Contents

The Tezos Platform

Tezos

- decentralized blockchain platform that supports smart contracts
- blockchain 3.0
  - live-upgrade (no hard forks)
  - on-chain governance
- proof of stake
- emphasis on contracts rather than cryptocurrency
- mathematically verified components

Excursion: On-chain Governance

Definition

We call **governance** any system for managing and implementing changes to cryptocurrency blockchains. In **on-chain governance**, rules for instituting changes are encoded into the blockchain protocol. Developers propose changes through code updates and each node votes on whether to accept or reject the proposed change.

Excursion: On-chain Governance

Definition

We call **governance** any system for managing and implementing changes to cryptocurrency blockchains. In **on-chain governance**, rules for instituting changes are encoded into the blockchain protocol. Developers propose changes through code updates and each node votes on whether to accept or reject the proposed change.

Background

- Bitcoin and Ethereum: informal governance, relies on decentralized ethos.
- changes are proposed by developers who seek consensus with the main stakeholders
- ideally, the consensus encompasses developers, miners, and users
- but actually, it's developers and miners

Examples for On-Chain Governance

Tezos

- self-amending ledger
- proposed changes are implemented and rolled out on a test chain
- if planned changes are successful, they are finalized to a production chain
- otherwise, they are rolled back.

Examples for On-Chain Governance

Tezos

- self-amending ledger
- proposed changes are implemented and rolled out on a test chain
- if planned changes are successful, they are finalized to a production chain
- otherwise, they are rolled back.

Dfinity

- mission: building the world's biggest virtual computer based on blockchain
- goal: more flexible than "code is law", but avoiding hard forks
- hardcoded constitution
- triggers passive and active actions
  - passive: increase in reward size for blocks
  - active: quarantining parts of the network for updates or roll backs

Tezos Smart Contracts

A decentralized platform that supports for smart contracts

- registered together with a private data storage:
  only the contract can interact, but the data are publicly visible.
- executed by performing specific transactions to their associated account
- data passed as program parameters and viewed as a procedure call
- account based
- smart contract language: Michelson

Michelson Smart Contracts

Tezos's native smart contract language

- stack-based, high-level data types, strict static type checking
  - the types of the input and output stack are fixed and monomorphic
  - the program is typechecked before it's injected into the system
  - $\Rightarrow$ contract execution cannot fail because an instruction has been executed on a stack of unexpected length or contents
- a sequence of instructions
  - input: stack resulting from the previous instruction
  - rewrites it for the next one
  - all values are immutable and garbage collected.

Reminder: Stack machines

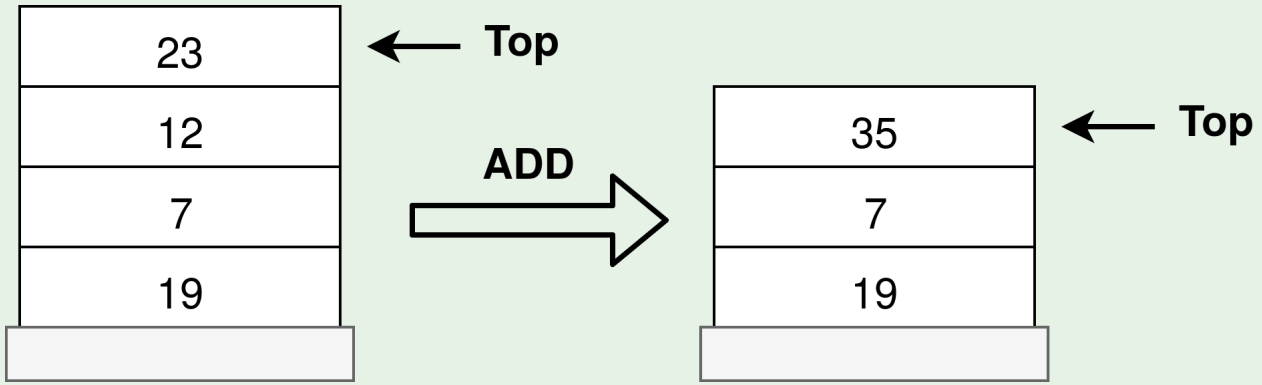
Stacks: last-in, first-out management of temporary values

example stack :: $23 : 12 : 7 : 19 : [\ ]$

Instructions: inputs from the stack, and results placed in the stack

ADD

ADD / a : b : S $\Rightarrow$ (a + b) : S



ADD / $23 : 12 : 7 : 19 : [\ ]$

$\Rightarrow (23 + 12) : 7 : 19 : [\ ]$

$\Rightarrow 35 : 7 : 19 : [\ ]$

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: `'a: 'S → 'a: 'a: 'S`
  - the notation `'a` (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation `'S` stands for any stack (type)

  example: the instruction ADD :: `int: int: 'S → int: 'S`
- types of subsequent instructions must compose

  example: DUP; ADD

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: `'a: 'S → 'a: 'a: 'S`
  - the notation `'a` (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation `'S` stands for any stack (type)

  example: the instruction ADD :: `int: int: 'S → int: 'S`
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: `'a: 'S → 'a: 'a: 'S` ; `int: int: 'S → int: 'S`

