Michelson: the language of Smart Contracts in Tezos

The language is stack based, with high level data types and primitives and strict static type checking. Its design cherry picks traits from several language families. Vigilant readers will notice direct references to Forth, Scheme, ML and Cat.

A Michelson program is a series of instructions that are run in sequence: each instruction receives as input the stack resulting of the previous instruction, and rewrites it for the next one. The stack contains both immediate values and heap allocated structures. All values are immutable and garbage collected.

A Michelson program receives as input a single element stack containing an input value and the contents of a storage space. It must return a single element stack containing an output value and the new contents of the storage space. Alternatively, a Michelson program can fail, explicitly using a specific opcode, or because something went wrong that could not be caught by the type system (e.g. division by zero, gas exhaustion).

The types of the input, output and storage are fixed and monomorphic, and the program is typechecked before being introduced into the system. No smart contract execution can fail because an instruction has been executed on a stack of unexpected length or contents.

This specification gives the complete instruction set, type system and semantics of the language. It is meant as a precise reference manual, not an easy introduction. Even though, some examples are provided at the end of the document and can be read first or at the same time as the specification.

Table of contents

I - Semantics

This specification gives a detailed formal semantics of the Michelson language. It explains in a symbolic way the computation performed by the Michelson interpreter on a given program and

initial stack to produce the corresponding resulting stack. The Michelson interpreter is a pure function: it only builds a result stack from the elements of an initial one, without affecting its environment. This semantics is then naturally given in what is called a big step form: a symbolic definition of a recursive reference interpreter. This definition takes the form of a list of rules that cover all the possible inputs of the interpreter (program and stack), and describe the computation of the corresponding resulting stacks.

Rules form and selection

The rules have the main following form.

```
> (syntax pattern) / (initial stack pattern) => (result stack pattern)
  iff (conditions)
  where (recursions)
```

The left hand side of the `=>` sign is used for selecting the rule. Given a program and an initial stack, one (and only one) rule can be selected using the following process. First, the toplevel structure of the program must match the syntax pattern. This is quite simple since there is only a few non trivial patterns to deal with instruction sequences, and the rest is made of trivial pattern that match one specific instruction. Then, the initial stack must match the initial stack pattern. Finally, some rules add extra conditions over the values in the stack that follow the `iff` keyword. Sometimes, several rules may apply in a given context. In this case, the one that appears first in this specification is to be selected. If no rule applies, the result is equivalent to the one for the explicit `FAIL` instruction. This case does not happen on well-typed programs, as explained in the next section.

The right hand side describes the result of the interpreter if the rule applies. It consists in a stack pattern, whose part are either constants, or elements of the context (program and initial stack) that have been named on the left hand side of the `=>` sign.

Recursive rules (big step form)

Sometimes, the result of interpreting a program is derived from the result of interpreting another one (as in conditionals or function calls). In these cases, the rule contains a clause of the following form.

```
where (intermediate program) / (intermediate stack) => (partial result)
```

This means that this rules applies in case interpreting the intermediate state on the left gives the pattern on the right.

The left hand sign of the `=>` sign is constructed from elements of the initial state or other partial results, and the right hand side identify parts that can be used to build the result stack of the rule.

If the partial result pattern does not actually match the result of the interpretation, then the result of the whole rule is equivalent to the one for the explicit `FAIL` instruction. Again, this case does not

happen on well-typed programs, as explained in the next section.

Format of patterns

Code patterns are of one of the following syntactical forms.

- `INSTR` (an uppercase identifier) is a simple instruction (e.g. `DROP`);
- `INSTR (arg) ...` is a compound instruction, whose arguments can be code, data or type patterns (e.g. `PUSH uint8 3`) ;
- `{ (instr) ; ... }` is a possibly empty sequence of instructions, (e.g. `IF { SWAP ; DROP } { DROP }`), nested sequences can drop the braces ;
- `name` is a pattern that matches any program and names a part of the matched program that can be used to build the result ;
- _ is a pattern that matches any instruction.

Stack patterns are of one of the following syntactical forms.

- `[FAIL]` is the special failed state ;
- `[]` is the empty stack ;
- `(top) : (rest)` is a stack whose top element is matched by the data pattern `(top)` on the left, and whose remaining elements are matched by the stack pattern `(rest)` on the right (e.g. `x : y : rest`) ;
- `name` is a pattern that matches any stack and names it in order to use it to build the result ;
- _ is a pattern that matches any stack.

Data patterns are of one of the following syntactical forms.

- integer literals, (e.g. `3`) ;
- string literals, (e.g. `"contents"`) ;
- `Tag` (capitalized) is a symbolic constant, (e.g. `Unit`, `True`, `False`) ;
- `(Tag (arg) ...)` tagged constructed data, (e.g. `(Pair 3 4)`) ;
- a code pattern for first class code values ;
- `name` to name a value in order to use it to build the result ;
- _ to match any value.

The domain of instruction names, symbolic constants and data constructors is fixed by this specification. Michelson does not let the programmer introduce its own types.

Be aware that the syntax used in the specification may differ a bit from the concrete syntax, which is presented in Section VIII. In particular, some instructions are annotated with types that are not present in the concrete language because they are synthesized by the typechecker.

Shortcuts

Sometimes, it is easier to think (and shorter to write) in terms of program rewriting than in terms of big step semantics. When it is the case, and when both are equivalents, we write rules of the form:

```
p / S   =>   S''
where   p' / S'   =>   S''
```

using the following shortcut:

```
p / S   =>   p' / S'
```

The concrete language also has some syntax sugar to group some common sequences of operations as one. This is described in this specification using a simple regular expression style recursive instruction rewriting.

II - Introduction to the type system and notations

This specification describes a type system for Michelson. To make things clear, in particular to readers that are not accustomed to reading formal programming language specifications, it does not give a typechecking or inference algorithm. It only gives an intentional definition of what we consider to be well-typed programs. For each syntactical form, it describes the stacks that are considered well-typed inputs, and the resulting outputs.

The type system is sound, meaning that if a program can be given a type, then if run on a well-typed input stack, the interpreter will never apply an interpretation rule on a stack of unexpected length or contents. Also, it will never reach a state where it cannot select an appropriate rule to continue the execution. Well-typed programs do not block, and do not go wrong.

Type notations

The specification introduces notations for the types of values, terms and stacks. Appart from a subset of value types that appear in the form of type annotations in some places througout the language, it is important to understand that this type language only exists in the specification.

A stack type can be written:

- `[]` for the empty stack ;
- `(top) : (rest)` for the stack whose first value has type `(top)` and queue has stack type `(rest)`.

Instructions, programs and primitives of the language are also typed, their types are written:

```
(type of stack before) -> (type of stack after)
```

The types of values in the stack are written:

- `identifier` for a primitive data-type (e.g. `bool`),
- `identifier (arg)` for a parametric data-type with one parameter type `(arg)` (e.g. `list uint8`),
- `identifier (arg) ...` for a parametric data-type with several parameters (e.g. `map string uint8`

),
- `[ (type of stack before) -> (type of stack after) ]` for a code quotation, (e.g. `[ uint8 : uint8 : [] -> uint8 : [] ]`),
- `lambda (arg) (ret)` is a shortcut for `[ (arg) : [] -> (ret) : [] ]`.

Meta type variables

The typing rules introduce meta type variables. To be clear, this has nothing to do with polymorphism, which Michelson does not have. These variables only live at the specification level, and are used to express the consistency between the parts of the program. For instance, the typing rule for the `IF` construct introduces meta variables to express that both branches must have the same type.

