Extraction in Coq: an Overview

Pierre Letouzey

Laboratoire PPS, Université Paris Diderot - Paris 7
Case 7014, F-75205 Paris Cedex 13, France
letouzey@pps.jussieu.fr

Abstract. The extraction mechanism of Coq allows one to transform Coq proofs and functions into functional programs. We illustrate the behavior of this tool by reviewing several variants of Coq definitions for Euclidean division, as well as some more advanced examples. We then continue with a more general description of this tool: key features, main examples, strengths, limitations and perspectives.

1 Introduction

This article describes the current status of the extraction mechanism available in the Coq proof assistant [7, 8]. The extraction mechanism of Coq is a tool for automatic generation of programs out of Coq proofs and functions. These extracted programs are expressed in functional languages such as Ocaml, Haskell or Scheme, these three languages being the ones currently supported by Coq extraction. The main motivation for this extraction mechanism is to produce certified programs: each property proved in Coq will still be valid after extraction.

Through a series of examples about Euclidean division, we will review several alternatives that allow the user to express in Coq an algorithm that does not fit naturally in this system. We will also see how these alternatives influence the shape of the program obtained by extraction. We will then mention two advanced situations that illustrate the fact that Coq's current extraction can handle any Coq objects, even the ones defined via high-end features of Coq and without direct counterpart in Ocaml or Haskell. We will summarize the key features of Coq extraction, mention some significant Coq developments taking advantage of the extraction, and conclude on the current strengths of this tool, its limitations and future research perspectives.

2 Extraction in practice : div

In this section, we illustrate the use of Coq extraction on a small yet revealing example: Euclidean division amongst natural numbers. For sake of simplicity, we will use the unary `nat` datatype for representing these natural numbers: every number is stored as either zero or the successor `S` of another number. Even if this representation is inherently inefficient, the discussion that follows would be quite

similar with more clever coding of numbers. Coq's Standard Library provides basic operations and relations on `nat`, such as `+`, `*`, `<`, $\leq$. In Coq, logical relations do not necessarily have corresponding boolean test functions, but here a result named `le_lt_dec`, noted `≤?` afterwards, can be used as an effective comparison function for determining whether n≤m or m<n for any numbers `n` and `m`.

Each Coq snippet proposed below is taken verbatim from a valid session[1] with Coq 8.2, including unicode notations and other syntactic improvements.

2.1 A division that fulfills the structural constraint

One usual algorithm for division on natural numbers is to proceed by successive subtractions: `div x y = 0` when `x<y` and `div x y = S (div (x-y) y)` otherwise. But this cannot be written directly in Coq. Due to the intimate relationship between proofs and programs in Coq, no Coq objects may be allowed to trigger infinite computations. A rather drastic constraint is hence required on recursive functions in order to ensure their termination: they should have at least one inductive parameter such that recursive calls are done on an immediate subterm of this parameter[2]. Here, our recursive call fails this criterion, since `(x-y)` is not an immediate subterm of `x`, and second parameter `y` has not changed. Even worse, trying this algorithm with `y=0` leads to an infinite computation: Coq's rejection is here quite legitimate.

For defining nonetheless our division in Coq, a first solution is to try to live with this structural constraint, and adapt our algorithm accordingly. For instance:

```
Fixpoint div x y := match x with
  | 0 => 0
  | S x' =>
     let z := div x' y in
     if (S z)*y ≤? x then S z else z
end.
```

Knowing the quotient for the predecessor of `x` can indeed be used to infer the quotient for `x`. But proceeding this way leads to a costly test repeated `x` times. This is a common situation with Coq: intended algorithms can be adapted to be "structural", but this may result in an awkward and/or less efficient algorithm.

Command `Extraction div` can then be used to convert this division to Ocaml code[3]:

```
let rec div x y =
  match x with
    | O -> O
    | S x' ->
        let z = div x' y in if le_lt_dec (mult (S z) y) x then S z else z
```

[1] See `http://www.pps.jussieu.fr/~letouzey/download/examples_CiE2008.v`

[2] For a more precise definition of this structural constraint, see Chap. 4 of Coq Reference Manual at `http://coq.inria.fr`.

