Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts

Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson(✉)

Nomadic Labs, Paris, France
{bruno.bernardo,raphael.cauderlier,zhenlei.hu, basile.pesin,julien.tesson}@nomadic-labs.com

Abstract. Tezos is a blockchain launched in June 2018. It is written in OCaml and supports smart contracts. Its smart contract language is called Michelson and it has been designed with formal verification in mind. In this article, we present Mi-Cho-Coq, a Coq framework for verifying the functional correctness of Michelson smart contracts. As a case study, we detail the certification of a Multisig contract with the Mi-Cho-Coq framework.

Keywords: Certified programming · Programming languages · Blockchains · Smart contracts

1 Introduction

Tezos is a public blockchain launched in June 2018. An open-source implementation, in OCaml [16], is available [3]. Tezos is an account based smart-contract platform with a Proof-of-Stake consensus algorithm [2]. Each account has a balance of tokens (called *tez*) and some of them, named *smart contracts*, can also store code and data. A smart contract's code is triggered when a transaction is sent to the associated account. Tezos' smart contracts language is called Michelson.

Our long-term ambition is to propose certified code in the whole Tezos codebase as well as certified smart contracts. The choice of OCaml as an implementation language is an interesting first step: OCaml gives Tezos good static guarantees since it benefits from OCaml's strong type system and memory management features. Furthermore, formally verified OCaml code can be produced by a variety of tools such as F* [21], Coq [22], Isabelle/HOL [17], Why3 [13], and FoCaLiZe [18]. Another specificity of Tezos is the use of formally verified cryptographic primitives. Indeed the codebase uses the HACL* library [23], which is certified C code extracted from an implementation of Low*, a fragment of F*.

This article presents Mi-Cho-Coq, a framework whose ultimate purpose is two-sided: giving strong guarantees – down to the interpreter implementation – related to the semantics of the Michelson language; and providing a tool able to prove properties of smart contracts written in Michelson.

E. Sekerinski et al. (Eds.): FM 2019 Workshops, LNCS 12232, pp. 368–379, 2020.
https://doi.org/10.1007/978-3-030-54994-7_28

Currently, the correspondence between Mi-Cho-Coq's Michelson interpreter implemented in Coq and the Michelson interpreter from a Tezos node implemented in OCaml is unproven. However, in the long run, we would like to lift this limitation by replacing the interpreter in the node with the extraction of Mi-Cho-Coq's interpreter. It would provide a strong confidence that all properties proven in Mi-Cho-Coq would actually hold for on-chain executions. Note that achieving this will not only require engineering efforts, but also the approval of Tezos token holders. Indeed, Tezos has an on-chain governance mechanism: changes to the *economic ruleset*, a subset of the codebase that contains amongst other things the Michelson interpreter, must be approved by a vote of token holders.

In this paper we will present how Mi-Cho-Coq can be used to prove functional properties of smart contracts. It is organised as follows: Section 2 gives an overview of the Michelson smart contract language, the Mi-Cho-Coq framework is then presented in Sect. 3, a case study on a Multisig smart contract is then conducted in Sect. 4, Sect. 5 presents some related work and finally Sect. 6 concludes the article by listing directions for future work.

The Mi-Cho-Coq framework, including the Multisig contract described in Sect. 4, is available at https://gitlab.com/nomadic-labs/mi-cho-coq/tree/FMBC_2019.

2 Overview of Michelson

Smart contracts are Tezos accounts of a particular kind. They have private access to a memory space on the chain called the *storage* of the smart contract, each transaction to a smart contract account contains some data, the *parameter* of the transaction, and a *script* is run at each transaction to decide if the transaction is valid, update the smart contract storage, and possibly emit new operations on the Tezos blockchain.

Michelson is the language in which the smart contract scripts are written. The Michelson language was designed before the launch of the Tezos blockchain. The most important parts of the implementation of Michelson, the typechecker and the interpreter, belong to the economic ruleset of Tezos so the language can evolve through the Tezos amendment voting process.

2.1 Design Rationale

Smart contracts operate in a very constrained context: they need to be expressive, evaluated efficiently, and their resource consumption should be accurately measured in order to stop the execution of programs that would be too greedy, as their execution time impacts the block construction and propagation. Smart contracts are non-updatable programs that can handle valuable assets, there is thus a need for strong guarantees on the correctness of these programs.

