Preliminaries for our Compiler

January 25, 2021

Before we're able to work on our compiler, it's necessary to get things straight concerning what the end result should look like in detail and which approach provides the best means to get there efficiently. I thought it's the best idea to take a look at the whole pipeline from parsing until the compilation result first in order to get an overview of what the process steps and their intermediate results are. A crucial aspect of the compilation, IMO, is what approach we want to choose to connect the parent SC and its assertion contract. This document is an attempt at playing out different possibilities of doing this and assessing the impact this design decision has on the whole pipeline and it's requirements to our "toolchain" (esp. the compiler).

1 Type check

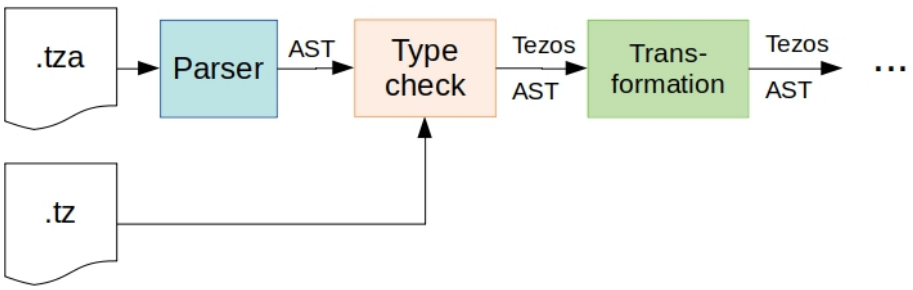

Figure 1: Type check placement in the pipeline

- Tasks:
  - Check for unsupported expressions & types; transforms to Tezos-specific AST
  - Replace the wildcards with actual types
  - Type match the assertion contract types with parent contract types

- How to extract type from parent contract, s.t. we can compare them easily?
  - Using Tezos parsing libraries to retrieve type of parent `->` not comparable on the fly
  - Write own parser who returns the same AST type `->` comparable
    - Handle entrypoints given w/ name and w/o name
  - Include Michelson type syntax to existing parser and require copy of parent type in assertion contract.
    - our syntax includes each type anyway - counter-argument from earlier not valid anymore
    - ~~Not compatible with Ethereum~~
    - ~~Using a universal syntax to state parent type again introduces possibility for errors and therefore type mismatches~~
    - ~~Extracting type from parent directly is therefore better IMO~~
- Position in pipeline: before transformation to avoid unnecessary work
  - Requires adaption of current transformation code

2 Assembly

After compilation, the parent and assertion contract somehow need to be "connected" to each other - let's call that stage the assembly. We have 2 different approaches, that we still need to decide on:

a) **Assemble 2 separate contracts** - Both parent and assertion contract are originated separately on the blockchain and the latter one needs to be called for assertion checking.

b) **Assemble a single, monolithic contract** - Both are merged into a single contract which contains the smart contract and the assertion code

As the assembly strategy will have an impact on the compilation process and result in different generation pipelines, especially for the Liquidity approach, we should take a closer look on both strategies and how they would be implemented in more detail.

2.1 Assembly strategy 1: Separate contracts

- Contract type of assertion contract needs to be the same
  - Allows to simply pass on the parameter as is
  - Unused eps return "unit"
  - restriction may be lifted if ep names are guaranteed to exists and are equivalent

– also depends on how we handle "normal" eps

IF_LEFT {code for used EP} {UNIT; NIL operation; PAIR}

- The address of the assertion contract needs to be provided in some way

  a) Parent storage contains the address of its assertion contract
     * Parent contract needs to be modified (storage declaration, every storage access & return value)
     * Assertion contract must be originated first, so that address can be passed into initial storage of parent
     * Baker needs to retrieve address from contract storage (no obvious standard way to do this - storages can look differently)
       -> BigMap?

  b) Address of the assertion contract is a parameter of the transaction
     * Address must be known to the caller
     * Decouples parent and assertion contract ~~(if calling assertion from parent is not necessary)~~
     * Safety issue: can be passed a contract with correct type that never returns a counterexample
     * Parent contract doesn't need to be touched

The interactions between baker, validators and the contracts are sketched in figures 2 and 3.

