Functional Verification of Tezos Smart Contracts in Coq

Bruno Bernardo, *Raphaël Cauderlier*, Zhenlei Hu, Basile Pesin, Julien Tesson

Nomadic Labs, Paris, France

FMBC, Porto
October 11, 2019

Nomadic Labs

- R&D company focused on distributed, decentralised and formally verified software.
- Involved in the development of the core software of the Tezos blockchain.
- Based in Paris, France.

Tezos

- Public blockchain
- Live since June 2018
- Implemented in OCaml
- Open source project (MIT License)
  `https://gitlab.com/tezos/tezos`

Tezos

- Liquid Proof of Stake
  - Block production rights come from owned tokens
  - Rights can be delegated
  - Delegations are revocable
- On-chain governance
  - Ruleset can be amended through on-chain vote
    Consensus, smart contract language, voting procedure
  - Voting right is the same as block production right
- Formal methods
  - Recognised as crucial since the Tezos whitepaper
  - Static analysis, model checking, and functional verification (in F* and Coq)
  - Targeting both the OCaml implementation of the Tezos node and the smart-contract layer

Michelson: Tezos smart contract language

- stack-based Turing-complete language
  - low-level computation paradigm
  - high-level data structures
- designed with software verification in mind
  - static typing
  - language specification (syntax, typing, semantics)
  - failure is explicit
    - integers do not overflow
    - division returns an option
- implemented using an OCaml GADT
  - subject reduction for free

Michelson example: vote

```
storage (map string int);
parameter string;
code {
      # Check that at least 5tz have been sent
      AMOUNT;
      PUSH mutez 5000000; COMPARE; GT; IF { FAIL } {};

      # Pair and stack manipulation
      DUP; DIP { CDR; DUP }; CAR; DUP;

      DIP { # Get the number of votes for the chosen option
           GET; IF_NONE { FAIL } {};
           # Increment
           PUSH int 1; ADD; SOME };
      UPDATE;
      NIL operation; PAIR
    }
```

Mi-Cho-Coq

- Deep embedding of the Michelson language in Coq
- Weakest-precondition calculus
- Free software (MIT License)
  `https://gitlab.com/nomadic-labs/mi-cho-coq/`

Syntax of types

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

Inductive type : Set :=
| Comparable_type (_ : comparable_type)
| unit | option (_ : type) | lambda (_ _ : type)
| pair (_ _ : type) | or (_ _ : type)
(* Domain specific *)
| key | signature | operation | contract (_  : type)
(* Data structures *)
| list (_ : type) | set (_ : comparable_type)
| map (_ : comparable_type) (_ : type).

Coercion Comparable_type : comparable_type >-> type.

Definition stack_type := Datatypes.list type.
```

Syntax and typing of instructions

```
Inductive instruction
   : stack_type -> stack_type -> Set :=
| 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
| COMPARE {a : comparable_type} {S} :
          instruction (a :: a :: S) (int :: S)
| DROP {a S} : instruction (a :: S) S
| DUP {a S} : instruction (a :: S) (a :: a :: S)
| SWAP {a b S} : instruction (a :: b :: S) (b :: a :: S)
| ....
```

Semantics

```
Fixpoint eval {A B : stack_type}
      (i : instruction A B) (input : stack A) : stack B :=
    match i, input with
    | FAILWITH, (x, _) =>
        ...
    | SEQ i1 i2, input =>
        eval i2 (eval i1 input)
    | IF bt bf, (b, st) =>
        if b then eval bt st else eval bf st
    | LOOP body, (b, st) =>
        if b then eval (SEQ body (LOOP body)) st
        else st
    ...
```

Semantics

```
Fixpoint eval {A B : stack_type}
      (i : instruction A B) (input : stack A) : M (stack B) :=
    match i, input with
    | FAILWITH, (x, _) =>
       Failed _ (Assertion_Failure _ x)
    | SEQ i1 i2, input =>
       bind (eval i2) (eval i1 input)
    | IF bt bf, (b, st) =>
       if b then eval bt st else eval bf st
    | LOOP body, (b, st) =>
       if b then eval (SEQ body (LOOP body)) st
       else Return _ st
    ...
```

Semantics

