FORMAL LAND

VERIFICATION OF INTERNAL ERRORS

Prepared For :
The Tezos Foundation

September - October
2022

ABOUT

At Formal Land, we do formal verification on the OCaml implementation of the protocol of Tezos. We proceed by translating the code to the proof system Coq using the translator coq-of-ocaml. We then write specifications and proofs on the generated Coq code.

We host the result of our work on the website coq-tezos-of-ocaml.

INTRODUCTION

Let us first say a few words about the work we have done on the verification of the absence of internal errors in the protocol of Tezos. This work was done in September and October 2022. The goal is to formally verify that for any protocol inputs, the errors classified as “unexpected” can never occur. The only errors that should occur are related to mistakes in the user inputs, and should be properly reported to the user.

The verification of internal errors we covered in the last two months can be estimated at around half of the protocol code depending on "alpha_context.ml". For that, we fully axiomatized the storage system ("storage.ml" and "*_storage.ml" files). Once we have actually verified the storage files, we will probably discover new invariants to check, and that will propagate in the proofs of the project. We have already verified most of the "*_repr.ml" files, or completed the lemmas if needed.

While verifying the absence of internal errors, we also have to verify the preservation of the invariants of the data structures of the protocol. This intermediate step is necessary to show that some internal errors cannot be reached. This also helps making sure that the protocol code is sound, and documents the data invariants. We plan to backport these invariants as comments in the protocol code, as suggested by some OCaml developers.

We have not found critical errors in the code. However, we found a few potential integer overflows that we reported to the OCaml developers. These overflows would require further analysis to know if they are reachable, or changes in the protocol code to make sure they cannot happen.

For the parts we have verified, our proofs show that the functions are indeed correctly preserving the invariants and cannot raise internal errors. As the protocol evolves, our formalization will help making sure that no existing properties are broken by new code. The current proofs are also the basic structure to propagate the invariants on the storage once we have a complete verification of it.

We will present some salient points of our methodology to track the internal errors, and describe the main files we have verified.

Protocol version: Unless specified otherwise, we wrote our proofs for the protocol "proto_alpha" translated to Coq at the date of 2022-09-08. As a reference, the folder for the protocol L was created the 2022-10-08 with the Merge request 6419. Our next task will be to adapt these proofs to the current version of the protocol L and "proto_alpha".

METHODOLOGY

We proceeded in three steps:

1. Classification of the errors
2. Handling of assert and exceptions
3. Formal proofs in Coq

1) CLASSIFICATION OF THE ERRORS

We manually classified the protocol errors in our previous grant, and made the following reports:

- asserts report
- exceptions report
- error monad report

In our analysis, most of the errors from the error monad are user errors rather than internal errors. The exceptions and assert are almost always internal errors (there are a few cases of exceptions with an error handler). For our current verification effort, we focus on:

- the assert errors,
- the error "Storage_error" of the error monad, corresponding to unexpected errors in the storage file storage.ml (non-existing keys, unexpected encoding errors),
- a few other errors from the error monad, local to certain files.

The full list of internal errors that we handled is given in the variable Error.internal_errors. There is still discussion among the OCaml developers about which errors are considered internal and which should be considered as user-induced. The list "Error.internal_errors" might thus be completed over time. The proofs we have already written should stay the same except at the points where the newly added internal errors are raised, which should be captured by refining the validity predicates which are pivotal in our code (see below).

2) HANDLING OF ASSERT AND EXCEPTIONS

To verify the assert and exceptions we proceed in two ways, depending on the part of the code that we cover:

- For the Michelson part (the "script_*.ml" files), we rely on the fact that we have written dependent simulations for most of the code (for the interpreter and type-checker / parser). In our translation from OCaml to Coq, we represent exceptions and assert instructions as axioms. Since our dependent simulations do not use the primitives assert and the exceptions, the only way we can write equivalent simulations is if the errors are unreachable. We have defined simulations for most of "script_interpreter.ml" and "script_ir_translator.ml" that are the two main files for Michelson. These simulations are proven correct for most of the interpreter and a part of the translator. These proofs and definitions are for the version K of the protocol.

