Electronic Notes in Theoretical Computer Science 82 No. 2 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume82.html 17 pages

# A Java Card CAP converter in PVS

Thomas Genet [1] Thomas Jensen [1] Vikash Kodati
David Pichardie [1]

*IRISA*
*Campus de Beaulieu*
*F-35042 Rennes*

**Abstract**

The Java Card language is a trimmed down dialect of Java aimed at programming smart cards. Java Card specifies its own class file format (the Java Card Converted APplet (CAP) format) that is optimised with respect to the limited space resources of smart cards. This paper deals with the certified development of algorithms necessary for the conversion of ordinary Java class files into the CAP format. More precisely, these algorithms are concerned with constructing and compressing method tables and constant pools. The main contribution of this paper is to specify and prove the correctness of these algorithms using the theorem prover PVS.

## 1 Introduction

The Java Card language [7] is a trimmed down dialect of Java aimed at programming smart cards. As with Java, Java Card is compiled into bytecode, which is then verified and executed on a virtual machine [4], installed on a chip on the card itself. However, the memory and processor limitations of smart cards necessitate a further stage, in which the bytecode is optimised from the standard class file format of Java, to the *CAP file* format [8]. The core of this optimisation is a *tokenization* in which names (strings) are replaced with tokens (integer values). Replacing strings with integers reduces the size of the code and enables a faster lookup of virtual methods using the standard object-oriented technique of *vtables*. Additional optimisations are obtained by a *componentisation* that merges class files from the same package into one *CAP file*. This means that data which is common to several class files can be shared and that symbolic references between classes from the same CAP file can be transformed into memory offsets relative to the CAP file.

[1] Email: {Thomas.Genet, Thomas.Jensen, David.Pichardie}@irisa.fr

In a companion article [3] we have described a semantic framework for proving the correctness of Java Card tokenization. The basic idea underlying that framework was to give an abstract description of the constraints from the official specification of the tokenization and show that any program transformation satisfying these constraints is 'correct'. Notice that this is independent of showing that there actually exists a collection of functions satisfying these constraints (which is not done in *op. cit.*). 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 [8] (given certain assumptions) are sufficient.

The aim of the work reported here is to construct a provably correct program that will transform Java Card class files into equivalent CAP files—we will call such a program a *CAP converter*. The result mentioned above reduces considerably the proof obligations for constructing a provably correct converter. For the tokenization, it is enough to verify that the CAP converter respects constraints on the tokenization imposed in the official language definition (see 4) to ensure correctness. For the componentisation, the proof is facilitated by first developing an abstract theory of merging tables and then instantiating this theory to the relevant Java Card structures such as the constant pool (see 5).

The paper is structured as follows. We first provide an intuitive explanation of the purpose of tokenizing Java Card class files by describing the differences between method resolution for Java Card class files and for Java Card CAP files (Section 2). We then proceed (Section 3) to describe the PVS formalization of the class file and the CAP format on which the CAP converter operates. Section 4 presents the development and accompanying proofs of the tokenization part of the converter. Section 5 on componentisation describes the specification and the implementation of constant pool merging in the CAP format.

## 2 Virtual method dispatch in Java Card

In the Java language, when calling a method $m$ on an object of class $c$, the bytecode is found using a method lookup function:

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

which implements the resolution of virtual method invocation. It takes a class reference $c$ (the actual class of the receiver object), a method reference $m$ (the signature and the class in which the $m$ is declared), and returns the reference to the class where $m$ was last redefined (which can be above $c$ in the class inheritance hierarchy), together with the bytecode itself. In the class file format, starting from class $c$, the algorithm recursively searches upwards in the class hierarchy for a (re-)definition of the method $m$. The Java visibility mod-

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

Fig. 1. The `lookup` function for the class file format.

ifiers have to be taken into account when deciding whether or not a method redefines another—see below. The central steps of this search are

(i) Get class information for the actual class.

(ii) If $m$ is public: if $m$ is defined then *found* else *lookup super*
If $m$ is package-visible: if $m$ is defined and visible then *found* else *lookup super*

The procedure `lookup` for method lookup in Java class files is defined in Figure 1 using a pseudo-ML notation. The basic functionality is standard: in order to find the implementation of a method `m_sig` that was declared in the class `dec_class` we recursively search for the last overriding of `m_sig`, starting from the current class and moving up in the class hierarchy towards `dec_class`. Differences in the literature occur when it comes to taking visibility modifiers for methods into account. For example, [1,6] use a 'naive' lookup which does not take account of visibility modifiers. Our definition of method lookup takes these into account by making the test

