KxcRefcode: CKAK02TSCAFR

Frozen: a mechanically verified Tezos smart contract for vesting deposit

KAZUNARI TANAKA, Nagoya University, Japan
HAOCHEN XIE*, Kotoi-Xie Consultancy, Japan

Keywords: Smart contract, Formal verification, Tezos, Blockchain, Vesting contract

1 INTRODUCTION

We authored a Tezos smart contract that we call “Frozen” and formally verified its correctness using the Mi-cho-coq framework.

The contract implements a simple vesting deposit functionality, i.e. the contract originator deposit a certain amount of tezzes while specifying the following parameters:

- `frozen_until` - a timestamp before which the fund will be locked into the contract
- `fund_owners` - a list of implicit accounts that could withdraw the locked fund at a later time than the `frozen_until`.

We authored the contract directly on Coq with the help of the Mi-cho-coq framework and automatically converted the Coq representation of Michelson AST into the format expected by the Tezos network. The resulting contract in Michelson could be found in Code Listing 1.

The frozen contract stores the parameters in its storage and only expects invocations of valid withdrawal request. The contract code could be roughly divided into two logical parts: (1) the part that checks the validity of the withdrawal request, and (2) the part that assembles the token transfer in response to the withdraw request.

We formally verified the following properties of this contract, and the technical details will be discussed in subsequent sections.

(1) no additional fund could be added to the contract
(2) only invocations issued by one of the accounts listed in the `fund_owners` could be used to withdraw from the fund
(3) the fund could only be withdrawn at or after the timestamp `frozen_until`
(4) the contract storage, i.e. the parameters, can never be changed once the contract was originated

Code Listing 1 (Generated Michelson Contract).

```
{ parameter (pair mutez address) ;
  storage (pair (set address) timestamp) ;
  code { DUP ;
         AMOUNT ;
         PUSH mutez 0 ;
         COMPARE ;
         NEQ ;
```

*Corresponding author

This is a registered document in the KXC Technical Note Archives.

Authors’ addresses: Kazunari Tanaka, tzskp1@gmail.com, Nagoya University, Chikusaku Furocho, Nagoya, Japan, 464-8601; Haochen Xie, haochenx@acm.org, Kotoi-Xie Consultancy, Nagoya, Japan
Full reference code: HXRD.CKAK02TSCAFR~ba06g. Document rendered at 2021-02-14 15:28 JST.

```
        IF { FAILWITH } {} ;
        { DUP ; CAR ; DIP 1 { CDR } } ;
        DIP 1 { { DUP ; CAR ; DIP 1 { CDR } } } ;
        SWAP ;
        SOURCE ;
        MEM ;
        IF {} { FAILWITH } ;
        SWAP ;
        NOW ;
        COMPARE ;
        LT ;
        IF { FAILWITH } {} ;
        { DUP ; CAR ; DIP 1 { CDR } } ;
        DUP ;
        BALANCE ;
        COMPARE ;
        LT ;
        IF { FAILWITH } {} ;
        PUSH mutez 0 ;
        COMPARE ;
        EQ ;
        IF { FAILWITH } {} ;
        DROP 1 ;
        { DUP ; CAR ; DIP 1 { CDR } } ;
        { DUP ; CAR ; DIP 1 { CDR } } ;
        SWAP ;
        CONTRACT unit ;
        IF_NONE { FAILWITH } { SWAP ; PUSH unit Unit ; TRANSFER_TOKENS } ;
        DIP 1 { NIL operation } ;
        CONS ;
        PAIR } }
```

2 TECHNICAL DETAILS

Using the Mi-cho-coq framework, we verified the correctness of the contract using a Hoare logic style approach examining by establishing the connection between precondition and postcondition of the contract execution. The actual proof of the correctness statement is almost trivial by reducing the proof obligations.

In addition, we prepared a mechanism to automatically convert the Michelson AST we developed in Coq into Michelson code that is recognized by the Tezos network. We used a combination of formatting mechanism provided by Mi-cho-coq and some custom shell script.

