Verifying, testing and running smart contracts in ConCert

Danil Annenkov, Mikkel Milo, Jakob Botsch Nielsen, and Bas Spitters

Concordium Blockchain Research Center, Aarhus University

Introduction Smart contracts are programs running on top of a blockchain. They often control big amounts of cryptocurrency and cannot be changed after deployment. Unfortunately, many vulnerabilities have been discovered in smart contracts and this has lead to huge financial losses (e.g. TheDAO, Parity's multi-signature wallet). So, smart contract verification is becoming increasingly important. Functional smart contract languages (FSCLs) are becoming increasingly popular: e.g. Simplicity [9], Liquidity[1], Plutus[2], Scilla [10] and Midlang[3]. A contract in such a language is just a function from a message type and a current state to a new state and a list of actions (transfers, calls to other contracts), making smart contracts more amenable for formal verification. We build on the ConCert framework [3] for embedding smart contracts in Coq and the execution model introduced in [8]. In the present work, we extend ConCert with an extraction functionality, implement anonymous voting based on the Open Vote Network protocol and integrate property-based testing using QuickChick.[4]

Extraction The Coq proof assistant features a possibility of extracting the executable content of Gallina terms into OCaml, Haskell and Scheme [6]. The extraction procedure is non-trivial since Gallina is a dependently-typed functional language. The extraction code itself might contain errors and current Coq extraction adds to the trusted computing base (TCB). Projects such as MetaCoq [11] and CertiCoq [1] address this issue by verifying the extraction procedure in Coq, but do not extract to smart contract languages. The general idea of extraction is to find and mark all parts of a program that do not contribute to computation. That is, types and propositions in terms are replaced with □ (a box). One of the important results of [6] and [11] is that the computational properties of the erased terms are preserved. We extend on the work on the certified erasure and implement *deboxing* — a simple optimisation procedure for removing some redundant constructs (boxes) left after the erasure step. Following [6], we identify a safe way of removing boxes: all the constants must be applied to all the logical arguments (the ones that correspond to applied boxes). The validation procedures are implemented in Coq using the formally verified type checker from the MetaCoq project. After the deboxing step, the code is pretty-printed to the target language. Currently, we support Liquidity and Midlang as target languages, but the technique applies to the other languages mentioned above.

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 → Z → option (list action ∗ Z)`. The main functionality is given by the two functions `inc_counter` and `dec_counter`. We use *refinement 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 then the previous state.

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

The Liquidity and Midlang extractions of this code can be seen in the listings below.

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

Listing 1: Liquidity

```
proj1_sig : Sig a → a
proj1_sig e = case e of Exist a → a
inc_counter : Z → Sig Z → Sig Z
inc_counter st new_balance =
  Exist (add st (proj1_sig new_balance))
```

Listing 2: Midlang

As one can see, the extraction procedure removes all "logical" parts from the original 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.

Despite the apparent similarity of our target languages and OCaml, there are many semantic differences and restrictions that makes extraction non-trivial. E.g. the absence of unsafe coercions (`Obj.magic`), data types are limited to non-recursive inductive types, support for recursive definitions is limited to tail recursion on a single argument[5].

[1] https://www.liquidity-lang.org/

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

[3] https://developers.concordium.com/midlang

[4] Our development is available at https://github.com/AU-COBRA/ConCert

[5] These restrictions apply to FSCLs to a different extent, but unsafe coercions are missing in all the FSCL we considered.

We successfully applied the developed extraction to several variants of the counter contract, to the crowdfunding contract described in [3] and to an interpreter for a simple expression language. The latter example shows the possibility of extracting certified interpreters for domain-specific languages such as Marlowe[6] and CL [4, 2] representing an important step towards safe smart contract programming.

Boardroom Voting Hao, Ryan and Zielisky developed the Open Vote Network protocol [5], an e-voting protocol that allows a small number of parties ('a boardroom') to vote anonymously on a topic. Their protocol allows tallying the vote while still maintaining maximum voter privacy, meaning that each vote is kept private unless all other parties collude. Each party proves with zero-knowledge to all other parties that they are following the protocol correctly and that their votes are well-formed.