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: `'a: 'S → 'a: 'a: 'S`
  - the notation `'a` (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation `'S` stands for any stack (type)

  example: the instruction ADD :: `int: int: 'S → int: 'S`
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: `'a: 'S → 'a: 'a: 'S` ; `int: int: 'S → int: 'S`
  - these types only fit together if `'a = int`

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: `'a: 'S → 'a: 'a: 'S`
  - the notation `'a` (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation `'S` stands for any stack (type)

  example: the instruction ADD :: `int: int: 'S → int: 'S`
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: `'a: 'S → 'a: 'a: 'S` ; `int: int: 'S → int: 'S`
  - these types only fit together if `'a = int`
  - `'S` does not matter and remains variable

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: `'a: 'S → 'a: 'a: 'S`
  - the notation `'a` (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation `'S` stands for any stack (type)

  example: the instruction ADD :: `int: int: 'S → int: 'S`
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: `'a: 'S → 'a: 'a: 'S` ; `int: int: 'S → int: 'S`
  - these types only fit together if `'a = int`
  - `'S` does not matter and remains variable
  - DUP; ADD :: `int: 'S → int: int: 'S` ; `int: int: 'S → int: 'S`

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: $'a: 'S \rightarrow 'a: 'a: 'S$
  - the notation $'a$ (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation $'S$ stands for any stack (type)

  example: the instruction ADD :: $int: int: 'S \rightarrow int: 'S$
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: $'a: 'S \rightarrow 'a: 'a: 'S$; $int: int: 'S \rightarrow int: 'S$
  - these types only fit together if $'a = int$
  - $'S$ does not matter and remains variable
  - DUP; ADD :: $int: 'S \rightarrow int: int: 'S$; $int: int: 'S \rightarrow int: 'S$
  - the intermediate types fit together

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: $'a: 'S \rightarrow 'a: 'a: 'S$
  - the notation $'a$ (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation $'S$ stands for any stack (type)

  example: the instruction ADD :: $int: int: 'S \rightarrow int: 'S$
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: $'a: 'S \rightarrow 'a: 'a: 'S$; $int: int: 'S \rightarrow int: 'S$
  - these types only fit together if $'a = int$
  - $'S$ does not matter and remains variable
  - DUP; ADD :: $int: 'S \rightarrow int: int: 'S$; $int: int: 'S \rightarrow int: 'S$
  - the intermediate types fit together
  - DUP; ADD :: $int: 'S \rightarrow$ $int: int: 'S$; $int: int: 'S$ $\rightarrow int: 'S$

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: $'a: 'S \rightarrow 'a: 'a: 'S$
  - the notation $'a$ (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation $'S$ stands for any stack (type)

  example: the instruction ADD :: $int: int: 'S \rightarrow int: 'S$
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: $'a: 'S \rightarrow 'a: 'a: 'S$; $int: int: 'S \rightarrow int: 'S$
  - these types only fit together if $'a = int$
  - $'S$ does not matter and remains variable
  - DUP; ADD :: $int: 'S \rightarrow int: int: 'S$; $int: int: 'S \rightarrow int: 'S$
  - the intermediate types fit together
  - DUP; ADD :: $int: 'S \rightarrow$ $int: int: 'S$; $int: int: 'S$ $\rightarrow int: 'S$
  - . . . can be elided

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: $'a: 'S \rightarrow 'a: 'a: 'S$
  - the notation $'a$ (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation $'S$ stands for any stack (type)

  example: the instruction ADD :: $int: int: 'S \rightarrow int: 'S$
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: $'a: 'S \rightarrow 'a: 'a: 'S$; $int: int: 'S \rightarrow int: 'S$
  - these types only fit together if $'a = int$
  - $'S$ does not matter and remains variable
  - DUP; ADD :: $int: 'S \rightarrow int: int: 'S$; $int: int: 'S \rightarrow int: 'S$
  - the intermediate types fit together
  - DUP; ADD :: $int: 'S \rightarrow$ $int: int: 'S$; $int: int: 'S$ $\rightarrow int: 'S$
  - ... can be elided
  - DUP; ADD :: $int: 'S \rightarrow int: 'S$

