Verifying a Compiler for Java Threads*

Andreas Lochbihler

Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany
andreas.lochbihler@kit.edu

Abstract. A verified compiler is an integral part of every security infrastructure. Previous work has come up with formal semantics for sequential and concurrent variants of Java and has proven the correctness of compilers for the sequential part. This paper presents a rigorous formalisation (in the proof assistant Isabelle/HOL) of concurrent Java source and byte code together with an executable compiler and its correctness proof. It guarantees that the generated byte code shows exactly the same observable behaviour as the semantics for the multithreaded source code.

1 Introduction

In a recent "research highlights" article in CACM [14], the CompCert C compiler [13] by Leroy was praised as follows: "I think we are on the verge of a new paradigm for safety-critical systems, where we rely upon formal, machine checked verification, instead of human audits. Leroy's compiler is an impressive step toward this goal." And indeed, Leroy's work can be seen as a door opener, in particular because his verification includes various optimisations and generates assembler code for a real machine. However, concurrent programs call for formal methods even louder, because many bugs show up only in some interleavings of the threads' executions, which makes the bugs nearly impossible to find and reproduce. But so far, nobody has ever never included the compilation of thread primitives and multithreaded (imperative) programs.

In this paper, we present a compiler from a substantial subset of multithreaded Java source code to byte code and show semantic preservation w.r.t. interleaving semantics in the proof assistant Isabelle/HOL [23]. To our knowledge, this is the first verified compiler for a realistic concurrent language. The verification addresses the fundamental challenges of concurrency: nondeterministic interleaving and different granularity of atomic operations between source and byte code. At present, we ignore the Java Memory Model (JMM) and assume sequential consistency. Like Sun's `javac` compiler, ours does not optimise. Thus, we expect that the verification also works for the full JMM.

We show how to address nondeterminism by applying a bisimulation approach (Sec. 3) like in [24,27] to a compiler for a realistic concurrent language. We cope with interleaving by decomposing the correctness proof for the compiler into a correctness proof for individual threads. To that end, we introduce a generic

* This work is partially supported by DFG grant Sn 11/10-1.

framework semantics which interleaves the individual threads and manages locks and wait sets. Since the observable behaviour includes all accesses to shared memory, method calls and synchronisation, we obtain a bisimulation for the multithreaded semantics from bisimulations for single threads. Bisimulation also solves the atomicity issue for us: unobservable steps may be decomposed into arbitrarily many unobservable steps, observable ones into multiple unobservable ones followed by an observable one.

We have based our work on the Jinja project [11], which contains formal semantics and a verified compiler for sequential Java. Our semantics JinjaThreads (Sec. 2) adds multithreading and concurrency primitives for arbitrary dynamic thread creation, synchronisation on monitors, the wait-notify mechanism and joining of threads. Our compiler (Sec. 4) may seem a straightforward extension of Jinja's, but its verification (Sec. 5) posed two fundamental challenges: In striking contrast to Jinja's big-step semantics and simulation-only proof, we had to (i) verify the compiler w.r.t. small-step interleaving semantics and (ii) show both directions of the bisimulation. Accordingly, our verification comprises 47kL of Isabelle code, whereas the original Jinja verification needed only 15kL. Finally, we discuss our design decisions that enabled the verification to succeed (Sec. 6).

Using Isabelle's code generator, we have mechanically extracted an executable implementation of our compiler in standard ML. It compiles source code programs in abstract syntax to byte code programs in abstract syntax. The full formalisation of JinjaThreads with all details is available online [19].

2 Jinja with Threads

In this section, we present the features of JinjaThreads that are relevant for our compiler verification. JinjaThreads is a complex model of Java that supports a broad spectrum of concepts, all of which must be correctly handled by the compiler all the way from source code to byte code: local variables, objects and fields, inheritance, dynamic dispatch and recursion, arrays and exception handling; for details see [11,18]. Here, we focus on Java's concurrency language features as specified in the Java Language Specification (JLS) [8, Ch. 17]: synchronisation via locks, the *wait-notify* mechanism, thread creation and joining. The interrupt mechanism is not modelled, but could be added at little cost to the formalisation. Fig. 1 illustrates the life cycle of a Java thread. After a thread has been spawned by invoking its *start* method, it keeps running (i) until it is *final*. If, however, it invokes the *wait* method on an object o, it temporarily releases its locks on o's monitor and is entered in o's wait set (ii). If another thread calls *notify* on o, a thread t is removed from o's wait set (iii), but t must reacquire its locks before it can continue to run.

Java source and byte code have the same thread and concurrency model, which is captured in our multithreaded semantics (Sec. 2.1). We use it as an interleaving semantics for all languages in JinjaThreads: source code J, (Sec. 2.3), byte code *JVM* (Sec. 2.4), and one intermediate language J_1 (Sec. 2.5).

2.1 The Framework Semantics

In this section, we present the multithreaded semantics for all JinjaThreads languages. To that end, we assume a small-step semantics for single threads, written $\langle x, h\rangle \xrightarrow{ta}\triangleright \langle x', h'\rangle$, which contains all atomic execution steps for the individual threads. It takes a thread-local state x and the shared heap h: the result are a new thread-local state x' and heap h', and a thread action ta which spawns a new thread, locks a monitor, joins another thread, etc. For thread joining, we use the predicate *final* to identify thread states that have terminated. Then, the **framework semantics** *redT* takes such a semantics as a parameter to form the set of small-step reductions for the multithreaded case. Here, we only give a short summary of it, see [18] for full details.

A **multithreaded state** s consists of four components: (i) The **lock state** *locks* s stores for every monitor how many times it is locked by a thread, if at all. Locks are mutually exclusive. (ii) The **thread state** *thr* s stores for every thread t in s all information that is specific to it, i.e. the thread-local state x and the multiset ln of locks on monitors that this thread has temporarily released, e.g. when it was suspended to a wait set. (iii) The **shared memory** *shr* s. (iv) The **wait sets** *wset* s: a thread is in at most one wait set at a time.

A single thread t uses the thread action ta to query and update the state of the locks, threads and wait sets. Currently, the framework semantics provides the following **basic thread actions**: (i) Locking, unlocking, temporarily releasing, and testing whether it has (not) locked a monitor. (ii) Creating a new thread, testing whether a thread has been started, and joining a thread. (iii) Suspending to a wait set, notifying one (all) threads in a wait set. A **thread action** (**TA**) ta consists of multiple basic thread actions, written as a list ($[]$ denotes the empty list). The whole list is checked and executed atomically. A call to the *wait* method, e.g., issues [*HasLock* l, *Release* l, *Suspend* w] to test whether the thread t has locked the monitor l, to temporarily release all locks on l, and to suspend itself to the wait set w. By composing TAs that affect multiple aspects of the multithreaded state from basic thread actions, we were able to keep the framework semantics flexible and the proofs about it simple.

The **framework semantics** *redT* has reductions (written $s \xrightarrow{t \triangleright ta}\triangleright s'$) of two kinds. First, a reduction $\langle x, h\rangle \xrightarrow{ta}\triangleright \langle x', h'\rangle$ of the thread t in s that is not in a wait set and has not temporarily released any locks (state (i) in Fig. 1). In that case, tests of the TA ta must hold in s. Then, *redT* atomically executes ta on s and updates t's local state to x' and the shared heap to h', which yields s'. Second, *redT* can choose a thread t in s that is not in a wait set, but has temporarily released some locks ln – state (iii) in Fig. 1. If t can reacquire all of them, *redT* assigns them to t again and resets ln to $\{\}$ in s'. In that case, everything else remains unchanged from s to s'. A reduction $s \xrightarrow{t \triangleright ta}\triangleright s'$ is unobservable (written *is-m*τ s (t, ta) s') iff it results from a τ-move of a thread (cf. Sec. 2.2). Note that a thread gets from (ii) to (iii) in Fig. 1 only if an *other* thread notifies it.

