Towards Rewriting in Coq[⋆]

Jacek Chrząszcz and Daria Walukiewicz-Chrząszcz

Institute of Informatics, Warsaw University
ul. Banacha 2, 02-097 Warsaw, Poland
{chrzaszcz,daria}@mimuw.edu.pl

This work is dedicated to Jean-Pierre Jouannaud who is unquestionably a spiritus movens of the research on bringing rewriting to Coq.

Abstract. Equational reasoning in Coq is not straightforward. For a few years now there has been an ongoing research process towards adding rewriting to Coq. However, there are many research problems on this way. In this paper we give a coherent view of rewriting in Coq, we describe what is already done and what remains to be done.
We discuss such issues as strong normalization, confluence, logical consistency, completeness, modularity and extraction.

1 Introduction

Large part of research in modern theoretical computer science is concerned with formalizing mathematical reasoning. On one hand various formal calculi are being developed which model mathematical notions and proofs. On the other hand computer programs are written which implement these formalisms together with tools which help the users formalize and solve their mathematical problems.

In this paper we concentrate on Coq [18], a proof assistant based on type theory and the Curry-Howard correspondence, which relates formulas to types and their proofs to terms of these types.

Calculus of constructions. The first version of Coq (which was called CoC at that time) was designed in the late 80s by Coquand and Huet. It implemented the calculus of constructions, i.e. the lambda calculus, equipped with a powerful typing discipline containing polymorphism, dependent types and type constructors. Let us mention one typing rule, conversion, which will be important to us in the rest of the paper.

$$\frac{E \vdash a : t \quad E \vdash t' : \mathrm{s}}{E \vdash a : t'} \quad \text{if } t \approx t'$$

[⋆] This work was partly supported by the Polish government grant 3 T11C 002 27 and the EU FP6 project IST-15905 Mobius.

This rule says that a proof of t is also a proof of any correct formula t' which is convertible to t. In the pure calculus of constructions, the convertibility relation $\approx$ is only β-equality, which due to normalization and confluence of β-reduction can be checked automatically. The latter property is also crucial for the decidability of type-checking.

In the calculus of constructions it is possible to define natural numbers, lists, booleans and other inductive types by using the so-called impredicative encoding. For example, natural numbers can be represented as polymorphic Church numerals with their type $Nat \equiv \forall C : \star, C \rightarrow (C \rightarrow C) \rightarrow C$. Nevertheless this coding has some important drawbacks concerning both logical and computational aspects. It is for example impossible to prove that 0 is different from 1, induction principles, for example on natural numbers, are not provable, and some trivial functions, like predecessor on natural numbers, cannot be computed in constant time.

Calculus of inductive constructions. In early 90s Coquand and Paulin proposed to extend the calculus with new kinds of syntactic objects: inductive definitions introducing a type and its constructors and elimination schemes for that type [20]. The elimination schemes come together with their reduction rules called ι-reduction, which extends convertibility. The real novelty in the calculus of inductive constructions is strong elimination, which allows one to build types by recursion over inductive types.

Using strong elimination one can show that 0 is different from 1. Other problems of impredicative encoding of inductive types are now also solved.

The calculus of inductive constructions preserves all essential metatheoretical properties of the original system: it is terminating, confluent, logically consistent and has decidable type-checking [46].

Unfortunately using recursors to write function definitions is not very easy and the definitions, once written, are not very readable. A solution to these problems is to replace the elimination schemes by two separate mechanisms: one for pattern-matching (or case-analysis), and one for constructing well-founded recursive functions.

Definitions by pattern-matching were added to Martin-Löf's type theory by Coquand [19]. Consequently, constants may be defined not only explicitly, by giving a term, but also implicitly by a set of defining equations of the form $f(u_1, \dots u_n) = e$, which must be exhaustive and unambiguous.

An adaptation of the above idea to the context of the calculus of constructions is due to Christine Paulin and was implemented in Coq in 1994. The problem of checking unambiguity and exhaustiveness is eliminated by choosing a simple format of case analysis using the `match` operator. The simple format was later extended to more complex patterns in [21]. The recursive definitions are built using the fixpoint operator `fix`. Normalization is guaranteed within the type-checking rule of `fix`. Every recursive call must operate on arguments that are structurally smaller than the original ones, i.e. are deconstructed from the original arguments by a `match`.

Although quite natural and general enough to encode many interesting functions, the `fix` rule with its guard condition causes problems, both at the practical and the theoretical levels: complicated meta-theory, non-incremental proof-checking, etc. For these reasons several versions of type annotations were proposed to hide the guardedness condition in the type system [25, 36, 26, 9, 5]. There is a running prototype implemented for the system [5].

In order to simplify writing definitions of functions, several authors consider direct translation of functional programs to Coq [35, 39]. Another possibility is to include in Coq the style of pattern-matching definitions as proposed by Coquand. But pattern-matching equations are just a restricted form of rewrite rules.

Rewriting in the calculus of constructions. There is an ongoing research process aiming at adding rewriting into theorem provers based on type theory and Curry-Howard Isomorphism such as Coq. Jean-Pierre Jouannaud, who was supervisor of our PhD theses [16, 42], is one of the people who actively pushed this research on.

Defining functions by rewriting is as simple and elegant as the definitions by pattern matching, but the user has a possibility to add more rules, that would otherwise have to be proved as axioms or lemmas in the form of Leibniz equalities or heterogeneous equalities. Moreover, rules can be ambiguous (their left-hand sides may overlap) as long as the rewriting system is confluent. By transforming equalities into rewrite rules, one makes the conversion richer and therefore proofs shorter and more automatic. Moreover more terms are now typable, so the calculus becomes more expressive.

