Correctness of Java Card Tokenisation

Ewen Denney

N°3831
Décembre 1999

THÈME 2
Correctness of Java Card Tokenisation

Ewen Denney

Thème 2 — Génie logiciel et calcul symbolique  
Projet Lande

Rapport de recherche n°3831 — Décembre 1999 — 34 pages

Abstract: We present 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 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.

Key-words: Java Card, bytecode, optimisation, logical relations, data refinement
La correction de la tokenisation de Java Card

Résumé : Nous présentons une formalisation de l'optimisation de bytecode du langage Java Card de Sun qui est réalisé par une transformation du format “class file” au format “CAP file”. La formalisation s'exprime comme un ensemble de contraintes entre les deux formats. Le bytecode de Java Card est formalié comme une sémantique opérationnelle abstraite, qui peut alors être instanciées dans les deux formats. L'optimisation est donnée comme une relation logique telle que les sémantiques instanciées sont observablement égales.

Mots-clés : Java Card, bytecode, optimisation, relations logiques, raffinement de données
The Java Card language [Sun97] 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 [LY97], 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 [Sun99]. The core of this optimisation is a tokenisation in which names are replaced with tokens enabling a more direct 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 report concentrates on proving the correctness of the specification. The formal development of an algorithm will be 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 [Sun99] (given certain assumptions) are sufficient. Moreover, the technique used to develop an algorithm is orthogonal to this proof.

1 The Conversion

We give a brief sketch of the transformation process. We assume that the reader has a basic understanding of the various elements of the Java Virtual Machine and class file format.

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 fact, the conversion process also takes a number of export files as input, but we will ignore these here. Indeed, this is just one of several simplifying assumptions we make.

The ‘transformation’ is presented in [Sun99] as a collection of constraints on the CAP file, rather than as an explicit correspondence between class and CAP formats. Instead, we adopt a simplified definition of the transformation, only considering classes, constant pools, fields and methods. In particular, we ignore exceptions and interfaces.

In the class file format, methods, fields and so on are referred to using a certain naming convention. In CAP files, instead, tokens are ascribed to the various entities. The idea is that if a method, say, is publicly visible1, then it is ascribed a token. If the method is only visible within its package, then it is referred 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).

CAP files consist of a number of components, of which we will consider the constant pool, class, method, static field and descriptor components. 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 tokens to offsets into the method component. The method tables contain the information necessary for resolving any method call in that class. If a class inherits a method from a superclass then it may be that the method token is included in the relevant table, or 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.

Another optimisation concerns method references. These are tagged to indicate whether they correspond to the call of a ‘supermethod’, that is, the method of a superclass. This comes from using the super keyword in the source code (thus avoiding overriding). Retaining this information in the bytecode allows a more efficient location of the information in the tables.

What we have described are those aspects concerned with the rearrangement into CAP format. There are also a number of mandatory optimisations such as the inlining of final fields, and the type-based specialisation of instructions. The order of these stages is not specified. Indeed, a converter is at liberty to implement further optimisations.

Most of the work in the proof lies in the various definitions: defining the semantics of the virtual machine independently of the underlying format, and formalising the main stages of the transformation. Given this

1We follow the terminology of [Sun99], 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.
framework, the proof, as such, is relatively small. It is natural when proving correctness to consider all the transformation steps simultaneously. The modularity is in the definition. Thus, by “Java card tokenisation” we mean both the assignment of tokens to the various named items, and the rearrangement in the CAP file format. We also take the compression of method tables into account.

There is a more detailed discussion of the differences between Java and Java Card in the official documentation [Sun97, Sun99].

2 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 [Ber97] gives an operational semantics which we have used as a starting point. He also considers the verification conditions, which considerably complicates the rules, however. Börger and Schulte [BS98] have a different approach, though also make use of auxiliary functions to a certain extent. Pusch has formalised the JVM in HOL [Pus98]. 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 [LR98] 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, this is an aspect which we have ignored completely but have, however, specified the conversion. Their formalism is based on the B method, so the specification and proof are presented as a series of refinements.

In [Pus96], 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 states. 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.

3 Overview of Formalisation

We will present the transformation from class file to CAP file as a transformation of virtual machines, that is, from the JVM to the JCVM. Since Java Card is a sublanguage of Java it can be executed on a JVM, although the intention is that the conversion is an integral part of the compilation process, and that only the CAP file is executed.

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, for example, 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.

We adopt a simple form of equivalence based on the notion of representation independence [Mit96]. 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.

Representation independence may be proven by defining coupling relations between the two formats, which respect the tokenisation and are the identity at observable types. This can be seen as formalising a data refinement from class to CAP files.

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. 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.

We follow numerous researchers in this area and formalise the virtual machines in an operational style, as transition relations over abstract machines. We adopt the action semantics formalism of Mosses [Mos98].
This is convenient as by presenting the bytecode semantics in 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. However, 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.

The semantics is given in a mixture of operational and denotational styles. We formalise the JCVM operationally, parameterised with respect to a number of auxiliary functions which are then interpreted denotationally.

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 rules giving the semantics of the method invocation instructions, presented in Section 5.1, are parameterised with respect to the lookup function. Then in Sections 6 and 7 two possible interpretations of lookup (and the other auxiliary functions) are given. A further choice would be to give an abstract interpretation to the auxiliary functions or, going in the opposite direction, to include error information. For example, if the bytecode is not assumed to be verified, the lookup function could return NoSuchMethodError or IllegalAccessException.

Although the equivalence of dynamic method lookup could be regarded as the aim of the proof, in fact ‘correctness’ is distributed throughout all of the transformation and equivalence of lookup uses equivalence of the transformation of classes, and so on.

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 this spirit, we will use ‘name interpretation’ and ‘token interpretation’ to refer to the semantics of the JVM and JCVM respectively.

Hence we consider the tokenisation to be correct if we can give a family of coupling relations which respects the tokenisation, and which is the identity on observable types. We set this as the goal of the proof. Here we will just be concerned with the correctness of the tokenisation and conversion to CAP format, and not any inlining or further optimisation that might take place.

We give types to the various entities converted during tokenisation, such as Class_ref and Constant_pool. We include a type, Bytecode, since the bytecode itself changes during tokenisation. This is due, amongst other reasons, to the presence of constant pool indices as arguments to instructions.

Let us write $\llbracket \cdot \rrbracket_{\text{name}}$ for the name interpretation (class format), and $\llbracket \cdot \rrbracket_{\text{tok}}$ for the token interpretation (CAP format). We interpret both types and auxiliary functions. For example

$$
\llbracket \text{Method} \rrbracket_{\text{name}} = \text{Class} \times \text{Method} \times \text{Type}
$$

$$
\llbracket \text{Method} \rrbracket_{\text{tok}} = \text{Class} \times \text{Method} \times \text{Token}
$$

The lookup function

$$
\text{lookup} : \text{Class} \times \text{Method} \rightarrow \text{Class} \times \text{Bytecode}
$$

is interpreted, in turn, as the two functions defined below. Then for each type, $\theta$, we define a relation $R_\theta \subseteq \llbracket \theta \rrbracket_{\text{name}} \times \llbracket \theta \rrbracket_{\text{tok}}$.

One technical problem is that some types are most naturally considered as being local to a certain context, or dependent on another type. For example, class references in the token interpretation can either be external to a package or internal. In the second case, they are given as an offset which does not make sense out of the package. Thus $R_{\text{Class}\_ref}$ must relate class names to both internal and external token references. Another example is that constant pool indices in the bytecode are assumed to have a label indicating the relevant constant pool (whereas, in reality, this is evident from the context.) We can get round this by assuming that data is paired with something to indicate its context whenever necessary. A more elegant approach would use dependent types.

We start from the observation that not all instructions use the heap or the environment. It is these which are sensitive to the alterations in the layout of the constant pool and class hierarchy, which take place during tokenisation. Thus we will ignore the instructions concerned with immediate operations, stack manipulation, local variables, and branching.  

In fact, we will not consider the array access instructions either, one of which (aastore) accesses the environment, but we could easily make this extension.

RR n° 3831
Secondly, since the bytecode is assumed to have been verified (and is not self-modifying), we can regard it as having been assembled into an abstract syntax. This simplifies the presentation and lets us abstract away from details of program counters and use a form of structural operational semantics. We will not consider jump instructions here, so need only consider one instruction at a time.\(^3\)

The operational semantics is given in terms of configurations of abstract syntax and labelled arrows. We regard configurations themselves as a form of instruction; in effect, a closure. Any changes to the heap or environment are given implicitly in the arrow rather than being explicitly part of the configuration. This affords some modularity for then the semantics of those instructions which do not use the heap or environment can be given as such and then extended verbatim.

In Section 4, we define abstract types which are common to the two formats. 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, so define auxiliary functions for accessing such data.

In Sections 6 and 7, we give the specific details of the name and token formats, respectively, defined as interpretations of types and auxiliary functions, \([\_]_{\text{name}}\) and \([\_]_{\text{tok}}\).

In Section 8, we define the logical relation, \{\(R_\theta\)\}_{\theta \in \text{Abstract_type}}. It is convenient to informally group the definition into several levels. First of all, there are various basic types (byte, short, etc.), \(\gamma\), for which we have \(R_\gamma = id_\gamma\). Then there are the references, \(\iota\), such as package and class references, for which the relation \(R_\iota\) represents the tokenisation of named items.

The constraints on the componentisation are expressed in \(R_\kappa\), where \(\kappa\) 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 \(R_\theta\) for each type, \(\theta\), where

\[
\theta ::= \gamma \mid \iota \mid \kappa \mid \theta \times \theta' \mid \theta \rightarrow \theta' \mid \theta + \theta' \mid \theta^\prime
\]

The family of relations, \{\(R_\theta\)\}_{\theta \in \text{Type}}\), represents the overall construction of components in the CAP file format from a class file. The relations are ‘logical’ in sense that the definitions for defined types follows automatically. For example, we define the type of the environment and heap as

\[
\text{Environment} = \text{Package_ref} \rightarrow \text{Package}
\]

\[
\text{Heap} = \text{Object_ref} \rightarrow \text{Object}
\]

and so the definition of \(R_{\text{Environment}}\) follows from those of \(R_{\text{Package_ref}}, R_{\text{Package}}\) and the (standard\(^4\)) construction of \(R_{\text{ref}}\); similarly for \(R_{\text{Heap}}\). In Section 9, we use this semantic format to prove the correctness. The proof has two parts:

1. We prove that all auxiliary functions are representation independent; that is, if \(f : \theta \rightarrow \theta'\) then we have \([f]_{\text{name}} \circ R_\theta \circ [f]_{\text{tok}}\).

2. Then, it is straightforward to prove that all instructions are representation independent, using part 1. It is convenient to view the operational semantics of bytecode as giving an interpretation

\[
[code] : \text{State} \rightarrow \text{Bytecode} \times \text{State}
\]

where

\[
\text{State} = \text{Global_state} \times \text{Local_state}
\]

\[
\text{Global_state} = \text{Environment} \times \text{Heap}
\]

\[
\text{Local_state} = \text{Operand_stack} \times \text{Local_variables} \times \text{Class_ref}
\]

We can conclude, therefore, 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.

Finally, we make some concluding remarks in Section 10.

\(^3\)Actually, jumps should not present a problem. We could add labels to the abstract syntax, and keep the code as a component of the global state. An auxiliary function would search for the label and evaluation would proceed from that point onwards.

\(^4\)It is simpler to allow functions to be partial.
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. We write elements of sum types in the form \( \langle \text{tag}, \text{value} \rangle \). Occasionally we use terms as singleton types, such as \( 0xFFFF \) and \( 0 \). We use a set-theoretic notation where convenient.

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, \( \text{CP\_index} \) 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., \( \text{Class\_name} \times \text{Index} \). 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 \( \text{Package\_ref} \to \text{Package} \), with \( \text{Package} \) interpreted as \( \text{Class\_name} \to \text{Class\_file} \) or \( \text{CAP\_file} \).

