CoqJVM: An Executable Specification of the Java Virtual Machine using Dependent Types

Robert Atkey

LFCS, School of Informatics, University of Edinburgh
Mayfield Rd, Edinburgh EH9 3JZ, UK
bob.atkey@ed.ac.uk

Abstract. We describe an executable specification of the Java Virtual Machine (JVM) within the Coq proof assistant. The principal features of the development are that it is executable, meaning that it can be tested against a real JVM to gain confidence in the correctness of the specification; and that it has been written with heavy use of dependent types, this is both to structure the model in a useful way, and to constrain the model to prevent spurious partiality. We describe the structure of the formalisation and the way in which we have used dependent types.

1 Introduction

Large scale formalisations of programming languages and systems in mechanised theorem provers have recently become popular [4–6, 9]. In this paper, we describe a formalisation of the Java virtual machine (JVM) [8] in the Coq proof assistant [11]. The principal features of this formalisation are that it is executable, meaning that a purely functional JVM can be extracted from the Coq development and – with some O'Caml glue code – executed on real Java bytecode output from the Java compiler; and that it is structured using dependent types.

The motivation for this development is to act as a basis for certified consumer-side Proof-Carrying Code (PCC) [12]. We aim to prove the soundness of program logics and correctness of proof checkers against the model, and extract the proof checkers to produce certified stand-alone tools.

For this application, the model should faithfully model a realistic JVM. For the intended application of PCC, this is essential in order to minimise and understand the unavoidable semantic gap between the model and reality. PCC is intended as a secure defence against hostile code; the semantic gap is the point that could potentially be exploited by an attacker. To establish and test of our model we have designed it to be executable so that it can be tested against a real JVM. Further, we have structured the design of the model using Coq's module system, keeping the component parts abstract with respect to proofs about the model. This is intended to broaden the applicability of proofs performed against the model and to prevent "over-fitting" to the specific implementation.

In order to structure the model we have made heavy use of Coq's feature of dependent types to state and maintain invariants about the internal data

structures. We have used dependent types as a local structuring mechanism to state the properties of functions that operate on data structures and to pass information between them. In some cases this is forced upon us in order to prove to Coq that our recursive definitions for class loading and searching the class hierarchy are always terminating, but they allow us to tightly constrain the behaviour of the model, reducing spurious partiality that would arise in a more loosely typed implementation.

To demonstrate the removal of spurious partiality, we consider the implementation of the `invokevirtual` function. To execute this function, we must resolve the method reference within the instruction; find the object in the heap; search for the method implementation starting from the object's class in the class pool and then, if found invoke this method. In a naive executable implementation, we would have to deal with the potential failure of some of these operations. For example, the finding of the class for a given object in the heap. We know from the way the JVM is constructed that this can never happen, but we still have to do something in the implementation, even if it just returns an error. Further, every proof that is conducted against this implementation must prove over again that this case cannot happen. We remove this spurious partiality from the model by making use of dependent types to maintain invariants about the state of the JVM. These invariants are then available to all proofs concerning the model. Our belief is that this will make large-scale proofs using the model easier to perform, and we have some initial evidence that this is the case, but detailed research of this claim is still required.

There are still cases when the model should return an error. We have attempted to restrict these to when the code being executed is not type safe. A basic design decision is to construct the model so that if the JVM's bytecode verifier would accept the code, then the model should not go wrong.

Overview In the next section we give an overview of our formalisation, detailing the high-level approach and the module design that we have adopted. We describe the modelling of the class pool and its operations in Section 3. In Section 4 we describe the formalisation of object heaps and static fields, again using dependent types. In Section 5 we describe our modelling of the execution of single instructions. The extraction to O'Caml is discussed in Section 6. We discuss related work in Section 7 and conclude with notes for future work in Section 8.

The Formalisation Due to reasons of space, this paper can only offer a high-level overview of the main points of the formalisation. For more information the reader is referred to the formalisation itself, which is downloadable from `http://homepages.inf.ed.ac.uk/ratkey/coqjvm/`.

2 High-level Structure of the Formalisation

The large-scale structure of the formalisation is organised using Coq's module facilities. We use this for two reasons: to abstract the model over the implementation of several basic types such as 32-bit integers and primitive class, method

and field names; and also to provide an extra-logical reason to believe that we are modelling a platonic JVM, rather than fitting to our implementation.

The interface for the basic types is contained within the signature BASICS. We assume a type Int32.t with some arithmetic operations. This is instantiated with O'Caml's `int32` type after extraction. We also require types to represent the class, field and method names. Since these types are used as the domains of finite maps throughout the formalisation, we stipulate that they must also have an ordering suitable for use with Coq's implementation of balanced binary trees.

