Mastering Michelson

Part 0: Outline

Raphaël Cauderlier

Nomadic Labs training
October 17, 2019

Outline

Mastering Michelson

Part 1: Getting started

Raphaël Cauderlier

Nomadic Labs training
October 17, 2019

Outline

Smart contracts in Tezos

Smart contracts are:

- a bag of tokens (the balance), a piece of code, a storage space
- all stored on the blockchain at a specific address

They:

- decide whether a transaction is accepted or rejected
- keep track of transactions in their storage,
- initiate transfers to other smart contracts,
- take a parameter and emit operations

Michelson vs. EVM

- Like EVM:
  - stack language
  - on-chain storage
  - gas model
  - Turing complete
- Unlike EVM:
  - static typing
  - atomic computations
  - explicit failure
  - strict syntax

Stack language

- low level enough for good intuition on gas consumption,
- ideal underlying model for formal verification,
- between a high level language and a typed bytecode,

Stack language

- instructions rewrite an input stack into an output stack,
- do not modify input values (immutable data structures),
- the contract rewrites the stack
  - from `pair parameter storage`
  - to `pair (list operation) storage`

Static typing

Instructions operate on a *stack*.

Each instruction pops 0, 1, or several elements on the top of the stack and pushes back 0, 1, or several elements on the stack.

For example:

- `SWAP` pops two elements `a` and `b` and pushes back `a`, and `b` on top of `a`
- `UNIT` pops nothing and pushes the constant `Unit`
- `DROP` pops an element and pushes nothing
- `NOT` pops a boolean and pushes its negation

The number and nature of the element popped and pushed can be seen in the type of the instruction

`SWAP`	::	`'a : 'b : 'A`	$\rightarrow$	`'b : 'a : 'A`
`SENDER`	::	`'A`	$\rightarrow$	`address : 'A`
`DROP`	::	`'a : 'A`	$\rightarrow$	`'A`
`NOT`	::	`bool : 'A`	$\rightarrow$	`bool : 'A`

Static typing

Data are typed.

- `PUSH int 2` vs. `PUSH mutez 2`
- `PUSH (list int) { 1 ; 2 ; 3 }`
  vs. `PUSH (set int) { 1 ; 2 ; 3 }`

Static typing

Data are typed.

- `PUSH int 2` vs. `PUSH mutez 2`
- `PUSH (list int) { 1 ; 2 ; 3 }`
  vs. `PUSH (set int) { 1 ; 2 ; 3 }`

Each time a piece of data enters a Michelson contract, its type must be given

- `PUSH` (and variants)
- `storage`
- `parameter`

Static typing

Data are typed.

- `PUSH int 2` vs. `PUSH mutez 2`
- `PUSH (list int) { 1 ; 2 ; 3 }`
  vs. `PUSH (set int) { 1 ; 2 ; 3 }`

Each time a piece of data enters a Michelson contract, its type must be given

- `PUSH` (and variants)
- `storage`
- `parameter`

The type of each instruction is determined by the type of its input

$$
\begin{array}{lrcl}
\texttt{ADD} & :: \quad \texttt{int : int : 'S} & \rightarrow & \texttt{int : 'S} \\
\texttt{ADD} & :: \quad \texttt{mutez : mutez : 'S} & \rightarrow & \texttt{mutez : 'S} \\
\texttt{SWAP} & :: \quad \texttt{'a : 'b : 'S} & \rightarrow & \texttt{'b : 'a : 'S}
\end{array}
$$

Static typing

Data are typed.

