Efficient reasoning about executable specifications in Coq

Gilles Barthe and Pierre Courtieu

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

Abstract. We describe a package to reason efficiently about executable specifications in Coq. 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) of the size of the proofs in a large-scale case study on reasoning about JavaCard.

1 Introduction

Proof assistants based on type theory, such as Coq [9] and Lego [15], 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:

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

In our opinion, the functional style has some distinctive advantages over its relational counterpart, especially for formalizing complex programming languages. In particular, the functional style 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:

- 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 more cumbersome than their relational counterpart;

– secondly, a proof assistant like Coq offers, through a set of inversion [10, 11] and elimination tactics, effective support to reason about relational specifications. In contrast, there is no similar level of support for reasoning about executable functions.

Here we address this second issue by giving a package that provides effective support for reasoning about complex recursive definitions, see Section 2. Further, we illustrate the benefits of the package in reasoning about executable specifications of the JavaCard platform [3, 4], and show how its use yields compact proofs scripts, up to 10 times shorter than proofs constructed "by hand", see Section 3. Related work is discussed in Section 3. 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) :=
<[x:nat](P x)>  Cases n of      O ⇒ f
                           | (S n0) ⇒ (f0 n0 (F n0))
  end}
```

The expression `<[x:nat](P x)>` is a specific type scheme for dependently typed case expressions (dependent types are necessary to represent predicates like `P`). It is used 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 second case is accepted because `(f0 n0 (F n0))` has type `(P (S n0))`, which is equal to `([x:nat](P x) (S n0))`.

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)`.

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

Basic idea 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 *reasoning 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 automation of 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 λ-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). In Subsection 2.5 we concentrate on the structure of this incomplete term and its automatic generation from the definition of the function.

Once built, this incomplete term can be applied to a particular goal with the **`Refine`** tactic, or completed into a general reusable induction principle. To achieve the latter, we replace the `?`'s by abstracted variables (`Hypx` in the example below). We also abstract the predicate (`P`) and finally obtain a general principle. On our example this leads to:

```
[P:(nat→Prop); Hyp0:(P O);  Hyp1:(P (S O));
 Hyp2:(P (S (S O))); Hyp3:(P (S (S (S O))));
 Hyp4:(x:nat) (P x) → (P (S (S (S (S x))))))]
Fixpoint Q [n:nat]: P :=
<[x:nat](P x)> Cases n of
  | O ⇒  Hyp0
  | (S O) ⇒ Hyp1
  | (S S O) ⇒ Hyp2
  | (S S S O) ⇒ Hyp3
  | (S S S S m) ⇒ Hyp4 m (Q m)
  end
     : (P:(nat→Prop)) (P O) →  (P (S O)) → (P (S (S O)))
     → (P (S (S (S O)))) →   ((x:nat) (P x) → (P (S (S (S (S x))))))
     →(n:nat)(P n)
```

This term is well-typed and thus defines a correct induction principle.

Capturing the environment of each branch In the examples above 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 n of
 | O ⇒ O
 | (S m) ⇒
   Cases (isfourtime n) of
   | true ⇒ n
   | false ⇒ (findfourtime m)
   end
 end.
```

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

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

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

```
Fixpoint Q2 [n:nat]: (P n) :=
(<[x:bool](isfourtime n)=x → (P n)>
Cases n of
 | O ⇒ (?:(P O))
 | (S m) ⇒
   <[b:bool](isfourtime (S m))=b→(P (S m))>
   Cases (isfourtime (S m)) of
   | true ⇒ (?:(isfourtime (S m))=true→(P (S m)))
   | false ⇒ (?:(P m)→(isfourtime (S m))=false→(P (S m))) (Q2 m)
   end (refl_equal bool (isfourtime (S m))))
end
```

where `(reflequal bool (isfourtime (S m)))` is the proof of the different equalities induced by the case analysis. Recall that `reflequal` is the constructor of the inductive type `eq` representing the equality, so `{(reflequal A x)}` is of type `(eq A x x)`.

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 bool (isfourtime (S m)))`, which is of type `(isfourtime (S m))=(isfourtime (S m))`. 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 (S m))=(isfourtime (S m))→(P (S m))`. This is the type of the whole case expression, but not of each branch taken separately. So moving the `reflequal` expression inside the `Cases` would result in an ill-typed term.

2.3 Contents of the package

The main components of the 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 the PO_i's are the proof obligations generated by the algorithm described above, and $\boldsymbol{x_i}$'s 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;
- A *tactic* **`Analyze`** which applies the above algorithm to a particular goal. This tactic allows for more automation, see for that section 2.6.

