the wait set of an object by calling the object's method *wait* [21, §17.8]. To enter the wait set, the thread must have locked the object's monitor and must not be interrupted; otherwise, an *IllegalMonitorState* or *Interrupted* exception, respectively, is raised. 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 *Interrupted* exception. 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. After the notifying thread has released the lock, the notified thread competes with the other threads for acquiring the lock.

When a thread calls *join* on another thread, it blocks until (i) the thread associated with the receiver object 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 *Interrupted* exception; otherwise, it returns normally.

Interruption [21, §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 *Interrupted* exception, 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.

Beyond threads and synchronisation, Java also specifies how shared memory behaves under concurrent accesses, which is known as the Java memory model [21, §17.4]. Volatile fields in particular are used to implement lock-free synchronisation constructs within Java that do not rely on the above kinds. To that end, a compare-and-set operation is crucial: it atomically checks whether a volatile field stores a given value and, if so, replaces it with another given value; and returns whether the comparison was successful. No knowledge of the JMM is needed for this article; a detailed account can be found in [50].

1.2 Notation

In this article, I mostly use 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 $\longrightarrow$ or $\Longrightarrow$ and associates to the right. Since the latter form stems from Isabelle's natural deduction framework, it separates assumptions from conclusions in proof rules, but cannot occur inside other HOL formulae. Multiple assumptions are enclosed in $[\![$ and $]\!]$ with the separator ";". For example, modus ponens is written $[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q$. Biimplication $P \longleftrightarrow Q$ is shorthand for $P \longrightarrow Q$ and $Q \longrightarrow P$.

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 τ. 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 $fst :: 'a \times 'b \Rightarrow 'a$ and $snd :: 'a \times 'b \Rightarrow 'b$. 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 $Inl :: 'a \Rightarrow 'a + 'b$ and $Inr :: 'b \Rightarrow 'a + 'b$. Like $\times$, $+$ associates to the right.

Sets (type $'a$ *set*) are isomorphic to predicates (type $'a \Rightarrow$ *bool*) with bijections $\_ \in \_$ and $\{x \mid \_\}$. $\emptyset$ denotes the empty set. The predicate *finite* on sets characterises all finite sets. The operator $\uplus :: 'a\ set \Rightarrow 'b\ set \Rightarrow ('a + 'b)\ set$ denotes disjoint union on sets.

The definite description operator $\iota x.\ P\ x$ is known as Russell's ι-operator. It denotes the unique x such that $P\ x$ holds, provided exactly one exists. Hilbert's choice, 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 $'a$ *list*) come with the empty list $[]$ and the infix constructor $\cdot$ for consing. Variable names ending in "*s*" usually stand for lists. The 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 the first element of *xs* and *take n xs* returns the first *n* elements of *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*; *zip xs ys* combines *xs* and *ys* elementwise into a list of pairs; and *foldl f a xs* reduces the list *xs* with the binary operator *f* and start value *a*, associating to the left.

datatype $'a$ *option* = *None* | *Some* $'a$ adjoins a new element *None* to $'a$, all existing elements in type $'a$ are also in $'a$ *option*, but prefixed by *Some*. For succinctness, I write $\lfloor a \rfloor$ for *Some a*. For example, *bool option* consists of the three values *None*, $\lfloor True \rfloor$, and $\lfloor False \rfloor$. Variables whose name ends in "o" usually have *option* type.

Case distinctions on datatypes use guard-like syntax. For example, *case xo of None* $\Rightarrow$ *a* | $\lfloor x \rfloor \Rightarrow f$ pattern-matches on *xo*. If *xo* is *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, $f(a := b) = \lambda x.$ *if* $x = a$ *then b else* $f\ x$.

As all functions in HOL are total, *partial functions* are modelled as functions of type $'a \Rightarrow$ $'b$ *option* where *None* represents undefinedness and $f\ x = \lfloor y \rfloor$ means that *f* maps *x* to *y*. I abbreviate $'a \Rightarrow 'b$ *option* by $'a \rightharpoonup 'b$ 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 $f\big(x_{[0]} \mapsto y_{[0]}\big) \ldots \big(x_{[i]} \mapsto y_{[i]}\big)$ where *i* is the minimum of $|xs|-1$ and $|ys|-1$. The everywhere undefined map $\lambda\_.$ *None* is written *empty*. Updates of *empty* are written $[x \mapsto y]$ and $[xs\,[\mapsto]\,ys]$. The domain of *f* (written *dom f*) is the set of points at which *f* is defined. The order $m_1 \subseteq_{\mathrm{m}} m_2$ on maps denotes that m_2's domain contains m_1's and m_1 and m_2 are equal on *dom* m_1.

Locales [4] 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). A locales can import other locales where parameters may be instantiated and names prefixed as necessary. A locale context collects declarations such as theorems and definitions, which may depend on the parameters and assumptions. Locale interpretations instantiate the parameters of a locale and discharge the assumptions. This specialises all collected declarations to the given arguments.

For example, the locale *semigroup* below declares a context whose parameter, i.e., abstract operation, is the binary operator $\odot$ that is assumed to be associative. The locale *monoid* imports the *semigroup* context with the name prefix *p* and renames the parameter to $\oplus$. It also adds another operation *e* and assumes that *e* is the neutral element. The type of the semigroup and monoid elements is a type variable $'a$ rather than an opaque type. This way, the locale interpretation *free-monoid* can instantiate $'a$ with the type of lists.

locale *semigroup* = fixes $\odot :: 'a \Rightarrow 'a \Rightarrow 'a$ assumes *assoc*: $(a \odot b) \odot c = a \odot (b \odot c)$
locale *monoid* = *p*: *semigroup* $\oplus$ for $\oplus$ + fixes *e* :: $'a$ assumes $a \oplus e = a$ and $e \oplus a = a$
interpretation *free-monoid*: *monoid* @ [] ⟨⟨proof⟩⟩

2 Modelling multithreaded Java source code and bytecode

In this section, I present a language-independent interleaving semantics (Section 2.2) and the JinjaThreads multithreaded semantics for Java source code (Section 2.3) and bytecode (Section 2.4) after having discussed the design principles and core ideas (Section 2.1).

2.1 High-level overview

Two design principles guided the modelling of multithreaded Java: separation of concerns and sharing (and reuse) of definitions and proofs. Both are key to obtain a tractable model.

To disentangle concurrency from sequential features, I developed a language-independent interleaving semantics, which is parametrised by the semantics of individual threads. So, the source code and bytecode formalisations share the multithreading semantics. This makes sense as they provide the same multithreading features. Moreover, it simplifies the type safety proofs as the concurrency features are dealt with in the multithreading semantics. Without the separation, I would have had to consider them twice, once for the source code and once for the bytecode.

The multithreading semantics relieves the single-thread semantics from the burdens of multithreading. It manages the multithreaded state (i.e., the locks, wait sets, the thread pool, and interrupts) and interleaves the individual threads. It also isolates the local states of the threads from each other. I achieve this separation and isolation by ensuring that neither can the single-threaded semantics directly access the multithreaded state nor does the multithreaded semantics look at the thread-local states. Instead, every single-threaded transition is labelled by a so-called thread action, which communicates to the interleaving semantics the preconditions on and atomic updates to the multithreaded state. When the interleaving semantics executes a step, it changes the multithreaded state according to the thread action. These actions are the only means of communication between the two levels. Since this is unidirectional, the multithreaded semantics can transfer information to the single-threaded semantics only by picking one step that the latter offers. Hence, the single-threaded semantics must anticipate in its steps all possible answers it is willing to accept. Thanks to this clear and restricted interface, I can reduce proofs about the multithreaded semantics to the level of threads, as threads interact only via thread actions (and the shared heap).

Some synchronisation primitives perform complicated checks and updates to the multithreaded state, which must all be executed atomically. For example, a call to `wait` on some monitor a must check that the current thread holds the lock on a and that it does not have a pending interrupt; it then releases the lock on a and suspends itself to a's wait set. So this single transition checks and updates the locks, the interrupts, and the wait sets. To avoid redundancies in the definitions and proofs, thread actions are built from basic thread actions (BTA), each of which deals with only one aspect. There are BTAs for checking, acquiring and releasing the lock state of a monitor, for spawning a new thread, for setting, clearing, and testing for pending interrupts, notifying other threads, etc. The single-threaded semantics modularly combine these BTAs into one thread action that encodes all the desired checks and updates. Although the BTAs are clearly inspired by Java's synchronisation mechanism, they are more abstract and modular. So they could be used for modelling synchronisation primitives of other languages, too. In fact, thread actions can express more than what is needed for Java threads. This generality sometimes complicates reasoning as proofs must deal with BTA combinations that the single-threaded semantics never use.

In return, the decomposition into BTAs ensures that the proofs about the multithreaded semantics are simple and modular, too, as they can also focus on a particular aspect of concurrency, e.g., on locking. To that end, the multithreaded state is also partitioned into five components (locks, thread pool, shared heap, wait sets, and pending interrupts).[1] This way, every BTA except for one depends on and affects only one component of the state. The BTA semantics can thus be defined for component individually and then composed at the end.

The strict separation into single-threaded and multithreaded semantics also helps with reuse. As the single-threaded semantics need not manage the multithreaded state, I was able

[1] 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 (Section 3.3).

to reuse the existing sequential Jinja semantics with only small modifications (but significantly extended with the concurrency features). As I have been careful to make the extensions fit into the existing design, many theorems about the semantics, in particular about the sequential aspects, needed only small adaptations. I was thus able to benefit from many years of experience that have gone into Jinja.

The key ideas of the source code formalisation are the following: The semantics is a standard small-step semantics where subexpression reduction rules determine the evaluation order. Raised exceptions slowly propagate outwards to an exception handler via exception propagation rules. The `synchronized` blocks ensure that the lock is released when the block is left, normally or due to an exception. Method calls are inlined dynamically; local variables are nevertheless statically scoped as the type system ensures that all method bodies are closed expressions. Native methods implement many of the concurrency features such as the wait-notify mechanism, interruption, and thread spawns. Like in Java, JinjaThreads programs must declare native methods as methods without body. This ensures that method lookup and overriding work uniformly for ordinary and native methods. The compare-and-set operation is a language construct of its own, as opposed to a native method like in Java, because the field must be given as an argument. This is done using reflection in Java, but JinjaThreads does not model reflection. I therefore formalise compare-and-set as a separate language construct.

Bytecode programs differ from source code only in the method bodies. As class, field and method declarations use the same abstract syntax as source code, the lookup functions are shared, too. The virtual machine interprets the instructions of a method body on a stack-based state machine. When an exception is raised, the VM looks for a suitable exception handler in the current method; execution immediately jumps to the handler or reraises the exception at the call site of the current method. The instructions for locking and unlocking need not follow a block structure, so unlocking may fail in case the lock is not held. Native methods share the formalisation with source code. Like in source code, compare-and-set is formalised as an instruction instead of a native method. There are two VM versions. The aggressive VM assumes that there are always sufficiently many operands of the expected types on the stack; if not, the behaviour is unspecified. The defensive VM checks these conditions and aborts the execution with an error if the check fails. The bytecode verifier abstractly interprets the instructions and ensures that these checks always succeed.

To support different memory models, both semantics operate on an abstract heap model. In Section 4, I instantiate it with sequential consistency and the Java memory model. The abstract model is formalised as a locale that fixes the abstract heap operations (empty heap, allocation, reading, writing, and dynamic address types) and their properties. Allocation, reading, and writing are formalised as relations such that they may fail or be non-deterministic, which is necessary under the JMM. The abstract heap model differs from other memory model formalisations [41, 69] in that the assumed properties are very weak. For example, I do not assume that writing a value to some heap location l and then immediately reading from l again yields the same value, as this is not guaranteed and may be sometimes forbidden under the JMM. Instead, my assumptions identify the properties on which the type safety and compiler correctness proofs rely. For example, dynamic address types do not change once they have been determined.

Figure 1 shows the structure of JinjaThreads. In comparison to Jinja, new parts are set in bold, adapted ones normally, and dropped ones in grey with dotted lines. Source code and bytecode share some general infrastructure and the multithreading semantics with two memory models (sequential consistency and the JMM). The compiler translates source code into bytecode in two stages.

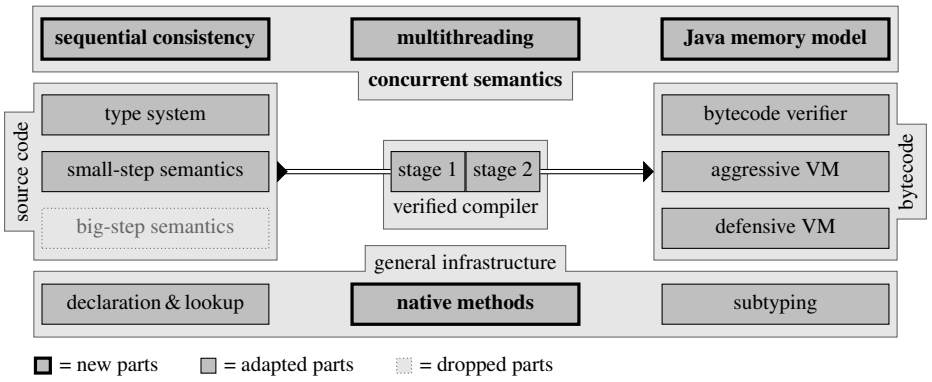


Fig. 1 Structure of JinjaThreads in comparison with Jinja's

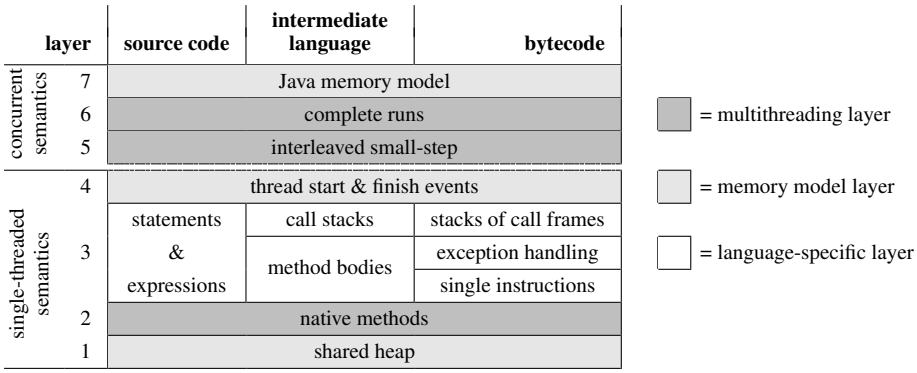


Fig. 2 JinjaThreads stack of semantics

The semantics themselves are organised in a stack of seven layers that separate the sequential aspects, the concurrency features, and the memory model from one another (Figure 2). 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 holds for both memory models as they differ only in layers 1, 4, and 7.

2.2 Interleaving semantics

The multithreading semantics relieves the single-thread semantics from the burdens of multithreading. It manages the multithreaded state (i.e., the locks, wait sets, the thread pool, and interrupts) and interleaves the individual threads. It also isolates the local states of the threads from each other. Interaction between individual threads and the interleaving semantics happens only via thread actions. This way, the multithreading semantics is oblivious of thread-local states. Thus, I can use it for both source code and bytecode.

2.2.1 The multithreaded state

The multithreaded state $s = (ls, tp, h, ws, is)$ consists of five components:

$$
\begin{array}{ll}
\textit{has-locks}\ \ \textit{None}\ \ t = 0 & \textit{unlock}\ \ \textit{None}\ \ = \textit{None} \\
\textit{has-locks}\ \lfloor (t',n) \rfloor\ t = (\textit{if}\ t = t'\ \textit{then}\ n+1\ \textit{else}\ 0) & \textit{unlock}\ \lfloor (t,n) \rfloor = (\textit{if}\ n = 0\ \textit{then None else}\ \lfloor (t,n-1) \rfloor) \\
\textit{may-lock}\ \ \textit{None}\ \ t = \textit{True} & \textit{acquire}\ L\ t\ n = \\
\textit{may-lock}\ \lfloor (t',n) \rfloor\ t = (t = t') & \quad (\textit{if}\ n = 0\ \textit{then}\ L\ \textit{else acquire}\ (\textit{lock}\ L\ t)\ t\ (n-1)) \\
\textit{lock}\ \ \ \textit{None}\ \ t = \lfloor (t,0) \rfloor & \textit{release}\ \ \textit{None}\ \ t = \textit{None} \\
\textit{lock}\ \ \ \lfloor (t',n) \rfloor\ t = \lfloor (t',n+1) \rfloor & \textit{release}\ \lfloor (t',n) \rfloor\ t = (\textit{if}\ t' = t\ \textit{then None else}\ \lfloor (t',n) \rfloor)
\end{array}
$$

Fig. 3 Implementation of lock operations

1. The *lock status ls* stores in a map of type $'l \rightharpoonup ('t \times nat)$ (denoted by $('l,'t)$ *locks*) for every lock l how many times it is held by a thread, if any, where $'l$ and $'t$ denote the types of lock and thread identifiers, respectively. A thread with ID t holding the lock l exactly $n+1$ times is represented by $ls\ l = \lfloor (t,n) \rfloor$. If l is not held by any thread, then $ls\ l = None$. Using a map with a counter ensures that 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.
   The state of a single lock L of type $('t \times nat)$ *option* (denoted $'t$ *lock*) is manipulated using the six operations given in Figure 3: (i) *has-locks L t* denotes the number of times t has acquired L; (ii) *may-lock L t* tests whether t may lock L; (iii) *lock L t* acquires L for t once; (iv) *unlock L* release L once; (v) *acquire L t n* acquires L for t n times; and (vi) *release L t* completely releases L if t holds L.[2]
2. The *thread pool tp* is a map of type $'t \rightharpoonup ('x \times ('l \Rightarrow nat))$, denoted by $('l,'t,'x)$ *tp*. Free thread IDs are mapped to *None*. If $tp\ t = \lfloor (x,ln) \rfloor$, then t identifies a thread with local state x (type variable $'x$) and temporarily released locks *ln*. 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, the thread must reacquire the lock before it can continue. The multiset *ln* records how often the thread has held the lock and the multithreaded semantics ensures that the locks are acquired before the thread executes again. This way, the single-threaded semantics need neither remember how many times the lock had been acquired, nor reacquire it explicitly afterwards.
3. The *shared heap* $h :: 'h$ which is passed from one thread to the other.
4. The *wait set status* $ws :: 't \rightharpoonup ('w\ \textit{wait-set-status})$ stores for every thread t its wait set status, where $'w$ *wait-set-status* consists of the values *InWS w* and *WS-Notified* and *WS-WokenUp*, and $'w$ represents the type of wait set identifiers. The wait set $w :: 'w$ of a monitor contains all threads whose wait status is $\lfloor InWS\ w \rfloor$. The predicate *waiting wo* tests whether the wait set status $wo :: 'w\ \textit{wait-set-status option}$ is of the form $\lfloor InWS\ w' \rfloor$, i.e., the associated thread is in a wait set.
   Figure 4 shows the possible transitions between the wait set status as an automaton. Initially, after the thread has been spawned, its wait set status is *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 w (status $\lfloor InWS\ w \rfloor$). When another thread notifies or wakes up (i,e., interrupts) the thread (dashed lines), the latter's status changes to $\lfloor WS\text{-}Notified \rfloor$ or $\lfloor WS\text{-}WokenUp \rfloor$, respectively. From either of them, it leaves the wait set cycle and returns to the normal state *None* by processing the notification or wake-up, respectively. If the thread has temporarily released some locks when it suspended it-

[2] Some operations on locks should be partial. For example, $lock\ \lfloor (t',n) \rfloor\ t$ only makes sense if $t' = t$. As usual in HOL, I extend *lock* to a total function and ignore the mismatch. This choice removes some preconditions from a few lemmata. For example, if *has-lock L t*, then also *has-lock* $(acquire\ L\ t'\ n)\ t$ where *has-lock L t* is short-hand for $has\text{-}locks\ L\ t > 0$. Other implementations of *lock* would require the precondition *may-lock L t'*.

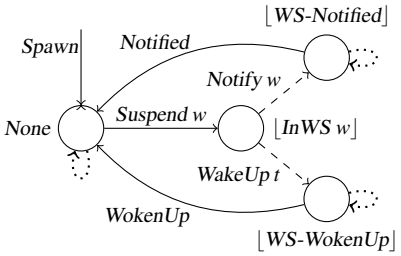


Fig. 4 Wait sets, notification, and interruption

thread `t1`

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

thread `t2`

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

Fig. 5 Example program with two threads

self to the wait set, it must reacquire them in states ⌊*WS-Notified*⌋ and ⌊*WS-WokenUp*⌋ (dotted arrow).[3] Following the Java language specification [21, §17.8], reacquisition precedes processing the removal from the wait set, i.e., notification or wake-up, although the order is semantically irrelevant.

5. The set of *pending interrupts is* :: *'t set* stores the IDs of all interrupted threads.

The projection functions *locks*, *tp*, *shr*, *wset*, and *intrs* return the locks, the thread pool, the shared heap, the wait sets, and the pending interrupts of a state, respectively.

For example, Figure 5 shows a program with two threads `t1` and `t2`. Suppose that `t1` executes first until it enters the wait set of monitor `g` and then `t2` executes until is has interrupted `t1`. Suppose further that f identifies the monitor referenced by `f`, and similarly for `g`. Then, this state is represented by the tuple (ls, tp, h, ws, is) where

$ls = [f \mapsto (\texttt{t1}, 1), g \mapsto (\texttt{t2}, 0)]$, i.e., threads `t1` and `t2` hold the locks f and g twice and once, respectively; all other locks are free;
$tp = [\texttt{t1} \mapsto (\ldots, (\lambda\_.\ 0)(g := 1)), \texttt{t2} \mapsto (\ldots, \lambda\_.\ 0)]$, where the thread-local states have been omitted, thread `t1` has temporarily released the lock on g which it had held once before, and thread `t2` has not temporarily released any locks;
$h = \ldots$ is some shared heap;
$ws = [\texttt{t1} \mapsto \mathit{WS\text{-}WokenUp}]$, i.e., an interrupt has woken up thread `t1`; and
$is = \{\texttt{t1}\}$, i.e., thread `t1` has a pending interrupt.

2.2.2 Thread actions

Thread actions encode the preconditions on and the updates to the multithreaded state of a single-thread execution step. A thread action is composed of multiple basic thread actions. My formalisation implements 16 different BTAs, which are split in five groups (Figure 6):

Locking The four lock actions (type *lock-act*) query and manipulate a single lock. Like locks in the lock status, lock actions are grouped by lock ID on which they operate (type *'l lock-acts*, the list orders the lock BTAs for the same lock).
- *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. Apart from the single lock, it also updates the temporarily released locks in the thread pool.
- *UnlockFail* tests whether t does not hold a lock, i.e., unlocking it would fail.

[3] In Java, all suspended threads have temporarily released a lock. So reacquisition always takes place for JinjaThreads source code and bytecode. This invariant does not hold on the language-independent interleaving level, though. The reacquisition step is therefore conditional.

datatype *lock-act* = *Lock* | *Unlock* | *UnlockFail* | *Release*
type_synonym $'l$ *lock-acts* = $'l \Rightarrow$ *lock-act list*
datatype $('t,'x,'h)$ *nt-act* = *Spawn* $'t$ $'x$ $'h$ | *ThreadEx* $'t$ *bool*
datatype $'t$ *cond-act* = *Join* $'t$
datatype $('t,'w)$ *wait-act* = *Suspend* $'w$ | *Notify* $'w$ | *NotifyAll* $'w$ | *WakeUp* $'t$ | *Notified* | *WokenUp*
datatype $'t$ *intr-act* = *Intr* $'t$ | *ClearIntr* $'t$ | *IsIntrd* $'t$ *bool*
type_synonym $('l,'t,'x,'h,'w,'o)$ *thread-action* =
$'l$ *lock-acts* × $('t,'x,'h)$ *nt-act list* × $'t$ *cond-act list* × $('t,'w)$ *wait-act list* × $'t$ *intr-act list* × $'o$ *list*

Fig. 6 Type definitions for thread actions

There is no BTA for testing whether a thread holds a lock. This can be simulated by using the two lock BTAs *Unlock* and *Lock* in this order in one thread action: *Unlock* tests that t holds the lock and *Lock* undoes the effect of *Unlock*.

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 (Sections 3.2 and 5.2.2), 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.

Thread join *Join* t joins on thread t, i.e., t must have terminated before.

Wait sets *Suspend* w inserts the current thread t into the wait set w. *Notify* w and *NotifyAll* w wake up one or all of the threads in the wait set w. Their wait set status becomes $\lfloor$*WS-Notified*$\rfloor$ (see Figure 4). If w is empty, no thread is woken up. *WakeUp* t changes t's wait set status to $\lfloor$*WS-WokenUp*$\rfloor$, if it has been in a wait set before. Otherwise, nothing happens. *Notified* and *WokenUp* label steps 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 t has a pending interrupt; b :: *bool* denotes the result.

A thread action consists of a family for the lock BTAs and one list for each of the other groups. Since thread actions are used as labels for the steps, they include a sixth component of type $'o$ *list*, which will be used by the memory models (Section 4). It maintains the grouping of BTAs instead of putting them all into one list because each group affects only one part of the state, so their semantics can be defined independently of other groups. The projections $\langle ta \rangle_\mathrm{l}$, $\langle ta \rangle_\mathrm{t}$, $\langle ta \rangle_\mathrm{c}$, $\langle ta \rangle_\mathrm{w}$, $\langle ta \rangle_\mathrm{i}$, and $\langle ta \rangle_\mathrm{o}$ extract from the thread action ta the BTAs for locks, thread creation, conditions, wait sets, interrupts, and the memory model part.

All BTAs of a thread action are executed in a single step; if the precondition of at least one BTA in the thread action of a step is not met, the interleaving semantics does not select the step. Thus, a single-thread semantics can query and change multiple components of the multithreaded state in one step by composing BTAs.

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 (where the memory model events have been omitted):

$$((\lambda\_.\ [])(l := [\mathit{Unlock}, \mathit{Lock}, \mathit{Release}]), [], [], [\mathit{Suspend}\ w], [\mathit{IsIntrd}\ t\ \mathit{False}], \ldots)$$

that is, this thread action performs the lock BTAs *Unlock*, *Lock*, *Release* on l in this order and no lock BTAs on other locks, no thread creation and thread join BTAs, one wait set BTA *Suspend* w, and one interruption BTA *IsIntrd* t *False*. Instead of this cumbersome notation, I use a list-like notation for thread actions, with lock identifiers added to lock BTAs. Isabelle's parser and pretty printer are set up such that they convert it into the corresponding thread action. Hence, the above thread action is written as

$$\begin{array}{ll}
\textit{upd-L} :: {'t}\ \textit{lock} \Rightarrow {'t} \Rightarrow \textit{lock-act} \Rightarrow {'t}\ \textit{lock} & \textit{ok-L} :: {'t}\ \textit{lock} \Rightarrow {'t} \Rightarrow \textit{lock-act} \Rightarrow \textit{bool} \\
\textit{upd-L}\ L\ t\ \textit{Lock} = \textit{lock}\ L\ t & \textit{ok-L}\ L\ t\ \textit{Lock} = \textit{may-lock}\ L\ t \\
\textit{upd-L}\ L\ t\ \textit{Unlock} = \textit{unlock}\ L & \textit{ok-L}\ L\ t\ \textit{Unlock} = (\textit{has-locks}\ L\ t > 0) \\
\textit{upd-L}\ L\ t\ \textit{UnlockFail} = L & \textit{ok-L}\ L\ t\ \textit{UnlockFail} = (\textit{has-locks}\ L\ t = 0) \\
\textit{upd-L}\ L\ t\ \textit{Release} = \textit{release}\ L\ t & \textit{ok-L}\ L\ t\ \textit{Release} = \textit{True} \\
\\
\textit{upd-Ls} :: {'t}\ \textit{lock} \Rightarrow {'t} \Rightarrow \textit{lock-act list} \Rightarrow {'t}\ \textit{lock} & \textit{ok-Ls} :: {'t}\ \textit{lock} \Rightarrow {'t} \Rightarrow \textit{lock-act list} \Rightarrow \textit{bool} \\
\textit{upd-Ls}\ L\ t\ [] = L & \textit{ok-Ls}\ L\ t\ [] = \textit{True} \\
\textit{upd-Ls}\ L\ t\ (la \cdot las) = \textit{upd-Ls}\ (\textit{upd-L}\ L\ t\ la)\ t\ las & \textit{ok-Ls}\ L\ t\ (la \cdot las) = \\
 & \quad \textit{ok-L}\ L\ t\ la \wedge \textit{ok-Ls}\ (\textit{upd-L}\ L\ t\ la)\ t\ las \\
\\
\textit{upd-locks} :: ('l,{'t})\ \textit{locks} \Rightarrow {'t} \Rightarrow {'l}\ \textit{lock-acts} \Rightarrow ('l,{'t})\ \textit{locks} & \textit{ok-locks} :: ('l,{'t})\ \textit{locks} \Rightarrow {'t} \Rightarrow {'l}\ \textit{lock-acts} \Rightarrow \textit{bool} \\
\textit{upd-locks}\ \textit{ls}\ t\ \textit{las} = (\lambda l.\ \textit{upd-Ls}\ (\textit{ls}\ l)\ t\ (\textit{las}\ l)) & \textit{ok-locks}\ \textit{ls}\ t\ \textit{las} = (\forall l.\ \textit{ok-Ls}\ (\textit{ls}\ l)\ t\ (\textit{las}\ l))
\end{array}$$

Fig. 7 Semantics for lock BTAs: update functions (left) and precondition tests (right)

$$\begin{array}{ll}
\textit{upd-trl} :: \textit{nat} \Rightarrow {'t}\ \textit{lock} \Rightarrow {'t} \Rightarrow \textit{lock-act} \Rightarrow \textit{nat} & \textit{upd-trls} :: \textit{nat} \Rightarrow {'t}\ \textit{lock} \Rightarrow {'t} \Rightarrow \textit{lock-act list} \Rightarrow \textit{nat} \\
\textit{upd-trl}\ n\ L\ t\ \textit{Release} = n + \textit{has-locks}\ L\ t & \textit{upd-trls}\ n\ L\ t\ [] = n \\
\textit{upd-trl}\ n\ L\ t\ _ = n & \textit{upd-trls}\ n\ L\ t\ (la \cdot las) = \\
 & \quad \textit{upd-trls}\ (\textit{upd-trl}\ n\ L\ t\ la)\ (\textit{upd-L}\ L\ t\ la)\ t\ las
\end{array}$$

$$\begin{array}{l}
\textit{upd-TRL} :: {'l}\ \textit{tr-locks} \Rightarrow ('l,{'t})\ \textit{locks} \Rightarrow {'t} \Rightarrow {'l}\ \textit{lock-acts} \Rightarrow {'l}\ \textit{tr-locks} \\
\textit{upd-TRL}\ \textit{ln}\ \textit{ls}\ t\ \textit{las} = (\lambda l.\ \textit{upd-trls}\ (\textit{ln}\ l)\ (\textit{ls}\ l)\ t\ (\textit{las}\ l))
\end{array}$$

Fig. 8 Update functions for temporarily released locks

$$(\!|\textit{Unlock}{\rightarrow}l, \textit{Lock}{\rightarrow}l, \textit{Release}{\rightarrow}l, \textit{Suspend}\ w, \textit{IsIntrd}\ t\ \textit{False}, \ldots|\!)$$

This thread action achieves its goal as follows. First, $\textit{Unlock}{\rightarrow}l, \textit{Lock}{\rightarrow}l$ checks that the current thread holds l without effectively changing the lock status. Then, $\textit{Release}{\rightarrow}l$ releases l and $\textit{Suspend}\ w$ adds the thread to the wait set. $\textit{IsIntrd}\ t\ \textit{False}$ 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, $(\!|\textit{Lock}{\rightarrow}l, \textit{Unlock}{\rightarrow}l|\!)$ does not alter the lock state either, but checks that no other thread holds l. Conversely, BTAs of different groups are unordered, even though the $(\!|\ldots|\!)$ notation might conjure up the illusion of a total ordering.

2.2.3 Semantics of thread actions

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

Lock actions The semantics for lock BTAs is shown in Figure 7. The function *upd-L* maps lock BTAs to the operations on individual locks given in Figure 3. Note that *UnlockFail* does not change the lock because this BTA only queries the lock status. Similarly, *ok-L* checks the preconditions of the lock actions. The functions *upd-Ls*, *upd-locks* and *ok-Ls*, *ok-locks* lift these function to lists of lock BTAs for a single lock and to functions for all locks. Note how $\textit{ok-Ls}\ L\ t\ (la \cdot las)$ updates the lock L such that checking the remaining BTAs las takes the effect of the first BTA 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, too (Figure 8). They follow the same pattern. Recall 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 *upd-tp* and *upd-tps* update the thread pool (Figure 9). Spawned threads are stored under their thread ID with the initial

```
upd-tp :: ('l,'t,'x) tp ⇒ ('t,'x,'h) nt-act ⇒ ('l,'t,'x) tp
upd-tp tp (Spawn t x m)  = tp(t ↦ (x, λ_. 0))
upd-tp tp (ThreadEx t b) = tp

upd-tps ::
  ('l,'t,'x) tp ⇒ ('t,'x,'h) nt-act list ⇒ ('l,'t,'x) tp
upd-tps tp []         = tp
upd-tps tp (nt · nts) = upd-tps (upd-tp tp nt) nts
```

```
ok-tp :: ('l,'t,'x) tp ⇒ ('t,'x,'h) nt-act ⇒ bool
ok-tp tp (Spawn t x m)  = (tp t = None)
ok-tp tp (ThreadEx t b) = (b = (tp t ≠ None))

ok-tps :: ('l,'t,'x) tp ⇒ ('t,'x,'h) nt-act list ⇒ bool
ok-tps tp []         = True
ok-tps tp (nt · nts) =
  ok-tp tp nt ∧ ok-tps (upd-tp tp nt) nts
```

