Type Safe Nondeterminism – A Formal Semantics of Java Threads

Andreas Lochbihler

Universität Passau

lochbihl@fim.uni-passau.de

Abstract

We present a generic framework to transform a single-threaded operational semantics into a semantics with interleaved execution of threads. Threads can be dynamically created and use locks for synchronisation. They can suspend themselves, be notified by other threads again, and interact via shared memory. We formalised this in the proof assistant Isabelle/HOL along with theorems to carry type safety proofs for the instantiating semantics (progress and preservation in the style of Wright and Felleisen [24]) over to the multithreaded case, thereby investigating the role of deadlocks and giving an explicit formalisation for them. We apply this framework to the Java thread model using an extension of the Jinja [12] source code semantics to have type safety for multithreaded Java machine-checked. The Java Memory Model is not included.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]: Semantics – type safety; F.3.2 [*Semantics of Programming Languages*]: Operational semantics; D.3.3 [*Language Constructs and Features*]: Concurrent programming structures

General Terms Languages, Theory

Keywords Threads, Deadlock, Type Safety, Java

1. Introduction

There are many formal approaches based on operational semantics and type systems to model certain safety properties of the Java programming language in the literature. A lot of work has been done in the context of type safety for sequential Java [5, 12, 19, 21, 22] on the source code level. In [19], Stärk et al. also give an abstract state machine semantics for threads, for which they show preservation of a number of invariants. Also, there is a large number of formal semantics for subsets of Java bytecode, see [3, Ch. 2] for an overview. All of them are single-threaded or lack other important features of concurrent Java, except for [3] and [15], which contain a pretty comprehensive semantics of Java bytecode features. However, both [3] and [15] only contain the semantics, but neither type system nor other safety features.

To the best of our knowledge, there is no formal type safety proof for multithreaded Java with dynamic thread creation and synchronisation, for neither source nor byte code, although Java's concurrency features are widely used in practise. Thus, the primary objective of this paper is a machine-checked type safety proof for a large and faithful subset of multithreaded Java, including classes with objects, fields and methods, inheritance with method overriding and dynamic dispatch, arrays, exception handling, dynamic thread creation, synchronisation via monitors and the wait/notify mechanism. To separate the semantics of these mechanisms from the low-level details of allowed compiler/runtime optimisations, we abstract from the complicated Java Memory Model (JMM) [9] by using a single shared memory like in [19]. Every properly synchronised program has the same semantics with or without the JMM.

Applying a standard approach [24] to show type safety, we experience the following in the multithreaded case: While the subject reduction theorem, i.e. preservation of well-typedness under reductions, is usually easily carried over from threads in isolation, the progress theorem, which shows that the evaluation of well-typed expressions does not get stuck, is particularly nontrivial in the presence of potential deadlocks and non-deterministic executions. In fact, most formalisations of type soundness for concurrent programming languages do not allow dynamic thread creation or synchronisation that can lead to deadlocks. If they do, they leave out the progress theorem or their notion of deadlock is given implicitly by the theorem's assumptions. This way, one cannot be sure that the theorem's notion coincides with the standard understanding of deadlock, especially because deadlocks can arise in many different ways (cf. Sec. 1.1 below). Thus, another major contribution of this paper is an explicit formalisation of deadlock in a theorem prover. Moreover, we then prove type safety with respect to this notion.

The basis of our work is the Jinja project [12], which already contains most features of the Java language subset that are not related to threads. On the source code level, Jinja consists of both a big-step and a small-step semantics, which are shown to be equivalent, a type system and a type safety proof through progress and preservation in the style of [24]. We add support for both threads and arrays, but since the latter is pretty orthogonal to the development here, we do not show this feature in detail. On the way, we develop a generic formal framework for lifting a sequential operational semantics to the concurrent case, which gives modularity in proving progress and preservation with respect to the sequential and concurrent aspects. We then instantiate this framework with the modified small-step semantics of Jinja to carry both progress and preservation over to the multithreaded case, and type safety then easily follows. In modelling threads, we closely follow Ch. 17 in the Java Language Specification (JLS) [9] for Java 5. Our semantics faithfully covers arbitrary dynamic thread creation, synchronisation on monitors and the wait/notify mechanism. Interaction takes place via shared memory. We have not included the deprecated methods `stop`, `suspend` and `resume`, neither `yield` and `sleep`, which are just hints to schedulers from which we abstract. Neither have we included `interrupt` and `join`, but we believe that they can be added easily. We have formalised every notion and proved every lemma and theorem in this paper in the theorem prover Isabelle/HOL [17], i.e. every single proof is machine-checked. Even though the presentation is quite technical at times, we show only the most important

Thread (I)

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

Thread (II)

```
synchronized (e) {
  synchronized (f) {
    .
    .
    .
  }
}
```

Thread (III)

```
synchronized (g) {
  synchronized (e) {
    ...
    g.notify();
    ...
  }
}
```

Figure 1. Three Java threads with different deadlock possibilities.

definitions, lemmata and theorems. The framework formalisation consists of about 5000 lines of definitions and proof scripts with approximately 250 lemmata. Multithreading for Jinja added about another 5000 lines to the Jinja source code formalisation. The current development of the formalisation is available online [16].

This contribution is organised as follows: The rest of this section gives an example on deadlocks and introduces some notation. The framework semantics is formalised in detail in Sec. 2. Sec. 3 contains the formal treatment of deadlocks. The machinery for lifting well-formedness constraints from the instantiating semantics to the multithreaded level is outlined in Sec. 4. In Sec. 5, we extend the Jinja semantics, apply the framework to it and show type safety. Sec. 6 discusses related and Sec. 7 concludes this paper.

1.1 Deadlock example

For an example on how different forms of deadlock can arise in Java, consider Fig. 1. For a start, suppose there are only threads (I) and (II), which share the object referenced by `f`. Thread (I) first locks the monitor for the object `f`, then acquires the lock on object `g`'s monitor, and later suspends itself to the wait set of `g`'s monitor, thereby releasing the lock on `g` again. Once it is woken up (by some notify invocation on `g` by some other thread),[1] it then competes for the lock on `g` again until it reacquires it and then finishes execution after releasing the locks on both `g` and `f`. Thread (II) has to acquire locks on objects `e` and `f` to execute its critical section. Now, if we start (I) first and then interleave threads (I) and (II), both end up in a deadlock: (I) is first to acquire the lock on `f`, so later it suspends itself to `g`'s wait set and waits for being woken up by some other thread, but there is only thread (II) which waits on the lock on `f`, which is still held by (I). Conversely, if (II) acquires the lock on `f` first, (I) will end up deadlocked in the wait set because there will be no other thread to wake up (I) again. Note that in this case, there is only a single thread in deadlock. This is not possible if deadlock is due to locks only.

Now, suppose thread (III) is run in parallel with (I) and (II) and shares objects `e` and `g`. (III) acquires locks on `g` and `e` and possibly wakes up thread (I). However, e.g., if thread (I) acquires its first lock and next, thread (II) acquires its first lock and then (III) does so, all threads end up in deadlock, because they are waiting cyclically on each other: (I) is waiting for the lock on `g`, which is held by (III), but (III) is waiting on the lock on `e`, which is held by (II), and (II) itself is waiting for the lock on `f` held by (I). There are many more deadlock possibilities in this example, but not all schedules lead to deadlock: If e.g. (II) acquires both locks first and (I) acquires the lock on `g` before (III) does, no deadlock can occur.

1.2 Notation

Both our framework and the Jinja semantics are formalised in the proof assistant Isabelle/HOL, i.e. all formulae and propositions are written in HOL, which is close to standard mathematical notation. This section introduces further non-standard notation, a few basic data types and their primitive operations.

Types is the set of all types which contains, in particular, the type of truth values *bool*, natural numbers *nat*, and integers *int*. The space of total functions is denoted by $\Rightarrow$. Type variables are written $'a$, $'b$ etc. The notation $t{::}\tau$ means that the HOL term t has type τ.

Pairs come with two projection function $\mathit{fst} :: 'a \times 'b \Rightarrow 'a$ and $\mathit{snd} :: 'a \times 'b \Rightarrow 'b$. We identify tuples with pairs nested to the right: (a, b, c) is identical to $(a, (b, c))$ and $'a \times 'b \times 'c$ to $'a \times ('b \times 'c)$.

Sets (type $'a\ \mathit{set}$) follow the usual mathematical convention.

Lists (type $'a\ \mathit{list}$) come with the empty list $[]$, the infix constructor $\cdot$ for consing and the infix @ that concatenates two lists. Variable names ending in "s" usually stand for lists or maps (see below) and $|xs|$ is the length of xs. If $n < |xs|$, then $xs_{[n]}$ denotes the n-th element of xs. The standard function $\mathit{map}\ f\ xs$ applies the function f to every element in the list xs. $\mathit{flatten}\ xss$ flattens a list xss of lists by concatenating all its elements. $\mathit{list\text{-}all2}\ P\ xs\ ys$ is shorthand for $|xs| = |ys| \wedge (\forall i{<}|xs|.\ P\ xs_{[i]}\ ys_{[i]})$.

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

The **option** data type $'a\ \mathit{option} = \mathit{None} \mid \mathit{Some}\ 'a$ adjoins a new element *None* to a type $'a$. All existing elements in type $'a$ are also in $'a\ \mathit{option}$, but are prefixed by *Some*. For succinctness, we write $\lfloor a \rfloor$ for $\mathit{Some}\ a$. Hence, for example, *bool option* has the values *None*, $\lfloor \mathit{True} \rfloor$ and $\lfloor \mathit{False} \rfloor$.

Case distinctions on data types are written with guards. For x of option type, e.g., $\mathsf{case}\ x\ \mathsf{of}\ \mathit{None} \Rightarrow f \mid \lfloor y \rfloor \Rightarrow g\ y$ means that if x is *None* then the expression is f, and if x is some y then the expression is $g\ y$ where g may refer to the value y of the *Some* constructor.

Partial functions are modelled as functions of type $'a \Rightarrow 'b\ \mathit{option}$ where *None* represents undefined and $f\ x = \mathit{Some}\ y$ means x is mapped to y. Instead of $'a \Rightarrow 'b\ \mathit{option}$ we write $'a \rightharpoonup 'b$ and call such functions **maps**. We write $f(x \mapsto y)$ as shorthand for $f(x{:=}\lfloor y \rfloor)$. The map $\lambda x.\ \mathit{None}$ is written *empty* and $\mathit{empty}(\ldots)$, where $\ldots$ are updates, abbreviates to $[\ldots]$. For example, $\mathit{empty}(x \mapsto y)$ becomes $[x \mapsto y]$.

The **indefinite description** $\epsilon x.\ Q\ x$ is known as Hilbert's ϵ-operator. It denotes some x such that $Q\ x$ is true, provided one exists.

2. Formalisation of the framework semantics

Our framework consists of two major parts. On the one hand, there is the framework semantics *redT* which handles the threads for an arbitrary (single-threaded) semantics. It is instantiated with an operational small-step semantics r such that every single thread can be reduced according to r. *redT* takes care of all bookkeeping, e.g. managing locks and thread reductions. On the other hand, we give a number of well-formedness conditions, both for the initial program configuration and for the instantiating semantics, which enable us to easily carry type safety proofs for the instantiating semantics (when we view it as a single-threaded semantics) to the multithreaded setting of the framework. In this section, we only treat the framework semantics, well-formedness conditions are presented in Sec. 3.2 and 4.

Our framework operates on a small-step operational semantics which is modelled by a set of single-step reductions. When applying the framework, we always deal with two different small-step semantics. We do not use a big-step semantics because any such semantics would have to appeal to small-step semantics features to allow for interleaved evaluations of different threads.