2.4 Using the package on an example

We now prove the following property:

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

Using the tactic ***Analyze****.* In order to benefit from the rewriting steps of the tactic, we first unfold the definition of the function, and then apply the tactic:

```
Lemma P:(n:nat)(le (findfourtime n) n).
Intro n. Unfold findfourtime.
Analyze findfourtime params n.
```

At this point we have the following three subgoals corresponding to the three branches of the definition of `findfourtime`, notice the induction hypothesis on the last one:

```
1: (le O O)
2: [(isfourtime (S m))=true] ⊢ (le (S m) (S m))
3: [(le (findfourtime m) m); (isfourtime (S m))=false]
     ⊢ (le (findfourtime m) (S m))
```

Each of these subgoals is then proved by the **Auto** tactic.

Using the general principle. We set:

```
Functional Scheme findfourtime_ind := Induction for findfourtime.
```

Then the proof follows the same pattern as above:

```
Lemma Q':(n:nat)(le (findfourtime n) n).
Intro n. Unfold findfourtime.
Elim n using findfourtime_ind.
```

Again we have three subgoals:

```
1: (le O O)
2: [eq:(isfourtime (S m))=true]
     ⊢ (le (if (isfourtime (S m))
           then (S m) else (findfourtime m)) (S m))
3:[(le (findfourtime m) m); eq:(isfourtime (S m))=false]
     ⊢ (le (if (isfourtime (S m))
           then (S m) else (findfourtime m)) (S m))
```

We see that some rewriting steps must be done by hand to obtain the same result than with the tactic. Finally the whole script is the following:

```
Lemma Q':(n:nat)(le (findfourtime n) n).
Intro n. Unfold findfourtime.
Elim n using findfourtime_ind;Intros;Auto;Rewrite eq;Auto.
Save.
```

In more complex cases, like in section 3.2, the rewriting operations done by the **Analyze** tactic make the script significantly smaller than when using the general principle.

2.5 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 λ-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 λ-abstractions from above. The result of this algorithm is an incomplete term, like `Q2` above, from which we can either apply the **`Refine`** tactic or build a general principle.

Grammar Coq's internal representation of the expressions of the Calculus of Inductive Constructions (CIC) is given in Figure 1. Although our package deals with mutually recursive functions, we omit them from the presentation for clarity reasons. The construction $Ind(x : T)\{\boldsymbol{T}\}$, where $\boldsymbol{T}$ stands for a vector of terms (types actually), is an inductive type whose n-th constructor is denoted by $C_{n,T}$ (in the sequel we often omit T). In the **`Fix`** expression X is bound in f and corresponds to the function being defined, t is its type and f is the body of the function. In the **`Case`** expression t_i's are functions taking as many arguments as the corresponding constructors take, see [16, 17].

Judgment Judgments are of the form: $t, X, G, \Gamma_1, \Gamma_2 \vdash P$ 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), Γ_1 is the list of bound variables (initially empty), Γ_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, $(\boldsymbol{X\ t_i})$ 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.

2.6 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.

- 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 Γ_2 where E has been replaced by $C_i(\boldsymbol{x_i})$ in the right members.

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

Fig. 1. Syntax of terms of CCI

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

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

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

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

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

Fig. 2. Elimination algorithm

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

Fig. 3. (Case) rule with rewriting

– Second, we can 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 Γ_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. Therefore we also provide a rewriting database that contains all the equalities of Γ_2 for each branch. The database can be used later with the **AutoRewrite** tactic.
– Third, we can optimize the **Analyze** tactic in the particular case of a non recursive function applied to constructor terms, i.e. terms of which head symbol is a constructor of an inductive type. For example, suppose we have a goal of the form:

```
(n:nat)(P (f O (S (S (S n)))))
```

Where `f` is a non recursive function. It is clear that we do not want to consider all possible constructors for the arguments of `f`. For example the case `(f (S ...) ...)` is not interesting for us. The idea is to focus on the case branch corresponding to the constructors. We use a simple trick to achieve this: we apply the function to its arguments and reduce. The resulting term is a new function containing only the relevant case analysis. We apply the tactic to this term instead of the initial function.
Note that this optimization is hardly useful for recursive functions because some recursive hypothesis can be lost when focusing on a particular branch.