We keep the constituent parts of the formalisation abstract from each other by use of the module system. This has the advantage of reducing complexity in each part and keeping the development manageable. It is also an attempt to keep proofs about the model from "over-fitting" to the concrete implementation used. For example, we only expose an abstract datatype for the set of loaded classes and some axiomatised operations on it. The intention is that any implementation of class pools will conform to this specification, and so proofs against it will have wider applicability than just the implementation we have coded[1]. Thirdly, the use of modules makes the extracted code safer to use from O'Caml. Many of the datatypes we use in the formalisation have been refined by invariants expressed using dependent types. Since O'Caml does not have dependent types they are thrown away during the extraction process. By using type abstraction we can be sure that we are maintaining the invariants correctly.

The main module is Execution, which has the following functor signature:

$$
\begin{aligned}
&\textbf{Module}\ \mathsf{Execution}\ (B : \mathsf{BASICS})\\
&\qquad (C : \mathsf{CLASSDATATYPES}\ \textbf{with Module}\ B := B)\\
&\qquad (CP : \mathsf{CLASSPOOL}\ \textbf{with Module}\ B := B\\
&\qquad\qquad \textbf{with Module}\ C := C)\\
&\qquad (RDT : \mathsf{CERTRUNTIMETYPES}\ \textbf{with Module}\ B := B\\
&\qquad\qquad \textbf{with Module}\ C := C\\
&\qquad\qquad \textbf{with Module}\ CP := CP)
\end{aligned}
$$

The module types mentioned in the signature correspond to the major components of our formalisation. The signature BASICS we have already mentioned. CLASSDATATYPES contains the basic datatypes for classes, methods, instructions and the like. CLASSPOOL is the interface to the set of loaded classes and the dynamic loading facilities, described in Section 3. CERTRUNTIMETYPES is the interface to the object heap and the static fields, described in Section 4.

Note the use of sharing constraints in the functor signature. These are required since the components that fulfil each signature are themselves functors. We need sharing constraints to state that all their functor arguments must be equal. The heavy use of sharing constraints exposed two bugs in Coq's implementation, one to do with type checking code using sharing and one in extraction.

[1] This technique is also used in Bicolano: `http://mobius.inria.fr/bicolano`.

3 Class Pools and Dynamic Loading

Throughout execution the JVM maintains a collection of classes that have been loaded into memory: the class pool. In this section we describe how we have modelled the class pool; how new classes are loaded; how functions that search the class pool are written; and how the class pool is used by the rest of the formalisation.

3.1 The Class Pool

Essentially, the class pool is nothing but a finite map from fully qualified class names to data structures containing information such as the class's superclass name and its methods. We store both proper classes and interfaces in the same structures, and we differentiate between them by a boolean field classInterface, which is true when the structure represents an interface, and false otherwise. The map from class names to class structures directly mimics the data structure that would be used in a real implementation of a JVM. In order to construct an executable model within Coq though, the basic data structure is not enough; we have to refine the basic underlying data structure with some invariants.

The motivation for adding invariants to the class pool data structure was originally to enable the writing of functions that search over the class hierarchy. Each class data structure is a record type, with a field classSuperClass : option className indicating the name of that class's superclass, if any. Searches over the class hierarchy operate by following the superclass links between classes. In a real JVM implementation it is known that, due to the invariants maintained by the class loading procedures, if a class points to a superclass, then that superclass will exist, and that `java.lang.Object` is the top of the hierarchy. Therefore, every search upwards through the inheritance tree will always terminate.

When writing these functions in Coq we must convince Coq that the function actually does terminate, i.e. we must have a proof that the class hierarchy is well founded so that we can write functions that recurse on this fact. To this end, we package the basic data structure for the class pool, a finite map from class names to class structures, with two invariants:

$$
\begin{array}{ll}
\mathit{classpool} & : \mathsf{Classpool.t} \\
\mathit{classpoolInvariant} & : \forall \mathit{nm}\ c.\, \mathsf{lookup}\ \mathit{classpool}\ \mathit{nm} = \mathsf{Some}\ c \rightarrow \\
 & \qquad\qquad \mathsf{classInvariant}\ \mathit{classpool}\ \mathit{nm}\ c \\
\mathit{classpoolObject} & : \exists c.\, \mathsf{lookup}\ \mathit{classpool}\ \texttt{java.lang.Object} = \mathsf{Some}\ c \\
 & \quad \wedge\ \mathsf{classSuperClass}\ c = \mathsf{None} \\
 & \quad \wedge\ \mathsf{classInterfaces}\ c = [\,] \wedge \mathsf{classInterface}\ c = \mathsf{false}
\end{array}
$$

We call the type of these records certClasspool. The type Classpool.t represents the underlying finite map. The function lookup looks up a class name in a given class pool, returning an option class result. There are two invariants that we maintain on class pools, the first covers every class in *classpool*, we describe this

below. The second states that there is an entry for `java.lang.Object` and that it has no superclass and no superinterfaces, and is a proper class.