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

\[
\text{Type} = \{\text{Boolean, Byte, Short}\} + \text{Reference\_type}
\]

We have not included Int, which is optional.

\[
\text{Reference\_type} = \text{Array\_type} + \text{Class\_ref}
\]

We use a separate type, \( \text{Object\_ref} \), 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 a platform specific abstract unit of storage. 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

\[
\text{Word} = \text{Object\_ref} + \text{Null} + \text{Boolean} + \text{Byte} + \text{Short}
\]

Thus a word is (i.e. represents) either a reference (possibly null) or an element of a primitive type. Furthermore, we define

\[
\text{Value} = \text{Word}
\]

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\(^6\):

\[
\text{Package\_ref} | \text{Class\_ref} | \text{Field\_ref} | \text{Method\_ref} | \text{Interface\_method\_ref}
\]

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. 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. By contrast, an identifier is a name or a token. We do not use identifiers at the abstract level though. There is not a specific form of reference for interfaces. These are taken to be a particular form of class reference.

\[
\text{CP\_info} = \text{Class\_ref} + \text{Method\_ref} + \text{Field\_ref}
\]

Since, in Java, only constants of ‘big’ type can appear in the constant pool, these are not present in Java Card. Moreover, since we consider the constant pool to be fully resolved, we do not need separate entries for names.

---

\(^6\)Which we distinguish from Reference\_type

RR n° 3831
The index type, CP_index, expresses the rearrangement of the constant pools. We introduce a separate index type, SM_index, for the method references used by the invokespecial instruction. These are treated differently since the corresponding entries are replaced with supermethod references during conversion.

There is a type for method information structures

\[ \text{Method\_info} \]

Since types are coded differently in the two formats we introduce an abstract type

\[ \text{Type\_code} \]

We follow Bertelsen [Ber97] in considering the constant pool to be partially resolved. For example, rather than taking a method reference in the class file format to be a tag (which can be ignored) plus a class index (the reference at which, in turn, contains an index to a string constant) and a signature (that is, a name-and-type index, which also contains indices), we just represent it as a tuple of class name and descriptor.

Finally, we use types for grouping parts of the transformation.

\[ \text{Package} \mid \text{Class} \mid \text{Pack\_methods} \mid \text{Pack\_fields} \mid \text{Constant\_pool} \]

Using these basic types, we can then construct complex types using the usual type constructors. We will use (non-dependent) sum, product, function and list types (denoted \( \sum^\oplus \)). We do not use lists much since it is simpler to represent tables as functions with a domain of indices rather than lists.

The virtual machines are formalised as follows:

\[
\begin{align*}
\text{Locals} &= \text{Nat} \to \text{Word} \\
\text{Config} &= \text{Bytecode} \times \text{Word}^\ast \times \text{Locals} \times \text{Class\_ref} \\
\text{Bytecode} &= \text{Instruction} + (\text{Bytecode} \times \text{Bytecode}) + \text{Config} \\
\text{Instruction} &= \text{Nop} + \text{Invokevirtual} \, \text{CP\_index} + \text{Checkcast} \, \text{Typecode} + \text{Invokespecial} \, \text{SM\_index} \cdots
\end{align*}
\]

We treat the instructions as types, and give full details in the next section. Since bytecode contains indices into the constant pool, it changes during tokenisation. We could model this in a number of ways. One possibility would be to treat the indices themselves as auxiliary functions (Börger and Schulte [BS98] do something similar to this), although this is essentially equivalent to the approach we have taken. The essential point is that Bytecode depends on CP_index and, indeed, other types.

The tokenisation requires us to make various distinctions, such as between static and instance fields, which are not needed for some auxiliary functions.

There are other constructed types, which we need not be directly concerned with. For example:

\[ \text{Class\_inst\_obj} = \text{Class\_ref} \times \text{IV} \]

In Java, array types contain the array dimension, whereas array objects contain the length. Since arrays are unidimensional in Java Card, the dimension is unnecessary.

\[
\begin{align*}
\text{Array\_obj} &= \text{Nat} \times \text{Array\_type} \times (\text{Nat} \to \text{Value}) \\
\text{Array\_type} &= \text{Primitive\_type} + \text{Class\_ref} \\
\text{Object} &= \text{Class\_inst\_obj} \times \text{Array\_obj} \\
\text{IV} &= \text{Field\_ref} \to \text{Value}
\end{align*}
\]

We will assume that the classes are grouped correctly according to their package.

\[
\begin{align*}
\text{Environment} &= \text{Package\_ref} \to \text{Package} \\
\text{Heap} &= \text{Object\_ref} \to \text{Object}
\end{align*}
\]
5 Operational Semantics

The official specification of the JCVM (and JVM) is given in terms of frames. This is 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 will 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. 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 model local variables as a partial function, but represent this as a list.

To account for method invocations, it is convenient to allow a configuration itself to be considered as an instruction. When a method is invoked, the next instruction becomes a current 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. Since execution does not terminate, as such, we introduce an artificial instruction \( \text{nop} \) to signify the termination of an instruction. The following two rules are standard for SOS:

\[
\begin{align*}
\langle b_1, \text{ops}, l, c \rangle &\Rightarrow \langle b'_1, \text{ops}', l', c' \rangle \\
\langle b_1, b_2, \text{ops}, l, c \rangle &\Rightarrow \langle b'_1, b_2, \text{ops}', l', c' \rangle \\
\langle b_1, \text{ops}, l, c \rangle &\Rightarrow \langle \text{nop}, \text{ops}', l', c' \rangle \\
\langle b_1, b_2, \text{ops}, l, c \rangle &\Rightarrow \langle b_2, \text{ops}', l', c' \rangle
\end{align*}
\]

For configurations we use the rule:

\[
\frac{f \Rightarrow f'}{(\text{Config } f, \text{ops}, l, c) \Rightarrow (\text{Config } f', \text{ops}, l, c)}
\]

We will write \( \text{Config}(b, o, l, c) \) as \( \langle b, o, l, c \rangle \).

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.

Thus we define a transition relation

\[
\Rightarrow \subseteq \text{Config} \times \text{Arrow} \times \text{Config}
\]

where

\[
\begin{align*}
\text{Config} &= \text{Bytecode} \times \text{Word}^+ \times \text{Locals} \times \text{Class}\_\text{ref} \\
\text{Arrow} &= \text{Global}\_\text{state} \rightarrow \text{Global}\_\text{state} \\
\text{Global}\_\text{state} &= \text{Environment} \times \text{Heap} \\
\text{Bytecode} &= \text{Instruction} + (\text{Bytecode} \times \text{Bytecode}) + \text{Config} \\
\text{Instruction} &= \text{Nop} + \text{Invokevirtual}\_\text{CP}\_\text{index} + \text{Invokestatic}\_\text{CP}\_\text{index} + \\
&+ \text{Invokeinterface}\_\text{CP}\_\text{index} + \text{Invokespecial}\_\text{SM}\_\text{index} + \text{Return} + \\
&+ \text{New}\_\text{CP} + \text{Putstatic}\_\text{CP}\_\text{index} + \text{Getstatic}\_\text{CP}\_\text{index} + \\
&+ \text{Putfield}\_\text{CP}\_\text{index} + \text{Getfield}\_\text{CP}\_\text{index} + \text{Checkcast}\_\text{Typedef} + \\
&+ \text{Instanceof}\_\text{Typedef}
\end{align*}
\]

The rules are given in the following form.

\[
\begin{align*}
\text{hypothesis 1} \\
\vdots \\
\text{hypothesis } n
\end{align*}
\]

\[
\text{config} \xrightarrow{\text{state change}} \text{config}
\]

Hypotheses are either conditions or assignments. In fact, almost all hypotheses here are assignments. The only conditions used are in invokespecial, where the predicate \( \text{super}\_\text{invocation} \) is used to choose between two behaviours, and in checkcast, where an exception can be raised. We make liberal use of wildcards ‘\( \_ \)’ in assignments to suppress unimportant details.

RR n° 3831
When the heap and environment do not change, we will not bother to write the label explicitly. Since at most one transition can be a hypothesis, we adopt the convention that, unless stated otherwise, the label on the arrow of the conclusion is the same as that on this hypothesis. Moreover, the arrow on a transition with no transitions for hypotheses is the identity, $id_{heap,env}$. We can implicitly use $heap$ and $env$ in the hypotheses.

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

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

We factor out those tedious parts of the semantics which are common to most instructions into a number of auxiliary selector functions. We adopt the convention of using capitalised names for types, and lower case names for the corresponding selector functions. So, we have:

$$\text{constant_pool}: \text{Class_ref} \to \text{Constant_pool}$$

where $\text{Constant_pool}$ is the abstract type of the constant pool.

We give the semantics of those instructions which make use of the constant pool and class hierarchy, namely:\n
- the various method invocation instructions,
- instructions for instantiating and accessing classes, and
- the two type checking instructions.

By virtue of the verification phase, we can assume that various static checks have been carried out. We use a number of auxiliary functions. Certain functions have preconditions, which we take as concomitant with the well-formedness of the class file. Some functions use the environment and the heap. Rather than pass these as explicit arguments we will assume them to be globally accessible.

The $lookup$ function 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. We assume that the declared and actual class are in the same class hierarchy.

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

We use a separate function $lookup\_int$ for looking up interface methods, since they are treated differently.

$$\text{lookup\_int}: \text{Class_ref} \times \text{Interface_method_ref} \to \text{Class_ref} \times \text{Bytecode}$$

$\text{instance\_fields}(c)$ returns the default values for the instance fields in $c$.

$$\text{instance\_fields}: \text{Class_ref} \to (\text{Field_ref} \to \text{Value})$$

$\text{static\_val}(f)$ returns the static value in field $f$.

$$\text{static\_val}: \text{Field_ref} \to \text{Value}$$

$\text{instOf}(r,o)$ returns a boolean corresponding to whether the object $o$ can be cast to reference type $r$.

$$\text{instOf}: \text{Reference\_type} \times \text{Object} \to \text{Bool}$$

The functions $add$ and $update$ are used in formalising the operational semantics. $add(r,o)$ puts the new binding $r \mapsto o$ on the heap, and $update(f,v)$ changes the binding of static field $f$ to value $v$. We can define $add: \text{Object_ref} \times \text{Object} \to \text{Statechange}$ independently of the underlying format, so do not regard it as an auxiliary function. In contrast, since we assume that the values of static fields are stored in the class/CAP file (though this is not clear), the $update$ function changes the environment and so depends on the format.

$$\text{update}: \text{Field_ref} \times \text{Value} \to \text{Statechange}$$

where

$$\text{Statechange} = \text{Global\_state} \to \text{Global\_state}$$

\(^a\)With the exception of \text{astore}. We also ignore \text{invokevirtual} on array objects, and native methods.
Global_state = Environment × Heap

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 or class, such as the appropriate constant pool.

\[ \text{super}(c) \text{ returns a reference to the superclass of } c. \]

\[ \text{super} : \text{Class_ref} \rightarrow \text{Class_ref} \]

The function super_invocation is used in the semantics of the invokespecial instruction, and returns a boolean corresponding to whether or not an invocation should be of a supermethod.

\[ \text{super_invocation} : \text{Class_ref} \times \text{Method_ref} \rightarrow \text{Bool} \]

The functions method_class, method_code and method_nargs return the class, code and number of arguments for a given method reference.

\[ \text{method_class} : \text{Method_ref} \rightarrow \text{Class_ref} \]

We only use method_class for static (and super) method references.

\[ \text{method_code} : \text{Method_ref} \rightarrow \text{Bytecode} \]

\[ \text{method_nargs} : \text{Method_ref} \rightarrow \text{Nat} \]

The function method_code assumes that the appropriate method is defined at the given class and does not do any searching. It is used in the semantics of the invokestatic and invokespecial instructions.

Finally, since reference types are represented differently in the two formats, we use a function reference_type to abstract away from the concrete format (expressed using a natural and/or an index into the constant pool) and return the actual type:

\[ \text{reference_type} : \text{Type_code} \times \text{Class_ref} \rightarrow \text{Reference_type} \]

5.1 Method Invocation

Note that whereas some of these operations have class loading aspects in Java, Java Card does not have dynamic class loading.

