Towards a Smart Contract Verification Framework in Coq *

Danil Annenkov and Bas Spitters

Aarhus University

Abstract

We propose a novel way of embedding functional smart contract languages into the Coq proof assistant using meta-programming techniques. Our framework allows for developing the meta-theory of smart contract languages using the deep embedding and provides a convenient way for reasoning about concrete contracts using the shallow embedding. The proposed approach allows to make a connection between the two embeddings in a form of a soundness theorem. As an instance of our approach we develop an embedding of the Oak smart contract language in Coq and verify several important properties of a crowdfunding contract. The developed techniques are applicable to all functional smart contract languages.

1 Introduction

The concept of blockchain-based smart contracts has evolved in several ways since its appearance. Starting from the restricted and non Turing complete Bitcoin script[1] designed to validate transactions, the idea of smart contracts expanded to fully featured languages like Solidity running on the Ethereum Virtual Machine (EVM).[2] Recent research on the smart contract verification discovered the presence of multiple vulnerabilities in many smart contract written in Solidity [3, 6]. Several times the issues in smart contract implementations resulted in huge financial losses (for example, the DAO contract and the Parity multi-sig wallet on Ethereum). The setup for smart contracts is quite unique: once deployed, they cannot be changed and any small mistake in the contract logic may lead to serious financial consequences. This shows not only the importance of formal verification of smart contracts, but also the importance of principled programming language design. Next generation smart contract languages tends to employ the functional programming paradigm. 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 [5], Michelson[3]) to fully-featured OCaml- and Haskell-like languages (Liquidity [2], Plutus[4]). There is a very good reason for this tendency. Statically typed functional programming languages can rule out many mistakes. Moreover, due to the absence (or more precise control) of side effects programs in functional languages behave as mathematical functions that makes reasoning about them easier. However, one cannot hope to perform only stateless computations: the state is inherent for blockchains. One way to approach this is to limit the ways of changing the state. While Solidity allows arbitrary state modifications at any point of execution, many modern smart contract languages represent smart contract execution as a function from a current state to a new state. This functional nature of modern smart contract languages makes them well-suited for formal reasoning.

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

[1] Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf

[2] Ethereum's white paper: https://github.com/ethereum/wiki/wiki/White-Paper

[3] https://www.michelson-lang.com/

[4] https://cardanodocs.com/technical/plutus/introduction/

The Ethereum Virtual Machine and the Solidity smart contract language remains one of the most used platforms for writing smart contacts. Due to the permissiveness of the underlying execution model and complexity of the language verification in this setting is quite challenging. On the other hand, many new generation languages such as Oak,[5] Liquidity and Scilla, offer a different execution model and a type system allowing to rule out many errors by means of type checking. Of course, many important properties are not possible to capture even with powerful type systems of functional smart contract languages. For that reason, to provide even higher guarantees, such as functional correctness, one has to resort to stronger type systems/logics for reasoning about programs and employ deductive verification techniques. Among various tools for that purpose proof assistants provide a versatile solution for that problem.

Proof assistants, or interactive theorem provers are tools that allow for stating and proving theorems by interacting with users. Proof assistants often offer some degree of proof automation by implementing decision and semi-decision procedures, or interacting with automated theorem provers (SAT and SMT solvers). Some proof assistants allow for writing user-defined automation scripts, or write extensions using a plug-in system. This is especially important, since many problems in the verification of programming languages are undecidable and providing users with a convenient way of interactive proving while retaining a possibility to do automatic reasoning makes proof assistants very flexible tools for verification of smart contracts.

Existing formalisations of functional smart contract languages mostly focus on meta-theory[6] with the exception of Scilla [7], which features verification of particular smart contracts in Coq by means of shallow embedding by hand. Simplicity [5] is a low-level combinator based functional language and its formalisation allows for translating from deep to shallow embeddings for purposes of meta-theoretic reasoning. None of these developments combine deep and shallow embeddings for a *high-level* functional smart contract language in one framework or provide an automatic way of converting smart contracts to Coq programs for convenient verification of concrete smart contract. We are making a step towards this direction by allowing for deep and shallow embeddings to coexist and interact in Coq.

The contributions of this paper are the following: (1) we develop an approach allowing for developing in one framework the meta-theory of smart contract languages and convenient reasoning about concrete contracts; (2) we combine deep and shallow embedings using the metaprogramming facilities of the MetaCoq plug-in [1]; (3) as an instance of our approach we define the syntax and semantics of the Oak language (the deep embedding) and the corresponding translation of Oak programs into Coq functions (the shallow embedding); (4) we prove properties of a crowdfunding contract given as a deep embedding (abstract syntax tree) of an Oak program. We discuss details of our approach in Section 2 and provide an example of a crowdfunding contract in Section 3.

