Albert, an intermediate smart-contract language for the Tezos blockchain

Bruno Bernardo, Raphaël Cauderlier, Basile Pesin, and Julien Tesson

Nomadic Labs, Paris, France
{first_name.last_name}@nomadic-labs.com

Abstract. Tezos is a smart-contract blockchain. Tezos smart contracts are written in a low-level stack-based language called Michelson. In this article we present Albert, an intermediate language for Tezos smart contracts which abstracts Michelson stacks as linearly typed records. We also describe its compiler to Michelson, written in Coq, that targets Mi-Cho-Coq, a formal specification of Michelson implemented in Coq.

Keywords: Certified programming · Certified compilation · Programming languages · Linear types · Blockchains · Smart contracts.

1 Introduction

Tezos is an account-based public blockchain and smart-contract platform. It was launched in June 2018 and an open-source implementation is available [3]. The Tezos blockchain distinguishes itself through its on-chain amendment procedure by which a super-majority of stakeholders can modify a large part of the codebase, through its liquid Proof-of-Stake consensus algorithm [2], and through its focus on formal methods which is especially visible in the design and implementation of Michelson, its smart-contract language.

Indeed, the Michelson interpreter is implemented using a GADT that statically ensures the subject reduction property. Moreover, Michelson is formally specified in the Coq proof assistant. This Coq specification is called Mi-Cho-Coq [15] and its main application today is the certification of Michelson smart contracts by deductive verification [7].

However, the stack paradigm used by Michelson is too low-level for complex applications. For this reason, several high-level languages have been developed [5,14,6,9,12,13]. Unfortunately, their compilers to Michelson are not formally verified which limits the application of formal methods for these languages.

In this article, we propose an intermediate language named Albert to avoid the duplication of effort put into compilers to Michelson and to ease the certification of these compilers. The main feature of Albert is that the Michelson stack is abstracted through named variables. The duplication and destruction of resources are however explicit operations in both Albert and Michelson, this is reflected in Albert by the use of a linear type system.

We have formally specified the Albert language in the Ott tool [18] from which the Albert lexer, parser, and LaTeX documentation are generated. Ott can

also generate typing and semantic rules for Coq and other proof assistants. We have written the Albert compiler in Coq as a function from the generated Coq output for the Albert grammar to the Michelson syntax defined in Mi-Cho-Coq.

This article is organised as follows: Section 2 gives an overview of the Michelson smart-contract language. Section 3 presents the Albert intermediate language, the figures of this section have been produced by the LaTeX output of Ott. The Albert compiler is then presented in Section 4. Section 5 discusses some related work and finally Section 6 concludes the article by listing directions for future work.

The Albert specification and compiler are available at `https://gitlab.com/nomadic-labs/albert/tree/WTSC20`.

2 Overview of Michelson

Smart contracts are Tezos accounts of a particular kind. They have private access to a memory space on the chain called the *storage* of the smart contract, each transaction to a smart contract account contains some data, the *parameter* of the transaction, and a *script* is run at each transaction to decide if the transaction is valid, update the smart contract storage, and possibly emit new operations on the Tezos blockchain.

Michelson is the language in which the smart contract scripts are written. The most important parts of the implementation of Michelson, the typechecker and the interpreter, belong to the economic ruleset of Tezos which evolves through the Tezos on-chain amendment voting process.

2.1 Design rationale

Smart contracts operate in a very constrained context: they need to be expressive, evaluated efficiently, and their resource consumption should be accurately measured in order to stop the execution of programs that would be too greedy, as their execution time impacts the block construction and propagation. Smart contracts are non-updatable programs that can handle valuable assets, there is thus a need for strong guarantees on the correctness of these programs.

The need for efficiency and more importantly for accurate account of resource consumption leans toward a low-level interpreted language, while the need for contract correctness leans toward a high-level, easily auditable, easily formalisable language, with strong static guarantees.

To satisfy these constraints, Michelson was made a Turing-complete, low-level, stack based interpreted language (*à la* Forth), facilitating the measurement of computation costs, but with some high-level features *à la* ML: polymorphic products, options, sums, lists, sets and maps data-structures with collection iterators, cryptographic primitives and anonymous functions. Contracts are pure functions that take a stack as input and return a stack as output. This side-effect free design is an asset for the conception of verification tools.

The language is statically typed to ensure the well-formedness of the stack at any point of the program. This means that if a program is well typed, and if it is being given a well-typed stack that matches its input expectation, then at any point of the program execution, the given instruction can be evaluated on the current stack.