[3] The complete list of extraction commands can be found in the Coq Reference Manual.

This first extracted `div` highlights the fact that on basic Coq functions, extraction is mainly performing a straightforward syntactic translation. But even on such a simple function, some proof elimination occurs during extraction. In fact, comparison `le_lt_dec a b` is not producing a mere boolean, but rather a proof-carrying boolean type `{a≤b}+{b<a}`, which is an inductive type internally named `sumbool`, with two constructors `left` and `right` both having a proof as parameter, here respectively a proof of `a≤b` and a proof of `b<a`. Extraction removes these proofs, hence obtaining an extracted `sumbool` datatype with two constant constructors, isomorphic to `bool`. In order to get precisely the extracted code shown above, one could then teach Coq to take advantage of this isomorphism, via : `Extract Inductive sumbool => bool [ true false ]`.

One should note that the proof elimination done during extraction is based on earlier declarations by the user (or by the library designer). Here, proof-carrying boolean `{a≤b}+{b<a}` is exactly isomorphic to logical disjunction `a≤b ∨ b<a` (instead of `left` and `right`, constructors are named `or_introl` and `or_intror`). Simply, the former is declared in the logical world named `Prop` and is pruned during extraction whereas the latter is declared in `Set`, the world of Coq programs, and simply loses at extraction the logical parameters of its constructors. Similarly, two existential types coexist in Coq: the logical one `∃x:A,P x` and the informative one `{ x:A | P x }`.

2.2 A division with an explicit counter

Let's now try to implement a function closer to our intended division algorithm, instead of the ad-hoc structural version of the last section. A solution is to artificially add a new structurally decreasing parameter that will control the number of allowed recursive calls. Here for instance, if `y≠0`, it is clear that at most `x` successive subtractions can occur before the algorithm stops. A common presentation is to separate the function to iterate `div_F` from the actual counter-based recursive iterator `div_loop`. The main function `div` is then a simple call to `div_loop` with the right initial counter value.

```
Definition div_F div x y := if y ≤? x then S (div (x-y) y) else 0.

Fixpoint div_loop (n:nat) :=
  match n with
    | 0 => fun _ _ => 0
    | S n => div_F (div_loop n)
  end.

Definition div x y := div_loop x x y.
```

One more time, extraction is straightforward and mostly amounts to replacing Coq keywords with Ocaml ones. The counter, whose type is `nat`, is kept by the extraction, even though it is morally useless for the computation. At the same time, removing it and replacing `div_loop` by an unbounded loop would change the semantics of the program at least for `y=0`: with the above definition,

`div 5 0` computes to `5`, while a Ocaml version without counter would loop forever. As a consequence, the extraction cannot be expected to detect and remove automatically such a "useless" parameter.

Using such an explicit counter is often an interesting compromise: the written Coq code is not exactly what we intended in the first place, but is close to it, there is no complex internal Coq object as with the methods we will study in the next sections, computations can be done both in Coq and after extraction, and the additional cost induced by the presence of the counter is often modest. Here for instance the `x` value would have been computed anyway. Another example of this technique can be found in module `Numtheory` of the Standard Library, where a `gcd` function is defined on binary numbers thanks to a counter that can be the depth (i.e. the logarithm) of these binary numbers.

