Defining and Reasoning About Recursive Functions: A Practical Tool for the Coq Proof Assistant

Gilles Barthe[1], Julien Forest[1], David Pichardie[1], and Vlad Rusu[2]

[1] EVEREST Team, INRIA Sophia-Antipolis, France
{Gilles.Barthe, Julien.Forest, David.Pichardie}@sophia.inria.fr
[2] VERTECS Team, IRISA/INRIA Rennes, France
rusu@irisa.fr

Abstract. We present a practical tool for defining and proving properties of recursive functions in the Coq proof assistant. The tool generates from pseudo-code the *graph* of the intended function as an inductive relation. Then it proves that the relation actually represents a function, which is by construction the function that we are trying to define. Then, we generate *induction and inversion principles*, and a *fixpoint equation* for proving other properties of the function. Our tool builds upon state-of-the-art techniques for defining recursive functions, and can also be used to generate executable functions from inductive descriptions of their graph. We illustrate the benefits of our tool on two case studies.

1 Introduction

Dependent type theory provides a powerful language in which programs can be specified, proved and executed. Yet defining and reasoning about some simple recursive functions is surprisingly hard, and may rapidly become overwhelming for users, as too much effort is spent on dealing with difficult concepts not related to their verification problems. We detail these difficulties below, with an emphasis on the Coq proof assistant [13, 7], which is the focus of this article.

Difficulties with Defining Functions. In order to ensure decidability of type-checking (and hence of proof-checking), proof assistants based on type theory require that functions are provably terminating, total, and deterministic. Within Coq [13], totality and determinacy are enforced by requiring definitions by pattern matching to be exhaustive and unambiguous, and termination is enforced through a guard criterion that checks that recursive calls are performed on structurally smaller arguments. The guard predicate is designed to enable a translation from definitions by pattern matching to definitions by recursors, but it is difficult to use. As a result, users may be tempted to circumvent its use by adding a counter as an extra argument to recursive functions, and by performing a "large enough" number of iterations. This pragmatic solution is acceptable for specific programs, see e.g. [18], but cannot be adopted systematically.

M. Hagiya and P. Wadler (Eds.): FLOPS 2006, LNCS 3945, pp. 114–129, 2006.

A more general solution consists in defining functions by well-founded recursion. Two approaches prevail in the current state-of-the-art: the so-called *accessibility predicate* approach [21] (and its variant *converging iterations* [1]), and the so-called *ad-hoc predicate* approach [9] (which has been developed for a type theory different from Coq, and whose adaptation to Coq involves some subtleties); the two approaches make an advanced use of type theory, and are described in Section 2.

Both approaches help users in modeling functions. However, for lack of an appropriate and tool supported method, defining general recursive functions in Coq remains more difficult than defining those functions in proof assistants based on different logicals frameworks. For example, PVS [23], Isabelle [20] and HOL [15] allow users to provide a measure to prove termination, and generate proof obligations that must be discharged by the user in order for the system to accept the function as terminating. The possibility of relying on measures to guarantee termination makes the task of defining functions significantly easier. Yet the approach has never been implemented in proof assistants such as Coq.

Difficulties with reasoning about functions. Proof assistants based on type theory only offer limited support for reasoning about (structurally or generally recursive, total or partial) functions. There are three principles that can be used to reason about recursive functions: induction principles (which allow to prove properties about the function's output), inversion principles (which allow to deduce possible values of the function's input knowing the output), and fixpoint equations (which allow to unfold the function's definition in proofs). Unfortunately, users are not systematically provided with these results: for the Coq proof assistant, the tool of Balaa and Bertot [1] only generates the fixpoint equation for generally recursive functions, whereas the new tactic **`functional induction`**, which stems from the work of Barthe and Courtieu [2], only generates the induction principle for structurally recursive functions. The *ad-hoc predicate* approach of Bove and Capretta [9] provides an induction principle, but does not help in showing the totality of the function (and as stated above, cannot be adapted immediately to Coq).

Our objectives and contributions. The purpose of this article is to rectify this situation for the Coq proof assistant by proposing a tool that considerably simplifies the tasks of writing **and reasoning** about recursive functions. The tool takes as input the definition of a recursive function as pseudo-code (a language similar to type theory, but not subject to the restrictions imposed by the guard condition or by any criterion for totality), and generates Coq terms for its representation as a partial recursive function, its induction and inversion principles, and its fixpoint equation. Furthermore, the tool allows users to provide a well-founded order or a measure to prove termination, and generates from this information proof obligations that, if discharged, prove the termination of the function considered. Thus, if the pseudo-code describes a total function $f : A \to B$ that can be proved to be terminating, our tool will generate a Coq function $\bar{f}$ of the same type. The function $\bar{f}$ may not have exactly the same code as f, but this is of

no concern to the user, because the tool has provided her with all the principles needed to reason about $\bar{f}$ (so she does not need to use the definition). Moreover, Coq will extract from $\bar{f}$ a Caml function which is (almost) the same program as f.

