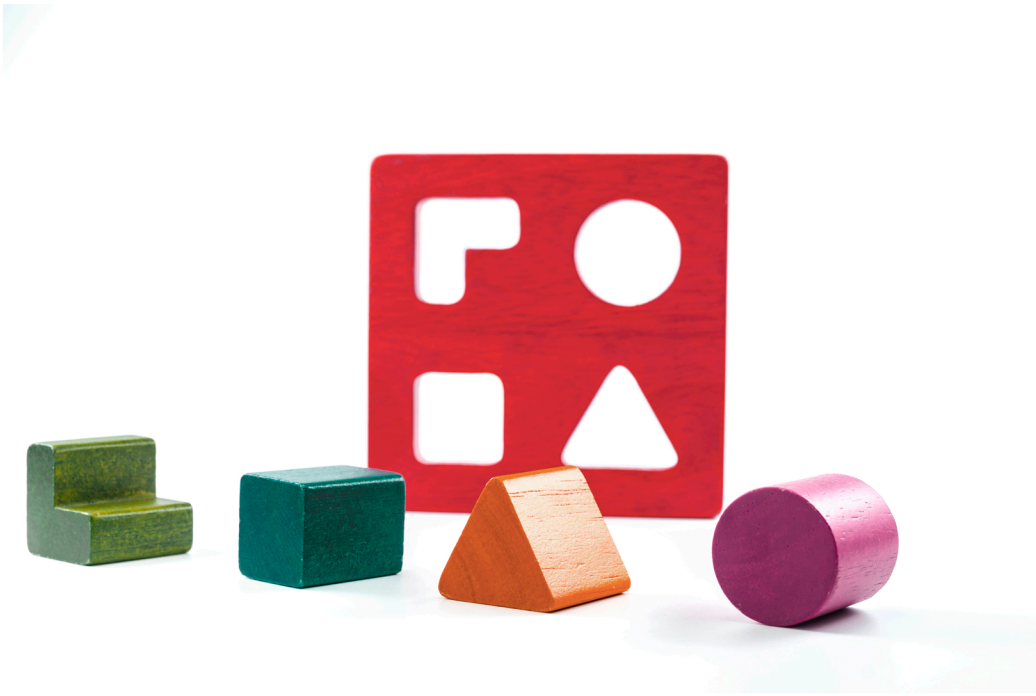
Erlang/OTP

VOLUMEN II
Las bases de OTP



MANUEL ÁNGEL RUBIO JIMÉNEZ

Erlang/OTP

Volumen II: Las bases de OTP

MANUEL ÁNGEL RUBIO JIMÉNEZ

El desarrollo en Erlang se fundamenta en dos bases bien definidas por su autor Joe Armstrong. La primera son la potencia de los procesos que implementa la máquina virtual de Erlang y la segunda es la metodología de Programación Orientada a la Concurrencia que se facilita con el framework OTP.

El uso de Erlang queda incompleto si no se emplean ambas cosas. Su potencia no se descubre hasta no hacer uso de los procesos como fuente principal de programación, y de los *comportamientos* que integra el framework OTP. Para la realización de un proyecto profesional es indispensable el conocimiento y dominio de esta tecnología.

Este segundo volumen de Erlang/OTP cubre el conocimiento de este framework, la creación de proyectos profesionales con él y la Programación Orientada a la Concurrencia o Modelo Actor como se conoce más extensamente. Termina el recorrido necesario para conocer las potencias del lenguaje y de su plataforma. Da al lector un recorrido por la teoría y la práctica acercando aún más herramientas y más código útil para lanzarse a desarrollar.

