Smart Contracts in LIGO

Christian Rinderknecht

rinderknecht@free.fr

Ligo LANG

6 March 2020

Tezos Developers Day

- I joined Nomadic Labs in July 2018.
- Two of my colleagues were Suzanne Dupéron and Gabriel Alfour.
- Mi-2019, Gabriel founded Ligo LANG, a spin-off company and Suzanne and I joined him.
- Ligo LANG is funded by the Tezos Foundation to develop tools to ease the creation of distributed applications on Tezos.
- We are currently a small group of engineers located around the globe. (Matej Šima of Stove Labs worked with us and is still close.)
- We have designed a language for smart contract on Tezos, called LIGO.
- We have written a compiler from LIGO to Michelson.

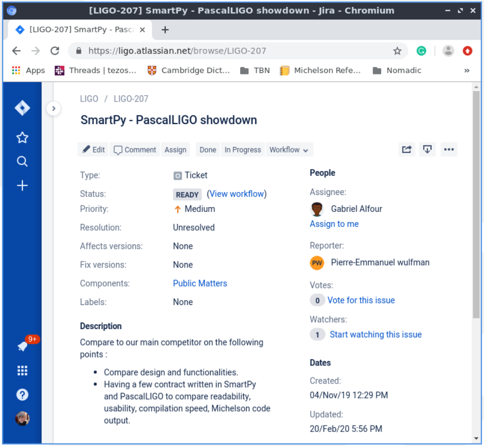
- Several tools have been developed, aiming at facilitating the adoption of LIGO.
- A VSCode plug-in is available, featuring
    - syntax highlighting,
    - one-click compilation to Michelson,
    - *dry runs* to locally execute contracts on a sandbox.
- The architecture of VSCode, with its Language Server Protocol, hopefully opens the door to plugins written in any programming language, e.g., static analysis in OCaml.
- A web-based IDE with the same set of features:
  `https://ide.ligolang.org/`

LIGO

- Ligo LANG also helps with the training of Tezos users.
- Unlike Michelson, LIGO is a language akin to what a mainstream programmer would expect.
- This means that LIGO features variables, expressions, function calls, data types, pattern matching etc.
- Nevertheless, LIGO is a Domain Specific Language (DSL) for Tezos.
- It is an open source, collective project (under the MIT licence).
- It is hosted here: `https://ligolang.org/`
- Anyone can register an issue with our gitlab server: `https://gitlab.com/ligolang/ligo/`

LIGO

- We are asked sometimes to compare LIGO with other languages.
- This is difficult because we do not know well enough the other languages.



LIGO is polyglot

- Perhaps the most striking feature of LIGO is that it comes in different concrete syntaxes, and even different programming paradigms.
- In other words, LIGO is not defined by one syntax and one paradigm, like imperative versus functional.
- There is PascaLIGO, which is inspired by Pascal, hence is an imperative language with lots of keywords, where values can be locally mutated from within the scope where they have been declared.
- There is CameLIGO, which is inspired by the pure subset of OCaml, hence is a functional language with few keywords, where values cannot be mutated, but still require type annotations (unlike OCaml, whose compiler performs almost full type inference).
- There is ReasonLIGO, which is inspired by the pure subset of ReasonML, which is a JavaScript syntax on top of OCaml.

LIGO is polyglot

- Even within PascaLIGO, two styles are possible: terse or verbose. We illustrate the terse style here and in the documentation. We plan to offer automatic style checking and two-way conversion by pretty-printing.
- We plan to provide transpilation between syntaxes, which includes pretty-printing. Two reasons:
  1. Some large owners of contracts may not want to depend too much on the given skill set of their maintainers.
  2. The more reviewed a smart contract is, the better, therefore it is beneficial to present one contract in the syntax that is better understood by a reviewer.

Some non-Tezos-specific LIGO features

- At the call site of a function, **the arguments and the environment are copied**, therefore any mutation (in PascaLIGO) will have no effect on the caller's arguments or the environment at the call site.
- LIGO features **higher-order functions**, that is, functions can be passed as arguments to others.
- There are **no user-defined recursive data types** (only predefined, like lists, sets and maps).
- Currently, there are no user-defined recursive functions, but we are going to enable very soon **tail recursive functions**.

Variant Types

- A variant type is a user-defined or a built-in type (in case of options) that defines a type by cases, so a value of a variant type is either this, or that or... and nothing else. The simplest variant type is equivalent to the enumerated types found in Java.
- In PascaLIGO:

```
type coin is Head | Tail
const head : coin = Head // Equivalent to Head (Unit)
const tail : coin = Tail // Equivalent to Tail (Unit)
```

- In CameLIGO:

```
type coin = Head | Tail
let head : coin = Head
let tail : coin = Tail
```

- In ReasonLIGO:

```
type coin = Head | Tail;
let head : coin = Head;
let tail : coin = Tail;
```

Variant Types

- A variant type is a user-defined or a built-in type (in case of options) that defines a type by cases, so a value of a variant type is either this, or that or... and nothing else. The simplest variant type is equivalent to the enumerated types found in Java.
- In PascaLIGO:

```
type coin is Head | Tail
const head : coin = Head // Equivalent to Head (Unit)
const tail : coin = Tail // Equivalent to Tail (Unit)
```

- In CameLIGO:

```
type coin = Head | Tail
let head : coin = Head
let tail : coin = Tail
```

- In ReasonLIGO:

```
type coin = Head | Tail;
let head : coin = Head;
let tail : coin = Tail;
```

