Mastering Michelson

Part 0: Outline

Raphaël Cauderlier (presented by Arvid Jakobsson)

Nomadic Labs training
February 12, 2020

Outline

Mastering Michelson

Part 1: Getting started

Raphaël Cauderlier (presented by Arvid Jakobsson)

Nomadic Labs training
February 12, 2020

Outline

1. Introduction
2. Type System
3. Tooling
4. References

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`
`UNIT`	::	`'A`	$\rightarrow$	`unit : '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

```
ADD  ::       int : int   : 'S  →  int : 'S
ADD  ::   mutez : mutez : 'S  →  mutez : 'S
SWAP ::          'a : 'b : 'S  →  'b : 'a : 'S
```

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   →   int : 'S
ADD  ::      mutez : mutez : 'S   →   mutez : 'S
SWAP ::          'a : 'b   : 'S   →   'b : 'a : 'S
```

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     →   int : 'S
ADD  ::   mutez : mutez : 'S →   mutez : 'S
SWAP ::   'a : 'b : 'S       →   '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 reentrancy 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 reentrancy bugs.
Contracts are not functions, they do not return values.
To get a value from another contract, we need continuation passing style.

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

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 'a 'b`, `or 'a 'b`, `option 'a`
- Addresses:
  `key_hash`, `address`, `contract 'a`
- Data structures:
  `list 'a`, `set 'a`, `map 'a 'b`, `big_map 'a 'b`
- Cryptography:
  `bytes`, `key`, `signature`
- Other:
  `string`, `lambda 'a 'b`, `operation`

Casts

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

Casts

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

$$
\begin{aligned}
\texttt{IMPLICIT_ACCOUNT} &:: \texttt{key_hash} \rightarrow \texttt{contract unit} \\
\texttt{ADDRESS} &:: \texttt{contract _} \rightarrow \texttt{address} \\
\texttt{CONTRACT 'ty} &:: \texttt{address} \rightarrow \texttt{option (contract 'ty)}
\end{aligned}
$$

Casts

```
INT    :: nat → int
ISNAT  :: int → option nat
ABS    :: int → nat

IMPLICIT_ACCOUNT :: key_hash → contract unit
ADDRESS          :: contract _ → address
CONTRACT 'ty     :: address → option (contract 'ty)

PACK        :: 'ty → bytes
UNPACK 'ty  :: bytes → option 'ty
```

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>]
- tezos-client expand macros in <src>

Editors

- Emacs
  https://gitlab.com/tezos/tezos/tree/master/emacs
- Vim
  https://github.com/rnestler/michelson.vim
- IntelliJ (and derived editors like PyCharm)
  https://://www.plugin-dev.com/plugins/tezos-michelson/
- VS Code
  https://gitlab.com/kinokasai/vscode-michelson

Try-Michelson

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

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

References

Michelson whitepaper:

- `https://tezos.gitlab.io/whitedoc/michelson.html`

Michelson reference:

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

Tezos Stack Exchange:

- `https://tezos.stackexchange.com/`

References

Michelson whitepaper:

- https://tezos.gitlab.io/whitedoc/michelson.html

Michelson reference:

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

Tezos Stack Exchange:

- https://tezos.stackexchange.com/

Questions?

Mastering Michelson

Part 2: Michelson by Example

Raphaël Cauderlier (presented by Arvid Jakobsson)

Nomadic Labs training
February 12, 2020

Outline

1. First example: Deposit
2. Second example: Voting
3. Third example: Multisig
4. Fourth example: Weather Insurance

Deposit contract

First example: a deposit contract

- Storage contains the owner's address

Two entry points:

1. Deposit tokens (callable by anybody)
2. Withdraw tokens (only callable by the owner)

Goal

- Starting example
- Simple, address-based 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` → `'A`
- Sequence: `{ code`$_1$ `; code`$_2$ `; ... ; code`$_n$ `}` :: `A`$_1$ → `A`$_{n+1}$
  if `code`$_i$ :: `A`$_i$ → `A`$_{i+1}$
- Explicit failure: `FAILWITH` :: `'a : 'A` → `'B`
- Conditional: `IF { bt } { bf }` :: `bool : 'A` → `'B`
  if `bt, bf` :: `'A` → `'B`
- Loop: `LOOP { body }` :: `bool : 'A` → `'A`
  if `body` :: `'A` → `bool : 'A`

Stack manipulation

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

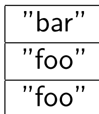
if code :: 'A → 'B

Stack manipulation example

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

Stack manipulation example

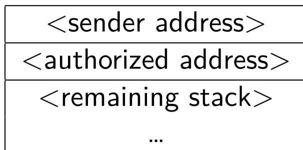
```
PUSH string "foo"; PUSH string "bar";
DIP { DUP; PUSH string "baz"}; SWAP; DROP
```



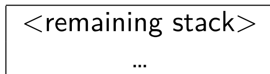
Comparison

- `COMPARE` :: `'a : 'a : 'A` → `int : 'A`
  - Returns: $\begin{cases} -1 & \text{if } x < y \\ 0 & \text{if } x = y \\ 1 & \text{if } x > y \end{cases}$
- `EQ`, `NEQ`, `LT`, `GT`, `LE`, `GE` :: `int : 'A` → `bool : 'A`

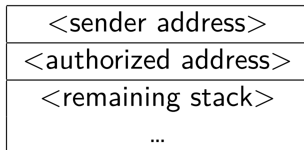
Address-based authentication



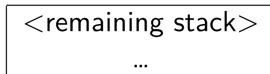
$\rightarrow$



Address-based authentication



→