- `PUSH int 2` vs. `PUSH mutez 2`
- `PUSH (list int) { 1 ; 2 ; 3 }`
  vs. `PUSH (set int) { 1 ; 2 ; 3 }`

Each time a piece of data enters a Michelson contract, its type must be given

- `PUSH` (and variants)
- `storage`
- `parameter`

The type of each instruction is determined by the type of its input

$$\begin{array}{lrcl} \texttt{ADD} :: & \texttt{int : int : 'S} & \rightarrow & \texttt{int : 'S} \\ \texttt{ADD} :: & \texttt{mutez : mutez : 'S} & \rightarrow & \texttt{mutez : 'S} \\ \texttt{SWAP} :: & \texttt{'a : 'b : 'S} & \rightarrow & \texttt{'b : 'a : 'S} \end{array}$$

type-checking at origination + each call

Static typing

Data are typed.

- `PUSH int 2` vs. `PUSH mutez 2`
- `PUSH (list int) { 1 ; 2 ; 3 }`
  vs. `PUSH (set int) { 1 ; 2 ; 3 }`

Each time a piece of data enters a Michelson contract, its type must be given

- `PUSH` (and variants)
- `storage`
- `parameter`

The type of each instruction is determined by the type of its input

`ADD`	::	`int : int : 'S`	$\rightarrow$	`int : 'S`
`ADD`	::	`mutez : mutez : 'S`	$\rightarrow$	`mutez : 'S`
`SWAP`	::	`'a : 'b : 'S`	$\rightarrow$	`'b : 'a : 'S`

type-checking at origination + each call
Michelson interpreter cannot run on an ill-typed contract

Static typing

- sound: well-typed contracts do not go wrong,
- inter-contract type safety checks!

Strong type system:

- no nulls, no implicit casts, no overflow, no division by 0 etc.
- high level types: $\mathbb{Z}$, options, pairs, lists, immutable sets and maps,

Atomic computation

A Michelson contract runs completely before calling other contracts.
This avoids many re-entrancy bugs.
Contracts are not functions, they do not return values.
To get a value from another contract, we need continuation passing style.

Atomic computation

A Michelson contract runs completely before calling other contracts.
This avoids many re-entrancy bugs.
Contracts are not functions, they do not return values.
To get a value from another contract, we need continuation passing style.

$$\text{A} \to \text{B(request, address of A)} \to \text{A(answer)}$$

Explicit failure

All possible runtime failures:

- Explicit failure (`FAILWITH` instruction)
- Gas exhaustion
- Mutez overflow

No modular arithmetic, invalid opcode, invalid instruction, stack over/underflow at runtime.

Syntax

- as unambiguous as possible for humans,
- enforced indentation (alignment of sequences and arguments),
- enforced case: (`INSTR`, `Data`, `type`),
- enforced delimitation of code blocks

Types

- Arithmetic:
  `int`, `nat`, `mutez`, `timestamp`
- Compound types:
  `unit`, `bool`, `pair _ _`, `or _ _`, `option _`
- Addresses:
  `key_hash`, `address`, `contract _`
- Data structures:
  `list _`, `set _`, `map _ _`, `big_map _ _`
- Crypto:
  `bytes`, `key`, `signature`
- Other:
  `string`, `lambda _ _`, `operation`

Casts

$$
\begin{array}{llcl}
\texttt{INT} & :: & \texttt{nat} & \rightarrow & \texttt{int} \\
\texttt{ISNAT} & :: & \texttt{int} & \rightarrow & \texttt{option nat} \\
\texttt{ABS} & :: & \texttt{int} & \rightarrow & \texttt{nat}
\end{array}
$$

Casts

$$
\begin{array}{lclcl}
\texttt{INT} & :: & \texttt{nat} & \rightarrow & \texttt{int} \\
\texttt{ISNAT} & :: & \texttt{int} & \rightarrow & \texttt{option nat} \\
\texttt{ABS} & :: & \texttt{int} & \rightarrow & \texttt{nat}
\end{array}
$$

$$
\begin{array}{lcrcl}
\texttt{IMPLICIT\_ACCOUNT} & :: & \texttt{key\_hash} & \rightarrow & \texttt{contract unit} \\
\texttt{ADDRESS} & :: & \texttt{contract \_} & \rightarrow & \texttt{address} \\
\texttt{CONTRACT 'ty} & :: & \texttt{address} & \rightarrow & \texttt{option (contract 'ty)}
\end{array}
$$

