# Efficient reasoning about executable specifications in Coq

Gilles Barthe and Pierre Courtieu

INRIA Sophia-Antipolis, France
{Gilles.Barthe,Pierre.Courtieu}@sophia.inria.fr

**Abstract.** In previous papers, we report on a formal executable specification of the JavaCard platform in Coq. In this paper, we describe a package to reason efficiently about such specifications. The package provides a command for synthesizing a customized induction principle for a recursively defined function, and a tactic that combines the application of the customized induction principle with automatic rewriting. We further illustrate how the package leads to a drastic reduction (by a factor of 10 approximately) in proofs in a large-scale case study on reasoning about JavaCard.

## 1 Introduction

Proof assistants based on type theory, such as Coq [26] and Lego [19], combine an expressive specification language (featuring inductive and record types) and a higher-order predicate logic (through the Curry-Howard Isomorphism) to reason about specifications. Over the last few years, these systems have been used extensively, in particular for the formalization of programming languages. Two styles of formalizations are to be distinguished:

- *functional specifications*. Here specifications are written in a functional programming style, using pattern-matching and recursion;
- *relational specifications*. Here specifications are written in a logic programming style, using inductively defined relations;

In a series of papers [5, 6], we report on a formal executable (functional) specification of the JavaCard[1] platform, and argue that such a style of specification provides a better alternative for formalizing complex programming languages, as it offers support for testing the specification and comparing it with a reference implementation, and the possibility to generate programs traces upon which to reason using e.g. temporal logic. Yet, it is striking to observe that many machine-checked accounts of programming language semantics use inductive relations. In the case of proof assistants based on type theory, two factors contribute to this situation:

[1] JavaCard is a dialect of Java tailored to programming with smartcards, see http://java.sun.com/products/javacard.

– firstly, type theory requires functions to be total and terminating (in fact, they should even be provably so, using a criterion that essentially captures functions defined by structural recursion), therefore specifications written in a functional style may be difficult and more cumbersome than their relational counterpart;
– secondly, a proof assistant like Coq offers, through a set of inversion [12, 13] and elimination tactics, effective support to reason about relational specifications. In contrast, there is no similar level of support for reasoning about executable functions.

The first issue has been partially addressed in a number of works, including [2, 3, 10]. In this paper, we address this second issue and report on a package to provide effective support for reasoning about executable specifications in Coq. Further, we report on our positive experience in using this package to reason about the JavaCard platform; as explained in Section 3, the package leads to a decrease of a factor 10 in the size of proofs scripts.

In our opinion, JavaCard is a particularly interesting case study because of the complexity of the functions involved in the formalizations. Nevertheless, it should be stressed that the package is useful in any context where complex recursive definitions are used, and not only for reasoning about JavaCard or programming language semantics.

*Organization* The remaining of the paper is organized as follows: in Section 2, we describe the principles underlying the package, illustrate its working on simple examples and provide some implementation details. In Section 3, we report on its practical benefits in our work on JavaCard. In Section 4, we turn to related and further work. We conclude in Section 5.

## 2 Elimination principles for functions

### 2.1 Elimination on inductive types and properties

One of the powerful tools provided by proof assistants like Coq for reasoning on inductive types is the *elimination principle*, which is usually generated automatically from the definition of the type. We see here the definition in Coq of the type `nat` and the type of its associated principle:

```
Inductive nat : Set :=
    O : nat
  | S : nat→nat.

nat_ind:
  (P:(nat→Prop))
  (P O)
  →((n:nat)(P n)→(P (S n)))
  →(n:nat)(P n).
```

The logical meaning of an elimination principle attached to an inductive type $T$ is that closed normal terms of type $T$ have a limited set of possible forms, determined by the constructors of $T$. It also captures the recursive structure of the type. It is important to notice that elimination principles are nothing else than recursive functions, for example the definition of `nat_ind` is the following:

```
[P:(nat→Prop); f:(P O); f0:((n:nat)(P n)→(P (S n)))]
 Fixpoint F [n:nat] : (P n) :=
  Cases n of
      O ⇒f
   | (S n0) ⇒(f0 n0 (F n0))
  end}
```

Reasoning by elimination on an inductive *property* $P$ can be understood intuitively as reasoning by induction on the proof of $P$. We see here the principle associated to the relation `le`:

```
Inductive le [n : nat] : nat→Prop :=
     le_n : (le n n)
   | le_S : (m:nat)(le n m)→(le n (S m)).

le_ind:
    (n:nat; P:(nat→Prop))
    (P n)
    →((m:nat)(le n m)→(P m)→(P (S m)))
    →(m:nat)(le n m)→(P m).
```

Suppose we are trying to prove `(P m)` for all $m$ such that `H:(le n m)`, we can apply the theorem `le_ind`, which leads to two subgoals corresponding to the possible constructors of `le`. This way of reasoning, which has proved to be very useful in practice, makes relational specifications a popular solution.