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

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

```
lookup_tok (act_class_ref, (method_tok, dec_class_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, (method_tok, dec_class_ref))
   else /* look in superclass */
      lookup_tok(super, (method_tok, dec_class_ref))
else /* package */
   if method_tok >= package_base and
       same_package(dec_class_ref, act_class_ref)
    then
      let method_offset = package_table[method_tok mod 128 - package_base]
        in (act_class_ref, methods[method_offset].Bytecode)
   else /* look in superclass */
      lookup_tok(super, (method_tok, dec_class_ref))
```

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

```
class A {
 foo() { <code1> } ;
 bar() { <code2> } ;
}

class B extends A {
 bar() { <code3> } ;
 baz() { <code4> } ;
}
```

Tokenization

| | |
|---|---|
| A.foo | 1 |
| A.bar | 2 |

Method table

| | |
|---|---|
| 1 | &code1 |
| 2 | &code2 |

| | |
|---|---|
| B.bar | 2 |
| B.baz | 3 |

| | |
|---|---|
| 1 | &code1 |
| 2 | &code3 |
| 3 | &code4 |

Fig. 3. Tokenization of methods

### *2.1 Method lookup in the token format*

The Java Card CAP format is based on *tokens i.e.*, integers that replace names (strings) when referencing a package, a class or a method. Using tokens rather than names reduces the code size and allows for a more direct method lookup using the token as index into a method table, as explained in the following.

The lookup function

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

for the CAP format (given in Figure 2) differs from the class file format lookup function in that it uses method tables to speed up the search for a method definition. Method tables associate a method token with the address of the code that implements the method, so the lookup function can jump directly to the method instead of searching for it in the class hierarchy (see Figure 3). Using tokens as indices into the method tables imposes some constraints on the way in which methods are tokenized. In particular, it is important that a method that redefines another method receives the same token as the method that it redefines. In Figure 3, the method `bar` in class `B` receives the same token (2) as method `bar` in class `A`. A method invocation `X.bar()` where `X` has type `A` can thus be compiled into 'jump to the code in entry 2 of the method table'. We return to the formal description of the constraints in Section 4.

The Java Card CAP format stipulates that two method tables should be generated for a class: one for the methods that are visible outside the package of the class (we call this the *public* methods), and one for the methods that are only visible inside the package (we call this the *package* methods). Method tables can be partial in the sense that not all methods defined or inherited by a class need have an entry in the method tables. The Java Card CAP format contains for each method table a *base* ; method with tokens smaller than this base are not included in the table. Hence, these methods cannot be found by looking them up in the method table but must be found by searching the class hierarchy, as is done in the class file format (see Figure 2). The motivation for this feature is saving space. Leaving out certain methods from the table means that method tables take up less space at the cost of slowing down method lookup.

## 3 Class file and CAP formats

In this section we present the PVS specification of Class file format and CAP format. Each of them is built as a PVS Abstract Data Type, defined by its set of constructors, associated recognizer predicates and accessors. The PVS models are strongly idealized versions of Java Card class and CAP files. We abstract away from all details that are not relevant for our algorithmic considerations. The proofs developed in the following extend to a more accurate model but the details would obscure both the development and the presentation.

### *3.1 Class file format*

The class file format is an idealized version of Java (Card) class file hierarchies. At the top-most, class hierarchies are represented as forests (rather than trees) as follows:

```
ClassHier: DATATYPE
 BEGIN
  empty: empty?
  nodeHier(class: ClassFile,
           sons: ClassHier,
           brothers: ClassHier): nodeHier?
 END ClassHier
```

This more general representation has the advantage that it equally well models the class tree of a complete Java Card program and the forest of classes making up an individual Java Card package. Furthermore, from a proof-theoretic point of view this representation has good properties. In particular, using proofs by induction over a *ClassHier* structure *ch*, yields some inductive hypothesis over the son and brother hierarchies of *ch* which are of the same type as *ch*. This is not the case when using a structure where a hierarchy *ch* has a list of sons: the related induction hypothesis is over an object of type *list[ClassHier]*. In this case, the induction principle is more complex since it is a mutual induction over *ClassHier* and *list[ClassHier]*.

A class hierarchy is a forest that can be *empty* or which is of the form *nodeHier(c, s, b)* where *c* is a class, *s* is the forest of its sons and *b* is the forest of its brothers. For instance, a class *c1* having two children *c2* and *c3* will be represented by a term *nodeHier(c1, nodeHier(c2, empty, nodeHier(c3, empty, empty)), empty)*, where *c1* has a son *c2* which has a brother *c3*. In the above datatype, *class*, *sons* and *brothers* are the accessors and *empty?*, *nodeHier?* are the recognizers.

The ClassFile type itself is a PVS record structure with four fields:

```
ClassFile: TYPE =
[# super: Maybe[ClassRef_name],
   methods: Methods_name,
   name: ClassRef_name,
   cp: CP_name #]
```

Each class file contains a reference to the super class *super* (which maybe empty for the top-most classes), the list of methods that are defined in the class *methods*, a reference to the class itself *name* and its constant pool *cp*. The constant pool (represented by an array of constants) will be defined more precisely in section 5.