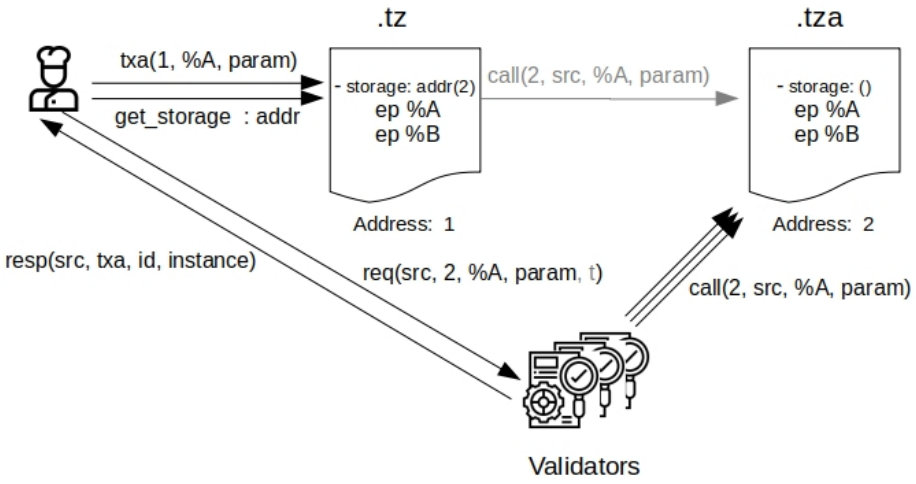

Figure 2: Interaction with two separate contracts v1: storing the address of the assertion contract in the parent

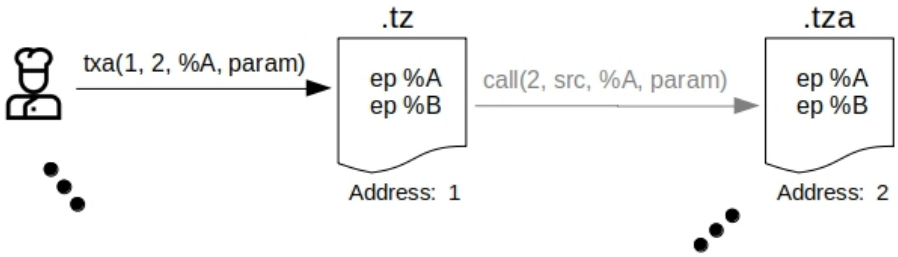

Figure 3: Interaction with two separate contracts v2: passing the address of the assertion contract in the transaction

2.2 Assembly strategy 2: Monolithic contract

- Assertions are appended to the parent as separate entry points
- Parent contract has to be modified (parameter type & addendum)
- Problems/Drawbacks:
  - entrypoint names necessary - otherwise no unambiguous assignment
  - parameter types of original ep & assertion ep are different
  - e.g. %A and %A_a cannot be called with the same parameter (if not called explicitly; ref. 2 line 24, 25.)
  - Possible solution:
    - all eps in the parent and assertion contract need to have a %name -> calling both eps explicitly allows using the same parameter (ASAIK) ; ref. 1 line 24, 25.
    - Assign dummy names if none are given - effort?
    - what to do with calls to %default?
      - Baker would need to
      - 1) look up entrypoints of contract
      - 2) check which type matches & find out the ep name
      - 3) derive assertion ep name from that
    - how to assign entry points given without names in .tza to their respective parent eps? 2 line 10, 11.
- EP names of the assertion code need to have a fixed pattern, s.t. the bakers know how to call them
- Similarly to strategy 1, there probably needs to be "empty" assertion code for normal eps

3:

