Division of Informatics, University of Edinburgh

Centre for Intelligent Systems and their Applications

The Synthesis of a Java Card Tokenisation Algorithm

by

Ewen Denney

Informatics Research Report EDI-INF-RR-0143

Division of Informatics
http://www.informatics.ed.ac.uk/

November 2001

The Synthesis of a Java Card Tokenisation Algorithm

Ewen Denney

Informatics Research Report EDI-INF-RR-0143

DIVISION *of* INFORMATICS
Centre for Intelligent Systems and their Applications

November 2001

appears in Proceedings of the 16th IEEE International Conference on Automated Software Engineering, San Diego, USA

Abstract :
We describe the development of a Java bytecode optimisation algorithm by the methodology of program extraction. We develop the algorithm as a collection of proofs and definitions in the Coq proof assistant, and then use Coq's extraction mechanism to automatically generate a program in OCaml. The extraction methodology guarantees that this program is correct. We discuss the feasibility of the methodology and suggest some improvements that could be made.

Keywords : program synthesis, Java Card, type theory, Coq, specification

The Synthesis of a Java Card Tokenisation Algorithm

Ewen Denney
Division of Informatics
University of Edinburgh
Scotland
ewd@dai.ed.ac.uk

Abstract

We describe the development of a Java bytecode optimisation algorithm by the methodology of program extraction. We develop the algorithm as a collection of proofs and definitions in the Coq proof assistant, and then use Coq's extraction mechanism to automatically generate a program in OCaml. The extraction methodology guarantees that this program is correct. We discuss the feasibility of the methodology and suggest some improvements that could be made.

1 Introduction

We describe the development of a substantial algorithm by the methodology of *program extraction*. We develop the algorithm as a collection of proofs and definitions in the Coq proof assistant, and then use Coq's extraction mechanism to automatically generate a program in OCaml. The beauty of the extraction methodology is that this program is guaranteed to be *correct*. Of course, there is a lot of work involved in formulating the appropriate concepts and actually doing the proofs, and this is what we describe here.

This work was part of a project[1] concerned with *Java Card* [11, 12], a dialect of Java aimed at programming smart cards. Interest in formal methods is high within the smart card industry, due to the potentially disastrous effects of bugs in card software. Our project looked at formalising a particular optimisation on the Java card bytecode used on cards, and proving its correctness. This work is reported elsewhere [3, 4]. In this paper we will develop an algorithm to carry out this optimisation.

This has been one of the largest developments to date which uses Coq's extraction methodology. The combined size of the proofs is about 2700 lines and the generated (unoptimised) code in OCaml is about 500 lines.

We start by giving an overview of the methodology of program extraction, and a brief introduction to Coq and to Java Card. In Section 4, we give the details of the formalisation of Java Card we adopt, followed in Section 5, by the formalisation of the optimisation as a set of constraints. In the next section, we describe a theory we need to develop for the proof: enumeration of datatypes (Section 6). Then, in the next two sections, we describe the proofs that correspond directly to optimisations of (parts of) Java bytecode files. The core of the algorithm is developed in Section 7, and this is used to construct the tokenisation in Section 8. Section 9 describes how an extraction is carried out in Coq, and, finally, Section 10 draws conclusions on our application of program extraction in Coq.

2 Program Extraction

The theoretical basis of program extraction is the *Curry-Howard isomorphism*, the correspondence between constructive proofs and lambda-terms [7]. From the lambda-calculus, it is a short step to programs in a conventional functional programming language.

It should not be thought, though, that extraction somehow gives programs for free. Without automation, it is far more work to obtain a program by extraction than it would be to just write it directly. A fairer comparison is between program extraction and formal verification of a program after it has been written. The advantage of extraction over post hoc verification stems from the fact that the difficulty of verification proofs is due, in large part, to trying to match the logical decomposition of properties to the structure of the program. This is immediate with a development by extraction.