Moreover, to ease the formalisation of Michelson, ambiguous or hidden behaviours have been avoided. In particular, unbounded integers are used to avoid arithmetic overflows and division returns an option (which is `None` if and only if the divisor is 0) so that the Michelson programmer has to specify the behaviour of the program in case of division by 0; she can however still *explicitly* reject the transaction using the **FAILWITH** Michelson instruction.

2.2 Quick tour of the language

The full language syntax, type system, and semantics are documented in [1], we give here a quick and partial overview of the language.

Contracts' shape A Michelson smart contract script is written in three parts: the parameter type, the storage type, and the code of the contract. A contract's code consists of one block that can only be called with one parameter, but multiple entry points can be encoded by branching on a nesting of sum types and multiple parameters can be paired into one.

When the contract is deployed (or *originated* in Tezos lingo) on the chain, it is bundled with a data storage which can then only be changed by a contract's successful execution. The parameter and the storage associated to the contract are paired and passed to the contract's code at each execution. The execution of the code must return a list of operations and the updated storage.

Seen from the outside, the type of the contract is the type of its parameter, as it is the only way to interact with it.

Michelson Instructions As usual in stack-based languages, Michelson instructions take their parameters on the stack. All Michelson instructions are typed as a function going from the expected state of the stack, before the instruction evaluation, to the resulting stack. For example, the **AMOUNT** instruction used to obtain the amount in *µtez* (*i.e.* a millionth of a *tez*, the smallest token unit in Tezos) of the current transaction has type `'S → mutez:'S` meaning that for any stack type `'S`, it produces a stack of type `mutez:'S`. Michelson uses an ordered type system which means that the number of times values are used and the order in which they are introduced and consumed matter and are visible at the type level. Some operations such as **SWAP** `:: 'a:'b:'S → 'b:'a:'S`, **DUP** `:: 'a:'S → 'a:'a:'S`, and **DROP** `:: 'a:'S → 'S` have to be used to respectively change the order of the values on the Michelson stack, to duplicate a value, and to pop a value from the stack without actually using it. Some instructions, like comparison or arithmetic operations, exhibit non-ambiguous

ad-hoc polymorphism: depending on the input arguments' type, a specific implementation of the instruction is selected, and the return type is fixed. For example **SIZE** has the following types:

```
bytes:'S → nat:'S                set 'elt:'S → nat:'S
string:'S → nat:'S               map 'key 'val:'S → nat:'S
                                 list 'elt:'S → nat:'S
```

While computing the size of a string or an array of bytes is similarly implemented, under the hood, the computation of map size has nothing to do with the computation of string size.

Finally, the contract's code is required to take a stack with a pair *parameter-storage* and returns a stack with a pair *operation list-storage*: `(parameter_ty*storage_ty):[] → (operation list*storage_ty):[]`.

The operations listed at the end of the execution can change the delegate of the contract, originate new contracts, or transfer tokens to other addresses. They will be executed right after the execution of the contract. The transfers can have parameters and trigger the execution of other smart contracts: this is the only way to perform *inter-contract* calls.

3 The Albert intermediate language

Michelson, as a stack-based language, is a difficult and unusual target for compiler writers. In addition to the usual effort to translate high-level constructions to lower-level types and control-flow, they have to deal with stack manipulation to make values available at the right stack position when calling an Michelson opcode, and to cope with the consumption of values by the opcode execution.

These additional difficulties also hinder the effort of teams developing static analysers and verification frameworks.

As a first simplification step, we have decided to build an intermediate language that abstracts away the ordering of values in the stack and provides a named binding to values. This intermediate language still keeps track of the resources as variables are typed by a linear type system, which enforces each value to be consumed exactly once. When a value is needed more than once, it must be explicitly duplicated with a `dup` operation. Generation of `dup`s is left to a future higher-level intermediate language.

In the process of defining the language, we thought that it would also be helpful to define some abstractions over the datatypes so we provide support for *records* which compile to nestings of Michelson's binary product type `pair` and *variants* which compile to nestings of Michelson's binary sum type `or`.

We also offer to define separate non-recursive function definitions used to define programming libraries. These functions are inlined at compile time.

3.1 Base language

The Albert language is defined as a collection of small language fragments that can be studied independently. Each fragment is defined in a separate Ott file.

The first fragment to consider is called the *base* fragment. As its name suggests, this fragment is the basis on top of which the other fragments are defined.

The base fragment contains the two main features of Albert: the stack is abstracted by named variables and Michelson binary pairs are generalized as records. We use the metavariable l to denote record labels and the metavariable x to denote variables but these two notions are unified in Albert.

Records and linear typing As we have seen it Section 2.2, Michelson uses an ordered type system that tracks both the order of the values on the stack and the number of uses of the values. Most high-level languages however bind values to named variables and implicitly handle the ordering and number of uses of variables. The required stack manipulation instructions are introduced at compile time. Albert is an intermediate language between these two extremes. In Albert, the order of values is abstracted but not the number of uses which is still explicitly handled.