At the heart of our tool lies a simple observation: inductive relations provide a convenient tool to describe mathematical functions by their graph (in particular they do not impose any restriction w.r.t. termination and totality), and to reason about them (using automatically generated induction and inversion principles derived from inductive definitions). Thus, relating a recursive function to its graph gives us for free reasoning principles about the function. Despite its simplicity, the observation does not seem to have been exploited previously to help defining and reasoning about recursive functions. Technically, the observation can be exploited both following the general accessibility approach and the *ad-hoc* predicate approach, leading to two mechanisms whose relative merits are discussed in Section 3. In Section 4, we illustrate the benefits of the tool.

2 A Critical Review of Existing Methods

Programming a nontrivial application inside a proof assistant is typically a long and tedious process. To make such developments easier for the user, mechanisms to define recursive functions should comply with the requirements below. They are by no means sufficient (see the conclusion for further requirements), but they definitely appear to be necessary:

- **deferred termination (DT):** the system should allow the user to define her functions without imposing that this very definition includes a termination proof;
- **support for termination proofs (ST):** termination proofs, which are usually not the main concern of the user, should be automated as much as possible;
- **support for reasoning (SR):** the system should provide the user with reasoning principles for proving all other properties of her functions;
- **executability (X):** when applied to structurally recursive definitions, mechanisms should yield the same result as, e.g., the **`Fixpoint`** construction of Coq.

Unfortunately, the current version of the Coq proof assistant does not satisfy these requirements, neither for structurally recursive nor for general recursive functions.

In the sequel, we briefly review the state-of-the-art approaches for general recursive definitions, and other related works, assessing in each case its benefits and limitations. A summary is presented in Figure 1.

For the sake of concreteness, we center our discussion around the fast exponentiation function, which is informally defined by the clauses $2^n = 2 \cdot 2^{n-1}$ if n is odd and $(2^{n/2})^2$ if n is even. The natural definition of the function in Coq would be:

Approach	Deferred termination	Induction principle	Inversion principle	Fixpoint equation	Support for termination	Implemented in Coq
Balaa & Bertot	no	no	no	yes	no	prototype
Barthe & Courtieu	n.a.	yes*	no	no	n.a.	yes
Bove & Capretta	yes	yes	no	no	no	no
this paper	yes	yes	yes	yes	yes	prototype

* for structurally recursive functions

Fig. 1. Comparaison of state-of-the-art approaches

```
Fixpoint pow2 (n: nat) :  nat  :=
  match n with
   |0 ⇒ 1
   |S q ⇒ match (even_odd_dec (S q)) with
             |left _ ⇒ square (pow2 (div2 (S q)))
             |right _ ⇒ n * (pow2 q)
           end
  end.
```

where `even_odd_dec` is a function, returning either `left` and a proof of the fact that its argument is even, or `right` and a proof of the fact that its argument is odd. Proofs are irrelevant here and are replaced by the _ placeholder. However this definition is not structurally recursive, as the guard predicate does not see `div2 (S q)` as structurally smaller than `q`. As a consequence, the current Coq tool rejects this definition and we must use one of the approaches described later in this paper for defining the function `pow2: nat →nat`.

Then, assuming the function `pow2: nat →nat` is defined in Coq, we need the following tools in order to reason about it:

- *a fixpoint equation* which allows to unfold the definition of the `pow2` in a proof:

```
Lemma pow2_fixpoint : ∀ n, pow2 n  =
match n with
|0 ⇒ 1
|S q ⇒
   match (even_odd_dec (S q)) with
        |left _ ⇒  square (pow2 (div2 (S q)))
        |right _ ⇒ 2 * (pow2 q)
   end
end.
```

- *a general induction principle* which only focuses on the domain of the function:

```
Lemma pow2_ind_gen : ∀ P : nat → Prop,
  P 0 →
  (∀ q, even_odd_dec (S q) = left _ → P (div2 (S q)) → P (S q) ) →
  (∀ q, even_odd_dec (S q) = right _ → P q → P (S q)) →
∀ n, P n.
```

  This principle was initially proposed by Slind [24] and is currently implemented in Coq by the command **`functional induction`** (only for structural recursive functions). It often needs to be combined with the fixpoint equation to simulate the inputs/outputs induction principle below.

