Programación Declarativa Avanzada

Erlang 3

Juan Rodríguez Hortalá

Listas intensionales

Similares a las listas intensionales de Haskell Una sintaxis compacta para definir listas, parecida a la notación de conjuntos empleada en matemáticas

```
[ Expression || Generators, Guards, Generators, ... ]
```

- los generadores son de la forma `Pattern <- List`, donde '`<-`' se puede entender como el símbolo de pertenencia para conjuntos
- las guardas son iguales que las de las definiciones de función, devuelven `true` o `false`
- `Expression` especifica la forma que tendrán los elementos del resultado

```
> [2 * X || X <- lists:seq(0, 3), X > 1, Y <- lists:seq(1, X)] .
> [{2 * X, Y} || X <- lists:seq(0, 3), X > 1, Y <- lists:seq(1, X)] .
> (fun(Libro, Db) -> [{P, D} || {L, P, D} <- Db, L == Libro] end)("Programming Erlang",
[{"Programming Erlang", "Juan", 30}, {"Erlang Programming", libre, 0}, {"Programming Erlang",
"Enrique", 20}, {"Erlang and OTP in Action", "Pepe", 40}]) .
```

Muchas funciones estándar sobre listas se pueden definir usando listas intensionales

```
map(F,Xs)    -> [ F(X) || X <-Xs ].
filter(P,Xs) -> [ X || X <-Xs, P(X) ].
append(Xss)  -> [ X || Xs <- Xss, X <- Xs ].
```

Records

Es habitual implementar un servidor como una función recursiva final con un argumento que es su estado

- el estado será un término Erlang complejo, normalmente una tupla con varias otras tuplas anidadas
- el servidor ejecutará una instrucción `receive`, y al recibir un mensaje ejecutará diversas alternativas según el mensaje recibido y el estado
- accedemos al estado mediante pattern matching, por lo que **la estructura del estado queda reflejada en el código**

En un servidor que representa una biblioteca el estado puede ser una lista de tuplas `{Libro, Persona, Dias}` utilizaremos varias funciones auxiliares como la siguiente

```
prestamos_de_libro(Libro, Db) -> [{P, D} || {L, P, D} <- Db, L == Libro] .
```

¿Qué hacer si necesitamos añadir más información al estado?

Lo normal es añadir más elementos a la tupla del estado

- el estado pasa a ser listas de tuplas `{Libro, Persona, Dias, Facultad}`
- entonces hay que cambiar el código de todas las funciones auxiliares

Además en programas reales las tuplas que representan los estados son muy grandes y anidadas, por lo que aumenta mucho la probabilidad de cometer errores

Records

Records azúcar sintáctico de las tuplas que permiten escribir programas más legibles y extensibles, y con menos errores se traducen a tuplas

- nos permiten abstraernos de la forma concreta de la tuplas
- equivalente Erlang de los struct de C

Declaración de un record

se pueden especificar valores por defecto

```
-record(prestamo, {libro, cliente, dias=0, facultad}) .
```

forma general: los records pueden contener a su vez otros records

```
-record{name, {field1 [ = default1 ], field2 [ = default2 ], ... fieldn [ = defaultn ] }
```

Creación de una instancia del record

no hace falta respetar el orden de los campos

```
> #prestamo{cliente = "Enrique", dias = 20, libro = "Programming Erlang", facultad = fdi}
```

si no se especifica algún campo se usa el valor por defecto o `undefined` si este no se ha especficado

```
> #prestamo{libro = "Erlang Programming", cliente = "libre"},
```

Demo `records.erl`

Para usar los records desde la consola

`rr(Modulo)` carga todas las definiciones de records en `Modulo` para poder usarlas desde consola

`rd(name, {field1 [ = default1 ], ... fieldn [ = defaultn ])` define el record para la consola

```
> rd(fruta, {nombre,color=naranja,forma}) .
> #fruta{nombre = "Naranja", forma = redonda} .
```