This choice is reflected in Albert's type system by the use of linear typing. Each expression of the Albert language is typed by a pair of record types whose labels are the variables touched by the instruction or expression; the first record type describes the consumed values and the second record type describes the produced values.

Thanks to the unification of variable names and record labels, records in Albert generalize both the Michelson stack types and the Michelson pair type. In the base fragment of Albert, all types are possibly-empty record types.

The grammar of types of the base fragment given in Figure 1.

$$
\begin{array}{lcll}
\textit{label},\ l & ::= & \text{Label / variable} & \\
& \mid & \textit{id} & \\
\end{array}
\qquad
\begin{array}{lcll}
\textit{ty} & ::= & \text{Type} & \\
& \mid & \textit{rty} & \text{Record type} \\
\end{array}
$$

$$
\begin{array}{lcll}
\textit{rty} & ::= & & \text{Record type} \\
& \mid & \{l_1 : \textit{ty}_1 ; \ .. \ ; l_n : \textit{ty}_n\} & \\
\end{array}
$$

Fig. 1: Syntax of the record types

In the record type $\{l_1 : \textit{ty}_1 ; \ .. \ ; l_n : \textit{ty}_n\}$, we assume the labels to be distinct and lexicographically ordered.

This constraint is formalized by the well-formedness judgement $\Gamma \vdash \textit{ty}$ defined in Figure 2. The typing context Γ is always empty here but other cases for typing contexts will be added in other language fragments.

The grammar for the base fragment is defined in Figure 3. Albert's grammar is more stratified than Michelson's grammar because we adopt from imperative languages the usual distinction between expressions and instructions. An instruction is either the `noop` instruction that does nothing, a sequence of instructions separated by semicolons, or an assignment *lhs*`=`*rhs* where the left-hand side *lhs* is either a variable or a record of variables and the right-hand side is an expression.

$\boxed{\Gamma \vdash ty}$ Type well-formedness

$$\frac{\begin{array}{l} l_1 < .. < l_n \\ \Gamma \vdash ty_1 \quad .. \quad \Gamma \vdash ty_n \\ \Gamma \vdash \end{array}}{\Gamma \vdash \{l_1 \ : \ ty_1 \, ; \ .. \, ; \ l_n \ : \ ty_n\}}$$

Fig. 2: Type well-formedness judgment

Contrary to usual imperative expressions, arbitrary nesting of expressions is not allowed and intermediate values should be named. This restriction, inspired by the static single assignment form commonly used in intermediate compilation languages, is designed to ease the production of Michelson code and to allow for more optimisations at the level of the Albert language in the future. In practice, this restriction means that an expression is either a variable x, a value val, the application of a user-defined function to a variable $f\,x$, a record projection $x.l$, or a record update $\{var\ \texttt{with}\ l_1{=}var_1\,;\ \ldots\,;\ l_n{=}var_n\}$.

instruction, I, *ins*	::=		Instruction
	|	`noop`	No operation
	|	$instruction_1\,;\ instruction_2$	Sequencing
	|	$lhs{=}rhs$	Assignment
	|	`drop` var	Resource dropping
lhs	::=		Left-hand side of assignement
	|	var	
	|	$\{l_1{=}var_1\,;\ ..\,;\ l_n{=}var_n\}$	
rhs	::=		Right-hand side of assignments
	|	arg	
	|	$f\ arg$	
	|	$var.l$	
	|	$\{var\ \texttt{with}\ l_1{=}var_1\,;\ \ldots\,;\ l_n{=}var_n\}$	

f	::=	Function symbol	arg	::=	Fun arg
	|	`dup`		|	var
				|	$value$
				|	$\{l_1{=}var_1\,;\ \ldots\,;\ l_n{=}var_n\}$

Fig. 3: Syntax of the base fragment

The type system of the base fragment is presented in figure 4. In the case of instruction sequencing $instruction\,;\ instruction'$, we do not want to restrict $instruction'$ to consume exactly the values produced by $instruction$. To avoid

this limitation, we have added the framing rule FRAME. This rule can be used to extend both record types rty and rty' used for typing an instruction *instruction* by the same record type rty''. This extension is performed by the join operator $\odot$, a partial function computing the disjoint union of two record types.

Operational semantics The semantics of the Albert base language is defined in big-step style in Figure 5. The definition of this semantic relation is unsurprising because the base fragment is very simple and the type system does not let much freedom at this point.

