Animating the Formalised Semantics of a Java-like Language

Andreas Lochbihler[1] and Lukas Bulwahn[2]

[1] Karlsruher Institut für Technologie, andreas.lochbihler@kit.edu
[2] Technische Universität München, bulwahn@in.tum.de

Abstract. Considerable effort has gone into the techniques of extracting executable code from formal specifications and animating them. We show how to apply these techniques to the large *JinjaThreads* formalisation. It models a substantial subset of multithreaded Java source and bytecode in Isabelle/HOL and focuses on proofs and modularity whereas code generation was of little concern in its design. Employing Isabelle's code generation facilities, we obtain a verified Java interpreter that is sufficiently efficient for running small Java programs. To this end, we present refined implementations for common notions such as the reflexive transitive closure and Russell's definite description operator. From our experience, we distill simple guidelines on how to develop future formalisations with executability in mind.

1 Introduction

In the last few years, substantial work has been devoted to the techniques and tools for executing formal specifications from Isabelle/HOL, on the levels of both prover infrastructure [5,8,9] and formalisations of foundational notions and concepts [6,11,18]. But so far, applications (e.g. [4,19,20]) have been designed for executability and restricted to purely functional specifications. A benchmark to test whether the aforementioned techniques mix well and scale to large formalisations has been missing.

In this work, we study how to apply code generation techniques to the *JinjaThreads* project [15,16,17], which formalises a substantial subset of multithreaded Java source and bytecode. JinjaThreads constitutes a good benchmark for three reasons: (i) It is a large formalisation (70k lines of definitions and proofs) that involves a broad range of advanced Isabelle features. (ii) As a programming language, type system, and semantics, it has a built-in notion of execution. This sets the goal for what should be executable. (iii) It focuses on proofs and modularity rather than code generation, i.e. complications in specifications and proofs for the sake of direct code generation were out of the question. Hence, it tests if code generation works "in the wild" and not only for specialised developments.

Our main contribution here is to discuss what was needed to automatically generate a well-formedness checker and an interpreter for JinjaThreads programs from the Isabelle formalisation, and what the pitfalls were. Thereby, we demonstrate how to combine the different techniques and tools such that changes to the existing formalisation stay minimal. Our contributions fall into two parts.

On the system's side, we enhanced Isabelle's code generator for inductive predicates (§2.1) to obtain a mature tool for our needs. It now compiles inductive definitions and first-order predicates, interpreted as logic programs, to functional implementations. Furthermore, we present a practical method to overcome the poor integratability of Isabelle's code generator into Isabelle's module system (§2.2). Finally, we describe a tabled implementation of the reflexive transitive closure (§2.3) and an executable version of Russell's definite description operator (§2.4), which are now part of the Isabelle/HOL library.

On JinjaThreads' side, we animated the formalisation (see §3.1 for an overview) through code generation: Many of its inductive definitions, we had to refine for compilation or, if this was impossible, implement manually (§3.2 and §3.3). To obtain execution traces of JinjaThreads programs, we adapted the state representation and formalised two schedulers (§3.4). In §3.5, we explain how to add memoisation to avoid frequently recomputing common functions, e.g. lookup functions, without polluting the existing formalisation. Clearly, as the generated code naively interprets source code programs, we cannot expect it to be as efficient as an optimising Java virutal machine (JVM). Nevertheless, we evaluated the performance of the generated interpreter (§3.6). Simple optimisations that we describe there speed up the interpreter by three orders of magnitude. Hence, it is sufficiently efficient to handle Java programs of a few hundred lines of code.

We conclude our contributions by distilling our experience into a few guidelines on how to develop formalisations to be executable ones. Rather than imposing drastic changes on the formalisation, they pinpoint common pitfalls. §4 explains why and how to avoid them.

The interpreter and the full formalisation is available online in the Archive of Formal Proofs [17]. To make the vast supply of Java programs available for experimenting and testing with the semantics, we have written the (unverified) conversion tool Java2Jinja as a plug-in to the Eclipse IDE. It converts Java class declarations into JinjaThreads abstract syntax. The latest development version is available at `http://pp.info.uni-karlsruhe.de/git/Java2Jinja/`.

1.1 Related work

Code generation (of *functional implementations*) from Isabelle/HOL is a well-established business. Marić [19] presents a formally verified implementation of a SAT solver. In the CeTA project, Thiemann and Sternagel [20] generate a self-contained executable termination checker for term rewriting systems. The Flyspeck project uses code generation to compute the set of tame graphs [4]. All these formalisations were developed with executability in mind. Complications in proofs to obtain an efficiently executable implementation were willingly taken and handling them are large contributions of these projects.

Code generation in Coq [13] has been used in various developments, notably the CompCert compiler [12] and the certificate checkers in the MOBIUS project [3]. Like in Isabelle, functional specifications pose no intrinsic problems. Although code extraction is in principle possible for any Coq specification, mathematical theories can lead to "a nightmare in term of extracted code efficiency

and readability" [13]. Hence, Coq's users, too, are facing the problem of how to extract (roughly) efficient code from specifications not aimed towards executability. ACL2 and PVS translate only functional implementations to Common Lisp.

In [5], we have reported on generating code from *non-functional specifications*. Recently, Nipkow applied code generation for inductive predicates to animate the semantics and various program analyses of an educational imperative language (personal communication). All these applications were tiny formalisations compared to JinjaThreads.

Some *formalisations of the JVM* in theorem provers are directly executable. The most complete is the M6 model of a JVM by Lui and Moore [14] in ACL2, which covers the CLDC specification. Farzan et al. [7] report on a JVM formalisation in Maude's rewriting logic. ACL2's and Maude's logics are directly executable, i.e., they force the user to write only executable formalisations. While JinjaThreads studies meta-language properties like type safety for a unified model of Java and Java bytecode, these JVM formalisations aim at verifying properties of individual programs. Atkey [1] presents an executable JVM model in Coq. He concentrates on encoding defensive type checks as dependent types, but does not provide any data on the efficiency.

