Typed Template Coq – Certified Meta-Programming in Coq
Abhishek Anand, Simon Boulier, Nicolas Tabareau, Matthieu Sozeau

To cite this version:

HAL Id: hal-01671948
https://inria.hal.science/hal-01671948
Submitted on 22 Dec 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
Typed Template Coq
Certified Meta-Programming in Coq

Abhishek Anand
Cornell University
Ithaca, NY, U.S.A.

Simon Boulier
Nicolas Tabareau
Gallinette Project-Team, Inria
Nantes, France

Matthieu Sozeau
Pi.R2 Project-Team, Inria and IRIF
Paris, France

Abstract
Template-Coq\(^1\) is a plugin for Coq, originally implemented by Malecha [7], which provides a reifier for Coq terms and global declarations, as represented in the Coq kernel, as well as a denotation command. Initially, it was developed for the purpose of writing functions on Coq’s AST in Gallina. Recently, its use was extended for the needs of the CertiCoq certified compiler project [2], which uses it as its front-end language and to derive parametricity properties [1], and the work of [5] on extracting Coq terms to a CBV \(\lambda\)-calculus. However, the syntax currently lacks semantics, be it typing semantics or operational semantics, which should reflect, as formal specifications in Coq, the semantics of Coq itself. This is an issue for CertiCoq where both a non-deterministic small step semantics and a deterministic call-by-value big step semantics had to be defined and preserved, without an “official” reference specification to refer to. Our hope with this work is to remedy this situation and provide a formal semantics of Coq’s implemented type theory, that can independently be refined and studied. By implementing a (partial) independent checker in Coq, we can also help formalize certified translations from Coq to Coq (Section 3).

ACM Reference Format:

1 Reification of Coq Terms
The reification of Coq terms follows Malecha’s original work, where the AST of Coq terms is defined in Coq and an ML plugin allows users to automatically construct reified versions of terms or constants with their references. Currently, the system refines all Coq terms, including fixpoints and cofixpoints, but the unquoting/reflection mechanism is partial. Apart from the use of OCAML’s native arrays and strings in the implementation of Coq, the reified syntax is identical to the constr datatype representing syntax in Coq’s OCAML code. The reifier can also crawl the environment for the dependencies of a given definition to output a program, which is the reification of a (minimal subset of) the environment and the definition’s reified body. A typical example is given in figure 1. As can be seen there, declarations of inductive types, with their constructors, and constants with their types and bodies, including lambda abstractions, pattern-matching (tCase) and fixpoint constructs (tFix) are all present.

\(^1\)https://template-coq.github.io/template-coq

Figure 1. Example reification of addition on natural numbers.

Modularity of reification. Template-Coq allows us to write many OCAML plugins, especially meta-theoretic translations of Gallina programs (§3), in Gallina itself. Writing such translations in Gallina is advantageous because it provides an opportunity to prove the correctness of such translations correct in Coq. A drawback, however, is that such Gallina programs run much slower than native OCAML plugins. Template-Coq now allows us to get the best of both worlds. It provides a mechanism to extract any Gallina program written over Template-Coq to OCAML and then run the result as an OCAML plugin. To achieve this, we have functorized our reification function, which is written in OCAML. The reification function can be instantiated differently to support both running inside Coq and after extraction.

For example, we can implement a minimal wrapper for CertiCoq’s compiler:

\textbf{Definition threeplusfour := \text{Nat.add}} 3 4.

\textbf{CertiCoq Compile threeplusfour.}

The new command goes through the extracted versions of the compiler to produce a file threeplusfour.c that can be compiled by CompCert or GCC and linked to a runtime library to compute the value of the threeplusfour definition.

2 Type Checking Coq in Coq
The verification of Coq in Coq has been a long term goal, tackled notably by BARRAS [3]. While we would like to ultimately prove metatheoretical results for the system described by our typing rules, our initial goals are slightly different in the sense that we focus on reflecting the actual implementation of Coq terms and providing a comfortable setting to work with these definitions. At the time of writing, we have specified:

1. \(\alpha\)-equivalence of terms and a “syntactic” cumulativity test that extends \(\alpha\)-equivalence.
We can now also use internal translation is given by two functions defined by induction on the syntax: \([\lambda] \) which translates terms and \([\lambda] \) which translates types. We define such functions in Coq, using the reified syntax of Template-Coq. Given a term \(t\) of type \(A\) we expect \([t]\) to be of type \([A]\). As term and types share the same syntax in Coq, both functions have signature \(\text{term} \rightarrow \text{term}\).

Translating Constants. A constant \(c\) is translated, using an association table, by another constant \(c'\). If \(c\) has a body \(t\) (i.e. \(c\) is a definition) then \(c'\) has \([t]\) as body. Otherwise \(c\) is an axiom, then \(c'\) is either a definition (and the axiom is justified by the translation) or another axiom.

Global Contexts. Translations are also parameterized by a global context which records the declared inductive types and the declared constants. It is mandatory to get, for instance, arities of inductive types, types of constructors, … For each inductive type, we need to specify how to translate its type, its constructors and its eliminator; and enrich accordingly the global context and the translation table.

Under Specified Constructors. Coq’s functions and applications are not fully typed. This can be problematic when we want to translate them. For instance, in the first translation of [4], we have:

\[
[\lambda(x : A). t] := ([\lambda(x : A)]. [t], \text{true})
\]

But in Coq the pair is typed and in fact we have:

\[
[\text{fun} (x:A) \Rightarrow t] := \langle \text{pair} (\forall x : A . ? ?) \text{ fun} (x : A) \Rightarrow t \rangle \text{ true}
\]

and we can not recover the type ? from the source term. Thankfully, Coq is designed so that type inference is decidable. That’s why, provided we also kept track of the local context (the types of lambda variables crossed), we can use the type inference algorithm of Section 2 to recover the missing information.

Putting all this together our translation functions have signature:

\[
\text{global_ctx} \rightarrow \text{trans_table} \rightarrow \text{local_ctx} \rightarrow \text{term} \rightarrow \text{term}
\]

3.2 Coq Plugin

Given the two translation functions, we can extract them to ML code. We wrote a plugin that uses those extracted functions to let the user use the translation directly in Coq in the spirit of [4]. The plugin implements three commands:

- **translate c** which derives the translation of a definition
- **implement ax : A** which adds the axiom \(ax : A\) to the global environment provided the user is able to inhabit the translated type \([A]\) (i.e. it switches to proof-mode).
- **implement Existing c** which allows to define the translation of a constant for which the translation can not be automatically derived (inductive type, constructor or axiom).

Parametricity Anand and Morrisett [1] have already formalized a variant of the parametricity translation using Template-Coq, providing stronger theorems for free on propositions. We are also looking at re-implementing Lasson’s plugin\(^2\) for unmodified parametricity and giving its formal correctness proof.

3.3 Future Work

Our long term goal is to leverage this translation+plug-in technique to extend the logical and computational power of Coq using, for instance, the forcing translation [6] or the weaning translation [5].

References


\(^2\)https://github.com/parametricity-coq/paramcoq