lfe_types(7)

Robert Virding

2021

NAME

lfe_types - LFE Types and Functions Specifications

TYPES

This is a description of the type syntax.

LFE type	Erlang type
`(any)`	`any()`
`(none)`	`none()`
`(atom)`	`atom()`
`(integer)`	`integer()`
`(range i1 i2)`	`I1..I2`
`(float)`	`float()`
`(bitstring m n)`	`<<_:M,_:_*N>>`
`(binary)`	`<<_:0,_:_*8>>`
`(bitstring)`	`<<_:0,_:_*1>>`
`(nil)`	`[]  %% nil`
`...`	`...`
`(lambda any <type>)`	`fun((...) -> <type>)`
`(lambda () <type>)`	`fun(() -> <type>)`
`(lambda (<tlist>) <type>)`	`fun((<tlist>) -> <type>)`
`(map)`	`map()`
`#M()`	`#{}`
`#M(<key> <value> ...)`	`#{<pairlist>}`
`(tuple)`	`tuple()`
`#()`	`{}`
`#(<tlist>)`	`{<tlist>}`
`(UNION <tlist>)`	`<type> | <type>`

Apart from the predefined types in the Erlang type system we also have the following predefined types which cannot be redefined: `UNION`, `call`, `lambda` and `range`. The usage of `bitstring`, `tuple` and `map` have also been extended.

Note that the type `#M()` is the empty map and the type `#()` is the empty tuple. We can still use the older `(map <key valuelist>)` and `(tuple <tlist>)` formats when declaring types for maps and tuples.

The general form of bitstrings is `(bitstring m n)` which denotes a bitstring which starts with `m` bits and continues with segments of `n` bits. `(binary)` is a short form for a sequence of bytes while `(bitstring)` is a short form for a sequence of bits. There is currently no short form for an empty binary, `(bitstring 0 0)` must be used.

Type Declarations of User-Defined Types

`(deftype (type-name) type-def)`

`(defopaque (type-name) type-def)`

`(deftype (type-name par1 par2) type-def)`

`(defopaque (type-name par1 par2) type-def)`

For unparameterised types the parentheses around the type name are optional. An example:

```
(deftype foo (tuple 'foo (integer) (list)))

(deftype bar (tuple 'bar (integer) (list)))
```

Type Information in Record Declarations

`(defrecord rec (field1 default1 type1) (field2 default2) (field3))`

Fields with type annotations *MUST* give a default value and fields without type annotations get the default type `(any)`.

SPECIFICATIONS

Type specifications of User-Defined Functions

`(defspec (func-name arity) function-spec ...)`

where

```
function-spec = (arg-type-list ret-type)
function-spec = (arg-type-list ret-type constraint-list)
function-spec = #M(arg-types arg-type-list ret-type ret-type)
function-spec = #M(arg-types arg-type-list ret-type ret-type
                   contraints constraint-list)
arg-type-list = (arg-type ...)
constraint-list = (constraint ...)
constraint = (var var-type)
```

For multiple types add more function specs. The parentheses around the function name and the arity are optional. For example from the docs:

```
(defspec foo ([(pos_integer)] (pos_integer)))

(defspec (foo 1)
  ([(pos_integer)] (pos_integer))
  ([(integer)] (integer)))

(defspec (remove-if 2)
  ([(lambda ((any)) (boolean)) (list)] (list)))
```

Or with constraints:

```
(defspec id ((X) X ((X (tuple)))))

(defspec (foo 1)
  ([(tuple X (integer))] X ((X (atom))))
  ([(list Y)] Y ((Y (number)))))

(defspec (remove-if 2)
  ([pred (list)] (list) [(pred (lambda ((any)) (boolean)))]))
```

Note that a constraint variable doesn't need to start with an upper-case like an Erlang variable, though in some case it may be easier to read.

Note we are using the alternate list form with `[ ]` instead of parentheses to make it easier to see the function arguments.

Types and function specifications in the module definition

Types can also be defined in the module declaration, for example:

```
(defmodule this-module
  ...
  (type ((foo-type) (tuple 'foo (integer) (list)))
        ((bar-type) (tuple 'bar (integer) (list))))
  (spec ((foo 1) ([(integer)] (foo-type)))
        ((id 1) ([x] x ((x (tuple))))))
  ...)
```