LIGO/Marigold: Smart Contracts and Layer-2

Christian Rinderknecht
Christian.Rinderknecht@nomadic-labs.com
Nomadic Labs

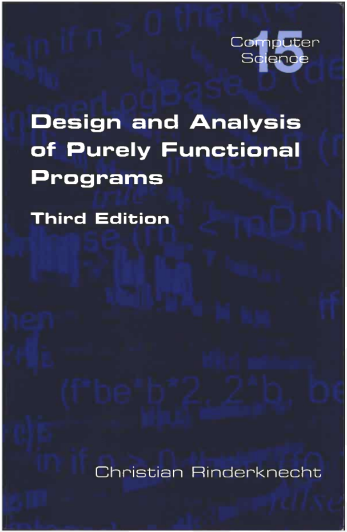
15 August 2019

Tezos London Meetup

A personal introduction

- My alma mater is *Université Pierre et Marie Curie* (UPMC, a.k.a. Paris 6).
- I did my doctoral studies at Inria, one of the most prestigious research institutes in informatics in France.
- I was a member of the team that developed the programming language OCaml.
- I went on to work as an engineer, a researcher and a professor for many years, across several countries (France, Korea, Hungary, Sweden), both in academia and private companies.
- In 2018, I joined Nomadic Labs, where most of the maintenance of the Tezos blockchain is done. My expertise is in compiler construction and functional programming. I have been working on a high-level language for writing smart contracts on Tezos.

My book about functional programming is published in London!



Nomadic Labs

- Nomadic Labs is an engineering company based in Paris.
- It is funded by the Tezos Foundation, which tasked it with developing the Tezos ecosystem.
- It focuses on R&D and is the main contributor to the Tezos blockchain itself.
- Most of the 35 engineers are doctors in informatics, specialised in programming language theory and practice, functional programming (OCaml), distributed systems, and formal verification.
- Nomadic Labs is also the home of Cortez, a mobile wallet for Tezos.
- It also helps with the training of Tezos users and other trainers.

- The economic protocol of Tezos handles a ledger of transactions as a **blockchain**. In a nutshell: a replicated database with strong access control, immutable entries and resilience to malicious users.
- Those transactions consist in the exchange of assets (tokens) between peers of the network. Each peer has an address and an account (an amount of tokens), but **a physical person can be several peers.**
- **Public-key cryptography** is used to secure the identity of the senders of transactions and to ensure that past transactions are not tempered with. An address is the hash of the peer's public key.
- The business logic between the peers is *not* recorded in the chain, that is, the knowledge of what they agree should trigger transfers of tokens.
- The economic protocol of Tezos offers the possibility to record in the chain that logic as a **smart contract**.

- A smart contract is a peer that also has a program associated to a private storage.
- That storage is writable only by the program, but is publicly readable (except by other contracts, to make them more deterministic by not depending on the current storage of another contract).
- The smart contract (by synecdoche) is executed each time a specific transaction is sent to its address.
- The transaction may include some parameters.
- Any call to a contract in a block is replicated by all the nodes in the chain, to check whether the block is valid before including it in their local view of the head of the chain.
- Once the block is validated, it is broadcasted by the underlying peer-to-peer, gossip network (as usual).

Smart Contracts on Tezos

- The validation of smart contracts therefore adds a delay in the validation of a block: the execution time of all the smart contracts in it must fit the interblock time for the chain (about one minute on Tezos).
- Therefore, each smart contract is allowed a given and fixed quantity of computation, measured in gas.
- Each instruction has an associated cost expected to be proportional to the wall-clock time and an ad hoc estimation is given by the node's client.
- A node is a server process that is accessed by a command-line client or RPCs. A node comprises a view of the chain so far, the context of the chain (a map from addresses to token amounts, used to validate transactions), and the mempool.
- When the execution exceeds the alloted gas, it is stopped, its effect on the storage is rolled back, a failed transaction is included in the block, and thus fees are collected to discourage spamming attacks.
- The economic protocol sets limits on gas per block and per transaction.

- Each smart contract has a **code size** and a **storage size**, which are allocated on every node on the chain (in their context).
- To limit the memory needed to synchronise the chain and store it, a fee is set per byte and collected when a contract is deployed (*originated*, in Tezos parlance).
- It is only at the normal termination of a contract that the effects of a contract becomes atomically visible to the network: the storage may appear modified and a list of operations may have been returned (transactions, contract creations and delegate settings) and validated.
- In particular, a smart contract can transfer tokens to other smart contracts, enabling the design of **complex distributed applications** in Tezos.