2 Our approach

There are various ways of reasoning about properties of a functional 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 are focused on functional smart contract languages and many proof assistants come with a built-in functional language, it is reasonable to assume that we can reuse the programming

[5] The Oak language is an ML-style functional smart contract language with Elm-like syntax. Oak is currently under development at the Concordium foundation.

[6] Michelson meta-theory: https://gitlab.com/nomadic-labs/mi-cho-coq/
Plutus core meta-theory: https://github.com/input-output-hk/plutus/tree/master/metatheory
Simplicity meta-theory: https://github.com/ElementsProject/simplicity/tree/master/Coq.

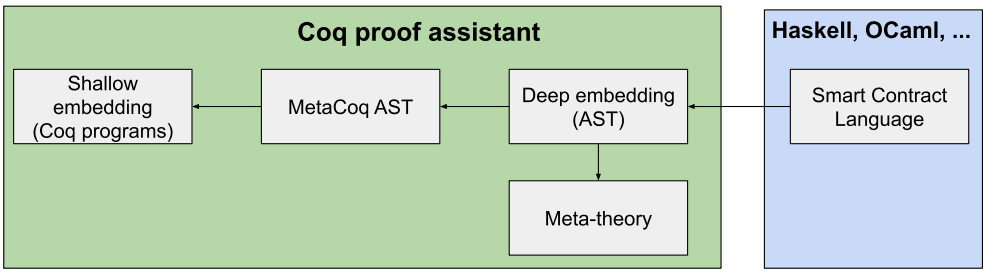

Figure 1: The structure of the framework

language of a proof assistant to express smart contracts and reason about their properties. A somewhat similar approach is taken by the authors of the hs-to-coq library [8], which translates total Haskell programs to Coq by means of source-to-source transformation. Unfortunately, in this case it is impossible to reason about the correctness of the translation.

We would like to have two representations of functional programs within the same framework: a deep embedding in the form of an abstract syntax tree (AST), and a shallow embedding as a Coq function. While the deep embedding is suitable for meta-theoretical reasoning, the shallow embedding is convenient for proving properties of concrete programs. We use the meta-programming facilities of the MetaCoq plug-in [1] to connect the two ways of reasoning about functional programs.

The overview of the structure of the framework is given in Figure 1. As opposed to source-to-source translations in the style hs-to-coq[8] and coq-of-ocaml[7] we would like for all the non-trivial transformations to happen in Coq. This makes it possible to reason within Coq about the translation and formalize the required meta-theory for the language. That is, we start with an AST of a program in a smart contract language implemented in Haskell, OCaml or some other language, then we generate an AST represented using the constructors of the corresponding inductive type in Coq (deep embedding) by printing the desugared AST of the program. By printing we mean a recursive procedure of converting the AST into a string consisting of the constructors of our Coq representation. The main idea is that this procedure should be as simple as possible and does not involve any non-trivial manipulations, since it will be a part of a trusted code base. If any non-trivial transformations are required, they should happen within the Coq implementation.

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 transformations we produce a MetaCoq AST, which is then interpreted into a program in Coq's Gallina language (shallow embedding). The transformations include conversion from the named to the nameless representation (if required) and translation into the MetaCoq AST. The deep embedding also serves as input for developing meta-theory of the smart contract language.

As an instance of our approach we develop an embedding of the Oak smart contract language to Coq.[8] The semantics of Oak is given as a definitional interpreter. This gives us an executable semantics for the language. 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