- *an inputs/outputs induction principle* which allows to deduce a relation between inputs and outputs from the proof that the relation holds for all the

pairs consisting of input and output to the recursive calls, for all situations in which these calls are performed:

```
Lemma pow2_ind : ∀ P : nat → nat → Prop,
P 0 1 →
(∀ q, even_odd_dec (S q) = left _ →
          P (div2 (S q)) (pow2 (div2 (S q))) →
          P (S q) (square (pow2 (div2 (S q))))) →
(∀ q, even_odd_dec (S q) = right _ → P q (pow2 q) →
            P (S q) (2*pow2 q)) →
∀ n, P n (pow2 n).
```

This principle is less general than the previous induction principle. Nevertheless, it produces more instantiated proof obligations. As a consequence, it often facilitates proof automation.

– *an inversion principle* to deduce some information about the inputs of a function from the value of the result of the function on these inputs. Such inversion lemmas are of great help in reasoning about executable semantics (most clauses in such semantics yield to error states, see e.g. [3], and the inversion principle allows users to deal directly with states which do not go wrong). Due to space constraints, we do not discuss inversion principles further.

The inputs/outputs induction principles allow the user to reason about functions: consider a proof of a simple theorem stating that the fast exponentation is equivalent to the slow (structural) one, shown below together with a lemma:

```
Fixpoint pow2_slow (n: nat) :  nat  :=
  match  n  with
    |0 ⇒  1
    |S  q ⇒ 2 * (pow2_slow q)
  end

Lemma pow2_slow_div2: ∀ n, even n  →  square (pow2_slow (div2 n)) = pow2_slow n.
```

The theorem can be proved simply by applying the induction principle `pow2_ind` to the predicate **`fun`** `n res ⇒res = pow2_slow n`. This leaves us with one unproved subgoal, which is solved by applying the `pow2_slow_div2` lemma:

```
Theorem pow2_prop: ∀ n, pow2 n = pow2_slow n.
Proof.
  apply pow2_ind with (P:=fun n res ⇒ res = pow2_slow n); intros;
     subst; try reflexivity.
  apply pow2_slow_div2; auto.
Qed.
```

On the other hand, simple induction principles such as that on natural numbers are too weak for proving properties about non-structural recursive functions such as `pow2_prop`. We may of course use a *well-founded* induction principle, but using it is as hard as building the function in the first place: indeed, the well-founded induction principle for *reasoning* about a function is almost identical to that for building the function (`well_founded_induction` discussed in Section 2.1 below).

2.1 General Accessibility

This approach is based on so-called *accessibility predicates* and consists in defining the function by induction on some well-founded relation for which recursive

calls are decreasing (i.e. made on strictly smaller arguments). This approach originates from [21], and is embodied in Coq by the `well_founded_induction` principle from its library: using this principle, one can define a general recursive function `f: A →B` from a relation `R:A →A →` **`Prop`**, a proof `P` that `R` is well-founded, and an "induction step" `I` of type `∀ y:A, (∀z: A, R z y →B) →B`. Formally, `f` is defined as `(well_founded_induction A R P I)`.

It is possible, but cumbersome, to circumvent the lack of reasoning principles. For example, assume that we want to prove that some post-condition `(Post x (f x))` holds, whenever the argument `x` satisfies a pre-condition `(Pre x)`. Then, it is enough if we *define* `f` right from the start to have the dependent type `∀ x: A, (Pre x) →{y : B | (Post x y)}`. In this way, the pre/post condition information about `f` is immediately available. However, this approach assumes that one *knows* already when defining `f` *all* the properties that one will ever need about it! This is not realistic, because any nontrivial development typically requires the user to state and prove many intermediate lemmas, *not known in advance*, between the definition of a function and the final theorem about the function. The above approach would then require to *re-define* `f` with *each* new lemma, and this definition requires to *prove* the new lemma *while* the function is defined, in addition to re-proving all lemmas previously proved.

Moreover, the approach does not easily scale up to mutually recursive functions such as those considered in Section 4.2. Hence, this approach is clearly not useable in any nontrivial development.

2.2 Ad-Hoc Predicate

The second approach has been described by Bove and Capretta [9] in the context of Martin-Löf's type theory.

The approach is based so-called on *ad-hoc predicates*, and consists in providing for each function an inductive predicate, called its *domain*, that characterizes its domain *and* the way recursive calls are performed. The domain is defined in such a way that if an element belongs to it, then all the arguments of the recursive calls needed to compute the result of the function on this argument also belong to it. For example, the domain `Dpow2` of fast exponentiation is defined as follows:

```
Inductive Dpow2 : nat → Set :=
  DPow2_0 : Dpow2 0
| DPow2_S_even : ∀ (q : nat) (H : even (S q)),
                  even_odd_dec (S q) = left _ H →
                  Dpow2 (div2 (S q)) → Dpow2 (S q)
| DPow2_S_odd : ∀ (q : nat) (H : odd (S q)),
                 even_odd_dec (S q) = right _ H →
                 Dpow2 q → Dpow2 (S q)
```

Using the closure property of `Dpow2`, one can define an auxiliary (dependently typed) function `Ppow2: ∀a:nat, Dpow2 a →nat` that takes an additional argument a proof of membership in the domain; the function is defined by induction on its last argument. Finally, one can obtain a function in one argument by showing that `Dpow2 a` is inhabited for each natural number `a`.

