Extending MetaCoq Erasure: Extraction to Rust and Elm

Danil Annenkov[1], Mikkel Milo[2], Jakob Botsch Nielsen[1], and Bas Spitters[1]

[1] Concordium Blockchain Research Center, Aarhus University, Denmark
[2] Department of Computer Science, Aarhus University, Denmark

Introduction The Coq extraction feature is essential for producing executable code in conventional functional languages that can be integrated with existing components. Currently, Coq features extraction into OCaml, Haskell and Scheme [2]. Nowadays, there are many new important target languages that are not covered by the standard Coq extraction, moreover, the extraction procedure is written on OCaml and is not verified. An example of a domain that experiences rapid development and the increased importance of verification is the smart contract technology. We explored this direction and addressed the verification issue in our previous work [1], where we presented an extraction pipeline based on MetaCoq's verified erasure. We would like to present an extension of our previous work with (i) an improved extraction pipeline; (ii) a new target language Rust; (iii) a verified web application that can be extracted to Elm.

The pipeline Our pipeline is presented in Figure 1. The **green** items are our contributions (including the previous work) and the items marked with * are verified in Coq. Compared to [1], we have added a pre-processing step to the pipeline at the Template Coq level, featuring new transformations (inlining, case branch expansion). This step allows for extending the pipeline with new conversion-preserving transformations at a low cost. We extensively use the features provided by the MetaCoq project in our development [4]. We start by developing a program in Gallina that can use rich Coq types in the style of certified programming. In the case of smart contracts, we can use the machinery available in ConCert to test and verify properties of interacting smart contracts. We obtain a Template Coq representation (which closely follows the actual kernel representation) by *quoting* the term. We then apply a number of *certifying* transformations to this representation. This means that we produce a transformed term and a proof term, witnessing that the transformed term is equal to the original in the theory of Coq. The term in Template Coq is then translated to PCUIC.[1] We obtain an *erased* term by applying the verified erasure of MetaCoq and our erasure procedure for types. By $\lambda\Box^+$, we mean the untyped calculus of extracted programs (also part of MetaCoq), which we enrich with data structures required for type extraction. From the $\lambda\Box^+$ representation we can obtain code in one of the supported target languages. In this work, we focus our attention on Rust and Elm (see more about smart contract extraction in [1]). Our development is available on GitHub: `https://github.com/AU-COBRA/ConCert`

Extracting to Rust Rust is a mixed paradigm general-purpose programming language that features many of the same concepts as functional programming languages (pattern-matching, higher-order functions, a Hindley-Milner based type system, etc.). These features make Rust a suitable and relatively straightforward target for printing from $\lambda\Box^+$. However, Rust is a low-level language that gives a lot of control. This also provides some challenges:

- For recursive data structures, e.g. lists, it is necessary to use indirection, otherwise, the size of the data type would be infinite. We solve it by using Rust's borrowed references.
- Closures do not behave uniformly as in Coq (e.g. making an array of closures is not directly possible). Therefore, we allocate the closures and use trait objects to make the closures behave like closures in Coq.
- Rust is an unmanaged language without a garbage collector. In several cases, like closures and recursive data types, we need to manage memory manually.

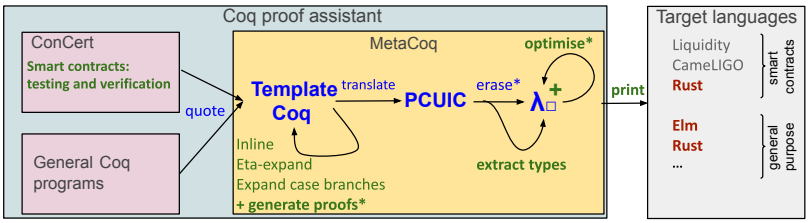

Figure 1: The pipeline

[1] Predicative calculus of cumulative inductive constructions, a "cleaned-up" version of the kernel representation [3]

Currently, we solve it by using a simple region-based memory allocation. In principle, a different off-the-shelf solution could be used.