[7] The coq-of-ocaml github page: https://github.com/clarus/coq-of-ocaml
[8] Our Coq development https://github.com/annenkov/FMBC19-artefact/, including examples from Section 3

```
(* Defining AST using customised notations     *)

(* Brackets [\ \] delimit the scope of  global *)
(* definitions and [| |] the scope of programs *)

Definition state_syn : global_dec :=
  [\ record State :=
     { balance   : Money ;
       donations : Map;
       owner     : Money;
       deadline  : Nat;
       done      : Bool;
       goal      : Money } \].

Make Inductive (trans_global_dec state_syn).

Definition action_syn : global_dec:=
  [\ data Action :=
        Transfer : Address → Money → Action
      | Empty    : Action; \].

Make Inductive (trans_global_dec action_syn).

Definition result_syn : global_dec :=
  [\ data Result :=
       Res   : State → Action → Result
     | Error : Result; \].

Make Inductive (trans_global_dec result_syn).

Definition msg_syn : global_dec :=
  [\ data Msg :=
       Donate   : Msg
     | GetFunds : Msg
     | Claim    : Msg; \].

Make Inductive (trans_global_dec msg_syn).
```

```
Definition crowdfunding : expr :=
[| \c : Ctx ⇒ \s : State ⇒ \m : Msg ⇒
   let bal : Money := balance s in
   let now : Nat := cur_time c in
   let tx_amount : Money := amount c in
   let sender : Address := ctx_from c in
   let own : Address := owner s in
   let accs : Map := donations s in
   case m : Msg return Result of
      | GetFunds →
      if (own == sender) && (deadline s < now) && (goal s ≤bal)
      then Res (mkState 0 accs own (deadline s) True (goal s))
               (Transfer bal sender)
       else Error : Result
     | Donate → if now ≤deadline s then
       (case (mfind accs sender) : Maybe return Result of
         | Just v →
           let newmap : Map := madd sender (v + tx_amount) accs in
           Res (mkState (tx_amount + bal) newmap own
                        (deadline s) (done s) (goal s)) Empty
         | Nothing →
           let newmap : Map := madd sender tx_amount accs in
           Res (mkState (tx_amount + bal) newmap own
                        (deadline s) (done s) (goal s)) Empty)
       else Error : Result
     | Claim →
       if (deadline s < now) && (bal < goal s) && (  done s) then
       (case (mfind accs sender) : Maybe return Result of
        | Just v → let newmap : Map := madd sender 0 accs in
            Res (mkState (bal−v) newmap own
                 (deadline s) (done s) (goal s)) (Transfer v sender)
        | Nothing → Error)
       else Error : Result
   |].

Make Definition entry :=
    Eval compute in (expr_to_term  (indexify crowdfunding)).
```

Figure 2: The crowdfunding contract

natural number).

Since the development of the meta-theory of Coq itself is one of the aims of the MetaCoq we can 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. We compare the results of evaluation of Oak expressions with the weak head call-by-value evaluation relation of MetaCoq up to appropriate conversion of values. Currently, the full formalisation of this proof is under development. Being able to relate the semantics of Oak to the semantics of Coq through Coq's meta-theory formalisation gives stronger guarantees that our shallow embedding reflects the actual behaviour of Oak programs. The described approach provides a more principled way of embedding functional language, in contrast to the source-to-source based approaches.

3 The crowdfunding contract

As an example of our approach we consider verification of some properties of a crowdfunding contract (Figure 2). Such a contract allows arbitrary users to donate money within a deadline. If the crowdfunding goal is reached, the owner can withdraw the total amount from the account after the deadline has passed. Also, users can withdraw their donations after the deadline if the goal has not been reached. Contracts like this are standard applications of smart contracts and appear in a number of tutorials.[9] We follow the example of Scilla [7] and adopt (with minor variations) a crowdfunding contract as a good instance to demonstrate verification techniques.

We extensively use a new feature of Coq called "custom entries" to provide a convenient

[9] The idea of a crowdfunding contract appears under different names: crowdsale, kickstarter-like contract, ICO contract, ect. Many Ethereum-related resources contain variations of this idea in tutorials (including Solidity and Vyper documentation). A simplified verision of a crowdfunding contract is also available for Liquidity: https://github.com/postables/Tezos-Developer-Resources/blob/master/Examples/Crowdfund/Basic.ml

notation for our deep embedding.[10] The program texts in Figure 2 written inside the special brackets [\ ... \] and [| ... |] are parsed according to the custom notation rules. For example, without using notations the definition of `action_syn` looks as follows:

```
gdInd Action 0 [("Transfer", [(nAnon, tyInd "nat"); (nAnon, tyInd "nat")]);("Empty", [])] false.
```

This AST otherwise would be printed directly from the smart contract AST by a simple procedure (as we outlined in Section 2). We start with defining the required data structures such as `State`, `Action`, `Result` and `Msg` meaning contract state, resulting contract actions, the type of results (equivalent to the `option` type of Coq) and messages accepted by this contract. We pre-generate string constants for corresponding names of inductive types, constructors, etc. using the MetaCoq template monad.[11] This allows for more readable presentation using our notation mechanism. Currently, we use the `nat` type of Coq to represent account addresses and currency. Eventually, these types will be replaced with corresponding formalisations of these primitive types.

