

A New Extraction for Coq

Pierre Letouzey

Université Paris-Sud, Laboratoire de Recherche en Informatique,
Bâtiment 490, F-91405 Orsay Cedex, France
`letouzey@lri.fr`

Abstract. We present here a new extraction mechanism for the Coq proof assistant. By extraction, we mean automatic generation of functional code from Coq proofs, in order to produce certified programs. In former versions of Coq, the extraction mechanism suffered several limitations and in particular worked only with a subset of the language. We first discuss difficulties encountered and solutions proposed to remove these limitations. Then we give a proof of correctness for a theoretical model of the new extraction. Finally we describe the actual implementation.

1 Introduction

The extraction mechanism of Coq is a tool for automatic generation of functional programs from Coq proofs. The main motivation for this mechanism is to produce certified programs. This concept of extraction is not new: such a tool exists in Coq since 1989 [10, 11], and also in other theorem provers like PX [5] or Nuprl [6]. But the previous implementation of the extraction tool in Coq suffers several limitations, and in particular accepts only a subset of Coq language. In this paper, we describe why and how this tool has been completely redesigned since Coq version 7.0.

The ability of extracting programs from proofs can be seen as a consequence of the Curry-Howard isomorphism, which explains in particular that a constructive proof is isomorphic to a functional program. Furthermore, since the internal representation of proofs inside Coq are λ -terms, one might say that a Coq proof term *is* a functional program. But we must temper this approach: these original Coq λ -terms need some transformations in order to get real programs. The following two sections present the two main kinds of such transformations.

Deletion of Logical Parts in Proofs. In a constructive proof, we can distinguish between informative and logical parts. The first ones are effectively used to construct the output value whereas the second ones are only present in the proof to ensure properties, for example termination of the algorithm. These logical parts are of no use during computation, and eliminating all this dead code usually leads to drastic gain in code size and execution speed. Although lot of work has been done in the domain of automatic dead code analysis [1, 2], our extraction relies on a declarative distinction made by the user. When defining

an object in `Coq`, the user has indeed to decide whether he puts it in the `Set` universe of informative objects, or in the `Prop` universe of logical objects. Then `Coq` typing system ensures that logical objects do not interfere during computations of informative objects. And finally, during extraction all objects of sort `Prop` will be eliminated. In fact, our method is different from automatic dead code analysis: although we do not eliminate informative dead code, we can simplify parts that do not satisfy the dead code criterion (see in particular the logical singleton elimination in the following).

Translation to a Real Functional Language. Since the beginning, the successive versions of `Coq` extraction are all aimed at producing source code for real functional languages. Today's available output languages are `Objective Caml` [8] and `Haskell` [3], and a `Scheme` [7] version is underway. There are three reasons for using external languages as output:

- First, certified code obtained this way can be easily integrated in larger developments. This is made possible by the production of readable interfaces¹. You can for example obtain a stand-alone program by adding hand-written I/O parts around a certified core, or design a certified library reused in several applications. This way, wider communities of programmers can benefit from extraction.
- A second practical interest in the use of external language is the speed gain. As said before, a `Coq` proof can be seen directly as a program. And its execution inside `Coq` is possible via the $\beta\delta\iota\zeta$ -reduction (using for instance the command `Eval Compute`). But this way of execution is quite comparable to interpreting, which is known to be far less efficient than compiling. A natural idea is then to use external compilers, such these of `Objective Caml` or `Haskell`. Notice however this internal `Coq` reduction will be improved soon: Benjamin Grégoire is currently working on a compiler for `Coq` terms [4]. But this compiler cannot perform elimination of logical parts, since it must be able to deal with open terms and reductions under lambdas (see Sect. 3.4).
- From a theoretical point of view, anyway, an external target language for extraction seems unavoidable when dealing with today's whole `Coq` theory. Of course, systems where extracted terms are internal objects have been proposed, like by Paulin [10] or more recently by Severi-Szasz [13]. But the theory in these studies are different from today's `Coq` theory. In particular, `Coq` now allows the definition of an informative fixpoint based on a logical decreasing measure (see for example the accessibility relation `Acc` and the associated fixpoint `Acc_rec`). An internal extraction would lead to an unguarded fixpoint that cannot be part of a strongly-normalizing system.

Overview. Section 2 presents the difficulties associated with our choices (acceptance of any `Coq` terms, removal of logical parts and translation to real pro-

¹ In fact, even the produced code itself is made as readable as possible.

gramming languages) and the solutions proposed. Section 3 formalizes our extraction in the Coq theoretical framework and states a correction result. Section 4 describes finally our current implementation of Coq extraction based on the previous theoretical study.

2 Challenges

All implementations of Coq extraction share the two principles given in introduction, that are removal of logical parts and translation to a real language. What characterizes our new implementation is that we now accept any Coq term, and that reduction of extracted terms is ensured to be correct both with lazy and strict evaluation. All these points combined lead to two kinds of challenges.

2.1 Problems with Elimination of Logical Parts

A first trouble with deletion of logical parts is the possible modification of the evaluation order. Let f be a function of type $(x : A)(P x) \rightarrow B$, that is a function expecting an informative argument x and a proof argument ensuring that x satisfies the precondition $(P x)$. Consider the two Coq terms $(f t)$ and $(f t p)$. They will evaluate quite differently, in particular the lack of the argument p will rapidly block the evaluation of $(f t)$. Now, if $\mathcal{E}$ is our extraction function, the expected extraction $\mathcal{E}(f)$ for f is a term of type $\mathcal{E}(A) \rightarrow \mathcal{E}(B)$, with only one argument, the informative one. And more generally it seems reasonable to remove all arguments of logical type, as in previous Coq extractions. But if we do this, $(f t)$ and $(f t p)$ will have the same extraction $(\mathcal{E}(f) \mathcal{E}(t))$, and this term will evaluate similarly to $(f t p)$, and quite differently from $(f t)$.

