Modules in Coq Are and Will Be Correct

Jacek Chrząszcz

Institute of Informatics, Warsaw University
ul. Banacha 2, 02-097 Warszawa, Poland
`chrzaszcz@mimuw.edu.pl`

Abstract. The paper presents the system of named modules implemented in Coq version 7.4 and shows that this extension is conservative. It is also shown that the implemented module system is ready for the future planned extension of Coq with definitions of functions by means of rewrite rules. More precisely, the impact of the module system on the acceptance criterion for rewrite rules is carefully studied, leading to the formulation of four closure properties that have to be satisfied by the acceptance criterion in order to validate the conservativity proof. It turns out that syntactic termination criteria such as Higher Order Recursive Path Ordering or the General Schema can be adapted to satisfy these closure properties.

1 Introduction

Computer aided theorem proving has become an important part of modern theoretical computer science. Various proof assistants gain more and more popularity and industrial size problems begin to be addressed. To become applicable in the industry, such systems must provide ways to structure large developments and a high degree of automation.

In this paper, we concentrate on the Coq proof assistant [5], a system that is based on the so-called Curry-Howard isomorphism, relating logical formulas to types and their proofs to terms inhabiting these types. The logical formalism implemented in Coq is an extension of the calculus of constructions of Coquand and Huet [7] with inductive types [8] and a predicative hierarchy of universes [14]. Coq is actively developed for more than 10 years now and is used both for fomalizing mathematics (for example the fundamental theorem of algebra) and for program verification (data structures, telecommunication protocols, Javacard platform etc.)

In order to address the large development issues, proof developments in Coq may be divided into files and an ML-style module system has recently been implemented [3]. The latter allows clear modelisation of parametrized theories and certified data structures and their convenient instantiation, therefore encouraging proof reuse.

The automation issues are being addressed by various tactics, but the main show-stopper is the treatment of equations. Even though equational reasoning is so common in mathematics it is still quite difficult to use it in Coq, especially compared to its competitors such as Isabelle [16] or PVS [15]. For that reason the inclusion of an efficient term rewriting engine is also planned for Coq.

In order to make rewriting as efficient as possible it is desirable to include it in the internal conversion relation of Coq. In this way, arbitrarily long reduction sequences would result in very small proof-terms. The other possibility of using rewriting based on the Leibniz equality is much more space- and time-consuming, as each rewriting step generates a fragment of the proof-term proportional to the size of the original goal.

Since the logical correctness of Coq depends on good metatheoretical properties of conversion, like confluence and strong normalization of reduction rules, we have to preserve these properties when extending conversion with user-defined rewrite rules. Since these properties of term rewriting systems are undecidable in general, the research in this area concentrates on incomplete acceptance criteria that guarantee termination and confluence and are flexible enough to accept most rewriting systems already known to be terminating and confluent [1,18].

The goal of this paper is to prove that the modular extension of Coq is conservative and that it will remain so after rewriting is added to Coq. We study the impact of the module system on acceptance criteria for rewrite rules and formulate closure properties that have to be satisfied by the criteria in order to make the conservativity proof work. The closure properties turn out to be compatible with syntactic acceptance criteria available in the literature.

In the type theory setting ML-style modules were first introduced in 1992 in the LF logical framework [12]. Later Courant [9,10] proposed a module calculus suitable for pure type systems, an example of which is the calculus of constructions and many other known type theories. In the calculus of Courant modules are second-class anonymous objects, reductions on modules have subject reduction and termination properties and therefore the modular extension of a given pure type system is conservative.[1] Thanks to the consequent use of module interfaces, Courant's module calculus allows smooth composition of libraries, separate checking of dependent parts of large proof developments and guarantees correctness of proofs upon a conservative upgrade of their components.

Unfortunately, given currently available acceptance criteria, definitions by rewriting cannot be integrated in the calculus of anonymous modules. Indeed, reductions on modules would impose very strong closure properties on acceptance criteria, such as closure by arbitrary substitutions, which does not hold for any of the available acceptance criteria.

The module system implemented in Coq is therefore a syntactic restriction of the calculus of Courant, allowing to use existing acceptance criteria for rewrite rules, without sacrificing good properties of the module system mentioned above. The proof of conservativity is changed accordingly: instead of relying on arbitrary reductions on anonymous modules we chose a particular reduction strategy preserving syntactic constraints, trying to make closure properties imposed on acceptance criteria as weak as possible.

[1] Courant also writes that his proof would still be valid if anonymous inductive types were added to the pure type system.

We start our presentation by formalizing pure type systems with generative definitions, a compromise between anonymous inductive types and global rewriting systems. On one hand, in theoretical papers inductive types are usually treated as anonymous entities [8,20], even though in Coq they are implemented as named. On the other hand, rewriting is usually given in a form of one global rewriting system which is treated as a parameter of typing rules. This approach is of course incompatible with interactive proof assistant, where users are used to construct their proofs and definitions gradually. However, known syntactic acceptance criteria are modular and allow to extend the global set of rewrite rules by entering one accepted rewriting system after another.

In our formalism, definitions by rewriting and inductive definitions are entered in the global environment, and they can be accessed in terms through the names assigned to them by the environment.

Next, this formalism is extended with a system of named modules and the generative definitions now become part of modules and module interfaces. We show, that a potential proof of `False` in the calculus with modules can be transformed into a proof of `False` in the calculus without modules. The closure properties on acceptance criteria for definitions by rewriting result from the analysis of this transformation. We conclude the paper by an argument that these properties are satisfiable.

2 Pure Type Systems with Generative Definitions

We present here a formalization of a pure type system with generative inductive definitions and generative definitions by rewriting, which is quite close to the way these elements are and will be implemented in Coq. The formalism is built upon a set of PTS sorts $\mathcal{S}$, a binary relation $\mathcal{A}$ and a ternary relation $\mathcal{R}$ over $\mathcal{S}$ governing the typing rules **(Term/Ax)** and **(Term/Prod)** respectively (Fig. 4). The syntactic class of pseudoterms is defined as follows:

$$e, t ::= v \mid \mathrm{s} \mid (e_1\ e_2) \mid \lambda v{:}t.\, e \mid \Pi v{:}t_1.\, t_2$$

The difference between e and t is only intuitive, to help the reader distinguish between a role of a term e and a type t, but both these letters denote elements of the syntactic class of pseudoterms. A pseudoterm can be a variable, a sort from $\mathcal{S}$, an application, an abstraction or a product.