2.1 The instantiating semantics

The semantics with which we instantiate the framework contains all single-step reductions for a single thread in isolation. We call such a semantics an **instantiating semantics** and usually denote it by r. Formally, a single-step reduction in $r\ P$, where P is a program environment parameter to the semantics, is a tuple $((e, c, x), \mathit{tas}, (e', c', x'))$, which we write as $P \vdash \langle e,(c, x)\rangle\ {-}\mathit{tas}{-}r{\rightarrow}\ \langle e',(c', x')\rangle$. Intuitively, this denotes that in r with program environment P, the

[1] We abstract from "spurious wake-ups" which are allowed by the JLS [9].

expression e with common/shared memory c and local state x can reduce to the expression e' with new shared memory c' and local state x'. *tas* is a list of thread actions which tell the framework semantics to create a new thread, to acquire a lock, to wake another thread, etc. r has type $('p,\ 'e,\ 'l,\ 't,\ 'x,\ 'c,\ 'w)$ *semantics*, which is shorthand for $'p \Rightarrow (('e \times 'c \times 'x) \times ('e,\ 'l,\ 't,\ 'x,\ 'c,\ 'w)$ *thread-action list* $\times\ 'e \times 'c \times 'x)$ *set*, where we assign the following meaning to type parameters:

$'p$ Type of the program environment, e.g. Java class definitions

$'e$ Type of a thread expression

$'l$ Type of the locks threads acquire and release, e.g. Java monitors

$'t$ Type of the thread IDs, by which threads are identified

$'x$ Type of the state information that is local to a single thread, e.g. local variables

$'c$ Type of the common/shared memory, e.g. type of the heap

$'w$ Type of the identifiers for wait sets, e.g. monitors in Java

In what follows, to denote concrete values of these types we use the same letters (without the leading $'$, but with various decorations), e.g. t, t', t_1, etc. for thread IDs.

2.2 The multithreaded semantics

The framework semantics takes an instantiating semantics r as a parameter and forms the set of single-step reductions for the multithreaded case, in which a **state** consists of four components:

1. The **lock status** in the first component, usually denoted by *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. A thread (identified by its thread ID t) holding the lock l $(n + 1)$ 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 ensures that a lock is always held by at most one thread at a time.
2. The second component, usually denoted by *es*, stores the **thread information** in a map of type $'t \rightharpoonup 'e \times 'x$, denoted by $('e,\ 't,\ 'x)$ *thread-info*. For every ID t for which a thread is present, *es* maps t to the current expression of the thread and the thread-local state.
3. The third component of type $'c$ is the **shared memory**, which we usually denote with c.
4. The last component keeps track of the **wait sets**. Every thread can be in at most one wait set at a time. We therefore model them as a map *ws* of type $'t \rightharpoonup 'w$ (denoted by $('w,\ 't)$ *waitsets*) where a thread t is waiting in the wait set w iff $ws\ t = \lfloor w \rfloor$. t is ready for execution iff $ws\ t = None$.

Suppose, e.g., in Fig. 1 with threads (I) and (II) only, we are in the deadlock state as described in the first scenario in Sec. 1.1. Suppose `e` references an object at address e, and similarly for `f` and `g`. Then, this state is represented by the tuple $(ls,\ es,\ c,\ ws)$ where $ls = [f \mapsto (I,\ 0),\ e \mapsto (II,\ 0)]$, i.e. the locks on f and e are held once by (I) and (II) resp., $es = [I \mapsto (sync(locked(f))\ \{\ sync(g)\ \{\ \ldots\ \}\ \},\ x_1),\ II \mapsto (sync(locked(e))\ \{\ sync(f)\ \{\ \ldots\ \}\ \},\ x_2)]$ stores the thread expressions and the local data for (I) and (II). The shared memory $c = [e \mapsto Obj\ \ldots,\ f \mapsto Obj\ \ldots,\ g \mapsto Obj\ \ldots]$ contains the objects referenced by `e`, `f`, and `g` and (I) is in the wait set of g: $ws = [I \mapsto g]$

When a single thread expression is reduced in the instantiating semantics r, it can ask the framework to perform finitely many thread actions of type $('e,\ 'l,\ 't,\ 'x,\ 'c,\ 'w)$ *thread-action*, which can alter the state of the locks, threads and wait sets. Note that these actions are the only means of "communication" between the two semantics. Since this is unidirectional, the framework semantics can only transfer information to the instantiating semantics by picking one reduction offered by the instantiating semantics. Hence, the instantiating semantics must anticipate in its reductions all possible answers it is willing to accept from the framework semantics. At the moment, our framework supports eight different **thread actions**, which can be split into three groups:

Locking *Lock l* acquires a lock on l for the current thread t. If l is held by another thread, this reduction is not possible. For *Unlock l*, which releases one lock on l, t must hold at least one lock on l. *UnlockFail l* is only possible if t does not hold a lock on l, i.e. releasing a lock on l would fail.

Thread creation *NewThread t e c x* creates a new thread with ID t, initial expression e and local state x. c must be the equal to the common memory after the reduction step of the executing thread. The new thread is ready for execution and does not hold any locks. *NewThreadFail* tests whether all IDs of type $'t$ are assigned to a thread, i.e., normal thread creation would fail.

Wait sets *Suspend w* inserts the current thread t in the wait set w, any previous assignment of the current thread to a wait set is lost. *Notify w* non-deterministically wakes up one of the threads in the wait set w. If w is empty, no thread is woken up. *NotifyAll w* wakes up all threads in the wait set w.

Note that most actions with preconditions are paired with another one whose precondition is the negation of the former's. A thread should always know, e.g., if it is able to release a lock that it ought to have acquired before – if not, i.e., the lock status is inconsistent with the thread's state or the thread simply does not care whether it owns the locks it tries to release, the framework semantics can tell the instantiating semantics. The only exception is the *Lock* action, since a thread cannot decide on its own if it will be able to acquire a lock. We deliberately chose this asymmetry in the mutually-exclusive locks because they are the source of deadlocks in practice.

Similarly, *NewThread t e c x* contains more information than immediately necessary, e.g. the shared memory c, which is saved for later use. The thread ID for the new thread is actually assigned by the framework semantics, which picks non-deterministically a fresh thread ID if available, but since some semantics might be interested to know the IDs of those threads they have created, we include this extra information in the action itself already. Hence, a sensible semantics should - when it creates a new thread - offer a reduction for every possible thread ID. Similarly, when the maximum number of threads is reached, the *NewThreadFail* action tells the semantics so.

The list *tas* of thread actions issued by a reduction step of the instantiating semantics tells the framework what changes to do to the multithreaded state. The framework then performs these changes using three update functions for single thread actions:

- $ls\ {}^{\ell}\!\!\rightsquigarrow_t ta$ updates the lock state *ls* according to *ta* for thread t,
- $es \wr\!\!\rightsquigarrow ta$ adds the new thread in *ta* to the thread information *es*,
- $ws\ \bullet\!\!\rightsquigarrow_t ta$ changes the wait sets *ws* according to *ta* requested by thread t.

Their definitions are shown in Fig. 2, the cases in which the map is not updated have been omitted. The function *lock-lock ls t l* (*unlock-lock ls t l*) increases (decreases) the lock count of t on l, *new-thread-id es* gives an unassigned thread ID if there is one left. There are also functions which fold the update functions over lists of thread actions *tas*, denoted by $ls\ [{}^{\ell}\!\!\rightsquigarrow]_t\ tas$, $es\ [\wr\!\!\rightsquigarrow]\ tas$, and $ws\ [\bullet\!\!\rightsquigarrow]_t\ tas$ resp. They are combined in one state update function $(ls,\ es,\ ws)\ [\rightsquigarrow]_t\ tas$ for thread actions *tas*. Let us look briefly at wake-ups: The equation for *Notify w* says that if there is a thread waiting in w then pick any such thread, say t, and remove it from w, otherwise do nothing. Similarly, *NotifyAll w* removes all threads from w and leaves all other threads unchanged.

Before updating the multithreaded state $(ls,\ es,\ c,\ ws)$, we first have to check whether all changes in *tas* requested by the thread

$$
\begin{array}{lcl}
ls\ {}^{\ell}\!\rightsquigarrow_t\ \mathit{Lock}\ l & \equiv & \mathit{lock\text{-}lock}\ ls\ t\ l \\
ls\ {}^{\ell}\!\rightsquigarrow_t\ \mathit{Unlock}\ l & \equiv & \mathit{unlock\text{-}lock}\ ls\ t\ l \\
ls\ [{}^{\ell}\!\rightsquigarrow]_t\ [] & \equiv & ls \\
ls\ [{}^{\ell}\!\rightsquigarrow]_t\ ta{\cdot}tas & \equiv & (ls\ {}^{\ell}\!\rightsquigarrow_t\ ta)\ [{}^{\ell}\!\rightsquigarrow]_t\ tas \\
 & & \\
es\ \wr\!\rightsquigarrow\ \mathit{NewThread}\ t'\ e\ c\ x & \equiv & \textsf{if}\ \mathit{new\text{-}thread\text{-}id}\ es = \lfloor t' \rfloor \\
 & & \textsf{then}\ es(t' \mapsto (e, x))\ \textsf{else}\ \mathit{arbitrary} \\
es\ \wr\!\rightsquigarrow\ \mathit{NewThreadFail} & \equiv & \textsf{if}\ \mathit{new\text{-}thread\text{-}id}\ es = \mathit{None} \\
 & & \textsf{then}\ es\ \textsf{else}\ \mathit{arbitrary} \\
es\ [\wr\!\rightsquigarrow]\ [] & \equiv & es \\
es\ [\wr\!\rightsquigarrow]\ ta{\cdot}tas & \equiv & (es\ \wr\!\rightsquigarrow\ ta)\ [\wr\!\rightsquigarrow]\ tas \\
 & & \\
ws\ \bullet\!\rightsquigarrow_t\ \mathit{Notify}\ w & \equiv & \textsf{if}\ \exists\, t.\ ws\ t = \lfloor w \rfloor \\
 & & \textsf{then let}\ t = \epsilon\, t.\ ws\ t = \lfloor w \rfloor \\
 & & \qquad \textsf{in}\ ws(t := \mathit{None}) \\
 & & \textsf{else}\ ws \\
ws\ \bullet\!\rightsquigarrow_t\ \mathit{NotifyAll}\ w & \equiv & \lambda t.\ \textsf{if}\ ws\ t = \lfloor w \rfloor\ \textsf{then}\ \mathit{None}\ \textsf{else}\ ws\ t \\
ws\ \bullet\!\rightsquigarrow_t\ \mathit{Suspend}\ w & \equiv & ws(t \mapsto w) \\
ws\ [\bullet\!\rightsquigarrow]_t\ [] & \equiv & ws \\
ws\ [\bullet\!\rightsquigarrow]_t\ ta{\cdot}tas & \equiv & (ws\ \bullet\!\rightsquigarrow_t\ ta)\ [\bullet\!\rightsquigarrow]_t\ tas \\
 & & \\
(ls, es, ws)\ [\rightsquigarrow]_t\ ta & \equiv & (ls\ [{}^{\ell}\!\rightsquigarrow]_t\ ta,\ es\ [\wr\!\rightsquigarrow]\ ta,\ ws\ [\bullet\!\rightsquigarrow]_t\ ta)
\end{array}
$$

Figure 2. Update functions for locks, thread data and wait sets.

$$
\begin{array}{lcl}
ls \vdash_t \mathit{Lock}\ l\ \surd_{\ell} & \equiv & \mathit{may\text{-}lock}\ ls\ t\ l \\
ls \vdash_t \mathit{Unlock}\ l\ \surd_{\ell} & \equiv & \mathit{has\text{-}lock}\ ls\ t\ l \\
ls \vdash_t \mathit{UnlockFail}\ l\ \surd_{\ell} & \equiv & \neg\ \mathit{has\text{-}lock}\ ls\ t\ l \\
ls \vdash_t []\ \surd_{[\ell]} & \equiv & \mathit{True} \\
ls \vdash_t ta{\cdot}tas\ \surd_{[\ell]} & \equiv & ls \vdash_t ta\ \surd_{\ell} \wedge (ls\ {}^{\ell}\!\rightsquigarrow_t\ ta) \vdash_t tas\ \surd_{[\ell]} \\
 & & \\
es,c \vdash \mathit{NewThread}\ t\ e\ c'\ x\ \surd_{\wr} & \equiv & \mathit{new\text{-}thread\text{-}id}\ es = \lfloor t \rfloor \wedge c = c' \\
es,c \vdash \mathit{NewThreadFail}\ \surd_{\wr} & \equiv & \mathit{new\text{-}thread\text{-}id}\ es = \mathit{None} \\
es,c \vdash []\ \surd_{[\wr]} & \equiv & \mathit{True} \\
es,c \vdash ta{\cdot}tas\ \surd_{[\wr]} & \equiv & es,c \vdash ta\ \surd_{\wr} \wedge (es\ \wr\!\rightsquigarrow\ ta),c \vdash tas\ \surd_{[\wr]}
\end{array}
$$

Figure 3. Predicates on the state satisfying the preconditions of a thread action list

t can actually be granted to it. For this purpose, we introduce the predicates $ls \vdash_t ta\ \surd_{\ell}$ and $es,c \vdash ta\ \surd_{\wr}$ for an action ta. Again, they are paired with list versions $ls \vdash_t tas\ \surd_{[\ell]}$ and $es,c \vdash tas\ \surd_{[\wr]}$, respectively, which work through the list of thread actions, temporarily updating the state after each successful check. We show their definitions in Fig. 3, leaving out the cases in which the predicates are constantly true. *may-lock ls t l* checks that no thread other than t holds a lock on l in ls, *has-lock ls t l* that t does so. Note that we do not have to check for conditions on the wait set because wait set actions have no preconditions.