Moreover, in addition to efficiency problems, this modification of the evaluation order can be dangerous when combined with `False_rec`. This Coq term of type $(P : \text{Set})\text{False} \rightarrow P$ is used to deal with any absurd sub-cases. For example, when defining a function f of type $(x : \text{nat})(x \neq 0) \rightarrow \text{nat}$, it can be used in any sub-case where $x = 0$. During extraction this `False_rec` is translated into an exception, meaning that execution should never come to this point. Now the Coq partial application $(f 0)$ is legal, despite the fact that it will be impossible to provide a second argument of type $0 \neq 0$. The extraction $(\mathcal{E}(f) 0)$ will then raise this `False_rec` exception when executed. Our new extraction solve this problem by leaving some dummy abstractions `(fun _ → ...)` when needed.

Another issue is that Coq distinction between logical and informative parts is not black and white. We can form hybrid terms like `if b then nat else True` that is either informative (since `nat : Set`) or logical (since `True : Prop`) depending on the value of boolean b . This construction is made possible in Coq by the `Type` universe, which in particular contains both `Set` and `Prop`. In our example, `nat : Set` implies `nat : Type`, and similarly `True : Prop` implies `True : Type`. Finally `if b then nat else True` is well-typed of type `Type`. The previous extraction simply refused to extract such a term, and more generally any term using the sort `Type`. This drastic restriction allows a complete elimination of logical parts.

On the opposite, the goal of our new extraction is to be able to deal with any Coq term. We must then use a dummy constant (noted $\square$ in this paper) to fill all logical places that remains, like the `True` place above. Our approach is here quite similar to “pruning” methods [1, 2].

2.2 Translation Problems

As said before, the extracted code must be linkable with other parts of a development. So at least the types of the extracted terms must be understandable. We then have two choices: we can either generate source code for a particular language, or directly byte-code or binary object file associated with a readable interface. But in addition to the difficulties of this binary object generation, this would lead to a “black box” solution. We prefer to generate source code and benefit from compilers optimizations and also allow users to read the produced code if they want to. The Open Source community has shown that confidence in programs also comes via the readability of their sources.

This choice raises a new question: which language should we use as target language? All we need is a λ -calculus with inductive types. This explains the choice of ML-like languages (`Objective Caml` and `Haskell`). But these languages are typed, and their type systems are quite different from the Coq one. In particular in these languages there is no dependent types nor universes. As said before, the pragmatic choice of the old Coq extraction was to refuse Coq terms of sort `Type`. But even this restriction was not enough for always obtaining well-typed ML terms. For example the notion of polymorphism differs between ML and Coq. So it is clear that a straightforward translation of Coq λ -terms towards ML λ -terms will lead to potentially non-typable terms.

In the same time, we would like to stick as much as possible to this straightforward translation. First, a vast majority of usual Coq terms and inductives have a direct ML-typable counterpart. Secondly, the interface problem also claims for simplicity of translation. And finally any workaround by convenient encoding seems to correspond to a pre-compilation we precisely do not want to do. One particular encoding has been tried by Loïc Pottier [12], but it still produces ML-untypable terms.

So how do we intend to use typed compilers with potentially some non-typable extracted terms? Concerning `Objective Caml` we plan to use, as Pottier, an undocumented feature called `Obj.magic`. This function gives a generic `'a` type to any term. With this function we can bypass locally the `Objective Caml` type-checker. Automatic generation of these `Obj.magic` is underway. Concerning `Haskell`, some implementations propose a undocumented function `unsafeCoerce` equivalent to `Obj.magic`.

A last approach is to directly use an untyped functional language like `Scheme`. But unfortunately `Scheme` has no first-class inductive types (except the `Bigloo` implementation) so we need to encode them. And first speed tests made with an experimental `Scheme` extraction are clearly in favor of `Haskell/Objective Caml`.

3 Theoretical Framework

In this section, after a description of Coq logic system, we present a formalization of the extraction mechanism core, that is the removal of logical parts. Then we show that this removal does not affect the answers of informative computations.

3.1 The Calculus of Inductive Constructions

Due to lack of space, we only give here a partial presentation of the Calculus of Inductive Constructions (CIC for short), which is the underlying formal system of Coq. The reader will find a complete description of CIC in Chap. 4 of the Coq Reference Manual [14]. We use all the notations introduced there except environments and contexts. In fact, for simplicity reasons, we unify these two notions: here we use contexts as generic objects containing either assumptions $(x : T)$ or definitions $(x := t : T)$ or inductive declarations $\text{Ind}_n(\Gamma_I := \Gamma_C)$. In such an inductive declaration, n is the number of parameters, Γ_I is a context declaring the inductive types (like $\text{nat} : \text{Set}$) and Γ_C is a context declaring the constructors (like $(\text{O} : \text{nat}) :: (\text{S} : \text{nat} \rightarrow \text{nat})$).

Let us remind the CIC term syntax:

$$\begin{aligned} t ::= & s \mid x \mid (x : t)t \mid [x : t]t \mid [x := t]t \mid (t \ t) \\ & \mid \langle t \rangle \text{Cases } t \text{ of } t \dots t \text{ end} \\ & \mid \text{Fix } x_i \{x_1/k_1 : t := t \dots x_n/k_n : t := t\} \end{aligned}$$

s is here a sort, either **Set**, **Prop** or **Type**. x is an identifier, bound either locally (by a lambda, a product, ...) or somewhere in a context Γ . Then follow syntaxes for product, lambda, let-in, application, case elimination and fixpoint. The $\langle t \rangle$ annotation gives the type of the case elimination. Concerning fixpoints, for each i , k_i is a number expressing that the component x_i expects at least k_i arguments, the last one being the “guard” argument, i.e. an inductive argument used to control the reduction of the fixpoint (see reductions below). In Coq there is also a co-fixpoint construction, but we will not consider it here.

We will also use the CIC typing judgment $\Gamma \vdash t : T$ meaning that T is a valid type for t in the context Γ . And t is said to have sort s in the context Γ if there exists T such as $\Gamma \vdash t : T$ and $\Gamma \vdash T : s$. Note that we cannot speak of “the” type and “the” sort of t , since in CIC unicity of type does not hold in general, and so is unicity of sort. For example, an object of sort **Prop** is at the same time of sort **Type** (see conversion typing rule).