### *3.2 CAP format*

The CAP format is the result of substituting tokens for (method, class and package) names and regrouping the constant pools into one constant pool component for an entire package. As a result the datatype of the CAP file format is very similar to the Class File datatype except that tokens replace names everywhere and there is a unique constant pool for the whole package. Thus, a Package is a tokenized hierarchy associated with a tokenized constant pool:

```
Package: TYPE = [# hier: ClassHier_tok, cp: CP_tok #]

ClassHier_tok: DATATYPE
 BEGIN
  empty: empty_tok?
  nodeHier(class: ClassFile_tok,
      sons: ClassHier_tok,
      brothers: ClassHier_tok): nodeHier_tok?
 END ClassHier_tok

ClassFile_tok: TYPE =
[# super: Maybe[ClassRef_tok],
   methods: Methods_tok,
   name: ClassRef_tok #]
```

## 4 Tokenization of Java Card class files

In this section we first present the algorithm that for each class in a class hierarchy builds the corresponding method tables, according to the definition of the CAP format. In Section 4.2 we then prove a number of theorems that together proves the validity of the tokenization algorithm. Algorithms are expressed in the PVS specification language—see [5] for a description.

### *4.1 The program*

We present here the algorithm for computing the *token tables* for the class hierarchy. Token tables map a method to its corresponding token, and are the key data structure for building method tables. The algorithm is composed of several functions that traverse the class hierarchy and for each class first tokenize the methods defined in the class and then build the method tables for the class.

*Iterating over the class hierarchy*
The function *tokHier* transforms a hierarchy $h$ (of type *ClassHier*) into a similar hierarchy (of type *ClassHier_wtt*) where the type of classes is enriched with a table of tokens corresponding to the list of methods defined in the class as well as the inherited methods. Note that for a given Class Hierarchy $h$, since the token numbering of methods of $h$ depends on the token numbering

of methods defined above $h$, the token table of the superclass of $h$ are needed by *tokHier*. This token table is given to *tokHier* in two parts: public ones *mt_super_pub* and the package ones *mt_super_pack*. If $h$ has no superclass, those two tables are empty. The *tokHier* function constructs the token tables for the class on the top of $h$ using the *tokClass* function and recursively converts the sons and brothers hierarchy of $h$. Note that, the PVS *MEASURE* keyword introduces the measure used to prove well-foundedness of the recursion and thus termination of the function. In the particular case of *tokHier* the termination is guaranteed since the hierarchy $h$ is decreasing (w.r.t. recursive the subterm ordering ¡¡) along the recursive calls.

```
tokHier(h: ClassHier, mt_super_pub, mt_super_pack: TokenTable): RECURSIVE
ClassHier_wtt =
  (CASES h
    OF empty: empty,
      nodeHier(c, h_sub, h_rest):
        LET (mt_cPub, mt_cPack) = tokClass(c, mt_super_pub, mt_super_pack),
            mt_sub = tokHier(h_sub, mt_cPub, mt_cPack),
            mt_rest = tokHier(h_rest, mt_super_pub, mt_super_pack)
         IN
         nodeHier((# super := c`super,
                     methods := c`methods,
                     name := c`name,
                     tabPublic := mt_cPub,
                     tabPackage := mt_cPack,
                     cp := c`cp #),
                  mt_sub, mt_rest)
    ENDCASES)
   MEASURE h BY <<
```

*Tokenization of classes*

The function *tokClass* computes the public and package tables of token for a class $c$, knowing the public and package tables of token of the superclass. First, the method list of $c$ is splitted into a public and a private list (function *split_methods*). Then, *tokClassList* is called to compute the table of token for each list.

```
tokClass(c: ClassFile, mt_super_pub, mt_super_pack: TokenTable):
  [TokenTable, TokenTable] =
      LET (lPub, lPack) = split_methods(c`methods) IN
        (tokClassList(c`name, 0, lPub, mt_super_pub),
         tokClassList(c`name, 0, lPack, mt_super_pack))
```

For a given class name $c$, with a method list *listMet* and the token table of the superclass *mt_super*, the function *tokClassList* constructs the token table corresponding to *listMet* and the inherited methods. The integer field *next* is used during the recursive calls to count every method defined in *mt_super* in order to find the number of the first token that can be used to tokenize the methods in *listMet*. If a method $s$ is associated with a token $t$ in *mt_super* and if $s$ occurs in *listMet* then the *searchMethod* function ensures that $s$ will be numbered by the same token $t$ in the class $c$.

```
tokClassList(c: ClassName, next: MethodToken, listMet: Methods_name,
        mt_super: TokenTable): RECURSIVE TokenTable =
   (CASES mt_super
     OF null: listMetToTokenTable(c, listMet, next),
        cons(s, rest):
          LET (new_s, new_listMet) = searchMethod(c, s, listMet) IN
            cons(new_s, tokClassList(c, next+1, new_listMet, rest))
     ENDCASES)
   MEASURE mt_super BY <<
```