If a list of thread actions contains at least one action whose preconditions are not met by the current state at that position, then the reduction step cannot be executed that time. This is a powerful means to an instantiating semantics for checking that some preconditions on the locks hold, even without changes to the lock status. It can check e.g. whether it holds (does not hold) the lock l by using the list [*Unlock l*, *Lock l*] ([*UnlockFail l*]), without altering the locks. Note that the order of thread actions in the list can be important: [*Lock l*, *Unlock l*], e.g., consists of the same actions, but checks that no other thread is holding a lock on l. In the same way, [*Unlock l*, *Unlock l*, *UnlockFail l*, *Lock l*] releases one lock on l, but also tests that it has had exactly two before the reduction.

	t_1:	t_2:		t_3:	t_4:
1	*Lock* l_1	|	1	|	|
2	|	*Lock* l_2	2	|	*Suspend w*
3	|	(*Lock* l_1)	3	|	|
4	(*Lock* l_2)	|	4	*final*	|

Figure 4. Two thread schedules that lead to deadlocked threads.

In case the list contains multiple *Suspend* actions, the thread will be listed only in the wait set specified by the last such action after the reduction. It is not forbidden to have a *Suspend w* action followed by some *Notify w* or *NotifyAll w* action later in the same list. Note that this is the only case in which a thread can possibly wake up itself, because whenever a thread is in a wait set, this thread is not considered for execution by the framework.

Now, we put together everything we have so far to obtain the set of reductions in the framework semantics *redT* for the instantiating semantics r in the program environment P. A single reduction step in *redT r P* is denoted by $P,r \vdash \langle ls|es,c|ws\rangle\ {-t,tas\rightarrow}\ \langle ls'|es',c'|ws'\rangle$, which means that in the multithreaded state (ls, es, c, ws), thread t can reduce with thread actions tas which yields the multithreaded state (ls', es', c', ws'). The only reduction rule for the **framework semantics** then reads:

$$
\frac{\begin{array}{c} es\ t = \lfloor (e, x) \rfloor \qquad ws\ t = \mathit{None} \\ P \vdash \langle e,(c, x)\rangle\ {-tas{-}r\rightarrow}\ \langle e',(c', x')\rangle \qquad ls \vdash_t tas\ \surd_{[\ell]} \\ es,c' \vdash tas\ \surd_{[\wr]} \qquad (ls', es', ws') = (ls, es(t \mapsto (e', x')), ws)\ [\rightsquigarrow]_t\ tas \end{array}}{P,r \vdash \langle ls|es,c|ws\rangle\ {-t,tas\rightarrow}\ \langle ls'|es',c'|ws'\rangle}
$$

Intuitively, $P,r \vdash \langle ls|es,c|ws\rangle\ {-t,tas\rightarrow}\ \langle ls'|es',c'|ws'\rangle$ is a reduction in *redT r P* iff

- there is a thread t in es, say $es\ t = \lfloor (e, x) \rfloor$,
- which is not in a wait set in ws and
- which can do a reduction $P \vdash \langle e,(c, x)\rangle\ {-tas{-}r\rightarrow}\ \langle e',(c', x')\rangle$ in r with thread actions tas such that lock and thread creation conditions are met by the state (ls, es, c, ws) and
- the old state – where the thread information es gets updated with the single-thread reduction result (e', x') – is updated according to tas and combined with the new shared memory c' to yield (ls', es', c', ws')

We write reductions in the transitive and reflexive closure of $P,r \vdash \langle\cdot|\cdot,\cdot|\cdot\rangle{-}\cdot,\cdot{\rightarrow}\langle\cdot|\cdot,\cdot|\cdot\rangle$ as $P,r \vdash \langle ls|es,c|ws\rangle\ {-tas\rightarrow*}\ \langle ls'|es',c'|ws'\rangle$. tas now is a list of pairs of thread IDs and thread action lists and keeps track of the thread ID and thread actions for every reduction step, i.e., in step n, thread $\mathit{fst}\ tas_{[n]}$ reduces with the actions $\mathit{snd}\ tas_{[n]}$. Note that we do not model a specific scheduler here, i.e. a reschedule is possible between any two single-step reductions. However, restricting to a specific scheduling scheme can be done easily by selecting the appropriate reduction tuples from the framework semantics, e.g. based on the data given in tas.

Similarly, every reduction step of the instantiating semantics is considered to be atomic. Conversely, there is no direct means to force multiple reduction steps of a thread being atomic. However, an instantiating semantics can easily use the locking mechanism to ensure atomicity, if necessary: Replace, e.g., the lock type *'l* by *'l option* to introduce one extra lock. Then, every atomic reduction sequence should start (end) with a *Lock None* (*Unlock None*) action. If all single-step reductions, which are supposed to be atomic themselves, also prefix and postfix their thread actions by *Lock None* and *Unlock None* respectively, every thread can only be reduced if it can lock *None*. Since atomic sequences hold *None* in their intermediate steps (and locks are mutually exclusive), rescheduling is not possible inside atomic actions.

3. Type safety with deadlocks

In the sequential case, type safety is usually shown in the syntactic way [24] by showing progress and preservation. Progress means that every well-formed and well-typed expression that is not final can be reduced, preservation requires that well-typedness is preserved under reductions and the expression's type may become only more specific. Usually, preservation can easily be carried over from the instantiating semantics. Deadlocks, however, can break the progress property.

3.1 Formalising deadlocks

A thread is said to be in deadlock if it is waiting for an event that will never occur. In operating systems, deadlock of processes has four preconditions on resource usage: mutual exclusion, hold and wait, circular waiting, and no preemption (cf. e.g. [18]). In our setting, we have two different possibilities for deadlocks:

- A thread that is in a wait set is deadlocked if all threads are either in a wait set or have completed its execution or are already deadlocked themselves.
- A thread is deadlocked if it is waiting to obtain a lock which is held by another thread which itself is deadlocked, i.e. threads are waiting circularly on each other.

We assume that the function *final* tells whether a given thread's expression has completed its execution and is not supposed to be reduced any further. An instantiating semantics must define *final* appropriately. The predicate *wf-final* then ensures that final expressions indeed cannot be reduced.

Consider, e.g., the schedules shown in Fig. 4. On the left-hand side, thread t_1 acquires the lock l_1, then thread t_2 acquires the lock l_2. To proceed, t_2 needs the lock l_1, too, so the *Lock* l_1 is postponed. However, t_1 requests the lock l_2 which is held by t_2, hence both threads are deadlocked. On the right-hand side, we see an example with wait sets. Suppose there are only two threads, t_3 and t_4. Some time while being reduced, t_4 suspends itself to the wait set w. Once t_3 has reduced to a final value, t_4 is deadlocked because there are no more threads to wake it up again.

In our framework, things are a little more tricky because threads can atomically request and release any number of different locks at one reduction step, and the instantiating semantics need not be deterministic, i.e. a thread usually can reduce in many different ways requesting many different locks. To get a hold on this, we introduce two abstractions in our formalisation:

- $P,r \vdash \langle e,(c,\, x)\rangle\ L \wr$ denotes that in the semantics r with program P, the expression e can reduce in the state $(c,\, x)$ with a thread action list which contains at least one *Lock* l action for every lock l in L and in which all *Lock* actions are on locks in L.
- $P,r \vdash \langle e,(c,\, x)\rangle \wr$ denotes that in the semantics r with program P, the thread action list of every possible reduction of e in state $(c,\, x)$ contains at least one *Lock* action.

Note that we do not care about unlock actions because only a thread itself would be able to remedy the missing lock, not others. With these two abstractions, we can now define the set of **threads in deadlock** *deadlocked r P ls es ws c* as a co-inductive set. Fig. 5 shows the introduction rules, where the predicate $\mathit{final}(es \backslash M)$ checks whether each thread of es is either final or its thread ID is in M. The first rule ensures that those threads t are in deadlock which must obtain a lock and can be reduced and for every reduction of which there must be another thread t' in deadlock which holds one of the locks requested by that reduction. The second rule says that if a thread is in a wait set with all other non-final threads being either also in a wait set or deadlocked, then it is deadlocked. This is the case since there is no rule that eliminates that thread from the coinductive set, which is in fact a greatest fixpoint. As a minor technical detail, we consider a final thread which has suspended itself in its last reduction to a wait set to be deadlocked, too.

$$\frac{es\ t = \lfloor (e,\, x)\rfloor \qquad P,r \vdash \langle e,(c,\, x)\rangle \wr \qquad P,r \vdash \langle e,(c,\, x)\rangle\ L \wr \qquad \begin{array}{c}\forall L.\ P,r \vdash \langle e,(c,\, x)\rangle\ L \wr \longrightarrow \\ \exists\, t' {\in} \mathit{deadlocked}\ r\ P\ ls\ es\ ws\ c.\ t' \neq t \wedge (\exists\, l {\in} L.\ \mathit{has\text{-}lock}\ ls\ t'\ l)\end{array}}{t \in \mathit{deadlocked}\ r\ P\ ls\ es\ ws\ c}$$

$$\frac{es\ t = \lfloor (e,\, x)\rfloor \qquad \mathit{final}(es \backslash \mathit{deadlocked}\ r\ P\ ls\ es\ ws\ c) \qquad ws\ t = \lfloor w \rfloor}{t \in \mathit{deadlocked}\ r\ P\ ls\ es\ ws\ c}$$

Figure 5. Introduction rules for the coinductive set *deadlocked r P ls es ws c*

I: $[\mathit{Unlock}\ l_1, \mathit{Lock}\ l_2]$
II: $[\mathit{Unlock}\ l_2, \mathit{Lock}\ l_1]$, $[\mathit{Lock}\ l_3]$, $[\mathit{Lock}\ l_4]$
III: $[\mathit{Lock}\ l_6]$
IV: $[]$
V: $[\mathit{Lock}\ l_2, \mathit{Lock}\ l_3]$
VI: $[\mathit{Lock}\ l_3]$

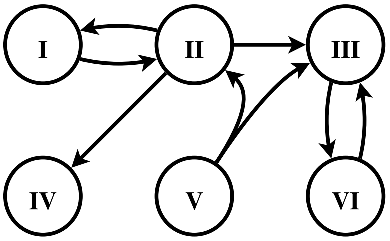

Figure 6. Deadlock example.

Consider Fig. 6 for an example of different deadlock situations. Suppose there are six threads which at the moment can reduce with the thread action lists shown on the left-hand side. If there are multiple lists for one thread, then there is one reduction for each list. Suppose that no thread is in a wait set and that the i-th thread holds the lock l_i. The graph on the right-hand side shows which thread is waiting to obtain a lock held by another thread. Then, thread III and VI are waiting on each other without other reduction options. Clearly, both of them are deadlocked. Although I and II are also waiting on each other, they are not deadlocked at the moment: II has two more reduction options. Waiting on lock l_3 will be in vain, because III is deadlocked. However, IV is not waiting on anyone, hence II may still hope to obtain the lock l_4 some time. Since thread action lists must be executed atomically, we may not appropriately interleave the action lists of I and II. Note that V is waiting *simultaneously* on II and III. Since III is already in deadlock, so is V. Clearly, IV is not deadlocked, since the empty list is always possible. Now, suppose thread IV is in a wait set. Then, all threads are deadlocked, since every thread except IV is waiting on some other thread to release a lock, and the only thread that could be reduced (i.e. IV) is waiting for some other thread waking it up.