Records

Acceso a los campos de un record `RecordExp#name.fieldName`

```
-record(prestamo, {libro, cliente, dias=0, facultad}) .
> P = #prestamo{libro = "Programming Erlang",cliente = "Juan", dias = 30, facultad = fdi} .
> P#prestamo.libro .
```

en un acceso a campos de un record ni **`name`** ni **`fieldName`** pueden ser variables

```
> L = libro, P#prestamo.L .
```

Demo `records:titulos_en/1, records:entradas_para/2`

Modificación de los campos de un record

```
> P2 = P#prestamo{cliente = "Pepe", dias = 0} .
```

Demo `records:devuelve_libro/3`

Pattern matching de una instancia de un record

```
> #prestamo{cliente = C} = P2 .
> C .
```

Inciso: as-patterns similares a los de Haskell

```
> (fun(X = {0, _}) -> X end)({0, hola}) .
```

Demo `records:devuelve_libro_matching/3, records:prestamos_de_libro/2`

Macros

Similares a los macros de C permiten escribir abreviaturas de expresiones Erlang que el preprocesador de Erlang ***expande*** en tiempo de compilación una especie de forma limita de función que se ejecuta en tiempo de compilación

Se definen empleando la directiva -define, y en su forma más sencilla se usan para definir constantes la costumbre es que los nombres de las macros vayan en mayúsculas estilo C

estos son ejemplos de usos típicos al escribir un módulo para un servidor

```
   % nombre con el que registraremos al pid del servidor
-define(SERVER_NAME, biblioteca) .
   % timeout que usaremos en los receive de las llamadas síncronas al servidor
-define(TIMEOUT, 1000) .
```

También se pueden definir macros con parámetros como suele pasar en cualquier lenguaje, no se debe abusar de las macros porque a menudo dificultan la comprensión del programa (por ejemplo los errores en la compilación de la expansión de una macro se darán en los usos de la macro, no es su definición)

```
-define(TIMES(X,Y), X * Y) .
```

Un uso típico legítimo de las macros con parámetros consiste en emplearlas para imprimir mensajes de depuración

```
% solo una de las dos definiciones de macro estará sin comentar, activando o desactivando de
esta manera el modo de depuración
%-define(DBG(Str, Args), ok).
-define(DBG(Str, Args), io:format(Str, Args)).
```

Para usar una macro utilizamos `?MACRO` el preprocesador sustituirá `?MACRO` por su definición durante la compilación

```
init() -> register(?SERVER_NAME, spawn(fun loop/0)) .
```

Demo `macros.erl`

Macros

Algunas macros predefinidas interesantes

`?MODULE` se expande al nombre del módulo en que se usa, útil para llamadas recursivas en los bucles de los servidores, para activar el cambio de código en caliente

`?MODULE_STRING` se expande al nombre del módulo en que se use, como un string

`?FILE` se expande al nombre del fichero en que se use, como un string

`?LINE` se expande al número de linea en que aparece esta instrucción, útil para depuración

`?MACHINE` se expande al nombre de la máquina virtual que se está usando

Demo `macros:info/0`

También se permiten macros condicionales al estilo de C su evaluación depende de otras macros o de flags que se pasan al compilador

`-undef(Macro)` Causes the macro to behave as if it had never been defined.

`-ifdef(Macro)` Evaluate the following lines only if Macro is defined.

`-ifndef(Macro)` Evaluate the following lines only if Macro is not defined.

`-else` Only allowed after an ifdef or ifndef directive. If that condition was false, the lines following else are evaluated instead.

`-endif.` Specifies the end of an ifdef or ifndef directive.

Para activar o desactivar las flags se puede usar `c(macros, [{d, debug}])` (define) y `c(macros, [{u, debug}])` (undefine) o las opciones correspondientes de `compile:file/2`.