In this section, we discuss the way we built the frozen contract in Coq, followed by a brief discussion of the formal correctness statements we achieved to verify mechanically. For readers who are interested in how the correctness statements are verified, we advice the reader to learn the Mi-cho-coq framework and read the proof script listed in Appendix A.

2.1 Contract definition

We used Coq as the "macro" language to define the Frozen contract as AST of Michelson code in the Mi-cho-coq framework. Using the technique describe above, we obtained the Michelson program directly deployable as a Tezos contract listed in Code Listing 1.

The Coq definitions we used to define the contract is listed below.

frozen.v

```
Definition parameter_ty := pair (mutez (* amount *)) (address (* beneficiary *)).
Definition storage_ty :=
  pair (set address (* fund_owners *)) (timestamp (* unfrozen *)).

Definition validate_invocation {self_type S} :
  instruction_seq self_type false (pair parameter_ty storage_ty ::: S) S :=
```

```
  {
    AMOUNT; PUSH mutez zero; COMPARE; NEQ;
    IF_TRUE {FAILWITH} { };
    UNPAIR; DIP1 {UNPAIR}; SWAP; SOURCE; @MEM _ _ _ (mem_set _) _;
    IF_TRUE { } {FAILWITH};
    (* source of operation is not whitelisted for withdrawal operations *)
    SWAP; NOW; COMPARE; LT;
    IF_TRUE {FAILWITH} { };
    (* deposit still frozen *)
    UNPAIR; DUP;
    BALANCE; COMPARE; LT;
    IF_TRUE {FAILWITH} { };
    (* requested withdrawal amount exceeds the balance *)
    PUSH mutez zero; COMPARE; EQ;
    IF_TRUE {FAILWITH} { };
    (* frozen contract cannot accept positive amount transfer *)
    DROP1
  }.

Definition perform_withdraw {self_type S} :
  instruction_seq self_type false (parameter_ty ::: S) (operation ::: S) :=
  {
    UNPAIR; SWAP; CONTRACT (Some "") unit;
    IF_NONE {FAILWITH} {SWAP; PUSH unit Unit; TRANSFER_TOKENS}
  }.

Definition frozen : full_contract false parameter_ty None storage_ty :=
  DUP;; validate_invocation;;;
  UNPAIR;; perform_withdraw;;;
  {DIP1 {NIL operation}; CONS; PAIR}.
```

Here, `validate_invocation` checks the validity of the invocation, which is expected to be a withdrawal request, `perform_withdraw` is to perform the actual payment, `frozen` combines them into the form of a contract.

We discuss the properties we have mechanically verified about `frozen`, i.e. our contract code, in the rest of this section.

2.2 Property verification

To make sure that our contract code behave exactly how it is designed to, we formalized the properties discussed in Section 1 and expressed then in Coq under the Mi-cho-coq framework as the following two lemmas.

```
Lemma frozen_correct
      (env : @proto_env (Some (parameter_ty, None)))
      (fuel : Datatypes.nat)
      (m : tez.mutez)
      (addr : data address)
      (unfrozen : data timestamp)
      (fund_owners : data (set address))
      (psi : stack (pair (list operation) storage_ty ::: [::]) -> Prop) :
fuel > 5 ->
precond (eval_seq env frozen fuel ((m, addr), (fund_owners, unfrozen), tt)) psi
<-> match contract_ env (Some "") unit addr with
  | Some c =>
    psi ([:: transfer_tokens env unit tt m c], (fund_owners, unfrozen), tt)
    /\ tez.compare (extract (tez.of_Z BinNums.Z0) I) (amount env) = Eq
    /\ set.mem address_constant address_compare (source env) fund_owners
    /\ (BinInt.Z.compare (now env) unfrozen = Gt
       \/ BinInt.Z.compare (now env) unfrozen = Eq)
    /\ (tez.compare (balance env) m = Gt
       \/ tez.compare (balance env) m = Eq)
    /\ tez.compare (extract (tez.of_Z BinNums.Z0) I) m = Lt
```

```
21      | None => false
22      end.
23  
24  Lemma frozen_preserve_storage
25        (env : @proto_env (Some (parameter_ty, None)))
26        (fuel : Datatypes.nat)
27        (m : tez.mutez)
28        (addr : data address)
29        (unfrozen : data timestamp)
30        (fund_owners : data (set address))
31        returned_operations new_storage :
32    fuel > 5 ->
33    eval_seq env frozen fuel ((m, addr), (fund_owners, unfrozen), tt)
34  = Return (returned_operations, new_storage, tt) ->
35    new_storage = (fund_owners, unfrozen).
```