3.2 Language extensions

The full Albert language is obtained by adding to the base fragment that we have just defined a series of language extensions. The main purpose of these extensions is to reflect all the features available in Michelson. The only new main feature compared to Michelson is the generalisation of the binary sum type `or` into n-ary non-recursive variants with named constructors.

Albert's variant types generalize the `or`, `option`, and `bool` types of Michelson. Variants are the dual of records, with the caveat that it is not possible to construct an empty variant 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.

Constructors 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 because the same constructor name is allowed to appear in different variant types. We use the syntax `[`C_1 `:` ty_1 `|` .. `|` C_n `:` ty_n`]` for the variant type whose constructors are the $C_1, \ldots, C_n$ where each C_k expects an argument of type ty_k. The types `or a b`, `option a`, and `bool` in Albert are aliases for the variant types `[Left : a | Right : b]`, `[Some : a | None : {}]` and `[False : {} | True : {}]` respectively.

Pattern matching can be used on variants either as a right-hand side or as an instruction. In both cases, the Albert syntax for pattern matching is similar to the OCaml syntax of pattern matching; for right-hand sides, the syntax is `match x with | pattern_1 →rhs_1 | ... | pattern_n →rhs_n end`.

3.3 Example: a voting contract

We present in figure 6 a simple voting contract written in Albert. The user of the contract can vote for a predefined set of options by sending tokens and its choice (represented by a string) to the contract.

The storage of the contract (line 1) is a record with two fields: a `threshold` that represents a minimum amount that must be transferred to the contract for the vote to be considered, and an associative map, `votes`, with strings as keys (the options of the vote) and integers as values (the number of votes for each associated key).

$$\boxed{\Gamma \vdash \mathit{instruction} : \mathit{ty} \Rightarrow \mathit{ty}'}$$

Instruction typing

$$\frac{\begin{array}{l}\Gamma \vdash \mathit{rty}_1 \\ \Gamma \vdash \mathit{rty}_2 \\ \mathit{rty} \odot \mathit{rty}'' = \mathit{rty}_1 \\ \mathit{rty}' \odot \mathit{rty}'' = \mathit{rty}_2 \\ \Gamma \vdash \mathit{instruction} : \mathit{rty} \Rightarrow \mathit{rty}'\end{array}}{\Gamma \vdash \mathit{instruction} : \mathit{rty}_1 \Rightarrow \mathit{rty}_2}\ \textsc{Frame}$$

$$\frac{}{\Gamma \vdash \texttt{noop} : \{\,\} \Rightarrow \{\,\}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{instruction} : \mathit{ty}_1 \Rightarrow \mathit{ty}_2 \\ \Gamma \vdash \mathit{instruction}' : \mathit{ty}_2 \Rightarrow \mathit{ty}_3\end{array}}{\Gamma \vdash \mathit{instruction}\texttt{;}\ \mathit{instruction}' : \mathit{ty}_1 \Rightarrow \mathit{ty}_3}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{rhs} : a \Rightarrow b \\ \Gamma \vdash \mathit{lhs} : b \Rightarrow c\end{array}}{\Gamma \vdash \mathit{lhs}\texttt{=}\mathit{rhs} : a \Rightarrow c}$$

$$\frac{}{\Gamma \vdash \texttt{drop}\ \mathit{var} : \{\mathit{var} : \mathit{ty}\} \Rightarrow \{\,\}}$$

$$\boxed{\Gamma \vdash \mathit{lhs} : \mathit{ty} \Rightarrow \mathit{ty}'}$$

Left-hand sides typing

$$\frac{}{\Gamma \vdash \mathit{var} : \mathit{ty} \Rightarrow \{\mathit{var} : \mathit{ty}\}}$$

$$\frac{}{\Gamma \vdash \{l_1\texttt{=}x_1\texttt{;}\ \ldots\texttt{;}\ l_n\texttt{=}x_n\} : \{l_1 : \mathit{ty}_1\texttt{;}\ \ldots\texttt{;}\ l_n : \mathit{ty}_n\} \Rightarrow \{x_1 : \mathit{ty}_1\texttt{;}\ \ldots\texttt{;}\ x_n : \mathit{ty}_n\}}$$

$$\boxed{\Gamma \vdash \mathit{rhs} : \mathit{ty} \Rightarrow \mathit{ty}'}$$

Right-hand side typing

$$\frac{\Gamma \vdash_a \mathit{arg} : \mathit{ty} \Rightarrow \mathit{ty}'}{\Gamma \vdash \mathit{arg} : \mathit{ty} \Rightarrow \mathit{ty}'}$$