For the framework semantics working properly, we need two basic well-formedness conditions. First, we require (via the predicate $es \vdash_f ls\ \surd$) that all locks are held by actual, non-final threads:

$$\begin{aligned} es \vdash_f ls\ \surd \equiv \forall\, l.\ & \mathsf{case}\ ls\ l\ \mathsf{of}\ \mathit{None} \Rightarrow \mathit{True} \\ & \mid \lfloor (t,\, n)\rfloor \Rightarrow \exists\, e\ x.\ es\ t = \lfloor (e,\, x)\rfloor \wedge \neg\, \mathit{final}\ e \end{aligned}$$

Second, let *wf-lock r P c c′* denote that both $P,r \vdash \langle e,(c,\, x)\rangle\ L \wr$ and $P,r \vdash \langle e,(c,\, x)\rangle \wr$ are invariant if c is changed (via reduction by another thread) to c'. Then, we show the following properties for the set of deadlocked threads. The first lemma is shown by case analysis on t being deadlocked. Lem. 2 and 3 are shown by coinduction on the set of deadlocked threads.

LEMMA 1. *Deadlocked threads are irreducible in the framework semantics. Formally:*

$$\frac{t \in \mathit{deadlocked}\ r\ P\ ls\ es\ ws\ c}{\neg\, P,r \vdash \langle ls|es,c|ws\rangle\ {-t,tas\!\longrightarrow}\ \langle ls'|es',c'|ws'\rangle}$$

$$\frac{t' \notin \textit{deadlocked } r\ P\ ls\ es\ ws\ c \qquad es\ t' = \lfloor(e, x)\rfloor \qquad \neg\, \textit{final } e \qquad es \vdash_f ls\ \surd}{\exists\, t\ e\ x.\ es\ t = \lfloor(e, x)\rfloor \wedge \neg\, \textit{final } e \wedge ws\ t = \textit{None} \wedge (P,r \vdash \langle e,(c, x)\rangle \wr \wedge (\exists\, L.\ P,r \vdash \langle e,(c, x)\rangle\ L \wr) \longrightarrow (\exists\, L.\ P,r \vdash \langle e,(c, x)\rangle\ L \wr \wedge (\forall\, l{\in}L.\ \textit{may-lock } ls\ t\ l)))}$$

Figure 7. Formal restatement of Lem. 3

LEMMA 2. *Deadlocked threads remain deadlocked after reductions of other threads.*

$$\frac{\textit{wf-final } r \qquad P,r \vdash \langle ls|es,c|ws\rangle\ {-t,tas\rightarrow}\ \langle ls'|es',c'|ws'\rangle \qquad \textit{wf-lock } r\ P\ c\ c'}{\textit{deadlocked } r\ P\ ls\ es\ ws\ c \subseteq \textit{deadlocked } r\ P\ ls'\ es'\ ws'\ c'}$$

LEMMA 3 (cf. Fig. 7). *If there is a non-final thread which is not deadlocked – and all locks are held by non-final threads – then there is a thread which is not final, not in a wait set and if it can be reduced and must obtain a lock when being reduced then there is a reduction whose Lock actions can all be performed.*

3.2 Well-formedness conditions for progress

To show progress in the multithreaded case, we naturally need some progress lemma for the instantiating semantics. We write *wf-progress r P es c* to denote that all non-final threads in *es* can be reduced in *r* if the shared memory is *c*. However, this is not sufficient for progress in the multithreaded case. Suppose, e.g., there is only a single thread *t* which tries to release a lock *l* by the *Unlock l* action although it is not holding *l*. If this was the only reduction possible for *t*, then this would break the progress property of the framework semantics, because the expression in the instantiating semantics is not stuck, but its embedding as a thread in the multithreaded semantics is. Therefore, we introduce the predicate *ex-red r P ls es c* which requires that for every thread *t* in *es* which is reducible in *r* with thread actions *tas* there is also a reduction in *r* with thread actions *tas'* where

- all actions in *tas'* that are related to thread creation are possible,
- for every lock *l*, all (un)lock actions on *l* in *tas'* are possible or the first one not possible is a *Lock l* action, and
- for every *Lock l* action in *tas'* that is not preceded by a *Unlock l* action in *tas'* there must already be a *Lock l* action in *tas*.

ex-red r P ls es c ensures that if a thread *t* in *es* can be reduced in *r*, then either *t* is deadlocked in (*ls*, *es*, *c*, *ws*) or there exists a thread in *es* and a reduction of it in *r* that is possible in (*ls*, *es*, *c*, *ws*). However, it is frequently more convenient to show for every thread ID *t* a different kind of well-formedness denoted by *wf-r-progress r P ls es c t*, which implies *ex-red* (cf. Lem. 4). Intuitively, *wf-r-progress r P ls es c t* imposes six constraints provided that there is a thread *t* in *es*, say $es\ t = \lfloor(e, x)\rfloor$:

- The length of the thread action list for reductions of *e* in state (*c*, *x*) must be bounded.
- For all reductions of *t* with actions *tas*, all *NewTread* actions must have the same shared memory as the resulting state.
- For every reduction of *t* which creates a new thread with ID *t'*, there must also be a reduction creating a thread with any other thread ID and, if all thread IDs are used up, another one which has a *NewThreadFail* action instead.
- Conversely, if not all thread IDs are used, for every *NewThreadFail* action, there must be a reduction actually creating a new thread with an arbitrary ID.
- If there is a reduction of *t* with actions *tas* which contain an *Unlock l* action, there must be a reduction with an *UnlockFail l* action instead, if the *Unlock l* action is not possible.
- Conversely, if *UnlockFail l* is requested by some reduction of *t*, but *t* holds sufficiently many locks on *l*, then there is also a reduction with an *Unlock l* action instead.

In fact, both *ex-red r P ls es c* and *wf-r-progress r P ls es c t* are more complex because we allow for partially rearranging of thread actions in the substitute reductions, but this technical detail is not important for the further development.

The next lemma shows that *wf-r-progress r P ls es c t* for every thread *t* is indeed sufficient for *ex-red r P ls es c*. It is shown by induction on prefixes of thread action lists and a large case distinction.

LEMMA 4. If $\forall\, t.$ *wf-r-progress r P ls es c t* then *ex-red r P ls es c*.

With Lem. 1, 2, and 3, we finally show the following key theorem in proving progress for the instantiating semantics:

THEOREM 1 (Progress). *Let t be a non-final thread which is not deadlocked. Suppose r satisfies the progress conditions wf-progress and ex-red r P ls es c. Then, if all locks are held by non-final threads, the multithreaded semantics can make progress. Formally:*

$$\frac{es\ t = \lfloor(e, x)\rfloor \qquad \neg\, \textit{final } e \qquad t \notin \textit{deadlocked } r\ P\ ls\ es\ ws\ c \qquad \textit{wf-progress } r\ P\ es\ c \qquad es \vdash_f ls\ \surd \qquad \textit{ex-red } r\ P\ ls\ es\ c}{\exists\, t'\ tas'\ es'\ ls'\ ws'\ c'.\ P,r \vdash \langle ls|es,c|ws\rangle\ {-t',tas'\rightarrow}\ \langle ls'|es',c'|ws'\rangle}$$

4. Lifting thread-local well-formedness conditions

Often one wants to consider only program expressions which satisfy some well-formedness condition. This section introduces the machinery provided by the framework to lift such conditions to the multithreaded case at no cost. An instantiating semantics can define a predicate Q of type $'e \Rightarrow\ 'c \Rightarrow\ 'x \Rightarrow \textit{bool}$ and the operator $\Uparrow\!\cdot\!\Uparrow$ automatically lifts Q to a predicate of type $('e,\ 't,\ 'x)\ \textit{thread-info} \Rightarrow\ 'c \Rightarrow \textit{bool}$. Formally:

$$\Uparrow\! Q \!\Uparrow es\ c \equiv \forall\, t.\ \textsf{case}\ es\ t\ \textsf{of}\ \textit{None} \Rightarrow \textit{True} \mid \lfloor(e, x)\rfloor \Rightarrow Q\ e\ c\ x$$

Suppose, e.g., that on the single-thread level, we have a definite assignment requirement, i.e. every variable must be assigned before being used. Suppose further, the predicate $\mathcal{D}$, which takes an expression *e* and a store for local variables *x*, guarantees that evaluating *e* satisfies the definite assignment condition with variables in the local store *x* having already been initialised. Then, $\Uparrow\!\mathcal{D}\!\Uparrow es\ c$ says that every thread *t* in *es*, say $es\ t = \lfloor(e, x)\rfloor$, satisfies $\mathcal{D}\ e\ x$.

In the context of type safety proofs, such well-formedness conditions are usually preserved under reductions in *r*. The predicate $P,r \vdash Q\ \surd\!\!\rightarrow$ gives three conditions that are sufficient for Q being also preserved under reduction in the framework semantics:

1. Q must be preserved under reductions of the instantiating semantics.
2. Q must also hold for new threads at the time of creation.
3. Q is preserved even if another thread, which also satisfies Q, changes the shared memory in a single-step reduction in *r*.

The next lemma shows that a predicate Q satisfying $P,r \vdash Q\ \surd\!\!\rightarrow$ is in fact preserved under multithreaded reductions, both single-step and multi-step. The first part is shown by a case analysis and induction on *tas*, the second by induction on the number of reduction steps.

LEMMA 5 (Soundness of the predicate $P,r \vdash \cdot \surd\!\!\rightarrow$).

$$\frac{P,r \vdash \langle ls|es,c|ws\rangle \;-t,tas\!\rightarrow\; \langle ls'|es',c'|ws'\rangle \qquad \Uparrow\!Q\!\Uparrow\ es\ c \qquad P,r \vdash Q \surd\!\!\rightarrow}{\Uparrow\!Q\!\Uparrow\ es'\ c'}$$

$$\frac{P,r \vdash \langle ls|es,c|ws\rangle \;-tas\!\rightarrow\!*\; \langle ls'|es',c'|ws'\rangle \qquad \Uparrow\!Q\!\Uparrow\ es\ c \qquad P,r \vdash Q \surd\!\!\rightarrow}{\Uparrow\!Q\!\Uparrow\ es'\ c'}$$

Returning to the example above, if we can show for r that $\mathcal{D}\ e\ x$ is preserved under reductions and all new threads also satisfy the predicate, then $P,r \vdash \mathcal{D} \surd\!\!\rightarrow$ holds and $\Uparrow\!\mathcal{D}\!\Uparrow\ es\ c$ is preserved under multithreaded reductions, too.

Similarly, such predicates on the thread level sometimes also needs some extra data, which is thread-specific, but invariant under reductions, e.g. a typing environment for local variables. We model such extra invariant data as maps from thread IDs to some type $'i$. As before, suppose Q is a predicate of type $'i \Rightarrow {'e} \Rightarrow {'c} \Rightarrow {'x} \Rightarrow bool$ on the invariant data and the thread state. We automatically lift Q to the framework semantics, denoted by $\Uparrow\!Q\!\Uparrow$ of type $('t \rightharpoonup {'i}) \Rightarrow ({'e}, {'t}, {'x})\ thread\text{-}info \Rightarrow {'c} \Rightarrow bool$.

We say a **map** I (type $'t \rightharpoonup {'i}$) **to invariant data** is well-formed w.r.t. a thread map es, denoted by $I \vdash_i es \surd$, iff I and es are defined on the same set of thread IDs. The partial order **extends** $I \trianglelefteq I'$ on maps to invariant data denotes that I' is an extension of I, i.e. I' is defined whenever I is defined, and in that case, I and I' coincide.

