Correctness of Java Card Method Lookup via Logical Relations

Ewen Denney and Thomas Jensen

Projet Lande, IRISA, Rennes Cedex 35042, France

Abstract. We formalise the Java Card bytecode optimisation from class file to CAP file format as a set of constraints between the two formats, and define and prove its correctness. Java Card bytecode is formalised as an abstract operational semantics, which can then be instantiated into the two formats. The optimisation is given as a logical relation such that the instantiated semantics are observably equal. The proof has been automated using the Coq theorem prover.

Using a high-level language for programming embedded systems may require a transformation phase in order that the compiled code fits on the device. In this paper we describe a method for formally proving the correctness of such a transformation. The method makes extensive use of types to describe the various run-time structures and relies on the notion of logical relation to relate the two representations of the code. We present the method in the setting of mapping Java onto smart cards. The Java Card language [10] is a trimmed down dialect of Java aimed at programming smart cards. As with Java, Java Card is compiled into bytecode, which is then verified and executed on a virtual machine [4], installed on a chip on the card itself. However, the memory and processor limitations of smart cards necessitate a further stage, in which the bytecode is optimised from the standard class file format of Java, to the *CAP file* format [11]. The core of this optimisation is a *tokenisation* in which names are replaced with tokens enabling a faster lookup of various entities.

We describe a semantic framework for proving the correctness of Java Card tokenisation. The basic idea is to give an abstract description of the constraints given in the official specification of the tokenisation and show that any transformation satisfying these constraints is 'correct'. This is independent of showing that there actually exists a collection of functions satisfying these constraints. This article concentrates on proving the correctness of the specification. The formal development of an algorithm is the subject of another report. The main advantage of decoupling 'correctness' into two steps is that we get a more general result: rather than proving the correctness of one particular algorithm, we are able to show that the constraints described in Sun's official specification [11] (given certain assumptions) are sufficient. We give a formalisation and correctness proof for the part concerned with dynamic method lookup. A comprehensive formalisation appears as a technical report [2].

1 The Conversion

Java source code is compiled on a class by class basis into the *class file* format. By contrast, Java Card *CAP files* correspond to packages. They are produced by the *con-*

version of a collection of class files. In the class file format, methods, fields and so on are referred to using strings. In CAP files, however, tokens are ascribed to the various entities. The idea is that if a method, say, is publically visible[1], then it is ascribed a token. If the method is only visible within its package, then it is referred to directly using an offset into the relevant data structure. Thus references are either internal or external. The conversion groups entities from different class files into the components of a CAP file. For example, all constant pools of the class files forming a package are merged into one constant pool component, and all method implementations are gathered in the same method component. One significant difference between the two formats is the way in which the method tables are arranged. In a class file, the methods item contains all the information relevant to methods defined in that class. In the CAP file, this information is shared between the class and method components. The method component contains the implementation details (*i.e.* the bytecode) for the methods defined in this package. The class component is a collection of class information structures. Each of these contains separate tables for the package and public methods, mapping method tokens to offsets into the method component. The method tables contain the information necessary for resolving any method call in that class.

The conversion is presented in [11] as a collection of constraints on the CAP file, rather than as an explicit mapping between class and CAP formats. For example, if a class inherits a method from a superclass then the conversion can choose to include the method token in the relevant method table or, instead, that the table of the superclass should be searched. There is a choice, therefore, between copying all inherited methods, or having a more compressed table. The specification does not constrain this choice. We adopt a simplified definition of the conversion, only considering classes, constant pools, and methods (with inheritance and overwriting). In particular, we ignore fields, exceptions and interfaces. The conversion also includes a number of mandatory optimisations such as the inlining of final fields, and the type-based specialisation of instructions [10, 11], which we do not treat here.

2 Overview of Formalisation

The conversion from class file to CAP format is a transformation between formats of two virtual machines. The first issue to be addressed is determining in what sense, exactly, the conversion to token format should be regarded as an equivalence. We cannot simply say that the JVM and JCVM have the same behaviour for all bytecodes, in class and CAP file format respectively, because, *a priori*, the states of the virtual machines are themselves in different formats. Instead, we adopt a simple form of equivalence based on the notion of *representation independence* [5]. This is expressed in terms of so-called *observable* types. This limits us to comparing the two interpretations in terms of words, but this is sufficient to observe the operand stack and local variables, where the results of execution are stored.

[1] We follow the terminology of [11], where a method is *public visible* if it has either a `protected` or a `public` modifier, and *package visible* if it is declared `private` or has no visibility modifier.

Representation independence may be proven by defining *coupling relations* between the two formats that respect the tokenisation and are the identity at observable types. This can be seen as formalising a *data refinement* from class to CAP file. We formalise the relations nondeterministically as any family of relations which satisfies certain constraints, rather than as explicit transformations. This is because there are many possible tokenisations and we wish to prove any reasonable optimisation correct.

The virtual machines are formalised in an operational style, as transition relations over abstract machines. We adopt the action semantics formalism of Mosses [6], using a mixture of operational and denotational styles: the virtual machines are formalised operationally, parameterised with respect to a number of auxiliary functions, which are then interpreted denotationally. This modular presentation of the semantics facilitates the comparison between the two formats. We illustrate this for dynamic method lookup, used in the semantics of the method invocation instructions. The lookup function which searches for the implementation of a method is dependent on the layout of the method tables. The operational rule giving the semantics of the method invocation instructions, presented in Section 5, is parameterised with respect to the lookup function. Then in Section 6 two possible interpretations of lookup are given.