Here are the notations for meta type variables:

- `'a` for a type variable,
- `'A` for a stack type variable,
- _ for an anonymous type or stack type variable.

Typing rules

The system is syntax directed, which means here that it defines a single typing rule for each syntax construct. A typing rule restricts the type of input stacks that are authorized for this syntax construct, links the output type to the input type, and links both of them to the subexpressions when needed, using meta type variables.

Typing rules are of the form:

```
(syntax pattern)
:: (type of stack before) -> (type of stack after) [rule-name]
   iff (premisses)
```

Where premisses are typing requirements over subprograms or values in the stack, both of the form `(x) :: (type)`, meaning that value `(x)` must have type `(type)`.

A program is shown well-typed if one can find an instance of a rule that applies to the toplevel program expression, with all meta type variables replaced by non variable type expressions, and of which all type requirements in the premisses can be proven well-typed in the same manner. For the reader unfamiliar with formal type systems, this is called building a typing derivation.

Here is an example typing derivation on a small program that computes `x+5/10` for a given input `x`, obtained by instanciating the typing rules for instructions `PUSH`, `ADD` and for the sequence, as found in the next sections. When instanciating, we replace the `iff` with `by`.

```
{ PUSH uint8 5 ; ADD ; PUSH uint8 10 ; SWAP ; DIV }
:: [ uint8 : [] -> uint8 : [] ]
```

```
    by { PUSH uint8 5 ; ADD }
       :: [ uint8 : [] -> uint8 : [] ]
          by PUSH uint8 5
             :: [ uint8 : [] -> uint8 : uint8 : [] ]
                by 5 :: uint8
           and ADD
             :: [ uint8 : uint8 : [] -> uint8 : [] ]
   and { PUSH uint8 10 ; SWAP ; DIV }
       :: [ uint8 : [] -> uint8 : [] ]
          by PUSH uint8 10
             :: [ uint8 : [] -> uint8 : uint8 : [] ]
                by 10 :: uint8
           and { SWAP ; DIV }
             :: [ uint8 : uint8 : [] -> uint8 : [] ]
                by SWAP
                   :: [ uint8 : uint8 : [] -> uint8 : uint8 : [] ]
                and DIV
                   :: [ uint8 : uint8 : [] -> uint8 : [] ]
```

Producing such a typing derivation can be done in a number of manners, such as unification or abstract interpretation. In the implementation of Michelson, this is done by performing a recursive symbolic evaluation of the program on an abstract stack representing the input type provided by the programmer, and checking that the resulting symbolic stack is consistent with the expected result, also provided by the programmer.

Side note

As with most type systems, it is incomplete. There are programs that cannot be given a type in this type system, yet that would not go wrong if executed. This is a necessary compromise to make the type system usable. Also, it is important to remember that the implementation of Michelson does not accept as many programs as the type system describes as well-typed. This is because the implementation uses a simple single pass typechecking algorithm, and does not handle any form of polymorphism.

III - Core data types and notations

- `string`, `u?int{8|16|32|64}`: The core primitive constant types.
- `bool`: The type for booleans whose values are `True` and `False`
- `unit`: The type whose only value is `Unit`, to use as a placeholder when some result or parameter is non necessary. For instance, when the only goal of a contract is to update its storage.
- `list (t)`: A single, immutable, homogeneous linked list, whose elements are of type `(t)`, and that we note `Nil` for the empty list or `(Cons (head) (tail))`.
- `pair (l) (r)`: A pair of values `a` and `b` of types `(l)` and `(r)`, that we write `(Pair a b)`.
- `option (t)`: Optional value of type `(t)` that we note `None` or `(Some v)`.
- `or (l) (r)`: A union of two types: a value holding either a value `a` of type `(l)` or a value `b` of type `(r)`, that we write `(Left a)` or `(Right b)`.
- `set (t)`: Immutable sets of values of type `(t)` that we note `(Set (item) ...)`.
- `map (k) (t)`: Immutable maps from keys of type `(k)` of values of type `(t)` that we note `(Map`

```
(Item (key) (value)) ...).
```

IV - Core instructions

Control structures

- `FAIL`: Explicitly abort the current program. `:: _ -> _`
  This special instruction is callable in any context, since it does not use its input stack (first rule below), and makes the output useless since all subsequent instruction will simply ignore their usual semantics to propagate the failure up to the main result (second rule below). Its type is thus completely generic. `> FAIL / _ => [FAIL]`
  `> _ / [FAIL] => [FAIL]`
- `{ I ; C }`: Sequence. `:: 'A -> 'C`
  `iff I :: [ 'A -> 'B ]`
  `C :: [ 'B -> 'C ]`

  ```
  > I ; C / SA => SC
    where I / SA => SB
      and C / SB => SC
  ```
- `IF bt bf`: Conditional branching. `:: bool : 'A -> 'B`
  `iff bt :: [ 'A -> 'B ]`
  `bf :: [ 'A -> 'B ]`

  ```
  > IF bt bf / True : S => bt / S
  > IF bt bf / False : S => bf / S
  ```
- `LOOP body`: A generic loop. `:: bool : 'A -> 'A`
  `iff body :: [ 'A -> bool : 'A ]`

  ```
  > LOOP body / True : S => body ; LOOP body / S
  > LOOP body / False : S => S
  ```
- `DIP code`: Runs code protecting the top of the stack. `:: 'b : 'A -> 'b : 'C`
  `iff code :: [ 'A -> 'C ]`

  ```
  > DIP code / x : S => x : S'
    where code / S => S'
  ```
- `DII+P code`: A syntactic sugar for working deeper in the stack. `> DII(\rest=I*)P code / S => DIP (DI(\rest)P code) / S`
- `EXEC`: Execute a function from the stack. `:: 'a : lambda 'a 'b : 'C -> 'b : 'C`

  ```
  > EXEC / a : f : S => r : S
    where f / a : [] => r : []
  ```

Stack operations

- `DROP`: Drop the top element of the stack. `:: _ : 'A -> 'A`

```
> DROP / _ : S => S
```

- `DUP`: Duplicate the top of the stack. `:: 'a : 'A -> 'a : 'a : 'A`

```
> DUP / x : S => x : x : S
```

- `DUU+P`: A syntactic sugar for duplicating the `n`th element of the stack. `> DUU(\rest=U*)P / S => DIP (DU(\rest)P) ; SWAP / S`
- `SWAP`: Exchange the top two elements of the stack. `:: 'a : 'b : 'A -> 'b : 'a : 'A`

```
> SWAP / x : y : S => y : x : S
```

- `PUSH 'a x`: Push a constant value of a given type onto the stack. `:: 'A -> 'a : 'A iff x :: 'a`

```
> PUSH 'a x / S => x : S
```

- `UNIT`: Push a unit value onto the stack. `:: 'A -> unit : 'A`

```
> UNIT / S => Unit : S
```

Generic comparison

Comparison only works on a class of types that we call comparable. A `COMPARE` operation is defined in an ad hoc way for each comparable type, but the result of compare is always an `int64`, which can in turn be checked in a generic manner using the following combinators. The result of `COMPARE` is `0` if the compared values are equal, negative if the first is less than the second, and positive otherwise.

