Liquidity : OCaml pour la blockchain

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

25 janvier 2018

OCaml PRO

Blockchain

- Structure de données distribuée et persistante
- Ajout de nouveaux blocs en concurrence
- Critère (consensus) pour choisir la bonne chaîne

Blockchain et *smart contracts*

OCaml PRO

Blockchain

- Structure de données distribuée et persistante
- Ajout de nouveaux blocs en concurrence
- Critère (consensus) pour choisir la bonne chaîne

Bitcoin

2008

- Crypto-monnaie
- Transferts entre adresses

Blockchain et *smart contracts*

OCaml PRO

Blockchain

- Structure de données distribuée et persistante
- Ajout de nouveaux blocs en concurrence
- Critère (consensus) pour choisir la bonne chaîne

Bitcoin 2008

- Crypto-monnaie
- Transferts entre adresses

Ethereum 2014

- + Smart contracts

Blockchain et *smart contracts*

OCaml PRO

Blockchain

- Structure de données distribuée et persistante
- Ajout de nouveaux blocs en concurrence
- Critère (consensus) pour choisir la bonne chaîne

Bitcoin 2008

- Crypto-monnaie
- Transferts entre adresses

Ethereum 2014

- + Smart ~~contracts~~

Blockchain et *smart contracts*

OCaml PRO

Blockchain

- Structure de données distribuée et persistante
- Ajout de nouveaux blocs en concurrence
- Critère (consensus) pour choisir la bonne chaîne

Bitcoin 2008

- Crypto-monnaie
- Transferts entre adresses

Ethereum 2014

- + ~~Smart contracts~~

Blockchain et *smart contracts*

OCaml PRO

Blockchain

- Structure de données distribuée et persistante
- Ajout de nouveaux blocs en concurrence
- Critère (consensus) pour choisir la bonne chaîne

Bitcoin 2008

- Crypto-monnaie
- Transferts entre adresses

Ethereum 2014

- + ~~Smart contracts~~ Programmes

Blockchain et *smart contracts*

OCaml PRO

Blockchain

- Structure de données distribuée et persistante
- Ajout de nouveaux blocs en concurrence
- Critère (consensus) pour choisir la bonne chaîne

Bitcoin 2008

- Crypto-monnaie
- Transferts entre adresses

Ethereum 2014

- + ~~Smart contracts~~ Programmes
- Immuables «*code is law*»
- Exécutés par les nœuds participant à la blockchain

Tezos et Michelson

OCaml PRO

Tezos

2014

- Proof of stake
- Auto-modification par référendum
- Accent sur les méthodes formelles
  - Développé en OCaml

Michelson

- Langage de *smart contracts*
  - À pile
  - Fonctionnel
  - Fortement typé

Exemple de programe Michelson

OCaml PRO

```
parameter string;
storage (map string int);
return unit;
code
  { # Pile = [ Pair parameter storage ]
    PUSH tez "5.00"; AMOUNT; COMPARE; LT;
    IF # Est-ce que AMOUNT < 5 tz ?
      { FAIL }
      {
        DUP; DUP; CAR; DIP { CDR }; GET; # GET parameter storage
        IF_NONE # Est-ce un vote invalide ?
          { FAIL }
          { # Some x, x dans la pile
            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 nouveau_stockage
          }
      };
  }
```

Liquidity

- Syntaxe est un sous ensemble de OCaml [1]
- Proche de Michelson dans l'esprit
- Fonctionnel et fortement typé (sans polymorphisme)
- Compilé vers Michelson

1. Utilise une version légèrement modifiée du parseur OCaml

Types de base du langage

OCaml PRO

Types de base

- `unit` : avec unique constructeur `()`
- `bool` : Booléens
- `int` : entiers non bornés
- `nat` : naturels non bornés
- `tez` : type des montants
- `string` : chaînes de caractères
- `timestamp` : dates et durées
- `key` : clefs cryptographiques
- `key_hash` : hashs de clefs cryptographiques
- `signature` : signatures cryptographiques

Types composés

- `'a option = None | Some of 'a`
- `('a, 'b) variant = Left of 'a | Right of 'b`
- `(t1 * t2 * t3)` : tuples
- `'a list` : listes
- `'a set` : ensembles
- `('a, 'b) map` : maps
- `'a -> 'b` : fonctions
- `('a, 'b) contract` : pour les contrats dont le paramètre est de type `'a` et la valeur de retour est de type `'b`

Définitions (non récursives)

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

Constructions du langage

- let
- Séquence
- Application de fonctions
- If-then-else
- Pattern matching (sur constructeurs et listes, non-profond)
- Lambda-abstractions
- Fermetures