Casts

$$
\begin{array}{llcll}
\texttt{INT} & :: & \texttt{nat} & \rightarrow & \texttt{int} \\
\texttt{ISNAT} & :: & \texttt{int} & \rightarrow & \texttt{option nat} \\
\texttt{ABS} & :: & \texttt{int} & \rightarrow & \texttt{nat}
\end{array}
$$

$$
\begin{array}{llrll}
\texttt{IMPLICIT\_ACCOUNT} & :: & \texttt{key\_hash} & \rightarrow & \texttt{contract unit} \\
\texttt{ADDRESS} & :: & \texttt{contract \_} & \rightarrow & \texttt{address} \\
\texttt{CONTRACT 'ty} & :: & \texttt{address} & \rightarrow & \texttt{option (contract 'ty)}
\end{array}
$$

$$
\begin{array}{llrll}
\texttt{PACK} & :: & \texttt{\_} & \rightarrow & \texttt{bytes} \\
\texttt{UNPACK 'ty} & :: & \texttt{bytes} & \rightarrow & \texttt{option 'ty}
\end{array}
$$

CLI

- `tezos-client typecheck script <file>`
- `tezos-client run script <file> on storage <data> and input <data> [--trace-stack]`
- `tezos-client originate contract <contract_name> transferring <balance> from <payer> running <file> --init <storage> --burn-cap <cap>`
- `tezos-client transfer <amount> from <sender> to <contract> --arg <parameter> [--burn-cap <cap>]`

Editors

- Emacs
  https://gitlab.com/tezos/tezos/tree/master/emacs
- IntelliJ (and derived editors like PyCharm)
  https://www.plugin-dev.com/plugins/tezos-michelson/

Try-Michelson

`https://try-michelson.tzalpha.net/`
(or locally on the Training VM)

- Web editor
- Type-checking
- Simulation (incl. inter-contract interaction)

Reference manual

Michelson whitepaper:

- Athens: `https://tezos.gitlab.io/master/whitedoc/michelson.html`
- Babylon: `https://tezos.gitlab.io/zeronet/whitedoc/michelson.html`

WIP Michelson reference:

- `https://arvidj.eu/michelson/`

Reference manual

Michelson whitepaper:

- Athens: `https://tezos.gitlab.io/master/whitedoc/michelson.html`
- Babylon: `https://tezos.gitlab.io/zeronet/whitedoc/michelson.html`

WIP Michelson reference:

- `https://arvidj.eu/michelson/`

Questions?

Mastering Michelson

Part 2: Michelson by example (and Babylon novelties)

Raphaël Cauderlier

Nomadic Labs training
October 17, 2019

Outline

Deposit contract

First example: a deposit contract

- Storage contains the owner's address
- Can recieve tokens from anybody
- The owner can withdraw tokens

Goal

- Starting example
- Simple authentication
- Token transfers

Base types

- `unit`: trivial type (the only value is `Unit`)
- `bool`: either `True` or `False`
- `string`: character strings surrounded by double quotes `"`

Control structures

- No operation: `{}` :: `'A` $\rightarrow$ `'A`
- Sequence: `{ code`$_1$ `; code`$_2$ `; ... ; code`$_n$ `}` :: `A`$_1$ $\rightarrow$ `A`$_{n+1}$
  if `code`$_i$ :: `A`$_i$ $\rightarrow$ `A`$_{i+1}$
- Explicit failure: `FAILWITH` :: `'a : 'A` $\rightarrow$ `'B`
- Conditional: `IF { bt } { bf }` :: `bool : 'A` $\rightarrow$ `'B`
  if `bt, bf` :: `'A` $\rightarrow$ `'B`
- loop: `LOOP { body }` :: `bool : 'A` $\rightarrow$ `'A`
  if `body` :: `'A` $\rightarrow$ `bool : 'A`