The Coq reductions are the following:

- (beta) $([x : X]t \ u) \rightarrow_\beta t\{x/u\}$
- (delta) $c \rightarrow_\delta t$ if the current context Γ contains $(c := t : T)$.
- (zeta) $[x := t]u \rightarrow_\zeta u\{x/t\}$
- (iota) $\langle P \rangle \text{Cases } C_i \ p_1 \dots p_k \ u_1 \dots u_n \text{ of } f_1 \dots f_n \text{ end} \rightarrow_\iota f_i \ u_1 \dots u_n$
if C_i is the i -th constructor of an inductive type with k parameters.
- (iota) Let F be the declarations $f_1/k_1 : A_1 := t_1 \dots f_n/k_n : A_n := t_n$. Then:
 $(\text{Fix } f_i \{F\} \ u_1 \dots u_{k_i}) \rightarrow_\iota (t_i\{f_j/\text{Fix } f_j \{F\}\}_{\forall j} \ u_1 \dots u_{k_i})$
if u_{k_i} (the “guard” argument) begins with a constructor.

The Coq reductions are *strong*: they can occur at any position internally thanks to the usual compatibility rules. We will also consider *weak* reductions later on, i.e. reductions occurring only at head positions. We will use $\rightarrow_r$ as an abbreviation for one step of any of Coq reductions $\rightarrow_\beta$, $\rightarrow_\delta$, $\rightarrow_\iota$ or $\rightarrow_\zeta$.

Before stating some stability properties of the CIC typing system used later on in the proofs, let us distinguish a particular class of CIC terms:

Definition 1. *An type scheme is a well-typed term accepting at least one type of the form $(x_1 : X_1) \dots (x_n : X_n)s$ with s a sort.*

In other words, a type scheme is a term that will become a type (that is something of type a sort) when applied to enough arguments.

Lemma 2 (Stability Lemma). *We have the following results:*

- (Subject Reduction) *When a term t reduces to u , if T is a type of t , then it is also a type of u . And if s is a sort of t , then it is also a sort of u .*
- *Secondly, when substituting a variable in a term, type might change. But there are some critical cases for which we have stability:*
 - *if t has sort Prop , so has $t\{x/u\}$*
 - *if t has an inductive type, so has $t\{x/u\}$*
 - *if t is a type scheme, so is $t\{x/u\}$*
- *Lastly, concerning applications, if t has sort Prop , so has $(t\ u)$, and if t is a type scheme, so is $(t\ u)$.*

When comparing with previous Coq extraction, notice that we can now have an application $(t\ u)$ of sort Prop without t having sort Prop . This comes from the Type universe: consider for example $([X : \text{Type}][x : X]x\ \text{True})$.

3.2 The Extraction Function

As explained before, we will eliminate any sub-term of sort Prop because they correspond to logical parts. In addition to this, we will also eliminate sub-terms corresponding to types, and more generally sub-terms which are type schemes. Why do we remove these type schemes? This choice is less natural than the Prop elimination. We are in particular aware of one Coq development whose main result is to built the type of some particular lattice [9]. Then the extraction of such a development only gives a dummy constant for this type. But we think this situation is exceptional. In usual developments, the computed results belong to some data types, for example inductive types like `bool`, `nat` or `Z`. And in these cases, we will see that type scheme sub-terms are dead code. Another justification for this choice is that unlike Coq, ML languages forbid the use of types as regular terms, enforcing a clear distinction between terms and types.

Let $\text{CIC}_\square$ be CIC plus one constant $\square$. Unlike CIC the new $\text{CIC}_\square$ is untyped. But the reductions in $\text{CIC}_\square$ are exactly the same as in CIC with $\square$ being non-reducible. Let us now define an extraction function from CIC to $\text{CIC}_\square$.

Definition 3. The extraction function $\mathcal{E}$ is defined by structural induction over any term t typable in a context Γ :

($\square$) If t is a type scheme or has sort *Prop* in context Γ , then $\mathcal{E}_\Gamma(t) = \square$

Otherwise we proceed structurally:

(*id*) $\mathcal{E}_\Gamma(x) = x$
 (*lam*) $\mathcal{E}_\Gamma([x : T]t) = [x : \square]\mathcal{E}_{\Gamma'}(t)$ where $\Gamma' = \Gamma :: (x : T)$
 (*let*) $\mathcal{E}_\Gamma([x := t]u) = [x := \mathcal{E}_\Gamma(t)]\mathcal{E}_{\Gamma'}(u)$ where $\Gamma' = \Gamma :: (x := t : T)$ and T is a type of t
 (*app*) $\mathcal{E}_\Gamma(u \ v) = (\mathcal{E}_\Gamma(u) \ \mathcal{E}_\Gamma(v))$
 (*cases*) $\mathcal{E}_\Gamma(<P>\text{Cases } e \text{ of } f_1 \dots f_n \text{ end}) =$
 $<\square>\text{Cases } \mathcal{E}_\Gamma(e) \text{ of } \mathcal{E}_\Gamma(f_1) \dots \mathcal{E}_\Gamma(f_n) \text{ end}$
 (*fix*) $\mathcal{E}_\Gamma(\text{Fix } f_i \{f_1/k_1 : A_1 := t_1 \dots f_n/k_n : A_n := t_n\}) =$
 $\text{Fix } f_i \{f_1/k_1 : \square := \mathcal{E}_{\Gamma'}(t_1) \dots f_n/k_n : \square := \mathcal{E}_{\Gamma'}(t_n)\}$
 where $\Gamma' = \Gamma :: (f_1 : A_1) :: \dots :: (f_n : A_n)$

And the extraction of a context is defined by:

(*nil*) $\mathcal{E}(\square) = \square$
 (*def*) $\mathcal{E}(\Gamma :: (c := t : T)) = \mathcal{E}(\Gamma) :: (c := \mathcal{E}_\Gamma(t) : \square)$
 (*ax*) $\mathcal{E}(\Gamma :: (x : T)) = \mathcal{E}(\Gamma) :: (x : \square)$
 (*ind*) $\mathcal{E}(\Gamma :: \text{Ind}_n(\Gamma_I := \Gamma_C)) = \mathcal{E}(\Gamma) :: \text{Ind}_n(\mathcal{E}(\Gamma_I) := \mathcal{E}(\Gamma_C))$

Clearly, $\mathcal{E}$ is a “pruning” function: it only replaces some sub-terms by $\square$. In particular no modification of the structure can occur. In the actual implementation, another step of extraction is dedicated to structure modifications, see Sect. 4 for a brief description.

