Concurrent Programming

CS511

(Lack of) Types

Documenting Types using `spec`

Tail Recursion

Exceptions

Control Structures

Erlang is Strongly Typed

```
1> 6+"1".
** exception error: an error occurred when evaluating an
arithmetic expression
     in operator  +/2
        called as 6 + "1"
```

Good, but there is no static type-checking...

Recall from Previous Class

```
drivers_license(Age) when Age < 16 ->
    forbidden ;
drivers_license(Age) when Age == 16 ->
    'learners permit' ;
drivers_license(Age) when Age == 17 ->
    'probationary license' ;
drivers_license(Age) when Age >= 65 ->
    'vision test recommended but not required' ;
drivers_license(_) ->
    'full license'.
```

Types

```
2> c1:drivers_license(45).
'full license'
3> c1:drivers_license("hi").
'vision test recommended but not required'
```

- What is going on?
- Recall the comparison order

$$\textit{number} < \textit{atom} < \textit{reference} < \textit{fun} < \textit{port} < \textit{pid} < \textit{tuple} < \textit{map} < \textit{nil} < \textit{list} < \textit{bitstring}$$

```
...
drivers_license(Age) when Age >= 65 ->
    'vision test recommended but not required' ;
...
```

Types

```
drivers_license(Age) when not(is_number(Age)) ->
    throw(wrong_argument_type);
drivers_license(Age) when Age < 16 ->
    forbidden ;
% the rest follows without change
```

- Other type-checking predicates:

  `is_atom/1, is_function/1, is_boolean/1, is_record/1,...`

- More on exceptions later

```
9> c1:drivers_license("hi").
** exception throw: wrong_argument_type
     in function  c1:drivers_license/1 (c1.erl, line 6)
```

Documenting Types

- Type specifications:

  ```
  -spec Function(ArgType1, ..., ArgTypeN)->ReturnType.
  ```

  or

  ```
  -spec Function(ArgName1 :: Type1, ..., ArgNameN :: TypeN)->RT
  ```

- Type specifiers are used for:
  - Documentation of intended usage
  - Automatic detection of type errors
- The compiler does not check type but there are tools for doing this

Type Declarations – Examples

```
-spec drivers_license(integer()) -> atom().

drivers_license(Age) when Age < 16 ->
    forbidden ;
drivers_license(Age) when Age == 16 ->
    'learners permit' ;
drivers_license(Age) when Age == 17 ->
    'probationary license' ;
drivers_license(Age) when Age >= 65 ->
    'vision test recommended but not required' ;
drivers_license(_) ->
    'full license'.
```

Dialyzer

- Checks that given specifications agree with call patterns
  - Also detects exceptions and dead code
- It does so loosely using so called “Success Typings” http://www.it.uu.se/research/group/hipe/papers/succ_types.pdf
  - Assume that all is good in terms of typing (start from most general possible type) and then refining this view as the code analysis progresses

Dialyzer

- Before using this tool you must initialize its internal tables (Persistent Lookup Tables)
- This process can take 5 minutes or more

```
$ dialyzer --build_plt --apps erts kernel stdlib crypto
    mnesia sasl common_test eunit
  Creating PLT /Users/ebonelli/.dialyzer_plt ...
Unknown functions:
  compile:file/2
  compile:forms/2
  compile:noenv_forms/2
  compile:output_generated/1
  cover:analyse/2
  cover:analyse_to_file/2
  cover:analyse_to_file/3
  cover:compile_beam/1
  cover:export/1
  cover:get_main_node/0
  cover:import/1
  cover:imported_modules/0
  cover:start/0
  cover:start/1
  cover:stop/0
  cover:stop/1
  cover:which_nodes/0
```

Checking Type Declarations

```
-spec drivers_license(integer()) -> atom().

drivers_license(Age) when Age < 16 ->
    forbidden ;
drivers_license(Age) when Age == 16 ->
    'learners permit' ;
drivers_license(Age) when Age == 17 ->
    'probationary license' ;
drivers_license(Age) when Age >= 65 ->
    'vision test recommended but not required' ;
drivers_license(_) ->
    'full license'.
```

