Compiling a low-level language to Michelson

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

July 30, 2019

Acknowledgements

I'd first like to thank Bruno Bernardo for offering this internship and learning opportunity to me. I also thank him, along with Raphaël Caurdelier and Julien Tesson, both for helping me when I encountered difficulties, and for proofreading this document. Finally, I'd also like to thank the whole Nomadic Labs team for being so welcoming.

Contents

Context

0.1 The Tezos blockchain

The Tezos ecosystem is centered around the Tezos blockchain. A blockchain is a decentralized database, usually focused on storing data about transactions made by its users. Blockchains usually use a dedicated token, or cryptocurrency, to quantify these transactions. These currencies, while they can't really (yet) be used to buy physical goods outside of the blockchain, can usually be exchanged for traditional government-established currencies, at varying rates. The currency of the Tezos blockchain is called *tez*.

The Tezos blockchain, while currently an outsider compared to older, more successful blockchains (like Bitcoin or Ethereum), has several unique features that make it appealing. Namely, the chain gives token holders governance over the chain, by allowing its core protocol to be amended by votes of the community. Nomadic Labs is one of the most active actors in the Tezos ecosystem, maintaining and improving the Tezos open-source codebase; in particular, they've worked on updates of the protocol, including the Babylone update, which is being voted at time of writing. More details on these features, which won't be discussed in this report, can be found in the Tezos White Paper [1].

More to the point of this work, Tezos puts a strong emphasis on safety : the chain is implemented in OCaml, a statically typed programming language, which prevents some runtime errors. As we will see below, formal verification is also a focus on the work around the chain, and will be the focus of this report.

0.2 The Michelson programming language

Tezos, as well as allowing regular "humans" to create accounts (referred to as tz1 accounts), also allows users to run programs on the blockchain. These programs are often called "smart contracts", since most of them are used to automate transactions between two parties. Once the contract has been uploaded (originated) on the blockchain, it can then be called by any other account (being a human user or another smart-contract) by sending a transaction, containing at least a small amount of tez to cover processing fees, as well as the parameters of the contract. The contract itself holds a balance and can use its tokens to forge its own transactions.

The programming language used to write smart contracts for Tezos is called Michelson [2]. Michelson is a statically typed stack-based programming language, meaning that the programmer has to explicitly manipulate the typed stack of the interpreter, using low level instructions. Examples of Michelson contracts (and there specification) are given in section 1.1.1

As Michelson is a low level language, it can be very tedious to write: to manipulate the stack explicitly, the programmer needs to keep in mind the state of the stack at every point in the program (fortunately an Emacs michelson-mode exists to display such information). The language was purposefully designed to be simple, which makes it easy to specify and proof specifications of programs, as we'll see below. However, it also makes writing smart-contracts in Michelson a bit tedious, and creates a barrier to entry for new programmers wanting to work in the Tezos ecosystem.

Since the Michelson interpreter is already part of the Tezos protocol (and can only be changed by amendment and vote of the community), it's a good base to build upon : creating higher-level programming languages that would compile to Michelson would facilitate the writing of smart-contracts. To accomplish this goal, a first step will be to establish an intermediate programming language, Albert, abstracting away some of the low-level hurdles of Michelson (the stack manipulations in particular). To keep formal verification in mind, the Albert language semantics will be specified, and the compiler proved correct.

Chapter 1

The Mi-Cho-Coq framework

The Mi-Cho-Coq project consists in a formalisation of Michelson's semantics, written in Coq. Coupled with a weakest-precondition calculus, it allows its users to specify properties of Michelson programs, and to prove them using the Coq proof assistant. We'll see below a few proofs of relatively simple contracts.

1.1 Proving the specification of smart-contracts

1.1.1 Weather insurance

The first contract we'll prove using Mi-Cho-Coq is the weather insurance contract. Its principle is simple : the contract is originated containing, in its storage the addresses of two other accounts, a threshold of rain, and the public key of a third party oracle. The contract is also originated with a balance. At any point, the oracle can send an integer representing the actual level of rain, as well as its signature. The contract then verifies that the signature is correct, and if it is, compares the level of rain with its threshold : it then sends its whole balance to one of the addresses saved in its storage, depending on the level of rain relative to its threshold.

```
parameter (pair (signature %received_sig) (nat :rain %rain_level));
storage (pair (pair (address %under_addr)
                    (address %over_addr))
              (pair (nat %rain_thresh) (key %weather_service_key)));
code { DUP; DUP;
       CAR; MAP_CDR{PACK ; BLAKE2B};
       SWAP; CDDDR;
       DIP {UNPAIR}; CHECK_SIGNATURE ; # Check if the data has been correctly signed
       ASSERT; # If signature is not correct, end the execution
       DUP; DUP; DUP; DIIIP{CDR}; # Place storage type on bottom of stack
       DIIP{CDAR}; # Place contracts below numbers
       DIP{CADR}; # Get actual rain
       CDDAR; # Get rain threshold
       CMPLT; IF {CAR} {CDR}; # Select contract to receive tokens
       CONTRACT unit; BALANCE; UNIT; TRANSFER_TOKENS; # Setup and execute transfer
       NIL operation; SWAP; CONS;
       PAIR };
```

Let's take a closer look at how this contract works. We start by duplication the parameters twice on line 1. We then PACK the sent rain level and hash it using the BLAKE2B instruction on line2. We use this encrypted data to check that the sent signature was indeed the one of the weather service on lines 4-5; if not, the execution fails.

On lines 6 to 8, we manipulate the stack so has to have the stored contracts after the actual rain quantity and the threshold. We then compare on line 9 the threshold to the level; if the level is below the threshold, we chose the `under_addr`, and if it is above, we choose the `over_addr`. We use this address to forge a transaction, sending all the tokens held by the contract on line 10. Finally, on line 11-12, we construct the return type of the contract (a pair containing the sent operations, i.e. the transfer, and

the unchanged storage).

Given the description above, we can easily give a specification of the contract. In the followings specifications, I'll note as "preconditions" the conditions that must be verified for the contract not to call FAIL (or a related macro). The "postconditions" fully describe the new state of the storage at the end of the execution, as well as the potential generated operations.

Preconditions:	`SignatureCorrect`(weather_service_key, received_sig)
Postconditions:	new_storage = storage
	rain_thresh $<$ rain_level $\Rightarrow$ returned_operations = [transfer_tokens balance over_addr]
	rain_thresh $\geq$ rain_level $\Rightarrow$ returned_operations = [transfer_tokens balance under_addr]

Although this is a pretty simple specification that was pretty simple to prove (the proof can be found at `https://gitlab.com/Vertmo/mi-cho-coq/blob/more-contract-proofs/src/contracts_coq/testsuite/mini_scenarios/weather_insurance.v`), one point stands out : we only specify the production of the operations at the end of the contract, and not their effect (in this case, that the balance of the contract would be drained after any successful execution). This is because Mi-Cho-Coq only specifies the execution of the contract, and not the interactions with the exterior world.

1.1.2 Vote

Another specified smart-contract is the Vote contract. As its name implies, it allows users to vote for a candidate among a list set at origination. The contracts retains the number of votes towards each candidate. Any user can vote any number of times, but must send 5 tez (or 5000000μtez) with each vote, otherwise the transaction is refused.

```
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
     }
```

Let's look in more details at our voting program: First, the description of the storage and parameter types is given on lines `1-2`. Then the code of the contract is given. On line `5`, AMOUNT pushes on the stack the amount of (in μtez) sent to the contract address by the user. The threshold amount (5tez) is also pushed on the stack on line `6` and compared to the amount sent: COMPARE pops the two top values of the stack, and pushes either -1, 0 or 1 depending on the comparison between the value. GT then pops this value and pushes `true` if the value is 1. If the threshold is indeed greater than the required amount, the first branch of the IF is executed and FAIL is called, interrupting the contract execution and cancelling the transaction.