The paper [9] presents the theoretical background to extraction in Coq, and illustrates this with a small example. It does not really explain how to go about developing a significantly sized piece of software. Caldwell [2] explains how

[1]Most of the work was done as part of the Action Incitative *Java Card* at IRISA, Rennes.

to extract small recursive programs from inductive proofs in Nuprl. The most significant extraction in Coq to date appears to be Théry's synthesis of Buchberger's algorithm [13].

3 Coq

The Coq proof assistant [1] is an implementation of the Calculus of Inductive Constructions, allowing interactive proof development. The type theory is sufficiently powerful to concisely formalise logical and programming concepts. It is a lambda-calculus extended with inductive definitions [8]. Declaring a type to be inductive automatically gives induction and recursion terms.

Since the underlying logic is constructive, each completed proof is assigned a *proof object*, actually a lambda-term. The extraction mechanism converts these lambda-terms to programs in a chosen functional language (either OCaml, CamlLight, or Haskell), a process which we assume to be correct. The system will automatically extract all the proofs and definitions on which the selected proof depends and output the resulting code in the same file.

A well-known problem with extracting computational content from constructive proofs is deciding which parts of the proof are computationally relevant, and which are purely logical. For example, given a constructive proof of $\exists x : \tau \,.\, P[x]$, should we extract the witness to τ or not? Coq's solution to this problem is to have separate universes for computational and logical data. Thus, as well as a logical existential, Coq also has a computational existential, defined as a subset type — $\{x : \tau \mid P\}$ — the x of type τ such that P holds. Witnesses are only extracted for computational existentials.

For legibility, we will avoid Coq notation in this paper, and simply write both as $\exists$. Where necessary, though, and where it is best thought of as a type, we do distinguish the computational existential as a subset type.

4 Java Card

In this section, we describe the class and CAP file formats, and their formalisation in Coq.

As with Java, Java Card is compiled into bytecode, for which various constraints are verified, and then executed on a virtual machine [6] installed on a chip on the card itself. However, the memory and processor limitations of smart cards necessitate a further stage, in which the bytecode is optimised from the standard class file format of Java, to the *CAP file* format [12]. The core of this optimisation is a *tokenisation* in which names are replaced with tokens, enabling a more direct lookup of various entities.

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. We discuss this in more detail later.

The conversion is presented in [12] 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 conversion, only considering classes and methods.

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 visible[2], then it is ascribed a token. If the method is only visible within its package, then it is referred to directly using an offset into the relevant data structure. Thus references are either internal or external.

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.

The proof described in [3, 4] is based on formulating an abstract definition of the bytecode formats, which can be instantiated to either class or CAP files. This is organised into a family of *abstract types*. The two formats are formalised as recursive interpretations of these types, and the correctness constraints (in the next section) are defined recursively (formally, as a *logical relation*) between corresponding interpretations.

```
Inductive Abstract_type : Set :=
   Package_ref           : Abstract_type
 | Ext_class_ref         : Abstract_type
 | Class_ref             : Abstract_type
```

[2]We follow the terminology of [12], 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.

```
| Virtual_method_ref    : Abstract_type
| Class                 : Abstract_type
| Method_info           : Abstract_type
| Pack_methods          : Abstract_type
| Package               : Abstract_type
| Fun  : Abstract_type → Abstract_type
                        → Abstract_type.
```

When a type is defined inductively, Coq will automatically generate proof terms giving the corresponding principles of induction and recursion. These are used to give interpretations of abstract types in the two models. (Here, and below, we sometimes just give the types of definitions, and omit the actual body.)

```
Definition name_interp :
                Abstract_type → Set.
Definition tok_interp :
                Abstract_type → Set.
```

$$\begin{tikzcd}
& \mathit{Abstract\ Model} \arrow[dl, "\texttt{name\_interp}"'] \arrow[dr, "\texttt{tok\_interp}"] & \\
\mathit{Class\ files} \arrow[rr, leftrightarrow, "\mathit{correctness\ relation}"'] & & \mathit{CAP\ files}
\end{tikzcd}$$