We also lift the *final* predicate to a multithreaded state s: s is **final** iff for all threads t in s, say *thr* s $t = \lfloor(x, ln)\rfloor$ ($\lfloor\_\rfloor$ denotes definedness for a partial function), t's local state x is *final* and t has not temporarily released any locks, i.e. $ln = \{\}$.

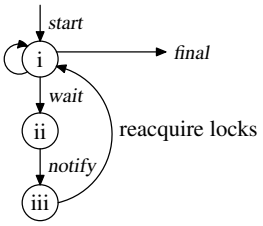

Fig. 1. Life cycle of a Java thread

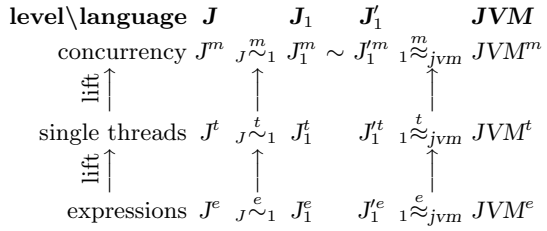

Fig. 2. Composition of bisimulations for the correctness proof.

2.2 Concepts for all Languages in JinjaThreads

Each JinjaThreads language has three semantics levels, which Fig. 2 shows together with the delay bisimulations used in Sec. 5 for the verification. The expression level semantics, which is marked with ^{e}, contains all execution steps of a single thread except for method calls and returns. The semantics for a thread (marked ^{t}) lifts the expression level semantics to call stacks and adds method calls and returns. The multithreaded semantics (marked ^{m}) models the full behaviour for multithreaded programs. In all languages, this is the framework semantics instantiated with the call-stack semantics for single threads.

There are five kinds of **values**: booleans *Bool b*, integers *Intg i*, addresses *Addr a*, the null reference *Null*, and a dummy value *Unit*. Addresses reference objects or arrays on the heap, which is a map from addresses to heap objects. To avoid redundancies with the instruction for object creation in the formalisation, all system exceptions (like *NullPointer* and *IllegalMonitorState*) are preallocated on the heap. &_ denotes the address of a preallocated system exception.

In standard Java, only monitors – of which every object and array has exactly one – can be locked. Hence, addresses identify monitors in the framework instantiations. Since every monitor manages its own wait set, addresses also identify wait sets. JinjaThreads uses the same heap representation in all languages. Hence, every instantiation need only to specify the thread-local state.

All JinjaThreads languages use the same format for class and method declarations, only the method definitions depend on the language. Hence, a compiler *Comp* need to be specified only for method bodies. The generic function *compP* then uniformly applies *Comp* to all methods of all classes, i.e. the program *P* is compiled to *compP Comp P*. This generic approach ensures that compilation does not affect the class hierarchy and lookup functions for methods and fields.

JinjaThreads comes with standard well-formedness conditions (e.g. typeability, acyclic class hierarchy), see [11] for details. In the following, we will assume that all programs are well-formed.

JinjaThreads has two kinds of method calls: First, standard calls to methods that are implemented in the program *P*. Second, calls to methods that cannot be implemented in JinjaThreads syntax, e.g. native methods such as *wait*, *notify*, or *start* on *Thread* objects. We call them **external calls** and hardwire them in the semantics. Such a call executes atomically, written $P \vdash \langle a.M(vs),\ h\rangle \xrightarrow{ta}_{ext} \langle va,\ h'\rangle$

where a is the address of the object, M the method name and vs the list of parameter values. It returns va, which is either a normal value v or the address a of an exception, a thread action ta, and the new heap h'. Currently, the following native methods are provided: *wait*, *notify* and *notifyAll* implement the wait set mechanism for all objects and arrays; they simply translate the call into the TAs [*HasLock* a, *Release* a, *Suspend* a], [*HasLock* a, *Notify* a], and [*HasLock* a, *NotifyAll* a], resp., where a is the address of the object or array being called. For these methods, additional reductions with the TA [*HasNoLock* a] raise an *IllegalMonitorState* exception. The framework semantics selects the right reduction according to its lock status. Moreover, method *start* in class *Thread* spawns a new thread, or fails with an *IllegalThreadState* exception if the thread has already been started before. Finally, *join* joins the called thread. Via this mechanism, we could add more native methods and even model I/O easily.

Regarding the **observable behaviour**, we consider the following operations as observable moves: calling and returning from a method, locking and unlocking, creating objects and arrays and accessing data on the heap other than type information.[1] Since thread creation, joining and the wait-notify mechanism are implemented as external calls, all of them are, in particular, observable. Conversely, all control flow constructs, including exception throwing and handling, and local variable manipulation are only relevant to the thread that executes them, so these generate only τ-moves.

2.3 The Source Code Language *J*

In the source language J, everything is an expression with a return value: statements are treated as expressions that return *Unit*. An expression is *final* if it is either a value *Val* v (normal termination) or a thrown exception *throw* (*Addr* a), which we abbreviate as *Throw* a. For a program P, let $P \vdash \langle e,(h, xs)\rangle \xrightarrow{ta}_J \langle e',(h', xs')\rangle$ denote that the expression e executes in a single step to e' with TA ta, thereby changing the heap h to h' and the store for local variables from xs to xs'. J^e contains 84 reduction rules, but we only show those for synchronisation. For details on the syntax and the full sequential semantics, see [11,18].

Synchronisation in Java source code is done via the synchronized statement, which is specified in the JLS [8, Sec. 14.19]. The synchronized modifier for methods behaves as if its body was statement-synchronized on *this*, so we only need to consider synchronized blocks in J. Fig. 3 shows the reduction rules for the synchronized statement *sync*: JS1 reduces the monitor subexpression. If it raises an exception, rule JS2 propagates it.[2] If the monitor subexpression evaluates

[1] These observable moves strictly include all JMM inter-thread actions except for thread divergence actions. We omit the latter because the JMM is inconsistent for infinite executions [3]. Object creation, e.g., must be observable in our approach, because it changes the heap, but it is no inter-thread action: The JMM assumes that all objects have been preallocated, which is unrealistic for an actual semantics.

[2] This is a typical example of how J^e handles exceptions: For every language construct, rules propagate thrown exceptions (*Throw* a) from subexpressions until a matching *try-catch* block is reached or there are no surrounding expressions any more.

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{ta}_J \langle e',s'\rangle}{P \vdash \langle \mathit{sync}(e)\ e_2,s\rangle \xrightarrow{ta}_J \langle \mathit{sync}(e')\ e_2,s'\rangle}\ \textbf{JS1}$$

$$P \vdash \langle \mathit{sync}(\mathit{Throw}\ a)\ e,s\rangle \xrightarrow{[]}_J \langle \mathit{Throw}\ a,s\rangle\ \textbf{JS2}$$

$$P \vdash \langle \mathit{sync}(\mathit{null})\ e,s\rangle \xrightarrow{[]}_J \langle \mathit{Throw}\ \&\mathit{NullPointer},s\rangle\ \textbf{JS3}$$

$$P \vdash \langle \mathit{sync}(\mathit{addr}\ a)\ e,s\rangle \xrightarrow{[\mathit{Lock}\ a]}_J \langle \mathit{insync}(a)\ e,s\rangle\ \textbf{JS4}$$

$$\frac{P \vdash \langle e,s\rangle \xrightarrow{ta}_J \langle e',s'\rangle}{P \vdash \langle \mathit{insync}(a)\ e,s\rangle \xrightarrow{ta}_J \langle \mathit{insync}(a)\ e',s'\rangle}\ \textbf{JS5}$$

$$P \vdash \langle \mathit{insync}(a)\ \mathit{Val}\ v,s\rangle \xrightarrow{[\mathit{Unlock}\ a]}_J \langle \mathit{Val}\ v,s\rangle\ \textbf{JS6}$$

$$P \vdash \langle \mathit{insync}(a)\ \mathit{Throw}\ ad,s\rangle \xrightarrow{[\mathit{Unlock}\ a]}_J \langle \mathit{Throw}\ ad,s\rangle\ \textbf{JS7}$$

Fig. 3. Source code reductions for the synchronized statement.

to the *null* value, a *NullPointer* exception is thrown (JS3). If it reduces to some monitor address a, the thread can only reduce further (JS4) by acquiring a lock on a. To remember that the lock has been granted, the expression is rewritten to $\mathit{insync}(a)\ e$, a variant of the *sync* expression that does not occur in programs. Then, the synchronized block's body is executed (JS5). If this terminates normally with a value v or with an exception at address ad, JS6 and JS7 release the lock on a and propagate the return value or exception.