If the value was `false`, the execution continues on line `9`, where we prepare the stack for the next action: DUP copies the top of the stack, we then manipulate the tail of the stack while preserving it's head using DIP: there, we take the right element of the `(chosen, candidates)` pair with CDR, and we duplicate it again. By closing the block guarded by DIP we recover the former stack's top, and the following line takes its left element with CAR, and duplicates it.

On line `12`, we use DIP to protect the top of the stack again. GET then pops `chosen` and `candidates` from the stack, and pushes an option containing the number of votes of the candidate, if it was found in the map. If it was not found, ASSERT_SOME makes the program fail. On line `15`, the number of votes is incremented by ADD, and packed into an option type by SOME.

We then leave the DIP block to regain access to value at the top of the stack (`chosen`). On line `18`, UPDATE pops the three values remaining on top of the stack, and pushes the `candidates` map updated with the incremented value for `chosen`. Finally, we push an empty list of operations with NIL `operation`, and pair the two elements on top of the stack to get the correct return type.

We can then specify this smart-contract as follows (`amount` refers to the quantity of μtez sent by the caller for the transaction):

Preconditions:	`amount` > 5000000 μtez
	chosen $\in$ `Keys`(storage)
Postconditions:	returned_operations = [ ]
	$\forall$ c, c $\in$ `Keys`(storage) $\iff$ c $\in$ `Keys`(new_storage)
	new_storage[chosen] = storage[chosen] + 1
	$\forall$ c $\in$ `Keys`(storage), c $\neq$ chosen $\Rightarrow$ new_storage[c] = storage[c]

Despite not looking much more complicated than the weather insurance smart contract, the specification of the vote smart-contract was slightly harder to prove (the Coq proof script is available at `https://gitlab.com/nomadic-labs/mi-cho-coq/blob/master/src/contracts_coq/vote.v`). Indeed, in order to prove most of the postconditions, we had to prove a few lemmas on the map datastructure, and the relations between the functions (*mem*, *get* and *update*) used to manipulate it. These lemmas were integrated in the library of Mi-Cho-Coq, and will most likely be useful to prove other contracts in the future.

1.2 Tooling Michocott

Michocott is an implementation of Mi-Cho-Coq, using the Ott [3] programming language. From a set of grammar definitions and rules, Ott can generate typing and semantic rules for Coq (and other proof assistants), a Menhir parser, as well as LaTeXdocumentation. We'll use Ott later on to define the grammar, typing and semantic rules for the Albert programming language in section 2.2.

1.2.1 Documentation generation

In order to adapt Ott to Michelson's needs, and especially the format of its online documentation (`http://tezos.gitlab.io/mainnet/whitedoc/michelson.html`), which uses the ReStructuredText (reST) format, I added a reST output mode to Ott. Given the architecture of the Ott compiler, which separates frontend from backend pretty well, it was fairly easy to add a new RST backend.
To make the output practical for integrating pieces of the documentation with human-written text, I also added a mode allowing to generate one *.rst* file per typing/semantic rule. The fragments can then be integrated into a single documentation page using the reST `.. include::` directive.

Another goal of automated document generation is to check that the typing and semantic rules specified in Michocott (and Mi-Cho-Coq) are indeed the same as the ones actually implemented in the Michelson interpreter running in the chain. In order to easily compare the two, we need a structured output which can be automatically generated from both Michocott and the interpreter. Luckily for us, on the interpreter side, we already have the start of a solution : Alexandre Doussot developped, for his own tool (try-michelson `https://gitlab.com/nomadic-labs/try-michelson`) a tool capable to parse the Michelson documentation (which, while not being automatically generated from the interpreter code, is kept updated and in check by the Tezos developers).

In order to complete the chain between the interpreter and Michocott, I added a JSON output to Ott. An example of the JSON output is as follows :

```
{"op":"CDR",
  "ty":[{
     "name":"t_fun1_CDR",
     "premises":[],
     "conclusion":"CDR :: pair ty1 ty2 -> ty2"}],
  "semantics":[{
      "name":"bs_CDR",
      "premises":[],
      "conclusion":"CDR / ( Pair d d' ) : S => d' : S"}]
},
```

The implementation of the output is unfortunately, highly specific to Michelson's need (in particular, it sorts and organizes the rules according to the instruction they relate to), so it's very unlikely it will be merged into Ott proper, and will probably remain in a Nomadic-specific fork.

1.2.2 Parser and lexer improvements

In addition to its specification and documentation capabilities, Ott can also, using the grammar rules defined for a language, generate both an Ocamllex lexer and a Menhir parser (only for OCaml, but a port to use the Coq version of Menhir should be doable with minimal modifications). These functionalities however are still, in the main branch of Ott, a bit lackluster, and I proposed the following changes to fix some of the issues I encountered while using the lexer generation (these changes can be seen in this GitHub PR : `https://github.com/ott-lang/ott/pull/52`)

- Order of the tokens : In order for the tokens to be correctly lexed, a token prefixing another must be placed after the longer token in the *.mll* file : this is because OCamllex always select the first matching token it encounters, and therefore would always select the prefix if it was first in the list. Fortunately, it's sufficient to sort the tokens by decreasing length to solve this problem. More difficult is the problem of regular expressions that can be included in one another. I decided to simply place the metavars (which are most likely to use regex) at the end of the list; a better method could surely be implemented to construct topological sort of regex, but this would really complicate the program and the running time of Ott, for really small benefits.

- Type conversion : Regarding the metavars, some of them are often used to define a token containing a value of a type different than string (typically, to parse integer or floats). The lexer generation didn't account for this, so I added a feature to automatically add a `string_of_XXX` conversion to the token lexing. The features get the relevant type from the metavar declaration in the Ott source file. As of this PR, I've only added the conversions `string_of_int`, `string_of_bool` and `string_of_float`, but it could be interesting to allow users to define and use their own arbitrary conversion function (however, that would require some heavier modification of the code, and most likely be a really niche feature)

- Location : Finally, I completed the calculations of lexer locations, which didn't update when uncountering a new line ("\n" character), because the call to `Lexing.new_line lexbuf` was missing. This is a tiny change, but it allows for parsing error message to be correct and helpful.

Chapter 2

The Albert Programming Language

2.1 Overview

Albert is a higher-level programming language than Michelson [4]. The main difference with Michelson is that, in Albert, the stack is abstracted by a record with named variables, which makes their manipulations easier. The variables in the main "stack" record are handled linearly, which means they are consumed when used, and must be explicitly duplicated if they need to be used twice. This reflects Michelson's behaviour, where values on the stack are indeed consumed by instructions using them.

The linear typing is handled by Albert's typing rules, which keeps track of the content of the stack (without its ordering) at any point during the execution. For instance, below is a type of an instruction that assigns the value contained in a variable `foo` to a variable `bar`.

$$\overline{\Gamma \vdash bar = foo : \{foo : ty\}@rty \Rightarrow \{bar : ty\}@rty}$$

Where `rty` is the type of the other values contained in the stack, and `@` denotes the join operation for record types. It's important to note that the join operator is not a simple append, and is commutative. It's actually equivalent to the `merge` operation in MergeSort, with the fields of the record sorted by lexicographic order of the labels. This canonical form is also extended to record types, where fields must be sorted in lexicographic order for the type to be deemed "well formed".

