Summary

This document defines patterns that make it easier for developers to build smart contracts in Michelson.

Abstract

Michelson is a low-level typed stack-based functional programming language, which is an unusual combination of features, to put it mildly. While Michelson is surprisingly expressive for a language of its simplicity, it lacks many generally typical programming language features, such as variables or references. This leads to developer discomfort, limits contract complexity and inhibits interoperability of contracts written by different individuals or teams. The problem of interoperation is particularly exacerbated with regards to higher level languages that generate or compile to Michelson, since the Michelson code produced by two different compilers or generators can have radically different structure.

Fortunately, Michelson's expressivity allows us to define common conventions, abstractions and best practices which we hope will mitigate these challenges. We hope these patterns will prove informative and useful to Michelson developers, as well as developers of languages and tools which target Michelson.

Patterns defined in this document:

- Entrypoints
- `Pair`s and `Or`s syntax sugar
- CASE macro
- View Entrypoints
- Void Entrypoints

Entrypoints

Often a single contract is supposed to provide multiple functions, or as we will say, contain several *entrypoints*. At the time of writing, Michelson does not explicitly support entrypoints in Mainnet.

However, the recent update formalizes entrypoints and facilitates work with them (see also the Michelson tutorial for Zeronet). In short, the contract parameter should be a tree of `Or`s, there each leaf of this tree stands for an argument of some entrypoint. Leaves are annotated with entrypoint names. When a value of `contract t` type refers to a contract with a correctly defined parameter, type `t` stands for the argument of some of its entrypoints. `CONTRACT` instruction used to construct `contract t` value now can be annotated with the name of particular entrypoint.

For now, when this feature is not yet available in Mainnet, one can still use a similar approach to encoding entrypoints in contract parameter. When calling such a contract, one will have to wrap passed argument with a sequence of `Left`s and `Right`s instructions required for this particular entrypoint.

This standard does not define a conventional structure that `or`-trees must follow. Examples of compatible tree structure include:

- Right-hand comb:`(or a (or b (or c (or d e))))`
- Left-hand comb:`(or (or (or (or a b) c) d) e)`
- Right-hand balanced tree: e.g `(or (or a b) (or c (or d e)))`
- Left-hand balanced tree: e.g `(or (or (or a b) c) (or d e))`

All such structures are `A1` compliant, so long as all entrypoints are indicated with field annotations, and all top-level union type arguments are unannotated.

Nevertheless, it is advisable that specific implementations adhere to a consistent tree-struct convention, examples of which can be seen in various extensions of this standard (such as `TZIP-006`).

Example

Consider the following simple Michelson contract that implements a counter:

```
parameter
  (or
    unit %bump_counter
    nat  %reset_counter
  );
storage nat;
code {UNPAIR;
      IF_LEFT
        {DROP; PUSH nat 1; ADD; NIL operation; PAIR;} # %bump_counter
        {DIP {DROP}; NIL operation; PAIR;}            # %reset_counter
     };
```

This contract exports two entrypoints: `%bump_counter` and `%reset_counter`, which add one to the `nat` in storage, or replace the `nat` with a new `nat`.

Notice that after we enter the `IF_LEFT` in the `code` block, the two branches of the contract are completely logically separate from one another.

The sequence of instructions for `%bump_counter` has type

```
{DROP; PUSH nat 1; ADD; NIL operation; PAIR;}

 :: unit : nat : [] -> pair (list operation) nat : []
```

The sequence of instructions for `%reset_counter` has type

```
{DIP {DROP}; NIL operation; PAIR;}

  :: nat : nat : [] -> pair (list operation) nat : []
```

In a sense, each entrypoint almost acts like it's its own contract. We can imagine a contract that implements only `%bump_counter`:

```
parameter unit;
storage nat;
code { UNPAIR; DROP; PUSH nat 1; ADD; NIL operation; PAIR;}
```

and a contract that implements only `%reset_counter`:

```
parameter nat;
storage nat;
code {UNPAIR; DIP {DROP}; NIL operation; PAIR;}
```

Our entrypoint pattern allows us to combine both these separate contracts into a single contract with a two "methods" that operate on a common storage.