As long as conversion is decidable, extending the conversion can be seen as a means to separate reasoning from computing in a similar manner that is used in Deduction Modulo [24]: while the proof term must record all deduction steps, the computation steps can be hidden away in conversion and performed automatically. This could be most helpful when working with axiomatic equational theories, like e.g. group theory, that can be transformed into confluent and terminating rewriting systems.

There exist tools that assist users in performing such transformations (see e.g. [17]).

In order to maintain conversion and hence type-checking decidable one must be careful to add only rewrite rules which would not spoil subject reduction, strong normalization and confluence. The easiest and the most natural way to preserve subject-reduction is to require that the left- and right-hand sides of a rule have the same type. More flexible approaches allow for the left- and right-hand sides that do not need to be well-typed (see discussion about underscore variables in the next section). Since it is not possible to check automatically whether a given set of rewrite rules in strongly normalizing, one has to come up with decidable criteria ensuring strong normalization and flexible enough to accept most of the known and useful definitions by rewriting. In [1] it is shown that the calculus of constructions can be safely extended with any terminating and confluent first-order rewriting system. For the higher-order case, there are two such criteria: HORPO [42] by the second author and the General Schema [6] by Frédéric Blanqui. They are both discussed in detail in Section 3. In the paper of Blanqui, the second problem, confluence, is also discussed.

But ensuring termination and confluence is of course not enough. To trust theorems proved with the help of a proof assistant based on a formalism one needs logical consistency. Without rewriting, consistency is guaranteed for all developments without axioms. With rewriting, one can show it for all developments where rewriting systems are complete [10, 45]. Practical way of checking completeness is also provided in [45] and discussed in Section 5.

In this paper we give a general vision of rewriting in Coq, how we think it will look and feel. We particularly care to maintain the important characteristics of Coq, such as decidable conversion and type-checking, interactive proof development, canonicity of inductive types, logical consistency etc. At the same time we aim to be able to express by rewriting the elimination schemes for inductive types and definitions by case-analysis and therefore to drop ι-reduction from the system.

2 Rewriting — Look and Feel

Let us imagine a future version of Coq with rewriting, where definitions by rewriting will be entered just as all other definitions:[1]

[1] The syntax of the definition by rewriting is inspired by the experimental "recriture" branch of Coq developed by Frédéric Blanqui.

```
Welcome to Coq 10.1

Coq < Symbol + : nat → nat → nat
Rules
| 0 + y ⟶  y
| x + 0 ⟶  x
| (S x) + y ⟶ S (x + y)
| x + (S y) ⟶ S (x + y)
| x + (y + z) ⟶ (x + y) + z.
```

The above fragment defines addition on unary natural numbers. This function is defined by induction on both arguments simultaneously and the last rule expresses associativity.

Introducing a new definition by rewriting to an environment can be done just like for inductive definitions.

$$\frac{E \vdash \mathtt{ok} \qquad E \vdash \mathsf{Rew}(\Gamma; R) : \mathsf{correct}}{E; \mathsf{Rew}(\Gamma; R) \vdash \mathtt{ok}} \tag{1}$$

If the environment E is correct and if the new definition by rewriting is correct, then we can add it to the environment. The right premise stands for all tests that have to be performed before environment extension. First, the definition must be well-formed, e.g. the local environment Γ must contain function symbols only, their types must be correct etc. Second, the rewriting system must verify the chosen acceptance condition, which should guarantee subject reduction, strong normalization and confluence of the system after adding the given definition by rewriting.

Note that at this point we only require properties that are needed to keep type-checking decidable. The user is free to add a rewriting system which causes inconsistency, just as he is free to add an inconsistent axiom but not a non-terminating fixpoint definition in the current version of Coq. Consistency of environments containing definitions by rewriting is a separate issue which is discussed in Section 5.

Rewrite rules are to be used in the conversion rule, and since the set of available rules depends on the environment, so does the conversion relation. The correct version of conversion rule is:

$$\frac{E \vdash a : t \qquad E \vdash t' : \mathrm{s} \qquad E \vdash t \approx t'}{E \vdash a : t'}$$

Going back to our example, the definition of + is well-typed, confluent and terminating, so we can assume that it is correct and safely add it to the environment. Our definition is also complete in the sense that for all pairs of canonical natural numbers (i.e. made of constructors), their sum computes into a canonical natural number.

With this definition, `+` becomes much more useful than the one usually defined using `match` and `fix`. Both lemmas $\forall x : nat.\ 0 + x = x$ and $\forall x : nat.\ x + 0 = x$ can be proved by $\lambda x : nat.\ \mathit{refl}\ nat\ x$, where *refl* is the only constructor of the Leibniz equality inductive predicate. Since the definition of addition is now symmetric we do not have to use induction for any of the two lemmas.

By enriching conversion, we also make more terms typable. Hence the logical language becomes more expressive, especially when dependently typed programs and their properties are considered.

The most prominent example of this kind is the append function on lists with length. For the sake of simplicity, let us assume that we have a list of boolean values.

```
Inductive nlist : nat → Set :=
| nnil : nlist 0
| ncons : bool → forall n:nat, nlist n → nlist (S n).

Symbol append : forall n m:nat, nlist n → nlist m → nlist (n+m).
Rules
| append 0 m nnil lm ⟶ lm
| append (S n) m (ncons b1 n ln) lm ⟶ ncons b1 (n+m) (append n m ln lm)
| append n 0 ln nnil ⟶ ln.
```

Note that without the symmetric `+` in the conversion, either the first two rules or the last rule would not be well-typed. Indeed, in the first rule, the type of the left hand side is `nlist (0+m)` and the type of the right hand side is `nlist m` and in the third rule the corresponding types are respectively `nlist (n+0)` and `nlist n`.