Here, `frozen_correct` states that

- *the transaction amount of an invocation must equal to zero (line 14).* This ensures that no additional fund could be added to the contract.
- *the transaction source of an invocation must be in fund_owners (line 15).* This ensures that the fund could only be accessed by authorized parties.
- *the timestamp of an invocation must be greater than or equal to unfrozen (line 16–17).* This ensures that the fund could not be accessed before it matures.
- *the amount requested in the withdrawal must be less or equal to the contract balance (line 18–19).* This ensures that the requested amount is available in the contract and the specified amount would be withdrawn if the invocation was successful.
- *the amount requested in the withdrawal must be greater than zero (line 20).* This ensures the validness and usefulness of a successful withdrawal.

and `frozen_preserve_storage` states that

- *The storage is maintained as the same before and after the transaction. (line 35).* This ensures that the contract parameters (e.g. the list of `fund_owners`) never changes during the lifetime of the contract.

If one examine closely and carefully, it is not difficult to see that `frozen_preserve_storage` is actually a corollary that follows `frozen_correct` and the exact proof for the latter could be found in Appendix A.

3 CONCLUSION

(to be added)

ACKNOWLEDGMENTS

(to be finished)

- TF for Tezos Ecosystem Grant
- Mi-cho-coq
- helps from the Tezos developer community, esp the Mi-cho-coq developers

A PROOF SCRIPT

In this appendix, we include the proof script in Coq that we used to mechanically verify the properties as discussed in Section 2.2. Note that for the sake of conciseness, we omitted repeating line 17–52, which is the definition of the frozen contract and could be found in Section 2.1.

frozen.v

```
1  From mathcomp Require Import all_ssreflect.
2  From Michocoq Require Import semantics util macros.
```

```
Import syntax comparable error.

Set Implicit Arguments.
Unset Strict Implicit.
Unset Printing Implicit Defensive.

Module frozen(C : ContractContext).
Module semantics := Semantics C. Import semantics.
Require Import String.
Open Scope string_scope.

Definition zero :=
  Comparable_constant syntax_type.mutez (extract (tez.of_Z BinNums.Z0) I).
```