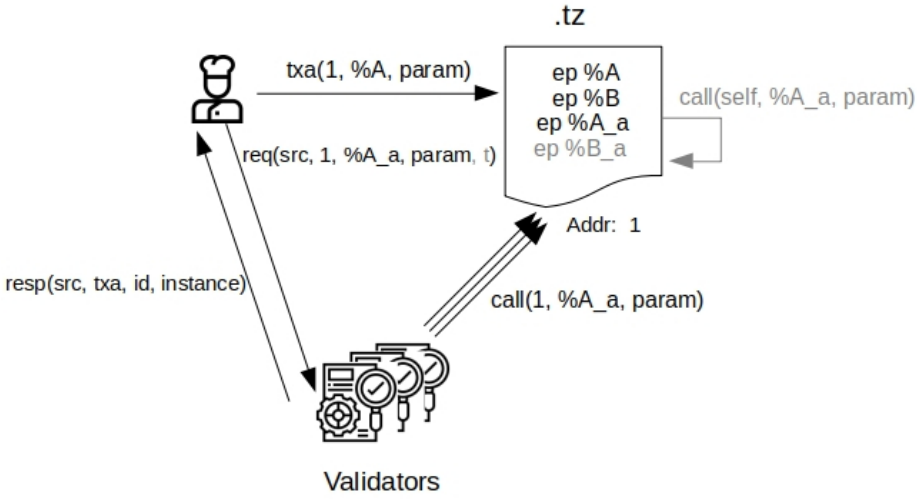

Figure 4: Interaction with a monolithic contract

Listing 1: Monolithic assembly w/ ep names

```
1  (* example.tz *)
2  parameter (or (int %A) (list %B int));
3  code {
4  ...
5  IF_LEFT { ... } (* %A*)
6          { ... } (* %B *)
7  }
8
9  (* example.tza *)
10 entrypoint %A (i: int)
11 entrypoint %B (l : (list int)
12
13 (* example_monolithic.tz *)
14 parameter (or (int %A) (or (list %B int) (or (int %A_a) (list %B_a int))));
15 code {
16 ...
17 IF_LEFT { ... } (* %A *)
18         { IF_LEFT { ... } (* %B *)
19                   { IF_LEFT { ... } (* %A_a *)
20                             { ... } (* %B_a *)
21 }}}
22
23 (* calling the eps and their assertions *)
24 ./tezos-client call example from xy --entrypoint %A --arg "1"
25 ./tezos-client call example from xy --entrypoint %A_a --arg "1"
```

Listing 2: Monolithic assembly w/o ep names

```
1  (* example.tz *)
2  parameter (or int (list int));
3  code {
4  ...
5  IF_LEFT { ... } (* %A *)
6          { ... } (* %B *)
7  }
8
9  (* example.tza *)
10 entrypoint (left (i: int))          (* %A *)
11 entrypoint (right (l : (list int)) (* %B *)
12
13 (* example_monolithic.tz *)
14 parameter (or int (or (list int) (or int (list int))));
15 code {
16 ...
17 IF_LEFT { ... } (* %A *)
18         { IF_LEFT { ... } (* %B *)
19                   { IF_LEFT { ... } (* %A_a *)
20                             { ... } (* %B_a *)
21 }}}
22
23 (* calling the eps and their assertions *)
24 ./tezos-client call example from xy --arg "(left 1)" (* %A *)
25 ./tezos-client call example from xy --arg "(right (right (left 1)))" (* A_a *)
```

3 Compiler

There are two different possibilities to compile the assertion contract to Michelson: compiling directly to Michelson or compiling to Liquidity as an intermediate representation. The advantages and disadvantages, as well as the respective pipelines for both approaches are discussed in the following.

3.1 Compilation directly to Michelson

Writing an own compiler from the Tezos-AST to Michelson directly. Some general aspects/thoughts about this approach are:

- Compilation is complex
  - no routine/experience in Michelson
  - increased effort to obtain optimized/most efficient code
- Complex testing `->` from Michelson code it's not obvious to see if it does the right thing

- We only have to extend Michelson to include our new instructions
- Implementation in a single "all-in-one" pipeline from parsing to assembly
- After compilation the Michelson code needs to be type checked -> Tezos protocol libraries are needed for this, which are protocol dependent; our code thus needs proper versioning.