Inductive definitions and definitions by rewriting are stored in the environment and used in terms only through names assigned to them by the environment. Therefore an environment is a sequence of declarations, each of them being a constant definition $v : t := e$, a variable declaration $v : t$, an inductive definition $\mathsf{IndDef}(E^I := E^C)$, where E^C and E^I are environments of (possibly mutually defined) inductive types and their constructors, or a definition by rewriting $\mathsf{RewDef}(E^\Gamma, R)$, where E^Γ is an environment of (possibly mutually defined) function symbols and R is a set of rewrite rules defining them. Environments E^I, E^C in inductive definitions and E^Γ in definitions by rewriting contain only variable declarations. We assume that names of all declarations in environments are pairwise disjoint.

Definition 1. *A* pure type system with generative definitions *is defined by the typing rules in Fig. 1, 2, 3 and 4. The relation* $\approx$ *used in the rule* **(Term/Conv)** *is the congruence generated by the sum of beta, delta and rewrite reductions.*

As in [2,19] elimination of inductive types is supposed to be expressed by rewriting.

The rules for correctness of definitions contain side-conditions. The side conditions $\mathsf{POS}_E(E^I := E^C)$ stands for a positivity condition on inductive definitions as given for example in [8,20]. The condition $\mathsf{ACC}_E(E^\Gamma, R)$ stands for an acceptance condition as given for example in [2,19]. Both conditions are meant to assure the decidability of type-checking in any correct environment.

Consider the following sequence of declarations in the Coq-like syntax:[2]

```
Inductive  nat : Set  :=  O : nat | S : nat → nat.
Symbol plus : nat → nat → nat
  Rules
    plus O y    ⟶  y
    plus (S x) y ⟶ S (plus x y)
    plus x (plus y z) ⟶ plus (plus x y) z
    plus x O    ⟶  x
    plus x (S y) ⟶ S (plus x y).
Definition two : nat := plus (S O) (S O).
Parameter n : nat.
```

It can be interpreted as an environment E consisting of the inductive definition of natural numbers, symmetric definition by rewriting of addition, the definition of the constant two and the declaration of a variable n of type *nat*. Assuming that the definition of natural numbers satisfies the condition $\mathsf{POS}()$ and the definition of addition satisfies the condition $\mathsf{ACC}()$, we can derive the judgment $E \vdash \texttt{ok}$ using the rules in Fig. 1.

The definition of logical consistency for a calculus with generative definitions has to be more involved that the usual requirement that `False` is not inhabited in the empty environment. Indeed, the latter formulation does not account for generative definitions at all. Therefore we define logical consistency by the requirement that `False` is not inhabited in any *closed* environment, i.e. an environment without variable declarations and where all functions are total.

The requirement that functions defined by rewriting are total could very well be included in the condition $\mathsf{ACC}()$. However, we decided to assume the existence of a separate condition on definitions by rewriting, called $\mathsf{COMP}_E(E^\Gamma, R)$ for *completeness*, that is satisfied only if all functions from E^Γ are total, i.e. completely defined by R in the environment E.

The separation between $\mathsf{ACC}()$ and $\mathsf{COMP}()$ is motivated by the idea of working with abstract function symbols, equipped with some rewrite rules not defining them completely. For example if *plus* were declared using only the third rule from the system give above, we could develop a theory of an associative

[2] The syntax of the definition by rewriting is inspired by the experimental "Rewriting" branch of Coq developed by Blanqui. For the sake of clarity we omit certain details, like the environment of rule variables.

$$\frac{}{\epsilon \vdash \mathsf{ok}}$$

$$\frac{E \vdash \mathsf{ok} \quad E \vdash t : \mathrm{s}}{E; v : t \vdash \mathsf{ok}} \qquad \frac{E \vdash \mathsf{ok} \quad E \vdash t : \mathrm{s} \quad E \vdash e : t}{E; v : t := e \vdash \mathsf{ok}}$$

$$\frac{E \vdash \mathsf{ok} \quad E \vdash \mathsf{IndDef}(E^I := E^C) : \mathsf{correct}}{E; \mathsf{IndDef}(E^I := E^C) \vdash \mathsf{ok}} \qquad \frac{E \vdash \mathsf{ok} \quad E \vdash \mathsf{RewDef}(E^\Gamma, R) : \mathsf{correct}}{E; \mathsf{RewDef}(E^\Gamma, R) \vdash \mathsf{ok}}$$

Fig. 1. Environment correctness

Let $E^I = v_1^I : t_1^I \dots v_k^I : t_k^I$ and $E^C = v_1^C : t_1^C \dots v_n^C : t_n^C$

$$\frac{\begin{array}{c} E \vdash t_j^I : \mathrm{s}_j \text{ for } j = 1 \dots k \\ E; E^I \vdash t_i^C : \mathrm{s}_i' \text{ for } i = 1 \dots n \end{array}}{E \vdash \mathsf{IndDef}(E^I := E^C) : \mathsf{correct}} \quad \text{if } \mathsf{POS}_E(E^I := E^C)$$

Let $E^\Gamma = v_1 : t_1 \dots v_n : t_n$ and
$R = \{E_i : e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\dots m}$, where $E_i = v_1^i : t_1^i; \dots; v_{n_i}^i : t_{n_i}^i$

$$\frac{\begin{array}{c} E \vdash t_k : \mathrm{s}_k \quad \text{for } k = 1 \dots n \\ \left.\begin{array}{cc} E; E_i \vdash \mathsf{ok} & E; E^\Gamma; E_i \vdash \hat{t}_i : \hat{s}_i \\ E; E^\Gamma; E_i \vdash e_i^L : \hat{t}_i & E; E^\Gamma; E_i \vdash e_i^R : \hat{t}_i \end{array}\right\} \text{for } i = 1 \dots m \end{array}}{E \vdash \mathsf{RewDef}(E^\Gamma, R) : \mathsf{correct}} \quad \text{if } \mathsf{ACC}_E(E^\Gamma, R)$$

Fig. 2. Correctness of definitions

$$\frac{E_1; v : t; E_2 \vdash \mathsf{ok}}{E_1; v : t; E_2 \vdash v : t} \qquad \frac{E_1; v : t := e; E_2 \vdash \mathsf{ok}}{E_1; v : t := e; E_2 \vdash v : t} \qquad \frac{E_1; v : t := e; E_2 \vdash \mathsf{ok}}{E_1; v : t := e; E_2 \vdash v \longrightarrow_\delta e}$$

$$\frac{E \vdash \mathsf{ok}}{E \vdash v_i^I : t_i^I} \qquad \frac{E \vdash \mathsf{ok}}{E \vdash v_i^C : t_i^C} \qquad \text{where } \begin{cases} E = E_1; \mathsf{IndDef}(E^I := E^C); E_2 \\ E^I = v_1^I : t_1^I \dots v_n^I : t_n^I \\ E^C = v_1^C : t_1^C \dots v_n^C : t_n^C \end{cases}$$

$$\frac{E \vdash \mathsf{ok}}{E \vdash v_i : t_i} \qquad \frac{E \vdash \mathsf{ok} \quad \delta : E_i \to E}{E \vdash e_i^L \delta \longrightarrow_R e_i^R \delta} \qquad \text{where } \begin{cases} E = E_1; \mathsf{RewDef}(E^\Gamma, R); E_2 \\ E^\Gamma = v_1 : t_1 \dots v_n : t_n \\ R = \{E_i : e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\dots m} \end{cases}$$

