Compiling a low-level language to Michelson

Basile Pesin, under the supervision of Bruno Bernardo, Raphaël Cauderlier, and Julien Tesson

September 19, 2019

Nomadic Labs

Context, and the Michelson programming language

The Tezos Blockchain

A **blockchain** (distributed transaction ledger) with...

Self-amending protocol : Token holders vote for updates

Focus on safety :

- Strongly typed OCaml implementation
- Formally verifiable smart-contract language

The Michelson language

- Stack manipulations : PUSH `ty lit`, DROP, SWAP, DUP, DIP `{ ins }`
- Numeric types: `int`, `nat`, `mutez`, `timestamp` and ops: ADD, COMPARE ...
- Booleans, control structures: IF `{ then } { else }`, LOOP `{ body }`
- Product types (PAIR, CAR, ...) and sum types (LEFT, IF_LEFT {}, ...)
- List, Set: NIL `ty`, CONS, IF_CONS `{ then } { else }`, ITER `{ body }` ...
- Map: EMPTY_MAP `tyk tyv`, UPDATE, MEM, GET ...
- Domain specific operations

A short Michelson program

The **voting** smart contract, a classic

```
storage (map string int %candidates);
parameter string %chosen;
code { AMOUNT; PUSH mutez 5000000; COMPARE; GT;
       IF { FAIL } {};
       DUP; DIP { CDR; DUP }; CAR; DUP;
       DIP {
            GET; ASSERT_SOME;
            PUSH int 1; ADD; SOME
           };
       UPDATE; NIL operation; PAIR
     }
```

Representations of Michelson

- In-chain interpreter (in the protocol)
- Documentation (`http://tezos.gitlab.io/`)
- MiChoCoq typed syntax

  ⇅

  MiChoCoq untyped syntax

  ⇅

  Micheline (S-expressions)
- MiChoCott

The Albert Programming language

Albert general ideas

- Imperative-looking language
- Abstract the stack with variables in a global record
- Generalize pairs by n-ary records
- Generalize Or, Options, Booleans by n-ary variants
- Global, non-recursive functions

Syntax

```
instr :=
  | lhs = rhs
  | instr ; instr
  | drop x
  | loop x do instr done
  | ...
lhs :=
  | x
  | { l_1 = x_1 ; ... ; l_n = x_n }
  | (x_1, x_2)
```

```
rhs :=
  | arg
  | x_1 + x_2
  | x.l
  | { l_1 = x_1 ; ... ; l_n = x_n }
  | ...

arg :=
  | x
  | 42 | "Hello" | 500utz
  | ...
```

The voting contract, in Albert

```
type storage_ty = { threshold : mutez; votes: map string nat }
def vote :
  { param : string ; store : storage_ty } →
  { operations : list operation ; out_storage : storage_ty } =
    (store0, store1) = dup store; threshold = store1.threshold; (threshold, threshold_copy) = dup threshold;
    am = amount; ok = am >= threshold;
    match ok with
      False f →failwith "you_are_so_cheap!"
      | True t →drop t; state = store0.votes ;
              (state0, state1) = dup state; (param0, param1) = dup param;
              prevote_option = state1[param1];
              { res = prevote } = assert_some { opt = prevote_option };
              one = 1; postvote = prevote + one; postvote = Some postvote;
              final_state = {| state0 with param0 |→ postvote |};
              out_storage = {threshold = threshold_copy; votes = final_state};
              operations = ([] : list operation)
    end
```

Linear typing

Values are consumed: $G \vdash y = x : \{x : t\} \Rightarrow \{y : t\}$

Explicit duplication: $G \vdash y = dup\, x : \{x : t\} \Rightarrow \{y : (t \times t)\}$
with left-hand side destructuring: $G \vdash (y, z) = dup\, x : \{x : t\} \Rightarrow \{y : t; z : t\}$
Explicit destruction: $G \vdash drop\, x : \{x : t\} \Rightarrow \{\}$

Other examples:
$G \vdash z = x + y : \{x : int; y : int\} \Rightarrow \{z : int\}$
$G \vdash n = size\, s : \{s : string\} \Rightarrow \{n : nat\}$

Albert “source”