1.2 Background: the Code Generator Framework and Refinement

Isabelle's code generator [9] turns a set of equational theorems into a functional program with the same equational rewrite system. As it builds on equational logic, the translation guarantees partial correctness by construction and the user may easily refine programs and data without affecting her formalisation globally. *Program refinement* can separate code generation issues from the rest of the formalisation. As any (executable) equational theorem suffices for code generation, the user may *locally* derive new (code) equations to use upon code generation. Hence, existing definitions and proofs remain unaffected, which has been crucial for JinjaThreads.

For *data refinement*, the user may replace constructors of a datatype by other constants and derive equations that pattern-match on these new (pseudo-)constructors. Neither need the new constructors be injective and pairwise disjoint, nor exhaust the type. Again, this is local as it affects only code generation, but not the logical properties of the refined type. Conversely, one cannot exploit the type's new structure inside the logic. Only type constructors can be refined; some special types (such as $'a \Rightarrow 'b$ *option* for maps) must first be wrapped in an (isomorphic) type of their own (e.g. $('a,'b)$ *mapping*).

Isabelle's standard library defines such special-purpose types for sets and maps with standard operations. Associative lists and red-black trees implement them via data refinement. FinFuns [18] are almost-everywhere constant functions; they provide an executable universal quantifier thanks to data refinement to associative lists. The Isabelle Collections Framework (ICF) [11] advocates dealing with refinement *in* the logic instead of hiding it in the code generator. Locales, i.e. Isabelle modules, specify the abstract operations, concrete implementations interpret them. This allows for executing truly underspecified functions.

2 Code Generation in Isabelle

In this section, we present our contributions that JinjaThreads has motivated, but that are generally applicable. Consequently, they have been integrated into Isabelle's main system and library. First, we present the code generator for inductive predicates and our improvements to it (§2.1). Then, we describe our approach to overcome the problematic situation with code generation and locales (§2.2). Finally, we sketch formalisations for enhanced implementations for the reflexive transitive closure (§2.3) and the definite description operator (§2.4), which are employed in JinjaThreads' type system, for example.

2.1 The Predicate Compiler

The *predicate compiler* [5] translates specifications of inductive predicates, i.e. the introduction rules, into executable equational theorems for Isabelle's code generator. The translation is based on the notion of *modes*. A mode partitions the arguments into input and output. For a given predicate, the predicate compiler infers the set of possible modes such that all terms are ground during execution. Lazy sequences handle the non-determinism of inductive predicates. By default, the equations implement a Prolog-style depth-first execution strategy. Since its initial description [5], we improved the predicate compiler in four aspects:

First, mode annotations restrict the generation of code equations to modes of interest. This is necessary because the set of modes is exponential in the number of arguments of a predicate. Therefore, the space and time consumption of the underlying mode inference algorithm grows exponentially in that number; for all applications prior to JinjaThreads, this has never posed a problem. In case of many arguments (up to 15 in JinjaThreads), the plain construction of this set of modes burns up any available hardware resource. To sidestep this limitation, modes can now be declared and hence they are not inferred, but only checked to be consistent.

Second, we also improved the compilation scheme: The previous one sequentially checked which of the introduction rules were applicable. Hence, the input values were repeatedly compared to the terms in the conclusion of each introduction rule by pattern matching. For large specifications, such as JinjaThreads' semantics (contains 88 rules), this naive compilation made execution virtually impossible due to the large number of rules. To obtain an efficient code expression, we modified the compilation scheme to partition the rules by patterns of the input values first and then only compose the matching rules – this resembles similar techniques in Prolog compilers, such as clause indexing and switch detection. We report on the performance improvements due to this modification in §3.6.

Third, the predicate compiler now offers non-intrusive program refinement, i.e., the user can declare alternative introduction rules. For an example, see §3.3.

Fourth, the predicate compiler was originally limited to the restricted syntactic form of introduction rules. We added some preprocessing that transforms definitions in predicate logic to a set of introduction rules. Type-safe method overriding (§3.2) gives an example.

2.2 Isabelle Locales and Code Generation

Locales [2] in Isabelle allow parametrised theory and proof development. In other words, locales allow to prove theorems abstractly, relative to a set of *fixed parameters and assumptions*. Interpretation of locales transfers theorems from their abstract context to other (concrete) contexts by instantiating the parameters and proving the assumptions. JinjaThreads uses locales to abstract over different memory consistency models (§3.3) and schedulers (§3.4), and to underspecify operations on abstract data structures.

As code generation requires equational theorems in the (foundational) theory context, theorems that reside in the context of a locale cannot serve as code equations directly, but must be transferred into the theory context. For example, consider a locale L with one parameter p, one assumption $A\ p$ and one definition $f = \ldots$ that depends on p. Let g be a function in the theory context for which $A\ (g\ z)$ holds for all z. We want to generate code for f where p is instantiated to $g\ z$.

The Isabelle code generator tutorial proposes *interpretation and definition*: One instantiates p by $g\ z$ and discharges the assumption with $A\ (g\ z)$, for arbitrary z. This yields the code equation $f\ (g\ z) = \ldots$ which is ill-formed because the left-hand side applies f to the non-constructor constant g. For code generation, one must manually define a new function f' by $f'\ z = f\ (g\ z)$ and derive $f'\ z = \ldots$ as code equation. This approach is unsatisfactory for two reasons: It requires to manually re-define all dependent locale definitions in the theory context (and for each interpretation), and the interpretation must be unconditional, i.e., $A\ (g\ z)$ must hold for *all* z. In JinjaThreads, the latter is often violated, e.g. $g\ z$ satisfies A only if z is well-formed.

