

# Formal Methods for Michelson

Raphaël Cauderlier

Nomadic Labs

Chain Accelerator Training Tezos Certification  
June 27, 2019

# Introduction

- Raphaël Cauderlier
- Nomadic Labs
- Verification of Tezos Smart Contracts in Coq

- 1 Introduction
- 2 Formal Methods for smart contracts
- 3 Multisig Contracts
- 4 Mi-Cho-Coq
- 5 Conclusion

# Motivation

- Smart contracts manipulate money (sometimes a lot)
- They are here to stay: in case of bug, they are hard to update
- Security: bugs may become exploits

Before uploading them, we want to be sure there is no bug in them!

# Motivation

- Smart contracts manipulate money (sometimes a lot)
- They are here to stay: in case of bug, they are hard to update
- Security: bugs may become exploits

Before uploading them, we want to be sure there is no bug in them!

- infinitely-many possible input values so testing cannot be exhaustive

# Definition

Formal methods: methods for mathematically reasoning about programs

# Definition

Formal methods: methods for mathematically reasoning about programs

- Semantics: Description of the meaning of all instructions of the programming language

# Definition

Formal methods: methods for mathematically reasoning about programs

- Semantics: Description of the meaning of all instructions of the programming language
- Specification: Formula in some logic describing the expected behaviour of the program



# Definition

Formal methods: methods for mathematically reasoning about programs

- Semantics: Description of the meaning of all instructions of the programming language
- Specification: Formula in some logic describing the expected behaviour of the program
- Goal: verify that the program satisfies the specification
  - Mathematical proof, more or less automatized

# Approaches

- Model Checking  
Abstract the program into a state automaton called the *model* that can be checked on all inputs.
- Abstract Interpretation  
Abstract the values as *domains* (for example intervals). Refine the abstraction when needed.
- Deductive Verification  
Reduce to the *theorem proving* problem.

# Model Checking

Abstract the program into a state automaton called the *model* that can be checked on all inputs.

- Specifications:
  - Safety No bad state can be reached
  - Liveness Good states are reached infinitely often
  - Temporal properties
- Problem:
  - Finding the model
  - Linking it to the concrete program

# Abstract Interpretation

Abstract the values as *domains* (for example intervals). Refine the abstraction when needed.

- Specifications:
  - Safety
  - Arithmetic
- Problem
  - False alarms

# Deductive Verification

Reduce to the *theorem proving* problem.

- Specifications:
  - *Functional* properties { precondition } Program { postcondition }
  - Very rich logics
- Problem
  - Requires a lot of user interaction

# Michelson design

Michelson has been designed to ease formal methods

- Static typing
- Explicit failure
- No overflow nor division by zero
- Clear documented semantics

Michelson contracts are necessarily small and simple

# Formal methods for Michelson

- Model Checking:
  - Example: auction
  - Spec: Anybody either win the auction or lose no money
  - Tool: Cubicle Model-Checker
- Abstract Interpretation:
  - Bound on gas
  - Token freeze
- Deductive Verification:
  - Example: multisig
  - Spec: multisig succeeds IFF enough valid signatures
  - Tool: Mi-Cho-Coq

# 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:
  - list of operations (to be run atomically)
  - changing the list of owners and the threshold



# The multisig contract

- Michelson implementation:

`https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/generic.tz`

# The multisig contract

- Michelson implementation:

`https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/generic.tz`

- Integrated in the `tezos-client`
  - `tezos-client deploy multisig ...`
  - `tezos-client sign multisig transaction ...`
  - `tezos-client from multisig contract ... transfer ...`

# Multisig anti-replay

- Multisig uses cryptographic signatures to
  - authenticate the owners
  - ensure they agree to perform the specific action

# Multisig anti-replay

- Multisig uses cryptographic signatures to
  - authenticate the owners
  - ensure they agree to perform the specific action
- But each signature should be usable only once
  - Once in the lifetime of the contract
  - On no other contract

# Multisig anti-replay

- Multisig uses cryptographic signatures to
  - authenticate the owners
  - ensure they agree to perform the specific action
- But each signature should be usable only once
  - Once in the lifetime of the contract
  - On no other contract
- Signed data = (action, counter, multisig address)  
Counter incremented at each successful run

# Multisig storage

We need to store

- the keys
- the threshold
- the anti-replay counter

```
storage
  (pair (nat %stored_counter)
    (pair (nat %threshold)
      (list %keys key))) ;
```

# Multisig parameter

Two entrypoints

- Default: take my tokens (by anybody)
- Main: Perform an action (requires enough signatures)

```
parameter
  (or (unit %default)
      (pair %main
        (pair :payload
          (nat %counter)
          (or :action
            (lambda %operation unit (list operation))
            (pair %change_keys
              (nat %threshold)
              (list %keys key))))
        (list %sigs (option signature))));
```

# Multisig code

```
https://github.com/murbard/smart-contracts/blob/  
master/multisig/michelson/generic.tz
```



# The Coq interactive theorem prover

- Developed for more than 30 years
- Non-trivial mathematical theorems: 4-color, odd-order
- CompCert: certified C compiler

# Coq: Dependent Types

```
Parameter A : Type.
```

# Coq: Dependent Types

```
Parameter A : Type.
```

```
Parameter a : A.
```

# Coq: Dependent Types

```
Parameter A : Type.
```

```
Parameter a : A.
```

```
Parameter B : A -> Prop.
```

# Coq: Dependent Types

```
Parameter A : Type.
```

```
Parameter a : A.
```

```
Parameter B : A -> Prop.
```

```
Parameter f : forall x : A, B x.
```

# Coq: Dependent Types

```
Parameter A : Type.
```

```
Parameter a : A.
```

```
Parameter B : A -> Prop.
```

```
Parameter f : forall x : A, B x.
```

```
Check f a. (* Answer: f a : B a *)
```

## Coq: Implicit arguments

Explicit polymorphism: Types are regular terms

```
Definition identity (A : Type) (a : A) := a.
```

# Coq: Implicit arguments

Explicit polymorphism: Types are regular terms

```
Definition identity (A : Type) (a : A) := a.
```

Inference of first argument

```
Lemma identity_2 : identity _ 2 = 2.  
Proof. reflexivity. Qed.
```



# Coq: Implicit arguments

Explicit polymorphism: Types are regular terms

```
Definition identity (A : Type) (a : A) := a.
```

Inference of first argument

```
Lemma identity_2 : identity _ 2 = 2.  
Proof. reflexivity. Qed.
```

Implicit argument

```
Definition id {A : Type} (a : A) := a.  
  
Lemma id_2 : id 2 = 2.  
Proof. reflexivity. Qed.
```

# Coq: Inductive Types

Generalisation of ADT to dependent types:

```
Inductive vector (A : Type) : nat -> Type :=  
  | Nil : vector A 0  
  | Cons n : A -> vector A n -> vector A (1 + n).
```

# Mi-Cho-Coq

Coq formalisation of Michelson

- syntax
- semantics
- typing

Functional verification of a few contracts

# Verification of the Multisig

- Proven:

Characterisation of the relation between input and output:

`eval multisig input = Success output <-> R input`  
`output`

- Not proven:

Security property:

- signatures cannot be forged
- signatures sent to a multisig cannot be replayed

# Future work

- Prove more contracts
- Improve automation
- Formalise more of Michelson (gas, contract life, security)
- Higher-level languages with certified compilers
- Extract the Coq interpreter

# Conclusion

- Formal methods are complementary.
- Language design matters.