The approach is elegant, allows users to defer the proof of totality of the function, and provides induction principles for free: thus it satisfies the criteria

(DT) and (SR). However, it does not satisfy the criteria (ST) and (X). More importantly, in the above we have departed from standard practice by defining Dpow2 as a dependent set, rather than as a proposition (observe that the type of Dpow2 is nat → **Set**). As a result, the extracted function is not that expected because it looks like:

```
let rec ppow2 x = function
  | DPow2_0 → S O
  | DPow2_S_odd (q, x0) →
      mult (ppow2 (div2 (S q)) x0) (ppow2 (div2 (S q)) x0)
  | DPow2_S_even (q, x0) → mult (S (S O)) (ppow2 q x0)
```

This phenomenon is due to the fact that, in Coq's type theory, the sorts **Set** and **Prop** are not equivalent: the sort **Set** is the sort of datatypes (which are relevant from a computational point of view and are therefore translated to Caml code by Coq's extraction mechanism), whereas **Prop** is the sort of logical properties (which are not translated to Caml).

Bertot and Castéran [7] point out a possible approach for adapting the *ad-hoc predicate* approach to the Calculus of Inductive Constructions—which amounts to defining Dpow2 with type nat →**Prop**, and still being able to define pow2 afterwards. Once more, the difficulty arises from the guard predicate which requires users to prove in a very specific manner so-called inversion theorems:

```
Lemma Dpow2_even_inv : ∀ n (p:Dpow2 n) (q : nat) (H_eq:n = S q )
    (H : even (S q)),even_odd_dec (S q) = left _ H → Dpow2 (div2 (S q)).

Lemma Dpow2_odd_inv :  ∀ n (p:Dpow2 n) (q : nat) (H_eq:n = S q )
   (H : odd (S q)), even_odd_dec (S q) = right _ H → Dpow2 q.
```

These lemmas show how to deduce from a proof of Dpow2 (S q), either a proof of Dpow2 (div2 (S q)) if (S q) is even or a proof of Dpow2 q if (S q) is odd. Whether or not we can define the recursive function pow2 depends on the way one proves these lemmas—a blatant violation of the principle of proof irrelevance—and the guard condition makes it impossible for users to apply existing tools to reason about relations or for dealing with equalities. If we prove them in the correct way, we can define Ppow2 by structural induction on an argument of type Dpow2 n. The corresponding extracted Caml program is the one we expected:

```
let rec ppow2 = function
  | O → S O
  | S q →
      (match even_odd_dec (S q) with
         | Left → let rec_res = ppow2 (div2 (S q)) in mult rec_res rec_res
         | Right → mult (S (S O)) (ppow2 q))
```

but using the *ad-hoc predicate* approach in Coq definitively requires a lot of expertise and pushes Coq (and users) to their limits.

2.3 Functional Induction

functional induction is a new Coq tactic that originates from work by Barthe and Courtieu [2]; it generates automatically induction principles which have proved very useful in reasoning about executable semantics [3], but its scope of application is limited to structurally recursive functions. In addition, the tactic

builds the induction principle from the internal representation of the function instead of building it from the user definition of the function, which leads to unnecessarily verbose induction principles which in some circumstances are hard to use. For example, it was not possible to use this tactic in the work [18].

2.4 Other Approaches in Coq

The "converging iterations" approach In [1], a recursive function f is defined as the solution of a fixpoint equation involving a functional F. The definition involves proving that the fixpoint equation terminates in a finite number of iterations. The fixpoint equation of f is then obtained almost for free. Induction principles are not provided. The approach has been implemented in a prototype tool that automatically generates some of the definitions, proof obligations, and corresponding proofs.

The tool of Balaa and Bertot [1], which is currently the sole tool for Coq based on this approach, does not satisfy the criteria (DT) and (ST) and (X), and only partially satisfies the criterion (SR), as it only provides the fixpoint equation for the function.

Combining [9] and [1]. In [6], the "converging iterations" and "ad-hoc predicate" approaches are merged into a powerful technique for recursive function definition, allowing for partial functions with nested recursive calls. The method is implementable in Coq. It satisfies criteria (DT) and (SR) but it also adds up all the difficulties of the two combined approaches, which makes it quite hard to grasp and to use for practical verification purposes.

Using coinductive types. Capretta [12] uses co-inductive types to encode the tick monad, which associates to every type its type of partial elements, and shows how the monad can-be used to formalize all recursive functions between two given types.