3 Applications to JavaCard

In this section, we illustrate the benefits of our package in establishing correctness results for the JavaCard platform. Note that for the clarity of presentation, the Coq code presented below is a simplified account of [3, 4].

3.1 Background

JavaCard is a dialect of Java tailored towards programming multi-application smartcards. Once compiled, JavaCard applets, typically electronic purses and loyalty applets, are verified by a bytecode verifier (BCV) and loaded on the card, where they can be executed by a JavaCard Virtual Machine (JCVM).

Correctness of the JavaCard platform The JCVM comes in two flavours: a defensive JCVM, which manipulates typed values and performs type-checking at run-time, and an offensive JCVM, which manipulates untyped values and relies on successful bytecode verification to eliminate type verification at run-time. Following a strategy streamlined in [14], we want to prove that the offensive and defensive VMs coincide on those programs that pass bytecode verification. This involves:

– formalizing both VMs, and show that both machines coincide on those programs whose execution on the defensive VM does not raise a type error;
– formalizing a BCV as a dataflow analysis of an abstract VM that only manipulates types and show that the defensive VM does not raise a type error for those programs that pass bytecode verification.

In both cases, we need to establish a correspondence between two VMs. More precisely, we need to show that the offensive and abstract VMs are sound (non-standard) abstract interpretations of the defensive VMs, in the sense that under suitable constraints "abstraction commutes with execution".

CertiCartes [3, 4] is an in-depth feasibility study in the formal verification of the JavaCard platform. CertiCartes contains formal executable specifications of the three VMs (defensive, abstract and offensive) and of the BCV, and a proof that the defensive and offensive VM coincide on those programs that pass bytecode verification. The bulk of the proof effort is concerned with the soundness of the offensive and abstract VMs w.r.t. the defensive VM. In our initial work, such proofs were performed by successive unfoldings of the case analyses arising in the definition of the semantics of each bytecode, leading to cumbersome proofs which were hard to produce, understand and modify. In contrast, the **Analyze** tactic leads to (up to 10 times) smaller proofs that are easier to perform and understand, as illustrated in the next subsections.

3.2 Applications to proving the correspondence between virtual machines

In order to establish that the offensive VM is a sound abstraction of the defensive VM, one needs to prove that abstraction commutes with execution for every bytecode of the JavaCard instruction set. In the particular case of method invocation, one has to show:

```
nargs:nat
nm: class_method_idx
state: dstate
cap:jcprogram
=======================================
let res=(dinvokevirtual nargs  nm state cap) in
let ostate=(alpha_off state) in
let ores=(oinvokevirtual nargs  nm  ostate cap)
in  (INVOKEVIRTUAL_conds res) → (alpha_off res) = ores
```

where:

– the abstraction function `alpha_off` maps a defensive state into an offensive one;
– the predicate `INVOKEVIRTUAL_conds` ensures that the result state `res` is not a type error;
– the functions `dinvokevirtual` and `oinvokevirtual` respectively denote the defensive and offensive semantics of virtual method invokation.

The definition of dinvokevirtual is:

```
Definition dinvokevirtual :=
[nargs:nat][nm:class_method_idx][state:dstate][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.
```

In our initial work on CertiCartes, such statements were proved "by hand", by following the successive case analyses arising in the definition of the semantics of each bytecode. The proofs 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:

```
Simpl.
Unfold dinvokevirtual.
Analyze dinvokevirtual 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. In the case of a successful method call, the corresponding subgoal is:

```
...
_eg_8 : (get_method c nm)=(value m)
_eg_7 : (Nth_elt (classes cap) (get_obj_class_idx nhp))
          =(value c)
_eg_7 : (Nth_func hp (absolu vx))=(value nhp)
_eg_6 : (test_NullPointer vx)=false
_eg_5 : t=(Ref t0)
_eg_4 : x=((Ref t0),vx)
_eg_3 : (Nth_func (opstack h) (S n))=(value ((Ref t0),vx))
_eg_2 : nargs=(S n)
_eg_1 : state=(sh,(hp,(cons h lf)))
...
H : (INVOKEVIRTUAL_conds
 (new_frame_invokevirtual (S n) m nhp (sh,(hp,(cons h lf))) cap))
=======================================
(alpha_off
 (new_frame_invokevirtual (S n) m nhp (sh,(hp,(cons h lf))) cap))
=(oinvokevirtual (S n) nm (alpha_off (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 representation of JavaCard programs.