Fig. 9 Semantics for thread creation BTAs: update functions (left) and precondition tests (right)

```
locale final-thread = fixes final :: 'x ⇒ bool

ok-cond :: ('l,'t,'x,'h,'w) state ⇒ 't ⇒ 't cond-act ⇒ bool
ok-cond s t (Join t') = case tp s t' of None ⇒ True
                      | ⌊(x, ln)⌋ ⇒ t ≠ t' ∧ final x ∧ ln = (λ_. 0) ∧ wset s t' = None

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

Fig. 10 Semantics for thread join actions

state given in the BTA and no temporarily released locks. The predicates *ok-tp* and *ok-tps* check the preconditions, i.e., t is a free thread ID for *Spawn* t x m, and b in *ThreadEx* t b expresses whether t is not a free thread ID. Thread IDs are free iff they are not in the domain of the thread pool map.

Thread join actions Thread join actions do not affect the multithreaded state. Hence, there are no update functions, but only predicates for the preconditions (Figure 10). A thread t successfully joins on the thread t' iff (i) t' has not yet been started, i.e., $tp\ s\ t = None$, or (ii) t' is not the executing thread itself, t' has been fully evaluated, not temporarily released any locks, and is not in any wait set. 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. In Isabelle, the locale *final-thread* hides the *final* parameter.

Wait set actions For wait set actions, the semantics $t \vdash ws \models wa \Rightarrow ws'$ is defined as a relation, where t denotes the executing thread, ws and ws' the original and successor wait sets, and wa the wait set action to be executed. The rules in Figure 11 implement the wait set automaton from Figure 4. Unlike the other update functions, it is a relation because *Notify* w non-deterministically picks one thread t' from the wait set 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 $\_ \vdash \_ \models \_ \Rightarrow \_$ to lists of BTAs. To that end, I define the reflexive, transitive closure $r^{***} :: 'a \Rightarrow 'b\ list \Rightarrow 'a \Rightarrow bool$ of a ternary relation $r :: 'a \Rightarrow 'b \Rightarrow 'a \Rightarrow bool$ as

$$\frac{}{r^{***}\ a\ []\ a} \qquad \frac{r^{***}\ a\ bs\ a' \qquad r\ a'\ b\ a''}{r^{***}\ a\ (bs\ @\ [b])\ a''}$$

Then, $t \vdash \_\ [\models \_ \Rightarrow]\ \_$ is the reflexive, transitive closure of $t \vdash \_ \models \_ \Rightarrow \_$, i.e., the former folds the latter over a list of wait set actions.

The BTAs for wait sets do not have individual preconditions on t's wait set state. Rather, *Notified* and *WokenUp* characterise reductions of t that are meant to process notifications and interruptions. The predicate *ok-wsets* formalises that such reductions require the wait

$$\frac{}{t \vdash ws \models \mathit{Suspend}\ w \Rightarrow ws(t \mapsto \mathit{InWS}\ w)}$$

$$\frac{ws\ t' = \lfloor \mathit{InWS}\ w \rfloor}{t \vdash ws \models \mathit{Notify}\ w \Rightarrow ws(t' \mapsto \mathit{WS\text{-}Notified})} \qquad \frac{\forall t'.\ ws\ t' \neq \lfloor \mathit{InWS}\ w \rfloor}{t \vdash ws \models \mathit{Notify}\ w \Rightarrow ws}$$

$$\frac{}{t \vdash ws \models \mathit{NotifyAll}\ w \Rightarrow \lambda t.\ \mathit{if}\ ws\ t = \lfloor \mathit{InWS}\ w \rfloor\ \mathit{then}\ \lfloor \mathit{WS\text{-}Notified} \rfloor\ \mathit{else}\ ws\ t}$$

$$\frac{ws\ t' = \lfloor \mathit{InWS}\ w \rfloor}{t \vdash ws \models \mathit{WakeUp}\ t' \Rightarrow ws(t' \mapsto \mathit{WS\text{-}WokenUp})} \qquad \frac{\forall w.\ ws\ t' \neq \lfloor \mathit{InWS}\ w \rfloor}{t \vdash ws \models \mathit{WakeUp}\ t' \Rightarrow ws}$$

$$\frac{}{t \vdash ws \models \mathit{Notified} \Rightarrow ws(t := \mathit{None})} \qquad \frac{}{t \vdash ws \models \mathit{WokenUp} \Rightarrow ws(t := \mathit{None})}$$

Fig. 11 Semantics for wait set actions

$$\begin{array}{l}
\mathit{upd\text{-}int} :: {'t}\ \mathit{intrs} \Rightarrow {'t}\ \mathit{intr\text{-}act} \Rightarrow {'t}\ \mathit{intrs} \\
\mathit{upd\text{-}int}\ \mathit{is}\ (\mathit{Intr}\ t) \quad = \mathit{is} \cup \{t\} \\
\mathit{upd\text{-}int}\ \mathit{is}\ (\mathit{ClearIntr}\ t) = \mathit{is} - \{t\} \\
\mathit{upd\text{-}int}\ \mathit{is}\ (\mathit{IsIntrd}\ t\ b) \ = \mathit{is} \\
\\
\mathit{upd\text{-}ints} :: {'t}\ \mathit{intrs} \Rightarrow {'t}\ \mathit{intr\text{-}act\ list} \Rightarrow {'t}\ \mathit{intrs} \\
\mathit{upd\text{-}ints}\ \mathit{is}\ [] \quad = \mathit{is} \\
\mathit{upd\text{-}ints}\ \mathit{is}\ (\mathit{ia} \cdot \mathit{ias}) = \mathit{upd\text{-}ints}\ (\mathit{upd\text{-}int}\ \mathit{is}\ \mathit{ia})\ \mathit{ias}
\end{array}$$

$$\begin{array}{l}
\mathit{ok\text{-}intr} :: {'t}\ \mathit{intrs} \Rightarrow {'t}\ \mathit{intr\text{-}act} \Rightarrow \mathit{bool} \\
\mathit{ok\text{-}intr}\ \mathit{is}\ (\mathit{Intr}\ t) \quad = \mathit{True} \\
\mathit{ok\text{-}intr}\ \mathit{is}\ (\mathit{ClearIntr}\ t) = \mathit{True} \\
\mathit{ok\text{-}intr}\ \mathit{is}\ (\mathit{IsIntrd}\ t\ b) \ = (b = (t \in \mathit{is})) \\
\\
\mathit{ok\text{-}intrs} :: {'t}\ \mathit{intrs} \Rightarrow {'t}\ \mathit{intr\text{-}act\ list} \Rightarrow \mathit{bool} \\
\mathit{ok\text{-}intrs}\ \mathit{is}\ [] \quad = \mathit{True} \\
\mathit{ok\text{-}intrs}\ \mathit{is}\ (\mathit{ia} \cdot \mathit{ias}) = \\
\quad \mathit{ok\text{-}intr}\ \mathit{is}\ \mathit{ia} \wedge \mathit{ok\text{-}intrs}\ (\mathit{upd\text{-}int}\ \mathit{is}\ \mathit{ia})\ \mathit{ias}
\end{array}$$

Fig. 12 Semantics for interruption BTAs: update functions (left) and precondition test (right)

set status to be $\lfloor \mathit{WS\text{-}Notified} \rfloor$ and $\lfloor \mathit{WS\text{-}WokenUp} \rfloor$, respectively, and all other reductions require the wait set status to be *None*.

$$\begin{array}{l}
\mathit{ok\text{-}wsets}\ ws\ t\ was = (\mathit{if}\ \mathit{Notified} \in \mathit{set}\ was\ \mathit{then}\ ws\ t = \lfloor \mathit{WS\text{-}Notified} \rfloor \\
\qquad\qquad \mathit{else\ if}\ \mathit{WokenUp} \in \mathit{set}\ was\ \mathit{then}\ ws\ t = \lfloor \mathit{WS\text{-}WokenUp} \rfloor \\
\qquad\qquad \mathit{else}\ ws\ t = \mathit{None})
\end{array}$$

Interrupt actions Figure 12 defines the update functions *upd-int* and *upd-ints* and predicates *ok-intr* and *ok-intrs* for interrupt actions. The multithreaded semantics merely manages the set of interrupts, but it attaches no specific behaviour to them. So, *Intr* and *ClearIntr* have no preconditions, i.e., 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 are checked and executed in one step, it is straightforward to implement a compare-and-swap operation on the interrupt status. For example, $\langle\!|\mathit{IsIntrd}\ t\ \mathit{True}, \mathit{ClearIntr}\ t|\!\rangle$ checks that t is interrupted and clears it atomically.

Thread actions Finally, the semantics for thread actions combines these building blocks. The test *ok-ta* checks that the preconditions of all five BTA groups hold. The update relation *upd-ta* combines the update functions and relations, except for *upd-TRL*, which the interleaving semantics will apply directly when it updates the thread-local state.

$$\frac{\begin{array}{c}\mathit{ok\text{-}locks}\ (\mathit{locks}\ s)\ t\ \langle ta \rangle_{\mathrm{l}} \qquad \mathit{ok\text{-}tps}\ (\mathit{tp}\ s)\ \langle ta \rangle_{\mathrm{t}} \\ \mathit{ok\text{-}conds}\ s\ t\ \langle ta \rangle_{\mathrm{c}} \qquad \mathit{ok\text{-}wsets}\ (\mathit{wset}\ s)\ t\ \langle ta \rangle_{\mathrm{w}} \qquad \mathit{ok\text{-}intrs}\ (\mathit{intrs}\ s)\ \langle ta \rangle_{\mathrm{i}}\end{array}}{\mathit{ok\text{-}ta}\ s\ t\ ta}$$

$$\frac{\begin{array}{c}ls' = \mathit{upd\text{-}locks}\ ls\ t\ \langle ta \rangle_{\mathrm{l}} \qquad tp' = \mathit{upd\text{-}tps}\ tp\ \langle ta \rangle_{\mathrm{t}} \\ t \vdash ws\ [\models \langle ta \rangle_{\mathrm{w}} \Rightarrow]\ ws' \qquad is' = \mathit{upd\text{-}ints}\ is\ \langle ta \rangle_{\mathrm{i}}\end{array}}{\mathit{upd\text{-}ta}\ (ls, tp, h, ws, is)\ t\ ta\ (ls', tp', h, ws', is')}$$

2.2.4 Interleaving semantics

Now, I put everything together that I have defined so far to obtain the multithreaded semantics (level 5 in Figure 2). It takes the single-threaded semantics $r :: ('l,'t,'x,'h,'w,'o)$ *semantics* as a parameter that source and bytecode will instantiate accordingly. A single step in r is written $t \vdash (x,h) -ta\rightarrow (x',h')$. It denotes that the thread with ID t can atomically reduce in the thread-local state x with shared heap h to the thread-local state x' with the new shared heap h' with thread action ta. In Isabelle, the locale *multithreaded* fixes the parameters *final* (inherited from *final-thread*), r, and *acq-events*. The parameter *acq-events* produces the label (i.e., the sixth component of a thread action) for when a thread reacquires its temporarily released locks. The locale requires two basic properties: (i) *final* states are final, i.e., they cannot execute any further; and (ii) the shared heap in the thread creation BTAs indeed records the shared heap at spawn time.

type_synonym $('l,'t,'x,'h,'w,'o)$ *semantics* =
$'t \Rightarrow 'x \times 'h \Rightarrow ('l,'t,'x,'h,'w,'o)$ *thread-action* $\Rightarrow 'x \times 'h \Rightarrow$ *bool*

locale *multithreaded* = *final-thread* +
fixes $r :: ('l,'t,'x,'h,'w,'o)$ *semantics* $\quad (\_ \vdash \_ -\_\rightarrow \_)$
and *acq-events* $:: 'l$ *tr-locks* $\Rightarrow 'o$ *list*
assumes *final-no-red*: $[\![t \vdash (x,h) -ta\rightarrow (x',h');\ final\ x]\!] \Longrightarrow False$
and *Spawn-heap*: $[\![t \vdash (x,h) -ta\rightarrow (x',h');\ Spawn\ t''\ x''\ h'' \in set\ \langle ta\rangle_t]\!] \Longrightarrow h'' = h'$

The atomic steps of the multithreaded semantics *redT* with syntax $s -t{:}ta\rightarrow s'$ are given by two rules:

$$\textsc{Normal}\ \frac{tp\ s\ t = \lfloor (x, \lambda\_.\ 0)\rfloor \qquad t \vdash (x, shr\ s) -ta\rightarrow (x',h') \qquad ok\text{-}ta\ s\ t\ ta \qquad upd\text{-}ta\ (locks\ s, tp\ s, h', wset\ s, intrs\ s)\ t\ ta\ (ls', tp', h'', ws', is')}{s -t{:}ta\rightarrow (ls', tp'(t \mapsto (x', upd\text{-}TRL\ (\lambda\_.\ 0)\ ls\ t\ \langle ta\rangle_l)), h'', ws', is')}$$

$$\textsc{Acq}\ \frac{tp\ s\ t = \lfloor (x, ln)\rfloor \qquad ln \neq (\lambda\_.\ 0) \qquad \neg\ waiting\ (wset\ s\ t) \qquad may\text{-}acq\ (locks\ s)\ t\ ln}{s -t{:}(\lambda\_.\ [], [], [], [], [], acq\text{-}events\ ln)\rightarrow upd\text{-}acq\ s\ t\ x\ ln}$$

Each step is labelled with the ID t of the executing thread and the executed thread action ta. The rule NORMAL injects the atomic steps of the threads into the multithreaded semantics. Given a thread t with local state x and without any temporarily released locks ($\lambda\_.\ 0$), if t can reduce with the shared state $shr\ s$ to x' and h' with thread action ta, the interleaving semantics tests whether 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 state to h'. The other rule ACQ reacquires the locks ln that a thread t has temporarily released. This step requires that t is not in a wait set and that t may acquire all temporarily released locks. The predicate *may-acq* $:: ('l,'t)$ *locks* $\Rightarrow 't \Rightarrow 'l$ *tr-locks* $\Rightarrow$ *bool* checks the latter condition.

$$may\text{-}acq\ ls\ t\ ln = (\forall l.\ ln\ l > 0 \longrightarrow may\text{-}lock\ ls\ t\ l)$$

If so, *upd-acq* $s\ t\ x\ ln$ updates the multithreaded state to

$$(\lambda l.\ acquire\ (locks\ s\ l)\ t\ (ln\ l), (tp\ s)(t \mapsto (x, \lambda\_.\ 0)), shr\ s, wset\ s, intrs\ s)$$

where t's multiset of temporarily released locks is empty.

Complete runs are obtained by stringing together the atomic transitions of the interleaving semantics. Formally, a complete run consists of a possibly infinite list of transition labels and—if there are only finitely many labels—the terminal state. Possibly infinite lists where the constructor for the empty list carries a symbol $'b$ are defined by the codatatype of

terminated lazy lists

$$\textbf{codatatype}\ ('a,'b)\ tllist = []_{'b} \mid {}'a \cdot ('a,'b)\ tllist$$

Complete runs from a state s can be defined for arbitrary labeled transition systems $\_ -\!\!\triangleright \_$ coinductively by the two rules below, where $s \not\rightarrow$ denotes $\forall tl\ s'.\ \neg\ s \xrightarrow{tl}\triangleright s'$. (I use double rules to distinguish coinductive definitions from inductive ones.)

$$\textsc{Stop}\ \frac{s \not\rightarrow}{s \Rightarrow []_s} \qquad \textsc{Step}\ \frac{s \xrightarrow{tl}\triangleright s' \qquad s' \Rightarrow tls}{s \Rightarrow tl \cdot tls}$$

For $s \xrightarrow{(t,ta)}\triangleright s' = s -t{:}ta\rightarrow s'$, this gives the complete runs of the multithreaded semantics. The reflexive and transitive closure $redT^{***}$ of the interleaving semantics contains all the finite prefixes of complete runs and is written as $s -ttas\rightarrow^* s'$.

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

$$\frac{tp\ s\ t = \lfloor (x, \lambda\_.\ 0) \rfloor \qquad final\ x \qquad wset\ s\ t = None}{t \in final\text{-}threads\ s}$$

$$mfinal\ s = (dom\ (tp\ s) \subseteq final\text{-}threads\ s)$$

By the assumptions of the locale *multithreaded*, final states are indeed final:

Lemma 1 *If mfinal s, then there are no t, ta, s′ such that* $s -t{:}ta\rightarrow s'$.

2.3 JinjaThreads source code

I now present the JinjaThreads source code semantics for Java threads in detail, which in particular instantiates the framework from the previous section. In modelling Java threads, I closely follow Chapter 17 in the Java Language Specification [21] for Java 8, making minor abstractions where special cases would have unnecessarily complicated the formalisation.

2.3.1 Abstract syntax and typing

This section briefly presents the abstract syntax for JinjaThreads source code, bottom up. I focus on the parts relevant for threads; more details on the sequential aspects can be found in the Jinja paper [32] and my PhD thesis [49].

Abstract syntax falls into a generic part and one specific to source code. This way, bytecode can reuse the generic parts (Section 2.4). JinjaThreads defines only an abstract syntax, but no concrete input syntax. There is also a converter Java2Jinja (Section 8) that translates the concrete syntax of (a fragment of) Java into JinjaThreads abstract syntax [49, §6.5].

There are HOL types *cname* for class names, *mname* for method names, and *vname* for variable names and field names. In the following, C and D range over class names, M over method names, V over variable names, and F over field names. There are five kinds of values (HOL type *'addr val*, variable convention v): a dummy value *Unit*, booleans *Bool b* where b :: *bool*, 32-bit integers *Intg i* where i :: *word32*, the null reference *Null*, and references *Addr a* where a :: *'addr*. Addresses are a type variable such that the memory models can instantiate the type as needed (Section 4). JinjaThreads types (HOL type *ty*, variable convention T) are the type *Void* for *Unit*, primitive types *Boolean* and *Integer*, and three kinds of reference

expression	description	expression	description
new C	object allocation for class *C*	*Val v*	literal value
new T[*e*]	array allocation with element type *T*	$e_1 \ll bop \gg e_2$	binary operator
Cast T e	checked cast of *e* to type *T*	*Var V*	local variable access
e instanceof T	check for assignment compatibility	$V := e$	local variable assignment
$e.F\{D\}$	field access	$e_1[e_2]$	array cell access
$e_1.F\{D\} := e_2$	field assignment	$e_1[e_2] := e_3$	array cell assignment
$e.CAS(D.F, e', e'')$	atomic compare-and-set operation	*e.length*	array length
$e.M(es)$	method call	$e_1;;\ e_2$	sequential composition
throw e	exception throwing	*if* (e_1) e_2 *else* e_3	conditional
try e_1 *catch*$(C\ V)$ e_2	exception handling	*while* (e_1) e_2	while loop
sync (e_1) e_2	lock monitor e_1 for executing block e_2		
insync (a) *e*	execute *e* while having locked monitor *a* (not part of the input syntax)		
$\{V : T = vo; e\}$	local variable declaration with optional initial value *vo*		

Table 1 JinjaThreads expressions

types: *NT* for the null reference, *Class C* for classes, and $T[]$ for arrays with element type T. The predicate *is-refT* on types *ty* tests for reference types.

JinjaThreads source code, which I call *J*, is an imperative language where everything is an expression (HOL type *'addr expr*, variable convention *e*) with a return value: statements are modelled as expressions that return *Unit*. The datatype *expr* has 23 constructors, one for each kind of operation (Table 1). The following abbreviations are frequently used:

$$null = Val\ Null \qquad unit = Val\ Unit \qquad addr\ a = Val\ (Addr\ a)$$
$$Throw\ a = throw\ (addr\ a) \qquad \{V : T;\ e\} = \{V : T = None; e\}$$

An expression is considered to be final iff it is a value *Val v* or a thrown exception *Throw a*. The compare-and-set operation $e.CAS(D.F, e', e'')$ is a separate language construct, instead of Java native method. The reason is that the field F and the declaring class D cannot be passed as arguments to methods in JinjaThreads as it does not model reflection.

A program declaration (of type *'m prog*, variable convention *P*) is a list of class declarations, each of which consists of the class name and the class itself. The class declares its direct superclass, its fields and methods. A field declaration is a tuple of field name, type and a flag whether the field is volatile. A method declaration consists of the method name, a list of the parameters' types, the return type, and an optional method body (type *'m option*). If the method body is *None* (written *Native*), a native method is declared. The actual body is left as a type parameter *'m* such that all JinjaThreads languages can use this 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*.

A program declaration P induces a subtype relation $P \vdash T \leq T'$ on reference types with *Class Object* at the top, *NT* at the bottom, and all classes in between according to the subclass relation. The array type constructor $\_[]$ is covariant like in Java.

JinjaThreads requires the system classes *Object*, *Thread*, and *Throwable*, and the system exceptions *NullPointer*, *ClassCast*, *OutOfMemory*, *Arithmetic*, *ArrayIndexOutOfBounds*, *NegativeArraySize*, *ArrayStore*, *IllegalThreadState*, *Interrupted*, and *IllegalMonitorState*. System classes and exceptions differ from ordinary ones only in that every proper program must declare them under the specified name.

The type system for *J* is modelled as type judgements of the form $P, E \vdash e :: T$ where the environment E (of type $vname \rightharpoonup ty$) assigns types to local variables. $P, E \vdash es\ [::]\ Ts$ extends $P, E \vdash \_ :: \_$ pointwise to lists of expressions and types. For example,

$$\text{WTSYNC}\ \frac{P,E \vdash e_1 :: T_1 \qquad \textit{is-refT}\ T_1 \qquad T_1 \neq NT \qquad P,E \vdash e_2 :: T}{P,E \vdash \textit{sync}\ (e_1)\ e_2 :: T}$$

$$\text{WTCALL}\ \frac{\begin{array}{c} P,E \vdash e :: T \qquad P,E \vdash es\ [::]\ Ts' \\ \textit{class-of}\ T = \lfloor C \rfloor \qquad P \vdash C\ \textit{sees}\ M{:}Ts \to T_r = \textit{meth}\ \textit{in}\ D \qquad P \vdash Ts'\ [\leq]\ Ts \end{array}}{P,E \vdash e.M(es) :: T_r}$$

$$\text{WTCAS}\ \frac{\begin{array}{c} P,E \vdash e :: T \qquad P,E \vdash e' :: T' \qquad P,E \vdash e'' :: T'' \qquad \textit{class-of}\ T = \lfloor C \rfloor \\ P \vdash C\ \textit{sees}\ F{:}T_f\ (\textit{True})\ \textit{in}\ D \qquad P \vdash T' \leq T_f \qquad P \vdash T'' \leq T_f \end{array}}{P,E \vdash e.CAS(D.F,e',e'') :: \textit{Boolean}}$$

The typing rule WTSYNC for `synchronized` statements $\textit{sync}\ (e_1)\ e_2$ requires that the monitor expression e_1 must have a reference type, but not NT, and the return type of $\textit{sync}\ (e_1)\ e_2$ is e_2's type.

Rule WTCALL deals with method calls $e.M(es)$. The partial function *class-of* T determines the class C where the method lookup starts. It satisfies (i) $\textit{class-of}\ (\textit{Class}\ C) = \lfloor C \rfloor$, and (ii) $\textit{class-of}\ (T[]) = \lfloor \textit{Object} \rfloor$, and (iii) $\textit{class-of}\ T = \textit{None}$ for all other types T. In particular, $\textit{class-of}\ NT = \textit{None}$ disallows expressions like $\textit{null.wait}([])$ because `null` cannot directly be dereferenced in Java [21]. The notation $P \vdash C\ \textit{sees}\ M{:}Ts \to T_r = \textit{meth}\ \textit{in}\ D$ 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 $\textit{meth} :: 'm\ \textit{option}$ the optional method body. As programs must declare native methods, method lookup and typing work uniformly for native and non-native methods.

In the rule WTCAS for the compare-and-set operation, $P \vdash C\ \textit{sees}\ F{:}T_f\ (\textit{vol})\ \textit{in}\ D$ expresses that the class C has a field F declared in class D with type T_f and volatility status *vol*. So compare-and-set operations can only be used on volatile fields. This ensures that the Java memory model correctly treats the compare-and-set operation as an atomic synchronisation action. The return type is *Boolean* as the operation returns whether the comparison was successful.

JinjaThreads imposes standard well-formedness conditions on programs, which fall into two groups. First, the language-independent conditions in particular require that the class hierarchy must be acyclic, method overriding contravariant in the parameters and covariant in the return types, and the signatures of native methods must match the signature that JinjaThreads expects. Among others, the following native methods are allowed:[4]

$\textit{Thread.start}([]) :: \textit{Void}$	$\textit{Object.interrupted}([]) :: \textit{Boolean}$	$\textit{Object.wait}([]) :: \textit{Void}$
$\textit{Thread.join}([]) :: \textit{Void}$	$\textit{Object.currentThread}([]) :: \textit{Class Thread}$	$\textit{Object.notify}([]) :: \textit{Void}$
$\textit{Thread.interrupt}([]) :: \textit{Void}$		$\textit{Object.notifyAll}([]) :: \textit{Void}$
$\textit{Thread.isInterrupted}([]) :: \textit{Boolean}$		

The first column lists the methods of *Thread* for spawning (*start*), joining on (*join*), interrupting (*interrupt*), and testing for interruption (*isInterrupted*) of a thread. The second column 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 both operate on the current thread: *interrupted* checks and clears the interrupt status, and *currentThread* returns the associated *Thread* object. The last column specifies *Object*'s methods for the wait-notify-mechanism.

Second, the language-specific conditions are imposed on the bodies of all declared methods. For J, the parameter names must be distinct and equally many as the parameter types,

[4] JinjaThreads provides further native methods (e.g., printing) that are not related to multithreading and therefore not covered in this article. The details can be found in [49].

datatype *hty* = *Class cname* | *Array ty nat* **datatype** *loc* = *Field cname vname* | *Cell nat*

locale *heap* =
fixes *typeof-addr* :: *'heap* ⇒ *'addr* ⇀ *hty*
and *empty-heap* :: *'heap*
and *alloc* :: *'heap* ⇒ *hty* ⇒ (*'heap* × *'addr*) *set*
and *read* :: *'heap* ⇒ *'addr* ⇒ *loc* ⇒ *'addr val* ⇒ *bool*
and *write* :: *'heap* ⇒ *'addr* ⇒ *loc* ⇒ *'addr val* ⇒ *'heap* ⇒ *bool*
and *P* :: *'m prog*

assumes *alloc-type*: ⟦(*h'*, *a*) ∈ *alloc h T*; *is-type P T*⟧ ⟹ *typeof-addr h' a* = ⌊*T*⌋
and *alloc-hext*: (*h'*, *a*) ∈ *alloc h T* ⟹ *h* ⊴ *h'*
and *write-hext*: *write h a al v h'* ⟹ *h* ⊴ *h'*

Fig. 13 Interface for the shared heap

and the bodies must be well-typed with a subtype of the declared return type and pass the definite assignment check. Formally, the well-formedness predicate *wf-prog wf-md P* is parametrized by the language-specific well-formedness check for method declarations *wf-md*. For *J*, *wf-J-prog P* instantiates *wf-md* with the checks described.

2.3.2 Abstract heap module

The shared heap h is formalised as an abstract datatype (ADT) using a locale (Figure 13). The ADT offers the advantage that the different memory models can provide different implementations and all definitions and theorems immediately carry over. The type variables *'heap* and *'addr* represent the abstract types of the heap and of addresses.

The operation *typeof-addr* returns the dynamic type of addresses (type *hty*) in a given heap, which can either be a class name *Class C* or an array *Array T n* with n cells of type T. I overload *Class* :: *cname* ⇒ *ty* and *Class* :: *cname* ⇒ *hty* and similarly for *is-type*.

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. As the JMM requires allocation to be non-deterministic [50], *alloc* returns the set of all possible updated heaps h' with the allocated address a. The set is empty if the allocation fails, e.g., due to insuffcient memory. If the allocation succeeds, a's type information in h' must be correct—provided that the allocated type is valid (assumption *alloc-type*). A type T is valid (notation *is-type P T*) iff all classes it refers to exist in the program and, in case of array types, the element type is valid and not *NT*.

The relations *read* and *write* model access to the heap. The member *al* :: *loc* specifies which field (*al* = *Field D F*) or array cell (*al* = *Cell n*) of an address to access. An address a and a member *al* identify a location (a, al). The predicate *read h a al v* holds if v can be read from the location (a, al) in the heap h. Similarly, writing to a location updates the heap (*write h a al v h'*). Again, the JMM requires that these operations be non-deterministic. Allocations and writes must be implemented such that they extend the heap ($h \trianglelefteq h'$), i.e., dynamic type information grows monotonically. Formally, $h \trianglelefteq h'$ iff *typeof-addr h* $\subseteq_{\text{m}}$ *typeof-addr h'*.

All heap-dependent definitions and theorems reside in this locale or its descendants.

2.3.3 Semantics of native methods

Most of Java's concurrency features are implemented in native methods. I now present the semantics of these native methods (level 2). $P,t \vdash \langle a.M(vs), h\rangle -ta\rightarrow_{\text{native}} \langle va, h'\rangle$ denotes that when thread t calls method M on the receiver object at address a with parameters vs in the heap h, then the thread action ta is issued, va returned and the heap updated to h'. The

framework type variable		source code	bytecode
lock ID	$'l$	$'addr$	$'addr$
thread ID	$'t$	$'addr$	$'addr$
thread-local state	$'x$	$'addr\ expr \times (vname \rightharpoonup 'addr\ val)$	$'addr\ option \times 'addr\ frame\ list$
shared state	$'h$	$'heap$	$'heap$
wait set ID	$'w$	$'addr$	$'addr$
memory model	$'o$	not needed for this article, see [49, 50] for details	

Table 2 Instantiation of type variables of the framework semantics

result *va* can either be a value (*Ret-Val v*), the address of a system exception *Ret-sys-xcpt C*, or the special value *Ret-Again*, which is used for non-atomic native calls.

Thanks to the multithreading semantics, the semantics of native methods merely issues suitable thread actions. Therefore, I first specify how source code and bytecode instantiate the type variables in the multithreading semantics (Table 2). Addresses identify locks and wait sets as every object has one monitor and one wait set. Threads are identified by their associated object. The thread-local states are discussed in Sections 2.3.4 and 2.4.2.

The JLS specifies the native methods only incompletely or not at all. Hence, I relied on the Java API [26] 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 14 shows the semantics for the native methods. The rules for the methods *start*, *join*, *interrupt*, and *isInterrupted* have the premises *typeof-addr h a* $= \lfloor Class\ C \rfloor$ and $P \vdash C \preceq^* Thread$, where $\_ \vdash \_ \preceq^* \_$ denotes the reflexive and transitive closure of the subclass relation.

Fork and join Rule 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, START cannot include the concrete initial state of the new thread in the *Spawn* BTA, because their state representations differ. Instead, it specifies to execute the parameter-less *run* method that class *C* sees with receiver object *a*. Source code and bytecode convert this BTA into the state representation as required. If the thread has already been started, STARTFAIL raises an *IllegalThreadState* exception.

The *join* method waits for the receiver thread *a* to terminate, so JOIN includes the basic thread action *Join a*. However, the API specifies that *join* first has to test whether the current thread *t* has not been interrupted. Otherwise, it raises an *Interrupted* exception (JOININTR). Recall that the interleaving semantics picks the reductions JOIN or JOININTR only if their thread action's precondition is satisfied. So, if *a* is not *final* and *t* is not interrupted, the call to *join* gets stuck until either *a* terminates or *t* gets interrupted.

Although the implementation of class *Thread* in Oracle's JDK SE 8 declares (and has always declared in previous versions) the methods *start* and *join* as `synchronized`, neither the JLS nor the API require this. Hence, none of the rules for *start* and *join* includes *Lock* and *Unlock* actions, as such synchronisation would erroneously hide data races.

Interruption When a thread interrupts another thread t' via 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*). But 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 (ISINTRD), whereas the *interrupted* method returns and clears the interrupt status of the executing thread (INTRDT and INTRDF). A call to *currentThread* returns the address of the object associated with the current thread (CURRTH).

START: $P,t \vdash \langle a.start([]),h\rangle -(\!|Spawn\ a\ (C,run,a)\ h|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

STARTFAIL: $P,t \vdash \langle a.start([]),h\rangle -(\!|ThreadEx\ a\ True|\!)\rightarrow_{\text{native}} \langle Ret\text{-}sys\text{-}xcpt\ IllegalThreadState,h\rangle$

JOIN: $P,t \vdash \langle a.join([]),h\rangle -(\!|Join\ a,IsIntrd\ t\ False|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

JOININTR: $P,t \vdash \langle a.join([]),h\rangle -(\!|IsIntrd\ t\ True,ClearIntr\ t|\!)\rightarrow_{\text{native}} \langle Ret\text{-}sys\text{-}xcpt\ Interrupted,h\rangle$

INTR: $P,t \vdash \langle a.interrupt([]),h\rangle -(\!|ThreadEx\ a\ True,WakeUp\ a,Intr\ a|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

INTRINEX: $P,t \vdash \langle a.interrupt([]),h\rangle -(\!|ThreadEx\ a\ False|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

ISINTRD: $P,t \vdash \langle a.isInterrupted([]),h\rangle -(\!|IsIntrd\ a\ b|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ (Bool\ b),h\rangle$

CURRTH: $P,t \vdash \langle a.currentThread([]),h\rangle -(\!|\,|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ (Addr\ t),h\rangle$

INTRDT: $P,t \vdash \langle a.interrupted([]),h\rangle -(\!|IsIntrd\ t\ True,ClearIntr\ t|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ (Bool\ True),h\rangle$

INTRDF: $P,t \vdash \langle a.interrupted([]),h\rangle -(\!|IsIntrd\ t\ False|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ (Bool\ False),h\rangle$

WAITFAIL: $P,t \vdash \langle a.wait([]),h\rangle -(\!|UnlockFail{\rightarrow}a|\!)\rightarrow_{\text{native}} \langle Ret\text{-}sys\text{-}xcpt\ IllegalMonitorState,h\rangle$

WAITINTRD1: $P,t \vdash \langle a.wait([]),h\rangle -(\!|Unlock{\rightarrow}a,Lock{\rightarrow}a,IsIntrd\ t\ True,ClearIntr\ t|\!)\rightarrow_{\text{native}}$ $\langle Ret\text{-}sys\text{-}xcpt\ Interrupted,h\rangle$

WAIT: $P,t \vdash \langle a.wait([]),h\rangle -(\!|Suspend\ a,Unlock{\rightarrow}a,Lock{\rightarrow}a,Release{\rightarrow}a,IsIntrd\ t\ False|\!)\rightarrow_{\text{native}}$ $\langle Ret\text{-}Again,h\rangle$

NTFD: $P,t \vdash \langle a.wait([]),h\rangle -(\!|Notified|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

WAITINTRD2: $P,t \vdash \langle a.wait([]),h\rangle -(\!|WokenUp,ClearIntr\ t|\!)\rightarrow_{\text{native}} \langle Ret\text{-}sys\text{-}xcpt\ Interrupted,h\rangle$

SPURIOUS: $P,t \vdash \langle a.wait([]),h\rangle -(\!|Unlock\ a,Lock\ a,Release\ a,IsIntrd\ t\ False|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

NTF: $P,t \vdash \langle a.notify([]),h\rangle -(\!|Unlock{\rightarrow}a,Lock{\rightarrow}a,Notify\ a|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

NTFFAIL: $P,t \vdash \langle a.notify([]),h\rangle -(\!|UnlockFail{\rightarrow}a|\!)\rightarrow_{\text{native}} \langle Ret\text{-}sys\text{-}xcpt\ IllegalMonitorState,h\rangle$

NTFALL: $P,t \vdash \langle a.notifyAll([]),h\rangle -(\!|Unlock{\rightarrow}a,Lock{\rightarrow}a,NotifyAll\ a|\!)\rightarrow_{\text{native}} \langle Ret\text{-}Val\ Unit,h\rangle$

NTFALLFAIL: $P,t \vdash \langle a.notifyAll([]),h\rangle -(\!|UnlockFail{\rightarrow}a|\!)\rightarrow_{\text{native}} \langle Ret\text{-}sys\text{-}xcpt\ IllegalMonitorState,h\rangle$

Fig. 14 Semantics of native methods. The rules for methods *start*, *join*, *interrupt*, and *isInterrupted* additionally have the premises *typeof-addr h a* $= \lfloor Class\ C \rfloor$ and $P \vdash C \preceq^* Thread$.

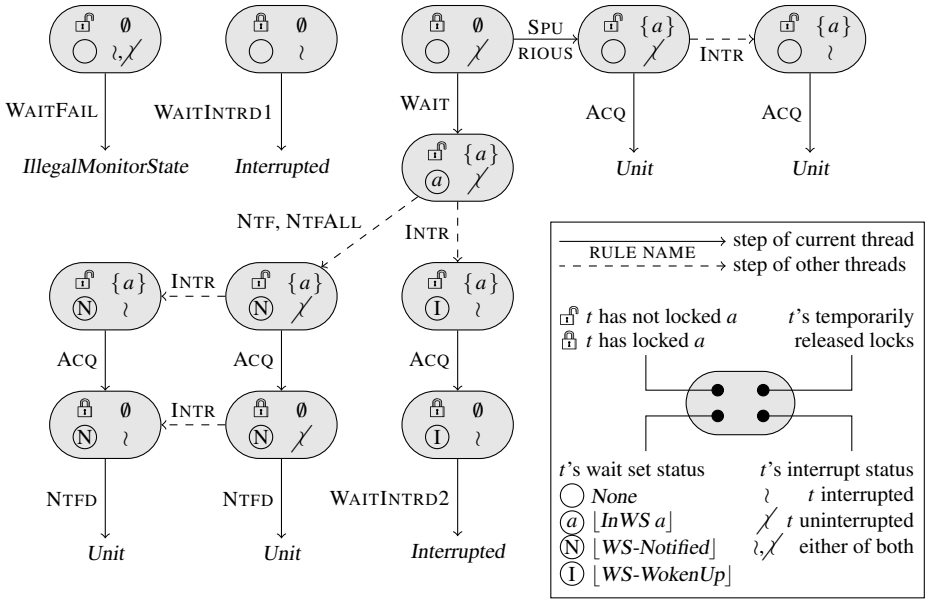


Fig. 15 Steps for t executing a call to *wait* on object a

Waiting and notification The rules for *wait*, *notify*, and *notifyAll* are more complicated. I start with *notify* and *notifyAll*. A call to either of them first tests via $Unlock{\rightarrow}a, Lock{\rightarrow}a$ 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 (NTF and NTFALL). Otherwise, $UnlockFail{\rightarrow}a$ checks that the thread does not hold the lock; rules NTFFAIL and NTFALLFAIL then raise an *IllegalMonitorState* exception.

Figure 15 illustrates as a transition system how the rules for *wait* implement the wait-notify mechanism. Every state is defined by four relevant components which are explained in the box on the right. The transitions are labelled by the rules that generate them – solid lines denote steps by the thread calling *wait*, dashed lines denote steps of other threads. From the three inital states without incoming arrows, the transitions lead to what the call returns, i.e., *Unit* or a system exception. Observe that the transitions from the initial state in the top centre step through the wait set automaton from Figure 4.

If the thread has not locked the receiver object a (top left state), WAITFAIL raises an *IllegalMonitorState* exception. If t does hold the lock, *wait* tests whether t has a pending interrupt.[5] If so (second state in the top row), WAITINTRD1 clears it and raises an *Interrupted* exception. Otherwise, the thread is in the state in the top 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, WAIT suspends t to a's wait set and temporarily releases all locks on a. Whether t will be interrupted or notified determines whether *wait* should return normally or raise an *Interrupted* exception. Thus, the return value cannot be determined at this point of time. Therefore, WAIT returns the special token *Ret-Again*, which indicates that the thread's next step should be to call the same method with the same parameters once more. This effectively splits the call to *wait* into two steps. Since WAIT suspends t to the wait set a, the multithreading semantics expects t to process its removal from the wait set in its next step (NTFD and WAITINTRD2 use the BTAs *Notified* and *WokenUp*, respectively). But before the multithreading semantics picks one of these, t must have been removed from the wait set and reacquired the locks on a. Hence, when another thread calls *notify* or *notifyAll* on a, or *interrupt* on t, t is removed from the wait set a. Note that this determines the result of the call to *wait*. So, t's next step is to reacquire the locks on a. There is no need to add a rule for that to t's semantics because ACQ of the interleaving semantics takes care of this. Next, the second call to *wait* processes t's removal from the wait set, which NTFD and WAITINTRD2 implement. The latter also clears t's interrupt status as required by the JLS [21, §17.8.1]. Another thread may interrupt t after t is has been notified, but before NTFD processes t's removal. In that case, the call to *wait* returns normally and t's interrupt status is still set.

The JLS allows, but does not require, spurious wake-ups [21, §17.8.1], i.e., a call to *wait* may return without interruption and notification. Spurious wake-ups are relevant to determine when programs are correctly synchronized [50]. In JinjaThreads, threads can wake up spuriously, but they do not have to. Instead of suspending itself upon a call to *wait*, a thread can choose to instantaneously wake up spuriously (SPURIOUS), i.e., it only temporarily releases the lock on the monitor a. Since I do not model a specific scheduler, t may postpone to reacquire a's lock until other threads make progress. Hence, instantaneous wake-ups cover all spurious wake-ups because the other threads cannot tell whether another thread has woken up spontaneously.

A simpler model for spurious wake-ups would be to include a rule without preconditions that removes any thread from any wait set any time. However, this does not work well with

[5] Neither the JLS [21, Ch. 17.8] nor the Java API [26] 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.

considering complete runs as discussed in Section 2.2.4: If there is at least only one possible step, one such will be taken. This assumption is also the basis for type safety proofs via progress and preservation (Section 3). However, this can enforce spurious wake-ups (rather than discourage them as the JLS does). Consider, e.g.,

