Coq version 7

Safer, Lighter, Faster

Jean-Christophe Filliâtre
(joint work with Hugo Herbelin)

SRI International, December 11, 2000

Overview

1. A brief history of Coq's development
2. Design of a proof assistant
3. A purely functional type-checker for the CIC
4. A few words about Objective Caml and Literate programming

History

- 1984 : first implementation by G. Huet and T.Coquand
  Written in CAML (INRIA, Formel team)
- 1989 : first release (version 4.10)
- 1991 : inductive types and extraction by C. Paulin
- 1993 : ported to Caml-Light (X. Leroy, D. Doligez)
- 1995 : new implementation by C. Murthy
- 1996 : ported to Objective Caml (native compiler)

Part I

Design of the Coq Proof Assistant

The main picture

- Coq is a type-checker for the Calculus of Inductive Constructions, a typed λ-calculus
- Propositions as types, proofs as λ-terms
- The proof engine builds terms, by means of tactics

Safety:

- the CIC is consistent (strong normalization property)
- the proof term is type-checked when the proof is completed

The ideal world

Ideally, the kernel is only composed of an abstract type for well-formed typing environments, together with functions to add elements (safely)

type *env*

val *empty* : *env*

val *push_var* : $env \to identifier \times constr \to env$

val *add_constant* : $env \to constant\_declaration \to env$

...

val *lookup_var* : $env \to identifier \to constr$

...

The real world

In the real world, we also want

- an interactive system, which means some imperative state somewhere
- undo operations
- an extensible system, where users can write tactics, add tables, etc.
- efficiency (tactics should not spend all the time doing type-checking)

Coq versions 5.10 and 6.*

Chet's implementation satisfies all the "real world requirements"

Objects can be registered with methods to perform their declarations and to load them from the disk. Global tables can be registered with methods to freeze and unfreeze their states.

Then the system keeps a stack of declarations and provides undo by mean of unfreeze + redo.

Terms of the CIC are just a particular kind of objects

But it is unsafe: you can delete / modify / add constants without any check (in ML only)

Coq version 7: a conciliation

A purely functional kernel on one side, with an abstract type for well-formed typing environment

Chet's generic undo mechanism on the other side

Both are conciliated by declaring a reference on the current typing environment as "a global table"

The "imperative" environment

open *Kernel*

let r = ref *empty*

let *push_var* (id, c) = r := *push_var* !r (id, c)

let *add_constant* d = r := *add_constant* !r d

...

let *lookup_var* v = *lookup_var* !r v

...

The imperative environment (interface)

val $\mathit{push\_var} : \mathit{identifier} \times \mathit{constr} \rightarrow \mathit{unit}$

val $\mathit{add\_constant} : \mathit{constant\_declaration} \rightarrow \mathit{unit}$

...

val $\mathit{lookup\_var} : \mathit{identifier} \rightarrow \mathit{constr}$

...

Benefits

- A functional kernel: simpler to reason about, simpler to extend, and a bootstrap is now possible
- *freeze* and *unfreeze* functions for the typing environment are immediate (the datatype is persistent since it is purely functional)

  let *freeze* () $= !r$

  let *unfreeze* $f = r := f$

Summary of the architecture

Lib	Utility libraries	15	1700
Kernel	Type-checker for CIC	18	7800
Library	Undo mechanism and modules	14	2000
Pretyping	Translation from *rawconstr* to *constr*	12	2000
Parsing	Translation from *AST* to *rawconstr* / PP	11	3700
Proofs	Proof engine	11	4000
Tactics	Basic tactics and tacticals	14	4000
Toplevel	Assembling layer	8	2900
	Total V7	103	28100
	Total V6	*119*	*42000*

Part II

Design of the Kernel

Type reconstruction

$$U_0, \Gamma \vdash M : T : S,\ U_1$$

A circular dependency

Adding a term in an environment requires to type-check it

Type-checking a term requires to lookup in an environment

Defining all the functions as mutually recursive in a single module is a bad solution

A 3 steps solution

1. First, define a datatype *unsafe_env* for typing environments
2. Then, define reduction functions, typing rules, etc. over that datatype; they usually make assumptions about the well-formedness of typing environments
3. Finally, in a single module "*Safe*", define the type of typing environments *env* as equal to *unsafe_env*, write the type-checker and define operations over type *env* so that they perform type-checking

The datatype for typing environments

- is purely functional
- mainly uses maps (dictionnaries) from the ocaml standard library
- is an abstract datatype, so that it is easy to change its implementation
- is reused is other parts of the system when safety is not involved, as well as functions over that datatype (reductions, etc.)

The main module of the kernel

type *env* = *unsafe_env*

let *typecheck* e t =

$\quad\ldots$

let *push_var* e (id, c) =

$\quad$ let (j, u) = *typecheck* e c in

$\quad\ldots$

$\quad$ *push_var* (*merge_constraints* e u) (id, c)

let *add_constant* e d =

$\quad\ldots$

Part III

Tools

Objective Caml (X. Leroy, D. Doligez)

- Functional and imperative features (strict), objects and classes
- A powerful module calculus, separate compilation
- Type inference, strong typing
- Bytecode and native-code compilers
- A rich standard library

  (sets, maps, buffers, format, bignums, str, Unix, ...)
- debugger, profiler, lex and yacc, C/Ocaml interface, ...

Camlp4 (D. de Rauglaudre)

- Pre-Processor-Pretty-Printer for Objective Caml
- provides high-level grammars

  (precedences, associativity, operators for lists, optional entries, ...)
- grammars are extensible dynamically

  (c.f. Coq command `Grammar`)

Literate programming

Coq modules interfaces are documented using ocamlweb, a literate programming tool for Objective Caml

A single human-readable document is produced, which describes the design and all the interfaces (types, functions, exceptions, ...)

Currently, this document is 110 pages long (but a lot of modules are not yet fully documented), plus a 30 pages index

111. States of the system. In that module, we provide functions to get and set the state of the whole system. Internally, it is done by freezing the states of both *Lib* and *Summary*. We provide functions to write and restore state to and from a given file.

val $intern\_state\ :\ string \rightarrow\ unit$

val $extern\_state\ :\ string \rightarrow\ unit$

type $state$

val $freeze\ :\ unit \rightarrow\ state$

val $unfreeze\ :\ state\ \rightarrow\ unit$

112. Rollback. $with\_heavy\_rollback\ f\ x$ applies f to x and restores the state of the whole system as it was before the evaluation if an exception is raised.

val $with\_heavy\_rollback\ :\ (\alpha\ \rightarrow\ \beta)\ \rightarrow\ \alpha\ \rightarrow\ \beta$