### *4.2 Verification of the tokenization*

To prove correctness of the specification of the tokenization algorithm, as it was pointed out in the introduction, it is enough to prove that this algorithm satisfies the constraints described in Sun's official specification. This proof includes more than 100 lemmas proved with PVS and is separated into four parts: tokenization of a redefined method, tokenization of a new method, tokenization of an inherited method and token distribution. Here, we give a flavour of each part of the proof.

#### *4.2.1 Tokenization of a redefined method*

One of the constraints state that when a class defines a method which is already present in the class' superclass (in this superclass the method may have been defined or inherited), it must receive the same token as the method in the superclass. As an example, here is the lemma to prove for the particular case of the public methods, in PVS syntax:

```
methods_redefined1: LEMMA
  ∀ (h: ClassHier, c1, c2: ClassFile_wtt,
      mt_super_pub, mt_super_pack: TokenTable, m: [Sig, MethodInfo_name]):
    LET h2 = tokHier(h, mt_super_pub, mt_super_pack) IN
       ValidHierarchy(h) ∧ subClass(c1, c2, h2) ∧
        member(m, c1`methods) ∧
          member(m, c2`methods) ∧ m`2`visibility = Pub
       ⊃
       (∃ (s1, s2: TokenTableStruct):
          member(s1, c1`tabPublic) ∧
           s1`method = m`1 ∧
            s1`class = c1`name ∧
```

$$\begin{array}{l}
\quad member(s_2,\ c_2\text{‘}tabPublic)\ \wedge \\
\quad\ s_2\text{‘}method\ =\ m\text{‘}1\ \wedge \\
\quad\quad s_2\text{‘}class\ =\ c_2\text{‘}name\ \wedge \\
\quad\quad\ s_1\text{‘}token\ = s_2\text{‘}token)
\end{array}$$

The lemma *methods_redefined1* says that if $c_1$ is a subclass of $c_2$ in the hierarchy $h_2$ (the tokenized version of the class hierarchy $h$) and $m$ is a method defined in both classes then $m$ will appear in the two corresponding tables with the same token but with a different name of relative class. The hierarchy $h$ is supposed to be a so-called *valid hierarchy* meaning *e.g.*, that there is no class occurring twice in the hierarchy. The proof of this lemma is based on an induction on the structure of the hierarchy $h$.

- if $h$ is empty, it is contradictory with the fact that $c_1$ is a subclass of $c_2$ in $h$, so there's nothing to prove.
- if $h$ is a node of a class $c$, a son-hierarchy $h_{sub}$ and a brother-hierarchy $h_{rest}$, there are three cases to study

(i) $c_2$ is equal[2] to $c$ and $c_1$ is a class of $h_{sub}$
(ii) $c_1$ is a subclass of $c_2$ in the hierarchy $h_{sub}$
(iii) $c_1$ is a subclass of $c_2$ in the hierarchy $h_{rest}$

The proof concerning the package table (lemma *methods_redefined2*) is similar.

### *4.2.2 Tokenization of a new method*

When a class defines a new public (resp. package) method (*i.e.* a method which does not appear in the superclass), it must have a token greater than all the tokens used in the superclass public (resp. package) table. This condition is expressed for the new public methods in the following way:

$$\begin{array}{l}
new_methods1:\ \text{LEMMA} \\
\quad \forall\ (h:\ ClassHier,\ c_1,\ c_2:\ ClassFile_wtt, \\
\qquad mt_super_pub,\ mt_super_pack:\ TokenTable, \\
\qquad m_1,\ m_2:\ [Sig,\ MethodInfo_name]): \\
\qquad \text{LET}\ h_2\ =\ tokHier(h,\ mt_super_pub,\ mt_super_pack)\ \text{IN} \\
\qquad\quad ValidHierarchy(h)\ \wedge \\
\qquad\quad\ member(c_1,\ h_2)\ \wedge \\
\qquad\quad\quad member(m_1,\ c_1\text{‘}methods)\ \wedge \\
\qquad\quad\quad\ m_1\text{‘}2\text{‘}visibility\ =\ Pub\ \wedge \\
\qquad\quad\quad\quad \neg\ member(m_1\text{‘}1, mt_super_pub)\ \wedge \\
\qquad\quad\quad\quad\ (\forall\ (c_2:\ ClassFile_wtt): \\
\qquad\quad\quad\quad\qquad subClass(c_1,\ c_2,\ h_2)\ \supset\ \neg\ member(m_1\text{‘}1,\ c_2\text{‘}methods))\ \wedge \\
\qquad\quad\quad\quad\quad subClass(c_1,\ c_2,\ h_2)\ \wedge \\
\qquad\quad\quad\quad\quad\ member(m_2,\ c_2\text{‘}methods)\ \wedge\ m_2\text{‘}2\text{‘}visibility\ =\ Pub \\
\qquad\quad \supset \\
\qquad\quad (\exists\ (s_1,\ s_2:\ TokenTableStruct):
\end{array}$$