Thanks to the fact that associativity of `+` is in conversion, one could write and prove the following equational property of append.

```
append k (n+m) lk (append n m ln lm) = append (k+n) m (append k n lk ln) lm
```

The types of the two sides are `listn ((n+m)+k)` and `listn (n+(m+k))` respectively, and since standard Leibniz equality requires terms to compare to have convertible types, associativity of `+` must be in conversion.

Rewrite rules can also be used to define higher-order and polymorphic functions, like the `map` function on polymorphic lists.

```
Inductive list (A : Set) : Set :=
    nil : list A | cons : A → list A → list A.
```

```
Symbol map : forall A B:Set, (A → B) → list A → list B
Rules
  map A B f (nil A) ⟶ nil B
  map A B f (cons A a l) ⟶ cons B (f a) (map A B f l).
```

Even though we consider higher-order rewriting, we think that it is enough to choose the simple matching modulo α-conversion. Higher-order matching is useful for example to encode logical languages by higher-order abstract syntax, but it is not really used in Coq where modeling relies rather on inductive types.

Instead of higher-order matching, one rather needs the possibility to underspecify some arguments. Consider for example transforming the equation for associativity of `append` into a rewrite rule:

```
append k (n+m) lk (append n m ln lm)
               ⟶   append (k+n) m (append k n lk ln) lm
```

The `(n+m)` argument of `append` is needed there just for the sake of correct type-checking of the left-hand side, because the type of `append n m ln lm` is `list (n+m)`. It has no meaning for actual matching of this rewrite rule against terms, because in all well-typed terms, the second argument of append is the length of the fourth argument anyway.

Besides, putting `(n+m)` in the rewrite rule creates many critical pairs with `+`, which cannot be resolved without adding many new rules (either by hand or through an automatic completion procedure) to make the rewriting system confluent.

Instead, we could replace `(n+m)` by a fresh variable (or by an underscore standing for a "don't care" variable) and write this rule as:

```
append k _ lk (append n m ln lm)
               ⟶   append (k+n) m (append k n lk ln) lm
```

This new rule matches all well-typed terms matched by the old rule, and more. Moreover, there is no critical pairs between this rule and the rules for `+` and the rule becomes left-linear, which is both easier to match and may help with confluence proof. The downside is that the left-hand side of the rule is not well-typed anymore which might make the proof of subject reduction harder.

This way of writing left-hand sides of rules was already used by Werner in [46] to define elimination rules for inductive types, making them orthogonal (the left-hand sides are of the form $I_{elim}\ P\ \vec{f}\ \vec{w}\ (c\ \vec{x})$, where P, $\vec{f}$, $\vec{w}$, $\vec{x}$ are distinct variables and c is a constructor of I). In [10], Blanqui gives a precise account of these omissions using them to make more rewriting rules left-linear. Later, the authors of [14] show that these redundant subterms can be completely removed from terms (in a calculus without rewriting however). In [4], a new optimized convertibility test algorithm is presented for Coq, which ignores testing equality of these redundant arguments.

It is also interesting to note that when the second argument of `append` is a fresh variable then we may say that this argument is matched modulo conversion and not syntactically.

3 Strong Normalization

In this section we concentrate on the part of acceptance criterion for rewrite rules which is supposed to guarantee strong normalization.

The first article about higher-order rewriting in the calculus of constructions is [2, 3], where the authors extend the termination criterion called the General Schema, originally defined in [28]. This result is further extended in [11] by adding a powerful mechanism, called the computable closure and further on in [6, 10, 8, 7, 9] where rewriting on types, rewriting modulo AC, extended recursors and type-based termination are considered.

Another method for proving strong normalization of higher-order rewriting is the Higher Order Recursive Path Ordering (HORPO). HORPO was originally presented in [29], in the context of the simply typed lambda calculus. A version of HORPO for the calculus of constructions was first presented in [41] and its journal version [43]. An extended and elaborated version of the results can be found in [42].

The works on HORPO and the General Schema share the approach to rewriting in the calculus of constructions presented in the previous section. Rewriting is introduced by rewrite rules on function symbols that are constants added to the system. Function symbols can have dependent and polymorphic types. In [42] (HORPO approach) both sides of every rewrite rules must have the same type, in [6] (General Schema) they are meant to have the same type for any typable instance of the left-hand side.

In order to have strong normalization, the form of the rules is further restricted: it is required that all meaningful type parameters of the head function symbol of left-hand sides must be different variables.

Both termination criteria, the General Schema and HORPO, are based on a well-founded ordering on function symbols, called precedence. Inductive types are built from type constructors and constructors satisfying some positivity conditions, and elimination schemes are just function symbols with associated rewrite rules. It is shown that most elimination schemes can be accepted by the General Schema and HORPO.

In order to deal with elimination schemes, the structural ordering associated with inductive definitions is incorporated in both HORPO and General Schema. They share also the use of computable closure (first used in the context of the simply-typed λ-calculus [12]), which is a set of terms derived from the left-hand side by some syntactic reducibility-preserving operations.

While the General Schema consists essentially of the computable closure, its use in HORPO is just one of the possibilities. Nevertheless, it is not the case that all object level rules accepted by the General Schema are accepted by HORPO. Apart from some technical conditions, the reason is mainly hidden in the different approaches to inductive types and constructors; in [6], where the General Schema is used, every function symbol whose output type is an instance of an inductive type I is considered as a constructor of I, in [42], where HORPO is used, the standard vision of constructors as symbols that do not rewrite is adopted.

