Making Tezos smart contracts more reliable with Coq

Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, 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. This article gives an overview of efforts using the Coq proof assistant to have stronger guarantees on Michelson smart contracts: the Mi-Cho-Coq framework, a Coq library defining formal semantics of Michelson, as well as an interpreter, a simple optimiser and a weakest-precondition calculus to reason about Michelson smart contracts; Albert, an intermediate language that abstracts Michelson stacks with a compiler written in Coq that targets Mi-Cho-Coq.

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

The final authenticated publication is available online at https://doi.org/10.1007/978-3-030-61467-6_

1 Introduction

Tezos [15,16,5] is a public blockchain launched in June 2018. An open-source implementation of a Tezos node in OCaml is available [2]. Tezos has smart-contracts capabilities, a proof-of-stake consensus algorithm, and a voting mechanism that allows token holders to vote for changes to a subset of the codebase called the *economic protocol*. This subset contains, amongst other elements, the voting rules themselves, the consensus algorithm, and the interpreter for Michelson, the language for Tezos smart contracts.

Michelson [1] is a stack-based Turing-complete domain-specific language with a mix of low-level and high-level features. Low-level features include stack manipulation instructions. High-level features are high-level data types (option types, sum types, product types, lists, sets, maps, and anonymous functions) as well as corresponding instructions. Michelson is strongly typed: data, stacks and instructions have a type. Intuitively the type of a stack is a list of the types of its values, and the type of an instruction is a function type from the input stack to the output stack. The combination of high and low level features is the result of a trade-off between the need to meter resource consumption (computation *gas* and storage costs) and the willingness to have strong guarantees on the Michelson programs.

Michelson has been designed with formal verification in mind: its strong type system guarantees that there can be no runtime error apart from explicit failure and *gas* or token exhaustion. Furthermore, its OCaml implementation uses GADTs [25] which gives subject reduction for free. In this article, we describe a couple of efforts using Coq to make Tezos smart contracts more reliable. The first one is Mi-Cho-Coq, a Coq library that implements a Michelson interpreter, a weakest precondition calculus enabling the functional verification of Michelson programs as well as a very simple optimiser.

The second one is Albert, an intermediate language that abstracts Michelson stacks into records with named variables, for which we have implemented a compiler in Coq that targets Mi-Cho-Coq. Because of its low-level aspects, it is hard to write Michelson programs, and as a consequence, higher-level languages compiling to Michelson, such as LIGO [4] or SmartPy [7] have been developed in the Tezos ecosystem. Ideally, there would be certified compilers from these high-level languages to Michelson, and formal proofs of smart-contracts would be done directly at the higher level and not at the Michelson level, as it is done with Mi-Cho-Coq. The goal of Albert is to facilitate the implementation of certified compilers to Michelson by being used as a target for certified compilers from high-level languages.

This article is organised as follows: in Section 2 we illustrate the Michelson language with an example, we describe the Mi-Cho-Coq library in Section 3 and Albert in Section 4. Future and related work is discussed in Section 5.

2 Example of a Michelson contract

The goal of this section is to give the reader an intuitive feeling of the Michelson language. Our explanations will be illustrated by an example of a Michelson program, a voting contract, presented in figure 1a. This contract allows any voter to vote for a predefined set of choices. Voting requires a fee of 5 tez. It is possible to vote multiple times. The predefined set of choices is the initial storage chosen at the deployment of the contract. In this example, we assume that we want to vote for our favourite proof assistant amongst Agda, Coq and Isabelle (cf. fig. 1b). Initially, each tool has obviously 0 vote.

Smart contracts are accounts that can contain code and storage. Calls to a smart-contract provokes the execution of the code contained in the account with the data sent during the transaction as input arguments for the code. The execution of the code can lead to a modification of the storage, it can also generate other transactions. The storage must be initialised when the contract is deployed on the chain.

A Michelson program is thus composed of three fields: `storage`, `parameter` and `code` that respectively contain types of the *storage* of the account, of the *parameter* that is sent during the transaction, or the *code* contained in the account.