The need for efficiency and more importantly for accurate account of resource consumption leans toward a low-level interpreted language, while the need for

contract correctness leans toward a high level, easily auditable, easily formalisable language, with strong static guarantees.

To satisfy these constraints, Michelson was made a Turing-complete, low level, stack based interpreted language (*à la* Forth), facilitating the measurement of computation costs, but with some high level features *à la* ML: polymorphic products, options, sums, lists, sets and maps data-structures with collection iterators, cryptographic primitives and anonymous functions. Contracts are pure functions that take a stack as input and return a stack as output. This side-effect free design is an asset for the conception of verification tools.

The language is statically typed to ensure the well-formedness of the stack at any point of the program. This means that if a program is well typed, and if it is being given a well-typed stack that matches its input expectation, then at any point of the program execution, the given instruction can be evaluated on the current stack.

Moreover, to ease the formalisation of Michelson, ambiguous or hidden behaviours have been avoided. In particular, unbounded integers are used to avoid arithmetic overflows and division returns an option (which is `None` if and only if the divisor is 0) so that the Michelson programmer has to specify the behaviour of the program in case of division by 0; she can however still *explicitly* reject the transaction using the **FAILWITH** Michelson instruction.

2.2 Quick Tour of the Language

The full language syntax, type system, and semantics are documented in [1], we give here a quick and partial overview of the language.

Contracts' Shape. A Michelson smart contract script is written in three parts: the parameter type, the storage type, and the code of the contract. A contract's code consists of one block of code that can only be called with one parameter, but multiple entry points can be encoded by branching on a nesting of sum types and multiple parameters can be paired into one.

When the contract is deployed (or *originated* in Tezos lingo) on the chain, it is bundled with a data storage which can then only be changed by a contract's successful execution. The parameter and the storage associated to the contract are paired and passed to the contract's code at each execution. The execution of the code must return a list of operations and the updated storage.

Seen from the outside, the type of the contract is the type of its parameter, as it is the only way to interact with it.

Michelson Instructions. As usual in stack-based languages, Michelson instructions take their parameters on the stack. All Michelson instructions are typed as a function going from the expected state of the stack, before the instruction evaluation, to the resulting stack. For example, the **AMOUNT** instruction used to obtain the amount in μtez (*i.e.* a millionth of a *tez*, the smallest token unit in Tezos) of the current transaction has type `'S` → `mutez:'S` meaning that

for any stack type `'S`, it produces a stack of type `mutez:'S`. Some instructions, like comparison or arithmetic operations, exhibit non-ambiguous ad-hoc polymorphism: depending on the input arguments' type, a specific implementation of the instruction is selected, and the return type is fixed. For example **SIZE** has the following types: `bytes:'S → nat:'S`, `string:'S → nat:'S`, `set 'elt:'S → nat:'S`, `map 'key 'val:'S → nat:'S`, `list 'elt:'S → nat:'S`

While computing the size of a string or an array of bytes is similarly implemented, under the hood, the computation of map size has nothing to do with the computation of string size.

Finally, the contract's code is required to take a stack with a pair *parameter-storage* and returns a stack with a pair *operation list-storage*: `(parameter_ty*storage_ty):[] → (operation list*storage_ty):[]`.

The operations listed at the end of the execution can change the delegate of the contract, originate new contracts, or transfer tokens to other addresses. They will be executed right after the execution of the contract. The transfers can have parameters and trigger the execution of other smart contracts: this is the only way to perform *inter-contract* calls.

A Small Example - The Vote Contract. We want to allow users of the blockchain to vote for their favorite formal verification tool. In order to do that, we create a smart contract tasked with collecting the votes. We want any user to be able to vote, and to vote as many times as they want, provided they pay a small price (say 5 *tez*). We originate the contract with the names of a selection of popular tools: Agda, Coq, Isabelle and the K framework, which are placed in the long-term storage of the contract, in an associative map between the tool's name and the number of registered votes (of course, each tool starts with 0 votes).