2.3 A division by general recursion, historical approach

We can in fact build a Coq `div` function that will produce exactly the intended algorithm after extraction. Before presenting the modern ways of writing such a function with two frameworks recently added to Coq, let us first mention the historical approach. For a long time, the only possibility has been to play with accessibility predicates and induction principles such as `well_founded_induction`[4]. In this case, recursive functions do satisfy the structural constraint of Coq, not via their regular arguments, but rather via an additional logical argument expressing that some quantity is accessible. Recursive calls can then be done on quantities that are "more easily accessible" than before. This extra logical parameter is then meant to disappear during extraction. In practice, non-trivial functions are impossible to write as a whole with this approach, due to the numerous logical details to provide. Such functions are hence built piece by piece using Coq interactive tactics, as for proofs. Reasoning a posteriori on the body of such functions is also next to impossible, so key properties of these functions are to be attached to their output, via post-conditions `{ a:A | P a }`. Pre-conditions can also be added to restrict functions on a certain domain: for instance, `div` will be defined only for `y≠0`. Here comes the complete specification of our `div` and its implementation in a proof-like style:

```
Definition div : ∀x y, y≠0 → { z | z*y ≤ x < (S z)*y }.
Proof.
 induction x as [x Hrec] using (well_founded_induction lt_wf).
 intros y Hy.
 destruct (y ≤? x) as [Hyx|Hyx]. (* do we have y≤x or x<y ? *)
 (* first case: y≤x *)
 assert (Hxy : x-y < x) by omega.
 destruct (Hrec (x-y) Hxy y Hy) as [z Hz]. (* ie: let z = div (x-y) y *)
 exists (S z); simpl in *; omega. (* ie:  z+1 fits as (div x y) *)
 (* second case: x<y *)
 exists 0; omega.
Defined.
```

[4] See for instance Chap. 1 of [8] for more details on this topic.

We use `lt_wf`, which states that `<` is well-founded on natural numbers. When combined with `well_founded_induction`, this allows us to perform recursive calls at will on any strictly smaller numbers. Doing such a recursive call can be quite cumbersome: for calling `Hrec` on `x-y`, we need to have already built a proof `Hxy` stating that `x-y < x`. Without additional help such as comments, it is also very tedious to keep track on the algorithm used in such a proof. Fortunately, extraction can do it for us:

```
let rec div x y =
  if le_lt_dec y x then S (div (minus x y) y) else O
```

2.4 A division by general recursion with the Russell framework

The function-as-proof paradigm of the last section can be used on a relatively large scale, see for instance `union` and the few other non-structural operations on well-balanced trees in early versions of module `FSetAVL` in the Standard Library. But such Coq functions are hardly readable and maintainable, consume lots of resources during their definitions and in practice almost always fail to compute in Coq.

Recent versions of Coq include Russell, a framework due to M. Sozeau [10] that greatly eases the design of general recursive and/or dependently-typed functions. With this framework, bodies of functions can be written without being bothered by proof parts or by structural constraints. Simply, such definitions are fully accepted by Coq only when some corresponding proof obligations have been proved later on. For instance:

```
Definition id (n:nat) := n.

Program Fixpoint div (x:nat)(y:nat|y≠0) { measure id x }
  : { z | z*y ≤ x < (S z)*y }
  := if y ≤? x then S (div (x-y) y) else O.

Next Obligation. (* Measure decreases on recursive call : x-y < x *)
 unfold id; simpl; omega.
Qed.

Next Obligation. (* Post-condition enforcement : z*y ≤ x < (S z)*y *)
 destruct_call div; simpl in *; omega.
Qed.
```

After this definition and the proofs of corresponding obligations, a Coq object `div` is added to the environment, mixing the pure algorithm and the logical obligations. This object is similar to the dependently-typed `div` of the previous section, and its extraction produces the very same Ocaml code.

Russell framework can be seen as a sort of anti-extraction, in the spirit of C. Parent's earlier works [9]. Even if it is still considered as experimental, it is already quite usable. For instance, we have a version of `FSetAVL` where the aforementioned non-structural operations on well-balanced trees are written and proved using Russell.