### 2.2 Elimination following the shape of a function

When we choose a functional style of specification, recursive functions are used instead of inductive relations and elimination principles are usually not automatically available. However, once a function is defined (i.e. proved terminating which is automatic in Coq for structural recursion) it is relatively natural to build an induction principle, which follows its shape. It allows to *reason by induction on the possible branches* of the definition of the function. Since the new term follows the same recursive calls as the function, we know that this new term defines a correct induction scheme and that it will be accepted by Coq. For example the following function:

```
Fixpoint isfourtime [n:nat] : bool :=
 Cases n of
  | O ⇒true
  | (S S S S m) ⇒(isfourtime m)
  | _ ⇒false
 end.
```

yields an induction principle with a nested case analysis of depth 5. Redoing all the corresponding steps each time we want to consider the different paths of the function would be long and uninteresting (see Section 3 for more examples).

We have designed a Coq package allowing to automate the inductive/case analysis process. According to the Coq paradigm, our package builds *proof terms* in the Curry-Howard philosophy. Surprisingly the presence of proof terms is helpful: since functions and proofs are all represented by $\lambda$-terms, we can build proof terms by transforming the term of the function we consider. Since moreover we focus on functions defined by structural recursion (see section 4.2 for extension to more general recursion schemes), termination of the function (and consequently the correctness of the elimination principles we generate) is ensured by the type checking of Coq.

The proof of a property $\forall x.P$ using `isfourtime` as a model will be a term $Q$ obtained by transforming `isfourtime` (notice how implicit cases have been expanded):

```
Fixpoint Q [n:nat]: P :=
<[x:nat](P x)> Cases n of
  | O ⇒(?: (P O))
  | (S O) ⇒(?: (P (S O)))
  | (S S O) ⇒(?: (P (S S O)))
  | (S S S O) ⇒(?: (P (S S S O)))
  | (S S S S m) ⇒
        ((?:(x:nat)(P x)→(P (S S S S x))) m (Q m))
 end.
```

where the five `(?:H)` stand for properties `H` yet to be proved (subgoals).

The expression `<[x:nat](P x)>` is a specific type scheme for dependently typed case expressions. It allows to build case branches having different types, provided that each one corresponds to the scheme applied to the corresponding pattern. For example the first branch is correct because `([x:nat](P x) O)=(P O)`. Similarly the last case is accepted because `((?:(x:nat)(P x) → (P (S S S S x))) m (Q m))` has type `(P (S S S S m))`, which is `([x:nat](P x) (S S S S m))`.

The type of the whole case expression is obtained by applying the scheme to the term on which the case analysis is done, here `n`, i.e. `(P n)`.

We see here that subgoals contain nothing more than induction hypotheses, but generally this will not be enough. Indeed, we need to capture for each subgoal the *environment* induced by the branches of **Cases** expressions which have been chosen. We illustrate this situation with the following function (which computes for every $n$ the largest $m \leq n$ such that $m = 4k$ for some $k$):

```
Fixpoint findfourtime [n:nat]: nat :=
 Cases (isfourtime n) of
  | true ⇒n
  | false ⇒ Cases n of
            | (S m) ⇒(findfourtime m)
            | _ ⇒O (* impossible *)
             end
```

```
end.
```

In this case it is necessary to remember that in the first branch `(isfourtime n)` is equal to `true` and that in the second and third branches it is equal to `false`. Otherwise it will be impossible to prove for example the following property:

$$P \equiv \forall n : nat.\texttt{(isfourtime (findfourtime } n\texttt{))}.$$

Finally the proof of the elimination principle generated from `findfourtime` should have the form:

```
Fixpoint Q [n:nat]: (P n) :=
 (<[x:bool](isfourtime n)=x →(P n)>
 Cases (isfourtime n) of
   | true ⇒
      (?: (isfourtime n)=true→(P n))
   | false ⇒
      Cases n of
       | O ⇒
         (?:(isfourtime O)=false →(P O))
       | (S m) ⇒
          ((?:(P m) →(isfourtime n)=false →(P (S m))) (Q m))
      end
  end
  (reflequal nat (isfourtime n))).
```

where `(reflequal nat (isfourtime n))` is the proof of the different equalities induced by the case analysis[2].

These proofs are gathered outside the case expression so as to obtain a well typed term. This point is subtle: equality proofs have the form `(reflequal nat (isfourtime n))`, which is of type `(isfourtime n)=(isfourtime n)`. We see therefore that to be accepted by the case typing rule explained above, we must apply this term to a term of type `(isfourtime n)=(isfourtime n) → (P n)`. This is the type of the whole case expression, but not of each branch taken separately. So moving the `(reflequal nat (isfourtime n))` inside the case expression would result in an ill-typed term.