Let $I\ [\mathcal{I}\!\leadsto]_Q\ tas$ denote the extension of I with invariant data for all threads created in tas. These updates then preserve well-formedness of maps to invariant data, which is shown by case analysis, induction on tas, and by induction on the number of reduction steps.

LEMMA 6.

$$\frac{P,r \vdash \langle ls|es,c|ws\rangle \;-t,tas\!\rightarrow\; \langle ls'|es',c'|ws'\rangle \qquad es \vdash_i I \surd}{es' \vdash_i (I\ [\mathcal{I}\!\leadsto]_Q\ tas) \surd}$$

$$\frac{P,r \vdash \langle ls|es,c|ws\rangle \;-tas\!\rightarrow\!*\; \langle ls'|es',c'|ws'\rangle \qquad es \vdash_i I \surd}{es' \vdash_i (I\ [\mathcal{I}\!\leadsto]_Q\ flatten\ (map\ snd\ tas)) \surd}$$

Here we see the reason for storing the shared memory in *NewTread* actions. Only with this trick are we able to update I correctly in the case of multiple reductions because the data chosen for a new thread may depend on the shared memory at thread creation time.

Now, let the predicate $P,r,R \vdash Q \surd\!\!\rightarrow$ impose the following conditions on an predicate Q sufficient for Q being preserved under a multithreaded reduction in $redT\ r\ P$, provided that the well-formedness predicate R holds in the initial state. (This extra predicate R is, strictly speaking, not necessary for the proofs, but allows for some modularity in applications.)

1. Q is preserved under reductions in $r\ P$ for threads satisfying R.
2. For all actions $NewThread\ t\ e\ c\ x$ issued by threads satisfying R, there is some invariant data i for (e, c, x) with $Q\ i\ e\ c\ x$.
3. Q is unaffected by changes to the shared memory by another thread that also satisfies both Q and R.

The next lemma shows that these conditions are sufficient for Q being preserved under multithreaded reductions. Again, the first part is shown by a case analysis and induction on tas, the second part by induction on the number of reduction steps.

LEMMA 7 (Soundness of the predicate $P,r,\cdot \vdash \cdot \surd\!\!\rightarrow$).

$$\frac{\begin{array}{c} P,r \vdash \langle ls|es,c|ws\rangle \;-t,tas\!\rightarrow\; \langle ls'|es',c'|ws'\rangle \\ \Uparrow\!Q\!\Uparrow\ I\ es\ c \qquad \Uparrow\!R\!\Uparrow\ es\ c \qquad P,r,R \vdash Q \surd\!\!\rightarrow \end{array}}{\Uparrow\!Q\!\Uparrow\ (I\ [\mathcal{I}\!\leadsto]_Q\ tas)\ es'\ c'}$$

$$\frac{\begin{array}{c} P,r \vdash \langle ls|es,c|ws\rangle \;-tas\!\rightarrow\!*\; \langle ls'|es',c'|ws'\rangle \\ \Uparrow\!Q\!\Uparrow\ I\ es\ c \qquad \Uparrow\!R\!\Uparrow\ es\ c \qquad P,r \vdash R \surd\!\!\rightarrow \qquad P,r,R \vdash Q \surd\!\!\rightarrow \end{array}}{\Uparrow\!Q\!\Uparrow\ (I\ [\mathcal{I}\!\leadsto]_Q\ flatten\ (map\ snd\ tas))\ es'\ c'}$$

5. Multithreaded Jinja

In this section, we instantiate the framework with an extension of the Jinja source code semantics [12] for Java threads (without the JMM) and present the formalisation in detail. In modelling Java threads, we closely follow Ch. 17 in the Java Language Specification [9] for Java 5, making minor abstractions where special cases would have unnecessarily complicated the formalisation. First, we explain how Java threads are modelled in the framework. Then, we present the well-formedness conditions we need to impose on Jinja programs and expressions in order to show type safety (progress and preservation) in the last part.

In Jinja, source code expressions have the type *expr*, which is a data type with a constructor for every kind of operation, i.e. for creating objects and arrays, for casts, literals, access and assignment to local variables, arrays and fields, binary operations, method call, nested blocks, sequential composition, if and while statements, exception throwing and catching, and synchronized blocks. An expression is considered to be final iff it is a value or a thrown exception object on the heap. Although the syntax is quite different from Java, a compiler can translate any program which only uses Jinja features to the Jinja syntax.

5.1 Modelling the Java thread concept in Jinja

In our formalisation, the "program" P (type *J-prog*) contains the class declarations. Every thread stores its expression (*expr*) and its local variables (*locals*); since method calls are dynamically inlined, we do not need an explicit call stack. The shared memory (*heap*) is a map from addresses (*addr*) to heap objects (*heapobj*), thread IDs are natural numbers. In Java, only monitors – of which every object has one – can be locked, so locks are addresses, too. So are the wait set identifiers, because every monitor manages its own wait set. Thus, the Jinja semantics *red* is of type (*J-prog*, *expr*, *addr*, *nat*, *locals*, *heap*, *addr*) *semantics* and, accordingly, a reduction in it is denoted by $P \vdash \langle e,(h, x)\rangle \;-tas\!-\!red\!\rightarrow\; \langle e',(h', x')\rangle$.

Having explained the different instantiations of type parameters to the framework, we now present the introduction rules for the set of reductions *red P* which are relevant for our modelling Java threads. Fig. 8 and Fig. 9 show the reduction rules for method calls and the synchronized statement, respectively. See [12] for all other reduction rules, none of which issues thread actions on its own.

5.1.1 Thread creation

In Java, threads are associated with objects of the class Thread or its subclasses. A new thread is spawned by invoking the Thread object's start method. Hence, in our model, we treat objects of class Thread (or subclasses thereof) like standard objects, except that the well-formedness condition requires that no class comparable to Thread in the type hierarchy declares a method called start. Only when invoking the start method on such an object, rules NT1 and NT2 apply instead of CALL, which is for invoking nonnative methods and dynamically inlines the called method's code. In contrast, NT1 generates a new thread with an arbitrary thread ID, initial expression $Var\ this{\cdot}run([])$ (which represents the call to the thread object's run method), the heap as the shared memory component, and, as thread local information, *this* initialised to the correct thread object, given by its address. Similarly, NT2 models the reduction in case there are no more free thread IDs, which results in an *OutOfMemory* exception being thrown in the spawning thread.

5.1.2 Wait, notify and notifyAll

We cannot implement the *wait*, *notify*, and *notifyAll* methods of Object inside the Jinja language, because they are native just like spawning a thread. Hence, we include a pair of extra rules for each of these (rules W1, W2, N1, N2, NA1 and NA2), emulating the behaviour of the natives methods in the JVM:

$$\frac{h\ a = \lfloor \mathit{Obj}\ C\ \mathit{fs} \rfloor \qquad P \vdash C\ \mathit{sees}\ M\colon \mathit{Ts}{\rightarrow}T = (\mathit{pns}, \mathit{body})\ \mathit{in}\ D \qquad |\mathit{vs}| = |\mathit{pns}| \qquad |\mathit{Ts}| = |\mathit{pns}| \qquad \neg\, P \vdash C \preceq^* \mathit{Thread} \vee M \neq \mathit{start}}{P \vdash \langle \mathit{addr}\ a{\bullet}M(\mathit{map}\ \mathit{Val}\ \mathit{vs}),(h,\, x)\rangle \ {-}[]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{blocks}\ (\mathit{this}{\cdot}\mathit{pns},\, \mathit{Class}\ D{\cdot}\mathit{Ts},\, \mathit{Addr}\ a{\cdot}\mathit{vs},\, \mathit{body}),(h,\, x)\rangle}\,\textsc{Call}$$

$$\frac{h\ a = \lfloor \mathit{Obj}\ C\ \mathit{fs} \rfloor \qquad P \vdash C \preceq^* \mathit{Thread}}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{start}([]),(h,\, x)\rangle \ {-}[\mathit{NewThread}\ t\ (\mathit{Var}\ \mathit{this}{\bullet}\mathit{run}([]))\ h\ [\mathit{this} \mapsto \mathit{Addr}\ a]]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{unit},(h,\, x)\rangle}\,\textbf{NT1}$$

$$\frac{h\ a = \lfloor \mathit{Obj}\ C\ \mathit{fs} \rfloor \qquad P \vdash C \preceq^* \mathit{Thread}}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{start}([]),(h,\, x)\rangle \ {-}[\mathit{NewThreadFail}]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{THROW}\ \mathit{OutOfMemory},(h,\, x)\rangle}\,\textbf{NT2}$$

$$\frac{h\ a = \lfloor q \rfloor}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{wait}([]),(h,\, x)\rangle \ {-}[\mathit{Suspend}\ a,\, \mathit{Unlock}\ a,\, \mathit{Lock}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{unit},(h,\, x)\rangle}\,\textbf{W1}$$

$$\frac{h\ a = \lfloor q \rfloor}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{wait}([]),(h,\, x)\rangle \ {-}[\mathit{UnlockFail}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{THROW}\ \mathit{IllegalMonitorState},(h,\, x)\rangle}\,\textbf{W2}$$

$$\frac{h\ a = \lfloor q \rfloor}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{notify}([]),(h,\, x)\rangle \ {-}[\mathit{Notify}\ a,\, \mathit{Unlock}\ a,\, \mathit{Lock}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{unit},(h,\, x)\rangle}\,\textbf{N1}$$

$$\frac{h\ a = \lfloor q \rfloor}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{notify}([]),(h,\, x)\rangle \ {-}[\mathit{UnlockFail}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{THROW}\ \mathit{IllegalMonitorState},(h,\, x)\rangle}\,\textbf{N2}$$

$$\frac{h\ a = \lfloor q \rfloor}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{notifyAll}([]),(h,\, x)\rangle \ {-}[\mathit{NotifyAll}\ a,\, \mathit{Unlock}\ a,\, \mathit{Lock}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{unit},(h,\, x)\rangle}\,\textbf{NA1}$$

$$\frac{h\ a = \lfloor q \rfloor}{P \vdash \langle \mathit{addr}\ a{\bullet}\mathit{notifyAll}([]),(h,\, x)\rangle \ {-}[\mathit{UnlockFail}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{THROW}\ \mathit{IllegalMonitorState},(h,\, x)\rangle}\,\textbf{NA2}$$

Figure 8. Jinja reduction rules for method call.

$$\frac{P \vdash \langle o',s\rangle\ {-}\mathit{tas}{-}\mathit{red}{\longrightarrow}\ \langle o'',s'\rangle \qquad \neg\ \mathit{lock\text{-}granted}\ o'}{P \vdash \langle \mathit{sync}(o')\ e,s\rangle\ {-}\mathit{tas}{-}\mathit{red}{\longrightarrow}\ \langle \mathit{sync}(o'')\ e,s'\rangle}\,\textbf{S1}$$

$$P \vdash \langle \mathit{sync}(\mathit{null})\ e,s\rangle\ {-}[]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{THROW}\ \mathit{NullPointer},s\rangle\ \textbf{S2}$$

$$P \vdash \langle \mathit{sync}(\mathit{throw}\ a)\ e,s\rangle\ {-}[]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{throw}\ a,s\rangle\ \textbf{S3}$$

$$P \vdash \langle \mathit{sync}(\mathit{addr}\ a)\ e,s\rangle\ {-}[\mathit{Lock}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{sync}(\mathit{locked}(a))\ e,s\rangle\ \textbf{S4}$$

$$\frac{\begin{array}{c} P \vdash \langle e,s\rangle\ {-}\mathit{tas}{-}\mathit{red}{\longrightarrow}\ \langle e',s'\rangle \\ \forall\, \mathit{tas}'.\ \mathit{tas} \neq \mathit{Suspend}\ a{\cdot}\mathit{tas}' \wedge \mathit{tas} \neq \mathit{UnlockFail}\ a{\cdot}\mathit{tas}' \end{array}}{P \vdash \langle \mathit{sync}(\mathit{locked}(a))\ e,s\rangle\ {-}\mathit{tas}{-}\mathit{red}{\longrightarrow}\ \langle \mathit{sync}(\mathit{locked}(a))\ e',s'\rangle}\,\textbf{S5}$$

$$\textbf{S6}\quad \frac{P \vdash \langle e,s\rangle\ {-}\mathit{tas}{-}\mathit{red}{\longrightarrow}\ \langle e',s'\rangle \qquad \mathit{tas} = \mathit{Suspend}\ a{\cdot}\mathit{tas}'}{P \vdash \langle \mathit{sync}(\mathit{locked}(a))\ e,s\rangle\ {-}\mathit{tas}\ @\ [\mathit{Unlock}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{sync}(\mathit{addr}\ a)\ e',s'\rangle}$$

$$P \vdash \langle \mathit{sync}(\mathit{locked}(a))\ \mathit{Val}\ v,s\rangle\ {-}[\mathit{Unlock}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{Val}\ v,s\rangle\ \textbf{S7}$$

$$P \vdash \langle \mathit{sync}(\mathit{locked}(a))\ \mathit{throw}\ a',s\rangle\ {-}[\mathit{Unlock}\ a]{-}\mathit{red}{\longrightarrow}\ \langle \mathit{throw}\ a',s\rangle\ \textbf{S8}$$

Figure 9. Jinja reduction rules for the synchronized statement.