```
m = new Object(); synchronized (m) { m.wait(); } print "X";
```

When run with Oracles Hotspot and OpenJDK VMs, this program with only one thread deadlocks, because the thread waits forever for being notified or interrupted, but there is no thread to do so. Hence, it never prints `X`. With the simpler model for spurious wake-ups, such a deadlock could not occur. The semantics would spuriously wake-up the thread and run it to completion, i.e., the program would always print `X` and terminate. My semantics produces both behaviours: If the call to wait chooses to instantaneously wake up, the program prints `X` and terminates. Otherwise, it chooses to wait in the wait set and deadlocks.

Interaction of interruption and notification While a thread t is in a wait set, it may be notified and interrupted simultaneously. The JLS demands that t determines an order over these causes and behave accordingly. That is, if the notification comes first, t's interrupt is pending and the call returns normally. If the interrupt comes first, t's call throws an *Interrupted* exception, but the notification must not be lost, i.e., another thread in the wait set must be notified, if there is any.

JinjaThreads meets this requirement as follows: Notifications use the BTAs *Notify* or *NotifyAll* whereas interrupts use *WakeUp*. Therefore, the wait set status $\lfloor \mathit{WS\text{-}Notified} \rfloor$ and $\lfloor \mathit{WS\text{-}WokenUp} \rfloor$ records the cause of the removal. This determines whether NTFD or WAITINTRD2 will process the removal, i.e., whether *wait* returns normally or throws an *Interrupted* exception. In particular, NTFD and WAITINTRD2 ignore on the interrupt status when they execute. Otherwise, if they did consider it in the obvious way, 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 to the left in Figure 15). Hence, when t_1 processes its removal, its interrupt status has been set, i.e., it raises an *Interrupted* exception. But now, the notification is lost, because t_2 remains in the wait set. This violates the above requirement.

The JLS does not require that the order over the concurrent notification and interruption be consistent with other orderings. In fact, Oracle's HotSpot 6 and 7 sometimes choose an inconsistent order [50]. In JinjaThreads, the interleaving of threads defines an order on notifications (BTA *Notify* and *NotifyAll*) and interrupts (BTA *WakeUp*) from different threads which is consistent with all other orderings. Thus, the JinjaThreads cannot produce behaviours with inconsistent orders.

2.3.4 Semantics of JinjaThreads source code

The core semantics for *J* on level 3 is a small-step semantics written $P,t \vdash \langle e,s \rangle -ta\rightarrow \langle e',s' \rangle$. I say that the expression e reduces in state s to e' and state s' with thread action ta. The states consists of the shared heap and a store for local variables (type $\mathit{vname} \rightharpoonup {}'\mathit{addr\ val}$). It is a standard semantics with rules for subexpression reduction and exception propagation. An expression is *final*, i.e., fully reduced, when it is a value *Val v* or a thrown exception *Throw a*.

Figure 16 shows the rules relevant for multithreading, namely for `synchronized` blocks and method calls. These rules illustrate the main ideas; for an extensive discussion, see [32, 49]. The expression *sync* (e_1) e_2 models Java's `synchronized` statements as specified in

RSYNC1:
$$\frac{P,t \vdash \langle e_1, s\rangle -ta\rightarrow \langle e_1', s'\rangle}{P,t \vdash \langle \mathit{sync}\ (e_1)\ e_2, s\rangle -ta\rightarrow \langle \mathit{sync}\ (e_1')\ e_2, s'\rangle}$$

RSYNCN:
$$P,t \vdash \langle \mathit{sync}\ (\mathit{Val\ Null})\ e, s\rangle -\langle\!|\!|\rightarrow \langle \mathit{THROW\ NullPointer}, s\rangle$$

RSYNCX:
$$P,t \vdash \langle \mathit{sync}\ (\mathit{Throw}\ a)\ e, s\rangle -\langle\!|\!|\rightarrow \langle \mathit{Throw}\ a, s\rangle$$

RLOCK:
$$P,t \vdash \langle \mathit{sync}\ (\mathit{addr}\ a)\ e, s\rangle -\langle\!|\mathit{Lock}{\rightarrow}a|\!|\rightarrow \langle \mathit{insync}\ (a)\ e, s\rangle$$

RSYNC2:
$$\frac{P,t \vdash \langle e, s\rangle -ta\rightarrow \langle e', s'\rangle}{P,t \vdash \langle \mathit{insync}\ (a)\ e, s\rangle -ta\rightarrow \langle \mathit{insync}\ (a)\ e', s'\rangle}$$

RUNLOCK:
$$P,t \vdash \langle \mathit{insync}\ (a)\ (\mathit{Val}\ v), s\rangle -\langle\!|\mathit{Unlock}{\rightarrow}a|\!|\rightarrow \langle \mathit{Val}\ v, s\rangle$$

RUNLOCKX:
$$P,t \vdash \langle \mathit{insync}\ (a)\ (\mathit{Throw}\ a'), s\rangle -\langle\!|\mathit{Unlock}{\rightarrow}a|\!|\rightarrow \langle \mathit{Throw}\ a', s\rangle$$

RCALL:
$$\frac{\begin{array}{c}\mathit{typeof\text{-}addr}\ h\ a\ = \lfloor T \rfloor \\ P \vdash \mathit{class\text{-}of}'\ T\ \mathit{sees}\ M{:}Ts{\rightarrow}T_r = \lfloor (pns, body) \rfloor\ \mathit{in}\ D \qquad |vs| = |pns| \qquad |Ts| = |pns|\end{array}}{\begin{array}{l}P,t \vdash \langle \mathit{addr}\ a.M(\mathit{map\ Val}\ vs), (h,x)\rangle \\ \quad -\langle\!|\!|\rightarrow \langle \mathit{blocks}\ (\mathit{this}\cdot pns)\ (\mathit{Class}\ D\cdot Ts)\ (\mathit{Addr}\ a\cdot vs)\ body, (h,x)\rangle\end{array}}$$

RNATIVE:
$$\frac{\mathit{typeof\text{-}addr}\ h\ a = \lfloor T \rfloor \qquad P \vdash \mathit{class\text{-}of}'\ T\ \mathit{sees}\ M{:}Ts{\rightarrow}T_r = \mathit{Native}\ \mathit{in}\ D \qquad P,t \vdash \langle a.M(vs), h\rangle -ta\rightarrow_{\text{native}} \langle vx, h'\rangle \qquad e' = \mathit{native\text{-}Ret2J}\ (\mathit{addr}\ a.M(\mathit{map\ Val}\ vs))\ vx}{P,t \vdash \langle \mathit{addr}\ a.M(\mathit{map\ Val}\ vs), (h,x)\rangle -\mathit{native\text{-}TA2J}\ P\ ta\rightarrow \langle e', (h',x)\rangle}$$

RCAS:
$$\frac{\mathit{read}\ h\ a\ (\mathit{Field}\ D\ F)\ v \qquad \mathit{write}\ h\ a\ (\mathit{Field}\ D\ F)\ v\ h'}{P,t \vdash \langle \mathit{addr}\ a.\mathit{CAS}(D.F, \mathit{Val}\ v, \mathit{Val}\ v'), (h,x)\rangle -\langle\!|\!|\rightarrow \langle \mathit{true}, (h',x)\rangle}$$

RCASF:
$$\frac{\mathit{read}\ h\ a\ (\mathit{Field}\ D\ F)\ v'' \qquad v \neq v''}{P,t \vdash \langle \mathit{addr}\ a.\mathit{CAS}(D.F, \mathit{Val}\ v, \mathit{Val}\ v'), (h,x)\rangle -\langle\!|\!|\rightarrow \langle \mathit{false}, (h,x)\rangle}$$

Fig. 16 Semantics of `synchronized` blocks, method calls, and compare-and-set operations

[21, §14.19]. JinjaThreads does not model `synchronized` methods explicitly, because they are syntactic sugar for ordinary methods with their body inside a `synchronized` statement.

Rule RSYNC1 reduces the monitor subexpression. If the monitor subexpression becomes *Null*, a *NullPointer* exception is raised (RSYNCN), where *THROW NullPointer* denotes the address of the pre-allocated *NullPointer* exception. If an exception is raised while reducing the monitor subexpression, RSYNCX 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, RLOCK rewrites the *sync* $(\mathit{addr}\ a)\ e$ expression to *insync* $(a)\ e$ to remember that the lock has been granted (*insync* $(a)\ e$ expressions are not part of the input language as there is no typing rule for them). Then, RSYNC2 executes the body. Once it has become a value or raised an exception, RUNLOCK and RUNLOCKX 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 *insync* $(a)\ e$ calls *wait* on a. The basic thread action *Release* in WAIT and temporarily released locks in the interleaving semantics (cf. ACQ) take care of this.

RCALL and RNATIVE are the main rules for calling a normal and native method, respectively. The rules for evaluating the receiver expression and the parameters are standard [32] and not shown. So, suppose that the receiver expression has evaluated to an address a with dynamic type T. If the called method M for the receiver is not native, RCALL looks up the method definition in P and inlines the method body. (The function *class-of*′ computes the class where the method lookup starts: *class-of*′ $(\mathit{Class}\ C) = C$ and *class-of*′ $(\mathit{Array}\ T\ n) = \mathit{Object}$, and *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, RNATIVE dispatches the call to the semantics for native methods (Section 2.3.3). Then, the thread action *ta* and result *vx* are converted into *J* using the functions *native-TA2J* and *native-Ret2J*. The function *native-TA2J* replaces the initial states (C, M, a) of threads spawned in *ta* with the method body that the method lookup finds for *M* starting at *C* and sets the *this* pointer to *Addr a*. Similarly, *native-Ret2J* converts the result into *J* syntax:

$$\begin{aligned}
&\textit{native-Ret2J}\ e\ (\textit{Ret-Val}\ v) &&= \textit{Val}\ v\\
&\textit{native-Ret2J}\ e\ (\textit{Ret-sys-xcpt}\ C) &&= \textit{THROW}\ C\\
&\textit{native-Ret2J}\ e\ \textit{Ret-Again} &&= e
\end{aligned}$$

In particular, in the case of WAIT, the expression of the statement does not change, so the next step will be another call to *wait* as required.

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. The progress theorem (Lemma 12) shows that the type system rules out such cases.

Compare-and-set uses the operations of the abstract heap in the rules RCAS and RCASF. When the expected value *v* can be read from the location $(a, \textit{Field}\ D\ F)$ in the heap, then RCAS immediately writes the other value v' to the same location and the result *true* records the success of the comparison. Conversely, when an other value v'' can be read, RCASF does not update the heap and returns *false*. So, only a successful compare-and-set operations acts like an atomic read and write on the location; a failing one behaves just like a read.[6] The rules for subexpression reduction, exception propagation, and throwing of a *NullPointer* exception are similar to those for `synchronized` blocks except for unlocking and not shown.

I obtain the interleaving semantics for *J* by instantiating the locale *multithreaded* from Section 2.2.4 with $\textit{J-red} = \lambda t\ ((e,x),h)\ ta\ ((e',x'),h').\ P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow \langle e',(h',x')\rangle$ for *r* and $\textit{J-final} = \lambda(e,x).\ \textit{final}\ e$ for *final*.

Lemma 2 *J-final and J-red P are well-formed with respect to the interleaving semantics.*

Proof I must discharge the assumptions of the locale *multithreaded*. *final-no-red* follows by case analysis of the rules. *Spawn-heap* holds by induction on the small step semantics and case analysis on the semantics for native methods. □

The start state *J-start P C M vs* for program *P* has exactly one thread *start-tID* with thread-local state $(\textit{blocks}\ (\textit{this}\cdot \textit{pns})\ (\textit{Class}\ D\cdot Ts)\ (\textit{Null}\cdot vs)\ \textit{body}, \textit{empty})$ where $P \vdash C\ \textit{sees}\ M{:}Ts\rightarrow\_ = \lfloor(\textit{pns},\textit{body})\rfloor\ \textit{in}\ D$. Hence, *start-tID* is about to execute the non-native method *M* in class *C* with parameters *vs*. Setting the *this* pointer to *Null* simulates a static method. 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 pending interrupts. A start state is well-formed (written *wf-start P C M vs*) iff *C* sees a non-native method *M* and the parameters *vs* conform to *M*'s parameter types.

2.4 Bytecode and the virtual machine

This section presents JinjaThreads's bytecode language (Section 2.4.1) and virtual machine (VM, Section 2.4.2). They model Java bytecode and the Java VM according to the Java Virtual Machine Specification (JVMS) [42].

[6] This behaviour models the Java 9 specification of the compare-and-set operation. In Java 8 and before, it was unclear whether a failing compare-and-set operation has also the synchronisation effect of writing the old value [73].

2.4.1 Bytecode

The bytecode language JVM reuses many concepts from source code. For program declarations, it suffices to specify the type of method bodies *'addr jvm-method* to be plugged in for *'m*. Everything else remains unchanged, in particular, the lookup functions, subtyping and generic well-formedness.

A method body (msl, mxl, ins, xt) consists of an instruction list *ins*, an exception table *xt*, the maximum stack length *msl*, and the number *mxl* of required registers, not counting the *this* pointer and parameters. The lookup functions *instrs-of P C M* and *ex-table-of P C M* extract the instruction list and exception table for method *M* in class *C* from *P*.

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*

The exception table *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 *Any* = *None*. The exception handler starting at the index *pc* in *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 *Any*, it handles all.

Any might seem redundant because all exceptions must be subclasses of *Throwable*, i.e., $\lfloor Throwable \rfloor$ could replace *Any*. However, I include *Any* for two reasons: First, the Java Virtual Machine specification (JVMS) [42, §4.7.3] also specifies such a "catch-all" value, which is meant for compiling `finally` blocks. Second, $\lfloor Throwable \rfloor$ and *Any* are interchangeable only if one can prove that all raised exceptions are subclasses of *Throwable*, which requires a type safety proof. With *Any*, the compiler verification can avoid the subject reduction and preservation proofs for the intermediate language by not relying on such invariants.

JinjaThreads supports 24 different bytecode instructions (Table 3). If not provided explicitly, operands are taken from the stack and results pushed onto the stack. In comparison to Java bytecode, JinjaThreads unifies instructions that only differ in their operand types (e.g., `aload` and `iload`) in polymorphic ones (e.g., *Load*), but the instructions have not been simplified conceptually. The instruction *CAS* does not exist in Java bytecode where compare-and-set operations are native methods taking a variable handle as the field specification. As JinjaThreads does not model such reflection, I added the compare-and-set instruction. Moreover, a few instructions for stack and register manipulation (e.g., `dup2`, `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 `byte` and `float` nor advanced control flow instructions like `tableswitch` and `jsr` for subroutines.

2.4.2 The virtual machine

The state space is taken from the Jinja VM [32]. The state (xcp, h, frs) of type *jvm-state* consists of an exception flag *xcp* ($\lfloor a \rfloor$ corresponds to *Throw a* in *J* and *None* denotes none), a heap *h* and a stack of call frames. A state is final iff the call stack is empty.

type_synonym $('addr, 'heap)$ *jvm-state* = *'addr option* × *'heap* × *'addr frame list*
type_synonym *'addr frame* = *'addr opstack* × *'addr registers* × *cname* × *mname* × *pc*
type_synonym *'addr opstack* = *'addr val list*
type_synonym *'addr registers* = *'addr val list*
type_synonym *pc* = *nat*

Load i	load from register *i*	*Getfield F C*	fetch field *F* declared in class *C*
Store i	store into register *i*	*Putfield F C*	set field *F* declared in class *C*
Push v	push literal value *v* on stack	*CAS F C*	compare-and-set on field *F* of class *C*
Pop	pop value from stack	*Checkcast T*	ensure that value conforms to type *T*
Dup	duplicate top value on stack	*Instanceof T*	check for assignment compatibility
Swap	swap top elements on stack	*Invoke M n*	invoke method *M* with *n* parameters
BinOp bop	apply binary operator *bop*	*Return*	return from method
New C	create object of class *C*	*Goto i*	relative jump
NewArray T	create array with element type *T*	*IfFalse i*	branch if top of stack is *Bool False*
ALoad	fetch array cell	*ThrowExc*	raise top of stack as exception
AStore	set array cell	*MEnter*	acquire lock on monitor
ALength	get length of array	*MExit*	release lock on monitor

Table 3 Instructions of the JinjaThreads virtual machine

$$
\begin{array}{l}
\textit{exec-instr MEnter } P\ t\ h\ stk\ loc\ C\ M\ frs = \\
(\textit{let } v \cdot stk' = stk; \textit{Addr } a = v \\
\ \textit{in if } v = \textit{Null then } \{\ ((\!|\!|), \lfloor \textit{addr-of-sys-xcpt NullPointer} \rfloor, h, (stk, loc, C, M, pc) \cdot frs)\ \} \\
\quad \textit{else } \{\ ((\!|\textit{Lock}{\to}a|\!|), \textit{None}, h, (stk', loc, C, M, pc+1) \cdot frs)\ \})
\end{array}
$$

$$
\begin{array}{l}
\textit{exec-instr MExit } P\ t\ h\ stk\ loc\ C\ M\ frs = \\
(\textit{let } v \cdot stk' = stk;\ \textit{Addr } a = v \\
\ \textit{in if } v = \textit{Null then } \{\ ((\!|\!|), \lfloor \textit{addr-of-sys-xcpt NullPointer} \rfloor, h, (stk, loc, C, M, pc) \cdot frs)\ \} \\
\quad \textit{else } \{\ ((\!|\textit{Unlock}{\to}a|\!|), \textit{None}, h, (stk', loc, C, M, pc+1) \cdot frs), \\
\qquad ((\!|\textit{UnlockFail}{\to}a|\!|), \lfloor \textit{addr-of-sys-xcpt IllegalMonitorState} \rfloor, h, (stk, loc, C, M, pc) \cdot frs)\ \})
\end{array}
$$

$$
\begin{array}{l}
\textit{exec-instr } (\textit{Invoke } M'\ n)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs = \\
(\textit{let } ps = \textit{rev } (\textit{take } n\ stk);\ r = stk_{[n]}; \textit{Addr } a = r;\ \lfloor T \rfloor = \textit{typeof-addr } h\ a \\
\ \textit{in if } r = \textit{Null then } \{\ ((\!|\!|), \lfloor \textit{addr-of-sys-xcpt NullPointer} \rfloor, h, (stk, loc, C, M, pc) \cdot frs)\ \} \\
\quad \textit{else let } (D, Ts, T_r, m) = \iota(D, Ts, T_r, m).\ P \vdash \textit{class-of}'\ T\ \textit{sees } M' : Ts \to T_R = m\ \textit{in } D \\
\qquad \textit{in case } m \textit{ of } \textit{Native} \Rightarrow \{\ (\textit{native-TA2jvm } P\ t\ a, \textit{native-Ret2jvm } n\ h'\ stk\ loc\ C\ M\ pc\ frs\ vx)\ | \\
\qquad\qquad P, t \vdash \langle a.M'(ps), h\rangle -ta\rightarrow_{\text{native}} \langle vx, h'\rangle\ \} \\
\qquad\ |\ \lfloor (msl, mxl, ins, xt) \rfloor \Rightarrow \textit{let } fr' = ([], r \cdot ps\ @\ \textit{replicate } mxl\ \textit{dummy-val}, D, M', 0) \\
\qquad\qquad \textit{in } \{\ ((\!|\!|), \textit{None}, h, fr' \cdot (stk, loc, C, M, pc) \cdot frs)\ \})
\end{array}
$$

$$
\begin{array}{l}
\textit{exec-instr } (\textit{CAS } F\ D)\ P\ t\ h\ stk\ loc\ C\ M\ pc\ frs = \\
(\textit{let } v'' \cdot v' \cdot v \cdot stk' = stk;\ \textit{Addr } a = v \\
\ \textit{in if } v = \textit{Null then } \{\ ((\!|\!|), \lfloor \textit{addr-of-sys-xcpt NullPointer} \rfloor, h, (stk, loc, C, M, pc) \cdot frs)\ \} \\
\quad \textit{else } \{\ ((\!|\!|), \textit{None}, h', (\textit{Bool True} \cdot stk', loc, C, M, pc+1) \cdot frs)\ | \\
\qquad \textit{read } h\ a\ (\textit{Field } D\ F)\ v' \wedge \textit{write } h\ a\ (\textit{Field } D\ F)\ v''\ h'\ \} \cup \\
\qquad \{\ ((\!|\!|), \textit{None}, h, (\textit{Bool False} \cdot stk', loc, C, M, pc+1) \cdot frs)\ |\ \textit{read } h\ a\ (\textit{Field } D\ F)\ v''' \wedge v' \neq v'''\ \}
\end{array}
$$

Fig. 17 Semantics of the instructions for monitors, method calls, and the compare-and-set operation

Each method executes in its own call frame (type *frame*). A call frame (stk, loc, C, M, pc) contains the operand stack *stk*, an array *loc* of registers for the *this* pointer, the parameters, and local variables, the class name *C*, the method name *M*, and the program counter *pc*. Although *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.

The JinjaThreads VM is defined in a functional style, like the Jinja VM. The function *exec-instr* :: *instr* $\Rightarrow$ *jvm-prog* $\Rightarrow$ $'$*addr* $\Rightarrow$ $'$*heap* $\Rightarrow$ *opstack* $\Rightarrow$ *registers* $\Rightarrow$ *cname* $\Rightarrow$ *mname* $\Rightarrow$ *pc* $\Rightarrow$ $'$*addr frame list* $\Rightarrow$ (($'$*addr*, $'$*heap*) *jvm-TA* $\times$ ($'$*addr*, $'$*heap*) *jvm-state*) *set* defines the semantics of a single instruction, where ($'$*addr*, $'$*heap*) *jvm-TA* is an abbreviation for ($'$*addr*, $'$*addr*, $'$*addr option* $\times$ $'$*addr frame list*, $'$*heap*, $'$*addr*, $'o$) *thread-action*. Given the instruction, the program, the thread ID, the heap, and the curried non-empty call stack, it produces a non-empty set of thread actions and successor states by pattern-matching on the instruction.

Figure 17 presents the semantics of the instructions *MEnter*, *MExit*, *Invoke*, and *CAS* for monitors, method calls, and the compare-and-set operation; Both instructions for mon-

$$
\begin{array}{lll}
\mathit{exec}\ P\ (\mathit{xcp}, & h, [])\ & = \emptyset \\
\mathit{exec}\ P\ (\mathit{None}, h, (\mathit{stk}, \mathit{loc}, C, M, \mathit{pc}) \cdot \mathit{frs}) & & = \mathit{exec\text{-}instr}\ (\mathit{instrs\text{-}of}\ P\ C\ M)_{[\mathit{pc}]}\ P\ h\ \mathit{stk}\ \mathit{loc}\ C\ M\ \mathit{pc}\ \mathit{frs} \\
\mathit{exec}\ P\ (\lfloor a \rfloor, & h, \mathit{fr} \cdot \mathit{frs}) & = \{\ \mathit{xcpt\text{-}step}\ P\ a\ h\ \mathit{fr}\ \mathit{frs}\ \}
\end{array}
$$

Fig. 18 Combining normal execution with exception handling in the VM

itors *MEnter* and *MExit* raise a *NullPointer* exception if the value v on top of the stack is *Null*. Here, the function *addr-of-sys-xcpt* returns the address of pre-allocated system exceptions. Otherwise, *MEnter* acquires the lock with the thread action $\langle\!|\mathit{Lock}{\rightarrow}a|\!\rangle$ where a is the address that v denotes. The same thread action is used by the source code semantics (RLOCK). Like RUNLOCK, *MExit* unlocks the monitor a with the thread action $\langle\!|\mathit{Unlock}{\rightarrow}a|\!\rangle$. Unlike *sync* $(\_)$ $\_$ expressions in J, 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.

Calls to native methods reuse the semantics from Section 2.3.3 and non-native calls push the new call frame fr' on top of the call stack, where the registers contain the receiver object r, the parameters ps (which are in reverse order on the stack), and *replicate mxl dummy-val* fills the remaining registers with the dummy value *dummy-val*.

Like in source code, compare-and-set *CAS* uses the abstract heap operations *read* and *write* to access and update the specified field. When several values can be read from memory, this instruction has several successor states, for comparison success and for comparision failure. The Java memory model eliminates this non-determinism at a higher level in the semantics stack, i.e., compare-and-set is always deterministic for volatile fields.

The function $\mathit{exec} :: {'\mathit{addr}}\ \mathit{jvm\text{-}prog} \Rightarrow ({'\mathit{addr}}, {'\mathit{heap}})\ \mathit{jvm\text{-}state} \Rightarrow (({'\mathit{addr}}, {'\mathit{heap}})\ \mathit{jvm\text{-}TA} \times ({'\mathit{addr}}, {'\mathit{heap}})\ \mathit{jvm\text{-}state})\ \mathit{set}$ incorporates exception handling in the semantics (Figure 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* (not shown) 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.

This formalisation yields an aggressive VM: As can be seen in Figure 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 (Section 3) shows that these cases cannot occur for well-formed programs.

JinjaThreads also formalises 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 exec_d adds these checks on top of the aggressive VM *exec*.

datatype ${'a}\ \mathit{type\text{-}error} = \mathit{TypeError} \mid \mathit{Normal}\ {'a}$

$\mathit{exec}_d\ P\ s = (\mathit{if}\ \mathit{check}\ P\ s\ \mathit{then}\ \mathit{Normal}\ (\mathit{exec}\ P\ s)\ \mathit{else}\ \mathit{TypeError})$

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, *MEnter* requires a non-empty stack with a reference value at the top:

$\mathit{check\text{-}instr}\ \mathit{MEnter}\ P\ t\ h\ \mathit{stk}\ \mathit{loc}\ C\ M\ \mathit{pc}\ \mathit{frs} = (0 < |\mathit{stk}| \wedge \mathit{is\text{-}Ref}\ (\mathit{hd}\ \mathit{stk}))$

where *is-Ref v* predicates that *v* is *Null* or some *Addr a*. The checks for *MExit* are the same. For *CAS F D*, at least three values must be on the stack with the third being a reference or *Null*; if it is a reference, the associated class (*Object* in case of an array) must see the field *F* declared in class *D* as volatile and the other two values' types must be subtypes of the *F*'s declared type.

As there are two VMs, bytecode instantiates the multithreading semantics twice, too. The type variable instantiations 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* for the aggressive and defensive VM, respectively. Like for *J-red*, they require some glue to adjust the types.

$$\textit{jvm-final}\ (xcp,frs) = (frs = [])$$

$$\textit{jvm-exec}\ P\ t\ ((xcp,frs),h)\ ta\ ((xcp',frs'),h') = (ta,(xcp',h',frs')) \in \textsf{exec}\ P\ t\ (xcp,h,frs)$$

$$\textit{jvm-execd}\ P\ t\ ((xcp,frs),h)\ ta\ ((xcp',frs'),h') =$$
$$(\exists S.\ \textsf{exec}_d\ P\ t\ (xcp,h,frs) = \textsf{Normal}\ S \wedge (ta,(xcp',h',frs')) \in S)$$

Note that *jvm-execd* turns the defensive VM into a strict VM as it does not raise *TypeError*s any more, but gets stuck. Nevertheless, I keep referring to it as the defensive VM. Dropping *TypeError* avoids duplications as the same representation for thread-local states can be used for both the aggressive and the defensive VM. For example, I can use the same state invariants for both VMs in the type safety proof (Section 3.5).

To distinguish between the two locale instances for bytecode, I use the name prefixes *jvm.* and *jvmd.* for the aggressive and defensive VM, respectively. Also, $P \vdash s\ -t{:}ta{\rightarrow}_{\text{jvm}}\ s'$ denotes *jvm.redT P s* (t,ta) s' and $P \vdash s\ -t{:}ta{\rightarrow}_{\text{jvmd}}\ s'$ denotes *jvmd.redT P s* (t,ta) s'. Also, $P \vdash s\ -ttas{\rightarrow}^*_{\text{jvm}}\ s'$ and $P \vdash s\ -ttas{\rightarrow}^*_{\text{jvmd}}\ s'$ are the reflexive and transitive closures of *jvm.redT P* and *jvmd.redT P*.

Lemma 3 *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 (Figure 18). *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. □

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

$$(\textit{None}, [([], \textit{Null} \cdot \textit{vs}\ @\ \textit{replicate mxl dummy-val}, D, M, 0)])$$

where $P \vdash C\ \textit{sees}\ M{:}\_{\rightarrow}\_ = \lfloor(mxs,mxl,ins,xt)\rfloor\ \textit{in}\ D$. That is, 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.

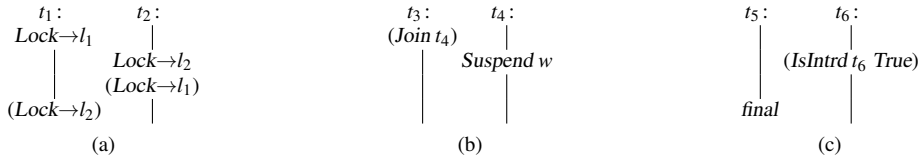


Fig. 19 Three schedules with two threads each that lead to deadlock

3 Type safety

In this section, I prove type safety for both source code (Section 3.4) and bytecode (Section 3.5). Type safety expresses that when a program starts in a good state, then the program will not run into type problems such as non-existing methods and fields or operands of unexpected type, and if the program terminates, the result is of the expected type. To that end, I formalise the notion of good state and prove that the semantics preserves the good-state property and no type error occurs in good states. As multithreaded programs can end up in deadlock, the type safety statements also require a formalisation of deadlocks (Section 3.3).

3.1 High-level overview

For languages like *J*, type safety is usually expressed using two syntactic properties: progress and preservation [87]. Progress means that every well-formed and well-typed expression can be reduced unless it has already been fully evaluated. So the semantics is not missing any rules. Full evaluation is determined by a syntactic predicate *final*. Preservation requires that reductions preserve well-typedness and well-formedness. In Section 3.2, I develop the machinery to transfer preservation proofs from single threads to the interleaving semantics. Deadlocks, however, can break the progress property.

I therefore formalise the concept of deadlock and prove progress up to deadlock for the interleaving semantics. Although progress typically identifies allowed stuck states syntactically, I formalise deadlock (and thus the allowed stuck states) semantically. The reason is that a deadlock typically involves several threads. A syntactic characterisation would have to examine all of them together and would therefore 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.

Intuitively, a system is said to be in deadlock iff all threads are waiting for something that will never occur. In the interleaving semantics, there are four things 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 basic thread actions, I can formally define deadlock solely in terms of the reductions of a thread—independent of the language.

Figure 19 shows three schedules that lead to different kinds of deadlocks. On the left, in Figure 19a, 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 transition with $Lock{\to}l_1$ is postponed. However, t_1 next requests the lock l_2, which t_2 is holding. Hence, both threads are in deadlock. In the centre (Figure 19b), t_3 waits for t_4's termination, but t_4 suspends itself to the wait set w and does not wake up spuriously. 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 19c) shows a similar example with interruption: t_6 waits for being interrupted, but t_5 terminates without doing so. Thus, t_6 is deadlocked.

I: $(\!|Unlock{\to}l_1, Lock{\to}l_2|\!)$
II: $(\!|Unlock{\to}l_2, Lock{\to}l_1|\!)$, $(\!|Lock{\to}l_3|\!)$, $(\!|Lock{\to}l_4|\!)$
III: $(\!|Lock{\to}l_6|\!)$
IV: $(\!|\,|\!)$
V: $(\!|Lock{\to}l_2, Lock{\to}l_3|\!)$
VI: $(\!|Lock{\to}l_3|\!)$

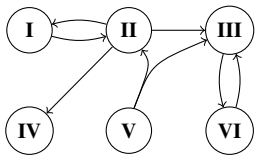


Fig. 20 Example with deadlocked threads

In the interleaving semantics, things are a bit more tricky than in these 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 (JOIN) or for the executing thread to be interrupted (JOININTR). It therefore makes sense to formalise and study deadlock in the full generality of the interleaving semantics.

Figure 20 shows an example of different deadlocks with locks. The example is abstract because such a situation cannot arise in Java as, e.g., threads cannot acquire several locks in one step or non-deterministically wait for different locks. There are six threads which can execute with the thread actions shown on the left-hand side (every thread action is written as a list of basic thread actions enclosed in $(\!|$ and $|\!)$). If there are multiple thread actions, then there is one transition 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. Threads III and VI are waiting for each other without other transition 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 transition options. Waiting on lock l_3 will be in vain because III is deadlocked. However, IV is not waiting for any resource, so II may still hope to obtain the lock l_4 at some later 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 as it needs the locks l_2 and l_3 to proceed. Since III is already in deadlock, so is V. Clearly, IV is not deadlocked as it is not in a wait set. 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 proceed (i.e., IV) is waiting for some other thread waking it up.

To get a hold on this, I first formalise what a thread non-deterministically waits for in a deadlock, by looking at the thread actions of the reductions in the single-threaded semantics. I then define the set of threads in deadlock coinductively. Coinductivity naturally captures that a thread is not deadlocked iff one can deduce in finitely many steps that it is not. That is, finitely many steps (of other threads) suffice to allow the thread under consideration to continue. A multithreaded state is in deadlock iff all its non-final threads are deadlocked. So deadlock subsumes states in which all threads have terminated, as I do not require that there be a non-final thread. I could have formalised deadlock also as a state property instead of individual threads in deadlock, but the thread-wise definition makes the type safety statements more intuitive (Section 3.4).

There are two kinds of reasons for a thread being deadlocked:

– It waits for a condition that is controlled by some other thread which itself is in deadlocked. Acquiring a lock or joining on a thread fall into this category.

– It waits for an event that any thread could possibly perform, but all potential threads are deadlocked. Waiting for an interrupt and for being removed from a wait set are examples of this kind.

While there are still running threads in a state, only deadlock of the first kind is possible. When the last running thread has terminated or has ended up in deadlock, also the second kind can apply. So this case assumes that the set of threads is closed with respect to the outside, i.e., one cannot add a spinning thread, which would "undeadlock" all waiting threads.

I prove two sanity theorems for deadlocks. First, threads and states in deadlock cannot proceed. Second, threads in deadlock remain in deadlock even if other threads keep running. The latter requires that the single-threaded semantics is well-behaved, i.e., the synchronisation options of one thread are not affected by other threads changing the heap. Well-behavedness rules out in particular blocking synchronisation primitives other than those provided by the interleaving semantics. Non-blocking primitives like compare-and-set are fine as failure is always an option when there is no success.

The main theorem about deadlock is progress up to deadlock (Theorem 10). It expresses that the multithreaded semantics can make a step in a non-deadlocked state. For this theorem, I identify and formalise sufficient conditions on the single-threaded semantics, which source code and bytecode satisfy. Hence, source code and bytecode can apply the progress theorem (Sections 3.4 and 3.5). The conditions are chosen such that the sequential Jinja invariants and their preservation proof can be reused. Technically, these conditions are collected in Isabelle locales. So Isabelle specialises the theorems automatic to the two semantics after the conditions have been discharged.

The type safety proofs for source code and bytecode themselves require assumptions about the abstract heap model. Progress needs that read and write relations are total for locations (fields and array cells indexed by addresses) that exist according to the dynamic type of addresses. Preservation relies on the following:

– The initial heap is conformant.
– Only type-correct values can be read from an existing location in a conformant heap.
– Allocation and writing type-correct values to existing locations preserves heap conformance.

The restriction to existing locations is crucial for the type-safety proof of the Java memory model [50, §5.2], as otherwise the type of a location and the set of type-correct values is not (yet) determined.

3.2 Lifting thread-local invariants

To separate constraints due to multithreading from the language-specific constraints, I first define some machinery to transfer such thread-local constraints and their preservation lemmas to the multithreaded semantics. Suppose that the predicate $Q :: 't \Rightarrow 'x \Rightarrow 'h \Rightarrow bool$ denotes a constraint on the thread local states which may depend on the shared state. The operator $\uparrow\_\uparrow$ lifts Q to a predicate of type $('l, 't, 'x)\ tp \Rightarrow 'h \Rightarrow bool$ on the thread pool and shared state such that $\uparrow Q \uparrow$ imposes Q on all threads in the thread pool.

$$\uparrow Q \uparrow\ tp\ h = (\forall t.\ case\ tp\ t\ of\ None \Rightarrow True \mid \lfloor (x, ln) \rfloor \Rightarrow Q\ t\ x\ h)$$

To transfer preservation lemma for Q to the multithreaded state, I define the locale *lifting-wf* (see Figure 21). It fixes the constraint Q and assumes that

locale *lifting-wf* = *multithreaded* + fixes $Q :: 't \Rightarrow 'x \Rightarrow 'h \Rightarrow bool$
  assumes $[\![t \vdash (x,h) -ta\rightarrow (x',h'); Q\ t\ x\ h]\!] \Longrightarrow Q\ t\ x'\ h'$
  and $[\![t \vdash (x,h) -ta\rightarrow (x',h'); Q\ t\ x\ h; Spawn\ t''\ x''\ h' \in set\ \langle ta\rangle_t]\!] \Longrightarrow Q\ t''\ x''\ h'$
  and $[\![t \vdash (x,h) -ta\rightarrow (x',h'); Q\ t\ x\ h; Q\ t''\ x''\ h]\!] \Longrightarrow Q\ t''\ x''\ h'$

locale *lifting-inv* = *multithreaded* + fixes $Q :: 'i \Rightarrow 't \Rightarrow 'x \Rightarrow 'h \Rightarrow bool$
  assumes $[\![t \vdash (x,h) -ta\rightarrow (x',h'); Q\ i\ t\ x\ h]\!] \Longrightarrow Q\ i\ t\ x'\ h'$
  and $[\![t \vdash (x,h) -ta\rightarrow (x',h'); Q\ i\ t\ x\ h; Spawn\ t''\ x''\ h' \in set\ \langle ta\rangle_t]\!] \Longrightarrow \exists i''.\ Q\ i''\ t''\ x''\ h'$
  and $[\![t \vdash (x,h) -ta\rightarrow (x',h'); Q\ i\ t\ x\ h; Q\ i''\ t''\ x''\ h]\!] \Longrightarrow Q\ i''\ t''\ x''\ h'$

Fig. 21 Definition of locales *lifting-wf* and *lifting-inv*

1. single-thread steps preserve Q,
2. Q holds for new threads at the time of creation, and
3. Q is preserved even if another thread, which also satisfies Q, changes the heap.

Under these assumptions, the interleaving semantics preserves $\uparrow Q\uparrow$, too.

Lemma 4 (Preservation for $\uparrow\_\uparrow$) *Let $\uparrow Q\uparrow\ (tp\ s)\ (shr\ s)$. If $s -t{:}ta\rightarrow s'$ or $s -ttas\rightarrow^* s'$, then $\uparrow Q\uparrow\ (tp\ s')\ (shr\ s')$, too.*

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 $Q :: 'i \Rightarrow 't \Rightarrow 'x \Rightarrow 'h \Rightarrow bool$ also include the invariant data. The operator $\Uparrow Q\Uparrow$ lifts Q to thread pools similar to $\uparrow\_\uparrow$.

$$\Uparrow Q\Uparrow\ I\ tp\ h = (\forall t.\ case\ tp\ t\ of\ None \Rightarrow True \mid \lfloor (x,ln)\rfloor \Rightarrow \exists i.\ I\ t = \lfloor i\rfloor \wedge Q\ i\ t\ x\ h)$$

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

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

$$\begin{aligned} I(&[] \rightsquigarrow Q) = I \\ I(&Spawn\ t\ x\ h \cdot nts \rightsquigarrow Q) = (I(t \mapsto \varepsilon i.\ Q\ i\ t\ x\ h))(nts \rightsquigarrow Q) \\ I(&ThreadEx\ t\ b \cdot nts \rightsquigarrow Q) = I(nts \rightsquigarrow Q) \end{aligned} \tag{1}$$

For labels *ttas* of the reflexive and transitive closure $\_ -\_\rightarrow^* \_$, let $I(ttas\ [\rightsquigarrow]\ Q)$ denote the extension $I(concat\ (map\ (\lambda(t,ta).\ \langle ta\rangle_t)\ ttas) \rightsquigarrow Q)$, where the term $map\ (\lambda(t,ta).\ \langle ta\rangle_t)\ ttas$ extracts all thread creation BTAs from *ttas* and *concat* combines them in one list. The extension preserves well-formedness of maps to invariant data.

Lemma 5 *Suppose $thr\ s \vdash_i I$. If $s -t{:}ta\rightarrow s'$, then $tp\ s' \vdash_i I(\langle ta\rangle_t \rightsquigarrow Q)$. If $s -ttas\rightarrow^* s'$, then $tp\ s' \vdash_i I(ttas\ [\rightsquigarrow]\ Q)$.*

Equation 1 shows why it is necessary to remember the shared heap in *Spawn* actions. $\_(\_ \rightsquigarrow Q)$ 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 store it in the thread actions.

Similarly to *lifting-wf*, the locale *lifting-inv* collects the assumptions for lifting the preservation theorems (Figure 21). The main difference is that *lifting-inv* existentially quantifies over the invariant data for spawned threads. In fact, *lifting-wf* is just the special case of *lifting-inv* with the constraint instantiated to $\lambda\_.\ Q$.

Analogous to Lemma 4, the next lemma shows that these assumptions are sufficient for the interleaving semantics preserving $\Uparrow Q\Uparrow$.

$$\text{MWLOCK:}\ \frac{\textit{has-lock}\ (\textit{locks}\ s\ l)\ t' \qquad t' \neq t \qquad t' \in T}{\textit{must-wait}\ s\ t\ (\textit{Inl}\ l)\ T}$$

$$\text{MWJOIN:}\ \frac{\textit{not-final-thread}\ s\ t' \qquad t' \in T}{\textit{must-wait}\ s\ t\ (\textit{Inr}\ (\textit{Inl}\ t'))\ T} \qquad \text{MWINTR:}\ \frac{\textit{all-final-except}\ s\ T \qquad t' \notin \textit{intrs}\ s}{\textit{must-wait}\ s\ t\ (\textit{Inr}\ (\textit{Inr}\ t'))\ T}$$

Fig. 22 Thread t must wait for a resource indefinitely

Lemma 6 (Preservation for $\Uparrow\_\Uparrow$) *Suppose that* $\Uparrow Q \Uparrow\ I\ (\textit{tp}\ s)\ (\textit{shr}\ s)$.

(i) If $s -t{:}ta\rightarrow s'$, *then* $\Uparrow Q \Uparrow\ I(\langle ta\rangle_{\mathrm{t}} \rightsquigarrow Q)\ (\textit{tp}\ s')\ (\textit{shr}\ s')$.
(ii) If $s -ttas\rightarrow^* s'$, *then* $\Uparrow Q \Uparrow\ I(ttas\ [\rightsquigarrow]\ Q)\ (\textit{tp}\ s')\ (\textit{shr}\ s')$.

3.3 Deadlock

As explained in Section 3.1, I first define what a thread may wait for in a deadlock. The function *waits* $ta :: ('l + 't + 't)\ \textit{set}$ extracts all the resources for which a thread may be waiting, i.e., the locks it acquires, the threads it joins on, and the threads which must be interrupted. Formally,

$$\textit{waits}\ ta = \{\, l \mid \textit{Lock} \in \textit{set}\ (\langle ta\rangle_{\mathrm{lf}}\ l)\,\} \uplus \{\, t \mid \textit{Join}\ t \in \textit{set}\ \langle ta\rangle_{\mathrm{c}}\,\} \uplus \textit{intr-waits}\ \langle ta\rangle_{\mathrm{i}}$$