This protocol was implemented as an Ethereum smart contract by McCorry, Shahandashti and Hao [7]. In their implementation, the smart contract serves as the orchestrator of the vote by verifying the zero-knowledge proofs and computing the final tally.

We implement a similar contract in the ConCert framework. Under the assumption that all participants behave correctly, we prove that our contract cannot compute the wrong tally. We additionally implement the zero-knowledge proofs that allow the contract to verify the behaviour of all participants. This consists of functions to create zero-knowledge proofs and functions to verify these, used by the contract itself. We prove the zero-knowledge proofs correct in the sense that we show that proofs constructed using our functions also verify correctly.[7]

Both the tallying and the zero-knowledge proofs are based on finite field arithmetic, so we develop some required theory about $\mathbb{Z}_p$ including Fermat's theorem and the extended Euclidean algorithm. In the future, we hope that the stdlib2 project will integrate parts of MathComp such as finite field arithmetic into the standard library of Coq and look forward to using that. For now, we choose to reimplement the parts that we need. Additionally, since ConCert includes an executable specification we use the Bignums library to implement computation in $\mathbb{Z}_p$ in an efficient way, allowing us to compute with the boardroom voting contract directly in Coq. We are currently working on using the extraction mechanism described above to extract and run the boardroom voting contract on existing blockchains.

Testing smart contracts With ConCert's executable specification our contracts are fully testable from within Coq. This enables us to integrate property-based testing into ConCert using QuickChick.[8] This serves as a cost-effective, semi-formal, semi-automated approach to discover bugs and increases reliability that the implementation is correct. It may be used either as a preliminary step to support formal verification or as a complementary approach whenever the properties become too involved to prove.

The testing framework is semi-automated in the sense that the user must implement a *generator* function for the message type of the contract they want to test, i.e. a function which generates "arbitrary" messages to be sent to the contract. The framework then generates thousands of "arbitrary" blockchain execution traces and uses QuickChick to test if the supplied properties hold.

We demonstrate the usability of the framework by testing complex contracts such as the congress contract (the essence of TheDAO) and the Tezos FA2 token standard.[9] The testing framework currently supports testing of functional properties, as well as temporal properties (i.e. involving reachability of states).

References

[1] Abhishek Anand et al. "CertiCoq: A verified compiler for Coq". In: *CoqPL'2017.*

[2] Danil Annenkov and Martin Elsman. "Certified Compilation of Financial Contracts". In: *PPDP'2018.*

[3] Danil Annenkov, Jakob Botsch Nielsen, and Bas Spitters. "ConCert: A Smart Contract Certification Framework in Coq". In: *CPP'2020.*

[4] Patrick Bahr, Jost Berthold, and Martin Elsman. "Certified Symbolic Management of Financial Multi-Party Contracts". In: *SIGPLAN Not.* (2015), 315327.

[5] Feng Hao, Peter YA Ryan, and Piotr Zieliński. "Anonymous voting by two-round public discussion". In: *IET Information Security* 4.2 (2010).

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

[7] Patrick McCorry, Siamak F Shahandashti, and Feng Hao. "A smart contract for boardroom voting with maximum voter privacy". In: *FC 2017.*

[8] Jakob Botsch Nielsen and Bas Spitters. "Smart Contract Interactions in Coq". In: *FMBC'2019.*

[9] Russell O'Connor. "Simplicity: A New Language for Blockchains". In: *arXiv:1711.03028* (2017).

[10] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. "Scilla: a Smart Contract Intermediate-Level LAnguage". In: *CoRR* abs/1801.00687 (2018). arXiv: 1801.00687. URL: http://arxiv.org/abs/1801.00687.

[11] Matthieu Sozeau et al. "Coq Coq Correct! Verification of Type Checking and Erasure for Coq, in Coq". In: *POPL'2019.*

[6] https://github.com/input-output-hk/marlowe

[7] We do not prove the inverse, i.e. that incorrect proofs do not verify, which for our case is a probabilistic statement better suited for tools like EasyCrypt.

[8] Our development is available at https://github.com/mikkelmilo/ConCert/

[9] https://medium.com/@TQTezos/introducing-fa2-a-multi-asset-interface-for-tezos-55173d505e5f