Variant Types

- The names `Head` and `Tail` in the definition of the type `coin` are called **data constructors**, or **variants**.
- In general, it is interesting for variants to carry some information, and thus go beyond enumerated types.
- In the following, we show how to define different kinds of users of a system.

```
type id is nat

type user is
  Admin   of id
| Manager of id
| Guest

const u : user = Admin (1000n)
const g : user = Guest        // Equivalent to Guest (Unit)
```

Variants in CameLIGO

```
type id = nat

type user =
  Admin   of id
| Manager of id
| Guest

let u : user = Admin 1000n
let g : user = Guest
```

Variants in ReasonLIGO

```
type id = nat;

type user =
| Admin   (id)
| Manager (id)
| Guest;

let u : user = Admin (1000n);
let g : user = Guest;
```

- **Pattern matching** is similiar to the switch construct in Javascript, and can be used to route the program's control flow based on the value of a variant. Consider for example the definition of a function `flip` that flips a coin.
- In PascaLIGO:

```
type coin is Head | Tail

function flip (const c : coin) : coin is
  case c of
    Head -> Tail
  | Tail -> Head
  end
```

Pattern Matching in CameLIGO and ReasonLIGO

- In CameLIGO:

```
type coin = Head | Tail

let flip (c : coin) : coin =
  match c with
    Head -> Tail
  | Tail -> Head
```

- In ReasonLIGO:

```
type coin = Head | Tail;

let flip = (c : coin) : coin =>
  switch (c) {
  | Head => Tail
  | Tail => Head
  };
```

General Iteration in PascaLIGO

- General iteration in PascaLIGO takes the shape of general loops, which should be familiar to programmers of imperative languages as while loops.
- Those loops are of the form

```
while <condition> <block>
```

- Their associated block is repeatedly evaluated until the condition becomes true, or never evaluated if the condition is false at the start. The loop never terminates if the condition never becomes true.
- Because we are writing smart contracts on Tezos, when the condition of a `while` loops fails to become true, the execution will run out of gas and stop with a failure anyway.
- Loops make sense only in PascaLIGO because the conditional expression needs to be mutated by the body of the loop.

General Iteration in PascaLIGO

- Here is how to compute the greatest common divisors of two natural numbers by means of Euclid's algorithm:

```
function gcd (var x : nat; var y : nat) : nat is block {
  if x < y then {
    const z : nat = x;
    x := y; y := z
  }
  else skip;
  var r : nat := 0n;
  while y =/= 0n block {
    r := x mod y;
    x := y;
    y := r
  }
} with x
```

General Iteration in CameLIGO

- CameLIGO is a functional language where user-defined values are constant, therefore it makes no sense in CameLIGO to feature loops, which we understand as syntactic constructs where the state of a stopping condition is mutated, as with `while` loops in PascaLIGO.
- Instead, CameLIGO implements a **folded operation** by means of a predefined function named `Loop.fold_while`.
- It takes an initial value of a certain type, called an **accumulator**, and repeatedly calls a given function, called **folded function**, that takes that accumulator and returns the next value of the accumulator, until a condition is met and the fold stops with the final value of the accumulator.
- The iterated function needs to have a special type: if the type of the accumulator is `t`, then it must have the type `(bool * t)` (not simply `t`). It is the boolean value that denotes whether the stopping condition has been reached.

- Here is how to compute the greatest common divisors of two natural numbers by means of Euclid's algorithm:

```
let iter (x,y : nat * nat) : bool * (nat * nat) =
  if y = 0n then false, (x,y) else true, (y, x mod y)

let gcd (x,y : nat * nat) : nat =
  let x,y = if x < y then y,x else x,y in
  let x,y = Loop.fold_while iter (x,y)
  in x
```

- To ease the writing and reading of the iterated functions (here, `iter`), two predefined functions are provided: `Loop.continue` and `Loop.stop`:

```
let iter (x,y : nat * nat) : bool * (nat * nat) =
  if y = 0n then Loop.stop (x,y) else Loop.resume (y, x mod y)

let gcd (x,y : nat * nat) : nat =
  let x,y = if x < y then y,x else x,y in
  let x,y = Loop.fold_while iter (x,y)
  in x
```

Inlining

- The Michelson generator of the LIGO compiler performs several kinds of optimisations.
- One of them is **inlining**, that is, the expansion of the body of a function at its call site (with its parameters also expanded with the arguments).
- Inlining is controlled by **attributes** of constant and function declarations.
- In PascaLIGO:

```
function fst (const p : nat * nat) : nat is p.0;
attributes ["inline"];

function main (const p : nat * nat; const s : nat * nat)
  : list (operation) * (nat * nat) is
  ((nil : list (operation)), (fst (p.0,p.1), fst (p.1,p.0)))
```

Research & Development

- We are working on a more **powerful type system** which will enable the writing of more expressive contracts, featuring more type inference (less annotations) and enabling a greater variety of programming paradigms (e.g., object-oriented).
- We are writing a **certified backend in Coq**, that is, a Michelson code generator proven correct w.r.t. a formal semantics and extracted to OCaml.
- Those endeavours are not just engineering, they are instances of **applied research** and require a strong background on programming language theory.

Structure of a LIGO contract

- A LIGO contract is a series of constant and function declarations.
- The scope of those is called top-level, to distinguish declarations that may occur within functions.
- In particular, even in PascaLIGO, you cannot have mutable variables at the top-level.
- As a design pattern, there is usually one special function, which we call main function that is called with a parameter when the contract is invoked.
- The main function calls other functions according to the value of the contract parameter.
- Those functions are called entrypoints, following the Michelson convention.