Michelson Smart Contracts

- Michelson ascribes a **type** to each stack instruction
- The type describes the values on the stack before and after the instruction.

  example: the instruction DUP :: $'a : 'S \rightarrow 'a : 'a : 'S$
  - the notation $'a$ (a type variable) stands for any type (cf. generics in Java or TypeScript)
  - the notation $'S$ stands for any stack (type)

  example: the instruction ADD :: $int : int : 'S \rightarrow int : 'S$
- types of subsequent instructions must compose

  example: DUP; ADD
  - :: $'a : 'S \rightarrow 'a : 'a : 'S$; $int : int : 'S \rightarrow int : 'S$
  - these types only fit together if $'a = int$
  - $'S$ does not matter and remains variable
  - DUP; ADD :: $int : 'S \rightarrow int : int : 'S$; $int : int : 'S \rightarrow int : 'S$
  - the intermediate types fit together
  - DUP; ADD :: $int : 'S \rightarrow$ $int : int : 'S$; $int : int : 'S$ $\rightarrow int : 'S$
  - ... can be elided
  - DUP; ADD :: $int : 'S \rightarrow int : 'S$
  - DUP; ADD :: $a : S \mapsto (a+a) : S$

Michael Smart Contract Types

- contract :: (Parameter p, Storage s) $\rightarrow$ ([Operation], Storage s)
  - takes a pair of arguments: an input parameter p and a storage value s
  - returns a pair
    - a list of network operations and
    - a storage value (to replace the input storage)
  - the type of Parameter and Storage depends on the contract
- contract is implemented by a sequence $\mathcal{I}$ of Michelson instructions
- $\mathcal{I}$ transforms a stack with an input pair to a stack with an output pair
- $\mathcal{I}$ :: (pair 'Parameter 'Storage) : [] $\rightarrow$ (pair (list operation) 'Storage) : []

Contents

Michelson Smart Contracts: Hello world!

```
parameter unit;
storage string;
code { DROP;
       PUSH string "Hello World!";
       NIL operation; PAIR;
     };
```

type of input stack :: (pair unit string) : [ ]

Michelson Smart Contracts: Hello world!

Running with storage ”””’ and input Unit

initial stack value :: (Pair Unit ””) : [ ]

initial stack type :: (pair unit string) : [ ]

```
code { DROP;
       PUSH string ”Hello World!”;
       NIL operation; PAIR;
     };
```

Michelson Smart Contracts: Hello world!

DROP instruction

drops (i.e., removes) the top element of the stack

stack :: DROP / _ : S $\mapsto$ S

type :: 'a : 'S $\rightarrow$ 'S

Michelson Smart Contracts: Hello world!

DROP instruction

drops (i.e., removes) the top element of the stack

stack :: DROP / _ : S $\mapsto$ S

type :: 'a : 'S $\rightarrow$ 'S

after executing DROP:

stack value :: [ ]

stack type :: [ ]

```
code { PUSH string "Hello World!";
       NIL operation; PAIR;
     };
```

Michelson Smart Contracts: Hello world!

PUSH instruction

adds a value with a certain type onto the top of the stack

stack :: PUSH 'a x / S $\mapsto$ x : S

type :: 'S $\rightarrow$ 'a : 'S

Michelson Smart Contracts: Hello world!

PUSH instruction

adds a value with a certain type onto the top of the stack

stack :: PUSH 'a x / S $\mapsto$ x : S

type :: 'S $\rightarrow$ 'a : 'S

PUSH string "Hello World!"

stack :: PUSH string "Hello World!" / S $\mapsto$ "Hello World!" : S

type :: 'S $\rightarrow$ string : 'S

Michelson Smart Contracts: Hello world!

PUSH instruction

adds a value with a certain type onto the top of the stack

stack :: PUSH 'a x / S $\mapsto$ x : S

type :: 'S $\rightarrow$ 'a : 'S

PUSH string "Hello World!"

stack :: PUSH string "Hello World!" / S $\mapsto$ "Hello World!" : S

type :: 'S $\rightarrow$ string : 'S

after executing PUSH

stack value :: "Hello World!" : [ ]

stack type :: string : [ ]

```
code { NIL operation; PAIR;
       };
```

Michelson Smart Contracts: Hello world!