In the case of the voting contract, the `storage` (l.1) is a map from strings to integers: the keys are the different choices for the vote, the values are the number of votes for a given choice. The `parameter` (l.2) is a string that represents the ballot that has been chosen by the caller of the contract.

As mentioned in the introduction, Michelson is a stack based language. The calling convention of Michelson is the following: the initial stack contains one element that is a pair of which the left member is the parameter sent by the transaction and the right element is the initial storage. At the end of the execution of a Michelson script, the stack must have one element that is a pair containing on the left a list of operations (*e.g.* a transaction) that will be executed afterwards and on the right the updated storage. Each instruction takes an input stack and rewrites it into an output stack. In the comments of the example are written the content of the stack before and after the execution of some groups of instructions. The program starts by verifying that enough tokens were sent by the voters. This is implemented in lines 5 to 8. The amount sent by the voter is pushed to the stack (`AMOUNT`) followed by the minimum amount required (5000000 μtez *i.e* 5 *tez*). `COMPARE` pops the two amounts and pushes 1,0,-1 whether the 5 *tez* threshold is greater, equal or smaller than the amount sent by the voter. If the threshold is greater then the contract will `FAIL`: the execution of the contract is stopped and the transaction is cancelled. Lines 9 to 11 contain stack manipulations that duplicate (with instruction `DUP`) the ballot and the current vote count. `DIP {code}` protects the top of the stack by executing `code` on the stack without its top element. The next block, from line 12 to 18, tries to `UPDATE` (l.18) the map with an incremented number of votes for the chosen candidate. This only happens if a candidate is a valid one, that is, if it is equal to one of the keys of the map. Indeed, in line 13 `GET` tries to retrieve the number of votes: it returns None if the chosen candidate is not in the list or Some `i` if the candidate is in the list and has `i` votes. `ASSERT_SOME` will fail if None is at the top of the stack and will pop Some `i` and push `i` at the top. The incrementation by one of the number of votes for the chosen candidate is done in l.15.

3 Mi-Cho-Coq: defining clear semantics of Michelson

Mi-Cho-Coq [8] is a Coq library that contains an implementation of the Michelson syntax and semantics as well as a weakest precondition calculus that facilitates functioning verification of Tezos smart contracts. Also, we have recently implemented a certified optimiser that performs basic simplifications of Michelson programs.

Mi-Cho-Coq has already been presented in [8] and we refer the reader to this publication for more details. Here we present Mi-Cho-Coq more succintly and focus on high-level additions and changes, as the implementation has evolved significantly.

3.1 Syntax, Typing and Semantics of Michelson in Coq

Syntax and typing The data stored in the stacks have a type defined in the `type` inductive. The type of a stack is a list of `type`. Instructions are defined in

```
storage (map string int); # candidates
parameter string; # chosen
code {
  # (chosen, candidates):[]
  AMOUNT; # amount:(chosen, candidates):[]
  PUSH mutez 5000000; COMPARE; GT;
  # (5 tez > amount):(chosen, candidates):[]
  IF { FAIL } {}; # (chosen, candidates):[]
  DUP; DIP { CDR; DUP };
  # (chosen, candidates):candidates:candidates:[]
  CAR; DUP; # chosen:chosen:candidates:candidates:[]
  DIP { # chosen:candidates:candidates:[]
        GET; ASSERT_SOME;
        # candidates[chosen]:candidates:[]
        PUSH int 1; ADD; SOME
        # (Some (candidates[chosen]+1)):candidates:[]
      }; # chosen:(Some (candidates[chosen]+1)):candidates:[]
  UPDATE; # candidates':[]
  NIL operation; PAIR # (nil, candidates'):[]
}
```

(a)

```
{Elt "Agda" 0 ; Elt "Coq" 0 ; Elt "Isabelle" 0}
```

(b)

Fig. 1: A simple voting contract a and an example of initial storage b

the instruction inductive type. instruction is indexed by the type of the input and output stacks. This indexing implies that only well-typed Michelson instructions are representable in Mi-Cho-Coq. [1] A full contract is a sequence of instructions respecting the calling convention of Michelson mentioned above:

```
instruction ((pair params storage) :: nil) ((pair (list operation) storage) :: nil).
```

where storage is the type of the storage of the contract and params the type of its parameter.

