Liquidity : OCaml for the Blockchain

Çagdas Bozman, Mohamed Iguernlala, Michael Laporte,
Fabrice Le Fessant and **Alain Mebsout**

January 25th, 2018

OCaml PRO

Blockchain

- Distributed and persistant data structure
- Concurrently adding new blocks
- Criterion (consensus) for choosing the right chain

Blockchains and *smart contracts*

OCaml PRO

Blockchain

- Distributed and persistant data structure
- Concurrently adding new blocks
- Criterion (consensus) for choosing the right chain

Bitcoin

2008

- Crypto-currency
- Transfers between addresses

Blockchains and *smart contracts*

OCaml PRO

Blockchain

- Distributed and persistant data structure
- Concurrently adding new blocks
- Criterion (consensus) for choosing the right chain

Bitcoin 2008

- Crypto-currency
- Transfers between addresses

Ethereum 2014

- + Smart contracts

Blockchains and *smart contracts*

OCaml PRO

Blockchain

- Distributed and persistant data structure
- Concurrently adding new blocks
- Criterion (consensus) for choosing the right chain

Bitcoin 2008

- Crypto-currency
- Transfers between addresses

Ethereum 2014

- + Smart ~~contracts~~

Blockchains and *smart contracts*

OCaml PRO

Blockchain

- Distributed and persistant data structure
- Concurrently adding new blocks
- Criterion (consensus) for choosing the right chain

Bitcoin 2008

- Crypto-currency
- Transfers between addresses

Ethereum 2014

- + ~~Smart contracts~~

Blockchains and *smart contracts*

OCaml PRO

Blockchain

- Distributed and persistant data structure
- Concurrently adding new blocks
- Criterion (consensus) for choosing the right chain

Bitcoin 2008

- Crypto-currency
- Transfers between addresses

Ethereum 2014

- + ~~Smart contracts~~ Programs

Blockchains and *smart contracts*

OCaml PRO

Blockchain

- Distributed and persistant data structure
- Concurrently adding new blocks
- Criterion (consensus) for choosing the right chain

Bitcoin 2008

- Crypto-currency
- Transfers between addresses

Ethereum 2014

- + ~~Smart contracts~~ Programs
- Unalterable "*code is law*"
- Executed by nodes participating in the blockchain

Tezos and Michelson

OCaml PRO

Tezos

2014

- Proof of stake
- Auto-amending by referendum
- Stress on formal methods
  - Developed in OCaml

Michelson

- *Smart contracts* language
  - Stack based
  - Functional
  - Strongly typed

Michelson program example

OCaml PRO

```
parameter string;
storage (map string int);
return unit;
code
  { # Pile = [ Pair parameter storage ]
    PUSH tez "5.00"; AMOUNT; COMPARE; LT;
    IF # Is AMOUNT < 5 tz ?
      { FAIL }
      {
        DUP; DUP; CAR; DIP { CDR }; GET; # GET parameter storage
        IF_NONE # Is it a valid vote ?
          { FAIL }
          { # Some x, x now in the stack
            PUSH int 1; ADD; SOME; # Some (x + 1)
            DIP { DUP; CAR; DIP { CDR } }; SWAP; UPDATE;
            # UPDATE parameter (Some (x + 1)) storage
            PUSH unit Unit; PAIR; # Pair Unit new_storage
          }
      };
  }
```

Liquidity

- Syntax is a subset of OCaml[1]
- Close to Michelson in spirit
- Functional and strongly typed (without polymorphism)
- Compiled to Michelson

[1] Uses a slightly modified version of the OCaml parser

Base Types of the Language

OCaml PRO

Base types

- `unit`: with unique constructor `()`
- `bool`: Booleans
- `int`: unbounded integers
- `nat`: unbounded naturals
- `tez`: type of amounts
- `string`: text strings
- `timestamp`: dates et durations
- `key`: cryptographic keys
- `key_hash`: hashes of cryptographic keys
- `signature`: cryptographic signatures

Composed types

- `'a option = None | Some of 'a`
- `('a, 'b) variant =`
  `Left of 'a | Right of 'b`
- `(t1 * t2 * t3)` : tuples
- `'a list` : lists
- `'a set` : sets
- `('a, 'b) map` : maps
- `'a -> 'b` : functions
- `('a, 'b) contract` : for contracts whose parameter is of type `'a` and return value is of type `'b`