- `EQ`: Checks that the top of the stack EQuals zero. `:: int64 : 'S -> bool : 'S`

```
> EQ ; C / Int64 (0) : S => C / True : S
> EQ ; C / _ : S => C / False : S
```

- `NEQ`: Checks that the top of the stack does Not EQual zero. `:: int64 : 'S -> bool : 'S`

```
> NEQ ; C / Int64 (0) : S => C / False : S
> NEQ ; C / _ : S => C / True : S
```

- `LT`: Checks that the top of the stack is Less Than zero. `:: int64 : 'S -> bool : 'S`

```
> LT ; C / Int64 (v) : S => C / True : S iff v < 0
> LT ; C / _ : S => C / False : S
```

- `GT`: Checks that the top of the stack is Greater Than zero. `:: int64 : 'S -> bool : 'S`

```
> GT ; C / Int64 (v) : S   =>   C / True : S   iff  v > 0
> GT ; C / _ : S           =>   C / False : S
```

- `LE`: Checks that the top of the stack is Less Than of Equal to zero. `:: int64 : 'S   ->   bool : 'S`

```
> LE ; C / Int64 (v) : S   =>   C / True : S   iff  v <= 0
> LE ; C / _ : S           =>   C / False : S
```

- `GE`: Checks that the top of the stack is Greater Than of Equal to zero. `:: int64 : 'S   ->   bool : 'S`

```
> GE ; C / Int64 (v) : S   =>   C / True : S   iff  v >= 0
> GE ; C / _ : S           =>   C / False : S
```

Syntactic sugar exists for merging `COMPARE` and comparison combinators, and also for branching.

- `CMP{EQ|NEQ|LT|GT|LE|GE} > CMP(\op) ; C / S   =>   COMPARE ; (\op) ; C / S`
- `IF{EQ|NEQ|LT|GT|LE|GE} bt bf > IF(\op) ; C / S   =>   (\op) ; IF bt bf ; C / S`
- `IFCMP{EQ|NEQ|LT|GT|LE|GE} bt bf > IFCMP(\op) ; C / S   =>   COMPARE ; (\op) ; IF bt bf ; C / S`

V - Operations

Operations on booleans

- `OR :: bool : bool : 'S   ->   bool : 'S`

```
> OR ; C / x : y : S   =>   C / (x | y) : S
```

- `AND :: bool : bool : 'S   ->   bool : 'S`

```
> AND ; C / x : y : S   =>   C / (x & y) : S
```

- `XOR :: bool : bool : 'S   ->   bool : 'S`

```
> XOR ; C / x : y : S   =>   C / (x ^ y) : S
```

- `NOT :: bool : 'S   ->   bool : 'S`

```
> NOT ; C / x : S   =>   C / ~x : S
```

Operations on integers

Integers can be of size 1, 2, 4 or 8 bytes, signed or unsigned. Integer Operations are homogeneous, so that performing computations between values of different int types must be done via explicit casts.

For specifying arithmetics, we consider that integers are all stored on 64 bits (the largest integer size) so that we can express the operations, in particular casts, using usual bitwise masks. In this context, the type indicator functions are defined as follows (which can be read both as a constraint on the bitpatttern and as a conversion operation).

```
Uint64 (x) = Int64 (x) = x
Uint32 (x) = x & 0x00000000FFFFFFFF
Int32 (x) = x & 0x00000000FFFFFFFF
          | (x & 0x80000000 ? 0xFFFFFFFF00000000 : 0)
Uint16 (x) = x & 0x000000000000FFFF
Int16 (x) = x & 0x000000000000FFFF
          | (x & 0x8000 ? 0xFFFFFFFFFFFF0000 : 0)
Uint8 (x) = x & 0x00000000000000FF
Int8 (x) = x & 0x00000000000000FF
         | (x & 0x80 ? 0xFFFFFFFFFFFFFF00 : 0)
```

We also use the function `bits (t)` that retrieve the meaningful number of bits for a given integer type (e.g. `bits (int8) = 8`).

- `NEG:: t : 'S -> t : 'S for t in int{8|16|32|64}`

  `> NEG ; C / t (x) : S => C / t (-x) : S`

  With cycling semantics for overflows (min (t) = -min (t)).
- `ABS :: t : 'S -> t : 'S for t in int{8|16|32|64}`

  `> ABS ; C / t (x) : S => C / t (abs (x)) : S`

  With cycling semantics for overflows (abs (min (t)) = min (t)).
- `ADD :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> ADD ; C / t (x) : t (y) : S => C / t (x + y) : S`

  With cycling semantics for overflows.
- `SUB :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> SUB ; C / t (x) : t (y) : S => C / t (x - y) : S`

  With cycling semantics for overflows.
- `MUL :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> MUL ; C / t (x) : t (y) : S => C / t (x * y) : S`

  Unckeched for overflows.

- `DIV :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> DIV ; C / t (x) : t (0) : S => C / [FAIL]`

  `> DIV ; C / t (x) : t (y) : S => C / t (x / y) : S`
- `MOD :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> MOD ; C / t (x) : t (0) : S => C / [FAIL]`

  `> MOD ; C / t (x) : t (y) : S => C / t (x % y) : S`
- `CAST t_to` where `t_to in u?int{8|16|32|64} :: t_from : 'S -> t_to : 'S for t_from in u?int{8|16|32|64}`

  `> CAST t_to ; C / t_from (x) : S => C / t_to (x) : S`

Alternative operators are defined that check for overflows.

- `CHECKED_NEG :: t : 'S -> t : 'S for t in int{8|16|32|64}`

  `> CHECKED_NEG ; C / t (x) : S => [FAIL] on overflow`

  `> CHECKED_NEG ; C / t (x) : S => C / t (-x) : S`
- `CHECKED_ABS :: t : 'S -> t : 'S for t in int{8|16|32|64}`

  `> CHECKED_ABS ; C / t (x) : S => [FAIL] on overflow`

  `> CHECKED_ABS ; C / t (x) : S => C / t (abs (x)) : S`
- `CHECKED_ADD :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> CHECKED_ADD ; C / t (x) : t (y) : S => [FAIL] on overflow`

  `> CHECKED_ADD ; C / t (x) : t (y) : S => C / t (x + y) : S`
- `CHECKED_SUB :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> CHECKED_SUB ; C / t (x) : t (y) : S => [FAIL] on overflow`

  `> CHECKED_SUB ; C / t (x) : t (y) : S => C / t (x - y) : S`
- `CHECKED_MUL :: t : t : 'S -> t : 'S for t in u?int{8|16|32|64}`

  `> CHECKED_MUL ; C / t (x) : t (y) : S => [FAIL] on overflow`

  `> CHECKED_MUL ; C / t (x) : t (y) : S => C / t (x * y) : S`
- `CHECKED_CAST t_to` where `t_to in u?int{8|16|32|64} :: t_from : 'S -> t_to : 'S for t_from in u?int{8|16|32|64}`

  `> CHECKED_CAST t_to ; C / t_from (x) : S => C / t_to (x) : S`
  `iff t_from (x) = t_to (x)`

  `> CHECKED_CAST t_to ; C / t_from (x) : S => [FAIL]`

Bitwise logical operators are also available on unsigned integers.