J^e also includes all external calls into J^e, but it has no rule for standard method calls. It uses the predicate *is-ext-call* to determine, based on type information, whether the call is external. Standard method calls are left to the semantics J^t, which lifts J^e to call stacks. The lifting is standard: as long as the frame's expression at the top of the call stack is not *final*, it is being reduced according to J^e. In case of a standard method call, J^t pushes a new call frame with the called method's body as expression on top of the stack. If the top frame's expression is *final*, the return value or thrown exception replaces the method call subexpression in the frame below. A thread in J^t is final iff the call stack contains only one expression, which is also *final*.

Originally, the JinjaThreads source code small-step semantics [18] did not model a call stack and dynamically inlined method calls in the expressions instead. But the compiler verification requires an explicit call stack, so we use this alternative semantics. We have also shown that they are strongly bisimilar: The strong bisimulation relates the call stack es to the expression e' iff folding es with method inlining equals e'.

2.4 The Byte Code Language *JVM*

The byte code language and the JinjaThreads virtual machine (VM) model Java byte code and the Java VM according to the Java Virtual Machine Specification (JVMS) [16]. A thread-local state (xcp, frs) consists of an exception flag xcp ($\lfloor a \rfloor$ corresponds to *Throw* a in J and $\bot$ denotes none), and a stack frs of frames. A frame $fr = (stk, loc, C, M, pc)$ consists of the stack stk, an array loc for the

parameters and local variables, the class C and method name M of the method, and the program counter *pc*. A state is *JVM-final* iff the frame stack is empty.

A method body (*msl*, *mxs*, *ins*, *xt*) consists of an instruction list *ins*, an exception table *xt*, the maximum stack length *msl* and the size *mxs* of the array for local variables. The exception table is a list of entries (*from*, *to*, *C*, *pc*, *d*) where *C* is either a class name or the special value *Any*. The exception handler starting at index *pc* in *ins* expects *d* elements on the stack and handles exceptions that are raised by instructions in the interval [*from*, *to*). If *C* is a class name, it handles only those that are a subclass of *C*; if *C* is *Any*, it handles all.

Regarding the JinjaThreads' instruction set, Java byte code instructions which only differ on their operand types (e.g. `iload` and `aload`) are combined in polymorphic ones (e.g. *Load*), but the instructions have not been simplified conceptually. Moreover, operations that directly manipulate the stack (e.g. `dup`) or the local variables like `iinc` are not part of the Jinja VM. Since they are all silent instructions, our silent instructions can easily simulate them.

The semantics of a single instruction is defined by the function *exec-instr*. Given the instruction, the heap and the frame stack, it produces a list of successor states together with the corresponding TAs. Like for *J*, we only explain method invocation, synchronisation and exception handling.

Method calls are very similar to *J*: the *Invoke* instruction decides via the predicate *is-ext-call* whether the call is external. If so, it uses the reductions from $P \vdash \langle a.M(vs), h\rangle \xrightarrow{ta}_{ext} \langle va, h'\rangle$ to determine the successor states. Otherwise, it looks up the method in *P* and pushes a new call frame on top of the frame stack with the parameters and local variables correctly initialised.

The instructions *MEnter* and *MExit* for entering and exiting a monitor implement synchronisation. Both throw a *NullPointer* exception if the top stack element *v* is *null*. Otherwise, they increment the program counter and issue a *Lock* or *Unlock* action on the address *a* in *v*, resp. Additionally, *MExit* can also raise an *IllegalMonitorState* exception with the TA [*HasNoLock* *a*]. The latter possibility is to allow for unstructured locking, where unlocking may fail.

The function *exec P* (*xcp*, *h*, *fr*·*frs*) incorporates exception handling in the semantics: If no exception is flagged, this just executes the current instruction via *exec-instr*. Otherwise ($xcp = \lfloor a \rfloor$), *a* is checked against the exception handlers for the program counter of *fr*: If one is found, the stack is trimmed to the length specified in the exception table, *a* is pushed onto the stack and the program counter is set to the start of the handler. Otherwise, *fr* is popped and *a* is rethrown at the *Invoke* statement in the previous call frame.

The VM model *exec* is aggressive: it assumes that there are always sufficiently many operands of the right types on the stack. If not, the result is undefined. JinjaThreads also contains a defensive VM, which performs such checks and raises a type error in case they fail. The byte code verifier, which is also part of JinjaThreads, ensures that for verified byte code programs and conform states, the type checks are always met and no type errors occur, i.e. aggressive and defensive VM agree. A separate proof shows (using a type compiler) that the byte code verifier accepts all programs generated by the JinjaThreads compiler [11].

$$\frac{V < |xs_1| \qquad xs_1' = xs_1[V := Addr\ a]}{P \vdash \langle sync_V(addr\ a)\ e_1,\ (h,\ xs_1)\rangle \xrightarrow{[Lock\ a]}_{J_1} \langle insync_V(a)\ e_1,\ (h,\ xs_1')\rangle}\ \mathbf{J_1S4}$$

$$\frac{V < |xs_1| \qquad xs_{1\,[V]} = Addr\ a'}{P \vdash \langle insync_V(a)\ Throw\ ad,\ (h,\ xs_1)\rangle \xrightarrow{[Unlock\ a']}_{J_1} \langle Throw\ ad,\ (h,\ xs_1)\rangle}\ \mathbf{J_1S7}$$

$$\frac{V < |xs_1| \qquad xs_{1\,[V]} = Null}{P \vdash \langle insync_V(a)\ Val\ v,\ (h,\ xs_1)\rangle \xrightarrow{[]}_{J_1'} \langle Throw\ \&NullPointer,\ (h,\ xs_1)\rangle}\ \mathbf{J_1'S8}$$

$$\frac{V < |xs_1| \qquad xs_{1\,[V]} = Addr\ a'}{P \vdash \langle insync_V\ (a)\ Val\ v,\ (h,\ xs_1)\rangle \xrightarrow{[HasNoLock\ a']}_{J_1'} \langle Throw\ \&IllegalMonitorState,\ (h,\ xs_1)\rangle}\ \mathbf{J_1'S9}$$

Fig. 4. Example reduction rules for *sync* statements in J_1 and J_1'.

In the compiler verification, we mostly use the defensive VM for the bisimulation proof. As before, we have three levels of semantics: JVM_d^e (JVM_a^e) contains all execution steps of the defensive (aggressive) VM that manipulate only the top frame on the call stack, i.e. all instructions except for *Invoke* and *Return*, including method-local exception handling. The single-threaded VM semantics JVM^t also includes the execution steps that JVM_d^e has omitted. Then, the multi-threaded VM JVM^m is again the framework semantics instantiated with JVM^t.