- For the rest of the code, we translated as many "assert"s as possible to the error monad. We made this modification by hand and maintain this change in our fork of the protocol https://gitlab.com/formal-land/tezos/-/merge_requests/7. All the changes are in the two commits authored by Andrey Klaus on this fork. We counted 93 asserts in the protocol version we are working on, and 35 in our forked version. The main remaining "assert"s, that we have not eliminated yet, are in the data-encodings or deeply nested in some algorithms such as the Deque in "sc_rollup_arith.ml". We have not yet done a monadic translation on the exceptions, which appear to be less frequent than the assert.

In the following, we will focus on the verification of errors that are in the error monad of the protocol.

3) FORMAL PROOFS IN COQ

Let us describe how we specify and verify that the protocol code in the error monad does not contain internal errors.

Soundness predicate and validity conditions

We use the following predicate to express that an expression "e" of type "M? a" (returning a value of type "a" in the error monad "M?") cannot raise an internal error, and returns a value satisfying the predicate "P":

```
letP? x := e in P x
```

Actually, "letP?" is a monadic notation for "bind_prop" defined as:

```
Definition bind_prop {a : Set} (e : M? a)
  (P : a → Prop) : Prop :=
  match e with
  | Pervasives.Ok x ⇒ P x
  | Pervasives.Error err ⇒ Error.not_internal err
  end.
```

This definition specifies that:

- when the expression "e" is successful, it satisfies the predicate "P" ;
- when "e" raises a list of errors "err" (usually composed of a single error), no one is classified as internal.

In practice, the verification of a function "f" articulates the "letP?" both with validity pre- and post-conditions on the inputs and outputs of "f" as follows:

```
Lemma f_is_valid : (* in mock-code *)
  forall (x1 ... xn), Valid1 x1 /\ ... /\ Validn xn
  -> letP? (y1,...,yp) := f x1 ... xn in
     Valid1' y1 /\ ... /\ Validp' yp
```

Intuitively, the above lemma specifies that, when the inputs of "f" are valid, then if the computation of "f" succeeds, it ouputs a valid value and if it does not, it does not cause any internal error.

We have specified and proved such lemmas for all the functions in "main.ml". This expresses the fact that the calls to the functions in "main.ml" cannot return an internal error. These proofs are complete for the file "main.ml", but depend on other files which can have admitted proofs. We will detail in the Results section below the files we have worked on.

Example

An example of proof on a simple function is the following. For the function named "ticket_diffs_of_lazy_storage_diff" from the file "ticket_accounting.ml" (here in the Coq version as automatically produced by coq-of-ocaml):

```
Definition ticket_diffs_of_lazy_storage_diff
  (ctxt : Alpha_context.context)
  (storage_type_has_tickets : Ticket_scanner.has_tickets)
  (lazy_storage_diff : list Alpha_context.Lazy_storage.diffs_item)
  : M? (Ticket_token_map.t Z.t × Alpha_context.context) :=
  if Ticket_scanner.has_tickets_value storage_type_has_tickets then
    let? '(diffs, ctxt) :=
      Ticket_lazy_storage_diff.ticket_diffs_of_lazy_storage_diff ctxt
        lazy_storage_diff in
    Ticket_token_map.of_list_with_merge ctxt diffs
  else
    return? (Ticket_token_map.empty, ctxt).
```

we write the following Coq specification:

```
Lemma ticket_diffs_of_lazy_storage_diff_is_valid
  ctxt storage_type_has_tickets lazy_storage_diff :
  Raw_context.Valid.t ctxt →
  List.Forall Lazy_storage_diff.diffs_item.Valid.t lazy_storage_diff →
  letP? '(map, ctxt) :=
    Ticket_accounting.ticket_diffs_of_lazy_storage_diff
      ctxt storage_type_has_tickets lazy_storage_diff in
  Ticket_token_map.Valid.t (fun _ ⇒ True) map ∧
  Raw_context.Valid.t ctxt.
```

stating that given input values, namely "ctxt", "storage_type_has_tickets" and "lazy_storage_diff", which are valid, the function "ticket_diffs_of_lazy_storage_diff":

- does not return an internal error, and
- returns two values "map" and "ctxt" satisfying their corresponding validity predicates.