For example, we interpret package references as package names and tokens, in the name and token interpretations respectively, and (abstract) functions are recursively interpreted as either partial functions (on names) or total functions (on tokens). We will discuss the definition of (concrete) partial functions using `PFun` in Section 6.1 below.

```
name_interp Package_ref = Package_name
tok_interp Package_ref = Package_tok

name_interp (Fun X Y)
  = PFun (name_interp X) (name_interp Y)
tok_interp (Fun X Y)
  = (tok_interp X) → (tok_interp Y)
```

Similarly, packages are interpreted as partial functions from `Class_name` to `Class_file`, and as the type `CAP_file`, respectively. We omit details of how the other abstract types are interpreted, as well as definitions of the concrete types.

In actual Java Card bytecode, these formats are sequences of bytes, so would be more faithfully represented using lists than functions. We will adopt a more high-level representation using partial functions from indices to elements, which is more convenient for reasoning. A further reason is compatibility with the proof of correctness described in [10, 3, 4]. Although total functions were used there, simplifying the formalisation considerably, we must take account of the function domains for developing the algorithm.

5 Optimisation Constraints

The input to the conversion process is a collection of class files, which we take to be arranged into an *environment*. At the abstract level, we define this as a function `Abs_environment = Fun Package_ref Package`. Roughly speaking, the correctness theorem which we want to prove is

```
∀env_name : name_interp Abs_environment.
  ∃env_tok : tok_interp Abs_environment.
             correct(env_name, env_tok).
```

where `correct` expresses the relation between environments in the two formats. However, we must alter this in a number of ways. Firstly, since the proof will be for arbitrary abstract types, we need separate correctness relations at each abstract type.

Secondly, although the goal is to construct conversion functions, we must also construct correctness relations, themselves subject to constraints.

In practice, however, combining the specification of the conversion functions with the specification of the corresponding correctness relations proved too unwieldy.

One particular problem was that proving the correctness of the conversion functions requires access to the definition of the correctness relations. Now, if the relations are combined with their own proof of correctness, then, in addition to the content giving the relation, we also get a large amount of proof annotation. This doesn't just make the proof hard to read — it typically becomes too large for Coq to expand the definitions. Presumably, in retrospect, we should have strengthened the specification sufficiently so its properties would be enough to reason with and we would never need to look at the proof terms, but it is often more natural to reason using the structure of the term directly. Unfortunately, we found this to be a frequent problem with the subset types. Thus, in the end, we were forced to separate the definitions into:

```
Definition JCrelation :
 ∀X:Abstract_type .
 (name_interp X) → (tok_interp X) → Prop.

Definition Relation_constraints :
 ∀X:Abstract_type .
 ((name_interp X) → (tok_interp X) → Prop)
        → Prop
```

and

```
Lemma JCrelation_correct :
 ∀X:Abstract_type .
  (Relation_constraints X
                (JCrelation X)).
```

Here, the (polymorphic) relation `JCrelation` is expressed as a map into `Prop` (*i.e.* for each `X`, the proposition whether or not a `(name_interp X)` and `(tok_interp X)` are related), `Relation_constraints` is a (polymorphic) predicate on these relations, and `JCrelation_correct` asserts that each such relation satisfies the corresponding predicate).

Thus, we construct the relations and then prove their correctness separately. This is a pity, since we are going against the spirit of extraction where, ideally, the computational content is constructed implicitly.

The third modification to our original theorem is that we must account for relation domains. The conversion functions convert entities in a *given* environment (of class files). Here, `dom X x` holds when `x` is in the domain of the type `X`. For example, `dom Package_ref p` holds when `p` is the name of a package in the set of class files we want to convert.

```
Definition dom :
  ∀X:Abstract_type .
       (name_interp X) → Prop.
```

The specifications of the relations, conversion functions, and domains, must be parameterised over a `name_interp Abs_environment`, but in this case it is clearer to exploit a feature of Coq which lets us declare this as a local variable rather than an explicit parameter. Hence we can leave this implicit.

