Implementing Modules in the Coq System

Jacek Chrząszcz*

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

Abstract. The paper describes the implementation of interactive ML-style modules in the recent version 7.4 of Coq proof assistant. Modules (especially higher-order) provide a very convenient tool for parametrized theories which was lacking in Coq for years. Their interactive character extends naturally the interactive environment provided by the proof assistant. The implementation follows the paradigm of recent versions of Coq to separate the correctness-critical code from the rest of the system, using abstraction barriers.

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 a high degree of automation and ways to structure large developments.

From a long time ago, large developments in the Coq system [Coq] can be split across many files and a structure of one file can be nicely presented using sections. Moreover, sections provide a mechanism of abstracting individual definitions and theorems. However, what was lacking in Coq, was the possibility to conveniently define parametrized theories or parametrized certified data structures, easy to use in other developments.

If we were to implement such data structures in a functional programming language from the ML family, we would use modules. First, modules provide a means of bundling together related types and definitions into a structure. Second, the structure can be given an interface (signature), specifying what elements are to be used by the exterior of that structure and hiding its implementation details. Third, higher order modules (functors) provide a way of abstracting a structure over a module verifying a given interface. The abstracted module can later be conveniently applied to actual parameters. These three mechanisms are extended with nested modules, functor types and higher order functors, so that the module system resembles a typed lambda calculus with records, record types and monotone record subtyping.

All of this is now available in version 7.4 of Coq. First developments realized with modules prove that extension very promising and useful [MPMU03]. In this

* Partly supported by KBN Grant 7 T11C 028 20

paper we present modules from the implementation point of view. We show that our implementation follows the paradigm of the recent versions of Coq (from V7.0 on) according to which correctness-critical code should be separated by abstraction barriers from the rest of the system, responsible for user interface, interactive proof development, etc. [Fil00]. We discuss implementation choices and important modifications in Coq mechanisms.

From the theoretical point of view, a version of ML-style module system suitable for type theory has been proposed by Judicaël Courant in [Cou97,Cou98]. In his work, an arbitrary pure type system is extended with a calculus of anonymous modules. Courant shows the conservativity of the resulting system over the base PTS and termination and confluence of module reductions.

Unfortunately his calculus is too complicated for direct implementation in a proof assistant, which not only provides the type-checker of a suitable calculus but also contains various extra-logical elements, like grammar and pretty-printing rules, implicit arguments declarations, tactic definitions etc.

Therefore we decided to implement a module system with some simplifying restrictions, similar to those described in [Ler94]. First, every module expression that can be used in a term must be a name (or more exactly, an access path). Second, since complex module expressions cannot appear in terms, there are no reductions on modules and the module equality is restricted to equality of names. The exact set of typing rules and formal analysis of the implemented module system can be found in [Chr03].

Many other proof assistants provide a means to structure proofs. For example PVS theories and parametric theories [OS97] roughly correspond to our structures and first order functors respectively. Isabelle locales [Kam00] are rather concerned with convenient management of assumptions than with general structuring of big proof developments. They correspond to Coq sections with much extended functionnality. It is for example possible to reopen an already closed locale, have several open locales at the same time or interactively decide to abstract a given theorem a given number of levels.

The module system that is closest to ours appears to be the one implemented in Maude [DM98]. The main difference however is the lack of implicit module subtyping forcing users to write explicit coercions, called *views*, which are absent in our system.

The rest of the paper is organized as follows. We start by giving a general overview of Coq and its architecture and an overview of the module system. Then we present module related modifications in the Coq kernel and in the management of extra-logical features. The reader interested in the implementation details is kindly invited to consult the enhanced version of this paper available at `http://www.mimuw.edu.pl/~chrzaszc/papers/` or the Coq sources.

2 The Coq System

Coq is a proof assistant implementing the calculus of inductive constructions [CPM90]. It is based on the so-called Curry-Howard isomorphism, which relates

formulas to types and proofs to terms. Giving a proof of a theorem amounts to constructing a term of the type corresponding to the theorem. Coq provides a set of tactics to help the user interactively construct their proof-terms. One can also abbreviate complex terms by names (using definitions) and define inductive types.

Coq is entirely written in Objective Caml, the full system has about 120 000 lines of code in almost 500 files. The source tree is divided into the following directories, corresponding to implemented functionality:

kernel — the type-checker of the calculus of inductive constructions,
library — the backtracking mechanism, management of theory files and directories,
parsing — abstract syntax trees and management of standard and user defined parsing and pretty-printing rules,
interp — translation from the syntax-tree to untyped terms,
pretyping — analysis of untyped terms (generation of implicit arguments, etc.),
proofs — the structure and management of an interactive proof,
tactics — predefined tactics and the tactic language,
toplevel — interactive loop.