Revisión Técnica

José Luis Gordo Romero | Juan Sebastián Pérez Herrero

ISBN 978-84-945523-2-8

Capítulo 1. Especificación de Tipos

La diferencia entre la comprobación de tipos dinámicos y estáticos es la diferencia entre X debe ser un coche y X es un coche.
—Bjarne Stroustrup

Antes de empezar con la exposición del Modelo Actor y entrar de lleno en la explicación de OTP, conviene revisar una vez más como funciona la especificación de tipos. Erlang es un lenguaje de tipado dinámico: la comprobación de tipos se realiza durante la ejecución y no durante la compilación. No obstante es posible especificar los tipos de nuestros parámetros para que herramientas como Dialyzer o el compilador puedan tener más información sobre los tipos de datos que manejamos en nuestro código. De esta forma podemos localizar más fácilmente errores de tipado en nuestros programas.

En este capítulo hablaremos sobre las directivas ***spec***, ***type*** y ***opaque***. Estas directivas nos permiten especificar tipos de datos y parámetros de entrada y salida en la definición de funciones.

1. Especificación de Funciones

Cuando queremos utilizar una función nos pueden surgir dudas sobre qué parámetros debemos emplear al llamarla. Sabiendo sólo el número de parámetros que recibe (o arity en inglés) no podemos determinar ni el orden esperado ni el tipo de datos soportado por cada uno:

```
get_value/2
```

Si al definir la función usamos una especificación con tipos podemos reconocer fácilmente los parámetros de entrada y los tipos que retorna:

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

Con esta definición ya tenemos suficiente información. En base a los nombres con que hemos llamado los parámetros sabemos que ***Key*** solo puede ser un átomo y que ***Lista*** debe contener tuplas cuyo primer elemento sea un átomo y el segundo una lista de caracteres.

Como valor devuelto vemos que se pueden dar dos opciones. Podemos recibir una lista de caracteres o bien un átomo específico: ***undefined***.

Nota

Si lo que estamos definiendo puede ser de diferentes tipos, debemos emplear el símbolo tubería (|) entre las alternativas.

También podemos especificar los tipos de forma polimórfica sin mezclarlos. Por ejemplo, una función que retorna un binario si recibe un binario, pero que retorna una lista si recibe una lista, se especifica de la siguiente manera:

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

Existe todavía otra forma más de definir las funciones: separando la especificacion de los tipos de la propia función. Un ejemplo:

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

En la documentación de Erlang suelen emplear una sintaxis parecida al especificar todos los datos. Puedes ver más información en ApéndiceA, *Documentación de Erlang: EDoc*.

2. Tipos básicos

Erlang dispone de muchos tipos que podemos emplear directamente en nuestras especificaciones. Los más elementales son los tipos básicos del sistema: atom, integer, float, binary, string, pid, node o reference.

Para definir listas, por ejemplo una lista de átomos, podemos emplear tanto la sintaxis ***list(atom())*** como ***[atom()]***. Con esta notación indicamos que se trata de una lista de átomos compuesta por cero, uno o más átomos.

En el ejemplo del apartado anterior ya vimos cómo definir las tuplas: ***{atom(), string()}***. Esta tupla se compone de dos elementos, ni más ni menos, ya que el tamaño de la tupla viene delimitado en la definición.

También podemos utilizar registros en nuestras especificaciones. Para emplear por ejemplo un registro llamado state en una especificación, se añade sin presentar su composición interna, directamente ***#state{}***. En el volumen anterior ya vimos que este tipo de dato compuesto es un poco

especial ya que permite declarar tanto los valores por defecto de cada uno de sus campos como su tipo. Por ejemplo:

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

3. Tipos compuestos

Usamos tipos para dar mayor semántica a los datos que definimos. Erlang dispone de otro conjunto de tipos más concretos para ayudarnos a definir mejor nuestros datos, funciones y registros. Estos son:

term() | any()

Es un término. Un dato general en el que entra todo. Ambos son equivalentes. En realidad ***term()*** está definido como ***any()***.

binary()

El tipo binario se puede emplear con concordancias o directamente a través de su sintaxis. Pero si queremos indicar que se trata de cadenas de bytes, podemos emplear ***binary()***.

bitstring()

Bitstring se refiere también a los binarios pero en este caso de 1 bit de tamaño mientras que binary es de 8 bits.

boolean()

Los átomos ***true*** o ***false***, verdadero y falso respectivamente.