3.1.1 Assembly strategy 1

- Contract type can be taken from original contract (it must be the same?); storage is simply ()
- Add empty code for the unused entry points
- Parent contract has to be modified as well:
  - Parent contract needs to be parsed -> Tezos libraries
    - AST can be manipulated (location of each subsequent node must be increased though -> no purely local changes)
    - storage/parameter manipulation relatively easy; adapting code to changed storage hard
  - Address of .tza in storage
    - Add address into storage canonically -> Bakers need to be able to extract it in a standard way
    - Possible idea: Wrap original storage inside bigmap/map
      - Add boilerplate code at beginning of contract to extract original storage -> keeps parent code valid for large parts
      - Return values need to be adapted
- Alternative: Enforcing the necessary storage structure
  - Expect e.g. a structure like `storage (pair address _);`
  - Parse parent storage type and reject storages which do not match this type
  - Easy to implement; more responsibility to the devs
  - Removes any need to consider the parent contract in the compilation process -> move to the type check stage of the pipeline

3.1.2 Assembly strategy 2

- check if eps are present in type check stage
- parameter type must be generated with unions
- parent code for the original eps must be adapted (cf. lines 5-6 & 17-18 listing 1)
- rename assertion entry points

3.1.3 The pipeline

The pipeline for this approach looks as follows:

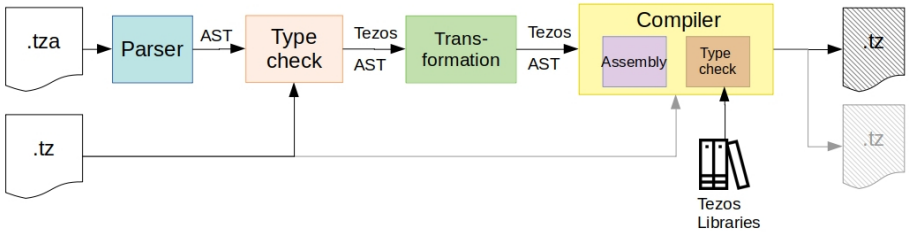

Figure 5: The pipeline for generating smart contracts with distributed assertion checking using direct compilation to Michelson

3.2 Compilation to Liquidity

- Compilation target = Liquidity instead of Michelson
- Use Liquidity's compiler to generate Michelson code
  - optimizations
  - type-checking
- Liquidity syntax is OCaml-like `->` compilation less complex
- Liquidity must be extended for new instructions as well
  - Adds an additional component which needs to be kept up to date with the newest protocols
  - Fork code
  - extend compiler (and decompiler)
  - "template" on how to add instructions : `https://github.com/OCamlPro/liquidity/commit/c8666106c9e57f397bfbf8c6f72857e09c5a11c2`

Provided there is a feasible way to import the Liquidity code into our code base, the pipeline would look as shown in fig. 6. All extra steps explained in 3.2.3, 3.2.4 and shown in figs. 7, 8 would then be integrated in "our" compilation stage (yellow).

3.2.1 Liquidity

- High-level language for programming Smart Contracts for Tezos
- Full coverage of Michelson
  - Last release June 2020

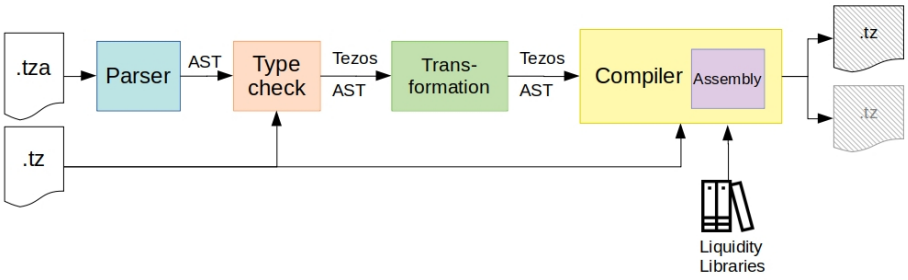

Figure 6: The generation pipeline importing Liquidity libraries/code in our code base

  - Not clear from docs which protocols are supported
  - Michelson instructions added in Edo are not yet included