```
COMPARE; NEQ; IF { FAILWITH } {}
```

Operations

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

$$\texttt{TRANSFER_TOKENS} :: \texttt{'a : mutez : contract 'a : 'A} \rightarrow \texttt{operation : 'A}$$

$$\texttt{AMOUNT}, \texttt{BALANCE} :: \texttt{'A} \rightarrow \texttt{mutez : 'A}$$
$$\texttt{SENDER} :: \texttt{'A} \rightarrow \texttt{address : 'A}$$
$$\texttt{SELF} :: \texttt{'A} \rightarrow \texttt{contract 'a : 'A}$$

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

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

Pairs, unions, and options

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

Constructors

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

Destructors

$$\texttt{CAR} :: \quad \texttt{pair 'a 'b : 'A} \quad \rightarrow \quad \texttt{'a : 'A}$$
$$\texttt{CDR} :: \quad \texttt{pair 'a 'b : 'A} \quad \rightarrow \quad \texttt{'b : 'A}$$

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

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

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

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

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

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

Parameters and storage

A contract has 3 sections: *storage*, *parameter* and *code*

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

Example: Deposit contract

Example: Deposit contract

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

Vote contract

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

Goal

This example demonstrates:

- paywall
- arithmetics
- manipulation of a `map`

Integer arithmetics

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

Arithmetics

- ABS :: `int : 'A → nat : 'A`
- NEG :: `nat : 'A → int : 'A`
- NEG :: `int : 'A → int : 'A`
- ADD :: `nat : nat : 'A → nat : 'A`
- ADD :: `nat : int : 'A → int : 'A`
- ADD :: `int : nat : 'A → int : 'A`
- ADD :: `int : int : 'A → int : 'A`
- SUB, MUL
- EDIV :: `nat : nat : 'A → option(pair nat nat) : 'A`
- EDIV :: `nat : int : 'A → option(pair int nat) : 'A`
- EDIV :: `int : nat : 'A → option(pair int nat) : 'A`
- EDIV :: `int : int : 'A → option(pair int nat) : 'A`

bitwise operations:

- `LSL, LSR` :: `nat : nat : 'A → 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 { # Paywall
       AMOUNT; PUSH mutez 5000000;
       COMPARE; LE; IF { } {FAILWITH};
       DUP; DIP {CDR; DUP}; CAR; DUP;
       # Verify and update vote
       DIP { GET;
             IF_NONE
               { PUSH string "Not a valid option";
                 FAILWITH}
               {};
             PUSH nat 1; ADD; SOME};
       UPDATE;
       NIL operation; PAIR }
```

The Multisig contract

- n persons share the ownership of the contract.
- They agree on a threshold t (an integer).
- To perform an action with the contract, at least t owners must agree.
- Possible actions:
  - Return a list of operations (to be run atomically)

Goal

This example demonstrates:

- Advanced, signature-based authentication
- Security

Types

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

Instructions

- `PACK` :: `'a : 'A` → `bytes : 'A`
  Serialize a value
- `UNPACK 'a` :: `bytes` → `option 'a : 'A`
- `BLAKE2B`, `SHA256`, `SHA512` :: `bytes : 'A` → `bytes : 'A`
  Hash a byte sequence
- `CHECK_SIGNATURE` :: `key : signature : bytes : 'A` → `bool : 'A`
  Verify a signature

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
  - Takes deposits from clients
  - Refunds the insurer or its client depending on the rain level
- Oracle contract
  - Stores rain levels
  - A paid service

Goal

This example demonstrates:

- Communication between smart contracts
- Deposits and refunds

Continuation-Passing Style

$\texttt{A} \rightarrow \texttt{B(lookup, address of A)} \rightarrow \texttt{A(answer)}$

Continuation-Passing Style

`A → B(lookup, address of A) → A(answer)`

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

Continuation-Passing Style

`A` $\rightarrow$ `B(lookup, address of A)` $\rightarrow$ `A(answer)`

- Smart contract A (insurance contract) needs an extra entrypoint for receiving the answer:
  - `callback(answer)`
- Smart contract B (oracle) needs:
  - `query(lookup, callback)`: request that weather data for lookup is sent to callback against fee

Continuation-Passing Style

```
A → B(lookup, address of A) → A(answer)
```

- Smart contract A (insurance contract) needs an extra entrypoint for receiving the answer:
  - `callback(answer)`
- Smart contract B (oracle) needs:
  - `query(lookup, callback)`: request that weather data for lookup is sent to callback against fee

```
A  →  B.query(lookup, contract %callback A)
   →  A.callback(answer)
```

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 (presented by Arvid Jakobsson)

Nomadic Labs training
February 12, 2020

Outline

1. Formal Methods for Michelson smart contracts

Motivation

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

Before uploading them, we want to be sure there are no bugs in them!

Motivation

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

Before uploading them, we want to be sure there are no bugs 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 behavior 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 behavior 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: e.g. something happens eventually.
- Challenges:
  - 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
- Challenges:
  - False alarms

Deductive Verification

Reduce to the *theorem proving* problem.

- Specifications:
  - *Functional* properties:

    ```
    { precondition } Program { postcondition }
    ```

  - Very rich logics
- Challenges:
  - 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: Participants either win the auction or lose no money
  - Tool: Cubicle Model Checker
- Abstract Interpretation:
  - Bounds on gas
  - Token freeze
- Deductive Verification:
  - Example: multisig
  - Spec: multisig succeeds IFF enough valid signatures
  - Tool: Mi-Cho-Coq