5.1.1 Invokevirtual

The procedure is:

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 \( \text{heap}(r) = \langle \text{act_cref}, \_ \rangle \), the actual class reference (fully qualified name or a package/class token pair).
5. We then do \( \text{lookup}(\text{act_cref}, \text{dec_mref}) \), 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 new configuration is created for this method and evaluation proceeds from there.

\[
\begin{align*}
\text{dec_mref} & := \text{constant_pool}(c)(i) & \text{get declared method reference from constant pool} \\
n & := \text{method_nargs}(\text{stat_mref}) & \text{get number of arguments} \\
\langle \text{act_cref}, \_ \rangle & := \text{heap}(r) & \text{get actual class reference from heap} \\
\langle \text{m_class}, \text{m_code} \rangle & := \text{lookup}(\text{act_cref}, \text{dec_mref}) & \text{look up method} \\
\langle \text{invokevirtual}, i, a_1 \ldots a_n :: r :: s, l, c \rangle & \Rightarrow \langle \langle \text{m_code}, \_ \rangle, a_1 \ldots a_n :: r, \text{m_class}, s, l, c \rangle
\end{align*}
\]

RR n° 3831
5.1.2 Invokestatic

Invocation of class (static) methods.

\[
\begin{align*}
\text{dec}_m \text{ref} & := \text{constant}_p ool(c)(i) & \text{get declared method reference from constant pool} \\
n & := \text{method}_n arg(s(stat_mref)) & \text{get number of arguments} \\
m_\text{class} & := \text{method}_\text{class}(\text{dec}_m\text{ref}) & \text{get class of method} \\
m_\text{code} & := \text{method}_\text{code}(\text{dec}_m\text{ref}) & \text{get method code} \\
\langle \text{invokestatic} \ t, a_1, \ldots, a_n \ :: \ s, l, c \rangle & \Rightarrow \langle (m_\text{code}, \{}), a_1, \ldots, a_n, m_\text{class}, s, l, c \rangle
\end{align*}
\]

5.1.3 Invokeinterface

By virtue of the verification phase, we can assume that various static checks have been carried out and that, for example, the resolved method is not an initialization method (<init> or <c1init>), that \text{act cref} implements the interface, and that there are \(n - 1\) arguments on the operand stack. According to [LY97], the \(n\) is a historical redundancy. A run-time exception can be thrown if the object reference \(r\) is null.

\[
\begin{align*}
\text{int}_m \text{ref} & := \text{constant}_p ool(c)(i) & \text{get declared method reference in constant pool} \\
n & := \text{method}_n arg(s(int_mref)) & \text{get number of arguments} \\
\langle \text{act cref}, \_ \rangle & := \text{heap}(r) & \text{get class name from heap} \\
\langle m_\text{class}, m_\text{code} \rangle & := \text{lookup}_\text{int}(\text{act cref}, int_mref) & \text{look up interface method} \\
\langle \text{invokeinterface} \ t, a_1, \ldots, a_n \ :: \ r \ :: \ s, l, c \rangle & \Rightarrow \langle (m_\text{code}, \{}), a_1, \ldots, a_n, m_\text{class}, s, l, c \rangle
\end{align*}
\]

5.1.4 Invokespecial

The \text{invokespecial} instruction has two behaviours, depending on the modifiers of the actual class, and the kind of method invoked. Either the superclass of the actual class is searched, or a method of the actual class itself is used, which is allowed to be private or an initialization method. The reasoning is described in [LY97] as:

\[
\begin{align*}
\text{if} & \not<\text{init}> \land \not\text{private} \land (\text{actual}_\text{class} < \text{method}_\text{class}) \land \text{super}(\text{actual}) \\
\text{then} & \ \text{super}_\text{method} \\
\text{else} & <\text{init}> \land \text{uninitialised}(\text{object}_\text{ref}) \\
\text{then} & \ \text{init}_\text{method} \\
\text{else} & \text{*/may be private */} \\
\text{actual}_\text{class}_\text{method}
\end{align*}
\]

An analysis is carried out to determine which of these cases holds. Although this might be carried out at run-time in the class format, in the CAP format it is analysed statically as part of the conversion process, so that the method reference indicates explicitly whether or not it is a super invocation. However, we abstract this test out into a function \text{super invocation}.

Hence it is unnecessary to get the class name from the heap at runtime. A static dataflow analysis during verification ensures that methods are always initialized before use.

For \text{invokespecial} \(i\), on a ‘superclass’ instance method. Note that we call the lookup with the superclass of the actual class:

\[
\begin{align*}
\text{dec}_m \text{ref} & := \text{constant}_p ool(c)(i) & \text{get declared method reference in constant pool} \\
n & := \text{method}_n arg(s(dec_mref)) & \text{get number of arguments} \\
\text{super}_\text{invocation}(c, dec_mref) & & \text{check for super invocation} \\
\langle \text{super}_m \text{ref}, \text{m}_\text{code} \rangle & := \text{super}_\text{of}(\text{method}_\text{class}(\text{dec}_m\text{ref})) & \text{get superclass of method call} \\
\langle m_\text{code}, \_ \rangle & := \text{lookup}(\text{super}_m \text{ref}, \text{dec}_m\text{ref}) & \text{look up method from superclass} \\
\langle \text{invokespecial} \ t, a_1, \ldots, a_n \ :: \ r \ :: \ s, l, c \rangle & \Rightarrow \langle (m_\text{code}, \{}), a_1, \ldots, a_n, m_\text{cref}, s, l, c \rangle
\end{align*}
\]
For `invokespecial i` on instance initialization methods, and methods of the actual class (which may be private), there is no lookup:

\[
\begin{align*}
\text{dec}_\text{mref} & := \text{constant}_\text{pool}(c)(i) \quad \text{get declared method reference in constant pool} \\
n & := \text{method_nargs}(\text{dec}_\text{mref}) \\
\neg \text{super_invocation}(c, \text{dec}_\text{mref}) & \quad \text{check not super invocation} \\
m\text{class} & := \text{method_class}(\text{dec}_\text{mref}) \quad \text{get class of method} \\
m\text{code} & := \text{method_code}(\text{dec}_\text{mref}) \\
\langle \text{invokespecial } i, a_1, \ldots, a_n : s, l, c \rangle & \Rightarrow \langle m\text{code}, \langle\rangle, a_1, \ldots, a_n, m\text{class}, s, l, c \rangle
\end{align*}
\]

### 5.1.5 Method Return

The method return instructions do not use the configuration of the current method.

\[
\langle \langle \text{return}_\ldots \ldots \rangle, \text{ops}, l, c \rangle \Rightarrow \langle \text{nop}, \text{ops}, l, c \rangle
\]

### 5.2 Class Instructions

As mentioned above, Java Card does not support dynamic class loading so these operations all assume that the appropriate classes are loaded.

#### 5.2.1 Class Instantiation

We can assume that the class is not abstract or an interface, that the class is accessible, and that the operand stack will not overflow.

The function `instance_fields` returns a mapping which gives the default field values of a class:

\[
\begin{align*}
\text{instance_fields} : & \text{Class}\_\text{ref} \rightarrow (\text{Field}\_\text{ref} \rightarrow \text{Value}) \\
\text{static_val} : & \text{Field}\_\text{ref} \rightarrow \text{Value}
\end{align*}
\]

Recall that we define `add(r, o)`, which adds the binding of reference `r` to object `o` to the heap, as:

\[
\text{add}(r, o) = \lambda(e, h) : (e, h + \{r \mapsto o\})
\]

Since we can define `add` independently of format we do not regard it as an auxiliary function.

\[
\begin{align*}
c\_\text{ref} & := \text{constant}_\text{pool}(c)(i) \quad \text{get class reference} \\
\text{iv} & := \text{instance_fields}(c\_\text{ref}) \quad \text{compute default values} \\
o & := \langle c\_\text{ref}, \text{iv} \rangle \\
r & \in \text{Object}\_\text{ref}\_\text{dom}(\text{heap}) \quad \text{find new reference} \\
\langle \text{new } i, \text{ops}, l, c \rangle & \xrightarrow{\text{add}(r, o)} \langle \text{nop}, r : \text{ops}, l, c \rangle
\end{align*}
\]

We assume the existence of a deterministic choice function for selecting a new object reference. We do not define an auxiliary function, however, since we have not given a concrete implementation of the type `Object\_ref` in either model. This makes the proof of equivalence slightly easier though, strictly speaking, all that is necessary is that the transition rules are *observably* deterministic.

#### 5.2.2 Class Fields

We use 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. It is because of this that the environment can change.

\[
\begin{align*}
f\_\text{ref} & := \text{constant}_\text{pool}(c)(i) \quad \text{get field reference} \\
\langle \text{putstatic } i, v : \text{ops}, l, c \rangle & \xrightarrow{\text{update}(f\_\text{ref}, v)} \langle \text{nop}, \text{ops}, l, c \rangle \\
f\_\text{ref} & := \text{constant}_\text{pool}(c)(i) \quad \text{get field reference} \\
v & := \text{static_val}(f\_\text{ref}) \quad \text{get static value} \\
\langle \text{getstatic } i, \text{ops}, l, c \rangle & \Rightarrow \langle \text{nop}, v : \text{ops}, l, c \rangle
\end{align*}
\]

RR n° 3831
5.2.3 Instance Fields

The instructions for putting and getting the instance field of an object, \( r \), where the reference \( r \) is on the heap, assume that \( r \) is initialized.

\[
\begin{align*}
  f \_ref & := \text{constant\_pool} (c)(i) & \text{get field reference} \\
  \langle c' \_ref, iv \rangle & := \text{heap}(r) & \text{get object from heap} \\
  o & := \langle c' \_ref, iv + \{f \_ref \mapsto v\} \rangle & \text{update object} \\
  \langle \text{putfield} i, v :: ops, l, c \rangle & \mapsto (\text{nop}, ops, l, c)
\end{align*}
\]

\[
\begin{align*}
  f \_ref & := \text{constant\_pool} (c)(i) & \text{get field reference} \\
  \langle c' \_ref, iv \rangle & := \text{heap}(r) & \text{get object from heap} \\
  v & := iv(f \_ref) & \text{get field value} \\
  \langle \text{getfield} i, r :: ops, l, c \rangle & \Rightarrow (\text{nop}, v :: ops, l, c)
\end{align*}
\]

5.3 Type Checking

The \( \text{instOf} : \text{Reference\_type} \times \text{Object} \to \text{Bool} \) predicate formalises when an object can be cast to a type. The object reference, \( r \), is left on the stack. If the condition fails, the instruction throws a \text{CheckCastException}. We do not model this.

\[
\begin{align*}
  d & := \text{reference\_type}(tc, c) & \text{get type descriptor} \\
  r \neq \text{null} & \Rightarrow r \in \text{dom(heap)} \land \text{instOf}(d, \text{heap}(r)) & \text{check cast is valid} \\
  \langle \text{checkcast} tc, r :: s, l, c \rangle & \Rightarrow (\text{nop}, r :: s, l, c)
\end{align*}
\]

The \text{instanceof} instruction has similar semantics to \text{checkcast}, though does not throw an exception. Instead, the object reference on the operand stack is replaced with a bit representing the result of the check.

\[
\begin{align*}
  d & := \text{reference\_type}(tc, c) & \text{get type descriptor} \\
  b & := \text{if instOf}(d, \text{heap}(r)) \text{ then } 1 \text{ else } 0 & \text{compute ‘instance bit’} \\
  \langle \text{instanceof} tc, r :: s, l, c \rangle & \Rightarrow (\text{nop}, b :: s, l, c)
\end{align*}
\]

6 Name Interpretation

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 \text{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 detail of those parts used here. We group the class files by package into a global environment. Thus \( \text{env\_name}(p)(c) \) denotes the class file in package \( p \) with name \( c \).