$$\frac{\begin{array}{c}\Gamma \vdash_a \mathit{arg} : \mathit{ty} \Rightarrow \mathit{ty}' \\ \Gamma \vdash f : \mathit{ty}' \Rightarrow \mathit{ty}''\end{array}}{\Gamma \vdash f\ \mathit{arg} : \mathit{ty} \Rightarrow \mathit{ty}''}$$

$$\frac{\begin{array}{c}\{l : \mathit{ty}\} \odot \mathit{rty}\texttt{=}\mathit{rty}' \\ \Gamma \vdash \mathit{rty}'\end{array}}{\Gamma \vdash \mathit{var}.l : \mathit{rty}' \Rightarrow \mathit{ty}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{rty}' \\ \{l_1 : \mathit{ty}_1\texttt{;}\ \ldots\texttt{;}\ l_n : \mathit{ty}_n\} \odot \mathit{rty}\texttt{=}\mathit{rty}'\end{array}}{\Gamma \vdash \{\mathit{var}\ \texttt{with}\ l_1\texttt{=}\mathit{var}_1\texttt{;}\ \ldots\texttt{;}\ l_n\texttt{=}\mathit{var}_n\} : \{\mathit{var} : \mathit{rty}'\texttt{;}\ \mathit{var}_1 : \mathit{ty}_1\texttt{;}\ \ldots\texttt{;}\ \mathit{var}_n : \mathit{ty}_n\} \Rightarrow \mathit{rty}'}$$

$$\boxed{\Gamma \vdash f : \mathit{ty} \Rightarrow \mathit{ty}'}$$

Function symbol typing

$$\frac{}{\Gamma \vdash \texttt{dup} : \mathit{ty} \Rightarrow \{\texttt{car} : \mathit{ty}\texttt{;}\ \texttt{cdr} : \mathit{ty}\}}$$

$$\boxed{\Gamma \vdash_a \mathit{arg} : \mathit{ty} \Rightarrow \mathit{ty}'}$$ Argument typing

$$\frac{}{\Gamma \vdash_a \mathit{var} : \{\mathit{var} : \mathit{ty}\} \Rightarrow \mathit{ty}}$$

$$\frac{\Gamma \vdash \mathit{value} : \mathit{ty}}{\Gamma \vdash_a \mathit{value} : \{\,\} \Rightarrow \mathit{ty}}$$

$$\frac{}{\Gamma \vdash_a \{l_1\texttt{=}x_1\texttt{;}\ \ldots\texttt{;}\ l_n\texttt{=}x_n\} : \{x_1 : \mathit{ty}_1\texttt{;}\ \ldots\texttt{;}\ x_n : \mathit{ty}_n\} \Rightarrow \{l_1 : \mathit{ty}_1\texttt{;}\ \ldots\texttt{;}\ l_n : \mathit{ty}_n\}}$$

Fig. 4: Typing rules for the base fragment

$\boxed{lhs/val \implies val'}$ Left-hand side evaluation

$$\frac{}{var/val \implies \{var{=}val\}}$$

$$\frac{}{\{l_1{=}x_1\,;\ ..\ ;\ l_n{=}x_n\}/\{l_1{=}val_1\,;\ ..\ ;\ l_n{=}val_n\} \implies \{x_1{=}val_1\,;\ ..\ ;\ x_n{=}val_n\}}$$

$\boxed{arg/_a\, val \implies val'}$ Argument evaluation

$$\frac{}{var/_a\{var{=}val\} \implies val}$$

$$\frac{}{val/_a\{\,\} \implies val}$$

$$\frac{}{\{l_1{=}x_1\,;\ ..\ ;\ l_n{=}x_n\}/_a\{x_1{=}val_1\,;\ ..\ ;\ x_n{=}val_n\} \implies \{l_1{=}val_1\,;\ ..\ ;\ l_n{=}val_n\}}$$

$\boxed{f/val \implies val'}$ Function symbol evaluation

$$\frac{}{\texttt{dup}/val \implies \{\texttt{car}{=}val\texttt{;}\ \texttt{cdr}{=}val\}}$$

$\boxed{rhs/val \implies val'}$ Right-hand side evaluation

$$\frac{arg/_a\, val \implies val'}{arg/val \implies val'}$$

$$\frac{arg/_a\, val \implies val' \qquad f/val' \implies val''}{f\ arg/val \implies val''}$$

$$\frac{\{l{=}val\} \odot rval{=}rval'}{var.l/rval' \implies val}$$

$$\frac{\{l_1{=}val_1'\,;\ ..\ ;\ l_n{=}val_n'\} \odot rval{=}rval' \qquad \{l_1{=}val_1\,;\ ..\ ;\ l_n{=}val_n\} \odot rval{=}rval''}{\{var\ \texttt{with}\ l_1{=}var_1\,;\ ..\ ;\ l_n{=}var_n\}/\{var{=}rval'\texttt{;}\ var_1{=}val_1\,;\ ..\ ;\ var_n{=}val_n\} \implies rval''}$$