```
Fixpoint eval {A B : stack_type}
      (i : instruction A B) (input : strack A)
      (fuel : nat) {struct fuel} : M (stack B) :=
  match fuel with
  | 0 => Failed _ Out_of_fuel
  | S n =>
    match i, input with
    | FAILWITH, (x, _) =>
       Failed _ (Assertion_Failure _ x)
    | SEQ i1 i2, input =>
       bind (eval i2 n) (eval i1 n input)
    | IF bt bf, (b, st) =>
       ...
    | LOOP body, (b, st) =>
       ...
```

Semantics of domain specific operations

- `mutez` and `timestamp` arithmetics are supported
- serialisation, cryptographic primitives, and access to the chain context are axiomatized

Verification

```
Definition correct_smart_contract {A B : stack_type}
  (i : instruction A B) min_fuel spec : Prop :=
  forall (input : stack A) (output : stack B) fuel,
    fuel >= min_fuel input ->
    eval i fuel input = Return (stack B) output <->
      spec input output.
```

Verification

```
Definition correct_smart_contract {A B : stack_type}
  (i : instruction A B) min_fuel spec : Prop :=
  forall (input : stack A) (output : stack B) fuel,
    fuel >= min_fuel input ->
    eval i fuel input = Return (stack B) output <->
      spec input output.
```

Full functional verification: we characterize all successful runs of the contract.

Computing weakest precondition

```
Fixpoint wp {A B} (i : instruction A B) fuel
  (post : stack B -> Prop) : (stack A -> Prop) :=
  match fuel with
  | 0 => fun _ => False
  | S fuel =>
     match i, input with
     | FAILWITH, _ => False
     | SEQ B C, input => wp B fuel (wp C fuel post) input
     | IF bt bf, (b, input) =>
        if b then wp bt fuel post input
        else wp bf fuel post input
     | LOOP body, (b, input) =>
        if b then wp (SEQ body (LOOP body)) fuel post input
        else post input
     | …
```

Correctness of wp

```
Lemma wp_correct {A B} (i : instruction A B)
  fuel (post : stack B -> Prop) (input : stack A) :
  wp i fuel psi input <->
    exists output,
      eval i fuel input = Return _ output /\ psi output.
Proof. … Qed.
```

Proven smart contracts

- Vote example
- Default "manager" smart contract
- Multisig
  - n persons share the ownership of the contract.
  - they agree on a threshold t (an integer).
  - to do anything with the contract, at least t owners must agree.
  - possible actions:
    - transfer from the multisig contract to somewhere else
    - change the list of owners and the threshold

Conclusion

- The Michelson smart-contract language is formalized in Coq.
- This formalisation can be used to prove interesting Michelson smart-contracts.

Ongoing and Future Work

- certify compilers to Michelson
- formalize the Michelson cost model
- use code extraction to replace the current GADT-based implementation in OCaml
- formalize the contract life, mutual and recursive calls
- implement serialisation and cryptography

Thank you!

- Tezos
  https://gitlab.com/tezos/tezos
- Mi-Cho-Coq
  https://gitlab.com/nomadic-labs/mi-cho-coq/
- Multisig contract in Michelson
  https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/multisig.tz
- Multisig proof
  https://gitlab.com/nomadic-labs/mi-cho-coq/blob/master/src/contracts_coq/multisig.v

Mutlisig Specification

```
Definition multisig_spec
           (counter : N)
           (action : data action_ty)
           (sigs : Datatypes.list (Datatypes.option (data sign
           (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 operati
  let params : data parameter_ty := ((counter, action), sigs)
  let storage : data storage_ty := (stored_counter, (threshold
  counter = stored_counter /\
  exists first_sigs remaining_sigs,
    sigs = (first_sigs ++ remaining_sigs)%list /\
    length first_sigs = length keys /\
```

Mutlisig Specification

```
check_all_signatures
  first_sigs keys
  (fun k sig =>
     check_signature
       env k sig
       (pack env pack_ty ((chain_id_ env, address_ env ST.self
                          (counter, action)))) /\
(count_signatures first_sigs >= threshold)%N /\
new_stored_counter = (1 + stored_counter)%N /\
```

Mutlisig Specification

```
match action with
| inl (amout, contr) =>
  new_threshold = threshold /\
  new_keys = keys /\
  returned_operations = (transfer_tokens env unit tt amout con
| inr (inl kh) =>
  new_threshold = threshold /\
  new_keys = keys /\
  returned_operations = (set_delegate env kh :: nil)%list
| inr (inr (nt, nks)) =>
  new_threshold = nt /\
  new_keys = nks /\
  returned_operations = nil
end.
```