The semantics of this simple assign operation mirrors its typing (the `@` join operation used here is similar to the one used for record types).

$$\overline{E \vdash bar = foo/\{foo = val\}@rval \rightarrow \{bar = val\}@rval}$$

2.1.1 Types

In order to fully use the possibilities of Michelson, Albert implements all the basic types implemented by Michelson, that is : `int`, `nat`, `mutez`, `timestamp` for numeric types, `string` and `bytes`. Albert also implements the same data structures as Michelson, `list`, `set` and `map`.

Moreover, Albert generalizes `pair` types with `record`, and `or`, `option` and `bool` by n-ary `variant` with arbitrary constructors. For ease of use, Albert also adds these specific types to its core language, making them equivalent to the similar record type (for instance, `option ty` is equivalent to the variant `[None : unit | Some : ty]`).

2.1.2 Example smart-contract

Below is the classic vote smart-contract, rewritten in Albert. It's notable that although the code is clearly longer than the Michelson one, it's also more comprehensible from an imperative programming frame of mind.

```
type storage_ty = { threshold : mutez; votes: map string nat }

def vote :
  { store : storage_ty ; param : string } →
```

```
{ operations : list operation ; out_storage : storage_ty } =

 { car = store0; cdr = store1 } = dup store;
 threshold = store1.threshold;
 { car = threshold; cdr = threshold_copy } = dup threshold;
 ok = amount < threshold;
 match ok with
    True →state = store0.votes ;
            { car = state0; cdr = state1 } = dup state;
            { car = param0; cdr = param1 } = dup param;
            prevote_option = state1[param1];
            prevote = assert_some prevote_option;
            one = 1;
            postvote = prevote + one;
            postvote = Some postvote;
            final_state = {| state0 with param0 |→ postvote |};
            out_storage = { votes = final_state; threshold = threshold_copy };
            operations = []
  | False →
            failwith "you're␣so␣cheap!"
 end
```

Two difficulties still arise from writing an Albert contract, and create verbosity : first, the need to constantly assign every intermediate value to a variables, and second, the need to explicitly duplicate each resource the program has to use twice. These are unfortunately necessary in order to keep Albert's resource management close to Michelson's stack based management.

More Albert smart-contracts can be found in appendix A.

2.2 Formal semantics

I started my work on Albert by writing down the formal semantics of the language, mostly by following the typing rules (already written before the start of my internship). I'll give below the highlights of the rules.

2.2.1 Bases of the language

$$(\text{INSTR\_ASSIGN})\ \frac{E \vdash rhs/val- > val' \qquad E|-lhs/val' \rightarrow val''}{E \vdash lhs = rhs/val \rightarrow val''}$$

Most of the language is actually based on the `lhs = rhs` assignment instruction (as seen above), which is necessary to ensure all calculated values are indeed named and stored. An Albert program therefore resembles a series of assignments, with the left-hand side being either a simple variable, or a record destructuring. It's important to note that, since an instruction always manipulates a record (representing Michelson's stack) the input value of `rhs` and output value of `rhs` are necessarily records, while the value passed from `rhs` to `lhs` can be anything. We'll note below the formal semantics of these two left-hand sides.

$$(\text{LHS\_VAR})\ \frac{}{E \vdash var/val \rightarrow \{var = val\}}$$

$$(\text{LHS\_RECORD})\ \frac{}{E \vdash \{l_1 = x_1; ..; l_n = x_n\}/\{l_1 = val_1; ..; l_n = val_n\} \rightarrow \{x_1 = val_1; ..; x_n = val_n\}}$$

Actual calculations are performed in the right-hand side of assignment instructions. Right-hand sides can first be a simple argument: either a constant value, a variable or a record constructed from variables; as arguments are called from `rhs`, their input value is always a record (representing the stack).

$$(\text{ARG\_VAR})\ \frac{}{E \vdash_{arg} var/\{var = val\} \rightarrow val}$$

$$(\text{ARG\_VAL})\ \frac{}{E \vdash_{arg} val/\{\} \to val}$$

$$(\text{ARG\_RECORD})\ \frac{}{E \vdash_{arg} \{l_1 = x_1; ..; l_n = x_n\}/\{x_1 = val_1; ..; x_n = val_n\} \to \{l_1 = val_1; ..; l_n = val_n\}}$$

$$(\text{RHS\_ARG})\ \frac{E \vdash_{arg} arg/val \to val'}{E \vdash_{rhs} arg/val \to val'}$$

Right-hand sides can also be a function application (to arguments). Functions can either be primitives (for instance, the dup instruction is represented as a function) or user-defined. A user defined function is found in the environment (E), and is simply seen as an instruction (as in Michelson, all functions are global).

$$(\text{F\_FVAR})\ \frac{fvar = instruction \in E \qquad E \vdash_{ins} instruction/val \to val'}{E \vdash_{f} f/val \to val'}$$

$$(\text{F\_DUP})\ \frac{}{E \vdash_{f} dup/val \to \{car = val; cdr = val\}}$$

$$(\text{RHS\_APP})\ \frac{E \vdash_{arg} arg/val \to val' \qquad E| -_{funct} f/val' \to val''}{E \vdash_{rhs} farg/val \to val''}$$

Right-hand sides can also act directly on record, either by projection (taking a specific fields of the record by name) or by updating them (which of course consumes other values on the stack).

$$(\text{RHS\_PROJECTION})\ \frac{\{l = val\}@rval = rval'}{E \vdash_{rhs} var.l/rval' \to val}$$

$$(\text{RHS\_UPDATE})\ \frac{\{l_1 = val'_1; ..; l_n = val'_n\}@rval = rval' \qquad \{l_1 = val_1; ..; l_n = val_n\}@rval = rval''}{E \vdash_{rhs} \{var\ with\ l_1 = var_1; ..; l_n = var_n\}/\{var = rval'; var_1 = val_1; ..; var_n = val_n\} -> rval''}$$

Right-hand sides also include most of the arithmetic operations on numerical data. I give here both the + rule, and the $/mod$ rule, interesting in that it returns a pair containing the quotient and the remainder of the euclidean division. The − and × rules are similar to the + ones.

$$(\text{RHS\_PROJECTION})\ \frac{}{E \vdash_{rhs} (x_1 + x_2)/\{x_1 = nv_1; x_2 = nv_2\} \to nv_1 + nv_2}$$

$$(\text{RHS\_UPDATE})\ \frac{}{E \vdash_{rhs} (x_1\ /mod\ x_2)/\{x_1 = nv_1; x_2 = nv_2\} \to \{quotient = nv_1/nv_2; remainder = nv_1\%nv_2\}}$$

Apart from assignments, the instruction language is also completed with a few specific instructions : for instance, the drop instruction, which behave similarly to the DROP instruction in Michelson. It's a bit less useful in Albert however, as useless variable existing in the main record is not an annoyance for the programmer, who can simply ignore it.

$$(\text{RHS\_UPDATE})\ \frac{}{E \vdash_{ins} DROPvar/\{var = val\} \to \{\}}$$

The instructions are linked together as sequences of instructions. The sequence rule is the following:

$$(\text{INS\_SEQ})\ \frac{E \vdash_{ins} I_1/val \to val' \qquad E \vdash_{ins} I_2/val' \to val''}{E \vdash_{ins} I_1; I_2/val \to val''}$$

This seems pretty straightforward; however, we quickly see that the input and output values used by instructions (which are record values representing the stack) are incomplete: as the current rules are laid out, the only variables in the stack present in val are the one actually consumed by I_1, and the ones present in val' are therefore either the one produced by I_1, or the one consumed by I_2, which are not necessarily the same. We therefore need to add a new rule, to allow for the passed stack fragments to be completed by unused variables. We call this rule `frame`, and specify it below.

$$(\text{INS\_SEQ})\ \frac{E \vdash_{ins} instruction/rval \to rval' \qquad rval@rval'' = rval_1 \qquad rval'@rval'' = rval_2}{E \vdash_{ins} instruction/rval_1 \to rval_2}$$

This rule is, of course, mirrored in the typing of the instruction.