Stack manipulation

`PUSH 'a x`	::	`'A`	$\rightarrow$	`'a : 'A`
`DROP`	::	`'a : 'A`	$\rightarrow$	`'A`
`DUP`	::	`'a : 'A`	$\rightarrow$	`'a : 'a : 'A`
`SWAP`	::	`'a : 'b : 'A`	$\rightarrow$	`'b : 'a : 'A`
`DIP { code }`	::	`'a : 'A`	$\rightarrow$	`'a : 'B`

if code :: `'A` $\rightarrow$ `'B`

Stack manipulation example

```
PUSH string "foo"; PUSH string "bar";
DIP { DUP; PUSH string "baz"}; SWAP; DROP
```

Stack manipulation example

```
PUSH string "foo"; PUSH string "bar";
DIP { DUP; PUSH string "baz"}; SWAP; DROP
```

"bar"
"foo"
"foo"

Comparison

- `COMPARE` :: `'a : 'a : 'A` $\rightarrow$ `int : 'A` Returns −1, 0, or 1
- `EQ, NEQ, LT, GT, LE, GE` :: `int : 'A` $\rightarrow$ `bool : 'A`

Sender authentication

<sender address>
<authorized address>
<remaining stack> ...

→

<remaining stack> ...

Sender authentication

<sender address>
<authorized address>
<remaining stack> ...

$\rightarrow$

<remaining stack> ...

```
COMPARE; EQ; IF { FAILWITH } {}
```

Operations

- `address`: untyped address
- `contract 'a`: smart contract expecting a parameter of type `'a`
- `mutez`: amount of tokens (in μꜩ)

$\mathtt{TRANSFER\_TOKENS} :: \mathtt{'a} : \mathtt{mutez} : \mathtt{contract}\ \mathtt{'a} : \mathtt{'A} \rightarrow \mathtt{operation} : \mathtt{'A}$

$$
\begin{array}{llcl}
\mathtt{AMOUNT}, \mathtt{BALANCE} :: & \mathtt{'A} & \rightarrow & \mathtt{mutez} : \mathtt{'A} \\
\mathtt{SENDER} :: & \mathtt{'A} & \rightarrow & \mathtt{address} : \mathtt{'A} \\
\mathtt{SELF} :: & \mathtt{'A} & \rightarrow & \mathtt{contract}\ \mathtt{'a} : \mathtt{'A}
\end{array}
$$

$$\mathtt{CONTRACT}\ \mathtt{'a} :: \mathtt{address} : \mathtt{'A} \rightarrow \mathtt{option}\ (\mathtt{contract}\ \mathtt{'a}) : \mathtt{'A}$$

$$\mathtt{ADDRESS} :: \mathtt{contract}\ \mathtt{'a} : \mathtt{'A} \rightarrow \mathtt{address} : \mathtt{'A}$$

Pairs, ors, and options

- `pair 'a 'b`: a pair of values `Pair x y` with `x` :: `'a` and `y` :: `'b`
- `or 'a 'b`: either `Left x` with `x` :: `'a` or `Right y` with `y` :: `'b`
- `option 'a`: either `Some x` with `x` :: `'a` or `None`

Constructors

$$
\begin{array}{llcccccl}
\texttt{PAIR} & :: & \texttt{'a} : & \texttt{'b} : & \texttt{'A} & \rightarrow & \texttt{pair 'a 'b} : \texttt{'A} \\
\texttt{LEFT 'b} & :: & & \texttt{'a} : & \texttt{'A} & \rightarrow & \texttt{or 'a 'b} : \texttt{'A} \\
\texttt{RIGHT 'a} & :: & & \texttt{'b} : & \texttt{'A} & \rightarrow & \texttt{or 'a 'b} : \texttt{'A} \\
\texttt{NONE 'a} & :: & & & \texttt{'A} & \rightarrow & \texttt{option 'a} : \texttt{'A} \\
\texttt{SOME} & :: & & \texttt{'a} : & \texttt{'A} & \rightarrow & \texttt{option 'a} : \texttt{'A}
\end{array}
$$

