Types

FunC has the following built-in types:

Atomic types

- `int` is the type of 257-bit signed integers. By default, overflow checks are enabled and lead to integer overflow exception.
- `cell` is the type of TVM cells. All persistent data in TON Blockchain is stored in trees of cells. Every cell has up to 1023 bits of arbitrary data in it and up to 4 references to other cells. Cells play a role of memory in stack-based TVM.
- `slice` is the type of cell slices. Cell can be transformed to a slice, and then the data bits and references to other cells from the cell can be obtained by loading them from the slice.
- `builder` is the type of cell builders. Data bits and references to other cells can be stored into a builder, and then the builder can be finalized to a new cell.
- `tuple` is the type of TVM tuples. Tuple is an ordered collection of up to 255 components, having arbitrary value types, possibly distinct.
- `cont` is the type of TVM continuations. Continuations are used for controlling the flow of TVM program execution. It is rather a

low-level object from the perspective of FunC, although somewhat paradoxically quite general.

Note that any of the types above occupy only a single entry in the TVM stack.

Absence of boolean type

In FunC booleans are represented as integers: `false` is represented as `0` and `true` is represented as `-1` (257 ones in binary notation).

Logical operations are done as bitwise operations. When a condition is checked, every non-zero integer is considered as `true` value.

Null values

By the value `null` of TVM type `Null` FunC represents absence of a value of some atomic type. Some primitives from the standard library may be typed as ones returning an atomic type and actually return `null`s in some cases. Others may be typed as ones excepting a value of an atomic type, but work fine with `null` values too. Such behavior is explicitly stated in the primitive specification. By default `null` values are prohibited and lead to a run-time exception.

In such a way, an atomic type `A` may be implicitly transformed into type `A^?` a.k.a. `Maybe A` (type-checker is agnostic to such a transformation).

Hole type

FunC has support for type inference. Types `_` and `var` represent type "holes", which can later be filled with some actual type during type checking. For example, `var x = 2;` is a definition of variable `x` equal to `2`. Type-checker can infer that `x` has type `int`, because `2` has type `int`, and left and right sides of an assignment must have equal types.

Composite types

Types can be composed in more complex ones.

Functional type

Types of the form `A -> B` represent functions with specified domain and codomain. For example, `int -> cell` is the type of functions of one integer argument, which return a TVM cell.

Internally values of such types are represented as continuations.

Tensor types

Types of the form `(A, B, ...)` essentially represent ordered collections of values of types `A`, `B`, `...`, which all together occupy more than one TVM stack entry.

For example, if a function `foo` has type `int -> (int, int)`, it means that the function takes one integer and returns pair of them.

A call of this function may look like `(int a, int b) = foo(42);`. Internally the function consumes one stack entry and leaves two of them.

Note that although in low-level perspective value `(2, (3, 9))` of type `(int, (int, int))` and value `(2, 3, 9)` of type `(int, int, int)` are represented in the same way as three stack entries `2`, `3` and `9`, for FunC type-checker they are values of **different** types: for example, code `(int a, int b, int c) = (2, (3, 9));` wouldn't be compiled.

Special case of tensor type is the **unit type** `()`. It is usually used for representing the fact that a function doesn't return any value, or has no arguments. For example, a function `print_int` would have type `int -> ()` and the function `random` has type `() -> int`. It has unique inhabitant `()`, which occupy 0 stack entries.

Type of form `(A)` is considered by type-checker as the same type as `A`.

Tuples types

Types of the form `[A, B, ...]` represent TVM tuples with concrete length and types of components, known in compile time. For example, `[int, cell]` is the type of TVM tuples, having length exactly 2, which first component is an integer, and the second is a cell. `[]` is the type of empty tuples (having the unique inhabitant – the empty tuple). Note that in contrast to unit type `()`, the value of `[]` occupy 1 stack entry.

Polymorphism with type variables

FunC has Miller-Rabin type system with support for polymorphic functions. For example, function

```
forall X -> (X, X) duplicate(X value) {
  return (value, value);
}
```

is a polymorphic function which takes a (single stack entry) value and returns two copies of this value. `duplicate(6)` will produce values `6 6`, and `duplicate([])` will produce two copies `[] []` of empty tuple.

In this example `X` is a type variable.

See more info on this topic in the functions section.

User-defined types

Currently FunC has no support for defining types except for type constructions described above.

Type width

As you may have noticed, every value of a type occupies some number of stack entries. If it is the same number for all values of the type, this number is called **type width**. Polymorphic functions currently can be defined only for types with fixed and known in advance type width.