Theoretical Computer Science 283 (2002) 305–331

Theoretical Computer Science

www.elsevier.com/locate/tcs

Correctness of Java Card method lookup via logical relations ☆

Ewen Denney[a], Thomas Jensen[b,*]

[a] Mathematical Reasoning Group, Division of Informatics, University of Edinburgh, Edinburgh EH1 1HN, Scotland, UK

[b] IRISA/CNRS, Campus de Beaulieu, F-35042 Rennes Cedex, France

Abstract

This article presents a formalisation of the bytecode optimisation of Sun's Java Card language from the class file to CAP file format as a set of constraints between the two formats, and defines and proves its correctness. Java Card bytecode is formalised using 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. © 2002 Elsevier Science B.V. All rights reserved.

Keywords: Compiler correctness; Optimisation; Operational semantics; Java; Smart cards

1. Introduction

Using a high-level language for programming embedded systems may require a transformation phase in order that the compiled code fits on the device. This is the case when mapping Java onto smart cards. 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.

The Java Card language [14] 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 [6], 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 [15]. The core of this optimisation is a *tokenisation* in which

☆ This work was supported by the INRIA *Action de recherche coopérative* Java Card.

* Corresponding author.

E-mail addresses: ewd@dai.ed.ac.uk (E. Denney), thomas.jensen@irisa.fr (T. Jensen)

0304-3975/02/$ - see front matter © 2002 Elsevier Science B.V. All rights reserved.
PII: S0304-3975(01)00138-4

names (strings) are replaced with tokens (integer values). Replacing strings with integers reduces the size of the code and enables a faster lookup of, e.g., virtual methods. Additional optimisations are obtained by a *componentisation* that merges class files from the same package into one *CAP file*. This means that symbolic references between classes from the same CAP file can be transformed into memory offsets relative to the CAP file.

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 from the official specification of the tokenisation and show that any transformation satisfying these constraints is 'correct'. Notice that this is independent of showing that there actually exists a collection of functions satisfying these constraints (which is not done here). 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 [15] (given certain assumptions) are sufficient. Moreover, the technique used to develop an algorithm is orthogonal to this proof. In this article we give a formalisation and correctness proof for the part concerned with dynamic method lookup. A comprehensive formalisation appears as a technical report [4].

A distinguishing feature of our approach is the use of types and logical relations to structure and describe the transformation of low-level code. The proof is presented in the setting of Java Card byte code but the technique itself is not limited to this particular language.

2. 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 *conversion* 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. In addition, 'top-level' references, to packages (and applets) are made using *application identifiers* (AIDs).

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. There are a number of such components, of which we will consider the constant pool, class, and method components. One significant

[1] We follow the terminology of [15], 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.

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 [15] 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. 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, which we do not treat here—see the official documentation [4, 15] for details.

3. 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* [11, 7]. This is expressed in terms of so-called *observable* types. This limits us to comparing the two interpretations in terms of words (there are no double words in Java Card), but this is sufficient to observe the operand stack and local variables, where the results of execution are stored.

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 non-deterministically 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. Formally, we say that a function is representation independent if it maps related inputs to related outputs. This is the definition of a *logical relation* at function types.

The virtual machines are formalised in an operational style, as transition relations over abstract machines. We adopt a style of semantics proposed by Mosses [8], 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. Transitions are decorated with operations that describe the effect of the transition on the global state. By presenting the bytecode semantics in such a modular manner we can more easily make the comparison between the two formats where significant. We prove the correctness of tokenisation with respect to these semantics.

It should be noted that the particular formalisation of the semantics is orthogonal to the technique used for proving equivalence. The main point is to give a set of operational rules which can be used for both virtual machines, with all the semantic differences abstracted out into a number of auxiliary functions. This means that we only have to prove that the auxiliary functions behave equivalently in the two formats in order to prove the semantics equivalent. This reduces the proof burden considerably.

In Section 5, 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 6 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 7 and 8, 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.

To illustrate how it is natural to conceive the operational semantics independently of certain auxiliary functions, we consider 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. There are also a number of choices for how it is affected by method modifiers, each of which is apparently consistent with the official specification. The operational rule giving the semantics of the virtual method invocation instruction, presented in Section 6, is parameterised with respect to the lookup function. Then in Sections 7 and 8 two possible interpretations of lookup are given. The operational semantics, together with the interpretations of the auxiliary functions, induces an 'interpretation' of the bytecode, and it is in terms of this that we compare the two formats.

In Section 9, we define the logical relation, $\{R_\theta\}_{\theta \in Abstract_type}$. It is convenient to informally group the definition into several levels. First of all, 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 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

```
Environment = Package_ref → Package
```

and so the definition of $R_{\mathtt{Environment}}$ follows from those of $R_{\mathtt{Package_ref}}$, $R_{\mathtt{Package}}$ and the (standard [2]) construction of $R_{_\to_}$. Intuitively, this imposes the restriction that in order for two environments e_n and e_t to be related. a package reference $pref_n$ and its conversion $pref_t$ must be mapped into packages p_n and p_t such that p_t is a conversion of p_n. In Section 10, we use this semantic format to prove the correctness. Finally, we make some concluding remarks in Section 11.