### 2.3 Inference system

In this subsection, we give an inference system to build an elimination principle for a particular property $G$ from a function $t$. The main goal of this system is to reproduce the structure of the term $t$, including **`Case`**, **`Fix`**, **`let`** and $\lambda$-abstractions, until it meets an application, a constant or a variable. The remaining terms are replaced by proof obligations whose type is determined by the **`Case`**, **`Fix`**, **`let`** and $\lambda$-abstractions from above.

[2] `reflequal` is the constructor of the inductive type `eq` representing the equality: `Inductive eq [A:Set; x:A]: A->Prop := reflequal : (eq A x x).`

**Grammar** Figure 1 presents the grammar of expressions. For the of the presentation, it is a simplified version of the expressions of the Calculus of Inductive Constructions (CIC) where mutual recursive functions are ommitted. Since our implementation operates on Coq terms, we use the internal representation of Coq for fixpoints, that splits the elimination on inductive types into constructions **`Fix`** and **`Cases`**. **`Fix`**$(X, t, f)$ is a binder for $X$ in $f$, $X$ corresponds to the function being defined by **`Fix`**, $t$ is its type and $f$ is the body of the function. The construction $Ind(x : T)\{\boldsymbol{T}\}$ is an inductive type. $C_{n,T}$ is the nth constructor of inductive type $\boldsymbol{T}$, we shall leave $T$ implicit when possible. In the **`Case`** expression $t_i$'s are functions taking as many arguments as the corresponding constructors take, see [20, 21].

$$
\begin{aligned}
\text{Variables} : V &::= x, y \ldots \\
\text{Sorts: } S &::= Set \ \mid Prop \mid Type \\
\text{Terms: } T &::= V \mid S \mid \lambda V : T.T \mid \forall V : T.T \mid (T\ T) \\
&\mid \mathtt{\textbf{Fix}}(V, T, T) \mid \mathtt{\textbf{Case}}\ T\ \mathtt{\textbf{of}}\ \boldsymbol{T}\ \mathtt{\textbf{end}} \mid \mathtt{\textbf{let}}\ T = T\ \mathtt{\textbf{in}}\ T \\
&\mid Ind(x : T)\{\boldsymbol{T}\} \mid C_{n,T}
\end{aligned}
$$

**Fig. 1.** Syntax of terms of CCI

**Judgment** Judgments are of the form: $t, X, G, \Gamma \vdash P$ where $t$ is the function to use as a model, $X$ is the variable corresponding to the recursive function (bound by **`Fix`** in $t$, used to find recursive calls), $G$ is the property to be proved (the goal), $\Gamma_1$ is the list of bounded variables (initially empty), $\Gamma_2$ is the list of equalities corresponding to the case analysis, and $P$ is the proof term of the principle. When proving a property $G$, we build a term $P$ containing proof obligations represented by $(? : T)$ where $T$ is the awaited property (i.e. type) that must be proved.

**Rules** are given in Figure 2. In the last rule, $(\boldsymbol{X}\ \boldsymbol{t_i})$ represents all recursive calls found in $t$. In the present implementation, no nested recursion is allowed. In the rule (Case), we note $(C_i\ \boldsymbol{x_i})$ the fully applied constructor $C_i$.

The main tools provided by our package are:

- A Coq *command* **`Functional Scheme`** which builds a general induction principle from the definition of a function $f$. It is a theorem of the form:

  $$(P : (\forall \boldsymbol{x_i} : \boldsymbol{T_i}.Prop))(H_1 : PO_1)...(H_n : PO_n) \rightarrow \forall \boldsymbol{x_i} : \boldsymbol{T_i}.(P\ \boldsymbol{x_i})$$

  where $PO_i$ are the proof obligations generated by the algorithm described above, and $\boldsymbol{x_i}$ correspond to the arguments of $f$. To make an elimination, the user just applies the theorem. The advantage of this method is that the structure of the function (and its type checking) is not duplicated each time we make an elimination;

$$\frac{t, X,\ G,\ x : T \cup \Gamma_1,\ \Gamma_2 \vdash P}{\lambda x : T.t,\ X,\ G,\ \Gamma_1,\ \Gamma_2 \vdash \lambda x : T.P}\text{(Lambda)}$$

$$\frac{\forall i.\{t_i, X,\ G,\ \boldsymbol{x_i} : \boldsymbol{T_i} \cup \Gamma_1,\ E = (C_i\ \boldsymbol{x_i}) \cup \Gamma_2 \vdash P_i\}}{\begin{array}{l}\texttt{Case}\ E\ \texttt{of}\ \boldsymbol{t_i}\ \texttt{end},\ X,\ G,\ \Gamma_1,\ \Gamma_2 \\ \qquad\qquad\qquad \vdash (\texttt{Case}\ E\ \texttt{of}\ \boldsymbol{P_i}\ \texttt{end}\ (\texttt{reflequal}\ T_E\ E))\end{array}}\text{(Case)}$$