The proof is as follows:

```
Proof.
  intros H_ctxt H_lazy_storage_diff.
  unfold Ticket_accounting.ticket_diffs_of_lazy_storage_diff.
  destruct Ticket_scanner.has_tickets_value; simpl.
  { eapply Error.split_letP. {
      now apply
        ticket_diffs_of_lazy_storage_diff_is_valid.
    }
    clear ctxt H_ctxt; intros [diffs ctxt] [H_diffs H_ctxt].
    now apply Ticket_token_map.of_list_with_merge_is_valid.
  }
  { split; trivial.
    apply Carbonated_map.Make.empty_is_valid.
  }
Qed.
```

On line 4, we reason by cases over the if. On line 5 we split the reasoning over the error monad binder "let?" into two parts. With the tactic "apply" we call the validity lemma of the three external functions we are referring to:

- "Ticket_lazy_storage_diff.ticket_diffs_of_lazy_storage_diff"
- "Ticket_token_map.of_list_with_merge"
- "Ticket_token_map.empty"

RESULTS

We are writing the proofs of the absence of internal errors for each protocol file. We axiomatized, for now, the validity of all the storage-related files, such as "storage.ml" and the "*_storage.ml" files.

For a quick overview of the files that we still need to review, specify or verify, there is our milestone page listing all the files of the protocol that we have/have not checked. Some of the files in “TODO” are already partially verified (like the "script_*" files considering the Michelson simulations). Some of the files in “Done” are specified but not verified, like the storage files.

We have verified:

- All of the "main.ml" file, most of "validate.ml" and half of "apply.ml". These proofs can be found in "Main.v", "Validate.v", and "Apply.v". We also verified the largest parts of related files such as "amendment.ml" or "baking.ml".
- Almost all of the "ticket_*.ml" files. An example of proof file is "Ticket_accounting.v".

- A part of the "sc_rollup*.ml" and "tx_rollup*.ml" files. An example is "Tx_rollup_l2_context.v", verifying that the two internal errors named "Key_cannot_be_serialized" and "Value_cannot_be_serialized" cannot be reached.
- The file "dal_apply.ml" in "Dal_apply.v"
- All the "zk_*.ml" files. An example of proof file is given with "Zk_rollup_apply.v".
- All the "*_repr.ml files". These files were already mostly completely verified, including for the absence of unexpected errors, during previous grants. However, we continued to extend the proofs, in particular for the new protocol files.

For the storage, we use a specification by simulation. We simulate the storage by a record containing all the sub-stores (there are around one hundred sub-stores). As this part is long, verbose, and error-prone, we used a set of Ruby scripts to generate the simulation and its validity lemmas (admitted for now). These scripts are in "scripts/alpha/templates/storage".

The outputs of these scripts are in:

- "Context_generated.v" for the definition of the simulation,
- "Storage_generated.v" for the equality between this simulation and the code in storage.ml

Then we specified all the "*_storage.ml" files by hand (36 files). An example is "Sc_rollup_stake_storage.v".

Our next target will be to verify the storage system. We think that it will make the specifications of the protocol files more complex, as new conditions will appear, such as the existence of a certain key in a certain part of the store. But as this code part is complex, it is also crucial to get the correct specification.

COMMUNICATION

We made a presentation at the proto-call meeting on Tuesday, October 11th, to present our work to verify the protocol. We now use the public Slack channel "#coq-tezos-of-ocaml" to post the changelog of our work on a weekly basis.

We published three blog posts:

- Verifying the skip-list: verification of the skip-list data-structure, a work that we have done during this summer;
- Skip-list verification. Using inductive predicates: continuation of the previous post on the verification of skip-list;
- Absence of internal errors: a general introduction to the work on internal errors.

CONCLUSION

We are continuing to verify the internal errors for the remaining protocol files. We think that these proofs are useful as they cover a large part of the code base, and check for the propagation of the invariants of the data structures. We thank the Tezos Foundation to have allowed us to work on this verification effort.


THANKS!