byte()

Un número en el rango 0..255 o de 8 bits.

char()

Un número en el rango 0..16#10ffff. Este limite se establece por UTF-16, la tabla de caracteres más grande que tiene limitado su número de caracteres a 10FFFF (21 bits).

number()

Es un alias para indicar la elección: ***integer() |float()***.

list()

Si no se indica nada entre paréntesis es equivalente a [***any()***].

tuple()

La forma genérica de una tupla de cualquier número de elementos.

nonempty_list()

Una lista que debe de tener al menos un valor. Si no se especifica nada entre paréntesis es equivalente a ***nonempty_list(any())***.

nonempty_string()

Es como ***string()*** pero debe contener al menos un carácter.

module()

Se toma como un átomo pero su semántica nos refiere el nombre de un módulo.

arity()

Rango numérico 0..255. Empleado para indicar el número de parámetros (aridad) que puede aceptar una función.

mfa()

Definición equivalente a ***{module(), atom(), arity()}***.

node()

Un átomo que identificaremos como el nombre de un nodo Erlang.

timeout()

Es un valor que se emplea de forma regular para definir el tiempo que un proceso tiene que esperar a una respuesta al realizar una llamada o en general para establecer una espera. Se define como: ***'infinity' | non_neg_integer()***.

non_neg_integer()

Se define como el rango abierto 0.. es un número igual o mayor a cero.

pos_integer()

Se define como el rango abierto 1.. es un número igual o mayor a uno.

neg_integer()

Se define como el rango abierto ..-1 es un número igual o menor a menos uno.

Aviso

Aunque existe la posibilidad de indicar la mayoría de parámetros y datos como ***any()*** o ***term()*** esto se considera una mala práctica.

Siempre es preferible emplear tipos específicos en lugar de genéricos. Los tipos genéricos pueden emplearse únicamente cuando el dato no es conocido y podría ser cualquiera, como cuando se obtienen datos mediante un JSON o se deserializa un dato obtenido mediante comunicación de red, fichero u otro método.

4. Literales y Rangos

Como parte de la definición de tipos se pueden emplear también literales. Aunque no todos son válidos, se pueden emplear:

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

Números o rangos numéricos. Estos rangos pueden estar acotados o abiertos.

atom

Átomos. Se puede emplear cualquier átomo como ya vimos en la definición de ***timeout()***.

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

Clausuras. De forma general como en el primer ejemplo o indicando los parámetros de entrada y el tipo de retorno. El uso de los puntos suspensivos indica que no se tiene en cuenta el número de parámetros.

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

Binarios de distintos tamaños.

5. Creando Tipos de Datos

Para agregar todavía mayor semántica a la definición de funciones podemos crear nuestro propios tipos. Éstos pueden ser una composición

de los tipos vistos anteriormente o simplemente un nuevo nombre dado a un tipo existente para hacerlo más afín al dato que representa.

Por ejemplo, vamos a crear varios tipos:

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

Estos tipos se pueden emplear para simplificar o ayudar en la creación de nuevos tipos y la especificación de funciones. En muchos casos, utilizando simplemente estos tipos no es necesario agregar el nombre del parámetro.

Tomando el ejemplo de `get_value/2` visto anteriormente podemos reescribirlo de esta forma:

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

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

Podemos parametrizar la definición de un tipo genérico. Estos parámetros pueden utilizarse en la definición de tipos derivados más concretos. Por ejemplo, el tipo `mydict/0` es un `orddict/2` en el que hemos especificado el tipo de la clave y de los valores que se almacenarán en el diccionario:

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

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

Los tipos de datos pueden ser exportados a través de la directiva ***-export_type***. Hay muchos módulos que exportan sus tipos haciéndolos disponibles para nuestras definiciones como por ejemplo ***proplists***. Si quisiéramos exportar los tipos definidos en el caso anterior para poder emplearlos en otros módulos, agregaríamos la directiva:

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

Lo lógico es exportar los tipos de datos que consideremos que pueden ser empleados en otros módulos.