$$\frac{t, X,\ G,\ \boldsymbol{x_i} : \boldsymbol{T_i} \in \boldsymbol{u} \cup \Gamma_1,\ u = v \cup \Gamma_2 \vdash P}{\texttt{let}\ u = v\ \texttt{in}\ t,\ X,\ G,\ \Gamma_1,\ \Gamma_2 \vdash \texttt{let}\ u = v\ \texttt{in}\ P}\text{(Let)}$$

$$\frac{f, X, G, \Gamma_1, \Gamma_2 \vdash P}{\texttt{Fix}(X, T, f),\ \_,\ G,\ \Gamma_1,\ \Gamma_2 \vdash \texttt{Fix}(X, G, P)}\text{(Fix)}$$

$$\frac{(\boldsymbol{X}\ \boldsymbol{t_i}) \in t}{t, X, G, \Gamma_1, \Gamma_2 \vdash ((? : \forall \Gamma_1.\boldsymbol{X}\ \boldsymbol{t_i} \to \Gamma_2 \to G)\ \Gamma_1)}\text{(Rec)}$$
$$\text{if } t \neq \texttt{Fix},\ \texttt{Case}, \texttt{let} \text{ or } \lambda x : T.t.$$

**Fig. 2.** Elimination algorithm

– A *tactic* **`Analyze`** which applies the above algorithm to a particular goal. This second tactic allows more automation, see next section.

### 2.4 More automation

We can use the equalities generated by the tactic to perform rewriting steps automatically, thereby reducing the user interactions to achieve the proof. This is possible on two levels:

– first we can rewrite in the generated hypothesis of each branch. We give a modified version of the (Case) rule in figure 3, where $\Gamma_2[E \leftarrow C_i(\boldsymbol{x_i})]$ is the set of equalities $\Gamma_2$ where $E$ has been replaced by $C_i(\boldsymbol{x_i})$ in the right members.

$$\frac{\forall i.\{t_i[E \leftarrow C_i(\boldsymbol{x_i})], X,\ G,\ \boldsymbol{x_i} : \boldsymbol{T_i} \cup \Gamma_1,\ E = (C_i\ \boldsymbol{x_i}) \cup \Gamma_2[E \leftarrow C_i(\boldsymbol{x_i})] \vdash P_i\}}{\begin{array}{l}\texttt{Case}\ E\ \texttt{of}\ \boldsymbol{t_i}\ \texttt{end},\ X,\ G,\ \Gamma_1,\ \Gamma_2 \\ \qquad\qquad\qquad \vdash (\texttt{Case}\ E\ \texttt{of}\ \boldsymbol{P_i}\ \texttt{end}\ (\texttt{reflequal}\ T_E\ E))\end{array}}\text{(Case)}$$

**Fig. 3.** (Case) rule with rewriting

This rewriting is applied for both the command and the tactic;

- second we can also propagate these rewriting steps in the goal itself. This is possible in the tactic **`Analyze`**, where we call the **`Rewrite`** tactic of Coq, which performs substitutions in the goal, with the set of equalities $\Gamma_2$.
  As Coq provides a mechanism for folding/unfolding constants, it is possible that rewriting steps become possible later during the proofs of the generated subgoals. This is specially true when dealing with complex specifications where unfolding all definitions is not comfortable (unfolding all definitions in our applications to JavaCard leads to goals of over 8000 lines). Therefore we also provide a rewriting database that contains all the equalities of $\Gamma_2$ for each branch. The database can be used later with the **`AutoRewrite`** tactic.

## 3 Applications to JavaCard

In this section, we illustrate the benefits of our package in reasoning about the JavaCard platform. More specifically, we show how the package has been used to good effect for establishing the correctness of the JavaCard bytecode verifier, and the preservation of certain memory invariants during a program's execution. For the sake of readability, we begin with a brief review of our work on the JavaCard platform.

### 3.1 Background

*Smartcards* Open platform smartcards are small devices designed to integrate multiple applications on-board, for example identity documents, electronic purses, loyalty applets and health records. Such applications are intended to coexist on the same card and communicate securely, putting security issues at stake [14][3].

*JavaCard* [17, 16] is a dialect of Java tailored towards smartcard programming, and the most common platform for multi-application smartcards. Once compiled, JavaCard programs are verified by a bytecode verifier and loaded on the card, where they can be executed by the JavaCard Virtual Machine.