3.3 Strong Reduction in a Restriction of $\text{CIC}_\square$

We want to establish that evaluating an extracted term leads to a meaningful result, for example *true* or *false* for a boolean original CIC term. And of course, we also want this result to be compatible with the answer of the corresponding CIC computation.

We will proceed by simulating CIC derivations inside $\text{CIC}_\square$ and vice-versa. The difficulty is that the simulation might end on a term still having redexes in CIC but whose counterpart in $\text{CIC}_\square$ is no more reducible. In fact, we have potentially three cases of CIC redexes corresponding to $\text{CIC}_\square$ non-redexes:

1. a β -redex $([x : X]t \ u)$ corresponding to a non-redex $(\square \ u')$
2. a ι -redex $<\dots>\text{Cases } e \text{ of } \dots \text{ end}$ corresponding to a non-redex $<\dots>\text{Cases } \square \text{ of } \dots \text{ end}$
3. a fixpoint redex $(\text{Fix } f_i \{\dots\} u_1 \dots u_n)$ corresponding to a non-redex, either
 - (a) $(\square \ u'_1 \dots u'_n)$
 - (b) $(\text{Fix } f_i \{\dots\} u_1 \dots \square)$ (the “guard” argument is now a blocking $\square$)

In case 1, we should morally have $([x : X]t \ u)$ corresponding to $\Box$, since stability lemma says that **Prop**-sorted terms and type scheme are preserved by application. This “lack of precision” of an extracted term can appears after some steps of reduction, as in the next example:

Example 4.

$$\begin{aligned} t &= ([X : \text{Type}][f : \text{nat} \rightarrow X][g : X \rightarrow \text{nat}](g \ (f \ 0)) \ \text{Prop} \ [_ : \text{nat}]\text{True}) \\ \mathcal{E}(t) &= ([X : \Box][f : \Box][g : \Box](g \ (f \ 0)) \ \Box \ \Box) \rightarrow_{\beta}^* [g : \Box](g \ (\Box \ 0)) \end{aligned}$$

If we need to, we will bypass this problem (and case 3a) via an ad-hoc reduction:

Definition 5. *The $\Box$ -reduction is defined by the following rule: $(\Box \ u) \rightarrow_{\Box} \Box$*

Situation of case 2 is quite different. Unlike Ex. 4 where a lambda becomes logical after reduction, a **Cases** cannot change the inductive type upon which it is performed. And $\mathcal{E}$ eliminates all **Cases** sub-terms of sort **Prop**. So how can case 2 occur afterwards? Because the **CIC** typing system allows a **Cases** logical elimination to produce something informative when the elimination is performed upon a term whose logical inductive type has either:

1. zero constructor (empty inductive)
2. one constructor whose arguments are all logical, parameters put aside (singleton inductive)

This is in fact a first exception to our introduction statement “logical objects do not interfere during computations of informative objects”. The second exception is case 3b: the “guard” argument of a fixpoint can be of a logical inductive type whereas the whole fixpoint term is informative.

Just suppose for a while that we forbid these particular typing features. Let us consider until the end of this section two systems CIC^- and $\text{CIC}_{\Box}^-$ that are **CIC** and $\text{CIC}_{\Box}$ with the following restrictions:

- (i) Logical empty elimination should not produce informative terms.
- (ii) Logical singleton elimination should not produce informative terms.
- (iii) For every component f_i of a fixpoint, its “guard” argument should not be logical when the type of f_i is not.

Then we have the following result for strong reductions of extracted terms:

Definition 6. *A type T is said to be logic-free if for all closed normal term t of type T we have $\mathcal{E}(t) = t$*

Theorem 7. *Let t be a well-typed closed CIC^- term whose type T is logic-free. Then all reductions of $\mathcal{E}(t)$ terminate on the CIC^- normal form of t .*

We will not prove this result here, since it is somehow less important than incoming Theorem 15 whereas the proofs of these two theorems are similar. Note that the use of $\Box$ -reductions is not needed here, thanks to the logic-free hypothesis combined with Restrictions (i), (ii) and (iii). This will change afterwards.

3.4 Weak Reductions in the Complete CIC_□

Since our goal is an extraction mechanism accepting any Coq term, we now need to remove these Restrictions (i), (ii) and (iii). Restriction (i) concerning empty inductive is in fact easy to remove, since ι -reduction upon an empty inductive will never happen. We can just ignore these **Cases**, and translate them later on to exceptions (see the **False_rec** discussion in Sect. 2.1). Now, removing restrictions (ii) and (iii) will oblige us to adapt reduction stately of extracted terms, leaving strong reduction (i.e. reduction allowed under lambdas) for weak reduction. Anyway, this has to be done at some point, since our functional target languages do not allow strong reduction.

Singleton Elimination. If H is a logical equality, $\langle \text{nat} \rangle \text{Cases } H \text{ of } O \text{ end}$ can be reduced to give O without knowing the exact value of H , hidden behind a $\square$. We can similarly reduce any logical singleton elimination, but this is dangerous when combined with strong reduction, and may lead to type errors. Consider for example the following **cast** function that transforms an integer into a boolean if we can prove that integers and booleans are equal:

```
cast = [H : nat == bool][n : nat] <[t : Set] t > Cases H of n end
```

Then from previous remarks, in

```
[H : nat == bool] <... > Cases (cast H O) of ... end
```

the $(\text{cast } H \text{ } O)$ would reduce to O , whereas the surrounding **Cases** expect a boolean. Clearly, if we forbid reduction under lambdas, the problem disappears.