[2] More precisely the restriction of $c_2$ where we omit the tables.

$$
\begin{array}{l}
member(s_1,\ c_1\text{`}tabPublic)\ \wedge \\
\ member(s_2,\ c_2\text{`}tabPublic)\ \wedge \\
\ \ s_1\text{`}method\ =\ m_1\text{`}1\ \wedge \\
\ \ \ s_1\text{`}class\ =\ c_1\text{`}name\ \wedge \\
\ \ \ \ s_2\text{`}method\ =\ m_2\text{`}1\ \wedge \\
\ \ \ \ \ s_1\text{`}token\ >\ s_2\text{`}token)
\end{array}
$$

This lemma states a property where $c_1$ is a subclass of $c_2$ in a hierarchy $h_2$ which is the tokenized version of a class hierarchy $h$. If $c_1$ defines a method $m_1$ which is new (since $\neg\ member(m_1\text{`}1, \mathrm{mt\_super\_pub})$) and if $c_2$ defines a method $m_2$ then the token used for $m_1$ ($s_2$'token) is greater than the token used for $m_2$ ($s_2$'token). The proof needs a double induction on the hierarchy. The proof is similar for the package tables.

#### *4.2.3 Tokenization of an inherited method*

Each method defined in a class must have a token but not only these methods. There's a token for an inherited method too. It is thus necessary to prove that a public method defined in class $c_2$ will appear in the public token table of all subclasses with the same token until a subclass redefines the method.

#### *4.2.4 Token distribution*

Some lemmas are also necessary to prove a final requirement of the specification *viz.*, that all tokens in a public table are in the interval $[0, length(table)-1]$ and every value in this interval correspond to a token in the table.

## 5 Merging constant pools

In the Java bytecode format, constants like integers, real numbers, strings, class and method names are all stored in an array called a *constant pool*. There is one constant pool per class. In the bytecode of the methods of the class, every occurrence of a constant is replaced by its corresponding index in the constant pool.

This section deals with the *componentisation* of Java Card packages. The Java Card definition specifies how a package of class files must be split into a number of components (method, class, export, constant pool, *etc*) that together represent the package. We will not deal with all these aspects of componentisation but focus on how to build the constant pool component. More precisely, we will show how to define and prove correct a function that is given a class hierarchy and produces two results: a global package constant pool and an offset function $f$. The global constant pool is produced by merging all the constant pools of the classes contained in the hierarchy. For every class of the hierarchy, the offset function gives the jump to perform in the global constant pool to find back the content of the constant pool of the class. In particular, the offset function will be used to backpatch the bytecode of every class of name $n$ of a Java Card package: in the bytecode of class named $n$, every occurrence of index $i$ will be replaced by $i + f(n)$.

Thus, the main property to prove on the componentisation function is that for every class $c$ of name $n$ which is in the hierarchy, the content of the $i$-th cell of the global constant pool of $c$ is equal to the content of the $(i+f(n))$-th cell of the global package constant pool. The function and its related proof can be divided into two simpler problems:

- merging two constant pools together, and
- apply this simple merging function over the class hierarchy to gather all the constant pools into a global constant pool with a correct offset function.

In section 5.1, we define a specific theory to deal with basic array copying and merging problems. In section 5.2, we show how to use this simple theory in order to achieve the componentisation function and its proof on class hierarchies.

### *5.1 A theory of merging arrays*

For representing constant pools in PVS, we chose a relatively low level representation in order to model precisely arrays and their bounds. The chosen representation makes use of PVS dependent typing:

$$cpool\colon \text{TYPE} = [\#\ size\colon nat,\ tab\colon [below[size] \to Content]\ \#]$$

Hence, a constant pool is a record type with a *size* field representing the size of the array and a *tab* field containing the array itself. The array is a total function mapping every index of $0 \ldots size-1$ to an element of type *Content*. Depending on the format including this constant pool the *Content* type will be instantiated by different types: method names in the Class File format and method tokens in the CAP format. Note that the *size* of a constant pool can be 0 as it is mentioned in the Java specification [8]. Then, defining array access and array modification is straightforward:

$$read(t\colon\ cpool,\ k\colon below[size(t)])\colon\ Content\ = tab(t)(k)$$

$$put(t\colon\ cpool,\ k\colon\ below[size(t)],\ e\colon\ Content)\colon\ (\{t_1\colon\ cpool\ \mid\ size(t) = size(t_1)\}) = \\ (\#\ size := size(t),\ tab := tab(t)\ \text{WITH}\ [(k) := e]\ \#)$$