Definitions (non-recursive)

```
type point = { x : int; y : int }
type result = Ok of point | Error of string
```

Languages constructs

OCaml PRO

- let
- Sequence
- Function applications
- If-then-else
- Pattern matching (on contributors and lists, non-deep)
- Lambda-abstractions
- Closures

Languages constructs

- let
- Sequence
- Function applications
- If-then-else
- Pattern matching (on contributors and lists, non-deep)
- Lambda-abstractions
- Closures

```
match%nat x with (* x : int *)
| Pos p -> (* p : nat = x *) ...
| Neg n -> (* n : nat = -x *) ...
```

Languages constructs

- let
- Sequence
- Function applications
- If-then-else
- Pattern matching (on contributors and lists, non-deep)
- Lambda-abstractions
- Closures

```
match%nat x with (* x : int *)
| Pos p -> (* p : nat = x *) ...
| Neg n -> (* n : nat = -x *) ...
```

Contract calls

```
let result, storage = Contract.call c amount storage param in ...
```

Example

```
[%%version 0.14]

type votes = (string, int) map

let%init storage (myname : string) =
  Map.add myname 0 (Map ["ocaml", 0; "pro", 0])

let%entry main (parameter : string) (storage : votes)
               : unit * votes =
  let amount = Current.amount() in
  if amount < 5.00tz then
    Current.fail ()
  else
    match Map.find parameter storage with
    | None -> Current.fail ()
    | Some x ->
       let storage = Map.add parameter (x+1) storage in
       ( (), storage )
```

Simplifications et encodings

```
type t1 = { a: int; b: string; c: bool}
                ⇓
type t1 = (int * (string * bool))
```

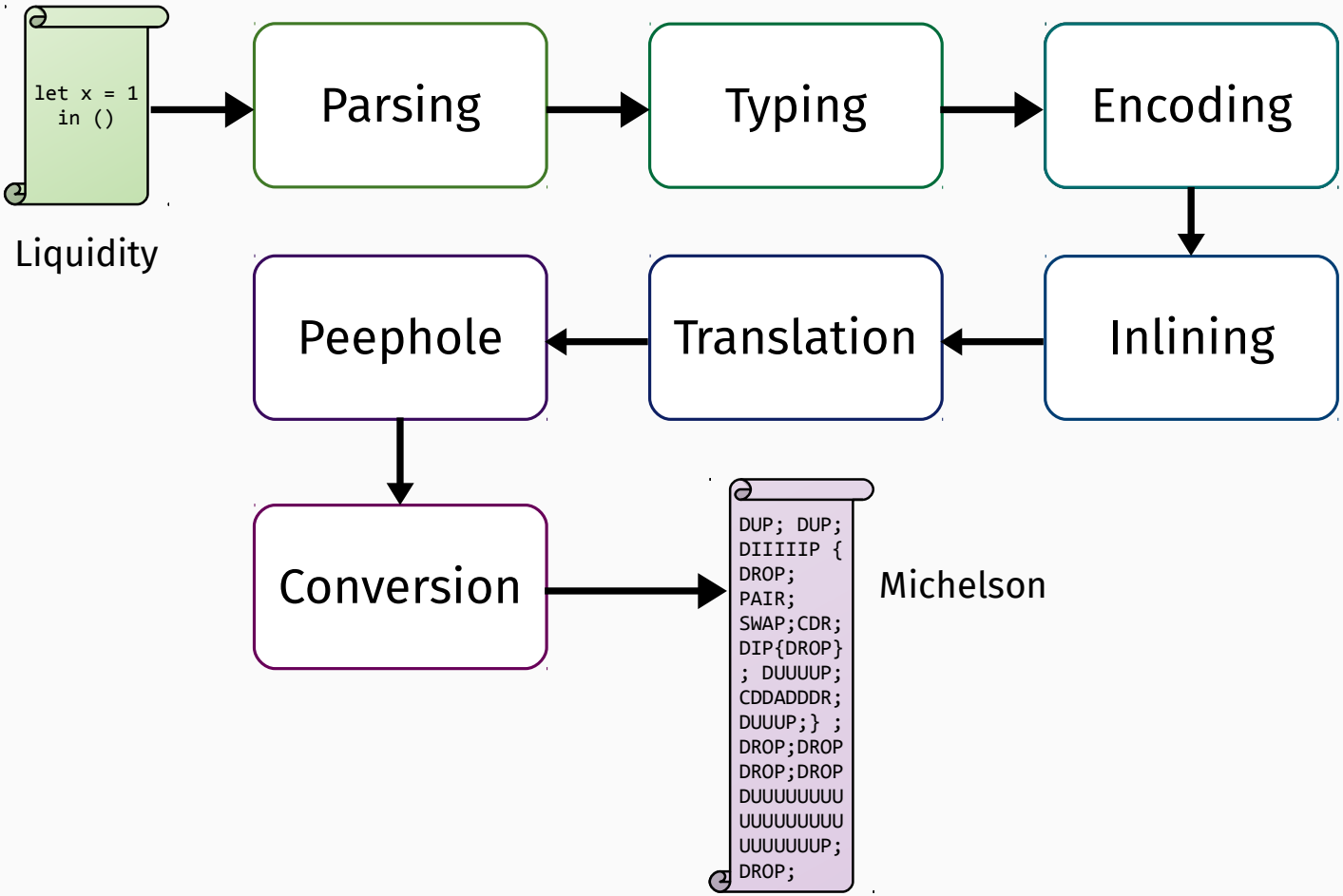
```
type t2 = A of int | B of string | C
                ⇓
type t2 = (int, (string, unit) variant) variant
```

```
match x with
| A i -> qqch1(i)
| B s -> qqch2(s)
| C -> qqch3
```