The notation $\delta : E_i \to E$ means $E \vdash v\delta : t\delta$ for all $v : t \in E_i$

Fig. 3. Enviroment lookup

(Term/Prod)
$$\frac{E \vdash t_1 : \mathrm{s}_1 \quad E; v : t_1 \vdash t_2 : \mathrm{s}_2}{E \vdash \Pi v{:}t_1.\, t_2 : \mathrm{s}_3}$$
where $(\mathrm{s}_1, \mathrm{s}_2, \mathrm{s}_3) \in \mathcal{R}$

(Term/Abs)
$$\frac{E; v : t_1 \vdash e : t_2 \quad E \vdash \Pi v{:}t_1.\, t_2 : \mathrm{s}}{E \vdash \lambda v{:}t_1.\, e : \Pi v{:}t_1.\, t_2}$$

(Term/Ax)
$$\frac{E \vdash \mathsf{ok}}{E \vdash \mathrm{s}_1 : \mathrm{s}_2}$$
where $(\mathrm{s}_1, \mathrm{s}_2) \in \mathcal{A}$

(Term/App)
$$\frac{E \vdash e : \Pi v{:}t_1.\, t_2 \quad E \vdash e' : t_1}{E \vdash e\, e' : t_2\{v \mapsto e'\}}$$

(Term/Conv)
$$\frac{E \vdash e : t \quad E \vdash t' : \mathrm{s} \quad E \vdash t \approx t'}{E \vdash e : t'}$$

Fig. 4. PTS rules

function over natural numbers. The usefulness of abstract functions equipped with rewrite rules will become clearer after the introduction of modules, because they will allow us to instantiate abstract elements with concrete ones.

Definition 2. *A correct environment E is* closed *if it contains no variable declarations and if all definitions by rewriting are complete, i.e. every time E can be split into $E_1; \mathsf{RewDef}(E^\Gamma, R); E_2$ the condition $\mathsf{COMP}_{E_1}(E^\Gamma, R)$ is satisfied.*

Definition 3. *A pure type system with generative definitions is* logically consistent *if* `False` *is not inhabited in any closed environment.*

Summarizing, conditions $\mathsf{ACC}()$ and $\mathsf{POS}()$ are supposed to guarantee that typing in every correct environment is decidable and $\mathsf{COMP}()$ is supposed to guarantee that `False` is not provable in any closed environment.

The problem of consistency for the calculus of constructions with rewriting has not been very well studied yet. The only condition that we know that guarantees consistency is given in [2]. Please see the discussion of this question at the end of Sect. 5.

3 Calculus of Named Modules

In this section we define a calculus built on top of a pure type system, containing inductive definitions, definitions by rewriting and a system of named modules. The latter is an adaptation of an ML-style module system with the following usual features:

Structures bundle together related definitions and lemmas. They correspond to records.

Signatures – module types of structures – play the role of interfaces. They correspond to record types with manifest fields [13,11].

Functors are parametric modules. Their application to modules makes instantiation very convenient. They correspond to dependent functions.

Functor types – module types of functors – correspond to dependent products.

Higher order functors are functors taking parameters which are themselves functors.

Subtyping allows to apply functors to modules with more components. It is monotone for signatures and contravariant in the argument type and covariant in the result type for functor types.

Nested modules – modules can be components of stuctures and signatures.

The reader is invited to consult Sect. 1.5.1 of [4] for simple examples of all these features. Below we present the details of the calculus of named modules. We deliberately ignore the problem of name clashes in nested structures and signatures, referring the reader to [4] again for details.

Access paths	p	$::= v \mid p.v$
Terms	e, t	$::= p \mid \mathsf{s} \mid (e_1\ e_2) \mid \lambda v{:}t.\, e \mid \Pi v{:}t_1.\, t_2$
Blocks	B	$::= \overline{p} \mid \mathsf{Rew}(E^{\Gamma}, R) \mid \mathsf{Ind}(E^I := E^C)$
Block types	BT	$::= \mathsf{RewSpec}(E^{\Gamma}, R) \mid \mathsf{IndSpec}(E^I := E^C)$
Rewriting systems	R	$::= E_1 : l_1 \longrightarrow r_1 : t_1 \ldots E_n : l_n \longrightarrow r_n : t_n$
Modules		
	m	$::= p \mid \mathsf{Struct}\ \overline{v}_1 : S_1 := P_1 \ldots \overline{v}_n : S_n := P_n\ \mathsf{End} \mid \mathsf{Functor}[v{:}M]\ m \mid (p_1\ p_2)$
Module types	M	$::= \mathsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \mathsf{End} \mid \mathsf{Funsig}(v{:}M_1)\ M_2$
Implementations	P	$::= e \mid B \mid m$
Specifications	S	$::= \mathsf{Ty}(t) \mid \mathsf{Eq}(e : t) \mid \mathsf{Ty}(BT) \mid \mathsf{Eq}(\overline{p} : BT) \mid \mathsf{Ty}(M) \mid \mathsf{Eq}(p : M)$
Specification sorts	SS	$::= \texttt{modtype} \mid \texttt{blocktype} \mid \texttt{spec}$
Environments	E	$::= \epsilon \mid \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n$

Fig. 5. Syntax

Specifications. In order to avoid giving many sets of similar rules, for example for extracting information from the environment and for extracting information from module signatures, we decided (after Courant) to introduce the auxiliary notion of specifications. They allow to factorize extraction rules into one rule for extracting a specification from the environment, one rule for extracting a specification from a signature and one set of rules to extract information from a specification. Specifications are also used to factorize rules for environment creation, typing rules for structures and subtyping rules for signatures.

There are two kinds of specifications: abstract $\mathsf{Ty}(\Phi)$ and manifest $\mathsf{Eq}(\psi : \Phi)$. Using specifications, a variable declaration is written $v : \mathsf{Ty}(\Phi)$, which means that v has type Φ, and a constant declaration is written $v : \mathsf{Eq}(\psi : \Phi)$, meaning that v is of type Φ and equal to ψ.