The design of Michelson

- The language native to the Tezos blockchain for writing smart contracts is **Michelson**, a Domain-Specific Language (DSL) inspired by Lisp and Forth.
- This unusual lineage aims at satisfying unusual constraints, but entails some tensions in the design.
- First, to measure stepwise gas consumption, **Michelson is interpreted**.
- On the one hand, to assess gas usage per instruction, instructions should be simple, which points to **low-level features** (a RISC-like language).
- On the other hand, it was originally thought that users will want to write in Michelson instead of lowering a language to Michelson, because the gas cost would otherwise be harder to predict.
- This means that **high-level features** were deemed necessary (like a restricted variant of Lisp lambdas, a way to encode algebraic data types, as well as built-in sets, maps and lists).

The design of Michelson

- To avoid ambiguous and otherwise misleading contracts, the layout of Michelson contracts has been constrained (e.g., indentation, no UTF-8), and a canonical form was designed and enforced when storing contracts on the chain.
- To avoid the origination of contracts that would fail due to inconsistent assumptions on the storage and temporary values, a strong static type system was designed for Michelson.
- To reduce the size of the code, Michelson was designed as a stack-based language, whence the lineage from Forth and other concatenative languages like PostScript, Joy, Cat, Factor etc. (Java bytecode would count too.)
- Programs in those languages are compact because they assume an implicit stack in which some input values are popped, and output values are pushed, according to the current instruction being executed.

The semantics of Michelson

- The semantics of each Michelson instruction is determined by the change of a prefix of the stack, that is, a segment starting at the top.
- An abstraction of that change can be captured by the **type of the instruction**.
- For example, the instruction DUP has type $\alpha :: A \rightarrow \alpha :: \alpha :: A$, meaning that if the stack before the instruction is executed has a topmost item (also called *head*) of type α, where the substack (also called *tail*) has type A, then the instruction leaves a stack with two items of type α, on top of a stack of type A. (We use the OCaml notation for consing, that is, pushing items.)
- This does not specify entirely the semantics, only the **static semantics**.
- We need a **dynamic semantics**, also called *evaluation* or *interpretation*, defined as a transition system, which states what happens to values in the stack. Here, the evaluation mimics exactly the typing judgement above:

$$\mathrm{DUP}\, x :: S \rightarrow x :: x :: S$$

The type system of Michelson

- Whilst the types of instructions can be polymorphic, their instantiations must be monomorphic, hence instructions are not first-class values and cannot be partially interpreted.
- This enables a simple static type checking, as opposed to a complex type inference. It can be performed efficiently: contract type checking consumes gas.
- Basically, type checking aims at validating the composition of instructions, therefore is key to safely composing contracts (concatenation, activations).
- Once a contract passes type checking, it cannot fail due to inconsistent assumptions on the storage and other values (there are no null values, no casts), but it can still fail for other reasons: division by zero, token exhaustion, gas exhaustion, or an explicit FAILWITH instruction. This property is called type safety. Also, such a contract cannot remain stuck: this is the progress property.
- The existence of a formal type system for Michelson, of a formal specification of its dynamic semantics (evaluation), of a Michelson interpreter in Coq, of proofs in Coq of properties of some typical contracts, all those achievements are instances of formal methods in Tezos.

A voting contract in Michelson

- The following example has been ripped off the paper *Introduction to the Tezos Blockchain*, V. Allombert, M. Bourgoin, J. Tesson, HPCS, 2019, July 15–19, Dublin, Ireland.
- Disclaimer: They are my colleagues.
- We want to write a smart contract in Michelson that records votes of peers for their favourite candidate.
- For the sake of simplicity, a candidate is denoted by a string.
- The vote is open to all peers for a fee, and they have to pick one candidate in a fixed list.
- The state of the stack at a given point will be written in a comment starting with #.

A voting contract in Michelson

- We begin by defining the storage as a mapping from the candidates (`string`) to the number of corresponding votes (`int`):

```
storage (map string int);
```

- Then we specify the type of the parameter of the contract, that is, the vote:

```
parameter string;
```

- The execution starts with a stack containing a single pair made of the parameter and the storage:

```
code {
  # (vote, storage)
```

A voting contract in Michelson

- First, we check if the caller sent us enough tokens to vote, and, if not, we fail:

```
AMOUNT;
# amount::(vote, storage)
PUSH mutez 5000;
# 5000::amount::(vote, storage)
IFCMPGT { PUSH string "Insufficient fee."; FAILWITH } {};
# (vote, storage)
```

- The instruction `AMOUNT` pushes onto the stack the amount sent by the voter.
- The instruction PUSH pushes a typed constant.
- The instruction `IFCMPGT` compares the first two numbers on the stack, and, if the topmost is greater than the next, the first sequence (between braces) is executed, else the second one (which is empty here because of `FAILWITH`).
- The instruction `FAILWITH` reads the value on top of the stack, terminates the execution, and signals an error (denoted by the value) to the node and the client.