1. One (W1, N1, NA1) models normal execution with three thread actions: The first one tells the framework to manipulate the wait sets according to the method's meaning, the *Unlock* and subsequent *Lock* actions are to check if the current thread holds a lock on monitor associated with the wait set. If not, this reduction will not be chosen by the multithreaded semantics.
2. In that case, the other rule (W2, N2, NA2) instead raises an *IllegalMonitorState* exception if the thread does not hold a lock on the monitor associated with the object. The *UnlockFail* action ensures that this rule can only be chosen if the thread does not hold a lock on the monitor.

Note that the framework semantics does not allow for "spurious wake-ups", which are permitted by the JLS [9].

5.1.3 Synchronisation

Synchronisation in Java is done via synchronized statements. Sec. 14.19 in the JLS [9] determines its behaviour: "A synchronized statement acquires a mutual-exclusion lock on behalf of the executing threads, executes a block and then releases the lock." Fig. 9 shows the rules for the synchronized statement. A synchronized statement is written $\mathit{sync}(o')\ e$ where o' is the expression for the object on whose monitor a lock is acquired and e is the block's expression. To remember syntactically that a synchronized statement has already acquired the lock on a, we use the expression $\mathit{locked}(a)$ at o'. The predicate $\mathit{lock\text{-}granted}\ o'$ checks whether o' is of the form $\mathit{locked}(a)$ for some a. We now discuss the rules in Fig. 9 in detail.

S1 reduces the monitor subexpression, provided it is not the expression denoting the granted lock on the monitor. If the monitor subexpression reduces to the *null* value, a *NullPointer* exception is thrown (S2). If an exception is raised while reducing the monitor subexpression, the same exception is propagated by S3. If it reduces to some address a, the thread can only reduce further (S4) by acquiring the lock on a.[2] Once the lock has been granted, the body e of the synchronized statement is reduced: If e's reduction does

[2] In [4], in rule **SYN2**, the monitor subexpression may evaluate to an address only if the lock can be acquired at the same time. This simplification, which we avoid, may have some strange effects. For instance, suppose we have a thread θ_1 with expression

$$\mathit{sync}(o{\cdot}f)\ \{\ o{\cdot}f = \mathit{new}\ \mathit{Object}();\ \mathit{sync}(o)\ \{\ \ldots\ \}\}$$

and another one θ_2 with expression

$$\mathit{sync}(o)\ \{\ \mathit{sync}(o{\cdot}f)\ \{\ \ldots\ \}\}$$

where o is a shared object, say O, with a field f initially referencing another object, say F. If θ_1 first obtains the lock on F, which is referenced by $o{\cdot}f$, and then θ_2 is evaluated as far as possible, i.e. it is then waiting on the lock on F, this schedule leads to a deadlock in our semantics. In [4], however, θ_2 cannot evaluate that far, but stops evaluating $o{\cdot}f$ at $O{\cdot}f$. Once θ_1 changes $O{\cdot}f$ to another object F', θ_2 can continue with a lock on F', i.e., there is no possibility for a deadlock here. Although the JMM introduces much complexity about which values a thread sees when, an example with a volatile field f reveals the same issue in the presence of the memory model.

not request that the thread is suspended to the wait set a, nor that the thread does not hold a lock on a, $\mathit{sync}(\mathit{locked}(a))\ e$ is reduced accordingly by S5. If e's reduction corresponds to a call to the *wait* method of a, which is characterised by a *Suspend a* action at the action list's head, additionally, the lock on a is released (by appending an *Unlock a* action to the end) (S6).

If the body reduces to a value in a normal way, the lock is released and the synchronized statement returns the value by S7. If an exception is thrown while the body is reduced, the lock on a is released and the exception propagated (S8).

Since synchronized methods are just syntactic sugar for ordinary methods with their whole body inside a synchronized statement on *this*, we omit this option in our formalisation for simplicity.

5.2 Well-formedness constraints

In Sec. 4, we have shown how local well-formedness constraints can be lifted to the multithreaded case. In this section, we introduce the well-formedness conditions for Jinja and show how they are lifted in the above way.

There are already a number of constraints for well-formed Jinja programs (see [12] for the details). For the threads, we need to impose some more on the class declarations. We say, a class with name C is well-formed if it is a well-formed Jinja class and satisfies additionally:

- if C is *Object*, it must not have fields or methods which are not native (and thus would be hardwired into the semantics)
- if C is *Thread*, it must have a *run* method with no parameters and return type *Void*,
- if C is comparable to *Thread* in the subtype relation, it must not have a *start* method (which is hardwired in the semantics), and
- it must not have methods called *wait*, *notify*, nor *notifyAll*.

A well-formed program P, denoted by *wf-J-prog P*, consists of well-formed class declarations only.

Since we use the special expression $\mathit{locked}(a)$ in monitor expressions of synchronized statements to remember that the lock on the monitor a has already been granted, we must also ensure that $\mathit{locked}(a)$ cannot occur in a monitor expression via subexpression reduction (rule S1 in Fig. 9). To this end, we define the predicate $\vdash e \surd_{\&}$ for an expression e. Intuitively, $\vdash e \surd_{\&}$ ensures that in an expression e with multiple subexpressions, if an explicit address value occurs in such a subexpression e', then all other subexpressions of e which are evaluated before e' according to the evaluation order imposed by the semantics must have already been completely evaluated. For example, in an array assignment $a[i] := e$, if a has evaluated to some address a and the index expression i has not yet been completely reduced to an integer, e must not contain an explicit address. We write $\vdash \cdot \Uparrow\surd_{\&}\Uparrow$ for $\vdash \cdot \surd_{\&}$ lifted to the multithreaded case with the $\Uparrow\cdot\Uparrow$ operator.

LEMMA 8. If $P \vdash \langle o',s\rangle -\mathit{tas}-\mathit{red}\rightarrow \langle o'',s'\rangle$ and $\vdash \mathit{sync}(o')\ e \surd_{\&}$ then $\neg$ *lock-granted* o''.

By this lemma, shown by case analysis on the reduction steps, $\vdash \cdot \surd_{\&}$ in fact does the job. The next lemma, which proves that $\vdash \cdot \surd_{\&}$ is also preserved under reductions for well-formed programs, is shown by induction on the set of reductions.

LEMMA 9. If *wf-J-prog P* then $P,\mathit{red} \vdash (\vdash \cdot \surd_{\&}) \surd\!\rightarrow$.

Locks are stored twice in the multithreaded Jinja semantics: On the one hand, there is the map *ls* for storing locks in the framework. On the other hand, we remember locks in the monitor subexpression of synchronized statements, which are stored in the map *es*. Naturally, we want them to be consistent, which is expressed by the predicate $es \vdash_{\mathrm{e}} ls \surd$:

$$\begin{array}{l} es \vdash_{\mathrm{e}} ls \surd \equiv \forall t.\ \mathsf{case}\ es\ t\ \mathsf{of}\ \mathit{None} \Rightarrow \forall l.\ \neg\ \mathit{has\text{-}lock}\ ls\ t\ l \mid \lfloor(e, x)\rfloor \Rightarrow \\ \quad \forall l.\ \mathsf{if}\ 0 < \boldsymbol{\ell}\ e\ l\ \mathsf{then}\ ls\ l = \lfloor(t, \boldsymbol{\ell}\ e\ l - 1)\rfloor\ \mathsf{else}\ \neg\ \mathit{has\text{-}lock}\ ls\ t\ l \end{array}$$

$$P \vdash T \le T \qquad P \vdash NT \le \mathit{Class}\ C \qquad \frac{P \vdash A \le B}{P \vdash A[\,] \le B[\,]}$$

$$\frac{P \vdash C \preceq^* D}{P \vdash \mathit{Class}\ C \le \mathit{Class}\ D} \qquad P \vdash NT \le A[\,] \qquad P \vdash A[\,] \le \mathit{Class\ Object}$$

Figure 10. Jinja subtyping rules

The function $\boldsymbol{\ell}\ e\ l$ counts the number of $\mathit{sync}(\mathit{locked}(l))$ subexpressions in e. Note that $es \vdash_{\mathrm{e}} ls \surd$ implies $es \vdash_f ls \surd$:

LEMMA 10. If $es \vdash_{\mathrm{e}} ls \surd$ then $es \vdash_f ls \surd$.

LEMMA 11. *For well-formed programs and well-formed thread expressions,* $\cdot \vdash_{\mathrm{e}} \cdot \surd$ *is preserved under reductions:*

$$\frac{\mathit{wf\text{-}J\text{-}prog}\ P \qquad \begin{array}{l} P,\mathit{red} \vdash \langle ls|es,c|ws\rangle -t,\mathit{tas}\rightarrow \langle ls'|es',c'|ws'\rangle \\ \vdash es \Uparrow\surd_{\&}\Uparrow \qquad es \vdash_{\mathrm{e}} ls \surd \end{array}}{es' \vdash_{\mathrm{e}} ls' \surd}$$

$$\frac{\mathit{wf\text{-}J\text{-}prog}\ P \qquad \begin{array}{l} P,\mathit{red} \vdash \langle ls|es,c|ws\rangle -\mathit{tas}\rightarrow\!* \langle ls'|es',c'|ws'\rangle \\ \vdash es \Uparrow\surd_{\&}\Uparrow \qquad es \vdash_{\mathrm{e}} ls \surd \end{array}}{es' \vdash_{\mathrm{e}} ls' \surd}$$

This lemma is shown by an extensive case analysis and induction on the thread action list, and the number of reduction steps.

Another feature of both Jinja and Java is definite assignment (cf. example in Sec. 4): It ensures that every variable must be assigned before being used. As before, we write $\mathcal{D}\ e\ x$, which takes an expression e and a store for local variables x, for the definite assignment test. The details for $\mathcal{D}$ can be found in [12], the definitions for the synchronized statements are straight forward. The next lemma, which is shown by rule induction on the reductions, shows that definite assignment is preserved under reductions for well-formed programs.

LEMMA 12. If *wf-J-prog P* then $P,\mathit{red} \vdash \mathcal{D} \surd\!\rightarrow$.

5.3 The type safety proof

In this section, we introduce the Jinja type system and show the progress and subject reduction theorem, which we combine in the end to get type safety.

5.3.1 The type system

Jinja has two primitive types *Boolean* and *Integer*, for *Unit* the type *Void*, and reference types *NT*, *Class C*, and $T[\,]$, where $\cdot[\,]$ is a recursive type constructor for arrays. This way, array types of arbitrarily many dimensions are allowed in Jinja whereas the Java VM supports at most 255 dimensions [14].

On the types of a program P, we define a **widening relation** whose rules are shown in Fig. 10: $P \vdash T \le T'$ denotes that T is a subtype of T'. Subtyping behaves as in Java: The subclass relation $P \vdash \cdot \preceq^* \cdot$ is injected into the subtype relation, *NT* is a subtype of reference types, arrays are subtypes of *Class Object* and $\cdot[\,]$ is covariant. We extend $\le$ to lists pointwise, denoted by $[\le]$.