The kernel (about 10% of the code) is the only critical part of the system. Bugs in other parts of Coq cannot compromise the mathematical correctness of the theorem prover state [Fil00].

The implementation of modules follows closely this practice. Module type-checking has been added to the kernel, while the management of extra-logical module features is implemented outside, in the **library** directory.

3 Overview of the Module System

The goal of this section is to briefly describe the module system that we implemented and to show most typing rules concerning modules. The complete set of rules may be found in [Chr03].

The basic module expressions are structures, which represent sequences of toplevel Coq entries. For the sake of brevity of module typing rules, we ignore inductive definitions in this presentation and adopt the convention from [Cou97,Cou98] to represent the toplevel entries as triples: a name, a specification and an implementation. The specification is used for typing subsequent entries, and the implementation is provided only as a witness of the validity of the specification. According to this convention `Definition` $\mathtt{v}_1 : e_1 := e_1'$ is represented as $v_1 : \mathsf{Ty}(e_1)\mathsf{Eq}(e_1') := e_1'$, and `Lemma` $\mathtt{v}_2 : e_2$ with proof e_2' as $v_2 : \mathsf{Ty}(e_2) := e_2'$. The manifest specification $\mathsf{Ty}(e_1)\mathsf{Eq}(e_1')$ of v_1 indicates that the equality $v_1 \approx e_1'$ may be used in the conversion relation for typing subsequent entries. On the other hand, the abstract specification $\mathsf{Ty}(e_2)$ of v_2 does not export the equality $v_2 \approx e_2'$ and therefore it cannot be used in the conversion.

The basic module types used to type structures are signatures. They contain only the information relevant for typing, i.e. names v_i's and specifications t_i's. The typing rule for structures is the following:

(Struct)

$$\frac{E; v_1 : t_1 \ldots v_{i-1} : t_{i-1} \vdash q_i : t_i \quad \text{for all } i = 1 \ldots n}{E \ \vdash\ \mathsf{Struct}\ v_1 : t_1 := q_1 \ldots v_n : t_n := q_n\ \mathsf{End}\ :\ \mathsf{Sig}\ v_1 : t_1 \ldots v_n : t_n\ \mathsf{End}}$$

which shows that implementations q_i's are not used for typing indeed. Note that the signature assigned to a structure by this rule reflects all manifest specifications of the structure components.

Since structures can possibly be nested (i.e. contain module components), a triple $v : t := q$ may as well denote a module definition of the form $v : S := m$, where S is a module type and m a module expression.

In our system, the only way a module may be used in a term is when it is named. To access components of modules one uses the dot notation, and since modules can be nested, the module expressions allowed in terms are access paths (given by the grammar on the left). The specification of a variable v can simply be looked up in the environment and the specification of a longer access path $p.v$ can be inferred by the following typing rule (on the right):

$$p ::= v \mid p.v$$

(Access)

$$\frac{E\ \vdash\ p\ :\ \mathsf{Sig}\ v_1 : t_1 \ldots v_n : t_n\ \mathsf{End}}{E\ \vdash\ p.v_i\ :\ t_i\{v_j \mapsto p.v_j\}_{j<i}}$$

Note that the syntactic restriction that only access paths are allowed in terms is preserved by the substitution applied in the conclusion of the (Access) rule.

A functor is a module abstracted with respect to a module verifying a given module type. Functors informally correspond to λ-abstractions and their types to dependent products. In the rules below V is a variable, S and S' module types, m a module expression, and p_1, p_2 access paths.

(Functor)

$$\frac{E; V : S\ \vdash\ m : S'}{E\ \vdash\ \mathsf{Functor}[V : S]m\ :\ \mathsf{Funsig}(V : S)S'}$$

(App)

$$\frac{E \vdash p_1 : \mathsf{Funsig}(V : S)S' \quad E \vdash p_2 : S}{E \vdash p_1\ p_2 : S'\{V \mapsto p_2\}}$$

Note that in order to apply a functor to a module expression, both elements must be named. Due to this restriction, the resulting module type is always well-formed.

The last very important element of the module typing is the subtyping relation on module types, denoted $S_1 <: S_2$. On signatures the subtyping is a monotone extension of the subtyping on specifications of components (allowing to forget manifest specifications, to reorder and to drop fields in the supertype). For example, if $S_2 <: S_2'$ then

$$\mathsf{Sig}\ v_1 : \mathsf{Ty}(e_1);\ v_2 : S_2;\ v_3 : \mathsf{Ty}(e_3)\mathsf{Eq}(e_3')\ \mathsf{End}\ \ <:\ \ \mathsf{Sig}\ v_3 : \mathsf{Ty}(e_3);\ v_2 : S_2'\ \mathsf{End}$$

On functor types, subtyping is contravariant in the argument type and covariant in the result. The subtyping is used by the following subsumption rule:

(Sub)

$$\frac{E \vdash m : S \qquad E \vdash S <: S'}{E \vdash m : S'}$$

4 Modules in the Kernel

In the type-theory tradition, the kernel of a proof assistant is the type-checker of the calculus implemented in the system. Up to version 7.3 the Coq kernel implemented the type-checker for the calculus of inductive constructions. Now it also implements the type-checker for the module language.