In those two definitions, note that we make use of subtyping. Indeed, what the type definition of *read* says is that one can only read in a constant pool $t$ at a position of type $below[size(t)]$ i.e. the interval $0 \ldots size-1$ which is a subtype of the naturals. In the same way, calling the *put* function on a constant pool $t$ returns a constant pool of the subtype $\{t_1 : cpool \mid size(t) = size(t_1)\}$ i.e. type of constant pools having the same size as $t$. PVS has a great ability to deal with subtyping in proofs where subtyping information are heavily used by the decision procedures and results into greater automation. However, subtyping has to be used carefully. For instance, the recursive function *arraycopy*, whose type is:

$$
\begin{array}{l}
\textit{arraycopy}\,(t_1\colon\ \textit{cpool}\,,\ \textit{from\_pos}\colon\ \textit{below}\,[\textit{size}\,(t_1)]\,,\\
\qquad\qquad \textit{to\_pos}\colon\ \{i\ \mid\ \textit{from\_pos}\ \le\ i\ \wedge\ i\ <\ \textit{size}\,(t_1)\}\,,\ \textit{at\_pos}\colon\ \textit{nat}\,,\\
\qquad\qquad t_2\colon\ \{t\colon\ \textit{cpool}\ \mid\ (\textit{size}(t) - \textit{at\_pos})\ \ge\ (1 + (\textit{to\_pos} - \textit{from\_pos}))\})\colon\\
\text{RECURSIVE}\ \ (\{t\colon\ \textit{cpool}\ \mid\ \textit{size}\,(t) =\ \textit{size}\,(t_2)\})
\end{array}
$$

copies the content of an array $t_1$ between indexes *from_pos* and *to_pos* into an array $t_2$ at a position *at_pos*. The type of the result of this function could be more precise. In particular, we could specify in the type that in the resulting constant pool, (1) we must find the values of $t_1$ between the indexes *at_pos* and *at_pos + (to_pos - from_pos)* and (2) we must find back the original values of $t_2$ outside of these indexes. However, using such a very precise subtype tends to generate Type Checking Conditions (TCCs for short) that are more intricate and more difficult to prove than if the property is proven outside the typing, as additional lemmas: *correct_copy* for property (1) and *copy_not_modif_outside_values* for property (2):

$$
\begin{array}{l}
\textit{correct\_copy}\colon\ \text{LEMMA}\\
\quad \forall\ (t_1\,,\ \textit{acopy}\colon\ \textit{cpool})\colon\\
\qquad \forall\ (i\,,\ j\,,\ k\colon\ \textit{nat})\colon\\
\qquad\quad \forall\ (t_2\colon\ \textit{cpool})\colon\\
\qquad\qquad (i\ \le\ j\ \wedge\\
\qquad\qquad\ \ j\ <\ \textit{size}\,(t_1)\ \wedge\\
\qquad\qquad\quad k + (j - i)\ <\ \textit{size}\,(t_2)\ \wedge\ \textit{acopy}\ =\ \textit{arraycopy}\,(t_1\,,\ i\,,\ j\,,\ k\,,\ t_2))\\
\qquad\qquad\ \ \supset\\
\qquad\qquad\ \ (\forall\ (\textit{ind}\colon\ \{v\colon\ \textit{nat}\ \mid\ i\ \le\ v\ \wedge\ v\ \le\ j\})\colon\\
\qquad\qquad\qquad \textit{read}\,(t_1\,,\ \textit{ind})\ =\\
\qquad\qquad\qquad\ \ \textit{read}\,(\textit{acopy}\,,\ (\textit{ind} + k) - i))
\end{array}
$$

$$
\begin{array}{l}
\textit{copy\_not\_modif\_outside\_values}\colon\ \text{LEMMA}\\
\quad \forall\ (t_1\,,\ \textit{acopy}\colon\ \textit{cpool})\colon\\
\qquad \forall\ (i\,,\ j\,,\ k\colon\ \textit{nat})\colon\\
\qquad\quad \forall\ (t_2\colon\ \textit{cpool})\colon\\
\qquad\qquad (i\ \le\ j\ \wedge\\
\qquad\qquad\ \ j\ <\ \textit{size}\,(t_1)\ \wedge\\
\qquad\qquad\quad k + (j - i)\ <\ \textit{size}\,(t_2)\ \wedge\ \textit{acopy}\ =\ \textit{arraycopy}\,(t_1\,,\ i\,,\ j\,,\ k\,,\ t_2))\\
\qquad\qquad\ \ \supset\\
\qquad\qquad\ \ (\forall\ (\textit{ind}\colon\ \{v\colon\ \textit{nat}\ \mid\ v\ <\ k\ \vee\ (k + (j - i)\ <\ v\ \wedge\ v\ <\ \textit{size}\,(t_2))\})\colon\\
\qquad\qquad\qquad \textit{read}\,(t_2\,,\ \textit{ind})\ =\ \textit{read}\,(\textit{acopy}\,,\ \textit{ind}))
\end{array}
$$