4. 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 [2] gives an operational semantics which we have used as a starting point. He also considers the bytecode verification conditions, which considerably complicates the rules, however. Pusch has formalised the JVM in HOL [10]. 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. Börger and Schulte [3] present a formalisation of the JVM using the formalism of abstract state machines (ASMs). By progressively adding language features such as classes, objects and exceptions to a core set of bytecode instructions, they provide an incremental development of a JVM semantics as an abstract machine for executing JVM bytecode. By coupling this semantics with an ASM semantics for Java, Börger and Schulte are able to provide a set of constraints that are sufficient for a Java compiler to be correct. All these works make various simplifications and abstractions. Since these are formalisations of Java rather than Java Card they do not consider the CAP file format. We have chosen a small-step operational semantics for this paper but the other formalisms could likely have been used as well. Our main criteria was to obtain a semantics that allows to isolate the differences between the bytecode and the CAP formats in a few auxillary functions and our choice fulfills this goal.

[2] In fact, we allow functions to be partial.

In contrast to the above-mentioned works, the work of Lanet and Requet [5] 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 [9], 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. In his Ph.D. Thesis [12], Schellhorn proves a similar correctness result for a WAM-implementation of Prolog using an Abstract State Machines semantics. The correctness of the translation is expressed in Dynamic Logic and checked using the verification tool KIV.

5. Abstract types

This section introduces the fundamental notions of abstract and concrete types that are used to structure the transformation. These are not the types of the Java Card language, but rather standard types such as sums, products, functions and lists. Here and in later sections, we use record types with the actual types of fields (drawn from the official specification where not too confusing) serving as labels. We write elements of sum types in the form $\langle tag, value \rangle$.

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.

$$[\![\texttt{CP_index}]\!]_{name} = \texttt{Class_name} \times \texttt{Index}$$

where `Index` is a concrete type. In the token format, however, since all the constant pools are merged, we have

$$[\![\texttt{CP_index}]\!]_{\text{tok}} = \texttt{Package_tok} \times \texttt{Index}.$$

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 the environment containing the hierarchy of defined classes as a mapping from package references to packages

$$\texttt{Environment} = \texttt{Package_ref} \rightarrow \texttt{Package}$$

and then define the different representations of packages by having `Package` interpreted as `Class_name` → `Class_file` in the class file format and as a `CAP_file` in the CAP file format.

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.

```
Type = {Boolean, Byte, Short} + Reference_type

    Reference_type = Array_type + Class_ref.
```

We have not included `Int`, which is optional. 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

```
Word = Object_ref + Null + Boolean + Byte + Short.
```