In Section 4, we define abstract types for the various entities converted during tokenisation, which are common to the two formats. For example, `Class_ref` and `Environment`. It is this type structure which is used to define the logical relations. In Section 5 we give an operational semantics which is independent of the underlying class/CAP file format. The structure of the class/CAP file need not be visible to the operational semantics. We need only be able to extract certain data corresponding to a particular method, such as the appropriate constant pool. In Sections 6, we give the specific details of the class file and CAP file formats, respectively, defined as interpretations of types and auxiliary functions, $[\![.]\!]_{name}$ and $[\![.]\!]_{tok}$. We refer to these as the *name* and the *token* interpretation, respectively.

In Section 7, we define the logical relation, $\{R_\theta\}_{\theta \in Abstract\_type}$. It is convenient to group the definition into several levels. First, there are various basic observable types (byte, short, *etc.*), γ, for which we have $R_\gamma = id_\gamma$. Second, there are the references, ι, such as package and class references, for which the relation R_ι represents the tokenisation of named items. Third, the constraints on the organisation into components (which we will call the *componentisation*) are expressed in R_κ, where κ includes method information structures, constant pools, and so on. This represents the relationship between components in CAP files and the corresponding entities in class files. Using the above three families of relations we can define R_θ for each type, θ, where

$$\theta ::= \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \to \theta' \mid \theta + \theta' \mid \theta^*.$$

The family of relations, $\{R_\theta\}_{\theta \,\in\, Abstract\_type}$, represents the overall construction of components in the CAP file format from a class file. The relations are 'logical' in the sense that the definitions for defined types follows automatically. For example, we define the type of the environment that contains the class hierarchy as

$$\mathtt{Environment} = \mathtt{Package\_ref} \to \mathtt{Package}.$$

and so the definition of $R_{\mathtt{Environment}}$ follows from those of $R_{\mathtt{Package\_ref}}$, $R_{\mathtt{Package}}$ and the standard construction of $R_{\_ \to \_}$; similarly for $R_{\mathtt{Heap}}$.

3 Related Work

There have been a number of formalisations of the Java Virtual Machine which have some relevance for our work here on Java Card. Bertelsen [1] gives an operational semantics which we have used as a starting point. He also considers the verification conditions, which considerably complicates the rules, however. Pusch has formalised the JVM in HOL [8]. Like us, she considers the class file to be well-formed so that the hypotheses of rules are just assignments. The operational semantics is presented directly as a formalisation in HOL, whereas we have chosen (equivalently) to use inference rules. All these works make various simplifications and abstractions. However, since these are formalisations of Java rather than Java Card they do not consider the CAP file format. In contrast, the work of Lanet and Requet [3] is specifically concerned with Java Card. They also aim to prove the correctness of Java Card tokenisation. Their work can be seen as complementing ours. They concentrate on optimisations, including the type specialisation of instructions, and do not consider the conversion as such. In contrast, we have specified the conversion but ignored the subsequent optimisations. Their formalism is based on the B method, so the specification and proof are presented as a series of refinements. In [7], Pusch proves the correctness of an implementation of Prolog on an abstract machine, the WAM. The proof structure is similar to ours, although there are refinements through several levels. There are operational semantics for each level, and correctness in expressed in terms of equivalence between levels. The differences between the semantics are significant, since they are not factored out into auxiliary functions as here. She uses a big-step operational semantics, which is not appropriate for us because we wish to compare intermediate results. Moreover, she uses an abstraction function on the initial state, the results being required to be identical, whereas we have a *relation* for both initial and final states.

4 Abstract Types

We use types to structure the transformation. These are not the types of the Java Card language, but rather are based on the simply-typed lambda calculus with sums, products and lists. We use record types with the actual types of fields (drawn from the official specification where not too confusing) serving as labels. Occasionally we use terms as singleton types, such as `OxFFFF` and `0`. There are two sorts of types: *abstract* and *concrete*. The idea is that abstract types are those we can think of independently of a particular format. The concrete types are the particular realisations of these, as well as types which only make sense in one particular model. For example, `CP_index` is the abstract type of indices into a constant pool for a given package. In the name interpretation, this is modelled by a class name and an index into the constant pool of the corresponding class file, *i.e.* $\mathtt{Class\_name} \times \mathtt{Index}$ where `Index` is a concrete type. In the token interpretation, however, since all the constant pools are merged, we have $[\![\mathtt{CP\_index}]\!]_{tok} = \mathtt{Package\_tok} \times \mathtt{Index}$. Another example is the various distinctions that are made between method and field references in CAP files, but not class files, and which are not relevant at the level of the operational semantics, which concerns terms of abstract types. We arrange the types so that as much as possible is common between the two formats. For example, it is more convenient to uniformly define environments as mappings of type $\texttt{Package\_ref} \to \texttt{Package}$, with Package interpreted as $\texttt{Class\_name} \to \texttt{Class\_file}$ or CAP_file.

There is a 'type of types' for the two forms of data type in Java Card — primitive types, *i.e.*, the simple types supported directly on the card, and reference types.

$$\begin{aligned}\texttt{Type} &= \{\texttt{Boolean}, \texttt{Byte}, \texttt{Short}\} + \texttt{Reference\_type}\\ \texttt{Reference\_type} &= \texttt{Array\_type} + \texttt{Class\_ref}.\end{aligned}$$

We use a separate type, Object_ref, to refer to objects on the heap. The objects themselves contain a reference to the appropriate class or array of which they form an instance.