2.5 Local Variables in an Array: the Intermediate Language J_1

Our compiler operates in two stages: The first stage allocates local variables to array indices, the second generates the byte code instructions. The intermediate language J_1 stores local variable values in an array (like byte code does), but the expressions from the source code have not yet been replaced by instructions. Hence, local variables in J_1 are no longer identified by their name, but by an index in the array. A $sync_V(e_1)\ e_1'$ block is now annotated with a variable index V. Following the JVMS [16, Sec. 7.14], this variable will be used in the byte code to store the monitor address between the *MEnter* and *MExit* instructions that implement the monitor locking and unlocking. Since J_1 behaves like the byte code w.r.t. local variables, J_1 already uses this local variable for *sync* blocks.

We define a new semantics J_1^e for expressions (written $P \vdash \langle e_1,\ (h,\ xs_1)\rangle \xrightarrow{ta}_{J_1} \langle e_1',\ (h',\ xs_1')\rangle$) with new rules for the expressions that operate on the variable array xs_1. In J_1^e, J_1S4 and J_1S7 (Fig. 4), e.g., replace JS4 and JS7. J_1S4 not only locks the monitor, but also stores its address in the variable array xs_1. (The first premise ensures that the variable index does not exceed the size of the array.) Accordingly, J_1S7 reads the monitor address for unlocking from the array.

Analogously to J, J_1^t lifts J_1^e to call stacks, which are again the thread-local states for the multithreaded semantics J_1^m. Like J^m and JVM^m, J_1^m is the framework semantics instantiated with J_1^t.

To be in line with the *MExit* semantics, we introduce a variant $J_1'^e$ of the J_1^e semantics. Apart from the reductions from J_1^e, it also includes in the un-

locking for $sync_V$ blocks the cases where the entry V in xs_1 (written $xs_{1[V]}$) is *Null* or the thread does not hold the lock on the monitor at the address $xs_{1[V]}$. In these cases, it raises a *NullPointer* or *IllegalMonitorState* exception resp. J'_1S8 and J'_1S9 in Fig. 4 show these if the block has terminated normally with a value v. J'^e_1 contains analogous reductions for abnormal termination with an exception *Throw a*. As above, J'^t_1 lifts J'^e_1 to call stacks and J'^m_1 is the framework semantics instantiated with J'^t_1.

3 Semantic Preservation via Bisimulations

We now introduce the notion of semantic preservation (Sec. 3.1) and the bisimulation infrastructure (Sec. 3.2 and 3.3) for showing preservation for the compiler.

3.1 Semantic Preservation

Semantic preservation aims to show that semantic properties established on the source code also hold for the target code and vice versa. Such properties or specifications (e.g. a safety property like no null pointer exceptions) are typically modelled as predicates on the traces of observable behaviour, i.e. the observable steps of a program execution, or on the sets of possible traces (for nondeterministic programs). Thus, a correct compiler *Comp* must ensure that the (sets of) traces of the source program P and of the compiled program *Comp* P are equal.

Formally, *Comp* **preserves the semantics** of P iff the following holds: Let s_1 and s_2 be the initial states for P and *Comp* P, resp. For every execution of P that starts in s_1 and terminates in s'_1, there must be an execution of *Comp* P from s_2 to s'_2 such that both the executions' traces and the observable data in s'_1 and s'_2 (such as the result values or exceptional termination) are the same. Conversely, every terminating execution of *Comp* P from s_2 must be matched that way by one of P from s_1.

As multithreaded programs are inherently nondeterministic, both directions are essential. The compiled code must not miss any observable nondeterministic choice, neither may it introduce additional observable behaviour. Some atomic high-level statements are translated into a sequence of simple instructions, which allow more interleavings. A correct compiler must ensure that these new interleavings do not lead to new behaviours. Conversely, some constructs (like exception handling) are atomic in the compiled code, but require many steps in the source code semantics. Although the compiled code has less interleavings, no observable behaviour must be missed.

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 and byte code may have different behaviours under this scheduler. In this sense, semantic preservation means: If there is a scheduler for P such that s_1 terminates in s'_1 with trace t, then there is also a scheduler for *Comp* P such that s_2 ends in s'_2 with trace t.

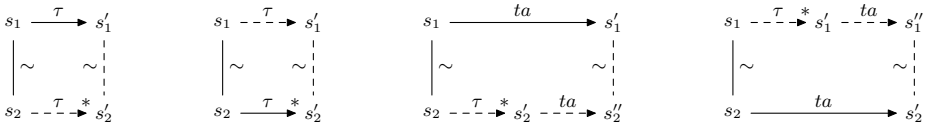

Fig. 5. Diagrams for delay bisimulation. Solid lines denote assumptions, dashed lines conclusions.

3.2 Simulation Properties

For semantic preservation, we must show trace equivalence for the source code and the compiled code. To do this, it is standard to show bisimilarity. The latter implies trace equivalence and can be shown by inspecting individual steps of execution instead of whole program executions. For the verification, we have chosen delay bisimilarity [20,1], as it is easy to obtain a delay bisimulation for multithreaded states from one for individual thread states (cf. Sec. 2.1). As it is transitive, we can decompose the compiler into smaller transformations and verify each on its own. Transitivity ensures that the overall compiler is correct, too.

Abstractly, programs define labelled transition systems whose states are the program states and whose labels constitute the observable behaviour. We write $s \xrightarrow{ta}\triangleright s'$ for a single **transition** (move), i.e. execution step in the small-step semantics, from state s to state s' with label ta. A predicate *is-*τ s ta s' determines whether the transition $s \xrightarrow{ta}\triangleright s'$ is unobservable. Such transitions are called silent or **τ-moves**. Since their labels are irrelevant, we don't keep track of them and write $s \xrightarrow{\tau}\triangleright s'$ instead. Moreover, $\xrightarrow{\tau}\triangleright^*$ denotes the reflexive and transitive closure of $\xrightarrow{\tau}\triangleright$. A **visible** move consists of a finite sequence of τ-moves followed by an observable transition. In this paper, we will often have states, labels, reductions, and the like for two or more programs and semantics. We will index variables and arrows with numbers to assign them to one of the semantics, i.e. s_1, $\xrightarrow{}\triangleright_1$, etc. for the first, s_2, $\xrightarrow{}\triangleright_2$, etc. for the second and so on.

A relation $\sim$ on states is a **(delay) bisimulation** [20,1] iff (i) $s_1 \sim s_2$ for the initial states s_1 and s_2 and (ii) it satisfies the simulation diagrams in Fig. 5: Every τ-move is simulated by a finite (possibly empty) sequence of τ-moves, and observable moves are simulated by visible moves such that the resulting states are again $\sim$-related. Two programs (transition systems) are **(delay) bisimilar** iff there exists a delay bisimulation for them. A special case of delay bisimulation is **strong bisimulation** [21] where every move is simulated by exactly one move.

Note that the relational composition $\sim_1 \circ \sim_2$ of two delay bisimulations $\sim_1$ and $\sim_2$ is again a delay bisimulation [1], where $s_1 \sim_1 \circ \sim_2 s_3 \equiv \exists s_2.\ s_1 \sim_1 s_2 \wedge s_2 \sim_2 s_3$. Hence, delay bisimilarity is transitive.