Implementation-wise, Coq's canonical structures are used to deal with the ad-hoc polymorphism of some Michelson instructions (*e.g* `ADD` that can add integers to `timestamp` or `mutez` or integers). Coq's notations make contracts'appearance in Mi-Cho-Coq very close to actual contracts.

Also a lexer, parser and typechecker have been implemented, making it possible to generate a Mi-Cho-Coq AST from a string representing a Michelson contract. Support for Michelson entry-points and annotations has been added.

Semantics An interpreter for Michelson has been implemented as an evaluator eval. Its simplified type is `forall` {A B : list type}, instruction A B → nat → stack A

[1] This is also the case in the OCaml Michelson interpreter via the use of GADTs.

→ M (stack B). Intuitively the interpreter takes a sequence of instructions and an input stack and returns an output stack. Since Michelson programs can explicitly fail the output stack is embedded in an error monad M. A *fuel* argument is added to enforce termination of the interpreter. This argument will decrease every time in any recursive call to eval. Note that the notion of *fuel* is different from *gas*, which measures computation costs.

3.2 Functional verification of Michelson smart contracts

We have verified the functional correctness of Michelson contracts with Mi-Cho-Coq, including complex ones such as a multisig contract (cf. section 4 of [8]) or a daily spending limit contract [2] used in the Cortez mobile wallet [3]. Our correctness results are statements that condition successful runs of a contract with the respect of a specification:

```
Definition correct_smart_contract {A B : stack_type}
     (i : instruction A B) min_fuel spec : Prop :=
     forall (input : stack A) (output : stack B) fuel,
       fuel >= min_fuel input →
       eval i fuel input = Return (stack B) output <→
         spec input output.
```

For example, for the voting contract described in section 2, the specification would be that (preconditions) the amount sent is greater than or equal to 5 tez, the chosen candidate is one the possible choices and that (postconditions) the evaluation of the contract generates no operation, and modifies only the votes count by incrementing by 1 the number of votes of the chosen candidate.

In order to facilitate these functional proofs, a weakest precondition calculus eval_precond is implemented. Its type is `forall` {fuel A B}, instruction A B → (stack B → `Prop`) → (stack A → `Prop`) that for an instruction and a postcondition (a predicate over the output stack) returns the weakest precondition (a predicate over the input stack).

The correctness of eval_precond has been proven:

```
Lemma eval_precond_correct {A B} (i : instruction A B) fuel st psi :
  eval_precond fuel i psi st <→
    match eval i fuel st with Failed _ _ => False | Return _ a => psi a end.
```

Intuitively, eval_precond_correct states that (left to right) the computed by eval_precond is a precondition and that (right to left) it is the weakest. This lemma is heavily used in the proofs of correctness.

3.3 Optimiser

A Michelson optimiser has been implemented in Mi-Cho-Coq. The purpose of this optimiser is to simplify Michelson programs, thus reducing the gas costs of

[2] https://blog.nomadic-labs.com/formally-verifying-a-critical-smart-contract.html
[3] https://gitlab.com/nomadic-labs/cortez-android

executing them, without modifying their semantics. Simplifications are basic at the moment: the goal is mainly to clean programs generated from higher level languages by removing useless stack manipulations instructions.