Demo `macros:condicional/0, macros:condicional2/0`

Ficheros de cabecera Erlang

Las definiciones de records y macros se suelen incluir en ficheros de cabecera con extensión **`.hrl`** que luego son importados por todos los módulos que necesiten dichas definiciones

```
%% includes.hrl
-record(prestamo, {libro, cliente, dias=0, facultad}) .

-ifdef(debug).
     -define(DBG(Call), (fun(V) -> io:format("llamada ~p = ~p~n", [??Call, V]), V end)
(Call)) .
-else.
     -define(DBG(Call), Call).
-endif.

%% includes.erl
-module(includes) .
-include("includes.hrl") .
....
```

en este ejemplo usamos `??Call` que convierte el argumento `??Call` de la macro **`DBG`** en un string

Demo `includes.erl, includes.hrl`

Módulo Dict, ETS y Dets

Equivalentes Erlang de la librería de colecciones de Java

Módulo `dict` implementa un diccionario de claves-valor

- La representación del diccionario no está definida y puede cambiar sin previo aviso (TAD)
- Rendimiento modesto, sólo para diccionarios pequeños

`dict:new()` crea un nuevo diccionario vacío

`dict:append(Key, Value, Dict)` añade `Value` a la lista de valores asociados a `Key` en `Dict` si el valor asociado no es una lista salta una excepción

`dict:store(Key, Value, Dict)` añade o sustituye el valor asociado a Key en Dict1 por Value

`dict:find(Key, Dict)` busca el valor asociado a Key en Dict devolviendo {ok, Value} si lo encuentra o error en otro caso

`dict:to_list/1`, `dict:from_list/1` funciones de conversión a y desde a listas de parejas clave-valor

Demo

ETS

Tablas ETS = Erlang Term Storage colección de BIFs agrupadas en el módulo **`ets`** que sirven para almacenar en memoria grandes colecciones de datos de forma eficiente

- Almacen tuplas en la cuales uno de sus componentes sirve de clave de la tupla
- Se implementan utilizando tablas hash y árboles binarios, distintas representaciones para cada tipo de colección

Cuatro clases de ETSs

Set no pueden aparecer dos tuplas con la misma clave al insertar una con la misma clave que otra anterior entonces se elimina la antigua

Ordered Set igual que los set pero optimizadas para ser recorridas en orden

Bag pueden aparecer varias tuplas con la misma clave pero no dos tuplas iguales

Duplicate bag pueden aparecer varias tuplas idénticas

ETS

Creación `ets:new(Nombre, Opciones)` devuelve un identificador de tabla que podemos utilizar para acceder a la tabla más adelante posibles opciones

`set, ordered_set, bag, duplicate_bag` clase de ETS que se creará
`{keypos, Pos}` indica cuál de las componentes de las tuplas se usará de clave
`named_table` permite usar el nombre de la tabla para acceder a ella si no se especifica entonces `Nombre` no vale para nada
`public, protected, private` donde `public` = la tabla se puede leer y escribir por cualquier proceso; `protected` = la tabla se puede leer por cualquier proceso pero sólo escribir por el proceso dueño; `private` = sólo el proceso dueño puede acceder a la tabla

opciones por defecto (pasando la lista vacía como opciones) entre otras `set, {keypos, 1}, protected`

No hay recogida de basura para las tablas ETS: se debe usar `ets:delete(Tabla)`, también se destruyen cuando se termina el proceso dueño

El comando `tv:start()` inicia la herramienta gráfica de visualización de tablas

ETS

Operaciones

`insert(Tab, ObjectOrObjects)` inserta la tupla o lista de tuplas especificada en la tabla de acuerdo a la clase de tabla que sea

`lookup(Tab, Key)` devuelve las entradas asociada a la clave correspondiente

`delete(Tab, Key)` borra todas las entradas asociadas a la clave correspondiente

`tab2list(Tab)` devuelve la lista de elementos en el tabla como una lista

`first(Tab)` devuelve la primera clave en la tabla más sentido para ordered sets

`next(Tab, Key)` devuelve la siguiente clave en la tabla más sentido para ordered sets