6.1 Types

\[
\begin{align*}
  \text{[Package]}_{\text{name}} & = \text{Class\_name} \to \text{Class\_file} \\
  \text{[Package\_ref]}_{\text{name}} & = \text{Package\_name} \\
  \text{[Class\_ref]}_{\text{name}} & = \text{Class\_name} \\
  \text{[Field\_ref]}_{\text{name}} & = \text{Class\_name} \times \text{Field\_name} \\
  \text{[Method\_ref]}_{\text{name}} & = \text{Class\_name} \times \text{Sig} \\
  \text{Sig} & = \text{Method\_name} \times \text{[Type]}_{\text{name}}^* \\
  \text{[Interface\_method\_ref]}_{\text{name}} & = \text{Class\_name} \times \text{Sig} \\
  \text{[Constant\_pool]}_{\text{name}} & = \text{[CP\_index]}_{\text{name}} \to \text{[CP\_info]}_{\text{name}}
\end{align*}
\]
Correctness of Java Card Tokenisation

\[
\begin{align*}
[\text{CP\_index}]_{\text{name}} &= \text{Class\_name} \times \text{Index} \\
[\text{Class}]_{\text{name}} &= \text{Class\_file} \\
\text{Class\_file} &= \text{Class\_flags} \times (\text{Class\_name}+\text{Void}) \times \text{Fields\_item} \times \text{Methods\_item} \times \text{Constant\_pool\_item} \times \text{Class\_name} \\
\text{Class\_flags} &= \text{Class\_flag} \rightarrow \text{Bool} \\
\text{Class\_flag} &= \text{Public} + \text{Final} + \text{Super} + \text{Interface} + \text{Abstract}
\end{align*}
\]

Since there is no overloading of fields, a field signature is just a name.

\[
\begin{align*}
\text{Fields\_item} &= \text{Field\_name} \rightarrow \text{Field\_info} \\
\text{Field\_info} &= \text{Field\_flags} \times [\text{Type}]_{\text{name}} \times \text{Value}
\end{align*}
\]

The value field is only required for static fields.

\[
\begin{align*}
\text{Field\_flags} &= \text{Field\_flag} \rightarrow \text{Bool} \\
\text{Field\_flag} &= \text{Public} + \text{Private} + \text{Protected} + \text{Static} + \text{Final} \\
[\text{Pack\_fields}]_{\text{name}} &= \text{Class\_name} \rightarrow \text{Fields\_item} \\
[\text{Pack\_methods}]_{\text{name}} &= \text{Class\_name} \rightarrow \text{Methods\_item} \\
\text{Methods\_item} &= \text{Sig} \rightarrow \text{Method\_info}
\end{align*}
\]

\[
\begin{align*}
\text{Method\_info} &= \text{Method\_flags} \times \text{Sig} \times ([\text{Type}]_{\text{name}} + \text{Void}) \times \text{Exception\_classes} \times \text{Maxstack} \times \text{Maxlocals} \times \text{Bytecode} \times \text{Exception\_handlers}
\end{align*}
\]

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. We represent the flags as a predicate (boolean function) over the possible settings. There are various constraints on the flags which we do not consider here.

\[
\begin{align*}
\text{Method\_flags} &= \text{Method\_flag} \rightarrow \text{Bool} \\
\text{Method\_flag} &= \text{Public} + \text{Private} + \text{Protected} + \text{Static} + \text{Final} + \text{Native} + \text{Abstract} \\
\text{Constant\_pool\_item} &= \text{Index} \rightarrow \text{CP\_info}
\end{align*}
\]

Finally,

\[
\begin{align*}
[\text{Type\_code}]_{\text{name}} &= [\text{CP\_index}]_{\text{tok}} \\
[\text{CP\_info}]_{\text{name}} &= [\text{Class\_ref}]_{\text{name}} + [\text{Field\_ref}]_{\text{name}} + [\text{Method\_ref}]_{\text{name}} + [\text{Interface\_method\_ref}]_{\text{name}}
\end{align*}
\]

### 6.2 Auxiliary Functions

We use an extended lambda calculus to define the functions. In addition to typed abstractions and pairs, we use conditionals, case expressions, let expressions, function overloading, and pattern matching in both lets and abstractions. We also use set-theoretic constructs, such as union and the map notation.

\[
\begin{align*}
(\text{lookup} : \text{Class\_ref} \times \text{Method\_ref} \rightarrow \text{Class\_ref} \times \text{Bytecode}) \\
[\text{lookup}]_{\text{name}} &= \text{lookup\_name}
\end{align*}
\]

There are a number of possibilities for how method lookup should be defined, depending on the definition of inheritance. For example, [Ber97, Pus98] use a ‘naïve’ lookup which does not take account of visibility modifiers. A fuller discussion of this appears in [Seg99].
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))

(lookup_int : Class_ref × Interface_method_ref → Class_ref × Bytecode)
lookup_int [name] = lookup_name

(static_val : Field_ref → Value)
λ(c,⟨f,t⟩) : Field_ref.env_name(pack_name(c))(c).Fields_item(f)

(update : Field_ref × Value → Statechange)
update ((c_name, f_name, _, _), v) = λ(c, h) : Environment × Heap.
  ⟨λ' : Package_name. λ cn : Class_name. if cn = c_name then
  let ⟨f, s, fields, m, cp, n⟩ = e(p')(cn)
  ⟨flags, d, value⟩ = fields(f_name) in
  ⟨f, s, fields + {f_name ← ⟨flags, d, v⟩}, m, cp, n⟩
else e(p')(cn),
  h⟩

Because of our assumption on the well-formedness of environments, p' = pack_name(cn).

(instance_fields : Class_ref → (Field_ref → Value))

instance_fields(c_name) =
  let (., super, ., fields, ., , , _) = env_name(pack_name(c_name))(c_name) in
  if c_name = javacard.lang.Object then instance_fields1(c_name, fields)
  else instance_fields1(super) ∪ instance_fields1(c_name, fields)

The function default computes the default values for each type.

instance_fields1(c_name, fields) =
  { (⟨c_name, f_name⟩ → default(d) ) | fields(f_name) = ⟨flags, d, _⟩ ∧ ¬flags(static) }

(instOf : Reference_type × Object → Bool)
instOf(d, o) =
  case o of
  Class_inst_obj(c_ref, _) → compat(d, c_ref)
  Array_obj(n, a, _) → compat(d, n, a))

We use the function compat : Reference_type × Reference_type → Bool which, in turn, uses local functions compatArray, and supers and interfaces. These latter two return the set of subclasses and superinterfaces, respectively, of a class. Since we have not considered interfaces here, we assume for now that interfaces returns the empty set.

compat(d_t, d_s) =
  case d_s of
Class_ref _ \rightarrow d_t \text{ in supers(d_s) or } d_t \text{ in interfaces(d_s)}
Array_type _ \rightarrow \text{ case } d_t \text{ of}
   \begin{align*}
   \text{Class_ref } _ \rightarrow & d_t = \text{javacard.lang.Object} \\
   \text{Array_type } _ \rightarrow & \text{ compatArray}(d_t, d_s)
   \end{align*}

compatArray(d_t, d_s) =
let (_ , d_1), (_ , d_2) = d_t , d_s in
case d_1 of
   \text{Primitive } _ \rightarrow d_1 = d_2
   \text{- } & \text{ not primitive(d_1) and compat(d_1, d_2)}

(constant_pool : Class_ref \rightarrow (CP_index \rightarrow CP_info))
\lambda c : \text{Class_name.}\lambda\langle c', t \rangle : \text{Class_name \times Index. } \text{let } \langle \ldots cp \ldots \rangle = \text{env_name(pack_name(c))(c) in } cp(i)

(method_class : Method_ref \rightarrow Class_ref)
\lambda\langle t, c_name, sig \rangle : \text{Method_ref. }\text{c_name}

(super : Class_ref \rightarrow Class_ref)
\lambda c : \text{Class_name. env_name(pack_name(c))(c). Super}

(super_invocation : Class_ref \times Method_ref \rightarrow \text{Bool})
\lambda\langle c, \langle c', \langle m_name, t \rangle \rangle \rangle : \text{Class_name \times Method_ref. }
\text{env_name(pack_name(c))(c).Class_flags(super) \land }
\text{m_name \neq } \langle \text{init} \rangle \land 
\neg \text{env_name(pack_name(c'))}(c').\text{Methods_item(\langle m_name, t \rangle).Method_flags(Private)}

The super class flag does not seem to be semantically significant. It exists only for reasons of backward compatibility.

(method_code : Method_ref \rightarrow Bytecode)
\lambda\langle t, c_name, sig \rangle : \text{Method_ref. }\text{env_name(pack_name(c_name))(c_name). Methods_item(sig). Bytecode}

(method_nargs : Method_ref \rightarrow \text{Nat})
\lambda\langle c_name, \langle m_name, types \rangle \rangle : \text{Method_ref. }\text{length(types)}

The function length returns the length of the list of types.

(reference_type : Type_code \times Class_ref \rightarrow Reference_type)
\lambda i, c : f(env(pack_name(c))(c). Constant_pool_item(i))

We assume a partial coercion function, f, of type CP_info \rightarrow Array_type, which takes a class name representing an array type, and returns the actual array type.

7 Token Interpretation

In the token format, the data is arranged by packages into CAP files. Each CAP file consists of a number of components.

There are tokens for the various entities — packages, classes, static fields, static methods, instance fields, virtual methods, and instance methods — each with a particular range and scope. Although package tokens should be scoped within a particular CAP file, (being indices into the package table which gives an AID), we will assume they are externally visible here.

The export file contains a list of tokens for imported entities. We only make use of this in assuming the existence of a token for the class javacard.lang.Object.

References to items external to a package are via tokens, which are used to find internal offsets. For example, the class component consists of a list of class information structures, each of which has method tables indexed by tokens that give offsets into the method component, where the method information is found. Internal references use the offsets directly.
7.1 Types

\[ \text{Package}_{\text{tok}} = \text{CAP\_file} \]
\[ \text{Package\_ref}_{\text{tok}} = \text{Package\_tok} \]

It follows that an environment consists of a mapping from package tokens to their corresponding CAP file.

\[ \text{env}_{\text{tok}} : \text{Package\_tok} \rightarrow \text{CAP\_file} \]

A CAP file consists of 11 components, though not all are used for method lookup (or, indeed, the rest of the operational semantics). We just include those components we need.

\[ \text{CAP\_file} = \text{Constant\_pool\_comp} \times \text{Class\_comp} \times \text{Method\_comp} \times \text{Static\_field\_comp} \times \text{Descriptor\_comp} \]

The class reference is either an internal offset into the class component of the CAP file of this 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. Thus we define

\[ \text{Class\_ref}_{\text{tok}} = \text{Package\_tok} \times (\text{Class\_tok} + \text{Offset}) \]
\[ \text{Field\_ref}_{\text{tok}} = \text{Static\_field\_ref} + \text{Instance\_field\_ref} \]
\[ \text{Static\_field\_ref} = \text{Class\_ref}_{\text{tok}} \times (\text{Static\_field\_tok} + \text{Offset}) \]
\[ \text{Instance\_field\_ref} = \text{Class\_ref}_{\text{tok}} \times \text{Instance\_field\_tok} \]
\[ \text{Method\_ref}_{\text{tok}} = \text{Static\_method\_ref} + \text{Virtual\_method\_ref} + \text{Super\_method\_ref} \]
\[ \text{Static\_method\_ref} = \text{Class\_ref}_{\text{tok}} \times (\text{Static\_method\_tok} + \text{Offset}) \]
\[ \text{Virtual\_method\_ref} = \text{Class\_ref}_{\text{tok}} \times \text{Virtual\_method\_tok} \]
\[ \text{Super\_method\_ref} = \text{Class\_ref}_{\text{tok}} \times \text{Virtual\_method\_tok} \]
\[ \text{Interface\_method\_ref}_{\text{tok}} = \text{Class\_ref}_{\text{tok}} \times \text{Interface\_method\_tok} \]
\[ \text{CP\_index}_{\text{tok}} = \text{Package\_tok} \times \text{Index} \]

Let \[ \text{SMP} = \text{SM\_index} \rightarrow \text{SM\_ref} \] (‘supermethod pool’).

\[ \text{Constant\_pool}_{\text{tok}} = \text{CP\_index} \rightarrow \text{CP\_intro}_{\text{tok}} \times \text{SMP} \]
\[ \text{Class}_{\text{tok}} = \text{Class\_info} \]
\[ \text{Class\_comp} = \text{Classes} \times \text{Interfaces} \]

We ignore the tag and size items.

\[ \text{Classes} = \text{Offset} \rightarrow \text{Class\_info} \]
\[ \text{Class\_info} = \text{Class\_flags} \times \text{Interface\_count} \times \text{Super} \times \text{Public\_table} \times \text{Package\_table} \times \text{Class\_ref}_{\text{tok}} \]

The class reference is to the class itself.

\[ \text{Class\_flags} = \text{Class\_flag} \rightarrow \text{Bool} \]
\[ \text{Class\_flag} = \text{Interface} + \text{Shareable} \]

Access information is not given by the flags in CAP format. We assume that each Class\_info structure is labelled with the corresponding class reference. We do not regard OxFFFF as an offset.

\[ \text{Public\_table} = \text{Public\_base} \times \text{Public\_size} \times (\text{Index} \rightarrow \text{Offset} + \{\text{OxFFFF}\}) \]
\[ \text{Package\_table} = \text{Package\_base} \times \text{Package\_size} \times (\text{Index} \rightarrow \text{Offset}) \]
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.