To overcome these deficiencies, our new approach *splits the locale* L into two: L_0 and L_1. L_0 fixes the parameter p and defines f; L_1 inherits from L_0, assumes $A\ p$, and contains the proofs from L. Since L_0 makes no assumptions on p, the locale implementation exports the equation $f = \ldots$ in L_0 as an unconditional equation $L_0.f\ p = \ldots$ in the theory context which directly serves as code equation. For execution, we merely pass $g\ z$ to $L_0.f$. We use this scalable approach throughout JinjaThreads. Its drawback is that the existence of a model for f, as required for its definition, must not depend on L's assumptions; e.g. the termination argument of a general recursive function must not require L's assumptions. Many typical definitions (all in JinjaThreads) satisfy this restriction.

2.3 Tabling the Reflexive Transitive Closure

The reflexive transitive closure (RTC) is commonly used in formalisations, also in JinjaThreads' subtyping relation. Here, we present how a simple refinement implements a tabling depth-first execution of the RTC. By default, the predicate compiler uses the two introduction rules below for code generation.

$$\frac{}{rtc\ r\ x\ x} \qquad\qquad \frac{r\ x\ y \qquad rtc\ r\ y\ z}{rtc\ r\ x\ z}$$

Compiling them in a Prolog-style depth-first fashion leads to non-termination when the underlying relation r has reachable cycles. Hence, Berghofer imple-

mented a tabled version of RTC that detects cycles and short-circuits the search in that case (cf. acknowledgements). The predicate *rtc-tab r xs x z* expresses that z is reachable in r from x without visiting any node in xs:

$$\frac{}{\textit{rtc-tab}\ r\ xs\ x\ x} \qquad \frac{x \notin \textit{set}\ xs \qquad r\ x\ y \qquad \textit{rtc-tab}\ r\ (x \cdot xs)\ y\ z}{\textit{rtc-tab}\ r\ xs\ x\ z}$$

For execution, the terminating *rtc-tab* implements RTC via program refinement with the equality $\textit{rtc}\ r = \textit{rtc-tab}\ r\ []$.

2.4 An Executable Definite Description Operator

Russell's definite description operator ι and Hilbert's choice ε extract a deterministic function from a relational formulation. Like any underspecified function, they pose a challenge for code generation [8], because their axiomatisations are not unconditional equations. Hence, we can only execute them via program refinement, i.e., we must derive such an equation from the axiomatisation. This is only possible for inputs for which the specification fixes a unique return value, e.g. singleton sets for ι and ε, as any implementation returns a fully specified value. Now, we construct an executable implementation for ι using the predicate compiler. Our execution strategy is as follows: We enumerate all values satisfying the predicate. If there is exactly one such value, we return it; otherwise, we throw a exception. To enumerate values efficiently, we rely on the predicate compiler.

For technical reasons, it works in terms of the type $'a\ \textit{pred}$ [5], which is isomorphic to $'a \Rightarrow \textit{bool}$. The *Pred* constructor and the *eval* selector allow to convert between $'a\ \textit{pred}$ and $'a \Rightarrow \textit{bool}$. Then, the type of sequences $'a\ \textit{seq}$ implements $'a\ \textit{pred}$ via data refinement with the lazy constructor $\textit{Seq} :: (\textit{unit} \Rightarrow 'a\ \textit{seq}) \Rightarrow 'a\ \textit{pred}$. The type $'a\ \textit{seq}$ has a richer structure with the constructors *Empty*, *Insert*, and *Join*. *Empty* and *Insert* are self-explanatory; *Join P xq* represents the union of the enumeration P and the values in the sequence xq.

First, we lift ι to $'a\ \textit{pred}$ by defining $\textit{the}\ A = (\iota x.\ \textit{eval}\ A\ x)$. Then, we define by (1) the operation $\textit{singleton} :: (\textit{unit} \Rightarrow 'a) \Rightarrow 'a\ \textit{pred} \Rightarrow 'a$ that returns for a singleton enumeration the contained element and a (lazy) *default* value otherwise. We prove (2) to implement *the* via *singleton*, which exploits reflexivity of HOL's equality for non-singleton enumerations.

$$\textit{singleton}\ \textit{default}\ A = (\textit{if}\ \exists!x.\ \textit{eval}\ A\ x\ \textit{then}\ \iota x.\ \textit{eval}\ A\ x\ \textit{else}\ \textit{default}\ ()) \quad (1)$$

$$\textit{the}\ A = \textit{singleton}\ (\lambda\_.\ \textit{the}\ A)\ A \quad (2)$$

Having refined $'a\ \textit{pred}$ to $'a\ \textit{seq}$, we prove (3) as code equation for *singleton*:

$$\begin{array}{l}\textit{singleton}\ \textit{default}\ (\textit{Seq}\ f) = (\textit{case}\ f\ ()\ \textit{of}\\ \quad \textit{Empty} \Rightarrow \textit{throw}\ \textit{default}\\ \mid \textit{Insert}\ x\ P\ \Rightarrow \textit{if is-empty}\ P\ \textit{then}\ x\\ \quad\quad \textit{else let}\ y = \textit{singleton}\ \textit{default}\ P\ \textit{in if}\ x = y\ \textit{then}\ x\ \textit{else throw}\ \textit{default}\\ \mid \textit{Join}\ P\ xq\ \Rightarrow \textit{if is-empty}\ P\ \textit{then the-only}\ \textit{default}\ xq\\ \quad\quad \textit{else if null}\ xq\ \textit{then singleton}\ \textit{default}\ P\\ \quad\quad\quad \textit{else let}\ x = \textit{singleton}\ \textit{default}\ P;\ y = \textit{the-only}\ \textit{default}\ xq\ \textit{in}\\ \quad\quad\quad\quad \textit{if}\ x = y\ \textit{then}\ x\ \textit{else throw}\ \textit{default})\end{array} \quad (3)$$

The predicate *is-empty* (*null*) tests if the enumeration (the sequence) contains no element. The operation *the-only* is *singleton*'s analogon for *'a seq* with a similar code equation. In HOL, *throw*, defined by *throw f = f ()*, just applies the *unit* value. The generated code for *throw* raises an exception without evaluating its argument, a *unit* closure. This ensures partial correctness for *singleton* and *the*, i.e., if the code terminates normally, the computed value is correct.