Fixpoints with Logical “Guards”. The problem is now to reduce an informative fixpoint whose “guard” argument is logical. Of course, the first idea is to remove the “guard” condition on this kind of fixpoint reduction. But combined with strong reduction, this may lead to looping evaluation. The following **loop**² expects an hypothetic proof of accessibility of O via **gt** (greater than) and then does infinitely many recursive calls: $(F \text{ } n)$ calls $(F \text{ } (S \text{ } n))$.

```
loop = [Ax : (Acc nat gt O)]
      Fix F {F/2 : (a : nat) (Acc nat gt a) → nat :=
        [a : nat][b : (Acc nat gt a)] (F (S a) H)}
      0 Ax
```

where $H = (\text{Acc_inv } a \text{ } b \text{ } (S \text{ } a) \text{ } (\text{gt_Sn_n } a))$ is a proof³ of accessibility for $(S \text{ } a)$. Extraction gives:

```
ℰ(loop) = [Ax : □]
          Fix F {F/2 : □ :=
            [a : nat][b : □] (F (S a) □)}
          0 □
```

² built on the **Acc_rec** model, see Coq standard library

³ see the Coq standard library for definitions of **Acc_inv** and **gt_Sn_n**

If we remove the “guard” condition here, then this term is strongly reducible even without being applied, and gives $[Ax : \Box]\text{Fix } F \{ \dots \} (\text{S O}) \Box$ and so on ...

Modification of the Reduction. We first need some extra annotations on Cic well-typed terms: If the type of t is a singleton inductive type whose sole constructor expects n logical arguments, then $\langle \dots \rangle \text{Cases } t \text{ of } f \text{ end}$ will now be noted $\langle \dots \rangle \text{Cases}_n t \text{ of } f \text{ end}$. These new annotations should be kept by the $\mathcal{E}$ function. We can now modify the reductions of $\text{Cic}_\Box$ to deal with $\Box$ blocking ι -reduction.

Definition 8 (New ι -reduction). *The ι -reduction upon $\text{Cic}_\Box$ terms is now:*

(iota) $\langle P \rangle \text{Cases } C_i p_1 \dots p_k u_1 \dots u_n \text{ of } f_1 \dots f_n \text{ end} \rightarrow_\iota f_i u_1 \dots u_n$

(iota) $\langle P \rangle \text{Cases}_n \Box \text{ of } f \text{ end} \rightarrow_\iota f \underbrace{\Box \dots \Box}_n$

(iota) *Let F be the declarations $f_1/k_1 : A_1 := t_1 \dots f_n/k_n : A_n := t_n$. Then:*
 $(\text{Fix } f_i \{F\} u_1 \dots u_{k_i}) \rightarrow_\iota (t_i \{f_j / \text{Fix } f_j \{F\}\}_{\forall j} u_1 \dots u_{k_i})$
if u_{k_i} is equal to $\Box$ or begins with a constructor.

Let us now restrain the reductions to forbid strong reduction. For every possible reduction, we define an associated weak reduction.

Definition 9 (Weak Reductions). *The reductions $\rightarrow_{\beta_w}, \rightarrow_{\iota_w}, \rightarrow_{\delta_w}, \rightarrow_{\zeta_w}$ and $\rightarrow_\Box$ are defined from the same base cases as $\rightarrow_\beta, \rightarrow_\iota, \rightarrow_\delta, \rightarrow_\zeta, \rightarrow_\Box$ respectively, and from the following restricted compatibility rules:*

$$\frac{u \rightarrow_\Box v}{(u \ t) \rightarrow_\Box (v \ t)} \qquad \frac{u \rightarrow_\Box v}{(t \ u) \rightarrow_\Box (t \ v)}$$

$$\frac{u \rightarrow_\Box v}{\langle P \rangle \text{Cases } u \text{ of } \dots \text{ end} \rightarrow_\Box \langle P \rangle \text{Cases } v \text{ of } \dots \text{ end}}$$

And as for $\rightarrow_r$, the full weak reduction $\rightarrow_{r_w}$ is $\rightarrow_{\beta_w} \cup \rightarrow_{\iota_w} \cup \rightarrow_{\delta_w} \cup \rightarrow_{\zeta_w}$.

In fact, this $\rightarrow_{r_w}$ can be seen as a generalization of both Objective Caml CBV strategy and Haskell CBN strategy. The last main step toward the actual reduction strategy of these languages is to fix an evaluation order: should we reduce first the head or the arguments?

An important point to mention: from now to the end of this paper we will only consider contexts with no assumptions. This restriction is needed because reduction in presence of an axiom is analog to strong reduction under a lambda. In particular we may loose the fundamental fact that a closed inductive term will necessarily reduce to a term beginning with a constructor. Of course not every axiom breaks this property, but for simplicity sake we will forbid all of them.

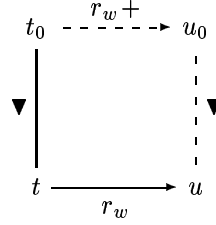
To study evaluation of extracted terms, we need an invariant $\blacktriangleleft$ preserved by reduction. Studying $\mathcal{E}$ directly is not a good choice, since $\mathcal{E}$ does not behave well with respect to reduction: if $t \rightarrow_r u$, we might not have $\mathcal{E}(t) \rightarrow_r \mathcal{E}(u)$, see for example term t of Ex. 4.

Definition 10. Let Γ_0 and t_0 be a context and a term of CIC . Let Γ and t be a context and a term in $\text{CIC}_\square$. We say that $(\Gamma, t) \triangleleft (\Gamma_0, t_0)$ iff:

- ($\triangleleft_1$) t_0 is well-typed in context Γ_0
- ($\triangleleft_2$) t and t_0 differ only at positions where t contains $\square$, and Γ and Γ_0 differ only at positions where Γ contains $\square$
- ($\triangleleft_3$) any sub-term in t_0 and Γ_0 corresponding to a $\square$ in t or Γ has sort *Prop* or is a type scheme
- ($\triangleleft_4$) all *Cases* (or *Cases_n*) in t and Γ are upon inductive types that are either informative or logical singleton or empty.