Using these rules (and the typing rules), I proved a few important properties on the base fragment of the language. Firstly, I proved the subject reduction property, which specifies that reducing an instruction according to the semantic rules preserves typing. We formally write this property (for instructions) as

$$\text{SUBJECT\_REDUCTION}\ \frac{G \vdash_{val} v : ty \qquad G \vdash_{ins} ins : ty \Rightarrow ty' \qquad G \vdash_{ins} ins/v \rightarrow v'}{G \vdash_{val} v' : ty'}$$

with similar properties on right-hand sides, arguments, functions and left-hand sides. This property allows us to "connect" typing and semantic rules. Although it seems simple to prove in most cases, since typing and semantic rules are already quite close, in practice it proved a hassle for some of the rules, as the Coq representation of the rules generated by Ott often uses `List.map` with pattern destructuring and restructuring, and can be difficult to manipulate.

Using this first property, it becomes easy to prove the progress property, which tells us that a well-typed term can always be evaluated. This property is formally defined as follows:

$$\text{PROGRESS}\ \frac{G \vdash_{val} v : ty \qquad G \vdash_{ins} ins : ty \Rightarrow ty'}{\exists v', G \vdash_{ins} ins/v \rightarrow v'}$$

again, with similar properties defined on the other terms of our language. The progress property is extremely important, as it guarantees that interpretation of the program cannot get stuck if the program is well-typed. In the case of Albert, we're not planning on building an interpreter, but this property is still useful to show the completeness of the semantic rules, as it can only be proved if every typing rule as a corresponding semantic rule. We would also want to be able to prove that typing and semantics are preserved when compiling to Michelson, which already respects both subject reduction and progress (as most well defined programming languages do).

2.2.2 Data structures

Variants

As we've seen above, Albert's `variant` types generalize the `or`, `option` and `bool` types. Variants are therefore the dual of records, with the caveat that it is not possible to construct an empty variant (by choice, as Michelson does not have an empty type it could correspond to). Variants offer two main operations to the user : constructing a `variant` value using a constructor, and pattern-matching on a `variant` value.

Constructor are determined by a label, and applied (as a function) on a single value. When constructing a `variant` value, the user must indicate the full type of the variant value (meaning, all the different constructors and their types). This choice was made in order to simplify the work of the type-checker of Albert (since Albert is an intermediate language designed to be compiled for, and not for programs to be manually written for, this should not be too much of an issue for future users). Below is the typing rule for `variant` value and constructor application, as well as the related semantic rule.

$$(\text{VAL\_CONSTR})\ \frac{G \vdash_{val} val : ty \qquad [constructor : ty]@vty' = vty}{G \vdash_{val} (constructor\ val : vty) : vty}$$

$$(\text{F\_CONSTR})\ \frac{[constructor : ty]@vty' = vty}{G \vdash_{funct} constructor\ vty : ty \Rightarrow vty}$$

$$(\text{F\_CONSTR})\ \frac{}{E \vdash_{funct} constructor\ vty/val \rightarrow (constructor\ val : vty)}$$

Like records, variants have a "canonical form", where constructors are ordered by lexicographic order. Therefore, the `@` join relation used above behaves exactly like the one on records.

Pattern matching can be used on variants either as a right-hand side (in which case every branch will be a right-hand side) or as an instruction (and every branch will be an instruction). Below is the typing rule, as well as the semantic rules for the right-hand side version of the pattern matching (the instruction version is similar).

$$\text{(RHS\_MATCH)}\ \frac{\begin{array}{c}\{var : [cons_1 : ty_1|\ldots|cons_k : ty_k]\}@rty = rty' \\ \{var_1 : ty_1\}@rty = rty_1 \ldots \{var_k : ty_k\}@rty = rty_k \\ G \vdash_{rhs} rhs_1 : rty_1 \Rightarrow ty \ldots G \vdash_{rhs} rhs_k : rty_k \Rightarrow ty\end{array}}{G \vdash_{rhs} match\ var\ with\ cons_1\ var_1 \rightarrow rhs_1|\ldots|cons_k\ var_k \rightarrow rhs_k\ end : rty' \Rightarrow (constructor\ val : vty)}$$

$$\text{(RHS\_MATCH\_FOUND)}\ \frac{E \vdash_{rhs} rhs/\{var' = val'\} \rightarrow val}{E \vdash_{rhs} match\ var\ with\ cons\ var' \rightarrow rhs| < branches\_rhs > end/\{var = (cons\ val' : vty)\} \rightarrow val}$$

$$\text{(RHS\_MATCH\_NEXT)}\ \frac{\begin{array}{c}cons \neq cons' \\ E \vdash_{rhs} match\ var\ with\ branches_rhs\ end/\{var = (cons\ val' : vty)\} \rightarrow val'\end{array}}{E \vdash_{rhs} match\ var\ with\ cons'\ var' \rightarrow rhs| < branches\_rhs > end/\{var = (cons\ val' : vty)\} \rightarrow val}$$

While these rules seem pretty verbose, they are quite simple to explain: every branch should have the same type (in an environment where the pattern's `var` has the type related to the constructor). Evaluation of a match simply consists in evaluating the correct right-hand side (by typing, all the branches must be specified, so the match always succeed). By typing, we also see that every-branch of the pattern-matching must consume the exact same variables from the stack. This is a severe restriction which could be difficult to deal with while programming in Albert, but is necessary in order for the compiled Michelson program to be correctly typed. The right-hand side match is particularly constrained, since the program can't even `drop` unnecessary variables in a branch.

Lists

As in Michelson, Albert implements linked lists. The language first gives the user a few operations to construct list, either in extension or by `cons` operations.

$$\text{(RHS\_LIST)}\ \frac{}{E \vdash_{rhs} [var_1; ..; var_n]/\{var_1 = val_1; ..; var_n = val_n\} -> [val_1; ..; val_n]}$$

$$\text{(RHS\_CONS)}\ \frac{}{x_1 :: x_2/\{x_1 = val_1; x_2 = lval_2\} -> val_1 :: lval_2}$$

More interestingly, programmers can also pattern-match on an empty or non-empty list (to mimic the IF_CONS instruction of Michelson). As in the previously-described variant pattern-matching, list pattern-matching exists both as a right-hand side and as an instruction (we show below the instruction version of the semantic rules, for a change).

$$\text{(INS\_LIST\_MATCH\_NIL)}\ \frac{\{var = []\}@rval = rval' \qquad E \vdash_{ins} I_1/rval \rightarrow val}{E \vdash_{ins} match\ var\ with[] \rightarrow I_1|var_1 :: var_2 \rightarrow I_2\ end/rval' \rightarrow val}$$

$$\text{(INS\_LIST\_MATCH\_CONS)}\ \frac{\{var_1 = val_1 :: tl\}@rval = rval' \qquad \{var_1 = val_1; var_2 = tl\}@rval = rval''}{E \vdash_{ins} match\ var\ with[] \rightarrow I_1|var_1 :: var_2 -> I_2\ end/rval' \rightarrow val}$$

More specific to lists (and to maps, as we'll see) are the map (as a right-hand-side) and for (as an instruction) operations, which mirror the ITER instruction. We'll show below the map semantic rules.