- Uses local variables, no stack manipulation!
- Efficient & optimizing compiler
- Also provides a decompiler Michelson `->` Liquidity
- Includes type checking!
- AFAIK: not formally verified
- Integration into the pipeline:
  - Liquidity uses ocpbuild & OCaml 4.07. (we currently use dune & OCaml 4.09)
  - a) Integrate fully into pipeline, i.e. use their libs/modules (adaption of build systems necessary); shown in fig. **??**
  - b) Use it as a standalone tool; shown in figs. **??**, **??**

3.2.2 Example

Consider the following transformed assertion contract:

Listing 3: A transformed assertion in the assertion syntax

```
(entrypoint %ep1 (l: (list int))
  (exists (i: int)
    (if (lt i (size l))
        (assert (ge (nth l i) 10)))))
```

If the compilation target were Liquidity, the compilation result would look as follows:

Listing 4: A transformed assertion in the assertion syntax

```
let%entry ep1 (l : int list) () =
  let size = List.length l in
  let i = Int.random(0, size) in
  if l.(i) >= 10 then failwith "assertion failed";
  ([], ())
```

3.2.3 Assembly strategy 1

- Liquidity builds contract type from the order of given entry points

```
let%entry A (i: int) = ...
let%entry B (b: bool) = ...
->
parameter (or int bool);
```

- The order of eps in the .liq file must match the parent contract
- Liquidity allows only 1 default entry point declaration; if assertion contract doesn't define names
  a) they need to be merged into the default entry points using `match-cases`
  b) keep them separate and assign dummy names; BUT types need to be stripped down (e.g. strip `left int` to `int`)

Listing 5: Liquidity: handle all eps in the default ep

```
type storage = unit

let%entry default (input: (int, bool) variant) _ =
   begin match input with
   | Left i -> if i > 10 then failwith ()
   | Right b -> if b then failwith ()
   end;
   ([], ())
```

- The parent contract would need to be modified in the same way as described in 3.1.1.

Fig. 7 shows the pipeline for this approach if Liquidity is used as a standalone tool. If the libraries can be imported in our code base, the required steps would be the same - however they would be part of the yellow compilation stage.

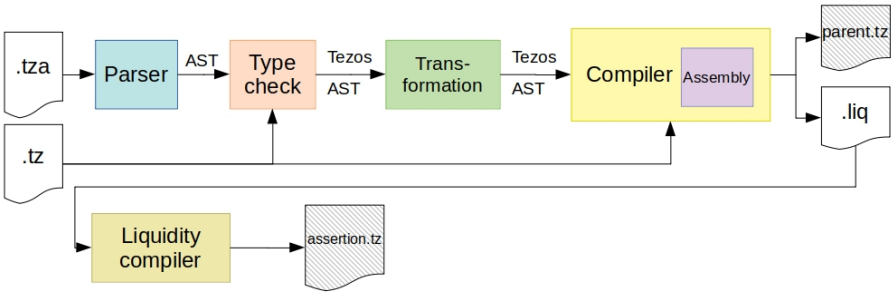

Figure 7: The generation pipeline for assembly strategy 1 and Liquidity as a standalone tool

3.2.4 Assembly strategy 2

- Contracts need to be merged in one .liq file `->` decompilation of parent contract necessary
  - The decompiler of Liquidity needs to be extended as well as the compiler
- Append the assertion entrypoints to the decompiled parent `->` rename assertion ep names canonically
- Compile merged .liq file to .tz
- Parent code doesn't need to be modified; Liquidity compiler takes care of that

Fig. 8 shows the pipeline for this approach if Liquidity is used as a standalone tool. It would require a 3-step workflow until the contracts are generated. If the libraries can be imported in our code base, the required steps would be the same - however they would be part of the yellow compilation stage.