Thus, a word is (i.e. represents) either a reference (possibly null) or an element of a primitive type. Furthermore, we define `Value` = `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 reference [3] used during tokenisation:

```
Package_ref | Class_ref | 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. For example, in the CAP format a class reference can be in internal or external forms depending on whether the reference is internal to a package or to a class in another package. 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

[3] Which we distinguish from `Reference_type`.

did when defining the environment of classes at the end of Section 3. The heap of dynamically allocated objects is another example. It is modelled by

$$\begin{aligned}
\texttt{Heap} &= \texttt{Object_ref} \ \rightarrow \ \texttt{Object}\\
\texttt{Object} &= \texttt{Class_inst_obj + Array_obj}\\
\texttt{Class_inst_obj} &= \texttt{Class_ref} \times (\texttt{Field_ref} \rightarrow \texttt{Value})\\
\texttt{Array_obj} &= \texttt{Nat} \times \texttt{Array_type} \times (\texttt{Nat} \rightarrow \texttt{Value})\\
\texttt{Array_type} &= \texttt{Primitive_type + Class_ref.}
\end{aligned}$$

For completeness, we mention that the abstract type for the constant pool would be defined as follows:

$$\texttt{Constant_pool=CP_index} \rightarrow \texttt{(Class_ref + Method_ref + Field_ref),}$$

but in this article we are not concerned with the componentisation of the constant pool and do not deal with it in any further detail.

6. Operational semantics

In this section 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, as shown in Section 10.

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. There is some choice for how to model frames and the various formalisations of bytecode semantics in the literature differ slightly in their approach. Although the official specification also mentions a reference to the current constant pool we can calculate this from the current class reference. We abstract away from details of program counters and (literal) byte codes, and instead formalise the code as abstract syntax. The state of the virtual machine is captured by 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 equivalent to the traditional idea of a call stack.

We use a single-step SOS with rules given in a particular semantics style (proposed by Mosses [8]) as

$$\mathit{config} \stackrel{\mathit{statechange}}{\Longrightarrow} \mathit{config}'$$

meaning that the configuration *config* becomes *config'* with a change in the environment and heap given by *statechange*. We use *heap* and *env* as 'global variables' in the hypotheses. When the heap and environment do not change, we will not write the label explicitly. If an instruction changes the heap or the environment, then we label the arrow with an operation which abstracts the effect on the global state. For example, $add(r,o)$ is the arrow

$$\langle h,e\rangle \mapsto \langle h+(r\mapsto o),e\rangle$$

which extends the heap with the binding of reference r to object o, and leaves the environment unchanged. Similarly, the instruction, `putstatic`, is defined using the function, `update`, which takes a class, a static field of that class, and a value of compatible type, and overlays the change to the environment given by updating the field with that value:

$$\frac{f_ref := \texttt{constant_pool}\ (c)(i)}{\langle \texttt{putstatic}\ i, v :: ops, l, c\rangle \stackrel{\texttt{update}(f_ref,v)}{\Longrightarrow} \langle nop, ops, l, c\rangle}$$

Since execution does not terminate, as such, we introduce an artificial instruction `nop` to signify the termination of an instruction. The following two rules are standard for SOS:

$$\frac{\langle b_1, ops, l, c\rangle \Rightarrow \langle b_1', ops', l', c'\rangle}{\langle b_1; b_2, ops, l, c\rangle \Rightarrow \langle b_1'; b_2, ops', l', c'\rangle} \quad \frac{\langle b_1, ops, l, c\rangle \Rightarrow \langle \texttt{nop}, ops', l', c'\rangle}{\langle b_1; b_2, ops, l, c\rangle \Rightarrow \langle b_2, ops', l', c'\rangle}$$

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:

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

By using configurations we can model return from methods very simply by the transition

$$\langle\langle \texttt{return}, _, _, _\rangle, ops, l, c\rangle \Rightarrow \langle \texttt{nop}, ops, l, c\rangle$$

that replaces the current configuration with the `nop` instruction. This, together with the rule for sequential composition will result in the control being transferred to the instruction following the invocation of the method:

$$\frac{}{\langle\langle \texttt{return}, _, _, _\rangle, ops, l, c\rangle \Rightarrow \langle \texttt{nop}, ops, l, c\rangle}$$

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

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

The `Instruction` type lists all those instructions whose behaviour depends on the particular format.

$$\begin{aligned}
\texttt{Config} &= \texttt{Bytecode} \times \texttt{Word}^* \times \texttt{Locals} \times \texttt{Class_ref}\\
\texttt{Arrow} &= \texttt{Global_state} \to \texttt{Global_state}\\
\texttt{Global_state} &= \texttt{Environment} \times \texttt{Heap}\\
\texttt{Bytecode} &= \texttt{Instruction + (Bytecode} \times \texttt{Bytecode) + Config}\\
\texttt{Instruction} &= \texttt{Nop + Invokevirtual CP_index + Invokestatic CP_index +}\\
&\quad \texttt{Invokeinterface CP_index + Invokespecial SM_index +}\\
&\quad \texttt{Return + New CP_index + Putstatic CP_index +}\\
&\quad \texttt{Getstatic CP_index + Putfield CP_index +}\\
&\quad \texttt{Getfield CP_index + Checkcast Typecode +}\\
&\quad \texttt{Instanceof Typecode}.
\end{aligned}$$

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

$$\texttt{lookup} : \texttt{Class_ref} \times \texttt{Method_ref} \to \texttt{Class_ref} \times \texttt{Bytecode}$$

that takes the class reference where a method is declared, together with the actual method reference (which contains the actual class reference), and returns the class reference where the method is defined together with the code. The function $\texttt{method_nargs} : \texttt{Method_ref} \to \texttt{Nat}$ returns the number of arguments for a given method reference.

6.1. Invokevirtual

The instruction 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 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.

Formally, this is expressed by the following inference rule:

$$\frac{\begin{array}{ll} dec_mref := \texttt{constant_pool}\ (c)(i) & \text{get declared method reference} \\ n := \texttt{method_nargs}(dec_mref) & \text{get number of arguments} \\ \langle act_cref, _\rangle := heap(r) & \text{get actual class reference from heap} \\ \langle m_cl, m_cd\rangle := \texttt{lookup}(act_cref, dec_mref) & \text{look up method} \end{array}}{\langle \texttt{invokevirtual}\ i, a_1 \ldots a_n :: r :: s, l, c\rangle \Rightarrow \langle\langle m_cd, \langle\rangle, a_1 \ldots a_n :: r, m_cl\rangle, s, l, c\rangle}$$

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.

7. Name interpretation

The name interpretation gives semantics to Java class files (see Fig. 1). Since this is fairly standard we only give a brief description. Classes are described by fully qualified names (the set Class_name). We assume a function *pack_name* which gives the package name of a class name. Methods and fields, on the other hand, are given signatures, consisting of an unqualified name and a type, together with the class of definition. The signature is not considered to include the return type of a method.

The class files are grouped by package into a global environment that is accessed by the function *env_name*. Thus $env_name(p)(c)$ denotes the class file in package p with name c. A package is modelled as a function from class names to class files. A class file contains all the information corresponding to a particular class, viz., the various flags (public, final, *etc*) given to the class, a reference to the super class (which is void for java.lang.Object), a collection Methods_item of the methods *defined* in the class, a constant pool and the fully qualified name of the class. We only give the interpretation of those parts used here (for more details see [4]) and so leave, e.g. the constant pool unspecified.

Each method defined in a class is represented by a method item that lists the flags of the method, its signature and return type, the maximal size of the operand stack and number of local variables and the bytecode of the method.

The procedure for method lookup in class files

$$\texttt{lookup} : \texttt{Class_ref} \times \texttt{Method_ref} \rightarrow \texttt{Class_ref} \times \texttt{Bytecode}$$

Types

```
Class_file =
Class_flags × Super × Methods_item × Constant_pool × Class_name

Method_info =
Method_flags × Sig × ([[Type]]_name + Void) × Maxstack × Maxlocals×
Bytecode

    [[Package]]_name = Class_name → Class_file
    [[Class_ref]]_name = Class_name
   [[Method_ref]]_name = Class_name × Sig
               Sig = Method_name × [[Type]]*_name
      [[Class]]_name = Class_file
             Super = Class_name + Void
[[Pack_methods]]_name = Class_name → Methods_item
      Methods_item = Sig → Method_info
```

Fig. 1. Name interpretation.

is defined in Fig. 2 using an extended lambda calculus with conditionals, case expressions, let expressions, and pattern matching in both lets and abstractions. There are a number of possibilities for how method lookup should be defined, depending on the definition of inheritance. The basic functionality is standard: in order to find the implementation of a method `sig` that was declared in the class `dec_class` we recursively search for the last overriding of `sig`, starting form the current class and moving up in the class hierarchy towards `dec_class`. Differences in the literature occur when it comes to taking visibility modifiers for methods into account. For example, [2, 10] use a 'naive' lookup which does not take account of visibility modifiers. Our definition of method lookup takes these into account by making the test