$$\text{(RHS\_LIST\_MAP\_NIL)}\ \frac{}{E \vdash_{rhs} map\ var\ in\ var'\ do\ rhs\ done/\{var' = []\} \rightarrow []}$$

$$\text{(RHS\_LIST\_MAP\_CONS)}\ \frac{\begin{array}{c}E \vdash_{rhs} rhs/\{var = val\} \rightarrow val' \\ E \vdash_{rhs} map\ var\ in\ var'\ do\ rhs\ done/\{var' = tl\} \rightarrow tl'\end{array}}{E \vdash_{rhs} map\ var\ in\ var'\ do\ rhs\ done/\{var' = val :: tl\} \rightarrow val' :: tl'}$$

Maps

Albert also implements associative maps, like Michelson does. These maps implement classic `get` and `update` operations (both as right-hand sides), where `get` returns takes a key and returns an option, and `update` takes both a key and an option (to allow to remove a value from the map). The semantics for the `get` operation are outlined below.

$$\text{(GET\_NIL)}\ \frac{}{E \vdash_{rhs} var[x]/\{var = \{\}; x = val\} \rightarrow None\{\}}$$

$$\text{(GET\_FOUND)}\ \frac{}{E \vdash_{rhs} var[x]/\{var = \{val \mapsto val'; mval\}; x = val\} \rightarrow Some\ val'}$$

$$\text{(GET\_NEXT)}\ \frac{val_1 \neq val_2 \qquad E \vdash_{rhs} var[x]/\{var = mval; x = val_2\} \rightarrow val_2'}{E \vdash_{rhs} var[x]/\{var = val_1 |- > val_1'; mval; x = val_2\} \rightarrow val_2'}$$

And the semantics for the `update` operation are the following.

$$\text{(UPDATE\_NIL\_NONE)}\ \frac{}{E \vdash_{rhs} \{var\ with\ x \mapsto y\}/\{var = \{\}; x = v_1; y = None\{\}\} \rightarrow \{\}}$$

$$\text{(UPDATE\_NIL\_SOME)}\ \frac{}{E \vdash_{rhs} \{var\ with\ x \mapsto y\}/\{var = \{\}; x = v_1; y = Some\ v_2\} \rightarrow \{v_1 \mapsto v_2\}}$$

$$\text{(UPDATE\_CONS\_NONE)}\ \frac{}{E \vdash_{rhs} \{var\ with\ x \mapsto y\}/\{var = \{v_1 \mapsto v_3; mval\}; x = v_1; y = None\{\}\} \rightarrow mval}$$

$$\text{(UPDATE\_CONS\_SOME)}\ \frac{}{E \vdash_{rhs} \{var\ with\ x \mapsto y\}/\{var = \{v_1 \mapsto v_3; mval\}; x = v_1; y = Some\ v_2\} \rightarrow \{v_1 \mapsto v_2; mval\}}$$

$$\text{(UPDATE\_NEXT)}\ \frac{v_1 \neq v_1' \qquad E \vdash_{rhs} \{var\ with\ x \mapsto y\}/\{var = mval; x = v_1; y = v_2\} \rightarrow mval'}{E \vdash_{rhs} \{var\ with\ x \mapsto y\}/\{var = \{v_1' \mapsto v_3; mval\}; x = v_1; y = v_2\} \rightarrow \{v_1' \mapsto v_3; mval'\}}$$

Moreover, like the list, maps can be iterated (and mapped) upon, in which case both the key and the value can be used in the loop body. Below we show the semantic rules for the for loop instruction on a map.

$$\text{(INS\_MAP\_FOR\_NIL)}\ \frac{\{var' = \{\}\}@rval = rval'}{E \vdash_{ins} for\ var \mapsto var'\ in\ var'\ do\ I_1\ done/rval' \rightarrow rval}$$

$$\text{(INS\_MAP\_FOR\_CONS)}\ \frac{\begin{array}{c}\{var = val; var' = val'\}@rval = rval' \\ E \vdash_{ins} I_1/rval' \rightarrow rval1 \qquad \{var'' = mval\}@rval1 = rval'' \\ E \vdash_{ins} for\ var \mapsto var'\ in\ var''\ do\ I_1\ done/rval'' \rightarrow rval2\end{array}}{E \vdash_{ins} for\ var \mapsto var'\ in\ var''\ do\ I_1\ done/\{var'' = \{val \mapsto val'; < mval >\}\} \rightarrow rval2}$$

Lastly, the language implements sets, which are simply maps where the key is the set element, and the value is simply unit.

2.3 Compiler to Mi-Cho-Coq

With all these rules written down (plus the typing rules), we now have to write a compiler answering to this formal specification, that is a compiler that, from a well-typed Albert program, compiles to a well typed Michelson program, with the type and semantics of the Michelson program being equivalent to the those of the Michelson's program. In order to specify and prove these properties, we'll first have

to define a relation between Albert's type and Michelson's type, as well as a relation between Albert's values and Michelson's values (as we'll see in 2.3.2).

In order to be able to use a formal specification of Michelson (which is already written in Coq), we'll compile to Mi-Cho-Coq's AST. More specifically, we'll target Mi-Cho-Coq's untyped syntax, which can then be turned back into the typed syntax using Mi-Cho-Coq's typer (as long as the untyped program respects Michelson's typing rules of course). This is simpler than compiling to the typed syntax, but means we'll also have to prove the compiled Michelson program is correctly typed separately.

2.3.1 Architecture of the program and parser

As we want to specify and prove the correctness of our compiler, we decided to implement it in Coq. This allows us to easily take advantage of Ott's definition, as well as to easily compile to Mi-Cho-Coq's AST. Moreover, using Coq's extraction facilities, we'll be able to transpile our compiler to OCaml code, which will be easier to run. The compiler follows a classic frontend/middle-end/backend architecture.

Frontend Albert's parser and lexer are the only part not implemented in Coq. Indeed, we use Ott functionalities (improved with the modifications outlined in 1.2.2) to generate both a Menhir parser as well as an Ocamllex lexer for the language. As Ott does not yet support generating Menhir-Coq parser, we've decided to have the generated parser directly in OCaml. We therefore must call the parser's API from the OCaml extracted code (or rather, from a "main" function written by hand to tie all the code together) instead of directly from Coq code. Although this does work well, we still need to adapt the code of the parser a little to fix discrepancies between the OCaml code generated by Menhir and the OCaml code extracted from Coq.

The main difference lies in the name of constructors for the AST generated by Ott : while Coq's extraction adds a `Coq_` prefix in front of the constructor, Ott does not when producing the *parser.mly* file used by Menhir to generate the parser. Our (admittedly dirty) fix is to rewrite part of the *parser.mly* file with appropriate regular expressions. While this may seems like a bad idea for certified compiler, we have deemed reasonable not to certify the parser for Albert (as more high-level language compiling to Albert will directly compile to the its AST anyway, and therefore sidestep any weakness in the parser).

Middle-end In order to compile Albert's AST to Mi-Cho-Coq's AST, a first necessary phase is to type-check the Albert program, in order both to make sure the Michelson program will also be well-typed, and to infer the explicit types needed to convert Albert's values to Michelson's values (see 2.3.2). We therefore implemented the typing-rules of Albert, described in Ott as relations, in functional form.

The type checker processes in three separate passes : the first one consists in getting rid of type aliases declared by the user, and replacing them by their actual declaration. This simplify the next phases, as we don't have to worry about declared types when verifying type-equivalence during the type-checking proper. As declared types are simple aliases (types can't be recursively declared), these phases amount to (recursive) inlining of the type aliases wherever they're found in the program. This phase can fail (and therefore returns in the error monad) if an undeclared type alias is found in the program.