*CertiCartes: a formal executable semantics of JavaCard* In our previous work on CertiCartes, we give a formal executable semantics of the JavaCard platform in Coq. With over 15,000 lines of Coq scripts, our formalization constitutes the most in-depth machine-checked account of the JavaCard platform to date.

CertiCartes contains formal specifications of:

- a defensive JavaCard virtual machine, i.e. a virtual machine where values are tagged by their type and where typing information is verified at run-time;
- an offensive JavaCard virtual machine, i.e. a virtual machine which relies on successful bytecode verification to eliminate type verification at run-time and where values are untyped;

[3] Furthermore applications can in principle be loaded on the card after its issuance to users, which adds further security threats.

– an abstract JavaCard virtual machine, i.e. a virtual machine where values are types, and which is used in the construction of the bytecode verifier—the bytecode verifier is constructed as a data-flow analyzer based on the abstract virtual machine.

All specifications are written in an executable style and model one-step execution of the virtual machines. Formally, the defensive (resp. offensive, abstract) virtual machine is defined as a function mapping a defensive (resp. offensive, abstract) state to a returned defensive (resp. offensive, abstract) state—the result of one-step execution is tagged to witness normal or abrupt termination.

Further, CertiCartes includes a correctness proof for our bytecode verifier, using a strategy streamlined in [11, 18, 24] and further refined and expanded in [5, 6]. The strategy proceeds in three steps:

– show that the abstract virtual machine is a sound abstraction of the defensive virtual machine, as illustrated by the diagram in Figure 4, where `dexec` denotes one-step execution of the defensive virtual machine, `aexec` denotes one-step execution of the abstract virtual machine (the result is a list of abstract states because of the non-determinism introduced by forgetting about values), and `alpha_abs` is the function mapping states to abstract states (by projecting values to types);
– similarly, show that for programs that do not raise a type error, the offensive abstract virtual machine is a sound abstraction of the defensive virtual machine, as illustrated by the diagram in Figure 5, where `oexec` denotes one-step execution of the offensive virtual machine, and `alpha_off` is the function mapping states to offensive states (by omitting types from values);
– conclude that the offensive and defensive virtual machines coincide on those programs that pass bytecode verification.

$$\begin{tikzcd}
\texttt{state} \arrow[r, "\texttt{dexec}"] \arrow[d, "\texttt{alpha_abs}"] & \texttt{returned_state} \arrow[d, "\preceq\texttt{alpha_abs_rs}"] \\
\texttt{abs_state} \arrow[r, "\texttt{aexec}"] & \texttt{(list abs_returned_state)}
\end{tikzcd}$$

**Fig. 4.** Commutative diagram of defensive and abstract execution

### 3.2 Applications to proving the correspondence between virtual machines

In this subsection, we illustrate the usefulness of the package in proving that the diagrams in Figures 5 and 4 commute. In order to establish the commutation property, one needs to prove that abstraction commutes with execution

$$
\begin{tikzcd}
\texttt{state} \arrow[r, "\texttt{dexec}"] \arrow[d, "\texttt{alpha\_off}"] & \texttt{returned\_state} \arrow[d, "\texttt{alpha\_off\_rs}"] \\
\texttt{off\_state} \arrow[r, "\texttt{oexec}"] & \texttt{off\_returned\_state}
\end{tikzcd}
$$

**Fig. 5.** Commutative diagram of defensive and offensive execution

for every bytecode of the JavaCard instruction set. In the particular case of the commutation of Figure 5 for method invocation, one has to show in Coq:

```
Lemma INVOKEVIRTUAL_commut :
let res= (exec_instruction (invokevirtual nargs nm) state cap) in
let ostate=(alpha_off state) in
let ores=(oexec_instruction (invokevirtual nargs nm) ostate cap)
in (INVOKEVIRTUAL_conds res) →(alpha_off_rs res) = ores
```