This way an environment can be uniformly presented as an assignment of specifications to names. Inductive definitions and definitions by rewriting are also tailored to match this framework, but this will be explained later.

Structures and Signatures. From the point of view of typing, signatures correspond to fragments of correct environments (see the rule **(Sig/Form)** in Fig. 7) and structures correspond to fragments of closed environments.

Like environments, signatures assign specifications to names. Structures assign specifications and implementations to names. The typing rule for structures **(Sig/Struct)** explains the role of these elements: every implementation satisfies its corresponding specification, but implementations are not used to type-check subsequent components.

This gives us a nice way to formally distinguish between lemmas with proofs and definitions of constants. A lemma is represented as a triple $v : \mathsf{Ty}(t) := e$ where v is its name, t its formulation and e its proof, which is not used to type subsequent components. A constant definition is a triple $v : Eq(e : t) := e$, where the equality $v \approx e$ can be deduced from its specification and hence used to type subsequent components. Even though the necessity of duplicating e may look strange, advantages of using specifications are worth it.

Blocks and Block Types. Apart from lemmas and constant definitions, structures can also contain inductive definitions and definitions by rewriting (and also modules themselves). Suppose m is a structure containing the first three definitions given in Sect. 2, i.e. the inductive definition of natural numbers, the function *plus* defined by rewriting and the definition of *two*. If we bind m to a name, say A, we expect to be able to use the elements of m, prefixed by A. In particular $A.O$ and $A.S$ should both be constructors of the inductive type $A.nat$ and the rewrite rule $A.plus\ A.O\ x \longrightarrow x$ should be available in the conversion.

Since we want our module system to allow separate checking of modules, the extraction of information from modules should be based on module types and not on module expressions. Therefore it is necessary to put information about inductive definitions and definitions by rewriting inside signatures. In order to enforce the parallel between structures and closed environments, rewriting in structures should be subject to both $\mathsf{ACC}()$ and $\mathsf{COMP}()$ conditions, and rewriting in signatures — just to $\mathsf{ACC}()$.

This is exactly the case in our system. We decided to split $\mathsf{RewDef}(E^\Gamma, R)$ into two constructions $\mathsf{RewSpec}(E^\Gamma, R)$ and $\mathsf{Rew}(E^\Gamma, R)$, the first being an element of specifications and subject to the condition $\mathsf{ACC}(E^\Gamma, R)$, and the latter an implementation and subject to the condition $\mathsf{COMP}(E^\Gamma, R)$ (see rules **(RewSpec/Form)** and **(RewSpec/Rew)** in Fig. 10).

The reason why the syntactic class of specifications does not directly contain $\mathsf{RewSpec}(E^\Gamma, R)$ comes from considering the operation of module renaming:

```
Module B:=A.
```

After that operation, the module type of B should reflect both the fact that the *plus* component is a symbol defined by rewriting and the fact that it is equal to *A.plus*. Since we want to have the principal types property for module types, we decided to wrap $\mathsf{RewSpec}(E^\Gamma, R)$ into specifications $\mathsf{Ty}()$ and $\mathsf{Eq}()$.

Inductive definitions are treated similarly: we also split $\mathsf{IndDef}(E^I := E^C)$ into $\mathsf{IndSpec}(E^I := E^C)$ and $\mathsf{Ind}(E^I := E^C)$, but the rule **(IndSpec/Ind)** for typing $\mathsf{Ind}(E^I := E^C)$, has no additional side-condition compared to the rule **(IndSpec/Form)** from Fig. 9.

Note that inductive definitions and definitions by rewriting both have a potential to mutually define many symbols. Therefore we decided to create a new syntactic class of *blocks*, denoting a possibly mutually defined sequence of terms. There is also a whole typing hierarchy for blocks. Next to terms, types, sorts and term specifications, we introduce blocks, block types, the sort `blocktype` and block specifications. A similar hierarchy is also available for modules: there are module expressions, module types, the sort `modtype` and module specifications.

Note that the rules **(IndSpec/Form)** and **(RewSpec/Form)**, correspond to rules from Fig. 2 and a sequence **(Ty/Form)** or **(Eq/Form)** followed by **(Env/Insert)** corresponds to environment insertion rules from Fig. 1. Similarly, a sequence **(Env/Lookup)**, followed by **(Ty/Type)** or **(Eq/Type)** or

$$
\textbf{(Env/Empty)}\quad \frac{}{\epsilon \vdash \mathtt{ok}}
\qquad\qquad
\textbf{(Env/Insert)}\quad \frac{E \vdash \mathtt{ok} \quad E \vdash S : \mathtt{spec}}{E; \overline{v} : S \vdash \mathtt{ok}}
\qquad\qquad
\boxed{\textbf{(Env/Lookup)}\quad \frac{E_1; \overline{v} : S; E_2 \vdash \mathtt{ok}}{E_1; \overline{v} : S; E_2 \vdash \overline{v} : S}}
$$

Fig. 6. Environment

(Eq/Comp), followed optionally by **(IndSpec/IndType)**, **(IndSpec/Constr)**, **(RewSpec/Fun)** or the rule **(RewSpec/Comp)** correspond to environment lookup rules from Fig. 3.

Calculus of Named Modules. Our calculus contains a module system which is a syntactic restriction of the module calculus of Courant. Indeed, the syntactic class of terms does not contain the general selection operator $m.v$, where m is a module expression, but only its restriction to so-called access paths [13]. The same restriction also appears in module and block manifest specifications and in module application.

Definition 4. *The* Calculus of Named Modules *is defined by the typing rules on Fig. 4,*[3] *6, 7, 8, 9, 10 and 11.*[4] *The relation* $\approx$ *appearing in the premises of the rules* **(Term/Conv)** *and* **(Block/Conv)** *is the congruence defined by beta reduction and rules* **(Eq/Comp)** *and* **(RewSpec/Comp)**.