Thanks to the previous *array_copy* function it is easy to define a function *sum* that concatenates two constant pools. Moreover, using the two previous lemmas it is easy to show that the *sum* function is correct i.e. in the sum of two constant pools $t_1$ and $t_2$, $t_1$ can be found back in the sum from indexes 0 to $\textit{size}(t_1) - 1$ and $t_2$ can be found in the sum between indexes $\textit{size}(t_1)$ to

$size(t_1) + size(t_2) - 1$. The jump necessary to find back $t_2$ in the sum will be used in the next section to construct the offset function.

```
correct_sum: THEOREM
  ∀ (t1, asum, t2: cpool):
    asum = sum(t1, t2) ⊃
     (∀ (i: below[size(asum)]):
        ((i < size(t1)) ⇒ read(asum, i) = read(t1, i)) ∧
         ((i ≥ size(t1)) ⇒
           read(asum, i) = read(t2, i − size(t1))))
```

### 5.2 Constant pools

As shown in section 4.1, the tokenization transforms a hierarchy of type *ClassHier* into a hierarchy of type *ClassHier_wtt* where the type of classes is enriched with a table of tokens corresponding to the list of methods defined in the class. Now, in order to convert the *ClassHier_wtt* type into the final CAP format (type *Package*) the only two transformations to perform are: discard the method lists and build the global constant. The purpose of this section is to detail the last one. Starting from a class hierarchy (of type *ClassHier_wtt*), the proposed componentisation function constructs the global constant pool and the offset function at the same time. The offset function is represented by an association list i.e. a list of pairs $(n, i)$ where $n$ is a class name and $i$ is the jump to achieve in the global constant pool to find back the constant pool of class named $n$. The componentisation function recursively travels through the class hierarchy structure following a depth-first path. The componentisation function takes two parameters: the class hierarchy *ch* and *jump* of type natural used in recursive calls to increment the offset by the size of the constant pools already merged. The result of this function is a triplet $(t, l, j)$ where $t$ is the merged constant pool, $l$ is the association list representing the offset function and $j$ is the size of the merged constant pool (used for recursive calls). This function recursively applies componentisation to sons and brothers of *ch* and uses the resulting constant pools and offset to build the global constant pool for *ch*. The componentisation is defined in PVS as follows:

```
componentisation(ch: ClassHier_wtt, jump: nat): RECURSIVE [CP_tok, off-
set_fun, nat] =
  CASES ch
    OF empty: (new_array(0), null, jump),
       nodeHier(cf, sons, brothers):
         LET (sons_CP, sons_fun, new_jump) = componentisation(sons, jump) IN
         LET (broth_CP, broth_fun, new2_jump) =
                   componentisation(brothers, new_jump)
         IN
             (sum(sons_CP, sum(broth_CP, cp(cf))),
```

```
                cons((name(cf), new2_jump), append(broth_fun, sons_fun)),
                size(cp(cf)) + new2_jump)
    ENDCASES
   MEASURE ch BY <<
```

Like in section 4.2 for the tokenization functions, the componentisation function is supposed to be applied to a valid hierarchy. However, for componentisation an additional assumption is necessary: we assume that there is a bijection between classes and their names. The following main theorem states that for all valid class hierarchies, if *comp_cp* is the componentised constant pool, if *off_fun* is the related offset function, then for every class file *cf* in this hierarchy (having a non empty constant pool), we can find back the $i$-th cell of the constant pool of *cf* at position $jump + i$ in *comp_cp*, where $jump$ =get(get_assoc(name(cf), off_fun)). The value of the jump is in fact the value associated to the name of the class *cf* into the association list *off_fun*.

```
correctness_cp_component: THEOREM
  ∀ (ch: ClassHier_wtt):
    ∀ (cf: ClassFile_wtt):
      LET cp = cp(cf),
          ctok = name(cf),
          comp_res = componentisation(ch, 0),
          comp_cp = PROJ_1(comp_res),
          off_fun = PROJ_2(comp_res)
        IN
        member(cf, ch) ∧ size(cp) > 0 ∧ ValidHier(ch) ⊃
         (∀ (i: below(size(cp))):
            read(cp, i) =
             read(comp_cp,
                  get(get_assoc(ctok, off_fun)) + i))
```