Optimisations are defined in one file, `optimizer.v`. A first step (`dig0dug0`) removes useless instructions (`DROP 0`, `DIG 0` and `DUG 0`), needless uses of `DIP` (`DIP 0 i` is replaced with `i`) and replaces `DIG 1` and `DUG 1` with `SWAP`. A second step (`digndugn`) removes `DIG n; DUG n` sequences. A third step (`swapswap`) removes `SWAP ; SWAP` sequences. A fourth step (`push_drop`) removes `PUSH ; DROP 1` and rewrites `PUSH ; DROP n+1` into `DROP n` (for `n > 0`). The `visit_instruction` function, similarly to the Visitor Pattern [12], traverses a Michelson sequence of instructions and applies one optimisation received as an argument. Finally, the `cleanup` function applies the four optimisations (in the order of their presentation above) to a sequence of instructions.

We prove that the semantics of Michelson instructions are preserved by the optimisations. This is implemented in `typed_optimizer.v`. The main theorem `optimize_correct` states that if an instruction sequence can by typechecked then its optimised version can also be typechecked with the same type; furthermore if the initial sequence runs successfully on some stack, then the optimised version runs also successfully on the same stack and they both return the same value.

4 Albert

Albert is an intermediate language for Tezos smart contracts with a compiler written in Coq and that targets Mi-Cho-Coq.

We present in this section a high level overview of Albert's design and features. A more detailed presentation of Albert's syntax, typing rules and semantics can be found in [9].

4.1 Design overview

The key aspect of Albert's design is the abstraction of Michelson stacks by records with named fields. This gives two practical benefits: unlike in Michelson, in Albert we do not need to care about the order of the values and we can bind variables to names. Also, unlike Michelson where contracts can only contain one sequence of instructions, it is possible in Albert to define multiple functions, thus giving the possibility to implement libraries. An important limitation of Albert is that resources are still being tracked: variables are typed by a linear type system that enforces that each value cannot be consumed twice. A **dup** operation duplicates resources that need to be consumed multiple times. A next step would be to generate these operations in order to abstract data consumption.

In a nutshell, each expression or instruction 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 generalise both the Michelson stack types and the Michelson pair type.

Albert offers slightly higher-level types than Michelson: records generalise Michelson's pairs and non-recursive variants generalise Michelson's binary sum types as well as booleans and option types. Variants offer two main operations to the user: constructing a variant value using a constructor, and pattern-matching on a variant value.

The semantics of the Albert base language is defined in big-step style.

We present in figure 2 the translation in Albert of the voting contract described in section 2. The storage of the contract is a record with two fields: a `threshold` that represents the minimum amout that must be transferred, 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). The contract contains a `vote` function that checks that the parameter sent is one of the available options, fails if not and otherwise updates the vote count. The main function `guarded_vote` verifies that the amount of tokens sent is high enough and if so, calls `vote`.

4.2 Implementation overview

Albert is formally specified with the Ott tool [24] in a modular way (one `.ott` file per fragment of the language). From the Ott specification the Albert lexer and parser as well as typing and semantic rules are generated in Coq. The type checker is a Coq function that uses an error monad to deal with ill-typed programs. There is no type inference, which should not be a problem since Albert is supposed to be used as a compilation target.

The Albert compiler is written in Coq, as a function from the generated Albert grammar to the Michelson syntax defined in Mi-Cho-Coq. The compiler is extracted to OCaml code, which is more efficient and easier to use as a library. Compilation of types, data and instructions are mostly straightforward, apart from things related to records or variants. Records are translated into nested pairs of values, variants into a nesting of sum types. Projections of record fields are translated into a sequence of projections over the relevant components of a pair. Pattern matching over variants are translated into a nesting of `IF_LEFT` branchings. A mapping from variable names to their positions in the stack exists at every point in the program. This mapping is currently naive, variables are ordered by the lexicographic order of their names. This mapping is used in the translation of assignment instructions.