It is convenient to sometimes consider the two method tables together. We write Class _info . Tables for the union of the two tables.

We abstract from the details of the method and constant pool components and regard them as mappings from indices to entries. We use Index to access elements of lists, and Offset to access tables, but since we formalise both as functions, the distinction is not important.

\[
\begin{align*}
\text{Constant}_\text{s}_\text{comp} & = [\text{Constant}_\text{pool}]_{\text{tok}} \\
[\text{Pack}_\text{methods}]_{\text{tok}} & = \text{Method}_\text{comp} \\
\text{Method}_\text{comp} & = \text{Exception}_\text{handlers} \times \text{Methods} \\
\text{Methods} & = (\text{Offset} \to \text{Method}_\text{info}) \\
\text{Method}_\text{info} & = \text{Method}_\text{flags} \times \text{Maxstack} \times \text{Nargs} \times \text{Max}_\text{locals} \times \text{Bytecode} \\
\text{Method}_\text{flags} & = \text{Method}_\text{flag} \to \text{Bool} \\
\text{Method}_\text{flag} & = \text{Extended} + \text{Abstract} \\
\text{Exception}_\text{handlers} & = \text{Index} \to \text{Exception}_\text{handler}_\text{info} \\
[\text{Pack}_\text{fields}]_{\text{tok}} & = \text{Static}_\text{field}_\text{comp} \times \text{Descriptor}_\text{comp} \\
\text{The static field component contains details of the static fields in the classes of a package. The instance fields, on the other hand, only appear in the descriptor component. We assume offsets are into the appropriate list.} \\
\text{Static}_\text{field}_\text{comp} & = \text{Array}_\text{init} \times \text{Non}_\text{default}_\text{values} \\
\text{Array}_\text{init} & = \text{Offset} \to \text{Array}_\text{init}_\text{info} \\
\text{Array}_\text{init}_\text{info} & = \text{Count} \times (\text{Index} \to \text{Value}) \\
\text{Non}_\text{default}_\text{values} & = \text{Offset} \to \text{Value} \\
\text{The descriptor component is used for “parsing and verifying” the CAP file. The only use of it in the semantics seems to be to find the types of instance fields for the new instruction.} \\
\text{Descriptor}_\text{comp} & = \text{Class}_\text{ref} \to \text{Class}_\text{descriptor}_\text{info} \\
\text{Class}_\text{descriptor}_\text{info} & = (\text{Class}_\text{tok} + \{0xFFFF\}) \times \text{Access}_\text{flags} \times \text{Offset} \times \\
\text{Interface}_\text{count} \times \text{Field}_\text{count} \times \text{Method}_\text{count} \times \\
\text{Interfaces} \times \text{Field}_\text{descs} \times \text{Method}_\text{descs} \\
\text{Rather than represent arrays as lists, and then define functions to traverse the lists and find an entry with a given index, we choose to represent the arrays directly as functions of the various indices. In the class file, where entities have unique names, the name can be used as an index. In the CAP file, however, it is not so clear what to use as an index since some arrays contain both internal and external items. In the specification [Sun9! the Fields item (Field_descs here) is a list of field descriptors, of the form} \\
(\text{Token} + \{0xFFFF\}) \times \text{Access}_\text{flags} \times [\text{Field}_\text{ref}]_{\text{tok}} \times \text{Type}_\text{desc} \\
\text{The field descriptor items contain a token (or 0xFFFF) and a field reference. Presumably, the field reference gives the offset for static field references, and repeats the token for instance field references. We reformulate this in a triple Field_descs = Instance_fields \times Pub\text{static_fields} \times Pack\text{static_fields} where} \\
\text{Instance_fields = Token} \to \text{Access}_\text{flags} \times [\text{Field}_\text{ref}]_{\text{tok}} \times \text{Type}_\text{desc} \\
\text{Pub\text{static_fields = Token} \to \text{Access}_\text{flags} \times [\text{Field}_\text{ref}]_{\text{tok}} \times \text{Type}_\text{desc} \\
\text{Pack\text{static_fields = [Field}_\text{ref}]_{\text{tok}} \to \text{Access}_\text{flags} \times \text{Type}_\text{desc} \\
\text{Thus, the transformation should really have another pass, in which the functions would be flattened into arrays, and the true offsets calculated using the size of the various entries. We ignore this here.} \\
\text{We do not give the details for the Method_descs item since it is not used in the semantics.} \\
\text{Finally,} \\
[\text{Type}_\text{code}]_{\text{tok}} = \text{Nat} \times ([\text{CP}_\text{index}]_{\text{tok}} + \{0\})
7.2 Auxiliary Functions

The lookup function 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.

We use several local auxiliary functions:

\[
\text{class\_info} : \text{Class\_ref} \rightarrow \text{Class\_info}
\]
\[
\text{class\_info}' : \text{Package} \times \text{Class\_ref} \rightarrow \text{Class\_info}
\]
\[
\text{methods\_array} : \text{Class\_ref} \rightarrow (\text{Offset} \rightarrow \text{Method\_info})
\]
\[
\text{same\_package} : \text{Package\_ref} \times \text{Package\_ref} \rightarrow \text{bool}
\]

\[
\text{same\_package}(c\_ref, c'\_ref) =
\]
\[
\quad \text{let}(p\_tok, _,)(p'\_tok) = c\_ref, c'\_ref' \text{ in}
\]
\[
\quad p\_tok = p'\_tok
\]

For a given class reference, the function \text{class\_info} finds the corresponding class information structure in the global environment.

\[
\text{class\_info} (c\_ref) =
\]
\[
\quad \text{let} (...) \text{, cp\_comp, class\_comp, method\_comp, ...} : \text{CAP\_file} =
\]
\[
\text{case} c\_ref \text{ of}
\]
\[
\quad \text{Int} (\text{int\_pack\_tok}, _) \rightarrow \text{env\_tok(\text{int\_pack\_tok})}
\]
\[
\quad \text{Ext} (\text{ext\_pack\_tok}, _) \rightarrow \text{env\_tok(\text{ext\_pack\_tok})}
\]
\[
\text{in}
\]
\[
\text{case} c\_ref \text{ of}
\]
\[
\quad \text{Int} (_, \text{offset}) \rightarrow \text{class\_comp.Classes (offset)}
\]
\[
\quad \text{Ext} (_, \text{class\_tok}) \rightarrow
\]
\[
\quad \quad \text{let class\_offset} = \text{class\_offset(class\_tok)}
\]
\[
\quad \quad \text{in class\_comp.Classes (offset)}
\]

We define a variant, \text{class\_info}', which returns the class information structure in a particular CAP file.

\[
\text{methods\_array} (\text{class\_ref}) =
\]
\[
\quad \text{let} (...) \text{, method\_comp, ...} : \text{CAP\_file} =
\]
\[
\text{case} \text{class\_ref} \text{ of}
\]
\[
\quad \text{Int} (\text{int\_pack\_tok}, _) \rightarrow \text{env\_tok(\text{int\_pack\_tok})}
\]
\[
\quad \text{Ext} (\text{ext\_pack\_tok}, _) \rightarrow \text{env\_tok(\text{ext\_pack\_tok})}
\]
\[
\text{in}
\]
\[
\text{method\_comp.Methods}
\]

We assume the existence of several functions for resolving external tokens to internal offsets.\footnote{It is not clear how these should be implemented. One possibility is through the descriptor component; another is to use the export component.}

\[
\text{class\_offset} : \text{Package\_tok} \times \text{Class\_tok} \rightarrow \text{Offset}
\]
\[
\text{static\_field\_offset} : \text{Package\_tok} \times \text{Class\_tok} \times \text{Field\_tok} \rightarrow \text{Offset}
\]
\[
\text{method\_offset} : \text{Package\_tok} \times \text{Class\_tok} \times (\text{Virtual\_method\_tok} + \text{Static\_method\_tok}) \rightarrow \text{Offset}
\]

We extend these definitions in the obvious way to take arbitrary references.

The main steps of \text{lookup\_tok} for virtual methods are:

1. Get method array for the package of the actual class.

2. Get class information for the actual class.
lookup_tok (act_class_ref, (tag, dec_class_ref, method_tok)) =

case tag of

CONSTANT_virtual_method_ref, CONSTANT_super_method_ref ->

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, (tag, dec_class_ref, method_tok))
  else /* look in superclass */
    lookup_tok(super, (tag, dec_class_ref, method_tok))
else /* package */
  if method_tok >= package_base /
    (same_package(dec_class_ref, act_class_ref)
     \ tag = CONSTANT_super_method_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, (tag, dec_class_ref, method_tok))

3. If public:
  if defined then get info else lookup super
If package:
  if defined \ visible then get info else lookup super

(lookup : Class_ref \ Interface_method_ref \ Interface_method_ref \ Class_ref \ Bytecode)
[lookup]tok = lookup_tok

(lookup_int : Class_ref \ Interface_method_ref \ Interface_method_ref \ Class_ref \ Bytecode)
[lookup_int]tok = lookup_int_tok — not defined here

(static_val : Field_ref \ Value)
\(\lambda(p, c, f) : Field_ref.
  let offset = case f of
    Int i -> i
    Ext tok -> static_field_offset(p, c, tok)
in
  env_tok(p).Static_field_comp.Non_defualt_values(offset)
(update : Field_ref × Value → Stateschange)

\[ \text{update}(\langle p, c, f \rangle, v) = \lambda(e, h). \]
\[ \lambda f'. \]
\[ \text{if } y' = p \text{ then} \]
\[ \text{let } (\langle \_, \_, \_ \rangle, \langle \text{Array_init, Ndv} \rangle, \_ \rangle = c(p) \text{ in} \]
\[ \text{let } \text{offset} = \text{case } f \text{ of} \]
\[ \text{Int } i \rightarrow i \]
\[ \text{Ext } \text{tok} \rightarrow \text{static_field_offset}(p, c, \text{tok}) \]
\[ \text{in} \]
\[ \langle \_, \_, \_, \langle \text{Array_init, Ndv + \{offset \mapsto v\}} \rangle, \_ \rangle \]
\[ \text{else } e(p') \]
\[ , h \rangle \]

(constant_pool : Class_ref → (CP_index → CP_info))

\[ [\text{constant_pool}]_{\text{tok}} = \]
\[ \lambda(p_{\text{tok}}, \_). \lambda x. \text{let } \langle \text{cp, smp} \rangle = \text{env_tok}(p_{\text{tok}}). \text{Constant_pool_comp in} \]
\[ \text{case } x \text{ of} \]
\[ \text{Index} \langle \_, i \rangle \rightarrow \text{cp}(i) \]
\[ \text{SM} \langle \_, i \rangle \rightarrow \text{smp}(i) \]

(instance_fields : Class_ref → (Field_ref → Value))

\[ \text{instance_fields}(c_{\text{ref}}) = \]
\[ \text{let } (p_{\text{tok}}, c_{\text{tok}}) = c_{\text{ref}} \text{ in} \]
\[ \text{fields} = \text{env_tok}(p_{\text{tok}}). \text{Descriptor_comp.Fields.Instance_fields in} \]
\[ \text{super} = \text{class_info}(c_{\text{ref}}). \text{Super in}\]
\[ \text{if } c_{\text{tok}} = \text{javacard.lang.Object_token} \text{ then} \]
\[ \text{instance_fields1}(c_{\text{ref}}, \text{fields}) \]
\[ \text{else } \text{instance_fields}(\text{super}) \cup \text{instance_fields1}(c_{\text{ref}}, \text{fields}) \]

We assume the existence of a token, javacard.lang.Object_token. The function \text{default} computes the default values for each type.

\[ \text{instance_fields1}(c_{\text{ref}}, \text{fields}) = \]
\[ \{ (\langle c_{\text{ref}}, f_{\text{tok}} \rangle \mapsto \text{default}\langle d \rangle) \mid \exists f_{\text{tok}}. \text{fields}(f_{\text{tok}}) = \langle \text{flags}, f_{\text{ref}}, d \rangle \} \]

(instOf : Reference_type × Object → Bool)

The same as in the name interpretation, except for the definition of supers and interfaces.