A program execution $s_0 \xrightarrow{tas}\triangleright^* s_n$ is a finite sequence of transitions $s_0 \xrightarrow{ta_1}\triangleright s_1 \xrightarrow{ta_2}\triangleright \ldots \xrightarrow{ta_n}\triangleright s_n$ where tas is the list of all labels ta_i of observable steps $s_{i-1} \xrightarrow{ta_i}\triangleright s_i$. To characterise complete executions for semantic preservation, we assume a predicate *final* that identifies terminal states. We say that a relation $\sim$ **preserves final states** iff final states are $\sim$-related to final states only. Delay bisimulations that preserve final states also preserve the semantics:

Lemma 1. *Let $\sim$ be a delay bisimulation that preserves final states and $s_1 \sim s_2$. If $s_1 \xrightarrow{tas}\triangleright_1^* s_1'$ such that $final_1\ s_1'$, then there exists an s_2' such that $s_2 \xrightarrow{tas}\triangleright_2^* s_2'$, $final_2\ s_2'$ and $s_1' \sim s_2'$. If $s_2 \xrightarrow{tas}\triangleright_2^* s_2'$ with $final_2\ s_2'$, then there exists an s_1' such that $s_1 \xrightarrow{tas}\triangleright_1^* s_1'$, $final_1\ s_1'$ and $s_1' \sim s_2'$.*

Proof. This lemma is shown by an easy induction on $s_1 \xrightarrow{tas}\triangleright_1^* s_1'$ and $s_2 \xrightarrow{tas}\triangleright_2^* s_2'$, resp., where the simulation properties from Fig. 5 are used in the inductive step.

3.3 Lifting for Bisimulations

The delay bisimulations for showing semantic preservation always relate multithreaded states. As we use our framework semantics at all compilation stages, we 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:

First, we lift a relation $\sim$ on thread-local states and the shared memories for two instantiating semantics $-\!\!\triangleright_1$ and $-\!\!\triangleright_2$ to multithreaded states s_1 and s_2, denoted by $s_1 \sim_m s_2$: (i) The lock status and wait sets of s_1 and s_2 must be the same. (ii) All threads in s_1 also exist in s_2 and vice versa. (iii) For every thread t in s_1 and s_2, say $thr\ s_1\ t = \lfloor(x_1, ln_1)\rfloor$ and $thr\ s_2\ t = \lfloor(x_2, ln_2)\rfloor$, the temporarily released locks must be the same ($ln_1 = ln_2$) and the local states $\sim$-related: $(x_1, shr\ s_1) \sim (x_2, shr\ s_2)$. $\sim_m$ preserves final states iff $\sim$ does so.

Next, we show that the above definitions are sensible: if $\sim$ is a delay bisimulation, then so is $\sim_m$. However, this holds only if τ-moves are in fact not observable by other threads. To that end, we require that they neither execute any TAs, nor change the shared heap: $is\text{-}\tau\ (x, h)\ ta\ (x', h')$ implies $ta = []$ and $h = h'$ for all x, x', h, h', ta. Moreover, we must require that $\sim$ **is preserved by heap changes** by other executing threads: Let y_1 and y_2 be two thread-local states with $(y_1, h_1) \sim (y_2, h_2)$, each of which performs a visible move to (y_1', h_1') and (y_2', h_2') resp. such that $(y_1', h_1') \sim (y_2', h_2')$, i.e. the visible moves simulate each other. Then, whenever $(x_1, h_1) \sim (x_2, h_2)$ holds for the old heaps, $(x_1, h_1') \sim (x_2, h_2')$ must still hold for the new heaps.

Theorem 1. *Let $\sim$ be a delay bisimulation that is preserved by heap changes and preserves final states. Then $\sim_m$ is also a delay bisimulation.*

Proof. The proof shows that the multithreaded semantics can perform the reductions of the executing thread from the simulation diagrams in Fig. 5. Preservation of final states ensures that joining succeeds either in both states or in none. Preservation under heap changes is required to establish $\sim_m$ on the result states.

4 Compilation from Source Code to Byte Code

Jinja [11] already contains a nonoptimising compiler *J2JVM* from source code to byte code via the intermediate language J_1: $compE_1$ compiles J expressions to J_1 expressions. It allocates array indices to local variables and replaces all

references to local variables in e by their indices. $compE_2$ and $compxE_2$ generate instruction sequences and exception tables for J_1 expressions. All of them are recursive on the expression structure. $compP_1$ lifts $compE_1$ to programs using $compP$, and so does $compP_2$ with $compE_2$ and $compxE_2$. The overall compiler *J2JVM* is the composition of $compP_1$ with $compP_2$. For JinjaThreads, we have extended $compE_1$, $compE_2$ and $compxE_2$ to handle *sync* expressions, which we present in this section. For details on the other constructs, see [11].

$compE_1$ assigns indices to variables in the following order: the *this* pointer, method parameters, local variables in the order of block nesting level. For $sync(e)\ e'$ statements, $compE_1$ shifts local variables declared in e' by 1 and annotates *sync* with the index that it has freed this way.

The translation of a J_1 $sync_V(e_1)\ e_1'$ expression to byte code must ensure that the monitor is unlocked even if an unhandled exception occurs in e_1'. An exception handler, which applies to all exceptions, needs to do this. Thus, the instructions for $sync_V(e_1)\ e_1'$ are (where @ concatenates two lists):

$compE_2\ e_1$ @ [*Store* V, *Load* V, *MEnter*] @ $compE_2\ e_1'$ @ [*Load* V, *MExit*, *Goto* 4] @ [*Load* V, *MExit*, *Throw*]

First, the monitor expression e_1 is evaluated and the result (on the stack) is stored in V. *Load* V pushes the monitor address back onto the stack and *MEnter* locks the monitor. Then, the block is executed, the monitor address loaded again 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. For the exception tables, $compxE_2\ (sync_V(e_1)\ e_1')$ appends to the exception tables for e_1 and e_1' the entry $(pc_1, pc_2, Any, pc_2 + 3, d)$ such that $compE_2\ e_1'$ occupies the positions $[pc_1, pc_2)$ in the instruction list and the d bottom values (of surrounding expressions) remain on the stack. Hence, this handler applies to any exception unless it is handled inside the body e_1'.

For example, consider the following method declaration, whose body is $([f], sync(Var\ f)\ Var\ this.doIt([]))$ in abstract syntax:

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

This compiles to [*Load* 1, *Store* 2, *Load* 2, *MEnter*, *Load* 0, *Invoke doIt* 0, *Load* 2, *MExit*, *Goto* 4, *Load* 2, *MExit*, *Throw*, *Return*] with exception table [(4, 6, *Any*, 9, 0)]. For realistic examples, see the formalisation online [19].

5 Correctness Proofs

In this section, we present the correctness proof for the compiler: a delay bisimulation between the source program P and the compiled program *J2JVM* P. Fig. 2 from Sec. 2 shows how we build it from smaller delay bisimulations: Between J and J_1 (Sec. 5.1), and between J_1' and *JVM* (Sec. 5.2), we present three delay bisimulations, one for each level (expressions, singlethreaded call stacks and multithreaded). The delay bisimulations for the call stack level lift the ones on

the expression level to single threads and the multithreaded level is always $\sim_m$ from Sec. 3.3 instantiated with the ^{t} level. Finally, we show that J_1^m and $J_1'^m$ are equivalent for states of interest (Sec. 5.3). By transitivity, P and $J2JVM\ P$ are delay bisimilar, i.e. the compiler is correct (Thm. 2).

The delay bisimulation relations typically consist of two parts: (i) the actual relation between states of the two semantics and (ii) some well-formedness constraints on the states of either semantics (e.g. being typeable) required by the bisimulation proof. The latter restrict the set of "valid" states for which the bisimulation property holds. To increase proof automation, we have similarly split the bisimulation proofs: First, we show that simulating reductions exists under conditions (i) and (ii), and that the resulting states are again related in (i). Next, we show that the constraints in (ii) are preserved under reductions and that the initial states satisfy them.

