Erlang/OTP

VOLUME II
The OTP Basics

MANUEL ÁNGEL RUBIO JIMÉNEZ

Edited by **Ayanda Dube**
Translated by **Ana M. Rubio**

Erlang/OTP

Volume II: The OTP Basics

Manuel Angel Rubio Jimenez

Translated by
Ana Maria Rubio Jimenez

Edited by
Ayanda Dube

Erlang/OTP
Volume II: The OTP Basics

Manuel Angel Rubio Jimenez

Translated by
Ana Maria Rubio Jimenez

Edited by
Ayanda Dube

Resumen

The development in Erlang is based on two bases well defined by its creators. The first is the power of the processes that the Erlang virtual machine implements. The second is the Concurrency Oriented Programming methodology that is facilitated with the OTP framework.

The use of Erlang is incomplete if both are not used. Its power is not discovered until processes have been used as the main source of programming and of the ***behaviors*** that the OTP framework integrates. To carry out a professional project in Erlang, knowledge and mastery of this technology is essential.

This second volume of Erlang/OTP covers the knowledge of this framework, the creation of professional projects with it and the Concurrency Oriented Programming or Actor Model as it is more widely known. Finish the necessary tour to know the powers of language and its platform. It gives the reader a tour of theory and practice, giving you even more tools and more useful code to launch into development.

In addition we cover Erlang/OTP 25, the logger system and a chapter dedicated to distributed programming.

[1] http://altenwald.org/curriculum-vitae/manuel
[2] http://creativecommons.org/licenses/by-nc-sa/3.0/

Chapter 1. Type Specification

The difference between dynamic and static type checking is the difference between X must be a car and X is a car.
— Bjarne Stroustrup

Before beginning with the exposition of the Actor Model and going fully into the explanation of OTP, it is convenient to review once again how the type specification works. Erlang is a dynamically typed language: type checking is done at run time and not at compile time. However, it is possible to specify the types of our parameters so that tools like Dialyzer or the compiler can have more information about the types of data that we handle in our code. In this way we can more easily locate typing errors in our programs.

In this chapter we will talk about the ***spec***, ***type*** and ***opaque*** directives. These directives allow us to specify data types and input and output parameters in the function definition.

1. Function Specification

When we want to use a function we may have doubts about what parameters we should use when calling it. Knowing only the number of parameters it receives we cannot determine either the expected order or the data type supported by each one:

```
get_value/2
```

We can provide more information by writing a specification. The definition of a specification provides information about the types of the arguments and the return:

```
-spec get_value(Key::atom(),
                List::[{atom(), string()}]) ->
      string() | undefined.
```

With this definition we already have enough information. Based on the names used for the parameters we know that ***Key*** can only be an atom and that ***List*** must contain tuples whose first element is an atom and the second a list of characters.

As return value we see that two options can be given. We can receive a list of characters or a specific atom: ***undefined***.

Note

If what we are defining can be of different types, we must use the pipe symbol or ***pipe*** (|) among the alternatives.

We can also specify the types polymorphically without mixing them. For example, a function that returns a binary if received a binary, but returns a list if received a list, is specified as follows:

```
-spec to_lower(String::binary()) -> binary();
              (String::string()) -> string().
```

There is yet another way to define functions: by separating the specification of the types from the function itself. An example:

```
-spec to_lower(String1) -> String2
      when String1 :: string(),
           String2 :: string();
              (Binary1) -> Binary2
      when Binary1 :: binary(),
           Binary2 :: binary().
```

The Erlang documentation often uses a similar syntax when specifying all data. You can see more information in Appendix A, ***Erlang Documentation: EDoc***.

2. Basic types

Erlang has many types that we can use directly in our specifications. The most elementary are the basic types of the system: atom, integer, float, binary, string, pid, node or reference.

To define lists, for example a list of atoms, we can use either the ***list(atom())*** or ***[atom()]*** syntax. With this notation we indicate that it is a list of atoms composed of zero, one or more atoms.