```
dec_flags(protected) or dec_flags(public)
or act_pk = dec_pk
```

This test ensures that the declaration of the method being looked up is indeed visible from the class in which the candidate implementation is given. This is a necessary condition for the actual method to override a method with the same signature declared in package `dec_pk`. A fuller discussion of this appears in [13].

8. 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

```
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 = dec_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.

rest of the operational semantics). Here we limit the discussion to the 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. [4]

The CAP file interpretation of the abstract types is given in Fig. 3. The class component consists of a list of class information structures where the main difference with respect to the name interpretation is the replacement of the method item by method tables. There are two method tables: one for publicly visible methods and one for methods visible only in the package. The entries in the method tables give offsets into the method component, where the method implementations are found. The token of a method is used to calculate the entry of the method. First,

[4] Note that although package tokens should be scoped *within* a particular CAP file (being indices to an AID in the package table), we will assume they have global scope.

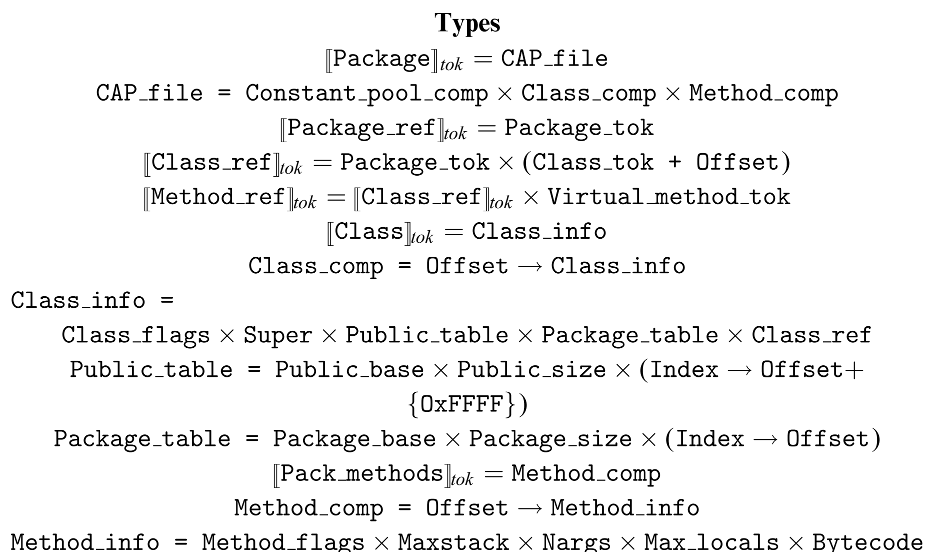

Fig. 3. Token interpretation.

public methods must have tokens in the range 0–127 and package-visible methods in the range 128–255. Thus, method access information is given implicitly by the tokens (rather than by flags); in particular they indicate which table to look in.

The Java Card specification includes the possibility of compressing the method tables. The two method tables each contain a base, size and 'list' of entries and the table only contains entries from the *base* to $base + size - 1$ inclusive. The methods with tokens lower than *base* (which by construction are methods inherited from the super-classes—see Section 9.2) must be looked up in the method tables of the super-class. This avoids duplicating parts of method tables at the expense of a more complicated method lookup. Finally, the entry for a public method will be `0xFFFF` if the method is defined in another package (this is the case for public methods that are inherited by classes in other packages).

The lookup function

$$\texttt{lookup}: \texttt{Class_ref} \times \texttt{Method_ref} \rightarrow \texttt{Class_ref} \times \texttt{Bytecode}$$

for the CAP format (given in Fig. 4) takes a class reference (the declared class), a method reference (in the actual class), and returns the reference to the class where the code is defined, together with the bytecode itself. The main steps of the algorithm 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*

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

let methods = method_array (act_class_ref)
in
let (_,super,(public_base,_,public_table),
             (package_base,_,package_table),_)
    = 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 and
       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.

If package: if defined and visible then *get info* else *lookup super*
It uses tokens to calculate the corresponding method table index as described above. We assume several local auxiliary functions:

```
class_info : Class_ref → Class_info
methods_array : Class_ref → (Offset → Method_info)
same_package : Package_ref × Package_ref → bool
class_offset : Package_tok × Class_tok → Offset
method_offset : Package_tok × Class_tok × Virtual_method_tok →
                Offset
```

For a given class reference, the function `class_info` finds the corresponding class information structure in the global environment. The function `method_array` simply

finds the method component for a given class reference. In addition, we assume the existence of functions `class_offset` and `method_offset` for resolving external tokens to internal offsets.

9. Formalisation of equivalence

We now formalise the correct conversions from class to CAP files as a family of relations,

$$\{R_\theta : [\![\theta]\!]_{name} \leftrightarrow [\![\theta]\!]_{tok}\}_{\theta \in Abstract_type}$$

indexed by abstract type, θ. The idea is that there is a fixed family of relations such that $x\,R_\theta\,y$ when y is a *possible* transformation of x.

