Functional Verification of Tezos Smart Contracts in Coq

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

Nomadic Labs

Semverif, Irif
May 20, 2019

Blockchains

Blockchains are:

- linked lists
  - link = cryptographic hash
- distributed databases: each node of the network
  - stores the complete chain
  - validates new incoming blocks
  - communicates with its peers toward consensus
- public ledgers for crypto-currencies

Smart Contracts

In practice, we want our crypto-currency accounts to be programmable:

- spending limits, access control

Smart Contracts

In practice, we want our crypto-currency accounts to be programmable:

- spending limits, access control
- votes, auctions, crowdfunding, timestamping, insurance, video games, etc...

Smart Contracts

In practice, we want our crypto-currency accounts to be programmable:

- spending limits, access control
- votes, auctions, crowdfunding, timestamping, insurance, video games, etc...

These programs are called smart contracts

The exploit of the Decentralized Autonomous Organisation

the DAO:

- a venture capital fund implemented as an Ethereum smart contract
- broke the crowdfunding world record
  - 15% of all Ethereum tokens (about 150 M$)

The exploit of the Decentralized Autonomous Organisation

the DAO:

- a venture capital fund implemented as an Ethereum smart contract
- broke the crowdfunding world record
  - 15% of all Ethereum tokens (about 150 M$)
- hacked in 2016
  - lead to a hard fork of the Ethereum blockchain

Smart contract verification

validating the chain $\Rightarrow$ running all the smart contracts

- smart contracts are necessarily small!

Smart contract verification

validating the chain ⇒ running all the smart contracts

- smart contracts are necessarily small!

Perfect set-up for formal methods
Let's verify them!

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
    - subject reduction for free

Mi-Cho-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) | contract (_  : type)
| pair (_ _ : type) | or (_ _ : type)
| lambda (_ _ : type)
| map (_ : comparable_type) (_ : 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_nat {S} : instruction (nat :: nat :: S) (nat :: S)
| ADD_int {S} : instruction (int :: int :: S) (int :: 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 characterize 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
  | 0 => 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
  - resetting the delegate of the multisig contract
  - changing the list of owners and the threshold

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
  - resetting the delegate of the multisig contract
  - changing the list of owners and the threshold

implemented in Michelson by Arthur Breitman
`https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/multisig.tz`

Multisig implementation in pseudo-ocaml

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

type action_ty =
| Transfer of
   {amount : mutez; destination : contract unit}
| SetDelegate of option key_hash
| 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 =
  let packed : bytes =
    pack (counter, address self, param.action) in
  assert (param.counter = storage.counter);
  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;
  assert (!valid_sigs > storage.threshold);
  storage.counter := 1 + storage.threshold;…
```

Multisig implementation in pseudo-ocaml

```
...

match param.action with
| Transfer {amount; destination} ->
    transfer amount destination
| SetDelegate new_delegate ->
    set_delegate new_delegate
| SetKeys {new_threshold; new_keys} ->
    storage.threshold := new_threshold;
    storage.keys := new_keys
```

Multisig proof: part 1 / 3

```
let multisig param storage =
  let packed : bytes =
    pack (counter, address self, param.action) in
  assert (param.counter = storage.counter);
  let valid_sigs : ref nat = ref 0 in…
```

Multisig proof: part 1 / 3

```
let multisig param storage =
  let packed : bytes =
    pack (counter, address self, param.action) in
  assert (param.counter = storage.counter);
  let valid_sigs : ref nat = ref 0 in…
```

```
Definition multisig_part_1 :
  instruction (pair parameter_ty storage_ty :: nil)
              (nat :: nat :: list (option signature) ::
               bytes :: action_ty :: storage_ty ::: nil) :=
  UNPAIR ; SWAP ; DUP ; DIP SWAP ;
  DIP (UNPAIR ; DUP ; SELF ; ADDRESS ; PAIR ;
       PACK ; DIP (UNPAIR ; DIP SWAP) ; SWAP) ;
  UNPAIR ; DIP SWAP ; ASSERT_CMPEQ ;
  DIP SWAP; UNPAIR; PUSH nat 0.
```

Multisig proof: part 1 / 3