In both works, strong normalization is shown using the method of reducibility candidates. In [42] the proof is done not for a particular rewrite system accepted by HORPO, but for the whole HORPO itself. In other words, the calculus of constructions is extended with the rewrite relation generated by all valid HORPO judgments and it is shown that the resulting calculus is strongly normalizing. This implies strong normalization of any set of rewrite rules accepted by HORPO. In [6] the proof is done for any set of rules accepted by the General Schema.

An important characteristic of the General Schema is the possibility to define rewriting rules at the level of types and not only at the level of objects. This kind of rewriting enables to write, for example large elimination rules for inductive types.

To deal with type-level rewriting, confluence is needed. For that reason, all rules in [6] have to be left-linear (see Section 4). Extension to type-level rewriting for HORPO cannot be shown this way, since HORPO is obviously non confluent.

Practical issues. In order to be suitable for implementation, the termination criteria must have two important properties: decidability and compatibility with further extensions of environments.

Both, HORPO and General Schema, are decidable criteria for accepting rewrite rules. Putting aside the convertibility tests, checking a rewrite rule has a polynomial complexity.

Environment extension corresponds to rule 1 in Section 2 and accounts for building one rewriting system on the top of another. Originally proofs of strong normalizations for both criteria were done for the static setting: signature, precedence on function symbols, and set of rewrite rules were given in advance. But this can be adapted to the situation where a new set of rewrite rules is defined for symbols from the new signature, and type-checked with the calculus of construction extended with rewriting coming from all previous rewrite systems.

Examples. Let us end this section with some examples explaining the condition about different variables as type parameters of head function symbols of the left-hand sides and illustrating what more can be done concerning strong normalization.

The following identity rule for polymorphic `map : forall A B:Set, (A → B) → list A → list B` can be accepted neither by HORPO nor by the General Schema:

```
  map C C λx.x l ⟶ l
```

In the General Schema approach rules have to be left-algebraic (cannot contain abstractions) and this one is not. In HORPO, the condition on type parameters is not satisfied, as `map` has the same variable as the first and the second argument. Nevertheless, it is believed that it cannot break strong normalization.

The following rule for the function symbol J is not accepted either:

```
Symbol J : forall A B:Set, A → B → A
Rules
  J C C a b ⟶ b
```

But this time it can be shown that this innocent-looking rule leads to nontermination. This example derives from the one presented by J.-Y. Girard in [27] and was shown to authors by Christine Paulin (see Chapter 6 in [42] for details concerning nontermination).

An evident difference between the two rules presented above is that there exists a well-typed instance of `J C _ a b` the type of which is different from the type of the corresponding instance of `b` and that it is not possible for `map`. Nevertheless HORPO from [42] cannot deal with the rule `map C _ λx.x l ⟶ l` either since it requires both sides of the rules to have the same type.

On the other hand the identity rule for monomorphic `mmap : forall A:Set, (A → A) → list A → list A`:

```
  mmap C λx.x l ⟶ l
```

satisfies the condition about different type variables. It is accepted by HORPO and rejected by the General Schema, because it is not left-algebraic.

The next example concerns the heterogeneous equality `JMeq`.

```
Inductive JMeq (A:Set)(a:A) : forall B:Set, B → Set :=
    JMeq_refl : JMeq A a A a.
```

The standard elimination scheme for this rule (`JMeq_std`) does not satisfy the condition about different type parameters and hence is rejected by HORPO. But it is known to be terminating already from the works on CIC (see [46] and also [7]).

```
Symbol JMeq_std : forall (A:Set)(a:A)(P : forall B:Set, B → Set),
  P A a → forall (B:Set)(b:B), JMeq A a B b → P B b
Rules
  JMeq_std A a P h A a (JMeq_refl A a) ⟶ h
```

However, its nonstandard (and more useful) elimination scheme `JMeq_nstd` satisfies the condition on type parameters and can be shown terminating by both HORPO and the General Schema.

```
Symbol JMeq_nstd : forall (A:Set)(a:A)(P:A → Set),
  P a → forall (b:A), JMeq A a A b → P b
Rules
  JMeq_nstd A a P h a (JMeq_refl A a) ⟶ h
```

4 Confluence

Confluence of the calculus of constructions with rewriting is less studied than strong normalization and it is known for two kinds of situations. First, if strong normalization can be established without confluence (it is the case for the object-level rewriting, see [6, 42]), then confluence is a consequence of the strong normalization and local confluence, i.e. joinability of critical pairs. If confluence is needed before the strong normalization proof then the result of [33] can be used. It states that the sum of beta reduction with confluent left-linear and left-algebraic rewrite system R is confluent. Confluence of R (without beta reduction) is usually simpler to achieve; in [6] it is shown for every R being a combination of a first-order system that is strongly normalizing and nonduplicating with a set of first- and higher-order rules satisfying the General Schema and such that critical pairs of R are joinable.

Left-linearity may seem an important restriction when using dependent types, but it is not really the case. In fact, nonlinearities due to typing can be avoided using underscore variables, like described in Section 2. Nevertheless, if one aims to deal with type-level rewriting, both type- and object-level rules have to be left-linear and left-algebraic. In order to lift this restriction one should probably consider a simultaneous proof of strong normalization and confluence.

5 Logical Consistency, Completeness of Definitions and Inductive Consequences

Adding arbitrary rewrite rules to the calculus of constructions may easily lead to logical inconsistency, just like adding arbitrary axioms. It is of course possible to put the responsibility on the user, but it is contrary to the current Coq policy to guarantee consistency of a large class of developments, namely those which do not contain axioms. Since we plan on using rewriting as a principal means of defining functions, we have to come up with a large decidable class of rewriting systems that are guaranteed not to violate consistency.

