Code Extraction from Coq to ML-like languages

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

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

Abstract

The Coq code extraction feature produces executable code in conventional functional languages that can be integrated with existing components. Currently, Coq features extraction into OCaml, Haskell and Scheme. Many new target languages are not covered by the standard Coq extraction, such as languages for smart contracts and web development. Moreover, the extraction procedure is written in OCaml and is not verified. Implementing extraction in Coq itself allows for verification of the extraction process. We implement an extraction pipeline by extending the MetaCoq verified erasure with proof-generating passes and verified optimisations. We support several languages from the ML family in our pipeline: two languages for smart contracts (Liquidity and CameLIGO) and the Elm programming language. Our experience shows the pipeline can handle practical use cases and can be extended with more target languages including the ML family.

1 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 [4]. Nowadays, there are many new target languages that are not covered by the standard Coq extraction. Moreover, the extraction procedure is written in OCaml and is unverified. 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 verified erasure. We would like to present an extension of our previous work with (i) an improved extraction pipeline; (ii) a new target smart contract language CameLIGO; (iii) a verified web application that can be extracted to the Elm programming language.

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. We extensively use the features provided by the MetaCoq project in our development [6]. We start by developing a program in Gallina that can use rich Coq types in the style of certified programming (see e.g. [3]). 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[1] 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.[2] 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]).[3]

2 Extracting to Liquidity and CameLIGO

Blockchain and distributed ledger technology experience rapid development at the moment. Smart contracts are an important part of this technology. Put simply, smart contracts are programs running on top of a blockchain, which often control big amounts of cryptocurrency and cannot be changed after deployment. Unfortunately, many vulnerabilities have been discovered in smart contracts and this has led to huge financial losses (e.g. TheDAO, Parity's multi-signature wallet), making them important targets for formal verification. *Functional* smart contract languages has been adopted by several blockchains and taking a step towards making smart contracts safer. Smart contracts in such languages are (partial) state transition functions: `contract : msg * storage` $\rightarrow$ `operation list * option storage` that take some user input `msg`, user-defined contracts state `storage` and return a list of operations (e.g. transfers, call to other contracts) and a new storage value.

[1]The Template Coq representation basically reflects the actual Coq kernel
[2]Predicative calculus of cumulative inductive constructions, a "cleaned-up" version of the kernel representation [5]
[3]Our development is available on Github: `https://github.com/AU-COBRA/ConCert`

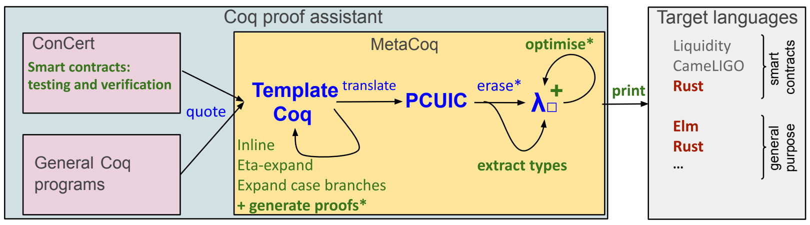

Figure 1: The pipeline

In ConCert, we are taking a further step by proving the properties of smart contracts that cannot be guaranteed by type checking. We consider two functional smart contract languages: Liquidity [2] and CameLIGO[4] for the Dune and Tezos blockchains. Both languages are inspired by OCaml and share many features with it. However, there are subtle differences both in the syntax and semantics making it impossible to use the standard extraction of Coq to OCaml directly. In our previous work [1] we presented extraction to Liquidity. Addressing some of the future work from this paper, we add the support for CameLIGO, since it is quite similar to Liquidity. Below, we summarise our experience with both languages, highlighting certain restrictions and differences.

Data types Data types are limited to non-recursive inductive types (also called *variant types*). Instead, the languages feature primitive container types like `list` and `map` along with operations on these types. Therefore, Coq functions on lists and finite maps must be replaced with "native" versions in the extracted code. We achieve this by providing a translation table that maps names of Coq functions to the corresponding language primitives. Note that extraction of recursive data types will produce code that will not compile.

Recursion Another limitation is that the support for recursive definitions is limited to tail recursion on a single argument. Therefore, for recursive functions of several arguments, the arguments need to be packed into a tuple. The same applies to data type constructors since the constructors take a tuple of arguments. Currently, the packing into tuples is done by the pretty-printer after verifying that constructors are fully applied. Recursive functions, which are not tail-recursive can be extracted, but will not be accepted by the compiler.

Type inference and polymorphism Both languages require type annotations in order to type-check a program. One source of ambiguity is the support of overloaded operations on numbers, which we solve this issue by providing a "prelude" for extracted contracts that specifies all required operations on numbers with explicit type annotations. CameLIGO requires even more type annotations: cases like an empty list, or a constructor `None` of the `option` must be annotated with the full type. We have implemented a general annotation mechanism that allows for attaching user-defined annotations to the AST nodes. We use this mechanism to annotate the extracted terms with types, which we then use to produce required type annotations at the pretty-printing stage.