where the predicate `INVOKEVIRTUAL_conds` ensures that the result state `res` is not a type error. Note that the functions defining the (defensive and offensive) semantics of method invokation require many case distinctions, as illustrated below by the definition of the function `INVOKEVIRTUAL` for defensive method invokation:

```
Definition INVOKEVIRTUAL :=
[nargs:nat][nm:class_method_idx][state:jcvm_state][cap:jcprogram]

(* The initial state is decomposed *)
Cases state of
(sh, (hp, nil)) ⇒(AbortCode state_error state) |
(sh, (hp, (cons h lf))) ⇒

(* nargs must be greater than zero *)
 Cases nargs of
 O    ⇒(AbortCode args_error state) |
 (S _) ⇒

  (* Extraction of the object reference (the nargsth element) *)
  Cases (Nth_func (opstack h) nargs) of
  error ⇒(AbortCode opstack_error state) |
  (value x) ⇒

   (* Tests if this element is a reference *)
   Cases x of
   ((Prim _), vx) ⇒(AbortCode type_error state) |
   ((Ref _), vx) ⇒

    (* tests if the reference is null *)
    (if (test_NullPointer vx)
```

```
        then (ThrowException NullPointer state cap)
        else

          (* Extraction of the referenced object *)
          Cases (Nth_func hp (absolu vx)) of
          error ⇒(AbortMemory heap_error state) |
          (value nhp) ⇒

            (* Get the corresponding class *)
            Cases (Nth_elt (classes cap) (get_obj_class_idx nhp)) of
            (value c) ⇒

              (* Get the corresponding method *)
              Cases (get_method c nm) of
              (* Successful method call *)
              (value m) ⇒(new_frame_invokevirtual nargs m nhp state cap) |
              error ⇒(AbortCap methods_membership_error state)
              end |
             _ ⇒(AbortCap class_membership_error state)
            end
          end
       )
      end
    end
  end
end.
```

This function is fairly representative of the CertiCartes formalization: it involves (1) decomposing the initial state and retrieving relevant fragments `F`—many partial functions are used at this stage—(2) performing some checks `O` on `F` (3) constructing the final state on the basis of `F` and `O`.

In our initial work on CertiCartes, such proofs were performed by successive unfoldings of the case analyses arising in the definition of `INVOKEVIRTUAL` leading to cumbersome proofs which were hard to produce, understand and modify. In contrast, the **Analyze** tactic leads to smaller proofs that are easier to perform and understand. For example, the following script provides the first steps of the proof:

```
Unfold exec_instruction.
Intros nargs nm state cap.
Analyze INVOKEVIRTUAL params nargs nm state cap.
```

At this point, nine different subgoals are generated, each of them corresponding to a full case analysis of the function. Initially, each subgoal is of the form:

```
l1 = r1 ∧ ... ∧ ln= rn →G
```

but, as explained in Section 2, the tactic performs an introduction step that shifts all equalities to the context and then rewrites all equalities in the remaining goal `G`. In the case of a successful method call, the corresponding subgoal is:

```
  (alpha_off_rs
   (new_frame_invokevirtual (S n) m nhp (sh,(hp,(cons h lf))) cap))
  =(oexec_instruction (invokevirtual (S n) nm)
    (alpha_off_rs (sh,(hp,(cons h lf)))) cap)
```

This goal, as all other goals generated by the tactic, can in turn be discharged using rewriting and some basic assumptions on the structure of the program (note that these assumptions do not impose any restriction on the JavaCard programs considered, but capture invariants that are enforced by the JavaCard specification).

### 3.3 Applications to proving memory invariants

Proving the commutation of the diagram in Figure 4 requires, in addition to reasoning by cases and equational reasoning, to establish a number of invariants on the memory of the virtual machine. Typically, these invariants are of the form: if a property $P$ holds for every element in the stack (resp. heap) before execution of the abstract machine, then it holds for every element in the stack (resp. heap) after execution of the abstract machine. Proving such properties is tedious and involves numerous case analyses. Consider for example the following lemma which states that the size of JavaCard types recorded in memory is bounded during execution of the abstract virtual machine:

```
Lemma tm_cap_rs_pred :
(i:Instruction)(s:abs_state) (rs:abs_returned_state)
(In rs (aexec i s)) →(boundedtypes s) →(boundedtypes_ret rs).
```

To prove the lemma, we need to proceed with a case analysis on the instruction set. In the case of the `load` bytecode, one is left with the goal:

```
 s : abs_state
 t : type
 l : locvars_idx
 ============================
 (boundedtypes s) →  (boundedtypes_ret (tload t l s))
```

Next one proceeds with a case analysis on the definition of `tload`, using the **Analyze** tactic. This generates 29 subgoals, out of which 26 are blown automatically by the tactic subsequently, leaving the user to prove three interesting cases corresponding to the successful execution of the `tload` bytecode. More generally, the use of this tactic in the Certicartes formalization allowed to reduce the size of proof scripts by a factor of 10.

## 4 Extensions and related work

### 4.1 Related work

Our work is most closely related to Slind's work on reasoning about terminating functional programs, see e.g. [25] where Slind presents a complete methodology to define a function from a list of equations and a *termination relation*. This relation is used to generate *termination conditions*, which need to be discharged to prove that the rewrite system defined by the equations terminates. From the proofs of the termination conditions, Slind automatically synthesizes an induction principle for the function. Slind's induction principle is closely related to ours but there are some differences:

- Slind's work focuses on proof-assistants based on higher-order logic, in particular on Isabelle [22] and HOL [15], that do not feature proof terms. In our framework, objects, properties and proofs are all terms of the Calculus of Inductive Constructions, and hence we need to provide a proof term for the customized principle attached to the function. As pointed out in Section 2, the construction of the proof term can be done directly by transforming the term corresponding to the function;
- Slind's work provides for each total and terminating function `f` a generic induction principle that can be used to prove properties about `f`. In contrast, the tactic **`Analyze`** starts from the property to be proven, say $\phi$, and proceeds to build directly a proof of $\phi$ by following the shape of the definition of `f`. Eventually the tactic returns some proof obligations that are required to complete the proof of $\phi$. This allows our package to perform rewriting and other operations during the building of the proof, leading to a drastic increase in automation. Since similar proof terms are constructed every time one applies the tactic on the function `f`, it could be argued that such a feature is costly in terms of memory. However, we have not experienced any efficiency problem although our JavaCard development is fairly large;
- Slind's work provides support for well-founded and nested recursion, which our package does not handle currently.

Slind's work, as well as ours, aims at providing efficient support for reasoning about executable specifications. An alternative strategy consists in developing tools for executing specifications that may be relational, at least in part. For example, Elf [23] combines type theory and logic programming and hence provides support for executing relational specifications. Further, there have been some efforts to develop tools for executing specifications written in Coq, Isabelle or related systems. In particular some authors have provided support for executing relational specifications: for example Berghofer and Nipkow [7] have developed a tool to execute Isabelle theories in a functional language, and used it in the context of their work on Java and JavaCard. Conversely, some authors have provided support to translate executable specifications into input for proof assistants: for example, D. Terrasse [27, 28] has developed a tool to translate Typol specifications in Coq. However, we believe that executable specifications are better suited for automating proofs, especially because they lend themselves well to automated theorem-proving and rewriting techniques.

A third possible strategy would consist in relating relational and functional specifications. For example, one could envision a tool that would derive automatically from a functional specification (1) its corresponding relational specification (2) a formal proof that the two specifications coincide. Such a tool would allow to combine the best of both worlds (one could be using elimination/inversion principles to reason about relational specifications, and then transfer the results to functional ones); yet we are not aware of any conclusive experience in this direction.

### 4.2 Extensions

As emphasized in the introduction, the main motivation behind the package is to develop effective tool support to specify and reason about the JavaCard platform. The following two extensions are the obvious next steps towards this objective:

- *support for conditional rewriting rules.* We would like to introduce functions using conditional rewriting rules in the spirit of [4]. Indeed, such definitions, which can be automatically translated into legal function declarations in Coq (provided the guard conditions are satisfied), lead to more readable specifications, especially for partial functions. It would also be interesting to combine the algorithms described in [4] with recent proposals to support partial functions in type theory, see e.g. [29].
- *automation of equational reasoning.* In order to achieve a higher degree of automation in the proofs of commuting diagrams, we need to automate equational reasoning in Coq further. We are currently considering the possibility of relying on an external tool to generate rewriting traces that are then translated in Coq. There are several candidates for such tools, including Elan [8] and Spike [9], and ongoing work to interface them with proof assistants such as Coq, see e.g. [1].

From a more general perspective, we are interested in the following extensions:

- *support for well-founded recursion.* We would like to introduce functions that are defined by well-founded recursion (whereas currently we only consider functions that are defined by structural recursion). Ongoing work by A. Balaa and Y. Bertot [2, 3] could help extending the package in this direction, but other recent work, see e.g. [10], could also prove useful there. In addition, it would be interesting to link Coq to external tools that can help synthesizing automatically the termination relation (and possibly its proof of well-foundedness).
- *automation of inversion principles.* We also would like to provide inversion principles like those that exist for inductive types [12]. Considering for example the property `(isfourtime x)=true`, we want to be able to deduce automatically that `x=O` or `x=(S S S S y)`$\wedge$`((isfourtime y)=true)`. Such principles do not use induction, but proceed by case analysis and use the fact that constructors of inductive types are injective.

## 5 Conclusion

We have described a package to automated reasoning about executable specifications in Coq. Although the primary motivation for this work comes from our earlier work on verifying the JavaCard platform, we believe the package is of general interest for any Coq user that manipulates complex recursive definitions. More generally, preliminary experiments with CertiCartes in Isabelle and PVS suggest that the combination of case analysis and rewriting performed by the **`Analyze`** tactic would be useful in other proof assistants.

In addition, we have illustrated the usefulness of our package in reasoning about the JavaCard platform. This positive experience supports the claim laid in [4] that proofs of correctness of the JavaCard platform can be automated to a large extent and indicates that proof assistants could provide a viable approach to validate novel type systems for future versions of Javacard.