If we now want to add a third method to our counter contract called `%antibump_counter`, we can extend the union by nesting another `or` type constructor:

```
parameter
 (or
    unit   %antibump_counter
    (or
      unit %bump_counter
      nat  %reset_counter
    )
  );
storage nat;
code {UNPAIR;
      IF_LEFT
        {DROP; PUSH nat 1; SUB; ISNAT; ASSERT_SOME;
         NIL operation; PAIR
        }                                              # %antibump_counter
        {IF_LEFT
          {DROP; PUSH nat 1; ADD; NIL operation; PAIR;} # %bump_counter
          {DIP {DROP}; NIL operation; PAIR;}            # %reset_counter
        };
     };
```

This pattern can be extended further to `n` entrypoints by adding additional `or` types and `IF_LEFT` instructions.

`Pair`s and `Or`s syntax sugar

To increase syntactic clarity, we propose a sugar for `pair` and `or` types according to the following reduction rules:

```
(a, b)                            ~> (pair a b)
(a, b) :t %f                      ~> (pair :t %f a b)
(a %foo, b %bar)                  ~> (pair (a %foo) (b %bar))

(a | b)                           ~> (or a b)
(a | b) :t %f                     ~> (or :t %f a b)
(a %foo | b %bar)                 ~> (or (a %foo) (b %bar))
```

Types written in this syntax sugar should relax any Michelson implementations indentation and whitespacing rules.

This standard defines only tuples and unions of size two. Introduced syntax sugar becomes especially useful in extensions of the standard where this syntax is generalized (see `TZIP-006` for example).

CASE macro

We propose a new multiary macro called `CASE`:

```
CASE a b (\rest) / S => IF_LEFT a {CASE b (\rest)} / S
CASE a b / S => IF_LEFT a b / S
```

This enables the code in the counter contract to be rewritten:

```
code {UNPAIR;
       CASE
         {DROP; PUSH nat 1; SUB; ISNAT; ASSERT_SOME; NIL operation; PAIR }
         {DROP; PUSH nat 1; ADD; NIL operation; PAIR;}
         {DIP {DROP}; NIL operation; PAIR;}
     };
```

This macro definition works when entrypoints form a right-hand comb in contract parameter, it should be adjusted in case another representation is chosen.

Optionally, the branches in `CASE` may be annotated with field annotations:

```
code {UNPAIR;
       CASE
        %antibump_counter {DROP; PUSH nat 1; SUB; NIL operation; PAIR;}
        %bump_counter     {DROP; PUSH nat 1; ADD; NIL operation; PAIR;}
        %reset_counter    {DIP {DROP}; NIL operation; PAIR;}
     };
```

which should generate a type error if the field annotations don't match.

View Entrypoints

We define the following synonym in a contract's parameter type declaration:

```
view a r = (a, contract r)
```

A `view` is an entrypoint which represents a computation that takes an argument of type `a` and returns a result of type `r`. This return type is represented as a callback. For example

```
parameter
  ( unit          %antibump_counter
  | ( unit          %bump_counter
    | ( nat           %reset_counter
      | view unit nat %getCount
      )
    )
  );
storage nat;
code {UNPAIR;
      CASE
        {DROP; PUSH nat 1; SUB; ISNAT; ASSERT_SOME; NIL operation; PAIR }
        {DROP; PUSH nat 1; ADD; NIL operation; PAIR;}
        {DIP {DROP}; NIL operation; PAIR;}
        {UNPAIR; DIP {AMOUNT; DUUUP}; PAIR;
         TRANFER_TOKENS; NIL operation; SWAP; CONS; PAIR}
     };
```

By convention, `view` must emit only a single transfer `operation` to the callback contract passed by the caller, and must not mutate the contract storage in any way.