Jinja has a static type system with typing judgements of the form $P,E \vdash e :: T$ where P is the Jinja program, E the typing environment, i.e. a map from variable names to types, e is the expression to be typed and T is e's Jinja type. Fig. 11 shows the typing rules for method calls and the synchronized statement. Due to space limitations, we cannot show all of the typing rules. For the remaining rules, see [12].

Rule WTC1 is the standard Jinja rule for method call where $P \vdash C\ \mathit{sees}\ M{:}\ \mathit{Ts}{\rightarrow}T = (\mathit{pns}, \mathit{body})\ \mathit{in}\ D$ denotes that in program P,

$$\frac{\begin{array}{c}P,E \vdash e :: \mathit{Class}\ C \qquad P \vdash C\ \mathit{sees}\ M\colon \mathit{Ts}{\to}T = (\mathit{pns}, \mathit{body})\ \mathit{in}\ D\\ \mathit{list\text{-}all2}\ (\lambda e\ T.\ P,E \vdash e :: T)\ \mathit{es}\ \mathit{Ts}' \qquad P \vdash \mathit{Ts}'\ [\le]\ \mathit{Ts}\end{array}}{P,E \vdash e{\bullet}M(\mathit{es}) :: T}\ \textbf{WTC1}$$

$$\frac{P,E \vdash e :: \mathit{Class}\ C \qquad P \vdash C \preceq^{*} \mathit{Thread}}{P,E \vdash e{\bullet}\mathit{start}([]) :: \mathit{Void}}\ \textbf{WTC2}$$

$$\frac{P,E \vdash e :: T \qquad \mathit{is\text{-}refT}\ T \qquad T \neq \mathit{NT}}{P,E \vdash e{\bullet}\mathit{wait}([]) :: \mathit{Void}}\ \textbf{WTC3}$$

$$\frac{P,E \vdash e :: T \qquad \mathit{is\text{-}refT}\ T \qquad T \neq \mathit{NT}}{P,E \vdash e{\bullet}\mathit{notify}([]) :: \mathit{Void}}\ \textbf{WTC4}$$

$$\frac{P,E \vdash e :: T \qquad \mathit{is\text{-}refT}\ T \qquad T \neq \mathit{NT}}{P,E \vdash e{\bullet}\mathit{notifyAll}([]) :: \mathit{Void}}\ \textbf{WTC5}$$

$$\frac{P,E \vdash o' :: T \qquad \mathit{is\text{-}refT}\ T \qquad T \neq \mathit{NT} \qquad P,E \vdash e :: T'}{P,E \vdash \mathit{sync}(o')\ e :: T'}\ \textbf{WTS}$$

Figure 11. Jinja typing rules for method call and the synchronized statement.

$$\frac{P,E,h \vdash e : \mathit{NT} \qquad \mathit{list\text{-}all2}\ (\lambda e\ T.\ P,E,h \vdash e : T)\ \mathit{es}\ \mathit{Ts}}{P,E,h \vdash e{\bullet}M(\mathit{es}) : T}\ \textbf{WTCN}$$

$$\frac{P,E,h \vdash o' : \mathit{NT} \qquad P,E,h \vdash e : T}{P,E,h \vdash \mathit{sync}(o')\ e : T'}\ \textbf{WTSN}$$

Figure 12. Jinja runtime typing rules for the *null* type in method call and the synchronized statement.

class C sees the method M with parameter types Ts and return type T in class D which has parameter names pns and body body, taking method overriding into account. $\mathit{list\text{-}all2}\ (\lambda e\ T.\ P,E \vdash e :: T)\ \mathit{es}\ \mathit{Ts}'$ denotes that $|\mathit{es}| = |\mathit{Ts}'|$ and the parameters es have types Ts'. As before, this rule cannot subsume the hard-wired methods for thread creation, *wait*, *notify*, and *notifyAll*, because they do not have an implementation in the Jinja language. Thus, we introduce one extra rule for each of them where *is-refT* is a predicate for reference types (rules WTC2, WTC3, WTC4 and WTC5). The rule for the synchronized statement WTS is straightforward. Note that we include the condition $T \neq \mathit{NT}$ in these rules to disallow expressions like $\mathit{null}{\bullet}\mathit{wait}([])$, since *null* cannot directly be dereferenced in Java [9].

However, the static type system is not preserved under reductions. In particular, during reduction, explicit addresses may occur inside expressions which are not typable without knowing the heap. Thus, we also have another (runtime) type system [5], which takes the heap into account and which is denoted by $P,E,h \vdash e : T$. It relaxes some constraints imposed by the static type system, for details see [12]. Most importantly, in addition to each rule in Fig. 11, which also exist in the runtime type system with the h added to them, there is another rule for typing the case when an expression typed with a non-*null* reference type reduces to *null*. Fig. 12 shows the typing rules for method call WTCN, which also subsumes the new native methods, and the synchronized statement WTSN. Note that, e.g., the expression $\mathit{null}{\bullet}\mathit{wait}([])$ now has arbitrary type, not even a unique least type, however, since this is only technical device for the type safety proof, this does not matter to us when typing an expression statically in the type system above.

LEMMA 13. *If* $P,E \vdash e :: T$ *then* $P,E,h \vdash e : T$.

5.3.2 Progress

For both progress and preservation, the initial configuration must conform to the type constraints given by the program in various aspects. First, all field contents of all objects on the heap h must have a type which conforms to the type given in the field declaration, which we denote by $P \vdash h \surd$. Similarly, every local variable in x must hold a value which is compatible with the typing environment E, denoted by $P,h \vdash x\ (:\le)\ E$. If both heap h and local store x are conform, we write $P,E \vdash (h, x) \surd$ as shorthand. Details for conformance can be found in [12], the extension to the Jinja thread semantics is straightforward. For progress, we lift typability to the multithreaded setting. $P,h \vdash_{::} \mathit{es} \surd$ holds iff for every thread t, there exists a typing environment in which t's expression is (runtime) typable. With the runtime type system, we can carry over the induction proof on red P for progress to the thread extension under the same conditions:

LEMMA 14 (Progress for threads in isolation).

$$\frac{\mathit{wf\text{-}J\text{-}prog}\ P \qquad P \vdash h \surd \qquad P,E,h \vdash e : T \qquad \mathcal{D}\ e\ x \qquad \neg\,\mathit{final}\ e}{\exists\, e'\ s'\ \mathit{tas}.\ P \vdash \langle e,(h, x)\rangle\ {-}\mathit{tas}{-}\mathit{red}{\rightarrow}\ \langle e',s'\rangle}$$

Hence, we have:

LEMMA 15 (*wf-progress* for *red P*).

$$\frac{\mathit{wf\text{-}J\text{-}prog}\ P \qquad P \vdash h \surd \qquad P,h \vdash_{::} \mathit{es} \surd \qquad \Uparrow\!\mathcal{D}\!\Uparrow \mathit{es}\ h}{\mathit{wf\text{-}progress}\ \mathit{red}\ P\ \mathit{es}\ h}$$

For progress of the framework semantics, we also need to show that the instantiating Jinja semantics is well-behaved:

LEMMA 16.
If $\vdash \mathit{es} \Uparrow\!\surd_{\&}\!\Uparrow$ and $\mathit{es} \vdash_{\mathrm{e}} \mathit{ls} \surd$ then $\mathit{wf\text{-}r\text{-}progress}\ \mathit{red}\ P\ \mathit{ls}\ \mathit{es}\ h\ t$.

Together with Thm. 1 and Lem. 4, we get progress for the framework semantics:

THEOREM 2 (Progress).

$$\frac{\begin{array}{cccc} & \mathit{wf\text{-}J\text{-}prog}\ P & P,h \vdash_{::} \mathit{es} \surd & \\ \Uparrow\!\mathcal{D}\!\Uparrow \mathit{es}\ h & \vdash \mathit{es} \Uparrow\!\surd_{\&}\!\Uparrow & \mathit{es} \vdash_{\mathrm{e}} \mathit{ls} \surd & P \vdash h \surd \\ \mathit{es}\ t = \lfloor (e, x) \rfloor & \neg\,\mathit{final}\ e & \multicolumn{2}{c}{t \notin \mathit{deadlocked}\ \mathit{red}\ P\ \mathit{ls}\ \mathit{es}\ \mathit{ws}\ h}\end{array}}{\exists\, t\ \mathit{tas}\ \mathit{es}'\ \mathit{ls}'\ \mathit{ws}'\ h'.\ P,\mathit{red} \vdash \langle \mathit{ls}|\mathit{es},h|\mathit{ws}\rangle\ {-}t,\mathit{tas}{\rightarrow}\ \langle \mathit{ls}'|\mathit{es}',h'|\mathit{ws}'\rangle}$$

5.3.3 Preservation

For preservation, we combine conformance conditions and typability in a single predicate. First, let $P,E,h \vdash e \le: T$ be a shorthand for $\exists\, T'.\ (P,E,h \vdash e : T' \wedge P \vdash T' \le T))$, i.e. e is typable in the environment E and heap h with a type T' which is a subtype of T. However, this is not the only such condition where the typing environment E is used: $P,E \vdash (h, x) \surd$ also depends on E. All in all, we define $P,(E, T) \vdash e,x,h \surd \equiv P,E \vdash (h, x) \surd \wedge P,E,h \vdash e \le: T$ and lift it to the multithreaded setting:

$$P,\mathit{Es} \vdash \mathit{es},h \Uparrow\!\surd\!\Uparrow\ \equiv\ \Uparrow\!(\lambda(E, T)\ e\ h\ x.\ P,(E, T) \vdash e,x,h \surd)\!\Uparrow \mathit{Es}\ \mathit{es}\ h$$

Note that this time, since the typing environment and the initial type must remain the same during reduction, we use a map to invariant data Es which is a pair of a typing environments E and a type T. Hence, for every thread t, $\mathit{Es}\ t$ stores both the typing environment for t's local variables and the type of t's initial expression.

The next lemma, which follows from a number of different other preservation lemmata not presented here, shows that $P,\mathit{Es} \vdash \mathit{es},h \Uparrow\!\surd\!\Uparrow$ is invariant for well-formed programs if threads respect definite assignment.

LEMMA 17. If $\mathit{wf\text{-}J\text{-}prog}\ P$ then $P,\mathit{red},\mathcal{D} \vdash (P,\cdot \vdash \cdot,\cdot,\cdot \surd) \surd\!\rightarrow$.