In Fig. 1a, we present a voting contract, annotated with the state of the stack after each line of code. When actually writing a Michelson contract, development tools (including an Emacs Michelson mode) can interactively, for any point of the code, give the type of the stack provided by the Michelson typecheck of a Tezos node.

Let's take a look at our voting program: First, the description of the storage and parameter types is given on lines `1--2`. Then the code of the contract is given. On line `5`, **AMOUNT** pushes on the stack the amount of (in μtez) sent to the contract address by the user. The threshold amount (5 *tez*) is also pushed on the stack on line `6` and compared to the amount sent: **COMPARE** pops the two top values of the stack, and pushes either -1, 0 or 1 depending on the comparison between the value. **GT** then pops this value and pushes `true` if the value is 1. If the threshold is indeed greater than the required amount, the first branch of the **IF** is executed and **FAIL** is called, interrupting the contract execution and canceling the transaction.

If the value was `false`, the execution continues on line `9`, where we prepare the stack for the next action: **DUP** copies the top of the stack, we then manipulate the tail of the stack while preserving it's head using **DIP**: there, we take the right

```
1  storage (map string int); # candidates
2  parameter string; # chosen
3  code {
4    # (chosen, candidates):[]
5    AMOUNT; # amount:(chosen, candidates):[]
6    PUSH mutez 5000000; COMPARE; GT;
7    # (5 tez > amount):(chosen, candidates):[]
8    IF { FAIL } {}; # (chosen, candidates):[]
9    DUP; DIP { CDR; DUP };
10   # (chosen, candidates):candidates:candidates:[]
11   CAR; DUP; # chosen:chosen:candidates:candidates:[]
12   DIP { # chosen:candidates:candidates:[]
13         GET; ASSERT_SOME;
14         # candidates[chosen]:candidates:[]
15         PUSH int 1; ADD; SOME
16         # (Some (candidates[chosen]+1)):candidates:[]
17       }; # chosen:(Some (candidates[chosen]+1)):candidates:[]
18   UPDATE; # candidates' :[]
19   NIL operation; PAIR # (nil, candidates' ):[]
20 }
```

(a)

```
{Elt "Agda" 0 ; Elt "Coq" 0 ; Elt "Isabelle" 0 ; Elt "K" 0}
```

(b)

Fig. 1. A simple voting contract a and an example of initial storage b

element of the `(chosen, candidates)` pair with **CDR**, and we duplicate it again. By closing the block guarded by **DIP** we recover the former stack's top, and the following line takes its left element with **CAR**, and duplicates it.

On line 12, we use **DIP** to protect the top of the stack again. **GET** then pops `chosen` and `candidates` from the stack, and pushes an option containing the number of votes of the candidate, if it was found in the map. If it was not found, **ASSERT_SOME** makes the program fail. On line 15, the number of votes is incremented by **ADD**, and packed into an option type by **SOME**.

We then leave the **DIP** block to regain access to value at the top of the stack (`chosen`). On line 18, **UPDATE** pops the three values remaining on top of the stack, and pushes the `candidates` map updated with the incremented value for `chosen`. Finally, we push an empty list of operations with **NIL** `operation`, and pair the two elements on top of the stack to get the correct return type.

3 Mi-Cho-Coq: A Verification Framework in Coq for Michelson

Mi-Cho-Coq consists of an implementation of a Michelson interpreter in Coq as well as a weakest precondition calculus à la Dijkstra [12].

Michelson Syntax and Typing in Coq. Michelson's type system, syntax and semantics, as described in the main documentation, are fully formalised in Mi-Cho-Coq.

The abstract syntax tree of a Michelson script is a term of an inductive type which carries the script type:

```
Inductive instruction : list type → list type → Set :=
| NOOP {A} : instruction A A
| FAILWITH {A B a} : instruction (a :: A) B
| SEQ {A B C} : instruction A B → instruction B C → instruction A C
| IF {A B} : instruction A B → instruction A B → instruction (bool :: A) B
| LOOP {A} : instruction A (bool :: A) → instruction (bool :: A) A ...
```

A Michelson code is usually a sequence of instructions (SEQ), which is one of the instruction constructors. It has type instruction stA stB where stA and stB are respectively the type of the input stack and of the output stack.