- `OR :: t : t : 'S -> t : 'S for t in uint{8|16|32|64}`

  `> OR ; C / t (x) : t (y) : S => C / t (x | y) : S`
- `AND :: t : t : 'S -> t : 'S for t in uint{8|16|32|64}`

  `> AND ; C / t (x) : t (y) : S => C / t (x & y) : S`
- `XOR :: t : t : 'S -> t : 'S for t in uint{8|16|32|64}`

  `> XOR ; C / t (x) : t (y) : S => C / t (x ^ y) : S`
- `NOT :: t : 'S -> t : 'S for t in uint{8|16|32|64}`

  `> NOT ; C / t (x) : S => C / t (~x) : S`
- `LSL :: t : uint8 (s) : 'S -> t : 'S for t in uint{8|16|32|64}`

  ```
  > LSL ; C / t (x) : uint8 (s) : S => C / t (x << s) : S
    iff s <= bits (t)
  > LSL ; C / t (x) : uint8 (s) : S => [FAIL]
  ```
- `LSR :: t : uint8 (s) : 'S -> t : 'S for t in uint{8|16|32|64}`

  ```
  > LSR ; C / t (x) : uint8 (s) : S => C / t (x >>> s) : S
    iff s <= bits (t)
  > LSR ; C / t (x) : uint8 (s) : S => [FAIL]
  ```
- `COMPARE`: Integer comparison (signed or unsigned according to the type). `:: t : t : 'S -> int64 : 'S for t in uint{8|16|32|64}`

Operations on strings

Strings are mostly used for naming things without having to rely on external ID databases. So what can be done is basically use string constants as is, concatenate them and use them as keys.

- `CONCAT`: String concatenation. `:: string : string : 'S -> string : 'S`
- `COMPARE`: Lexicographic comparison. `:: string : string : 'S -> int64 : 'S`

Operations on pairs

- `PAIR`: Build a pair from the stack's top two elements. `:: 'a : 'b : 'S -> pair 'a 'b : 'S`

  `> PAIR ; C / a : b : S => C / (Pair a b) : S`
- `P(A*AI)+R`: A syntactic sugar for building nested pairs in bulk. `> P(\fst=A*)AI(\rest=(A*AI)+)R`

```
; C / S  =>  P(\fst)AIR ; P(\rest)R ; C / S
> PA(\rest=A*)AIR ; C / S  =>  DIP (P(\rest)AIR) ; C / S
```

- `CAR`: Access the left part of a pair. `:: pair 'a _ : 'S  ->  'a : 'S`

```
> Car ; C / (Pair a _) : S  =>  C / a : S
```

- `CDR`: Access the right part of a pair. `:: pair _ 'b : 'S  ->  'b : 'S`

```
> Car ; C / (Pair _ b) : S  =>  C / b : S
```

- `C[AD]+R`: A syntactic sugar for accessing fields in nested pairs. `> CA(\rest=[AD]+)R ; C / S  => CAR ; C(\rest)R ; C / S`
  `> CD(\rest=[AD]+)R ; C / S  =>  CDR ; C(\rest)R ; C / S`

Operations on sets

- `EMPTY_SET 'elt`: Build a new, empty set for elements of a given type. `:: 'S  ->  set 'elt : 'S`
  The `'elt` type must be comparable (the `COMPARE` primitive must be defined over it).
- `MEM`: Check for the presence of an element in a set. `:: 'key : set 'elt : 'S  ->  bool : 'S`
- `UPDATE`: Inserts or removes an element in a set, replacing a previous value. `:: 'elt : bool : set 'elt : 'S  ->  set 'elt : 'S`
- `REDUCE`: Apply a function on a set passing the result of each application to the next one and return the last. `:: lambda (pair 'elt * 'b) 'b : set 'elt : 'b : 'S  ->  'b : 'S`

Operations on maps

- `EMPTY_MAP 'key 'val`: Build a new, empty map. The `'key` type must be comparable (the `COMPARE` primitive must be defined over it). `:: 'S -> map 'key 'val : 'S`
- `GET`: Access an element in a map, returns an optional value to be checked with `IF_SOME`. `:: 'key : map 'key 'val : 'S  ->  option 'val : 'S`
- `MEM`: Check for the presence of an element in a map. `:: 'key : map 'key 'val : 'S  ->  bool : 'S`
- `UPDATE`: Assign or remove an element in a map. `:: 'key : option 'val : map 'key 'val : 'S  ->  map 'key 'val : 'S`
- `MAP`: Apply a function on a map and return the map of results under the same bindings. `:: lambda (pair 'key 'val) 'b : map 'key 'val : 'S  ->  map 'key 'b : 'S`
- `REDUCE`: Apply a function on a map passing the result of each application to the next one and return the last. `:: lambda (pair (pair 'key 'val) 'b) 'b : map 'key 'val : 'b : 'S  -> 'b : 'S`

Operations on optional values

- `SOME`: Pack a present optional value. `:: 'a : 'S -> 'a? : 'S`

```
> SOME ; C / v :: S => C / (Some v) :: S
```

- `NONE 'a`: The absent optional value. `:: 'S -> 'a? : 'S`

```
> NONE ; C / v :: S => C / None :: S
```

- `IF_SOME bt bf`: Inspect an optional value. `:: 'a? : 'S -> 'b : 'S`

```
   iff  bt :: [ 'a : 'S -> 'b : 'S]
        bf :: [ 'S -> 'b : 'S]

> IF_SOME ; C / (Some a) : S  =>  bt ; C / a : S
> IF_SOME ; C / (None) : S  =>  bf ; C / S
```

Operations on unions

- `LEFT 'b`: Pack a value in a union (left case). `:: 'a : 'S -> or 'a 'b : 'S`

```
> LEFT ; C / v :: S => C / (Left v) :: S
```

- `RIGHT 'a`: Pack a value in a union (right case). `:: 'b : 'S -> or 'a 'b : 'S`

```
> RIGHT ; C / v :: S => C / (Right v) :: S
```

- `IF_LEFT bt bf`: Inspect an optional value. `:: or 'a 'b : 'S -> 'c : 'S`

```
   iff  bt :: [ 'a : 'S -> 'c : 'S]
        bf :: [ 'b : 'S -> 'c : 'S]

> IF_LEFT ; C / (Left a) : S  =>  bt ; C / a : S
> IF_LEFT ; C / (Right b) : S  =>  bf ; C / b : S
```

Operations on lists

- `CONS`: Prepend an element to a list. `:: 'a : list 'a : 'S -> list 'a : 'S`

```
> CONS ; C / a : l : S => C / (Cons a l) : S
```

- `NIL 'a`: The empty list. `:: 'S -> list 'a : 'S`

```
> NIL ; C / S => C / Nil : S
```

- `IF_CONS bt bf`: Inspect an optional value. `:: list 'a : 'S -> 'b : 'S`

```
   iff  bt :: [ 'a : list 'a : 'S -> 'b : 'S]
        bf :: [ 'S -> 'b : 'S]

> IF_CONS ; C / (Cons a rest) : S  =>  bt ; C / a : rest : S
> IF_CONS ; C / Nil : S  =>  bf ; C / S
```

- `MAP`: Apply a function on a list from left to right and return the list of results in the same order. `:: lambda 'a 'b : list 'a : 'S -> list 'b : 'S`
- `REDUCE`: Apply a function on a list from left to right passing the result of each application to the next one and return the last. `:: lambda (pair 'a 'b) 'b : list 'a : 'b : 'S -> 'b : 'S`

VI - Domain specific data types

- `timestamp`: Dates in the real world.
- `tez`: A specific type for manipulating tokens.
- `contract 'param 'result`: A contract, with the type of its code.
- `key`: A public cryptography key.
- `signature`: A cryptographic signature.

VII - Domain specific operations

Operations on timestamps

Timestamp immediates can be obtained by the `NOW` operation, or retrieved from script parameters or globals. The only valid operations are the addition of a (positive) number of seconds and the comparison.