The conversion functions are specified, then, as

```
Definition conversion :
  Abstract_type → Set :=
   λX:Abstract_type .
    ∃f:{x:(name_interp X) | (dom X) x}
                          → (tok_interp X).
      (respects (dom X)
                (JCrelation X) f).
```

where `respects d r f` holds when function `f` respects relation `r` on domain `d`. A constructive proof of this will give a conversion function for each `X`, which is correct with respect to `JCrelation X`.

The relation for environments, themselves, is different, though, since it relates, rather than assumes, environments, so we have a separate `R_Environment`. Similarly, the top-level theorem, for the conversion of environments, is not expressed in terms of the generic `conversion`.

Finally, then, the top-level theorem is:

```
Theorem convert_env :
  ∀e:(name_interp Abs_environment)
     ∃e':(tok_interp Abs_environment) .
                   (R_Environment e e').
```

This looks like our original theorem! The difference is that `R_Environment` defines the correctness condition in terms of conversion at smaller types, each of which is parameterised on `e`.

6 Enumeration

The tokenisation part of the conversion consists of assigning tokenised references to the name references, subject to various constraints. For the ‘atomic’ references, the token is simply a number and the constraint is (in most cases) bijectivity; for compound references, consisting of several names, we also require the relation to respect the relations on the component parts.

Computationally, this is easy — we simply enumerate the appropriate list of names. This gives the corresponding relation directly. The conversion function can then be constructed by treating the relation as an association list.

We need a number of basic definitions and lemmas for partial functions, injections, relations, and enumerations. The proofs are fairly straightforward but not trivial.

6.1 Basic Definitions

We need to use an explicit domain in our definition of partial functions, so a partial function from `A` to `B` is represented as a pair, consisting of a list for the domain, `listdom` : (`list A`) and a total function taking two arguments — a member of the domain, and a proof that the element is in the domain — to `B`. Since lists are an inductive datatype in Coq, they are finite. As well as being more natural to represent the domain as a list rather than a set in the type-theoretic setting of Coq, it also leads to a more straightforward extract. Most of the other definitions are self-explanatory. They all have constructive content. We use the notions of *countable* and *listable*. The constructive content of a proof of countability of a set is an enumeration — an embedding in `nat` — and the content of a proof of listability is a listing without duplicates of the members of the set.

```
Record PFun : Set :=
  mkPFun
   {listdom : (list A) ;
    parfun  : ∀a:A.(In a listdom) → B}.
```

```
Definition countable :=
    λA:Set . ∃f:A → nat .
        (fun_injection A nat f).
```

```
Definition lists :=
  λA:Set.λl:(list A) .
      (∀a:A . (In a l)) ∧ (nodups A l).

Definition listable :=
    λA:Set . ∃l: (list A) . (lists A l).
```

We want to extract to algorithms which use the equality of the programming language. What should this correspond to logically? We cannot use the built in notions of equality in Coq. The propositional equality is non-constructive and cannot be extracted, and giving an explicit definition for each type would lead to user-defined equalities in the extracted program.

Rather, with program extraction in mind, we will simply *assume* equalities for the primitive types we use. These can then be linked at extraction time to the equality of the programming language. Correctness of the extraction, then, requires that the OCaml equality correctly implements the Coq equality for those types we use it at (see Section 9). We make the following parametric definition of an equality type, defined as a record consisting of the equality function (but without any definition) paired with the property that it does correspond to Coq's own equality.

```
Record equality : Set :=
   mkeq {fun : A → A → bool;
         prop : ∀x,y:A .
                 (fun x y)=true ⟷ x=y}.
```