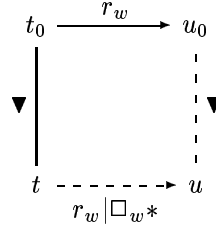
Lemma 11. If t is a CIC term typable in context Γ , then $(\mathcal{E}(\Gamma), \mathcal{E}_\Gamma(t)) \triangleleft (\Gamma, t)$.

Theorem 12. If $(\Gamma, t) \triangleleft (\Gamma_0, t_0)$ and $t \rightarrow_{r_w} u$, then there exists u_0 such as $t_0 \rightarrow_{r_w} u_0$ and $(\Gamma, u) \triangleleft (\Gamma_0, u_0)$.



Proof. See Appendix A. □

Theorem 13. Suppose that $(\Gamma, t) \triangleleft (\Gamma_0, t_0)$ and $t_0 \rightarrow_{r_w} u_0$. Then there exists u such as $(\Gamma, u) \triangleleft (\Gamma_0, u_0)$ and either $t \rightarrow_{r_w} u$ or $t \rightarrow_{\square_w}^* u$.



Proof. See Appendix B. □

Definition 14. A data-type is an inductive type D whose constructors expect only arguments of type D or of type another data-type.

Theorem 15. Let t be a well-typed closed CIC term whose type T is a logic-free data-type. Then all derivations of $\mathcal{E}(t)$ via $\rightarrow_{r_w \square_w}^*$ terminate on the CIC normal form of t .

Proof. See Appendix C. □

In practice, the logic-free data-type condition is not so restrictive. The usual data types like `bool`, `nat`, or `Z` verify it. And anyway we can state a generalized result for any type using an ad-hoc observational equivalence: plunged into a boolean context, t and $\mathcal{E}(t)$ will compute to the same value.

4 Implementation Considerations

After this theoretical proof of correctness, we now need to translate our $\text{CIC}_{\Box}$ system to real languages like Objective Caml or Haskell.

Removing Singleton Eliminations. Our theoretical study used a particular reduction rule for singleton eliminations. Of course this ad-hoc rule does not exist in Objective Caml nor in Haskell. But in fact we can eliminate all singleton Cases_n during extraction, via a more general singleton elimination rule $\langle \dots \rangle \text{Cases}_n e \text{ of } f \text{ end} \rightarrow (f \Box \dots \Box)$, used even under lambdas. We can prove that type error possibilities showed in Sect. 3.4 are avoided because all other reductions are still used in a weak way.

Implementing $\Box$. After this removal of singleton eliminations, $\Box$ now never comes in head position during a lazy evaluation of an informative term. We can then implement $\Box$ by an error in Haskell. Objective Caml situation is not so simple: we have seen previously that we may need $\Box$ -reduction like $(\Box u) \rightarrow_{\Box} \Box$. So the usual implementation of a dummy constant like $\Box$ by a “unit” value is inappropriate. We rather need something like `let rec  $\Box$  x = Obj.magic  $\Box$` , with a `Obj.magic` needed because this term is not well-typed. Gladly, most of the time the type of the sub-term replaced by $\Box$ shows it could be applied only to a limited number of arguments. Then a term like `fun _ ... _  $\rightarrow$  ()` can be used.

Implementing Fixpoints. The Objective Caml/Haskell fixpoint reduction we use is different from our theoretical weak fixpoint ι -reduction. First the real reduction has no control of the argument number. We emulate this control by translating a fixpoint component f/n to a function with at least n arguments. This is done via η -expansion if necessary, for example `Fix  $f \{ f/1 : A := t \}$` will give `let rec  $f$  x = ( $\mathcal{E}(t)$  x)` if t does not begin with a lambda. This way, the reduction of a fixpoint without enough arguments will be blocked. The second difference between real and theoretical reduction is the control on the “guard” argument. But in fact the Objective Caml evaluation strategy always satisfies this condition: the guard argument will be reduced to a value before evaluating the fixpoint. And the only values possible for inductive terms begin with a constructor. Haskell seems different at first look: the “guard” argument is not evaluated first. But in fact this does not change anything, since this “guard” argument will evaluate necessarily to a constructor as soon as it is used.

Removing Dummy Arguments. Our limited $\mathcal{E}$ function just does some pruning, and leaves unchanged the number of arguments expected by a function. If we consider again as in Sect. 2.1 the example of a function f of type $(x : A)(P x) \rightarrow B$, then if $\mathcal{E}(f)$ is typable it will have a type like $\mathcal{E}(A) \rightarrow \Box \rightarrow \mathcal{E}(B)$ if we also note $\Box$ the type of our implementation of $\Box$. Our implementation of extraction now removes these external dummy lambdas when extracting the body

of Coq constants. We just always leave at least one lambda to protect against undue evaluation and `False_rec` problem. For example, if our function f is of the form $[x : A][p : (P\ x)]t$, then we produce `let f x =  $\mathcal{E}(t)$` . Of course, since these optimized versions of the constants expect less arguments, we also need to compensate whenever these constants are used: $(f\ a\ p)$ in Coq becomes $(f\ \mathcal{E}(a))$ after extraction and $(f\ a)$ becomes `fun _  $\rightarrow$  (f  $\mathcal{E}(a)$ )`.

Code Optimizations. Several optimizations are done in order to produce more efficient and more readable code:

- After extraction, the existential inductive type `sig` has only one constructor `exist`, with exactly one informative argument. We can then say that $(\text{exist } x)$ is isomorphic to x , and remove completely this inductive type, as well as all similar inductive types.
- In a strict language like **Objective Caml**, it is generally a good idea to inline functions that might not need some of their arguments. This way, we skip the evaluation of these useless arguments. This situation typically occurs with the recursive function associated to an inductive type, like `nat_rec` for `nat`.
- There are also some small simplifications that can be done after extraction, like `<bool>Cases e of true true end  $\rightarrow$  true`. Of course no programmer will ever write such a `Cases`. But this kind of terms may appear in proofs, when the logical part of the proof has needed such an elimination of e .