4 Examples

Now we can show the representation expressed in our formal abstract syntax of the binding `Module A:=`m, where m is the structure mentioned in the previous section:

$$
\begin{array}{l}
A : \mathsf{Ty}(\ldots) := \mathsf{Struct} \\
\qquad\qquad \begin{array}{rcl}
nat, O, S & : & \mathsf{Ty}(\mathsf{IndSpec}(E^I_{nat} := E^C_{nat})) \;:=\; \mathsf{Ind}(E^I_{nat} := E^C_{nat}) \\
plus & : & \mathsf{Ty}(\mathsf{RewSpec}(E^{\Gamma}_{plus}, R_{plus})) \;:=\; \mathsf{Rew}(E^{\Gamma}_{plus}, R_{plus}) \\
two & : & \mathsf{Eq}(plus\ (S\ O)\ (S\ O) : nat) \;:=\; plus\ (S\ O)\ (S\ O)
\end{array} \\
\qquad\quad \mathsf{End}
\end{array}
$$

where $E^I_{nat} = nat : \mathsf{Ty}(\mathtt{Set})$, $E^C_{nat} = O : \mathsf{Ty}(nat); S : \mathsf{Ty}(nat \to nat)$, $E^{\Gamma}_{plus} = plus : \mathsf{Ty}(nat \to nat \to nat)$ and R_{plus} is the system with 5 rules from Sect. 2. Note that nat (and other names as well) appears three times here: first as a name of a structure component, second as a local binder inside $\mathsf{IndSpec}(E^I_{nat} := E^C_{nat})$ and third as a local binder inside $\mathsf{Ind}(E^I_{nat} := E^C_{nat})$. Again, the duplication here is only needed to make the presentation with specifications possible.

The principal signature of the structure m (omitted in $\mathsf{Ty}(\ldots)$ above) can be obtained by removing the implementations (elements after $:=$) and replacing

[3] To be 100% formal, in the premises of rules **(Term/Prod)** and **(Term/Abs)** the environment should be written $E; v : \mathsf{Ty}(t_1)$.

[4] The framed rules form a subsystem called *principal*, which is used in Sect. 5.

(Sig/Form)
$$\frac{E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \mathtt{ok}}{E \vdash \textsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \textsf{End} : \mathtt{modtype}}$$

$$\boxed{\textbf{(Sig/Access)}\quad \frac{E \vdash p : \textsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \textsf{End}}{E \vdash \overline{p.v_k} : S_k\{\overline{v_i} \mapsto \overline{p.v_i}\}_{i=1\ldots k-1}}}$$

(Sig/Struct)
$$\frac{\begin{array}{c} E \vdash \textsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \textsf{End} : \mathtt{modtype} \\ E; \overline{v}_1 : S_1 \ldots \overline{v}_{k-1} : S_{k-1} \vdash P_k : S_k \quad \text{for } k = 1 \ldots n \end{array}}{E \vdash \textsf{Struct}\ \overline{v}_1 : S_1 := P_1 \ldots \overline{v}_n : S_n := P_n\ \textsf{End} : \textsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \textsf{End}}$$

(Sig/Sub)
$$\frac{\begin{array}{c} E \vdash \textsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \textsf{End} : \mathtt{modtype} \\ E \vdash \textsf{Sig}\ \overline{v}'_1 : S'_1 \ldots \overline{v}'_{n'} : S'_{n'}\ \textsf{End} : \mathtt{modtype} \\ E; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n \vdash \overline{v}'_k : S'_k \quad \text{for } k = 1 \ldots n' \end{array}}{E \vdash \textsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \textsf{End} <: \textsf{Sig}\ \overline{v}'_1 : S'_1 \ldots \overline{v}'_{n'} : S'_{n'}\ \textsf{End}}$$

(Funsig/Form)
$$\frac{\begin{array}{c} E \vdash M_1 : \mathtt{modtype} \\ E; v : \textsf{Ty}(M_1) \vdash M_2 : \mathtt{modtype} \end{array}}{E \vdash \textsf{Funsig}(v{:}M_1)\ M_2 : \mathtt{modtype}}$$

(Funsig/Functor)
$$\frac{\begin{array}{c} E; v : \textsf{Ty}(M_1) \vdash m : M_2 \\ E \vdash \textsf{Funsig}(v{:}M_1)\ M_2 : \mathtt{modtype} \end{array}}{E \vdash \textsf{Functor}[v{:}M_1]\ m : \textsf{Funsig}(v{:}M_1) M_2}$$

(Funsig/Sub)
$$\frac{E \vdash M'_1 <: M_1 \quad E; v : \textsf{Ty}(M'_1) \vdash M_2 <: M'_2}{E \vdash \textsf{Funsig}(v{:}M_1)\ M_2 <: \textsf{Funsig}(v{:}M'_1)\ M'_2}$$

(Mod/App)
$$\frac{E \vdash p : \textsf{Funsig}(v : M_1)\ M_2 \quad E \vdash p' : M_1}{E \vdash p\ p' : M_2\{v \mapsto p'\}}$$

(Subsumption)
$$\frac{E \vdash m : M_1 \quad E \vdash M_1 <: M_2}{E \vdash m : M_2}$$

(Strengthenning)
$$\frac{E \vdash p : M \qquad E \vdash M : \mathtt{modtype}}{E \vdash p : M/p}$$

$$\begin{array}{rcl} \textsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \textsf{End}/p & = & \textsf{Sig}\ \overline{v}_1 : S_1/\overline{p.v_1}; \ldots ; \overline{v}_n : S_n/\overline{p.v_n}\ \textsf{End} \\ (\textsf{Funsig}(v{:}M_1)\ M_2)/p & = & \textsf{Funsig}(v{:}M_1)\ M_2 \end{array}$$

$$\begin{array}{rclrcl} \textsf{Ty}(t)/p & = & \textsf{Eq}(p : t) & \textsf{Eq}(e' : t)/p & = & \textsf{Eq}(e' : t) \\ \textsf{Ty}(BT)/\overline{p} & = & \textsf{Eq}(\overline{p} : BT) & \textsf{Eq}(\overline{p}' : BT)/\overline{p} & = & \textsf{Eq}(\overline{p}' : BT) \\ \textsf{Ty}(M)/p & = & \textsf{Eq}(p : M/p) & \textsf{Eq}(p' : M)/p & = & \textsf{Eq}(p' : M/p) \end{array}$$

Fig. 7. Modules

Symbols ϕ, Ψ, Ξ are either e, t, s, or $p, M, \mathtt{modtype}$, or $\overline{p}, BT, \mathtt{blocktype}$.