2.5 Yet Another Approach: Extending Coq

Another approach is to increase the class of recursive functions that are accepted by Coq's termination criterion by introducing increasingly powerful syntactic schemes [8], or more semantical schemes based on size information [4] (the latter also bears some resemblance with work of Xi [25] on enforcing termination using restricted dependent types). In fact, the implementation of the *ad-hoc* predicate approach could be significantly simplified by abandoning the guard predicate in favor of type-based termination methods, especially for inversion lemmas. However, it is unavoidable to fall back on general recursion in the end.

The developers of Epigram are exploring a more radical approach and elaborate a powerful theory of pattern-matching that lends well to programming with dependent types [19]. They use views, which play a similar role to inductive domain predicates and enlarge the set of acceptable fixpoint definitions.

3 Overview of the Tool

Our tool[1] is based on the notion of graph associated with a function. In link with the previous section, the function pow2 will be used as a running example for describing our tool. The tool takes as input functions defined in a pseudo-code style (without guard conditions on recursive calls). The new command by the user is:

```
GenFixpoint pow2 (n: nat) {wf nat_measure}: nat := 
  match n with
   |0 ⇒ 1
   |S q ⇒ match (even_odd_dec (S q)) with
             |left _ ⇒ square (pow2 (div2 (S  q)))
             |right _ ⇒ n * (pow2 q)
           end
  end.
```

3.1 Proof Obligations Delegated to the User

By default, the definition of the function must be provided with a relation (named nat_measure in the current example) between function arguments to justify the termination (with the **wf** keyword). Our tool then generates two kinds of proof obligations to be interactively proved: the user must first prove that all recursive calls respect the given relation (*compatibility property*) and the relation must be proved well founded (*well-founded property*). For the pow2 function, three subgoals are generated. Subgoals 1 and 2 deal with compatibility properties, and subgoal 3 deals with the well-founded property.

```
3 subgoals

  ============================
   ∀ (n q : nat) (h : even (S q)),
   n = S q  →  even_odd_dec (S q) = left h  →  nat_measure (div2 (S q)) n

subgoal 2 is:
 ∀ (n q : nat) (h : odd (S q)),
 n = S q  →  even_odd_dec (S q) = right h  →  nat_measure q n
subgoal 3 is:
 well_founded nat_measure
```

Our tool proposes other termination criteria, each of them with different kinds of proof obligations :

- {**measure** f}: the termination relies on a measure function f. The generated proof obligations require to prove that measures of recursive calls are smaller than measure of initial arguments. These kind of proof obligations are often easily discharged with arithmetic decision procedures.
- {**struct** arg}: the function is structurally recursive on argument arg (standard Coq criterion). No proof obligation is required.
- {**adhoc**}: the user doesn't want to prove termination. The generated function will be partial: pow2 : ∀n:nat, Dpow2 n →nat, where Dpow2 is the ad-hoc predicate associated with pow2.

[1] The current implementation of the tool is available at http://www-sop.inria.fr/everest/personnel/David.Pichardie/genfixpoint with several examples of generated coq files.

3.2 Automated Generation of the Function

The first task of the tool is to generate the graph relation of the function, as an inductive relation that completely "mimics" its definition. We use one constructor for each branch of the function pattern matching.

```
Inductive pow2_rel : nat → nat → Prop :=
|pow2_0 : pow2_rel 0 1
|pow2_1 : ∀ q res, (even_odd_dec (S q)) = left _ →
     pow2_rel (div2 (S q)) res → (pow2_rel (S q) (square res))
|pow2_2 : ∀ q res, (even_odd_dec (S q)) = right _ →
     pow2_rel q res → pow2_rel (S q) (2*res).
```

The next step is then to implement this relation : program a Coq function `pow2:nat→nat` which satisfies the relation `pow2_rel: ∀ n, pow2_rel n (pow2 n)`. We have experimented (and implemented) two techniques to achieve this task.

– A first approach is to use a well founded induction principle for defining an auxiliary function `pow2_rich` with a rich dependent type. This type specifies that the function satisfies its graph relation:

```
Definition pow2_type (n:nat) :=  {pow2_n: nat | pow2_rel n pow2_n}.
```

We automatically generate a proof script for this definition[2]. During the proof, applications of the induction hypothesis are justified using compatibility properties previously discharged by the user. It is then straightforward to define `pow2`

```
Definition pow2 (n:nat) : nat := let (f,_) := pow2_rich n in f.
```

– The second technique is based on ad-hoc predicates. The predicate is extracted from the graph relation by projection. We hence exactly generate the predicate `Dpow2` described in Section 2.2. Two approaches may then be used, depending on whether this predicate is in **`Set`** or **`Prop`**.