5.1 Strong Bisimulation between J and J_1

J and J_1 only differ in the treatment of local variables. Hence, the thread features do not introduce anything essentially new for the verification. Still, transferring the old correctness proof (which uses the big step semantics) required several substantial changes: (i) We adapted the small step semantics J_1^e such that it is strongly bisimilar to J^e, whereas the old semantics would only allow delay bisimilarity. This way, we need not distinguish observable from silent moves, which greatly simplifies the inuctive cases in the proofs. (ii) The strong bisimulation ${}_J\overset{e}{\sim}_1$ between J^e and J_1^e must now relate not only initial and final states, but also all intermediate states. We require that both expressions are identical in structure except for variable names, which are resolved according to $compE_1$'s numbering scheme. In addition to the old well-formedness constraints (e.g. a definite assignment test), the monitor address in the local variables in J_1^e must agree with the monitor address in the *insync* subexpression. (iii) We must now also show that small-step reductions preserve the well-formedness conditions.

Although the simulations are now much finer and must cover both directions, the old notions for the simulation proof [11] are still sufficient, i.e. the proofs do not pose any major difficulties. Establishing ${}_J\overset{e}{\sim}_1$ for the resulting states in the case for $sync_V(e)\ e'$ relies on V, the local monitor variable, not being accessed explicitly in the e', which the compiler numbering scheme guarantees.

5.2 Delay Bisimulation between J_1' and *JVM*

The translation from J_1 to *JVM* 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. Like between J and J_1, the key to correctness is delay bisimilarity on the expression level, on which we focus in this section. Calling and returning from methods works similarly in $J_1'^t$ and JVM^t, the laborious proof simply lifts delay bisimilarity. The multithreaded level is the framework semantics in both semantics. It is easy to show that the delay

bisimulation for the thread level preserves final states and is preserved by heap changes. Thus, Thm. 1 from Sec. 3.3 yields delay bisimilarity for $J_1'^m$ and JVM^m.

For the expression level, we make with a detour via the aggressive VM JVM_a^e. We show that JVM_a^e simulates $J_1'^e$, but that $J_1'^e$ simulates JVM_d^e. Since the byte code verifier accepts all compiled programs, the defensive VM JVM_d^e simulates the aggressive JVM_a^e step by step. This detour saves us from showing type safety for $J_1'^e$. If we used JVM_d^e directly, only full run-time typeability of the $J_1'^e$ expression would ensure that the JVM_d^e does not halt because of a type error where $J_1'^e$ still continues to execute. Conversely, the aggressive VM performs fewer checks than $J_1'^e$, so $J_1'^e$ might get stuck when JVM_a^e continues with undefined behaviour. Hence, bisimilarity holds only for conformant byte code states.

Note that this detour only affects the semantics, not the delay bisimulation relation ${}_1\overset{e}{\approx}_{jvm}$. $P,e,h \vdash (e_1, xs_1)\ {}_1\overset{e}{\approx}_{jvm}\ (stk, loc, pc, xcp)$ relates a J_1^e state (e_1, xs_1) (expression and local variables) to a JVM^e state (stk, loc, pc, xcp) (stack, local variables, program counter, and exception flag) for a heap h that is the same for both. P only defines the class hierarchy, whereas the J_1 expression e compiles to the instruction list $compE_2\ e$ with exception table $compxE_2\ e$. The inductive definition for ${}_1\overset{e}{\approx}_{jvm}$ mirrors the $J_1'^e$ reduction rules and relates instruction positions and the stack in the compiled code to partially evaluated expressions.

Fig. 6 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 its result value on top of the stack. Unfortunately, this does not translate to exceptions, because byte code does not propagate exceptions from subexpressions, but exception tables are used. Hence, ${}_1\overset{e}{\approx}_{jvm}$ contains separate exception propagation rules for all expressions, 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 J_1^e and JVM^e. 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 byte code instruction and intermediate J_1^e state. For example, B_4 relates the J_1^e state which next acquires a monitor's lock to the intermediate JVM^e state after executing the *Store V* instruction that saves the monitor address. Although J_1^e and JVM^e operate on the local variable array in the same way, they must not be equated in the bisimulation relation, because they differ in such intermediate states like in B_4, which J_1S4 skips.

The simulation proofs heavily rely on this value passing scheme. The next lemma, which is shown by induction on ${}_1\overset{e}{\approx}_{jvm}$, says that for related states, if one of them denotes a result values or thrown expressions, then the other can produce the same outcome using only τ-moves.

Lemma 2. *Let* $P,e,h \vdash (e_1, xs_1)\ {}_1\overset{e}{\approx}_{jvm}\ (stk, loc, pc, xcp)$. *If* $e_1 = Val\ v$, *then* (stk, loc, pc, xcp) *can silently execute to* $([v], xs_1, |compE_2\ e|, \bot)$. *If* $e_1 = Throw\ a$, *it can do so to* $([Addr\ a], xs_1, pc', \lfloor a \rfloor)$ *for some* pc'. *Conversely, if* $stk = [v]$ *and* $pc = |compE_2\ e|$, *then* (e_1, xs_1) *can silently become* $(Val\ v, loc)$. *If* $xcp = \lfloor a \rfloor$, *then* (e_1, xs_1) *can silently become* $(Throw\ a, loc)$.

$$P,e,h \vdash (Val\ v,\ xs)\ {}_1\overset{e}{\approx}_{jvm}\ ([v],\ xs,\ |compE_2\ e|,\ \bot)\ \mathbf{B}_1$$

$$\frac{P,e_1,h \vdash (Throw\ a,\ xs)\ {}_1\overset{e}{\approx}_{jvm}\ (stk,\ loc,\ pc,\ \lfloor a \rfloor)}{P,sync_V(e_1)\ e_2,h \vdash (Throw\ a,\ xs)\ {}_1\overset{e}{\approx}_{jvm}\ (stk,\ loc,\ pc,\ \lfloor a \rfloor)}\ \mathbf{B}_2$$

$$\frac{P,e_2,h \vdash (e,\ xs)\ {}_1\overset{e}{\approx}_{jvm}\ (stk,\ loc,\ pc,\ xcp)}{P,sync_V(e_1)\ e_2,h \vdash (insync_V(a)\ e,\ xs)\ {}_1\overset{e}{\approx}_{jvm}\ (stk,\ loc,\ 3 + |compE_2\ e_1| + pc,\ xcp)}\ \mathbf{B}_3$$

$$P,sync_V(e_1)\ e_2,h \vdash (sync_V(Val\ v)\ e_2, xs)\ {}_1\overset{e}{\approx}_{jvm}\ ([], xs[V := v], 1 + |compE_2\ e_1|, \bot)\ \mathbf{B}_4$$

Fig. 6. Example introduction rules for the ${}_1\overset{e}{\approx}_{jvm}$ bisimulation relation

Then, the simulation proofs consist of a huge induction on ${}_1\overset{e}{\approx}_{jvm}$ and case analysis of the execution steps. Control constructs like conditionals and loops, which are compiled to (conditional) jumps, are verified like in sequential settings.

5.3 Correctness of the Compiler

In Sec. 5.1 and 5.2, we have shown delay bisimilarity for the individual compiler stages, but w.r.t. two different semantics in the intermediate language J_1. To link J_1^m and $J_1'^m$ executions, we must show that the additional reductions in $J_1'^t$ due to e.g. J_1'S8 and J_1'S9 are never executed in $J_1'^m$, i.e. that the monitor exit instructions never raise *IllegalMonitorState* or *NullPointer* exceptions.