(Ty/Form)
$$\frac{E \vdash \Psi : \Xi}{E \vdash \textsf{Ty}(\Psi) : \mathsf{spec}}$$

(Ty/Sat)
$$\frac{E \vdash P : \Psi \qquad E \vdash \textsf{Ty}(\Psi) : \mathsf{spec}}{E \vdash P : \textsf{Ty}(\Psi)}$$

$$\boxed{\textbf{(Ty/Type)}\quad \frac{E \vdash P : \textsf{Ty}(\Psi)}{E \vdash P : \Psi}}$$

(Eq/Form)
$$\frac{E \vdash \phi : \Psi \qquad E \vdash \Psi : \Xi}{E \vdash \textsf{Eq}(\phi : \Psi) : \mathsf{spec}}$$

(Eq/Sat)
$$\frac{E \vdash P \approx \phi \qquad E \vdash P : \Psi \qquad E \vdash \textsf{Eq}(\phi : \Psi) : \mathsf{spec}}{E \vdash P : \textsf{Eq}(\phi : \Psi)}$$

$$\boxed{\textbf{(Eq/Type)}\quad \frac{E \vdash P : \textsf{Eq}(\phi : \Psi)}{E \vdash P : \Psi}} \qquad \boxed{\textbf{(Eq/Comp)}\quad \frac{E \vdash P : \textsf{Eq}(\phi : \Psi)}{E \vdash P \approx \phi}}$$

Fig. 8. Specifications

Let $E^I = v_1^I : \mathsf{Ty}(t_1^I) \ldots v_k^I : \mathsf{Ty}(t_k^I)$ and $E^C = v_1^C : \mathsf{Ty}(t_1^C) \ldots v_n^C : \mathsf{Ty}(t_n^C)$

(IndSpec/Form)

$$\frac{E \vdash t_j^I : s_j \text{ for } j = 1 \ldots k \qquad E; E^I \vdash t_i^C : s'_i \text{ for } i = 1 \ldots n}{E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype}}$$

if $\mathsf{POS}_E(E^I := E^C)$

(IndSpec/Ind)

$$\frac{E \vdash \mathsf{IndSpec}(E^I := E^C) : \mathtt{blocktype}}{E \vdash \mathsf{Ind}(E^I := E^C) : \mathsf{IndSpec}(E^I := E^C)}$$

(IndSpec/IndType)

$$\frac{E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)}{E \vdash p^j : t_j^I}$$

if $j \in \{1 \ldots k\}$

(IndSpec/Constr)

$$\frac{E \vdash \overline{p} : \mathsf{IndSpec}(E^I := E^C)}{E \vdash p^{k+i} : t_i^C\{v_j^I \mapsto p^j\}_{j=1\ldots k}}$$

if $i \in \{1 \ldots n\}$

Fig. 9. Inductive blocks

Let $E^\Gamma = v_1 : \mathsf{Ty}(t_1) \ldots v_n : \mathsf{Ty}(t_n)$ and
$R = \{E_i : e_i^L \longrightarrow e_i^R : \hat{t}_i\}_{i=1\ldots m}$, where $E_i = v_1^i : \mathsf{Ty}(t_1^i); \ldots; v_{n_i}^i : \mathsf{Ty}(t_{n_i}^i)$

(RewSpec/Form)

$$\frac{\begin{array}{c} E \vdash t_k : s_k \quad \text{for } k = 1 \ldots n \\ \left.\begin{array}{cc} E; E_i \vdash \mathtt{ok} & E; E^\Gamma; E_i \vdash \hat{t}_i : \hat{s}_i \\ E; E^\Gamma; E_i \vdash e_i^L : \hat{t}_i & E; E^\Gamma; E_i \vdash e_i^R : \hat{t}_i \end{array}\right\} \text{for } i = 1 \ldots m \end{array}}{E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}}$$

if $\mathsf{ACC}_E(E^\Gamma, R)$

(RewSpec/Rew)

$$\frac{E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype}}{E \vdash \mathsf{Rew}(E^\Gamma, R) : \mathsf{RewSpec}(E^\Gamma, R)}$$

if $\mathsf{COMP}_E(E^\Gamma, R)$

(RewSpec/Sat)

$$\frac{\begin{array}{c} E \vdash \mathsf{RewSpec}(E^\Gamma, R) : \mathtt{blocktype} \\ E \vdash p^j : t_j \text{ for } j = 1 \ldots n \\ E; E_i \vdash e_i^L\theta \approx e_i^R\theta \text{ for } i = 1 \ldots m \end{array}}{E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)}$$

where $\theta = \{v_j \mapsto p^j\}_{j=1\ldots n}$

(RewSpec/Fun)

$$\frac{E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R)}{E \vdash p^j : t_j}$$

(RewSpec/Comp)

$$\frac{E \vdash \overline{p} : \mathsf{RewSpec}(E^\Gamma, R) \qquad \delta : E_i \to E}{E \vdash e_i^L\theta\delta \approx e_i^R\theta\delta}$$

where $\theta = \{v_j \mapsto p^j\}_{j=1\ldots n}$

The notation $\delta : E_i \to E$ means that $E \vdash v\delta : t\delta$ holds for every $v : \mathsf{Ty}(t) \in E_i$.

Fig. 10. Rewriting blocks

(Block/Conv)

$$\frac{E \vdash B : BT \qquad E \vdash BT' : \mathtt{blocktype} \qquad E \vdash BT \approx BT'}{E \vdash B : BT'}$$

Fig. 11. Block conversion

Struct with Sig. The principal signature of B defined by `Module B:=A` is more interesting. It is obtained by using the **(Strengthening)** rule (Fig. 7):

$$\begin{array}{l}
\mathsf{Sig} \\
\quad nat, O, S \ : \ \mathsf{Eq}(A.nat, A.O, A.S : \mathsf{IndSpec}(E^I_{nat} := E^C_{nat})) \\
\quad plus \ : \ \mathsf{Eq}(A.plus : \mathsf{RewSpec}(E^\Gamma_{plus}, R_{plus})) \\
\quad two \ : \ \mathsf{Eq}(plus\ (S\ O)\ (S\ O) : nat) \\
\mathsf{End}
\end{array}$$

Note that using **(Sig/Access)** followed by **(Eq/Type)** we can derive the judgment $E \vdash B.nat, B.O, B.S : \mathsf{IndSpec}(E^I_{nat} := E^C_{nat})$, meaning that $B.nat$,$B.O$, $B.S$ form an inductive family, and using **(Eq/Comp)** instead, we derive the equalities $B.nat \approx A.nat$, $B.O \approx A.O$ and $B.S \approx A.S$.

