

The classical extraction module for Coq

Version 0.9

Alexandre Miquel

February 2009

Contents

1	Introduction	1
1.1	License	2
1.2	Availability	2
1.3	Credits	2
2	Usage	2
3	Commands	2
3.1	Classical Extraction " <i>file</i> " <i>qualid</i> ₁ <i>qualid</i> _n	2
3.2	Classical Extraction Reset	3
3.3	Classical Extraction Save " <i>file</i> "	3
3.4	Classical Extract <i>qualid</i> ₁ <i>qualid</i> _n	3
3.5	Classical Extract Witness <i>qualid</i> ₁ using <i>qualid</i> ₂	3
4	The extraction scheme	3
4.1	The basic mechanism	3
4.2	The default representation of pCIC data structures in λ_c	5
4.3	The <i>override</i> mechanism	5
4.4	Representing natural numbers using machine integers	6
4.5	Extracting (classical) axioms	7

1 Introduction

The classical extraction module for Coq enriches the Coq proof assistant [4] with a set of commands to extract programs from classical proofs built in the Coq environment. The target language of the extraction mechanism is the λ_c -calculus, an extension of λ -calculus with control primitives introduced by Jean-Louis Krivine to realize provable formulæ of classical logic. Extracted programs are stored in *lambda-calculus object* files (`.lco`) that can be inspected, loaded and evaluated using the `jivaro` toplevel that is distributed on the author's web page.

1.1 License

The classical extraction module for Coq is copyright © 2009 Alexandre Miquel.

The classical extraction module for Coq is free software: you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. You should have received a copy of the GNU Lesser General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

1.2 Availability

The classical extraction module for Coq is distributed on the author's web page at <http://perso.ens-lyon.fr/alexandre.miquel/kextraction/>

1.3 Credits

The classical extraction module has been designed and written by Alexandre Miquel (Alexandre.Miquel@ens-lyon.fr) using ideas developed in [1] and [3].

2 Usage

The classical extraction module for Coq comes as a bunch of ML files that can be statically or dynamically linked with Coq (version 8.2 or higher). The installation procedure is described in the file `INSTALL` in the archive.

When compiled as a dynamically linked module, the classical extraction module is loaded into the Coq environment using the command

```
Require ClassicalExtraction.
```

(typing `ClassicalExtraction` as one word). The classical extraction commands described in section 3 are then available in the Coq toplevel.

When compiled as a statically linked module, the classical extraction commands are immediately available, without the invocation above.

3 Commands

The classical extraction module provides the following commands:

3.1 Classical Extraction *"file" qualid₁ ... qualid_n*.

This command extracts λ_c -terms from the objects denoted by the qualified identifiers *qualid₁ ... qualid_n*, as well as from all the objects they may depend on in the current environment. The result is stored in a lambda-calculus object file *"file"*, with extension `.lco` added if necessary. This file can be inspected, loaded and executed in the `jivaro` toplevel.

The command `ClassicalExtraction "file" qualid1 ... qualidn` is equivalent to the sequence of commands

```

Classical Extraction Reset .
Classical Extract qualid1 ... qualidn .
Classical Extraction Save "file" .

```

3.2 Classical Extraction Reset .

This command clears the current list of λ_c -definitions. All the λ_c -definitions contained in the list before the command was executed are lost.

3.3 Classical Extraction Save "*file*" .

This command stores the current list of λ_c -definitions in a lambda-calculus object file "*file*", with extension `.lco` added if necessary. This file can be inspected, loaded and executed in the `jivaro` toplevel.

3.4 Classical Extract *qualid*₁ ... *qualid*_n .

This command extracts λ_c -terms from the objects denoted by the qualified identifiers *qualid*₁ ... *qualid*_n as well as from all the objects they may depend on in the current environment. The result is added to the list of λ_c -definitions already built by former extraction commands.

During extraction, global identifiers are turned into λ_c -symbols formed from the corresponding absolute section paths. For instance, the constructor `S` of type `nat → nat` is mapped to the λ_c -symbol `'Coq.Init.Datatypes.S'`.