The type Word is an abstract unit of storage and is platform specific. All we need know is that object references and the basic types, Byte, Short and Boolean, can be stored in a Word. Rather than use an explicit coercion, we assume

$$\texttt{Word} = \texttt{Object\_ref} + \texttt{Null} + \texttt{Boolean} + \texttt{Byte} + \texttt{Short}.$$

Thus a word is (*i.e.* represents) either a reference (possibly null) or an element of a primitive type. Furthermore, we define $\texttt{Value} = \texttt{Word}$. Although this is not strictly necessary, there is a conceptual distinction. If we were to introduce values of type int, then a value could be either a word or a double word.

There are several forms of references used during tokenisation, *viz.*, Package_ref, Class_ref and Method_ref. We distinguish Package from Package_ref, and similarly for the other items. Note that a *reference* is a composite entity which can be context dependent (*e.g.* in the CAP format a class reference can be in internal or external forms). We assume, however, that sufficient information is given so that references make sense globally. For example, class names are fully qualified, and class tokens are paired with a package token. We take field and method references to be to particular members of some class, and so contain a class reference. In contrast, an *identifier* is a name or a token (these are not used at the abstract level though). Using these basic types, we can then construct complex types using the usual type constructors: (non-dependent) sum, product, function and list types (denoted θ^*) as we did when defining the environment at the end of Sect. 2.

5 Operational Semantics

We define an operational semantics framework that allows us to model the execution of both class and CAP files. This is obtained by parameterising the semantics on a number of *auxiliary functions* that embody the differences between the two formats. This factorisation of the semantics reduces the equivalence proof considerably.

The official specification of the JCVM (and JVM) is given in terms of *frames*. A frame represents the state of the current method invocation, together with any other useful data. We introduce the notion of *configuration*, consisting of (the abstract syntax of) the code of the current method still to be executed, the operand stack, the local variables, and the current class reference. We write these as $\texttt{Config}(b, o, l, c)$ or just

$\langle b, o, l, c\rangle$. To account for method invocations, we allow a configuration itself to be considered as an instruction. When a method is invoked, the current instruction becomes a new configuration. Instead of a stack of frames, then, we have a single piece of 'code' (in this general sense). This form of closure is more general than the traditional idea of a call stack but helps simplify the proof. Method invocation is modelled by replacing the invoking instruction with a configuration that contains the code of the invoked method (see the detailed description of `invokevirtual` below). Execution of a method body is modelled by allowing transitions inside a configuration.

$$\frac{f \Rightarrow f'}{\langle \texttt{Config}\ f, ops, l, c\rangle \Rightarrow \langle \texttt{Config}\ f', ops, l, c\rangle}$$

The method invocation instructions (and others) take an argument which is an index into either the constant pool of a class file, or into the constant pool component of a CAP file. This means that the 'concrete' bytecode is itself dependent on the implementation and is therefore modelled by an abstract type. Formally, we define a transition relation

$$\Rightarrow\ \subseteq \texttt{Config} \times \texttt{Arrow} \times \texttt{Config}$$

using the types

$$\begin{aligned}
\texttt{Config} &= \texttt{Bytecode} \times \texttt{Word}^* \times \texttt{Locals} \times \texttt{Class_ref} \\
\texttt{Arrow} &= \texttt{Global_state} \rightarrow \texttt{Global_state} \\
\texttt{Global_state} &= \texttt{Environment} \times \texttt{Heap} \\
\texttt{Bytecode} &= \texttt{Instruction} + (\texttt{Bytecode} \times \texttt{Bytecode}) + \texttt{Config}
\end{aligned}$$

As mentioned above, the structure of the class/CAP file need not be visible to the operational semantics. We use a number of auxiliary functions, some of which have preconditions that we take as concomitant with the well-formedness of the class file. The definition of method invocation uses the `lookup` function

$$\texttt{lookup} : \texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Class\_ref} \times \texttt{Bytecode}$$

that takes the actual class reference, together with the declared method reference (which contains the class where the method is declared), and returns the class reference where the method is defined together with the code. Function $\texttt{method\_nargs} : \texttt{Method\_ref} \rightarrow \texttt{Nat}$ returns the number of arguments for a given method reference. The instruction for virtual method invocation is evaluated as follows:

1. The two byte index, i, into the constant pool is resolved to get the declared method reference containing the declared class reference and a method identifier (either a signature or token).
2. The number of arguments to the method is calculated.
3. The object reference, r, is popped off the operand stack.
4. Using the heap, we get $heap(r) = \langle act\_cref, \_\rangle$, the actual class reference (fully qualified name or a package/class token pair).
5. We then do $\texttt{lookup}(act\_cref, dec\_mref)$, getting the class where the method is implemented, and its bytecode. The lookup function is used with respect to the class hierarchy (environment).

6. A configuration is created for this method and evaluation proceeds from there.

$$\frac{\begin{array}{ll} \mathit{dec\_mref} := \texttt{constant\_pool}\ (c)(i) & \text{get declared method reference} \\ n := \texttt{method\_nargs}(\mathit{stat\_mref}) & \text{get number of arguments} \\ \langle \mathit{act\_cref}, \_\rangle := \mathit{heap}(r) & \text{get actual class reference from heap} \\ \langle m\_cl, m\_cd\rangle := \texttt{lookup}(\mathit{act\_cref}, \mathit{dec\_mref}) & \text{look up method} \end{array}}{\langle \texttt{invokevirtual}\, i, a_1 \ldots a_n :: r :: s, l, c\rangle \Rightarrow \langle\langle m\_cd, \langle\rangle, a_1 \ldots a_n :: r, m\_cl\rangle, s, l, c\rangle}$$

In the following sections we show how to instantiate the semantic framework (in particular the `lookup` function) to obtain a class file and a CAP file semantics.