The stack type is a list of Michelson type constructions, defined in the type inductive:

```
Inductive comparable_type : Set :=
| nat | int | string | bytes | bool | mutez | address | key_hash | timestamp.

Inductive type : Set :=
| Comparable_type (a : comparable_type) | key | unit | signature | operation
| option (a : type) | list (a : type) | set (a : comparable_type)
| contract (a : type) | pair (a b : type) | or (a b : type) | lambda (a b : type)
| map (key : comparable_type) (val : type)
| big_map (key : comparable_type) (val : type).
```

A full contract, for a given storage type storage and parameter type params is an instruction of type

```
instruction ((pair params storage) :: nil) ((pair (list operation) storage) :: nil).
```

Thanks to the indexing of the instruction inductive by the input and output stack types, only well-typed Michelson instructions are representable in Mi-Cho-Coq. This is very similar to the implementation of Michelson in the Tezos node which uses a similar feature in OCaml: generalised algebraic datatypes.

To ease the transcription of Michelson contracts into Mi-Cho-Coq AST we use notations so that contracts in Mi-Cho-Coq look very similar to actual Michelson code. The main discrepancy between Michelson and Mi-Cho-Coq syntax being that due to parsing limitations, the Michelson semi-colon instruction terminator has to be replaced by a double semi-colon instructions separator.

The ad-hoc polymorphism of Michelson instructions is handled by adding an implicit argument to the corresponding instruction constructor in Mi-Cho-Coq. This argument is a structure that carries an element identifying the actual implementation of the instruction to be used. As the argument is *implicit and maximally inserted*, Coq's type unifier tries to fill it with whatever value can fit with the known types surrounding it, *i.e.* the type of the input stack. Possible values are declared through the Coq's canonical structures mechanism, which is very similar to (Coq's or Haskell's) typeclasses.

Michelson Interpreter in Coq. Michelson semantics is formalised in Coq as an evaluator eval of type `forall {A B : list type}, instruction A B → nat → stack A → M (stack B)` where M is the error monad used to represent the explicit failure of the execution of a contract, and where stack A (resp. stack B) is the type of a stack data whose type matches A (resp. B), the list of types. As the stack is implemented as a tuple, stack constructs a product of types. The argument of type nat is called the *fuel* of the evaluator. It represents a bound on the depth of the execution of the contract and should not be confused with Michelson's cost model which is not yet formalised in Mi-Cho-Coq.

Some domain specific operations which are hard to define in Coq are axiomatised in the evaluator. These include cryptographic primitives, data serialisation, and instructions to query the context of the call to the smart contract (amount and sender of the transaction, current date, balance and address of the smart contract).

A Framework for Verifying Smart Contracts. To ease the writing of correctness proofs in Mi-Cho-Coq, a weakest precondition calculus is defined as a function eval_precond of type `forall {fuel A B}, instruction A B → (stack B → Prop) → (stack A → Prop)` that is a Coq function taking as argument an instruction and a predicate over the possible output stacks of the instruction (the postcondition) and producing a predicate on the possible input stacks of the instruction (the precondition).

This function is proved correct with respect to the evaluator:

```
Lemma eval_precond_correct {A B} (i : instruction A B) fuel st psi :
  eval_precond fuel i psi st <->
    match eval i fuel st with Failed _ _ => False | Return _ a => psi a end.
```

Note that the right-hand side formula is the result of the monad transformer of [5] which here yields a simple expression thanks to the absence of complex effects (exceptions, state, etc.) in Michelson.

A Small Example - The Vote Contract. We give below a formal specification of the voting contract seen previously, written in pseudo-code to keep it clear and concise. Section 4 presents a case study with a more detailed Coq specification.

We want the contract to take into account every vote sent in a transaction with an amount greater than 5 *tez*. Moreover, we want to only take into account the votes toward an actual available choice (the contract should fail if the wrong name is sent as a parameter). Finally, the contract should not emit any operation.

In the following specification, the *precondition* is the condition that must be verified for the contract to succeed. The *postcondition* fully describes the new state of the storage at the end of the execution, as well as the potentially emitted operations. `amount` refers to the quantity of μtez sent by the caller for the transaction.