The interface of the kernel to the rest of the Coq system is the abstract type `safe_environment` together with lookup functions and update functions, like `add_constant`, `add_inductive` etc. which return the updated environment only if the given element is correctly type-checked. The kernel is functional i.e. no in-place modification is performed and the update functions return the new environment or fail if type-checking is not successful.

An update function, called with `add_*` $E\ v\ (q,t)$, where E is a correct environment, v a name of the new element and (q,t) its body[1], implements the following typing rule:

$$(\textsc{Add}) \qquad \frac{E \vdash \texttt{ok} \qquad E \vdash q : t}{E; v : t := q \ \vdash \ \texttt{ok}} \quad \text{if } v \notin E$$

The left premise denotes the assumed correctness of the input environment E, the right premise denotes the type-checking of the new element, and the conclusion says that the returned updated environment $E; v : t := q$ is correct. It is important to note here that even though the environment sometimes contain implementations, they are never used for typing.

Adding modules requires the function `add_module` to be implemented, together with the module type-checker. Like previous `add_*` functions, this one also can be seen as implementing the rule (Add), with q a module body, and t a module specification (interface).

However it would be difficult for the user to write the whole definition of a module at once, especially for larger modules containing many definitions and proofs. Moreover, it would contradict the interactive character of Coq.

4.1 Interactive Modules

What seems natural in an interactive theorem prover is *interactive modules*. The user should have a possibility to declare the beginning of a new module, input definitions, declare and interactively prove some lemmas and then close the module, making all the entered definitions and lemmas the components of a new defined module. This is exactly what is implemented in version 7.4 of Coq. The following Coq session (on the left) results in a module body m (on the right) assigned to the name `M`:

[1] For the sake of clarity we ignore the fact that parts of the body may be synthesized.

```
Coq < Module M.
Interactive Module M started
Coq <   Definition T:=nat.
T is defined
Coq <   Definition x:=0.
x is defined
Coq <   Definition y:bool.
1 subgoal

  ============================
   bool
Coq <     Exact true.
Proof completed.
Coq <   Defined.
Exact true.
y is defined
Coq < End M.
Module M is defined
```

```
Struct
  Definition T:Set=nat.
  Definition x:nat=0.
  Definition y:bool:=true.
End
```

After closing a module, its components can be accessed using the dot notation:

```
Coq < Print M.x.
M.x = (0)
     : nat
```

A naive way to implement the `End M` operation would be to recover the whole body of the structure outside the kernel, retract the environment to the one before `Module M` and call `add_module` with the reconstructed module body and environment.

Let us see in details what would have to be done on the kernel side in order to perform all this. Let us call E the environment before `Module M` command. First, during the interactive session, the definitions of `T`, `x`, and `y` would have to be checked by the kernel. After `End M` the non-critical part of code would reconstruct the module body m and the kernel function `add_module` would be called with the environment E, the name `M` and the module body m. Like all `add_*` functions, `add_module` would type-check m before returning the new environment, according to the (ADD) rule. Now let us concentrate on the right premise of this rule. In our case q is m and therefore the rule (STRUCT) must be used, all the premises of which have already been verified during the interactive phase of module construction.

The naive implementation of module ending would make the kernel redo the job it did just a few moments before. For this reason we decided to turn the beginning and ending of the module into kernel operations. This allows us to eliminate the unnecessary double-checks.

So the interface of the kernel to the rest of the system has been extended with the functions `start_module`, marking the beginning of a module, and `end_module`, implementing (roughly) the rule:

$$\frac{E; v_1 : t_1 := q_1 \ldots v_n : t_n := q_n \vdash \texttt{ok}}{E;\ V\ :\ \mathsf{Sig}\ v_1 : t_1 \ldots v_n : t_n\ \mathsf{End} := \mathsf{Struct}\ v_1 : t_1 := q_1 \ldots v_n : t_n := q_n\ \mathsf{End} \vdash \texttt{ok}}$$

where V is the module name. The admissibility of this rule follows from the fact, that the only derivation of the premise is through a sequence of (ADD) rules. Their right premises imply

$$E \vdash \mathsf{Struct}\ v_1 : t_1 := q_1 \ldots v_n : t_n := q_n\ \mathsf{End}\ :\ \mathsf{Sig}\ v_1 : t_1 \ldots v_n : t_n\ \mathsf{End}$$

by an application of (STRUCT). Using (ADD) applied to $E \vdash \mathtt{ok}$ and the above, we get the desired conclusion.