(super : Class_ref → Class_ref)

\[ \lambda c_{\text{ref}} : \text{Class_ref}. \text{class_info}(c_{\text{ref}}). \text{Super} \]

(super_invocation : Class_ref × Method_ref → Bool)

\[ \lambda(\_, \langle \text{tag, \_, \_} \rangle) : \text{Class_ref} \times \text{Method_Ref}. \text{tag} = \text{CONSTANT_super_method_ref} \]

(method_class : Method_ref → Class_ref)

\[ \lambda(c_{\text{ref}}, \_): \text{Method_ref}. c_{\text{ref}} \]
We now formalise the equivalence between the class and CAP formats as a family of relations, 
\( \{ R_\theta : \llbracket \theta \rrbracket_{\text{name}} \leftrightarrow \llbracket \theta \rrbracket_{\text{tok}} \} \in \text{Abstract\_type} \), indexed by abstract type, \( \theta \). In fact, it is convenient to cheat a little by defining relations for certain types that do not correspond to any abstract types. The idea is that there is a fixed family of relations such that \( x R_\theta y \) when \( y \) is a possible transformation of \( x \). The relations are not necessarily total, i.e., for some \( x : \llbracket \theta \rrbracket_{\text{name}} \), there may not be a \( y \) such that \( x R_\theta y \). We make no restrictions on the relation domains.\(^8\)

We make various suppositions of the well-formedness of the input. For example, we assume that the class hierarchy is well-founded and that `javacard.lang.Object` is the top. We make no assumptions of well-formedness for CAP files, however. The only notion of well-formedness for a CAP file is that it is the result of transforming a collection of well-formed class files. Formally, the relations are defined as a mutually inductive collection of constraints, \( R_\theta \), for each type \( \theta \), where the types, \( \theta \), are given by the grammar:

\[^8\text{For example, arrays are not in the domain of } R_{\text{class\_ref}}. \text{ This is not important for the proof of correctness. However, this would have to be taken into account for the development of an algorithm.}\]
\[ \gamma ::= \text{Bool} \mid \text{Nat} \mid \text{Object}_\text{ref} \mid \text{Boolean} \mid \text{Byte} \mid \text{Short} \mid \text{Value} \mid \text{Word} \]
\[ i ::= \text{Package}_\text{ref} \mid \text{Class}_\text{ref} \mid \text{Static}_\text{field}_\text{ref} \mid \text{Instance}_\text{field}_\text{ref} \mid \text{Static}_\text{method}_\text{ref} \mid \text{Interface}_\text{method}_\text{ref} \mid \text{Virtual}_\text{method}_\text{ref} \mid \text{SM}_\text{ref} \]
\[ \kappa ::= \text{CP}_\text{index} \mid \text{SM}_\text{index} \mid \text{CP}_\text{info} \mid \text{Method}_\text{info} \mid \text{Type}_\text{code} \mid \text{Package} \mid \text{Class} \mid \text{Constant}_\text{pool} \mid \text{Pack}_\text{methods} \mid \text{Pack}_\text{fields} \]
\[ \theta ::= \gamma \mid i \mid \kappa \mid \theta \times \theta \mid \theta \rightarrow \theta \]

There are two sources of underspecification here. On the one hand, the relations really can be non-functional. On the other, there is a choice for what some of the relations are. For example, \( R_{\text{Class}_\text{ref}} \) is some bijection satisfying certain constraints. The relations between the ‘large’ structures, however, are completely defined in terms of those between smaller ones.

We first give the standard definitions of logical relations for the type constructors used here. These are used throughout the definition of the transformation. Then there are two parts to the transformation itself: the tokenisation, defined as the relations \( R_i \), and the ‘componentisation’, defined as the \( R_e \).

### 8.1 Logical Relations

There are standard definitions of \( R_\gamma \), \( R_\iota \rightarrow \iota \) and \( R_{\iota + \iota} \) in terms of \( R_\theta \) and \( R_{\psi} \). In addition, for each basic type \( \gamma \), we have \( R_\gamma = \text{id}_\gamma \).

\[
a R_\gamma a' \iff a = a' \\
f R_\iota \rightarrow \iota f' \iff \forall a R_\theta a' . fa R_{\psi} f'a'
\]

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

\[
\langle a, b \rangle R_{\gamma \times \iota} \langle a', b' \rangle \iff a R_\gamma a' \land b R_\iota b'
\]

\[
a R_{\iota + \iota} a' \iff (3b', a = \text{Theta} b \land a' = \text{Theta} b' \land b R_\theta b') \lor \\
(3c', a = \text{Theta} c \land a' = \text{Theta} c' \land c R_{\psi} c')
\]

Moreover, there is an obvious definition for lists:

\[
[] R_\theta [] \\
a :: as R_\theta a' :: as' \iff a R_\theta a' \land as R_\iota as'
\]

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 \( R_{\text{heap}} \) is defined as:

\[
\text{heap}_\text{name}, R_{\text{heap}} \text{heap}_\text{tok} \iff \forall r : \text{Object}_\text{ref} . \text{heap}_\text{name}(r) R_{\text{Object}} \text{heap}_\text{tok}(r)
\]

where \( R_{\text{Object}} \) is defined in terms of \( R_{\text{Class}_\text{ref}} \).

### 8.2 Tokenisation

The relations \( R_i \) 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, class tokens are scoped within a package, so the relation \( R_{\text{Ext}_\text{class}_\text{ref}} \) is between individual class names, and pairs of package and class 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 (with one exception to account for the copying of virtual methods in the token format). Formally, a relation, \( R_i \), is bijective when it is functional in both directions.

\[
a R b \land a' R b \Rightarrow a = a' \\
\text{and vice versa,} \\
a R b \land a R b' \Rightarrow b = b'
\]
However, relations are only bijective modulo the equivalence between internal and external references so we modify this to:

\[
\begin{align*}
R \land a' R b & \Rightarrow a = a' \\
R b & \Rightarrow (a R b' \iff \text{Equiv}(b, b'))
\end{align*}
\]

where equivalence, \text{Equiv}, of class references is defined as the reflexive symmetric closure of:

\[
\text{Equiv}((p, \text{offset}), (p, c)) \iff \text{class offset}(p, c) = \text{offset}
\]

We will say that the relation is 'externally bijective'. The second condition contains two parts: that the relation is injective modulo \text{Equiv}, and that it is closed under \text{Equiv}. We say that \text{R} is an external bijection when these conditions hold. We extend the definition of \text{Equiv} and external bijection to the other references. For example,

\[
\text{Equiv}((c, f), (c', f')) \iff f = f' \land \text{Equiv}(c, c')
\]

We cannot override static methods, but can override interface and virtual methods. The only bearing this has on the relations is in certain constraints for the tokenisation of overridden virtual methods.

The tokenisation process assigns tokens to external class references, static fields and static methods, and to all instance fields, virtual methods and interface methods.

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 \text{Equiv}. We write \( m \in c \) when a method with token \( m \) is declared in the class with reference \( c \). That is, \( m \in \text{dom}(\text{Table}) \) of \text{class info}(c) (defined on p. 30), and \text{pack_name}(c) for the package name of the class named \( c \).

We define functions for accessing flags:

\[
\text{Class flag}(c\_name, \text{flag}) = \text{env name}(c\_name). \text{Class flags}(\text{flag})
\]

\[
\text{Field flag}(c\_name, f\_name, \text{flag}) = \text{env name}(c\_name). \text{Fields}(f\_name). \text{Field flags}(\text{flag})
\]

The tokenisation uses the notion of external visibility.

\[
\text{Externally visible}(c\_name) = \text{Class flag}(c\_name, \text{Public})
\]

\[
\text{Externally visible}(c\_name, f\_name) = \text{Class flag}(c\_name, \text{Public}) \land \text{Field flag}(c\_name, f\_name, \text{Public})
\]

We will also write \text{public(sig)} and \text{package(sig)}.

The relations for \text{Instance field ref}, \text{Virtual method ref} and \text{Interface method ref} use \text{Class ref}, defined in the next section.

\text{(Package ref)} As mentioned above, we take package tokens to be externally visible. The relation \( R_{\text{Package ref}} \) is simply defined as any bijection between package names and tokens.

\text{(Ext class ref)} A bijection such that: \( c< c' \) \( R_{\text{Ext class ref}} (p, c) \Rightarrow \text{Externally visible}(c) \land \text{pack name}(c) \leq R_{\text{Package ref}} (p, c) \)

\text{(Instance field ref)} Package tokens must be higher than public tokens. Note that, in contrast to virtual method tokens, public and package tokens are drawn from the same namespace, and so this condition does not follow automatically.

\[
\begin{align*}
& (c, \text{sig}) R_{\text{Instance field ref}} (c, f) \land (c', \text{sig}) R_{\text{Instance field ref}} (c', f') \land \text{public(sig)} \land \text{package(sig)} \Rightarrow f < f'
\end{align*}
\]

The relation respects \( R_{\text{Class ref}} \):

\[
(c, \text{sig}) R_{\text{Instance field ref}} (c, f) \Rightarrow \text{Class ref}(c, \text{ref}) R_{\text{Class ref}} (c, \text{ref})
\]

Since the relation uses \( R_{\text{Class ref}} \) it is not bijective. However, it is functional from names to tokens, whereas in the other direction we have:

\[
\begin{align*}
& (c, \text{sig}) R_{\text{Instance field ref}} (c, f) \land (c, \text{sig}) R_{\text{Instance field ref}} (c', f')
\Rightarrow \text{Equiv} ((c, f), (c', f'))
\end{align*}
\]
(Ext_static_field_ref) An external bijection such that: \( \langle c\_name, f\_name \rangle R_{\text{Ext\_static\_field\_ref}} \langle c\_ref, f\_tok \rangle \Rightarrow \) 

\( \text{Externally\_visible}(c\_name, f\_name) \land \text{Field\_flag}(c\_name, f\_name, \text{Static}) \land c\_name R_{\text{Ext\_class\_ref}} c\_ref \)

(Ext_static_method_ref) An external bijection such that: \( \langle c\_name, sig \rangle R_{\text{Ext\_static\_method\_ref}} \langle c\_ref, m\_tok \rangle \Rightarrow \) 

\( \text{Externally\_visible}(c\_name, sig) \land \text{Method\_flag}(c\_name, sig, \text{Static}) \land c\_name R_{\text{Ext\_class\_ref}} c\_ref \)

(Virtual_method_ref) This is not a bijection because of the possibility of copying. Although ‘from names to tokens’ we do have:

\( \langle c\_name, sig \rangle R_{\text{Virtual\_method\_ref}} \langle c\_ref, m\_tok \rangle \land \langle c'\_name, sig' \rangle R_{\text{Virtual\_method\_ref}} \langle c\_ref, m\_tok \rangle \) 

\( \Rightarrow c\_name = c'\_name \land sig = sig' \) 

for a converse we have:

\( \langle c\_name, sig \rangle R_{\text{Virtual\_method\_ref}} \langle c\_ref, m\_tok \rangle \land \langle c\_name, sig \rangle R_{\text{Virtual\_method\_ref}} \langle c'\_ref, m'\_tok \rangle \) 

\( \Rightarrow (c\_ref \leq c'\_ref \lor c'\_ref \leq c\_ref) \land m\_tok = m'\_tok \)

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

\( \langle c\_name, sig \rangle R_{\text{Virtual\_method\_ref}} \langle c\_ref, m\_tok \rangle \land \) 

\( \langle c'\_name, sig \rangle R_{\text{Virtual\_method\_ref}} \langle c'\_ref, m'\_tok \rangle \land \) 

\( (c'\_name < c\_name \land (\text{package}(sig) \Rightarrow \text{same\_package}(c\_name, c'\_name))) \Rightarrow m\_tok = m'\_tok \)

The second condition says that the (public) tokens for introduced methods must have higher token numbers than those in the superclass. We assume a predicate, \( \text{new\_method} \), which holds of a method signature and class name when the method is defined in the class, but not in any superclass.

\( \text{public}(sig) \land \text{new\_method}(sig, c\_name) \land (c\_name, sig) R_{\text{Virtual\_method\_ref}} \langle c\_ref, m\_tok \rangle \Rightarrow \) 

\( \forall m'\_tok \in \text{super}(c\_ref). m\_tok > m'\_tok \)

Package tokens for introduced methods are similarly numbered, if the superclass is in the same package, but from 0 otherwise.