En los módulos donde queramos emplear estos tipos nos bastará con indicar el módulo del que provienen. Un caso bastante común es el tipo `proplists:property/0`. Podemos emplearlo de esta forma:

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

El módulo ***proplists*** define `property/0` como una tupla de dos elementos o un átomo. Especificando ese parámetro de entrada para `myfunc/1` aceptamos sólo listas de propiedades como único parámetro.

6. Dialyzer

Aunque el compilador **erlc** detecta errores de compilación en el código Erlang, existe otro comando llamado **dialyzer**[1] que se encarga de detectar posibles errores de tipado que pasan desapercibidos al compilador, analizando la declaración de tipos.

Dialyzer realiza una revisión del código intercambiando los tipos de las variables pasadas como parámetros por todos los tipos posibles que pueden contener según la definición de las funciones, reportando los errores que encuentre. Estos errores emergen si se detecta alguna llamada a una función con una variable que por su inicialización o procedencia, o bien no puede contener ninguno de los tipos aceptados o bien podría contener algún tipo no permitido.

6.1. Generando PLT

El análisis del código de Erlang es bastante costoso si se analiza cada función, incluidas las del propio sistema Erlang. Para acelerar este proceso Dialyzer provee la generación previa de unos ficheros PLT[2]. En estos ficheros se almacena el análisis realizado sobre las funciones base de las aplicaciones de Erlang.

No hay ficheros PLT generados por defecto para el código base, por eso es necesario crearlos. Sabiendo las dependencias que tendrá nuestro código podemos generar un fichero PLT que incluya las aplicaciones base. Esto lo hacemos con este comando:

```
$ dialyzer --build_plt❶ --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
```

[1] El comando Dialyzer es un acrónimo que significa ***DIscrepancy AnalYZer for ERlang*** (o Analizador de Discrepancias para Erlang).
[2] Persistent Lookup Table o Tabla de Búsqueda Persistente

```
  crypto:block_decrypt/4
  crypto:start/0
Unknown types:❺
  compile:option/0
 done in 0m58.79s❻
done (passed successfully)
```

❶ Parámetro para indicar que se cree el fichero PLT.

❷ Aplicaciones para generar el fichero PLT.

❸ El tiempo que se demora en analizar las aplicaciones. Cuantas más aplicaciones agreguemos en la lista más tiempo tomará esta fase. También dependerá de la máquina donde ejecutemos el comando.

❹ Si no se agregan todas las aplicaciones base en la generación del fichero PLT se indicará que hay funciones no incluidas. No se trata de un error, es solo un aviso. Todo lo que no sea analizado en el momento de crear el fichero PLT tendrá que ser analizado en el momento de análisis junto con nuestro código.

❺ Con los tipos sucede lo mismo que con las funciones no analizadas: Se analizarán junto con nuestro código.

❻ Este es el tiempo total que se ha empleado en la ejecución del comando.

Es recomendable intentar agregar de forma genérica todas las aplicaciones base en la creación del fichero PLT para evitarnos gastar todo ese tiempo en cada comprobación de nuestro código.

Podemos generar tantos ficheros PLT como queramos. Podemos generar un fichero PLT con la base de aplicaciones de Erlang y otro más específico con las dependencias de nuestro proyecto, por ejemplo. Ambos ficheros podremos usarlos después para comprobar nuestro código.

Nota

Por defecto el fichero generado se encontrará en el directorio base del usuario que lanzó el comando. Hemos cambiado este comportamiento empleando el parámetro ***--plt***.

Puedes ver más opciones para dialyzer escribiendo: **dialyzer --help**.

Para más información ver Apéndice B, ***Línea de Comandos: Dialyzer***.

6.2. Comprobando los tipos

Una vez tenemos el fichero PLT podemos proceder a comprobar nuestro código. Podemos realizar la ejecución sobre el directorio donde se encuentra el código o sobre los ficheros `erl` directamente.

