Design of a proof assistant: Coq version 7

Jean-Christophe Filliâtre

LRI, Université Paris-Sud, France
filliatr@lri.fr

Abstract

We present the design and implementation of the new version of the Coq proof assistant. The main novelty is the isolation of the critical part of the system, which consists in a type checker for the Calculus of Inductive Constructions. This kernel is now completely independent of the rest of the system and has been rewritten in a purely functional way. This leads to greater clarity and safety, without entailing efficiency. It also opens the way to the "bootstrap" of the system Coq, where the kernel will be certified using Coq itself.

1 Introduction

The system Coq [1] is a proof assistant. It provides mechanisms to write definitions, statements and to do formal proofs. It can be used interactively or as a batch checker. Its logical formalism is the Calculus of Inductive Constructions [4, 7] (CIC for short), a typed λ-calculus. Following Curry-Howard isomorphism, types are seen as propositions and terms as proofs. The proof engine builds terms by means of tactics written in ML. The safety of the system Coq is a consequence of the following two facts:

- The CIC is *consistent* (a proof of strong normalization is given in [8]);
- Once a proof is completed, it is type checked by a *trusted* piece of code implementing the typing rules of the CIC.

The typing rules of the CIC are given in appendix. The reader can check that there are not so numerous. Therefore, it should not be (too) complicated to implement them correctly. However, a proof assistant must also satisfy real world requirements, which include:

- an *interactive* system, which means some imperative state somewhere;
- an *undo* mechanism;
- an *extensible* system, where the user can write tactics in ML, add tables to be used by these tactics, etc;

- *efficiency*, which means that tactics should not spend all their time doing type checking.

Our objective when designing the new version of Coq was to conciliate a safe and independent kernel with all these requirements. The implementation of this new version of Coq is the work of several people in the Coq team. In particular, H. Herbelin and D. Delahaye improved several parts of the system outside of the kernel. However, this paper only concentrates on the kernel, whose design and partial re-implementation is the work of the author of this paper.

This paper is organized as follows. Section 2 presents the global architecture of the system, and how the kernel is taking place is that design. Section 3 presents the design of the kernel itself. Section 4 gives some implementation details.

2 The big picture

The first implementation of the Calculus of Constructions was realized by G. Huet and T. Coquand in 1984. The first public implementation of the system Coq, the version 4.10, was released in 1989. It was written in Caml, a dialect of ML written at INRIA. Inductive types were then added by C. Paulin in 1991. In 1993, D. de Rauglaudre ported the system Coq to Caml Light, a new implementation of Caml by X. Leroy and D. Doligez. In 1994–95, C. Murthy designed and implemented a completely new system, released as version 5.10. It was ported the next year to Objective Caml [2], the current implementation of Caml. All the following versions of Coq, up to the current version 6.3.1, are derivations of Murthy's system.

The new version of Coq, presented is this paper, is the version 7 of the system.

2.1 Architecture of versions 5.10 and 6.x

The main design of the current implementation of the system Coq (versions 5.10 and later) is due to C. Murthy. This implementation satisfies all the "real world requirements". In particular, the undo mechanism is implemented as follows. Any kind of *object* can be declared to the system, with methods[1] to perform its operation on the system and to load it from the disk. Global *tables* can be registered with methods to freeze and unfreeze their contents. Then the system keeps a stack of all operations and provides undo by unfreezing previous states and redoing operations.

In that context, terms of the CIC are just a particular kind of objects. But that design is *unsafe*: it is possible to delete, to modify or to add constants without any check—at the ML level only, not in the interactive system. Moreover, the type checker code is not clearly isolated. First, there is no data type for typing environments, since they are only pieces of the stack of operations. Second, the type checker does not come first, since its code relies on all the backtracking mechanism code (to access definitions in the environment).

[1] It is not implemented in an object oriented way, strictly speaking, since Caml did not support object oriented programming by the time of this implementation. However, C. Murthy's code is really simulating objects, using records.

2.2 An ideal kernel

The critical part of the system Coq consists in the type checker for the CIC. Indeed, once the proof of a lemma p is completed, the corresponding λ-term t is type checked and a new constant $p := t$ is added in the environment. To guarantee the consistency of the environment, the type checking has to be correct. Therefore, the kernel of the system should ideally provide an abstract type for well formed typing environments, together with functions to insert new elements that perform type checking. The signature of such a kernel would look like

```
type env
val empty : env
val push_var : env → identifier × constr → env
val add_constant : env → constant_declaration → env
...
val lookup_var : env → identifier → constr
...
```

Notice that the kernel does not have to provide a typing function: typing is done internally when insertion functions are called. Only the invariant of having a well formed typing environment is needed, and this can be maintained if the above type is abstract.