To execute definitions with Russell's ι operator, one proceeds as follows: Given a definition $c = (\iota x.\ P\ x)$, one runs the predicate compiler on P to obtain the function that enumerates x, i.e., the mode assigns the argument to be output. This yields an executable function *P-o* with $P = \mathit{eval}\ \mathit{P\text{-}o}$. Unfolding definitions, one obtains the code equation $c = \mathit{the}\ \mathit{P\text{-}o}$.

Note that the test $x = y$ in (3) requires that equality on the predicate's elements is executable. If this is not the case (e.g. functions as elements), we provide an altered equation where *throw default* replaces *if* $x = y$ *then* x *else throw default* in (3). Then, the computation also fails when the enumeration is actually a singleton, but contains the same element multiple times.

3 JinjaThreads: Well-formedness Checker and Interpreter

In this section, we first give an overview of JinjaThreads (§3.1). Then, we present how to obtain an executable well-formedness checker and interpreter for JinjaThreads, and what the pitfalls are (§3.2 to 3.4). We employ program and data refinement such that lookup functions are precomputed rather than recomputed whenever needed (§3.5). In §3.6, we evaluate the efficiency of the interpreter on a standard producer-consumer program.

3.1 Overview of JinjaThreads

Building on Jinja [10] by Klein and Nipkow, JinjaThreads models a substantial subset of multithreaded Java source and bytecode. Figure 1 shows the overall structure: The three major parts are the source and bytecode formalisations and a compiler between them. Source and bytecode share declarations of classes, fields and methods, the subtyping relation, and standard well-formedness constraints. The source code part defines the source code syntax, a single-threaded small-step semantics, and additional well-formedness constraints (such as a static type system and definite assignment). It contains a type safety proof via progress and preservation. The bytecode part formalises bytecode instructions, a virtual machine (VM) for individual threads, and a bytecode verifier. The type safety proof shows that verified bytecode cannot violate type checks in the defensive VM. For both parts, JinjaThreads defines two concurrent semantics: (i) interleaving semantics for the individual threads, which provides sequential consistency (SC) as memory consistency model (MCM) – schedulers allow to generate specific interleavings; (ii) the Java memory model (JMM) as an axiomatic specification of legal executions. Finally, the compiler translates source code into bytecode in two stages and is verified with respect to the concurrent semantics.

For all definitions in shaded boxes, we have generated code via Isabelle's code generator. We highlight the necessary steps using examples from well-formedness

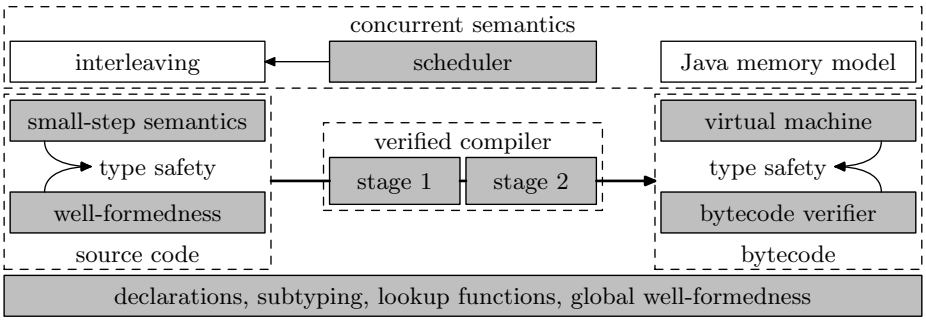

Fig. 1. Structure of JinjaThreads

(§3.2), the small-step semantics (§3.3), and the scheduler (§3.4). The compiler's definition, a functional implementation, is directly executable. The bytecode verifier requires adaptations similar to well-formedness. For the VM, we had to manually transform its functional specification to use Isabelle's special-purpose type for sets (c.f. §1.2). The JMM is purely axiomatic, finding an operational model would be a complicated task that we have not attempted.

3.2 The Type System and Well-formedness Conditions

A JinjaThreads program is given as a list of class declarations. Among others, *well-formedness* requires that its class hierarchy is acyclic with *Object* at the top, method overriding is type safe, and the program obeys the rules of the type system. Thus, a well-formedness checker must include a type checker. The type system relies on the subclass and subtype relation, least upper bounds (lub) w.r.t. the subtype relation, and lookup functions for fields and methods. To turn these into executable equations, we do the following:

The *subclass relation* $\preceq^*$ is the RTC of the direct subclass relation, which is defined inductively. As the standard execution mechanism for the RTC leads to non-termination in case of cyclic class hierarchies, we use the tabled RTC as described in §2.3. This ensures that querying $\preceq^*$ always terminates, i.e., we can reliably detect cyclic class hierarchies when checking well-formedness.

The *subtype relation* $:\leq$, another inductive predicate, extends $\preceq^*$ to arrays and primitive types. Checking whether one type is a subtype of another is executable. For acyclic class hierarchies with *Object* at the top, non-primitive types form an upper semi-lattice w.r.t. $:\leq$, i.e. unique lubs exist for existing types. However, compiling the declarative definition of lub to an executable function with the predicate compiler fails, because it would require to enumerate all supertypes of a given type. Therefore, we provide a functional implementation, join, to compute lubs for acyclic class hierarchies with *Object* at the top. For cyclic ones, lubs need not be unique, so the functional implementation's behaviour is undefined.