3.5 Classical Extract Witness *qualid*₁ using *qualid*₂ .

This command extracts a function that computes a witness (possibly depending on parameters) from an existence proof in `Prop` given by *qualid*₁ using a decidability proof given by *qualid*₂. Both qualified identifiers *qualid*₁ and *qualid*₂ have to be declared in the current environment with types of the form:

$$\begin{aligned}
\textit{qualid}_1 & : \text{forall } \vec{x} : \vec{T}, \text{exists } y : U, P \vec{x} y \\
\textit{qualid}_2 & : \text{forall } \vec{x} : \vec{T}, \text{forall } y : U, \{P \vec{x} y\} + \{\sim P \vec{x} y\}
\end{aligned}$$

The extracted function is a λ_c -term t such that for every sequence of semantical objects $\vec{x} : \vec{T}$ and for every sequence of realizers \vec{u} that represent the objects \vec{x} , the term $t\vec{u}$ computes a realizer representing a semantical object $y : U$ such that $P \vec{x} y$.

The extracted function is added to the current list of λ_c -definitions under the name `path1%witness`, where `path1` is the λ_c -symbol associated to the qualified identifier *qualid*₁ during extraction.

4 The extraction scheme

4.1 The basic mechanism

From a technical point of view, the classical extraction mechanism is a projection of pCIC-terms¹ onto λ_c -terms where all types are collapsed onto a special constant

¹Remember that pCIC (the current version of the *Calculus of Inductive Constructions*) assumes that the sort `Set` is predicative while the initial presentation (called CIC) assumes that the sort `Set` is impredicative.

`.type` that is computationally meaningless. (Classical extraction is not intended to build interesting programs from types or type formers.) The translation $M \mapsto M^*$ from pCIC to λ_c is defined on the underlying PTS as follows:

$$\begin{aligned}
x^* &\equiv x \\
(\text{fun } (x : T) \Rightarrow M)^* &\equiv \lambda x M^* \\
(M N)^* &\equiv M^* N^* \\
(\text{forall } (x : T), U)^* &\equiv \text{.type} \\
(\text{Prop})^* &\equiv \text{.type} \\
(\text{Set})^* &\equiv \text{.type} \\
(\text{Type})^* &\equiv \text{.type}
\end{aligned}$$

Note that unlike constructive extraction (such as implemented in Coq by Pierre Letouzey [2]), classical extraction treats the terms of sort `Prop` (a.k.a. proof-terms) the same way as the terms of sort `Set` or `Type`. Sorts of pCIC objects are actually not considered during classical extraction.

Defined objects The Coq environment allows the user to introduce many constants to represent (co)inductively defined type families, constructors, definitions and axioms. Each pCIC-constant c is translated as a λ_c -constant with the same full name

$$c^* \equiv c$$

For instance the constructor `S` in the inductive definition of the type `nat` is translated as the λ_c -constant `Coq.Init.Datatypes.S` corresponding to the full name of this constructor in the Coq environment. As an exception, every constant I representing a (co)inductive type or type family is translated as the computationally meaningless λ_c -constant `.type`, like every type former:

$$I^* \equiv \text{.type}.$$

Pattern matching The case analysis of a pCIC-term M belonging to a (co)inductive type family I with constructors c_1, \dots, c_k of arities n_1, \dots, n_k (excluding parameters) is translated as follows

$$\left(\begin{array}{l} \text{match } M \text{ with} \\ | c_1 x_1 \cdots x_{n_1} \Rightarrow N_1 \\ \quad \vdots \\ | c_k x_1 \cdots x_{n_k} \Rightarrow N_k \\ \text{end} \end{array} \right)^* \equiv \begin{array}{l} I \% \text{case } M^* (\lambda x_1 \cdots \lambda x_{n_1} N_1^*) \\ \quad \vdots \\ (\lambda x_1 \cdots \lambda x_{n_k} N_k^*), \end{array}$$

where `I % case` is a special λ_c -constant associated to the type family I . Intuitively, this λ_c -constant implements the case analysis of the extracted term M^* against the terms $\lambda x_1 \cdots \lambda x_{n_i} N_i^*$ ($1 \leq i \leq k$) extracted from the branches of the `match` construct. Note that the elimination predicate of the `match` construct and the parameters of the corresponding (co)inductive definition are lost during extraction.