The Current Implementation. All features described in this paper have been implemented and are part of the Coq distribution⁴ version 7.3. Usage of this extraction tool is explained in chapter 17 of the Reference Manual [14]. The extraction source code can be found in sub-directory `contrib/extraction` of the Coq sources, and is made of about 3,000 lines of **Objective Caml** including 900 lines for the theoretical core and 700 lines for the optimizations.

One Example. Let us try our implementation on a toy definition of a predecessor function with pre- and post-conditions. We first define our `pred` via tactics, then we print the internal `CIC` term generated and finally show the extraction of `pred` in **Objective Caml** syntax.

```
> Definition pred : (n:nat)~0=n -> {p:nat|n=(S p)}.
> Destruct n.
> Intros.
> Elim H; Trivial.
> Exists n0; Trivial.
> Defined.

> Print pred.
```

⁴ available at <http://coq.inria.fr>

```

pred =
[n:nat]
<[n0:nat]~0=n0->{p:nat | n0=(S p)}>
  Cases n of
    0 =>
      [H:(~0=0)](False_rec {p:nat | 0=(S p)} (H (refl_equal nat 0)))
    | (S n0) =>
      [_: (~0=(S n0))](
        (exist nat [p:nat](S n0)=(S p) n0 (refl_equal nat (S n0)))
      )
  end
: (n:nat)~0=n->{p:nat | n=(S p)}

> Extraction pred.
let pred = function
| 0 -> assert false (* absurd case *)
| S n0 -> n0

```

We can see in particular that the extraction has removed the logical argument of type $0 \neq n$, that the `False_rec` has been translated to an error, and that there is no trace left of the existential construction, seen as the identity.

A Benchmark Suite. Several contributions submitted by Coq users have been equipped with an automatic extraction test (see extraction documentation in [14]). This result in stand-alone certified programs in various domains, including:

- Boolean tautology checkers
- A Calculus of Constructions type-checker
- First-Order Unification
- Higman’s Lemma (with one manual `Obj.magic`)

for a total of more than 6,000 lines of extracted code. For example, one of the test computes (Fibonacci 10000) in less of one minute on a modern computer, using `Z` datatype and the matrix algorithm. We are currently aware of only one Coq development too huge to be tractable by our implementation.

5 Conclusion

To sum up, let’s compare situation of extraction in Coq version 6 and Coq version 7. Using Coq 6, obtaining a certified result via the computation of an extracted code was possible only under the following conditions:

1. The Coq term does not use sort `Type` nor strong elimination, otherwise the extraction rejects it.
2. The extracted term is accepted by the Objective Caml or Haskell type-checker.
3. The execution does not raise an exception due to the `False_rec` problem.
4. The execution terminates without shortage of resources (stack or memory).

That was quite restrictive. Since our new implementation in Coq version 7, points 1 and 3 are obsolete: the extraction accept any Coq term and the execution is now guaranteed to be correct. We are currently working on point 2, that is ensuring that extracted programs always type-check.

The readability of the extracted code has been greatly improved in our new extraction. And concerning efficiency, the old extraction and the new one are comparable. The new one is sometimes slightly better because of extra optimizations, and sometimes slightly worse because of residual occurrences of the dummy $\square$. We still work on transformations aimed at removing useless $\square$, and on other optimizations.

Acknowledgment:

Many thanks to Christine Paulin and Jean-Christophe Filliâtre, two great wizards of Coq, respectively for their help in the theoretical study and the implementation.

References

- [1] S. Berardi. Pruning simply typed λ -calculi. *Journal of Logic and Computation*, 6(2), 1996.
- [2] L. Boerio. Extending pruning techniques to polymorphic second order λ -calculus. In *Proceedings ESOP'94*, volume 788. Lecture Notes in Computer Science, 1994.
- [3] S. Peyton Jones et al. *Haskell 98, A Non-strict, Purely Functional Language*, 1999. Available at <http://haskell.org/>.
- [4] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proceedings ICFP'2002*. To appear.
- [5] S. Hayashi and H. Nakano. Px, a computational logic. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1987.
- [6] P. Jackson. *The Nuprl Proof Development System, Version 4.1 Reference Manual and User's Guide*. Cornell University, Ithaca, NY, 1994.
- [7] R. Kelsey, W. Clinger, and J. Rees (eds.). *Revised⁵ Report on the Algorithmic Language Scheme*, 1998. Available at <http://www.scheme.org/>.
- [8] X. Leroy, J. Vouillon, and D. Doliguez. *The Objective Caml system – release 3.04*, 2002. Available at <http://caml.inria.fr/>.
- [9] D. Monniaux. Réalisation mécanisée d'interpréteurs abstraits. Rapport de DEA, Université Paris VII, 1998.
- [10] C. Paulin-Mohring. Extracting F_ω 's programs from proofs in the Calculus of Constructions. In *Sixteenth Annual ACM Symposium on Principles of Programming Languages*, Austin, January 1989. ACM.
- [11] C. Paulin-Mohring. *Extraction de programmes dans le Calcul des Constructions*. Thèse d'université, Paris 7, January 1989.
- [12] L. Pottier. Extraction dans le calcul des constructions inductives. In *Journées Francophones des Langages Applicatifs*, 2001.
- [13] P. Severi and N. Szasz. Studies of a theory of specifications with built-in program extraction. *Journal of Automated Reasoning*, 27(1), 2001.
- [14] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 7.3*, May 2002. Available at <http://coq.inria.fr/doc-eng.html>.