Then, `equality Package_name`, for example, is the type of an equality for package names. We construct equalities for complex types, making use of the following theorem (which extracts to the identity).

```
Definition sub_equality :
  ∀A:Set . ∀P:A → Prop .
   (equality A) →
          (equality {a:A | P a}).
```

6.2 Counting Lists

We use most of the above definitions and lemmas to define `count_list`, which converts a list into an enumeration function. Once the elements of a type are listed we can enumerate them. The full statement is

```
Lemma count_list :
  ∀A:Set .
    (equality A) → (listable A)
                       → (countable A).
```

The proof uses two subsidiary lemmas. The `assoc` function is used to construct a function out of an association list. The lemma says that for list l and index a, if l contains a pair (a, b) (*i.e.* it exists 'logically'), then it can be found (*i.e.* it exists constructively).

```
Lemma assoc :
  ∀A,B:Set .
   (equality A) →
     ∀l:(list A*B) . ∀a:A . ∃b:B .
     (In (a,b) l) → {b:B | (In (a,b) l)}.
```

The `number` function pairs elements of a list with naturals, counting up from a given n. This cumbersome definition is needed to get the proofs to go through. It says that for n and l, we can construct a list of pairs, l', the first elements of which form l itself, which has no duplicates, and such that for each paired index, i, $n \leq i < n + 1 + \texttt{length}\, l$.

```
Lemma number :
 ∀A:Set . ∀n:nat . ∀l:(list A).
  ∃l':(list A*nat).
    (((map fst l')=l) ∧ (nodups A l') ∧
     ∀an:(A*nat) .
     (In an l') →
      (n ≤(snd an)) ∧
        ((snd an) < (n+1+(length l)))).
```

Then `count_list` $eq\ l\ x$ will be extracted to `assoc` eq $(\texttt{number}\ 0\ l)\ x$.

6.3 Inverses

Computationally, to say that the partial function, f, is surjective on its range, is for each element, b, in the range, to find an a in the domain, such that $f a = b$.

```
Lemma pfun_surjective_on_range :
   ∀A,B:Set . ∀f:(PFun A B) .
    ∀b:B . (In b (range f)) →
     ∃a:A .∃H:(In a (listdom f)) .
                         (f a H) = b.
```

To construct the inverse, g, of injection, f, we must give the domain and the function. The domain is given as `listdom` g = `range` f, and the function, `parfun` g, is constructed using `pfun_surjective_on_range`.

```
Theorem injection_inverse :
 ∀A,B:Set . ∀f:(PFun A B) .
  (pfun_injection A B f) →
   ∃g:(PFun B A).∀a:A.
    ∀H:(In a (listdom f)).
     ∃ H':(In (f a H) (listdom g)) .
                    (g (f a H) H') = a.
```

Here, `pfun_injection` defines the injectivity of a partial function.

7 Method Tables

As discussed in Section 4, one of the main differences between the class and CAP formats is the arrangement and location of method information. Thus, the construction of the method tables is central to the conversion. This is the most intricate part of the proof, and requires the most proof obligations. The proof structure is strongly guided by the algorithm and we describe it along these lines.

We make the simplification of only considering *package visible* references. The public visible references have different constraints, but could be handled similarly. The goal in this section is to construct a compressed (package) virtual method table for each class as well as global offset and tokenisation functions for the package virtual methods.

We construct the tables in a number of stages. We write $sig \in^{def} c$ when the method signature *sig* is defined in class c, and $sig \in^{inh} c$ when *sig* is inherited by c.

Signature and offset for defined methods

The first stage is to construct a function which lists the signatures of the methods *defined* in each class, together with their offsets.