$\boxed{instruction/val \implies val'}$ Instruction evaluation

$$\frac{instruction/rval \implies rval' \qquad rval \odot rval''{=}rval_1 \qquad rval' \odot rval''{=}rval_2}{instruction/rval_1 \implies rval_2}$$

$$\frac{}{\texttt{noop}/\{\,\} \implies \{\,\}}$$

$$\frac{I_1/val \implies val' \qquad I_2/val' \implies val''}{I_1\texttt{;}\ I_2/val \implies val''}$$

$$\frac{rhs/val \implies val' \qquad lhs/val' \implies val''}{lhs{=}rhs/val \implies val''}$$

$$\frac{}{\texttt{drop}\ var/\{var{=}val\} \implies \{\,\}}$$

Fig. 5: Big-step operational semantics of the base fragment

If the user sends less tokens that the threshold or if the parameter sent is not one of the options (the keys of the `votes` map), then the call to the contract will fail.

The contract contains two functions, `vote` and `guarded_vote`. Both functions respect Michelson's call conventions: they take as input the parameter and the storage combined and return a list of operations and an updated storage.

`vote` checks that the parameter is one of the voting options (l. 9 and 10). If not, the contract fails (due to assert_some in l.10). Otherwise, the number of votes associated to the parameter is increased by one (l. 11 and 12). `vote` returns an updated storage as well as an empty list of operations.

`guarded_vote`, the main function, checks that the amount of tokens sent (obtained with the amount primitive instruction l.21) is greater or equal to the threshold (l.22). If so, then `vote` is applied. Otherwise, it fails.

```
1  type storage_ty = { threshold : mutez; votes: map string nat }
2
3  def vote :
4    { param : string ; store : storage_ty } →
5    { operations : list operation ; store : storage_ty } =
6        {votes = state; threshold = threshold } = store ;
7        (state0, state1) = dup state;
8        (param0, param1) = dup param;
9        prevote_option = state0[param0];
10       { res = prevote } = assert_some { opt = prevote_option };
11       one = 1; postvote = prevote + one; postvote = Some postvote;
12       final_state = update state1 param1 postvote;
13       store = {threshold = threshold; votes = final_state};
14       operations = ([] : list operation)
15
16 def guarded_vote :
17   { param : string ; store : storage_ty } →
18   { operations : list operation ; store : storage_ty } =
19     (store0, store1) = dup store;
20     threshold = store0.threshold;
21     am = amount;
22     ok = am >= threshold0;
23     match ok with
24         False f →failwith "you_are_so_cheap!"
25       | True t →drop t;
26           voting_parameters = { param = param ; store = store1 };
27           vote voting_parameters
28     end
```

Fig. 6: A voting contract, in Albert

4 Compilation to Michelson

4.1 Compiler architecture

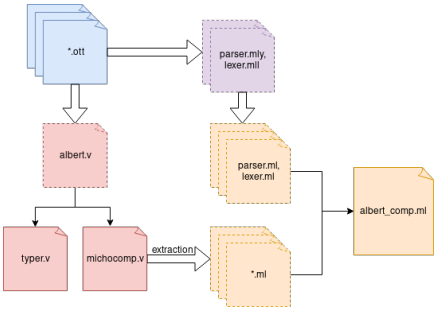

Fig. 7: Compiler architecture: dashed frames designate generated component, solid arrows represent relevant library dependencies.

As we want to be able to prove the correctness of our compiler in a near future, we decided to implement it in Coq. This allows us to easily take advantage of Ott's definitions automatically translated to Coq, as well as to easily compile to Mi-Cho-Coq's AST. Moreover, using Coq's extraction facilities, our compiler transpiles to OCaml code, which is more efficient and easier to use as a library.

The global architecture of the compiler is depicted in 7. The compiler pipeline, defined using OCaml glue code, roughly follows a classic architecture, notwithstanding the peculiar tools used: the lexer-parser, automatically generated from the grammar described in Ott, produces an AST which is then checked and annotated by the typer, extracted from a Coq development. Then, the compilation function, also written in Coq and extracted to OCaml, translates the typed Albert AST into an untyped Mi-Cho-Coq AST. Finally, the extracted Mi-Cho-Coq pretty-printer is used to produce a string which is a Michelson program, and which the glue code dumps into a file ready to be injected in the Tezos blockchain.

Typechecker The type checker phase can be divided in three steps.

First, type aliases declared by the user are replaced by their actual definition. This will simplify the verification of type equivalence in the next phases, as we will not have to worry about type variables. As declared types are simple aliases - types can't be recursively declared – this amounts to inlining the type aliases wherever they are found in the program.