2.5 A division by general recursion with the Function framework

An alternative framework can also be used to define our `div` function: Function, due to J. Forest and alii [4]. It is similar to Russell to some extent: algorithms can be written in a natural way, while proof obligations may have to be solved afterwards. Here, as for Russell, these proof obligations are trivial:

```
Function div (x y:nat)(Hy:y≠0) { measure id x } : nat :=
 if y ≤? x then S (div (x-y) y Hy) else 0.
Proof.
 intros; unfold id; omega.
Defined.
```

Moreover, as for Russell, this framework builds complex internal Coq objects, and extraction of these objects produces back precisely the expected code. But unlike Russell, Function is not meant to manipulate dependent types: in particular the `y≠0` pre-condition is possible here only since it is passed unmodified to the recursive call. On the contrary, Function focuses on the ease of reasoning upon functions defined with it, see for instance the `functional induction` tactics, allowing to prove separately properties of `div` that would have been post-conditions with Russell. Once again, the sensitive operations on well-balanced trees have be successfully tried and defined using Function.

3 Examples beyond ML type system

All our experiments on defining and extracting a division algorithm lead to legitimate Ocaml (or Haskell) code. But the type system of Coq allows us to build objects that have no counterparts in Ocaml nor Haskell. In this case, the type-checkers of these systems are locally bypassed by unsafe type casts (`Obj.magic` or `unsafeCoerce`). These unsafe type casts are now automatically inserted by the extraction in the produced code. We present now two of the various situations where such type casts are required.

3.1 Functions of variable arity

In Coq, a type may depend on an object such as a number. This allows us to write the type `nArrow` of n-ary functions (over `nat`), such that `nArrow 0 = nat` and `nArrow 1 = nat → nat` and so on.

```
Fixpoint nArrow n : Set := match n with
  | 0 => nat
  | S n => nat → nArrow n
end.
```

Furthermore, we can write a function `nSum` whose first parameter determines the number of subsequent parameters this function will accept (and sum together):

```
Fixpoint nSum n : nArrow (S n) :=
  match n return nArrow (S n) with
    | O => fun a => a
    | S m => fun a b => nSum m (a+b)
  end.

Eval compute in (nSum 2) 3 8 5.
```

The example `(nSum 2)` expects `(S 2) = 3` arguments and computes here 3+8+5=16. Without much of a surprise `nSum` cannot be typecheked in ML, so unsafe type casts are inserted during extraction:

```
let rec nSum n x =
  match n with
    | O -> Obj.magic x
    | S m -> Obj.magic (fun b -> nSum m (plus x b))
```

3.2 Existential structures

Another situation is quite common in developments on algebra: records can be used in Coq to define structures characterized by the existence of various elements and operations upon a certain type, with possibly some constraints on these elements and operations. For instance, let's define a structure of monoid, and show that `(nat,O,+)` is indeed a monoid:

```
Record aMonoid : Type :=
  { dom : Type;
    zero : dom;
    op : dom → dom → dom;
    assoc : ∀x y z:dom, op x (op y z) = op (op x y) z;
    zerol : ∀x:dom, op zero x = x;
    zeror : ∀x:dom, op x zero = x }.

Definition natMonoid :=
 Build_aMonoid nat 0 plus plus_assoc plus_0_l plus_0_r.
```

Proofs concerning monoids can then be done in a generic way upon an abstract object of type `aMonoid`, and be applied to concrete monoids such as `natMonoid`. This kind of approach is heavily used in CoRN development at Nijmegen. For the point of view of extraction, this `aMonoid` type hides from the outside the type placed in its `dom` field. Such dependency is currently translated to unsafe cast by extraction:

```
let natMonoid =
  { zero = (Obj.magic O); op = (Obj.magic plus) }
```

In the future, it might be possible to exploit recent and/or planned extensions of Haskell and Ocaml type-checkers to allow a nicer extraction of this example. Considering Haskell Type Classes and/or Ocaml's objects might also help.

4 Key features of extraction

Let us summarize now the current status of Coq extraction. The theoretical extraction function described in [7] is still relevant and used as the core of the extraction system. This function collapses (but cannot completely remove) both logical parts (living in sort `Prop`) and types. A complete removal would induce dangerous changes in the evaluation of terms, and can even lead to errors or non-termination in some situations. Terms extracted by this theoretical function are untyped λ-terms with inductive constructions, they cannot be accepted by Coq in general, nor by ML-like languages. Two separate studies of correctness have been done for this theoretical phase.