The second phase, is equally simple, and cannot even fail : it consists in sorting by lexicographic order both the fields of records and the constructors of variants in type declaration. This pass allows the user to write the fields of records in any order, while guarantying that the type-checker doesn't have to verify that two lists are permutations of each other. The sorting proper uses the Merge Sort algorithm, as implemented in Coq's **Sorting** library, and is therefore fairly efficient (which could end up mattering if higher-level compiler targeting Albert end up producing large records or variants to represent more complex data-structures or types).

The third, most complex and most important phase is of course the type-checking. Currently, the type-checking is entirely "unidirectional", as it starts at the top of the program, and simply checks program from top to bottom. This means that the type-checker is actually pretty stupid, as it does not perform any type inference. Since our type-system is monomorphic, this is not too much of a limitation, although we had to add some explicit type annotations in order for the type-checker to fully cover all cases (for instance, an empty list type cannot be inferred, so it needs to be annotated).

The type-checker follows the linear typing of Albert, and tracks the state of the record representing the stack throughout the program. Equivalence between two types (for instance, `bool` and its equivalent `variant`) is decided using a function `type_eq`, which follows the type equivalence defined in Ott (this

equivalence has only been partially proven). The type-checker returns a typed version of the AST, that is an AST where the instructions input and output type have been annotated (only the instructions are annotated as this is enough information for the compiler). Of course, the type-checker can fail if the program is ill-typed, in which case an error message is returned instead of the typed version of the AST. (admittedly, the current error messages could be improved).

The type-checker is supposed to be correct and complete relative to the typing-rules given in Ott, but I haven't had time to prove these properties (aside from some helpful auxiliary lemmas). I did however extensively test the type-checker, and I'm strongly confident it is correct.

Backend Once the program has been successfully type-checked, we move on to Michelson code generation. As we've specified, our compiler targets Michelson's untyped syntax, which will take care of the pretty-printing of the program; therefore, we only have to worry about producing a Mi-Cho-Coq untyped AST. As with the type-checker, the compiler returns its result in an error monad, although the compiler should not fail if the program is well typed (this is a property to be proven). We also want to have other properties of the compiler : namely, that it preserves typing (relative to the Albert-Michelson type equivalence described in 2.3.2) and semantics. As of now, the first property has only been partially proven, and I haven't had time to tackle the second one.

We'll focus in the following sections on the specifics of compiling an Albert program to Michelson, instruction by instruction.

2.3.2 Equivalence between Albert and Michelson

In order to compile an Albert program to a Michelson program, we'll need to be able to convert Albert's types to Michelson's types. Moreover, in order to be able to cover the `rhs_val` right-hand side to a PUSH `t v` instruction, where `v` is a Michelson value, we'll also need to be able to convert Albert's value literals to Michelson values. Fortunately, it seems obvious that if types are translatable (and they are, since Albert was specifically designed for them to be), values are too.

Primitives The basic types of Albert are pretty easy to translate to Michelson's types, as they have been specifically designed to. The numerical types `nat`, `int`, `timestamp` and `mutez` translate to the same types in Michelson. That is also the case for the `string` and `bytes` types. The basic collections types (`list`, `map` and `set`) are also translated to their counterparts in Michelson.

Records More interesting is the implementation of record types. As records are not present in Michelson, we must encode Albert's records in Michelson's type systems. Their are two uses of records in Albert. The simplest one is to generalize product types. In this case, we'll want to simply represent records by nested products. In order to know the order of the fields in our nested product, we'll reuse our lexicographic order to organize the fields. We also decide that the left element of the pair is always a field, and that the right element contains the rest of the elements of the record. For instance, the record type `{a : nat ; b : int ; c : string ; d : mutez}` translates to `(pair nat (pair int (pair string mutez)))`. An empty record translates to a `Unit` (this is coherent with the typing rules of Albert). In order to avoid using one more pair that needed, a singleton record also translates to a simple value. Using these conventions, Albert's pairs (which are equivalent to binary records) simply translate to Michelson's pair, which seems ideal.

The second use of records is to abstract Michelson's stack. In this case, there is no need to encode the record in any particular data structure, since it is used to represent the state of the stack. However, it's important to consider the actual order of values on the stack. Our first solution was to always push values on top of the stack, to keep track of their position, and then to retrieve them when needed using the DIG instruction (DIG `n` moves the n^{th} element of the stack to its top). However, while this approach worked fine with linear programs, when we started to manipulate branching instructions, we figured that this naive approach lost the guarantee of the two end-of-branch stacks being equal, while their Albert type were, as the order could possibly not be the same. In order to avoid this issue, we decided to always have the stack in the same order as the record type representing it between two Albert instructions. This requires to often DUG values into the stack (DUG `n` moves the first element of the stack to its n^{th} position, and is the opposite of DIG `n`), which creates an added cost. We already optimize the produced code by removing the sequences of DUG `n;` DIG `n`, as well as DIG `0` and DUG `0` (which both do nothing). Future optimisations could improve on these simple optimisations by eliminating some of the other useless DIG and DUG.

Variants As with records, we need a trick to encode variants in Michelson, since it does not contain them. We'll proceed similarly to our record technique, by encoding a variant in nested `or` types. Again, we'll use lexicographic order to sort the constructor of the variant, in order to predictably construct or pattern match on the variant. The type `[a : nat | b : int | c : string | d : mutez]` will then be translated to `(or nat (or int (or string mutez)))`. As we saw above, we don't allow for empty variants, as we'd have no way to encode an empty type in Michelson. A singleton variant is translated to the value without any constructor.

As was the case with records, the Albert `or` type is now equivalent to Michelson's (since `Left` comes in lexicographic order before `Right`). We then add special cases in the translater for the `bool` and `option` variants, which must simply be translated to their counterparts.

2.3.3 Compiling instructions

Now that we know how to translate Albert's types and values into Michelson's, its time to move on to the compilation of a program. Later on, we'll use the function `michType` to denote the translation from Albert's type to Michelson's type, and `michData` to denote the translation from Albert's values to Michelson's data.

Right-hand sides

We'll begin with the right-hand sides, which, has we've seen, contain most of the calculations present in the language. Compiling a right-hand side always put the resulting value on top of the stack.

First off, arguments are usually pretty easy to compile, as they consist mostly in either values, variable calls or record construction (the latter being more complex).

Values are also easy to compile by in pushing Michelson's data onto the stack. However, we have to take into account a small specific case : some types can't actually be used with PUSH in Michelson, as they are not storable. Currently, the main type concerned by this limitation is `operation`; since we need to build a `list operation` at the end of our program, in order for it to have a correct Michelson's contract return type, we need to add a specific case to take care of that issue. We simply decide to produce `NIL operation` in that case, instead of a PUSH.

Albert code	Michelson code
value (value $\neq$ `([] : list operation)`)	PUSH `michType(type(value)) michData(value)`
`([] : list operation)`	`NIL operation`

Retrieving a variable from the stack is about as simple, since the only thing to do is to DIG it back from its position, which we can predict given the type of the stack thanks to the lexicographic order on records. The record construction rule is a bit more complicated, as it generalizes the var rule, and also requires us to build the record as we `DIG` the variables from the stack. We'll note in the followings `var::stack` the operation of adding `var` in front of `stack`, `stack#var` the index of `var` in `stack`, and `stack/var`, the `stack` where the `var` has been removed. We'll transparently also use the name of the variable to designate its value on the stack.