Field and method lookup recurse over the class hierarchy. To avoid definitional problems in case of cyclic class hierarchies, JinjaThreads defines them relationally as inductive predicates and the lookup functions using the definite description operator ι. We refine them to use the executable operator *the* following §2.4.

$$P, t \vdash \langle null.M(map\ Val\ vs), s\rangle \xrightarrow{\varepsilon} \langle THROW\ NullPointer, s\rangle\ \textsc{CallNull}$$

$$\frac{is\text{-}Vals\ es}{P, t \vdash \langle null.M(es), s\rangle \xrightarrow{\varepsilon} \langle THROW\ NullPointer, s\rangle}\ \textsc{CallNull2}$$

Fig. 2. Original and alternative introduction rule of the small-step semantics.

The *type system* $E \vdash e :: T$ for source code statements is defined inductively, too. Even type checking requires type inference. Consider, e.g. the rule below for assignments to a local variable V whose type T is given by the environment E:

$$\frac{E\ V = \lfloor T \rfloor \qquad E \vdash e :: U \qquad U :\leq T \qquad V \neq this}{E \vdash V := e :: Void}$$

When the predicate compiler compiles $\_ \vdash \_ :: \_$, it must choose either to enumerate all subtypes of T and type-check e against each, or to infer e's type and check for $U :\leq T$. Note that in case $V := e$ is type-incorrect, the former approach might not terminate as e.g. *Object* has infinitely many subtypes. To force the predicate compiler to choose the latter, we disallow enumeration of subtypes via mode annotations.

For type inference, the rule for the conditional operator `_ ? _ : _` in Java requires to compute the lub of types of the second and third argument. As the declarative definition of the type system uses the declarative lub definition, type inference (and thus type checking) is not executable. For code generation, we therefore copy the definition for $\_ \vdash \_ :: \_$, replacing lub with the executable join function. Then, we prove that both versions agree on acyclic class hierarchies with *Object* at the top, but we cannot refine the declarative definition because equality only holds under acyclicity.

Overriding method M with parameter types Ts and return type T in class C with direct superclass D is *type-safe* if

$$\forall Ts'\ T'\ m.\ \vdash D\ sees\ M : Ts' \to T' = m \implies Ts'\ [:\leq]\ Ts \wedge T :\leq T'$$

where $\vdash D\ sees\ M : Ts' \to T' = m$ denotes that D sees M with parameter types Ts', return type T' and body m. The predicate compiler preprocesses the condition to an inductive predicate and compiles it to an executable equation (cf. §2.1).

After these preparations, *well-formedness* no longer poses any difficulties for code generation. Note that all the setup relies on program refinement only, the existing formalisation remains untouched. Stating and proving alternative equations requires between 5 lines for $:\leq$ and 220 lines for the type system.

3.3 The Semantics

The small-step semantics is parametric in the MCM. Thus, we model shared memory abstractly in terms of read and write functions for values and type information as locale parameters, following the splitting principle from §2.2.

The small-step semantics for source code is another inductive predicate. The predicate compiler processes 84 of 88 introduction rules automatically. For the

others, we must provide alternative introduction rules via program refinement. Fig. 2 shows the rule CallNull, which is representative for the four, for thread t invoking the method M with parameter values vs on the *null* pointer in the state s, which raises a *NullPointer* exception. Mapping the injection *Val* of values into expressions over the list of values vs expresses that all parameters have already evaluated to values. This rule violates the desired mode for executing the semantics because its execution would require pattern-matching against the term *map Val vs*. The remedy is to declare the alternative introduction rule CallNull2: We replace *map Val vs* by es and instead use the guard *is-Vals es* that predicates that all elements in es are of the form *Val* v for some v. To access vs in other parts of the rule (as is necessary in one of the others), we replace vs with *map the-Val es* where *the-Val* is the destructor for the constructor *Val*.

Mode annotations for executing the small-step semantics are crucial. The abstraction of the MCM in a locale adds 6 parameters to the small-step semantics in the theory context, which consequently allows a monstrous number of modes.

For code generation, we only use SC as MCM, because the JMM is axiomatic and thus not executable. SC models the shared heap as a function from addresses (natural numbers) to objects and arrays. Allocation must find a fresh address, i.e. one not in the heap's domain. Originally, this was defined via Hilbert's underspecified (and thus not executable) ε operator (4). For code generation, we had to change *new-Addr*'s specification to the least fresh address, replacing ε with *LEAST*. Then, we proved (5) and (6) to search for the least fresh address.

$$\textit{new-Addr}\ h = \textit{if}\ (\exists a.\ h\ a = \textit{None})\ \textit{then}\ \lfloor \varepsilon a.\ h\ a = \textit{None} \rfloor\ \textit{else}\ \textit{None} \quad (4)$$

$$\textit{new-Addr}\ h = \textit{gen-new-Addr}\ h\ 0 \quad (5)$$

$$\textit{gen-new-Addr}\ h\ n = \textit{if}\ (h\ n = \textit{None})\ \textit{then}\ \lfloor n \rfloor\ \textit{else}\ \textit{gen-new-Addr}\ h\ (n{+}1) \quad (6)$$

3.4 The Scheduler

Executing the interleaving semantics poses three problems:

1. The multithreaded state consists of functions of type $\_ \Rightarrow \_\ \textit{option}$ for locks, thread-local states and the monitor's wait sets. Neither quantifying over these maps' domains (e.g. to decide whether all threads have terminated) nor picking one of its elements (e.g. to remove an arbitrary thread from a wait set upon notification) are executable.
2. The state space of all possible interleavings is usually too large to be effectively enumerable. Therefore, one wants to pick one typical interleaving.
3. JinjaThreads programs that might not terminate should at least produce a prefix of the observable operations of such an infinite run.

To address the first, we previously [18] proposed to replace these maps with FinFuns, a generalisation of finite maps. Although quantification over the domain then becomes executable, it turned out that choosing an underspecified element remains unexecutable. We therefore only use them for lock management. For the pool of thread-local states and the wait set, we instead follow the ICF approach [11]. We replace the functions with abstract operations whose signatures

and properties we specify in two locales. Picking an arbitrary element remains underspecified, but this is now explicit inside the logic, not HOL's metalogic. Before code generation, we instantiate the locales with concrete data structure implementations like red-black trees and thus resolve the underspecification.