The same interactive mechanism can also be used to declare parametric modules, i.e. functors. Given a signature `SIG` (on the right[2]), one can define an interactive functor (on the left):

```
Coq < Module Two[X:SIG][Y:SIG].
Interactive Module Two started
Coq <   Definition T:=X.T * Y.T.
Coq <   Definition x:=(X.x, Y.x).
Coq < End Two.
Module Two is defined
```

```
SIG = Sig
        Parameter T:Set.
        Parameter x:T.
      End
```

In this case the environment operation performed by the function `end_module` shortcuts not only the (STRUCT) rule, but also (l applications of) the (FUNCTOR) rule, leading to the admissible rule:

$$\frac{E;\ V_1 : S_1 \ldots V_l : S_l;\ v_1 : t_1 := q_1 \ldots v_n : t_n := q_n\ \vdash\ \mathtt{ok}}{\begin{array}{l} E;\ V : \mathsf{Funsig}(V_1 : S_1) \ldots \mathsf{Funsig}(V_l : S_l) \\ \qquad\quad \mathsf{Sig}\ v_1 : t_1 \ldots v_n : t_n\ \mathsf{End} \\ \quad := \mathsf{Functor}[V_1 : S_1] \ldots \mathsf{Functor}[V_l : S_l] \\ \qquad\quad \mathsf{Struct}\ v_1 : t_1 := q_1 \ldots v_k : t_k := q_k\ \mathsf{End}\ \vdash\ \mathtt{ok} \end{array}}$$

where $V_1, \ldots, V_l$ are formal parameters of the functor, $S_1, \ldots, S_l$ are their module types, and triples $v_i : t_i := q_i$ are functor body components.

The optional element of a module opening declaration is the expected module type. After the following definition:

```
Coq < Module N:SIG.
Coq <   Definition T:=bool.
Coq <   Definition x:=true.
Coq < End N.
```

the signature of the module body will be matched against the signature `SIG` and if the matching succeeds, the interface of `N` will be restricted to `SIG`. In our example the conversion equalities `N.T`$\approx$`bool` and `N.x`$\approx$`true` will be forgotten, making both these components abstract.

If a restricting signature is present, the implementation of the `end_module` function also involves the application of the subsumption rule (SUB).

[2] Using the convention described in Sect. 3, `Parameter v:`e is understood as a signature component $v : \mathsf{Ty}(e)$.

4.2 Overview of the Environment Structure

In order to implement the function `end_module` the type representing correct environments has to contain the information of how many definitions must be transformed into structure components, how many module declarations must be turned into functor parameters and (optionally) against what signature the body must be matched.

Therefore the type `safe_environment` represents in fact a stack of growing correct environments $E_1 \subseteq \cdots \subseteq E_n$. Fortunately, thanks to functional data structures and sharing, the elements of the environment E_1 are not copied in each $E_2, \ldots E_n$.

The operations `add_*` affect only the last environment, i.e. the update of $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E_n$ results in $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq (E_n; v : t := q)$.

The operations which change the length of the environment stack are of course `start_module` and `end_module`. The first one is called with an environment stack $E_1 \subseteq \cdots \subseteq E_n$, a name for the new module V, its parameters $V_1 : S_1 \ldots V_l : S_l$ and optionally output signature S_{out}. It verifies correctness of $S_1, \ldots S_l$ and S_{out} and the fact that names $V, V_1, \ldots, V_l$ are new and returns $E_1 \subseteq \cdots \subseteq E_n \subseteq E_{n+1}$, with $E_{n+1} = E_n; V_1 : S_1 \ldots V_l : S_l; \textsc{struct}\langle V, S_{out}\rangle$, where $\textsc{struct}\langle V, S_{out}\rangle$ is a special mark that is transparent for typing rules. The mark is only used by `end_module` as a placeholder separating functor arguments from structure components and keeping the optional output signature S_{out}.

The correctness of the new environment follows from the fact that E_{n+1} can be obtained by applying l times the rule (ADD).

The function `end_module` called with $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E_n \subseteq E_{n+1}$, returns $E_1 \subseteq \cdots \subseteq E_{n-1} \subseteq E'_n$ where $E'_n = E_n; V : S := m$ with S and m synthesized from the information contained in E_{n+1}. More precisely, if E_{n+1} is

$$E_n; V_1 : S_1 \ldots V_l : S_l; \textsc{struct}\langle V, S_{out}\rangle; v_1 : t_1 := q_1 \ldots v_k : t_k := q_k$$

m and S become respectively

$$\mathsf{Functor}[V_1 : S_1] \ldots \mathsf{Functor}[V_l : S_l]\ \mathsf{Struct}\ v_1 : t_1 := q_1 \ldots v_k : t_k := q_k\ \mathsf{End}$$

$$\text{and} \qquad \mathsf{Funsig}(V_1 : S_1) \ldots \mathsf{Funsig}(V_l : S_l)\ S_{out}$$

after checking that $E_n; V_1 : S_1 \ldots V_l : S_l \vdash \mathsf{Sig}\ v_1 : t_1 \ldots v_k : t_k\ \mathsf{End} <: S_{out}$. In case S_{out} is missing, S becomes

$$\mathsf{Funsig}(V_1 : S_1) \ldots \mathsf{Funsig}(V_l : S_l)\ \mathsf{Sig}\ v_1 : t_1 \ldots v_k : t_k\ \mathsf{End}$$