In the example of the previous section we already saw how to define the tuples: ***{atom(), string()}***. This tuple is made up of two elements, neither more nor less, since the size of the tuple is delimited in the definition.

We may also use registers in our specifications. To use, for example, a register called state in a specification, add ***#state{}*** directly without presenting its internal composition. In the previous volume we already saw that this compound data type is a bit special since it allows you to declare both the default values of each of its fields and their type. For example:

```
-record(state, {
```

```
    name :: pid() | atom(),
    description = "no description" :: string(),
    counter = 0 :: non_neg_integer()
}).
```

We can define the maps by indicating optional or mandatory keys and the types expected for their definitions. For example:

```
-type spec() :: #{ id := child_id(),
                   start := mfa(),
                   significant => boolean(),
                   restart => restart(),
                   shutdown => shutdown(),
                   type => worker(),
                   modules => [module()] }.
```

In this example we can see mandatory data to be present within the map such as `id` or `start` for being indicated with `:=` and the rest of the optional values when indicated with `=>`.

3. Compound types

We use types to give more semantics to the data we define. Erlang has another set of more concrete types to help us better define our data, functions, and records. These are:

`term() | any()`

It is a general data that matches with everyone. Both are equivalent. Actually ***term()*** is defined as ***any()***.

`binary()`

The binary type can be used for matching or directly through its syntax. But if we want to indicate that these are byte strings, we can use ***binary()***.

`bitstring()`

Bitstring also refers to binaries but in this case 1 bit in size while binary is 8 bits.

`boolean()`

Boolean is a small set which includes only the ***true*** or ***false*** atoms.

`byte()`

A number in the range 0..255 or 8 bits.

`char()`

A number in the range 0..16#10ffff. This limit is set by UTF-16, the largest character table which has its number of characters limited to 10FFFF (21 bits).

`number()`

It is an alias to indicate the choice: ***integer() | float()***.

`list()`

If nothing is indicated between parentheses it is equivalent to ***[any()]***.

`tuple()`

The generic form of a tuple of any number of elements.

`map()`

If nothing is indicated between parentheses it is equivalent to ***#{any() => any()}***.

`nonempty_list()`

A list that must have at least one value. If nothing is specified between parentheses it is equivalent to ***nonempty_list(any())***.

`nonempty_string()`

It's like ***string()*** but must contain at least one character.

`module()`

It is taken as an atom but its semantics refer to the name of a module. However, it does not check that the module actually exists.

`arity()`

Numeric range 0..255. Used to indicate the number of parameters (arity) that a function can accept.

`mfa()`

Definition equivalent to ***{module(), atom(), arity()}***.

`node()`

An atom that we will identify as the name of an Erlang node. As in the case of `module()` this also refers to an atom.

`timeout()`

It is a value that is used on a regular basis to define the time that a process has to wait for a response when making a call or in general to establish a wait. It is defined as: ***infinity | non_neg_integer()***.

`non_neg_integer()`

It is defined as the open range `0..` is a number equal to or greater than zero.

`pos_integer()`

It is defined as the open range `1..` is a number equal to or greater than one.

`neg_integer()`

It is defined as the open range `..-1` is a number equal to or lesser than one.

Important

Although there is the possibility to indicate most parameters and data as ***any()*** or ***term()*** this is considered a bad practice.

It is always preferable to use specific types instead of generic ones. Generic types can only be used when the data is not known and could be anything, such as when obtaining data using JSON or deserializing data obtained through network communication, file, or another method.

4. Literals and Ranges

Literals can also be used as part of the type definition. Although not all are valid, you can use:

`0 | 0..255 | ..-1 | 0..`

Numbers or numeric ranges. These ranges can be bounded or open.

`atom()`

Any atom can be used as mentioned in the definition of ***timeout()***.