As to the second problem, we do not use the predicate compiler, as it would produce a depth-first search that enumerates all possible interleavings. The first few interleavings would be such that one thread executes completely (or until it blocks), then the next thread executes completely, etc. Interesting interleavings would occur only very much later – or never at all, if one of the preceding ones did not terminate. Instead, we let a scheduler pick the next thread at each step.

Formally, a scheduler consists of two operations (that we specify abstractly in two locales again): The function *schedule* takes the scheduler's state and the multithreaded state, and returns either a thread together with its next transition and the updated scheduler state, or *None* to denote that the interleaving has finished or is deadlocked. The other function *wakeup* chooses from a monitor's wait set the thread to be notified. In terms of these two functions, we define a deterministic, executable step function that updates the multithreaded state just like the non-deterministic interleaving semantics does. To obtain a complete interleaving as a potentially infinite trace, we corecursively unfold this step function. Then, we formally prove that this in fact yields a possible interleaving.

We have instantiated this specification with two concrete schedulers: a round-robin scheduler and a random scheduler based on a pseudo-random number generator. The most intricate problem is how to obtain (as a function) the thread's step from the (relational) small-step semantics, once the scheduler has decided which thread to execute. Fortunately, the semantics under SC is deterministic, if we purge transitions whose preconditions are not met by the current state. Thus, we use the *the* operator again, but without equality checks (§2.4), as the result states contain functions (the heap) for which checking equality is not executable.

Corecursive traces also solve the third problem. We instruct the code generator to implement possibly infinite lists *lazily*. For Haskell, this is the default; for the other target languages, data and program refinement provide an easy setup.

Formalising the scheduler did not affect the rest of the formalisation. It required 2357 lines of definitions and proofs, 20% of which only declare locales.

3.5 Tabulation

An execution of a JinjaThreads (or, similarly, Java) program frequently checks type casts and performs method lookups. However, with the above setup, the semantics recomputes the subtype relation and lookup functions at every type cast and method call from scratch. Here, we show how to leverage program and data refinement to avoid such recomputations with only minimal changes to the formalisation itself. We precompute the subclass relation, field and method lookup (a standard technique for VMs) and store them in mappings (cf. §1.2). Fig. 3 sketches the necessary steps.

In JinjaThreads, a program declaration used to be a list of class declarations, i.e. of type $'m\ cdecl\ list$, abbreviated as $'m\ prog$. For data refinement (cf. §1.2), we turn the abbreviation into a type of its own, wrapping the old type (l. 1 in Fig. 3).

1 **datatype** $'m\ prog = Program\ 'm\ cdecl\ list$
2 **definition** *prog-impl-invar* $P'\ c\ s\ f\ m = (c = Mapping\ (class\ (Program\ P')) \wedge \ldots)$
3 **typedef** $'m\ prog\text{-}impl = \{(P', c, s, f, m)\,|\,prog\text{-}impl\text{-}invar\ P'\ c\ s\ f\ m\}$
morphisms *impl-of Abs-prog*
4 **definition** $ProgDecl = Program \circ fst \circ impl\text{-}of$
5 **code_datatype** *ProgDecl*
6 **lemma** [code]: $class\ (ProgDecl\ P) = lookup\ (fst\ (snd\ (impl\text{-}of\ P)))$
7 **definition** $tabulate\ P' = Abs\text{-}prog\ (P', tabulate\text{-}class\ P', tabulate\text{-}subcls\ P', \ldots)$
8 **lemma** [code]: $Program = ProgDecl \circ tabulate$

Fig. 3. Tabulation for lookup functions and the subclass relation.

Next, we define the type $'m\ prog\text{-}impl$ (l. 3). Apart from the original program declaration (as a list P'), its elements (P', c, s, f, m) consist of mappings from class names to (i) the class declaration (c), (ii) the set of its superclasses (s), and (iii) two mappings for field and method lookup with field and method names as keys (f and m). The invariant *prog-impl-invar* (l. 2) states that the mappings correctly tabulate the lookup functions and subclass relation. Then, we define (l. 4) and declare (l. 5) the new constructor $ProgDecl :: 'm\ prog\text{-}impl \Rightarrow 'm\ prog$ for data refinement, which (in the logic) only extracts the program declaration.

For the lookup functions, the subclass relation, and the associated constants that the predicate compiler has introduced, we next prove code equations that implement them via lookup in the respective mapping – see l. 6 for class declaration lookup. This program refinement suffices to avoid recomputing lookup functions and the subclass relation during execution.

However, the generated code now expects the input program to come with the correctly precomputed mappings. Thus, we define *tabulate* (l. 7) and auxiliary functions that tabulate the lookup functions and subclass relation in these mappings for a given list P' of class declarations. Finally, we implement the former constructor *Program* (l. 8) in terms of *tabulate* and *ProgDecl*.

As most of JinjaThreads treats a program declaration opaquely, introducing $'m\ prog$ as a type of its own was painless; we edited just 143 lines out of 70k, i.e. .2%. The remaining program and data refinement took about 600 lines.

3.6 Efficiency of the Interpreter

Although we cannot expect the generated interpreter to be as efficient as an optimising JVM, to see whether it is suited to run small programs, we have evaluated it on a standard producer-buffer-consumer example. The producer thread allocates n objects and enqueues them in the buffer, which can store 10 elements at the same time. Concurrently, the consumer thread dequeues n objects from the buffer. Table 1 lists the running times for different code generator setups. All tests ran on a Pentium DualCore E5300 2.6GHz with 2GB RAM using Poly/ML 5.4.1 and Ubuntu GNU/Linux 9.10.

