Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts

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

Coq Workshop, September 8, 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.

Blockchains

- Distributed immutable ledger, replicated via a consensus protocol
- Smart contracts = programmable accounts
  - Accounts with space for code and data
  - Programs executed by each node (*must* be small!)
  - A scarce resource (gas) is needed to *pay* for computation
  - Execution can rollback if runtime fail

Tezos

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

Tezos

- Smart contract platform
- Proof-of-Stake consensus algorithm
- On-Chain governance mechanism
  - *Economic ruleset* can be changed through a vote
  - Includes the consensus algorithm, the smart contract language (and the voting rules)
- Focus on Formal Methods
  - Use of OCaml as a first step
    - Strong static guarantees of OCaml
    - Certified OCaml code can be produced by Coq, F*, Why3, etc.
  - Formally verified cryptographic primitives (HACL*)
  - Long-term goals
    - Certification of the whole Tezos codebase
    - Certified smart contracts

Michelson: the smart contract language in Tezos

- Small stack-based Turing-complete language
- Designed with software verification in mind:
  - Static typing
  - Clear documentation (syntax, typing, semantics)
  - Failure is explicit
    - Integers do not overflow
    - Division returns an option
- Implemented using an OCaml GADT
  - Representable programs are well typed

Mi-Cho-Coq: Michelson in Coq

”

`https://gitlab.com/nomadic-labs/mi-cho-coq/`
Free software (MIT License)

Syntax: Types

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

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

Coercion Comparable_type : comparable_type >-> type.

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

Syntax: 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)
| ADD {a b} {s : add_struct a b} {S} :
    instruction (a ::: b ::: S) (add_ret_type _ _ s ::: S)
| ....
```

Semantics

```
Fixpoint eval {A B : stack_type}
      (i : instruction A B) : stack A -> stack B :=
    match i in instruction A B
      return stack A -> stack B with
    | FAILWITH x =>
       ...
    | SEQ i1 i2 =>
       fun SA => eval i2 (eval i1 SA)
    | IF bt bf =>
       fun SbA => let (b, SA) := SbA in
         if b then eval bt SA else eval bf SA
    | LOOP body =>
       fun SbA => let (b, SA) := SbA in
         if b then eval (SEQ body (LOOP body)) SA
         else SA
    ...
```

Semantics

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

Semantics

```
Fixpoint eval {A B : stack_type}
      (i : instruction A B) (fuel : nat)
      {struct fuel} : stack A -> M (stack B) :=
  match fuel with
  | O => fun SA => Failed _ Out_of_fuel
  | S n =>
    match i in instruction A B
      return stack A -> M (stack B) with
    | FAILWITH x =>
       fun _ => Failed _ (Assertion_Failure _ x)
    | SEQ i1 i2 =>
       fun SA => bind (eval i2 n) (eval i1 n SA)
    | IF bt bf =>
       ...
    | LOOP body =>
       ...
```

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 characterise the successful runs of the contract.

Computing weakest precondition

```
Fixpoint wp {A B} (i : instruction A B) fuel
  (psi : stack B -> Prop) : (stack A -> Prop) :=
  match fuel with
  | O => fun _ => False
  | S fuel =>
     match i with
     | FAILWITH => fun _ => false
     | SEQ B C => wp B fuel (wp C fuel psi)
     | IF bt bf => fun '(b, SA) =>
         if b then wp bt fuel psi SA
         else wp bf fuel psi SA
     | LOOP body => fun '(b, SA) =>
         if b then wp (SEQ body (LOOP body)) fuel psi SA
         else psi SA
     | ...
```

Computing weakest precondition

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

The multisig contract

- 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
  - changing the list of owners and the threshold

Multisig implementation in pseudo-OCaml

```
type storage =
  {counter : nat; threshold : nat; keys : list key}

type action_ty =
| Transfer of
   {amount : mutez; destination : contract unit}
| SetKeys of {new_threshold : nat; new_keys : list key}

type parameter =
  {counter : nat;
   action : action_ty;
   signature_opts : list (option signature)}
```

Multisig implementation in pseudo-OCaml

```
let multisig param storage =
  (* pack bytes that should correspond to the input sigs *)
  let packed : bytes =
    pack (counter, address self, param.action) in
  assert (param.counter = storage.counter);
  (* check validity of signatures *)
  let valid_sigs : ref nat = ref 0 in
  List.iter2 (fun key signature_opt ->
      match signature_opt with | None -> ()
      | Some signature ->
        assert (check_signature signature key bytes);
        incr valid_sigs)
    storage.keys
    param.signature_opts;
  ...
```

Multisig implementation in pseudo-OCaml

```
...
(* checks and action *)
assert (valid_sigs >= storage.threshold);
storage.counter := 1 + storage.counter;
match param.action with
| Transfer {amount; destination} ->
    transfer amount destination
| SetKeys {new_threshold; new_keys} ->
    storage.threshold := new_threshold;
    storage.keys := new_keys
```

Multisig specification

```
Definition multisig_spec input output :=
  let '(((c, a), sigs), (sc, (t, keys))) := input in
  let '(ops, (nc, (nt, nkeys))) := output in
  c = sc /\ length sigs = length keys /\
  check_all_signatures sigs keys
    (pack (address self), (c, a)) /\
  count_signatures sigs >= t /\ nc = sc + 1 /\
  match a with
  | inl (amount, dest) => nt = t /\ nkeys = keys /\
    ops = [transfer_tokens unit tt amount dest]
  | inr (t, ks) => nt = t /\ nkeys = ks /\
    ops = nil
  end.
```

Multisig correctness

```
Theorem multisig_correct :
  correct_smart_contract multisig
    (fun '(keys, _) => 14 * length keys + 37)
    multisig_spec.
Proof. ... Qed.
```

Conclusion

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

Ongoing and Future Work

- Connect Michelson and Mi-Cho-Coq
  - Formalise the Michelson cost model
  - Use code extraction to replace the current GADT-based implementation in OCaml.
- Certify compilers from higher-level languages to Michelson
- Improve expressiveness of Mi-Cho-Coq
  - Improve proof automation
  - Formalise the contract life, mutual and recursive calls
  - Prove security properties

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`