6 Interpretations

The name interpretation gives semantics to Java class files (see Figure 1). Since this is fairly standard we give a brief description. Classes are described by fully qualified names, whereas methods and fields are given signatures, consisting of an unqualified name and a type, together with the class of definition. We assume a function $pack\_name$ which gives the package name of a class name. The data is arranged into class files, each of which contains all the information corresponding to a particular class. We only give the interpretation of those parts used here. We group the class files by package into a global environment so $env\_name(p)(c)$ denotes the class file in package p with name c.

$$[\![\texttt{Package}]\!]_{name} = \texttt{Class\_name} \rightarrow \texttt{Class\_file}$$
$$[\![\texttt{Class\_ref}]\!]_{name} = \texttt{Class\_name}$$
$$[\![\texttt{Method\_ref}]\!]_{name} = \texttt{Class\_name} \times \texttt{Sig}$$
$$\texttt{Sig} = \texttt{Method\_name} \times [\![\texttt{Type}]\!]^{*}_{name}$$
$$[\![\texttt{Class}]\!]_{name} = \texttt{Class\_file}$$
$$\texttt{Class\_file} = \texttt{Class\_flags} \times \texttt{Super} \times \texttt{Methods\_item} \times \texttt{Constant\_pool\_item} \times \texttt{Class\_name}$$
$$\texttt{Super} = \texttt{Class\_name} + \texttt{Void}$$
$$[\![\texttt{Pack\_methods}]\!]_{name} = \texttt{Class\_name} \rightarrow \texttt{Methods\_item}$$
$$\texttt{Methods\_item} = \texttt{Sig} \rightarrow \texttt{Method\_info}$$
$$\texttt{Method\_info} =$$
$$\texttt{Method\_flags} \times \texttt{Sig} \times ([\![\texttt{Type}]\!]_{name} + \texttt{Void}) \times \texttt{Maxstack} \times \texttt{Maxlocals} \times \texttt{Bytecode}$$

Fig. 1. Name Interpretation

The signature is not considered to include the return type. We assume that this signature is the same as the argument given to the methods item.

There are a number of possibilities for how method lookup should be defined, depending on the definition of inheritance. For example, [1, 8] use a 'naive' lookup which does not take account of visibility modifiers. A fuller discussion of this appears in [9].

```
lookup_name (act_class, (sig, dec_class)) =
let dec_pk  = pack_name(dec_class)
     act_pk  = pack_name(act_class)
     (_,_,meth_dec,_,_) = env_name (dec_pk) (dec_class)
     (_,super,_,meth_act,_,_)  = env_name (act_pk) (act_class)
     (dec_flags,_,_,_,_,_) = meth_dec(sig)  in
if meth_act(sig) = undefined
then lookup_name(super, (sig, dec_class))
else if
   dec_flags(protected) or dec_flags(public) or act_pk = act_pk
   then let (_,_,_,_,_,code) = meth_act(sig) in (act_class,code)
   else lookup_name(super, (sig, dec_class))
```

Fig. 2. The `lookup` function for the class file format.

6.1 Token Interpretation

In the JCVM, data is arranged by packages into CAP files. Each CAP file consists of a number of components, but not all are used for method lookup (or, indeed, the rest of the operational semantics). We just include those components we need here, namely, the constant pool, class and method components.

References to items external to a package are via tokens — for packages, classes, static fields, static methods, instance fields, virtual methods, and instance methods — each with a particular range and scope. These are then used to find internal offsets into the components. For example, a class reference is either an internal offset into the class component of the CAP file of the class' package, or an external reference composed of a package token and a class token. However, since we need to relate the reference to class names, we will assume that all references come with package information, even though this is superfluous in the case of internal references.

The class component consists of a list of class information structures, each of which has method tables, which give offsets into the method component, where the method implementations are found. The lookup algorithm uses tokens to calculate the corresponding method table index. There are separate tables for public and package methods. Method access information is given implicitly by the tokens rather than by flags. The two method tables each contain a base, size and 'list' of entries. The entries are defined from the $base$ to $base + size - 1$ inclusive. The entry for a public method will be `OxFFFF` if the method is defined in another package.

For a given class reference, the function `class_info` finds the corresponding class information structure in the global environment. The variant, `class_info`$'$ returns the class information structure in a particular CAP file. The function `method_array` simply finds the method component for a given class reference. We assume the existence of functions `class_offset` and `method_offset` for resolving external tokens to internal offsets It follows from the definition of the abstract type `Environment`, that the environment in the token format consists of a mapping from package tokens to their corresponding CAP file *i.e.*, env_{tok} : `Package_tok` $\rightarrow$ `CAP_file`. The lookup function takes a class reference (the declared class), a method reference (in the actual class), and