where $\uplus$ is disjoint union and *intr-waits* $\langle ta\rangle_{\mathrm{i}}$ is the set of thread IDs t for which $\langle ta\rangle_{\mathrm{i}}$ contains an element *IsIntrd* t *True* which is not preceeded by *Intr* t or *ClearIntr* t. The last constraint ignores interrupt checks whose result does not depend on the initial interrupt state, but is determined by the preceeding interrupt actions. For example,

$$\textit{intr-waits}\ (\!|\textit{IsIntrd}\ t\ \textit{True}, \textit{ClearIntr}\ t|\!) = \{t\} \qquad \textit{intr-waits}\ (\!|\textit{Intr}\ t, \textit{IsIntrd}\ t\ \textit{True}|\!) = \emptyset$$

The thread action on the left tests whether t has been interrupted and, if so, clears the interrupt status. Hence, it waits for t being interrupted. On the right, the test *IsIntrd* t *True* holds vacuously because *Intr* t sets the interrupt flag right before the test. Such tests clearly make no sense and eliminating them removes some assumptions from the theorems.

Note that *waits* ignores the 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 always succeeds or raises an exception, so it cannot deadlock.
wait sets A thread in a wait set cannot do anything to be removed. *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 deadlock formalisation is therefore tailored to Java more than the interleaving semantics.

Next, the predicate *must-wait* $s\ t\ w\ T$ determines that in state s, the thread t will wait indefinitely for resource $w :: 'l + 't + 't$ under the assumption that all threads in T are already deadlocked. Figure 22 shows the formal definition. The thread t must wait forever for the lock l (case $w = \textit{Inl}\ l$) if l is held by another thread t' which is deadlocked (MWLOCK). The

$$\textsc{Dactive:}\quad \frac{tp\ s\ t = \lfloor (x, \lambda\_.\ 0) \rfloor \qquad wset\ s\ t = None \qquad t \vdash (x, shr\ s)\wr \qquad \forall W.\ t \vdash (x, shr\ s)\ W\wr \longrightarrow \exists w \in W.\ must\text{-}wait\ s\ t\ w\ (deadlocked\ s \cup final\text{-}threads\ s)}{t \in deadlocked\ s}$$

$$\textsc{Dacquire:}\quad \frac{\neg\ waiting\ (wset\ s\ t) \qquad ln\ l > 0 \qquad tp\ s\ t = \lfloor (x, ln) \rfloor \qquad must\text{-}wait\ s\ t\ (Inl\ l)\ (deadlocked\ s \cup final\text{-}threads\ s)}{t \in deadlocked\ s}$$

$$\textsc{Dwait:}\quad \frac{tp\ s\ t = \lfloor (x, ln) \rfloor \qquad waiting\ (wset\ s\ t) \qquad all\text{-}final\text{-}except\ s\ (deadlocked\ s)}{t \in deadlocked\ s}$$

Fig. 23 Coinductive definition of the set of threads in deadlock

join on another thread t' fails forever (case $w = Inr\ (Inl\ t')$) if the thread t' is not final and already deadlocked (MWJOIN). The predicate *not-final-thread s t* denotes that t exists, but is not final, i.e., $t \in dom\ (tp\ s)$ and $t \notin final\text{-}threads\ s$. Note that *not-final-thread s t* negates the precondition of *Join* (Figure 10) 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, which the predicate *all-final-except* expresses:

$$all\text{-}final\text{-}except\ s\ T = \{\, t \mid not\text{-}final\text{-}thread\ s\ t \,\} \subseteq T$$

Finally, I define two abstractions: $t \vdash (x,h)\wr$ denotes that t can reduce in the local state x and heap h with a thread action ta that is not contradictory, i.e., there is a multithreaded state s such that $ok\text{-}ta\ s\ t\ ta$. For example, $(\!|Lock{\to}l, UnlockFail{\to}l|\!)$ and $(\!|Intr\ t, IsIntrd\ t\ False|\!)$ are contradictory. The predicate $t \vdash (x,h)\ W\wr$ 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.

With these preparations, I can now formalise when a thread is in deadlock. Figure 23 coinductively defines the set *deadlocked* of threads in deadlock. There are three ways a thread t can be deadlocked:

DACTIVE: t is ready to execute, say $tp\ s\ t = \lfloor (x, \lambda\_.\ 0) \rfloor$ and $wset\ s\ t = None$, and it can reduce, but no matter how it might reduce, it must wait for some deadlocked or final thread.

DACQUIRE: t has temporarily released some lock l and this lock is held by a deadlocked or final thread.

DWAIT: t is in a wait set and all non-deadlocked threads have already terminated.[7]

The first case is the default: $t \vdash (x, shr\ s)\wr$ ensures that the thread is not just stuck, i.e., universal quantification on W does not hold vacuously. Quantifying over all W with $t \vdash (x, shr\ s)\ W\wr$ allows a thread to non-deterministically wait for different "resources". The second case accounts for acquisition of temporarily released locks. Since it is the interleaving semantics that performs the acquisition (ACQ) instead of the single threads, a separate case is needed. The last case assumes that the set of threads is closed with respect to the outside, i.e., one cannot add a spinning thread, which would "undeadlock" again all waiting threads.

This coinductive definition is acceptable because *deadlocked* occurs in the rule premises only in monotone contexts:

Lemma 7 (Monotonicity of *all-final-except* and *must-wait*) *Let $T \subseteq T'$.*
If must-wait s t w T, then must-wait s t w T′. If all-final-except s T, then all-final-except s T′.

[7] Recall that the interleaving semantics does not force threads to wake up spuriously (cf. Section 2.3.3). Otherwise, a waiting thread would never be deadlocked because it could always be woken up spuriously.

locale *preserve-deadlocked* = *multithreaded* + fixes *wf-states* :: $('l,'t,'x,'h,'w)$ *state set*
  assumes $[\![s \in \mathit{wf\text{-}states};\ s -t{:}ta\rightarrow s']\!] \Longrightarrow s' \in \mathit{wf\text{-}states}$
  and $[\![s \in \mathit{wf\text{-}states};\ s -t'{:}ta\rightarrow s';\ \mathit{tp}\ s\ t = \lfloor(x, \lambda\_.\ 0)\rfloor;\ t \vdash (x, \mathit{shr}\ s)\wr]\!] \Longrightarrow t \vdash (x, \mathit{shr}\ s')\wr$
  and $[\![s \in \mathit{wf\text{-}states};\ s -t'{:}ta\rightarrow s';\ \mathit{tp}\ s\ t = \lfloor(x, \lambda\_.\ 0)\rfloor;\ t \vdash (x, \mathit{shr}\ s')\ W'\wr]\!] \Longrightarrow \exists W \subseteq W'.\ t \vdash (x, \mathit{shr}\ s)\ W\wr$

Fig. 24 The locale *preserve-deadlocked* collects the requirements for preservation of deadlock

A multithreaded state is in deadlock (*deadlock s*) iff all non-final threads are deadlocked. For simplicity, I do not require that there is at least a thread which is not final. Hence, every *mfinal* state is also in *deadlock*.

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

Lemma 8 *Every thread in deadlock is stuck, i.e., if* $s -t{:}ta\rightarrow s'$*, then* $t \notin \mathit{deadlocked}\ s$*. States in deadlock are stuck, i.e., if* $s -t{:}ta\rightarrow s'$*, then* $\neg\ \mathit{deadlock}\ s$*.*

Proof By case analysis on $s -t{:}ta\rightarrow s'$ and $t \in \mathit{deadlocked}\ s$. □

Execution may continue even if some threads are deadlocked, but threads in deadlock should remain deadlocked; otherwise, the deadlock definition would be flawed. 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 undeadlock it, either.

The locale *preserve-deadlocked* collects these assumptions (Figure 24). It fixes a set *wf-states* of well-formed states only for which the preservation requirements have to hold. The first assumption expresses that the interleaving semantics preserves *wf-states*. The other two assumptions express exactly the requirements from above. Note the covariance in the set W in the last assumption: Since $t \vdash \_\ W\wr$ expresses that one of t's reduction requires *all* resources in W, changes in the heap may only increase W. Under these assumptions, the set of deadlocked threads can only become larger:

Lemma 9 (Preservation of deadlock) *Let* $s \in \mathit{wf\text{-}states}$*. If* $s -t{:}ta\rightarrow s'$ *or* $s -ttas\rightarrow^* s'$*, then* $\mathit{deadlocked}\ s \subseteq \mathit{deadlocked}\ s'$*.*

Proof If $s -t{:}ta\rightarrow s'$, suppose that $t' \in \mathit{deadlocked}\ s$. I show $t' \in \mathit{deadlocked}\ s'$ by coinduction with *deadlocked s* as coinduction invariant. In the coinductive step, case analysis on $t' \in \mathit{deadlocked}\ s$ yields the interesting cases DACTIVE and DACQUIRE; the case DWAIT contradicts the assumption $s -t{:}ta\rightarrow s'$. In both these cases, the reduction from s to s' preserves *must-wait*. For DACTIVE, the assumptions of *preserve-deadlocked* relate $t' \vdash (x, \_)\wr$ and $t' \vdash (x, \_)\ \_\wr$ between *shr s* and *shr s'*, respectively.

The case $s -ttas\rightarrow^* s'$ follows from $s -t{:}ta\rightarrow s'$ by induction; the locale's first assumption reestablishes the induction invariant $s \in \mathit{wf\text{-}states}$ in the inductive step. □

Both source code and bytecode satisfy the assumptions of *preserve-deadlocked* where the well-formed states are the same as for the type safety proofs in the following sections.

I now prove the progress theorem for the interleaving semantics. The locale *progress* collects the necessary assumptions about the single-threaded semantics (Figure 25).

Theorem 10 (Progress up to deadlock) *Given the assumptions of locale* progress*, let* $s \in \mathit{wf\text{-}states}$*. If* $\neg\ \mathit{deadlock}\ s$*, then there are t, ta, and* s' *such that* $s -t{:}ta\rightarrow s'$*.*

locale *progress* = *multithreaded* + fixes *wf-states* :: $('l,'t,'x,'h,'w)$ *state set*
assumes *wf-stateD*: $s \in$ *wf-states* $\Longrightarrow$ *ok-locks-tp s* $\wedge$ *ok-wset-final s*
and *progress*: $[\![s \in$ *wf-states*; *tp s t* $= \lfloor(x, \lambda\_.\ 0)\rfloor$; $\neg$ *final x*$]\!] \Longrightarrow \exists ta\ x'\ h'.\ t \vdash (x, \mathit{shr}\ s) -ta\rightarrow (x', h')$
and *wf-ta*: $[\![s \in$ *wf-states*; *tp s t* $= \lfloor(x, \lambda\_.\ 0)\rfloor$; $t \vdash (x, \mathit{shr}\ s) -ta\rightarrow (x', h')]\!] \Longrightarrow \exists s'.$ *ok-ta s' t ta*
and *wf-red*:
$[\![s \in$ *wf-states*; *tp s t* $= \lfloor(x, \lambda\_.\ 0)\rfloor$; $\neg$ *waiting* (*wset s t*); $t \vdash (x, \mathit{shr}\ s) -ta\rightarrow (x', h')]\!]$
$\Longrightarrow \exists ta'\ x''\ h''.\ t \vdash (x, \mathit{shr}\ s) -ta'\rightarrow (x'', h'') \wedge$ (*ok-ta s t ta'* $\vee$ *ok-ta' s t ta'* $\wedge$ *waits ta'* $\subseteq$ *waits ta*)
and *Suspend-not-final*:
$[\![s \in$ *wf-states*; *tp s t* $= \lfloor(x, \lambda\_.\ 0)\rfloor$; $\neg$ *waiting* (*wset s t*);
$t \vdash (x, \mathit{shr}\ s) -ta\rightarrow (x', h')$; *Suspend w* $\in$ *set* $\langle ta\rangle_{\mathrm{w}}]\!] \Longrightarrow \neg$ *final x'*
and *Wakeup-waits*:
$[\![s \in$ *wf-states*; *tp s t* $= \lfloor(x, \lambda\_.\ 0)\rfloor$; $t \vdash (x, \mathit{shr}\ s) -ta\rightarrow (x', h')$;
Notified $\in$ *set* $\langle ta\rangle_{\mathrm{w}} \vee$ *WokenUp* $\in$ *set* $\langle ta\rangle_{\mathrm{w}}]\!] \Longrightarrow$ *waits ta* $= \emptyset$

Fig. 25 Definition of locale *progress*

The locale *progress* fixes a set *wf-states* of well-formed states. *wf-stateD* ensures that well-formed states satisfy two invariants: Only existing threads hold the locks (*ok-locks-tp s*) and all threads in *dom* (*wset s*) exist in *tp s* and their local state is not final (*ok-wset-final s*). Together with the other constraints, the latter invariant ensures that threads that have been removed from a wait set can 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 execute a step. The remaining assumptions restrict the single-thread semantics such that stuck 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, *wf-red* formalises that the transitions cover sufficiently many thread actions. *ok-ta'* formalises that *ta*'s conditions are met except for BTAs which are allowed to cause deadlock. It is like *ok-ta* with the following modifications:

- *ok-Ls* stops checking the lock preconditions when it encounters the first *Lock* BTA for l that t cannot acquire, but it does enforce the preconditions of *Unlock* and *UnlockFail* prior to this *Lock* BTA. For example, if $ta = (\!|\mathit{Lock}{\rightarrow}l, \mathit{Unlock}{\rightarrow}l, \mathit{Unlock}{\rightarrow}l|\!)$, then *ok-ta'* requires that t already holds the lock l once, or that it cannot acquire the lock l.
- *ok-intr* ignores conditions of BTAs of the form *IsIntrd* _ *True*.
- *ok-cond s t* (*Join t'*) is always *True*.

Thus, *wf-red* requires that every thread which is ready to execute, say with thread action *ta*, can reduce with thread action *ta'* such that either the current state *s* already meets *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 *ta'* compared to *ta*.

Assumption *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.

Wakeup-waits requires that when a thread processes the removal from a wait set, it does not execute BTAs which may cause deadlock. Although the interleaving semantics could deal with such BTAs, I disallow them to simplify the deadlock formalisation and proofs as neither source code nor bytecode semantics uses this.

Proof (Theorem 10) I first show that there is a thread, say *tp s t* $= \lfloor(x, ln)\rfloor$, such that

(a) t is not waiting, $ln = (\lambda\_.\ 0)$, not *final x*, and either not $t \vdash (x, \mathit{shr}\ s)\wr$ or there is a set W such that $t \vdash (x, \mathit{shr}\ s)\ W\wr$ and t need not wait for any $w \in W$, or
(b) t is not waiting, $ln \neq (\lambda\_.\ 0)$, and it need not wait for any of the locks in ln, or
(c) $ln = (\lambda\_.\ 0)$ and t's wait set status is $\lfloor$*WS-Notified*$\rfloor$ or $\lfloor$*WS-WokenUp*$\rfloor$.

Suppose there was none. Then it is easy to show by coinduction that $t \in \mathit{deadlock}\ s$ holds for all t such that $\mathit{not\text{-}final\text{-}thread}\ s\ t$. But this contradicts that s is not in deadlock. So let t be a thread with the above properties. I show for each case that t can take a step next.

In case (a), *progress* postulates a step $t \vdash (x, \mathit{shr}\ s) -ta\rightarrow (\_, \_)$. *wf-ta* ensures that the thread action of any step is not contradictory in itself, in particular *ta* is not. Hence, $t \vdash (x, \mathit{shr}\ s)\wr$ by definition. With (a), let W be such that $t \vdash (x, \mathit{shr}\ s)\ W\wr$ and t need not wait for any $w \in W$. Again by definition, there is a step $t \vdash (x, \mathit{shr}\ s) -ta'\rightarrow (\_, \_)$ with $W = \mathit{waits}\ ta'$. By *wf-red*, there is another step $t \vdash (x, \mathit{shr}\ s) -ta''\rightarrow (x', h')$ such that either (i) $\mathit{ok\text{-}ta}\ s\ t\ ta''$, or (ii) $\mathit{ok\text{-}ta'}\ s\ t\ ta''$ and $\mathit{waits}\ ta'' \subseteq \mathit{waits}\ ta'$. In case (i), I am done by NORMAL because *upd-ta* is a right-total relation. In case (ii), by choice of ta', all $w \in \mathit{waits}\ ta'' \subseteq \mathit{waits}\ ta'$ meet their precondition. Hence, $\mathit{ok\text{-}ta'}\ s\ t\ ta''$ implies $\mathit{ok\text{-}ta}\ s\ t\ ta''$, and I am back at case (i).

In case (b), the step directly follows with ACQ.

In case (c), t must process its removal from a wait set. By *wf-stateD*, *ok-wset-final s*, in particular not *final x*. By the same argument as in case (a), there is a step $t \vdash (x, \mathit{shr}\ s) -ta''\rightarrow (x', h')$ such that $\mathit{ok\text{-}ta}\ s\ t\ ta''$ or $\mathit{ok\text{-}ta'}\ s\ t\ ta''$. If $\mathit{ok\text{-}ta'}\ s\ t\ ta''$, $\langle ta''\rangle_{\mathrm{w}}$ contains *Notified* or *WokenUp* due to t's wait set status. By *Wakeup-waits*, $\mathit{waits}\ ta'' = \emptyset$, i.e., $\mathit{ok\text{-}ta'}\ s\ t\ ta''$ and $\mathit{ok\text{-}ta}\ s\ t\ ta''$ coincide. Thus, NORMAL yields the desired step. □

3.4 Type safety for source code

With the above preparations in place, I now prove type safety for *J* (Theorem 11). Thereby, I reuse the (adapted) lemmas from the type safety proof for Jinja [32].

Theorem 11 (Type safety for J) *Let wf-J-prog P and wf-start P C M vs and suppose that* $P \vdash \mathit{J\text{-}start}\ P\ C\ M\ vs -ttas\rightarrow^* s$ *such that* $\neg P \vdash s -t':ta'\rightarrow s'$ *for any* t', ta', s'. *Then, for every thread t in s, say* $\mathit{tp}\ s\ t = \lfloor((e, \_), ln)\rfloor$,

(i) if $e = \mathit{Val}\ v$, *then* $ln = (\lambda\_.\ 0)$ *and* $P, \mathit{shr}\ s \vdash v :\leq T$ *where*

$$(\mathit{start\text{-}ETs}\ P\ C\ M)(ttas\,[\rightsquigarrow]\,P, \_, \_ \vdash \_, \_\ \surd)\ t = \lfloor(E, T)\rfloor,$$

start-ETs P C M is the initial map $[\mathit{start\text{-}tID}\mapsto(\mathsf{empty}, T_r)]$, *and* T_r *is M's return type.*

(ii) if $e = \mathit{Throw}\ a$, *then* $ln = (\lambda\_.\ 0)$ *and* $\mathit{typeof\text{-}addr}\ (\mathit{shr}\ s)\ a = \lfloor\mathit{Class}\ C\rfloor$ *for some C such that* $P \vdash C \preceq^* \mathit{Throwable}$,

(iii) otherwise, $t \in \mathit{J.deadlocked}\ P\ s$.

In any case, t has an associated Thread *object at address 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 *vs* 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 $t = \mathit{start\text{-}tID}$ and *run* otherwise, (the notation $P, h \vdash v :\leq T$ is defined below) 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.

Thanks to the thread-wise notion for deadlocks, the type safety theorem exhaustively lists all possible cases. With *deadlock* only, I would have to omit case (iii) because the assumptions of the theorem already imply that $\mathit{J.deadlock}\ s$.

locale *typesafe* = *heap* + fixes *hconf* :: $'heap \Rightarrow bool$
assumes *hconf empty-heap*
and $[\![(h', a) \in alloc\ h\ T;\ hconf\ h;\ is\text{-}type\ P\ T]\!] \Longrightarrow hconf\ h'$
and $[\![write\ h\ a\ al\ v\ h';\ hconf\ h;\ P, h \vdash a{\cdot}al : T;\ P, h \vdash v :\leq T]\!] \Longrightarrow hconf\ h'$
and $[\![typeof\text{-}addr\ h\ a = \lfloor T \rfloor;\ hconf\ h]\!] \Longrightarrow is\text{-}type\ P\ T$
and $[\![hconf\ h;\ P, h \vdash a{\cdot}al : T]\!] \Longrightarrow \exists v.\ read\ h\ a\ al\ v$
and $[\![hconf\ h;\ P, h \vdash a{\cdot}al : T;\ P, h \vdash v :\leq T]\!] \Longrightarrow \exists h'.\ write\ h\ a\ al\ v\ h'$
and $[\![read\ h\ a\ al\ v;\ hconf\ h;\ P, h \vdash a{\cdot}al : T]\!] \Longrightarrow P, h \vdash v :\leq T$

Fig. 26 Type safety assumptions on the shared state ADT

In the remainder of this section, I develop the invariant necessary to prove Theorem 11 via progress and preservation. I always assume that P is well-formed, i.e., *wf-J-prog P*.

Recall that progress requires more than showing 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. The locale *progress* (Figure 25) collects sufficient conditions for a single-threaded progress property being extended to the interleaving semantics (Theorem 10). Now, I show that *J-red P* satisfies these conditions. First, I expand on the set of well-formed states *wf-states* for *J*.

3.4.1 Thread-local well-formedness constraints

The type safety lemmas in Jinja require the following invariants:

Conformance Conformance expresses that semantic objects conform to their syntactic description. A value v conforms to a type T (written $P, h \vdash v :\leq T$) iff v's dynamic type is a subtype of T:

$$P, h \vdash v :\leq T = (\exists T'.\ typeof_h\ v = \lfloor T' \rfloor \wedge P \vdash T' \leq T)$$

where $typeof_h\ v$ returns the type of a value v using *typeof-addr h* for addresses. This conformance notion naturally extends to list of values and types (written $P, h \vdash vs\ [:\leq]\ Ts$), stores and environments (written $P, h \vdash x\ (:\leq)\ E$), objects, and arrays. Conformance of the heap is also required, but as the heap is abstract (Section 2.3.2), I also leave heap conformance *hconf* abstract for now (the memory models in Section 4 define *hconf*). Then, a heap and a store are conformant $P, E \vdash (h, x)\surd$ iff *hconf h* and $P, h \vdash x\ (:\leq)\ E$. 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.

As *hconf* is abstract, the locale *typesafe* collects the assumptions needed for type safety (Figure 26). This requires the notion of the type of a location. Let $P, h \vdash a{\cdot}al : T$ denote that the location (a, al) is supposed to store values that conform to type T. Formally:

$$\frac{typeof\text{-}addr\ h\ a = \lfloor T \rfloor \qquad P \vdash class\text{-}of'\ T\ has\ F{:}T'\ (fm)\ in\ D}{P, h \vdash a{\cdot}Field\ D\ F : T'}$$

$$\frac{typeof\text{-}addr\ h\ a = \lfloor Array\ T\ n' \rfloor \qquad n < n'}{P, h \vdash a{\cdot}Cell\ n : T}$$

where $P \vdash C\ has\ F{:}T\ (fm)\ in\ D$ denotes that in program P, the class C has a field F of type T declared in class D with field modifiers *fm*.

Now, consider the locale *typesafe* in detail. Clearly, the empty heap must conform and the heap-manipulating operations *alloc* and *write* preserve heap conformance when valid types are allocated and type-conforming values written. Moreover, heap conformance must ensure that the dynamic types of all addresses are valid, too. The second group of assumption demands that typeable locations can be read from and written to (progress) and that values read conform to the location types (subject reduction).

Run-time type system The typing rules from Section 2.3.1 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 [16] $P,E,h \vdash e : T$ takes the heap into account and relaxes various preconditions that are not invariant. For example, the constraint $T_1 \neq NT$ in WTSYNC may be violated because e_1 may become *null*; the semantics throws a *NullPointer* exception in that case (RSYNCN). Thus, rule WTRTSYNC replaces WTSYNC. Moreover, the new rule WTRTINSYNC for *insync* (a) e statements requires the monitor address to be allocated and the body to be run-time typable.

$$\text{WTRTSYNC:}\ \frac{P,E,h \vdash e_1 : T_1 \qquad \textit{is-refT}\ T_1 \qquad P,E,h \vdash e_2 : T}{P,E,h \vdash \textit{sync}\ (e_1)\ e_2 : T}$$

$$\text{WTRTINSYNC:}\ \frac{\textit{typeof-addr}\ h\ a \neq \textit{None} \qquad P,E,h \vdash e : T}{P,E,h \vdash \textit{insync}\ (a)\ e : T}$$

The details for the other language constructs have been discussed at length elsewhere [16, 32], so I do not repeat them here.

Thread conformance The ID of the executing thread must have an associated thread object as *currentThread* returns its address (CURRTH). Thread conformance $P,h \vdash t\ \surd_t$ demands that *typeof-addr* $h\ t = \lfloor \textit{Class}\ C \rfloor$ for some C such that $P \vdash C \preceq^* \textit{Thread}$.

Now, progress and various preservation lemmas hold for the single-threaded semantics. I only present progress and subject reduction. They differ from JinjaThreads's predecessor Jinja [32, Lemma 2.8] only in the highlighted parts. This shows that I only minor modifications were necessary to reuse the theorems (and their proofs).

Lemma 12 (Single-threaded progress) *If* wf-J-prog *P and hconf h and* $P,E,h \vdash e : T$ *and* $\mathscr{D}\ e\ \lfloor \textit{dom}\ x \rfloor$ *and* $\neg$ *final e, then* $P,$ t $\vdash \langle e,(h,x)\rangle -$ *ta* $\rightarrow \langle e',s'\rangle$ *for some* *ta and* e' *and* s'.

Theorem 13 (Subject reduction) *If wf-J-prog P, and* $P,$ t $\vdash \langle e,s\rangle -$ *ta* $\rightarrow \langle e',s'\rangle$, *and* $P,E \vdash s\surd$, *and* $P,E,\textit{hp}\ s \vdash e : T$, *and* $P,\textit{hp}\ s \vdash t\ \surd_t$, *then there is a* T' *such that* $P,E,\textit{hp}\ s' \vdash e' : T'$ *and* $P \vdash T' \leq T$.

Lifting to the multithreaded semantics Clearly, *wf-states* imposes all these thread-local constraints. In Section 3.2, I have presented how to lift such predicates and preservation theorems to the interleaving semantics. For definite assignment, the lifted predicate (notation $\uparrow\mathscr{D}\uparrow$) is $\uparrow\lambda t\ (e,x)\ h.\ \mathscr{D}\ e\ \lfloor \textit{dom}\ x \rfloor\uparrow$. Conformance and typability depend on a typing environment E and the initial type T of the expression, which do not change during reduction. Thus, I model them as invariant data in a combined predicate. Let $P,(E,T),t \vdash (e,x),h\ \surd$ denote

$$\exists T'.\ P,E,h \vdash e : T' \wedge P \vdash T' \leq T \wedge P,E \vdash (h,x)\surd \wedge P,h \vdash t\ \surd_t \tag{2}$$

Then, $P,\_ \vdash \_,\_\ \Uparrow\surd\Uparrow$ lifts $P,\_,\_ \vdash \_,\_\ \surd$ to thread pools:

$$P,ET \vdash tp,h\ \Uparrow\surd\Uparrow = \Uparrow P,\_,\_ \vdash \_,\_\ \surd\Uparrow\ ET\ tp\ h$$

Lemma 14 (Preservation of definite assignment, typability, and conformance) *Suppose* $P \vdash s\ -t{:}ta\rightarrow\ s'$. *Let* $ET' = ET(\langle ta\rangle_t \rightsquigarrow P, \_, \_ \vdash \_, \_\ \surd)$.

(i) If $\uparrow\mathscr{D}\uparrow\ (tp\ s)\ (shr\ s)$, *then* $\uparrow\mathscr{D}\uparrow\ (tp\ s')\ (shr\ s')$.
(ii) If $P, ET \vdash tp\ s, shr\ s\ \Uparrow\surd\Uparrow$, *then* $P, ET' \vdash tp\ s', shr\ s'\ \Uparrow\surd\Uparrow$.

3.4.2 Inter-thread well-formedness constraints

The thread-local constraints do not suffice to discharge the assumption *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 such that one of them is always feasible. For example, START and STARTFAIL complement each other, and so do INTRDT and INTRDF. But if a thread-local state is inconsistent with the multithreaded state, *wf-red* may be violated in two cases:

1. The lock status assigns less locks to a thread than its *insync* (_) _ blocks remember. In that case, RUNLOCK and RUNLOCKX try to unlock a monitor that is not held, but there is no reduction with *UnlockFail*.
2. The wait set status is $\lfloor$*WS-Notified*$\rfloor$ or $\lfloor$*WS-WokenUp*$\rfloor$, but the next reduction is not a call to *wait*. Then, the semantics cannot issue a thread action with *Notified* or *WokenUp*.

For both cases, I introduce additional constraints that the reductions in J preserve.

For case 1, I define a function $\mathscr{I} :: expr \Rightarrow addr \Rightarrow nat$ such that $\mathscr{I}\ e\ a$ counts the *insync* (a) _ subexpressions in e for any monitor address a. I write *has-*$\mathscr{I}\ e$ if $\mathscr{I}\ e$ is not 0 everywhere, i.e., e contains at least one *insync* (_) _ subexpression. Then, the lock consistency invariant *lock-ok ls tp* expresses that for all thread IDs t,

(i) if t does not exist, it holds no locks, i.e., if $tp\ t = None$, then $\neg$ *has-lock* $(ls\ a)\ t$ for all monitor addresses a, and
(ii) if t exists, its *insync* (_) _ subexpressions remember its locks, i.e., if $tp\ t = \lfloor((e, \_), ln)\rfloor$, then $\mathscr{I}\ e\ a = has\text{-}locks\ (ls\ a)\ t + ln\ a$ for all a. Note that *lock-ok* must add t's temporarily released locks $(ln\ a)$ to the locks t actually holds (*has-locks* $(ls\ a)\ t$) as the *insync* (a) _ blocks remain when a call to *wait* on a temporarily releases the lock on a.

Preservation of *lock-ok* requires another invariant. Consider, for example, RSYNCN and suppose that e has an *insync* (a) _ subexpression. Then, $\mathscr{I}\ (sync\ (null)\ e)\ a > 0$ and $\mathscr{I}\ (THROW\ NullPointer)\ a = 0$, but a's lock state does not change. Hence, if the original state satisfies *lock-ok*, the successor state will not. The problem here is that e contains an *insync* (_) _ block although execution has not yet reached it.

To disallow such cases, I define the predicate *ok-*$\mathscr{I}\ e$ which ensures that *insync* (_) _ subexpressions occur only in subexpressions in which the next reduction will take place. Figure 27 shows representative cases of the definition. For expressions with subexpressions, the definition follows a common pattern; all subexpressions must satisfy *ok-*$\mathscr{I}$, too, and if *has-*$\mathscr{I}$ holds for any subexpression, then all subexpressions which are evaluated before must be a value, i.e., of the form *Val* _, which the predicate *is-Val* checks. For example, if *has-*$\mathscr{I}\ e_2$ in $e_1 \ll bop \gg e_2$, 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. Clearly, if $\neg$ *has-*$\mathscr{I}\ e$, then *ok-*$\mathscr{I}\ e$.

Like with the other thread-local well-formedness conditions, I lift *ok-*$\mathscr{I}$ to multithreaded states, written $\uparrow$*ok-*$\mathscr{I}\uparrow$, and show preservation with the locale *lifting-wf*.

$$
\begin{array}{ll}
\textit{ok-}\mathscr{I}\ (e_1 \ll bop \gg e_2) & = \textit{ok-}\mathscr{I}\ e_1 \wedge \textit{ok-}\mathscr{I}\ e_2 \wedge (\textit{has-}\mathscr{I}\ e_2 \longrightarrow \textit{is-Val}\ e_1) \\
\textit{ok-}\mathscr{I}\ (V := e) & = \textit{ok-}\mathscr{I}\ e \\
\textit{ok-}\mathscr{I}\ (e_1;;\ e_2) & = \textit{ok-}\mathscr{I}\ e_1 \wedge \neg\, \textit{has-}\mathscr{I}\ e_2 \\
\textit{ok-}\mathscr{I}\ (\textit{if}\ (e)\ e_1\ \textit{else}\ e_2) & = \textit{ok-}\mathscr{I}\ e \wedge \neg\, \textit{has-}\mathscr{I}\ e_1 \wedge \neg\, \textit{has-}\mathscr{I}\ e_2 \\
\textit{ok-}\mathscr{I}\ (\textit{while}\ (e_1)\ e_2) & = \neg\, \textit{has-}\mathscr{I}\ e_1 \wedge \neg\, \textit{has-}\mathscr{I}\ e_2 \\
\textit{ok-}\mathscr{I}\ (\textit{sync}\ (e_1)\ e_2) & = \textit{ok-}\mathscr{I}\ e_1 \wedge \neg\, \textit{has-}\mathscr{I}\ e_2 \\
\textit{ok-}\mathscr{I}\ (\textit{insync}\ (a)\ e) & = \textit{ok-}\mathscr{I}\ e
\end{array}
$$

Fig. 27 Definition excerpt of $\textit{ok-}\mathscr{I}$

Lemma 15 (Preservation of *ok-*$\mathscr{I}$ and $\uparrow$*ok-*$\mathscr{I}\uparrow$)

(i) If $P,t \vdash \langle e,s\rangle -ta\rightarrow \langle e',s'\rangle$ *and* $\textit{ok-}\mathscr{I}\ e$, *then* $\textit{ok-}\mathscr{I}\ e'$.
(ii) If $P \vdash s -t{:}ta\rightarrow s'$ *and* $\uparrow\textit{ok-}\mathscr{I}\uparrow\ (\textit{tp}\ s)\ (\textit{shr}\ s)$, *then* $\uparrow\textit{ok-}\mathscr{I}\uparrow\ (\textit{tp}\ s')\ (\textit{shr}\ s')$.

Proof Case (i) is proven by induction on the semantics.

Case (ii) holds by Lemma 4 and instantiating the locale *lifting-wf* (Figure 21). The first assumption is discharged by case (i). The second assumption holds because the body *body* of the *run* method of the spawned thread is well-typed as P is well-formed. Well-typed expressions have no *insync* $(\_)\ \_$ subexpressions because there is no typing rule for them. So, $\textit{ok-}\mathscr{I}\ \textit{body}$. The third assumption is vacuous as $\textit{ok-}\mathscr{I}$ does not depend on the heap. □

Lemma 16 (Preservation of *lock-ok*) *If* $\textit{lock-ok}\ (\textit{locks}\ s)(\textit{tp}\ s)$ *and* $\uparrow\textit{ok-}\mathscr{I}\uparrow\ (\textit{tp}\ s)\ (\textit{shr}\ s)$ *and* $P \vdash s -t{:}ta\rightarrow s'$, *then* $\textit{lock-ok}\ (\textit{locks}\ s')\ (\textit{tp}\ s')$.

Next, I turn to the second way in which *J-red* may violate *wf-red*. In principle, I could pursue the same path as for lock consistency 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 Section 3.5.

Given a set I of well-formed multithreaded states, *ok-Susp 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*):

$$
\begin{array}{l}
\textit{ok-Susp}\ I = \{\, s.\ s \in I \wedge (\forall t \in \textit{dom}\ (\textit{wset}\ s).\ \exists s_0 \in I.\ \exists s_1 \in I.\ \exists \textit{ttas}\ x_0\ \textit{ta}\ x\ w\ \textit{ln}\ \textit{ln}'. \\
\quad s_0 -t{:}ta\rightarrow s_1 \wedge s_1 -\textit{ttas}\rightarrow^* s \wedge \textit{tp}\ s_0\ t = \lfloor (x_0, \lambda\_.\ 0)\rfloor \wedge t \vdash (x_0, \textit{shr}\ s_0) -ta\rightarrow (x, \textit{shr}\ s_1) \wedge \\
\quad \textit{Suspend}\ w \in \textit{set}\ \langle ta\rangle_{\mathrm{w}} \wedge \textit{ok-ta}\ s_0\ t\ ta \wedge \textit{tp}\ s_1\ t = \lfloor (x, ln)\rfloor \wedge \textit{tp}\ s\ t = \lfloor (x, ln')\rfloor \,\}
\end{array}
$$

Clearly, *ok-Susp* preserves preservation of invariants by definition.

Lemma 17 (Preservation of *ok-Susp*) *If redT preserves the invariant I, then redT preserves ok-Susp I.*

3.4.3 Type safety

Now, the set *J-wf-states P* of well-formed states for the type safety proof is defined as

$$
\begin{array}{l}
\textit{J.ok-Susp}\ P\ \{\, s.\ \exists ET.\ P, ET \vdash \textit{tp}\ s, \textit{shr}\ s \Uparrow\surd\Uparrow \wedge \uparrow\mathscr{D}\uparrow\ (\textit{tp}\ s)\ (\textit{shr}\ s) \wedge \\
\qquad\qquad \uparrow\textit{ok-}\mathscr{I}\uparrow\ (\textit{tp}\ s)\ (\textit{shr}\ s) \wedge \textit{lock-ok}\ (\textit{locks}\ s)\ (\textit{tp}\ s)\,\}
\end{array}
$$

Lemma 18 *J satisfies the assumptions of* progress *for well-formed states J-wf-states P.*

Proof I proof the assumptions of locale *progress* (Figure 25) as follows. The well-formedness condition *ok-locks-tp* directly follows from *lock-ok* because this is just case (i) in

lock-ok's definition. *ok-wset-final* follows from *J.ok-Susp* and the assumption *Suspend-not-final*, which I discharge below. *progress* is just the progress lemma 12. 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*. The proof proceeds by induction over the small-step semantics. The interesting cases are RUNLOCK, RUNLOCKX and RNATIVE.

In case RUNLOCK, $\mathscr{I}$ (*insync* (*a*) _) *a* > 0, i.e., by lock consistency, t holds the lock a. Thus, unlocking a is possible, i.e., *ok-ta'* *s t ta* holds. The case RUNLOCKX is analogous. 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 a representative 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 19 *If s* ∈ *J-wf-states P is not in deadlock, then P* ⊢ *s* −*t*:*ta*→ *s'* *for some t, ta and s'.*

Proof This is Theorem 10 with Lemma 18 discharging the locale assumptions. □

Lemma 20 *The initial state J-start P C M vs is well-formed.*
If wf-J-prog P and wf-start P C M vs, then J-start P C M vs ∈ *J-wf-states P.*

Finally, I am able to prove type safety for *J*.

Proof (Proof of Theorem 11) By Lemma 20, *J-start P C M vs* ∈ *J-wf-states P*. Since reductions preserve *J-wf-states P*, I have *s* ∈ *J-wf-states P*, too. Hence, *s* ∈ *J.deadlock P* by Corollary 19, which subsumes all *mfinal* states. If *e* is *final*, cases (i) and (ii) follow from *s* being well-formed. Case (iii) follows by definition of *deadlock s*. The associated thread object exists because *s* ∈ *J-wf-states P* implies thread conformance. □

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 (Section 3.4), namely (i) identify necessary invariants, and (ii) prove the assumptions of locale *progress*. Instead of presenting the steps in detail once more, I focus on the similarities with and differences from source code.

3.5.1 Well-typings

When executing bytecode, the JinjaThreads VM relies on the following assumptions: The stack contains 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 that these assumptions are violated during execution, the bytecode must satisfy certain type constraints, similar to the typing rules for source code. As bytecode does not declare the types of the registers and the stack elements, JinjaThreads models type information separately, Since the formalisation does not differ from Jinja in any essential way, I only sketch the main ideas and introduce the relevant notation. For details, see [32].

A state type τ characterises a set of run-time states for one instruction by giving type information for the operand stack and registers. $\tau = \mathit{None}$ denotes that control flow cannot reach the instruction. Otherwise, say $\tau = \lfloor(ST, LT)\rfloor$, $ST :: \mathit{ty\ list}$ gives the types for the elements on the operand stack and LT the types for the register contents. The elements of LT are either Err or $\mathit{OK}\ T$ for some type $T :: \mathit{ty}$. Err denotes that a register is unusable and its type is unknown, e.g., if it has not been initialised yet. For example, $\lfloor([], [\mathit{OK}\ (\mathit{Class}\ C),\ \mathit{Err}])\rfloor$ 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 Null pointer), the second is unusable. A method type τs is a list of state types, one for each instruction of the method. A program typing for P is a function Φ such that $\Phi\ C\ M$ is the method type for every method M in every class C of P.

A state type τ for an instruction i is a state well-typing iff i can execute safely in any state that τ characterises and τ is consistent with the successor instruction's state type. Consistency is formalised using an abstract interpretation framework [32] that is not needed to understand this article. A method type τs is a method well-typing iff each of its state types is a state well-typing for its instruction.

A bytecode method declaration $(M, Ts, T, \lfloor(msl, mxl, ins, xt)\rfloor)$ in class D is well-typed with respect to τs iff

(i) τs is a well-typing for the method,
(ii) all state types in τs contain only valid types and respect the maximum stack length msl and the fixed number mxl of registers, and
(iii) τs satisfies the start condition, i.e., it is non-empty and the first state type $\tau s_{[0]}$ is at least as general as $\lfloor([], \mathit{OK}\ (\mathit{Class}\ D) \cdot \mathit{map\ OK\ Ts}\ @\ \mathit{replicate\ mxl\ Err})\rfloor$ in the abstract interpretation order, i.e., the stack is empty and the registers contain the *this* pointer and the parameters and all local variables are inaccessible.

A program typing Φ is a well-typing for P iff every method M in every class C of P is well-typed with respect to $\Phi\ C\ M$. A program P is well-formed (written $\mathit{wf\text{-}jvm\text{-}prog}\ P$) iff it satisfies the generic well-formedness constraints and there is a well-typing Φ 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 models a bytecode verifier in the style of Jinja that rephrases the abstract interpretation as a data flow analysis problem and computes a well-typing with Kildall's algorithm [27].

3.5.2 Type safety

Throughout this section, I assume that Φ is a well-typing for P. The type safety proof requires a thread-local invariant called conformance, written $P, \Phi \vdash t{:}s\surd$. Conformance requires that the state type correctly abstracts the run-time state (xcp, h, frs) of a thread t, i.e.,

(i) if $xcp = \lfloor a\rfloor$ flags the exception at address a, then a's dynamic type is a subclass of $\mathit{Throwable}$ and conforms to the exception specification in the abstract interpretation for the current instruction in the top-most call frame,
(ii) the heap conforms, i.e., $\mathit{hconf}\ h$,
(iii) the thread ID conforms, i.e., $P, h \vdash t\ \surd_t$,
(iv) for all call frames (stk, loc, C, M, pc) in frs, 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
(v) 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.

In comparison to Jinja's notion of conformance, condition (iii) is new and condition (i) strengthens a's type to be a subclass of *Throwable* instead of *Object*. The latter disallows arrays to be thrown, which Jinja does not model. All other conditions are unchanged.

For a well-typed program, I can now show type safety:

Theorem 21 (Type safety) *Let P be well-formed with well-typing Φ and the start state jvm-start P C M vs be well-formed. Then, the following hold:*

(a) The aggressive VM runs until all threads have terminated or are deadlocked. Formally: If $P \vdash$ jvm-start P C M vs $-ttas\rightarrow^*_{\text{jvm}} s$ such that $\neg P \vdash s\ -t{:}ta\rightarrow_{\text{jvm}} s'$ for any t, ta, s', then for every thread t in s, say tp s $t = \lfloor((xcp,frs),ln)\rfloor$,
(i) $P,\Phi \vdash t{:}(xcp, \text{shr}\ s, frs)\surd$, and
(ii) if $frs \neq []$ or $ln \neq (\lambda\_.\ 0)$, then $t \in$ jvm.deadlocked P s.

(b) Aggressive and defensive VM have the same behaviours. Formally:
(i) $P \vdash$ jvm-start P C M vs $-ttas\rightarrow^*_{\text{jvm}} s$ iff $P \vdash$ jvm-start P C M vs $-ttas\rightarrow^*_{\text{jvmd}} s$.
(ii) $P \vdash$ jvm-start P C M vs $\Rightarrow_{jvm}$ ttass iff $P \vdash$ jvm-start P C M vs $\Rightarrow_{jvmd}$ ttass.

Part (a) corresponds to the type safety statement for J (Theorem 11). Part (b) shows that it is safe to run the aggressive VM instead of the defensive VM. The two statements (b)(i) and (b)(ii) are not necessarily equivalent because (b)(i) refers to finite prefixes of executions whereas (b)(ii) talks about complete runs in the respective interleaving semantics ($P \vdash \_ \Rightarrow_{jvm} \_$ are the complete runs for the aggressive VM as defined by STOP and STEP and $P \vdash \_ \Rightarrow_{jvmd} \_$ similarly for the defensive VM).[8]

The proof is similar to the type safety proof for source code and therefore not shown. Instead, I compare it to the one for the sequential VM in Jinja [32, Theorem 4.10]. Klein and Nipkow showed that the defensive VM cannot reach a *TypeError* from a well-formed start state in a well-formed program. Recall from Section 2.4.2 that the defensive JinjaThreads VM cannot raise *TypeError*s, but gets stuck. This design choice is motivated by two considerations. First, without *TypeError*s, the defensive and the aggressive VM operate on the same state space, so the same proof invariants can be used. Second, when a single thread raises a *TypeError*, the whole VM should immediately halt. But this does not fit the structure of the interleaving semantics where one thread cannot abort the execution of other threads. Hence, I cannot express the absence of type errors directly. Instead, I show progress, which is equivalent to the absence of type errors for individual steps because type errors and normal reductions exclude each other in the defensive VM by construction. Still, Theorem 21 is slightly weaker with respect to type errors as it does not exclude the situation in which one thread is stuck at a type error and another thread runs for ever.