Destructors

$$\begin{array}{llcl} \texttt{CAR} :: & \texttt{pair 'a 'b : 'A} & \rightarrow & \texttt{'a : 'A} \\ \texttt{CDR} :: & \texttt{pair 'a 'b : 'A} & \rightarrow & \texttt{'b : 'A} \end{array}$$

$$\texttt{IF\_LEFT \{ bl \} \{ br \}} :: \quad \texttt{or 'a 'b : 'A} \quad \rightarrow \quad \texttt{'B}$$

$$\text{if } \texttt{bl} :: \texttt{'a : 'A} \rightarrow \texttt{'B}, \texttt{br} :: \texttt{'b : 'A} \rightarrow \texttt{'B}$$

$$\texttt{IF\_NONE \{ bn \} \{ bs \}} :: \quad \texttt{option 'a : 'A} \quad \rightarrow \quad \texttt{'B}$$

$$\text{if } \texttt{bn} :: \texttt{'A} \rightarrow \texttt{'B}, \texttt{bs} :: \texttt{'a : 'A} \rightarrow \texttt{'B}$$

$$\texttt{LOOP\_LEFT \{ body \}} :: \quad \texttt{or 'a 'b : 'A} \quad \rightarrow \quad \texttt{'b : 'A}$$

$$\text{if } \texttt{body} :: \texttt{'a : 'A} \rightarrow \texttt{or 'a 'b : 'A}$$

Parameters and storage

3 sections: *storage*, *parameter*, *code*

- *storage* and *parameter* are types.
- *code* :: `pair parameter storage : []` → `pair (list operation) storage`

Example: deposit contract

Example: deposit contract

```
parameter: (or unit mutez);
storage: address;
code:
  { DUP; CAR; DIP {CDR};
    IF_LEFT
      { # Deposit
        DROP; NIL operation }
      { # Withdraw
        DIP { DUP;
              # Access control:
              #  only the stored address can withdraw
              DUP; SENDER; COMPARE; EQ; IF {FAILWITH} {}
              CONTRACT unit; IF_NONE {FAILWITH} {}};
        PUSH unit Unit; TRANSFER_TOKENS;
        NIL operation; SWAP; CONS};
    PAIR}
```

Vote contract

- User can pay 5ꜩ to vote
- Fixed set of options to vote for

Goal

- paywall
- arithmetics
- manipulation of a `map`

Integer arithmetics

- `int`: arbitrary-precision integers
- `nat`: arbitrary-precision naturals

Arithmetics

- `ABS` :: `int : 'A` $\rightarrow$ `nat : 'A`
- `NEG` :: `nat : 'A` $\rightarrow$ `int : 'A`
- `NEG` :: `int : 'A` $\rightarrow$ `int : 'A`
- `ADD` :: `nat : nat : 'A` $\rightarrow$ `nat : 'A`
- `ADD` :: `nat : int : 'A` $\rightarrow$ `int : 'A`
- `ADD` :: `int : nat : 'A` $\rightarrow$ `int : 'A`
- `ADD` :: `int : int : 'A` $\rightarrow$ `int : 'A`
- `SUB`, `MUL`
- `EDIV` :: `nat : nat : 'A` $\rightarrow$ `option(pair nat nat) : 'A`
- `EDIV` :: `nat : int : 'A` $\rightarrow$ `option(pair int nat) : 'A`
- `EDIV` :: `int : nat : 'A` $\rightarrow$ `option(pair int nat) : 'A`
- `EDIV` :: `int : int : 'A` $\rightarrow$ `option(pair int nat) : 'A`
- bitwise operations
- `LSL, LSR` :: `nat : nat : 'A` $\rightarrow$ `nat : A`