Subject reduction, the key lemma for preservation, immediately follows from this using Lem. 7:

THEOREM 3 (Subject reduction).

$$\frac{\begin{array}{c}\mathit{wf\text{-}J\text{-}prog}\ P \qquad P,\mathit{red} \vdash \langle \mathit{ls}|\mathit{es},h|\mathit{ws}\rangle\ {-}t,\mathit{tas}{\rightarrow}\ \langle \mathit{ls}'|\mathit{es}',h'|\mathit{ws}'\rangle\\ \Uparrow\!\mathcal{D}\!\Uparrow \mathit{es}\ h \qquad P,\mathit{Es} \vdash \mathit{es},h \Uparrow\!\surd\!\Uparrow\end{array}}{P,(\mathit{Es}\ [\mathcal{I}{\leadsto}]_{P,\cdot \vdash \cdot,\cdot,\cdot \surd}\ \mathit{tas}) \vdash \mathit{es}',h' \Uparrow\!\surd\!\Uparrow}$$

5.3.4 Type safety

We use the preservation lemmata for the various conditions to lift all single-step reductions to the transitive reflexive closure by induction. With Thm. 2 and 3, this gives the final type safety theorem:

THEOREM 4 (Type safety).

$$\begin{array}{c}
\textit{wf-J-prog}\ P \\
es \vdash_i Es \ \surd \qquad P,Es \vdash es,h \Uparrow\surd\Uparrow \qquad \Uparrow\mathcal{D}\Uparrow\ es\ h \qquad es \vdash_{\mathrm{e}} ls \ \surd \\
\vdash es \Uparrow\surd_{\&}\Uparrow \qquad P,red \vdash \langle ls|es,h|ws\rangle \ {-tas{\longrightarrow}*}\ \langle ls'|es',h'|ws'\rangle \\
\nexists t\ ta\ es''\ ls''\ ws''\ h''.\ P,red \vdash \langle ls'|es',h'|ws'\rangle \ {-t,ta{\longrightarrow}}\ \langle ls''|es'',h''|ws''\rangle \\
Es' = Es\ [\mathcal{I}{\leadsto}]_{P,\cdot\vdash\cdot,\cdot,\cdot\ \surd}\ \textit{flatten}\ (\textit{map snd tas}) \\
\hline
\begin{array}{l}
Es \trianglelefteq Es' \wedge \\
(\forall t\ e'.\ \exists x'.\ es'\ t = \lfloor(e', x')\rfloor \longrightarrow \\
\quad (\exists v.\ e' = \textit{Val}\ v \wedge (\exists E\ T.\ Es'\ t = \lfloor(E, T)\rfloor \wedge P,h' \vdash v :\leq T)) \vee \\
\quad (\exists a.\ e' = \textit{Throw}\ a \wedge a \in \textit{dom}\ h') \vee \\
\quad (t \in \textit{deadlocked red}\ P\ ls'\ es'\ ws'\ h' \wedge (\exists E\ T.\ Es'\ t = \lfloor(E, T)\rfloor \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad P,E,h' \vdash e' \leq: T))
\end{array}
\end{array}$$

Let us examine what Thm. 4 states: Suppose we have:

- a well-formed Jinja program P, and
- there is a typing environment and an initial type in Es for thread ID t iff there is a thread with ID t in es, and
- for each thread t in es, say $es\ t = \lfloor(e, x)\rfloor$ and $Es\ t = \lfloor(E, T)\rfloor$, its expression e and local variable store x is typable in the type environment E and the heap h with a subtype of T, and
- every thread passes the definite assignment check, and
- locks are held only by non-final threads, and
- no thread can be reduced such that the monitor subexpression of a synchronized statement reduces to $\textit{locked}(a)$ for some address a, and
- the state (ls, es, c, ws) reduces to some state (ls', es', c', ws') with thread action lists tas such that
- (ls', es', h', ws') is in normal form.

Let Es' denote the map to typing environments and initial types Es which is extended with the environments and types for threads that are newly created in tas. Note that for threads t that have already been present in Es, $Es'\ t = Es\ t$, by $Es \trianglelefteq Es'$, i.e. type environments and initial types have remained the same for them. Then, for every thread t in es', say $es'\ t = \lfloor(e', x')\rfloor$, one of the following cases holds:

1. e' is a final value v, whose type is a subtype of t's initial type, or
2. e' throws an object that exists on the heap h, or
3. t is deadlocked and e' is typable with a subtype of t's initial type.

6. Related work

There are a number of formal semantics for Java on the source code level [21, 5, 22, 12], all of which model different subsets of sequential Java. Our basis for the sequential part is [12], which is a successor to [22]. In [4], Cenciarelli et al. give a formal semantics of multithreaded Java on the source code level, which includes most Java thread features such as dynamic thread running and stopping, synchronisation via monitors and the wait/notify mechanism. In particular, using event spaces, they carefully model the memory model for Java 2 [8], which is now out-dated. However, they neither give a type system nor do they prove any meta-theoretic results on their semantics. In contrast, they say:

> Event spaces are not necessarily "complete" [. . .] In fact, there are well-formed event spaces which are not completable, and this complicates the meta-theory of the semantics.

Since we are aiming for type safety via progress and preservation, we cannot resort to their semantics because non-completable event spaces would break the progress proof.

Stärk et al. [19] also present a multithreaded semantics of Java (without the JMM) based on abstract state machines together with a proof for preservation. However, they do not consider deadlocks, neither do they give a proof for progress. Moreover, their proofs are not checked by a theorem prover.

On the bytecode level, Belblidia and Debabbi present a formal small-step semantics for Java bytecode [3] which also features threads, but not the JMM. Like our approach, they have a semantics for threads in isolation and a second layer which manages the threads and receives thread actions, which they call labels, from them. In contrast to our framework semantics, at most one action can be issued at a time, but their single-thread semantics already takes care of the locks, which are stored in the shared memory, i.e. they only have actions for creating, killing, blocking and notifying threads. Yet, they do not model the wait/notify mechanism, which – strictly speaking – is not an integral part of Java bytecode, but of the `java.lang` package [14]; their block and notify actions are used by the second layer to keep track of which threads are ready for execution. Like Cenciarelli et al. in [4], they only give the semantics, but no type system and no discussion about deadlocks. Our framework semantics also handles the locks, i.e. an instantiating semantics need not care about other threads, and is generic and far more versatile, in particular, because arbitrary lists of thread actions can be passed in a single reduction. Their second layer, however, is tailored to their single-thread semantics for Java bytecode.

In [15], Liu and Moore present a monolithic formal semantics for multithreaded Java bytecode in form of an interpreter which also models class loading and initialisation. They aim for verifying JVM implementations w.r.t. the JVM specification and small Java programs in ACL2. However, they do not give a type system, nor do they comment on the deadlock issues.

Apart from showing type safety for the Java type system, type systems have proven useful for other safety features. Flanagan and Abadi [6] came up with an object calculus and a type system with dependent types to ensure that data races in accessing object members cannot occur. A data race occurs if two threads can access a location simultaneously without synchronisation, which can result in a corrupted state. Object members are annotated with locks' names, the type system ensures that accessing a member is only possible if the specified lock is held by the thread. An appropriate subject reduction theorem shows soundness. However, they do not provide any progress results. Flanagan and Freund [7] translated this calculus to full Java bytecode and implemented it in the `rccjava` tool. In [10], Grossman extends the approach of [6, 7] to multithreaded Cyclone, which is a type safe variant of C. He also shows the progress property that no well-typed thread can get "badly stuck". A thread t is "badly stuck" iff t cannot reduce any further even if t could acquire an arbitrary additional lock and, in case being final, t still holds some lock. Together with the subject reduction theorem, type safety, i.e. all threads reachable from a well-typed thread via reductions are not badly stuck, follows. In general, being in deadlock is stronger than being badly stuck because the latter does not involve the aspect of circular waiting.

Programs without data races are a desired kind of Java programs: The new JMM ensures that such programs have sequentially consistent behaviour. Aspinall and Ševčík [2] have formalised data race freedom and the memory model in Isabelle/HOL and proved this guarantee.

There are also approaches to eliminate potential deadlocks via type systems: For example, Suenaga and Kobayashi [20] propose a process calculus with thread creation, interrupts (which can be temporarily disabled), and synchronisation via structured locking. They assign to each syntactic occurrence of a lock a unique level tag. Their type system remembers bounds on the level of acquired locks in effect labels and ensures deadlock freedom by requiring that locks must be acquired in ascending order. Their deadlock formalisation is on the syntactic level only: A set of threads is in deadlock iff every "reducible" subexpression of the threads' expressions must be a synchronisation statement which has to acquire a lock which is already held. Consequently, they cannot express that some threads are in deadlock while others are still active. Moreover, in their model, a thread cannot acquire a lock multiple times as it is the case in Java. In contrast to that, our approach is defined in terms of the semantics, handles the extra cases introduced by wait sets and computes whether a given thread is in deadlock – even if some threads are not yet in deadlock.

Another interesting issue is to ensure that every lock acquired is eventually released and that a lock must have been acquired before being released, i.e. to avoid objects being locked forever and `IllegalMonitorStateExceptions` being raised at `monitorexit` instructions. For Java on the source level, this is no issue because locks are reliably acquired and released in a structured fashion by `synchronized` blocks and methods, but when we want to extend our formalisation to bytecode, `monitorenter` need not always be matched by `monitorexit`, which violates the $es \vdash_f ls \surd$ well-formedness condition of our framework. In particular, if an exception is raised while the lock is held and the exception handler is outside the synchronized section, things get tricky. Iwama and Kobayashi [11] propose to tag every object with a usage label which specifies a policy on how this object may be locked. A type system, for which they also give a type inference algorithm, guarantees that method implementations respect the usage tags, and the subject reduction theorem ensures soundness. However, their subset of Java bytecode is rather restricted in not allowing object fields, method invocations and inheritance. To tackle this problem of proper lock acquisition and release, Laneve [13] presents an operational semantics and a type system for a slightly larger subset of Java bytecode which includes both synchronisation and the wait/notify mechanism. It enforces the structured locking principle that is known from Java source code. Hence, the soundness proof gives that well-typed programs are free of `IllegalMonitorStateExceptions`, even for calls to `wait` and `notify`. He also discusses the intricacies introduced by exception handling in detail. However, his type system is not Java's and he does not mention progress.

7. Conclusion and future work

We have presented a general framework for adding concurrency to a single-threaded, operational semantics, in the proof assistant Isabelle/HOL. The framework semantics unhinges thread management and synchronisation from the instantiating semantics, thus allowing for modular proofs of meta-theoretic properties such as type safety. Further, we have given a formal definition of deadlock in terms of the semantics. In extending the Jinja source code semantics and applying the framework to it, we have type safety for multithreaded Java machine-checked, via progress and preservation.

For the future, we plan to extend the Jinja bytecode semantics to threads and to apply the framework there, too. [3] will be a good start there. Since our framework requires that every thread releases all locks it has acquired before terminating, we may want to recourse on [11] or [13]. When we then also add the JMM, Aspinall's and Ševčík's work [2] will be a good basis. Beyond Java, we also plan to add posix-like threads to a formal semantics of C++ [23] in Isabelle/HOL.

Acknowledgments

We would like to thank Daniel Wasserrab and Dennis Giffhorn for their valuable comments and the inspiring discussions we had with them.

References

[1] J. Alves-Foss, editor. *Formal Syntax and Semantics of Java*. LNCS 1523. Springer, 1999.

[2] D. Aspinall and J. Ševčík. Formalising Java's Data Race Free Guarantee. In *TPHOLs*, pages 22–37, 2007.

[3] N. Belblidia and M. Debbabi. A Dynamic Operational Semantics for JVML. *Journal of Object Technology*, 6(3):71–100, 2007.

[4] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Operational Semantics of Multi-Threaded Java. In Alves-Foss [1], pages 157–200.

[5] S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In Alves-Foss [1], pages 41–82.

[6] C. Flanagan and M. Abadi. Object Types against Races. In *CONCUR*, pages 288–303, 1999.

[7] C. Flanagan and S. N. Freund. Type-Based Race Detection for Java. In *PLDI*, pages 219–232, 2000.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.

[9] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. Addison-Wesley, 2005.

[10] D. Grossman. Type-Safe Multithreading in Cyclone. In *TLDI*, pages 13–25, 2003.

[11] F. Iwama and N. Kobayashi. A New Type System for JVM Lock Primitives. In *ASIA-PEPM*, pages 71–82, 2002.

[12] G. Klein and T. Nipkow. A Machine-Checked Model for a Java-Like Language, Virtual Machine and Compiler. *TOPLAS*, 28(4):619–695, 2006.

[13] C. Laneve. A Type System for JVM Threads. Technical Report TCS 290, University of Bolonga, 2003.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Addison-Wesley, 1999.

[15] H. Liu and J S. Moore. Executable JVM Model for Analytical Reasoning: A Study. In *IVME*, pages 15–23, 2003.

[16] Formalisation of the framework and the Java semantics online at http://www.infosun.fim.uni-passau.de/st/staff/lochbihl/jt.zip.

[17] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[18] G. Nutt. *Operating Systems*. Addison-Wesley, 2nd edition, 2000.

[19] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer, 2001.

[20] K. Suenaga and N. Kobayashi. Type-Based Analysis of Deadlock for a Concurrent Calculus with Interrupts. In *ESOP*, pages 490–504, 2007.

[21] D. Syme. Proving Java Type Soundness. In Alves-Foss [1], pages 83–118.

[22] D. von Oheimb and T. Nipkow. Machine-Checking the Java Specification: Proving Type-Safety. In Alves-Foss [1], pages 119–156.

[23] D. Wasserrab, T. Nipkow, G. Snelting, and F. Tip. An Operational Semantics and Type Safety Proof for Multiple Inheritance in C++. In *OOPSLA*, pages 345–362, 2006.

[24] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.