5 Future and related work

5.1 Towards stronger guarantees in the OCaml Michelson interpreter

The OCaml implementation of Tezos contains an interpreter for Michelson. This interpreter is implemented with GADTs in a way that gives subject reduction

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

def vote :
  { param : string ; store : storage_ty } →
  { operations : list operation ; store : storage_ty } =
      {votes = state; threshold = threshold } = store ;
      (state0, state1) = dup state;
      (param0, param1) = dup param;
      prevote_option = state0[param0];
      { res = prevote } = assert_some { opt = prevote_option };
      one = 1; postvote = prevote + one; postvote = Some postvote;
      final_state = update state1 param1 postvote;
      store = {threshold = threshold; votes = final_state};
      operations = ([] : list operation)

def guarded_vote :
  { param : string ; store : storage_ty } →
  { operations : list operation ; store : storage_ty } =
    (store0, store1) = dup store;
    threshold = store0.threshold;
    am = amount;
    ok = am >= threshold0;
    match ok with
        False f →failwith "you_are_so_cheap!"
      | True t →drop t;
          voting_parameters = { param = param ; store = store1 };
          vote voting_parameters
    end
```

Fig. 2: A voting contract, in Albert

for free: well-typed Michelson programs cannot go wrong: with the calling convention, we have the guarantee that well-typed programs will always be executed with stacks of the right length with data of the right type.

Nonetheless, because of the limitations of the logic implemented in OCaml, we are unable to reason directly about this interpreter. In this section, we sketch two possible solutions to this problem that we are currently exploring.

From Coq to OCaml An obvious solution is to use Coq's extraction to produce OCaml code. Coq's extraction mechanism relies on well-studied theoretical grounds [20,21], has been implemented for many years [19,18] and has even been partially certified [13,14]. However, OCaml code produced by Coq's extraction can contain `Obj.magic` to circumvent OCaml's less expressive type-system. That is in particular the case for Coq code that uses dependent types, such as Mi-Cho-Coq's interpreter.

Replacing the current OCaml Michelson interpreter with an extracted version containing `Obj.magic` occurrences would be problematic. The Michelson interpreter is part of the economic protocol which is sandboxed by a small subset of OCaml modules that, for obvious safety reasons, does not contain `Obj.magic`. Lifting this restriction would lower the guarantees provided by the OCaml type system and is not a path we would like to take.

A better solution would be to implement a second Michelson interpreter in Coq that would use simpler types so that its extraction is safe and compiles in the economic protocol sandboxing environment. In this second implementation, instructions' types would not be indexed by the input and output stacks and, as a consequence, the interpreter would be much more verbose, as all the ill-cases (e.g. executing `ADD` on a stack with only one element) would need to be dealt with. Proofs of equivalence between the typed interpreter and untyped interpreter would be needed, as the reference implementation would be the typed one, that uses dependent types, as the use of richer types would make it safer. It is also worth noting that any changes to the Michelson interpreter have to be approved by a community vote as it is part of the economic protocol that can be amended.

From OCaml to Coq with coq-of-ocaml A reverse approach is to use the coq-of-ocaml [10] tool to mechanically translate the OCaml Michelson interpreter into Coq code and then to manually prove in Coq the equivalence of the two interpreters.

coq-of-ocaml [10] is a work in progress effort to compile OCaml code to Coq. As a simple example to illustrate coq-of-ocaml, let us consider a polymorphic tree and a sum function that sums the values of a tree of integers. The OCaml code and its coq-of-ocaml translations are respectively in figure 3) and figure 4. Notice that in Coq, the type of the values stored in the trees appears in the type of the tree, as the rich type system of Coq does not allow this to remain implicit. It is possible to reason about the generated Coq program. For example one could prove manually that the sum of a tree containing only positive integers is positive (5). The Michelson interpreter is much more complex than this simple example

```
type 'a tree =
| Leaf of 'a
| Node of 'a tree * 'a tree

let rec sum tree =
  match tree with
  | Leaf n -> n
  | Node (tree1, tree2) -> sum tree1 + sum tree2
```

Fig. 3: Sum of a tree, in OCaml

```
Inductive tree (a : Type) : Type :=
| Leaf : a -> tree a
| Node : (tree a) -> (tree a) -> tree a.