The invariants that every class must satisfy are given by the predicate

$$\begin{aligned}&\mathsf{classInvariant} : \mathsf{Classpool.t} \to \mathsf{className} \to \mathsf{class} \to \mathsf{Prop}\\&\mathsf{classInvariant}\ \mathit{classpool}\ \mathit{nm}\ \mathit{c} \equiv\\&\quad \mathsf{className}\ \mathit{c} = \mathit{nm}\\&\quad \wedge\ (\mathsf{classSuperClass}\ \mathit{c} = \mathsf{None} \to \mathit{nm} = \texttt{java.lang.Object})\\&\quad \wedge\ \mathsf{classInvariantAux}\ \mathit{classpool}\ \mathit{c}\end{aligned}$$

which states that each class structure must be recorded under a matching name; only `java.lang.Object` has no superclass; and that further invariants hold: classInvariantAux *classpool* *c*. This varies depending on whether *c* is a proper class or an interface. In the case of a proper class, we require that two properties hold: that all the class's superclasses are present in *classpool* and likewise for all its superinterfaces. A proof that all a class's superclasses are present is recorded as a term of the following inductive predicate:

$$\begin{aligned}&\mathsf{goodSuperClass}\ \mathit{classpool} : \mathsf{option\ className} \to \mathsf{Prop}\\&\quad \mathsf{gscTop} : \mathsf{goodSuperClass}\ \mathit{classpool}\ \mathsf{None}\\&\quad \mathsf{gscStep} : \forall \mathit{cSuper}\ \mathit{nmSuper}.\\&\qquad \mathsf{lookup}\ \mathit{classpool}\ \mathit{nmSuper} = \mathsf{Some}\ \mathit{cSuper} \to\\&\qquad \mathsf{classInterface}\ \mathit{cSuper} = \mathsf{false} \to\\&\qquad \mathsf{goodSuperClass}\ \mathit{classpool}\ (\mathsf{classSuperClass}\ \mathit{cSuper}) \to\\&\qquad \mathsf{goodSuperClass}\ \mathit{classpool}\ (\mathsf{Some}\ \mathit{nmSuper})\end{aligned}$$

For a class record *c*, knowing goodSuperClass *classpool* (classSuperClass *c*), means that we know that all the superclasses of *c*, if there are any, are contained within *classpool*, finishing with a class that has no superclass. We also know that all the class structures in this chain are proper classes. By the other invariants of class pools, we know that the top class must be `java.lang.Object`. There is also a similar predicate goodInterfaceList *classes interfaceList* that states that the tree of interfaces starting from the given list is well founded.

For classInvariantAux *classpool* *c*, when *c* is an interface we again require that all the superinterfaces are present, but we also insist that the superclass must be `java.lang.Object`, to match the requirements in the JVM specification.

These predicates may be used to write functions that search the class hierarchy that are accepted by Coq by the technique of recursion on an ad-hoc predicate [3]. Unfortunately, they are not suitable for proving properties of these functions. We describe in Section 3.3 how we use an equivalent formulation to prove properties of these functions.

3.2 Dynamic Class Loading

New classes are loaded into the JVM as a result of the resolution of references to entities such as classes, methods and fields. For example, if the code for a method contains an `invokevirtual` instruction that calls a method `int C.m(int)`, then

the class name `C` must be resolved to a real class, and then the method `int C.m(int)` must be resolved to an actual method in the class resolved by `C`, or one of its superclasses. The process of resolving the reference `C` may involve searching for an implementation on disk and loading it into the class pool.

The JVM specification distinguishes between the loading and resolution of classes. It also takes care to maintain information on the different class loaders used to load classes, in order to prevent spoofing attacks on the JVM's type safety [7]. In our formalisation we only consider a single class loader, the bootstrap class loader. With this simplification, we can unfold the resolution and loading procedures described in the JVM specification to the following steps:

1. If the class `C` exists in the class pool as c then return c.
2. Otherwise, search for an implementation of `C`. If not found, return error. If found, check it for well-formedness and call it pc (preclass).
3. Recursively resolve the reference to the superclass, returning sc.
4. Check that sc is a proper class, and that `C` is not the name of sc or any of its superclasses.
5. Recursively resolve the references to superinterfaces.
6. Convert pc to a class c and add it to the class pool. Return c.