A voting contract in Michelson

- In order to check the current number of votes for the same candidate, we start by duplicating the topmost item:

```
DUP;
# (vote, storage)::(vote, storage)
```

- We then destructure the first pair:

```
UNPAIR;
# vote::storage::(vote, storage)
```

- We can now consult the map in storage for the current count:

```
GET;
# (Some current | None)::(vote, storage)
```

The top of the stack is `None` if `vote` was not a valid choice, and `(Some current)` if there were `current` votes already.

A voting contract in Michelson

- If None, we must fail; otherwise, we increment the number of votes:

```
IF_SOME { PUSH int 1; ADD; SOME }
        { PUSH string "Unknown candidate."; FAILWITH }
# Some (current+1) :: (vote, storage)
```

- We need now to update the storage with the new vote. For that, we need to fish out the vote:

```
DIP { UNPAIR };
# Some (current+1) :: vote :: storage
```

  The instruction DIP applies its sequence of instructions (here UNPAIR) to the item just below the top.

- Then we need to reorder the two first items, like so:

```
SWAP;
# vote :: Some (current+1) :: storage
```

A voting contract in Michelson

- We can now update the map:

```
UPDATE;
# storage'
```

- All contracts must normally end by leaving in the stack a pair made of a list of operations and the new storage.
- In this instance, the list is empty because all we do is update the internal storage of the contract. So we push an empty list of operations and pair it with the new storage:

```
  NIL operation;
  PAIR;
  # ([], storage')
}
```

The full Michelson contract

```
storage (map string int);
parameter string;
code {
  AMOUNT;
  PUSH mutez 5000;
  IFCMPGT { PUSH string "Insufficient fee."; FAILWITH } {};
  DUP;
  UNPAIR;
  GET;
  IF_SOME { PUSH int 1; ADD; SOME }
          { PUSH string "Unknown candidate."; FAILWITH }
  DIP { UNPAIR };
  SWAP;
  UPDATE;
  NIL operation;
  PAIR;
}
```

Assessment of the contract

- We could improve upon this design by enforcing a deadline on the ballot, or by granting the right to add new candidates in exchange of a fee.
- Anyhow, we had to modify, sometimes deeply, the stack in order to fetch and prepare the data for the instructions.
- Moreover, some instructions above were actually **macros** (`IF_SOME`, `IFCMPGT`), which are actually expanded into a sequence of atomic instructions by the Michelson parser, so the canonical contract on the chain is longer.
- Programming in **Michelson is fun** and might even save the Caps Lock key from extinction.
- *But does it scale up to larger and involved contracts with high stakes?*
- This is where **LIGO comes into play**.

- Like Michelson, LIGO is a programming language for writing smart contracts on Tezos.
- 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 remains a DSL for the Tezos blockchain, so not all the usual bells and whistles of a mainstream language are available.
- It is hosted here: `https://ligolang.org/`
- It is an open source, collective project (under the MIT licence).
- Go and try the tutorial!
- (If you guessed why the name “Michelson”, you will guess why “LIGO”.)

- 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 after they have been annotated with their types (declaration).
- 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).
- And more to come! You can propose your own front-end for LIGO and we can help with its integration in the distribution.

A voting contract in PascaLIGO

```
type store is map (string, nat)
type return  is list (operation) * store

function vote (const ballot : string; var store : store) : return is
  begin
    if   amount <= 5000mutez
    then failwith ("Insufficient fee.")
    else case store[ballot] of
                     None -> failwith ("Unknown candidate.")
          | Some (current) -> store[ballot] := current + 1n
          end
  end with ((nil : list (operation)), store)
```

- **Easier to understand**, even though the generated Michelson is not so neat.

A voting contract in CameLIGO

```
type store = (string, nat) map
type return = operation list * store

let vote (ballot, store : string * store) : return =
  if   amount <= 5000mutez
  then (failwith "Insufficient fee." : return)
  else match Map.find_opt ballot store with
          None -> (failwith "Unknown candidate." : return)
        | Some current ->
            let store = Map.update ballot (Some (current + 1n)) store
            in (nil : operation list), store
```

A crowdfunding contract in PascaLIGO

- A simple crowdfunding contract can be called in three manners:
  1. a backer sends some funds before the deadline has passed, and only once;
  2. a backer claims their funds after the deadline has passed and the goal has not been reached;
  3. the owner withdraws the funds after the deadline has passed and the goal has been reached;
  4. all other cases are errors.
- First, we define the storage (or *store*):

```
type store is record
    goal     : tez;
    deadline : timestamp;
    backers  : map (address, tez);
    funded   : bool;               // Optional terminator semicolon
  end
type return is list (operation) * store
```