- `ADD` Increment / decrement a timestamp of the given number of seconds. `:: timestamp : float : 'S -> timestamp : 'S`

```
> ADD ; C / t : period : S   =>   [FAIL]   iff   period < 0
> ADD ; C / t : period : S   =>   C / (t + period seconds) : S
```

- `ADD` Increment / decrement a timestamp of the given number of seconds. `:: timestamp : uint{8|16|32|64} : 'S -> timestamp : 'S`

```
> ADD ; C / t : seconds : S   =>   [FAIL]   on overflow
> ADD ; C / t : seconds : S   =>   C / (t + seconds) : S
```

- `COMPARE`: Timestamp comparison. `:: timestamp : timestamp : 'S   ->   int64 : 'S`

Operations on Tez

Operations on tez are limited to prevent overflow and mixing them with other numerical types by

mistake. They are also mandatorily checked for under/overflows.

- `ADD`: `:: tez : tez : 'S  ->  tez : 'S`

  `> ADD ; C / x : y : S  =>  [FAIL]  on overflow`

  `> ADD ; C / x : y : S  =>  C / (x + y) : S`
- `SUB`: `:: tez : tez : 'S  ->  tez : 'S`

  `> SUB ; C / x : y : S  =>  [FAIL]  iff  x < y`

  `> SUB ; C / x : y : S  =>  C / (x - y) : S`
- `MUL :: tez : u?int{8|16|32|64} : 'S  ->  tez : 'S`

  `> MUL ; C / x : y : S  =>  [FAIL]  on overflow`

  `> MUL ; C / x : y : S  =>  C / (x * y) : S`
- `COMPARE`: `:: tez : tez : 'S  ->  int64 : 'S`

Operations on contracts

- `MANAGER`: Access the manager of a contract. `:: contract 'p 'r : 'S  ->  key : 'S`
- `CREATE_CONTRACT`: Forge a new contract. `:: key : key? : bool : bool : tez : lambda (pair (pair tez 'p) 'g) (pair 'r 'g) : 'g : 'S -> contract 'p 'r : 'S`

  As with non code-emitted originations the contract code takes as argument the transfered amount plus an ad-hoc argument and returns an ad-hoc value. The code also takes the global data and returns it to be stored and retrieved on the next transaction. These data are initialized by another parameter. The calling convention for the code is as follows: `(Pair (Pair amount arg) globals)) -> (Pair ret globals)`, as extrapolable from the instruction type. The first parameters are the manager, optional delegate, then spendable and delegatable flags and finally the initial amount taken from the currently executed contract. The contract is returned as a first class value to be called immediately or stored.
- `CREATE_ACCOUNT`: Forge an account (a contract without code). `:: key : key? : bool : tez : 'S -> contract unit unit : 'S`

  Take as argument the manager, optional delegate, the delegatable flag and finally the initial amount taken from the currently executed contract.
- `TRANSFER_TOKENS`: Forge and evaluate a transaction. `:: 'p : tez : contract 'p 'r : 'g : [] -> 'r : 'g : []`

  The parameter and return value must be consistent with the ones expected by the contract, unit for an account. To preserve the global consistency of the system, the current contract's storage must be updated before passing the control to another script. For this, the script must put the partially updated storage on the stack ('g is the type of the contract's storage). If a recursive call to the current contract happened, the updated storage is put on the stack next to the return value. Nothing else can remain on the stack during a nested call. If some local values have to be kept for after the nested call, they have to be stored explicitly in a transient part of the storage. A trivial

example of that is to reserve a boolean in the storage, initialized to false, reset to false at the end of each contract execution, and set to true during a nested call. This thus gives an easy way for a contract to prevent recursive call (the contract just fails if the boolean is true).
- `BALANCE`: Push the current amount of tez of the current contract. `:: 'S -> tez :: 'S`
- `SOURCE 'p 'r`: Push the source contract of the current transaction. `:: 'S -> contract 'p 'r :: 'S`
- `SELF`: Push the current contract. `:: 'S -> contract 'p 'r :: 'S`
  `where contract 'p 'r is the type of the current contract`
- `AMOUNT`: Push the amount of the current transaction. `:: 'S -> tez :: 'S`

Special operations

- `STEPS_TO_QUOTA`: Push the remaining steps before the contract execution must terminate. `:: 'S -> uint32 :: 'S`
- `NOW`: Push the timestamp of the block whose validation triggered this execution (does not change during the execution of the contract). `:: 'S -> timestamp :: 'S`

Cryptographic primitives

- `H`: Compute a cryptographic hash of the value contents using the Sha256 cryptographic algorithm. `:: 'a : 'S -> string : 'S`
- `CHECK_SIGNATURE` Check that a sequence of bytes has been signed with a given key. `:: key : pair signature string : 'S -> bool : 'S`
- `COMPARE :: key : key : 'S -> int64 : 'S`

VIII - Concrete syntax

The concrete language is very close to the formal notation of the specification. Its structure is extremely simple: an expression in the language can only be one of the three following constructs.

1. A constant (integer or string).
2. The application of a primitive to a sequence of expressions.
3. A sequence of expressions.

Constants

There are two kinds of constants:

1. Integers in decimal (no prefix), hexadecimal (0x prefix), octal (0o prefix) or binary (0b prefix).
2. Strings with usual escapes `\n`, `\t`, `\b`, `\r`, `\\`, `\"`. Strings are encoding agnostic sequences of bytes. Non printable characters can be escaped by 3 digits decimal codes `\ddd` or 2 digit hexadecimal codes `\xHH`.

Primitive applications

In the specification, primitive applications always luckily fit on a single line. In this case, the concrete syntax is exactly the formal notation. However, it is sometimes necessary to break lines in a real program, which can be done as follows.

As in Python or Haskell, the concrete syntax of the language is indentation sensitive. The elements of a syntactical block, such as all the elements of a sequence, or all the parameters of a primitive, must be written with the exact same left margin in the program source code. This is unlike in C-like languages, where blocks are delimited with braces and the margin is ignored by the compiled.

The simplest form requires to break the line after the primitive name and after every argument. Argument must be indented by at least one more space than the primitive, and all arguments must sit on the exact same column.

```
PRIM
  arg1
  arg2
  ...
```

If an argument of a primitive application is a primitive application itself, its arguments must be pushed even further on the right, to lift any ambiguity, as in the following example.

```
PRIM1
  PRIM2
    arg1_prim2
    arg2_prim2
  arg2_prim1
```

It is possible to put successive arguments on a single line using a semicolon as a separator:

```
PRIM
  arg1; arg2
  arg3; arg4
```

```

```

It is also possible to add arguments on the same line as the primitive as a lighter way to write simple expressions. An other representation of the first example is:

```
PRIM arg1 arg2 ...
```

It is possible to mix both notations as in:

```
PRIM arg1 arg2
  arg3
  arg4
```

Or even:

```
PRIM arg1 arg2
  arg3; arg4
```

Both equivalent to:

```
PRIM
  arg1
  arg2
  arg3
  arg4
```

Trayling semicolons are ignored:

```
PRIM
  arg1;
  arg2
```

Calling a primitive with a compound argument on a single line is allowed by wrapping with parentheses. Another notation for the second example is:

```
PRIM1 (PRIM2 arg1_prim2 arg2_prim2) arg2_prim1
```

Sequences

Successive expression can be grouped as a single sequence expression using braces delimiters and semicolon separators.

```
{ expr1 ; expr2 ; expr3 ; expr4 }
```

A sequence block can be split on several lines. In this situation, the whole block, including the closing brace, must be indented with respect to the first instruction.

```
{ expr1 ; expr2
  expr3 ; expr4 }
```

Blocks can be passed as argument to a primitive.

```
PRIM arg1 arg2
  { arg3_expr1 ; arg3_expr2
    arg3_expr3 ; arg3_expr4 }
```

Conventions

The concrete syntax follows the same lexical conventions as the specification: instructions are represented by uppercase identifiers, type constructors by lowercase identifiers, and constant constructors are Capitalised.