The well-formedness check in step 2 checks that the class we have found has a superclass reference and that it has a matching name to the one we are looking for. In the case of interfaces, it checks that the superclass is always `java.lang.Object`. Since the formalisation only deals with a structured version of the `.class` data loaded from disk, we do not formalise any of the low-level binary well-formedness checks prescribed by the JVM specification. We separate class implementations (preclasses pc) from loaded classes c because preclasses contain information that is discarded by the loading procedure, such as the proof representations required for PCC.

If we start from a class pool that contains nothing but an implementation of `java.lang.Object` satisfying the properties in the previous section, and add classes by the above procedure then it is evident that we maintain the invariants.

We now describe how we have formalised this procedure as a Coq function. The first decision to be made is how to represent the implementations of classes "on disk". Following Liu and Moore's ACL2 model [9], we do this by modelling them as a finite map from class names to preclasses. The intention is that this represents a file system mapping pathnames to files containing implementations. We use the O'Caml glue code described in Section 6 to load the actual files from disk before any execution, parse them and put them in this structure.

With this, the implementation of the procedure above can look into this collection of preclasses in order to find class implementations and check them for well-formedness. However, a problem arises due to the recursive nature of the loading and resolution procedure. We must be able to prove that the resolution of a single class reference will always terminate; we must not spend forever trying to resolve an infinite chain of superclasses or interfaces. To solve this problem we define a predicate wfRemovals *preclasses* that states that *preclasses* may

be reduced to empty by removing elements one by one. With some work, one can prove $\forall preclasses.$ wfRemovals $preclasses$. Using this, we define a function loadAndResolveAux which has type

$$\begin{aligned}\mathsf{loadAndResolveAux}\ \ &(target : \mathsf{className})\ (preclasses : \mathsf{Preclasspool.t})\\ &(PI : \mathsf{wfRemovals}\ preclasses)\ (classes : \mathsf{certClasspool})\\ &: \{\mathsf{LOAD}\ classes, preclasses \Rightarrow classes' \ \&\ c : \mathsf{class}\\ &\qquad\qquad \mid \mathsf{lookup}\ classes'\ target = \mathsf{Some}\ c\}.\end{aligned}$$

This function takes the *target* class name to resolve, a *preclasses* to search for implementations, a PI argument stating that *preclasses* can be reduced to empty by removing elements, and the current class pool *classes*. The return type is an instance of the following type with two constructors:

$$\begin{aligned}&\mathsf{loadType}\ (A : \mathsf{Set})\ (P : \mathsf{certClasspool} \to A \to \mathsf{Prop})\ (classes : \mathsf{certClasspool})\\ &\qquad\qquad (preclasses : \mathsf{Preclasspool.t}) : \mathsf{Set}\\ &\quad \mathsf{loadOk} : \forall classes'\ a.\ \mathsf{preserveOldClasses}\ classes\ classes' \to\\ &\qquad \mathsf{onlyAddFromPreclasses}\ classes\ classes'\ preclasses \to\\ &\qquad P\ classes'\ a \to\\ &\qquad \mathsf{loadType}\ A\ P\ classes\ preclasses\\ &\quad \mathsf{loadFail} : \forall classes'.\ \mathsf{preserveOldClasses}\ classes\ classes' \to\\ &\qquad \mathsf{onlyAddFromPreclasses}\ classes\ classes'\ preclasses \to\\ &\qquad \mathsf{exn} \to\\ &\qquad \mathsf{loadType}\ A\ P\ classes\ preclasses\end{aligned}$$

and $\{\mathsf{LOAD}\ classes, preclasses \Rightarrow classes' \ \&\ a : A \mid Q\}$ is notation for

$$\mathsf{loadType}\ A\ (\lambda classes'\ a.\ Q)\ classes\ preclasses.$$

Hence, in the type of loadAndResolveAux, there are two possibilities: either a class structure is returned, along with a proof that this class is in the new class pool *classes′*. Or a new classpool *classes′* is returned, along with an error of type exn. These errors represent exceptions like `java.lang.ClassFormatError` that are turned into real Java exceptions by the code executing individual instructions.

The two common parts of the constructors, the predicates preserveOldClasses and onlyAddFromPreclasses relate the new class pool *classes′* to the old class pool *classes* and to *preclasses*. The predicate preserveOldClasses *classes classes′* states that any classes that were in *classes* must also be in *classes′*. The predicate onlyAddFromPreclasses *classes classes′ preclasses* states that any new classes in *classes′* that are not in *classes* must have been loaded from *preclasses*. These two properties are used to establish properties of the class pool as it evolves during the execution of the virtual machine. In particular, we use them to show that the invariants of previously loaded classes are not violated by loading new classes, and to allow the inheritance of known facts about *preclasses* to the class pool. This is intended to be used in consumer-side PCC for pre-checking the proofs in a collection of class implementations before execution begins.