⇒

```
match x with
| Left i -> qqch1(i)
| Right r -> match r  with
             | Left s -> qqch2(s)
             | Right _ -> qqch3
```

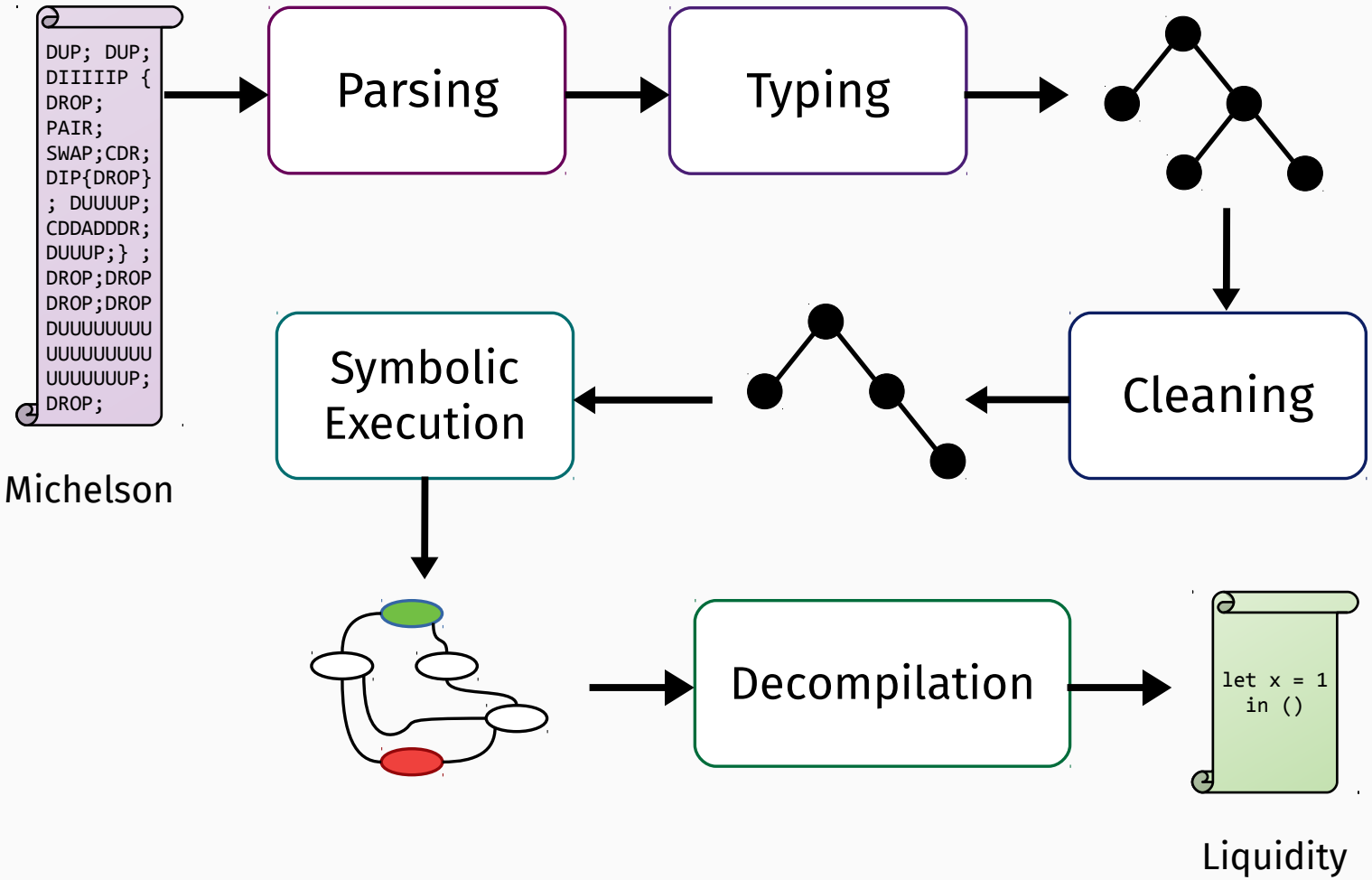
Compilation schema



Compilation: example

```
parameter string;
storage (map string int);
return unit;
code
  {
    DUP ;
    DIP {CDR ;} ;
    CAR ;
    PUSH
      tez
      "5.00" ;
    AMOUNT ; { @amount }
    COMPARE ;
    LT ;
    IF
      {
        FAIL ;
      }
      {
        DUUP ; { @storage }
        DUUP ; { @parameter }
        GET ;
        IF_NONE
          {
            FAIL ;
          }
          {
            DUUP ; { @storage }
            PUSH
              int
              1 ;
            DUUUP ; { @x }
            ADD ;
            DUUUUP ; { @parameter }
            DIP {SOME } ;
            UPDATE ; { @storage }
            DIP {DROP } ;
            PUSH
              unit
              Unit ;
            PAIR ;
          } ;
      } ;
    DIP
      {
        DROP ;
        DROP ;
      } ;
  }
```