$$\llbracket \texttt{Package} \rrbracket_{tok} = \texttt{CAP\_file}$$
$$\texttt{CAP\_file} = \texttt{Constant\_pool\_comp} \times \texttt{Class\_comp} \times \texttt{Method\_comp}$$
$$\llbracket \texttt{Package\_ref} \rrbracket_{tok} = \texttt{Package\_tok}$$
$$\llbracket \texttt{Class\_ref} \rrbracket_{tok} = \texttt{Package\_tok} \times (\texttt{Class\_tok} + \texttt{Offset})$$
$$\llbracket \texttt{Method\_ref} \rrbracket_{tok} = \llbracket \texttt{Class\_ref} \rrbracket_{\texttt{tok}} \times \texttt{Virtual\_method\_tok}$$
$$\llbracket \texttt{Class} \rrbracket_{tok} = \texttt{Class\_info}$$
$$\texttt{Class\_comp} = \texttt{Offset} \to \texttt{Class\_info}$$
$$\texttt{Class\_info} = \texttt{Class\_flags} \times \texttt{Super} \times \texttt{Public\_table} \times \texttt{Package\_table} \times \texttt{Class\_ref}$$
$$\texttt{Public\_table} = \texttt{Public\_base} \times \texttt{Public\_size} \times (\texttt{Index} \to \texttt{Offset} + \{\texttt{0xFFFF}\})$$
$$\texttt{Package\_table} = \texttt{Package\_base} \times \texttt{Package\_size} \times (\texttt{Index} \to \texttt{Offset})$$
$$\llbracket \texttt{Pack\_methods} \rrbracket_{tok} = \texttt{Method\_comp}$$
$$\texttt{Method\_comp} = \texttt{Offset} \to \texttt{Method\_info}$$
$$\texttt{Method\_info} = \texttt{Method\_flags} \times \texttt{Maxstack} \times \texttt{Nargs} \times \texttt{Max\_locals} \times \texttt{Bytecode}$$

Fig. 3. Token Interpretation

returns the reference to the class where the code is defined, together with the bytecode itself. The main steps of the algorithm (see Fig. 4) are:

1. Get method array for the package of the actual class.
2. Get class information for the actual class.
3. If public: if defined then *get info* else *lookup super*.
   If package: if defined and visible then *get info* else *lookup super*.

7 Formalisation of Equivalence

We formalise the equivalence between the class and CAP formats as a family of relations, $\{R_\theta : \llbracket\theta\rrbracket_{name} \leftrightarrow \llbracket\theta\rrbracket_{tok}\}_{\theta \in Abstract\_type}$ indexed by abstract type, θ. The idea is that $x\ R_\theta\ y$ when y is a *possible* transformation of x. The relations are not necessarily total, *i.e.* for some $x : \llbracket\theta\rrbracket_{name}$, there may not be a y such that $x\ R_\theta\ y$. Formally, the relations are defined as a mutually inductive collection of constraints, R_θ, for each type θ, where the types, θ, are given by the grammar:

$$\begin{aligned}
\gamma &::= \texttt{Bool} \mid \texttt{Nat} \mid \texttt{Object_ref} \mid \texttt{Boolean} \mid \texttt{Byte} \mid \texttt{Short} \mid \texttt{Value} \mid \texttt{Word} \\
\iota &::= \texttt{Package_ref} \mid \texttt{Ext_class_ref} \mid \texttt{Class_ref} \mid \texttt{Method_ref} \\
\kappa &::= \texttt{CP_index} \mid \texttt{CP_info} \mid \texttt{Method_info} \mid \texttt{Package} \mid \texttt{Class} \mid \\
&\quad\ \texttt{Constant_pool} \mid \texttt{Pack_methods} \\
\theta &::= \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \to \theta' \mid \theta + \theta' \mid \theta^*
\end{aligned}$$

where the observable types are built up inductively from the γ, *i.e.* do not contain the ι and κ. There are two sources of underspecification. First, the relations really can be non-functional. Second, there is a choice for what some of the relations are. For example,

```
lookup_tok (act_class_ref, (dec_class_ref, method_tok)) =

let methods = method_array (act_class_ref)
    (_, super, (public_base,_, public_table),
               (package_base,_, package_table),_)   : Class_info =
    class_info(act_class_ref) in
if method_tok div 128 = 0 then /* public */
   if method_tok >= public_base then
    let method_offset = public_table[method_tok-public_base] in
        if method_offset <>  0xFFFF
        then (act_class_ref, methods[method_offset].Bytecode)
        else /* look in superclass */
           lookup_tok(super, (dec_class_ref, method_tok))
   else /* look in superclass */
      lookup_tok(super, (dec_class_ref, method_tok))
else /* package */
   if method_tok >= package_base /\
      same_package(dec_class_ref, act_class_ref)
    then let method_offset =
         package_table[method_tok mod 128 - package_base]
      in (act_class_ref, methods[method_offset].Bytecode)
   else /* look in superclass */
      lookup_tok(super, (dec_class_ref, method_tok))
```

Fig. 4. The `lookup` function for the CAP file format.

$R_{\texttt{Class\_ref}}$ is *some* bijection satisfying certain constraints. The relations between the 'large' structures, however, are completely defined in terms of those between smaller ones. There are two parts to the transformation itself: the tokenisation, defined as the relations R_ι, and the 'componentisation', defined as the R_κ.

7.1 Tokenisation

The relations, R_ι, represent the tokenisation of items. The general idea is to set up relations between the names and tokens assigned to the various entities, subject to certain constraints described in the specification.

In order to account for token scope, we relate names to tokens paired with the appropriate context information. For example, method tokens are scoped within a class, so the relation $R_{\texttt{Method\_ref}}$ is between pairs of class names and signatures, and pairs of class references and method tokens. We must add a condition, therefore, to ensure that the package token corresponds to the package name of this class name.