We do not have space to go into the implementation of loadAndResolveAux. We have written the function in a heavily dependently typed style, making use of a "proof passing style". We describe this style in the next section.

3.3 Writing and Proving Functions that Search the Class Pool

Given a representation of class pools and functions that add classes to it, we also need functions that query the class pool. To execute JVM instructions, we need ways to determine when one class inherits from another; to search for virtual methods and to search for fields and methods during resolution.

The basic structure of these operations is to start at some point in the class hierarchy and then follow the superclass and superinterface links upwards until we find what we are looking for. We introduced goodSuperClass as a way to show that the hierarchy is well founded. While this definition is suitable for defining recursive functions over the superclass hierarchy, it is not suitable for proving properties of such functions. In the induction principle on goodSuperClass generated for a property P by Coq's **Scheme** keyword, the inductive step has P (classSuperClass *cSuper*) g as a hypothesis where g has type goodSuperClass *classes* (classSuperClass *cSuper*). The variable *cSuper* denotes the implementation of the superclass of the current class that goodSuperClass guarantees the existence of. However, during the execution of a recursive function over the class hierarchy we will have looked up the same class, but under a different Coq name *cSuper′*. We know from the other invariants maintained within certClasspool that *cSuper* and *cSuper′* are equal, because they are stored under the same class name, but our attempts at rewriting the hypotheses with this fact were defeated by type equality issues.

Although it may be possible to find a way to rewrite the hypotheses in such a way that allows us to apply the induction hypothesis, we took an easier route and defined an equivalent predicate, superClassChain:

```
superClassChain : certClasspool → option className → Prop :=
  sccTop : ∀classes. superClassChain classes None
  sccStep : ∀classes cSuper nmSuper.
    lookup classes nmSuper = Some cSuper →
    classInterface cSuper = false →
    (∀cSuper′. lookup classes nmSuper = Some cSuper′ →
               superClassChain classes (classSuperClass cSuper′)) →
    superClassChain classes (Some nmSuper).
```

The difference here is that the step to the next class in the chain is abstracted over the implementation of that class, removing the problem described above. We retain the original goodSuperClass predicate because it is easier to prove while adding classes to the classpool. These two are easily proved equivalent.

We can now define functions that are structurally recursive on the superclass hierarchy by recursing on the structure of the superClassChain predicate. To define such functions we must prove two inversion lemmas. These have the types

```
inv1 : ∀classes nm c optNm.
       optNm = Some nm → superClassChain classes optNm →
       ¬(lookup (classpool classes) nm = None)
```

```
fix search (classes : certClasspool)
           (supernameOpt : option className)
           (scc : superClassChain classes supernameOpt) :=
  match optionInformative supernameOpt with
  | inleft (exist supername snmEq) ⇒
      let notNotThere := inv1 snmEq scc in
      match lookupInformative classes supername with
      | inleft(exist superC superCExists) ⇒
          let scc' := inv2 superCExists snmEq scc in
          (* examine superC here, use
           * search classes (classSuperClass superC) scc' for
           * recursive calls *)
      | inright notThere ⇒
          match notNotThere notThere with end
      end
  | inright _ ⇒ (* search failed code here *)
  end
```

Fig. 1. Skeleton search function

$$\begin{aligned}\mathsf{inv2} : \forall &\mathit{classes}\ \mathit{nm}\ c\ \mathit{optNm}.\mathit{optNm} = \mathsf{Some}\ \mathit{nm} \rightarrow \\ &\mathsf{superClassChain}\ \mathit{classes}\ \mathit{optNm} \rightarrow \mathsf{lookup}\ (\mathsf{classpool}\ \mathit{classes})\ \mathit{nm} = c \rightarrow \\ &\mathsf{superClassChain}\ \mathit{classes}\ (\mathsf{classSuperClass}\ c).\end{aligned}$$

A skeleton search function is shown in Figure 1. In addition to inv1 and inv2, we use functions optionInformative : $\forall A\ o.\ \{a : A \mid o = \mathsf{Some}\ a\} + \{o = \mathsf{None}\}$ and lookupInformative : $\forall \mathit{classes}\ \mathit{nm}.\ \{c \mid \mathsf{lookup}\ \mathit{classes}\ \mathit{nm} = \mathsf{Some}\ c\} + \{\mathsf{lookup}\ \mathit{classes}\ \mathit{nm} = \mathsf{None}\}$.