Lists can be written in a single shot instead of a succession of `Cons`

```
(List 1 2 3) = (Cons 1 (Cons 2 (Cons 3 Nil)))
```

All domain specific constants are strings with specific formats:

- `tez` amounts are written using the same notation as JSON schemas and the command line client: thousands are optionally separated by comas, and centiles, if present, must be prefixed by a period.
  - in regexp form: `([0-9]{1,3}(,[0-9]{3})+)|[0-9]+(\.[0.9]{2})?`
  - `"1234567"` means 123456700 tez centiles
  - `"1,234,567"` means 123456700 tez centiles
  - `"1234567.89"` means 123456789 tez centiles
  - `"1,234,567.00"` means 123456789 tez centiles
  - `"1234,567"` is invalid
  - `"1,234,567."` is invalid
  - `"1,234,567.0"` is invalid
- `timestamp`s are written using `RFC 339` notation.
- `contract`s are the raw strings returned by JSON RPCs or the command line interface and cannot be forged by hand so their format is of no interest here.
- `key`s are `Sha256` hashes of `ed25519` public keys encoded in `base48` format with the following custom alphabet: `"eXMNE9qvHPQDdcFx5J86rT7VRm2atAypGhgLfbS3CKjnksB4"`.
- `signature`s are `ed25519` signatures as a series of hex-encoded bytes.

To prevent errors, control flow primitives that take instructions as parameters require sequences in the concrete syntax.

```
IF { instr1_true ; instr2_true ; ... } { instr1_false ; instr2_false ; ... }

IF
  { instr1_true ; instr2_true ; ... }
  { instr1_false ; instr2_false ; ... }
```

Main program structure

The toplevel of a smart contract file must be an undelimited sequence of four primitive applications (in no particular order) that provide its `parameter`, `return` and `storage` types, as well as its `code`.

See the next section for a concrete example.

Comments

A hash sign (`#`) anywhere outside of a string literal will make the rest of the line (and itself) completely ignored, as in the following example.

```
PUSH int8 1 # pushes 1
PUSH int8 2 # pushes 2
ADD         # computes 2 + 1
```

IX - Examples

Contracts in the system are stored as a piece of code and a global data storage. The type of the global data of the storage is fixed for each contract at origination time. This is ensured statically by checking on origination that the code preserves the type of the global data. For this, the code of the contract is checked to be of the following type lambda (pair (pair tez 'arg) 'global) -> (pair 'ret 'global) where 'global is the type of the original global store given on origination. The contract also takes a parameter and an amount, and returns a value, hence the complete calling convention above.

Empty contract

Because of the calling convention, the empty sequence is not a valid contract of type `(contract unit unit)`. The code for building a contract of such a type must take a `unit` argument, an amount in `tez`, and transform a unit global storage, and must thus be of type `(lambda (pair (pair tez unit) unit) (pair unit unit))`.

Such a minimal contract code is thus `{ CDR ; UNIT ; PAIR }`.

A valid contract source file would be as follows.

```
code { CDR ; UNIT ; PAIR }
```

```
storage unit
parameter unit
return unit
```

Reservoir contract

We want to create a contract that stores tez until a timestamp `T` or a maximum amount `N` is reached. Whenever `N` is reached before `T`, all tokens are reversed to an account `B` (and the contract is automatically deleted). Any call to the contract's code performed after `T` will otherwise transfer the tokens to another account `A`.

We want to build this contract in a reusable manner, so we do not hard-code the parameters. Instead, we assume that the global data of the contract are `(Pair (Pair T N) (Pair A B))`.

Hence, the global data of the contract has the following type

```
'g =
  pair
    pair timestamp tez
    pair (contract unit unit) (contract unit unit)
```

Following the contract calling convention, the code is a lambda of type

```
lambda
  pair (pair tez unit) 'g
  pair unit 'g
```

writen as

```
lambda
  pair (pair tez unit)
    pair
      pair timestamp tez
      pair (contract unit unit) (contract unit unit)
  pair unit
    pair
      pair timestamp tez
      pair (contract unit unit) (contract unit unit)
```

The complete source `reservoir.tz` is:

```
parameter timestamp ;
storage
  pair
    (pair timestamp tez) # T N
    (pair (contract unit unit) (contract unit unit)) ; # A B
return unit ;
```

```
code
  { DUP ; CDAAR ; # T
    NOW ;
    COMPARE ; LE ;
    IF { DUP ; CDADR ; # N
         BALANCE ;
         COMPARE ; LE ;
         IF { CDR ; UNIT ; PAIR }
            { DUP ; CDDDR ; # B
              BALANCE ; UNIT ;
              DIIIP { CDR } ;
              TRANSFER_TOKENS ;
              PAIR } }
       { DUP ; CDDAR ; # A
         BALANCE ;
         UNIT ;
         DIIIP { CDR } ;
         TRANSFER_TOKENS ;
         PAIR } }
```

Reservoir contract (variant with broker and status)

We basically want the same contract as the previous one, but instead of destroying it, we want to keep it alive, storing a flag `S` so that we can tell afterwards if the tokens have been transfered to `A` or `B`. We also want a broker `X` to get some fee `P` in any case.

We thus add variables `P` and `S` and `X` to the global data of the contract, now `(Pair (S, Pair (T, Pair (Pair P N) (Pair X (Pair A B)))))`. `P` is the fee for broker `A`, `S` is the state, as a string `"open"`, `"timeout"` or `"success"`.

At the beginning of the transaction:

```
 S is accessible via a CDAR
 T               via a CDDAR
 P               via a CDDDAAR
 N               via a CDDDADR
 X               via a CDDDDAR
 A               via a CDDDDDAR
 B               via a CDDDDDDR
```

For the contract to stay alive, we test that all least `(Tez "1.00")` is still available after each transaction. This value is given as an example and must be updated according to the actual Tezos minmal value for contract balance.

The complete source `scrutable_reservoir.tz` is:

```
parameter timestamp ;
storage
  pair
    string # S
```

```
    pair
      timestamp # T
      pair
        (pair tez tez) ; # P N
        pair
          (contract unit unit) # X
          pair (contract unit unit) (contract unit unit) ; # A B
return unit ;
code
  { DUP ; CDAR # S
    PUSH string "open" ;
    COMPARE ; NEQ ;
    IF { FAIL } # on "success", "timeout" or a bad init value
       { DUP ; CDDAR ; # T
         NOW ;
         COMPARE ; LT ;
         IF { # Before timeout
              # We compute ((1 + P) + N) tez for keeping the contract alive
              PUSH tez "1.00" ;
              DIP { DUP ; CDDDAAR } ; ADD ; # P
              DIP { DUP ; CDDDADR } ; ADD ; # N
              # We compare to the cumulated amount
              BALANCE ;
              COMPARE; LT ;
              IF { # Not enough cash, we just accept the transaction
                   # and leave the global untouched
                   CDR }
                 { # Enough cash, successful ending
                   # We update the global
                   CDDR ; PUSH string "success" ; PAIR ;
                   # We transfer the fee to the broker
                   DUP ; CDDAAR ; # P
                   DIP { DUP ; CDDDAR } # X
                   UNIT ; TRANSFER_TOKENS ; DROP ;
                   # We transfer the rest to A
                   DUP ; CDDADR ; # N
                   DIP { DUP ; CDDDDAR } # A
                   UNIT ; TRANSFER_TOKENS ; DROP } }
            { # After timeout, we refund
              # We update the global
              CDDR ; PUSH string "timeout" ; PAIR ;
              # We try to transfer the fee to the broker
              PUSH tez "1.00" ; BALANCE ; SUB ; # available
              DIP { DUP ; CDDAAR } ; # P
              COMPARE ; LT ; # available < P
              IF { PUSH tez "1.00" ; BALANCE ; SUB ; # available
                   DIP { DUP ; CDDDAR } # X
                   UNIT ; TRANSFER_TOKENS ; DROP }
                 { DUP ; CDDAAR ; # P
                   DIP { DUP ; CDDDAR } # X
                   UNIT ; TRANSFER_TOKENS ; DROP }
              # We transfer the rest to B
              PUSH tez "1.00" ; BALANCE ; SUB ; # available
              DIP { DUP ; CDDDDDR } # B
              UNIT ; TRANSFER_TOKENS ; DROP } }
    # return Unit
    UNIT ; PAIR }
```