A similar mechanism is also used to implement interactive module types, through the functions `start_modtype` and `end_modtype` and a mark $\textsc{sig}\langle V\rangle$.

Non-interactive module and module type definitions are implemented through the functions `add_module` and `add_modtype`. Note however that by the time of writing these words, parsing of whole structures and signatures is not implemented. Therefore non-interactive module definitions are basically limited to renamings and functor applications, like

```
Coq < Module DoubleN:=Two N N.
Module DoubleN is defined
```

4.3 Avoiding Name Clashes

The possibility of having nested modules forces us to take care of name clashes. Consider the following nested module:

```
Module F.
  Definition T:=nat.
  Definition x:T:=0.
  Module N.
    Definition T:=bool.
    Definition y:=x.
    Definition z:T:=true.
  End N.
  Definition z:=N.y.
End F.
```

The problem here is the two occurrences of the name `T` in the scope of the inner module. The simplest solution of rejecting every piece of code having two equal short names would be too restrictive, because the names of module components are important and the user cannot change them arbitrarily.[3] The second solution consisting of hiding completely the previous definition with the new one is also impossible, because the old definition may be accessed indirectly, as shown by our example (the type of `y` is the first `T`). Using the Coq tradition to internally represent names together with the section path (`Top.F.T` for the first `T` and `Top.F.N.T` for the second) does not work well with functors (if `F` was a functor) and module renamings, like `Module G:=F`.

A correct solution to this problem requires noticing that the name of a component is at the same time a label which has to be visible from the outside of the module and a binder which binds the name used in the rest of the same module. If we distinguish the two names, the binder could get alpha converted (like all other binders), and the label clash is not a problem.

This is the solution proposed in the literature on modules. For example, in the convention of Harper and Lillibridge [HL94], the module type of F would look as follows (labels in `typewriter` font, binders in *italics*):

$$
\begin{array}{l}
\mathsf{Sig} \\
\quad \mathtt{T} \triangleright T : \star = \mathrm{nat} \\
\quad \mathtt{x} \triangleright x : T = 0 \\
\quad \mathtt{N} \triangleright N \ : \ \mathsf{Sig} \quad \mathtt{T} \triangleright T' : \star = \mathrm{bool} \quad \mathtt{y} \triangleright y : T = x \quad \mathtt{z} \triangleright z : T' = \mathrm{true} \quad \mathsf{End} \\
\quad \mathtt{z} \triangleright z : T = N.\mathtt{y} \\
\mathsf{End}
\end{array}
$$

To calculate the specification of $F.\mathtt{z}$ one has to use the substitution $\{T \mapsto F.\mathtt{T}$, $x \mapsto F.\mathtt{x}, N \mapsto F.\mathtt{N}\}$ on "$: T = N.\mathtt{y}$"[4], obtaining

$$F.\mathtt{z} : F.\mathtt{T} = F.\mathtt{N}.\mathtt{y}$$

[3] This strict policy is applied for functor parameters, since once a functor is defined, its parameters names are no longer important.

[4] To be precise, we should have written $\mathsf{Ty}(T)\mathsf{Eq}(N)$ here and similar specifications in the signatures, but they would become completely unreadable.

Leroy in [Ler94] and in the implementation of Ocaml attaches a unique number to every component name. The name (character string) plays the role of label and the name with the unique number (called identifier) plays the role of binder. In order to calculate the specification of a module component the substitution still has to list the identifiers of all signature components.

The solution we adopted consists in assigning a unique number not to every component name, but only to the leading Sig keyword. The signature of F is then represented internally as follows:

$$
\begin{array}{l}
\mathsf{Sig}\,\boxed{1} \\
\quad T : \star = \mathrm{nat} \\
\quad x : \boxed{1}.T = 0 \\
\quad N \ : \ \mathsf{Sig}\,\boxed{2} \quad T : \star = \mathrm{bool} \quad y : \boxed{1}.T = \boxed{1}.x \quad z : \boxed{2}.T = \mathrm{true} \quad \mathsf{End} \\
\quad z : \boxed{1}.T := \boxed{1}.N.y \\
\mathsf{End}
\end{array}
$$

Now, to calculate the specification of $\boxed{0}.F.z$ one only has to use the short substitution $\{\boxed{1} \mapsto \boxed{0}.F\}$, obtaining

$$\boxed{0}.F.z : \boxed{0}.F.T = \boxed{0}.F.N.y$$

In the above, $\boxed{0}$ is a unique identifier attached to the outer block, as the unique identifiers are not only attached to every Sig, but to every Struct and marks $\textsc{struct}\langle\cdot\rangle$ and $\textsc{sig}\langle\cdot\rangle$ as well. Moreover we assume that the first element of the environment is the mark $\textsc{struct}\langle\cdot\rangle$ with the self identifier $\boxed{0}$. In other words the environment stack is never empty, and the unique identifier of the current structure or signature is stored in the corresponding mark.