Note that using `view` may potentially be insecure: users can invoke operations on an arbitrary callback contract on behalf of the contract that contains a view entrypoint. If you rely on `SENDER` value for authorization, please be sure you understand the security implications or avoid using views.

To make the process of writing the logic for a `view` easier, we define a `VIEW` macro:

```
VIEW :: view a r : storage : S -> (list operation, storage) : S
VIEW code =
  UNPAIR; DIP {DUUP}; PAIR; code; DIP {AMOUNT};
  TRANSFER_TOKENS; NIL operation; SWAP; CONS; PAIR;
  where
    code :: (a, storage) : S -> r : S
```

This macro allows us to rewrite our `%getCount` entrypoint as

```
code {UNPAIR;
      CASE
        {DROP; PUSH nat 1; SUB; ISNAT; ASSERT_SOME; NIL operation; PAIR }
        {DROP; PUSH nat 1; ADD; NIL operation; PAIR;}
        {DIP {DROP}; NIL operation; PAIR;}
        {VIEW {CDR}}
     };
```

Until mentioned multiple entrypoints feature is available, one challenge that arises from our construction of `view` entrypoints is that contracts whose parameters are unions cannot pass themselves as callbacks to contracts with a view whose type is a variant or "arm" of the caller's parameter.

For example, a contract with parameter type `(or nat bool)` will be unable use it's own address to call our counter contract's `%getCount` view, since `contract nat` and `contract (or nat bool)` are different types. While we could simply extend our counter with a `(unit, contract (or nat bool))` this is impractical and breaks our `view` abstraction. Defining `view`s in this way would greatly restrict the genericity of our `view` entrypoints, since a different `view` entrypoint would have to be added for each possible contracts with a parameter `nat` arm.

For now, this issue can be mitigated via using a proxy contract. In our case, this contract should store a reference to our contract `contract (or nat bool)` and have `nat` parameter. Once called, the proxy contract should wrap provided argument with the appropriate sequence of `Left`s and `Right`s instructions and then call the stored contract. Drawbacks of this approach are that a separate proxy contract should be originated for each entrypoint which is expected to be called recursively from our contract. Another one comes from the fact that proxy is used - securely passing the original `sender` from the call to our contract becomes non-trivial.

Void Entrypoints

We define the folling synonym in a contract's parameter type declaration

```
void a b = (a, lambda b b)
```

We also define a `VOID` macro:

```
VOID :: void a b : S -> b : _
VOID code = UNPAIR; SWAP; DIP {code}; SWAP; EXEC;
            PUSH string "VoidResult"; PAIR; FAILWITH;
```

A `void` is an entrypoint that ends in a `FAILWITH ("VoidResult", b)`. By construction they cannot be run on-chain. `void`s are designed to be run locally and may be very gas-expensive. They are included in the contract interface primarily to provide enable trust-less computation on the contract's storage. That is, the presence of the entrypoint on-chain implies that all parties can have certainty that anyone who calls a `void` with identical arguments will compute the same result.

The `lambda b b` in the void definition is intended to be used as a type proxy, and "callers" of void entrypoints should pass the identity function (the value produced by `LAMBDA b b {}`) here. That said, it is possible that there may be utility in some cases to using this parameter as a local continuation.

The object passed to `FAILWITH` is a pair with its first element equal to `"VoidResult"`. Herewith, given error mechanism is made compliant with possible custom errors defined by a contract.

```
parameter
  ( unit            %antibump_counter
  | ( unit            %bump_counter
    | ( nat             %reset_counter
      | ( view unit nat   %getCount
        | void unit bytes %hashCount
        )
      )
    )
  );
storage nat;
code {UNPAIR;
      CASE
        {DROP; PUSH nat 1; SUB; NIL operation; PAIR;}
        {DROP; PUSH nat 1; ADD; NIL operation; PAIR;}
        {DIP {DROP}; NIL operation; PAIR;}
        {CDR; AMOUNT; DUUUP; TRANFER_TOKENS; NIL operation; SWAP; CONS; PAIR}
        {VOID {DROP; PACK; SHA512}
     };
```