Arguments Leaf {_}.
Arguments Node {_}.

Fixpoint sum (tree : tree Z) : Z :=
  match tree with
  | Leaf n => n
  | Node tree1 tree2 => Z.add (sum tree1) (sum tree2)
  end.
```

Fig. 4: Sum of a tree, in Coq

```
Inductive pos : tree Z -> Prop :=
| PosLeaf : forall z, z > 0 -> pos (Leaf z)
| PosNode : forall t1, t2, pos t1 -> pos t2 -> pos (Node t1 t2).

Fixpoint positive_sum (t : tree Z) (H : pos t) : sum t > 0.
Proof.
  destruct H; simpl.
  - trivial.
  - assert (sum t1 > 0).
    now apply positive_sum.
    assert (sum t2 > 0).
    now apply positive_sum.
    lia.
Qed.
```

Fig. 5: The sum of a positive tree is positive, manual proof in Coq

and makes a heavy use of advanced OCaml features, such as GADTs. At the moment of writing, features such as side-effects, extensible types, objects and polymorphic variants, are not supported yet by coq-of-ocaml. Regarding GADTs, currently coq-of-ocaml translates them but in a way that generates axioms for casting. Work is being done to try to have an axiom-free translation. We have managed to translate the whole economic protocol of the Babylon[4] version of Tezos into Coq using coq-of-ocaml with the caveat that Coq axioms are needed and that OCaml annotations were added. As the Michelson interpreter is part of the economic protocol, this means we have a Coq translation of the OCaml interpreter. A next step would be to prove the equivalence of the translated interpreter with the Mi-Cho-Coq one.

5.2 Improvements to Mi-Cho-Coq and Albert

Mi-Cho-Coq Mi-Cho-Coq has axioms for domain specific opcodes that are harder to implement such as instructions to query the blockchain environment for relevant information (amount sent during the transaction, current time, sender of the transaction,...), data serialisation and cryptographic primitives. Work is being done to decrease the number of axioms being used. Another useful addition would be to extend the expressivity of the framework by supporting mutual calls and calls to other contracts, as well as reasoning on the lifetime of a contract. Another issue that would need to be dealt with would be to implement the gas model in Mi-Cho-Coq. A better or longer term solution would be to replace gas accounting with static computation of execution costs *à la* Zen Protocol [3].

Albert Albert is very much a work in progress. Next steps would be to have a smarter implementation of the compiler that would produce optimised code, as well as to prove the compiler correctness and meta-properties of the Albert language such as subject reduction. Longer term, we would like to implement a certified decompiler from Michelson to Albert as well as a weakest-precondition calculus to Albert in order to reason about Albert programs.

5.3 Related work

Despite the novelty of the field, many works regarding formal verification of smart contracts have been published or announced. The K framework has been used to formalise semantics of smart-contracts languages for Cardano[5], Ethereum[6] and Tezos[7]. Concordium has developed the ConCert certification framework [6], implemented in Coq, that permits to reason on the lifetime of a contract. Scilla [22,23], the smart-contract language of Zilliqa has been formalised in Coq as a shallow

[4] Babylon is the codename of the second amendment to the economic protocol voted by the Tezos community. It has been superseded by Carthage in March 2020.
[5] `https://github.com/kframework/plutus-core-semantics`
[6] `https://github.com/kframework/evm-semantics`
[7] `https://github.com/runtimeverification/michelson-semantics`

embedding. Its formalisation supports inter-contract interaction and multiple calls over the lifetime of the contract.