Of course the user never sees these unique identifiers. The name table mechanism, described in Sect. 5.5, efficiently separates user names from internal representation. It is also interesting to note that our unique identifiers bear a lot of similarities to the *self* identifier used in object oriented programming.

5 Modules Outside the Kernel

The most important challenge of the introduction of modules in Coq was not the implementation of the type-checking algorithm. Apart from managing the certified environment, Coq has extra-logical features like implicit arguments, an extensible pretty-printer and an extensible parser, databases for automated tactics, etc. They are controlled through commands which can very well be issued inside modules or module types. The management of these features in the presence of modules was technically much more difficult than the modifications in the kernel.

5.1 Overview

The generic mechanisms responsible for the management of extra-logical features of Coq are implemented in the directory **library**. They include:

1. Backtracking and state management
   Backtracking is implemented through the generic extensible mechanisms of *library objects* and *summaries*.
2. Theory file management
   Compiling theory files and loading them, management of dependencies and logical/physical paths.
3. Name visibility management
   Every Coq element has a unique long name, like `Coq.Datatypes.nat` for the type of natural numbers. The validity of its suffixes is stored in the name table, implemented in the file **nametab**.

The module operations on extra-logical features are implemented in a new file **declaremods**. Of course many other files were also modified. Before going into details, let us briefly present the generic mechanisms of Coq.

Backtracking and Library Objects. Since Coq is an interactive proof assistant, it is important to let users correct the mistakes they make. Therefore a backtracking mechanism is implemented in Coq (through the `Reset` command).

The commands issued by the user are pushed on a stack, and from time to time, the full state of the system is pushed there as well. To backtrack to a given point it is enough to "recall" the most recent copy of the state preceding the point and redo the commands following the state and preceding the point. See Fig. 1 for a graphical presentation (copies of the state are marked gray, and regular commands – white).

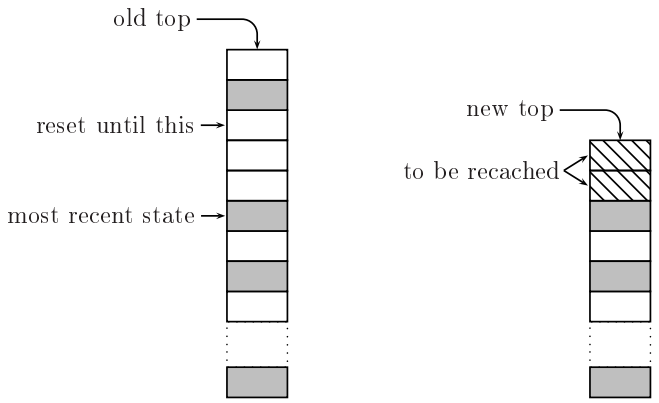

Fig. 1. Backtracking

In order to make this mechanism work properly and be easily extensible, the "commands on the stack" are given a common interface of so called *library objects*[5]. The object method `cache_function` implements the non-functional

[5] In fact for historical reasons, they are not Objective Caml objects.

part of command execution (usually table updates) and is called when pushing the object to the command stack, and while backtracking.

In previous versions of Coq the other uses of the generic object mechanism was to store and restore (to and from the disk) the extra-logical elements of compiled theory files and to partially control the discharge at the end of a section.

Summaries and State An important aspect of the implementation of backtracking is the possibility to periodically store the state of the system. Various Coq features require mutable information to work properly, for example the `Auto` tactic uses hint databases, the pretty-printing mechanism uses tables of pretty-printing rules, parsing — of parsing rules etc. Appropriate commands change the contents of these tables, so the contents of each table is a part of the state, and therefore should be stored. Of course copying the contents of each table would waste a lot of memory, so the data-structures used in Coq are all functional and their "copying" is just a pointer copying.

In order to facilitate the addition of new features in Coq, the tables able to store and restore their contents are given a common interface of so called *summaries*. The important remark here is that the global environment is also registered as a summary and managed through the generic mechanism.