Constructions du langage

- let
- Séquence
- Application de fonctions
- If-then-else
- Pattern matching (sur constructeurs et listes, non-profond)
- Lambda-abstractions
- Fermetures

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

Constructions du langage

- let
- Séquence
- Application de fonctions
- If-then-else
- Pattern matching (sur constructeurs et listes, non-profond)
- Lambda-abstractions
- Fermetures

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

Appel de contract

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

Exemple

```
[%%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 encodages

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

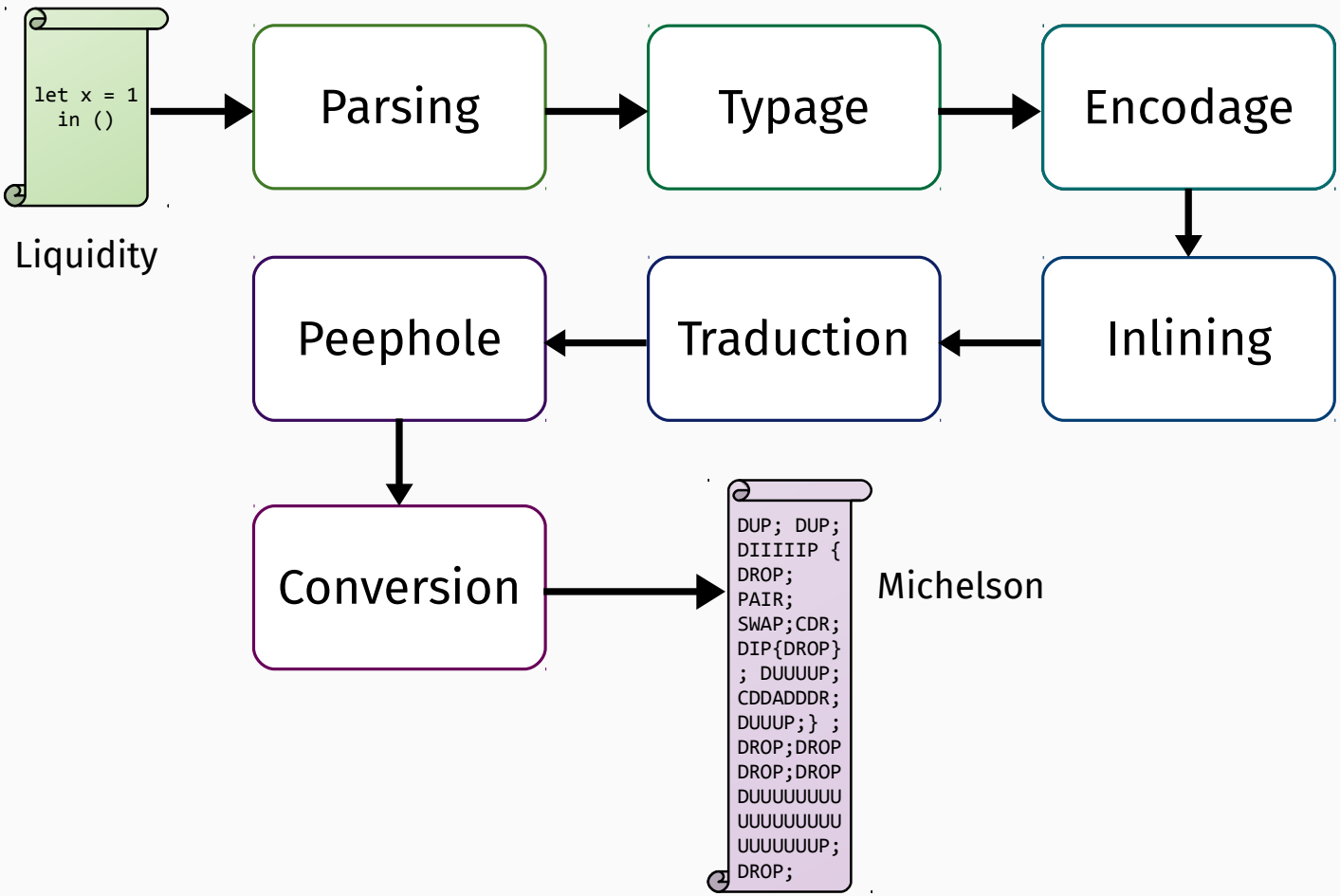
```
type t2 = A of int | B of string | C
                ⇓
type t2 = (int, (string, unit) variant) variant
```

