Functional Verification of Tezos Smart Contracts in Coq

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

Nomadic Labs

Deducteam Seminar
January 30, 2020

Who am I

- deducteammate (2012-2016)
- postdoc at IRIF (2016-2018)
- research engineer at Nomadic Labs (2019-?)

Who am I

- deducteammate (2012-2016)
- postdoc at IRIF (2016-2018)
- research engineer at Nomadic Labs (2019-?)
- Logipedia contributor (2020-?)

My PhD in one slide

Object-oriented mechanisms for interoperability between proof systems

- FoCaLiZe and Zenon (Modulo) $\rightarrow$ Dedukti
  - Focalide, Zenonide, Sukerujo
- operational semantics of an object-oriented calculus
  - Sigmaid, Meta Dedukti
- ITP interoperability methodology
  - automated theorem transfer
  - manual concept alignment in FoCaLiZe
  - no backward translation (final proof in Dedukti)
  - case study HOL and Coq on the sieve of Eratosthenes
- heuristic constructivisation of Zenon proofs

Nomadic Labs

`https://nomadic-labs.com`

- parisian start-up
- 35 permanent employees
  22 with a PhD in formal methods or programming languages
- R&D on the Tezos blockchain
- close to Inria and academia in general
- dedicated to free software

Blockchains

Blockchains are:

- linked lists where link = crypto hash
- distributed databases: each node
  - stores everything
  - validates new blocks
  - communicates toward consensus
- public ledgers for crypto-currencies

Tezos

https://tezos.com

https://tezos.gitlab.io

- in OCaml
- liquid proof of stake
- on-chain governance
- formal methods

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

What is contractual about smart contracts?

Once deployed, a smart contract's code:

- is public
- cannot change

What is contractual about smart contracts?

Once deployed, a smart contract's code:

- is public
- cannot change

we commit today on what we will do (and pay) in the future

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 $\Rightarrow$ 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

`https://michelson.nomadic-labs.com`

- 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

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

Mi-Cho-Coq

`https://gitlab.com/nomadic-labs/mi-cho-coq/`

Deep embedding in Coq of the Michelson language

- lexer, Menhir parser, macro expander, type checker, evaluator, pretty-printer

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.
```

Verified 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
- spending limit
  - two roles: admin and user
  - user can spend the contract's tokens up-to a stored limit
  - admin can change the limit and authentication keys

High level smart contract languages

Many languages compiled to Michelson:

- Ligo, SmartPy, Fi, Archetype, Morley, Juvix, SCaml, Liquidity, Iamtez, ...

High level smart contract languages

Many languages compiled to Michelson:

- Ligo, SmartPy, Fi, Archetype, Morley, Juvix, SCaml, Liquidity, lamtez, ...

no certified compiler

The Albert intermediate language

`https://albert-lang.io`

Goals:

- common suffix of most compilers to Michelson
- optimizing
- certified

Choices:

- abstract the stack

The Albert intermediate language

`https://albert-lang.io`

Goals:

- common suffix of most compilers to Michelson
- optimizing
- certified

Choices:

- abstract the stack
- and not much more

Type system

- same types as Michelson + n-ary variants and records
- explicit duplication
- explicit consumption
- implicit ordering

Type system

- same types as Michelson + n-ary variants and records
- explicit duplication
- explicit consumption
- implicit ordering

linear type system

Example: vote in Albert

```
type storage_ty = { threshold : mutez; votes: map string nat }

def vote :
  { param : string ; store : storage_ty } →
  { operations : list operation ; store : storage_ty } =
     {votes = state; threshold = threshold } = store ;
     (state0, state1) = dup state;
     (param0, param1) = dup param;
     prevote_option = state0[param0];
     { res = prevote } = assert_some { opt = prevote_option };
     one = 1; postvote = prevote + one; postvote = Some postvote;
     final_state = update state1 param1 postvote;
     store = {threshold = threshold; votes = final_state};
     operations = ([] : list operation)
```

Example: vote in Albert

```
def guarded_vote :
  { param : string ; store : storage_ty } →
  { operations : list operation ; store : storage_ty } =
    (store0, store1) = dup store;
    threshold = store0.threshold;
    am = amount;
    ok = am >= threshold0;
    match ok with
        False f →failwith "you are so cheap!"
      | True t →drop t;
          voting_parameters = { param = param ; store = store1 };
          vote voting_parameters
    end
```

Ott specification

- syntax, typing, and semantics specified in Ott
- modular specification (one file per language construction)
- from one source
  - OCaml AST
  - Menhir parser
  - Coq AST, typing, and semantic relations
  - LaTeX documentation

Compiler

- compiler written in Coq, certification in progress
- compiler target = Mi-Cho-Coq untyped AST
- proved optimisations at the Michelson level

Compiler pipeline

- inlining of type definitions
- sorting of record labels and variant constructors
- type checking
- function inlining + translation to Michelson

Meta theory

Subject reduction and progress proved on a fragment

$$(\Gamma \vdash instr : ty \rightarrow ty') \Rightarrow (\Gamma \vdash v : ty) \Rightarrow (E \models instr/v \Rightarrow v') \Rightarrow (\Gamma \vdash v : ty')$$

$$(\Gamma \vdash instr : ty \rightarrow ty') \Rightarrow (\Gamma \vdash v : ty) \Rightarrow (\exists v', E \models instr/v \Rightarrow v')$$

Conclusion

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

Ongoing and Future Work

- on Mi-Cho-Coq
  - formalize the Michelson cost model, contract life, mutual and recursive calls
  - prove smart contracts for other applications (security, finance, economy, ...)
  - prove equivalence between Mi-Cho-Coq and Michelson reference implementation
- on Albert
  - prove meta theory
  - improve and certify the compiler

Thank you!

Questions?