3.3 Albert

- Another intermediate high-level contract language
- Formally verified with Coq
- No decompiler available
- Uses Make as build tool
- Further research necessary
- References:
  - Website: `https://albert-lang.io/`
  - Publication: `https://arxiv.org/abs/2001.02630`

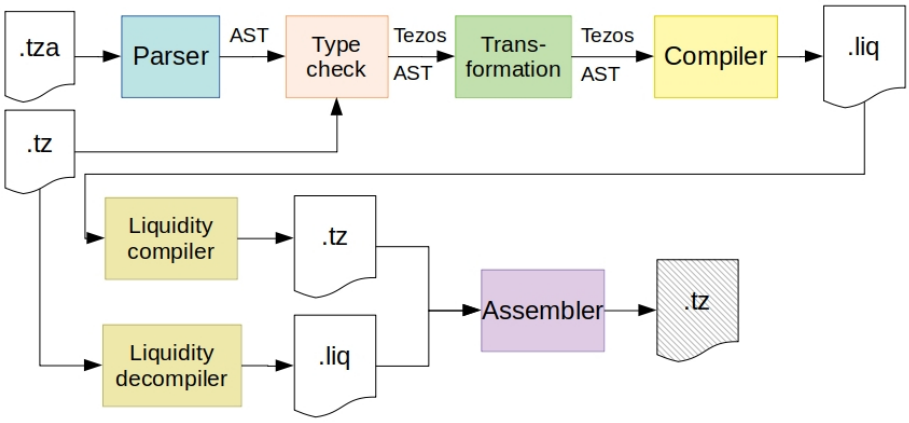

Figure 8: The generation pipeline for assembly strategy 2 and Liquidity as a standalone tool

3.4 Instructions

- If we use Liquidity as IR, the instruction have to be added to their compiler/parser as well
- Add instruction to Michelson
  - Fork Tezos code
  - Add to tokens
  - Implement evaluation → full OCaml functionality
  - Gas costs?
  - "template" on how to add instructions here: `https://gitlab.com/tezos/tezos/-/commit/01c321d751b9843359625fb4ee663e2b9857eaa7`
- Other instr. which might be necessary:
  - Internal call to assertion? (although I think a normal call should be fine)
  - Assertion fail
    - SC have no real return value
    - Is it possible to return something usable with the standard fail-with?
    - General question: how can we represent a counterexample/proof and how can we verify them? (as of now, there is no way to execute the assertion with a set of fixed variables)
    - Validators must somehow store the random values they generated (?)

3.4.1 Nth

- Data type: list
- For bytes & string use `SLICE`
- Typing cf. listing 6
- Implementation draft cf. listing 7

Listing 6: Type of the nth instruction

```
:: (list 'a) : nat : 'A  ->  option 'a : 'A

> NTH / l : index : S  =>  Some 'a: S
      iff index is within bounds
> NTH / l : index : S  =>  None 'a: S
      iff index is out of  bounds
```

Listing 7: Implementation draft of nth in Michelson interpreter

```
| (Nth ({elements = []; _}, (_, rest))) ->
  logged_return ((None, rest), ctxt)
| (Nth ({elements = elts; length}, (index, rest))) ->
  let l_length = Z.of_int length in
  if Compare.Z.(index <= l_length) then
    logged_return ( (Some (List.nth elts index), rest), ctxt )
  else logged_return ((None, rest), ctxt)
```

3.4.2 Random

- Data type: int (bool, nat, ... ?)
- Typing cf. listing 8
- Implementation draft cf. listing 9

Listing 8: Type of the rand instruction

```
:: (list 'a) : nat : 'A  ->  option 'a : 'A

> NTH / l : index : S  =>  Some 'a: S
      iff index is within bounds
> NTH / l : index : S  =>  None 'a: S
      iff index is out of  bounds
```

Listing 9: Implementation draft of rand in Michelson interpreter

```
| (Rand (offset, (length, rest))) ->
   Random.self_init;
   let ri = offset + Random.int(length) in
   logged_return (ri, rest), ctxt )
```

4 Further stuff

Here I collected all open questions I encountered when putting this together, but didn't follow up on:

- How do we differentiate between transactions to "normal" SC and SCs with assertions?
  - Intrinsic: the SC knows about it and triggers something (is this even possible? we can't do much from within a SC)
  - Extrinsic: the caller knows it, e.g. uses different transaction types (e.g. tx and txa)
    - SC containing normal eps and eps with assertions cause trouble:
    - what about txa which call a "normal" ep? –¿ empty assertions; unnecessary work
    - what about tx which call an ep with an associated assertion? That would dodge the assertion checking
- How to represent a counterexample/proof? How do we store the random numbers when they're generated during interpreting the contract?