- `Dpow2:nat→`**`Set`**: this is the simplest approach. A function `pow2_adhoc` of type `∀ n, Dpow2 n →nat` is directly programmed by pattern matching on its argument of type `Dpow2 n`. This technique has a more general application: it allows to generate executable code from inductive relations (see Section 3.4).
- `Dpow2:nat→`**`Prop`**: in this approach, we must first generate inversion lemmas to justify recursive calls, as explained in Section 2.2. These lemmas rely on a more general property which establishes that distinct branches of the graph are pairwise incompatible. All these lemmas (statements and proofs) are automatically generated by inspection of the graph relation. It is a particularly difficult part for the tool since statements must be proved in a very specific manner (they are proof-relevant, see Section 2.2).

In both cases the totality of the function must still be proved. We hence generate an implementation of the ad-hoc domain by well-founded induction over the relation `nat_measure`.

[2] We use here the possibility of programming by proof.

```
Lemma compute_Dpow2 : ∀ n, Dpow2 n.
```

The definition of pow2 is then straightforward by combining compute_Dpow2 and pow2_ahdoc.

```
Definition pow2 (n:nat) : nat := pow2_ahdoc n (compute_Dpow2 n).
```

As a final result, we obtain a Coq function verifying the graph relation whose extracted code is exactly as expected (except for the technique using an ad-hoc predicate in **Set**).

In the next section, we only require to have an implementation pow2 of the graph relation pow2_rel: the actual technique used for defining pow2 is not relevant.

3.3 Automatic Generation of Reasoning Principles

We now turn to generate the reasoning principles presented at the beginning of Section 2. All steps are carried automatically. The first step is to characterize pow2_rel as the graph of pow2.

```
∀ x y, pow2_rel x y →pow2 x = y
```

This is done by proving that the graph has the following "functional" property:

```
∀ x y1 y2, pow2_rel x y1 →pow2_rel x y2 →y1=y2
```

Using the now proved equivalence between pow2 and pow2_rel we generate:

- the inputs/outputs induction principle: starting from the induction principle of the relation pow2_rel (automatically generated by Coq, as for all inductive definitions) we remove all occurrences of pow2_rel using pow2.
- the general induction principle: the construction is similar to the previous principle, but using the induction principle of ad-hoc domain instead of the graph relation. This proof only relies on the fact that pow2 satisfies its associated graph.
- fixpoint equation: once again proving the fixpoint equation by reasoning on pow2_rel is easy. We only have to follow the definition of the pseudo code.
- inversion principle: an hypothesis of the form `e' = pow2 e` is first transformed into `pow2_rel e e'`, then we use the Coq tactic inversion to discriminate some incompatible cases, and we replace all generated hypotheses dealing with pow2_rel with their equivalent form dealing with pow2.

Definitions by pattern matching are internally represented in Coq by case analysis. As a result, users wanting to prove a property about a function they have defined with a few cases may be faced with an artificially large number of cases [18]. Our tool avoids the explosion of cases in reasoning about functions by providing appropriate induction principles, with exactly the same number of cases as the definition of the function. An extension of our tool is currently planned to correctly handle the default case which could occurs in a pattern matching.

3.4 Applications to relations

The tools we developed for the *ad-hoc* approach can serve a much more general purpose than defining and reasoning about recursive functions presented using pseudo-code. In fact, our work can be used to generate executable code from inductive relations. Such a possibility is interesting, because many existing developments, especially those concerned with programming language semantics, favor an inductive style of formalization. As mentioned above, inductive formalizations are attractive, because they elude issues of partiality, termination, and determinacy, and also because they make reasoning principles immediately available to users. However, defining a semantics inductively does not yield an executable semantics. Several works have attempted to overcome this weakness of inductive relations by providing mechanisms that transform inductively defined relations in a proof assistant based on type theory into recursive programs in a typed functional language, see e.g. [5]. While such a translation allows users to specify and verify inductively defined functions, and to extract an executable program for computing the result of the functions, it forces computations to take place *outside* the proof assistant. In contrast, our tool provides a means to generate *within* the proof assistant executable functions that realize inductive relations. By doing so, our tool makes it possible to prove the correctness of the executable semantics w.r.t the inductive one. This brings higher guarantees in the context of safety or security critical applications (which will typically use extracted code), because only the extraction mechanism of Coq needs to be trusted. We are currently experimenting with the tool for generating executable semantics of virtual machines from inductive semantics.