5.2 Module Operations on Objects

To understand the issues we had to solve when putting together modules and extra-logical features of Coq, consider the following code which uses `Syntactic Definition` as an example.

```
Coq < Module M.
Coq <   Definition id := [A:Set][a:A]a.
Coq <   Syntactic Definition Id := (id ?).
Coq <   Check (Id 0).
(id nat (0))
     : nat
Coq < End M.
Coq < Check (M.Id true).
(M.id bool true)
     : bool
Coq < Module N:=M.
Coq < Check (N.Id (M.Id None)).
(N.id (A:Set)(option A) (M.id (A:Set)(option A) None))
     : (A:Set)(option A)
Coq < Print N.Id.
 Syntactic Definition N.Id = (N.id ?)
```

When we rebind the module `M` to a new name `N`, in the same time we create a new syntactic definition `N.Id`, which now is a shorthand for `(N.id ?)`. So the same mechanism of substitutions that exists in the kernel for module components had

to be implemented for library objects[6]. This mechanism naturally covers objects defined inside a functor body and used after the functor is applied to a module (see the example on the left). Moreover, it is possible to attach objects to a module type. Such objects can be used inside a functor body or attached to a module defined with that module type (on the right).

```
Coq < Module Type SIG.
Coq <   Parameter id : (A:Set)A->A.
Coq < End SIG.
Coq < Module F[X:SIG].
Coq <   Syntactic Definition FId :=
Coq <     (X.id ?).
Coq < End F.
Coq < Module M1:=(F M).
Coq < Module N1:=(F N).
Coq < Check (M1.FId 0).
(M.id nat (0))
     : nat
Coq < Check (N1.FId true).
(N.id bool true)
     : bool
```

```
Coq < Module Type OBJECTS.
Coq <   Parameter id : (A:Set)A->A.
Coq <   Syntactic Definition Id :=
Coq <     (id ?).
Coq < End OBJECTS.
Coq < Module G[X:OBJECTS].
Coq <   Definition id:=
Coq <     [A:Set][a:A](X.Id a).
Coq < End G.
Coq < Module X.
Coq <   Definition id:=
Coq <     [A:Set][a:A]a.
Coq < End X.
Coq < Module P : OBJECTS := (G X).
Coq < Check (P.Id 0).
(P.id nat (0))
     : nat
```

The interface of objects has been extended with a new method `subst_function`, and in order to unify the treatment of objects for simple modules, functors and module types, we introduced a concept of *object closure*. It corresponds to a list of inactive objects that reference module names invalid in the current context, that wait to be substituted once the objects are assigned to an actual module and activated. An object closure is remembered for simple modules because of renamings, for functors because of applications, and for module types because of their possible attachment to modules.

Technically, an object closure is built from a list of objects, a list of bound module identifiers (functor formal parameters), a *self* identifier and a delayed substitution for efficient handling of partially substituted object closures.

5.3 Interactive Modules Revisited

When the user starts a new interactive module, `Declaremods.start_module` is called, which itself calls the kernel function `start_module` in order to start the new module in the global environment. Then, the state of the system is pushed on the library stack together with a special start mark.

Inside the module, user commands are executed and corresponding objects are pushed on the stack and their `cache_function` methods are called.

When the module is ended, `Declaremods.end_module` is called. Its operations are the following.

[6] Moreover, the same substitution that is used in the kernel, is also used in the library.

1. End the module in the global environment,
2. Get objects and the state from the library stack,
3. Compute the object closure,
4. Prepare the module object,
5. Restore summaries except for the environment,
6. Push the module object on the stack.

The operation on the global environment is performed through the kernel function `end_module` (point 1). Then we recover all objects[7] pushed on the stack after the most recent start mark and the state of the system stored at the time of starting the module (point 2).

Next, the object closure of the module is computed (point 3), according to the module specification: If the module was given an output signature when it was started, we take the object closure of the signature. Otherwise the object closure is constructed from the objects recovered in point 2, the list of functor arguments given when the module was started, the *self* identifier of the module body and the empty substitution.

Now, if the ended module is not a functor, the objects from its object closure are substituted (using the method `subst_function`) with the substitution contained within the closure extended with the mapping of the *self* identifier to the name of the module. The object closure and the substituted objects are packaged into a module object (point 4), but before putting it on the command stack, all tables except for the global environment are restored (point 5) to the state recovered in point 2.

Only now, the module object is pushed on the stack and its caching method is called (point 6). It stores the object closure in a table and if the module was not a functor, calls a caching method for every substituted object. This way, if the module has submodules, their objects also get activated.

5.4 Non-interactive Modules

Although the operations described in the previous section may seem complicated, our goal was to maximally unify the object operations for interactive and non-interactive modules (possibly nested), for modules loaded from disk, and for module types, having in mind that all these should cooperate with the backtracking mechanism.