4 Memory models

The heap has been left abstract in the semantics and type safety proofs in Sections 2 and 3. I now sketch the two implementations in JinjaThreads, sequential consistency and the JMM.

4.1 Sequential consistency

The first implementation is a standard heap, i.e., a map from addresses to objects and arrays. Addresses are modelled as natural numbers. Objects store the class name and a field table

[8] For example, abstractly, finite prefixes of executions do not distinguish between the two transition systems $3 \leftarrow 0 \rightarrow 1 \rightarrow 2$ and $2 \leftarrow 3 \leftarrow 0 \rightarrow 1 \rightarrow 2$ with initial state 0, but complete runs do. Conversely, complete runs identify $\ldots \leftarrow -2 \leftarrow -1 \leftarrow 0 \rightarrow 1 \rightarrow 2 \ldots$ with $0 \rightarrow 1 \rightarrow 2 \rightarrow \ldots$, but finite prefixes do not.

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 have an element type T, a list of cells each of which contains a value, and a field table for the fields inherited from *Object*. The length of the cell list determines the array length.

The heap operations are all deterministic. Allocation picks the least unallocated address and assigns it to an object or array with the specified fields and cells initialised to the appropriate default value (*Intg* 0, *Bool False*, *Unit*, or *Null*). Reading v from a location (a, al) is possible iff the current heap stores the value v in member al of the object or array at address a. Similarly, writing v to location (a, al) succeeds if a is allocated and updates a's member al to v. Heap conformance requires that 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 all array cells store type-conforming values.

Since the heap is passed as a shared state between the threads, this implementation leads to sequential consistency [38] as the memory model of the multithreaded semantics. This implementation satisfies the assumptions of locale *typesafe*. Therefore, JinjaThreads source code and bytecode are type safe under sequential consistency.

4.2 The Java Memory Model

The formalisation of the Java memory model has been described in another article [50]. Here, I only summarize the results relevant for this article.

In this implementation, it is the address and not the heap that stores the dynamic type information. So, an address *Address T n* consist of the dynamic type T :: *hty* and a sequence number n to distinguish objects of the same type. The shared state h stores only which addresses have been allocated. Allocation returns the set of all possible free unallocated addresses of the specified type. The function *typeof-addr* now depends on the program P, as it returns *None* for addresses with invalid types such as undeclared classes.

$$\textit{typeof-addr}\ P\ h\ (\textit{Address}\ T\ n) = (\textit{if}\ \textit{is-type}\ P\ T\ \textit{then}\ \lfloor T \rfloor\ \textit{else}\ \textit{None})$$

This restriction is necessary to meet the third assumption of locale *typesafe*. Reading is completely unrestricted, i.e., any value can be read from any member of any address at any time. Writing does not change the heap. Every heap is conformant.

On top of complete runs, at layer 7 in Figure 2, the JMM imposes constraints for well-formed and legal executions. In particular, every read of a value from a location must be matched by a visible write of that value to the same location. To that end, the semantics records in the last component $'o$ of thread actions all reads from and writes to the locations in memory (not shown in this article). It is not possible to decide whether an execution is legal by looking at this execution in isolation [6]. Thus, legality is a hyperproperty of executions.

Including the dynamic type information into the address makes sure that the semantics meets the JLS requirement that type information behave sequentially consistent [21, §17.4.5]. I have also experimented with storing type information in the heap instead of the addresses, but then type safety is violated [50].

Unfortunately, the above heap implementation does not satisfy the last assumption of *typesafe*. Therefore, JinjaThreads defines another version of the JMM implementation in which only type-conforming values can be read from memory. In [50], I have shown that both versions lead to the same set of legal executions, i.e., the JMM is type safe. In particular, the restricted implementation does satisfy the assumptions of *typesafe*. Therefore, JinjaThreads source code and bytecode are also type safe under the JMM.

In addition to type safety, I have shown that correctly synchronized JinjaThreads programs behave sequentially consistent under the JMM (the so-called data-race freedom guarantee) and that every sequentially consistent behaviour is allowed, i.e., the JMM indeed relaxes sequential consistency [50].

5 Compiler

JinjaThreads's compiler from source code to bytecode bridges the gap between the two languages; its correctness proof shows that both fit together. In this section, I extend Jinja's non-optimising compiler [32, §5] to JinjaThreads and prove that it preserves

well-formedness: It compiles well-formed source code into well-formed bytecode (Theorem 54).

the semantics: If the source program terminates or diverges or deadlocks, then so does the compiled program and vice versa (Theorem 57). In any case, 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 58).

correct synchronisation: The compiled program is correctly synchronised iff the source code is (Corollary 59).

5.1 High-level overview

Semantic preservation ensures that semantic properties established on the source code also hold for the compiled code. 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 non-deterministic programs). Thus, a correct compiler *Comp* must ensure that every behaviour of the compiled program *Comp P* is an acceptable behaviour of the source program *P*.

To make this formal, one must first identify what behaviour means. The JLS [21, §17.4.9] defines the behaviours of a Java program roughly as the observable external actions (e.g., reading external data or printing) in the finite prefixes of any execution; program termination and divergence are also observable. Here, divergence means that the program runs forever without producing any observable action. Since the JinjaThreads compiler does not optimize, I use a more fine-grained notion: a behaviour consists of a possibly infinite trace of thread actions of observable program transitions and—if the trace is finite—the final state or the fact that the program diverges. Formally, traces are modelled by the codatatype of possibly infinite lists where the constructor for the empty list is labelled by the final state or divergence. The JLS behaviours are projections of these behaviours because the thread actions include all external actions.

A compiler *Comp* therefore preserves the semantics of *P* iff the following holds: Let s_1 and s_2 be the initial states for *P* and *Comp P*, respectively. For every execution of *Comp P* that starts in s_2 and terminates in s_2', there must be an execution of *P* from s_1 to s_1' 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. Moreover, for every infinite execution (diverging or not) of *Comp P* that starts in s_2, *P* has an infinite execution with the same trace that starts in s_1; either both executions diverge or none.

It makes sense to consider complete traces rather than (finite) partial traces. The reason is that under the JMM, some Java programs generate legal infinite traces none of whose finite prefixes is legal [50]. Completeness can be defined abstractly for transition systems as a coinductive predicate: A trace is complete iff it is infinite, diverging, or the final state is stuck.

As is standard, I prove the trace inclusion using a simulation approach. The latter implies the former and can be shown by inspecting individual execution steps instead of whole program executions. Unfortunately, the standard simulation notions do not preserve trace completeness. Weak simulation, for example, demands that every observable transition can be simulated by a corresponding observable transition with arbitrary many unobservable transitions before and after and that every unobservable transition is simulated by finitely many unobservable transitions. So a finite complete trace, which ends in some stuck state s_2', has a finite simulating trace that ends in some state s_1'. However, as s_2' is stuck, the weak simulation condition does not impose any constraint on s_1', i.e., s_1' need not be stuck and the simulating trace therefore not complete. For example, it might be that s_2' is in deadlock whereas s_1' is not. In conclusion, such simulation notions allow the compiler to introduce deadlocks into a program that was deadlock-free. This is not acceptable.

I therefore use a bisimulation notion, i.e., I prove that every source code step can be simulated by bytecode steps and vice versa. This preserves completeness (if the bisimulation notion preserves divergence) because if s_2' is stuck and s_1' is not, then s_2' must be able to simulate s_1''s steps, but it cannot, so s_1' must get stuck after finitely many unobservable steps, as s_1' cannot diverge. This approach ensures preservation of deadlocks as deadlocks are formalised semantically. So, the compiler does not introduce deadlocks.[9] As a side effect, bisimulations establish trace equivalence instead of trace inclusion, i.e., the bytecode exhibits every behaviour of the source code. As the traces of source code and bytecode are the same, hyperproperties on traces such as possibilistic security properties [52] and JMM legality can also be transferred from source code to bytecode and back.

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 *Comp P* such that s_2 produces trace t and either ends in s_2' or runs infinitely, respectively.

As bisimulation notion, I have chosen delay bisimilarity [1,53] augmented with explicit divergence [9] because multithreaded states are delay bisimilar with explicit divergence if each of their threads is. Delay bisimulations differ from weak bisimulations in that an observable step must be simulated by a corresponding observable step and arbitrarily many unobservable steps before, but *not after*. This is crucial when a thread suspends itself to a wait set, which is an observable step. When such a suspension step is simulated, delay bisimilarity ensures that the suspension is the last simulating step. With weak bisimilarity, the simulation could include additional unobservable steps after the suspension, but as the

[9] It would probably have been possible to obtain preservation of deadlocks from a simulation proof in only one direction by deriving a syntactic characterisation of deadlock and analysing the simulation relation. This would break the abstraction layer of the interleaving semantics and destroy the modularity of the proofs, both for lifting single-threaded simulations and for showing semantic preservation under the JMM. For these reasons, I decided to go for a bisimulation approach, although showing both directions required considerable additional effort.

thread is already in the wait set, it cannot execute these steps in the interleaving semantics any more. Explicit divergence requires that for any two bisimilar states, either both can diverge (i.e., perform an infinite sequence of unobservable steps) or none can. This avoid the infinite stuttering problem where an infinite sequence of unobservable moves is simulated by no move at all. In other words, the compiler cannot introduce non-termination.

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 [39] as follows: A delay bisimulation (without explicit divergence) is well-founded iff whenever an unobservable step is simulated by no move at all, then the simulated step decreases in some fixed well-founded order on states. This ensures that such stuttering simulations can be applied only finitely many times in a row. Every well-founded delay bisimulation is a delay bisimulation with explicit divergence. The other direction does not hold. Conversely, delay bisimulations with explicit divergence compose better as they are transitive: If $\approx_1$ and $\approx_2$ are delay bisimulations with explicit divergence, so is their relational composition $\approx_1 \circ \approx_2$. Well-founded delay bisimulation do not have this property.

The two notions are used as follows: I decompose the compiler into smaller transformations and verify each on its own, i.e., prove delay bisimilarity with explicit divergence. Transitivity ensures that the overall compiler is correct, too. Each transformation itself is proven correct using a well-founded delay bisimulation, which implies delay bisimilarity with explicit divergence subject to a few technical side conditions (e.g., the bisimulation relation must preserve final states as otherwise thread joins cannot be simulated). Thus, it suffices to prove that the compiler preserves the behaviour of single threads that is observable to other threads. This separates the sequential challenges from the interleaving and deadlock issues, with which I deal on the abstract level of the interleaving semantics.

Compiling `synchronized` blocks is non-trivial in three respects: First, the source code semantics stores in a `synchronized` block the monitor address whereas bytecode caches it in a local register. Second, unlike the bytecode instructions for monitors, `synchronized` blocks enforce structured locking of monitors, i.e., unlocking never fails in source code. Hence, the compiler verification must show that unlocking a monitor never fails in compiled programs, either. Third, the monitor must also be unlocked when an exception abruptly terminates the block. The compiler adds an exception handler to ensure this.

The compiler *J2JVM* operates in two stages: The first stage $compP_1$ allocates local variables to registers (Section 5.4) and the second $compP_2$ generates bytecode (Section 5.5). Figure 28 shows its structure and the various semantics the verification involves. There is just one intermediate language J_1, but the verification spans five different semantics: For source code, I first develop a small-step semantics J_0 that makes call stacks explicit like in bytecode (Section 5.3) and prove it bisimilar to the source code semantics from Section 2.3. Two semantics for the intermediate language address the difficulty that unlocking a monitor in bytecode can fail: J_1' allows `synchronized` 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.[10]

The main verification effort is on the level of expressions and statements (last row in Figure 28), which contains all execution steps of a single thread except for calls to and returns

[10] 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 *exec-meth* and exploit that the defensive and aggressive VM have the same behaviours (Theorem 21). See Section 5.5.3 for details.

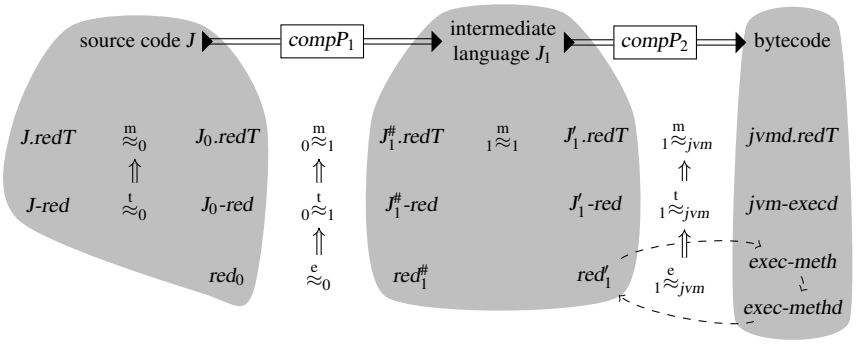


Fig. 28 Structure of the compiler (top row) and its verification. Each column corresponds to a semantics or bisimulations ($\approx$), the rows represent different levels (m = multithreaded, t = single-threaded, e = expressions). The grey areas group the semantics by the language that they belong to.

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 JMM plays only a minor role in the verification because the compiler does not optimise. The JMM legality constraints, which reside even higher in the stack of semantics, are therefore not affected and preservation of correct synchronisation follows easily from semantic preservation because the compiler preserves the set of complete runs (Section 5.2.3).

In comparison to Jinja, preservation of well-formedness follows the same lines because the languages are structured similarly. In contrast, multithreading changes the semantics drastically and, therefore, pervades the proofs, too. In particular, the verification is carried out against a small-step semantics rather than the big-step semantics as in Jinja. This complicates the proofs as the verification must deal with intermediate states and incomparable granularity of atomic steps. Nevertheless, I was able to reuse many proof invariants (although not their preservation proofs).

My simulation proofs at the expression level crunch through all the cases, in both directions. In CompCert [40], Leroy proves only that the compiled program simulates the source program, i.e., the downward direction. He then proves that CompCert's target language is deterministic and derives the diagrams for the upward direction from the downward ones. He claims that upward simulations are harder to show than downward ones and gives two reasons. First, source code typically gets stuck more often than the compiled code. In such a case, the upward simulation proof fails. Second, the compiled code typically has more intermediate states, so the upward simulation relation must relate more states, which leads to into more cases in the proof.

In JinjaThreads, this is not the case. For the second compiler stage, in particular, the upward direction is shorter than the downward direction (20 % less lines of proof script). Leroy's first reason also applies to the aggressive VM. I therefore use the defensive VM for the upward simulation proof and an almost aggressive VM for the downward direction, which are equivalent by the type-safety proof. Leroy's second argument does not hold for JinjaThreads. While the granularity of atomic steps differs between source code and bytecode, there is no clear direction that would favour an upward or downward simulation

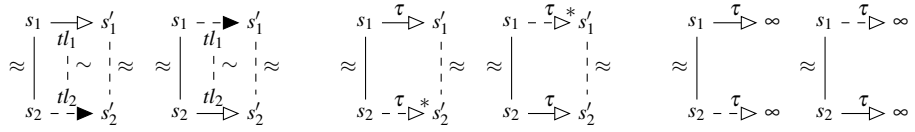


Fig. 29 Simulation diagrams for delay bisimulations with explicit divergence. Solid lines denote assumptions, dashed lines conclusions.

approach. Like in Java, most JinjaThreads language constructs in source code are compiled into one or (rarely) two bytecode instructions. `synchronized` blocks are the only exception. So, bytecode and source code granularity differ not too much for normal execution. Exception handling, in contrast, is much more fine grained in source code than in bytecode. The reason is that the VM directly jumps to the exception handler or pops the current call frame whereas exception propagation rules in source code slowly move the raised exception towards the handler. Consequently, there are much more intermediate states in source code than in byte code when it comes to exception handling.

5.2 Semantic preservation via bisimulations

In this section, I introduce delay bisimulations with explicit divergence as proof tool (Section 5.2.1) and show how preservation for single threads extends to the interleaving semantics (Section 5.2.2) and the JMM (Section 5.2.3).

5.2.1 Simulation properties

For semantic preservation, I show bisimilarity of the source code and the compiled code. I have chosen delay bisimilarity [1, 53] augmented with explicit divergence [9] because multithreaded states are delay bisimilar with explicit divergence if each of their threads is.

In this setting, programs define labelled transition systems (LTS) whose states are the program states and whose labels constitute the observable behaviour. I write $s \xrightarrow{tl}\triangleright\ s'$ for a single transition (move), i.e., an 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 $\_ -\_:\_\rightarrow \_$ fit into this format. A predicate $\tau\text{-}move\ s\ tl\ s'$ determines whether the transition $s \xrightarrow{tl}\triangleright\ s'$ 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 τ-moves. Since their labels are irrelevant, I don't keep track of them and write $s \xrightarrow{\tau}\triangleright\ s'$ for $\exists tl.\ s \xrightarrow{tl}\triangleright\ s' \wedge \tau\text{-}move\ s\ tl\ s'$. Moreover, $\_ \xrightarrow{\tau}\triangleright^+ \_$ denotes the transitive closure of $\_ \xrightarrow{\tau}\triangleright \_$, and $\_ \xrightarrow{\tau}\triangleright^* \_$ the reflexive and transitive closure. A state s can diverge (denoted $s \xrightarrow{\tau}\triangleright \infty$) iff an infinite sequence of τ-moves starts in s. A visible move $s \xrightarrow{tl}\blacktriangleright\ s'$ consists of a finite sequence of τ-moves followed by an observable transition, i.e., $s \xrightarrow{tl}\blacktriangleright\ s'$ abbreviates $\exists s''.\ s \xrightarrow{\tau}\triangleright^*\ s'' \wedge s'' \xrightarrow{tl}\triangleright\ s' \wedge \neg\ \tau\text{-}move\ s''\ tl\ s'$.

In this section, I often have states, labels, reductions, and the like for two or more programs and semantics. To keep the notation simple and clear, I usually index variables, arrows, etc. with numbers to assign them to one of them, i.e., $'x_1$, s_1, $t \vdash \_ -\_\rightarrow_1 \_$, etc. for the first, $'x_2$, s_2, $t \vdash \_ -\_\rightarrow_2 \_$, etc. for the second and so on.

type_synonym $('s,'t)\ lts = 's \Rightarrow 'tl \Rightarrow 's \Rightarrow bool$
type_synonym $('s_1,'s_2)\ bisim = 's_1 \Rightarrow 's_2 \Rightarrow bool$

locale *dbisim-base* =
fixes $\rightarrow\!\triangleright_1 :: ('s_1,'tl_1)\ lts$ and $\rightarrow\!\triangleright_2 :: ('s_2,'tl_2)\ lts$
and $\approx :: ('s_1,'s_2)\ bisim$ and $\sim :: ('tl_1,'tl_2)\ bisim$
and $\tau\text{-}move_1 :: ('s_1,'tl_1)\ lts$ and $\tau\text{-}move_2 :: ('s_2,'tl_2)\ lts$

locale *dbisim-div* = *dbisim-base* +
assumes *sim1*: $[\![s_1 \approx s_2;\ s_1 \xrightarrow{tl_1}\!\triangleright_1 s_1';\ \neg\ \tau\text{-}move_1\ s_1\ tl_1\ s_1']\!] \Longrightarrow \exists tl_2\ s_2'.\ s_2 \xrightarrow{tl_2}\!\blacktriangleright_2 s_2' \wedge s_1' \approx s_2' \wedge tl_1 \sim tl_2$
and *sim2*: $[\![s_1 \approx s_2;\ s_2 \xrightarrow{tl_2}\!\triangleright_2 s_2';\ \neg\ \tau\text{-}move_2\ s_2\ tl_2\ s_2']\!] \Longrightarrow \exists tl_1\ s_1'.\ s_1 \xrightarrow{tl_1}\!\blacktriangleright_1 s_1' \wedge s_1' \approx s_2' \wedge tl_1 \sim tl_2$
and *sim-τ1*: $[\![s_1 \approx s_2;\ s_1 \xrightarrow{\tau}\!\triangleright_1 s_1']\!] \Longrightarrow \exists s_2'.\ s_2 \xrightarrow{\tau}\!\triangleright_2^* s_2' \wedge s_1' \approx s_2'$
and *sim-τ2*: $[\![s_1 \approx s_2;\ s_2 \xrightarrow{\tau}\!\triangleright_2 s_2']\!] \Longrightarrow \exists s_1'.\ s_1 \xrightarrow{\tau}\!\triangleright_1^* s_1' \wedge s_1' \approx s_2'$
and *bisim-diverge*: $s_1 \approx s_2 \Longrightarrow s_1 \xrightarrow{\tau}\!\triangleright_1 \infty \longleftrightarrow s_2 \xrightarrow{\tau}\!\triangleright_2 \infty$

locale *dbisim-final* = *dbisim-base* + fixes $final_1 :: 's_1 \Rightarrow bool$ and $final_2 :: 's_2 \Rightarrow bool$
assumes $final_1$*-sim*: $[\![s_1 \approx s_2;\ final_1\ s_1]\!] \Longrightarrow \exists s_2'.\ s_2 \xrightarrow{\tau}\!\triangleright_2^* s_2' \wedge s_1 \approx s_2' \wedge final_2\ s_2'$
and $final_2$*-sim*: $[\![s_1 \approx s_2;\ final_2\ s_2]\!] \Longrightarrow \exists s_1'.\ s_1 \xrightarrow{\tau}\!\triangleright_1^* s_1' \wedge s_1' \approx s_2 \wedge final_1\ s_1'$

Fig. 30 Locale *dbisim-div* formalises the notion of delay bisimulations with explicit divergence; locale *dbisim-final* defines preservation of final states

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 29, which the locale *dbisim-div* in Figure 30 formalises:

(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 τ-move is simulated by a finite (possibly empty) sequence of τ-moves such that $\approx$ relates the resulting states.
(c) $\approx$ relates only states of which either both or none can diverge.

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 [54] 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. Condition (b) does not imply (c) because of the classic infinite stuttering problem. Infinitely many τ-moves may be simulated by no move at all.

Figure 31 on the right shows two LTSs with states $\{s_1, s_1', s_1''\}$ and $\{s_2, s_2', s_2''\}$ 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 with start state s_2 must decide immediately. Nevertheless, they are delay bisimilar with explicit divergence because divergence (Figure 29c) is a trace property and thus independent of the visited states.

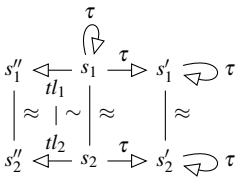


Fig. 31 Example of a delay bisimulation with explicit divergence that is not a well-founded delay bisimulation

A delay bisimulation $(\approx, \sim)$ preserves final states iff whenever one of the related states is final, then the other can silently reach a final state. The locale *dbisim-final* in Figure 30 formalises this notion and Figure 32 shows the simulation diagrams. 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 if the source code terminates in a final state, so does the compiled code, and vice versa.

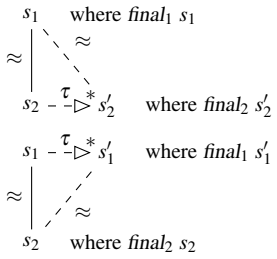
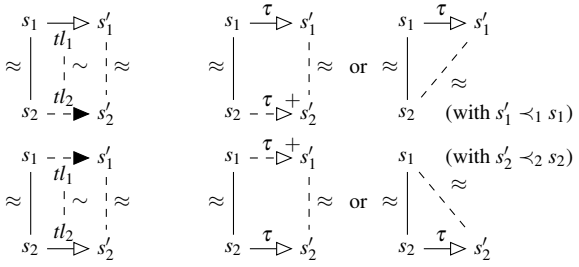


Fig. 32 Simulation diagrams for preservation of final states



(a) observable moves (b) τ-moves

Fig. 33 Simulation diagrams for well-founded delay bisimulations

Lemma 22 (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 transitivity for delay bisimulations without explicit divergence, i.e., the diagrams (a) and (b) of Figure 29. The diagrams in (c) are straightforward to prove. For preservation of final states, suppose $s_1 \approx_1 \circ \approx_2 s_3$ and $final_1\ s_1$. 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 $s_2 \xrightarrow{\tau}{\triangleright}_2^* s_2'$ and $s_1 \approx_1 s_2'$ and $final_2\ s_2'$. By induction on $s_2 \xrightarrow{\tau}{\triangleright}_2^* s_2'$ with invariant $s_2 \approx_2 s_3$, obtain s_3' such that $s_2' \approx s_3'$ and $s_3 \xrightarrow{\tau}{\triangleright}_3^* s_3'$, using *sim-τ1* in the inductive step. Since s_2' is $final_2$, there is an s_3'' with $s_3' \xrightarrow{\tau}{\triangleright}_3^* s_3''$ and $s_2' \approx s_3''$ and $final_3\ s_3''$. Then, $s_3 \xrightarrow{\tau}{\triangleright}_3^* s_3''$ and $s_1 \approx_1 \circ \approx_2 s_3''$ by transitivity and definition. This concludes this direction. The other direction follows by symmetry. □

Proofs by symmetry in this section are not only a matter of presentation. I appeal to symmetry also in the Isabelle formalisation. This way, only one direction needs a detailed proof. The usual trick of requiring that the bisimulation relation be symmetric cannot be used here because the state types of two transition systems differ. Instead, note that if $(\approx, \sim)$ is a delay bisimulation relation for $\rightarrow\!\triangleright_1$ and $\rightarrow\!\triangleright_2$, then $(flip \approx, flip \sim)$ is a delay bisimulation relation for $\rightarrow\!\triangleright_2$ and $\rightarrow\!\triangleright_1$, where $flip\ R\ x\ y = R\ y\ x$. Formally,

$$\frac{dbisim\text{-}div\ \rightarrow\!\triangleright_1\ \rightarrow\!\triangleright_2\ \approx\ \sim\ \tau\text{-}move_1\ \tau\text{-}move_2}{dbisim\text{-}div\ \rightarrow\!\triangleright_2\ \rightarrow\!\triangleright_1\ (flip \approx)\ (flip \sim)\ \tau\text{-}move_2\ \tau\text{-}move_1}$$

Similar relations hold for the other bisimulation locales in this section. Thus, it suffices to prove one of the diagrams in each of Figure 29a–c as a separate lemma. The other diagram then follows by instantiating the locale with the flipped transition systems using the rule.

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 [39] 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 33:

(a) Observable moves are simulated as in delay bisimulations with explicit divergence.
(b) A τ-move $s_i \xrightarrow{\tau}{\triangleright}_i s_i'$ ($i \in \{1, 2\}$) is either simulated by a finite *non-empty* sequence of τ-moves, or by no move at all. In the latter case, the τ-move being simulated must descend in $\prec_i$, i.e., $s_i' \prec_i s_i$.

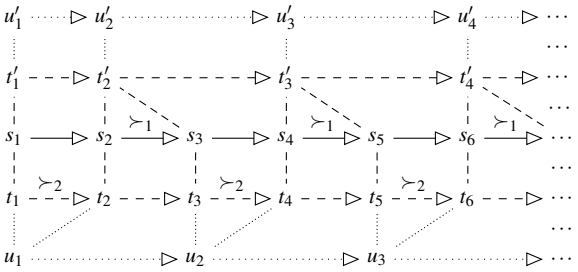


Fig. 34 Three transition systems (solid, dashed, and dotted arrows) and two well-founded delay bisimulations (dashed and dotted lines) whose composition is no well-founded delay bisimulation for the solid and dotted transition systems. All transitions are τ-moves.

Since $\prec_1$ and $\prec_2$ are well-founded, there are no infinitely decreasing chains and the infinite stuttering problem cannot occur. Proving well-founded delay bisimulation is easier than delay bisimulation with explicit divergence as all assumptions involve only a single transition.

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 23 *Let $(\approx,\sim)$ be a well-founded delay bisimulation for $\prec_1$ and $\prec_2$. Then, $(\approx,\sim)$ is a delay bisimulation with explicit divergence.*

Proof Since Figure 29a and Figure 33a are identical and Figure 33b trivially implies Figure 29b, only the simulation diagrams for divergence (Figure 29c) are interesting. The latter are shown by coinduction on $s_i \xrightarrow{\tau}\triangleright_i \infty$ ($i \in \{1,2\}$) with $s_1 \approx s_2$ and $s_{3-i} \xrightarrow{\tau}\triangleright_{3-i} \infty$ as coinduction invariant with a nested induction on the well-founded order $\prec_{3-i}$. □

The converse does not hold, as Figure 31 shows.[11] 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 τ-move $s_1 \xrightarrow{\tau}\triangleright_1 s_1$ according to Figure 33b. Clearly, $\approx'$ cannot relate s_1 and s'_2 because s_1 can do the observable move with label tl_1 and s'_2 cannot. This excludes the possibility on the left of Figure 33b. However, the right one is not feasible, either, as $s_1 \prec_1 s_1$ would violate well-foundedness of $\prec_1$.

The advantage of delay bisimulation with explicit divergence over well-founded delay bisimulation is that only the former is closed under composition (Lemma 22), but not the latter. Figure 34 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 $\prec'_1$ for the solid LTS. Suppose $\prec'_1$ were such. Then, for all $i > 0$, u_i can simulate the τ-move $s_{2i-1} \xrightarrow{\tau}\triangleright s_{2i}$ only by staying at u_i, hence $s_{2i} \prec'_1 s_{2i-1}$ by Figure 33b. But for all $j > 0$, u'_{j+1} is related to s_{2j} whose τ-move to s_{2j+1} it can simulate only by staying at u'_{j+1}. Hence, $s_{2j+1} \prec'_1 s_{2j}$ by Figure 33b. Therefore, $s_{i+1} \prec'_1 s_i$ for all i, i.e., $\prec'_1$ 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 bisimilarity is transitive as there may be delay bisimilarity relations other than the composition $\bigcirc_B$ for which a well-founded relation

[11] This example was found by Nitpick [10], a counter example generator for Isabelle/HOL, after several failed attempts of mine to prove equivalence).

exists. I have not attempted to prove this because directly composing bisimulation relations helps in breaking down the ultimate correctness result.

Semantic preservation needs a formal notion of execution. Here, an execution ξ 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 $\lfloor s \rfloor$ or the special symbol *None* (written $\xrightarrow{\infty}$) for divergence. That is, an execution is like a complete run (Section 2.2.4) in which all unobservable transitions have been removed. Formally, an execution is modelled by a terminated lazy list:

type_synonym $('s,'tl)$ *execution* $= ('tl,'s$ *option*$)$ *tllist*

An execution ξ terminates in the state s iff ξ is finite and the empty list constructor in ξ carries the symbol $\lfloor s \rfloor$.

The predicate $s \Downarrow\!\Downarrow \xi$ characterises all executions ξ that start in s. If s can reach via τ-moves a stuck state s', then an execution terminates in s' ($\Downarrow\!\Downarrow$STOP). If an infinite sequence of τ-moves starts in s, then an execution diverges ($\Downarrow\!\Downarrow$DIV). 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 ($\Downarrow\!\Downarrow$STEP).

$$\Downarrow\!\Downarrow\text{STOP}: \frac{s \xrightarrow{\tau\triangleright}{}^* s' \qquad \forall tl\ s''.\ \neg s' \xrightarrow{tl\triangleright} s''}{s \Downarrow\!\Downarrow [\,]_{\lfloor s' \rfloor}} \qquad \Downarrow\!\Downarrow\text{DIV}: \frac{s \xrightarrow{\tau\triangleright} \infty}{s \Downarrow\!\Downarrow [\,]^{\infty}_{\rightarrow}} \qquad \Downarrow\!\Downarrow\text{STEP}: \frac{s \xrightarrow{tl\blacktriangleright} s' \qquad s' \Downarrow\!\Downarrow \xi}{s \Downarrow\!\Downarrow tl \cdot \xi}$$

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 24 (Semantic preservation) *Let $(\approx,\sim)$ be a delay bisimulation with explicit divergence for $(\rightarrow\triangleright_1, \tau\text{-move}_1)$ and $(\rightarrow\triangleright_2, \tau\text{-move}_2)$ and $s_1 \approx s_2$. Then, the following holds:*

(i) Whenever $s_1 \Downarrow\!\Downarrow_1 \xi_1$, then $s_2 \Downarrow\!\Downarrow_2 \xi_2$ for some ξ_2 such that $\xi_1\ [\approx,\sim]\ \xi_2$.
(ii) Whenever $s_2 \Downarrow\!\Downarrow_2 \xi_2$, then $s_1 \Downarrow\!\Downarrow_1 \xi_1$ for some ξ_1 such that $\xi_1\ [\approx,\sim]\ \xi_2$.

Proof It suffices to prove (i) because (ii) follows from (i) by symmetry. Since ξ_2 is existentially quantified in (i), I must first construct it explicitly by corecursion from s_1 and ξ_1 before showing $s_2 \Downarrow\!\Downarrow_2 \xi_2$ and $\xi_1\ [\approx,\sim]\ \xi_2$ by coinduction. However, ξ_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 a coinduction invariant. Therefore, I define a variant $\Downarrow\!\Downarrow'$ of $\Downarrow\!\Downarrow$ where $\Downarrow\!\Downarrow$STEP not only prepends the transition label, but also remembers the intermediate state, i.e.,

$$\Downarrow\!\Downarrow'\text{STEP}: \frac{s \xrightarrow{tl\blacktriangleright} s' \qquad s' \Downarrow\!\Downarrow' \xi}{s \Downarrow\!\Downarrow' (tl,s') \cdot \xi}$$

Coinduction with invariant $s \Downarrow\!\Downarrow \xi_1$ shows that there is a ξ_1' such that $s \Downarrow\!\Downarrow_1' \xi_1'$ and ξ is the projection of ξ_1' on the transition labels. I construct $\xi_2' = \mathit{simulate}\ s_2\ \xi_1'$ by corecursion:

$\mathit{simulate}\ s_2\ [\,]^{\infty}_{\rightarrow} = [\,]^{\infty}_{\rightarrow}$
$\mathit{simulate}\ s_2\ [\,]_{\lfloor s_1' \rfloor} = (\mathit{let}\ s_2' = \varepsilon s_2'.\ s_2 \xrightarrow{\tau\triangleright}{}_2^* s_2' \wedge (\forall tl_2\ s_2''.\ \neg s_2' \xrightarrow{tl_2\triangleright}{}_2 s_2'') \wedge s_1' \approx s_2'\ \mathit{in}\ [\,]_{\lfloor s_2' \rfloor})$
$\mathit{simulate}\ s_2\ ((tl_1,s_1') \cdot \xi_1) =$
$\quad (\mathit{let}\ (tl_2,s_2') = \varepsilon(tl_2,s_2').\ s_2 \xrightarrow{tl_2\blacktriangleright}{}_2 s_2' \wedge s_1' \approx s_2' \wedge tl_1 \sim tl_2\ \mathit{in}\ (tl_2,s_2') \cdot \mathit{simulate}\ s_2'\ \xi_1)$

Coinduction with invariant $s_1 \approx s_2$ and $s_1 \Downarrow\!\Downarrow' \xi_1'$ proves that $s \Downarrow\!\Downarrow_2' \xi_2'$ and that ξ_1' and ξ_2' are related point-wise by $\approx$ and $\sim$. Then, the projection of ξ_2' to transition labels yields ξ_2. □

The characterisation of stuck states in $\downarrow\downarrow\text{STOP}$ explains the need for bisimulations. Indeed, each direction of semantic preservation needs both directions of the simulation diagrams. For example, let s_1 be a stuck state in $\multimap\!\triangleright_1$ and $s_2 \xrightarrow{\mathit{tl}} \multimap\!\triangleright_2 s_2'$ be an observable move and assume that $s_1 \approx s_2$. As there is nothing to simulate, $\approx$ trivially satisfies all the diagrams for $\multimap\!\triangleright_2$ simulating $\multimap\!\triangleright_1$ in Figure 29. Yet, $s_1 \downarrow\downarrow_1 []_{s_1}$, but not $s_2 \downarrow\downarrow_2 []_{s_2''}$ for any s_2''. So, semantic preservation is violated. The reason is that the definition of semantic preservation makes termination observable, but simulations do not talk about termination.

Theorem 25 (Preservation of final states) *Let $(\approx, \sim)$ preserve final states. Suppose $s_1 \downarrow\downarrow_1 \xi_1$ and $s_2 \downarrow\downarrow_2 \xi_2$ such that $\xi_1\ [\approx, \sim]\ \xi_2$. Then, ξ_1 terminates in a final$_1$-state iff ξ_2 terminates in a final$_2$-state.*

5.2.2 Lifting simulations in the interleaving framework

The delay bisimulations for showing semantic preservation always relate multithreaded states. As I use the language-independent 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, which I list below. Before that, I define the silent moves and the bisimulation relation for the multithreaded semantics.

A transition in the multithreaded semantics is a τ-move (predicated by *mτ-move*) iff it originates from a τ-move of the single-threaded semantics via NORMAL. In particular, reacquisition of temporarily released locks (rule ACQ) is no τ-move because another thread can observe the lock acquisition by no longer being able to acquire the lock.

Let $t \vdash (x_1, h_1) \approx (x_2, h_2)$ denote a bisimulation relation for thread t and $x_1 \approxeq x_2$ be a relation of thread-local states for threads in wait sets. The relation $s_1 \approx_{\text{m}} s_2$ on multithreaded states imposes the following conditions:

(i) Locks, wait setsm and interrupts are equal in s_1 and s_2, i.e., $\mathit{locks}\ s_1 = \mathit{locks}\ s_2$, and $\mathit{wset}\ s_1 = \mathit{wset}\ s_2$, and $\mathit{intrs}\ s_1 = \mathit{intrs}\ s_2$.
(ii) All threads in s_1 also exist in s_2 and vice versa, i.e., $\mathit{dom}\ (\mathit{tp}\ s_1) = \mathit{dom}\ (\mathit{tp}\ s_2)$.
(iii) For every thread in s_1 and s_2, say $\mathit{tp}\ s_1\ t = \lfloor (x_1, ln_1) \rfloor$ and $\mathit{tp}\ s_2\ t = \lfloor (x_2, ln_2) \rfloor$, the temporarily released locks are the same ($ln_1 = ln_2$), the thread-local states related ($t \vdash (x_1, \mathit{shr}\ s_1) \approx (x_2, \mathit{shr}\ s_2)$), and if t's wait set status is not *None* then $x_1 \approxeq x_2$.
(iv) All waiting threads exist, i.e., $\mathit{dom}\ (\mathit{wset}\ s_1) \subseteq \mathit{dom}\ (\mathit{tp}\ s_1)$.
(v) There are only finitely many threads, i.e., $\mathit{finite}\ (\mathit{dom}\ (\mathit{tp}\ s_1))$.

Since threads can observe the status of locks, wait sets, interrupts, and the existence of threads, conditions (i) and (ii) ensure that $\approx_{\text{m}}$-related states are indistinguishable in these respects to the threads. Constraint (iii) imposes the thread-wise bisimulation on all thread states. The last condition $x_1 \approxeq x_2$ imposes stronger simulation properties on threads in wait sets because the interleaving semantics does not allow a thread to execute τ-moves between its removal from the wait set and the (observable) reduction of processing the removal. Constraint (iv) ensures that spawned threads are not in a wait set, i.e., their thread-local states need not be related in $\approxeq$. The last constraint (v) ensures that $\approx_{\text{m}}$ preserves divergence.

To see that (v) is necessary, consider two pools with infinitely many threads. In one of them, each thread only does a single τ-move $x_1 \xrightarrow{\tau} \triangleright_1 x_1'$ 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 thread pool can diverge by executing one thread at a time, whereas the second is stuck. Hence, $\approx_{\text{m}}$ without (v) would not be a delay bisimulation with explicit divergence.

type_synonym $('l,'t,'x,'h,'w,'o)\ \tau\text{-move} = 'x \times 'h \Rightarrow ('l,'t,'x,'h,'w,'o)\ \mathit{thread\text{-}action} \Rightarrow 'x \times 'h \Rightarrow \mathit{bool}$

locale $\tau\text{-}\mathit{multithreaded} = \mathit{multithreaded} +$ fixes $\tau\text{-}\mathit{move} :: ('l,'t,'x,'h,'w,'o)\ \tau\text{-}\mathit{move}$
  assumes $\tau\text{-}ta$: $\tau\text{-}\mathit{move}\ (x,h)\ ta\ (x',h') \Longrightarrow ta = (\!|\!|)$
  and $\tau\text{-}\mathit{heap}$: $[\![t \vdash (x,h) -ta\rightarrow (x',h');\ \tau\text{-}\mathit{move}\ (x,h)\ ta\ (x',h')]\!] \Longrightarrow h' = h$

locale *m-dbisim-div* =
  $r1 : \tau\text{-}\mathit{multithreaded}\ \mathit{final}_1\ r_1\ \mathit{acq\text{-}events}\ \tau\text{-}\mathit{move}_1 +$
  $r2 : \tau\text{-}\mathit{multithreaded}\ \mathit{final}_2\ r_2\ \mathit{acq\text{-}events}\ \tau\text{-}\mathit{move}_2 +$
  fixes $\_ \vdash \_ \approx \_ :: 't \Rightarrow ('x_1 \times 'h_1, 'x_2 \times 'h_2)\ \mathit{bisim}$ and $\approx\!\!\!\!\approx :: ('x_1,'x_2)\ \mathit{bisim}$
  assumes $\forall t.\ \mathit{dbisim\text{-}div}\ (r_1\ t)\ (r_2\ t)\ (t \vdash \_ \approx \_) \sim \tau\text{-}\mathit{move}_1\ \tau\text{-}\mathit{move}_2$
  and $\forall t.\ \mathit{dbisim\text{-}final}\ (r_1\ t)\ (r_2\ t)\ (t \vdash \_ \approx \_) \sim \tau\text{-}\mathit{move}_1\ \tau\text{-}\mathit{move}_2\ (\lambda(x_1,h_1).\ \mathit{final}_1\ x_1)\ (\lambda(x_2,h_2).\ \mathit{final}_2\ x_2)$