We specify a function $f1$ as listing the signatures and being injective on the method offsets.

```
Lemma package_def_sig_os :
  ∃f1: {c : Class_name | dom Class_ref c}
                     → (list Sig*Offset).
  ∃m_offset :
   {(c,sig): Class_name*Sig | sig ∈^def c}
                          → Offset .
   (fun_injection m_offset) ∧
    ∀c:{c : Class_name | dom Class_ref c}.
     ∀sig:Sig . ∀os:Offset .
        (In (sig,os) (f1 c)) ←→
          (sig ∈^def c) ∧
             (m_offset (c,sig) = os).
```

Then $f1$ is defined, roughly, as

```
f1 c =
   number (sig_count c) (class_sigs c)
```

where `sig_count c` is defined by recursing through the class hierarchy as the number of method signatures above class c in the hierarchy, and `class_sigs c` is the list of method signatures in c.

Signature and Offset for Inherited Methods

For each class, we calculate the list of methods which it can inherit, together with their offsets. Having calculated the offsets at the previous stage, we ensure that *inherited* methods will have the same offset. We construct

```
f2: {c : Class_name | dom Class_ref c}
                       → (list Sig*Offset)
```

This is calculated from `package_def_sig_os` by class recursion, and using

```
merge : (list Sig*Offset)
→ (list Sig*Offset) → (list Sig*Offset)
```

which combines the method lists of a class and a subclass, taking account of overriding. We refine the specification of $f2\,c$ into a constructive definition in a number of stages. We have

$\langle sig, os\rangle \in f2\,c$

$$\begin{array}{ll}
\iff & sig \in^{inh} c \wedge \texttt{m\_offset}(c, sig) = os \\
\iff & ((sig \in^{inh} sup\ c \wedge sig \notin^{def} c) \vee sig \in^{def} c) \\
 & \wedge\ \texttt{m\_offset}(c, sig) = os \\
\iff & (sig \in^{inh} sup\ c \wedge sig \notin^{def} c\ \wedge \\
 & \texttt{m\_offset}(c, sig) = os) \vee \\
 & (sig \in^{def} c \wedge \texttt{m\_offset}(c, sig) = os) \\
\iff & (\langle sig, os\rangle \in f2\,(sup\,c) \wedge sig \notin^{def} c) \\
 & \vee\ \langle sig, os\rangle \in f1\,c \\
\iff & \langle sig, os\rangle \in \texttt{merge}\,(f2\,(sup\,c))(f1\,c).
\end{array}$$

Signature, Offset and Token for Inherited Methods

We then add the tokens. Again, we refine the specification to the definition in stages, using $f2$ and its properties.

```
f3: {c : Class_name | dom Class_ref c}
  → (list Sig*Offset*Virtual_method_tok)
```

The (package) tokens are calculated on a class by class basis, numbering the inherited methods upwards from 128 (as specified by [12]).

$\langle sig, os, m\_tok\rangle \in f3\,c$

$$\begin{array}{ll}
\iff & \langle sig, os\rangle \in f2\,c \wedge sig\ is\ (m\_tok - 128)'th\ in\ c \\
\iff & \langle sig, os\rangle \in f2\,c\ \wedge \\
 & \langle sig, m\_tok\rangle \in \texttt{number}\,128\,(\texttt{class\_sigs}\,c) \\
\iff & \langle sig, os, m\_tok\rangle \in \texttt{number}\,128\,(f2\,c).
\end{array}$$

Token and Offset for Compressed Method Tables

Now we construct the actual method tables.

```
f4: {c : Class_name | dom Class_ref c}
     → (list Virtual_method_tok*Offset)
```

It is at this point that we make the choice to compress. For each class, the base is computed as the minimum token of the methods defined in that class, and we cut off at that point, retaining all the methods from that token upwards.

By enforcing the copying of methods with token numbers greater than those overridden, we can ensure that for each class c,

$$\langle m\_tok, os\rangle \in f4\ c \Rightarrow m\_tok \geq base\, c.$$

8 Conversion Functions

In this section, we describe how some of the various conversion functions are constructed, using the machinery developed in the previous sections.