Several high-level languages for Tezos smart contracts are being developed [4,7,11,17]. Programs in the Archetype language[11] can contain security properties assertions or specifications expressed in logic formulae that are translated to the Why3 platform and then verified by automatic provers such as Alt-ergo, CVC or Z3. Juvix [17] implements a variant of the quantitative type theory which enables to track resources, similarly to Albert, as well as to specify and verify smart contracts.

References

1. Michelson: the language of Smart Contracts in Tezos. `https://tezos.gitlab.io/whitedoc/michelson.html`
2. Tezos code repository. `https://gitlab.com/tezos/tezos`
3. An introduction to the zen protocol. `https://www.zenprotocol.com/files/zen_protocol_white_paper.pdf` (2017)
4. Alfour, G.: LIGO: a friendly smart-contract language for Tezos. `https://ligolang.org`, accessed: 2020-05-12
5. Allombert, V., Bourgoin, M., Tesson, J.: Introduction to the tezos blockchain (2019)
6. Annenkov, D., Nielsen, J.B., Spitters, B.: Concert: a smart contract certification framework in coq. In: Blanchette, J., Hritcu, C. (eds.) Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2020, New Orleans, LA, USA, January 20-21, 2020. pp. 215–228. ACM (2020). https://doi.org/10.1145/3372885.3373829, `https://doi.org/10.1145/3372885.3373829`
7. Arena, S.C.: SmartPy. `https://smartpy.io`, accessed: 2019-12-12
8. 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`
9. Bernardo, B., Cauderlier, R., Pesin, B., Tesson, J.: Albert, an intermediate smart-contract language for the Tezos blockchain. In: Proceedings of the 4^{th} Workshop on Trusted Smart Contracts (to be published) (2020)
10. Claret, G.: Program in Coq. Theses, Université Paris Diderot - Paris 7 (Sep 2018), `https://hal.inria.fr/tel-01890983`
11. 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
12. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional Computing Series, Pearson Education (1994)
13. Glondu, S.: Extraction certifiée dans coq-en-coq. In: Schmitt, A. (ed.) JFLA 2009, Vingtièmes Journées Francophones des Langages Applicatifs, Saint Quentin sur Isère, France, January 31 - February 3, 2009. Proceedings. Studia Informatica Universalis, vol. 7.2, pp. 383–410 (2009)

14. Glondu, S.: Vers une certification de l'extraction de Coq. (Towards certification of the extraction of Coq). Ph.D. thesis, Paris Diderot University, France (2012), https://tel.archives-ouvertes.fr/tel-01798332
15. Goodman, L.M.: Tezos: A self-amending crypto-ledger. position paper. https://tinyurl.com/tezospp (2014)
16. Goodman, L.M.: Tezos: A self-amending crypto-ledger. white paper. https://tinyurl.com/tezoswp (2014)
17. Labs, C.: Juvix: a more elegant language for a more civilized age. https://github.com/cryptiumlabs/juvix, accessed: 2020-05-31
18. Letouzey, P.: Programmation fonctionnelle certifiée – L'extraction de programmes dans l'assistant Coq. Ph.D. thesis, Université Paris-Sud (Jul 2004)
19. Letouzey, P.: A New Extraction for Coq. In: Geuvers, H., Wiedijk, F. (eds.) Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002. Lecture Notes in Computer Science, vol. 2646. Springer-Verlag (2003)
20. Paulin-Mohring, C.: Extracting F_ω's programs from proofs in the Calculus of Constructions. In: Sixteenth Annual ACM Symposium on Principles of Prog ramming Languages. ACM, Austin (Jan 1989)
21. Paulin-Mohring, C.: Extraction de programmes dans le Calcul des Constructions. Thèse d'université, Paris 7 (Jan 1989), http://www.lri.fr/~paulin/PUBLIS/these.ps.gz
22. Sergey, I., Kumar, A., Hobor, A.: Scilla: a smart contract intermediate-level language. CoRR **abs/1801.00687** (2018), http://arxiv.org/abs/1801.00687
23. 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
24. Sewell, P., Nardelli, F.Z., Owens, S.: Ott. https://github.com/ott-lang/ott
25. Xi, H., Chen, C., Chen, G.: Guarded recursive datatype constructors. SIGPLAN Not. **38**(1), 224–235 (Jan 2003). https://doi.org/10.1145/640128.604150, https://doi.org/10.1145/640128.604150