`last(Tab)` devuelve la última clave en la tabla más sentido para ordered sets

```
TSet = ets:new(tset, [named_table, set]) .
ets:insert(tset, [{pepe, {nombre, pepe}}, {pepe, {edad, 30}}]) .
ets:lookup(tset, pepe) .
ets:insert(tset, [{lola, cuenca}, {maria, vigo}]) .
ets:tab2list(tset) .
Primero = ets:first(tset) .
ets:next(tset, Primero) .
ets:last(tset) .
```

Accesos concurrentes problemas cuando un proceso escribe en una ETS mientras otro la esta recorriendo `safe_fixtable(Tab, true)` durante un recorrido con `first/1` y `next/2` todos los elementos presentes en la tabla se devolverán una vez, aunque otro proceso los borre durante el recorrido. Si se añaden nuevos elementos es posible que se devuelvan con `next/2`, dependiendo del orden de la tabla. Al terminar se debe llamar a `safe_fixtable(Tab, false)` para liberar los elementos borrados durante el recorrido

ETS

Operaciones: match

`match(Tab, Pattern)` busca en la tabla con encaje de patrones. Los patrones pueden contener

- valores Erlang usados en patrones normales
- `'$0'`, `'$1'`, ... variables que se ligarán durante el matching
- `'_'` variables mudas

como resultado se devuelve una lista con un elemento para cada encaje del patrón con algún elemento de la tuplas. Estos elementos serán listas que contienen la ligadura de cada variable `'$i'`, en orden según las `i`

`match_object(Tab, Pattern)` como `match/2` pero devolviendo la tupla completa

`select(Tab, MatchingSpec)` forma más general de `match/2` las *matching specifications* tienen su propio formato, pero puede utilizarse `ets:fun2ms` para convertir un formato limitado de funciones a matching specifications

```
TBag = ets:new(tbag, [named_table, bag]) .
ets:insert(tbag, [{pepe, {nombre, pepe}}, {pepe, {edad, 30}}]) .
ets:tab2list(tbag) .
ets:lookup(tbag, pepe) .
ets:match(tbag, {pepe, '$1'}) .
ets:match(tbag, {pepe, {nombre, '$1'}}) .
ets:match(tbag, {'$1', {'$2', '$3'}}) .
ets:match(tbag, {'$1', {'$2', '$1'}}) .
ets:match(tbag, {'$2', {'$1', '$2'}}) .
ets:match_object(tbag, {'$1', {'_', '$1'}}) .
ets:select(tbag, ets:fun2ms(fun({Nombre, {Campo, Val}}) when Nombre == pepe -> Campo end)) .
```

ETS

Operaciones: escritura en disco

`tab2file(Tab, FileName)` guarda la tabla en disco

`file2tab(Tab, FileName)` lee la tabla del disco se respeta el nombre de la tabla

```
ets:new(countries, [bag,named_table]).
ets:insert(countries,{yves,france,cook}).
ets:insert(countries,{sean,ireland,bartender}).
ets:insert(countries,{marco,italy,cook}).
ets:insert(countries,{chris,ireland,tester}).
ets:tab2list(countries) .
ets:tab2file(countries, "tab.bin") .
q() .
$ cat tab.bin
ets:file2tab("tab.bin") .
ets:tab2list(countries) .
```

ETS

Records en ETS los records se traducen a tuplas en tiempo de compilación, por tanto pueden insertarse en una ETS.