Albert code	Michelson code	Resulting stack
var	DIG (stack#var)	$var :: (stack/var)$
$\{ x_1 = var_1; \ldots; x_k = var_k \}$	DIG $(stack\#var_k)$; DIG $((\_ :: (stack/var_k))\#var_{k-1})$; PAIR ; ... ; DIG $((\_ :: (stack/var_k/\ldots/var_2))\#var_1)$; PAIR	$(var_1, (\ldots, var_k)) :: (stack/var_1/\ldots/var_k)$

As well as simple argument retrieval, right-hand sides also include function calls on arguments, which are quite simple to compile. We simply compile the arguments, which results in the argument value being at the top of the stack (functions in Albert are unary, as we'll see later). We can then compile the function completely independently of the compiled argument. The resulting Michelson instruction will be one that pops a single elements from the top of the stack, and pushes its result as a single element to replace it. As we'll see below in 2.3.3, user-defined functions also respect this calling convention. We show below the compilation of a function call, as well as some of the functions present in Albert. We note $[|code|]_{stack \to stack'}$ the result of compiling *code*, which makes *stack* evolve to *stack'*.

Albert code	Michelson code	Stack evolution
f arg	$[\![\text{arg}]\!]_{stack \to stack'}$; $[\![\text{f}]\!]_{stack' \to stack''}$	$stack \to stack''$
dup	DUP ; PAIR	$d :: stack \to (d, d) :: stack$
contract `ty` address implicit_account	CONTRACT `michType`(ty) ADDRESS IMPLICIT_ACCOUNT	$a :: stack \to c :: stack$ $c :: stack \to a :: stack$ $k :: stack \to c :: stack$
$(c_i : [c_1 : t_1; \ldots c_i : t_i; \ldots; c_k : t_k])$	RIGHT `michType`(t_1) ; $\ldots$; LEFT `michType`$([c_{i+1} : t_{i+1}; \ldots; c_k : t_k])$	$v :: stack \to (Right(\ldots(Left\, v))) :: stack$

Right-hand sides also allow for some simple record manipulation, namely taking a specific field of a record (by projection) and updating a record. Projection is easy to compile by destructuring (using multiple CDR and a last CAR) the nested pairs and keeping only the correct field. Updating a record is a little more tricky, as it requires to destruct (using UNPAIR) and restructure the nested pairs, changing a subset of the fields with new values. In the compiled instruction below, we assume that *var* is of type $\{\ l_1 : ty_1\ ; \ldots ; l_k : ty_k\ \}$

Albert code	Michelson code
$var.l_i$	CDR ; $\ldots$; CAR
$\{$ var with $l_{i_1} = x_1\ ; \ldots ; l_{i_j} = x_j\ \}$	UNPAIR ; if the current label is to be updated : DROP ; DIG $(stack\#x_k)$ DIP `{ ... }` ; PAIR ;

Arithmetic operations and comparisons are very easy to translate, as we only have to dig the two operands from the stack, and apply the correct operator that is available in Michelson. We show below the code produced for the MUL arithmetic operator, as well as the LE comparison.

Albert code	Michelson code	Resulting stack
x_1 `*` x_2	DIG $(stack\#x_2)$; DIG $((x_2 : (stack/x_1))\#x_1)$; MUL	$(x_1 \times x_2) :: (stack/x_1/x_2)$
x_1 `<=` x_2	DIG $(stack\#x_2)$; DIG $((x_2 : (stack/x_1))\#x_1)$; COMPARE ; LE	$(x_1 \leq x_2) :: (stack/x_1/x_2)$

Left-hand sides

While right-hand sides allow for calculations, left-hand sides main use is to put the calculated values back at the right place in the stack. Indeed, as we've seen above, we cannot simply leave the calculated values on top of the stack, as we would risk having two stacks ordered differently at the end of a branching instruction, which Michelson does not allow. Therefore, we use the left hand side of the argument to know, using lexicographic order, the position of the value in the stack. The record destructuring left-hand sides generalize this behaviour, while UNPAIRing the related record. We'll extend our `stack#var` to also denote the index where `var` can be inserted into `stack` if it is not already present, keeping the lexicographic order.

Albert code	Michelson code
$var =$	DUG $stack\#var$
$\{\ x_1 = var_1 \ldots x_k = var_k\ \} =$	UNPAIR ; DUG $stack\#var_1$; UNPAIR $\ldots$

Control instructions

As we've seen, our language also contains some control instructions, allowing for branching and looping. These instruction can mostly be easily compile to Michelson code, as it includes similar control instruction in its core language.

As Albert uses variant to generalize `or`, `option` and `bool` types, we use pattern matching on the variant's constructors to generalize branching. In addition of the classic pattern matching, which can be used on both variants (using IF_LEFT), `bool` (using IF and `option` (using IF_SOME), and to pattern match on empty or non-empty `list` using IF_CONS. Branching exists both as right-hand sides (in which case they're mostly useful with user defined function calls, which is the only way to "insert" instructions into a right-hand side) and instructions, but we'll show the compilation of instructions, which are more interesting, below.

Albert code	Michelson code
match var with $c_1\ x_1 \rightarrow i_1$ | ... | $c_k\ x_k \rightarrow i_k$ end	DIG $stack\#var$; IF_LEFT { DUG $(stack/var)\#x_1$; $[\|i_1\|]$ } { IF_LEFT { ...} { ...} }
match var with [] $\rightarrow i_1$ | $hd::tl \rightarrow i_2$ end	DIG $stack\#var$; IF_CONS { DUG $(stack/var)\#hd$; DUG $(stack/var)\#tl$; $[\|i_2\|]$ } { $[\|i_1\|]$ }

Albert also gives access to the looping instruction present in Michelson (LOOP, LOOP_LEFT, ITER and MAP) via both instructions and right-hand sides (MAP in particular is of course used with the right-hand side map). We give below the two examples of map (which iterates on lists), and loop_left (which repeats while the variables contains the Left constructor of an or type), which are quite similar in how they compile, even if loop_left is used as an instruction and map as a right-hand side.

Albert code	Michelson code
map var in var' do rhs done	DIG $stack\#var$; MAP { DUG $(stack/var)\#var'$; $[\|rhs\|]$ }
loop_left var in var' do ins done	DIG $stack\#var$; LOOP_LEFT { DUG $(stack/var)\#var'$; $[\|ins\|]$ } ; DROP

The noticeable change between the two instructions is the addition of the DROP at the end of the loop_left compilation. This is necessary because Michelson actually puts the `Right` element extracted from the or at the end of the loop. Since we don't want to use it here, we DROP it from the stack.

User-defined functions

Finally, Albert gives users the possibility to split their code in multiple, non-recursive global functions. This possibility does not exist when all of the code is written in one block. The best way to compile user defined functions then becomes to inline their code.

Since a function is defined as an instruction, it makes sense that the input and output types of a function would be record types representing the stack manipulated during the function execution. However, as user defined functions are actually called on arguments (with the usual function call syntax), it is necessary to transform the values produced by these arguments (which must be record values, and are therefore represented in Michelson by nested pairs) into values corresponding to the stack which will be manipulated by the function. This, of course, means UNPAIRing the input value, and rePAIRing the output value at the end of the function. We simply add the code necessary to process this unpairing and re-pairing at the start and the end of the function code we'll inline.

We must also consider the case of the entry function, that is the function which defines the code of the Michelson compiled smart contract. A Michelson smart contract must start with its stack in the state `parameter_ty::storage_ty::nil`, and finish with the stack `(list operation)::storage_ty::nil`. This specific constraint means that our entry point function must have type `{ params : parameter_ty ; storage : storage_ty } → { operations : list operation ; storage : storage_ty}`. If the labels are in the correct (lexicographic) order, these types will be translated to the Michelson types `(parameter_ty * storage_ty) → (list operation * storage_ty)`, which, once unpaired by the inline call code, will give us the right Michelson stack types.