\( \text{package}(sig) \land \text{new\_method}(sig, c\_name) \land \) 

\( (c\_name, sig) R_{\text{Virtual\_method\_ref}} \langle c\_ref, m\_tok \rangle \land \text{same\_package}(c\_name, \text{super}(c\_name)) \Rightarrow \) 

\( \forall m'\_tok \in \text{super}(c\_ref). m\_tok > m'\_tok \)

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

\( \langle c\_name, sig \rangle R_{\text{Virtual\_method\_ref}} \langle c\_ref, m\_tok \rangle \Rightarrow (\text{public}(sig) \Rightarrow 0 \leq m\_tok \leq 127) \land (\text{package}(sig) \Rightarrow 128 \leq m\_tok \leq 255) \)

The specification [Sun99] also says that tokens must be contiguous but we will not enforce this.

(Interface_method_ref) \( \langle c\_name, sig \rangle R_{\text{Interface\_method\_ref}} \langle c\_ref, m\_tok \rangle \Rightarrow c\_name R_{\text{Class\_ref}} c\_ref \)

### 8.3 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 \( R_{\text{Class\_ref}}, R_{\text{Static\_field\_ref}} \) and \( R_{\text{Static\_method\_ref}} \), as relations 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. There are two ways we could ensure this. One possibility is, for example, to use the function \( \text{class\_info}\_c: \text{Class\_ref} \rightarrow \text{Class\_info} \) with the constraint:

\( c\_name R_{\text{Class\_ref}} \langle p\_tok, c\_tok \rangle \land c\_name R_{\text{Class\_ref}} \langle p\_tok, \text{offset} \rangle \Rightarrow \) 

\( \text{class\_info}((p\_tok, c\_tok)) = \text{class\_info}((p\_tok, \text{offset})) \)

The other possibility is to use the offset function \( \text{class\_offset}: \text{Package\_tok} \times \text{Class\_tok} \rightarrow \text{Offset} \) which returns the internal offset corresponding to an external token, and then define \( R_{\text{Class\_ref}} \) from this and \( R_{\text{Ext\_class\_ref}} \), and this is the solution we choose here. Clearly, therefore, \( R_{\text{Class\_ref}} \) is not a bijection.
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.

(CP_index) Define \([\text{CP_index}]_{p\_tok} = \text{Package}\_tok \times \text{Index}\). A bijection such that \(\langle c\_name, i\rangle R_{\text{CP_index}} (p\_tok, i') \Rightarrow \text{pack}\_name(c\_name) R_{\text{package_ref}} p\_tok\)

(CP_info) We have defined \(\text{CP_info} = \text{Class_ref} + \text{Method_ref} + \text{Interface_method_ref} + \text{Field_ref}\). We must define how the specific field and method references in the CAP file correspond to those in the class file.

(Field_ref) \(f\_ref R_{\text{field_ref}} (\text{tag}, f\_ref') \iff\)
\[\text{Field}\_flag(f\_ref, \text{Static}) \land \text{tag} = \text{CONSTANT}_\text{StaticFieldRef} \land f\_ref R_{\text{Static_field_ref}} f\_ref' \lor\]
\[\neg\text{Field}\_flag(f\_ref, \text{Static}) \land \text{tag} = \text{CONSTANT}_\text{InstanceFieldRef} \land f\_ref R_{\text{Instance_field_ref}} f\_ref'\]

(Method_ref) A method reference in the class file can become either a static, super or virtual method reference. Super method references also use virtual tokens. We use predicates, \text{static} and \text{super}, to determine whether a method reference in the class file can correspond to a static or super method invocation. If \text{static} holds, then the method must be a static method reference. However, the \text{super} predicate just indicates the possibility that a method could be a super reference. If it is called from \text{invokespecial}, this will be the case, but it will be a virtual method reference when called from \text{invokevirtual}. Let us write \(cp\) for \([\text{constant}\_\text{pool}]_{p\_tok} (c\_ref)\), where \(c\_ref\) is the relevant class reference.

\(\text{super}(m\_ref) \iff \exists i. \text{invoke\text{special}} i \in \text{code} \land cp(i) = m\_ref\)
The code can appear in any package.

\(m\_ref R_{\text{method_ref}} (\text{tag}, m'\_ref) \iff\)
\[\text{static}(m\_ref) \land \text{tag} = \text{CONSTANT}_\text{StaticMethodRef} \land m\_ref R_{\text{Static_method_ref}} m'\_ref \lor\]
\[\neg\text{static}(m\_ref) \land \text{super}(m\_ref) \land\]
\[\text{tag} = \text{CONSTANT}_\text{SuperMethodRef} \land m\_ref R_{\text{Virtual_method_ref}} m'\_ref \lor\]
\[\text{tag} = \text{CONSTANT}_\text{VirtualMethodRef} \land m\_ref R_{\text{Virtual_method_ref}} m'\_ref\]
(SM_index)
A bijection from the CP_index such that the entry is potentially a supermethod, that is, the \( i : \text{CP}_\text{index} \) such that \( \text{super}(cp(i)) \).

(SM_ref)
Relates virtual method references to the corresponding supermethod references.
Since \( \text{SMP} = \text{SM}_\text{index} \rightarrow \text{SM}_\text{ref} \) (‘supermethod pool’), the definition of \( R_{\text{SMP}} \) follows. Bytecode is defined using invokevirtual \( \text{SMP}_\text{index} \).

(Method_info)
We only treat certain parts of the method information here:
\[
\langle \text{flags}, \text{sig}, \_, \_, \text{maxstack}, \text{maxlocals}, \text{code}, \_ \rangle \rightarrow \langle \text{flags}', \text{maxstack}', \text{nargs}', \text{maxlocals}', \text{code}' \rangle \equiv
\langle \text{flags} \rightarrow \text{Method_flag flags}', \text{maxstack} \rightarrow \text{maxstack}', \text{maxlocals} \rightarrow \text{maxlocals}', \text{code} \rightarrow \text{Bytecode code} \rangle
\]
Flags are used less in the CAP format than in class files. Instead, access information is implicit in the use of tokens. For class flags, we simply have

\[
\text{Interface } R_{\text{Class_flag Interface}}
\]
Although, there is also a \text{Shareable} flag in the CAP format, we assume that no constraints are placed on this by a class file.

(Type_code)
Bytecode verification ensures that the constant pool entry must be a reference type.
\[
i R_{\text{Type_code}} \langle n, i \rangle \equiv
\begin{align*}
\text{let } \langle c, _ \rangle = i \text{ in } \\
\text{let } \text{cp_entry} = \text{constant_pool}(c)(i) \text{ in } \\
\text{let } v \text{ in } \\
\end{align*}
\[
\begin{align*}
\text{let } \text{cp_entry} = \text{constant_pool}(c)(i) \in \\
\text{let } \text{cp_entry} = \text{constant_pool}(c)(i) \in \\
\text{let } \text{cp_entry} = \text{constant_pool}(c)(i) \in \\
\text{let } \text{cp_entry} = \text{constant_pool}(c)(i) \in \\
\end{align*}
\]

We relate individual classes, but methods and constant pools are grouped together. Thus the name interpretation is all the information in one package and so, for example, \( [\text{Pack_methods}]_{\text{name}} : \text{Class_name} \rightarrow \text{Methods_item} \) is the ‘set’ of method data for all classes.

We use the auxiliary function, \( \text{method_offset} : \text{Package_token} \times \text{Class_token} \times \text{Method_token} \rightarrow \text{Offset} \). This is because the method information is spread between the two components in the token format. The relation \( R_{\text{Class}} \) ensures that a named method corresponds to a particular offset, and \( R_{\text{Pack_methods}} \) ensures that the entry at this offset is related by \( R_{\text{Method_info}} \).

(Constant_pool)
\[
\text{cp_name } R_{\text{Constant_pool}}(cp_token, \text{sm}) \equiv \text{cp_name } R_{\text{CP_index}} \rightarrow \text{cp_info cp_token} \wedge \text{cp_name } R_{\text{SMP}} \text{ sm}
\]
It is in \( \text{R}_{\text{CP_index}} : \text{Class_name} \times \text{Index} \leftrightarrow \text{Package_token} \times \text{Index} \) that the reorganisation is expressed essentially.

(Pack_methods) The method item and method component contain the implementations of both static and virtual methods.
\[
\forall (c, \text{name}, \text{sig}) R_{\text{Pack_methods}}(method_{\text{comp}}) \equiv \\
\forall (c, \text{name}, \text{sig}) R_{\text{Method_ref}}(p_token, c_token, m_token).
\]
\[
\text{methods}_{\text{name}}(c, \text{name}, \text{sig}) R_{\text{Method_info}}(\text{methods}_{\text{comp}}.\text{methods})(\text{method_offset}(p_token, c_token, m_token))
\]

INRIA
(Pack_fields) In the class file, all the field information is stored in the fields item. In the CAP file it is split between the static field and descriptor components. The former contains the values of the static fields, whereas the latter contains the flag and type information, for all fields. In the operational semantics we only use the type information, however.

As for the other components of the CAP file, we relate them to an aggregate, fields_name : class_name → Fields_item, which groups all the fields items in class files of a package. There are four clauses to the definition: for values, instance fields, public static fields and package static fields.

fields_name R_pack_fields (Desc_comp, Static_field_comp) ↔

∀(c_name, f_name) R_{field_ref} f_ref .
fields_name c_name f_name . Field_flags(Static) ⇒
fields_name c_name f_name . Value =

Static_field_comp. Non_default_values(static_field_offset(f_ref))
∧
∀(c_name, f_name) R_{instance_field} c_ref, f_tok .
(fields_name c_name f_name . (c_ref, f_tok))

R_{instance_field_info} Desc_comp(c_ref). Fields . Instance_fields(f_tok)
∧
∀(c_name, f_name) R_{ext_static_field_ref} c_ref, f_tok .
(fields_name c_name f_name . (c_ref, f_tok))

R_{pub_static_field_info} Desc_comp(c_ref). Fields . Pub_static_fields(f_tok)
∧
∀(c_name, f_name) R_{field_ref} f_ref .
pack_static(c_name, f_name) ⇒ fields_name c_name f_name

R_{pack_static_field_info} Desc_comp(c_ref). Fields . Pack_static_fields(f_ref)

We could have used R_field_ref in each of the four clauses, but give the more specific relations when possible. For the package visible static methods we have no choice but to use the more general relation.

We now define the relations for the field information structures:

⟨⟨flags|type, _, f_ref⟩⟩ R_{instance_field_info} ⟨f_tok, flags', f_ref, type'⟩
   ↔ flags R_{field_flags} flags' ∧ type R_{type'}

⟨⟨flags|type, value⟩⟩ R_{pub_static_field_info} ⟨f_tok, flags', f_ref, type'⟩
   ↔ flags R_{field_flags} flags' ∧ type R_{type'}

⟨⟨flags|type, value⟩⟩ R_{pack_static_field_info} ⟨flags', type'⟩
   ↔ flags R_{field_flags} flags' ∧ type R_{type'}

(Class) We define R_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 0xFFF, 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 (writing $c\_name$ for $cf.\_Class\_name$ and $c\_ref$ for $ci.\_Class\_ref$):

$$\begin{align*}
  cf@:\_Class\_file & R_{\_Class\_ci}@:\_Class\_info \iff \\
  cf.\_Class\_flags & R_{\_Class\_flags} \land \_Class\_flags \land \\
  cf.\_Super & R_{\_Class\_ref} \land \_Super \land \\
  \forall \_sig \in \_cf.\_methods \cdot \_item. \\
  \text{public}(\_sig) \Rightarrow \\
  \exists \_m\_tok, \_public\_base \leq \_m\_tok \land \_public\_base + \_public\_size \land \\
  \langle \_c\_name, \_sig \rangle \rightarrow R_{\_Virtual\_method\_ref} \langle \_c\_ref, \_m\_tok \rangle \land \_ci.\_Public\_table[\_m\_tok - \_ci.\_public\_base] = \_method\_offset(\_c\_ref, \_m\_tok) \\
  \land \\
  \text{package}(\_sig) \Rightarrow \\
  \exists \_m\_tok, \_package\_base \leq \_m\_tok \land \_package\_base + \_package\_size \land \\
  \langle \_c\_name, \_sig \rangle \rightarrow R_{\_Virtual\_method\_ref} \langle \_c\_ref, \_m\_tok \rangle \land \\
  \_ci.\_Package\_table[\_m\_tok \land _package\_base] = \_method\_offset(\_c\_ref, \_m\_tok) \\
  \land \\
  \forall \_m\_tok \in \_ci.\_Tables, \exists \_sig, \exists \_c'\_name. \\
  \langle \_c'\_name, \_sig \rangle \rightarrow R_{\_Virtual\_method\_ref} \langle \_c\_ref, \_m\_tok \rangle \land \_c\_name \leq \_c'\_name \land \\
  \text{public}(\_sig) \Rightarrow \\
  (\text{same\_package}(\_c\_name, \_c'\_name) \iff \_ci.\_public\_table[\_m\_tok - \_ci.\_public\_base] \neq 0xFFFF)
\end{align*}$$