The second step normalises type declarations by sorting in lexicographic order both the fields of records and the constructors of variants.

Finally, the third step checks that all defined functions are well typed. Currently, this type-checking proceeds in one pass from top to bottom and it does

not perform any type inference. It checks the linearity of variable usage, the compatibility of operands' types with their operator and the exhaustiveness of pattern matching. Each instruction is annotated with an input and output environments. The environment being a record type, associating a type to each variable name. One can note here that this record type is actually a description of the Michelson stack at each point of the program where position have been replaced by names.

The type checker is defined as a Coq function, thus is a total function. Its implementation uses an error monad to deal with ill-typed programs. If a program does not type check, an error message is returned instead of the typed version of the AST.

The lack of type inference is not too much of a limitation since the higher-level languages that will target Albert have enough type information to produce the explicit type annotations that are mandatory, as for example on variant constructors.

4.2 Compilation scheme

To compile an Albert program to a Michelson program, we need first to convert Albert's types to Michelson's types and Albert's data to Michelson's data, then to translate Albert instructions to an equivalent Michelson sequence of instructions.

Types and data Because Albert's primitive types reflect Michelson types, their translation is obvious. Only the translation of records and variants is not trivial. Records are translated into nested pairs of values, whereas variants are translated into a nesting of sum types. For the sake of simplicity, we use a comb shaped nesting, making access to records' fields and size of variant constructor linear in the size of the Albert type. A future task will be to provide a syntax to control the shape of the Michelson translation or to use a balanced tree shape.

Instructions The compilation scheme of instructions is rather straightforward. Projections of records fields are translated into a sequence of projections over the relevant parts of a pair. Pattern matching over variants are translated into a nesting of `IF_LEFT` branchings. Each branch of an Albert pattern-matching is translated in Michelson and inserted in the associated position of the Michelson `IF_LEFT` branchings tree.

At every point of the program we memorise a mapping from variable names to their positions in the stack. Each operation is then translated to the equivalent operation in Michelson, prefixed by `DIG n` operations that move the operands on top of the stack, `n` being the index of the variables used as operands.

Function arguments are brought back on top of the stack if they are variables and are pushed on it if they are literals. The Michelson translation of the function is then inlined.

Assignment instructions translate into a translation of the right hand side computation, followed by a reordering of data, guided by the shape of the left hand side: simple variable assignments `DUG` the result deeper in the stack for later use, while record patterns translate to a pairing destruction and then some stack reorganisation.

Our mapping from variable names to stack positions is currently naive and enforces the invariant that the elements of the stack are ordered by the lexicographic order of the variable names. This requires too much stack reorganisation and will be later replaced by an optimising placement algorithm.

5 Related Work

Formal verification of smart contracts is a recent but active field. The K framework has been used to formalise [11] the semantics of both low-level and high-level smart-contract languages for the Ethereum and Cardano blockchains. These formalisations have been used to verify common smart contracts such as Casper, Uniswap, and various implementations of the ERC20 and ERC777 standards. A formalization of Michelson in the K framework[19] is also under development.

Note also a formalisation of the EVM in the F* dependently-typed language [10], that was validated against the official Ethereum test suite. This formalisation effort led to formal definitions of security properties for smart contracts (call integrity, atomicity, etc).

Ethereum smart contracts, written in the Solidity high-level language, can also be certified using a translation to F* [8].

The Zen Protocol [4] directly uses F* as its smart-contract language so that smart contracts of the Zen Protocol can be proved directly in F*. Moreover, runtime tracking of resources can be avoided since computation and storage costs are encoded in the dependent types.

The Scilla [16] language of the Zilliqa blockchain has been formalised in Coq as a shallow embedding. This intermediate language is higher-level (it is based on λ-calculus) but also less featureful (it is not Turing-complete as it does not feature unbounded loops nor general recursion) than Michelson and Albert. Its formalisation includes inter-contract interaction and contract lifespan properties. This has been used to show safety properties of a crowdfunding smart contract. Moreover, Scilla's framework for writing static analyses [17] can be used for automated verification of some specific properties.

In the particular case of the Tezos platform, several high-level languages are being developed [5,14,6,9,12,13] to ease the development of smart contracts. Formal specification is featured in the Archetype language[9], the specification is then translated to the Why3 platform for automated verification. In Juvix[13], dependent types can be used to specify and verify smart contracts and resources are tracked in a similar fashion to Albert's linear type system thanks to a variant of quantitative type theory in Juvix's core language.