We assume that each of these relations is a bijection, modulo the equivalence between internal and external references (with one exception to account for the copying of virtual methods, explained below). Formally,

$$a\,R\,b \;\wedge\; a'\,R\,b \;\Rightarrow\; a = a'$$

$$a\,R\,b \;\Rightarrow\; (a\,R\,b' \iff \texttt{Equiv}(b, b'))$$

where equivalence, `Equiv`, of class references is defined as the reflexive symmetric closure of:

$$\texttt{Equiv}(\langle p\_tok, \mathit{offset}\rangle, \langle p\_tok, c\_tok\rangle) \iff \texttt{class\_offset}(p\_tok, c\_tok) = \mathit{offset}$$

The second condition contains two parts: that the relation is functional modulo `Equiv`, and that it is closed under `Equiv`. We say that R is an *external bijection* when these conditions hold. We extend the definition of `Equiv` and external bijection to the other references.

These relations are defined with respect to the environment (in name format). We use a number of abbreviations for extracting information from the environment. We write $c < c'$ for the subclass relation (*i.e.* the transitive closure of the direct subclass relation) and $\leq$ for its reflexive closure. In the token interpretation this is modulo `Equiv`. We write $m\_tok \in c\_ref$ when a method with token m_tok is declared in the class with reference c_ref, and $pack\_name(c)$ for the package name of the class named c.

We define function `Class_flag` for checking the presence of attributes such as `public`, `final`, *etc*. The tokenisation uses the notion of *external visibility*.

$$\texttt{Externally\_visible}(c\_name) = \texttt{Class\_flag}(c\_name, Public)$$

We will also write $public(sig)$ and $package(sig)$ according to the visibility of a method. The relation for `Method_ref` uses `Class_ref`, defined in the next section.

`Package_ref` : As mentioned above, we take package tokens to be externally visible. The relation $R_{\texttt{Package\_ref}}$ is simply defined as any bijection between package names and tokens.

`Ext_class_ref` : In order to define the relation for class references we first define the relation for external class references. We define $R_{\texttt{Ext\_class\_ref}}$ as a bijection between class names and external class references such that:
$c\_name\ R_{\texttt{Ext\_class\_ref}}\ (p\_tok, c\_tok) \Rightarrow$
$\texttt{Externally\_visible}(c\_name) \wedge\ pack\_name(c\_name)\ R_{\texttt{Package\_ref}}\ p\_tok$

`Method_ref` : This is not a bijection because of the possibility of copying. Although 'from names to tokens' we do have:

$$\begin{array}{l}\langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\ \langle c'\_name, sig'\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle\end{array} \Rightarrow \begin{cases} c\_name = c'\_name \wedge \\ sig = sig' \end{cases}$$

for a converse we have:

$$\begin{array}{l}\langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\ \langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c'\_ref, m'\_tok\rangle\end{array} \Rightarrow \begin{cases} (c\_ref \leq c'\_ref \vee c'\_ref \leq c\_ref) \\ \wedge\ m\_tok = m'\_tok \end{cases}$$

The first condition says that if a method overrides a method implemented in a superclass, then it gets the same token. Restrictions on the language mean that overriding cannot change the method modifier from public to package or vice versa.

$$\left.\begin{array}{l}\langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\ \langle c'\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c'\_ref, m'\_tok\rangle \wedge \\ c'\_name < c\_name \wedge \\ (package(sig) \Rightarrow same\_package(c\_name, c'\_name))\end{array}\right\} \Rightarrow m\_tok = m'\_tok$$

The second condition says that the tokens for public introduced methods must have higher token numbers that those in the superclass. We assume a predicate, *new_method*, which holds of a method signature and class name when the method is defined in the class, but not in any superclass.

$$\begin{array}{l} public(sig) \ \wedge \ new\_method(sig, c\_name) \ \wedge \\ (c\_name, sig) \ R_{\texttt{Method\_ref}} \ (c\_ref, m\_tok) \end{array} \Rightarrow \begin{cases} \forall m'\_tok \in super(c\_ref) \, . \\ \qquad\qquad m\_tok > m'\_tok \end{cases}$$

Package-visible tokens for introduced methods are similarly numbered, if the superclass is in the same package.

$$\begin{array}{l} package(sig) \ \wedge \ new\_method(sig, c\_name) \ \wedge \\ (c\_name, sig) \ R_{\texttt{Method\_ref}} \ (c\_ref, m\_tok) \ \wedge \\ same\_package(c\_name, super(c\_name)) \end{array} \Rightarrow \begin{cases} \forall m'\_tok \in super(c\_ref) \, . \\ \qquad\qquad m\_tok > m'\_tok \end{cases}$$

The third condition says that public tokens are in the range 0 to 127, and package tokens in the range 128 to 255.

$\langle c\_name, sig \rangle \ R_{\texttt{Method\_ref}} \ \langle c\_ref, m\_tok \rangle \ \Rightarrow (public(sig) \ \Rightarrow \ 0 \ \leq \ m\_tok \ \leq 127) \ \wedge \ (package(sig) \Rightarrow 128 \leq m\_tok \leq 255)$

The specification [11] also says that tokens must be contiguously numbered starting at 0 but we will not enforce this.

7.2 Componentisation

The relations in the previous section formalise the correspondence between named and tokenised entities. When creating the CAP file components, all the entities are converted, including the package visible ones. Thus at this point we define $R_{\texttt{Class\_ref}}$ as a relation between named items and either external tokens or internal references, subject to coherence constraints.