## References


1. C. Alvarado and Q-H. Nguyen. ELAN for equational reasoning in COQ. In J. Despeyroux, editor, *Proceedings of LFM'00*, 2000. Rapport Technique INRIA.
2. A. Balaa and Y. Bertot. Fix-point equations for well-founded recursion in type theory. In M. Aagaard and J. Harrison, editors, *Proceedings of TPHOLS'00*, volume 1689 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, 2000.
3. A. Balaa and Y. Bertot. Fonctions récursives générales par itération en théorie des types. In L. Rideau, editor, *Proceedings of JFLA'02*, 2002.
4. G. Barthe, G. Dufay, M. Huisman, and S. Melo de Sousa. Jakarta: a toolset to reason about the JavaCard platform. In I. Attali and T. Jensen, editors, *Proceedings of e-SMART'01*, volume 2140 of *Lecture Notes in Computer Science*, pages 2–18. Springer-Verlag, 2001.
5. G. Barthe, G. Dufay, L. Jakubiec, and S. Melo de Sousa. A formal correspondence between offensive and defensive JavaCard virtual machines. In A. Cortesi, editor, *Proceedings of VMCAI'02*, volume 2xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. To appear.
6. G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In D. Sands, editor, *Proceedings of ESOP'01*, volume 2028 of *Lecture Notes in Computer Science*, pages 302–319. Springer-Verlag, 2001.
7. S. Berghofer and T. Nipkow. Executing higher order logic. In P. Callaghan, Z. Luo, J. McKinna, and R. Pollack, editors, *Proceedings of TYPES'00*, volume 2xxx of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. To appear.
8. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *The Elan V3.4. Manual*, 2000.
9. A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23:47–77, 1997.
10. A. Bove and V. Capretta. Nested general recursion and partiality in type theory. In R. J. Boulton and P. B. Jackson, editors, *Proceedigns of TPHOLs'01*, volume 2152 of *Lecture Notes in Computer Science*, pages 121–135. Springer-Verlag, 2001.
11. L. Casset and J.-L. Lanet. A Formal Specification of the Java Byte Code Semantics using the B Method. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Proceedings of Formal Techniques for Java Programs*. Technical Report 251, 1999, Fernuniversität Hagen, Fernuniversität Hagen, 1999.
12. C. Cornes. *Conception d'un langage de haut niveau de representation de preuves: Récurrence par filtrage de motifs; Unification en présence de types inductifs primitifs; Synthèse de lemmes d'inversion*. PhD thesis, Université de Paris 7, 1997.

13. C. Cornes and D. Terrasse. Automating inversion and inductive predicates in Coq. In S. Berardi and M. Coppo, editors, *Proceedings of Types'95*, volume 1158 of *Lecture Notes in Computer Science*, pages 85–104. Springer-Verlag, 1995.
14. European Commission Information Society Directorate General. Research for the smart card of 2010, 2001. Available from `ftp://ftp.cordis.lu/pub/ist/docs/ka2/`.
15. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher-order logic.* Cambridge University Press, 1993.
16. JavaCard Forum. `http://www.javacardforum.org`
17. JavaCard Technology. `http://java.sun.com/products/javacard`
18. J.-L. Lanet and A. Requet. Formal Proof of Smart Card Applets Correctness. In J.-J. Quisquater and B. Schneier, editors, *Proceedings of CARDIS'98*, volume 1820 of *Lecture Notes in Computer Science*, pages 85–97. Springer-Verlag, 1998.
19. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, University of Edinburgh, May 1992.
20. C. Paulin-Mohring. Inductive definitions in the system Coq. Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of TLCA'93*, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer-Verlag, 1993.
21. C. Paulin-Mohring. *Définitions Inductives en Theorie des Types d'Ordre Superieur.* Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996.
22. L. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
23. F. Pfenning. Elf: a meta-language for deductive systems. In A. Bundy, editor, *Proceedings of CADE-12*, volume 814 of *Lecture Notes in Artificial Intelligence*, pages 811–815. Springer-Verlag, 1994.
24. A. Requet. A B Model for Ensuring Soundness of a Large Subset of the Java Card Virtual Machine. In S. Gnesi, I. Schieferdecker, and A. Rennoch, editors, *Proceedings of FMICS'00*, pages 29–46, 2000.
25. K. Slind. *Reasoning about Terminating Functional Programs.* PhD thesis, TU Münich, 1999.
26. The Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.2*, January 2002.
27. D. Terrasse. Encoding natural semantics in Coq. In V. S. Alagar, editor, *Proceedings of AMAST'95*, volume 936 of *Lecture Notes in Computer Science*, pages 230–244. Springer-Verlag, 1995.
28. D. Terrasse. *Vers un environnement d'aide au développement de preuves en Sémantique Naturelle.* PhD thesis, Ecole Nationale des Ponts et Chaussées, 1995.
29. F. Wiedijk and J. Zwanenburg. First order logic with domain conditions. Manuscript, 2001.