Since the realizability model [3] underlying classical extraction strongly relies on the fact that only `Prop` is impredicative, using classical extraction with flag `-impredicative-set` may not work.

Fixpoints and cofixpoints Fixpoints and cofixpoints are translated the very same way in the λ_c -calculus, regardless of problems of structural decreasing/production. Formally, this translation is defined by

$$\left(\begin{array}{l} (\text{co})\text{fix } f_1 : T_1 := M_1 \\ \vdots \\ \text{with } f_n : T_n := M_n \text{ for } f_i \end{array} \right)^* \equiv \begin{array}{l} .\text{fix}_{n.i} (\lambda f_1 \dots \lambda f_n M_1^*) \\ \vdots \\ (\lambda f_1 \dots \lambda f_n M_n^*) \end{array}$$

where $.\text{fix}_{n.i}$ (for $1 \leq i \leq n$) is a λ_c -constant implementing the fixpoint combinator defined by the reduction rule

$$.\text{fix}_{n.i} F_1 \dots F_n \succ F_i (. \text{fix}_{n.1} F_1 \dots F_n) \dots (. \text{fix}_{n.n} F_1 \dots F_n)$$

(Note the mutual dependency of the combinators $.\text{fix}_{n.1}, \dots, .\text{fix}_{n.n}$.) During the extraction process, these fixpoint combinators are automatically generated when needed, their definition being stored in the resulting object file.

4.2 The default representation of pCIC data structures in λ_c

The λ_c -representation of the data structures pertaining to a given (co)inductive type family I is defined by the implementation of the λ_c -constants c_1, \dots, c_k associated to the constructors of I and by the implementation of the λ_c -constant $I\%case$ that governs case analysis for the type family I . Since the objects of the family I are only accessed via the λ_c -constants c_1, \dots, c_k and $I\%case$ during the extraction process, changing the implementation of these constants is sufficient to change the λ_c -representation of all objects of the family I in the whole extracted program.

By default, (co)inductively defined data structures of pCIC are projected onto the λ_c -calculus using the standard impredicative encoding. Formally, if I is a (co)inductive type family depending on p parameters that is defined from k constructors c_1, \dots, c_k of arities n_1, \dots, n_k (excluding parameters), then, by default, every λ_c -constant c_i ($1 \leq i \leq k$) associated to the corresponding constructor is defined in the λ_c -calculus as an alias for the λ -term

$$c_i := \underbrace{\lambda y_1 \dots \lambda y_p}_{\text{parameters}} \underbrace{\lambda x_1 \dots \lambda x_{n_i}}_{\text{real args}} \underbrace{\lambda e_1 \dots \lambda e_k}_{\text{eliminators}} e_i x_1 \dots x_{n_i}$$

Case analysis is then implemented as the application of the matched value to all the branches of the `match` construct, so that the λ_c -constant $I\%case$ is simply defined as an alias for the identity function:

$$I\%case := \lambda z z$$

4.3 The override mechanism

For some data types, it is desirable to replace the default representation described above by a more efficient representation. The typical example is the type `nat` of natural numbers, whose default representation with the λ_c -constants

$$\begin{array}{ll} \text{Coq.Init.Datatypes.O} & := \lambda e_0 \lambda e_1 e_0 \\ \text{Coq.Init.Datatypes.S} & := \lambda x \lambda e_0 \lambda e_1 e_1 x \\ \text{Coq.Init.Datatypes.nat\%case} & := \lambda z z \end{array}$$

is even more inefficient than the traditional encoding à la Church.

For this purpose, the classical extraction module introduces a special set of λ_c -definitions to override some of the λ_c -definitions that would be normally produced during extraction. These λ_c -definitions are stored in an object file `override.lco` (in the `kextraction` directory) that is automatically loaded at the first invocation of a classical extraction command. Each time the extraction procedure produces a new λ_c -constant (representing a constructor, a case combinator, a definition, an axiom, etc.) the extractor first looks whether this constant is defined in the list of overriding definitions. If it is, the definition is taken from the list; otherwise, the constant is given its default definition using the mechanism described above.