NIL instruction

adds an empty list of a certain type onto the top of the stack

stack :: NIL 'a / S $\Rightarrow$ [ ] : S

type :: 'S $\rightarrow$ list 'a : 'S

Michelson Smart Contracts: Hello world!

NIL instruction

adds an empty list of a certain type onto the top of the stack

stack :: NIL 'a / S ⇒ [ ] : S

type :: 'S → list 'a : 'S

NIL operation

stack :: NIL operation / S ⇒ [ ] : S

type :: 'S → list operation : 'S

Michelson Smart Contracts: Hello world!

NIL instruction

adds an empty list of a certain type onto the top of the stack

stack :: NIL 'a / S ⇒ [ ] : S

type :: 'S → list 'a : 'S

NIL operation

stack :: NIL operation / S ⇒ [ ] : S

type :: 'S → list operation : 'S

after executing NIL

stack value :: [ ] : "Hello World!" : [ ]

stack type :: list operation : string : [ ]

```
code { PAIR;
       }
```

Michelson Smart Contracts: Hello World!

PAIR instruction

removes the top two elements of the stack, makes a pair of them, and pushes the pair onto the stack

stack :: PAIR / a : b : S $\Rightarrow$ (Pair a b) : S

type :: 'a : 'b : 'S $\rightarrow$ pair 'a 'b : 'S

Michelson Smart Contracts: Hello World!

PAIR instruction

removes the top two elements of the stack, makes a pair of them, and pushes the pair onto the stack

stack :: PAIR / a : b : S $\Rightarrow$ (Pair a b) : S

type :: 'a : 'b : 'S $\rightarrow$ pair 'a 'b : 'S

after executing PAIR

stack value :: (Pair [ ] "Hello World!") : [ ]

stack type :: pair (list operation) string : [ ]

```
code {}
```

Michelson Smart Contracts: Hello world!

END

```
storage
  "Hello World!"
emitted operations [ ]
```

Type check

```
Well typed
{ parameter unit ;
  storage string ;
  code { /* [ pair (unit @parameter) (string @storage) ] */
         DROP
         /* [] */ ;
         PUSH string "Hello World!"
         /* [ string ] */ ;
         NIL operation
         /* [ list operation, string ] */ ;
         PAIR
         /* [ pair (list operation) string ] */ } }
```

Contents

Michelson Smart Contracts: Hello _!

Another example

```
parameter string;
storage string;
code { CAR;
       PUSH string "Hello ";
       CONCAT;
       NIL operation; PAIR;
     };
```

Michelson Smart Contracts: Hello _!

Another example

```
parameter string;
storage string;
code { CAR;
       PUSH string "Hello ";
       CONCAT;
       NIL operation; PAIR;
     };
```

CAR instruction

select left component of pair

stack value :: CAR / (Pair a _) : S $\mapsto$ a : S

stack type :: pair 'a 'b : 'S $\rightarrow$ 'a : 'S

Michelson Smart Contracts: Hello _!

Another example

```
parameter string;
storage string;
code { CAR;
       PUSH string "Hello ";
       CONCAT;
       NIL operation; PAIR;
     };
```

CAR instruction

select left component of pair

stack value :: CAR / (Pair a _) : S $\mapsto$ a : S
stack type :: pair 'a 'b : 'S $\rightarrow$ 'a : 'S

CONCAT

string concatenation

stack :: CONCAT / a : b : S $\mapsto$ $a \uparrow b$: S
type :: string : string : 'S $\rightarrow$ string: 'S

where $a \uparrow b$ concatenates strings a and b (alternative writing a^b)

Michelson Smart Contracts: Hello _!

Type checking

```
{ parameter string ;
  storage string ;
  code { /* [ pair (string @parameter) (string @storage) ] */
         CAR
         /* [ @parameter string ] */ ;
         PUSH string "Hello "
         /* [ string, @parameter string ] */ ;
         CONCAT
         /* [ string ] */ ;
         NIL operation
         /* [ list operation, string ] */ ;
         PAIR
         /* [ pair (list operation) string ] */ } }
```

Michelson Smart Contracts: Hello _!