Module subtyping is governed by **(Sig/Sub)** and **(Funsig/Sub)**, as well as satisfaction rules **(Ty/Sat)**, **(Eq/Sat)** and **(RewSpec/Sat)**. Using the extraction rules followed by the satisfaction rules it is possible to prove the following:

$$\mathsf{Sig}\ nat, O, S : \mathsf{Ty}(\mathsf{IndSpec}(E^I_{nat} := E^C_{nat}))\ \mathsf{End}\ \ <:\ \ \mathsf{Sig}\ nat : \mathsf{Ty}(\mathtt{Set})\ \mathsf{End}$$

$$\mathsf{Sig}\ plus : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma_{plus}, R_{plus}))\ \mathsf{End}\ \ <:\ \ \mathsf{Sig}\ plus : \mathsf{Ty}(nat \to nat \to nat)\ \mathsf{End}$$

Rewrite specifications used in the parameter interface of a functor may ease the development of the functor body. They impose convertibility constraints on the functor arguments. Let us suppose that $P : nat \to \mathtt{Prop}$ is a predicate, and let

$$\begin{array}{ll}
E^\Gamma_{fg} = f{:}\,\mathsf{Ty}(nat \to nat);\quad g : \mathsf{Ty}(nat \to nat) & \\
 & R_{fg} = n{:}\,\mathsf{Ty}(nat) \mid f\ (g\ n) \longrightarrow g\ (f\ n)\ \ : nat \\
E^\Gamma_h = h{:}\,\mathsf{Ty}(nat \to nat) & R_h = m{:}\,\mathsf{Ty}(nat) \mid h\ m \longrightarrow S(S\ m)\ \ : nat
\end{array}$$

Let us now consider a functor F of the following type

$$\begin{array}{l}
\mathsf{Funsig}(X : \mathsf{Sig}\ f,g : \mathsf{Ty}(\mathsf{RewSpec}(E^\Gamma_{fg}, R_{fg}))\ \mathsf{End}) \\
\quad \mathsf{Sig} \\
\quad\quad lemma : \mathsf{Eq}(\ \lambda n{:}\,nat.\ \lambda p{:}\,P(X.f(X.f(X.g\ n))).\ p \\
\quad\quad\quad\quad :\ \Pi n{:}\,nat.\ P(X.f(X.f(X.g\ n))) \to P(X.g(X.f(X.f\ n)))\ ) \\
\quad \mathsf{End}
\end{array}$$

In the functor result type, correctness of the specification of its single $lemma$ component relies on the rewrite rule placed in the functor parameter signature. Now F can be applied to a module path of type

$$\begin{array}{l}
\mathsf{Sig} \\
\quad g : \mathsf{Eq}(\ \lambda n{:}\,nat.\ S\ n\ :\ nat \to nat\ ) \\
\quad f : \mathsf{Ty}(\ \mathsf{RewSpec}(E^\Gamma_h, R_h)\ ) \\
\mathsf{End}
\end{array}$$

In fact, the rule **(RewSpec/Sat)** allows any sequence of function symbols to satisfy $\mathsf{RewSpec}(E^\Gamma, R)$ as long as the corresponding conversion can prove equalities of left-hand and right-hand sides of all rules in R. In our example, the structure defines the functions f and g in such a way that both $f(g\,n)$ and $g(f\,n)$ are convertible with $S(S(S\ n))$, and therefore the equation expressed by the rewrite rule in R_{fg} is satisfied.

5 Conservativity

Our calculus of named modules is a conservative extension of the pure type system with generative definitions. This section presents the outline of the proof and a discussion about acceptance criteria for definitions by rewriring.

Closure properties. The calculus of named modules is parametrized by the side-conditions for acceptance of inductive definitions and definitions by rewriting. In order to make these definitions coexist with modules and in particular with module subtyping and functor applications, the acceptance conditions must be closed under some operations on environments. Formally, each side-condition C (POS, ACC and COMP) must satisfy the following closure properties:

$$\mathsf{C}_E(\Omega) \qquad \text{implies} \qquad \mathsf{C}_{E'}(\Omega') \qquad \text{if:}$$

$$\begin{array}{lll}
\text{(C1)}\ E = E_1; \overline{v} : S; E_2 & E' = E_1; E_2\{\overline{v} \mapsto \overline{p}\} & \text{with} \quad E_1 \vdash \overline{p} : S \\
\quad\ \ \Omega & \Omega' = \Omega\{\overline{v} \mapsto \overline{p}\}, & \\
\text{(C2)}\ E = E_1; E_3 & E' = E_1; E_2; E_3 & \\
\quad\ \ \Omega & \Omega' = \Omega & \\
\text{(C3)}\ E = E_1; v : \mathsf{Ty}(\mathsf{Sig}\ \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n\ \mathsf{End}); E_2 & & \\
\quad\ \ \Omega & E' = E_1; \overline{v}_1 : S_1 \ldots \overline{v}_n : S_n; E_2\{\overline{v.v_i} \mapsto \overline{v}_i\} & \\
 & \Omega' = \Omega\{\overline{v.v_i} \mapsto \overline{v}_i\} & \\
\text{(C4)}\ E = E_1; v : \mathsf{Ty}(\mathsf{Funsig}(\hat{v} : M)M'); E_2 & & \\
\quad\ \ \Omega & E' = E_1; E_2 & \\
 & \Omega' = \Omega &
\end{array}$$

In all of the above implications it is assumed that both environments E and E' are correct. Moreover in the last two properties, E_2 does not contain modules.

Even though these closure properties look complicated, the first two simply correspond to basic meta-theoretical properties of most type-systems: substitutivity and weakening. It has to be noticed that the condition (C1) is at the same time simpler and harder than usual substitutivity. On one hand, only paths can be substituted which is a crucial simplification for existing syntactic termination criteria for rewriting. On the other hand, the typing judgment $E_1 \vdash \overline{p} : S$ can result from module subtyping or block satisfaction rules, which means that $\overline{p}$ can be more precisely specified than the original variables $\overline{v}$.

The property (C2) simply corresponds to modularity of a given acceptance criterion and is generally needed for effective reloading of previously checked developments. The remaining conditions (C3) and (C4) are used in the conservativity proof presented below.

Conservativity. In a calculus with modules, a proof development can be represented as a structure well-typed in the empty environment. We show that every such structure ending in a proof of `False` can be transformed into a inconsistent closed environment of the underlying pure type system with generative definitions.

Theorem 1. *Every structure well-typed in the empty environment, ending in a component v : $\mathsf{Ty}(\texttt{False}) := e$ can be transformed into a closed environment of the underlying pure type system with generative definitions, ending with v : $\mathsf{Eq}(e' : \texttt{False})$ for some term e'.*

Proof sketch. Suppose first that the initial structure contains neither sub-modules nor manifest blocks. Then it is trivial to transform it into a closed environment of the underlying pure type system.

Therefore it remains to be shown that any structure can be transformed into one without sub-modules and manifest blocks. Since every manifest block $\overline{v} : \mathsf{Eq}(\overline{v}' : BT) := B$ in a structure without sub-modules can be eliminated by simply substituting $\overline{v}'$ to $\overline{v}$ in the rest of the structure, let us concentrate on sub-modules. The elimination of the latter is done in two phases. First, going from left to right, all module applications in the initial structure are recursively *evaluated* to weak-head normal forms: we replace $p_1\, p_2$ with $m\{v \mapsto p_2\}$ where p_1 is already evaluated to $\mathsf{Functor}[v\!:\!M]m$. Second, sub-modules are eliminated from right to left, by simply removing functors and *flattening* structures.