A Proof of Theorem 12

We proceed by case on the reduction used:

- The reduction done is a singleton ι_w -reduction like

$$\langle \dots \rangle \text{Cases}_n \square \text{ of } f \text{ end} \rightarrow_{\iota} (f \square \dots \square).$$

Then the definition of the compatibility rules for the ι_w -reduction combined with the no-axiom hypothesis implies that the counterpart a_0 in t_0 of the eliminated $\square$ is well-typed in an assumption-free context. Since a_0 has an inductive type, it can then be reduced to a term $(C \ p_1 \ \dots \ p_k \ v_1 \ \dots \ v_n)$ with C a constructor. And this reduction can even be done with a weak strategy. In fact C is the unique constructor of this singleton inductive type, and in addition to the parameters, C has exactly n arguments. So in t_0 the sub-term $\langle \dots \rangle \text{Cases}_n \ a_0 \text{ of } f_0 \text{ end}$ reduces to $(f_0 \ v_1 \ \dots \ v_n)$ in at least one step of r_w -reduction. Let u_0 be t_0 after those reductions. To check that $u \triangleleft u_0$ we only have to check that $(f \square \dots \square) \triangleleft (f_0 \ v_1 \ \dots \ v_n)$. And this is trivial, since in particular all v_i are logical as arguments of a singleton inductive constructor.

- The reduction is a fixpoint ι_w -reduction when the “guard” argument in t is $\square$. Then the corresponding “guard” argument g_0 in t_0 has a logical inductive type and can be reduced as in the previous case to a term h_0 beginning by a constructor. Then the fixpoint can be reduced in t_0 . And finally the obtained terms in $\text{Cic}_{\square}$ and in Cic are still linked by $\triangleleft$.
- The reduction is a β_w -reduction. Given the definition of $\triangleleft$, the β -redex in t has necessarily a β -redex counterpart in t_0 . Reducing this redex of t_0 leads to a term u_0 . And we have $u \triangleleft u_0$, because of the following property of $\triangleleft$:

Lemma 16. *If $a \triangleleft a_0$ and $b \triangleleft b_0$ then $a\{x/b\} \triangleleft a_0\{x/b_0\}$*

Proof.

- Points $(\triangleleft_1)$ and $(\triangleleft_2)$ are clear.
- Concerning $(\triangleleft_3)$, let c_0 be the counterpart in $a_0\{x/b_0\}$ of a $\square$ in $a\{x/b\}$. This $\square$ comes from a previous $\square$ either in a or in b . Let d_0 be the counterpart in a_0 or b_0 of this previous $\square$. We have that either $c_0 = d_0$ or $c_0 = d_0\{x/b_0\}$. Hypothesis $(\triangleleft_3)$ concerning a and b shows that d_0 is a type scheme or a **Prop**-sorted term. Using the stability Lemma 2, we have that c_0 is also a type scheme or a **Prop**-sorted term.
- Concerning $(\triangleleft_4)$, any **Cases** of $a\{x/b\}$ comes from a previous **Cases** in a or in b . And the inductive sub-term eliminated by this **Cases** cannot change of sort under substitution.

□

- Remaining cases $(\rightarrow_{\delta_w}, \rightarrow_{\zeta_w}$ and end of $\rightarrow_{\iota_w})$ are similar to the $\rightarrow_{\beta_w}$ case.

B Proof of Theorem 13

- If the redex r reduced in t_0 correspond to a similar redex in t , then we can reduce this redex in t and get a convenient u .
- If r is completely inside a sub-term of t_0 corresponding to a $\square$ of t , then we can just take $u = t$.
- We now comes to all other intermediate positions of r with respect to $\square$ (studied as cases 1, 2, 3a and 3b in Sect. 3.3):
 - If r is a β -redex, the only such situation is $r = ([x : X]a \ b)$ in t_0 corresponding to $(\square \ b')$ in t . We can then simulate the β -reduction in t_0 by a $\square$ -reduction in t .
 - If r is a δ - or ζ -redex, there is no intermediate situation.
 - If r is a **Cases** ι -redex, the remaining situation is


```

          <P>Cases e of ... end
          in t_0 corresponding in t to
          <P'>Cases □ of ... end.
          
```

 But we know via ($\blacktriangleleft_4$) that this **Cases** in t is either an informative or empty or singleton elimination. Informative elimination is impossible, otherwise e would have an informative inductive type, in contradiction with ($\blacktriangleleft_2$). An empty elimination cannot be reduced (no constructor). So the only possibility is the one of a singleton elimination we can precisely reduce now via the new ι -reduction.
 - If r is a **Fix** ι -redex, there is two sub-cases. If the **Fix** has disappeared from t but not all arguments composing the redex (case 3a), then we can simulate the ι -reduction in t_0 via some $\square$ -reductions. And if the **Fix** appears in t but not the “guard argument” (case 3b), then we reduce via the new fixpoint ι -reduction rule.

C Proof of Theorem 15

First, we clearly have that if $a \blacktriangleleft b$ and $a \rightarrow_{\square} c$, then $c \blacktriangleleft b$.

Consider now a derivation of $\mathcal{E}(t)$ via $\rightarrow_{r_w \square_w}^*$. We obtain an associated r_w -derivation of t by using alternatively the Theorem 12 or the previous remark at each step of $\rightarrow_{r_w \square_w}$. And for each step of r_w -reduction of $\mathcal{E}(t)$ there is at least one step of r_w -reduction done on t . Since in C1C r_w -reductions are always finite, then there is only a finite number of r_w -reductions during the derivation of $\mathcal{E}(t)$. And finally we cannot have infinitely many consecutive steps of $\square_w$ -reductions, since they decrease the size of the term. So our derivation of $\mathcal{E}(t)$ via $\rightarrow_{r_w \square_w}^*$ necessarily ends on a normal term.

Let u be such a normal term derived from $\mathcal{E}(t)$ and let u_0 be the corresponding C1C term derived from t via the previous method. Because of the definition of a data-type, u_0 can still be reduced via $\rightarrow_{r_w}$ zero or more times toward the C1C normal form t_0 of t . By Theorem 13, u can also be reduced accordingly via $r_w | \square_w^*$, except that u is already normal, so nothing is done, and we have $u \blacktriangleleft t_0$. By the logic-free hypothesis, we have also $\mathcal{E}(t_0) = t_0$, which means that t_0 has no **Prop**-sorted or type scheme sub-terms. So there cannot exist any $\square$ in u , and finally $u = t_0$.