However, typing this theorem raises two non trivial proof obligations (TCCs) revealing two implicit assumptions that have to be proved first, as lemmas. The first lemma states that by looking for the name of the class *cf* in the association list we will obtain at least one value for the jump, i.e. that the offset function is total with regards to the class names of the hierarchy. The second lemma states that for every class file *cf*, $jump + i$ is within the bounds of the global componentised constant pool *comp_cp*. Even if the details are rather different, the global proof sketches for those two lemmas and for *correctness_cp_component* are similar. We perform an induction on the structure of the hierarchy *ch*. The base case of the empty hierarchy is easily discarded since it is contradictory with the fact that *cf* is known to be in the hierarchy *ch*. Then for the general case, we achieve a proof by cases on the definition of *member(cf, ch)*:

- if *cf* is on the top of the hierarchy, we need to prove that the result is correctly constructed and that the final gathering with the results of the recursive calls on sons and brothers does not modify its validity.

- if *cf* is in the sons or the brothers of the current node then we need to use the induction hypothesis to obtain the properties of the partial result and then, as in the previous case, check that the property is preserved by the gathering of remaining results.

## 6 Conclusions

We have specified the core algorithms of a CAP converter in PVS. These algorithms deals with the tokenization and the componentisation of Java Card class files. The tokenization algorithm was proved correct by showing that it fulfills the constraints detailed in the Java Card definition of the CAP format. As was shown in previous work [3], this is sufficient to prove the correctness of the conversion. Componentisation was proved correct in the particular case of merging constant pools.

The work is carried out in the setting of Java Card but the underlying optimisations principles apply to any object-oriented language that uses method tables to implement dynamic method dispatch. Also, there is nothing in the approach that imposes the use of the PVS theorem prover. The choice of PVS was mainly due to its convenient specification language and its degree of proof automation which was clearly of benefit.

The proofs are based on program models that abstract away a number of features of the Java Card formats but we believe that the formats are the right ones for addressing the core algorithmic issues. There are no foundational reasons why the results shouldn't carry over to a more detailed model. However, the specification and the verification of the converter revealed to be a long and difficult task. In many verification works, the complexity of the proof is mainly due to the size of the program to check. In our case the program is rather small but the complexity is elsewhere and twofold. First, the CAP format is quite intricate and it requires a certain investment of time to study Sun's official specification in order to abstract Java and JavaCard formats into a relevant PVS model. Next, in order to include the low level actions of the program into the verification, some aspects of the PVS model are also low level and need some low level lemmas and verifications. In these two aspects, PVS Type Checking Conditions were very useful to detect specification weaknesses *e.g.* zero-sized constant pools or assumptions that were implicitly made *e.g.* well-formedness of the class hierarchy. The specification of the converter itself is only 1873 lines of PVS; however, the whole proof consists of more than one hundred lemmas and 6500 lines of tactics.

While a number of works have dealt with verifying compiler optimisations for various languages and with various degrees of automation, we are aware of only one piece of work directly addressing the type of optimisations studied here. Denney [2] shows how to develop a Java Card converter using the program extraction mechanism of Coq. More precisely, from a constructive proof that for each Java Card program there exists an equivalent program in converted format, it is shown how to extract such a converter. As with our work, a number of restrictions were imposed: only package-visible methods are considered and the merging of constant pools is not detailed. Nevertheless, this is one of the most substantial programs ever constructed using program extraction. As noted in [2], program extraction is far from being a "push-button" technology and requires considerable Coq expertise from the developer. We believe that our "program-then-prove" approach is more accessible although it still requires a certain level of expertise (in this case with PVS). Whether program extraction can be made equally accessible by combining it with a high degree of proof automation is open.

**Acknowledgements:** Thanks are due to Yoann Padioleau for detailed comments on a draft of this paper.

## References

[1] P. Bertelsen. Semantics of Java byte code. Technical report, Department of Information Technology, Technical University of Denmark, March 1997.

[2] E. Denney. The synthesis of a Java Card tokenisation algorithm. In *Proc. of 16th Int. Conf. on Automated Software Engineering (ASE 2001)*, pages 43–50. IEEE Press, 2001.

[3] E. Denney and T. Jensen. Correctness of Java Card method lookup via logical relations. *Theoretical Computer Science*, 283:305–331, 2002.

[4] T. Lindholm and F. Yelling. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[5] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In *Proc. of 11th Int. Conf. on Automated Deduction (CADE)*, volume 607 of *LNAI*, pages 748–752. Springer-Verlag, 1992.

[6] Cornelia Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical Report TUM-I9816, Institut für Informatik, Technische Universität München, 1998.

[7] Sun Microsystems. *Java Card 2.0 Language Subset and Virtual Machine Specification*, October 1997. Final Revision.

[8] Sun Microsystems. *Java Card 2.1 Virtual Machine Specification*, March 1999. Final Revision 1.0.