We must ensure that if a name corresponds to both an external token and to an internal offset, then the token and the offset correspond to the same entity. We ensure this by using the offset function $\texttt{class\_offset} : \texttt{Package\_tok} \times \texttt{Class\_tok} \rightarrow \texttt{Offset}$ which returns the internal offset corresponding to an external token, and then *define* $R_{\texttt{Class\_ref}}$ from this and $R_{\texttt{Ext\_class\_ref}}$. Clearly, therefore, $R_{\texttt{Class\_ref}}$ is not a bijection.

`Class_ref`**:** We define $R_{\texttt{Class\_ref}}$ as an external bijection which respects $R_{\texttt{Ext\_class\_ref}}$, that is, such that

$$c\_name \ R_{\texttt{Class\_ref}} \ (p\_tok, c\_tok) \iff c\_name \ R_{\texttt{Ext\_class\_ref}} \ (p\_tok, c\_tok)$$

Thus $R_{\texttt{Class\_ref}}$ extends $R_{\texttt{Ext\_class\_ref}}$ to internal references.

The three 'big' components are the constant pool, method, and class components. We mainly limit our definition of equivalence to these, though also consider the static field and descriptor components.

`Method_info`**:** We only treat certain parts of the method information here:

$$\begin{array}{c} \langle flags, sig, \_, \_, maxstack, maxlocals, code, \_ \rangle \\ R_{\texttt{Method\_info}} \\ \langle flags', maxstack', nargs', maxlocals', code' \rangle \end{array} \iff \begin{array}{l} flags \ R_{\texttt{Method\_flags}} \ flags' \ \wedge \\ maxstack = maxstack' \wedge \\ size(sig) = nargs' \wedge \\ maxlocals = maxlocals' \wedge \\ code \ R_{\texttt{Bytecode}} \ code' \end{array}$$

In the name interpretation is all the information in one package and so, for example, $[\![\texttt{Pack\_methods}]\!]_{name}$: $\texttt{Class\_name} \rightarrow \texttt{Methods\_item}$ is the 'set' of method data for all classes. In the token format the method information is spread between the two components. The coupling relations reflect this: the relation $R_{\texttt{Class}}$ ensures that a named method corresponds to a particular offset, and $R_{\texttt{Pack\_methods}}$ ensures that the entry at this offset is related by $R_{\texttt{Method\_info}}$.

`Pack_methods`**:** The method item and method component contain the implementations of both static and virtual methods.

$$\begin{array}{lcl} methods\_name & & \forall \langle c\_name, sig\rangle \; R_{\texttt{Method\_ref}} \; \langle p\_tok, c\_tok, m\_tok\rangle . \\ \;\; R_{\texttt{Pack\_methods}} & \Longleftrightarrow & \quad methods\_name(c\_name, sig) \; R_{\texttt{Method\_info}} \\ \; method\_comp & & \quad (methods\_comp.methods)(\texttt{method\_offset}(p\_tok, c\_tok, m\_tok)) \end{array}$$

`Class`**:** We define $R_{\texttt{Class}}$. There are a number of equivalences expressing correctness of the construction of the class component. For the lookup, the significant ones are those between the method tables. These say that if a method is defined in the name format, then it must be defined (and equivalent) in the token format. Since the converse is not required, this means we can copy method tokens from a superclass. Instead, there is a condition saying that if there is a method token, then there must be a corresponding signature in some superclass.

If a method is visible in a class, then there must be an entry in the method table, indicating how to find the method information structure in the appropriate method component. For package visible methods this implies that the method must be in the same package. For public methods, if the two classes are in the same package, then this entry is an offset into the method component of this package. Otherwise, the entry is `OxFFFF`, indicating that we must use the method token to look in another package.

The class component only contains part of the information contained in the class files. The full definition is given in Figure 5. (writing c_name for $cf.Class\_name$ and c_ref for $ci.Class\_ref$): The offset functions link the various relations. We make a global assumption (in fact, local to an environment) of the existence of `class_offset` and `method_offset`.

Equivalence proof: The full proof establishes that the auxiliary functions preserve the appropriate relations [2]. Here, we state the main lemma for the function `lookup` whose type is $\texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Class\_ref} \times \texttt{Bytecode}$.

Lemma 1. *If the heap and environment are related in the two formats, then:*

$$[\![lookup]\!]_{name} \; R_{\texttt{Class\_ref} \times \texttt{Method\_ref} \rightarrow \texttt{Class\_ref} \times \texttt{Bytecode}} [\![lookup]\!]_{tok}$$

In order to use the operational semantics with the logical relations approach it is convenient to view the operational semantics as giving an interpretation. We define $[\![code]\!](\langle env, heap, op\_stack, loc\_vars, m\_ref\rangle)$ as the resulting state from the (unique) transition from $\langle code, op\_stack, loc\_vars\rangle$ with environment env and heap $heap$. Thus interpreted bytecode has type $\texttt{State} \rightarrow \texttt{Bytecode} \times \texttt{State}$ where `State` is

$$\texttt{State} = \texttt{Global\_state} \times \texttt{Operand\_stack} \times \texttt{Local\_variables} \times \texttt{Class\_ref}$$