We prove that J_1^m and $J_1'^m$ are the same for a multithreaded state s_1, in which for every monitor a and thread t in s_1, the number of $insync_V(a)$ subexpressions of t equals the number of times t holds a in *locks* s_1 (written $\vdash s_1\ \surd$). In such a state, $J_1'^m$ never picks J_1'S9 as the TA [*HasNoLock* a'] never holds. Since all J_1^m reductions preserve $\vdash s_1\ \surd$, we add it as an additional well-formedness constraint to ${}_J\overset{m}{\sim}_1$. Similarly, J_1'S8 is never possible because ${}_J\overset{e}{\sim}_1$ (and thus ${}_J\overset{m}{\sim}_1$, too) does not allow *Null* being stored in the local variable for the monitor address. Thus, the augmented relation ${}_J\overset{m}{\sim}_1$ is also a delay bisimulation for J^m and $J_1'^m$.

We have shown all delay bisimulations from Fig. 2. By transitivity, J^m and JVM^m are delay bisimilar. In the initial state s_J in J, no monitor is locked, all wait sets are empty and there is only a single thread t whose expression is the body of some method M of class C in program P. For JVM, the initial state s_{jvm} is the same as s_J except that t's local state is the call frame ([], *loc*, C, M, 0) and no exception is flagged. For an *mfinal* J^m state s, the function *mxcp* s extracts the correct exception flag for every thread in s, i.e. $\bot$ for normal termination and $\lfloor a \rfloor$ if the exception at address a caused the abrupt termination. Then, Lem. 1 from Sec. 3.2 gives the following main correctness theorem:

Theorem 2. *Let s_J execute to s_J' in J^m for P with trace tas such that mfinal s_J'. Then, s_{jvm} executes to mxcp s_J' in JVM^m for J2JVM P with trace tas. Conversely, if s_{jvm} executes to s_{jvm}' in JVM^m for J2JVM P with trace tas such that mfinal s_{jvm}', then s_{jvm}' has the form mxcp s_J' and s_J executes to s_J' in J^m for P with trace tas.*

Proof. The full proof can be found online in the formalisation [19].

6 Discussion

Challenges due to concurrency. Verifying a compiler for a concurrent language adds three dimensions to compiler verification for sequential programs: (i) nondeterministic interleaving, (ii) different granularity of atomic operations between source and byte code, and (iii) memory models for optimisations. In JinjaThreads, we have addressed (i) and (ii).

For nondeterminism, bisimulation replaces the standard simulation approach for sequential programs, where only the compiled program simulates the source program. For bisimulation, it does not suffice to just show the other direction, but some subtleties arise: First, neither the source nor the compiled program may carry on if the other gets stuck, e.g. due to type errors. Our source code semantics is a small-step semantics, whereas the VM is an abstract state machine. Both naturally contain different type checks, only a full type system and type safety proof at every stage would ensure bisimilarity. By using both the aggressive and defensive VM in the simulation proofs, we only need a single type safety proof for byte code which ensures that both VMs are equivalent for verified byte code.

Second, the bisimulation must relate all states that are reachable from *either* initial state. Ordinary simulations do not have to relate intermediate states in the target code, which the source code skips, to any other state. This substantially increases the size of the bisimulation relation and consequently the number of cases the simulation proof has to consider. For ${}_1\overset{e}{\approx}_{jvm}$, Lem. 2 from Sec. 5.2 solves this problem for the numerous inductive steps in the simulation proof. For the base cases, we use similar lemmas for each expression, if necessary.

Our correctness result only mentions terminating executions, but a compiler should also preserve nontermination and deadlock. However, the standard (bi-)simulation approach with τ-moves cannot prove this because infinitely many consecutive τ-moves might be simulated by no moves at all, which is known as the *infinite stuttering problem*. Hence, our correctness result allows the byte code program to silently diverge even if all executions of the source program terminate, although this is not the case for our compiler. To prove this, we must strengthen the definition of delay bisimulation with an explicit notion of divergence like in [5], but we do not expect fundamental problems to arise from this.

Concerning (ii), several source code statements such as *sync* generate multiple byte code instructions. A single observable step in the source code program is decomposed into a number of silent steps and one observable step in between. Although it does not show up in the generated code, the number of atomic steps in the different semantics differs considerably. In particular, exceptions slowly propagate up in J whereas the VM directly jumps to the exception handler. The framework semantics, which we use at all stages, allows to decompose the multithreaded case to single threaded, where shared memory accesses and synchronisation must be observable. Hence, we do not have to worry about interleavings and atomicity in the main correctness proofs themselves.

Java vs. JinjaThreads. JinjaThreads is a generalised subset of Java and Java byte code. In terms of concurrency, it models all Java features of the JLS except

for time-dependent operations such as `Thread.sleep` and thread interruption. The latter could be easily added to the framework, but we cannot model the former because we have no notion of time. Other concurrency features like thread groups and the `java.util.concurrent` library are Sun's proprietary extensions, which we have not modelled. As to the sequential part, JinjaThreads inherits all features from Jinja: classes and objects, inheritance, dynamic dispatch, fields, arrays, exceptions, local variables, conditionals and loops, binary operators, etc. JinjaThreads models neither interfaces nor static fields and methods, but these are orthogonal to concurrency and could be added if desired. Thus, any Java program that uses only JinjaThreads features can be directly translated to JinjaThreads abstract syntax. JinjaThreads generalises Java in that it does not distinguish between statements and expressions to keep the formalisation simple. Unlike Java, e.g., the condition of a *while* loop may contain a *try catch* block.

For byte code, the situation is similar: all byte code instructions for the above features are modelled. The exception tables are slightly more general because for exception handling, the stack need only be trimmed to a specified size, but not completely cleared. For Jinja programs that respect the syntactic constraints of Java, our compiler only produces byte code that could be directly pretty-printed to Java byte code.

Although our formalisation completely ignores the memory model issue, it is still a sound model for real Java. For programs without data races in the sense of the JMM, the JMM guarantees sequential consistency [3], i.e. our interleaving semantics can reproduce all allowed executions. Hence, our results also apply to data race free Java programs. Moreover, our compiler is strictly nonoptimising; it just follows the recommendations in the JVMS [16, Ch. 7]. In fact, even Sun's `javac` compiler in Java 2 SE optimises only very little, but leaves this to the JIT compiler in the VM. Ševčík and Aspinall [25] showed that the JMM allows all program transformations that do not affect the JMM-observable behaviour. Since our compiler falls into this class, our verification will also work for the JMM.

Size of the formalisation. Currently, JinjaThreads consists of about 47k lines of Isabelle theories (without the a data flow analysis framework for the byte code verifer) , but not everything is relevant to the compiler itself. The framework semantics has approx. 6k lines. About 4.3k lines provide general infrastructure for JinjaThreads. The semantics for J, J_1 and *JVM*, the byte code verifier and type safety proofs are 14k lines. 1k lines show that our J semantics is bisimilar to the original source code semantics which dynamically inlines method calls. The translation from J to J_1 is verified in about 3.4k lines, but the by far largest part is the bisimilarity proof for J_1 and *JVM* with more than 17k lines. Replaying all proofs (including type safety and the byte code verifier) in Isabelle2009 takes 52 minutes on an Intel DualCore 2.33GHz processor with 2GB memory.

In comparison, Jinja [11], on which JinjaThreads was based, has only 15k lines of Isabelle code (excluding the data flow analysis). The compiler verification in terms of the big-step semantics is much easier: about 3.2k lines. Hence, going from big-step to small-step and from sequential to multithreaded has blown up the amount of proofs to be done. In particular, semantic preservation in Jinja

is only unidirectional from source code to byte code. Our proof scripts may be not optimal yet, and we expect that some improvements can be made, but the difference in size w.r.t. Jinja will remain immense.