The search function operates by first determining whether there is a superclass to be looked up. If so, optionInformative returns a proof object for $\mathit{supernameOpt} = \mathsf{Some}\ \mathit{supername}$. This is passed to inv1 to obtain a proof that *supername* does not *not* exist in *classes*. The function then must look up *supername* for itself: since Coq does not allow the construction of members of Set by the examination of members of Prop, a proof can only tell a function that its work will not be fruitless, not do its work for it. To dispose of the impossible case when the superclass is discovered not to exist, we combine *notNotThere* and *notThere* to get a proof of False which is eliminated with an empty **match** (recall that $\neg A$ is represented as $A \rightarrow \mathsf{False}$ in Coq). Otherwise, we use inv2 to establish superClassChain for the rest of the hierarchy, and proceed upwards.

3.4 Interface to the rest of the Formalisation

The implementation of the type certClasspool is kept hidden from the rest of the formalisation. To state facts about a class pool, external clients must make do with a predicate classLoaded : certClasspool → className → class → Prop.

To know classLoaded *classes nm c* is to know that *c* has been loaded under the name *nm* in *classes*. The resolution procedures for classes, methods and fields are also exposed, using the loadType type described above. Each of these functions returns witnesses attesting to the fact that, when they return a class, method or field, that entity exists and matches the specification requested.

This module also provides two other services: assignability (or subtype) checking, and virtual method lookup. These both work by scanning the class pool using the technique described in the previous subsection.

4 The Object Heap and Static Fields

The two other major data structures maintained by the JVM are the object heap and the static fields. In this section we describe their concrete implementations and the dependently typed interface they expose to the rest of the formalisation.

4.1 Object Heaps

As with the class pool, the object heap is essentially nothing but a finite map, this time from object references to structures representing objects. Object references are represented as natural numbers using Coq's positive type. As above, we apply extra invariants to this basic data structure to constrain it to more closely conform to object heaps that actually arise during JVM execution.

We build object heaps in two stages. First, we take a standard finite map data structure and re-package it as a *heap*. Heaps are an abstract datatype with the following operations, abstracted over a type obj of entities stored in the heap. We have operations lookup : heap → addr → option obj to look up items in the heap; update : heap → addr → obj → option t to update the heap, but only of existing entries; and new : heap → obj → heap × addr to create new entries.

Given a heap datatype with these operations and the obvious axioms, we build *object heaps* tailored to the needs of the JVM. The first thing to fix is the representation of objects themselves. We regard objects as pairs of a class name and a finite map from fields to values. As with class pools we require several invariants to hold about the representation of each object. Each of the class names mentioned in an object heap actually exist in some class pool. Thus, the type of an object heap depends on some class pool: certHeap *classes*. Second, we require that each of the fields in each object is well-typed. We are helped here by the fact that JVM field descriptors contain their type.

The type of the operation that looks up a field in an object is

heapLookupField *classes* (*heap* : certHeap *classes*) (*a* : addr)
(*fldCls* : className) (*fldNm* : fieldName) (*fldTy* : javaType)
: {*v* | objectFieldValue *classes heap a fldCls fldNm fldTy v*}
+ {¬objectExists *classes heap a*}.

This operation looks up an object at address *a*, and a field within that object. If the object exists then either the actual value of that field or a default value for

the field (based on *fldTy*) is returned, along with a proof that this is the value of that field in the object at a in *heap*. Otherwise, a proof that the object does not exist is returned. The predicates objectFieldValue and objectExists record facts about the heap that can be reasoned about by external clients, in a similar way to the classLoaded predicate for class pools.

A possible invariant that we do not maintain is that each field present in an object is actually present in that object's class and vice versa. We choose not maintain this invariant because it simplifies development at this stage of the construction of the model. As described above and as we will elaborate in Section 5, we use dependent types to reduce spurious partiality in the model and to make the model more usable for proving properties. At the moment, it is useful to know that a field's value is well-typed; when proving a property of the model that relies on type safety we do not have to keep around an additional invariant stating that all fields are well-typed. Also, it is useful to know that an object's class exists so that the implementation of the `invokevirtual` instruction can use this information to find the class implementation and search for methods. If the class does not exist there is no sensible action to take other than to just go wrong, which introduces spurious partiality into the model. However, there is an obvious action to take if a field does not exist – return a default value.

The interface that object heaps present to the rest of the formalisation is constructed in the same style as that for class pools. We present an abstract type certHeap *classes*, along with operations such as heapLookupField above. Operations heapUpdateField and heapNew update fields and create new objects respectively. All these operations are dependently typed so that they can be used in a proof-passing style within the implementations of the bytecode instructions.