We first illustrate the tokenisation, and then the componentisation. Algorithmically, the most natural split is to think of the conversion as being in two parts: on the one hand, there are the functions which are involved in or subsume the construction of the method tables, and on the other, those which use the tables. These latter are defined in a Coq section which is parameterised over the method offset and token functions, which are themselves computed from the tables.

8.1 Tokenisation

We show how package references are tokenised. The input is the list of package references in the current environment. This is constructed using the domain of the current environment.

```
Lemma list_packages :
 listable {p: Package_name |
                     dom Package_ref p}.
```

We construct an injection of names into tokens.

```
Lemma package_ref_inj :
  ∃f:{p:Package_name |
                dom Package_ref p}
     → Package_tok . (fun_injection f).
```

Proof. We must show the countability of $\{\mathtt{p} : \mathtt{Package\_name} \mid \mathtt{dom}\ \mathtt{Package\_ref}\ \mathtt{p}\}$. Applying the `count_list` lemma, this reduces to two subgoals: an equality for $\{\mathtt{p} : \mathtt{Package\_name} \mid \mathtt{dom}\ \mathtt{Package\_ref}\ \mathtt{p}\}$, and the proof of its listability. The first is proven by applying `sub_equality` to the equality for `Package_name` (recall that we assume primitive equalities for the basic types), and the second is just our lemma, `list_packages`. ∎

8.2 Componentisation

By 'componentisation' we mean the grouping of the various entities together into components for each package. Here we describe the construction of `conversion pack_methods`, which takes $methods\_name$ of type

$$\mathtt{Class\_name} \rightarrow \mathtt{Sig} \rightarrow (\mathtt{name\_interp}\ \mathtt{Method\_info}),$$

and returns a method component, of type

$$\mathtt{Offset} \rightarrow (\mathtt{tok\_interp}\ \mathtt{Method\_info}).$$

We use the bijection (for each package) between the defined $\mathtt{Class\_name} \times \mathtt{Signature}$ and `Offset`; in particular, $os2sig : \mathtt{Offset} \rightarrow \mathtt{Class\_name} \times \mathtt{Sig}$ is calculated using `injection_inverse`. Given $methods\_name : \mathtt{Class\_name} \rightarrow (\mathtt{Sig} \rightarrow \mathtt{name\_interp}\ \mathtt{Method\_info})$, rearranged as $uncurry(methods\_name) : \mathtt{Class\_name} \times \mathtt{Sig} \rightarrow \mathtt{name\_interp}\ \mathtt{Method\_info}$, we construct

$$\mathtt{Offset} \xrightarrow{os2sig} \mathtt{Class\_name} \times \mathtt{Sig} \xrightarrow{uncurry(methods\_name)} (\mathtt{name\_interp}\ \mathtt{Method\_info}) \xrightarrow{\mathtt{convert\_Method\_info}} (\mathtt{tok\_interp}\ \mathtt{Method\_info}).$$

Here, `convert_Method_info` does a straightforward rearrangement of a method information structure.

9 Extraction

We now describe what is involved in extracting code from the proof. All primitive constants and types (parameters in Coq), such as `Package_name`, must be realised.

```
Link Package_name := (list char).
```

We also want to override lists, booleans, naturals, and equality with definitions in the programming language. For example,

```
Extract Constant nat_128 => "128".

Extract Constant eq_pack_name => "(=)".

Extract Inductive bool => bool[true false].
```

We only link to equalities at non-function types, such as signatures and package names, so the OCaml and Coq equalities clearly match. We have some control over the degree of definition expansion and simplification involved in the extraction process, and it is worth using this to get more perspicuous code. Many of the proofs are computationally just the identity and these will be expanded automatically. There is some experimentation to get the most comprehensible program.

10 Concluding Remarks

This has been one of the largest developments to date which uses Coq's extraction methodology. The combined size of the proofs is about 2700 lines and the generated code in OCaml is about 500 lines.