```
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
```

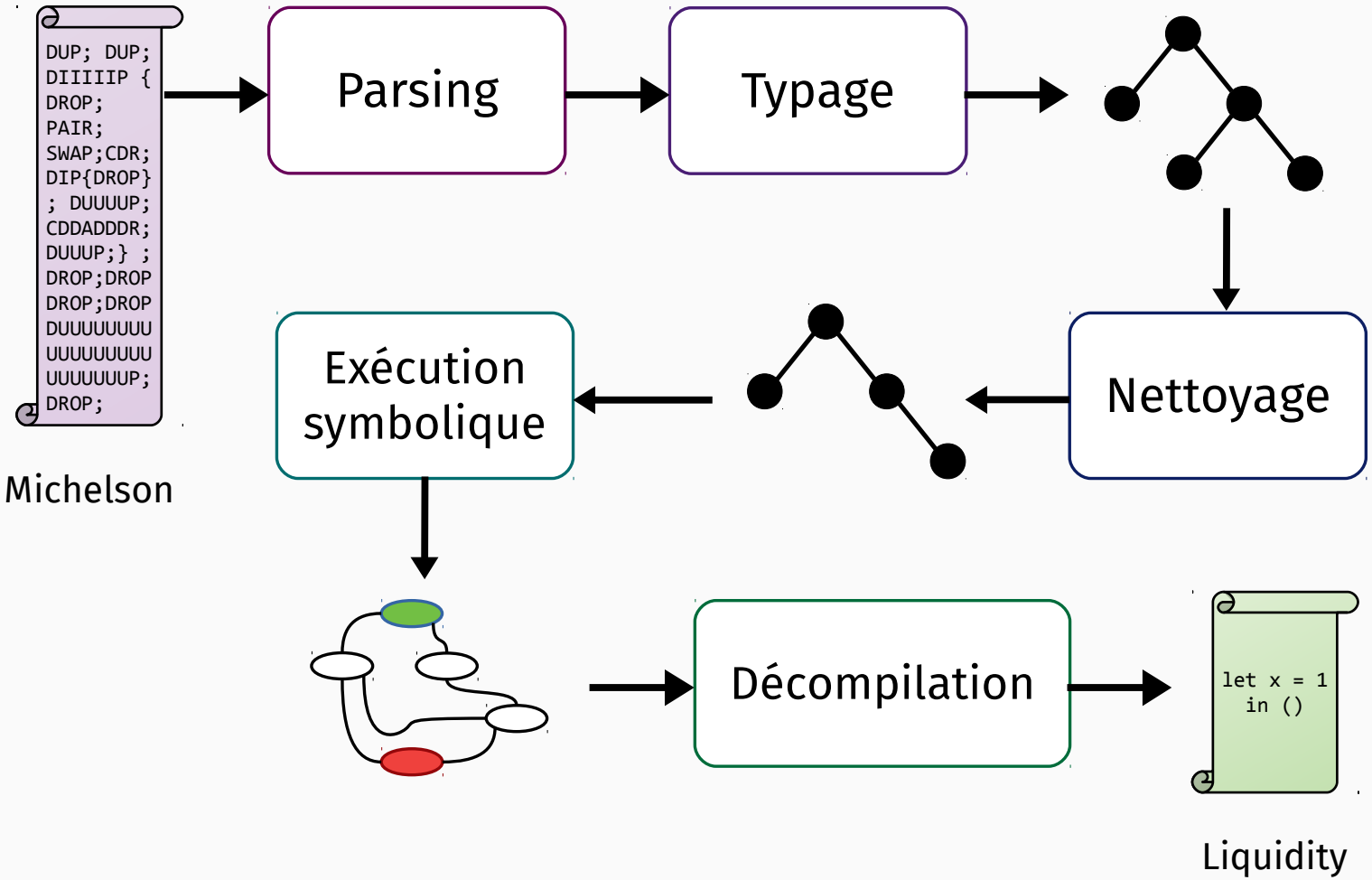
Schéma de compilation



Exemple : compilation