Un ejemplo usando el segundo caso:

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

❶ Los ficheros PLT que se emplearán para analizar el código. Si no indicamos este parámetro se utilizará el fichero `~/.dialyzer.plt`.

❷ Los ficheros que se analizarán.

❸ Comprueba si el fichero PLT cumple con las versiones de las aplicaciones que empleará el código a comprobar.

❹ El tiempo total en analizar el código.

El análisis nos puede informar que nuestros tipos han sido bien empleados o arrojar errores. Estos errores nos avisan de qué tipos deberíamos emplear en la especificación de las funciones allá donde se encuentran fallos o código inaccesible.

6.3. Tipos opacos

Cuando definimos tipos de datos y queremos emplear estos tipos de datos únicamente por las funciones especificadas en nuestro código podemos recurrir a crear un tipo opaco.

El tipo opaco no da visibilidad de la composición interna a la hora de mostrar errores de tipos con Dialyzer y facilita la visualización de errores. Esto facilita el cambio de la composición interna del tipo de dato en cualquier momento sin afectar a código externo.

Por ejemplo cuando definimos un tipo de dato como el siguiente:

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

El sistema lo empleará internamente como el rango que hemos definido y los errores y la información de dialyzer nos mostrará ese mismo rango.

El nombre es ignorado. En un error provocado adrede podemos ver la salida:

```
$ 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)
```

Un método para que Dialyzer y el sistema en sí no vaya más allá de la definición que le hemos dado, considerando ***ascii()*** diferente de ***0..127***, es hacer opaco nuestro tipo de dato. Esto se consigue empleado la definición del tipo de la siguiente forma:

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

De este modo Erlang considera que el tipo de dato que estamos usando es ***ascii()*** y no intentará resolverlo. Asímismo si sucediera algún error o warning en Dialyzer nos mostrará este dato y no el rango.

Por ejemplo, usando el tipo ***ascii()*** veremos errores donde se mostrará la salida de esta forma:

```
$ 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)
```

Siempre debemos exportar los tipos opacos porque son utilizados por otros módulos. Agregaremos una directiva ***export_type*** muy similar a la ya conocida ***export***:

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

De esta forma podemos emplear desde otros módulos la especificación usando ***ascii:ascii()***.

6.4. Tipos con parámetros

Aunque no son muy empleados porque agregan un nivel de complejidad en ocasiones innecesario, se pueden definir parámetros en los tipos para ser empleados dentro del propio tipo. Un ejemplo sería el tipo ***list/1*** definido así:

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

Haciendo la sustitución en el momento de ser usado podemos escribir algo como ***list(binary())***, por ejemplo, y sería equivalente a ***[binary()]***.

No solo los podemos emplear para listas, sino también para tuplas, registros, etc. Veamos otro ejemplo con tuplas. La lista de propiedades:

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

En esta ocasión no hemos creado solo un tipo con un parámetro, sino dos. En caso de pasar solo un parámetro obtendremos una lista de tuplas de dos elementos cuyo primer elemento tiene que ser un átomo. En el caso de pasar dos parámetros podemos definir el contenido tanto del primer como del segundo parámetro.

6.5. Especificaciones Incorrectas

Cuando cometemos errores con los tipos es cuando comienza a salir mucha más información en la ejecución de dialyzer. Por ejemplo, si cambiamos los tipos de ***inet:port_number()*** a ***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)
```

El error nos indica que debemos de emplear en la línea 14 como parámetro ***port()*** y el retorno será de tipo ***no_return()***.

Importante

Para algunas funciones cuando el retorno no es importante se indica su devolución como ***no_return()***. Principalmente suele no tener retorno cuando el final de la función sea dado por un lanzamiento de excepción, la ejecución de `halt/1` o `exit/1`.

Dialyzer nos informa la línea en la que se encuentra el posible error para su corrección. Además nos arroja información para poder subsanar el error. En este caso es un error de especificación de tipos. Pero en muchos casos los tipos pueden ser correctos y necesitemos corregir las llamadas a la función o el tratamiento interno de los datos.