`fun() | fun((...) -> Type) | fun(() -> Type) | fun((T) -> Type)`

Closures. In a general way as in the first example or indicating the input parameters and the return type. The use of the ellipsis indicates that the number of parameters is not taken into account.

<<>> | <<_:M>> | <<_:_*N>> | <<_:M, _:_*N>>

Binaries of different sizes.

5. Creating Data Types

To add even more semantics to the definition of functions we can create our own types. These can be a composition of the types seen above or simply a new name given to an existing type to make it more akin to the data it represents.

For example, let's create several types:

```
-type id() :: pos_integer().
-type user() :: binary().
-type domain() :: binary().
-type resource() :: binary().
-type jid() :: {user(), domain(), resource()}.
```

These types can be used to simplify or assist in the creation of new types and the specification of functions. In many cases, simply using these types does not require adding the parameter name.

Taking the `get_value/2` example seen above we can rewrite it like this:

```
-type key() :: atom().
-type value() :: string().
-type property() :: {key(), value()}.
-type properties() :: [property()].
-type result() :: value() | undefined.

-spec get_value(key(), properties()) -> result().
```

We can parameterize the definition of a generic type. These parameters can be used in the definition of more concrete derived types. For example, the type `keyword/0` is a `orddict/2` in which we have specified the type of the key and of the values that will be stored in the dictionary:

```
-type orddict(Key, Value) :: [{Key, Value}].

-type keyword() :: orddict(atom(), string()).
```

Data types can be exported via the ***-export_type*** directive. There are many modules that export their types to be used in our definitions such as ***proplists***. If we want to export the types defined in the previous case to use them in other modules, we add the directive:

```
-export_type([keyword/0]).
```

The logical thing is to export the types of data that we consider useful for other modules.

From the modules where we use these types, it will be enough to indicate the module from which they come. A fairly common case is the type `proplists:property/0`. We can use it like this:

```
-spec myfunc([proplists:property()]) -> ok.
```

The ***proplists*** module defines `property/0` as a tuple of two elements or an atom. By specifying that input parameter for `myfunc/1` we accept only property lists as a single parameter.

6. Dialyzer

Although the **erlc** compiler detects compilation errors in Erlang code, there is another command called **dialyzer**[1] that is responsible for detecting possible type errors that go unnoticed by the compiler, analyzing the type declaration.

Dialyzer performs a code review exchanging the types of the variables passed as parameters for all the possible types they can contain according to the definition of the functions, reporting any errors it finds. These errors appear if a call to a function is detected with a variable that, due to its initialization or origin, either cannot contain any of the accepted types or could contain some type that is not allowed.

Note

The use of Dialyzer is not mandatory and although it is strongly recommended, many do not use it regularly due to its slowness. However, in development environments like Emacs, VS Code, Sublime, and others that support language servers like Erlang LS[2], Dialyzer is executed with every code change and often be pretty fast.

A code that does not show Dialyzer warnings or errors may contain some bug, but for sure, a code with Dialyzer warnings or errors contains errors, even though they have not manifested themselves.

6.1. Generating PLT

Erlang code analysis is quite expensive if every function is analyzed, including those of the Erlang system itself. To speed up this process,

[1] Dialyzer is an acronym that stands for ***DIscrepancy AnalYZer for ERlang*** (or Discrepancy Analyzer for Erlang).
[2] https://erlang-ls.github.io/

Dialyzer provides the previous generation of some files PLT[3]. These files store the analysis performed on the base functions of the Erlang applications.

There are no PLT files generated by default for the codebase, so it is necessary to generate them. Knowing the dependencies that our code will have, we can generate an PLT file that includes the base applications. We do this with this command:

```
$ dialyzer --build_plt \ ❶
           --plt base.plt --apps erts kernel stdlib❷
  Compiling some key modules to native code... done in
 0m40.55s❸
  Creating PLT base.plt ...
Unknown functions:❹
  compile:file/2
  compile:forms/2
  compile:noenv_forms/2
  compile:output_generated/1
  crypto:block_decrypt/4
  crypto:start/0
Unknown types:❺
  compile:option/0
 done in 0m58.79s❻
done (passed successfully)
```

❶ Parameter to indicate that the file PLT is created.