Logical consistency for the calculus of constructions with rewriting was first studied in [10]. It was shown under an assumption that for every symbol f defined by rewriting, $f(t_1, \ldots, t_n)$ is reducible if $t_1 \ldots t_n$ are terms in normal form in the environment consisting of one type variable. But there were no details how to satisfy the assumption of the consistency lemma.

In [45] it is shown that logical consistency is an easy consequence of canonicity, which can be proved from completeness of definitions by rewriting (discussed below), provided that termination and confluence are proved first. More precisely, it can be shown that in every environment consisting only of inductive definitions and complete definitions by rewriting, every term of an inductive type can be reduced to a canonical form. This, by an easy analysis of normal forms, implies that there is no proof of $\Pi x{:}{*}.x$.

Completeness of definitions by rewriting. Informally, a definition by rewriting of a function symbol f is *complete* if the goal $f(x_1, \ldots, x_n)$ is covered, which means that all its canonical instances are head-reducible. In [45] the definition of completeness is precised in such a way that it guarantees logical consistency and there exist a sound and terminating algorithm for checking completeness of definitions.

If we adopt the view that properties of a rewriting system should be checked when it is being introduced to an environment (see typing rule 1 in Section 2), then completeness of the function symbol f has to be checked much earlier than it is used: one uses it in an environment $E = E_1; \mathsf{Rew}(f, R); E_2$ but it has to be checked when f is added to the environment, i.e. in the environment E_1. It follows that completeness checking has to account for environment extension and can be performed only with respect to arguments of such types which guarantee that their set of normal inhabitants would not change in the future. This is the case for inductive types whose normal inhabitants are always terms built from constructors.

In [45] there is also an algorithm for checking completeness. It checks that a goal is covered using successive splitting, i.e. replacement of variables of inductive types by constructor patterns. In presence of dependent types not all constructors can be put in every place. The `head` function below is completely defined since `nnil` is not of type `nlist (S n)`.

```
Symbol head : forall n:nat, nlist (S n) → bool
Rules
   head n (ncons b n l) ⟶ b
```

The algorithm is necessarily incomplete, since in the presence of dependent types emptiness of types trivially reduces to completeness and the former is undecidable. The algorithm accepts all definitions that follow dependent pattern matching schemes presented by Coquand and studied by McBride in his PhD thesis. Extended with the second run, it deals with all usual definitions by case analysis in Coq. It also accepts many definitions by rewriting containing rules which depart from standard pattern matching.

The rewriting systems for `+`, `append`, `map`, `JMeq_std`, `JMeq_nstd` presented earlier can be easily proved complete by the algorithm. This is also true for Streicher's axiom K:

```
Symbol K : forall (A:Set) (a:A) (P:eq A a a → Set),
   P (refl A a) → forall p: eq A a a, P p
Rules
   K A a P h (refl A a) ⟶ h
```

Another method for checking completeness of pattern matching equations in the Calculus of Constructions is presented in [34]. It consists in computing approximations of inductive types and is not based on splitting; for that reason it accepts some of the examples not accepted by the algorithm described above. Fortunately, it seems that the approximation method can be easily added to the algorithm from [45] as another phase, if the original version fails to show completeness.

Inductive consequences. During the completeness check of a definition by rewriting only some of the rules are used; usually they correspond to the pattern matching definition of a given symbol. An interesting question is how much the rules outside this part extend the conversion.

For first-order rewriting it is known that these rules are inductive consequences of the pattern matching ones, i.e. all their canonical instances are satisfied as equalities (see e.g. Theorem 7.6.5 in [40]). It is also true for higher-order and dependent rewriting in the calculus of constructions as long as there is no rewriting under a binder in the rewrite steps needed to join the critical pairs [44]. For example, the identity rule for the monomorphic `mmap` function from Section 3 is clearly an inductive consequence of the basic rules for `nil` and `cons`.

Unfortunately, the problem is more difficult for higher-order rules over inductive types with functional arguments. The defined function symbol might get under a binder and might be applied to a bound variable instead of a canonical term on which it is always reducible. Here is an example:

```
Inductive ord : Set :=
  o : ord
| s : ord → ord
| lim : (nat → ord) → ord.

Rewriting n2o : nat → ord
Rules
  n2o O ⟶ o
  n2o (S x) ⟶ s (n2o x)

Rewriting id : ord → ord
Rules
  id o ⟶ o
  id (s x) ⟶ s (id x)
  id (lim f) ⟶ lim (fun n ⇒ id (f n))

  id (id x) ⟶ id x
```

The last rewriting system is confluent (unlike the one in which the last rule is replaced with `id x ⟶ x` because the critical pair for `x=(lim f)` needs eta to be joinable). Now, for $l =$ `id (id x)`, $r =$ `id x`, $\sigma = \{$`x` $\mapsto$ `lim (fun n ⇒ n2o n)`$\}$ one has:

```
lσ = id (id (lim (fun n ⇒ n2o n)))
  ⟶  id (lim (fun n' ⇒ id ((fun n ⇒ n2o n) n')))
  ⟶  id (lim (fun n' ⇒ id (n2o n')))
  ⟶  lim (fun n'' ⇒ id ((fun n' ⇒ id (n2o n')) n''))
  ⟶  lim (fun n'' ⇒ id (id (n2o n'')))

rσ = id (lim (fun n ⇒ n2o n))
  ⟶  lim (fun n' ⇒ id ((fun n ⇒ n2o n) n'))
  ⟶  lim (fun n' ⇒ id (n2o n'))
```

and they are not equal.