With the adaptations from §3.2 to §3.4 only, the code is unbearably slow (column 1). For $n = 100$, interpreting the program takes 37 min, i.e. 2,240.3 s. As the main bottleneck, we identified the naive compilation scheme for the small-step semantics. By switching to the improved compilation scheme (column 2)

n	without adjustments	with indexing	almost strict	heap as red-black tree	with tabulation
10	229.9	1.9	.1	<.1	<.1
100	2,240.3	14.1	1.7	.7	.6
1,000	—	625.6	492.3	7.2	6.2
10,000	—	—	—	71.8	62.6

Table 1. Timing (in seconds) for running the producer-consumer example on n objects for different adjustments to the interpreter; — denotes timeout after 1h.

in the predicate compiler (§2.1), we sped up the interpreter by two orders of magnitude. The definite descriptor *the* that extracts the result configuration from the enumerations, strictly evaluates all branches. Hence, explicit laziness in the generated code is unnecessary. If we remove the most obvious constructions due to laziness from the code equations that we compiled under the improved scheme, a program run with $n = 100$ takes only 1.7 s (column 3).

As n increases, another bottleneck shows up: memory allocation (cf. §3.3). Since the heap is modelled as a function and writes as function updates, i.e. closures, finding the next fresh address takes time quadratic in the number of previous allocations. Thus, interpreting the example program is quadratic in n although the program itself only requires linearly many steps. To speed up allocation and read access, we replaced the function by a red-black tree with addresses as keys. Combined with the other improvements, this already provides a decent interpreter (column 4): Run times grow linearly in n as expected.

Finally, we also added tabulation (cf. §3.5), where the mappings are for simplicity implemented as associative lists. Surprisingly, the speed-up (less than 15%) is modest. The reason might be the tiny class hierarchy of the example program for which lookups functions terminate quickly.

We also ran the tests with the code generated in Haskell (compiled with Glasgow Haskell Compiler 6.10.4) and OCaml (compiled to native code with OCaml 3.11.1). The Haskell code is about 60% slower than the ML and the OCaml code takes between 2 to 5 times as much time as ML. Still, the different adjustments to the interpreter affect the run times similarly to ML.

As JinjaThreads also has a verified compiler and a virtual machine, we also ran the virtual machine on the compiled code. The virtual machine is 6 to 7 times faster than the source code interpreter with red-black trees for the heap: Pushing 10,000 objects through the buffer takes 9.6 s with tabulation and 11.9 s without. Clearly, rewriting expressions in the small-step semantics is slower than pattern-matching on instructions. Still, our interpreter and VM are still far from a commercial VM: The Java HotSpot VM takes only 30 ms for 10,000 objects.

In [14], Lui and Moore test their JVM formalisation M6 in ACL2 on a simple parallel factorial algorithm. To compare our interpreter with theirs, we have converted the Java program to JinjaThreads with our Java2Jinja tool. For computing 10! with five threads in parallel, our source code semantics takes 26.7 s and the VM just 0.2 s. The M6 takes 6.2 s when run in the ACL2 interpreter, version 2.7 with GNU CLISP 2.42.

4 Guidelines for Executable Formalisations

From our experience with JinjaThreads, we have distilled the following guidelines to easily obtain executable formalisations in Isabelle.

Avoid Hilbert's ε operator! Hilbert's choice cannot express underspecification adequately as, in HOL's model, its interpretation is fully specified. Partial correctness of the code generator guarantees that all evaluations in the functional language hold in *every model*. Thus, one cannot replace it by any implementing function that chooses one suitable value consistently and fixes the underspecified function to *one concrete model*. Instead, use one of the following alternatives:

1. Change the definition to make the choice deterministic and implementable, e.g. always pick the least element.
2. Use locales for intra-logical underspecification and instantiate the choice operator to a concrete implementation by locale interpretation.
3. Switch to a relational description and prove the correctness for all values.

The first is least intrusive to the formalisation, but requires changes to the original specification. To execute the deterministic choice, one needs to run the predicate compiler on the choice property and use the executable definite descriptor for predicates (§2.4), or implement a suitable search algorithm via program refinement, as we did for memory allocation (§3.3).

The second is the most flexible, but also tedious as the locale does not automatically setup proof automation and lacks true polymorphism. We use this approach e.g. to specify schedulers §3.4. Care must be taken in combination with data refinement via the code generator, as the choice must not depend on the additional structure that the interpretation introduces.

The last option completely avoids underspecification, but relinquishes the functional implementation. For code generation, one should either (i) apply the predicate compiler to obtain code that computes *all* possible implementations for the specification, or (ii) provide a functional implementation and show correctness (cf. §3.4). For this, one must typically replace the involved types with others that have additional structure.

Structure locales wisely! Modular specifications, i.e. locales, and code generation do not (yet) go well together (cf. §2.2). To combine them, one best adheres to the following discipline: One locale *Sig* fixes the parameters' signatures and contains all definitions that depend on the parameters. Another locale *Spec* extends *Sig* and states the assumptions about the parameters; all proofs that depend on the properties go into *Spec*. For functions and inductive predicates of *Sig*, one feeds the equational theorems and introduction rules exported into the theory context to the code generator or predicate compiler, resp. To obtain the (correctness) theorems, instantiate *Spec* and prove the assumptions.

Annotate predicates with modes! Mode annotations for predicates instruct the predicate compiler to generate only modes of interest, not all modes that its mode analysis can infer. They provide three benefits. First, if the predicate has many parameters, analysing all modes can quickly become computationally

intractable (cf. §3.3) – in this case, they are necessary. Second, they ease maintenance and debugging as they fail immediately after adjustments: If changes in the development disable a mode of interest, an error message indicates which clauses are to blame. Without annotations, the missing mode might remain undiscovered until much later, which then complicates correcting errors. Third, some not annotated, but inferable modes might lead to generation of slow or non-terminating functions. By disallowing them, the predicate compiler cannot accidentally pick one of them when it compiles a subsequent predicate.