The correctness of this theoretical phase is justified in several steps. First, we prove that the reduction of an extracted term is related to the reduction of the initial term in a bisimulation-alike manner (see [7] or Sect. 2.3 of [8]. Since this first approach is really syntactic and cannot cope for instance with the presence of axioms, we then started a semantical study based on realizability (see Sect. 2.4 of [8]). Finally, differences between theoretical reduction rules and the situation in real languages have been investigated, especially in the case of Haskell (see Sect. 2.6 of [8]).

Even if the actual implementation of the extraction mechanism is based on this theoretical study, it also integrates several additional features. First, the untyped λ-terms coming from the theoretical phase are translated to Ocaml, Haskell or Scheme syntax. In addition, several simplifications and optimizations are performed on extracted terms, in order to compensate the frequent awkward aspect of terms due to the incomplete pruning of logical parts. Indeed, complete removal of proof parts is often unsafe. Consider for instance a partial application of the `div` function of section 2.5, such as `div 0 0 : 0≠0→nat`. This partial application is quite legal in Coq, even if it does not produce much, being blocked by the need of an impossible proof of `0≠0`. On the opposite, an extraction that would brutally remove all proof parts would produce `div 0 0 : nat` for this exemple, leading to an infinite computation. The answer of our theoretical model of extraction is to be very conservative and produce anonymous abstractions corresponding to all logical preconditions such as this `Hy:y≠0`. The presence of these anonymous abstractions permits a simple and safe translation of all terms, including partial applications. At the same time, dangerous partial applications are quite rare, so our actual implementation favors the removal of these anonymous abstractions, at least in head position of extracted functions, leading here to the expected `div` of type `nat→nat→nat`, whereas a special treatment is done for corresponding partial applications: any occurrences of `div 0 0` would become `fun _ -> div 0 0`, preventing the start of an infinite loop during execution.

Moreover, the extraction embeds an type-checker based on [5] whose purpose is to identify locations of ML type errors in extracted code. Unsafe type cast `Obj.magic` or `unsafeCoerce` are then automatically inserted at these locations. This type-checking is done accordingly to a notion of approximation of Coq types into ML ones (see Chap. 3 of [8]). In addition, Coq modules and functors are

supported by the Ocaml extraction, while coinductive types can be extracted into Ocaml, Haskell or Scheme.

5 Some significant Coq developments using extraction

A list of user contributions related to extraction can be found at `http://coq.inria.fr/contribs/extraction-eng.html`. Let us highlight some of them, and also mention some developments not (yet?) in this list.

- **CoRN**: This development done in Nijmegen contains in particular a constructive proof of the fundamental theorem of algebra. But all attempts made in order to compute approximations of polynomial roots by extraction have been unsuccessful up to now [2]. This example illustrates how a large, stratified, mathematically-oriented development with a peculiar notion of logical/informative distinction can lead to a nightmare in term of extracted code efficiency and readability.
- **Tait**: This proof of strong normalization of simply typed λ-calculus produces after extraction a term interpretor [1]. This study with H. Schwichtenberg et alii has allowed us to compare several proof assistants and their respective extractions. In particular Minlog turned out to allow a finer control of what was eliminated or kept during extraction, while Coq `Prop/Set` distinction was rather rigid. At the same time, Coq features concerning proof management were quite helpful, and the extracted code was decent, even if not as nice as the one obtained via Minlog.
- **FSets**: Started with J.C. Filliâtre some years ago [3], this certification of Ocaml finite set and map libraries is now included in the Coq Standard Library. This example has allowed us to investigate a surprisingly wide range of questions, in particular concerning specifications and implementations via Coq modules, or concerning the best style for expressing delicate algorithms (tactics or Fixpoint or Russell or Function). It has been one of the first large example to benefit from extraction of modules and functors.
- **CompCert**: X. Leroy and alii have certified in Coq a compiler from C (with minor restrictions) to powerpc assembly [6]. While this development is quite impressive, its extraction is rather straightforward, since Coq functions have been written in a direct, structural way. The compiler obtained by extraction is performing quite well.
- **Fingertrees**: In [10], M. Sozeau experiments with his Russell framework. The fingertrees structure, relying heavily on dependent types, is a good testcase for both this framework and the extraction. In particular, the code obtained by extraction contains several unsafe type casts, its aspect could be improved but at least it can be executed and is reasonably efficient.