Forward contract

We want to write a forward contract on dried peas. The contract takes as global data the tons of peas `Q`, the expected delivery date `T`, the contract agreement date `Z`, a strike `K`, a collateral `C` per ton of dried peas, and the accounts of the buyer `B`, the seller `S` and the warehouse `W`.

These parameters as grouped in the global storage as follows:

```
Pair
  Pair (Pair Q (Pair T Z))
  Pair
    (Pair K C)
    (Pair (Pair B S) W)
```

of type

```
pair
  pair uint32 (pair timestamp timestamp)
  pair
    pair tez tez
    pair (pair account account) account
```

The 24 hours after timestamp `Z` are for the buyer and seller to store their collateral `(Q * C)`. For this, the contract takes a string as parameter, matching `"buyer"` or `"seller"` indicating the party for which the tokens are transfered. At the end of this day, each of them can send a transaction to send its tokens back. For this, we need to store who already paid and how much, as a `(pair tez tez)` where the left component is the buyer and the right one the seller.

After the first day, nothing cam happen until `T`.

During the 24 hours after `T`, the buyer must pay `(Q * K)` to the contract, minus the amount already sent.

After this day, if the buyer didn't pay enough then any transaction will send all the tokens to the seller.

Otherwise, the seller must deliver at least `Q` tons of dried peas to the warehouse, in the next 24 hours. When the amount is equal to or exceeds `Q`, all the tokens are transfered to the seller and the contract is destroyed. For storing the quantity of peas already delivered, we add a counter of type `uint32` in the global storage. For knowing this quantity, we accept messages from W with a partial amount of delivered peas as argument.

After this day, any transaction will send all the tokens to the buyer (not enough peas have been delivered in time).

Hence, the global storage is a pair, with the counters on the left, and the constant parameters on the right, initially as follows.

```
Pair
  Pair 0 (Pair 0_00 0_00)
  Pair
    Pair (Pair Q (Pair T Z))
    Pair
      (Pair K C)
      (Pair (Pair B S) W)
```

of type

```
pair
  pair unit32 (pair tez tez)
  pair
    pair uint32 (pair timestamp timestamp)
    pair
      pair tez tez
      pair (pair account account) account
```

The parameter of the transaction will be either a transfer from the buyer or the seller or a delivery notification from the warehouse of type `(or string uint32)`.

At the beginning of the transaction:

```
Q is accessible via a CDDAAR
T               via a CDDADAR
Z               via a CDDADDR
K               via a CDDDAAR
C               via a CDDDADR
B               via a CDDDDAAR
S               via a CDDDDADR
W               via a CDDDDDR
the delivery counter via a CDAAR
the amount versed by the buyer via a CDADAR
the amount versed by the seller via a CDADDR
the argument via a CADR
```

The contract returns a unit value, and we assume that it is created with the minimum amount, set to `(Tez "1.00")`.

The complete source `forward.tz` is:

```
parameter (or string uint32) ;
return unit ;
storage
  pair
    pair uint32 (pair tez tez) # counter from_buyer from_seller
```

```
    pair
      pair uint32 (pair timestamp timestamp) # Q T Z
      pair
        pair tez tez # K C
        pair
          pair (contract unit unit) (contract unit unit) # B S
          (contract unit unit); # W
code
  { DUP ; CDDADDR ; # Z
    PUSH uint64 86400 ; SWAP ; ADD ; # one day in second
    NOW ; COMPARE ; LT ;
    IF { # Before Z + 24
         DUP ; CADR ; # we must receive (Left "buyer") or (Left "seller")
         IF_LEFT
           { DUP ; PUSH string "buyer" ; COMPARE ; EQ ;
             IF { DROP ;
                  DUP ; CDADAR ; # amount already versed by the buyer
                  DIP { DUP ; CAAR } ; ADD ; # transaction
                  #  then we rebuild the globals
                  DIP { DUP ; CDADDR } ; PAIR ; # seller amount
                  PUSH uint32 0 ; PAIR ; # delivery counter at 0
                  DIP { CDDR } ; PAIR ; # parameters
                  # and return Unit
                  UNIT ; PAIR }
                { PUSH string "seller" ; COMPARE ; EQ ;
                  IF { DUP ; CDADDR ; # amount already versed by the seller
                       DIP { DUP ; CAAR } ; ADD ; # transaction
                       #  then we rebuild the globals
                       DIP { DUP ; CDADAR } ; SWAP ; PAIR ; # buyer amount
                       PUSH uint32 0 ; PAIR ; # delivery counter at 0
                       DIP { CDDR } ; PAIR ; # parameters
                       # and return Unit
                       UNIT ; PAIR }
                     { FAIL } } } # (Left _)
           { FAIL } } # (Right _)
       { # After Z + 24
         # test if the required amount is reached
         DUP ; CDDAAR ; # Q
         DIP { DUP ; CDDDADR } ; MUL ; # C
         PUSH uint8 2 ; MUL ;
         PUSH tez "1.00" ; ADD ;
         BALANCE ; COMPARE ; LT ; # balance < 2 * (Q * C) + 1
         IF { # refund the parties
              CDR ; DUP ; CADAR ; # amount versed by the buyer
              DIP { DUP ; CDDDAAR } # B
              UNIT ; TRANSFER_TOKENS ; DROP
              DUP ; CADDR ; # amount versed by the seller
              DIP { DUP ; CDDDADR } # S
              UNIT ; TRANSFER_TOKENS ; DROP
              BALANCE ; # bonus to the warehouse to destroy the account
              DIP { DUP ; CDDDDR } # W
              UNIT ; TRANSFER_TOKENS ; DROP
              # return unit, don't change the global
              # since the contract will be destroyed
              UNIT ; PAIR }
            { # otherwise continue
              DUP ; CDDADAR # T
```

```
               NOW ; COMPARE ; LT
               IF { FAIL } # Between Z + 24 and T
                  { # after T
                    DUP ; CDDADAR # T
                    PUSH uint64 86400 ; ADD # one day in second
                    NOW ; COMPARE ; LT
                    IF { # Between T and T + 24
                         # we only accept transactions from the buyer
                         DUP ; CADR ; # we must receive (Left "buyer")
                         IF_LEFT
                           { PUSH string "buyer" ; COMPARE ; EQ ;
                             IF { DUP ; CDADAR ; # amount already versed by the
buyer
                                  DIP { DUP ; CAAR } ; ADD ; # transaction
                                  # The amount must not exceed Q * K
                                  DUP ;
                                  DIIP { DUP ; CDDAAR ; # Q
                                         DIP { DUP ; CDDDAAR } ; MUL ; } ; # K
                                  DIP { COMPARE ; GT ; # new amount > Q * K
                                        IF { FAIL } { } } ; # abort or continue
                                  #  then we rebuild the globals
                                  DIP { DUP ; CDADDR } ; PAIR ; # seller amount
                                  PUSH uint32 0 ; PAIR ; # delivery counter at 0
                                  DIP { CDDR } ; PAIR ; # parameters
                                  # and return Unit
                                  UNIT ; PAIR }
                                { FAIL } } # (Left _)
                           { FAIL } } # (Right _)
                       { # After T + 24
                         # test if the required payment is reached
                         DUP ; CDDAAR ; # Q
                         DIP { DUP ; CDDDAAR } ; MUL ; # K
                         DIP { DUP ; CDADAR } ; # amount already versed by the buyer
                         COMPARE ; NEQ ;
                         IF { # not reached, pay the seller and destroy the contract
                              BALANCE ;
                              DIP { DUP ; CDDDDADR } # S
                              DIIP { CDR } ;
                              UNIT ; TRANSFER_TOKENS ; DROP ;
                              # and return Unit
                              UNIT ; PAIR }
                            { # otherwise continue
                              DUP ; CDDADAR # T
                              PUSH uint64 86400 ; ADD ;
                              PUSH uint64 86400 ; ADD ; # two days in second
                              NOW ; COMPARE ; LT
                              IF { # Between T + 24 and T + 48
                                   # We accept only delivery notifications, from W
                                   DUP ; CDDDDDR ; MANAGER ; # W
                                   SOURCE unit unit ; MANAGER ;
                                   COMPARE ; NEQ ;
                                   IF { FAIL } {} # fail if not the warehouse
                                   DUP ; CADR ; # we must receive (Right amount)
                                   IF_LEFT
                                     { FAIL } # (Left _)
                                     { # We increment the counter
                                       DIP { DUP ; CDAAR } ; ADD ;
```

```
                                    # And rebuild the globals in advance
                                    DIP { DUP ; CDADR } ; PAIR ;
                                    DIP { CDDR } ; PAIR ;
                                    UNIT ; PAIR ;
                                    # We test if enough have been delivered
                                    DUP ; CDAAR ;
                                    DIP { DUP ; CDDAAR } ;
                                    COMPARE ; LT ; # counter < Q
                                    IF { CDR } # wait for more
                                       { # Transfer all the money to the seller
                                         BALANCE ; # and destroy the contract
                                         DIP { DUP ; CDDDDADR } # S
                                         DIIP { CDR } ;
                                         UNIT ; TRANSFER_TOKENS ; DROP } } ;
                                  UNIT ; PAIR }
                                { # after T + 48, transfer everything to the buyer
                                  BALANCE ; # and destroy the contract
                                  DIP { DUP ; CDDDDAAR } # B
                                  DIIP { CDR } ;
                                  UNIT ; TRANSFER_TOKENS ; DROP ;
                                  # and return unit
                                  UNIT ; PAIR } } } } } } }
```