The `trans_global_dec` : `global_dec` → `mutual_inductive_entry` function takes the syntax of the data type declarations and produces an element of `mutual_inductive_entry` — a MetaCoq representation for inductive types. For each of our deeply embedded data type definitions we produce corresponding definitions of inductive types in Coq by using the `Make Inductive` command of MetaCoq that "unquotes" given instances of the `mutual_inductive_entry` type. Similar notation mechanism is used to write programs using the deep embedding. The definition of `crowdfunding` represents a syntax of the crowdfunding contract. We translate the crowdfunding contract's AST into a MetaCoq AST using the `expr_to_term` : `global_env` → `expr` → `term` function. Here, `global_env` is a global environment containing declarations of inductive types used in the function definition, `expr` is a type of Oak expressions, and `term` is a type of MetaCoq terms. Before translating the Oak AST we apply the `indexify` function that converts named variables into De Bruijn indices. The result of these transformations is unquoted with the `Make Definition` command. The corresponding function has the following type `entry` : `ctx` → `State_coq` → `Msg_coq` → `Result_coq`, where `ctx` is a call context containing current block time, transferred amount, sender's address and other information available for inspection during the contract call. The type names with the "coq" postfix correspond to the unquoted data types from the Figure 2.

The `entry` function corresponds to a transition from the current state of the contract to the new state. That allows for proving functional correctness properties using pre- and post-conditions. Similarly to [7], we a prove number of properties of the contract using the shallow embedding. Specifically, we proved the following properties: the contract does no leak funds; the donations can be paid back to the backers if the goal is not reached within a deadline; donations are recorded correctly in the contract's state. Moreover, in our Coq development, we show how one can verify library code for Oak by proving Oak functions equivalent to the corresponding functions from the standard library of Coq. In particular, we provide an example of such a procedure for certain functions on finite maps.

4 Related work

In this work we focus on modern smart contract languages based on a functional programming paradigm. In many cases various small errors in smart contracts can be ruled out by the type

[10] Custom entries are available starting from Coq 8.10

[11] The template monad is a part of the MetaCoq infrastructure. It allows for interacting with Coq's global environment: reading data about existing definitions, adding new definitions, quoting/unquoting definitions, etc.

systems of these languages. Capturing more serious errors requires employing such techniques as deductive verification (for verification of concrete contracts) and formalisation of meta-theory (to ensure soundness of type systems). Works related to formalisation of such languages are mentioned in Section 1 and include languages like Plutus, Michelson, Liquidity, Scilla and Simplicity.

5 Conclusion and future work

We have presented a work-in-progress on the smart contract verification framework. An important feature of our approach is the ability to both develop a meta-theory of a smart contract language and to conveniently reason about smart contracts. One can prove soundness theorems relating meta-theory of the smart contract language with the embedding. Such an option is usually not available for source-to-source translations. We applied our approach to the development of an embedding of the Oak smart contract language and provided an example of verification of a crowdfunding contract starting from the contract's AST. However, the approach is quite general and applies to other functional smart contract languages.

As future work, we would like to provide integration with Oak-language infrastructure allowing for a convenient translation of Oak programs to Coq. Since our framework is not focused on one particular smart contract language, we also consider benchmarking our development by developing "backends" for translation of other languages (e.g. Liquidity, Simplicity). Currently, our framework allows for proving functional correctness of contracts corresponding to one "step" from the current state to the new state. To be able to reason about the chain of contract calls one needs an execution model to be formalised in Coq as well. We plan to connect our development to the ongoing work on formalising such and execution model for the Oak programming language[4].

Extending the formalisation of the Oak language meta-theory is also among our goals for the framework. An important bit of Oak's meta-theory is the cost semantics allowing for reasoning about "gas". We would like to give a cost semantics for the deep embedding and explore how it can be extended on the shallow embedding.

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

[3] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *CCS16*, pages 254–269, 2016.

[4] Jakob Botsch Nielsen and Bas Spitters. Smart Contract Interactions in Coq. Submitted to FMBC19.

[5] Russell O'Connor. Simplicity: A New Language for Blockchains. PLAS17, pages 107–120. ACM, 2017.

[6] Ilya Sergey and Aquinas Hobor. A concurrent perspective on smart contracts. In *Financial Cryptography and Data Security*, pages 478–493, 2017.

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

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