Finally, we define $R_{\_package}$. Recall that

$$\begin{align*}
  \_{\text{Package}}\_name & = \_Class\_name \rightarrow \_Class\_file \\
  \_{\text{Package}}\_\_tok & = \_CAP\_file
\end{align*}$$

We use the notations $\text{class}\_info'(p\_tok, c\_tok)$ to indicate the class info for $c\_tok$ in the class component of CAP file $p\_tok$ (see p. 20), and $\text{pack}\_\_methods(p\_name)$ for the set of Method data extracted from each class file; similarly for $\text{pack}\_\_cps(p\_name)$ and $\text{pack}\_\_fields(p\_name)$.

$$\begin{align*}
  \text{pack}\_\_cps : \text{Package} \rightarrow \text{Constant}\_\_pool \\
  \text{pack}\_\_cps(p\_name) & = \lambda (c\_name, i). p\_name(c\_name) \cdot \_Constant\_\_pool\_item(i) \\
  \text{pack}\_\_\_methods : \text{Package} \rightarrow \text{Pack}\_\_methods \\
  \text{pack}\_\_\_methods(p\_name) & = \lambda c\_name. p\_name(c\_name) \cdot \_Methods\_\item \\
  \text{pack}\_\_\_fields : \text{Package} \rightarrow \_Fields \\
  \text{pack}\_\_\_fields(p\_name) & = \lambda c\_name. p\_name(c\_name) \cdot \_Fields\_\item
\end{align*}$$

$p\_name R_{\_package} p\_tok \iff \begin{align*}
  \text{pack}\_\_cps(p\_name) & R_{\_\_Constant}\_\_pool p\_tok.\_cp\_comp \land \\
  \forall c\_name R_{\_\_\_Class\_ref} c\_ref. c\_name \in \_dom(p\_name) \Rightarrow p\_name(c\_name) R_{\_\_\_Class}\_\_info'(p\_tok, c\_ref) \land \\
  \text{pack}\_\_\_methods(p\_name) & R_{\_\_\_Pack}\_\_methods p\_tok.\_method\_comp \land \\
  \text{pack}\_\_\_fields(p\_name) & R_{\_\_\_Pack}\_\_\_fields (p\_tok.\_desc\_comp, p\_tok.\_static\_field\_comp)
\end{align*}$

The offset functions link the various relations. We make a global assumption (in fact, local to an environment) of the existence of:

$$\text{class}\_\_\_offset, \text{method}\_\_\_offset, \text{static}\_\_\_field\_\_offset$$
9 Proof

We first prove that the auxiliary functions preserve the appropriate relations. Since we have not made the heap and environment explicit arguments, 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 9.1 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 \( [f]_{\text{name}} R_{\theta \to \theta'} [f]_{\text{tok}} \)

Proof: We consider two cases.

\((\text{lookup : Class_ref} \times \text{Method_ref} \to \text{Class_ref} \times \text{Bytecode})\)

For lookup, this is achieved by inducting over the class hierarchy and using the constraints on \( R_{\text{Class}} \) and \( R_{\text{Method}} \). Formally, we prove that

\[
\forall \text{name} \in \text{Class_ref} . \forall \text{act_name} \in \text{act_ref} . \forall \text{dec_name} \in \text{dec_ref} . \forall \text{sig} \in \text{sig} . \forall \text{tok} \in \text{tok} . \forall \text{ref} \in \text{ref} . \forall \text{m} \in \text{m} .
\]

\[
\text{lookup_name}(\text{name}, (\text{dec_name}, \text{sig})) R_{\text{Class_ref} \times \text{Method_ref} \times \text{Bytecode}} \text{lookup_tok}(\text{act_ref}, (\text{m_tok}, \text{dec_ref}))
\]

Induction over classes is possible since the subclass ordering is well-founded.

If the reference is to a virtual method, then:

The functions lookup_name and lookup_tok have similar structures. 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, that is, \( \text{methods\_item}(\text{sig}) \) is defined and the visibility condition holds.
  If the method token is public, then it must be that \( \text{m_tok} \geq \text{public\_base} \) and the offset is not \( 0x\text{FFFF} \).
  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 \( R_{\text{Class}} \), we have that there exists a token \( \text{m\_tok} \) such that \( \langle \text{name}, \text{sig} \rangle R_{\text{Virtual\_method\_ref}} \langle \text{act\_ref}, \text{m\_tok} \rangle \).

Using \( \text{act_name} R_{\text{Class_ref}} \text{act_ref} \) and \( \langle \text{dec_name}, \text{sig} \rangle R_{\text{Virtual\_method\_ref}} \langle \text{dec_ref}, \text{m_tok} \rangle \), and the fact that \( \text{dec_name} \) and \( \text{act_name} \) are in the same hierarchy, we deduce that \( \langle \text{name}, \text{sig} \rangle R_{\text{Virtual\_method\_ref}} \langle \text{act_ref}, \text{m_tok} \rangle \). Thus, again by \( R_{\text{Class}} \), it must be that \( \text{method\_offset}(\text{act_ref}, \text{m_tok}) \) is the entry in the method table computed by the lookup.

Then, by \( R_{\text{Pack}\_\text{methods}} \), we get that the corresponding method information structures are related by \( R_{\text{Method\_info}} \), and so 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 \( \text{method\_tok} \geq \text{package\_base} \) 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 \( R_{\text{Class}} \), and because the environments and actual class references are related. Equality follows from the inductive hypothesis at the superclass.

- If the function is not defined at the actual class in either format, then both algorithms look in the superclass and we appeal to the inductive hypothesis at the superclass. By the second constraint on \( R_{\text{Virtual\_method\_ref}} \), this must be because the token is less than the base.
  - In the case where the two functions differ, that is, 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 \( R_{\text{Virtual\_method\_ref}} \), we have that the results are equal in the current class.

\(^9\)Often this is taken as part of the definition of a logical relation.
(constant_pool:Class_ref → Constant_pool)

This follows directly from the assumption that the environments are related in the two formats and
places restrictions on the definitions of \(R_{\text{CP}}\_\text{index}, R_{\text{SM}}\_\text{index}\) and \(R_{\text{CP}}\_\text{info}\). Commutation ensures that the
constant pools are joined together without loss of data.

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 \(\text{code}_i[(\text{env}, \text{heap}, \text{op_stack}, \text{loc}_\text{vars}, m\_\text{ref})]\)
as the resulting state from the (unique) transition from \(\langle \text{code}, \text{op_stack}, \text{loc}_\text{vars} \rangle\) with environment \text{env} and
heap \text{heap}.

Thus we regard interpreted bytecode as having the type

\[
\text{State} \rightarrow \text{Bytecode} \times \text{State}
\]

where

\[
\text{State} = \text{Global}_\text{state} \times \text{Local}_\text{state}
\]

\[
\text{Global}_\text{state} = \text{Environment} \times \text{Heap}
\]

\[
\text{Local}_\text{state} = \text{Operand}_\text{stack} \times \text{Local}_\text{variables} \times \text{Class}_\text{ref}
\]

Now, the following fact is trivial to show: if \(R_B = \text{id}_B\) for all basic observable types, then \(R_\theta = \text{id}_\theta\) for all
observable \(\theta\). In combination with the following theorem, then, this says that if a transformation satisfies certain
constraints (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 \text{Word'}
) and the local variables (of observable type \text{Nat} \rightarrow \text{Word} so these are identical under the two
formats.

**Theorem 9.2** If

1. \(\text{name}_\text{env} R_{\text{Environment}} \text{env}_{\text{tok}},\)
2. \(\text{name}_\text{heap} R_{\text{Heap}} \text{heap}_{\text{tok}},\)
3. \(\text{ls} R_{\text{Local}_\text{state}} \text{ls'}, and\)
4. \(\text{code} R_{\text{Bytecode}} \text{code'}\)

then

\[
\text{[code]}_\text{name}_\text{env}_\text{name}_\text{heap}_\text{name}_{\text{ls}} R_{\text{Bytecode}} \times \text{State} \text{[code']}_\text{tok}(\text{env}_{\text{tok}}, \text{heap}_{\text{tok}}, \text{ls'})
\]

**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.

We take \text{invokevirtual} as an example. We use subscripts to distinguish the interpretations in the two
models. Suppose \(\text{name}_\text{heap} R_{\text{Heap}} \text{heap}_{\text{tok}}, \text{name}_\text{env} R_{\text{Environment}} \text{env}_{\text{tok}}, \text{name}_\text{m} R_{\text{Method}} \text{m}_{\text{tok}}\). Then, by induction
on \text{constant_pool}, we have \text{name}_\text{env} \text{name}_\text{m} \text{name}_\text{mref} \text{name}_\text{methref} \text{tok}. By the assumption on \text{heap} we have
\text{name}_\text{heap} \text{name}_\text{m} \text{name}_\text{mref} \text{tok}. Thus, by induction on \text{lookup}, we get that
\text{name}_\text{m} R_{\text{Class}_\text{ref}} \text{m}_\text{class} \text{tok} and \text{name}_\text{m} R_{\text{Bytecode}} \text{m}_\text{code} \text{tok}. Since the heap and environment do not change, we can conclude that
\text{invokevirtual} is representation independent. The cases of the other instructions are proven similarly.

**10 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.

It might seem that there is a circularity in the proof since the various relations make use of the environments,
the equivalence of which is tantamount to the correctness we seek to prove. However, although the relations are
defined using the environment, they do not use the relation between the environments. Moreover, the equivalence of environments does not assume the commutation of auxiliary functions but, rather, the equivalence of data that conforms to the specification.

There are a number of points, however, which are not entirely clear at this stage. We assumed functions to convert external tokens into internal offsets but it is not clear how they should be implemented. In this report we use the descriptor component to resolve tokens, but it may be that the export component should be used.

Also, it is not clear how array references are treated in the CAP file. In the constant pool in class file format, an array reference is a special form of class name, but this does not seem to be the case for class tokens, so have assumed a primitive function to convert an array as class name, into an array type.

For the semantics of the new instruction, we used a function instance_fields to compute the default values for the object fields. This requires the types of the fields, which are given in the descriptor component. Perhaps, though, it would suffice to use the declared_instance_size item in the corresponding class information structure.

Another unclear point is where the values of static fields are stored. We have assumed that they are stored in the field information structures of the fields item (in the class files) and the static field component (in the CAP files). This point is treated differently by [Ber97] and [BS98], and seemingly not considered by [Pus98].

The converter should produce an export file [Sun99] along with the CAP file but the details are not clear and we have not considered this. Finally, interface method references do not seem to appear from Draft 2 onwards of the JCVM 2.1 specification, but we have retained them here.

In addition to these problems, we have made a number of simplifications which could be relaxed. First of all, the proof should be extended to account for the rest of the transformation, accounting for interfaces, exceptions, and so on. 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. We found it more convenient to formalise various structures as functions where, in reality, they are actually laid out as tables. We could envisage another transformation pass where the functions are ‘flattened’ into tables.

We have used a simple definition of 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 [LR98]. 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. Although the specialisation of instructions could be handled by our technique (suitably combined with a type analysis), the extension is not 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 Bytecode).

These definitions have been formalised in Coq, and the lemmas verified [Seg99]. 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. The development is orthogonal to the proofs here and, in particular, independent of the formalisation of the bytecode semantics.

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. In fact, it suffices to think of the relations as being between all names and an infinite set of tokens but, in reality, we should use the actual names.
One significant improvement that could be made to the formalisation would be to use dependent types. This would offer a number of advantages. For example, if CP_index depended on a class reference, then we could avoid the explicit labelling of this. This would require an extension of the definition of logical relations, however.

In general, there are a number of changes which could be envisaged for the extraction. However, it was decided to ‘freeze’ the formalisation more or less in the form presented here so as to have a relatively stable version for the formalisation in Coq [Seg99].

References