6 Conclusion and future works

Coq extraction is hence a rich framework allowing to obtain certified programs expressed in Ocaml, Haskell or Scheme out of Coq developments. Even if some

details can still be improved, it is already quite mature, as suggested by the variety of examples mentioned above. This framework only seems to reach its limit when one tries to discover algorithm buried in large mathematical development such as CoRN, or when one seeks a fine control a la Minlog on the elimination performed by extraction. Most of the time, the `Prop/Set` distinction, which is a rather simple type-based elimination criterion, is quite efficient at producing reasonable extracted terms with little guidance by the user. Moreover, new tools such as Russel or Function now allow to easily define general recursive functions in Coq, hence allowing a wider audience to play with extraction of non-trivial Coq objects.

The correctness of this extraction framework currently rely on the theoretical studies made in [7, 8]. The next perspective is to obtain a mechanically-checked guarantee of this correctness. Work on this topic has already started with a student, S. Glondu. Starting from B. Barras CCI-in-Coq development, he has already defined a theoretical extraction in this framework and proved one of the main theorem of [7]. Another interesting approach currently under investigation is to use a Coq-encoded Mini-ML syntax as output of the current uncertified extraction, and have an additional mechanism try to build a proof of semantic preservation for each run of this extraction. Such extracted terms expressed in Mini-ML could then be fed to the certified ML compiler which is currently being built in the CompCert project of X. Leroy.

Some additional work can also be done concerning the typing of extracted code. For instance, thanks to advanced typing aspects of Haskell and/or Ocaml, examples such as the existential structure `aMonoid` may be typed some day without unsafe type casts. This would help getting some sensible program out of CoRN, which make extensive use of such structures. Manual experiments seem to show that Ocaml object-oriented features may help in this prospect. At the same time, some preliminary work has started in Coq in order to propose Haskell-like type classes, adding a support for these type classes to the Haskell extraction may help compensating the lack of module and functor extraction to Haskell.

References

[1] U. Berger, S. Berghofer, P. Letouzey, and H. Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82, 2005. Special issue.

[2] L. Cruz-Filipe and P. Letouzey. A Large-Scale Experiment in Executing Extracted Programs. In *12th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Calculemus'2005*, 2005.

[3] J.-C. Filliâtre and P. Letouzey. Functors for Proofs and Programs. In D. Schmidt, editor, *European Symposium on Programing, ESOP'2004*, volume 2986 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[4] D. Pichardie G. Barthe, J. Forest and V. Rusu. Defining and reasoning about recursive functions: a practical tool for the coq proof assistant. In *FLOPS'06*, volume 3945 of *LNCS*, 2006.

[5] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[6] Xavier Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd symposium Principles of Programming Languages*, pages 42–54. ACM Press, 2006.

[7] P. Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *TYPES 2002*, volume 2646 of *LNCS*. Springer-Verlag, 2003.

[8] P. Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq*. PhD thesis, Université Paris-Sud, July 2004. see `http://www.pps.jussieu.fr/~letouzey/download/these_letouzey_English.ps.gz`.

[9] C. Parent. *Synthèse de preuves de programmes dans le Calcul des Constructions Inductives*. thèse d'université, École Normale Supérieure de Lyon, January 1995.

[10] M. Sozeau. Program-ing Finger Trees in Coq. In *ICFP'07*, pages 13–24. ACM Press, 2007.