Recall that the set of abstract types is defined 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 γ are the observable types, ι the type of entities that are tokenised, κ the type of entities transformed by the componentisation and θ contains the type of the compound entities built from the basic entities (such as constant pools, class environments, etc). We structure the description according to the grammar. We first define the relations for observable and compound types (Section 9.1). These are essentially logical relations for the type constructors and are used throughout the definition of the transformation. Then there are two parts to the transformation itself: the tokenisation, defined as the relations R_ι (Section 9.2), and the componentisation, defined as the R_κ (Section 9.3). The relations are not necessarily total, i.e. for some $x : [\![\theta]\!]_{name}$, there may not be a y such that $x\,R_\theta\,y$. We make no restrictions on the relation domains. [5]

There are two sources of underspecification here. First, the relations really can be non-functional. Second, there is a choice for what some of the relations are. For example, $R_{\texttt{Class_ref}}$ is *some* bijection satisfying certain constraints. The relations between the 'large' structures, however, are completely defined in terms of those between smaller ones.

---

[5] For example, arrays are not in the domain of $R_{\texttt{Class_ref}}$. This is not important for the proof of correctness. However, this would certainly have to be taken into account for the development of an algorithm.

9.1. Observable and compound types

For each basic observable type γ, we have $R_\gamma = id_\gamma$. For the compound types there are standard definitions of $R_{\theta\times\theta'}$, $R_{\theta\to\theta'}$ and $R_{\theta+\theta'}$ in terms of R_θ and $R_{\theta'}$.

$$\begin{array}{rcl} a\,R_\gamma\,a' & \Leftrightarrow & a = a' \\ \langle a,b\rangle R_{\theta\times\theta'} \langle a',b'\rangle & \Leftrightarrow & a\,R_\theta\,a' \wedge b\,R_\theta\,b' \\ & [\,]R_{\theta*}[\,] & \\ a :: as\,R_{\theta*}\,a' :: as' & \Leftrightarrow & a\,R_\theta\,a' \wedge as\,R_{\theta*}\,as' \\ f\,R_{\theta\to\theta'}\,f' & \Leftrightarrow & \forall a\,R_\theta\,a' \,.\, f\,a\,R_{\theta'}\,f'a' \end{array}$$

In general, functions are partial. Thus if $f\,a$ is defined and $a\,R_\theta\,a'$, then $f'a'$ must be defined.

$$a\,R_{\theta+\theta'}\,a' \;\Leftrightarrow\; \left\{ \begin{array}{c} (\exists b,b' \,.\, a = \mathtt{Theta}\,b \wedge a' = \mathtt{Theta}\,b' \wedge b\,R_\theta\,b') \\ \vee \\ (\exists c,c' \,.\, a = \mathtt{Theta'}\,c \;\wedge\; a' = \mathtt{Theta'}\,c' \wedge c\,R_{\theta'}\,c') \end{array} \right.$$

Strictly speaking, because the types are mutually recursive, we should define the relations recursively, but we will gloss over this point. As an example of a derived relation, it follows that $R_{\mathtt{Heap}}$ is defined as

$$heap_{name}\;R_{\mathtt{Heap}}\;heap_{tok} \;\Leftrightarrow\; \forall r : \mathtt{Object_ref}\,.\,heap_{name}(r)\;R_{\mathtt{Object}}\;heap_{tok}(r)$$

where $R_{\mathtt{Object}}$, in turn, is defined in terms of $R_{\mathtt{Class_ref}}$.

9.2. Tokenisation

The tokenisation process assigns tokens to *external* class references, static fields and static methods, and to all instance fields, virtual methods and interface methods. 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. 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 $\leqslant$ 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* of public classes that are visible outside their defining package:

$$\mathtt{Externally_visible}(c_name) = \mathtt{Class_flag}(c_name, Public)$$

We will also write $public(sig)$ and $package(sig)$ according to the visibility of a method.

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. Therefore, we add a condition to ensure that the package token corresponds to the package name of this class name.

`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 such that

$$c_name\, R_{\texttt{Ext_class_ref}}\, (p_tok, c_tok) \Rightarrow \begin{cases} \texttt{Externally_visible}(c_name) \wedge \\ pack_name(c_name)\, R_{\texttt{Package_ref}}\ p_tok \end{cases}$$

`Method_ref`: The relation for `Method_ref` uses `Class_ref`, defined in Section 9.3. $R_{\texttt{Method_ref}}$ is not a bijection because we couple method tokens with class tokens to give the context of the method token. This results in a method token being coupled with class tokens for all the classes in which the method is inherited, thus we only have the property:

$$\begin{matrix} \langle c_name, sig\rangle\, R_{\texttt{Method_ref}}\, \langle c_ref, m_tok\rangle \\ \wedge \\ \langle c_name, sig\rangle\, R_{\texttt{Method_ref}}\, \langle c'_ref, m'_tok\rangle \end{matrix} \Rightarrow \begin{cases} (c_ref \leqslant c'_ref \vee c'_ref \leqslant c_ref) \\ \wedge \\ m_tok = m'_tok \end{cases}$$

However, ‘from names to tokens’ we *do* have:

$$\left.\begin{matrix} \langle c_name, sig\rangle\, R_{\texttt{Method_ref}}\, \langle c_ref, m_tok\rangle \\ \wedge \\ \langle c'_name, sig'\rangle\, R_{\texttt{Method_ref}}\, \langle c_ref, m_tok\rangle \end{matrix}\right\} \Rightarrow \begin{cases} c_name = c'_name \\ \wedge \\ sig = sig' \end{cases}$$

The following conditions on $R_{\texttt{Method_ref}}$ are formalisations of the constraints stated informally (though quite precisely) in the Java Card specification [15]. 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 means 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.

$$\left.\begin{array}{l} public(sig) \wedge \\ new_method(sig, c_name) \wedge \\ (c_name, sig)\ R_{\texttt{Method_ref}}\ (c_ref, m_tok) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \forall m'_tok \in super(c_ref). \\ m_tok > m'_tok \end{array}\right.$$