It seems that in case when critical pairs are joinable using rewriting under a binder, rules that are outside definitional part can also be considered as inductive consequences of the definition, but then the conversion needs to be functionally extensional (similarly to [34]) and some special care should be payed to the contexts under which rewriting occurs.

Summarizing, in many rewriting systems, especially simple ones, defining functions over non-functional inductive types, additional rules are inductive consequences of the complete subsystem. In other rewriting systems, even if we are not sure that this is the case, by [10, 45], a terminating, confluent and complete rewriting system cannot lead to inconsistency.

In practice it would be best to have two different keywords for complete and "not necessarily complete" definitions by rewriting. For example, the keyword `Complete Symbol` would mean that the set of rules must be checked for completeness by Coq and rejected if its completeness cannot be proved. The keyword `Symbol` (without `Complete`) would not incur any completeness checking and the user would understand that the responsibility for consistency in entirely in his hands.

6 Modularity

A very important issue in an interactive system like Coq is modularity. Coq developments are usually composed of many files and use the standard library or some libraries developed by third parties.

Once the given library file is checked, Coq metatheory guarantees that the compiled library can be read and included in any development without risking undecidability or logical inconsistency (the latter provided that there are no axioms in the library or the development).

In order to retain this status once rewriting is added to Coq one must be very careful to ensure good modularity properties on the rewriting systems included in library files.

In particular it should be impossible to define a rewrite rule $f(x) \longrightarrow g(x)$ in one file and $g(x) \longrightarrow f(x)$ in another one, because loading these two files together would break strong normalization. Although it is possible to design a new version of Coq in such a way that the whole set of rewriting rules is rechecked every time a new library is loaded, it would be very time consuming and therefore it is not a good solution.

Instead, the acceptance criteria for rewrite rules should take modularity into consideration. In other words, the following lemma should hold even if there are some definitions by rewriting in E_1, E_2 and/or $\mathcal{J}$:

For all judgments $\mathcal{J}$, if $E; E_1 \vdash \mathcal{J}$ and $E; E_2 \vdash \texttt{ok}$ then $E; E_1; E_2 \vdash \mathcal{J}$.

In practice, many modularity problems are avoided by allowing only rewrite definitions $\mathsf{Rew}(\Gamma, R)$ whose head symbols of the left hand sides of rules come from Γ. It is the case in all examples given in Section 2.

The module system. Additional modularity requirements for definitions by rewriting come from the module system. The Coq module system [15, 16] was designed with adding rewriting in mind. In particular, even though more theoretically advanced module systems existed at the time it was implemented (first-class modules, anonymous modules [38, 22, 23]), a simple named version was chosen, similar to [30], where each module construct must be given a name before being used in terms. Thanks to that, existing syntactic acceptance criteria (see Section 3) can be easily adapted to modules.

The module language resembles a simply typed lambda calculus with records and record types. A very important feature is module subtyping, with its subsumption rule permitting to give a less precise type to a module.

$$\frac{E \vdash M : T \qquad E \vdash T <: T'}{E \vdash M : T'}$$

This rule is useful in two cases. First, it permits to hide some implementation details of a module in order to be able to change them in the future without affecting other parts of the project. Second, it permits to define a functor with minimal requirements to its arguments and then apply it to a module at hand with more elements and more precise interface.

Once rewriting is added to Coq, definitions by rewriting will be allowed in module interfaces and in particular in argument types of functors.

Since functors can be applied to all modules whose interface is a subtype of the functor argument type, the subtyping on module interfaces has to be extended to interfaces containing definitions by rewriting.

It is clear that the convertibility properties required by the functor argument interface must be satisfied by the actual parameter's interface. Otherwise the functor result would not be well-typed.

In [16] no other restrictions on "rewriting-subtyping" are imposed. Note however, that such liberal definition of subtyping implies very strict modularity properties for the acceptance condition for definitions by rewriting. This means that the definition by rewriting must be guaranteed not only to be terminating and confluent in the current environment, but also in an environment, where some module type is replaced by a subtype. For example consider the following module type and functor:

```
Module Type T.
  Symbol g : bool → bool
  Rules
    g true → true.
End T.

Module F(X:T).
  Symbol f : bool → nat.
  Rules
    f true ⟶ 0
    f false ⟶ 0
    f (X.g false) ⟶ 1.
End F.
```

The definition of `f` inside the functor `F` is confluent, because no reduction rules are associated with `g false` and hence there are no critical pairs. Consider a possible implementation `M` of the module type `T` and the application of `F` to `M`.

```
Module M <: T.
  Symbol g : bool → bool
  Rules
    g true ⟶ true
    g false ⟶ false.
End M.

Module Z := F M.
```

Now, the signature of `Z` is the same as the body of `F`, but the formal parameter `X` is replaced by the actual parameter `M`. This gives the following set of rules defining `Z.f`:

```
f true ⟶ 0
f false ⟶ 0
f (M.g false) ⟶ 1
```

which is non-confluent, because on one hand `f (M.g false)` $\longrightarrow$ `1` and on the other hand `f (M.g false)` $\longrightarrow$ `f false` $\longrightarrow$ `0`.

It turns out that in presence of modules and subtyping, rewrite rules where left-hand sides mentions external symbols whose specification may still be completed is very dangerous. The easiest way to prevent this danger is to restrict the left-hand sides of the rewrite rules to contain only the symbols declared by the given rewrite definition. This is, again, the case in all examples from Section 2.