Data structures

- `list 'a`: a list with elements of type `'a`
- `set 'a`: a finite set of elements of type `'a`
- `map 'key 'val`: a finite map
- `big_map 'key 'val`: same but lazily deserialized

Instructions

- `NIL 'a`
- `EMPTY_SET 'elt`
- `EMPTY_MAP 'key 'val`
- `CONS`
- `UPDATE`
- `IF_CONS { bc } { bn }`
- `MEM`
- `MAP { body }`
- `SIZE`
- `ITER { body }`

Example: vote contract

Example: vote contract

```
storage (map string nat);
parameter string;
code { DUP; DIP {CDR; DUP}; CAR; DUP;
       DIP { GET;
             IF_NONE
               { PUSH string "Not a valid option";
                 FAILWITH}
               {};
             PUSH nat 1; ADD; SOME};
       UPDATE;
       NIL operation; PAIR }
```

Overview of Michelson changes in Babylon

More details here:

`https://blog.nomadic-labs.com/michelson-updates-in-005.html`

Gas

- Micro bench-marking
- Better gas computation
- `STEPS_TO_QUOTA` is deprecated

Account reorganisation

- `CREATE_ACCOUNT` is deprecated
- `CREATE_CONTRACT` now takes fewer arguments

`CREATE_CONTRACT { storage 'g ; parameter 'p ; code ... }` Parameters

Parameter type	Parameter	in Athens	in Babylon
`key_hash`	manager	Yes	No
`option key_hash`	initial delegate	Yes	Yes
`bool`	spendable flag	Yes	No
`bool`	delegatable flag	Yes	No
`mutez`	initial balance	Yes	Yes
`'g`	initial storage	Yes	Yes

Stack instructions

```
DIG n            ::    'a{1} : ... : 'a{n-1} : 'a{n} : 'C
                 →     'a{n} : 'a{1} : ... : 'a{n-1} : 'C

DUG n            ::    'a{1} : 'a{2} : ... : 'a{n} : 'C
                 →     'a{2} : ... : 'a{n} : 'a{1} : 'C

DROP n           ::    'a{1} : ... : 'a{n} : 'C
                 →     'C

DIP n { code }   ::    'a{1} : 'a{2} : ... : 'a{n} : 'C
                 →     'a{1} : 'a{2} : ... : 'a{n} : 'D
```

if code :: 'C → 'D

Partial application

Already in Athens:

```
EXEC ::    'a : lambda 'a 'b : 'C   →   'b : 'C
```

New in Babylon:

```
APPLY  ::  'a : lambda (pair 'a 'b) 'c : 'C
       →                lambda 'b 'c : 'C
```

Multiple big maps

In Athens:

- Each contract can store at most one `big_map`,
  `storage pair (big_map 'a 'b) 'c`
- `big_maps` can not be transferred

In Babylon:

- Both restrictions removed

Entrypoints

Already in Athens:

- A contract with 3 entrypoints of type `'a`, `'b`, and `'c` can be encoded as:

`(or 'a (or 'b 'c))`.

Entrypoints

Already in Athens:

- A contract with 3 entrypoints of type `'a`, `'b`, and `'c` can be encoded as:

`(or 'a (or 'b 'c))`.

- Entrypoints can be named using %-annotations:

`(or ('a %entry_A) (or ('b %entry_B) ('c %entry_C)))`.