X - Full grammar

```
<data> ::=
  | <int constant>
  | <string constant>
  | <timestamp string constant>
  | <signature string constant>
  | <key string constant>
  | <tez string constant>
  | <contract string constant>
  | Unit
  | True
  | False
  | Pair <data> <data>
  | Left <data>
  | Right <data>
  | Some <data>
  | None
  | List <data> ...
  | Set <data> ...
  | Map (Item <data> <data>) ...
  | instruction
<instruction> ::=
  | { <instruction> ... }
  | DROP
  | DUP
  | SWAP
  | PUSH <type> <data>
  | SOME
  | NONE <type>
  | IF_NONE { <instruction> ... } { <instruction> ... }
  | PAIR
```

```
    | CAR
    | CDR
    | LEFT <type>
    | RIGHT <type>
    | IF_LEFT { <instruction> ... } { <instruction> ... }
    | NIL <type>
    | CONS
    | IF_CONS { <instruction> ... } { <instruction> ... }
    | EMPTY_SET <type>
    | EMPTY_MAP <comparable type> <type>
    | MAP
    | REDUCE
    | MEM
    | GET
    | UPDATE
    | IF { <instruction> ... } { <instruction> ... }
    | LOOP { <instruction> ... }
    | LAMBDA <type> <type> { <instruction> ... }
    | EXEC
    | DIP { <instruction> ... }
    | FAIL
    | NOP
    | CONCAT
    | ADD
    | SUB
    | MUL
    | DIV
    | ABS
    | NEG
    | MOD
    | LSL
    | LSR
    | OR
    | AND
    | XOR
    | NOT
    | COMPARE
    | EQ
    | NEQ
    | LT
    | GT
    | LE
    | GE
    | CAST
    | CHECKED_ABS
    | CHECKED_NEG
    | CHECKED_ADD
    | CHECKED_SUB
    | CHECKED_MUL
    | CHECKED_CAST
    | FLOOR
    | CEIL
    | INF
    | NAN
    | ISNAN
    | NANAN
    | MANAGER
```

```
   | TRANSFER_TOKENS
   | CREATE_ACCOUNT
   | CREATE_CONTRACT
   | NOW
   | AMOUNT
   | BALANCE
   | CHECK_SIGNATURE
   | H
   | STEPS_TO_QUOTA
   | SOURCE <type> <type>
<type> ::=
   | int8
   | int16
   | int32
   | int64
   | uint8
   | uint16
   | uint32
   | uint64
   | unit
   | string
   | tez
   | bool
   | key
   | timestamp
   | signature
   | option <type>
   | list <type>
   | set <comparable type>
   | contract <type> <type>
   | pair <type> <type>
   | or <type> <type>
   | lambda <type> <type>
   | map <comparable type> <type>
<comparable type> ::=
   | int8
   | int16
   | int32
   | int64
   | uint8
   | uint16
   | uint32
   | uint64
   | string
   | tez
   | bool
   | key
   | timestamp
```

XI - Reference implementation

The language is implemented in OCaml as follows:

- The lower internal representation is written as a GADT whose type parameters encode exactly the typing rules given in this specification. In other words, if a program written in this representation

is accepted by OCaml's typechecker, it is mandatorily type-safe. This of course also valid for programs not handwritten but generated by OCaml code, so we are sure that any manipulated code is type-safe.In the end, what remains to be checked is the encoding of the typing rules as OCaml types, which boils down to half a line of code for each instruction. Everything else is left to the venerable and well trusted OCaml.

- The interpreter is basically the direct transcription of the rewriting rules presented above. It takes an instruction, a stack and transforms it. OCaml's typechecker ensures that the transformation respects the pre and post stack types declared by the GADT case for each instruction.The only things that remain to we reviewed are value dependent choices, such as that we did not swap true and false when interpreting the If instruction.
- The input, untyped internal representation is an OCaml ADT with the only 5 grammar constructions: `String`, `Int`, `Seq` and `Prim`. It is the target language for the parser, since not all parsable programs are well typed, and thus could simply not be constructed using the GADT.
- The typechecker is a simple function that recognizes the abstract grammar described in section X by pattern matching, producing the well-typed, corresponding GADT expressions. It is mostly a checker, not a full inferer, and thus takes some annotations (basically the input and output of the program, of lambdas and of uninitialized maps and sets). It works by performing a symbolic evaluation of the program, transforming a symbolic stack. It only needs one pass over the whole program.Here again, OCaml does most of the checking, the structure of the function is very simple, what we have to check is that we transform a `Prim ("If", ...)` into an `If`, a `Prim ("Dup", ...)` into a `Dup`, etc.