- Partial applications are not supported in general, but can easily be emulated through closures. We generate both a curried and an uncurried version of a function.

As an example, we provide Rust code for the standard Coq function

```
map : forall A B : Type, (A → B) → list A → list B
```

```
impl<'a> Program {
...
fn map<A: Copy, B: Copy>(&'a self, f: &'a dyn Fn(A)→ B, l: &'a List<'a, A>)→ &'a List<'a, B> {
  match l {
    &List::Nil(_)⇒ { self.alloc(List::Nil(PhantomData))},
    &List::Cons(_, a, t)⇒ { self.alloc(List::Cons(PhantomData, hint_app(f)(a), self.map(f, t)))},
  }}
fn map__curried<A: Copy, B: Copy>(&'a self)→ &'a dyn Fn(&'a dyn Fn(A)→ B)→ &'a dyn Fn(&'a List<'a, A>)
    → &'a List<'a, B> {
  self.closure(move |f| { self.closure(move |l| { self.map(f, l)})})} }
```

We use the `Fn` trait for the type of closures and a `hint_app` wrapper to guide Rust's type checking. We generate two versions of the function: `map` is used if the fully applied case and `map_curried` – in case of partial applications.

Rust can also be used as a smart contract language. Our development supports the integration of the extracted code with the Concordium blockchain infrastructure.

Verified Elm web application Elm is a general-purpose functional language used for web development. It is based on an extended variant of the Hindley-Milner type system and has all core features of functional languages. As a demonstration, we develop an Elm-specific example in Coq: a simple web app inspired by Elm guide.[2] We define the application model in Coq in the following way.

```
Record StoredEntry := { seName : string; sePassword : string }.
Definition ValidStoredEntry := { entry : StoredEntry | entry.(seName) ≠ "" ∧8 ≤length entry.(sePassword) }.
Record Model :=
  { (** A list of valid entries such with unique user names *)
    users : {l : list ValidStoredEntry | NoDup (seNames l)};
    (** A list of errors after validation *)
    errors : list string;
    (** Current user input *)
    currentEntry : Entry }.
```

As one can see, users in our model are represented as a list of valid entries without duplication of names. We apply the usual certified programming style in Coq (using `Program`) in order to implement the logic of a web application that manipulates only valid data. That is, we implement a model update function `updateModel : StorageMsg → Model → Model * Cmd StorageMsg` that takes a user input of type `StorageMsg`, a model instance and produces a new model and (potentially) a command, e.g. a server call. In our examples, we do not use this functionality. We validate user input to construct valid data instances that are used by the application's logic. The `Model` type guarantees that only valid data can be stored.

We use our extraction pipeline to obtain a fully functional Elm web application (provided that the rendering functionality — a view — is written directly in Elm).[3] The generated application is well-typed in Elm, even though we have used dependent types extensively. The Elm architecture guarantees that the only way our model is updated is when users interact with the web application by calling the `updateModel` function. Therefore, we know that in the extracted code the model invariant is preserved.

References

[1] Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters. "Extracting Smart Contracts Tested and Verified in Coq". In: CPP 2021. Virtual, Denmark: Association for Computing Machinery, 2021, 105–121. ISBN: 9781450382991. DOI: 10.1145/3437992.3439934. URL: https://doi.org/10.1145/3437992.3439934.

[2] Pierre Letouzey. "Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq". PhD thesis. Université Paris-Sud, 2004.

[3] Matthieu Sozeau, Simon Boulier, Yannick Forster, Nicolas Tabareau, and Théo Winterhalter. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq". In: *POPL'2019*. 2019. DOI: 10.1145/3371076.

[4] Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. "The MetaCoq Project". In: *Journal of Automated Reasoning* (2020). ISSN: 1573-0670. DOI: 10.1007/s10817-019-09540-0.

[2] The Elm guide: https://guide.elm-lang.org/architecture/forms.html

[3] The demo of our extracted code is available at https://ellie-app.com/d2gtJ7WkB9Xa1