Below we present the two phases of the conservativity proof in a bit more detailed way and explain the role of closure properties.

Evaluation. While replacing the functor application $p_1\, p_2$ by $m\{v \mapsto p_2\}$, one must make sure that the latter expression is well-typed given that m is well typed in an environment in which p_1 was defined, extended with $v : \mathsf{Ty}(M)$. If m contains definitions by rewriting or inductive definitions, their correctness after the substitution can only be proved if the side-conditions verify closure properties (C1) and (C2). The latter assures that m is still correct in the environment where p_1 is used extended with $v : \mathsf{Ty}(M)$ and (C1) assures that the substitution $\{v \mapsto p_2\}$ can safely be applied.

The other difficulty in the first phase lies in the termination proof. Fortunately our calculus is similar to the calculus of Courant and after solving some technical problems we can use his results to obtain termination.

Flattening. The second phase requires showing that module flattening preserves types. The simple proof by induction on the derivation does not work, so we have to introduce an auxiliary calculus, called principal, consisting of typing rules for terms (Fig. 4) and rules for extracting information from the environment (framed rules in Fig. 6, 7, 8, 9 and 10). We show that term judgments derivable in both calculi in a given correct environment are the same. Since type preservation of flattening in the principal calculus can be proved by simple induction on the derivation, using closure properties (C3) and (C4), we are done. The complete proof can be found in [4]. $\square$

Satisfying closure properties. Let us now fix the base pure type system to be the calculus of constructions. Since inductive types are often theoretically analysed as anonymous entities, all known positivity conditions POS for inductive types obviously satisfy the closure properties.

It turns out also that syntactic order-based termination criteria for object-level rewriting, as defined in [2,19], restricted by the requirement that all function symbols appearing in the left-hand sides of rules of R belong to E^{Γ} and the critical pairs test, also satisfy the above properties and therefore are suitable for ACC. In great simplification both termination criteria [2,19] restrict right-hand sides of rewrite rules to be constructed from local variables, previously defined function symbols, constructors of inductive types and recursive calls with *smaller* arguments. Smaller here means (extended) subterm relation for [2] or a recursive comparison for [19]. The rewriting system defined is Sec. 2 is accepted by both termination criteria and all its critical pairs are joinable.

Due to modularity constraints it seems unlikely to allow foreign function symbols in left-hand sides of rules as in the third rule below:

```
Functor [A : Sig Rewriting plus,mult : nat → nat → nat Rules R End]
Struct
  Rewriting exp : nat → nat → nat
  Rules
    exp x 0 ⟶ S 0
    exp x (S n) ⟶ A.mult x (exp x n)
    exp x (A.plus n m) ⟶ A.mult (exp x n) (exp x m)
End
```

Indeed, even if current rewrite rules R for `plus` and `mult` allow to prove local confluence of the system for `exp`, when some module is substituted to `A`, containing another definition of `plus` and `mult`, the system may stop being confluent.

The status of the condition COMP is the least clear, because little is known about consistency of the calculus of constructions with rewriting. It is believed that *completeness of definitions* techniques of [6] requiring that all constructor instances of a function symbol are reducible, can be extended to the calculus of constructions and proved to guarantee consistency. Such a condition COMP would certainly satisfy the closure properties. Some work in this direction has already been done in [2], but the completeness criterion given in this paper is not modular.

6 Conclusions

We have presented the formalization of a pure type system with generative inductive definitions and definitions by rewriting and showed that its extension with a system of named modules is conservative. The proof allowed us to formulate closure properies which must be satisfied by acceptance criteria for rewrite rules in order to be useful in the calculus with modules.

Since some existing acceptance criteria can already be adapted to satisfy the closure properties, the implementation of a term rewriting engine in Coq is by no means stopped by the modules. Still, finding really flexible acceptance criteria for rewriting is a challenging subject for future work.

Acknowledgments. The author would like to thank the anonymous referee for the extensive comments and suggestions.

References

1. Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. In *Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 9–18, Los Alamitos, USA, June 2001. IEEE Comp. Soc. Press.
2. Frédéric Blanqui. Definitions by rewriting in the Calculus of Constructions. *Mathematical Structures in Computer Science*, 2003. To appear.
3. Jacek Chrząszcz. Implementation of modules in the Coq system. In David Basin and Burkhart Wolff, editors, *Proceedings of the Theorem Proving in Higher Order Logics 16th International Conference*, volume 2758 of *LNCS*, pages 270–286, Rome, Italy, September 2003. Springer Verlag.
4. Jacek Chrząszcz. *Modules in Type Theory with Generative Definitions.* PhD thesis, Warsaw Univerity and University of Paris-Sud, Jan 2004. Available from `http://www.mimuw.edu.pl/~chrzaszc/papers`
5. The Coq proof assistant. `http://coq.inria.fr/`.
6. Thierry Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, Båstad, Sweden, 1992.
7. Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76:95–120, February 1988.
8. 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.
9. 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.
10. Judicaël Courant. *Un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.
11. Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL'94 [17], pages 123–137.
12. Robert Harper and Frank Pfenning. A module system for a programming language based on the LF logical framework. *Journal of Logic and Computation*, 1(8), 1998.
13. Xavier Leroy. Manifest types, modules, and separate compilation. In POPL'94 [17], pages 109–122.
14. Zhaohui Luo. *An Extended Calculus of Constructions.* PhD thesis, University of Edinburgh, 1990.
15. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. *PVS Language Reference.* Computer Science Laboratory, SRI International, Menlo Park, CA, September 1999.
16. Lawrence C. Paulson. Isabelle: the next 700 theorem provers. In P. Odifreddi, editor, *Logic and Computer Science.* Academic Press, 1990.
17. *Conference Record of the 21st Symposium on Principles of Programming Languages*, Portland, Oregon, 1994. ACM Press.
18. Daria Walukiewicz-Chrząszcz. Termination of rewriting in the calculus of constructions. *Journal of Functional Programming*, 13(2):339–414, 2003.
19. Daria Walukiewicz-Chrząszcz. *Termination of Rewriting in the Calculus of Constructions.* PhD thesis, Warsaw University and University Paris XI, 2003.
20. Benjamin Werner. *Méta-théorie du Calcul des Constructions Inductives.* PhD thesis, Université Paris 7, 1994.