Conclusion

At the end of the day, the product of our work is a working prototype of the Albert Compiler, able to generate well-typed Michelson program. I have written and tested a few programs, including the voting smart contract described above, as well as a multisig [5] contract (see A.2 for the Albert version). I was able to originate these contracts on a Tezos sandboxed node, and to run them, yielding the expected results. As a proof-of-concept for the Albert programming language, this is a success. Moreover, the improvements brought to Ott (in particular regarding the parser generation), although they have not yet been merged into the core software, will still be usable by Nomadic Labs to continue working on both Albert and future, higher-level languages targeting it. Although I didn't focus on it, Ott could also very well be used to generate documentation in the ReStructuredText format for Albert.

Further works include, of course, formally proving the correctness of the both the type-checker and compiler, which I unfortunately didn't have time to complete. Moreover, writing further optimisations of the compiler would also be useful, as the Michelson code currently generated consists mostly of explicit stack manipulations, which could possibly be improved using a stack optimization technique, such as the JUMBLE [6] macro. These optimizations would of course also need to be proven correct (that is that they keep both the typing and the semantics of the program intact). Improving error messages both for type-checking and parsing would go a long way in making the compiler more usable.

On a personal note, these works allowed to discover many interesting concept related to blockchains in general and stack-based programming languages such as Michelson in particular. Moreover, I was able to improve and develop my skills in both programming and verifying programs using Coq, especially through writing a full compiler, which was a very formative exercise. This knowledge, as well as the taste I've acquired for certified programming, will certainly inform my future study path, beginning as soon as next year. Moreover, working in Nomadic Labs, a company dedicating most of its ressources to research and development, and working close to public research has also given me a new interesting point of view on research, a sector which I already intend to make my career in. I therefore plan to continue working in related fields of verified programming.

Appendix A

More Albert contracts

A.1 Auction

This contract, inspired by `https://www.michelson-lang.com/contract-a-day.html#sec-1-11`, implements, as its name indicate, a simple auction process. An (out of chain) item is placed in auction, and any user can place a bid during a set time period. The bid is accepted only if higher than the previous one, and of course, the outbidden is refunded their previous bid at an address they provided when bidding.

```
type parameter_ty = key_hash
type storage_ty = { endDate : timestamp ; highestBid : mutez ; lastBidder : key_hash }

def bid : { params : parameter_ty ; storage : storage_ty } →
          { op : list operation ; storage : storage_ty } =
    // Check if the auction has ended
    { car = sto0 ; cdr = storage } = dup storage;
    endDate = sto0.endDate;
    no = now; ok = no < endDate;
    match ok with
    True u →drop u
    | False u →failwith "The␣auction␣is␣over"
    end;

    // Check if the new bid is higher than best
    { car = sto0 ; cdr = storage } = dup storage;
    highest = sto0.highestBid;
    am = amount; ok = am > highest;
    match ok with
    True u →drop u
    | False u →failwith "Not␣enough␣tez␣sent␣to␣replace␣highest␣bidder"
    end;

    // Refund the previous bidder
    { car = sto0 ; cdr = storage } = dup storage;
    { endDate = endDate ; highestBid = highest ; lastBidder = last } = sto0;
    drop endDate;
    un = {};
    acc = implicit_account last;
    transfer = transfer_tokens un highest acc;
    op = ([] : list operation);
    op = transfer :: op;

    // Save the new storage
    am = amount;
    storage = { storage with lastBidder = params ; highestBid = am }
```

A.2 The multisig contract

The multisig (multi-signature) smart-contract allows for a way of access control : the contract is originated with a list of authorized public keys, and a threshold. To issue a command to the multisig, the user must send more signature than the threshold specifies. If the user does not send enough signature, or if at least one of the signature is incorrect, the execution stops. Moreover, the contract also uses a counter that increases at each successful execution to avoid replay attacks (where an opponent would intercept the transaction and send it a second time). While multisigs can be written with intricate payloads in mind (up to and including starting the execution of another contract) the one below is quite simple : either it is asked to transfer a sum from it's balance to another contract, to change it's delegate or to change it's access-control parameters (set of key and threshold).

```
type parameter_ty = { payload :
                      { action : [ ChangeKeys : { keys : list key ; threshold : nat }
                                 | SetDelegate : option key_hash
                                 | Transfer : { am : mutez ; destination : contract unit }] ;
                        counter : nat } ;
                      sigs : list (option signature) }

type storage_ty = { keys : list key ; stored_counter : nat ; threshold : nat }

def multisig : { params : parameter_ty ; storage : storage_ty } →
               { ops : list operation ; storage : storage_ty } =
    // Pack the payload
    { car = storage0 ; cdr = storage } = dup storage;
    { car = params0; cdr = params } = dup params;
    { payload = payload ; sigs = sigs } = params0;
    se = self; add = address se;
    payload = { add = add ; payload = payload };
    packedPayload = pack payload;

    // Check the anti-replay counter
    { car = params0 ; cdr = params } = dup params;
    { car = storage0 ; cdr = storage } = dup storage;
    payload = params0.payload; counter = payload.counter;
    stored_counter = storage0.stored_counter;
    ok = counter == stored_counter;
    match ok with
    False f →failwith "Incorrect␣counter"
    | True t →drop t
    end;

    // Compute the number of valid signatures
    valid = 0;
    { keys = keys ; stored_counter = c ; threshold = threshold } = storage0;
    drop c;
    for k in keys do
        match sigs with
        [] →failwith "Not␣enough␣sigs␣sent"
        | s::sigs →
          match s with
          None u →drop u; drop k
          | Some s →
            { car = packedPayload0 ; cdr = packedPayload } = dup packedPayload;
            ok = check_signature k s packedPayload0;
            match ok with
            False u →failwith "Invalid␣signature"
            | True u →drop u; one = 1; valid = valid + one
            end
```

```
      end
    end
done; drop sigs;

// Check that the number of signatures is sufficient
ok = valid >= threshold;
match ok with
False u →failwith "Not␣enough␣signatures"
| True u →drop u
end;
drop packedPayload;

// Increment the counter
{ car = storage1 ; cdr = storage2 } = dup storage;
counter = storage2.stored_counter;
one = 1; counter = counter + one;
storage = { storage1 with stored_counter = counter };

// Execute the payload
ops = ([] : list operation);
payload = params.payload; action = payload.action;
match action with
Transfer t →
  { am = am ; destination = dest } = t;
  u = {};
  transf = transfer_tokens u am dest;
  ops = transf::ops
| SetDelegate k →
  setd = set_delegate k;
  ops = setd::ops
| ChangeKeys ck →
  { keys = keys ; threshold = thresh } = ck;
  storage = { storage with keys = keys ; threshold = thresh }
end
```

Bibliography

[1] L.M Goodman. Tezos: A self-amending crypto-ledger. white paper. https://tezos.com/static/white_paper-2dc8c02267a8fb86bd67a108199441bf.pdf, September 2014.

[2] Michelson: the language of Smart Contracts in Tezos. http://tezos.gitlab.io/mainnet/whitedoc/michelson.html.

[3] Peter Sewell, Francesco Zappa Nardelli, and Scott Owens. Ott. https://github.com/ott-lang/ott.

[4] Bruno Bernardo, Raphaël Cauderlier, and Julien Tesson. A formal specification of the Albert language (proposal), 2019.

[5] Arthur Breitman. Multisig contract in Michelson. https://github.com/murbard/smart-contracts/blob/master/multisig/michelson/multisig.tz.

[6] Arthur Breitman. Optimizing stack manipulation in michelson. https://hackernoon.com/optimizing-stack-manipulation-in-michelson-31ba7ff11a3a.