. Problema la posición de la clave por defecto en las ETS es 1, que corresponde al tipo del record en la traducción de records a tuplas

. Solución mediante `#RecordType.Field` obtenemos la posición del campo Field en la traducción de `RecordType` a tupla, que podemos utilizar luego como opción `{keypos, #RecordType.Field}` de la llamada a `ets:new/2`

```
rd(capital, {name, country, pop}).
ets:new(countries, [named_table, {keypos, #capital.name}]).
ets:insert(countries, #capital{name="Budapest", country="Hungary", pop=2400000}).
ets:insert(countries, #capital{name="Pretoria", country="South Africa", pop=2400000}).
ets:insert(countries, #capital{name="Rome", country="Italy", pop=5500000}).
ets:lookup(countries, "Pretoria").
ets:match(countries, #capital{name='$1',country='$2', _='_'}).
ets:match_object(countries, #capital{country="Italy", _='_'}).
MS = ets:fun2ms(fun(#capital{pop=P, name=N}) when P < 5000000 -> N end).
ets:select(countries, MS) .
```

- insertamos los records como si fueran tuplas
- podemos usar la notación de record en `ets:match/2`, nótese el uso de `_='_'` para ignorar el valor de los demás campos del record.

DETS

Tablas Dets = Disk Erlang Term Storage similares a las ETS pero asociadas a un archivo en disco

- Más lentas que las ETS, que funcionan en memoria, pero con persistencia ya que se auto-salvan en disco
- El tamaño del archivo asociado está limitado a 2GB para tamaños mayores se puede usar Mnesia, la base de datos nativa de Erlang, u otra base de datos más estándar
- Tres tipos de Dets: set, bag y duplicate bag

Creación `dets:open_file(Nombre, Opciones)` abre una tabla o la crea si no existía el archivo asociado. En caso de éxito devuelve `{ok, Nombre}` posibles opciones

`{type, bag | duplicate_bag | set}` indica el tipo de la tabla

`{auto_save, Intevalo}` indica cada cuanto tiempo se salvará en disco la tabla si no se accede en el intervalo especificado en milisegundos

`{file, FileName}` indica el nombre del archivo asociado a la tabla (por defecto el nombre de la tabla)

Las tablas Dets se cierran cuando todos los procesos que hayan abierto la tabla hayan terminado o llamado a `dets:close(Name)` si la tabla no se cierra antes de que se cierre la máquina virtual de Erlang entonces se deberá reparar la próxima vez que se abra

`dets:open_file(FileName)` abre una tabla existente. En caso de éxito devuelve `{ok, Ref}` donde `Ref` deberá utilizarse para futuros accesos a la tabla

DETS

Ejemplo

```
dets:open_file(food, [{type, bag}, {file, "./comida"}]).
dets:insert(food, {italy, spaghetti}).
dets:insert(food, {sweden, meatballs}).
dets:lookup(food, china).
dets:lookup(food, sweden).
dets:insert(food, {italy, pizza}).
NotItalian = ets:fun2ms(fun({Loc, Food}) when Loc /= italy -> Food end).
dets:select(food, NotItalian).
dets:close(food).
q() .
$ cat comida

{ok, Ref} = dets:open_file("./comida").
dets:lookup(Ref, italy).
dets:info(Ref).
dets:lookup(food, italy).
```

OTP Behaviors

OTP = Open Telecom Platform framework para el desarrollo de aplicaciones de telecomunicaciones en Erlang.

. Conjunto de módulos Erlang que lo convierten en un lenguaje de uso industrial, al proporcionar implementaciones fiables y escalables de la mayoría de las funcionalidades típicas requeridas en aplicaciones de ese tipo

. Parte de la distribución estándar de Erlang

OTP Behaviours corresponden a los interfaces de Java o las clases de tipos de Haskell

. Disponibles como módulos de OTP que se encargan de hacer el trabajo genérico de creación de procesos y manejo de errores

. El código específico de la aplicación se debe proporcionar en un módulo separado llamado callback module, que debe implementar unas funciones callback predefinidas igual que con los interfaces de Java o las clases de tipos de Haskell