3.3 Applications to proving memory invariants

In order to establish that the abstract VM is a sound abstraction of the defensive VM, one needs to establish a number of invariants on the memory of the virtual machine, for example that executing the abstract virtual machine does not create illegal JavaCard types such as arrays of arrays. Proving such a property is tedious and involves several hundreds of case analyses, including a topmost case analysis on the instruction set. In the case of the `load` bytecode, one is left with the goal:

```
s : astate
t : type
l : locvars_idx
=================================================
(legal_types s) → (legal_types (tload t l s))
```

Whereas a proof "by hand" leaves the user with 29 subgoals to discharge, an application of the **Analyze** tactic leaves the user with 3 subgoals to discharge, namely the three interesting cases corresponding to the successful execution of the `tload` bytecode. In such examples, the use of our tactic reduces 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. [20] 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 [18] and HOL [13], 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 ϕ, and proceeds to build directly a proof of ϕ by following the shape of the definition of `f`. Eventually the tactic returns some proof obligations that are required to complete the proof of ϕ. 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.

Our work is also closely related to the induction process defined by Boyer and Moore. In Nqthm [8], the directive `INDUCT (f v1...vn)` can be used during a proof to perform the same case analysis and recursion steps as in the definition of `f`. An interesting feature that we do not handle in this paper is the ability of *merging* different functions into one single induction principle.

These works, 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 [19] 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 [5] 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, Terrasse [21, 22] has developed a tool to translate Typol specifications into 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 of 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 the combination of 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 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 the format of conditional rewriting rules of [2]. Indeed, this format, which can be automatically translated into Coq (provided the guard conditions are satisfied), lead to more readable specifications, especially for partial functions.
- *support for inversion principles.* We also would like to provide inversion principles like those that exist for inductive types [10]. Such principles do not use induction; instead they proceed by case analysis and use the fact that constructors of inductive types are injective. For example, inversion principles for functions should allow us to deduce:

  ```
  x=O ∨ x=(S S S S y)∧((isfourtime y)=true)
  ```

  from `(isfourtime x)=true`.
- *support for merging.* Merging [8] allows to generate a single induction principle for several functions. This technique allows to define very accurate induction principles, that are very useful in fully automated tactics.

From a more general perspective, it would be interesting to provide support for nested and well-founded recursion. Furthermore, a higher automation of equational reasoning is required in the proofs of commuting diagrams. One possibility is to rely on an external tool to generate rewriting traces that are then translated in Coq. There are several candidates for such tools, including Elan [6] and Spike [7], and ongoing work to interface them with proof assistants such as Coq, see e.g. [1].

5 Conclusion

We have described a toolset for reasoning about complex recursive functions in Coq. A fuller account of the package, including a tutorial, can be found in [12].

In addition, we have illustrated the usefulness of our package in reasoning about the JavaCard platform. This positive experience supports the claim laid in [2] 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.

Thanks We would like to thank the anonymous referees for there constructive remarks, Yves Bertot for its precious advice, ideas and examples, Guillaume Dufay for the feedback on using the tactic intensively, and Konrad Slind for the precisions on TFL/Isabelle.

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. 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.
3. 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 2294 of *Lecture Notes in Computer Science*, pages 32–45. Springer-Verlag, 2002.
4. 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.
5. 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.
6. P. Borovanský, H. Cirstea, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, C. Ringeissen, and M. Vittek. *The Elan V3.4. Manual*, 2000.
7. A. Bouhoula. Automated theorem proving by test set induction. *Journal of Symbolic Computation*, 23:47–77, 1997.

8. R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988.
9. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 7.2*, January 2002.
10. 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.
11. 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.
12. P. Courtieu. Function Schemes in Coq: documentation and tutorial. See `http://www-sop.inria.fr/lemme/Pierre.Courtieu/funscheme.html`.
13. M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher-order logic*. Cambridge University Press, 1993.
14. 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.
15. Z. Luo and R. Pollack. LEGO proof development system: User's manual. Technical Report ECS-LFCS-92-211, LFCS, University of Edinburgh, May 1992.
16. 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.
17. C. Paulin-Mohring. *Définitions Inductives en Theorie des Types d'Ordre Superieur*. Habilitation à diriger les recherches, Université Claude Bernard Lyon I, 1996.
18. L. Paulson. *Isabelle: a generic theorem prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.
19. 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.
20. K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, TU Münich, 1999.
21. 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.
22. D. Terrasse. *Vers un environnement d'aide au développement de preuves en Sémantique Naturelle*. PhD thesis, Ecole Nationale des Ponts et Chaussées, 1995.