Since the type of object heaps depends on the current class pool to state its invariants, we have to update the invariants' proofs when new classes are added to the class pool. We use the preserveOldClasses predicate from Section 3.2:

$$\begin{aligned}\mathsf{preserveCertHeap} : \forall\, &\mathit{classesA}\ \mathit{classesB}.\ \mathsf{certHeap}\ \mathit{classesA} \rightarrow \\ &\mathsf{preserveOldClasses}\ \mathit{classesA}\ \mathit{classesB} \rightarrow \\ &\mathsf{certHeap}\ \mathit{classesB}\end{aligned}$$

Since every operation that alters the class pool produces a proof object of type preserveOldClasses, this can be passed into the above function to produce a matching object heap.

4.2 Static Fields

The static fields arc modelled in exactly the same way as the fields of a single object in the heap. The rest of the model is presented with a dependently typed interface that maintains the invariant that each field's value is well-typed according to the field's type. The type of well-typed static field stores is fieldStore *classes heap*.

5 Modelling Execution

All of the modules described above are arguments to the Execution functor, whose signature was in Section 2. We now describe the implementation of this module.

5.1 The Virtual Machine State

The state of the virtual machine is modelled as a record with the fields

stateFrameStack : list frame
stateClasses : certClasspool
stateObjectHeap : certHeap *stateClasses*
stateStaticFields : fieldStore *stateClasses* *stateObjectHeap*.

States contain the three major data structures for the class pool, object and static fields that we have covered above. The additional field records the current frame stack of the virtual machine. Individual stack frames have the fields:

frameOpStack : list rtVal *frameLVars* : list (option rtVal)
framePC : nat *frameCode* : code
frameClass : class

The type rtVal is used to represent run-time values manipulated by the virtual machine such as integers and references. There are entries for the current operand stack and the local variables. The use of option types in the local variables is due to the presence of values that occupy multiple words of memory. Values of types such as `int` only occupy a single 32-bit word of memory on the real hardware, but `long` and `double` values occupy 64-bits. When stored in the local variables, the second half of a 64-bit value is represented using None. The rest of the fields in frame are as follows: the *framePC* field records the current program counter; *frameCode* records the code being executed, this consists of a list of instructions and the exception handler tables; *frameClass* is the class structure for the code being executed, this is used to look up items in the class's constant pool.

5.2 Instruction Execution

The main function of the formalisation is exec : state → execResult. This executes a single bytecode instruction within the machine. If any exceptions are thrown then the catching and handling or the termination of the machine are all handled before exec returns. The type execResult sets out the possible results of a single step of execution:

```
execResult : Set
  cont : state → execResult
  stop : state → option rtVal → execResult
  stopExn : state → addr → execResult
  wrong : execResult.
```

An execution step can either request continuation to the next step; stop, possibly with a value; stop with a reference to an uncaught exception; or go wrong.

The basic operation of `exec` is simple. The current instruction is found by unpacking the current stack frame from the state and looking up by *framePC*. Each instruction is implemented by a different function. The non-object-oriented implementations are relatively straightforward; the object-oriented instructions more complex. They interact with the class pool, object heap and static field data structures described in earlier sections. The dependently typed interfaces are used to ensure that we maintain the invariants of each data structure, and that we only go wrong when the executed code is not type safe.

6 Extraction to O'Caml

The Coq development consists of roughly 5275 lines of specification and 2288 lines of proof, as measured by the `coqwc` tool. The proof component primarily comprises glue lemmas to allow the coding of proof-passing dependently typed functions. After extraction this becomes 16454 lines of O'Caml code, with a `.mli` file of 17132 lines. The expansion can be explained partially by the inclusion of some elements of Coq's standard library, but mainly by the repetition of module interfaces several times. This appears to be due to an internal limitation of Coq in the way it represents module signatures with sharing constraints.

To turn this extracted code into a simulator for the JVM we have written around 700 lines of O'Caml glue code. The bulk of this is code to translate from the representation of `.class` files used by the library that loads and parses them to the representation required by the extracted Coq code. The action of the O'Caml code is simply to generate a suitable *preclasses* by scanning the classpath for `.class` files, construct an initial state for a nominated static method, and then iterate the `exec` function until the machine halts.

The JVM so produced is capable of running bytecode produced by the `javac` compiler, albeit very slowly. We have not yet implemented arrays or strings, so the range of examples is limited, but we have used it to test the dynamic loading and virtual method lookup and invocation, discovering several bugs in the model.