OTP Behaviors - generic server

Generic server este behaviour proporciona funcionalidad genérica para implementar procesos servidores, por tanto permite

. iniciar el nuevo proceso servidor, posiblemente enlazándolo y/o registrándolo

. comunicarse con los clientes con llamadas síncronas y asíncronas, y definir un formato de mensajes para esa comunicación

. manejar el estado del servidor que se pasa como argumento de la función recursiva de cola que implementa el loop de receive del servidor

. detener el proceso servidor

Y además permite reutilizar una estructura de supervisión y código de gran calidad.
Se implementa en el módulo `gen_server`

`gen_server:start_link(ServerName, Module, Args, Options)` inicia el servidor genérico enlazándolo con el proceso invocador.

- `ServerName` nombre que se utiliza para registrar el proceso, y puede ser de la forma `{local, Name}` (para registros locales al nodo, como hasta ahora) o `{global, Name}` (para registros en el global_name_server del módulo `global`
- `Module` nombre del callback module
- `Args` argumento con el que se llamará a `Module:init/1` (donde `Module` es el callback module)
- `Options` opciones especiales de configuración

`gen_server:call(ServerRef, Request)` realiza una llamada síncrona al servidor
`gen_server:cast(ServerRef, Request)` realiza una llamada asíncrona al servidor

también disponible `gen_server:start/4` para iniciar el servidor sin enlazar los procesos

OTP Behaviors - generic server

Funciones a implementar en el **callback module** la directiva -behavior(gen_server) en el callback module indica que estamos implementando dicho behavior

`Module:init(Args)` llamada por gen_server:start_link/4, en caso de inicialización exitosa debe devolver `{ok, InitialState}`, o bien `{stop, Reason}` si algo falló en la inicialización. `InitialState` se utilizará como el valor inicial del estado en el bucle de receive del servidor

`Module:terminate(Reason, State)` realiza las acciones de limpieza (cerrar archivos o Dets, ...) necesarias antes de la terminación del servidor

`Module:handle_call(Request, From, State)` responde a una petición correspondiente a una llamada síncrona desde `From`, que toma la forma `{PidDelCliente, EtiquetaUnica}`

. En caso de éxito debe devolver `{reply, Reply, NewState}` de manera que el servidor continúa su ejecución actualizando su estado, y `Reply` se devuelve al cliente como resultado de la llamada a `gen_server:call/2` que inició la petición

. Si se devuelve `{stop, Reason, NewState}` se llama a `Module:terminate(Reason, State)` y se deteniene el servidor

`Module:handle_cast(Request, State)` similar a `Module:handle_call/3` pero para peticiones asíncronas correspondientes a llamadas a `gen_server:cast/2`, por lo que en caso de éxito se devuelve `{noreply, NewState}` y al proceso que llamó a `cast/2` se le devuelve `ok`

`Module:handle_info(Info, State)` similar a `Module:handle_cast/3` también atiende llamadas asíncronas pero generadas por envíos crudos de mensajes al servidor a través del operador `!`, en vez de mediante `gen_server:cast/2`

y un par más menos comunes

Demo `echo_gen.erl`

OTP Behaviors

Otros behaviors

Supervisor (módulo `supervisor`) facilita la implementación sistemas robustos con varias capas de supervisión, jerarquizando los procesos como trabajadores o supervisores. Ofrece políticas predefinidas para gestionar las caídas de los procesos hijos y terminar y/o reiniciar los procesos relacionados

Application (módulo `application`) en una aplicación Erlang el módulo principal debe implementar este behaviour, que se encarga de poner en marcha el sistema y cargar las otras aplicaciones de las que dependa. A menudo el módulo principal implementa tanto `application` como `supervisor`

Finite State Machines (módulo `gen_fsm`) para la implementación de máquinas de estados finitas, muy comunes en aplicaciones de telefonía

Event Handling (módulo `gen_event`) para implementar gestores de eventos que permitan registrar y desregistrar manejadores de eventos