Running on storage ”””’ and input ”’Tezos”’

```
storage
  ”Hello Tezos”
emitted operations
```

Contents

Michelson Smart Contracts: A voting contract

Voting for your favourite supercomputer

- an open vote with a fee
- a fixed list of choices
- voter's identity is not stored

Michelson Smart Contracts: A voting contract

Types

- the storage: a map from string to int
  domain: names of the candidates for voting
  range: number of votes cast for candidate
- the parameter: string
  should be one of the candidates to be counted

```
parameter string;
storage (map string int);
```

Michelson Smart Contracts: A voting contract

Verify that the caller sent enough tokens to be able to vote. If not, make the call fail

```
code{
#(name, storage)
AMOUNT;
# amount : (name, storage)
PUSH mutez 5000;
# 5000 : amount : (name, storage)
IFCMPGT{PUSH string "stingy!"; FAILWITH}
{};
# (name,storage)
```

- AMOUNT pushes the number of tokens received from the contract caller
- IFCMPGT is a macro: compares the two numbers on top (removing them in the process)
  - if the top number was greater, executes its first branch (code in braces)
    $\Rightarrow$ fail with an error message
  - otherwise the second

Michelson Smart Contracts: A voting contract

```
DUP;
# (name, storage) : (name, storage)
UNPAIR;
# name : storage : (name, storage)
GET;
# (Some current | None) : (name, storage)
```

- UNPAIR destructs the pair to get a stack with the key on top and the map beneath
- GET consumes the name and storage map on top to lookup the name in the map; returns
  - Some current, where current is the count for the voted name or
  - None if the key was not in the map

  a value of type option **int**

Michelson Smart Contracts: A voting contract

- if the count is some integer value, add 1 to this value,
- if not, fail because of an unknown name

```
IF_SOME
    {
      # current : (name, storage)
      PUSH int 1; ADD;
      # current+1 : (name, storage)
      SOME}
    {PUSH string "Unknown super computer";
     FAILWITH};
    # (Some (current + 1)) : (name,storage)
```

Michelson Smart Contracts: A voting contract

Reorder the elements and then update the map

```
# (Some (current + 1)) : (name, storage)
DIP{UNPAIR};
# (Some (current + 1)) : name : storage
SWAP;
# name:(Some (current+1)) : storage
UPDATE;
# updated storage
```

- DIP applies the code in braces one element below the stack top
- SWAP exchanges the two top elements of the stack
- UPDATE updates the map with the new count

Michelson Smart Contracts: A voting contract

Return updated storage

```
NIL operation;
PAIR;
# (nil, updated storage)
```

Michelson Smart Contracts: A voting contract

Type check

```
Well typed
{ parameter string ;
  storage (map string int) ;
  code { /* [ pair (string @parameter) (map @storage string int) ] */
         AMOUNT
          /* [ @amount mutez : pair (string @parameter) (map @storage string int) ] */ ;
          PUSH mutez 5000
          /* [ mutez : @amount mutez : pair (string @parameter) (map @storage string int) ] */ ;
         IFCMPGT
            { PUSH string "stingy!"
              /* [ string : pair (string @parameter) (map @storage string int) ] */ ;
             FAILWITH
              /* [] */ }
            { /* [ pair (string @parameter) (map @storage string int) ] */ }
          /* [ pair (string @parameter) (map @storage string int) ] */ ;
```

Michelson Smart Contracts: A voting contract

Type check

```
DUP
/* [ pair (string @parameter) (map @storage string int)
    : pair (string @parameter) (map @storage string int) ] */ ;
UNPAIR
/* [ @parameter string : @storage map string int
    : pair (string @parameter) (map @storage string int) ] */ ;
GET
/* [ option int : pair (string @parameter) (map @storage string int) ] */ ;
IF_SOME
  { /* [ @some int : pair (string @parameter) (map @storage string int) ] */
    PUSH int 1
    /* [ int : @some int : pair (string @parameter) (map @storage string int) ] */ ;
    ADD
    /* [ int : pair (string @parameter) (map @storage string int) ] */ ;
    SOME
    /* [ option int : pair (string @parameter) (map @storage string int) ] */ }
  { PUSH string "Unknown super computer"
    /* [ string : pair (string @parameter) (map @storage string int) ] */ ;
```

Michelson Smart Contracts: A voting contract

Type check

```
        FAILWITH
              /* [] */ }
         /* [ option int : pair (string @parameter) (map @storage string int) ] */ ;
         DIP { /* [ pair (string @parameter) (map @storage string int) ] */
               UNPAIR
               /* [ @parameter string : @storage map string int ] */ }
         /* [ option int : @parameter string : @storage map string int ] */ ;
         SWAP
         /* [ @parameter string : option int : @storage map string int ] */ ;
         UPDATE
         /* [ @storage map string int ] */ ;
         NIL operation
         /* [ list operation : @storage map string int ] */ ;
         PAIR
         /* [ pair (list operation) (map @storage string int) ] */ } }
```