```
Definition multisig_part_1_spec input output :=
  let '((((counter, action), sigs), storage), tt)
    := input in
  output = (storage, (counter,
           (pack (address self), (counter, action)),
           (sigs, (action, (storage, tt)))))).
```

```
Lemma multisig_part_1_correct :
  correct_smart_contract
    multisig_part_1 (fun _ => 14) multisig_part_1_spec.
Proof.
  (* Simple proof using wp_correct *)
Qed.
```

Multisig proof: part 2 / 3

```
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 proof: part 2 / 3

```
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…;
```

```
Definition multisig_loop_body :
  instruction
    (key :: nat :: list (option signature) ::
     bytes :: action_ty :: storage_ty :: nil)
    (nat :: list (option signature) ::
     bytes :: action_ty :: storage_ty :: nil)
:= …
```

Multisig proof: part 2 / 3

```
Definition multisig_loop_body :=
  DIP SWAP; SWAP;
  IF_CONS (IF_SOME
             (SWAP; DIP (SWAP; DIIP (DIP DUP; SWAP);
              CHECK_SIGNATURE; ASSERT;
              PUSH nat 1 ; ADD_nat))
             (SWAP; DROP))
          FAIL;
  SWAP.

Definition multisig_loop := ITER multisig_loop_body.
```

Multisig proof: part 2 / 3

```
Lemma multisig_loop_body_spec fuel input output :=
  let '(k, (n, (sigs, (packed, st)))) := input in
  match sigs with
  | nil => False
  | cons None sigs => output = (n, (sigs, (packed, st)))
  | cons (Some sig) sigs =>
    if check_signature k sig packed
    then output = (1 + n, (sigs, (packed, st)))
    else False
  end.

Lemma multisig_loop_body_correct :
  correct_smart_contract multisig_loop_body
    (fun _ => 14) multisig_fuel_body_spec.
Proof.
  (* Simple proof using wp_correct *)
Qed.
```

Multisig proof: part 2 / 3

```
Lemma multisig_loop_spec fuel input output :=
  let '(keys, (n, (sigs, (packed, st)))) := input in
  check_all_signatures sigs keys packed /\
  output =
    (count_signatures sigs + n, (nil, (packed, st))).

Lemma multisig_loop_correct :
  correct_smart_contract multisig_loop_body
    (fun '(keys, _) => length keys * 14 + 1)
    multisig_fuel_body_spec.
Proof.
  (* Not so simple inductive proof
     using multisig_loop_body_correct *)
Qed.
```

Multisig proof: part 3 / 3

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

```
Definition multisig_part_3 :
  instruction (nat :: nat :: list (option signature) ::
                bytes :: action_ty :: storage_ty :: nil)
               (pair (list operation) storage_ty :: nil)
```

Multisig proof: part 3 / 3

```
Definition multisig_part_3 :
  instruction (nat :: nat :: list (option signature) ::
                bytes :: action_ty :: storage_ty :: nil)
               (pair (list operation) storage_ty :: nil) :=
  ASSERT_CMPLE; ASSERT_NIL; DROP;
  DIP (UNPAIR; PUSH nat 1; ADD; PAIR);
  NIL operation; SWAP;
  IF_LEFT (UNPAIR; UNIT; TRANSFER_TOKENS; CONS)
          (IF_LEFT (SET_DELEGATE; CONS)
                   (DIP (SWAP; CAR); SWAP; PAIR; SWAP));
  PAIR.
```

Multisig proof: joining the parts

```
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 first_sigs >= t /\ nc = sc + 1 /\
  match a with
  | inl (amount, dest) => nt = t /\ nkeys = keys /\
    ops = [transfer_tokens unit tt amount dest]
  | inr (inl kh) => nt = t /\ nkeys = keys /\
    ops = [set_delegate kh]
  | inr (inr (t, ks)) => nt = t /\ nkeys = ks /\
    ops = nil
  end.
```

Multisig proof: joining the parts

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

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

- improve proof automation
- certify compilers to Michelson
- formalize the Michelson cost model
- formalize the contract life, mutual and recursive calls
- prove security properties
- use code extraction to replace the current GADT-based implementation in OCaml

Thank you!

Questions?