Using Ott to define the rules and generate AST in Coq.

```
instruction, I :: 'instr_' ::= {{com Instruction}}
  | drop var :: :: drop {{com Resource dropping}}

  defn
    g |- instruction : ty => ty' :: :: instr :: T_ by

    ---------------------------------- :: drop
    g |- drop var : {var : ty} => unit

  defn
    E |- instruction / val -> val' :: :: instr :: instr_ by

    ------------------------------------------------ :: drop
    E |- drop var / { var = val } -> {}
```

Difficulties

Ott representation of n-ary rules

```
g |- val_1 : ty_1 .. g |- val_n : ty_n
------------------------------------------------------------------------------ :: record
g |- { l_1 = val_1 ; .. ; l_n = val_n } : { l_1 : ty_1 ; .. ; l_n : ty_n }
```

Gives something like:

```
Inductive typing_arg :=
[...]
| typing_val_record : ∀ (l : list (label*val*ty))
  [...]
  (map (fun pat_ : label * value * ty ⇒
          let (p, _) := pat_ in let (l__, val__) := p in (l__, val__))
       l)))
  [...]
```

Nice properties of a language

Properties of well formed types

Notations : $G \vdash ty$, $G \vdash$

- $G \vdash ty \Rightarrow G \vdash$
- $ty \equiv ty' \Rightarrow (G \vdash ty \Leftrightarrow G \vdash ty')$
- $G \vdash instr : ty \rightarrow ty' \Rightarrow G \vdash ty \Rightarrow G \vdash ty'$

Subject reduction and progress

- $(G \vdash instr : ty \rightarrow ty') \Rightarrow (E \models v : ty) \Rightarrow (E \models instr/v \Rightarrow v') \Rightarrow (G \vdash v : ty')$
- $(G \vdash instr : ty \rightarrow ty') \Rightarrow (E \models v : ty) \Rightarrow (\exists v', E \models instr/v \Rightarrow v')$

Albert's compiler to Michelson

Compiler target

MiChoCoq typed syntax $\rightarrow$ Verify equivalence of typing and semantics

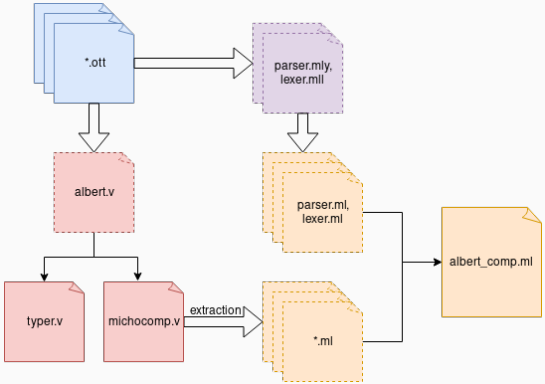
$\uparrow$

MiChoCoq untyped syntax

$\downarrow$

Micheline (S-expressions) $\rightarrow$ Pretty print to a *.tz* file

Compiler architecture



Almost all the code is extracted from Coq or generated from Ott.

Compiler pipeline

1. Inlining of type definitions
2. Sorting of record and variant fields
3. Type-checking (adds some type annotations)
4. Compilation to MiChoCoq + function inlining

Every step returns in an error monad (although compilation should not fail)

Stack manipulations

Carry a representation of the stack (list of the variable names) during the compilation.

Use DIG and DUG

Examples:

$[\![\texttt{y = x}]\!] =$ DIG `x` ; DUG `y`

$[\![\texttt{x + y}]\!] =$ DIG `y` ; DIG `x` ; ADD

Why DUG

```
match b with
True u →x = 14; y = ‘‘Hello’’
| False u →y = ‘‘World’’; x = 42
end
```

is of type `{ b : bool } →{ u : unit ; x : nat ; y : string }`

and compiles to `DIG b; IF { ... } { ... }`

Without DUGing, output stacks would be
`nat::string::unit::S` and `string::nat::unit::S`

Other solution: reorganize Albert's instructions (probably more efficient).

Conclusion

Accomplished work

- Additions to MiChoCoq (DIG and DUG instructions, Micheline pretty-printer)
- Some improvements on Ott (in particular on the lexer and parser generators)
- Formal semantics of Albert in Ott
- Type-checker + compiler for almost all Albert (missing sets and some domain specific operations)

Future works

- Proving type-checker and compiler properties
- Better error messages (locations)
- Optimization of stack manipulations
- And of course, writing compilers to Albert