Compiling overriding definitions The object file `override.lco` containing the list of overriding definitions is compiled from a source file `override.lc` in the `kextraction` directory (using the `jivaro toplevel`). Before loading the object file `override.lco`, the extractor looks for the source file `override.lc` in the `kextraction` directory. When this file exists and appears to be newer than the object file—or when the object file is missing—the extractor silently invokes the `jivaro toplevel` to (re)compile the list of overriding definitions before loading them.

4.4 Representing natural numbers using machine integers

The file `override.lc` accompanying the classical extraction module for Coq takes advantage of the presence of (unbounded) machine integers in the `jivaro toplevel` to redefine the representation of natural numbers as follows:

```

nat                :=  $\lambda n \lambda k k n$ 
Coq.Init.Datatypes.0 := nat 0
Coq.Init.Datatypes.S :=  $\lambda n n (\lambda n \text{int\_succ } n \text{ nat})$ 
Coq.Init.Datatypes.nat%case :=  $\lambda z \lambda e_0 \lambda e_1 z (\lambda n \text{int\_null } n e_0 (\text{int\_pred } n (\lambda p e_1 (\text{nat } p))))$ 

```

With these definitions, every natural number $n \in \mathbb{N}$ is represented in the extracted code as a *lazy integer*, that is, as a term t that passes the machine representation of the integer n to its first argument (formally: tk evaluates to kn for all terms k , still writing n the machine representation of the integer n).²

Although the (re)definitions above are sufficient to change the representation of natural numbers in the whole extracted code—thus changing the underlying space complexity—they do not change the fact that the code extracted from Coq functions still works with numerals on a unary basis, through the λ_c -constants implementing the constructors `O`, `S` and case analysis. To improve time complexity as well, this redefinition of natural numbers has to be accompanied with redefinitions of functions manipulating natural numbers in the extracted code. In this direction, the file `override.lc` redefines some of the λ_c -constants corresponding to arithmetic functions of the standard library of Coq, such as addition

```

Coq.Init.Peano.plus :=  $\lambda n \lambda m n (\lambda n m (\lambda m \text{int\_plus } n m \text{ nat}))$ 

```

and multiplication, as well as the bounded predecessor and subtraction functions.

²Note that we cannot represent natural numbers with machine integers directly: machine integers are pure data with no evaluation rule, so that we cannot put them into head position.

4.5 Extracting (classical) axioms

By default, each pCIC-axiom is extracted as a λ_c -constant (with the same full name) that is declared in the resulting object file as an undefined symbol (similarly to the special constant `.type` representing types).

Again, it is possible to override this behaviour by redefining the corresponding λ_c -constant in the file of overriding definitions. The file `override.lc` that comes with the classical extraction module for Coq contains a redefinition of the λ -constant that implements the axiom of excluded middle

```
classic : forall P : Prop, P ∨ ¬P
```

as well as of some other λ_c -constants associated to other results of the library for classical logic. (For instance, the λ_c -constant corresponding to Peirce's law is redefined as an alias for $\lambda P .cc$, where `.cc` is the constant for `call/cc`.)

In this way, it is possible to extract programs from classical proofs formalized in Coq using the library of classical logic in `Prop`, and to experience the fascinating computational behaviour of extracted programs working with continuations.

References

- [1] J.-L. Krivine. Realizability in classical logic. Unpublished lecture notes (available on the author's web page), 2005.
- [2] P. Letouzey. A new extraction for Coq. In H. Geuvers and F. Wiedijk, editors, *TYPES*, volume 2646 of *Lecture Notes in Computer Science*, pages 200–219. Springer, 2002.
- [3] A. Miquel. Classical program extraction in the calculus of constructions. In J. Duparc and T. A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 313–327. Springer, 2007.
- [4] The Coq Development Team (LogiCal Project). The Coq Proof Assistant Reference Manual – Version 8.1. Technical report, INRIA, 2006.