7 Related Work

Formal semantics for Java. There are a lot of formal semantics for different subsets of sequential Java source code and byte code, e.g. [2,11,22]. As for concurrent Java, AtomicJava [7] by Flanagan et al. models most Java source code features except inheritance and exception handling. Stärk et al. [26] present a semantics for multithreaded source code based on abstract state machines, a pen-and-paper proof for type preservation, and a model of a sequential JVM. Liu and Moore's interpreter M5 [17] in ACL2 provides a monolithic multithreaded semantics for byte code, which also models class loading and initialisation. They aim at verifying JVM implementations w.r.t. the JVMS. Huisman and Petri's [10] detailed model of the JVMS in Coq features all byte code instructions, the wait/notify mechanism and thread interruption, but they do not report on any proofs with the semantics.

Like in our approach, Belblidia and Debabbi [4] have a semantics for threads in isolation and a second layer which manages the threads from which it receives thread actions, which they call labels. In contrast to ours, their single-thread semantics already takes care of the locks, which are stored in shared memory. Nor do they model the wait/notify mechanism or thread interruption. Also, they only give the byte code semantics, but do not report on any proofs with it.

Formally verified compilers. Compiler verification in general has been an active research topic for more than 40 years; see [6] for an annotated bibliography. Rittri [24] and Wand [27] first used bisimulations for compiler verification for a simple, parallel functional language. They showed that running the compiled code on a virtual machine is weakly bisimilar to the source code's denotational semantics.

Most closely related to our compiler is the one by Stärk et al. [26], but it handles only sequential Java source code. Also, they lack the formal rigour required for machine-checked proofs, as already pointed out in [11].

As for compiler verification for concurrent Java, Ševčík and Aspinall [25] report on verifying individual compiler optimisations w.r.t. the JMM. They show that the JMM does not allow as many as intended by its designers for programs with data races. However, their proofs are only on paper for a toy core language without almost all sequential Java features.

Leroy's CompCert project [13,14,15] has been the most remarkable landmark in mechanised compiler verification recently. He has verified a complete compilation tool chain from a subset of C source code to PowerPC assembly language in Coq. CompCert focuses on low-level details and language features such as memory layout, register allocation and instruction selection. Leroy also plans to extend CompCert to concurrency [15, Sec. 17.7]. He wants to show semantic preservation only for pseudo-sequential executions, where threads are

rescheduled only at lock operations. By contrast, our approach directly covers all interleavings and all behaviours, since we use bisimulations instead of simulations. Hence, our proof also shows that the different granularity of atomicity in source code and byte code does not affect the possible behaviour of programs.

As part of the Verisoft project, Leinenbach [12] has verified a nonoptimising compiler from C0, a subset of C, directly to DLX assembler in Isabelle/HOL. Like CompCert, he focuses on low-level details and only proves a weak simulation theorem for sequential executions, but not for the backward direction.

8 Conclusion and Future Work

In the current paper, we presented the first verification of a compiler for multi-threaded Java to byte code. The proof technique is much more difficult than for sequential languages: (i) one must switch from big-step to small-step semantics in the source, target and intermediate languages, and (ii) one must show both directions of the required bisimulation to be semantics preserving. According to the more complex proof requirements, the verification required 47k lines of Isabelle formalisation compared to 15k lines for the sequential predecessor.

Our verified Java compiler is part of a larger project which aims to completely verify an infrastructure for language-based security [9,28]. Still, much remains to be done: Without a trusted VM, the guarantee of the verified compiler is vacuous. We are currently working together with the Isabelle team on mechanically extracting an executable VM from our formalisation. Moreover, our notion of bisimulation cannot distinguish a deadlocked program from a silently diverging one. Leroy's simulation property [15] might be a good starting point for a stronger notion. As the next step, we plan to add the JMM to our interleaving semantics. Using the techniques from [25], we expect to transfer our results to the JMM without meeting fundamental problems. This will further narrow the formalisation gap between Java and JinjaThreads. Finally, we are going to add the missing constructs from sequential Java to obtain a verified compiler for full Java. To automate the conversions, we are also working on a simple parser from Java source code to abstract syntax and a printer from byte code abstract syntax back to Java byte code.

Acknowledgements. We would like to thank G. Snelting, D. Wasserrab, D. Lohner, and C. Hammer for inspiring discussions about the formalisation and the anonymous reviewers for valuable comments on earlier drafts of this paper.

References

1. L. Aceto, R.J. van Glabbeek, W. Fokkink, and A. Ingólfsdóttir. Axiomatizing prefix iteration with silent steps. *Information and Computation*, 127(1):26–40, 1996.
2. J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*, vol. 1523 of *LNCS*. Springer, 1999.
3. D. Aspinall and J. Ševčík. Formalising Java's data-race-free guarantee. In *TPHOLs'07*, vol. 4732 of *LNCS*, pp. 22–37. Springer, 2007.

4. N. Belblidia and M. Debbabi. A dynamic operational semantics for JVML. *Journal of Object Technology*, 6(3):71–100, 2007.
5. J. A. Bergstra, J. W. Klop, and E. R. Olderog. Failurs without chaos: a new process semantics for fair abstraction. In *IFIP'87*, Formal Description of Programming Concepts III, pp. 77–103. Elsevier Science Publishing, 1987.
6. M. A. Dave. Compiler verification: a bibliography. *SIGSOFT Software Engineering Notes*, 28(6):2–2, 2003.
7. C. Flanagan, S. N. Freund, M. Lifshin, and S. Qadeer. Types for atomicity: Static checking and inference for Java. *ACM TOPLAS*, 30(4):1–53, 2008.
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.
9. C. Hammer and G. Snelting. Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs. *International Journal of Information Security*, 8(6):399–422, 2009.
10. M. Huisman and G. Petri. BicolanoMT: a formalization of multi-threaded Java at bytecode level. In *BYTECODE'08*, ENTCS, 2008.
11. G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM TOPLAS*, 28:619–695, 2006.
12. D. Leinenbach. *Compiler Verification in the Context of Pervasive System Verification*. PhD thesis, Saarland University, 2008.
13. X. Leroy. Formal certification of a compiler backend or: Programming a compiler with a proof assistant. In *POPL'06*, pp. 42–54. ACM, 2006.
14. X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
15. X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.
16. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.
17. H. Liu and J S. Moore. Executable JVM Model for Analytical Reasoning: A Study. In *IVME'03*, pp. 15–23, 2003.
18. A. Lochbihler. Type safe nondeterminism - a formal semantics of Java threads. In *FOOL'08*, 2008.
19. A. Lochbihler. Jinja with threads. In *The Archive of Formal Proofs*. http://afp.sf.net/devel-entries/JinjaThreads.shtml, 2009. Formal proof development.
20. R. Milner. A modal characterisation of observable machine-behaviour. In *CAAP'81*, vol. 112 of *LNCS*, pp. 25–34. Springer, 1981.
21. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
22. T. Nipkow, editor. *Special Issue on Java Bytecode Verification*, vol. 30(3–4) of *Journal of Automated Reasoning*. Springer, 2003.
23. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, vol. 2283 of *LNCS*. Springer, 2002.
24. M. Rittri. Proving the correctness of a virtual machine by a bisimulation. Licentiate thesis, Göteborg University, 1988.
25. J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP'08*, vol. 5142 of *LNCS*, pp. 27–51. Springer, 2008.
26. R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.
27. M. Wand. Compiler correctness for parallel languages. In *FPCA'95*, pp. 120–134. ACM, 1995.
28. D. Wasserrab, D. Lohner, and G. Snelting. On PDG-based noninterference and its modular proof. In *PLAS'09*, pp. 31–44. ACM, 2009.