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

$$\left.\begin{array}{l} package(sig) \wedge \\ new_method(sig, c_name) \wedge \\ (c_name, sig) R_{\texttt{Method_ref}}(c_ref, m_tok) \wedge \\ same_package(c_name, super(c_name)) \end{array}\right\} \Rightarrow \left\{\begin{array}{l} \forall m'_tok \in super(c_ref). \\ m_tok > m'_tok \end{array}\right.$$

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

$$\langle c_name, sig\rangle\ R_{\texttt{Method_ref}}\ \langle c_ref, m_tok\rangle$$

then

$$\left\{\begin{array}{l} (public(sig)) \Rightarrow 0 \leqslant m_tok \leqslant 127 \wedge \\ (package(sig)) \Rightarrow 128 \leqslant m_tok \leqslant 255 \end{array}\right.$$

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

9.3. Componentisation

The relations in the previous section formalise the correspondence between named and tokenised entities. The componentisation rearranges the class files into components. The three ‘big’ components are the constant pool, method, and class components. We limit our definition of equivalence to the method and class component since these are the ones that are of relevance for method lookup.

9.3.1. Class references

When creating the CAP file components, all the entities are converted, including the package visible ones. Thus, we need to extend the tokenisation relation for public classes to cover the package visible classes as well. However, within its defining package, a class (public or package visible) can now be referenced by an offset into the class component of the CAP file corresponding to the package. For public classes this means that it can be referenced either by its token or by its offset. For this reason the

relation for class references is not a bijection, and we have to impose further coherence constraints in order to ensure that if a class name corresponds to both an external token and to an internal offset, then the token and the offset correspond to the same entity. Formally, we define an equivalence, Equiv, of class references as the reflexive symmetric closure of:

$$\texttt{Equiv}(\langle p_tok, offset\rangle, \langle p_tok, c_tok\rangle) \Leftrightarrow \texttt{class_offset}(p_tok, c_tok) = offset$$

where $\texttt{class_offset} : \texttt{Package_tok} \times \texttt{Class_tok} \rightarrow \texttt{Offset}$ returns the internal offset corresponding to an external token. The coherence constraint on the relation is then

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

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

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

Class_ref: We define $R_{\texttt{Class_ref}}$ as an external bijection which respects the relation $R_{\texttt{Ext_class_ref}}$, that is, such that

$$c_name\; R_{\texttt{Class_ref}}(p_tok, c_tok) \;\Leftrightarrow\; c_name\; R_{\texttt{Ext_class_ref}}\,(p_tok, c_tok)$$

9.3.2. Methods and classes

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

$$\begin{array}{c}\langle flags, sig, _, _, maxstack, maxlocals, code, _\rangle \\ R_{\texttt{Method_info}} \\ \langle flags', maxstack', nargs', maxlocals', code'\rangle\end{array} \Leftrightarrow \begin{cases} flags\; R_{\texttt{Method_flags}}\; flags' \wedge \\ maxstack = maxstack' \wedge \\ size(sig) = nargs' \wedge \\ maxlocals = maxlocals' \wedge \\ code\; R_{\texttt{Bytecode}}\; code' \end{cases}$$

In the name interpretation all the information is 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:

$$methods_name \; R_{\texttt{Pack_methods}} \; method_comp$$
$$\Leftrightarrow$$
$$\begin{cases} \forall \langle c_name, sig \rangle R_{\texttt{Method_ref}} \langle p_tok, c_tok, m_tok \rangle . \\ \quad methods_name(c_name, sig) \\ \qquad R_{\texttt{Method_info}} \\ (methods_comp.methods)(\texttt{method_offset}(p_tok, c_tok, m_tok)) \end{cases}$$

`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 `0xFFFF`, indicating that we must use the method token to look in another package. The full definition of $R_{\texttt{Class}}$ is given in Fig. 5. (writing *c_name* for *cf.Class_name* and *c_ref* for *ci.Class_ref*):

10. Proof

We first prove that the auxiliary functions preserve the appropriate relations.[6] Since the heap and environment are not passed as explicit arguments to the functions, we need to assume the corresponding entities are related. Note that this proof is dependent on the specific implementations of auxiliary functions and, in particular, the choice of lookup algorithm used here.

Lemma 10.1. *If the heap and environment are related in the two formats, then: for all auxiliary functions $f : \theta \to \theta'$, given the corresponding preconditions, we have $[\![f]\!]_{name} \, R_{\theta \to \theta'} \, [\![f]\!]_{tok}$.*

Proof. We will consider the case of the lookup function.

$$\texttt{lookup : Class_ref} \times \texttt{Method_ref} \to \texttt{Class_ref} \times \texttt{Bytecode}$$

[6] Thus showing that we have indeed defined a *logical* relation.