We have simplified the model by only considering classes, methods, and package tokens, and by representing byte sequences by partial functions. As a methodology, extraction ranges from direct programming in a high-level language (effectively a form of automated data refinement), to full synthesis.

The definitions and theorems we have developed for injections, enumerations, partial functions, and inverses should be useful for similar constructions concerned with the transformation of data formats, although much more work is needed in writing libraries of certified software components.

The main conclusion we draw, in this regard, is that specification must take account of the intended implementation: develop a library of lemmas which extract appropriately, then build up the top-level specifications. In this case, partial functions would be represented as byte sequences.

As a case study in the extraction methodology, it has revealed a number of problems with Coq, both with the extraction mechanism, and with the proof development facilities, in general. As mentioned in Section 5, the main difficulty encountered in developing a program by extraction is when access is needed to a proof term. Even quite small programs can correspond to huge proofs. Nuprl [5] gets round this by allowing explicit access to the extract terms. Allowing explicit reference to the extraction process within the logic would be a way of handling axioms which, although unprovable, do hold for the corresponding extract terms.

Another problem is that the extraction mechanism of Coq (v6.3.1) can not cope with proofs which contain redexes containing terms which are 'too' higher-order (and invalid in the programming language), even though the reduct can be extracted. Short of this bug being fixed, the proofs were modified manually to make them amenable to extraction.

The fact that type expressions are sometimes not reduced, can leave them so large as to be incomprehensible. This tends to happen with recursive and parameterised types. A further inconvenience is that there can not be any dependencies between subgoals. It is not possible, for example, to split an existential goal, $\exists x : \tau \,.\, P$ into subgoals $x : \tau$ and P and work on P, thus implicitly instantiating x.

Finally, the generated program is large, unmodular, and rather difficult to understand. Further use could be made of automatic simplifications to create nicer code. We need some means of better reflecting architectural structure in the final program. In addition, it is unfortunate that Coq's natural language rendering mechanism can only be applied to proofs of propositions, and not computational constructions, so most of the constructions here can not make use of it.

Acknowledgements This work has benefited from discussions with colleagues at Rennes and Edinburgh. The author also wishes to thank the anonymous referees for their comments.

References

[1] B. Barras, S. Boutin, C. Cornes, J.-C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin-Mohring, A. Saibi, and B. Werner. The Coq Proof Assistant Reference Manual: Version 6.1. Technical Report RT-0203, Inria, Aug. 1997.

[2] J. Caldwell. Moving proofs-as-programs into practice. In *Proceedings, 12th IEEE International Conference on Automated Software Engineering*, IEEE Computer Society, pages 10–17, 1997.

[3] E. Denney. Correctness of Java Card Tokenisation. Technical Report 1286, Projet Lande, IRISA, 1999. Also appears as INRIA research report 3831.

[4] E. Denney and T. Jensen. Correctness of Java Card Method Lookup via Logical Relations. In G. Smolka, editor, *Proceedings of the 2000 European Symposium on Programming (ESOP'00)*, volume 1782 of *Lecture Notes in Computer Science*, pages 104–118. Springer Verlag, Mar. 2000.

[5] P. Jackson. *The Nuprl proof development system, version 4.2 reference manual and user's guide*. Computer Science Department, Cornell University, July 1995.

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

[7] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *Monographs on Computer Science*. Oxford University Press, 1990.

[8] C. Paulin-Mohring. Inductive definitions in the system Coq; rules and properties. In M. Bezem and J. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 664, pages 328–345. Springer-Verlag, 1993.

[9] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15(5–6):607–640, 1993.

[10] G. Segouat. Preuve en Coq d'une mise en oeuvre de Java Card. Master's thesis, Projet Lande, IRISA, 1999.

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

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

[13] L. Théry. A Certified Version of Buchberger's Algorithm. In C. Kirchner and H. Kirchner, editors, *Automated Deduction – CADE-15*, volume 1421 of *LNAI*, 1998.