También podemos definir nuestro propios behaviors aunque es bastante poco común, y la documentación está bastante escondida

http://www.erlang.org/doc/design_principles/des_princ.html

Success types

Como ya vimos en temas anteriores Erlang es un lenguaje con **tipado dinámico** en particular los tipos nunca pueden hacer que el compilador rechace un programa que funciona correctamente

Sin embargo con el tiempo se desarrollo un sistema de tipos para Erlang que sigue respetando el principio de ausencia de falsos positivos: los **success types** de Erlang

```
Type :: any() | none() | pid()  | port() | reference() | [] | Atom | Binary | float() | Fun |
Integer | List | Tuple | Union | UserDefined
Union :: Type1 | Type2
Atom :: atom() | Erlang_Atom
Fun :: fun() | fun((...) -> Type) | fun(() -> Type) | fun((Type, ..., Type) -> Type)
Integer :: integer() | Erlang_Integer | Erlang_Integer..Erlang_Integer
List :: list(Type) | improper_list(Type1, Type2)  | maybe_improper_list(Type1, Type2)
Tuple :: tuple()  | {} | {Type, ..., Type}
```

. Cualquier valor es un tipo, se usan tipos unión

. Tipos declarador por el usuario

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

. Declaraciones `spec` que especifican los tipos de las funciones se entienden como un **contrato** que el usuario se compromete a cumplir `-spec reverse([A]) -> [A] when A :: any().`

. Declaraciones de tipo en los `records`

```
-record(rec, {f1 = 42 :: integer(), f2 :: float(), f3 :: 'a' | 'b'}).
```

http://www.erlang.org/doc/reference_manual/typespec.html

Success types

Significado de los success types **sobreaproximaciones** de las evaluaciones exitosas de las expresiones Erlang

. `e :: T` implica que si `e` se evalúa a un valor `v` entonces `v` pertenece al conjunto representado por `T`

. `f :: fun((TIn) -> TOut)` implica que si `f(e)` se evalúa a un valor `v` entonces `e :: TIn` y `v :: TOut`

- La evaluación en Erlang es indeterminista se puede obtener más de un resultado debido a la ejecución concurrente o simplemente por la interacción con el usuario

- Todas las expresiones y funciones tienen varios success types posibles que son diversas aproximaciones más o menos finas a su valor. En particular `any()` es siempre válido

Dialyzer success types aplicados a la detección automática de errores definitivos (sin falsos positivos) en tiempo de compilación

. http://www.erlang.org/doc/apps/dialyzer/dialyzer_chapter.html

. el comando `typer` llama a Dialyzer para extraer la información de tipos implícita en el programa y devolverla en forma de declaraciones `spec`

Success types

Usando los success types para detectar errores definitivos dado el programa `errores.erl`

```
f(0) -> 1 .
g() -> f(1) .
```

```
$ dialyzer errores.erl
...
errores.erl:7: Function g/0 has no local return
errores.erl:7: The call errores:f(1) will never return since it differs in the 1st argument
from the success typing arguments: (0)
$ typer errores.erl
...
-spec f(0) -> 1.
-spec g() -> none().
```

En general, si `f :: fun((TIn) -> TOut)` y `e :: T` entonces

- `T ⊆ TIn` no implica una garantía de que el matching tendrá éxito

  si hemos sido demasiado optimistas con `TIn` sobreaproximando a un tipo demasiado grande: por ejemplo `f :: fun((any()) -> 1)` es un success type válido para `f/1` pero `f(1)` falla
- `T ⊃ TIn` no implica que el matching vaya a fallar necesariamente

  si hemos sido demasiado pesimistas con `T` sobreaproximando a un tipo demasiado grande: por ejemplo `0 :: any()` es un success type válido para `0` pero `f(0)` se evalúa exitosamente a `1`
- `T ∩ TIn = ∅` sí que implica una garantía de que el matching fallará siempre