2.3 Coq version 7: a conciliation

In Coq version 7, we provide the best of both worlds: an ideal kernel on one side, as described in the previous section, and Murthy's general backtracking mechanism on the other side. We proceed as follows:

1. First, we write a purely functional type checker for the CIC;
2. Second, we introduce the backtracking mechanism, but CIC terms are no more a particular kind of objects;
3. Finally, we conciliate both of them by declaring a *reference* on the current typing environment as a global table.

The code realizing the connection looks like:

```
open Kernel

let r = ref empty

let push_var (id,c) = r := push_var !r (id,c)
let add_constant cd = r := add_constant !r d
...
let lookup_var id = lookup_var !r id
...
```

Part	Description	Modules	Lines of code
Lib	Utility libraries	15	1700
Kernel	Type checker for CIC	18	7800
Library	Undo mechanism and modules	14	2000
Pretyping	Translation from AST to terms	12	2000
Parsing	Parsing and pretty-printing	11	3700
Proofs	Proof engine	11	4000
Tactics	Basic tactics and tacticals	14	4000
Toplevel	Assembling layer	8	2900
	Total	103	28100

Figure 1: The main parts of Coq version 7

In order to have the typing environment correctly backtracked during undo operations, we declare the above reference `r` as a global table. This is particularly simple, since typing environments are implemented with a purely functional datatype: freezing and unfreezing the state of `r` is then immediate.

```
let freeze () = !r
let unfreeze f = r := f
let _ = declare_table "typing env" freeze unfreeze
```

For all the rest of the system, there is no visible difference with respect to previous implementations. Whatever the implementation of the typing environment is, an imperative environment is still provided with the following signature:

```
val push_var : identifier × constr → unit
val add_constant : constant_declaration → unit
...
val lookup_var : identifier → constr
...
```

And indeed, all the upper parts of the system were reestablished almost "as is". The main parts of Coq version 7 are given in Figure 1. For each, the number of Ocaml modules and the number of lines of code is given. These parts are presented in the order of link edition.

3 Design of the kernel

As we explained in the previous section, we succeeded in isolating the critical part of the system Coq, that is a type checker for the CIC. For better clarity and safety, it is implemented in a purely functional way, following the signature sketched in Section 2.2.

The typing rules for CIC are given in appendix. If we try to implement them in a direct (and naive) way, we immediately face a problem of circular dependency:

Adding a term in an environment requires to type check it (rule Push)

and

Type checking a term requires to lookup in the environment (rule Var)

With definitions and inductive types, the definition of the convertibility $=_c$ would also need access to the environment. Defining the environment operations, the reductions and the typing function in a mutually recursive way, therefore in a single module, would not be a good design. It would result in a single module with thousands lines of code. Instead, we choose the following three steps solution:

1. First, we define a data type `env` for typing environments. This type is "unsafe": it does no type checking on the elements inserted in the environments.

2. Then we define reduction functions and typing rules over that data type `env`. They usually make assumptions about the well formedness of the typing environments they take as arguments.

3. Finally, in a single module `Safe`, we define the type of typing environment `safe_env`, as equal to `env`, we write the type checker and we define the operations over type `safe_env` so that they perform type checking. The type `safe_env` is abstract (its identity to `env` is not exported in the interface of module `Safe`) and therefore we maintain the invariant that all the values of type `safe_env` are well formed.

The implementation of module `Safe` looks like

```
type safe_env = env

let typecheck e t =
  ...

let push_var e (id,c) =
  let (j,u) = typecheck e c in
  ...
  push_var (merge_constraints e u) (id,c)

let add_constant e d =
  ...
```

and it implements the signature given in Section 2.2.

The implementation of type `env` is purely functional. It uses efficient data structures, like balanced trees provided in the Ocaml standard library. It is kept abstract, so that its implementation can be easily changed. The main point is that it is reused in many other parts of the system, as well as many functions over it like reduction functions, where safety is not involved. Therefore, most of the operations of the system are efficient, yet using the same data type as the safe kernel.

4 Implementation details

The system Coq is written in Objective Caml [2], a dialect of ML developed at INRIA. This language provides functional, imperative and object oriented programming styles and is equipped with a powerful module calculus which offers true separate compilation. Two compilers are available: one producing portable bytecode, allowing fast development and easy debugging, and one producing fast executables in native code. The rich and portable standard library of Ocaml is heavily used. Coq's grammar is written using the syntactic tool camlp4 [5] which allows a dynamic grammar extensible at the user level.

The code of Coq is documented using a literate programming tool, ocamlweb [6]. A single human-readable document is produced from the code, which describes the design and all the interfaces (types, functions with their specifications, exceptions, etc.). This document is currently describing only the kernel and is 110 pages long—but several modules are not yet fully documented. A 30 pages long index is also automatically produced by ocamlweb, giving the definition and use points of all the identifiers of the code.