❷ Applications to generate the file PLT.

❸ The time it takes to scan the applications. The more applications we add to the list, the longer this phase will take. It will also depend on the machine where we execute the command.

❹ If all the base applications are not added in the generation of the PLT file, it will indicate that there are functions not included. This is not an error, it is just a warning. Anything that is not parsed at the time the PLT file is created will have to be parsed at parse time along with our code.

❺ The same thing happens with types as with unparsed functions: They will be parsed along with our code.

[3]Persistent Lookup Table or Persistent Search Table

❻ This is the total time (58 seconds and 79 hundredths) that has been spent executing the command.

It is advisable to try to generically add all the base applications in the creation of the PLT file to avoid spending all that time in each check of our code.

We can generate as many PLT files as we want. We can generate an PLT file with the Erlang application base and a more specific one with the dependencies of our project, for example. Both files can be used later to check our code.

Note

By default, the generated file will be found in the base directory of the user who launched the command. We have changed this behavior by using the ***--plt*** parameter.

You can see more options for dialyzer by typing: **dialyzer --help**.

For more information see Appendix B, ***Command-line: Dialyzer***.

6.2. Checking the types

Once we have the PLT file we can proceed to check our code. We can carry out the execution on the directory where the code is located or on the files with the ***erl*** extension directly.

An example using the second case:

```
$ dialyzer --plt base.plt \ ❶
               tcpsrv.erl❷
  Checking whether the PLT base.plt is up-to-date... yes❸
  Proceeding with analysis... done in 0m0.63s❹
done (passed successfully)
```

❶ The PLT files that will be used to analyze the code. If we do not specify this parameter, the `~/.dialyzer.plt` file will be used.

❷ The files to be analyzed.

❸ It checks if the PLT file complies with the versions of the applications that the code to check will use.

❹ The total time to analyze the code.

The analysis can inform us that our types have been well used or throw errors. These errors tell us what types we should use in the function specification where there are bugs or inaccessible code.

6.3. Opaque types

When we define data types and we want to use these data types only for the functions specified in our code, we can resort to creating an opaque type.

The opaque type does not give visibility to the internal composition when displaying type errors with Dialyzer and makes it easier to display errors. This makes it easy to change the internal composition of the data type at any time without affecting external code.

For example when we define a data type like the following:

```
-type ascii() :: 0..127.
```

The system will use it internally as the range that we have defined and the errors and dialyzer information will show us that same range. The name is ignored. In an error caused on purpose we can see the output:

```
$ dialyzer --plt base.plt ascii.erl
  Checking whether the PLT base.plt is up-to-date... yes
  Proceeding with analysis...
ascii.erl:19: Function in_and_out_code/1 has no local return
ascii.erl:19: The call ascii:out_code(byte()) breaks the
 contract (binary()) -> binary()
 done in 0m0.52s
done (warnings were emitted)
```

One method so that Dialyzer and the system itself does not go beyond the definition we have given it, considering ***ascii()*** different from ***0..127***, is to make our data type opaque. This is achieved by using the type definition as follows:

```
-opaque ascii() :: 0..127.
```

This way Erlang considers that the data type we are using is ***ascii()*** and will not try to resolve it. Likewise, if an error or warning occurs in Dialyzer, it will show us this data and not the range.

For example, using the ***ascii()*** type we will see errors where the output will be displayed like this:

```
$ dialyzer --plt base.plt ascii.erl
  Checking whether the PLT base.plt is up-to-date... yes
  Proceeding with analysis...
ascii.erl:19: Function in_and_out_code/1 has no local return
ascii.erl:19: The call ascii:out_code(ascii:ascii()) breaks
 the contract (binary()) -> binary()
 done in 0m0.59s
done (warnings were emitted)
```

We should always export opaque types because they are used by other modules. We'll add an ***export_type*** directive very similar to the familiar ***export*** directive:

```
-export_type([ascii/0]).
```

In this way we can use the specification from other modules using ***ascii:ascii()***.