Note that our tools do not require determinacy in order to transform inductive relations into functions. Indeed, for non-deterministic relations the *ad-hoc* domain will not only encode the stack of recursive calls, but also the evaluation strategy used to compute a result. Thus our tool opens interesting perspectives in the context of encoding arbitrary term rewriting systems as functions (with an additional argument, i.e. the strategy) *within* the proof assistant. Potential applications include the design of reflective tactics, which encode decision procedures in Coq, and allow to achieve smaller and faster proofs, see e.g. [16] for a reflective tactic based on rewriting. The encoding is also of interest for confluent term rewriting systems, because it allows to defer the proof of confluence, or equivalently of functionality of its graph; this allows for example to rely on termination to prove confluence.

4 Case Studies

4.1 Interval Sets

We have built a library for finite sets of natural numbers encoded using ordered lists of intervals, it was planned to be part of the development of a framework for certified static analyses [10]. It includes functions for union, intersection, inclusion, and membership tests.

Among all those operations, the union operation is the most involved. We describe in some detail its definition and some of the proofs that we have written around it using our method.

First, interval sets are defined using the following two inductive relations (`::` is the Coq notation for *cons* on lists, and `fst`, `snd` return the first, resp. the second element of a pair):

```
Inductive  is_interval (i:Z*Z) : Prop :=
  |is_interval_cons :  fst i <= snd i  →  is_interval i.

Inductive is_interval_set : list (Z*Z)  →  Prop :=
    |Nil_set : is_interval_set nil
    |Single_set : ∀ i, is_interval i  →  is_interval_set (i::nil)
    |Cons_set : ∀ i j l,
      is_interval i  →  is_interval j  →   (1+ snd i) < fst j  →
      is_interval_set (j::l)  →  is_interval_set (i::j::l).
```

In particular, note that successive intervals in an interval set may not intersect each other, touch each other, or even be *adjacent*: $1 + snd\ i < fst\ j$, otherwise, they would form a single interval.

An inductive definition `compare_intervals_dec a1 a2`, not shown here, compares the relative positions of two intervals `a1`, `a2`. An implementation of the `union` operation linear in the size of its arguments cannot be structural, as it cannot predict in advance how the lists begin (more specifically, how many initial intervals of one list are included in the second one):

```
Fixpoint union (li1 li2: list (Z*Z)) : list (Z*Z) :=
match li1, li2 with
|nil, l' ⇒ l'
|l'', nil ⇒ l''
|a1::l1, a2::l2 ⇒
  match compare_intervals_dec a1 a2 with
    |snd_far_after  _   ⇒ a1::(union l1 (a2::l2))
    |snd_close_after _ _ _⇒ union l1 ((fst a1,snd a2)::l2) (*not structural*)
    |snd_includes _ _ _  ⇒ union l1 (a2::l2)
    |snd_equal_fst  _ _ ⇒ a1::(union l1 l2)
    |fst_includes _ _ _  ⇒  union (a1::l1) l2
    |fst_close_after _ _ _⇒ union ((fst a2,snd a1)::l1) l2 (*not structural*)
    |fst_far_after _ ⇒ a2::(union (a1::l1) l2)
       end
end.
```

One property that we need to prove about the `union` function is that, although its arguments are lists of pairs of integers, by applying it to two `interval_sets` one obtains an `interval_set`.

This turned out to be a nontrivial exercise. We have solved it as follows: we have defined a *weak interval set* structure, which is a list of intervals in which consecutive intervals may overlap, and a *weak union* operation, which is closed on weak interval sets (but not on proper interval sets). Then, a *grouping* operation transforms a weak interval set into a proper interval set. Weak union and grouping are also non-strucuturally recursive, but somewhat simpler that the proper union. We then show that the proper union is equal to the composition of the weak union and the grouping operations, and that the result of the composition is an interval set.

For this, we make a heavy use of the induction principles for proper union, union, and grouping functions. Several dozens of lemmas were proved using those principles; without them, we could not have completed the development.

4.2 Systems of Affine Recurrence Equations

Consider the following system:

```
W[n,i] =
    if  n<=N+D-1 then 0
    if  N+D<=n then W[n-1,i] +  (E[n-D] * x[n-i-D])
Y[n,i] =
    if i=-1 then 0
    if 0<=i then Y[n,i-1] + (W[n,i] * x[n-i])
E[n] = (d[n] - res[n]);
res[n] = Y[n,N-1];
```

which implements an auto adaptive filter [17] used in signal processing for noise cancellation or system identification. It takes an input signal `x` and a reference input `d` and updates its convolution coefficients so that the output signal `res` will converge towards the reference input.

The program can be rewritten in pseudo-code, yielding mutually recursive partial recursive functions. We now want to prove the following:

```
(∀ n, d n = 0)  →  (∀ n, res n = 0)
```

A basic analysis of the dependencies in the system of equations reveals a circularity in the proof: in order to prove the property, we must prove that `Y` is uniformly null, which requires to prove that `X` is uniformly, which requires to prove that `res` is uniformly null. Without appropriate induction principles on the structure of the function, the proof is intractable. In [11], Cachera and Pichardie report on a tool that generates induction principles for this specific class of systems. Our tool subsumes their work, as we dispose of a general tool which produces the same principles for the class of systems they consider. In particular, applying the induction principle generated by our tool allows us to conclude the proof with a few applications of the `ring` tactic of Coq.