Such restriction however turns out to be quite severe. Consider for example defining mathematical functions over natural numbers. It might be a good idea to add a rule defining how this function behaves for the arguments of the form `(a+b)`, but since `+` is not defined at the same time, this is impossible. Since `+` is already completely defined it is unlikely to introduce a new definition of `+` which would create more critical pairs.

This question, whether it is safe to allow external but completely defined symbols in left hand sides of rewrite rules, definitely needs to be studied further.

7 Extraction

The possibility to extract executable Ocaml, Haskell or Scheme code from Coq developments is one of the key features of Coq [37, 31, 32]. In this section we try to analyze the impact of introducing rewriting to Coq on the extraction mechanism.

The general problem with rewriting is that it does not immediately correspond to any mechanism present in functional languages. To extract a definition of a symbol defined by rewriting it is important to check whether this symbol is completely defined or not. If it is not, this means that the symbol is like an axiom and its successful extraction is impossible.

If it is complete then, as we explained in Section 5, its definition can be divided into two parts. The first part, which is a complete subset containing the rules used by the completeness checking procedure, usually consists of pattern matching rules. The second part is the remaining set of rules which, in most cases, are inductive consequences of the first part. These rules can also be called *shortcut* rules and they are only really important if the function's arguments are not ground terms, which is never the case when a functional program is executed.

So it is enough to just translate those rules which have the pattern-matching form and simply leave out all the others. The resulting definition will include the complete definition and some of the shortcut rules which have a pattern-matching form. Other shortcut rules should simply be dropped or they can be transformed into rewrite rules for optimization, available e.g. in the Glasgow Haskell Compiler.

For example the definition of + from Section 2 could be extracted to the following Ocaml code:

```
(** val plus : nat → nat → nat **)

let rec plus n m =
  match n, m with
  | O, y → y
  | x, O → x
  | S x, y → S (plus x y)
  | x, S y → S (plus x y);;
```

The second and fourth lines are not necessary for the completeness of the translation. However, the second line can speed up the definition if the second argument is O. Unfortunately, the fourth line is never used, even though it could also speed the computation up. Indeed, any natural number which does not match the first and the second rule, does match the third one. The associativity rule for + is not extracted, as it is not in the pattern-matching form.

The definition of append is extracted in the following way:

```
(** val append : nat → nat → nlist → nlist → nlist **)

let rec append n m ln lm =
  match n, m, ln, lm with
    | O, m, Nnil, lm → lm
    | (S n0), m, Ncons (b, n0', n1), lm →
        Ncons (b, (plus n0 m), (append n0 m n1 lm))
    | n, O, ln, Nnil → ln;;
```

A smart extraction procedure should also change the order of the second and third rule, because otherwise one of the first two rules always applies and the expected speed up from the third rule can never be achieved. Note also, that an extracted function has the same number of arguments as the original one, but the dependencies between them are broken. Consequently, while append is complete in the Coq world, its extracted version contains non-exhaustive pattern-matching, corresponding to unexpected cases when the list and its declared length do not match. Note however that the same problem it is already present for the extracted version of definitions by `fix` and `match` in the current Coq.

8 Conclusions

The goal of this paper was to present our vision of rewriting in Coq and to summarize the results already known in the field. We started from an incremental character of the calculus of constructions with rewriting, the form of the definitions by rewriting and matching used to apply rewrite rules to terms. Then we discussed such issues as strong normalization, confluence, logical consistency, completeness of definitions by rewriting, modularity and extraction.

We hope that this paper can be a basis for a deeper/more detailed discussion about rewriting in Coq, its future and its alternative views. Moreover we hope that it can serve for comparisons with conceptually different extensions of Coq, for example the one described in [13] aiming at extending Coq with decision procedures.

Acknowledgements. We would like to thank Paweł Urzyczyn, Frédéric Blanqui and the anonymous referees for their comments on the first version of our paper.

References

1. Franco Barbanera. Adding algebraic rewriting to the calculus of constructions: Strong normalization preserved. In *Proceedings of the Second International Workshop on Conditional and Typed Rewriting*, 1990.
2. Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization and confluence in the λ-algebraic-cube. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 406–415, Paris, France, July 1994. IEEE Comp. Soc. Press.
3. Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-λ-cube. *Journal of Functional Programming*, 7(6):613–660, 1997.
4. Bruno Barras and Benjamin Grégoire. On the role of type decorations in the calculus of inductive constructions. In L. Ong, editor, *Proceedings of the 19th Annual Conference of the European Association for Computer Science Logic*, volume 3634 of *Lecture Notes in Computer Science*, pages 151–166, Oxford, UK, 2005. Springer.
5. Gilles Barthe, Benjamin Grégoire, and Fernando Pastawski. CIC$\hat{}$: Type-based termination of recursive definitions in the Calculus of Inductive Constructions. In Miki Hermann and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence and Reasoning, 13th International Conference, Phnom Penh, Cambodia, November 13-17, 2006, Proceedings*, volume 4246 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2006.

