Deep and Shallow Embeddings in Coq *

Danil Annenkov and Bas Spitters

Aarhus University

Abstract. We demonstrate how deep and shallow embeddings of functional programs can coexist in the Coq proof assistant using meta-programming facilities of MetaCoq. While deep embeddings are useful for proving meta-theoretical properties of a language, shallow embeddings allow for reasoning about the functional correctness of programs.

Motivation. Functional languages are becoming increasingly popular in software development. Moreover, there is a demand for languages with well-understood semantics arising from the concept of "smart contracts" — computer programs running on blockchains. A number of blockchain implementations have already adopted certain variations of functional languages as an underlying smart contract language. These languages range from minimalistic and low-level (Simplicity, Michelson) to fully-featured OCaml- and Haskell-like languages (Liquidity, Plutus). There are several formalisations of these languages in proof assistants, mostly covering meta-theory[1][2] with the notable exception of Scilla [2], which features verification of particular smart contracts translated by hand to Coq. One of the motivations for the present work is formalisation of the Concordium Oak functional smart contract language. Although it seems obvious to use the functional language of a proof assistant to represent constructions of a language we would like to reason about, there are not that many tools available for that purpose. We propose to use the meta-programming facilities of MetaCoq [1] to develop such a tool in a principled way.

Embedding of Functional Languages. There are various ways of reasoning about properties of a programming language in a proof assistant. First, let us split the properties in two groups: meta-theoretical properties (properties of a language itself) and properties of programs written in the language. Since we limit ourselves to a functional programming language, it is reasonable to assume that we can reuse the programming language of a proof assistant to express functional programs and reason about their properties. A somewhat similar approach is taken by the authors of the hs-to-coq library [3], which translates total Haskell programs to Coq by means of source-to-source transformation. Unfortunately, in this case it is impossible to reason about correctness of the translation.

An alternative is to directly interpret the syntax into Coq functions in NbE style [4]. In the presence of inductive types this requires encodings that complicate reasoning about the properties of the embedded programs. However, we do not rule out this approach, since it can be quite useful for proving meta-theoretical properties.

We would like: (1) to have a way of translating programs written in our functional language into a Coq function that looks close to the original (the shallow embedding); (2) to have access to the AST of the language for meta-theoretical reasoning (the deep embedding), and (3) to make an explicit connection between the semantics of the language and its representation in Coq in the form of a soundness theorem.

*This work is supported by the Concordium Blockchain Research Center, Aarhus University, Denmark.

[1] Michelson in Coq: https://framagit.org/rafoo/michelson-coq/

[2] Plutus meta-theory in Agda: https://github.com/input-output-hk/plutus-metatheory.

Our approach. MetaCoq makes it possible to connect the two ways of reasoning about functional programs. We implement a functional language that corresponds to a "core" subset of an average functional language with a System F type system, algebraic data types and general recursion. We define a translation from this language to the syntax of MetaCoq. In contrast with the source-to-source translation, our translation is a Coq function from the AST of our language to the AST of MetaCoq. This makes it possible to reason within Coq about the translation and formalize the required meta-theory for the language.

MetaCoq allows us to convert an AST represented as an inductive type into a Coq term. Thus, starting with the syntax of a program in our functional language, through a series of translations we produce a MetaCoq AST, which is then interpreted into a program in Coq's Gallina language. Let us consider the following example:

```
Definition plus_syn : expr := [| fix "plus" (x : Nat) : Nat → Nat :=
                                  case x : Nat return Nat → Nat of
                                  | Z → \y : Nat → y
                                  | Suc y → \z : Nat → Suc ("plus" y z) |].
(* Unquoting the syntax into a Coq term *)
Make Definition my_plus := Eval compute in (expr_to_term (indexify plus_syn)).
Lemma my_plus_correct n m : my_plus n m = n + m.
Proof. induction n;simpl;auto. Qed.
(* Computing with the interpreter *)
Compute (eval 10 enEmpty [| {plus_syn} 1 1 |]).
(* = Ok (vConstr "nat" "Suc" [vConstr "nat" "Suc" [vConstr "nat" "Z" []]]) *)
```

The term `plus_syn` defines an AST of a program in our functional language using the extension of the notation mechanism called *custom entries*. After the application of `indexify` (which converts variable names into De Bruijn indices) and `expr_to_term` (which translates expressions to the terms of MetaCoq) we use `Make Definition my_plus := ...` of MetaCoq to produce a Coq program. The `my_plus_correct` lemma shows that the `my_plus` corresponds to the standard definition of addition. This example already allows us to verify correctness of certain functions by proving them equivalent to functions from the standard library of Coq.

The semantics of our functional language is defined as a definitional interpreter in Coq. As the example above shows, this allows to compute with the interpreter on the deeply embedded representation. The interpreter is implemented in an environment-passing style and works both with named and nameless representations of variables. To be able to interpret general fixpoints we evaluate fixpoints applications in the environment extended with the closure corresponding to the recursive call. Due to the potential non-termination, we define our interpreter using a *fuel idiom*: by structural recursion on an additional argument (a natural number).

Since MetaCoq aims to also formalise the meta-theory of Coq we use this development to show that the semantics of our functional language agrees with its translation to MetaCoq (on terminating programs) and our interpreter is sound with respect to the embedding. This paves a way for principled embeddings of functional languages to Coq.

References

[1] Abhishek Anand, Simon Boulier, Cyril Cohen, Matthieu Sozeau, and Nicolas Tabareau. Towards Certified Meta-Programming with Typed Template-Coq. In *ITP18*, volume 10895 of *LNCS*, pages 20–39, 2018.

[2] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a Smart Contract Intermediate-Level LAnguage. *CoRR*, abs/1801.00687, 2018.

[3] Antal Spector-Zabusky, Joachim Breitner, Christine Rizkallah, and Stephanie Weirich. Total Haskell is reasonable Coq. CPP18, pages 14–27. ACM, 2018.

[4] PawełWieczorek and Dariusz Biernacki. A Coq Formalization of Normalization by Evaluation for Martin-LöF Type Theory. CPP 2018, pages 266–279. ACM, 2018.