  and *heap-change*:
    $[\![t \vdash (x_1,h_1) \approx (x_2,h_2);\ t \vdash (x_1',h_1') \approx (x_2',h_2');\ ta_1 \sim ta_2;$
    $t \vdash (x_1,h_1) \xrightarrow{ta_1}_1 (x_1',h_1');\ t \vdash (x_2,h_2) \xrightarrow{ta_2}_2 (x_2',h_2');$
    $t' \vdash (x_1^*,h_1) \approx (x_2^*,h_2)]\!] \Longrightarrow t' \vdash (x_1^*,h_1') \approx (x_2^*,h_2')$
  and $\approx\!\!\!\!\approx I$:
    $[\![t \vdash (x_1,h_1) \approx (x_2,h_2);\ t \vdash (x_1',h_1') \approx (x_2',h_2');\ ta_1 \sim ta_2;$
    $t \vdash (x_1,h_1) \xrightarrow{ta_1}_1 (x_1',h_1');\ t \vdash (x_2,h_2) \xrightarrow{ta_2}_2 (x_2',h_2');$
    $\mathit{Suspend}\ w \in \mathit{set}\langle ta_1\rangle_{\mathrm{w}};\ \mathit{Suspend}\ w \in \mathit{set}\langle ta_2\rangle_{\mathrm{w}}]\!] \Longrightarrow (x_1',h_1') \approx\!\!\!\!\approx (x_2',h_2')$
  and $\mathit{sim}\text{-}\approx\!\!\!\!\approx 1$:
    $[\![t \vdash (x_1,h_1) \approx (x_2,h_2);\ x_1 \approx\!\!\!\!\approx x_2;\ t \vdash (x_1,h_1) -ta_1\rightarrow_1 (x_1',h_1');$
    $\mathit{Notified} \in \mathit{set}\ \langle ta_1\rangle_{\mathrm{w}} \vee \mathit{WokenUp} \in \mathit{set}\ \langle ta_1\rangle_{\mathrm{w}}]\!]$
    $\Longrightarrow \exists ta_2\ x_2'\ h_2'.\ t \vdash (x_2,h_2) -ta_2\rightarrow_2 (x_2',h_2') \wedge t \vdash (x_1',h_1') \approx (x_2',h_2') \wedge ta_1 \sim ta_2$
  and $\mathit{sim}\text{-}\approx\!\!\!\!\approx 2$:
    $[\![t \vdash (x_1,h_1) \approx (x_2,h_2);\ x_1 \approx\!\!\!\!\approx x_2;\ t \vdash (x_2,h_2) -ta_2\rightarrow_2 (x_2',h_2');$
    $\mathit{Notified} \in \mathit{set}\ \langle ta_2\rangle_{\mathrm{w}} \vee \mathit{WokenUp} \in \mathit{set}\ \langle ta_2\rangle_{\mathrm{w}}]\!]$
    $\Longrightarrow \exists ta_1\ x_1'\ h_1'.\ t \vdash (x_1,h_1) -ta_1\rightarrow_1 (x_1',h_1') \wedge t \vdash (x_1',h_1') \approx (x_2',h_2') \wedge ta_1 \sim ta_2$
  and *Ex-final-inv*: $(\exists x_1.\ \mathit{final}_1\ x_1) \longleftrightarrow (\exists x_2.\ \mathit{final}_2\ x_2)$

Fig. 35 Locale τ-*multithreaded* enforces that τ-moves are unobservable and locale *m-dbisim-div* collects the necessary assumptions for $\approx_{\mathrm{m}}$ being a delay bisimulation with explicit divergence.

The bisimulation $\_ \vdash \_ \approx \_$ for single threads also yields the relation on the thread actions as transition labels: $ta_1 \sim ta_2$ 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 $t \vdash (x_1,h_1) \approx (x_2,h_2)$, i.e, ta_1 and ta_2 may only differ in the initial states of spawned threads, which must be bisimilar. Note that storing the heap in the BTA *Spawn* again simplifies the definition because $t \vdash \_ \approx \_$ relates pairs of thread-local states and the current heaps.

The above definition for $\approx_{\mathrm{m}}$ is sensible. If $(t \vdash \_ \approx \_, \sim)$ is a delay bisimulation with explicit divergence, then so is $\approx_{\mathrm{m}}$ (Theorem 26), if threads communicate only through thread actions and the shared heap. Figure 35 formalises the communication restrictions as follows:

1. Most importantly, a thread must not be able to observe the τ-moves of other threads. To that end, I demand that τ-moves neither execute any BTAs, nor change the shared heap as expressed by the locale τ-*multithreaded*.
2. $t \vdash \_ \approx \_$ 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, i.e., t''s local state is final, then any $\approx_{\mathrm{m}}$-bisimilar state s_2 can reach via τ-moves a bisimilar state s_2' in which t''s local state is also final, i.e., t's join suceeds in s_2', too.
3. Since bisimilarity of threads involves the shared heap, I require that the heap changes by one thread preserve bisimilarity of other threads. Assumption *heap-change* of locale *m-dbisim-div* captures this formally: Let $t \vdash (x_1,h_1) \approx (x_2,h_2)$ be two bisimilar states with visible moves to (x_1',h_1') and (x_2',h_2') such that $t \vdash (x_1',h_1') \approx (x_2',h_2')$, 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, $t' \vdash (x_1^*,h_1') \approx (x_2^*,h_2')$ holds for the new heaps h_1' and h_2', too.

4. The wait-notify mechanism requires that when a thread has been removed from its wait set, its next step processes the removal. To that end, $\approx_m$ enforces that threads in wait sets are related in $\lessapprox$. The next three assumptions link $\lessapprox$ with the semantics. Assumption $\lessapprox I$ expresses that whenever a thread suspends itself to a wait set, $\lessapprox$ must relate the resulting states. Moreover (*sim-*$\lessapprox$*1* and *sim-*$\lessapprox$*2*), $\lessapprox$ is a "one-step" strong bisimulation for processing the removal from the wait set, i.e., whenever one of the states can process a removal, so can the other without any intervening τ-moves and $t \vdash \_ \approx \_$ relates new states.
5. Last, for technical reasons, *m-dbisim-div* requires that final states exist for either both single-threaded semantics or none. I discuss this assumption in more detail in Footnote 12 when I prove preservation for deadlocks (Theorem 27).

Then, $\approx_m$ is a delay bisimulation with explicit divergence that preserves final states.

Theorem 26 *Under the assumptions of locale* m-dbisim-div, $(\approx_m, \sim_m)$ *is a delay bisimulation for* r1.redT *and* r2.redT *that preserves final states. Formally:*

(i) *dbisim-div r1.redT r2.redT* $\approx_m \sim_m$ *r1.m*τ*-move r2.m*τ*-move*

(ii) *dbisim-final r1.redT r2.redT* $\approx_m \sim_m$ *r1.m*τ*-move r2.m*τ*-move r1.mfinal r2.mfinal*

Proof I show the simulation diagrams from Figures 29 and 32 for (i) and (ii), respectively. As before, I show just one direction of each diagram as the other follows by symmetry.

For an observable move $s_1 -t{:}ta_1 \rightarrow_1 s_1'$ and $\approx_m$-bisimilar state s_2 (Figure 29a), if the move originates from ACQ, 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 are the same in s_1 and s_2. Otherwise (NORMAL), the move originates from some observable move of t's semantics, say $t \vdash (x_1, shr\, s_1) -ta_1 \rightarrow_1 (x_1', shr\, s_1')$. 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 *None*—$x_1 \lessapprox x_2$. If t's wait set status is *None*, bisimilarity of single threads yields a visible move $t \vdash (x_2, shr\, s_2) \xrightarrow{ta_2}_2 (x_2', h_2')$ such that $t \vdash (x_1', shr\, s_1') \approx (x_2', h_2')$ and $ta_1 \sim ta_2$. 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 detail, if ta_2 joins on a thread t', i.e., $Join\, t' \in \langle ta_2 \rangle_c$, t' may be final in s_1, but need not be final in s_2. As $t' \vdash \_ \approx \_$ preserves final states, t' can silently reduce to a final state. Hence, the simulating visible move consists of (i) t's τ-moves, (ii) the silent reductions to final states of all threads ta_2 joins on, and (iii) 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 τ-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 where assumption $\lessapprox I$ establishes $x_1' \lessapprox x_2'$ if $wset\, s_1'\, t \neq None$, i.e., when ta_1 and ta_2 contain a *Suspend* BTA. Third, $ta_1 \sim ta_2$ guarantees that the local states of spawned threads are related; and $dom\, (wset\, s_1) \subseteq dom\, (tp\, s_1)$ ensures that their wait set status is *None*, so $\lessapprox$ need not relate their local states. Forth, by assumption *heap-change*, all other threads remain bisimilar. Fifth, the thread pool remains finite as a single step can spwan only finitely many threads.

The case for $wset\, s_1\, t \neq None$ is analogous except that assumption *sim-*$\lessapprox$*1* yields the simulating move and t does no τ-moves before the observable move.

Simulating a τ-move (Figure 29b) is easy. It must originate from a τ-move in the single-threaded semantics, so there is a simulating sequence of τ-moves. Since none of them generates any thread action, NORMAL simply 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, recall that all threads in an *mfinal* state are final. So suppose $r1.mfinal\ s_1$ and $s_1 \approx_m s_2$. By induction on the finite set of threads in s_1, the state s_2 can silently reach a state s_2' with $s_1 \approx_m s_2'$ and $r2.mfinal\ s_2'$. In the inductive step, the next thread t can silently reduce to an appropriate final state as $t \vdash \_ \approx \_$ preserves final states. These τ-moves together with the induction hypothesis yield the desired moves. □

Theorem 27 (Preservation of deadlocks) *Under the assumptions of locale* m-dbisim-div, $\approx_m$ *preserves deadlocks. Let* $s_1 \approx_m s_2$. *If* $t \in r1.deadlocked\ s_1$, *then* s_2 *can reduce via* τ*-moves to some* s_2' *such that* $s_1 \approx_m s_2'$ *and* $t \in r2.deadlocked\ s_2'$. *If* $t \in r2.deadlocked\ s_2$, *then* s_1 *can reduce via* τ*-moves to some* s_1' *such that* $s_1' \approx_m s_2$ *and* $t \in r1.deadlocked\ s_1'$.

Proof By symmetry, it suffices to prove only one direction. So suppose $t \in r1.deadlocked\ s_1$. Since $\_ \vdash \_ \approx \_$ preserves final states, there is an s_2^* which is reachable via τ-moves from s_2 such that $s_1 \approx_m s_2^*$ and all $final_1$ threads in s_1 are $final_2$ in s_2^*, too (by induction on the finite set of threads). Since $\_ \vdash \_ \approx \_$ preserves divergence, a similar induction shows that there is an s_2' which is reachable via τ-moves from s_2^* such that $s_1 \approx_m s_2'$ and all threads in s_1 that cannot do any τ-move cannot do any in s_2' either.

Then, I prove $t \in r2.deadlocked\ s_2'$ by coinduction with invariant $t \in r1.deadlocked\ s_1$. In the coinductive step, I show by case analysis of $t \in r1.deadlocked\ s_1$ that each case suffices to prove the corresponding case for $t \in r2.deadlocked\ s_2'$. For DACTIVE, this follows from the simulation for observable moves. Since t is deadlocked in s_1', it cannot do any τ-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 τ-moves. Therefore, $\approx_m$ preserves $t \vdash (\_, shr\ s)\ \wr$ and $t \vdash (\_, shr\ s)\ \_\wr$.[12] 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'. □

5.2.3 Semantic preservation for the Java memory model

Bisimulations ensure that both the source program and the compiled code have the same set of traces (Theorem 24). Since τ-moves produce no memory model events by assumption *τ-ta*, the JMM cannot distinguish the sets of complete runs, either. Thus, semantic preservation extends to the JMM:

Corollary 28 *Two programs that are delay bisimilar with explicit divergence have the same set of legal executions.*

5.3 Explicit call stacks for source code

The small-step semantics for source code dynamically inlines method calls (RCALL) whereas the VM models a call stack. To ease the compiler verification, I first define an alternative state representation and small-step semantics J_0 with explicit call stacks for the source code language. This change only affects the semantics, not the language. Then, after having defined the observable moves, I prove that both semantics are delay bisimilar.

[12] Recall that $t \vdash (x,h)\ \wr$ expresses that t with local state x can reduce with a thread action ta which is not contradictory in itself. This is where the technical assumption *Ex-final-inv* from Figure 35 becomes relevant for preservation. It excludes the case in which some x_1 satisfies $final_1$, but $final_2$ is unsatisfiable. Hence, $\_ \vdash \_\wr_1$ allows *Join* BTAs in the underlying execution, but $\_ \vdash \_\wr_2$ does not, which breaks semantic preservation. The other assumptions do not exclude this case: Only preservation of final states involves *final*, but if x_1 is unreachable, $\_ \vdash \_ \approx \_$ need not relate x_1 to any x_2, i.e., preservation of final states would be trivially satisfied.

RED_0: $$\frac{P,t \vdash \langle e,(h,\mathit{empty})\rangle -ta\rightarrow \langle e',(h',x')\rangle \qquad \mathit{no\text{-}call}\ P\ h\ e}{P,t \vdash \langle e,h\rangle -ta\rightarrow^{\mathrm{e}}_0 \langle e',h'\rangle}$$

R_0RED: $$\frac{P,t \vdash \langle e,h\rangle -ta\rightarrow^{\mathrm{e}}_0 \langle e',h'\rangle}{P,t \vdash \langle (e,es),h\rangle -ta\rightarrow^{\mathrm{t}}_0 \langle (e',es),h'\rangle}$$

R_0CALL: $$\frac{\mathit{call}\ e = \lfloor (a,M,vs)\rfloor \qquad \mathit{typeof\text{-}addr}\ h\ a = \lfloor T\rfloor \qquad P \vdash \mathit{class\text{-}of}'\ T\ \mathit{sees}\ M{:}Ts\rightarrow T_r = \lfloor (pns,body)\rfloor\ \mathit{in}\ D \qquad |vs| = |pns| \qquad |Ts| = |pns|}{P,t \vdash \langle e\cdot es,h\rangle -\langle\!|\!|\rightarrow^{\mathrm{t}}_0 \langle \mathit{blocks}\ (\mathit{this}\cdot pns)\ (\mathit{Class}\ D\cdot Ts)\ (\mathit{Addr}\ a\cdot vs)\ body\cdot e\cdot es,h\rangle}$$

R_0RET: $$\frac{\mathit{final}\ e}{P,t \vdash \langle e\cdot e'\cdot es,h\rangle -\langle\!|\!|\rightarrow^{\mathrm{t}}_0 \langle \mathit{inline}\ e\ e'\cdot es,h\rangle}$$

Fig. 36 Single-threaded source code semantics with explicit call stacks

5.3.1 State and semantics

On the call-stack level, local stores are irrelevant for the semantics as free variables can be bound by additional blocks. The rule for method calls RCALL and the start state *J-start* already do so for the *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.

Accordingly, 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 multithreaded programs as before.

To separate method calls and returns from the rest of the semantics, I introduce two auxiliary functions. First, the partial function *call e* returns $\lfloor (a,M,vs)\rfloor$ 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 $\mathit{call}\ e = \lfloor (a,M,vs)\rfloor$. This function *call e* traverses the expression e following the evaluation order of the small-step semantics. When it finds a call of the form $\mathit{addr}\ a.M(\mathit{map}\ \mathit{Val}\ vs)$, it returns $\lfloor (a,M,vs)\rfloor$; otherwise, it descends into the subexpression that the small-step semantics evaluates next; if there is none, it returns *None*. The other function $\mathit{inline}\ e_0\ e$ mimicks the small-step semantics' dynamic inlining of method calls. If e pauses at a call, $\mathit{inline}\ e_0\ e$ replaces this call with e_0. If not, it returns e.

The expression level semantics red_0 (notation $P,t \vdash \langle e,h\rangle -ta\rightarrow^{\mathrm{e}}_0 \langle e',h'\rangle$) is the same as in J except for calling (rule RED_0 in Figure 36).[13] To avoid redundancies, I do not define a new small-step semantics, but use the predicate *no-call P h e* to filter out all reductions due to RCALL from $P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow \langle e',(h',x')\rangle$. The predicate *no-call P h e* holds iff whenever e pauses at a call then the called method must be native. Note that red_0 discards the new local store x'; well-formedness ensures that x' is always *empty* (Corollary 32).

Figure 36 also shows the small-step semantics J_0-*red* for the call-stack level (notation $P,t \vdash \langle e\cdot es,h\rangle -ta\rightarrow^{\mathrm{t}}_0 \langle e'\cdot es',h'\rangle$). It consists of all reductions of the expression level semantics red_0 for the top of the call stack (R_0RED). Additionally, it reintroduces the reductions for method calls that red_0 has filtered out (R_0CALL). Rather than dynamically inlining the method body, R_0CALL pushes 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_0RET 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 a call. R_0RET has no counterpart in $\_,\_ \vdash \langle \_,\_\rangle -\_\rightarrow \langle \_,\_\rangle$ as dynamic inlining turns returns into no-ops.

[13] The thread-local state in *Spawn* actions in *ta* must be adapted, too, but I omit this technical detail.

A state $e \cdot es$ in J_0 is final, written $final_0\ (e \cdot es)$, iff $final\ e$ and $es = []$. The start state J_0-$start\ P\ C\ M\ vs$ has one thread $start\text{-}tID$ with local state

$$blocks\ (this \cdot pns)\ (Class\ D \cdot Ts)\ (Null \cdot vs)\ body \cdot []$$

where $P \vdash C\ sees\ M{:}Ts \rightarrow \_ = \lfloor (pns, body) \rfloor\ in\ D$.

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.

The following operations are observable: memory allocation, calls to native methods other than $currentThread$, access and assignment to fields and array cells, reading array lengths, and synchronisation. In particular, thread spawns, joining, interruption and the wait-notify mechanism are observable. Conversely, all control flow constructs, including exception throwing and handling, and local variable manipulations are only relevant to the thread that executes them, so these generate only τ-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 $ArrayIndexOutOfBounds$ exception. Thus, J and J_0 also treat throwing the $ArrayIndexOutOfBounds$ exception as an observable move. Fortunately, a larger set of observable moves only strengthens the correctness result. Formally, I define a predicate $\tau\text{-}move :: 'm\ prog \Rightarrow 'heap \Rightarrow 'addr\ expr \Rightarrow bool$ that identifies states in which τ-moves originate. Then, J-τ-$move$ and J_0-τ-$move$ determine the τ-moves for J and J_0, respectively, where

$$\begin{aligned} &J\text{-}\tau\text{-}move\ \ P\ ((e,x),h) \quad ta\ \_ \longleftrightarrow \tau\text{-}move\ P\ h\ e \wedge ta = \langle\!|\!| \\ &J_0\text{-}\tau\text{-}move\ P\ (e_0 \cdot es_0, h_0)\ ta\ \_ \longleftrightarrow (\tau\text{-}move\ P\ h_0\ e_0 \vee final\ e_0) \wedge ta = \langle\!|\!| \end{aligned}$$

Lemma 29 *J and J_0 satisfy the assumptions of locale τ-multithreaded.*

5.3.2 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 $V \in fv\ e$) iff e contains a subexpression $Var\ V$ that is not contained in a local-variable or catch block that declares V. An expression e is closed iff it contains no free variables, i.e., $fv\ e = \emptyset$. A call stack of expressions $e_0 \cdot es_0$ is well-formed (notation $wf_0\ (e_0 \cdot es_0)$) iff e_0 is closed, and all expressions in es_0 are closed and pause at a call. Formally:

$$wf_0\ (e_0 \cdot es_0) \longleftrightarrow fv\ e_0 = \emptyset \wedge (\forall e \in set\ es_0.\ fv\ e = \emptyset \wedge call\ e \neq None)$$

Closedness rules out references to global variables.[14] Hence, it is irrelevant that J_0 executes method bodies in an empty local store (R_0CALL) while inlining method calls executes the body in the local store of the caller (RCALL).

[14] Already in Jinja, closedness was crucial for the small-step semantics and its equivalence to the big-step semantics [32, §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." [32, §2.3.2].

Let *collapse* $(e_0 \cdot es_0)$ abbreviate the expression to which inlining collapses the call stack, i.e., *collapse* $(e_0 \cdot es_0) =$ *foldl inline* e_0 es_0. Then, the delay bisimulation relation $\overset{t}{\approx}_0$ for single threads relates the J state $((e,x),h)$ with $(e_0 \cdot es_0, h_0)$ iff x is empty, the heaps are the same, $e_0 \cdot es_0$ is well-formed, and $e_0 \cdot es_0$ collapses to e. Formally:

$$\frac{wf_0\ (e_0 \cdot es_0)}{((\mathit{collapse}\ (e_0 \cdot es_0), \mathit{empty}), h) \overset{t}{\approx}_0 (e_0 \cdot es_0, h)}$$

Since both $J.redT$ and $J_0.redT$ are instances of the interleaving semantics, I lift $\overset{t}{\approx}_0$ to multithreaded states as described in Section 5.2.2. Let $\overset{m}{\approx}_0$ and $\sim_0$ denote the corresponding instance of $\approx_m$ and $\sim$ where $(e,x) \approx\!\!\!\approx e_0 \cdot es_0$ iff $\neg$ *final* e_0.

Theorem 30 *If wf-J-prog P, then* $(\overset{t}{\approx}_0, \sim_0)$ *is a delay bisimulation with explicit divergence for J-red P t and* J_0*-red P t which preserves final states.*

The proof requires a number of lemmas first.

Lemma 31 *If wf-J-prog P and* $P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow \langle e',(h',x')\rangle$*, then fv* $e' \subseteq$ *fv e and dom* $x' \subseteq$ *dom* $x \cup$ *fv e*

Corollary 32 *If wf-J-prog P and* $P,t \vdash \langle e,(h,\mathit{empty})\rangle -ta\rightarrow \langle e',(h',x')\rangle$ *and e is closed, then* $x' = \mathit{empty}$*.*

Lemma 33 *If wf-J-prog P, then* J_0*-red P t preserves well-formedness.*

A call stack $e_0 \cdot es_0$ is normalised iff $\neg$ *final* e_0 or $es_0 = []$, i.e., R_0RET does not apply. The next lemma shows that J_0 can silently normalise call stacks; it is proven by induction on es_0.

Lemma 34 *For every call stack* $e_0 \cdot es_0$*, there is a normalised call stack* $e_0' \cdot es_0'$ *such that collapse* $(e_0 \cdot es_0) =$ *collapse* $(e_0' \cdot es_0')$ *and* $e_0 \cdot es_0$ *silently reduces to* $e_0' \cdot es_0'$*.*

A normalised call stack $e_0 \cdot es_0$ simulates the collapsed call stack *collapse* $(e_0 \cdot es_0)$ directly, i.e., without any additional τ-moves.

Lemma 35 *Let* (e_0, es_0) *be well-formed and normalised.*

(i) If $P,t \vdash \langle e_0,(h,\mathit{empty})\rangle -ta\rightarrow \langle e_0',(h',\mathit{empty})\rangle$*, then*
$P,t \vdash \langle \mathit{collapse}\ (e_0 \cdot es_0),(h,\mathit{empty})\rangle -ta\rightarrow \langle \mathit{collapse}\ (e_0' \cdot es_0),(h',\mathit{empty})\rangle$
(ii) If $P,t \vdash \langle \mathit{collapse}\ (e_0,es_0),(h,\mathit{empty})\rangle -ta\rightarrow \langle e',(h',\mathit{empty})\rangle$*, then* e' *is of the form collapse* (e_0',es_0) *and* $P,t \vdash \langle e_0,(h,\mathit{empty})\rangle -ta\rightarrow \langle e_0',(h',\mathit{empty})\rangle$*.*

Proof Note that inlining the top of the call stack preserves well-formedness. If *final* e_0, then $es_0 = []$ by normalisation and the lemma holds trivially. So suppose $\neg$ *final* e_0. Then, each direction follows by induction on es_0 from the following generalised one-step versions where e is arbitrary with *call* $e \neq$ *None*:

(i) If $P,t \vdash \langle e_0,(h,\mathit{empty})\rangle -ta\rightarrow \langle e_0',(h',\mathit{empty})\rangle$, then
$P,t \vdash \langle \mathit{inline}\ e_0\ e,(h,x)\rangle -ta\rightarrow \langle \mathit{inline}\ e_0'\ e,(h',x)\rangle$.
(ii) If $P,t \vdash \langle \mathit{inline}\ e_0\ e,(h,x)\rangle -ta\rightarrow \langle e',(h',x')\rangle$, then $x = x'$ and $e' = \mathit{inline}\ e_0'\ e$ for some e_0' such that $P,t \vdash \langle e_0,(h,\mathit{empty})\rangle -ta\rightarrow \langle e_0',(h',\mathit{empty})\rangle$.

Both of them are proved by induction on e. The only interesting case is when e is the call that *inline* replaces with e_0. Then, the local store in e_0's reduction must change from x to *empty* or vice versa. This follows from the next two easy lemmas (provable by induction) that Jinja uses to prove the big-step and small-step semantics equivalent [32, §2.5]:

(i) If $P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow \langle e',(h',x')\rangle$, then $P,t \vdash \langle e,(h,x_0 +\!\!+ x)\rangle -ta\rightarrow \langle e',(h',x_0 +\!\!+ x')\rangle$.
(ii) If $P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow \langle e',(h',x')\rangle$ and $fv\ e \subseteq W$,
then $P,t \vdash \langle e,(h,x\restriction_W)\rangle -ta\rightarrow \langle e',(h',x'\restriction_W)\rangle$,

where $f\restriction_A$ restricts the map f to A, i.e., $f\restriction_A = (\lambda x.\ \mathit{if}\ x \in A\ \mathit{then}\ f\ x\ \mathit{else}\ \mathit{None})$, and $f +\!\!+ g$ combines two maps: $f +\!\!+ g = (\lambda x.\ \mathit{case}\ g\ x\ \mathit{of}\ \mathit{None} \Rightarrow f\ x \mid \lfloor y\rfloor \Rightarrow \lfloor y\rfloor)$. □

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 36 *Let call* $e = \lfloor(a,M,vs)\rfloor$ *and* typeof-addr $h\ a = \lfloor T\rfloor$ *and* $P \vdash$ class-of$'$ T *sees* $M{:}Ts\rightarrow T_r = \lfloor(pns,body)\rfloor$ *in* D *and* $blks =$ blocks $(\mathit{this}\cdot pns)\ (\mathit{Class}\ D\cdot Ts)\ (\mathit{Addr}\ a\cdot vs)$ *body.*

(i) If $P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow \langle e',(h',x')\rangle$, *then* $e' =$ inline *blks e.*
(ii) If $|vs| = |pns|$ *and* $|Ts| = |pns|$, *then* $P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow \langle$inline *blks* $e,(h',x')\rangle$.

Now, I am ready to prove Theorem 30.

Proof (Theorem 30) By Lemma 23, it suffices to find two measures $\prec_1$ and $\prec_2$ for which $(\overset{t}{\approx}_0, \sim_0)$ is a well-founded delay bisimulation. Choose $\_ \not\prec_1 \_$, and $(e_0\cdot es_0,h_0) \prec_2 (e_0'\cdot es_0',h_0')$ iff $|es_0| < |es_0'|$, i.e., only returns R_0RET from method calls (when the call stack shrinks) need not have a counterpart in J.

For the simulation diagrams from Figure 33, I distinguish three cases:

1. Calls of non-native methods. For J_0 simulating J, it first normalises the call stack (Lemma 34). Then, Lemma 35 shows that the top call frame of the normalised call stack can reduce using the call, and Lemma 36 decomposes the resulting expression as necessary for the simulation with R_0CALL. For the other direction, Lemma 36 shows that the expression in the top call frame could also inline the call and so can the collapsed call stack by Lemma 35.
2. Returns from a method call (R_0RET) 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 35 shows that collapsing the call stack does not change the semantics. As for calls to non-native methods, J_0 first normalises the call stack.

In all cases, the new call stack is well-formed by Lemma 33. Preservation of final states is straightforward. □

Theorem 37 $(\overset{m}{\approx}_0, \sim_0)$ *is a delay bisimulation with explicit divergence for* J.redT P *and* J_0.redT P *that preserves final states.*

Proof By Theorem 26, it suffices to discharge the assumptions of locale *m-dbisim-div*. Theorem 30 discharges the inherited locales. As $\overset{t}{\approx}_0$ does not depend on the heap, *heap-change* holds trivially. For $\approx I$, case analysis and induction show that whenever J_0*-red* generates a *Suspend* BTA, the top call frame is not *final*. Since $\approx$ guarantees that the J_0 call stack is normalised, *sim-*$\approx$*1* follows easily because normalised call stacks simulate J's reductions without any additional τ-moves. Similarly, *sim-*$\approx$*2* holds because J simulates observable moves of J_0 without τ-moves. Finally, *Ex-final-inv* holds trivially. □

Finally, the bisimulation relation $\overset{m}{\approx}_0$ contains the well-formed start state.

Lemma 38 *If wf-J-prog P and wf-start P C M vs, then J-start P C M vs* $\overset{m}{\approx}_0$ *J*$_0$*-start P C M vs.*

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 (Section 5.4.1), the first compilation stage (Section 5.4.3), and the proof of its correctness (Sections 5.4.4 and 5.4.5).

5.4.1 Syntax of the intermediate language J_1

The intermediate language J_1 retains the expressions from source code, but stores local variable values in an array of registers similar to bytecode. 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 [32, §5.1] analogous to what *J* does in source code.

To avoid duplication, JinjaThreads parametrises the type of expressions $('a,'b,'addr)\ exp$ not only over the type of addresses $'addr$, but also over the variable names $'a$ and an annotation $'b$ for *sync* $(\_)\ \_$ and *insync* $(\_)\ \_$ blocks. $'addr\ expr$ abbreviates $(vname, unit, 'addr)\ exp$ where *unit* is the singleton type with value $()$. Expressions in J_1 are of type $'addr\ expr_1 = (nat, nat, 'addr)\ exp$, i.e., variable names are natural numbers and $sync_i\ (\_)\ \_$ and $insync_i\ (\_)\ \_$ blocks are now annotated with $i :: nat$.[15] Following the JVMS [42, §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 J_1 programs $'addr\ J_1\text{-}prog$ is $'addr\ expr_1\ prog$. Methods no longer declare parameter names because they have been replaced by indices.

J_1 requires a very specific layout of the registers, which $compP_1$ ensures. Register 0 holds the *this* pointer, the parameters occupy the registers 1 to n where n is the number of parameters. Then, the local variables follow according to the nesting depth: If a block $\{i\colon T = vo; e\}$ is nested in k local variable or catch blocks or bodies of $sync_{\_}\ (\_)\ \_$, then $i = 1 + n + k$; and similarly for $try\ e\ catch(C\ i)\ e'$ and $sync_i\ (e)\ e'$. For example,

$$try\ e_1\ catch(C\ 3)\ (sync_4\ (\{4\colon T_1 = vo_1; e_2\})\ \{5\colon T_2 = vo_2; e_3\})$$

is fine for a method with two parameters, but $\{3\colon T_1 = vo_1; \{5\colon T_2 = vo_2; e\}\}$, $\{4\colon T = vo; e\}$, and $\{3\colon T_1 = vo_1; \{2\colon T_2 = vo_2; e\}\}$ are not. Klein and Nipkow call this layout "an inverse de Bruijn numbering scheme" [32, §5.1.1]. The predicate $\mathscr{B}\ e\ i$ enforces this scheme, where i denotes the starting number for the outermost blocks. For example,

$$\mathscr{B}\ (sync_j\ (e_1)\ e_2)\ i \longleftrightarrow \mathscr{B}\ e_1\ i \wedge i = j \wedge \mathscr{B}\ e_2\ (i+1)$$

The typing rules for J_1 are almost identical to *J*. In $P, E \vdash_1 e :: T$, the program P has type $'addr\ J_1\text{-}prog$ and the typing environment E for local variables now is a list of types where the i-th element corresponds to variable i. The rule $\mathrm{WT}_1\mathrm{SYNC}$ for $sync_{\_}\ (\_)\ \_$ blocks demonstrates all relevant changes:

$$\mathrm{WT}_1\mathrm{SYNC}\colon\ \frac{P, E \vdash_1 e_1 :: T_1 \qquad is\text{-}refT\ T_1 \qquad T_1 \neq NT \qquad P, E\ @\ [Class\ Object] \vdash_1 e_2 :: T}{P, E \vdash_1 sync_j\ (e_1)\ e_2 :: T}$$

[15] Technically, they are annotated with $()$ in *J*. Hence, $sync\ (e)\ e'$ actually abbreviates $sync_{()}\ (e)\ e'$ and similarly for $insync\ (a)\ e$. Moreover, functions like *fv*, *final*, $\mathscr{D}$, *inline*, and *ok*-$\mathscr{I}$ are defined on $('a,'b,'addr)\ exp$ instead of $'addr\ expr$. So, they work on $'addr\ expr_1$, too.

$$\mathrm{R_1SYNC1}: \quad \frac{fail,P,t \vdash \langle e_1,s\rangle -ta\rightarrow^{\mathrm{e}}_1 \langle e_1',s'\rangle}{fail,P,t \vdash \langle sync_i\ (e_1)\ e_2,s\rangle -ta\rightarrow^{\mathrm{e}}_1 \langle sync_i\ (e_1')\ e_2,s'\rangle}$$

$$\mathrm{R_1SYNCN}: \quad \frac{i < |xs|}{fail,P,t \vdash \langle sync_i\ (null)\ e,(h,xs)\rangle -(\!|\!|)\rightarrow^{\mathrm{e}}_1 \langle THROW\ NullPointer,(h,xs[i := null])\rangle}$$

$$\mathrm{R_1SYNCX}: \quad fail,P,t \vdash \langle sync_i\ (Throw\ a)\ e,s\rangle -(\!|\!|)\rightarrow^{\mathrm{e}}_1 \langle Throw\ a,s\rangle$$

$$\mathrm{R_1LOCK}: \quad \frac{i < |xs|}{fail,P,t \vdash \langle sync_i\ (addr\ a)\ e,(h,xs)\rangle -(\!|Lock\rightarrow a|\!)\rightarrow^{\mathrm{e}}_1 \langle insync_i\ (a)\ e,(h,xs[i := Addr\ a])\rangle}$$

$$\mathrm{R_1SYNC2}: \quad \frac{fail,P,t \vdash \langle e,s\rangle -ta\rightarrow^{\mathrm{e}}_1 \langle e',s'\rangle}{fail,P,t \vdash \langle insync_i\ (a)\ e,s\rangle -ta\rightarrow^{\mathrm{e}}_1 \langle insync_i\ (a)\ e',s'\rangle}$$

$$\mathrm{R_1UNLCKN}: \quad \frac{xs_{[i]} = Null \qquad i < |xs|}{fail,P,t \vdash \langle insync_i\ (a)\ (Val\ v),(h,xs)\rangle -(\!|\!|)\rightarrow^{\mathrm{e}}_1 \langle THROW\ NullPointer,(h,xs)\rangle}$$

$$\mathrm{R_1UNLCK}: \quad \frac{xs_{[i]} = Addr\ a \qquad i < |xs|}{fail,P,t \vdash \langle insync_i\ (a')\ (Val\ v),(h,xs)\rangle -(\!|Unlock\rightarrow a|\!)\rightarrow^{\mathrm{e}}_1 \langle Val\ v,(h,xs)\rangle}$$

$$\mathrm{R_1UNLCKF}: \quad \frac{fail \qquad xs_{[i]} = Addr\ a \qquad i < |xs|}{\begin{array}{l} fail,P,t \vdash \langle insync_i\ (a')\ (Val\ v),(h,xs)\rangle \\ \qquad -(\!|UnlockFail\rightarrow a|\!)\rightarrow^{\mathrm{e}}_1 \langle THROW\ IllegalMonitorState,(h,xs)\rangle \end{array}}$$

$$\mathrm{R_1UNLCKXN}: \quad \frac{xs_{[i]} = Null \qquad i < |xs|}{fail,P,t \vdash \langle insync_i\ (a')\ (Throw\ a'),(h,xs)\rangle -(\!|\!|)\rightarrow^{\mathrm{e}}_1 \langle THROW\ NullPointer,(h,xs)\rangle}$$

$$\mathrm{R_1UNLCKX}: \quad \frac{xs_{[i]} = Addr\ a \qquad i < |xs|}{fail,P,t \vdash \langle insync_i\ (a')\ (Throw\ a'),(h,xs)\rangle -(\!|Unlock\rightarrow a|\!)\rightarrow^{\mathrm{e}}_1 \langle Throw\ a',(h,xs)\rangle}$$

$$\mathrm{R_1UNLCKXF}: \quad \frac{fail \qquad xs_{[i]} = Addr\ a \qquad i < |xs|}{\begin{array}{l} fail,P,t \vdash \langle insync_i\ (a')\ (Throw\ a'),(h,xs)\rangle \\ \qquad -(\!|UnlockFail\rightarrow a|\!)\rightarrow^{\mathrm{e}}_1 \langle THROW\ IllegalMonitorState,(h,xs)\rangle \end{array}}$$

Fig. 37 Semantics of `synchronized` blocks in J_1

$P,E \vdash_1 \_ :: \_$ implicitly relies on the numbering scheme with $|E|$ as start index, as $\mathrm{WT_1SYNC}$ ignores the annotation variable j. Instead, it extends the environment for the monitor variable with the type *Class Object*, to which all monitor references conform.

Since the compiler introduces these monitor variables, no expression should access them. The predicate $\mathscr{S}\ e$ ensures this by checking that $i \notin fv\ e'$ for all subexpressions of e of the form $sync_i\ (\_)\ e'$ or $insync_i\ (\_)\ e'$.

The well-formedness conditions *wf-J$_1$-mdecl* specific to J_1 are similar to J's: the body must be well-typed, pass the definite assignment check and satisfy the predicates $\mathscr{B}$ and $\mathscr{S}$. The constraints on parameter names have been dropped and the numbering scheme and no access to monitor variables are required. The predicate *wf-J$_1$-prog* $=$ *wf-prog wf-J$_1$-mdecl* checks the general and J_1-specific well-formedness conditions.

5.4.2 Semantics

The state space of J_1 is 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 red_1 is now of the form $fail,P,t \vdash \langle e,(h,xs)\rangle -ta\rightarrow^{\mathrm{e}}_1 \langle e',(h',xs')\rangle$ where $P :: {'addr\ J_1\text{-}prog}$ and $xs, xs' :: {'addr\ val\ list}$. The new parameter $fail :: bool$ 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 does. In particular, $sync_i\ (addr\ a)\ e$ stores the monitor address a in the local variable i upon locking the monitor. Accordingly, when unlocking the monitor, $insync_i\ (a)\ e$ ignores a, but retrieves the monitor address from register i. Figure 37 shows

$$\mathrm{R_1RED}: \quad \frac{\mathit{fail},P,t \vdash \langle e,(h,xs)\rangle -ta\rightarrow_1^{\mathrm{e}} \langle e',(h',xs')\rangle}{\mathit{fail},P,t \vdash \langle (e,xs)\cdot exs,h\rangle -ta\rightarrow_1^{\mathrm{t}} \langle (e',xs')\cdot exs,h'\rangle}$$

$$\mathrm{R_1CALL}: \quad \frac{\begin{array}{c}\mathit{call}_1\ e = \lfloor (a,M,vs)\rfloor \\ \mathit{typeof\text{-}addr}\ h\ a = \lfloor T\rfloor \qquad P\vdash \mathit{class\text{-}of}'\ T\ \mathit{sees}\ M{:}Ts\rightarrow T = \lfloor \mathit{body}\rfloor\ \mathit{in}\ D \qquad |vs| = |Ts| \\ e' = \mathit{blocks}_1\ 0\ (\mathit{Class}\ D\cdot Ts)\ \mathit{body} \qquad xs' = \mathit{Addr}\ a\cdot vs\ @\ \mathit{replicate}\ (\mathit{max\text{-}vars}\ \mathit{body})\ \mathit{dummy\text{-}val}\end{array}}{\mathit{fail},P,t \vdash \langle (e,xs)\cdot exs,h\rangle -\langle\!|\!|\rangle\rightarrow_1^{\mathrm{t}} \langle (e',xs')\cdot (e,xs)\cdot exs,h\rangle}$$

$$\mathrm{R_1RET}: \quad \frac{\mathit{final}\ e}{\mathit{fail},P,t \vdash \langle (e,xs)\cdot (e',xs')\cdot exs,h\rangle -\langle\!|\!|\rangle\rightarrow_1^{\mathrm{t}} \langle (\mathit{inline}\ e\ e',xs')\cdot exs,h\rangle}$$

Fig. 38 Semantics for call stacks in J_1

the rules for synchronisation. They explicitly check for the bounds of the register array. In comparison to Figure 16, there are two new pairs of rules for unlocking: First, $\mathrm{R_1UNLCKN}$ and $\mathrm{R_1UNLCKXN}$ raise a *NullPointer* exception if the register $xs_{[i]}$ for the monitor stores the *Null* pointer instead of an address. Second, $\mathrm{R_1UNLCKF}$ and $\mathrm{R_1UNLCKXF}$ allow unlocking to fail with an *IllegalMonitorState* exception like *MExit* (Figure 17) if the switch *fail* is set.

$\mathit{fail},P,t \vdash \langle e,(h,x)\rangle -ta\rightarrow_1^{\mathrm{e}} \langle e',(h',x')\rangle$ differs from $P,t \vdash \langle \_,\_\rangle -\_\rightarrow \langle \_,\_\rangle$ in two further respects. First, the small-step semantics treats local variable blocks with initialisation, say $\{i\colon T = \lfloor v\rfloor;\ e\}$, like $\{i\colon T = \mathit{None};\ i := \mathit{Val}\ v;;\ e\}$ and completely ignores uninitialised blocks. This ensures that J_1 and bytecode treat local variables identically. Second, there is no rule for calling non-native methods.

The single-thread semantics J_1-*red* (notation $\mathit{fail},P,t \vdash \langle (e,xs)\cdot exs),h\rangle -ta\rightarrow_1^{\mathrm{t}} \langle (e',xs')\cdot exs',h'\rangle$) takes care of method calls and returns similar to J_0-*red* (Figure 38). call_1 differs from *call* only for blocks. Initialised blocks never pause at a call because J_1 first "uninitialises" them. Rule $\mathrm{R_1CALL}$ initialises the registers of the new call frame just like *exec-instr* does in Figure 17. The function *max-vars* computes the maximum depth of nested local variables including the variables for $\mathit{sync}\_\ (\_)\ \_$ blocks. Analogous to *blocks*, $\mathit{blocks}_1\ n\ Ts\ \mathit{body}$ wraps *body* in uninitialised blocks for local variables n to $n+|ts|-1$ with types Ts.

Setting *fail* to *False* or *True* yields two different semantics of J_1, to which I refer as $J_1^{\#}$ and J_1', respectively; J_1 refers to both. Similarly, I sometimes omit the *fail* parameter from the semantics and instead decorate them with $'$ or #, e.g, $J_1^{\#}$-*red* and $\_,\_\vdash \langle \_,\_\rangle -\_\rightarrow_1^{\mathrm{t}\prime} \langle \_,\_\rangle$.

Like for source code, the multithreaded semantics $J_1^{\#}.\mathit{redT}$ and $J_1'.\mathit{redT}$ are the interleaving semantics instantiated with $J_1^{\#}$-*red* and J_1'-*red*, respectively. A J_1 thread is final, written J_1-*final* $((e,xs)\cdot exs)$, iff *final* e and $exs = []$. The start state J_1-*start* $P\ C\ M\ vs$ has one thread *start-tID* with local state

$$[(\mathit{blocks}_1\ 0\ (\mathit{Class}\ D\cdot Ts)\ \mathit{body}, \mathit{Null}\cdot vs\ @\ \mathit{replicate}\ (\mathit{max\text{-}vars}\ \mathit{body})\ \mathit{dummy\text{-}val})]$$

where $P\vdash C\ \mathit{sees}\ M{:}Ts\rightarrow\_ = \lfloor \mathit{body}\rfloor\ \mathit{in}\ D$.

On the single-thread level, $J_1^{\#}$ and J_1' are not bisimilar as unlocking a monitor can nondeterministically 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 τ-moves, J_1 defines a predicate τ-move_1 similar to τ-*move* and J_1-τ-*move* lifts it to call stacks: J_1-τ-*move* $P\ (((e,xs),exs),h)\ ta\ \_ \longleftrightarrow (\tau\text{-}\mathit{move}_1\ P\ h\ e \vee \mathit{final}\ e) \wedge ta = \langle\!|\!|\rangle$.

Lemma 39 *$J_1^{\#}$ and J_1' satisfy the assumptions of locale τ-multithreaded.*

$$
\begin{aligned}
&\mathit{compE}_1\ \mathit{Vs}\ (\mathit{Var}\ V) &&= \mathit{Var}\ (\mathit{index}\ \mathit{Vs}\ V)\\
&\mathit{compE}_1\ \mathit{Vs}\ \{V : T = vo; e\} &&= \{|\mathit{Vs}| : T = vo; \mathit{compE}_1\ (\mathit{Vs} @ [V])\ e\}\\
&\mathit{compE}_1\ \mathit{Vs}\ (\mathit{sync}\ (e_1)\ e_2) &&= \mathit{sync}_{|\mathit{Vs}|}\ (\mathit{compE}_1\ \mathit{Vs}\ e_1)\ (\mathit{compE}_1\ (\mathit{Vs} @ [\mathit{fresh\text{-}vname}\ \mathit{Vs}])\ e_2)\\
&\mathit{compE}_1\ \mathit{Vs}\ (\mathit{insync}\ (a)\ e) &&= \mathit{insync}_{|\mathit{Vs}|}\ (a)\ (\mathit{compE}_1\ (\mathit{Vs} @ [\mathit{fresh\text{-}vname}\ \mathit{Vs}])\ e)
\end{aligned}
$$

Fig. 39 Register allocation compE_1 for local variables, blocks, and synchronisation

5.4.3 Compilation stage 1

Jinja formalises a compiler compE_1 from $'\mathit{addr}\ \mathit{expr}$ to $'\mathit{addr}\ \mathit{expr}_1$, i.e., for method bodies. It assigns registers to variables [32, §5.2] in the order required by J_1: first the *this* pointer, then the method parameters, and finally local variables ordered by block nesting level. While traversing the expression, 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 $\mathit{index}\ \mathit{Vs}\ V$), i.e., the position of the last occurrence of V in *Vs*. Figure 39 shows an excerpt of compE_1's definition, the full definition can be found in [49]. For $\mathit{sync}\ (e_1)\ e_2$ blocks, compE_1 reserves the register $|\mathit{Vs}|$ to hold the monitor address. To shift the registers in e_2 by 1, it appends a fresh variable name $\mathit{fresh\text{-}vname}\ \mathit{Vs}$ to *Vs*. Freshness ($\mathit{fresh\text{-}vname}\ \mathit{Vs} \notin \mathit{set}\ \mathit{Vs}$) ensures that it does not hide any variables in surrounding blocks.

Jinja defines an operator *compP* to lift compilation at the level of expressions to whole programs [32, §5.4]. I have straightforwardly adapted it to JinjaThreads programs. The compiler compP_1 from J to J_1 applies compE_1 to all method bodies.

$$\mathit{compP}_1 = \mathit{compP}\ (\lambda C\ M\ Ts\ T\ (\mathit{pns}, \mathit{body}).\ \mathit{compE}_1\ (\mathit{this} \cdot \mathit{pns})\ \mathit{body})$$

For example, consider the following method declaration in Java