Both languages have limited support of polymorphism (CameLIGO does not support user-defined polymorphic definitions at all). This is problematic for programs that use, for example, the `option` monad, which is an instance of the general `Monad` type class in Coq. We overcome this issue by inlining such definitions using the proof-generating step of our pipeline. Without careful inlining, the code might contain polymorphic definitions after extraction. We plan to use more sophisticated techniques such as monomorphisation to specialise the extracted code.

No unsafe type casts Both languages offer no opportunity to force the type checking to succeed. That means that certain code (that used OCaml's `Obj.magic` in the standard Coq extraction) might be not well-typed after extraction without a possibility to enforce the typing. Therefore, we do our best effort to produce typable code. Providing various passes (like inlining) helps to overcome some limitations, like the usage of functions with rank-2 types.

Example As an example, let us consider a simple counter contract with the state being just an integer number and accepting increment and decrement messages: `counter : msg` $\to$ `Z` $\to$ `option (list action *` `Z)`. The main functionality is given by the two functions `inc_counter` and `dec_counter`. We use Coq's *subset types* to encode some invariants of these functions. E.g. for `inc_counter` we encode in the type that the result of the increment is greater than the previous state. The original and the extracted code for the `inc_counter` function is shown in Figure 2.

The extraction procedure removes all "logical" parts (e.g. proofs of being positive) from the original

[4]https://ligolang.org/

```
Program Definition inc_counter (st : Z)
        (new_balance : {z : Z | 0 < z})
 : {new_st : Z | st < new_st} :=
  exist (st + proj1_sig new_balance) _.
  (* the proof is omitted *)
```

(a) Coq code

```
type storage = int
let exist a = a
let inc_counter (st : storage) (new_balance : int)
 = exist (addInt st ((fun x→ x) new_balance))
```

(b) Liquidity code

Figure 2: Extraction of `inc_counter`

Coq code. This code is called from the `counter` function (not shown here) which performs input validation and constructs the argument of type `{z : Z | 0 < z}` to call `inc_counter`. Since the only way of interacting with the contract is by calling `counter`, it is safe to execute `inc_counter` without additional input validation.

3 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, which makes it an easier extraction target, compared to Liquidity and CameLIGO. However, there is still one problem: no functionality similar to OCaml's `Obj.magic`. Therefore, we do our best effort to produce typable code.

As a demonstration, we develop an Elm-specific example in Coq: a simple web application inspired by the Elm guide.[5]. We define the application model in Coq in the following way.

```
Record StoredEntry := { seName : string; sePassword : string }.
Definition ValidStoredEntry := { entry : StoredEntry | entry.(name) ≠ "" ∧8 ≤String.length p }.
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 }.
```

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.[6] 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).[7] 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.

4 Conclusions

We have presented an extraction pipeline implemented completely in the Coq proof assistant. This makes it suitable for providing strong correctness guarantees for the extraction process. The extraction relies on the MetaCoq verified erasure procedure, which we extend with data structures required for extraction to our target languages. The pipeline addresses new challenges originating from the target languages we have considered and can be extended with new transformations, if required. The proof-generating pass allows, for example, to inline and specialise some definitions which might not be typable after extraction, since our targets do not feature unsafe type casts, like OCaml's `Obj.magic`. New transformations can be added to this pass without much effort (no need to modify the proof-generating part), provided that they produce terms definitionally equal in the theory of Coq. More sophisticated transformations, such as partial evaluation, can be applied without breaking the definitional equality. Our pipeline can accommodate the techniques outlined in [7], which could be implemented in Coq directly (instead of a plugin) using the meta-programming facilities of MetaCoq.

As a demonstration of our approach, we have implemented extraction to Liquidity, CameLIGO, Elm and a subset of Rust. We believe that many other languages from the ML family can be added as targets to the development by implementing a pretty-printer in Coq.

[5]The Elm guide: https://guide.elm-lang.org/architecture/forms.html
[6]Commands can be e.g. calls to the server. We do not use this functionality in our example.
[7]The demo of our extracted code is available at https://ellie-app.com/d2gtJ7WkB9Xa1

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] Çagdas Bozman, Mohamed Iguernlala, Michael Laporte, Fabrice Le Fessant, and Alain Mebsout. “Liquidity: OCaml pour la Blockchain”. In: *JFLA18*. 2018.

[3] Adam Chlipala. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. ISBN: 9780262026659.

[4] Pierre Letouzey. “Programmation fonctionnelle certifiée – L’extraction de programmes dans l’assistant Coq”. PhD thesis. Université Paris-Sud, 2004.

[5] 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.

[6] 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.

[7] Akira Tanaka. “Coq to C Translation with Partial Evaluation”. In: *Proceedings of the 2021 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. PEPM 2021. 2021, 14–31. ISBN: 9781450383059. DOI: 10.1145/3441296.3441394.