6 Conclusion and Future Work

The Albert intermediate language has been formally specified in a very modular way using the Ott framework. This formal specification is the unique source from which Albert's parser (written in Menhir), Albert's typechecker and compiler (written in Coq) and the Section 3 of this article (written in LaTeX) are generated.

The current implementation of the compiler is rather naive and we plan to improve the performance of the produced code by sorting the values on the Michelson stack not by the name of the corresponding Albert variable but by their last use so that no work is performed after a variable assignment to dive it back to its position in the stack. This will however add some complexity in the compiler when several branches of a pattern-matching construction are joined because we will need to permute the stack in all but one of them to recover matching stack types in all branches.

The Coq versions of the language specification and the compiler open the possibility of certifying the compiler correctness and meta-properties of the Albert language such as subject reduction and progress. We have started proving these properties in Coq to improve the trust in the Albert tools.

Finally, we would like to add to Albert a specification language and support for deductive verification through the use of ghost code so that functional verification of Tezos smart contracts can be performed with the very high level of confidence offered by Coq and Mi-Cho-Coq but at a higher level than Michelson.

References

1. Michelson: the language of Smart Contracts in Tezos. `https://tezos.gitlab.io/whitedoc/michelson.html`
2. Proof-of-stake in Tezos. `https://tezos.gitlab.io/whitedoc/proof_of_stake.html`
3. Tezos code repository. `https://gitlab.com/tezos/tezos`
4. An introduction to the zen protocol. `https://www.zenprotocol.com/files/zen_protocol_white_paper.pdf` (2017)
5. Alfour, G.: LIGO: a friendly smart-contract language for Tezos. `https://ligolang.org`, accessed: 2019-12-12
6. Andrews, S., Ayotte, R.: fi: Smart coding for smart contracts. `https://fi-code.com`, accessed: 2019-12-12
7. Bernardo, B., Cauderlier, R., Hu, Z., Pesin, B., Tesson, J.: Mi-Cho-Coq, a framework for certifying Tezos Smart Contracts. In: Proceedings of the First Workshop on Formal Methods for Blockchains (to be published). FMBC 2019 (2019), `https://arxiv.org/abs/1909.08671`
8. Bhargavan, K., Delignat-Lavaud, A., Fournet, C., Gollamudi, A., Gonthier, G., Kobeissi, N., Kulatova, N., Rastogi, A., Sibut-Pinote, T., Swamy, N., Zanella-Béguelin, S.: Formal verification of smart contracts: Short paper. pp. 91–96. PLAS '16, ACM, New York, NY, USA (2016). https://doi.org/10.1145/2993600.2993611
9. Duhamel, G., Rognier, B., Sturb, P.Y., edukera: Archetype: a Tezos smart contract development solution dedicated to contract quality insurance. `https://docs.archetype-lang.org`, accessed: 2019-12-12

10. Grishchenko, I., Maffei, M., Schneidewind, C.: A semantic framework for the security analysis of ethereum smart contracts. In: Bauer, L., Küsters, R. (eds.) Principles of Security and Trust. pp. 243–269. Springer International Publishing, Cham (2018). https://doi.org/10.1007/978-3-319-89722-6_10
11. Hildenbrandt, E., Saxena, M., Zhu, X., Rodrigues, N., Daian, P., Guth, D., Moore, B., Zhang, Y., Park, D., Ştefănescu, A., Roşu, G.: Kevm: A complete semantics of the ethereum virtual machine. In: 2018 IEEE 31st Computer Security Foundations Symposium. pp. 204–217. IEEE (2018)
12. Inc., D.: SCaml: it's not a scam. `https://www.dailambda.jp/blog/2019-12-07-scaml/`, accessed: 2019-12-12
13. Labs, C.: Juvix: a more elegant language for a more civilized age. `https://github.com/cryptiumlabs/juvix`, accessed: 2019-12-12
14. Maurel, F., Arena, S.C.: SmartPy. `https://smartpy.io`, accessed: 2019-12-12
15. Nomadic Labs: Mi-Cho-Coq public repository. `https://gitlab.com/nomadic-labs/mi-cho-coq`
16. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR **abs/1801.00687** (2018), `http://arxiv.org/abs/1801.00687`
17. Sergey, I., Nagaraj, V., Johannsen, J., Kumar, A., Trunov, A., Hao, K.C.G.: Safer smart contract programming with scilla. PACMPL **3**(OOPSLA), 185:1–185:30 (2019). https://doi.org/10.1145/3360611
18. Sewell, P., Nardelli, F.Z., Owens, S.: Ott. `https://github.com/ott-lang/ott`
19. Verification, R.: A K semantics of Tezos' Michelson language. `https://github.com/runtimeverification/michelson-semantics`, accessed: 2019-12-12