Precondition: $\texttt{amount} \geq 5000000 \wedge \text{chosen} \in \texttt{Keys}(\text{storage})$
Postconditions: $\text{returned\_operations} = [\,] \wedge$
$\forall\ c,\ c \in \texttt{Keys}(\text{storage}) \iff c \in \texttt{Keys}(\text{new\_storage}) \wedge$
$\text{new\_storage}[\text{chosen}] = \text{storage}[\text{chosen}] + 1 \wedge$
$\forall\ c \in \texttt{Keys}(\text{storage}),\ c \neq \text{chosen} \Rightarrow \text{new\_storage}[c] = \text{storage}[c]$

Despite looking simple, proving the correctness of the vote contract still needs a fair number of properties about the map data structure. In particular we need some lemmas about the relations between the `mem`, `get` and `update` functions, which we added to the Mi-Cho-Coq library to prove this contract.

Once these lemmas are available, the contract can easily be proved by studying the three different situations that can arise during the execution: the contract can fail (either because the sender has not sent enough tez or because they have not selected one of the possible candidates), or the execution can go smoothly.

4 A Case Study: The Multisig Contract

The *Multisig* contract is a typical example of access-control smart contract. A Multisig contract is used to share the ownership of an account between several owners. The owners are represented by their cryptographic public keys in the contract storage and a pre-defined *threshold* (a natural number between 1 and the number of owners) of them must agree for any action to be performed by the Multisig contract.

Agreement of an owner is obtained by requiring a cryptographic signature of the action to be performed. To ensure that this signature cannot be replayed by an attacker to authenticate in another call to a Multisig contract (the same contract or another one implementing the same authentication protocol), a nonce is appended to the operation before signing. This nonce consists of the address of the contract on the blockchain and a counter incremented at each call.

Michelson Implementation. To be as generic as possible, the possible actions of our Multisig contract are:

- produce a list of operations to be run atomically
- change the threshold and the list of owner public keys

The contract features two entrypoints named `default` and `main`[1]. The `default` entrypoint takes no parameter (it has type `unit`) and lets unauthenticated users send funds to the Multisig contract. The `main` entrypoint takes as parameters an action, a list of optional signatures, and a counter value. It checks the validity and the number of signatures and, in case of successful authentication, it executes the required action and increment the counter.

The Michelson script of the Multisig contract is available at [9]. The code of the `default` entrypoint is trivial. The code for the `main` entrypoint can be divided in three parts: the header, the loop, and the tail.

[1] i.e. the parameter of the contract is a sum type branching two elements, cf. Sect. 2.2.

The header packs together the required action and the nonce and checks that the counter given as parameter matches the one stored in the contract.

The loop iterates over the stored public keys and the optional signatures given in parameter. It counts and checks the validity of all the signatures.

Finally the contract tail checks that the number of provided signatures is at least as large as the threshold, it increments the stored counter, and it runs the required action (it either evaluates the anonymous function passed in the contract parameter and emits the resulting operations or modifies the contract storage to update the list of owner public keys and the threshold).

Specification and Correctness Proof. Mi-Cho-Coq is a functional verification framework. It is well suited to specify the relation between the input and output stacks of a contract such as Multisig but it is currently not expressive enough to state properties about the lifetime of a smart contract nor the interaction between smart contracts. For this reason, we have not proved that the Multisig contract is resistant to replay attacks. However, we fully characterise the behaviour of each call to the Multisig contract using the following specification of the Multisig contract, where `env` is the evaluation environment containing among other data the address of the contract (`self env`) and the amount of the transaction (`amount env`).

```
Definition multisig_spec (parameter : data parameter_ty) (stored_counter : N)
           (threshold : N) (keys : Datatypes.list (data key))
           (new_stored_counter : N) (new_threshold : N)
           (new_keys : Datatypes.list (data key))
           (returned_operations : Datatypes.list (data operation))
           (fuel : Datatypes.nat) :=
  let storage : data storage_ty := (stored_counter, (threshold, keys)) in
  match parameter with
  | inl tt ⇒
    new_stored_counter = stored_counter ∧ new_threshold = threshold ∧
    new_keys = keys ∧ returned_operations = nil
  | inr ((counter, action), sigs) ⇒
    amount env = (0  Mutez) ∧ counter = stored_counter ∧
    length sigs = length keys ∧
    check_all_signatures sigs keys (fun k sig ⇒
          check_signature env k sig
            (pack env pack_ty (address_ env parameter_ty (self env),
                               (counter, action)))) ∧
    (count_signatures sigs >= threshold)%N ∧
    new_stored_counter = (1 + stored_counter)%N ∧
    match action with
    | inl lam ⇒
      match (eval lam fuel (tt, tt)) with
      | Return _ (operations, tt) ⇒
        new_threshold = threshold ∧ new_keys = keys ∧
        returned_operations = operations
      | _ ⇒ False
```

```
    end
  | inr (nt, nks) ⇒
    new_threshold = nt ∧ new_keys = nks ∧ returned_operations = nil
  end end.
```