6.4. Types with parameters

Although they are not widely used because they add a level of complexity that is sometimes unnecessary, parameters can be defined in the types to be used within the type itself. An example would be the type `list/1` defined like this:

```
-opaque list(T) :: [T].
```

Doing the substitution at the time of use we can write something like ***list(binary())***, for example, and it would be equivalent to ***[binary()]***.

We can not only use them for lists, but also for tuples, records, etc. Let's look at another example with tuples. The property list:

```
-opaque proplist(T) :: [{atom(), T} | atom()].
-opaque proplist(I, T) :: [{I, T} | I].
```

This time we have not created just one type with one parameter, but two. In case of passing only one parameter we will obtain a list of tuples of two elements whose first element has to be an atom. In the case of passing two parameters we can define the content of both the first and the second parameter.

6.5. Incorrect Specifications

When we make mistakes with the types is when much more information starts to come out in the dialyzer execution. For example, if we change the types from ***inet:port_number()*** to ***integer()***:

```
$ dialyzer --plt base.plt tcpsrv_errors.erl
  Checking whether the PLT base.plt is up-to-date... yes
  Proceeding with analysis...
tcpsrv_errors.erl:14: Invalid type specification for function
 tcpsrv_errors:srv_loop/1. The success typing is (port()) ->
 no_return()
 done in 0m0.64s
done (warnings were emitted)
```

The error tells us that we must use ***port()*** as parameter in line 14 and the return will be of type ***no_return()***.

Important

For some functions when there is no return because the function makes a call to `exit/1`, `halt/1` or `throw/1` the return is indicated as ***no_return()***.

Dialyzer informs us of the line where the possible error is located for its correction. It also gives us information to be able to correct the error. In this case it is a type specification error. But in many cases the types may be correct and we need to correct the calls to the function or the internal treatment of the data.

6.6. Dialyzer with rebar3

We have seen the need to create an PLT file manually and maintain it with the base applications. However, a project is often more complex and contains dependencies. Automating the generation of these files to speed up the testing of our code is a priority if we work regularly with Dialyzer.

In this task **rebar3** helps us enormously. Running `rebar3 dialyzer` generates the PLT file, both to create it and to check that it still has the last valid content and performs the type check as the last part of the execution.

If we do the test by creating a project and running the command, we will see an output like the following:

```
$ rebar3 dialyzer
===> Verifying dependencies...
===> Analyzing applications...
===> Compiling tcpsrv
===> Dialyzer starting, this may take a while...
===> Updating plt...
===> Resolving files...
===> Updating base plt...
===> Resolving files...
===> Building with 206 files in /Users/bombadil/.cache/rebar3/
rebar3_25.0.4_plt...
```

```
===> Copying /Users/bombadil/.cache/rebar3/rebar3_25.0.4_plt
 to /Users/bombadil/tcpsrv/_build/default/rebar3_25.0.4_plt...
===> Checking 206 files in _build/default/rebar3_25.0.4_plt...
===> Doing success typing analysis...
===> Resolving files...
===> Analyzing 1 files with _build/default/
rebar3_25.0.4_plt...
```

The generation of the PLT file can be useful if we want to manually launch the command against a single module at any time:

```
$ dialyzer --plt _build/default/rebar3_25.0.4_plt src/
tcpsrv.erl
  Checking whether the PLT _build/default/rebar3_25.0.4_plt is
 up-to-date... yes
  Proceeding with analysis... done in 0m0.14s
done (passed successfully)
```

So it's a good idea to create projects with **rebar3** and take advantage of its ease of using tools like **Dialyzer**.