The total redesign and (partial) reimplementation of the system was realized in less than four months. The whole code is roughly 30000 lines long, 10000 being dedicated to the kernel and the undo mechanism. (See Figure 1 for details.)

5 Conclusion

We have presented the new implementation of the system Coq. It combines the efficiency and safety requirements in a completely new design, where the critical part of the system, a type checker for the CIC, is clearly isolated.

This type checker is now written is a purely functional way. It uses efficient functional data structures and is even slightly faster than the previous type checkers that were partly imperative. Being functional, this critical kernel is now easier to maintain and to reason about. One can now think of formally certifying it. Following the work of B. Barras [3], it could even be "bootstrapped" *i.e.* certified by Coq itself.

Anyhow, this new implementation is surely already the safest proof assistant.

References

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

[2] The Objective Caml language. `http://caml.inria.fr/`.

[3] B. Barras. *Auto-validation d'un système de preuves avec familles inductives.* Thèse de doctorat, Université Paris 7, November 1999.

[4] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.

[5] D. de Rauglaudre. The Camlp4 Pre Processor. `http://caml.inria.fr/camlp4/`.

[6] J.-C. Filliâtre and C. Marché. Ocamlweb, a literate programming tool for Objective Caml. `http://www.lri.fr/~filliatr/ocamlweb/`.

[7] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in LNCS, 1993. Also LIP research report 92-49.

[8] B. Werner. *Une Théorie des Constructions Inductives*. Thèse de doctorat, Université de Paris VII, Mai 1994.

Appendix: Typing rules for the CIC

We give here the typing rules for the Calculus of Constructions. Definitions and inductive types are omitted here, since they do not introduce any additional difficulty with respect to the issues discussed in this paper. Sorts, terms and typing environments obey the following grammar:

Sorts ($\mathcal{S}$)	s	::=	$\mathsf{Set} \mid \mathsf{Prop} \mid \mathsf{Type}(i)$
Terms	t	::=	$s \mid x \mid [x:t]t \mid (x:t)t \mid (t\ t)$
Environments	Γ	::=	$[] \mid \Gamma, x:t$

Typing rules are given in Figure 2. The convertibility between terms, written $=_c$, is here the $\alpha\beta$-equivalence. In the general case, it is the $\alpha\beta\delta\iota$-equivalence, δ being the unfolding of definitions and ι the reduction associated to inductive types.

Empty
$$\frac{}{\mathcal{WF}([])}$$

Push
$$\frac{\Gamma \vdash t : s \quad s \in \mathcal{S} \quad x \notin \Gamma}{\mathcal{WF}(\Gamma, x : t)}$$

Axioms
$$\frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}(i)} \qquad \frac{\mathcal{WF}(\Gamma)}{\Gamma \vdash \mathsf{Set} : \mathsf{Type}(i)} \qquad \frac{\mathcal{WF}(\Gamma) \quad i < j}{\Gamma \vdash \mathsf{Type}(i) : \mathsf{Type}(j)}$$

Var
$$\frac{\mathcal{WF}(\Gamma) \quad x : t \in \Gamma}{\Gamma \vdash x : t}$$

Prod
$$\frac{\Gamma \vdash t : s_1 \quad \Gamma, x : t \vdash u : s_2 \quad s_1 \in \{\mathsf{Prop}, \mathsf{Set}\} \quad s_2 \in \{\mathsf{Prop}, \mathsf{Set}\}}{\Gamma \vdash (x : t)u : s_2}$$

$$\frac{\Gamma \vdash t : \mathsf{Type}(i) \quad \Gamma, x : t \vdash u : \mathsf{Type}(j) \quad i \leq k \quad j \leq k}{\Gamma \vdash (x : t)u : \mathsf{Type}(k)}$$

Lam
$$\frac{\Gamma \vdash (x : u')t' : s \quad \Gamma, x : t' \vdash u : u'}{\Gamma \vdash [x : t]u : (x : t')u'}$$

App
$$\frac{\Gamma \vdash u : (x : t')u' \quad \Gamma \vdash t : t'}{\Gamma \vdash (u\ t) : u'[x \leftarrow t]}$$

Conv
$$\frac{t =_c u}{t \leq_c u} \qquad \frac{i \leq j}{\mathsf{Type}(i) \leq_c \mathsf{Type}(j)} \qquad \frac{}{\mathsf{Prop} \leq_c \mathsf{Type}(i)} \qquad \frac{t =_c u \quad m \leq_c n}{(x : t)m \leq_c (x : u)n}$$

$$\frac{\Gamma \vdash u' : s \quad \Gamma \vdash t : t' \quad t' \leq_c u'}{\Gamma \vdash t : u'}$$

Figure 2: Typing rules for the Calculus of Constructions