Indeed, the management of non-interactive modules is similar to the points 3, 4 and 6 of the `end_module` operation. Let us take

```
Module M:=F N
```

as an example, and suppose that the formal parameter of the functor `F` is `X`. The object closure of `M` is constructed from the object closure of `F`, by extending the

[7] Intermediate copies of the state stored on the stack inside the module are forgotten.

substitution contained within the closure with the mapping from `X` to `N`[8]. Now the module object is prepared in the same way that in point 4 in the previous section. Once it is pushed on the command stack, the sequence of events is like in point 6 above, with the only exception that the `cache_function` of the module object starts by updating the global environment using the kernel function `add_module`.

Theory files are treated like special modules and their object operations are performed by module functions. Interactive and non-interactive module type operations are very similar to those for modules, but simpler, since only the object closure is important and no objects are substituted.

5.5 Long Names and Suffix Visibility

Since version 7 of Coq, all named elements, like definitions, inductives etc. have unique long names available to the user. For example the long name of the inductive type `list` from the file **PolyList** of the standard library of Coq is `Coq.Lists.PolyList.list`. Of course suffixes of the long name are also valid references to that inductive type. Without modules, suffix validity patterns were very simple: either all suffixes were valid, or all suffixes of length strictly greater than one. In the recent version of Coq, definitions of nested modules, combined with the use of the `Import` command (declaring components of the module to be directly available at toplevel), may lead to arbitrary suffix validity patters.

Validity of suffixes is managed by the file **nametab**, whose function is to maintain the correspondence between user names and internal names. During the implementation of modules, the file **nametab** has been substantially cleaned up and its functionality extended.

6 Conclusions

We have described the implementation of ML-style modules in the Coq proof assistant. Our implementation carefully follows the idea of separating correctness-critical code (the kernel) from the rest of the system.

The modules have been successfully used in several examples, even though some features, like parsing and printing of structures and signatures and precise error messages are still not implemented. In combination with the extraction mechanism capable to produce Ocaml modules from Coq modules (implementated by Pierre Letouzey), Coq may become a very convenient tool to develop certified parametric data structures. For example, a modular formalization of certified dictionaries implemented by binary search trees was realized by Pierre Casteran (contribution `Bordeaux/dictionaries`).

Another interesting example of using modules is KRAKATOA [MPMU03], a Coq frontend to certify JAVA programs. The authors prove quite a few generic

[8] If we had a module renaming, `Module M:=N`, we would take the object closure of `N`, and if the output type was given, `Module M:SIG:=...`, we would take the object closure of `SIG`.

properties of Java classes abstractly in a functor, and a user working on a concrete Java program simply applies the functor to a module representing the classes of his program and can conveniently use the properties already instantiated to his particular setting.

An interesting possibility opened by addition of modules is the adaptation to Coq of the EML methodology of gradual construction of functional programs [ST97], which strongly relies on the use of functors. In a proof assistant framework, correctness of refining steps could be established formally, using interactive proofs.

Acknowledgments. The author would like to thank the anonymous referees for their extensive comments, suggestions and numerous corrections.

References

[Chr03] Jacek Chrząszcz. Modules in type theory with generative definitions. PhD thesis, to be defended, 2003.

[Coq] The Coq Proof Assistant. `http://coq.inria.fr/`.

[Cou97] Judicaël Courant. A Module Calculus for Pure Type Systems. In *Typed Lambda Calculi and Applications 97*, Lecture Notes in Computer Science, pages 112 – 128. Springer-Verlag, 1997.

[Cou98] Judicaël Courant. *Un calcul de modules pour les systèmes de types purs.* Thèse de doctorat, Ecole Normale Supérieure de Lyon, 1998.

[CPM90] 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.

[DM98] F. Durán and J. Meseguer. An extensible module algebra for Maude. In *In 2nd International Workshop on Rewriting Logic and its Applications (WRLA'98)*, volume 15 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[Fil00] J.-C. Filliâtre. Design of a proof assistant: Coq version 7. Unpublished, 2000.

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In POPL'94 [POP94], pages 123–137.

[Kam00] Florian Kammüller. Modular reasoning in isabelle. In *17th International Conference on Automated Deduction, CADE-17*, volume 1831 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2000.

[Ler94] Xavier Leroy. Manifest types, modules, and separate compilation. In POPL'94 [POP94], pages 109–122.

[MPMU03] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for JML/JAVA program certification. submitted, 2003.

[OS97] Sam Owre and Natarajan Shankar. The formal semantics of PVS. Technical Report SRI-CSL-97-2, Computer Science Laboratory, SRI International, Menlo Park, CA, August 1997.

[POP94] *Conference Record of the 21st Symposium on Principles of Programming Languages*, Portland, Oregon, 1994. ACM Press.

[ST97] D. Sannella and A. Tarlecki. Essential concepts of algebraic specification and program development. *Formal Aspects of Computing*, 9:229–269, 1997.