A crowdfunding contract in PascaLIGO

- The function to contribute to the crowdfunding:

```
function back (var param : unit; var store : store) : return is
  begin
    if   now > store.deadline
    then failwith ("Past deadline.")
    else case store.backers[sender] of
             None -> store.backers[sender] := amount
           | Some (x) -> skip
           end
  end with ((nil: list (operation)), store)
```

A crowdfunding contract in PascaLIGO

- The function to claim funds is as follows:

```
function claim (var param : unit; var store : store) : return is
  begin
    var op : list (operation) := nil;
    if now <= store.deadline then failwith ("Too early.") else
    case store.backers[sender] of
        None -> failwith ("Not a backer.")
    | Some (asset) ->
          if balance >= store.goal or store.funded
          then failwith ("Goal reached: no refund.")
          else begin
                  const dest : contract (unit) = get_contract (sender);
                  op := list transaction (Unit, asset, dest) end;
                  remove sender from map store.backers
                end
     end
  end with (op, store)
```

- The function to withdraw the funds is as follows:

```
function withdraw (var param : unit; var store : store) : return is
  begin
    var op : list (operation) := nil;
    if sender = owner then
      if now >= store.deadline then
        if balance >= store.goal then
           begin
             store.funded := True;
             op := list transaction (unit, owner, balance)] end;
           end
        else failwith ("Below target.")
      else failwith ("Too early.")
    else skip
  end with (op, store)
```

Some peculiarities of LIGO

- LIGO features multiple syntaxes and paradigms.
- Even within PascaLIGO, two styles are possible: terse or verbose. We illustrated the verbose style here. We plan to offer automatic style checking and two-way conversion by pretty-printing.
- LIGO features **high-order quotations**, but not full-fledged closures.
- This means that, at the call site, **the arguments and the environment are always copied**, therefore any mutation (in PascaLIGO) will have no effect on the caller's arguments or the environment, except with built-in iterators.
- Quotations can be passed as arguments to others.

Quotations in LIGO

```
function iter (const delta : int; const l : list (int)) : int is
  var acc : int := 0;
  procedure aggregate (const i : int) is
    begin
      acc := acc + i
    end
  begin
    aggregate (delta);          // Has no effect on acc
    list_iter (l, aggregate); // Has an effect on acc
  end with acc
```

Tooling for LIGO

- Several tools are currently being developed, aiming at facilitating the adoption of LIGO.
- An IDE based on VSCode has been made available, featuring
  - syntax highlighting,
  - one-click compilation to Michelson,
  - *dry runs* to locally execute contracts on a sandbox,
  - a reactive counter estimating the gas consumption.
- 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.
- An SDK to make it simple for developers to make applications involving LIGO contracts, e.g., mobile smart wallets or web applications.

Research & Development on the LIGO compiler

- 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 and extracted to OCaml from its specification.
- Those endeavours are not just engineering, they are instances of **applied research** and require a strong background on programming language theory.
- But... you are welcome to join our effort, research or development, even if you do not have a PhD, of course.

The team behind LIGO

- I wrote "we" and I introduced myself as working for Nomadic Labs.
- Nomadic Labs has been helping with spinning off LIGO into a company, which I will formally join next month. Both companies are funded by the Tezos Foundation.
- We are an international team.
- Five of us are based in Paris, and eight are contractors spread around the globe, all working as one team (it is not just about deliverables).
- I became the COO, but I also coach new hires and work on an experimental parser.
- If you are passionate about compilers, distributed applications on the blockchain, the web, please get in touch with us.

- The LIGO project is the first stage of a larger project called **Marigold**.
- Marigold aims at **accelerating transactions**, including smart contracts, with **layer-2 innovations** (sometimes not adequately called *off-chain* or *side-chain* solutions).
- We plan to amend the Tezos protocol itself to create **tokens (Non Fungible Tokens) with Plasma**, which could be done with smart contracts, but much slower.
- This is possible because Tezos features a **metaconsensus protocol**, which empowers peers to propose amendments (i.e., source code) and submit their patches to a vote.
- For LIGO, we already proposed some Michelson instructions to make the code generated by the LIGO compiler faster (fetching and committing data deep within the stack).
- Cryptoeconomics Labs (Japan) (`https://www.cryptoeconomicslab.com`) work on a **general Plasma**, that is, a Plasma that is extended to encompass a larger class of contracts than just tokens, and of interest to Marigold.