7 Related Work

We know of two other large-scale executable specifications of the JVM constructed within theorem provers. Liu and Moore [9] describe an executable JVM model, called *M6*, implemented using ACL2's purely functional subset of Common Lisp. M6 is particularly complete: in common with the work described here it simulates dynamic class loading and interfaces, it also goes beyond our work in simulating class initialisation and instruction-level interleaving concurrency (though see comments on concurrency in the next section). Liu and Moore describe two applications of their model. They use the model to prove the invariant that if a class is loaded then all its superclasses and interfaces must also be loaded. Unlike in our model, where this invariant is built in, this proof is

an optional extra for M6. Our motivation for maintaining this invariant is to prove that searches through the class hierarchy terminate. In M6 this is proved by first establishing that there will always be a finite number of classes loaded, and so there will be a finite number of superclasses to any class; then recursion proceeds on this finite list. Note that, unlike our method, this does not guarantee that, at the point of searching through the hierarchy, all the classes will exist. Liu and Moore also directly prove some correctness properties of concurrent Java programs. Another executable JVM specification is that of Barthe *et al* [2]. This is an executable specification of JavaCard within Coq. With respect to our model, they do not need to implement dynamic class loading since all references are resolved before runtime in the JavaCard environment. They also prove the soundness of a bytecode verifier with respect to their model.

Klein and Nipkow [4] have formalised a Java-like language, Jinja, complete with virtual machine, compiler and bytecode verifier, in Isabelle/HOL. They have proved that the compilation from the high-level language to bytecode is semantics preserving. The language they consider is not exactly Java, and their model simplifies some aspects such as the lack of dynamic loading and interfaces. The formalisation is executable via extraction to SML. Another large-scale, and very complete formalisation is that of Stärk *et al* [13] in terms of abstract state machines. This formalisation is executable, but proofs about the model have not been mechanised. The Mobius project contains a formalisation of the JVM called Bicolano[2]. This formalisation uses Coq's module system to abstract away from the representation of the machine's data structures in a similar way to ours, and has been used to prove the soundness of the Mobius Base Logic. It does not model dynamic class loading.

Other large-scale formalisations of programming languages include Leroy's formalisation of a subset of C and PowerPC machine code for the purposes of a certified compiler [6]. This is a very large example of a useful program being extracted from a formal development. Another large mechanisation effort is Lee *et al*'s Twelf formalisation of an intermediate language for Standard ML [5].

8 Conclusions

We have presented a formal, executable model of a subset of the Java Virtual Machine structured using a combination of dependent types and Coq's module system. We believe that this use of dependent types as a structuring mechanism is the first application of such a strategy to a large program.

The model is incomplete at time of writing. In the immediate future we intend to add arrays and strings to the model in order to extend the range of real Java programs that may be executed. There are several extensions that require further research. Modelling I/O behaviour of JVM programs would be a useful feature. We speculate that a suitable way to do this would be to write the formalisation using a monad. The monad would be left abstract but axiomatised in Coq in

[2] `http://mobius.inria.fr/bicolano`

order to prove properties, but be implemented by actual I/O in O'Caml. Even more difficult is the implementation of concurrency. Liu and Moore's ACL2 model simulates concurrency by interleaving, but this does not capture all the possible behaviours allowed by the Java Memory Model [10]. There has been recent work on formalising the Java Memory Model in Isabelle/HOL [1], but it is difficult to see how this could be made into an executable model. A suitable approach may be to attempt to only model data-race free programs, for which the Java Memory Model guarantees the validity of the interleaving semantics.

Acknowledgement This work was funded by the ReQueST grant (EP/C537068) from the Engineering and Physical Sciences Research Council.

References

1. David Aspinall and Jaroslav Sevcík. Formalising Java's Data Race Free Guarantee. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *LNCS*, pages 22–37. Springer, 2007.
2. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 302–319. Springer-Verlag, 2001.
3. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.
4. Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.
5. Daniel K. Lee, Karl Crary, and Robert Harper. Towards a Mechanized Metatheory of Standard ML. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 173–184, New York, NY, USA, 2007. ACM Press.
6. Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 42–54, New York, NY, USA, 2006. ACM Press.
7. Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, New York, NY, USA, 1998. ACM Press.
8. Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
9. Hanbing Liu and J. Strother Moore. Executable JVM model for analytical reasoning: A study. *Sci. Comput. Program.*, 57(3):253–274, 2005.
10. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 378–391. ACM Press, 2005.
11. The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
12. George C. Necula. Proof-carrying code. In *Proceedings of POPL97*, January 1997.
13. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine – Definition, Verification, Validation*. Springer-Verlag, 2001.