6. Frédéric Blanqui. *Théorie des Types et Récriture.* PhD thesis, Université Paris-Sud, 2001.
7. Frédéric Blanqui. Inductive types in the Calculus of Algebraic Constructions. In M. Hofmann, editor, *Proceedings of 6th International Conference on Typed Lambda Calculi and Applications*, volume 2701 of *Lecture Notes in Computer Science*, Valencia, Spain, 2003.
8. Frédéric Blanqui. Rewriting modulo in Deduction modulo. In R. Nieuwenhuis, editor, *14th International Conference on Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, Valencia, Spain, 2003. Springer-Verlag.
9. Frédéric Blanqui. A Type-Based Termination Criterion for Dependently-Typed Higher-Order Rewrite Systems. In V. van Oostrom, editor, *Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2004.
10. Frédéric Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005.
11. Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. The Calculus of Algebraic Constructions. In P. Narendran and M. Rusinowitch, editors, *10th International Conference on Rewriting Techniques and Applications*, volume 1631 of *Lecture Notes in Computer Science*, Trento, Italy, July 1999. Springer.
12. Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive data type systems. *Theoretical Computer Science*, 272(1–2):41–68, 2002.
13. Frédéric Blanqui, Jean-Pierre Jouannaud, and Pierre-Yves Strub. Building decision procedures in the calculus of inductive constructions. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007*, volume 4646 of *Lecture Notes in Computer Science*, pages 328–342. Springer, 2007.
14. Edwin Brady, Connor McBride, and James McKinna. Inductive families need not store their indices. In S. Berardi, M. Coppo, and F. Damiani, editors, *Types for Proofs and Programs, TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2004.
15. Jacek Chrząszcz. Implementation of modules in the Coq system. In D. Basin and B. Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of *Lecture Notes in Computer Science*, pages 270–286, Rome, Italy, September 2003. Springer.
16. Jacek Chrząszcz. *Modules in Type Theory with Generative Definitions.* PhD thesis, Warsaw Univerity and University of Paris-Sud, Jan 2004.
17. Evelyne Contejean, Claude Marché, Benjamin Monate, and Xavier Urbain. CiME version 2, 2000. Available at `http://cime.lri.fr/`.
18. The Coq proof assistant. `http://coq.inria.fr/`.
19. Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.
20. Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
21. Cristina Cornes. *Conception d'un langage de haut niveau de réprésentation de preuves.* PhD thesis, Université Paris VII, 1997.
22. Judicaël Courant. A Module Calculus for Pure Type Systems. In *Typed Lambda Calculi and Applications 97*, volume 1210 of *Lecture Notes in Computer Science*, pages 112–128. Springer-Verlag, 1997.

23. Judicaël Courant. *Un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.
24. Gilles Dowek, Thérèse Hardin, and Claude Kirchner. Theorem proving modulo. *Journal of Automated Reasoning*, 31(1):33–72, 2003.
25. Eduardo Giménez. *Un Calcul de Constructions Infinies et son Application à la Vérification des Systèmes Communicants.* PhD thesis, Ecole Normale Supérieure de Lyon, 1996.
26. Eduardo Giménez. Structural recursive definitions in type theory. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 397–408, Aalborg, Denmark, July 1998. Springer.
27. Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la téorie des types. In J.E. Fenstad, editor, *Proceedings of the 2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, 1971.
28. Jean-Pierre Jouannaud and Mitsuhiro Okada. Executable higher-order algebraic specification languages. In *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 350–361. IEEE Comp. Soc. Press, 1991.
29. Jean-Pierre Jouannaud and Albert Rubio. The higher-order recursive path ordering. In Giuseppe Longo, editor, *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, Trento, Italy, July 1999. IEEE Comp. Soc. Press.
30. Xavier Leroy. Manifest types, modules, and separate compilation. In *Conference Record of the 21st Symposium on Principles of Programming Languages*, pages 109–122, Portland, Oregon, 1994. ACM Press.
31. Pierre Letouzey. A New Extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science.* Springer, 2003.
32. Pierre Letouzey. *Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq.* PhD thesis, Université Paris-Sud, 2004.
33. F. Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41(6):293–299, 1992.
34. Nicolas Oury. *Égalité et filtrage avec types dépendants dans le Calcul des Constructions Inductives.* PhD thesis, Université Paris-Sud, 2006.
35. Catherine Parent. Developing certified programs in the system Coq - the Program tactic. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs, TYPES'93*, volume 806 of *Lecture Notes in Computer Science*, pages 291–312. Springer, 1994.
36. Christine Paulin-Mohring. *Définitions inductives en théorie des types d'ordre supérieur.* Thèse d'habilitation, Ecole Normale Supérieure de Lyon, 1996.
37. Christine Paulin-Mohring and Benjamin Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.
38. Robert Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13:386–402, 2002.
39. Matthieu Sozeau. Subset coercions in coq. In *Types for Proofs and Programs, TYPES 2006*, Lecture Notes in Computer Science. Springer, 2007. To appear.
40. Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 2003.
41. Daria Walukiewicz-Chrząszcz. Termination of rewriting in the calculus of constructions. In J. Despeyroux, editor, *Proceedings of the 2nd Workshop on Logical Frameworks and Meta-Languages*, Santa Barbara, California, 2000.

42. Daria Walukiewicz-Chrząszcz. *Termination of Rewriting in the Calculus of Constructions.* PhD thesis, Warsaw University and University Paris XI, 2003.
43. Daria Walukiewicz-Chrząszcz. Termination of rewriting in the calculus of constructions. *Journal of Functional Programming*, 13(2):339–414, 2003.
44. Daria Walukiewicz-Chrząszcz and Jacek Chrząszcz. Inductive consequences in the calculus of constructions. Draft available at `http://www.mimuw.edu.pl/~chrzaszcz/papers/`.
45. Daria Walukiewicz-Chrząszcz and Jacek Chrząszcz. Consistency and completeness of rewriting in the calculus of constructions. In *Automated Reasoning, Third International Joint Conference, IJCAR 2006, Proceedings*, volume 4130 of *Lecture Notes in Artificial Intelligence*, pages 619–631. Springer, 2006.
46. Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives.* PhD thesis, Université Paris VII, 1994.