SCaml compiler

DaiLambda, Inc.
Jun FURUSE / 古瀬 淳

ReFX, Kyoto, 2019-09-10

Compilers for Tezos smart contracts

- Liquidity
- LIGO
- Fi
- SmartPy
- ..

Risky, premature, lots of bugs, or no documentation.

Bottleneck of adaptation

None uses Blockchain without a proper language.

We need a simple compiler for Tezos which **works**.

Small Compiler to Michelson.

Small implementation

For shorter development time.

Strict subset of an existing language

Users can learn it from the existing materials.

Serious tool for industry

All the features of Michelson should be covered.

Simple

No fancy features.

SCaml

Small implementation

Use of OCaml compiler library `compiler-libs`.

No need to write parser and type-checker.

Just around 1000 lines written in 1 week. Easy to extend.

Strict subset of OCaml

Valid SCaml programs are also valid OCaml programs.

OCaml tools are available for free: Tuareg, Merlin, PPX, etc.

Simulation in OCaml.

Serious tool for industry

All the technical challenges have been resolved.

Simple

No user defined data types. No polymorphism.

No pattern matching. No modules.

SCaml’s motto

Scam never call it a scam.

SCaml Features

Numeric types: `int`, `nat`, `tz`

`Int (-12)`, `Nat 42`, `Tz 1.23`

Monomorphic arithmetic operators: `(+)`, `(+^)`,`(+$)`

Options, Lists

`Some 1`, `[ 1; 2; 3 ]`

Sets, Maps

`Set [ 1; 2; 3 ]`, `Map [ (1, "one"), (2, "two") ]`

Pairs, Sums:

`int * int`, `(int, int) sum`

No user defined data types

But aliases are defineable:

ex. `type 'a t = (int * ('a, nat) sum)`

SCaml Features

Functions

```
fun x -> x + 1
```

Local lets, but no polymorphism

```
let x = e in e'
```

Switches (Not pattern matches)

```
match opt with
| None -> ...
| Some x -> ...
```

- No nested patterns
- No constants in patterns

Compilation Phases

Parsing

- OCaml parser

Typing

- OCaml type-checker
- SCaml specific typing

Compilation

- Shrink down to an intermidiate language, IML
- Closure conversion analysis in IML
- Compilation from IML to Michelson

SCaml typing

By unifying types available in OCaml’s typed AST:

Entry point

Force the type of the entry point to `'param -> 'storage -> operation list * 'storage`

`SELF`

Force the type of `SELF` opcode to `'param contract`

IML

Very small purely functional language.

- No polymorhpism
- No pattern match,
  but simple switches over Eithers/Lists/Options.

IML to Michelson

Quite normal compilation algorithm C to stack VM:

- Expression e is compiled to opcodes $C(e)$
- Environment E is compiled to stack $C(E)$

Property:

$$\begin{array}{l} E \models e \rightarrow v \\ eval(C(E), C(e)) = C(v) :: C(E) \end{array}$$

Adding Primitives

`SCaml` library module defines the primitives with their types:

```
type nat = Nat of int
let (+^) : nat -> nat -> nat = fun _ -> assert false
```

Table of primitives hard-coded in the compiler with their arities and opcodes:

```
let primitives = [
    ...
    "+^"       , (2, simple [ADD]);
    ...
  ]
```

Closure conversion

Michelson's `LAMBDA` is just a code block.

Closure conversion is required:

$$\lambda x.\, e \quad \rightarrow \quad Clos(E, x, e)$$

`fun x -> e` → `(E, LAMBDA(x, e))`

Problem: Michelson is typed.

```
if b then fun () -> b
 else fun () -> false
```

Closures may have different types.

Closure conversion in Michelson

Type inference of environment records.

```
if b then fun () -> b
     else fun () -> false
```

- `([Some b], LAMBDA (x,b))`
- `([None],    LAMBDA (x,false))`

Michelson will have real closures.

“Babylon” upgrade will introduce closures:

- `LAMBDA` generates a code block with free variables.
- `APPLY` pushes values of free variables in `LAMBDA`, which makes it a real closure.

Closure conversion will be no longer required💢