frozen.v

```
Local Lemma lem x :
  ~~ int64bv.sign x ->
  match int64bv.to_Z x with
  | BinNums.Z0 => true
  | BinNums.Zpos _ => true
  | BinNums.Zneg _ => false
  end.
Proof.
  move: x.
  rewrite /int64bv.sign /int64bv.to_Z /Bvector.Bsign
          /int64bv.int64 /Bvector.Bvector => x.
  apply: (@Vector.rectS _ (fun n x => forall x : Vector.t Datatypes.bool n.+1,
                          ~~ Vector.last x -> _) _ _ 63 x).
  + move=> _ x0 H.
    suff->: x0 = Vector.cons Datatypes.bool false 0 (Vector.nil Datatypes.bool)
      by [].
    apply: (@Vector.caseS' _ 0 x0
                          (fun x => ~~ Vector.last x -> x = Vector.cons
                                                            _ false
                                                            _ (Vector.nil _))
                          _ H).
    move=> h; apply: Vector.case0; by case: h.
  + move=> _ {x} n _ IH x.
    apply: (@Vector.caseS _ (fun n' (x0 : Vector.t Datatypes.bool n'.+1) =>
                              n' = n.+1 -> ~~ Vector.last x0 -> _) _ n.+1 x)
           => // {x} h n0 x C.
    move: C x => -> x.
    rewrite Zdigits.two_compl_value_Sn /= => /IH.
    case: (Zdigits.two_compl_value n x); by case: h.
Qed.

Lemma tez0 m :
  tez.compare (extract (tez.of_Z BinNums.Z0) I) m = Lt
\/ tez.compare (extract (tez.of_Z BinNums.Z0) I) m = Eq.
Proof.
  case: m => x.
  rewrite /tez.compare /int64bv.int64_compare /int64bv.compare /=.
  case H: (int64bv.sign x) => // _.
  move/negP/negP/lem: H.
  by case :(int64bv.to_Z x); auto.
Qed.

Lemma frozen_correct
      (env : @proto_env (Some (parameter_ty, None)))
      (fuel : Datatypes.nat)
      (m : tez.mutez)
      (addr : data address)
      (unfrozen : data timestamp)
      (fund_owners : data (set address))
      (psi : stack (pair (list operation) storage_ty ::: [::]) -> Prop) :
fuel > 5 ->
```

```
precond (eval_seq env frozen fuel ((m, addr), (fund_owners, unfrozen), tt)) psi
<-> match contract_ (Some "") unit addr with
  | Some c =>
    psi ([:: transfer_tokens env unit tt m c], (fund_owners, unfrozen), tt)
    /\ tez.compare (extract (tez.of_Z BinNums.Z0) I) (amount env) = Eq
    /\ set.mem address_constant address_compare (source env) fund_owners
    /\ (BinInt.Z.compare (now env) unfrozen = Gt
       \/ BinInt.Z.compare (now env) unfrozen = Eq)
    /\ (tez.compare (balance env) m = Gt
       \/ tez.compare (balance env) m = Eq)
    /\ tez.compare (extract (tez.of_Z BinNums.Z0) I) m = Lt
  | None => false
  end.
Proof.
  move=> F; have<-: 6 + (fuel - 6) = fuel by rewrite addnC subnK.
  split => H.
  + case C : (contract_ (Some "") unit addr).
    - move: H;
      rewrite eval_seq_precond_correct /eval_seq_precond /= C /=.
      case => + []+ []+ []+ []+ [][->].
      set C1 := (tez.compare _ _); case: C1 => // _ ->.
       set C2 := (tez.compare _ _); case: C2 => //.
        set C4 := (tez.compare _ _); case H4: C4 => //.
         set C3 := (BinInt.Z.compare _ _); case: C3 => //= *;
          repeat split => //; auto.
        subst C4; move: H4 (tez0 m) => ->; by case.
        set C4 := (tez.compare _ _); case H4: C4 => //.
         set C3 := (BinInt.Z.compare _ _); case: C3 => //= *;
          repeat split => //; auto.
        subst C4; move: H4 (tez0 m) => ->; by case.
    - move: H; rewrite eval_seq_precond_correct /eval_seq_precond /= C /=.
      by repeat case => ?.
  + rewrite eval_seq_precond_correct /eval_seq_precond /=.
    move: H; case: (contract_ (Some "") unit addr) => // a.
    case => []H1 []-> []-> [][]-> [][]-> ->;
    repeat split; by exists a.
Qed.

Lemma frozen_preserve_storage
      (env : @proto_env (Some (parameter_ty, None)))
      (fuel : Datatypes.nat)
      (m : tez.mutez)
      (addr : data address)
      (unfrozen : data timestamp)
      (fund_owners : data (set address))
      returned_operations new_storage :
  fuel > 5 ->
  eval_seq env frozen fuel ((m, addr), (fund_owners, unfrozen), tt)
= Return (returned_operations, new_storage, tt) ->
  new_storage = (fund_owners, unfrozen).
Proof.
  move=> f5; rewrite return_precond frozen_correct //=.
  by case: (contract_ (Some "") unit addr) => // ? [][] + ->.
Qed.
End frozen.
```