Decompilation

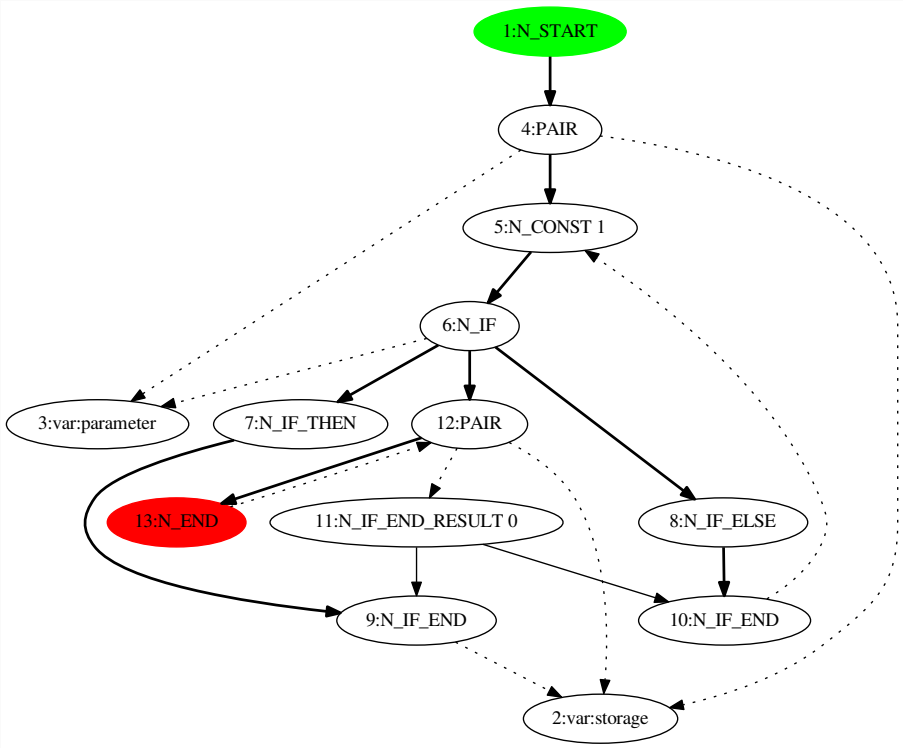


Decompilation: example

```
parameter bool;
return int;
storage int;
code {DUP; CAR;
      DIP { CDR; PUSH int 1 };
      IF
         { DROP; DUP; }
         { }
         ;
      PAIR;
     }
```

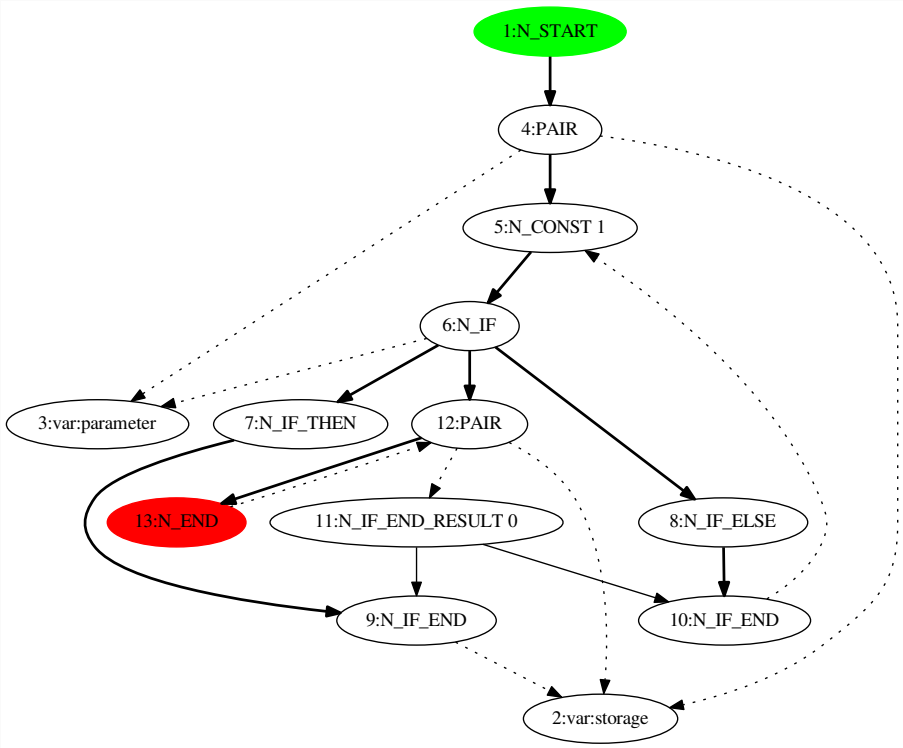
Decompilation: example

```
parameter bool;
return int;
storage int;
code {DUP; CAR;
      DIP { CDR; PUSH int 1 };
      IF
         { DROP; DUP; }
         { }
         ;
      PAIR;
     }
```



Decompilation: example

```
parameter bool;
return int;
storage int;
code {DUP; CAR;
      DIP { CDR; PUSH int 1 };
      IF
         { DROP; DUP; }
         { }
         ;
      PAIR;
     }
```




```
[%%version 0.14]
[%%entry
  let main (parameter : bool) (storage : int) : (int * int) =
    ((if parameter then storage else 1), storage) ]
```

Try-liquidity

Demo

Perspectives

OCaml PRO

- Multiple entry points
- Debugger (step-by-step execution)
- Verification
- Targer EVM (Ethereum Virtual Machine)
- Framework for deploying *dapps*

Thanks

http://liquidity-lang.org