$$
cf:\texttt{Class_file}\,R_{\texttt{Class}}\,ci:\texttt{Class_info} \Leftrightarrow
\begin{cases}
cf.Class_flags\ R_{\texttt{Class_flags}}\ ci.Class_flags\ \wedge \\
cf.Super\ R_{\texttt{Class_ref}}\,ci.Super\ \wedge \\
\forall sig \in cf.Methods_item. \\
public(sig) \Rightarrow \\
\exists m_tok\ .\ ci.Public_base \leqslant m_tok < ci.Public_base + ci.Public_size\ \wedge \\
\langle c_name, sig\rangle\ R_{\texttt{Method_ref}}\ \langle c_ref, m_tok\rangle\ \wedge \\
ci.Public_table[m_tok - ci.Public_base] = \texttt{method_offset}(c_ref, m_tok) \\
\wedge \\
package(sig) \Rightarrow \\
\exists m_tok.ci.Package_base \leqslant m_tok\&127 < ci.Package_base + ci.Package_size \\
\wedge \\
\langle c_name, sig\rangle\ R_{\texttt{Method_ref}}\ \langle c_ref, m_tok\rangle\ \wedge \\
ci.Package_table[m_tok\ \&\ 127 - ci.Package_base] = \\
\qquad\qquad\qquad\qquad \texttt{method_offset}(c_ref, m_tok) \\
\wedge \\
\forall m_tok \in ci.Public_table \cup ci.Package_table.\exists sig.\exists c'_name. \\
\langle c'_name, sig\rangle\ R_{\texttt{Method_ref}}\ \langle c_ref, m_tok\rangle \wedge c_name \leqslant c'_name\ \wedge \\
public(sig) \Rightarrow [(same_package(c_name, c'_name) \Leftrightarrow \\
ci.Public_table[m_tok - ci.Public_base] \neq \texttt{0xFFFF})]
\end{cases}
$$

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

The proof is by induction over the class hierarchy (possible since the subclass ordering is well-founded) using the constraints on $R_{\texttt{Class}}$ and $R_{\texttt{Method}}$. Formally, we prove that

$$act_name\ R_{\texttt{Class_ref}}\,act_ref \wedge (dec_name, sig) R_{\texttt{Method_ref}} (dec_ref, m_tok)$$

implies that

$$
\begin{cases}
\texttt{lookup_name}(act_name, (dec_name, sig)) \\
\qquad\quad R_{\texttt{Class_ref}\times\texttt{Bytecode}} \\
\texttt{lookup_tok}(act_ref, (m_tok, dec_ref)).
\end{cases}
$$

The functions `lookup_name` and `lookup_tok` have similar structures, cf. Figs. 2 and 4. `lookup_name` takes one of three branches and we show that the conditions and the results are equivalent for `lookup_tok`. Either the method is *defined and visible* in the actual class, or *defined and not visible*, or *undefined*.

Suppose the method is defined and visible in the actual class, i.e., $methods_item(sig)$ is defined and the visibility condition holds. If the method token is public, then it must be that $m_tok \geqslant public_base$ and the offset is not `0xFFFF`. If the method token is package visible, then it must be greater than the package base, and the packages must be the same. In both cases, we return the actual class together with the code at that class.

Now, by $R_{\texttt{Class}}$ we have that there exists a token m'_tok such that

$$\langle act_name, sig\rangle R_{\texttt{Method_ref}} \langle act_ref, m'_tok\rangle.$$

We deduce that $(act_name, sig)\ R_{\texttt{Method_ref}}(act_ref, m_tok)$. Thus, again by $R_{\texttt{Class}}$, it must be that method_offset(act_ref, m_tok) is the entry in the method table computed by the lookup. Then, by $R_{\texttt{Pack_methods}}$, we get that the corresponding method information structures are related by $R_{\texttt{Method_info}}$, and in particular, the bytecodes are equivalent.

Suppose the method is defined but not visible. This must be for a package token then, and we have $method_tok \geqslant package_base$ and the same_package condition is false. In both formats then we look at the superclass, which is the same due to the definition of $R_{\texttt{Class}}$, and because the environments and actual class references are related. Equality follows from the inductive hypothesis at the superclass.

Finally, there is the case where the function is not defined at the actual class in the class format. If it is not defined in the token format either then both algorithms look in the superclass and we appeal to the inductive hypothesis at the superclass. By the second constraint on $R_{\texttt{Method_ref}}$, this must be because the token is less than the base. Otherwise, if the method is undefined in the name format, but defined (and visible) in the token format, this must be because the method was copied from a superclass (and the token is greater than the base). We can then use the inductive hypothesis at this superclass, as in the previous case. This tells us that the results are equal at the superclass. Thus, by definition of lookup_name and the overriding constraint on $R_{\texttt{Method_ref}}$, we have that the results are equal in the current class. □

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

$$[\![code]\!](\langle env, heap, op_stack, loc_vars, c_ref\rangle)$$

as the resulting state from the (unique) transition from $\langle code, op_stack, loc_vars\rangle$ with environment env and heap $heap$. Thus, we regard interpreted bytecode as having the type

$$\texttt{State} \to \texttt{Bytecode} \times \texttt{State}$$

where

$$\begin{aligned}
&\texttt{State} = \texttt{Global_state} \times \texttt{Local_state}\\
&\texttt{Global_state} = \texttt{Environment} \times \texttt{Heap}\\
&\texttt{Local_state} = \texttt{Operand_stack} \times \texttt{Local_variables} \times \texttt{Class_ref}\\
&\texttt{Operand_stack} = \texttt{Word}^*\\
&\texttt{Local_variables} = \texttt{Nat} \to \texttt{Word}
\end{aligned}$$

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 → Word) so these are identical under the two formats.