$$
cf : \texttt{Class\_file}\ R_{\texttt{Class}}\ ci : \texttt{Class\_info} \iff
\begin{cases}
cf.Class\_flags\ R_{\texttt{Class\_flags}}\ ci.Class\_flags \wedge \\
cf.Super\ R_{\texttt{Class\_ref}}\ ci.Super \wedge \\
\\
\forall sig \in cf.Methods\_item. \\
public(sig) \Rightarrow \\
\exists m\_tok\ .\ ci.Public\_base \leq m\_tok < ci.Public\_base + ci.Public\_size \wedge \\
\langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\
ci.Public\_table[m\_tok - ci.Public\_base] = \texttt{method\_offset}(c\_ref, m\_tok) \\
\wedge \\
package(sig) \Rightarrow \\
\exists m\_tok\ .\ ci.Package\_base \leq m\_tok\ \&\ 127 < ci.Package\_base + ci.Package\_size \wedge \\
\langle c\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge \\
ci.Package\_table[m\_tok\ \&\ 127 - ci.Package\_base] = \texttt{method\_offset}(c\_ref, m\_tok) \\
\wedge \\
\forall m\_tok \in ci.Public\_table \cup ci.Package\_table. \exists sig. \exists c'\_name. \\
\langle c'\_name, sig\rangle\ R_{\texttt{Method\_ref}}\ \langle c\_ref, m\_tok\rangle \wedge c\_name \leq c'\_name \wedge \\
public(sig) \Rightarrow [\ (same\_package(c\_name, c'\_name) \iff \\
\qquad\qquad ci.Public\_table[m\_tok - ci.Public\_base] \neq \texttt{0xFFFF})]
\end{cases}
$$

Fig. 5. Definition of $R_{\texttt{Class}}$

Now, the following fact is trivial to show: if $R_B = id_B$ for all basic observable types, then $R_\theta = id_\theta$ for all observable θ. In combination with the following theorem, then, this says that if a transformation satisfies certain constraints (formally expressed by saying that it is contained in R) then it is correct, in the sense that no difference can be observed in the two semantics. In particular, we can observe the operand stack (of observable type `Word*`) and the local variables (of observable type `Nat` $\rightarrow$ `Word`) so these are identical under the two formats.

Theorem 1. *Assume that* $env_{name}\ R_{\texttt{Environment}}\ env_{tok}$, $heap_{name}\ R_{\texttt{Heap}}\ heap_{tok}$, $ls\ R_{\texttt{Local\_state}}\ ls'$, *and* $code\ R_{\texttt{Bytecode}}\ code'$. *Then*

$$[\![code]\!]_{name}(env_{name}, heap_{name}, ls)\ R_{\texttt{Bytecode}\times\texttt{State}}\ [\![code']\!]_{tok}(env_{tok}, heap_{tok}, ls')$$

8 Conclusion

We have formalised the virtual machines and file formats for Java and Java Card, and the optimisation as a relation between the two. Correctness of this optimisation was expressed in terms of observable equivalence of the operational semantics, and this was deduced from the constraints that define the optimisation. Although the framework we have presented is quite general, the proof is specific to the instantiations of auxiliary functions we chose. It could be argued that we might have proven the equivalence of two incorrect implementations of lookup. The remedy for this would be to specify the functions themselves, and independently prove their correctness. Furthermore, we have made a number of simplifications which could be relaxed. We have used a simple definition of $R_{\texttt{Bytecode}}$ here, which just accounts for the changing indexes into constant pools (as well as method references in configurations). We have not considered inlining or the specialisation of instructions, however. We expressed equivalence in terms of an identity at observable types but we should also account for the difference in word size, as in [3]. Although the specialisation of instructions could be handled by our technique, the extension is less clear for the more non-local optimisations.

We emphasise that the particular form of operational semantics used here is orthogonal to the rest of the proof. This version suffices for the instructions considered here, but could easily be changed (along with the definition of $R_{\texttt{Bytecode}}$). The auxiliary functions could be given different definitions; for example, an abstract interpretation or, going in the opposite direction, including error information.

The definitions have been formalised in Coq, and the lemmas verified [9]. The discipline this imposed on the work presented here was very helpful in revealing errors. Even just getting the definitions to type-check uncovered many errors. We take the complexity of the proofs (in Coq) as evidence for the merit in separating the correctness of a particular algorithm from the correctness of the specification. In fact, the operational semantics, correctness of the specification, and development of the algorithm are all largely independent of each other.

As mentioned in the introduction, there are two main steps to showing correctness:

1. Give an abstract characterisation of all possible transformations and show that the abstract properties guarantee correctness.
2. Show that an algorithm implementing such a transformation exists.

We are currently working on a formal development of a tokenisation algorithm using Coq's program extraction mechanism together with constraint-solving tactics.

References

1. P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.
2. Ewen Denney. Correctness of Java Card Tokenisation. Technical report, Projet Lande, IRISA, 1999.
3. Jean-Louis Lanet and Antoine Requet. Formal proof of smart card applets correctness. In *Third Smart Card Research and Advanced Application Conference (CARDIS'98)*, 1998.
4. T. Lindholm and F. Yelling. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.
5. J. Mitchell. *Foundations for Programming Languages*. Foundations of Computing Series. MIT Press, 1996.
6. Peter D. Mosses. Modularity in structural operational semantics. November 1998.
7. Cornelia Pusch. Verification of Compiler Correctness for the WAM. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics (TPHOLs'96)*, pages 347–362. Springer-Verlag, 1996.
8. Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.
9. Gaëlle Segouat. Preuve en Coq d'une mise en oeuvre de Java Card. Master's thesis, Projet Lande, IRISA, 1999.
10. Sun Microsystems. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997. Final Revision.
11. Sun Microsystems. *Java Card 2.1 Virtual Machine Specification*, March 1999.