5 Conclusion and Future Work

Originally, the JinjaThreads formalisation aimed to investigate semantics properties of concurrent Java; executability was of little concern throughout its development. At the start, subtleties in the formalisation inhibited executing the specifications. After we had substantially improved the code generation of inductive predicates and manually adapted and extended the formalisation, we obtained a Java interpreter with decent performance. We found solutions on how to marry code generation with locales and how to adequately handle underspecification and the definite description operator. From our experience, we extracted guidelines on how to develop future formalisations with executability in mind.

JinjaThreads' predecessor Jinja [10] has been developed eight years ago. Comparing the efforts and results to obtain executability, we note the following improvements: First, Jinja's code generator setup relied on manual and unsound translations, e.g. sets as raw lists and ad hoc implementations for Hilbert's ε operator. In contrast, we adapted the formalisation such that the unsound translations are no longer necessary. Instead, we use safe implementations for sets from Isabelle's library and model underspecification explicitly inside the logic. Second, the Jinja interpreter can loop infinitely when it executes ill-formed programs, but Jinja lacks a well-formedness checker. Employing our new implementations (cf. §2.3), JinjaThreads now offers a decision procedure for checking well-formedness. Third, the (now outdated) predicate compiler, which Jinja uses, generates code directly in the functional target language. Thus, interweaving purely functional and logical computations as, e.g. in the JinjaThreads scheduler would have been impossible within the logic, but required editing the generated code. Exploiting program and data refinement, we obtained a sound and executable definite description operator (§2.4) to link both worlds.

Thanks to these increased efforts, we reach a new level of confidence in the generated code, which would have been impossible with the tools eight years ago. Still, this extensive case study revealed some pressing issues for code generation:

To execute JinjaThreads' virtual machine specification we employ implementations for common set operations. The necessary refinement is conceptionally straightforward, but requires a tremendous effort if done manually. This step should be automated.

The lack of integration between locales and code generation requires all users to follow a rather strict discipline (cf. §2.2). A solution on Isabelle's side that integrates locales and code generation needs to be addressed in the future.

Acknowledgements We thank F. Haftmann for invaluable help with the code generator, S. Berghofer for improving the RTC, J. Thedering and A. Zea for their work on Java2Jinja, and J. Blanchette, A. Popescu, M. Hecker, D. Lohner, and the anonymous reviewers for comments on earlier drafts. We acknowledge funding from DFG grants Sn11/10-1,2 and DFG doctorate program 1480 (PUMA).

References

1. Atkey, R.: CoqJVM: An executable specification of the Java virtual machine using dependent types. In: TYPES'08. LNCS, vol. 4941. Springer 18–32 (2008)
2. Ballarin, C.: Interpretation of locales in Isabelle: Theories and proof contexts. In: MKM'06. LNAI, vol. 4108, pp. 31–43. Springer (2006)
3. Barthe, G., Crégut, P., Grégoire, B., Jensen, T., Pichardie, D.: The MOBIUS proof carrying code infrastructure. In: FMCO'08. LNCS, vol. 5382. Springer 1–24 (2008)
4. Bauer, G., Nipkow, T.: Flyspeck I: Tame graphs. In Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. `http://afp.sourceforge.net/entries/Flyspeck-Tame.shtml` (2006) Formal proof development.
5. Berghofer, S., Bulwahn, L., Haftmann, F.: Turning inductive into equational specifications. In: TPHOLs'09. LNCS, vol. 5674, pp. 131–146. Springer (2009)
6. Bulwahn, L., Krauss, A., Haftmann, F., Erkök, L., Matthews, J.: Imperative functional programming with Isabelle/HOL. In: TPHOLs'08. LNCS, vol. 5170, pp. 134–149. Springer (2008)
7. Farzan, A., Meseguer, J., Roşu, G.: Formal JVM code analysis in JavaFAN. In: AMAST'04. LNCS, vol. 3116, pp. 147–150. Springer (2004)
8. Haftmann, F.: Data refinement (raffinement) in Isabelle/HOL This is a draft of an envisaged publication still to be elaborated which, applying the usual rules of academic confidentiality, can be inspected at `http://www4.in.tum.de/~haftmann/pdf/data_refinement_haftmann.pdf`.
9. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: FLOPS'10. LNCS, vol. 6009, pp. 103–117. Springer (2010)
10. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Sys. 28, 619–695 (2006)
11. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: ITP'10. LNCS, vol. 6172, pp. 339–354. Springer (2010)
12. Leroy, X.: A formally verified compiler back-end. J. Autom. Reasoning 43(4), 363–446 (2009)
13. Letouzey, P.: Extraction in Coq: An overview. In: Logic and Theory of Algorithms. LNCS, vol. 5028. Springer 359–369 (2008)
14. Liu, H., Moore, J.S.: Executable JVM model for analytical reasoning: A study. In: IVME'03, pp. 15–23. ACM (2003)
15. Lochbihler, A.: Type safe nondeterminism – a formal semantics of Java threads. In: Workshop on Foundations of Object-Oriented Languages (FOOL'08). (2008)
16. Lochbihler, A.: Verifying a compiler for Java threads. In: ESOP'10. LNCS, vol. 6012, pp. 427–447. Springer (2010)
17. Lochbihler, A.: Jinja with threads. In Klein, G., Nipkow, T., Paulson, L. (eds.) The Archive of Formal Proofs. `http://afp.sourceforge.net/entries/JinjaThreads.shtml` (2007) Formal proof development.
18. Lochbihler, A.: Formalising FinFuns – generating code for functions as data from Isabelle/HOL. In: TPHOLs'09. LNCS, vol. 5674, pp. 310–326. Springer (2009)
19. Marić, F.: Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. Theor. Comput. Sci. 411(50), 4333–4356 (2010)
20. Thiemann, R., Sternagel, C.: Certification of termination proofs using CeTA. In: TPHOLs'09. LNCS, vol. 5674, pp. 452–468. Springer (2009)