Theorem 10.2. *Assume that*
(1) $env_{name}\ R_{\texttt{Environment}}\ env_{tok}$,
(2) $heap_{name}\ R_{\texttt{Heap}}\ heap_{tok}$,
(3) $ls\ R_{\texttt{Local_state}}\ ls'$, *and*
(4) $code\ R_{\texttt{Bytecode}}\ code'$.
Then

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

Proof. It is straightforward to show that the representation independence of instructions follows from that of the auxiliary functions. Most of the work was in formulating the operational semantics so as to be independent of the underlying format. For the case of `invokevirtual` (see Section 6) suppose

$$heap_{name}\ R_{\texttt{Heap}}\ heap_{tok},\ env_{name} R_{\texttt{Environment}}\ env_{tok},\ m_{name} R_{Method}\ m_{tok}.$$

Then, from the representation independence (Lemma 10.1) of the function `constant_pool` for accessing the constant pool, we have

$$dec_mref_{name}\ R_{\texttt{Method_ref}}\ dec_mref_{tok}.$$

By the assumption on *heap* we have

$$act_cref_{name}\ R_{\texttt{Class_ref}}\ act_cref_{tok}.$$

From Lemma 10.1 applied to the lookup function we get that

$$m_class_{name}\ R_{\texttt{Class_ref}}\ m_class_{tok} \quad \text{and} \quad m_code_{name}\ R_{\texttt{Bytecode}}\ m_code_{tok}.$$

Since the heap and environment do not change, we can conclude that `invokevirtual` is representation independent. The cases of the other instructions are proven similarly. □

11. 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, in particular, 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. In addition to these problems, we have made a number of simplifications which could be relaxed. It would also be easy to incorporate AID's and so make package tokens internal. Another extension would be to incorporate the export files and descriptor component. One detail that is important for the development of the algorithm is the domains of the various functions and relations. We have not been too precise about the domains of the partial functions, and have used a notion of relational bijection accordingly. Currently, the relations are between *all* names and an infinite set of tokens but, in reality, we should use the *actual* names.

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, more realistically, we should account for the difference in word size. This has been considered in [5]. Although it seems that 'conversion' and 'optimisation', to borrow their terminology, are orthogonal, it would, nevertheless, be interesting to extend our formalisation to include these aspects. The Java Card set of byte codes contains a number of instructions that operate on one particular type (e.g. there is a `getfiled_s` for fetching a value from a field of type `short`). The specialisation of instructions could be handled by our technique (suitably combined with a type analysis), however, the extension is less clear for the more non-local optimisations.

We emphasised 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. For example, if the bytecode is not assumed to be verified, the lookup function could return `NoSuchMethodError` or `IllegalAccessError`.

These definitions have been formalised in Coq [1], and the lemmas verified [13]. 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. It is worth reflecting on the fact that Sun presents their specification as a formal definition of Java Card, which we have 'formalised' here, and then used as the basis of a formalisation in Coq! 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. For extraction, we prove (roughly speaking)

$$\forall\theta.\forall x : \theta_{name}.\exists y : \theta_{tok}.\, xR_\theta y$$

and from the constructive proof of this proposition, we then extract the program that converts from name to token format. The constructive essence of an extraction proof lies in the definition of $R_{\texttt{Package}}$. There are at least two possibilities for how the theorem could be proven for $\theta = \texttt{Package}$. One possibility is to first prove it for $\theta = \texttt{Class}$ and use this to construct the elements of $\texttt{Package}_{\texttt{tok}}$. This is necessary if dependencies between packages are such that the transformation must be carried out on a class by class basis. However, if the dependencies are ensured to give a tree-structure we can prove the `Package` case directly. This would correspond to a package by package transformation. Either way, the structure of the induction mirrors the overall structure of the algorithm.

Acknowledgements

Thanks to Tommy Thorn and Gaëlle Segouat for the formalisation in Coq and to Pascal Fradet for all the discussions pertaining to the tokenisation and its proof.

References

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliâtre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, B. Werner, The Coq Proof Assistant Reference Manual—Version V6.1, Tech. Report 0203, INRIA, August 1997.
[2] P. Bertelsen, Semantics of Java byte code, Tech. Report, Department of Information Technology, Technical University of Denmark, March 1997.
[3] E. Börger, W. Schulte, Defining the Java virtual machine as platform for provably correct Java compilation, in: L. Brim, J. Grunska, J. Zlatusla (Eds.), Proc. 23rd Internat. Symp. on Mathematical Foundations of Computer Science, Lecture Notes in Computer Sciences, Vol. 1450, Springer, Berlin, 1998.
[4] E. Denney, Correctness of Java Card Tokenisation, Tech. Report 1286, Project Lande, IRISA, 1999. Also appears as INRIA research report 3831.
[5] J.-L. Lanet, A. Requet, Formal proof of smart card applets correctness, Third Smart Card Research and Advanced Conf. (CARDIS'98), 1998.
[6] T. Lindholm, F. Yelling, The Java Virtual Machine Specification, Addison-Wesley, Reading, MA, 1997.
[7] J. Mitchell, Foundations for Programming Languages, Foundations of Computing Series, MIT Press, Cambridge, MA, 1996.
[8] D. Mosses, Modularity in structural operational sematics, Extended abstract, November 1998.
[9] C. Pusch, Verification of Compiler Correctness for the WAM, in: J. von Wright, J. Grundy, J. Harrison (Eds.), Theorem Proving in Higher Order Logics (TPHOLs'96), Springer, Berlin, 1996, pp. 347–362.

[10] C. Pusch, Formalizing the Java Virtual Machine in Isabelle/HOL, Tech. Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.
[11] J.C. Reynolds, Types, abstraction and parametric polymorphism, Information Processing 83, North-Holland, Amsterdam, 1983.
[12] G. Schellhorn, Verification of abstract state machines, Ph.D. Thesis, University of Ulm, 1999.
[13] G. Segouat, Preuve en Coq d'une mise en oeuvre de Java Card, Master's Thesis, Project Lande, IRISA, 1999.
[14] Sun Microsystems, Java Card 2.0 Language Subset and Virtual Machine Specification, October 1997, Final Revision.
[15] Sun Microsystems, Java Card 2.1 Virtual Machine Specification, March 1999, Final Revision 1.0.