```
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 ;
          }
          {
            DUUUP ; { @storage }
            PUSH
              int
              1 ;
            DUUUP ; { @x }
            ADD ;
            DUUUUP ; { @parameter }
            DIP {SOME } ;
            UPDATE ; { @storage }
            DIP {DROP } ;
            PUSH
              unit
              Unit ;
            PAIR ;
          } ;
      } ;
    DIP
      {
        DROP ;
        DROP ;
      } ;
  }
```

Décompilation

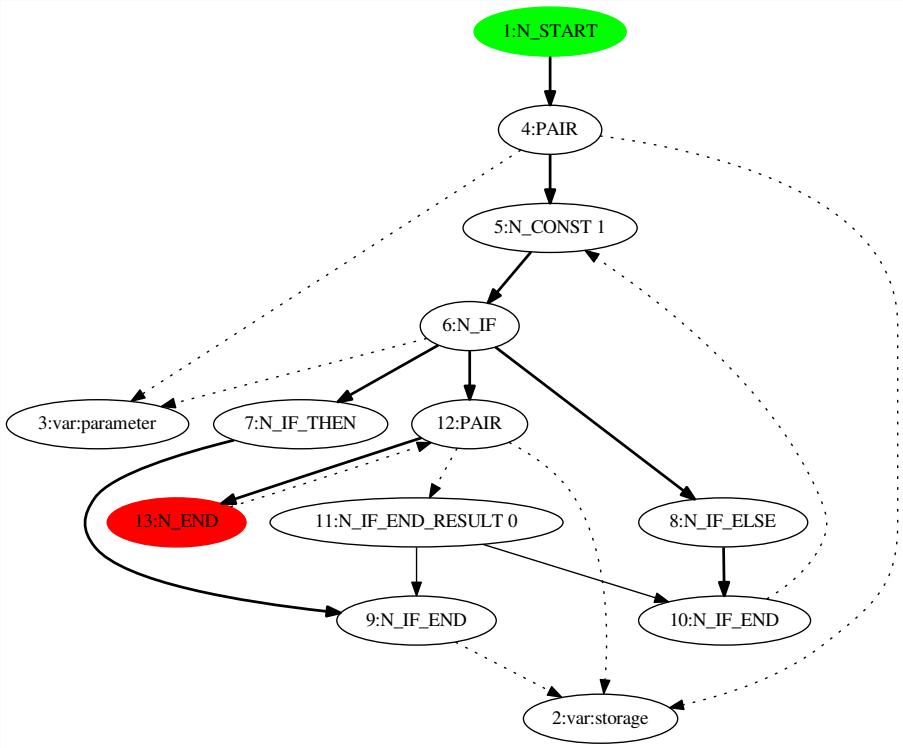


Décompilation : exemple

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

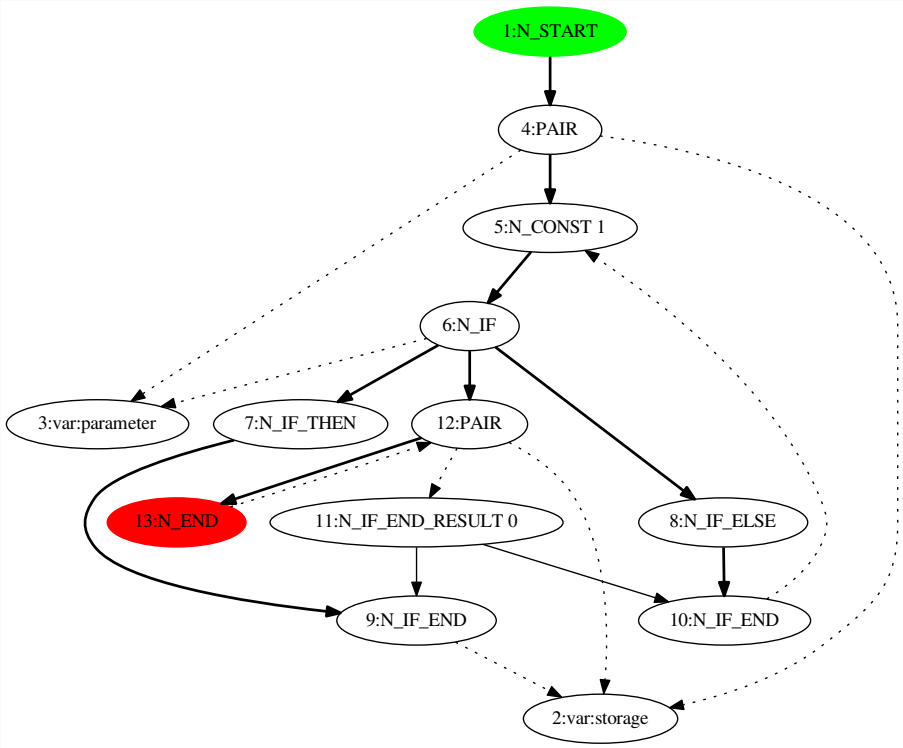
Décompilation : exemple

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



Décompilation : exemple

```
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

Démo

Perspectives

- Plusieurs points d'entrée
- Debugger (exécution pas à pas)
- Vérification
- Cibler EVM (Ethereum Virtual Machine)
- Framework pour déploiement d'applications

Merci

http://liquidity-lang.org