  dados `v` un valor para `e` y `D` el dominio de un valor para `f/1` entonces los success types de `e` y `f/1` implican `v ∈ T` y `D ⊆ TIn`. Como `T ∩ TIn = ∅` entonces `v ∉ TIn`, lo que combinado con `D ⊆ TIn` implica `v ∉ D`, es decir, un fallo de matching. En el ejemplo Dialyzer calcula `f :: fun((0) -> 1)` y `1 :: 1`, y como `0 ∩ 1 = ∅` entonces se expresa el fallo como `f(1) :: none()`

Success types

Usando los success types para detectar errores definitivos

```
-module(errores) .
-export([f/1, g/0, h0/1, h1/1, dead/1]).

% fallo de matching
f(0) -> 1 .
g() -> f(1) .

% refinamiento de dominio de funciones locales
h0(0) -> id(0) .
h1(1) -> id(1) .
id(X) -> X .
% codigo muerto
dead(X) -> case h0(X) of
                  0 -> ok;
                  2 -> error
              end .
$ dialyzer errores.erl
...
errores.erl:7: Function g/0 has no local return
errores.erl:7: The call errores:f(1) will never return since it differs in the 1st argument
from the success typing arguments: (0)
errores.erl:18: The pattern 2 can never match the type 1
$ typer errores.erl
...
-spec f(0) -> 1.
-spec g() -> none().
-spec h0(0) -> 0 | 1.
-spec h1(1) -> 0 | 1.
-spec id(0 | 1) -> 0 | 1.   % dominio de funcion local refinado
-spec dead(0) -> 'ok'.
```

Success types

Usando Dialyzer http://www.erlang.org/doc/man/dialyzer.html

. Dialyzer guarda los análisis de los módulos estándar en una PLT (Persistent Lookup Table) para no tener que volver a reanalizarlos en cada análisis

. Para usar Dialyzer primero tenemos que crear una PLT

`dialyzer --build_plt --apps erts kernel stdlib` crea una PLT en `$HOME/.dialyzer_plt` almacenando los análisis de las librerías estándar

`dialyzer --build_plt --apps erts kernel stdlib --output_plt file` como el anterior pero guardando la PLT en el archivo especificado

. Una vez creada la PLT Dialyzer utiliza la de `$HOME/.dialyzer_plt` por defecto, o si no la especificada

`$ dialyzer module.erl` analiza el archivo especificado

`$ dialyzer --plt <ruta a PLT> module.erl` analiza el archivo especificado utilizando la PLT especificada

Usando Typer `$ typer --help`

. Typer se limita a llamar a Dialyzer para pedirle la información de tipos que este infiere para los análisis, por lo que también necesita una PLT

`$ typer module.erl` infiere los success types del archivo especificado

`$ typer --plt <ruta a PLT> module.erl` infiere los success types del archivo especificado utilizando la PLT especificada

Success types

Usando los success types para detectar documentar los programas la ayuda de Erlang en http://www.erlang.org/erldoc ya está plagada de success types en las descripciones de las funciones

La herramienta EDoc http://www.erlang.org/doc/man/edoc.html de documentación de programas es similar a Javadoc pero adaptado a Erlang, y utiliza los success types para la documentación

```
%% @doc Este modulo sirve para programar el factorial
%% @copyright 2012 El inventor del factorial
%% @author Juan Rodriguez

-module(fact) .
-export([fact/1]) .

%% @doc Esta funcion calcula el factorial de un numero
-spec fact(non_neg_integer()) -> non_neg_integer() .
fact(0) -> 1 ;
fact(N) when N >0 -> N * fact(N-1) .
```

La documentación `html` correspondiente se genera llamando a `edoc:files(["fact.erl"])` sólo se genera documentación para las funciones en un `-export`, la directiva `-compile(export_all)` sólo es útil durante el desarrollo y la depuración