Using the Mi-Cho-Coq framework, we have proved the following theorem:

```
Lemma multisig_correct (params : data parameter_ty)
      (stored_counter new_stored_counter threshold new_threshold : N)
      (keys new_keys : list (data key))
      (returned_operations : list (data operation)) (fuel : nat) :
  let storage : data storage_ty := (stored_counter, (threshold, keys)) in
  let new_storage : data storage_ty :=
    (new_stored_counter, (new_threshold, new_keys)) in
  17 * length keys + 14 $\leq$ fuel →
  eval multisig (23 + fuel) ((params, storage), tt)
    = Return _ ((returned_operations, new_storage), tt) <→
  multisig_spec params stored_counter threshold keys
    new_stored_counter new_threshold new_keys returned_operations fuel.
```

The proof relies heavily on the correctness of the precondition calculus. The only non-trivial part of the proof is the signature checking loop. Indeed, for efficiency reasons, the Multisig contract checks the equality of length between the optional signature list and the public key list only after checking the validity of the signature; an optional signature and a public key are consumed at each loop iteration and the list of remaining optional signatures after the loop exit is checked for emptiness afterward. For this reason, the specification of the loop has to allow for remaining unchecked signatures.

5 Related Work

Formal verification of smart contracts is a recent but active field. The K framework has been used to formalise [15] the semantics of both low-level and high-level smart contract languages for the Ethereum and Cardano blockchains. These formalisations have been used to verify common smart contracts such as Casper, Uniswap, and various implementations of the ERC20 and ERC777 standards.

Note also a formalisation of the EVM in the F* dependently-typed language [14], that was validated against the official Ethereum test suite. This formalisation effort led to formal definitions of security properties for smart contracts (call integrity, atomicity, etc).

Ethereum smart contracts, written in the Solidity high-level language, can also be certified using a translation to F* [7].

The Zen Protocol [4] directly uses F* as its smart contract language so that smart contracts of the Zen Protocol can be proved directly in F*. Moreover, runtime tracking of resources can be avoided since computation and storage costs are encoded in the dependent types.

The Scilla [19,20] language of the Zilliqa blockchain has been formalised in Coq as a shallow embedding. This intermediate language is higher-level (it

is based on λ-calculus) but also less featureful (it is not Turing-complete as it does not feature unbounded loops nor general recursion) than Michelson. Its formalisation includes inter-contract interaction and contract lifespan properties. This has been used to show safety properties of a crowdfunding smart contract. To the best of our knowledge, no tool currently exists for interactive functional verification of Scilla smart contracts but Scilla's framework for writing static analyses can be used for automated verification of some specific properties.

6 Limits and Future Work

As we have seen, the Mi-Cho-Coq verification framework can be used to certify the functional correctness of non-trivial smart contracts of the Tezos blockchain such as the Multisig contract. We are currently working on several improvements to extend the expressivity of the framework; Michelson's cost model and the semantics of inter-contract interactions are being formalised.

In order to prove security properties, such as the absence of signature replay in the case of the Multisig contract, an adversarial model has to be defined. This task should be feasible in Coq but our current plan is to use specialised tools such as Easycrypt [6] and ProVerif [8].

No code is currently shared between Mi-Cho-Coq and the Michelson evaluator written in OCaml that is executed by the Tezos nodes. We would like to raise the level of confidence in the fact that both evaluators implement the same operational semantics. We could achieve this either by proposing to the Tezos stakeholders to amend the economic protocol to replace the Michelson evaluator by a version extracted from Mi-Cho-Coq or by translating to Coq the OCaml code of the Michelson evaluator using a tool such as CoqOfOCaml [11] or CFML [10] and then prove the resulting Coq function equivalent to the Mi-Cho-Coq evaluator.