```
int foo(Object f) { synchronized(f) { return this.m(); } }
```

In J's abstract syntax, the body is $([f], \mathit{sync}\ (\mathit{Var}\ f)\ (\mathit{Var}\ \mathit{this}.m([])))$. The compiler compP_1 compiles this declaration to $\mathit{sync}_2\ (\mathit{Var}\ 1)\ (\mathit{Var}\ 0.m([]))$.

5.4.4 Preservation of well-formedness

Jinja's proof of 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 (condition $\mathscr{S}\ \mathit{body}$). The next lemma (provable by induction) shows that compE_1 ensures this. The interesting cases $\mathit{sync}\ (e_1)\ e_2$ and $\mathit{insync}\ (a)\ e$ rely on $\mathit{fresh\text{-}vname}\ \mathit{Vs}$ being fresh.

Lemma 40 *If* $\mathit{fv}\ e \subseteq \mathit{set}\ \mathit{Vs}$, *then* $\mathscr{S}\ (\mathit{compE}_1\ \mathit{Vs}\ e)$.

Second, preservation of well-typedness requires the stronger induction on the structure of expressions instead of the usual induction on the derivation of the typing judgement:

Lemma 41 ([32, Lem. 5.5.]) *If* $\mathit{wf\text{-}prog}\ \mathit{wf\text{-}md}\ P$ *and* $P, [Vs\ [\mapsto]\ Ts] \vdash e :: T$ *and* $|Ts| = |Vs|$, *then* $\mathit{compP}_1\ P, Ts \vdash_1 \mathit{compE}_1\ \mathit{Vs}\ e :: T$.

Proof By induction on e. The interesting new case is $\mathit{sync}\ (e_1)\ e_2$. From $P, [Vs\ [\mapsto]\ Ts] \vdash \mathit{sync}\ (e_1)\ e_2 :: T$, there is a $T_1 \neq \mathit{NT}$ such that $\mathit{is\text{-}refT}\ T_1$, $P, [Vs\ [\mapsto]\ Ts] \vdash e_1 :: T_1$, and $P, [Vs\ [\mapsto]\ Ts] \vdash e_2 :: T$ by rule inversion. By the induction hypothesis, $\mathit{compP}_1\ P, Ts \vdash_1 \mathit{compE}_1\ \mathit{Vs}\ e_1 :: T_1$. For e_2, it does not suffice to apply the induction hypothesis directly because this would give $\mathit{compP}_1\ P, Ts \vdash_1 \mathit{compE}_1\ \mathit{Vs}\ e_2 :: T$ instead of

$$\mathit{compP}_1\ P, Ts @ [\mathit{Class}\ \mathit{Object}] \vdash_1 \mathit{compE}_1\ (\mathit{Vs} @ [\mathit{fresh\text{-}vname}\ \mathit{Vs}])\ e_2 :: T$$

as required by $\mathrm{WT}_1\,\mathrm{SYNC}$. This is the reason why the induction rule for $P,[Vs\,[\mapsto]\,Ts]\vdash e::T$, which Jinja uses, is too weak for this proof. Instead, since *fresh-vname Vs* is fresh,

$$P,[Vs\,@\,[\textit{fresh-vname}\ Vs]\,[\mapsto]\,Ts\,@\,[\textit{Class Object}]]\vdash e_2::T$$

follows from $[\![P,E\vdash e::T;\ E\subseteq_{\mathrm{m}} E']\!]\Longrightarrow P,E'\vdash e::T$ (provable by rule induction). Thus, the induction hypothesis applies. □

The remaining language-specific well-formedness constraints hold like in Jinja. Hence, preservation of well-formedness follows.

Theorem 42 *If wf-J-prog P, then wf-J_1-prog* $(\mathit{compP}_1\ P)$.

5.4.5 Semantic preservation for compE$_1$

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 $J_1^\#$ consist of a stack of semantics, the delay bisimulation at one layer composes language-specific constraints and bisimilulation relations from the level below (Figure 28), 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 cover both directions, the key ideas for the correctness proof [32, §5.5] are still sufficient.

In detail, the bisimulation relation ${}_0\overset{\mathrm{e}}{\approx}_1$ at the level of expressions is naturally the heart of the correctness proof because the translation's core is at this level. ${}_0\overset{\mathrm{t}}{\approx}_1$ extends ${}_0\overset{\mathrm{e}}{\approx}_1$ to call stacks; ${}_0\overset{\mathrm{m}}{\approx}_1$ lifts ${}_0\overset{\mathrm{t}}{\approx}_1$ to the interleaving semantics as described in Section 5.2.2. Hence, I want to prove that ${}_0\overset{\mathrm{t}}{\approx}_1$ 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 43 (forward direction) and Theorem 44 (backward direction).

Theorem 43 *Let wf-J-prog P and* $Vs\vdash(e_0,x_0)\approx(e_1,xs_1)$. *Let* $\mathit{fv}\ e_0\subseteq\mathit{set}\ Vs$ *and* $|Vs|+\mathit{max\text{-}vars}\ e_1\leq|xs_1|$. *Suppose that* $P,t\vdash\langle e_0,(h,x_0)\rangle-ta_0\rightarrow^{\mathrm{e}}_0\langle e_0',(h',x_0')\rangle$. *Then, there are* ta_1, e_1', *and* xs_1' *such that* $Vs\vdash(e_0',x_0')\approx(e_1',xs_1')$ *and the following hold:*

(i) If $\tau\text{-move}_0\ P\ h\ e_0$, *then* $h'=h$ *and* $ta_1=(\!|\,|\!)$ *and* $\langle e_1,(h,xs_1)\rangle$ *reduces in* $J_1^\#$ *with at least one* τ*-move to* $\langle e_1',(h',xs_1')\rangle$.
(ii) If $\neg\,\tau\text{-move}_0\ P\ h\ e_0$ *and* $\mathit{call}\ e_0\neq\mathit{None}$ *and* $\mathit{call}_1\ e_1\neq\mathit{None}$, *then* $\neg\,\tau\text{-move}_1\ (\mathit{compP}_1\ P)\ h\ e_1$ *and* $\mathit{compP}_1\ P,t\vdash\langle e_1,(h,xs_1)\rangle-ta_1\rightarrow^{\mathrm{e}\#}_1\langle e_1',(h',xs_1')\rangle$ *and* ta_0 *is* ${}_0\overset{\mathrm{t}}{\approx}_1$*-bisimilar to* ta_1.
(iii) Otherwise, there are e_1'' *and* xs_1'' *such that* $\langle e_1,(h,xs_1)\rangle$ *reduces in* $J_1^\#$ *with (possibly no)* τ*-moves to* $\langle e_1'',(h,xs_1'')\rangle$ *and* $\neg\,\tau\text{-move}_1\ (\mathit{compP}_1\ P)\ h\ e_1''$ *and* $\mathit{compP}_1\ P,t\vdash\langle e_1'',(h,xs_1'')\rangle-ta_1\rightarrow^{\mathrm{e}\#}_1\langle e_1',(h',xs_1')\rangle$ *and* ta_0 *is* ${}_0\overset{\mathrm{t}}{\approx}_1$*-bisimilar to* ta_1.

Theorem 44 *Let wf-J-prog P and* $Vs \vdash (e_0, x_0) \approx (e_1, xs_1)$. *Let fv* $e_0 \subseteq$ *set Vs and* $|Vs| +$ *max-vars* $e_1 \leq |xs_1|$ *and* $\mathscr{D}\ e_0\ \lfloor dom\ x_0 \rfloor$. *Suppose that* $compP_1\ P, t \vdash \langle e_1, (h, xs_1)\rangle -ta_1 \rightarrow_1^{e\#} \langle e_1', (h', xs_1')\rangle$. *Then, there are* ta_0, e_0', *and* x_0' *such that* $Vs \vdash (e_0', x_0') \approx (e_1', xs_1')$ *and the following hold:*

(i) If τ-$move_1\ (compP_1\ P)\ h\ e_1$, *then* $h' = h$ *and* $ta_0 = (\!|\,|\!)$ *and* $\langle e_0, (h, x_0)\rangle$ *reduces in* J_0 *with* τ*-moves to* $\langle e_0', (h', x_0')\rangle$. *If this involves no* τ*-moves, then cnt-IB* $e_1' <$ *cnt-IB* e_1.

(ii) If $\neg\ \tau$-$move_1\ (compP_1\ P)\ h\ e_1$ *and call* $e_0 \neq$ None *and call*$_1\ e_1 \neq$ None, *then* $\neg\ \tau$-$move_0\ P\ h\ e_0$ *and* $P, t \vdash \langle e_0, (h, x_0)\rangle -ta_0 \rightarrow_0^e \langle e_0', (h', x_0')\rangle$ *and* ta_0 *is* ${}_0\overset{t}{\approx}_1$*-bisimilar to* ta_1.

(iii) Otherwise, there are e_0'' *and* x_0'' *such that* $\langle e_0, (h, x_0)\rangle$ *reduces in* J_0 *with (possibly no)* τ*-moves to* $\langle e_0'', (h, x_0'')\rangle$ *and* $\neg\ \tau$-$move_0\ P\ h\ e_0''$ *and* $P, t \vdash \langle e_0'', (h, x_0'')\rangle -ta_0 \rightarrow_0^e \langle e_0', (h', x_0')\rangle$ *and* ta_0 *is* ${}_0\overset{t}{\approx}_1$*-bisimilar to* ta_1.

Consider the assumptions of the theorems first. The central relation $Vs \vdash (e_0, x_0) \approx (e_1, xs_1)$ fully encapsulates the relation between (e_0, x_0) and (e_1, xs_1). The others, *fv* $e_0 \subseteq$ *set Vs* and $|Vs| +$ *max-vars* $e_1 \leq |xs_1|$ and $\mathscr{D}\ e_0\ \lfloor dom\ x_0 \rfloor$, are only language-specific constraints that involve either of them. To improve proof automation, there are separate preservation lemmas for the latter. Consequently, only $Vs \vdash (e_0', x_0') \approx (e_1', xs_1')$ appears in the conclusion. In detail, $Vs \vdash (e_0, x_0) \approx (e_1, xs_1)$ predicates that

(a) the initialised local variables are the same, i.e., $x_0 \subseteq_m [Vs\,[\mapsto]\,xs_1]$,
(b) e_1 adheres to the numbering scheme for variables, i.e., $\mathscr{B}\ e_1\ |Vs|$,
(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 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 $insync_\_\ (\_)\ \_$ blocks, i.e., $\neg\ has\text{-}\mathscr{I}\ \_$.

Condition (c) is not required for $sync\ (\_)\ \_$ blocks 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 Vs* expresses that *Vs* captures all free variables in e. Second, $|Vs| +$ *max-vars* $e_1 \leq |xs_1|$ guarantees that xs_1 is large enough to hold all local variables during execution. The third constraint $\mathscr{D}\ e_0\ \lfloor dom\ x_0 \rfloor$ only appears in Theorem 44 and is new compared to Jinja. It ensures that J_0 does not get stuck when looking up a local variable in x_0. This could happen as $J_1^\#$ does not check that variables are initialised.

In the conclusions, case (i) corresponds to the τ-move simulation diagrams for well-founded delay bisimulations in Figure 33b. Theorem 43 always proves the left column, i.e., $J_1^\#$ simulates every τ-move in J_0 by at least one τ-move. In contrast, J_1 uninitialises local variable blocks before it executes the block's body (Section 5.4.1), which has no counterpart in J_0. Hence, Theorem 44(i) allows J_0 to stall when the number of initialised blocks decreases. The measure *cnt-IB* e_1 counts the initialised blocks in e_1.

Case (iii) corresponds to the visible moves simulating observable moves (Figure 33a). Case (ii) is the special case when both e_0 and e_1 pause at a call. In that case, no τ-moves may precede the simulating move. Remember that *sim-*$\approx$*1* and *sim-*$\approx$*2* of locale *m-dbisim-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 ${}_0\overset{t}{\approx}_1$-related. This is what well-formedness (premise *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 small-step semantics. The interesting cases are for local variables and synchronisation blocks, but Jinja's notions of hidden and unmodified variables [32, §5.5] suffice. Again, freshness of *fresh-vname Vs* is essential.

Now, it is clear what the bisimulation relations for expressions and threads should be. Recall that the expressions in J_0 call frames are closed, i.e., $Vs = []$ and $x_0 = empty$. Hence,

$$e_0\ {}_0\overset{e}{\approx}_1\ (e_1, xs_1) \longleftrightarrow [] \vdash (e_0, empty) \approx (e_1, xs_1) \wedge fv\ e_0 = \emptyset \wedge \mathscr{D}\ e_0\ \lfloor\emptyset\rfloor \wedge max\text{-}vars\ e_1 \leq |xs_1|$$

Lifting to single threads is straightforward. The heaps must be the same, the call stacks must be ${}_0\overset{e}{\approx}_1$-related pointwise, and all call frames except the top pause at a call.

$$(e_0 \cdot es_0, h_0)\ {}_0\overset{t}{\approx}_1\ ((e_1, xs_1) \cdot exs_1, h_1) \longleftrightarrow h_0 = h_1 \wedge e_0\ {}_0\overset{e}{\approx}_1\ (e_1, xs_1) \wedge |es_0| = |exs_1| \wedge$$
$$(\forall (e_0', (e_1', xs_1')) \in set\ (zip\ es_0\ exs_1).\ e_0'\ {}_0\overset{e}{\approx}_1\ (e_1', xs_1') \wedge call\ e_0' \neq None \wedge call_1\ e_1' \neq None)$$

For ${}_0\overset{m}{\approx}_1$, I take $\approx_m$ from Section 5.2.2 instantiated with ${}_0\overset{t}{\approx}_1$ and ${}_0\overset{w}{\approx}_1$, where threads in wait sets must pause at a call:

$$e_0 \cdot es_0\ {}_0\overset{w}{\approx}_1\ (e_1, xs_1) \cdot exs_1 \longleftrightarrow call\ e_0 \neq None \wedge call_1\ e_1 \neq None,$$

Lemma 45 *Let wf-J-prog P. Then,* ${}_0\overset{t}{\approx}_1$ *is a delay bisimulation with explicit divergence for* J_0*-red P t and* $J_1^{\#}$*-red* $(compP_1\ P)$ *t.*

Proof By Lemma 23, it suffices to show that ${}_0\overset{t}{\approx}_1$ is a well-founded delay bisimulation. This follows easily with Theorems 43 and 44 as J_0 and J_1 have similar call-stack semantics (Figures 36 and 38). The well-founded relations $\prec_0$ and $\prec_1$ are

$$e_0' \cdot es_0' \prec_0 e_0 \cdot es_0 \longleftrightarrow False$$
$$(e_1', xs_1') \cdot exs_1' \prec_1 (e_1, xs_1) \cdot exs_1 \longleftrightarrow cnt\text{-}IB\ e_1' < cnt\text{-}IB\ e_1$$

□

Theorem 46 *Let wf-J-prog P. Then,* ${}_0\overset{m}{\approx}_1$ *is a delay bisimulation with explicit divergence for* J_0*.redT P and* $J_1^{\#}$*.redT* $(compP_1\ P)$ *that preserves final states.*

Proof It suffices to show that the assumptions of locale *m-dbisim-div* are met. Lemma 45 discharges the first. Preservation of final states and heap changes is trivial because (i) finality is invariant under ${}_0\overset{t}{\approx}_1$ and (ii) ${}_0\overset{t}{\approx}_1$ only imposes equality on the heaps, but does not otherwise depend on it. $\approx\!\!\!\approx I$ is provable by induction and case analysis, whereas *sim-*$\approx\!\!\!\approx 1$ and *sim-*$\approx\!\!\!\approx 2$ follow from case (ii) of Theorems 43 and 44.

Lemma 47 *If wf-J-prog P and wf-start P C M vs, then*

$$J\text{-}start\ P\ C\ M\ vs\ {}_0\overset{m}{\approx}_1\ J_1\text{-}start\ (compP_1\ P)\ C\ M\ vs.$$

5.4.6 Equivalence of $J_1^{\#}$ *and* J_1'

The verification must show that unlocking a monitor in compiled code never fails. The intermediate language is the right place as its semantics already stores the monitor address in the registers like bytecode, but the syntax still enforces the structured locking discipline.

I prove that $J_1^{\#}$*.redT P* and J_1'*.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-ok*$_1$) is defined analogously to *lock-ok* in Section 3.4.2. In such a state, J_1'*.redT P* never picks R_1UNLCKF nor R_1UNLCKXF because the precondition of the thread action $(\!|UnlockFail{\rightarrow}a|\!)$ is violated. $J_1^{\#}$*.redT P* preserves *lock-ok*$_1$ under the following condition (notation $\overset{\text{1}}{\text{🔒}}\ \_\ \surd$) 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) $ok\text{-}\mathscr{I}\ e_1$, i.e., all $insync_\_\ (\_)\ \_$ subexpressions of e_1 lie on one path from the root in e_1's abstract syntax tree (Figure 27).

Hence, I define ${}_1\overset{m}{\approx}_1$ by $s_1\ {}_1\overset{m}{\approx}_1\ s_1' \longleftrightarrow s_1 = s_1' \wedge lock\text{-}ok_1\ (locks\ s_1)\ (tp\ s_1) \wedge \boxplus\ s_1\ \surd$

Theorem 48 *If wf-J_1-prog P, then ${}_1\overset{m}{\approx}_1$ is a strong bisimulation for $J_1^{\#}$.redT P and J_1'.redT P.*

Corollary 49 *If wf-J_1-prog P, then ${}_1\overset{m}{\approx}_1$ is a delay bisimulation with explicit divergence for $J_1^{\#}$.redT P and J_1'.redT P that preserves final states.*

Lemma 50 *If wf-J_1-prog P and wf-start P C M vs, then*

$$J_1\text{-}start\ P\ C\ M\ vs\ {}_1\overset{m}{\approx}_1\ J_1\text{-}start\ P\ C\ M\ vs.$$

5.5 Bytecode generation

The first compiler stage has already replaced variable names by register numbers. The second stage $compP_2$ now completes the translation in that it generates the bytecode instructions and exception tables for the expressions (Section 5.5.1). In this section, I show preservation of well-typedness (Section 5.5.2) and semantics (Section 5.5.3) for $compP_2$.

5.5.1 Compilation stage 2

The second compiler stage translates expressions into instruction lists (function $compE_2$ :: $'addr\ expr_1 \Rightarrow 'addr\ instr\ list$) and exception tables (function $compxE_2$ :: $'addr\ expr_1 \Rightarrow pc \Rightarrow nat \Rightarrow ex\text{-}table$). $compP_2 = compP\ compMb_2$ lifts $compE_2$ and $compxE_2$ to programs using $compP$ and computes the maximum stack size $max\text{-}stack$ and register size using $max\text{-}vars$.

$compMb_2\ C\ M\ Ts\ T\ body =$
$\quad (let\ ins = compE_2\ body\ @\ [Return];\ xt = compxE_2\ body\ 0\ 0$
$\quad\ in\ (max\text{-}stack\ body, max\text{-}vars\ body, ins, xt))$

JinjaThreads extends Jinja's $compE_2$ and $compxE_2$ to $sync_i\ (e_1)\ e_2$ expressions, on which I focus in this section. The other expressions are similar to Jinja [32, §5.3]. The translation of a $sync_i\ (e_1)\ e_2$ 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 $sync_i\ (e_1)\ e_2$ are

$compE_2\ e_1\ @\ [Dup, Store\ i, MEnter]\ @\ compE_2\ e_2\ @\ [Load\ i, MExit, Goto\ 4]\ @$
$[Load\ i, MExit, ThrowExc]$

First, the monitor expression e_1 is evaluated, its result on the stack duplicated and stored in register i; $MEnter$ locks the monitor. Then, the block e_2 is executed, the monitor address loaded back from register i and the monitor unlocked. $Goto\ 4$ 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, $compxE_2$ takes the current program counter pc and stack depth d as parameters. For $sync_i\ (e_1)\ e_2$, $compxE_2$ appends to the exception tables for e_1 and e_2 the entry $(pc_1, pc_2, Any, pc_2 + 3, d)$

where $pc_1 = pc + |compE_2\ e_1| + 3$ and $pc_2 = pc_1 + |compE_2\ e_2|$. 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.

Continuing the compilation example from Section 5.4.3, $compE_2$ compiles the method body $sync_2\ (Var\ 1)\ (Var\ 0.m([]))$ to

$[Load\ 1, Dup, Store\ 2, MEnter, Load\ 0, Invoke\ m\ 0, Load\ 2, MExit, Goto\ 4,$
$Load\ 2, MExit, ThrowExc, Return]$

with exception table $[(4, 6, Any, 9, 0)]$.

As I have already discussed in Section 2.4.1, *Any* is important for the verification, although $\lfloor Throwable \rfloor$ would also do in theory. Since the latter only applies to subclasses of *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.

5.5.2 Preservation of well-formedness

To show that $compP_2$ preserves well-formedness, Jinja defines a type compiler that computes a well-typing for the generated bytecode [32, §5.9]. Since JinjaThreads' extensions naturally fit in the compilation scheme, I only present the final theorem. The full details can be found in the formalisation [45].

Theorem 51 *If wf-J_1-prog P, then wf-jvm-prog* $(compP_2\ P)$.

5.5.3 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., τ-moves. Exception handling and the following instructions generate only τ-moves: *Load*, *Store*, *Push*, *Pop*, *Dup*, *Swap*, *BinOp*, *Checkcast*, *Instanceof*, *Goto*, *IfFalse*, *ThrowExc*, and *Return*. Additionally, *Invoke* generates a τ-move only if the called method is non-native or the native method *currentThread*.

Like between J_0 and $J_1^\#$, the key to correctness is delay bisimilarity on the call-frame 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 26 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 (Figure 28). A single step of execution is written $chk, P, ins, xt, t \vdash \langle (stk, loc, pc, xcp), h \rangle\ -ta\rightarrow^{e}_{\text{jvm}}\ \langle (stk', loc', pc', xcp'), h' \rangle$: If the exception flag *xcp* is *None*, it denotes that the check *chk* succeeds (see below), *pc* points to an instruction in *ins* and (stk', loc', pc', xcp') describes a possible successor state of executing the instruction $ins_{[pc]}$ with stack *stk* and registers *loc* according to *exec-instr*; *ta* is the thread action. If the exception flag is set, it denotes that *xt* contains a suitable exception handler at pc' and no stack underflow occurs. At the expression level, a step must preserve the call stack's length, i.e., neither return from a method nor call a non-native method.

B_1: $$P,e,h \vdash (Val\ v,xs) \approx ([v],xs,|compE_2\ e|,None)$$

B_2: $$\frac{P,e_1,h \vdash (Throw\ a,xs) \approx (stk,loc,pc,\lfloor a \rfloor)}{P,sync_i\ (e_1)\ e_2,h \vdash (Throw\ a,xs) \approx (stk,loc,pc,\lfloor a \rfloor)}$$

B_3: $$\frac{P,e_2,h \vdash (e,xs) \approx (stk,loc,pc,xcp)}{P,sync_i\ (e_1)\ e_2,h \vdash (insync_i\ (a)\ e,xs) \approx (stk,loc,|compE_2\ e_1|+3+pc,xcp)}$$

B_4: $$P,sync_i\ (e_1)\ e_2,h \vdash (sync_i\ (Val\ v)\ e_2,xs) \approx ([v],xs[i:=v],|compE_2\ e_1|+2,None)$$

Fig. 40 Example introduction rules for the relation $\approx$

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 the instruction list is enlarged at either end and the stack extended 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*. Since *exec-meth* gets stuck only when red'_1 is also stuck, I use *exec-meth* for simulating red'_1's reductions.

The other direction uses *exec-methd* because it gets stuck at least as often as red'_1. For *ThrowExc*, e.g., *check-instr* demands that the exception is a subclass of *Throwable*, but red'_1 does not. So, *exec-methd* cannot simulate red'_1 unless the bisimulation excludes such cases, e.g., by requiring bytecode conformance. This would further complicate the proofs, which are already tedious. Conversely, red'_1 cannot simulate *exec-meth* as red'_1 gets stuck when trying to access a field of an integer, but *exec-meth* carries on with unspecified behaviour.

As the generated bytecode is well-formed (Theorem 51), the defensive *exec-methd* simulates the almost aggressive *exec-meth* for conformant states and preserves bytecode conformance. 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 and sensitive to small changes, I have not attempted to do so.

This detour only affects the semantics, not the bisimulation relation ${}_1\overset{e}{\approx}_{jvm}$. As before, ${}_1\overset{e}{\approx}_{jvm}$ consists of two parts, (i) a relation between J_1 call frames and JVM expression-level states and (ii) well-formedness conditions of the states, which the individual semantics preserve. The relation $P,e,h \vdash (e_1,xs_1) \approx (stk,loc,pc,xcp)$ relates a J_1 call frame (e_1,xs_1) (expression and registers) 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 $e :: {}'addr\ expr_1$ compiles to the instruction list $ins = compE_2\ e$ and $xt = compxE_2\ e\ 0\ 0$. The inductive definition for $\approx$ mirrors the reduction rules of red'_1 and relates a partially evaluated expression e_1 with the corresponding stack *stk*, registers *loc*, and the instruction position *pc*.

Figure 40 shows some representative rules from the inductive definition. The single rule B_1 for all expressions exploits that the last instruction in a compiled expression always puts the result 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, $\approx$ contains separate exception propagation rules for every expression, similar to B_2. Still, it abstracts from computed 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 *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 since they differ in intermediate states like in B_4, which R_1LOCK skips.

The bisimulation relation ${}_1\overset{e}{\approx}_{jvm}$ specialises this relation to complete call frames:

$$\frac{P \vdash C \text{ sees } M{:}Ts \to T = \lfloor body \rfloor \text{ in } D \qquad P, blocks_1\ 0\ (Class\ D \cdot Ts)\ body, h \vdash (e_1, xs_1) \approx (stk, loc, pc, xcp) \qquad max\text{-}vars\ e_1 \le |xs_1|}{P, h \vdash (e_1, xs_1)\ {}_1\overset{e}{\approx}_{jvm}\ (xcp, stk, loc, C, M, pc)}$$

The main simulation theorems at the expression level are similar in structure to Theorems 43 and 44. Their proofs consist of a huge induction on the relation and case analysis of the reductions. Control constructs like conditionals and loops, which are compiled to (conditional) jumps, are verified like in sequential Jinja.

The bisimulation relation ${}_1\overset{t}{\approx}_{jvm}$ for single threads lifts ${}_1\overset{e}{\approx}_{jvm}$ 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.

Theorem 52 (Correctness of stage 2) *In locale typesafe, suppose that wf-J_1-prog P. Then,*

(i) ${}_1\overset{t}{\approx}_{jvm}$ *is a delay bisimulation with explicit divergence for J_1'-red P t and the defensive VM jvm-execd (compP$_2$ P) t.*

(ii) ${}_1\overset{m}{\approx}_{jvm}$ *is a delay bisimulation with explicit divergence for J_1'.redT P and jvmd.redT (compP$_2$ P) that preserves final states.*

Lemma 53 *If wf-J_1-prog P and wf-start P C M vs, then*

$$J_1\text{-}start\ P\ C\ M\ vs\ {}_1\overset{m}{\approx}_{jvm}\ jvm\text{-}start\ (compP_2\ P)\ C\ M\ vs.$$

5.6 Complete compiler

In the previous sections, I have shown that all relations in Figure 28 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 42 and 51.

Theorem 54 *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., $({}_J\approx_{jvm}, {}_J\sim_{jvm}) = \overset{m}{\approx}_0 \bigcirc_{B\,0} \overset{m}{\approx}_1 \bigcirc_{B\,1} \overset{m}{\approx}_1 \bigcirc_{B\,1} \overset{m}{\approx}_{jvm}$. Lemmas 38, 47, 50, and 53 show that ${}_J\approx_{jvm}$ relates the start states. And Lemma 22 composes the bisimulation theorems 37, 46, 52 and Corollary 49 to obtain the main correctness theorem 56.

Theorem 55 *If wf-J-prog P and wf-start P C M vs, then*

$$J\text{-}start\ P\ C\ M\ vs\ {}_J\approx_{jvm}\ jvm\text{-}start\ (J2JVM\ P)\ C\ M\ vs.$$

Theorem 56 *In locale typesafe, if wf-J-prog P, then $({}_J\approx_{jvm}, {}_J\sim_{jvm})$ is a delay bisimulation with explicit divergence that preserves final states.*

All proofs have been conducted independently of the heap implementation under the assumptions of the locale *typesafe*. As ${}_1\overset{t}{\approx}_{jvm}$ imposes bytecode conformance on the JVM state to show that *exec-meth* and *exec-methd* are equivalent, Theorem 56 holds for all implementations that satisfy *typesafe*, in particular sequential consistency and the JMM (Section 4).

Since ${}_{\mathrm{J}}\approx_{\mathrm{jvm}}$ decomposes into the relations for each stage, I now can break down correctness in terms of bisimilarity to concrete executions ξ, i.e., trace equivalence, using the Theorems 24 and 25 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 57 (Correctness) *Let wf-J-prog and wf-start P C M vs.*

(a) Let $P \vdash$ J-start P C M vs $\downarrow\!\downarrow_{\mathrm{J}}$ ξ. Then, there is a ξ' such that ξ $[{}_{\mathrm{J}}\approx_{\mathrm{jvm}}, {}_{\mathrm{J}}\sim_{\mathrm{jvm}}]$ ξ' and J2JVM $P \vdash$ jvm-start (J2JVM P) C M vs $\downarrow\!\downarrow_{\mathrm{jvmd}}$ ξ' and
  (i) If ξ terminates in s and J.mfinal s, then ξ' terminates in the state mexception s, which is jvm-final.
  (ii) If ξ deadlocks in state s, then ξ' deadlocks in some s', too.
  (iii) If ξ diverges or runs infinitely, so does ξ'.

(b) Let J2JVM $P \vdash$ jvm-start (J2JVM P) C M vs $\downarrow\!\downarrow_{\mathrm{jvmd}}$ ξ'. Then there is a ξ such that $P \vdash$ J-start P C M vs $\downarrow\!\downarrow_{\mathrm{J}}$ ξ and ξ $[{}_{\mathrm{J}}\approx_{\mathrm{jvm}}, {}_{\mathrm{J}}\sim_{\mathrm{jvm}}]$ ξ' and
  (i) If ξ' terminates in s' and jvm.mfinal s', then ξ terminates in some s such that J.mfinal s and $s' =$ mexception s.
  (ii) If ξ' deadlocks in state s', then ξ deadlocks in some s, too.
  (iii) If ξ' diverges or runs infinitely, so does ξ.

Proof (sketch) 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 24 and 25.

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 $\lfloor a \rfloor$ 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 27 shows that $\overset{m}{\approx}_0$, ${}_0\overset{m}{\approx}_1$, and ${}_1\overset{m}{\approx}_{jvm}$ preserve deadlocks. For ${}_1\overset{m}{\approx}_1$, it is easy to show that the semantics for $J_1^{\#}$ and J_1' differ only in states that cannot be in deadlock. Hence, ${}_1\overset{m}{\approx}_1$ preserves deadlocks, too. Since s and s' are stuck by construction of ξ and ξ', there are no τ-moves that Theorem 27 would allow, i.e., s and s' are both deadlocked. □

While the above theorem correctly describes semantic preservation for sequential consistency, the JMM adds another layer 7 that is not yet reflected. By Corollary 28, delay bisimiliarity enforces that the legal executions are the same.

Theorem 58 *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 J2JVM P with start state jvm-start (J2JVM P) C M vs.*

Corollary 59 *Let wf-J-prog P and wf-start P C M vs. Then, P is correctly synchronised iff J2JVM P is.*

6 Discussion

In this section, I reflect on some of my formalisation choices in JinjaThreads (Section 6.1) and look back to see (i) how JinjaThreads manages to recycle much of the previous efforts (Section 6.2) and (ii) what the motivations for the different parts were and what benefits they brought (Section 6.3).

6.1 Formalisation choices

JinjaThreads uses standard techniques in programming language formalisation: The source code semantics is a traditional small-step semantics with subexpression reduction and exception propagation rules and interleaving. Type safety is shown by progress and preservation. The compiler verification relies on a simulation argument. The virtual machine mimicks an abstract state machine. It thus shows that these standard techniques do scale to complicated languages. In this section, I take a look back and discuss the advantages and disadvantages over other formalisation choices.

The small-step semantics is defined as an inductive predicate with 98 rules for 23 language constructs. 28 rules reduce subexpressions (these are the only recursive rules) and 27 propagate exceptions. That is, more than half of the rules merely control what is (not) evaluated; they do not specify any actual behaviour. Consequently, the majority of cases in an induction proof about the small-step semantics deal only with subexpressions and exceptions. I therefore tried to prove these cases automatically. This often worked pretty well because many proof invariants were also defined inductively or recursively over the syntax. As all these small-step rules pattern-match on constructors of the abstract syntax tree, Isabelle's proof automation often automatically simplifies the proof invariants as needed and solves the corresponding subgoals. Subexpression reduction rules are thus less of a problem for proofs than for validating the semantics by inspection as some behaviours, in particular for exceptions, arise only from the interplay of many rules.

To reduce the number of rules, I tried to combine all subexpression reduction rules in the following evaluation context reduction rule as is often done in formalisations on paper [87]:

$$\textsc{Context}\,\frac{P,t \vdash \langle e,s\rangle -ta\rightarrow \langle e',s'\rangle}{P,t \vdash \langle C[e],s\rangle -ta\rightarrow \langle C[e'],s'\rangle}$$

where C denotes an evaluation context with a single hole and the context substitution $C[e]$ puts e into C's hole. I abandoned this attempt, though, as it made the formalisation more complicated instead of simpler. The inductive case for CONTEXT typically required a nested induction over the evaluation contexts, whose cases corresponded exactly to the subexpression reduction cases I had before. The only difference was that now there were two nested inductions instead of just one. All attempts to disentangle the induction lead to a lot of duplication. For example, proving that context substitution preserves well-typedness requires a type system for evaluation contexts, whose rules mimick the type system for expressions. Moreover, with CONTEXT, the small-step semantics is no longer executable. Isabelle's compiler for inductive predicates can generate an interpreter out of the described JinjaThreads semantics, but it would fail to invert the matching against context substitutions.

In summary, explicitly enumerating all subexpression reductions and exception propagations worked reasonably well for JinjaThreads. In retrospect, however, it would have been worth to try a continuation-passing style like in Krebbers's C semantics [36] and in the $\mathbb{K}$ framework [71]. Krebbers represents the evaluation context using a zipper data structure.

This technique has three benefits over subexpression reduction rules: First, the relation is not recursive any more, so case analysis instead of induction suffices. Second, the currently evaluated subexpression is exposed at the root of the zipper. This makes the interpreter more efficient as the evaluator need not search in every step for the next evaluation position. Third, as a by-product, the method call stack would be explicit, too, so I could have avoided the hackish call-stack semantics for J_0 and J_1. Conversely, like with CONTEXT, more definitions are needed, e.g., a type system for the zipper. Moreover, the zipper manipulations introduce even more intermediate states without a bytecode counterpart, which the downward simulation proof for the compiler must deal with. For local assignment, e.g., the configurations $V := e$ and $e \mid V := \square$ are equivalent. Subexpression reduction rules avoid the step that transforms the former into the latter.

The strict separation between single-threaded and multithreaded semantics worked very well and forced me to disentangle sequential and concurrent aspects. In particular, the decomposition of the Java synchronisation primitives into basic thread actions paid off, as every synchronisation aspect had to be formalised only once. Thanks to the interleaving, there is always one current multithreaded state on which the BTA conditions are checked and the updates performed. In hindsight, it would have been sensible to take the thread isolation one step further by eliminating the shared heap altogether. Every memory access and address type query would then be another basic thread action that the multithreaded semantics interprets. Then, there would be no thread interactions hidden from the multithreaded level. This would make obsolete all the locale assumptions about preservation under changes to the heap by other threads (for example the third assumption of locales *lifting-wf* and *lifting-inv* in Figure 21 and assumption *heap-change* in Figure 35). Moreover, a (coinductive) big-step semantics would be possible for individual threads, which produces the set of all possible traces of thread actions. Verifying the compiler against such a big-step semantics for the single-threaded case might have been easier. Conversely, the type safety proof would have been harder because type-correctness of values can no longer be determined for threads in isolation. The semantics would no longer be executable, either.

In the compiler verification, the need for bisimulations seems artificial, only due to the strict separation into single-threaded and multithreaded semantics and to quirks of the JMM. Starting from scratch, I would try harder to prove only one direction: the upward simulation for the single-threaded defensive VM or a downward simulation for a source code big-step semantics. The simulation could similarly be lifted to the multithreaded case, except that preservation of stuck states requires a syntactic characterisation. Breaking the abstraction appears in retrospect less effort than proving both simulation directions.

6.2 From Java^{light} to JinjaThreads

JinjaThreads has been designed to reuse Jinja's definitions and proofs, which itself builds on ideas in Java^{light} and μJava. This shows that formalisation reuse is possible at a large scale.

Reuse is most obvious in Jinja's program declaration infrastructure, lookup functions and generic well-formedness constraints, which can be traced back to early versions of Java^{light} [59]. These definitions, theorems, and proofs needed only few adaptations and some extensions such as exceptions being subclasses of *Throwable* and the formalisation of native methods (Section 2.3.1). Similarly, the type system, definite assignment checks and the bytecode verifier closely resemble Jinja's and μJava's.

The picture changes when we look at the semantics of the languages because this is where concurrency becomes pervasive. The abstract heap module (Sections 2.3.2 and 3.4.1) formally captures the abstract concept of a heap that was implicit in Jinja and its prede-

cessors, and identifies the assumptions on the heap. To reuse the sequential semantics of source code and bytecode, JinjaThreads defines a language-independent semantics for interleaving that separates concurrency issues from single-threaded concerns. The crucial design choice here is that the single-threaded semantics can only access the thread-local state and the shared heap, but not the whole multithreaded state. This way, it was enough to extend the sequential Jinja semantics slightly, namely with the thread ID and the thread action, and to adapt the VM to return a set of successor states due to the non-determinism. Indeed, I did not even change the order of arguments of the small-step semantics and the components of the VM's state. This can be seen in the definitions of *J-red*, *jvm-exec*, and *jvm-execd* in Sections 2.3.4 and 2.4.2, which require some regrouping of the state to fit into the format required by the interleaving semantics. Although not particularly nice, this glue code caused less effort than manually editing all the definitions, lemmas, and proofs to account for a different order of arguments and components. Unfortunately, Isabelle does not yet provide any tool support for such large-scale refactorisations.

Reuse of theorems and proof scripts is much more important than of the definitions, which make up only a small fraction of the JinjaThreads sources. Clearly, this is only possible if the definitions are changed and extended in ways that are in line with the previous design decision. Otherwise, only the abstract proof ideas can be transferred, but the actual proof script must be written anew. JinjaThreads contains examples of both.

The type safety proofs belong to the good cases. As the higlighting in Lemma 12 and Theorem 13 in Section 3.4.1 shows, the single-threaded progress and preservation statements for source code and bytecode hardly differ from Jinja's. Indeed, the same invariants (conformance, definite assignment, ...) are needed and the preservation proofs follow the same proof structure and many cases of the proofs are textually identical in Jinja and JinjaThreads. Clearly, threads require new invariants with their preservation lemmas and new properties (Figure 25 on page 38), but this is to be expected when new features are added. The important point is that the proofs for the existing features can be recycled. So, JinjaThreads benefits from the good choice of proven lemmas and the setup for the proof automation that had been developed for its predecessors.