5 Conclusion

We have presented a tool to define and reason about recursive functions in Coq, and shown on non-trivial examples that the tool significantly improves over earlier work by simplifying the definition of functions, and generating automatically all reasoning principles required to prove properties about the function. Our tool is very similar in spirit to the TFL tool developed by Slind [24] for Isabelle and HOL, but differently from TFL, our tool is based on the observation that the graph of a function is the most powerful property satisfied by the function, and exploits state-of-the-art techniques for defining recursive definitions in type theory.

Further work is required to enhance the applicability of the tool. In order to increase usability, we plan to allow in pseudo-code non-exhaustive and ordered

pattern matching (a specific instance of ordered pattern matching being default cases). Such an extension would solve common problems encountered by Coq users:

- *partial functions:* the prevailing approach with partial functions is to use the lift monad, i.e. to encode partial functions from A to B as total functions from A to $B_\perp$. Yet, experience in the formalization of programming language semantics has demonstrated that the use of the lift monad clutters definitions, and hence later proofs [3]. The extended tool shall simplify the definition of partial functions by omitting cases of undefinedness in pseudo-code, and reasoning about partial functions by combining the induction and inversion principles into a principle to reason about the defined cases of the function.
- *increasing support for termination proofs:* proving termination is made easier by our tool, but still requires some work from the users. In the future, we intend to extend pseudo-code to accept not only well-founded orders and measures, but also reduction orders. The extended tool shall enable users to benefit immediately from ongoing efforts to develop libraries for certifying termination proofs in Coq [22], and to reduce even further the effort dedicated to termination proofs.

It could also be interesting to extend our tool to nested recursive functions. One well-known difficulty is that applying the *ad hoc* predicate approach to nested functions requires to use inductive-recursive definitions [14], which are not part of the Calculus of Inductive Constructions. In an unpublished note, Capretta has shown how induction-recursion can be given an impredicative encoding in the Calculus of Inductive Constructions; unfortunately, the use of impredicativity severely restricts the possibilities of exploiting the encoding, and solutions that cater for predicative universes must be sought.

References

1. 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.
2. G. Barthe and P. Courtieu. Efficient Reasoning about Executable Specifications in Coq. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 31–46. Springer-Verlag, 2002.
3. 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.
4. G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. *Mathematical Structures in Computer Science*, 14:97–141, February 2004.

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 2277 of *Lecture Notes in Computer Science*, pages 24–40. Springer-Verlag, 2002.
6. Y. Bertot, V. Capretta, and K. Das Barman. Type-theoretic functional semantics. In V. Carreño, C. Muñoz, and S. Tahar, editors, *Proceedings of TPHOLs'02*, volume 2410 of *Lecture Notes in Computer Science*, pages 83–98. Springer-Verlag, 2002.
7. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development— Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.
8. F. Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, February 2005.
9. A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, 15:671–708, February 2005.
10. D. Cachera, T. Jensen, D. Pichardie, and V. Rusu. Extracting a data flow analyser in constructive logic. *Theoretical Computer Science*, 342, 2005. To appear.
11. D. Cachera and D. Pichardie. Embedding of Systems of Affine Recurrence Equations in Coq. In *Proceedings ofTPHOLs'03*, number 2758 in Lecture Notes in Computer Science, pages 155–170. Springer-Verlag, 2003.
12. Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 1(2):1–18, 2005.
13. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
14. P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 2000.
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. B. Grégoire and A. Mahboubi. Proving equalities in a commutative ring done right in coq. In J. Hurd and T. Melham, editors, *Proceedings of TPHOLs'05*, volume 3603 of *Lecture Notes in Computer Science*, pages 98–113. Springer-Verlag, 2005.
17. M. Katsushige, N. Kiyoshi, and K. Hitoshi. Pipelined LMS Adaptative Filter Using a New Look-Ahead Transformation. *IEEE Transactions on Circuits and Systems*, 46:51–55, January 1999.
18. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *Proceedings of POPL'06*. ACM Press, 2006.
19. C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14:69–111, 2004.
20. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
21. B. Nordström. Terminating general recursion. *BIT*, 28(3):605–619, 1988.
22. Color Project. `http://color.inria.fr`
23. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
24. K. Slind. *Reasoning about Terminating Functional Programs*. PhD thesis, TU Münich, 1999.
25. H. Xi. Dependent types for program termination verification. *Higher-Order and Symbolic Computation*, 15(1):91–131, March 2002.