Entrypoints

Already in Athens:

- A contract with 3 entrypoints of type `'a`, `'b`, and `'c` can be encoded as:

```
(or 'a (or 'b 'c)).
```

- Entrypoints can be named using %-annotations:

```
(or ('a %entry_A) (or ('b %entry_B) ('c %entry_C))).
```

- But two problems:
  - To call an entrypoint of a contract, we need to know where the entrypoint is in the `or`-tree
  - and the types of all entrypoints
  - interfaces are not extensible

Entrypoints

In Athens:

$$
\begin{array}{llr}
\texttt{CONTRACT 'ty} & :: & \texttt{address : 'S} \\
 & \rightarrow & \texttt{option (contract 'ty) : 'S} \\
\texttt{SELF} & :: & \texttt{'S} \\
 & \rightarrow & \texttt{contract <param\_type> : 'S}
\end{array}
$$

In Babylon:

$$
\begin{array}{llr}
\texttt{CONTRACT \%<entry> 'ty} & :: & \texttt{address : 'S} \\
 & \rightarrow & \texttt{option (contract 'ty) : 'S} \\
\texttt{SELF \%<entry>} & :: & \texttt{'S} \\
 & \rightarrow & \texttt{contract <entry\_ty> : 'S}
\end{array}
$$

`%default` is used if `%<entry>` is omitted.

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

Goal

- advanced, signature-based authentication
- security

Types

- `bytes`: non-readable sequence of bytes
- `key`: cryptographic public key
- `signature`: cryptographic signature

Instructions

- `PACK` :: `'a : 'A` $\rightarrow$ `bytes : 'A`
- `UNPACK 'a` :: `bytes` $\rightarrow$ `option 'a : 'A`
- `BLAKE2B, SHA256, SHA512` :: `bytes : 'A` $\rightarrow$ `bytes : 'A`
- `CHECK_SIGNATURE` :: `key : signature : bytes : 'A` $\rightarrow$ `bool : 'A`

Example: multisig

Example: multisig

```
parameter (pair (lambda unit (list operation)) (list (option signature)));
storage (pair nat (list key));
code
  { DUP; CDR; DIP {CAR}; DUP;
    DIP { SWAP; DUP; CAR; DIP {CDR}; DUP; DIP {SWAP}; PACK; SWAP };
    DUP; CAR;
    DIP
      { CDR; PUSH nat 0; SWAP;
        ITER
          { DIP {SWAP}; SWAP;
            IF_CONS
              { IF_SOME
                  { SWAP;
                    DIP { SWAP ; DIP {DIP {DIP {DUP}; SWAP}};
                          DIP {DIP {DIP {DUP}; SWAP}; SWAP}; SWAP;
                          DIP {CHECK_SIGNATURE}; SWAP;
                          IF {DROP} {PUSH string "bad signature"; FAILWITH};
                          PUSH nat 1; ADD }}
                  { SWAP; DROP }}
              { PUSH string "signature list is too short"; FAILWITH };
            SWAP }};
    COMPARE; LE; IF {} {PUSH string "not enough signatures"; FAILWITH};
    IF_CONS {PUSH string "signature list is too long"; FAILWITH} {}; DROP;
    UNIT; EXEC; PAIR }
```

Weather Insurance

- Insurance contract
  - Take deposits
  - Refunds the insurance or its client depending on the rain level
- Oracle contract
  - stores rain levels
  - paid service

Goal

- communication between smart contracts
- deposits and refunds

Continuation passing style

$$A \rightarrow B(\text{request, address of A}) \rightarrow A(\text{answer})$$

Continuation passing style

A $\rightarrow$ B(request, address of A) $\rightarrow$ A(answer)

- Smart contract A (insurance contract) needs an extra entrypoint for receiving the answer

Continuation passing style

$$A \rightarrow \text{B(request, address of A)} \rightarrow \text{A(answer)}$$

- Smart contract A (insurance contract) needs an extra entrypoint for receiving the answer
- Smart contract B (oracle) also needs two entrypoints

Oracle

https://gitlab.com/nomadic-labs/mi-cho-coq/blob/master/src/contracts/mutually_calling/oracle.tz

Insurance

https://gitlab.com/nomadic-labs/mi-cho-coq/blob/master/src/contracts/mutually_calling/weather_insurance_on_chain_oracle.tz

Mastering Michelson

Part 3: Formal Methods

Raphaël Cauderlier

Nomadic Labs training
October 17, 2019

Outline

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