Last but not least, to ease the development of certified compilers from high-level languages to Michelson, we are working on the design of an intermediate compilation language called Albert that abstracts away the Michelson stack.

References

1. Michelson: the language of Smart Contracts in Tezos. https://tezos.gitlab.io/whitedoc/michelson.html
2. Proof-of-stake in Tezos. https://tezos.gitlab.io/whitedoc/proof_of_stake.html
3. Tezos code repository. https://gitlab.com/tezos/tezos
4. An introduction to the zen protocol. https://www.zenprotocol.com/files/zen_protocol_white_paper.pdf (2017)
5. Ahman, D., et al.: Dijkstra monads for free. CoRR abs/1608.06499 (2016). http://arxiv.org/abs/1608.06499
6. Barthe, G., Dupressoir, F., Grégoire, B., Kunz, C., Schmidt, B., Strub, P.-Y.: EasyCrypt: a tutorial. In: Aldini, A., Lopez, J., Martinelli, F. (eds.) FOSAD 2012-2013. LNCS, vol. 8604, pp. 146–166. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10082-1_6

7. Bhargavan, K., et al.: Formal verification of smart contracts: short paper, pp. 91–96. PLAS 2016. ACM, New York (2016). https://doi.org/10.1145/2993600.2993611
8. Blanchet, B.: Modeling and verifying security protocols with the applied pi calculus and proverif. Found. Trends Priv. Secur. **1**(1–2), 1–135 (2016). https://doi.org/10.1561/3300000004
9. Breitman, A.: Multisig contract in Michelson. https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/generic_multisig.tz
10. Charguéraud, A.: Characteristic formulae for the verification of imperative programs. In: ICFP 2011, pp. 418–430. ACM, New York (2011)
11. Claret, G.: Program in Coq. Theses, Université Paris Diderot - Paris 7, September 2018. https://hal.inria.fr/tel-01890983
12. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975). https://doi.org/10.1145/360933.360975
13. Filliâtre, Jean-Christophe, Paskevich, Andrei: Why3—where programs meet provers. In: Felleisen, Matthias, Gardner, Philippa (eds.) ESOP 2013. LNCS, vol. 7792, pp. 125–128. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-37036-6_8. https://hal.inria.fr/hal-00789533
14. Grishchenko, Ilya, Maffei, Matteo, Schneidewind, Clara: A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, Lujo, Küsters, Ralf (eds.) POST 2018. LNCS, vol. 10804, pp. 243–269. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
15. Hildenbrandt, E., et al.: KEVM: a complete semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium, pp. 204–217. IEEE (2018)
16. Leroy, X., Doligez, D., Frisch, A., Garrigue, J., Rémy, D., Vouillon, J.: The OCaml system release 4.08: documentation and user's manual. User manual, Inria, June 2019. http://caml.inria.fr/pub/docs/manual-ocaml/
17. Nipkow, Tobias, Wenzel, Markus, Paulson, Lawrence C. (eds.): Isabelle/HOL: A Proof Assistant forHigher-Order Logic. LNCS, vol. 2283. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45949-9
18. Pessaux, F.: FoCaLiZe: inside an F-IDE. In: Workshop F-IDE 2014. Proceedings F-IDE 2014, Grenoble, France, May 2014. https://doi.org/10.4204/EPTCS.149.7
19. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR abs/1801.00687 (2018). http://arxiv.org/abs/1801.00687
20. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with scilla. PACMPL **3**(OOPSLA), 185:1–185:30 (2019). https://doi.org/10.1145/3360611
21. Swamy, N., et al.: Dependent types and multi-monadic effects in F*. In: POPL, pp. 256–270. ACM, January 2016. https://www.fstar-lang.org/papers/mumon/
22. The Coq development team: The Coq Reference Manual, version 8.9, November 2018. http://coq.inria.fr/doc
23. Zinzindohoué, J.K., Bhargavan, K., Protzenko, J., Beurdouche, B.: HACL*: a verified modern cryptographic library. Cryptology ePrint Archive, Report 2017/536