For the compiler, the reuse situation is mixed. As the compiler does not optimize, the language extensions for Java threads require only small adaptations and extensions to the existing definition of the compiler and the syntax of the intermediate language (Sections 5.4.1, 5.4.3, and 5.5.1). Similarly, the proofs that the compiler preserves well-formedness of programs (Theorems 42 and 51) are to a large extent taken over from Jinja, except for the change in the induction principle in Lemma 41. In contrast, semantic preservation merely relies on the same abstract ideas, but differs in all other respects as it is now carried out against the single-threaded small-step semantics instead of the sequential big-step semantics, which no longer exists. Interestingly, the equivalence proof for the small-step semantics J and J_0 reuses several invariants and preservation lemmas from Jinja's equivalence proof for the big-step and the small-step semantics (Lemma 31 and the last two claims in the proof of Lemma 35). This is possible because Jinja's equivalence proof, too, must prove that the implicit call stack of the big-step semantics can be flattened to the dynamic method inlining of the small-step semantics. So, proofs sometimes get reused for a different purpose.

In summary, although I have tried to reuse in JinjaThreads as much as possible from Jinja, often it is more the general ideas and concepts that have survived than their textual formulation in Isabelle/HOL. In the end, JinjaThreads is incompatible with Jinja, but every Jinja program can be trivially transformed into a JinjaThreads program. Still, reuse saved me from having to choose right from all those innumerable formalisation options, only a few of which lead to easy and automatic proofs.

Overall, the desire for reuse led to the modular architecture with a strict separation between sequential and concurrent features, and, in particular, to the stack of semantics in Figure 2. It probably would have been much easier to formalise the semantics of source code and bytecode as a single huge definition (Section 7 discusses other Java formalisations in this style). But the modularity made it possible to reuse many parts of JinjaThreads for both source code and bytecode, e.g., the interleaving semantics (Section 2.2), the memory models (Section 4), and the lifting of bisimulation relations for the compiler verification (Section 5.2.2). Overall, I am convinced that savings by reusing proofs clearly outweigh the additional effort for the modular definitions.

Reuse also has a downside in that certain design decisions prevent clean formalisations and extensions. For example, JinjaThreads's small-step semantics avoids the need to model call stacks like in Jinja, but call stacks are an important stepping stone in the compiler verification to make the proofs manageable. Therefore, I developed a version with call stacks (Section 5.3.1), but extracting the calls using *call* is not an elegant solution. A proper solution like Krebber's with zippers for C [36] would have meant that I have to restructure and redo the existing type safety proofs. Similarly, for the validation (Section 8), I tried to extend JinjaThreads with `break` and `continue` control statements. In the end, I aborted this attempt because these control statements break too many assumptions of the compiler verification, i.e., the required adaptation effort was too high given the priority of the extension. Similarly, JinjaThreads itself evolved over time, too, and early parts like the interleaving semantics influenced many later design decisions. For example, the JMM formalisation (layer 7) sitting on top of the interleaving semantics (layer 5) is in conflict with the ideas of true concurrency. In fact, the interleaving semantics introduces an artificial notion of global time and my formalisation therefore cannot produce certain out-of-order behaviours (Section 2.3.3). If formalising the JMM had been envisioned from the start, no interleaving semantics would have been needed and the JMM could have been defined on top of a (coinductive) single-threaded big-step semantics. Clearly, these cleaner re-developments could be realised in the future, but when I made these design decisions, reuse seemed to be a better use of my research time.

6.3 Formalisation motivations, efforts, and outcomes

The motivations and goals of formalising Java changed with the different Java models. Back in the 1990s, Oheimb and Nipkow formalised Java$^{\ell ight}$ in Isabelle/HOL to demonstrate that "the technology for producing machine-checked programming language designs has arrived" [64]. Meanwhile, mechanisation has become standard: many papers at top programming language conferences come with a formalisation of their core calculi and theorems in a proof assistant.

Bali and μJava were developed in the context of the VerifiCard project [82]. This project aimed at developing a specification and verification tool set for the JavaCard platform grounded in formal methods. Here, Bali and μJava provided the formal semantics that enables the rigorous verification of programs against their specification, and the verified bytecode verifier enables the safe execution of untrusted code in a sandbox. The VerifiCard project as a whole brought formal verification with model checkers and proof assistants closer to practice.

Next, Jinja [32] aimed at unifying the different models from Bali and μJava, i.e., finding a balance between an abundance of features, which previously had been studied in isolation, and tractability of the resulting model. Klein and Nipkow emphasized comprehensibility of

the model and readability of the formal proofs, so they simplified the languages and left out some of the features.

The Quis custodiet project [68] picked up Jinja as a formal foundation for verifying an information flow control algorithm. Wasserrab and Lohner [86] formalised control flow graphs, call graphs, and program dependence graphs for Jinja bytecode and connected them to Wasserrab's slicing framework [85]. This connection enabled them to show that program slicing can establish non-interference properties of Jinja programs. To that end, they showed that the different graphs correctly abstract the program. These proofs heavily rely on the bytecode verifier's correctness and type safety of the JVM because the graph constructions depend on static type information.

The concurrency extensions towards JinjaThreads described in this paper originate in the Quis custodiet project, too. The slicing algorithm had been extended to multithreaded programs (assuming interleaving semantics) [20] and so should its formal verification. In 2007, the work on the interleaving semantics started, which was supposed to develop the foundations for verifying the algorithm. Type safety and the correctness of the bytecode verifier were essential for the graph construction and its correctness. The compiler verification had two aims: (i) to complete the extension of Jinja to concurrency and to sanity-check the source code and bytecode formalisations against each other, and (ii) to enable the transfer of non-interference properties between source code (e.g., established by a type system) and bytecode. The verification of slicing-based non-interference for multithreaded programs is still ongoing [12].

The Java memory model study, which I have only touched on in this paper, had several goals. First, most Java program analyses, including slicing-based IFC, assume interleaving semantics. They are therefore only sound for programs without data races, as these behave as if threads were interleaved. Therefore, extending JinjaThreads with the JMM and proving the DRF guarantee is necessary for the verification of such algorithms. Second, and more importantly, the Java memory model and its interaction with the language had not been studied much before, unlike the sequential features of Jinja and the synchronisation primitives of the interleaving semantics. So I aimed at understanding the JMM better and at clarifying some of its dark corners. This study [50] has revealed subtle interactions with object allocation and type safety, and showed that values may appear "out of thin air" in some Java programs, which theoretically compromises the Java security architecture. The search for a better JMM specification has started, but no convincing solutions have been found so far.

Regarding the formalisation effort, I can only give numbers for extending Jinja to JinjaThreads. The first version of the interleaving semantics and its connection to the source code semantics as described in [46] took about nine person months. Verifying the compiler as published in [47], i.e., without the memory model and without preservation of nontermination and deadlocks, required one year. I spent on the JMM formalisation and the DRF guarantee ([48]) another year and on type safety for the JMM [50] about nine months. Code generation and validation (see Section 8, [51]) were spread out over a longer period, so I cannot give an estimate there. In parallel, I occasionally added new features and refactored the existing formalisation, which is not included in the above times. The effort to add a new feature varied a lot. Extensions that fit into the design of the language are easy. For example, adding the compare-and-set operation took me three days. Some extensions, e.g., arrays, required weeks of refactoring to accomodate them in the model.

In summary, these efforts lead to three main benefits: First, they advanced the state of the art in formalising programming languages and structuring the formalisations. In particular, they showed that standard techniques scale to large models. Second, they identified problems

with the Java programming language and proposed clarifications. Third, they produced an artefact on which further work can be and is being built.

7 Related Work

7.1 Formalisation of multithreaded Java

Formalisations of (aspects of) sequential Java and Java bytecode abound in the literature. Hartel and Moreau [24] provide a good overview and Alves-Foss [2] has collected many early works. Most formalisations cover either only Java source code or only Java bytecode. One notable exception is the semantics by Stärk et al. [79] for a subset of Java source code and bytecode in terms of abstract state machines, for which they prove subject reduction. They use neither machine support for the semantics nor for checking their proofs.

For concurrent Java source code, AtomicJava [19] 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.

In K-Java, Bogdănaş and Roşu [11] formalise all features of Java 1.4 using the $\mathbb{K}$ system. The multithreading features are similar to JinjaThreads': they include threads with forks and joins, `synchronized` blocks, interruption, and the wait-notify mechanism. There is no compare-and-set operation as the `java.util.concurrent` package was added only in Java 5. K-Java does not cover the Java memory model and assumes sequential consistency instead. Bogdănaş and Roşu focus on completeness of the syntactic features and validate their formalisation by running numerous test cases. Using the $\mathbb{K}$ framework, they generate several tools such as a Java interpreter and a model checker form the semantics. They do not prove any property about their formalisation, though.

There are more formalisations for multithreaded Java bytecode. First, Liu and Moore [43] 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 [13]. Like JinjaThreads, their VM semantics implements native methods from the CLDC standard library, in particular for class loading and concurrency, but not thread interruption. In their monolithic semantics, every instruction and native method can access any part of the state, which reduces modularity. They aim for verifying small Java programs [44, 56] and JVM implementations with respect to the JVMS. Thus, they do not define a type system for bytecode nor prove type safety. Instead, the M6 models bytecode much closer to the JVMS than JinjaThreads does. Like its predecessors, JinjaThreads abstracts from technical details like the constant pool and string literals.

Second, Bicolano [55] serves as the basis for the proof carrying code infrastructure in the Mobius project [5]. It provides a comprehensive model for CLDC except for concurrency and class loading in Coq. In BicolanoMT, Huisman and Petri [25] extend Bicolano with interleaving concurrency including interrupts and the wait-notify mechanism, but no spurious wake-ups. By using the extension framework by Czarnik and Schubert [14], they do not need to change the sequential Bicolano semantics at all, but the extension is tied to the bytecode language. In contrast, JinjaThreads adds the semantics of *MEnter* and *MExit* instructions to the single-threaded semantics and labels all rules with thread actions. In return, JinjaThreads uses the same interleaving semantics for source code and bytecode.

Third, Belblidia and Debbabi present a formal small-step semantics for multithreaded Java bytecode [7]. Like JinjaThreads, they have a semantics for threads in isolation and a second layer which manages the threads and receives basic thread actions. 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 unblocking threads. They do not model the wait-notify mechanism nor thread interruption. Like the M6 and BicolanoMT, they only give the semantics, but no type system and no proofs. In JinjaThreads, the interleaving semantics manages the complete multithreaded state which includes the locks, wait sets, and interrupts. This isolates them from one another and relieves 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 has simple semantics.

Forth, JavaFAN [18, 17] is a formal analyser for Java source and bytecode in Maude. Although two formal semantics for both source code and bytecode are provided, these are unconnected. 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 the JinjaThreads source code interpreter suffers. In JinjaThreads, dynamic method inlining encloses the statement to execute in increasingly many blocks for *this* and the parameters and execution has to traverse them at every step of execution. The JavaFAN VM follows a traditional style, but separates deterministic sequential instructions from non-deterministic multithreading to boost performance.

Beyond Java, Krebbers [36] has formalised a large (sequential) subset the C11 standard in Coq. He aims at modelling all the intricate corners of the C language as closely as feasible, including bit-wise memory access, non-local control flow, and unspecified and implementation-defined behaviour. The executable operational semantics traverses the program expression using a zipper data structure. This way, traversal rules replace the traditional subexpression reduction rules. This has two advantages: First, the interpreter becomes faster as it need not search over and over again for the next subexpression to evaluate. Second, non-local control flow like `goto` can freely traverse the whole program to find the jump target. Krebbers also formalised an axiomatic semantics using separation logic and proves it sound against the operational semantics. The implementation-defined behaviour is configurable using Coq type classes, which are remotely similar to locales in Isabelle.

7.2 Type safety proofs of concurrent languages

There are only few type safety proofs for multithreaded languages like Java. Often, only subject reduction is shown, which eliminates the need to deal with deadlocks.

Grossman [23] 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. Like *deadlocked* in JinjaThreads, badly stuck is defined semantically. In general, being deadlocked is stronger than being badly stuck because being badly stuck misses the aspect of circular waiting.

Goto et al. [22] 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 over-approximation of deadlock. Progress with this restriction no longer guarantees that the semantics is not missing any rule. For example, it holds even for a semantics without any rules for `synchronized` statements.

7.3 Verified compilers

Compiler verification has been an active research topic for more than 40 years; see [15] for an annotated bibliography. Rittri [70] and Wand [84] first used bisimulations for verifying a compiler for a parallel functional language. They showed that running the compiled code on a VM is weakly bisimilar to the programs denotational semantics, which ignores divergence.

Most closely related to JinjaThreads' compiler is naturally Jinja's [32, §5] which itself builds on Strecker's [80] for μJava. They handle subsets of sequential JinjaThreads and are verified with respect to the big-step semantics. Their correctness theorem cannot rule out that the compiler transforms an infinitely running program into a terminating one.

Leroy's CompCert project [40] 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 downward direction (the upper half of Figure 33). He claims that upward simulations are harder to show than downward ones for CompCert. In JinjaThreads, the situation is the opposite because exception handling in source code takes a lot more steps than in bytecode due to the exception propagation rules.

For CompCertTSO, Ševčík et al. [78] adapt Leroy's approach of deriving upward from downward simulations to the concurrent setting. They require that the single-threaded semantics does not exhibit internal non-determinism and that it is receptive. Receptiveness classifies observable single-thread steps (such as reading a value from memory) into equivalence classes and requires that a thread can perform all the steps in the equivalence class if it can do at least one of them. For example, when a thread reads in a step some specific value from a memory location, then there are also steps that read any other value from the same location. Then, thread-wise downward simulations yield thread-wise upward simulations if the thread running the compiled code is determinate, i.e., all observable steps from any fixed (reachable) state are in the same equivalence. Using the same argument, upward simulations can be derived from downward simulations.

Receptiveness could be a way to avoid proving both simulation directions on the expression and thread level in JinjaThreads. It would require some changes to the interleaving semantics, though, to satisfy the receptiveness requirements. For example, *ThreadEx a True* and *Spawn a* (C, run, a) h as generated by StartFail and Start (Figure 14) would have to be in the same class. But the additional arguments (C, run, a) and h of *Spawn* cannot be determined from the former BTA, so *ThreadEx a True* is related to all BTAs of the form *Spawn a* _ _. Hence, the BTA *ThreadEx* would have to include C and h, which are not needed anywhere else. Similarly, other BTAs would need to be duplicated. For example, *UnlockFail*$\rightarrow a$ is used by method calls to *wait*, *notify*, and *notifyAll*. Clearly, each of these usages should be in a separate equivalence class, i.e., different versions of *UnlockFail* are needed. These modifications would thus weaken the abstraction boundary that the interleaving semantics provides as the specific BTA usage of the single-threaded semantics shows up in the abstract definitions.

The CakeML compiler [37] compiles a sequential, ML-like language to assembly code. It has been verified using the HOL4 prover against a big-step semantics. Owens et al. [66] found that a functional big-step semantics rather than a relational one simplifies the CakeML correctness proofs, because the functional style avoids duplication in rules and works well with rewriting. In JinjaThreads, duplication in the small-step rules is only a minor issue because most statements have only few cases (array update is the only exception) and Isabelle can automaticaly derive the equations for rewriting from inductive definitions.

8 Conclusion

In this article, I have focused on studying the effects of Java multithreading in a *unified* model rather than in isolation. This ensures that important details cannot be missed easily. Tractability quickly becomes a major concern and modularity is the key to push the limits. I have demonstrated how to disentangle sequential aspects, concurrency features, and the memory model, and the proofs show that JinjaThreads is indeed a usable model despite being sizeable.

However, the size and complexity make it non-trivial to ensure that JinjaThreads faithfully models (a subset of) Java. Rushby [72] and Norrish [61] 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 Java$^{\ell ight}$, μJava, and Jinja. Hence, numerous publications in various venues [8, 28–35, 45–48, 50, 51, 58–60, 62–65, 67, 74–77, 80, 81] and its reuse in [3, 57] support the claim of faithfulness. Type safety (Section 3) and the compiler verification (Section 5) are excellent examples for the second. Yet, bugs may still hide in technical details that publications gloss over and in areas that sanity theorems fail to cover. Therefore, I have extracted an executable interpreter, VM, and compiler from the formal definitions using Isabelle's code generator and validated the semantics by running Java test programs and comparing the results with Sun's reference implementation [51]. To make the vast supply of Java programs available for testing, I also developed a translator Java2Jinja from Java to JinjaThreads abstract syntax.[16] Testing revealed a bug in the implementation of binary operators that all previous proofs were unable to spot. Although validation through testing can never prove the absence of errors, I am now confident that JinjaThreads faithfully models the Java subset it covers.

Acknowledgements I thank my PhD supervisor Gregor Snelting for supporting this work, Jonas Thedering and Antonio Zea for their work on Java2Jinja, Lukas Bulwahn and Florian Haftmann for their help with the code generator, Jasmin Blanchette for his Nitpick support, and Christoph Sprenger and the anonymous revierers for their suggestions how to improve the presentation.

References

1. Aceto, L., van Glabbeek, R.J., Fokkink, W., Ingólfsdóttir, A.: Axiomatizing prefix iteration with silent steps. Information and Computation **127**(1), 26–40 (1996). DOI 10.1006/inco.1996.0047
2. Alves-Foss, J. (ed.): Formal Syntax and Semantics of Java, *LNCS*, vol. 1523. Springer (1999). DOI 10.1007/3-540-48737-9
3. Backes, M., Busenius, A., Hriţcu, C.: On the development and formalization of an extensible code generator for real life security protocols. In: A.E. Goodloe, S. Person (eds.) NFM 2012, *LNCS*, vol. 7226, pp. 371–387. Springer (2012). DOI 10.1007/978-3-642-28891-3_34
4. Ballarin, C.: Locales: A module system for mathematical theories. J. Autom. Reasoning **52**(2), 123–153 (2014). DOI 10.1007/s10817-013-9284-7
5. Barthe, G., Crégut, P., Grégoire, B., Jensen, T., Pichardie, D.: The MOBIUS proof carrying code infrastructure. In: F. de Boer, M. Bonsangue, S. Graf, W.P. de Roever (eds.) Formal Methods for Components and Objects, *LNCS*, vol. 5382, pp. 1–24. Springer (2008). DOI 10.1007/978-3-540-92188-2_1
6. Batty, M., Memarian, K., Nienhuis, K., Pichon-Pharabod, J., Sewell, P.: The problem of programming language concurrency semantics. In: J. Vitek (ed.) ESOP 2015, *LNCS*, vol. 9032, pp. 283–307. Springer (2015). DOI 10.1007/978-3-662-46669-8_12

[16] Available at `http://pp.info.uni-karlsruhe.de/projects/quis-custodiet/Java2Jinja/`

7. Belblidia, N., Debbabi, M.: A dynamic operational semantics for JVML. Journal of Object Technology **6**(3), 71–100 (2007)
8. Berghofer, S., Strecker, M.: Extracting a formally verified, fully executable compiler from a proof assistant. In: COCV 2003, *ENTCS*, vol. 82(2), pp. 377–394 (2003). DOI 10.1016/S1571-0661(05)82598-8
9. Bergstra, J.A., Klop, J.W., Olderog, E.R.: Failures without chaos: A new process semantics for fair abstraction. In: Formal Description of Programming Concepts III (IFIP 1987), pp. 77–103. Elsevier Science Publishing (1987)
10. Blanchette, J.C., Nipkow, T.: Nitpick: A counterexample generator for higher-order logic based on a relational model finder. In: M. Kaufmann, L.C. Paulson (eds.) ITP 2010, *LNCS*, vol. 6172, pp. 131–146. Springer (2010). DOI 10.1007/978-3-642-14052-5_11
11. Bogdănaş, D., Roşu, G.: K-Java: A complete semantics of Java. In: POPL 2015, pp. 445–456. ACM (2015). DOI 10.1145/2676726.2676982
12. Breitner, J., Graf, J., Hecker, M., Mohr, M., Snelting, G.: On improvements of low-deterministic security. In: F. Piessens, L. Viganò (eds.) POST 2016, *Lecture Notes in Computer Science*, vol. 9635, pp. 68–88. Springer Berlin Heidelberg (2016). DOI 10.1007/978-3-662-49635-0_4
13. Connected limited device configuration (CLDC) specification 1.1. `http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html`
14. Czarnik, P., Schubert, A.: Extending operational semantics of the Java bytecode. In: G. Barthe, C. Fournet (eds.) TGC 2008, *LNCS*, vol. 4912, pp. 57–72. Springer (2008)
15. Dave, M.A.: Compiler verification: a bibliography. SIGSOFT Software Engineering Notes **28**(6), 2–2 (2003)
16. Drossopoulou, S., Eisenbach, S.: Describing the semantics of Java and proving type soundness. In: J. Alves-Foss (ed.) Formal Syntax and Semantics of Java, *LNCS*, vol. 1523, pp. 542–542. Springer (1999). DOI 10.1007/3-540-48737-9_2
17. Farzan, A., Chen, F., Meseguer, J., Roşu, G.: Formal analysis of Java programs in JavaFAN. In: R. Alur, D. Peled (eds.) CAV 2004, *LNCS*, vol. 3114, pp. 501–505. Springer (2004). DOI 10.1007/978-3-540-27813-9_46
18. Farzan, A., Meseguer, J., Roşu, G.: Formal JVM code analysis in JavaFAN. In: C. Rattray, S. Maharaj, C. Shankland (eds.) AMAST 2004, *LNCS*, vol. 3116, pp. 132–147. Springer (2004). DOI 10.1007/978-3-540-27815-3_14
19. Flanagan, C., Freund, S.N., Lifshin, M., Qadeer, S.: Types for atomicity: Static checking and inference for Java. ACM Transactions on Programming Languages and Systems **30**(4), 1–53 (2008). DOI 10.1145/1377492.1377495
20. Giffhorn, D.: Slicing of concurrent programs and its application to information flow control. Ph.D. thesis, Fakultät für Informatik, Karlsruher Institut für Technologie (2012)
21. Gosling, J., Joy, B., Steele, G., Bracha, G., Buckley, A.: The Java Language Specification: Java SE 8 Edition. Oracle America, Inc. (2015)
22. Goto, M., Jagadeesan, R., Pitcher, C., Riely, J.: Types for relaxed memory models. In: TLDI 2012, pp. 25–38. ACM (2012). DOI 10.1145/2103786.2103791
23. Grossman, D.: Type-safe multithreading in Cyclone. In: TLDI 2003, pp. 13–25. ACM (2003). DOI 10.1145/604174.604177
24. Hartel, P.H., Moreau, L.: Formalizing the safety of Java, the Java virtual machine, and Java Card. ACM Computing Surveys **33**, 517–558 (2001). DOI 10.1145/503112.503115
25. Huisman, M., Petri, G.: BicolanoMT: a formalization of multi-threaded Java at bytecode level. In: BYTECODE 2008, Electronic Notes in Theoretical Computer Science (2008)
26. Java platform, standard edition 8 API specification (2014). `http://docs.oracle.com/javase/8/docs/api/`
27. Kildall, G.A.: A unified approach to global program optimization. In: POPL 1973, pp. 194–206. ACM (1973). DOI 10.1145/512927.512945
28. Klein, G.: Verified Java bytecode verification. Ph.D. thesis, Institut für Informatik, Technische Universität München (2003)
29. Klein, G., Nipkow, T.: Verified lightweight bytecode verification. Concurrency and Computation: Practice and Experience **13**(13), 1133–1151 (2001)
30. Klein, G., Nipkow, T.: Verified bytecode verifiers. Theoretical Computer Science **298**(3), 583–626 (2002). DOI 10.1016/S0304-3975(02)00869-1
31. Klein, G., Nipkow, T.: Jinja is not Java. In: G. Klein, T. Nipkow, L.C. Paulson (eds.) The Archive of Formal Proofs. `http://www.isa-afp.org/entries/Jinja.shtml` (2005). Formal proof development
32. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Transactions on Programming Languages and Systems **28**(4), 619–695 (2006). DOI 10.1145/1146809.1146811

33. Klein, G., Nipkow, T., von Oheimb, D., Nieto, L.P., Schirmer, N., Strecker, M.: Java source and bytecode formalizations in Isabelle: Bali. Isabelle sources in Isabelle/HOL/Bali (2002)
34. Klein, G., Strecker, M.: Verified bytecode verification and type-certifying compilation. Journal of Logic and Algebraic Programming **58**(1–2), 27–60 (2004). DOI 10.1016/j.jlap.2003.07.004
35. Klein, G., Wildmoser, M.: Verified bytecode subroutines. Journal of Automated Reasoning **30**(3–4), 363–398 (2003). DOI 10.1023/A:1025095122199
36. Krebbers, R.: The C standard formalized in Coq. Ph.D. thesis, Radboud University Nijmegen (2015)
37. Kumar, R., Myreen, M.O., Norrish, M., Owens, S.: CakeML: A verified implementation of ML. In: POPL 2014, pp. 179–191. ACM, New York, NY, USA (2014). DOI 10.1145/2535838.2535841
38. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **28**(9), 690–691 (1979)
39. Leroy, X.: Formal certification of a compiler backend or: Programming a compiler with a proof assistant. In: POPL 2006, pp. 42–54. ACM (2006). DOI 10.1145/1111037.1111042
40. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning **43**(4), 363–446 (2009)
41. Leroy, X., Blazy, S.: Formal verification of a C-like memory model and its uses for verifying program transformations. Journal of Automated Reasoning **41**(1), 1–31 (2008). DOI 10.1007/s10817-008-9099-0
42. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, Second Edition. Addison-Wesley (1999)
43. Liu, H., Moore, J.S.: Executable JVM model for analytical reasoning: A study. In: IVME 2003, pp. 15–23. ACM (2003). DOI 10.1145/858570.858572
44. Liu, H., Moore, J.S.: Java program verification via a JVM deep embedding in ACL2. In: K. Slind, A. Bunker, G. Gopalakrishnan (eds.) TPHOLs 2004, *LNCS*, vol. 3223, pp. 117–125. Springer (2004). DOI 10.1007/978-3-540-30142-4_14
45. Lochbihler, A.: Jinja with threads. In: G. Klein, T. Nipkow, L.C. Paulson (eds.) The Archive of Formal Proofs. `http://www.isa-afp.org/entries/JinjaThreads.shtml` (2007). Formal proof development
46. Lochbihler, A.: Type safe nondeterminism - a formal semantics of Java threads. In: Proceedings of the 2008 International Workshop on Foundations of Object-Oriented Languages (FOOL 2008) (2008)
47. Lochbihler, A.: Verifying a compiler for Java threads. In: A.D. Gordon (ed.) ESOP 2010, *LNCS*, vol. 6012, pp. 427–447. Springer (2010). DOI 10.1007/978-3-642-11957-6_23
48. Lochbihler, A.: Java and the Java memory model – a unified, machine-checked formalisation. In: H. Seidl (ed.) ESOP 2012, *LNCS*, vol. 7211, pp. 497–517. Springer (2012)
49. Lochbihler, A.: A machine-checked, type-safe model of Java concurrency : Language, virtual machine, memory model, and verified compiler. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2012). DOI 10.5445/KSP/1000028867
50. Lochbihler, A.: Making the Java memory model safe. ACM Trans. Program. Lang. Syst. **35**(4), 12:1–12:65 (2014). DOI 10.1145/2518191
51. Lochbihler, A., Bulwahn, L.: Animating the formalised semantics of a Java-like language. In: M. van Eekelen, H. Geuvers, J. Schmalz, F. Wiedijk (eds.) ITP 2011, *LNCS*, vol. 6898, pp. 216–232. Springer (2011). DOI 10.1007/978-3-642-22863-6_17
52. McLean, J.: A general theory of composition for a class of "possibilistic" properties. IEEE Transactions on Software Engineering **22**(1), 53–67 (1996). DOI 10.1109/32.481534
53. Milner, R.: A modal characterisation of observable machine-behaviour. In: E. Astesiano, C. Böhm (eds.) CAAP 1981, *LNCS*, vol. 112, pp. 25–34. Springer (1981). DOI 10.1007/3-540-10828-9_52
54. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
55. Mobius consortium. Deliverable D3.1. Byte code level specification language and program logic (2006)
56. Moore, J.S., Porter, G.: The apprentice challenge. ACM Transactions on Programming Languages and Systems **24**(3), 193–216 (2002). DOI 10.1145/514188.514189
57. Moser, G., Schaper, M.: From Jinja bytecode to term rewriting: A complexity reflecting transformation. Information and Computation (To appear.)
58. Nipkow, T.: Verified bytecode verifiers. In: F. Honsell, M. Miculan (eds.) FOSSACS 2001, *LNCS*, vol. 2030, pp. 347–363. Springer (2001). DOI 10.1007/3-540-45315-6_23
59. Nipkow, T., von Oheimb, D.: Java$_{\ell ight}$ is type-safe — definitely. In: POPL 1998, pp. 161–170. ACM (1998). DOI 10.1145/268946.268960
60. Nipkow, T., von Oheimb, D., Pusch, C.: μJava: Embedding a programming language in a theorem prover. In: F.L. Bauer, R. Steinbrüggen (eds.) Foundations of Secure Computation, *NATO Science Series F: Computer and Systems Sciences*, vol. 175, pp. 117–144. IOS Press (2000)
61. Norrish, M.: A formal semantics for C++. Tech. rep., NICTA (2008). Available from `http://nicta.com.au/people/norrishm/attachments/bibliographies_and_papers/C-TR.pdf`.

62. Oheimb, D.v.: Analyzing Java in Isabelle/HOL. formalization, type safety and hoare logic. Ph.D. thesis, Fakultät für Informatik, Technische Universität München (2000). URL http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2001/oheimb.html
63. Oheimb, D.v.: Hoare logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience **13**(13), 1173–1214 (2001)
64. Oheimb, D.v., Nipkow, T.: Machine-checking the Java specification: Proving type-safety. In: J. Alves-Foss (ed.) Formal Syntax and Semantics of Java, *LNCS*, vol. 1523, pp. 119–156. Springer (1999)
65. Oheimb, D.v., Nipkow, T.: Hoare logic for NanoJava: Auxiliary variables, side effects and virtual methods revisited. In: L.H. Eriksson, P. Lindsay (eds.) FME 2002, *LNCS*, vol. 2391, pp. 89–105. Springer (2002). DOI 10.1007/3-540-45614-7_6
66. Owens, S., Myreen, M.O., Kumar, R., Tan, Y.K.: Functional big-step semantics. In: P. Thiemann (ed.) ESOP 2016, *LNCS*, vol. 9632, pp. 589–615. Springer (2016). DOI 10.1007/978-3-662-49498-1_23
67. Pusch, C.: Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In: R. Cleaveland (ed.) TACAS 1999, *LNCS*, vol. 1579, pp. 89–103. Springer (1999). DOI 10.1007/3-540-49059-0_7
68. Quis custodiet – machine-checked software security analyses. https://pp.info.uni-karlsruhe.de/projects/quis-custodiet/
69. Ramananandro, T., Dos Reis, G., Leroy, X.: Formal verification of object layout for C++ multiple inheritance. In: POPL 2011, pp. 67–80. ACM (2011). DOI 10.1145/1926385.1926395
70. Rittri, M.: Proving the correctness of a virtual machine by a bisimulation. Licentiate thesis, Göteborg University (1988)
71. Roşu, G., Şerbănuţă, T.F.: An overview of the K semantic framework. Journal of Logic and Algebraic Programming **79**(6), 397–434 (2010). DOI 10.1016/j.jlap.2010.03.012
72. Rushby, J.: Formal methods and the certification of critical systems. Tech. Rep. SRI-CSL-93-7, Computer Science Laboratory, SRI International (1993). URL http://www.csl.sri.com/papers/csl-93-7/
73. Sampson, J., Boehm, H., Otenko, O., Levart, P., Holmes, D., Haley, A., Buchholz, M., Lea, D., Davidovich, V., Terekhov, A., Diestelhorst, S.: Varieties of CAS semantics (another doc fix request). Thread the on concurrency-interest mailing list, first post at http://altair.cs.oswego.edu/pipermail/concurrency-interest/2015-January/013613.html (2015)
74. Schirmer, N.: Java definite assignment in Isabelle/HOL. In: Proceedings of ECOOP Workshop on Formal Techniques for Java-like Programs. Technical Report 408, ETH Zurich (2003)
75. Schirmer, N.: Analysing the Java package/access concepts in Isabelle/HOL. Concurrency and Computation: Practice & Experience - Formal Techniques for Java-like Programs **16**, 689–706 (2004). DOI 10.1002/cpe.v16:7
76. Schirmer, N.: A verification environment for sequential imperative programs in Isabelle/HOL. In: F. Baader, A. Voronkov (eds.) LPAR 2004, *Lecture Notes in Artificial Intelligence*, vol. 3452, pp. 398–414. Springer (2005)
77. Schirmer, N.: Verification of sequential imperative programs in Isabelle/HOL. Ph.D. thesis, Technische Universität München (2006)
78. Ševčík, J., Vafeiadis, V., Zappa Nardelli, F., Jagannathan, S., Sewell, P.: CompCertTSO: A verified compiler for relaxed-memory concurrency. J. ACM **60**(3), 22:1–22:50 (2013). DOI 10.1145/2487241.2487248
79. Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Springer (2001)
80. Strecker, M.: Formal verification of a Java compiler in Isabelle. In: CADE 2002, *LNCS*, vol. 2392, pp. 63–77. Springer (2002)
81. Strecker, M.: Investigating type-certifying compilation with Isabelle. In: M. Baaz, A. Voronkov (eds.) LPAR 2002, *LNCS*, vol. 2514, pp. 403–417. Springer (2002). DOI 10.1007/3-540-36078-6_27
82. Tool-assisted specification and verification of javacard programmes: Verificard. http://cordis.europa.eu/project/rcn/53643_en.html
83. Wagner, D., et al. (eds.): Ausgezeichnete Informatik-Dissertationen 2003, *LNI*, vol. D-3. Köllen Verlag (2003)
84. Wand, M.: Compiler correctness for parallel languages. In: FPCA 1995, pp. 120–134. ACM (1995)
85. Wasserrab, D.: From formal semantics to verified slicing – a modular framework with applications in language based security. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik (2010)
86. Wasserrab, D., Lohner, D.: Proving information flow noninterference by reusing a machine-checked correctness proof for slicing. In: 6th International Verification Workshop (VERIFY 2010) (2010)
87. Wright, A.K., Felleisen, M.: A Syntactic Approach to Type Soundness. Information and Computation **115**(1), 38–94 (1994)