We check our code with dialyzer

```
$ dialyzer c1.erl
  Checking whether the PLT /Users/ebonelli/.dialyzer_plt is
      up-to-date... yes
  Proceeding with analysis... done in 0m1.03s
done (passed successfully)
```

Checking Type Declarations

```erlang
-spec drivers_license(integer()) -> string().

drivers_license(Age) when Age < 16 ->
    forbidden ;
%...other clauses here...
```

We check our code with dialyzer

```
$ dialyzer c1.erl
  Checking whether the PLT /Users/ebonelli/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis...
c1.erl:5: Invalid type specification for function c1:drivers_license/1. The success typing is (_) -> 'forbidden' | 'full license' | 'learners permit' | 'probationary license' | 'vision test recommended but not required'
 done in 0m1.09s
done (warnings were emitted)
```

Checking Type Declarations

```
-spec drivers_license(integer()) -> string().

drivers_license(Age) when Age < 16 ->
    forbidden ;
drivers_license(Age) when Age == 16 ->
    'learners permit' ;
drivers_license(Age) when Age == 17 ->
    "probationary license" ;
drivers_license(Age) when Age >= 65 ->
    'vision test recommended but not required' ;
drivers_license(_) ->
    'full license'.
```

We check our code with dialyzer

```
$ dialyzer c1.erl
  Checking whether the PLT /Users/ebonelli/.dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.99s
done (passed successfully)
```

Type Declarations – More Examples

- Type variables can be used in specifications to specify relations for the input and output arguments of a function
- For example, the following specification defines the type of a polymorphic identity function:

```
-spec id(X) -> X.
```

- Notice that the above specification does not restrict the input and output type in any way

Type Declarations – More Examples

- Type variables can be constrained using a when clause
- The :: constraint should be read as "is a subtype of"

```
%% sum(L) returns the sum of the elements in L
-spec sum(List) -> number() when
      List :: [number()].

%% min(L) -> returns the minimum element of the list L
-spec min(List) -> Min when
      List :: [T,...],
      Min :: T,
      T :: term().

%% append(X, Y) appends lists X and Y
-spec append(List1, List2) -> List3 when
      List1 :: [T],
      List2 :: [T],
      List3 :: [T],
      T :: term().
```

Type Expressions 1/3

- Singletons can be either integers or atoms:
  - 1, 2 or 42
  - 'foo', 'bar' or 'atom'
  - foo, 42
- Unions of singletons, what we normally refer to as "types":
  - integer(): any integer value
  - float(): any floating point value
  - atom(): any atom
  - pid(): a process identifier
  - ref(): a reference
  - fun(): a function
  - ... and many more

Type Expressions 2/3

- Types for compound data structures:
  - tuple(): a tuple of any form
  - list(): a proper list of any length
- Union type constructor
  - type | type

```
-spec f('a' | 1) -> 'b' | 1.
f(1) ->
     1;
f(a) ->
     b.
```

Type Expressions 3/3

Some built-in types and how they are defined[1]:

term()	any()
boolean()	'false' — 'true'
byte()	0..255
char()	0..16#10ffff
nil()	[]
number()	integer() — float()
list()	[any()]
nonempty_list()	nonempty_list(any())
string()	[char()]
nonempty_string()	[char(),...]
function()	fun()
module()	atom()
no_return()	none()

[1] http://erlang.org/doc/reference_manual/typespec.html

Defining Types – An Example

- Use of `type` directive

```
%%% {empty}   -- Empty tree
%%% {node,Data, LeftTree, RightTree}  -- Non empty tree

-type btree() :: {empty} | {node,term(),btree(),btree()}.

-spec sizeT(btree()) -> number().

sizeT({empty}) ->
    0;
sizeT({node,_D,LT,RT}) ->
    1 + sizeT(LT) + sizeT(RT).
```

Defining Types – Another Example

We would like to define our own type that specifies what a card looks like.

```
-type value() :: 1..13.
-type suit() :: spade | heart | diamond | clubs.
-type card() :: {card, suit(), value()}.
-spec suit(card()) -> suit().
```

Define the type of a deck of cards.

```
-type deck() :: list(card())
```

An Example

```erlang
-module(cards).
-export([kind/1, main/0]).

-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.

kind({_, A}) when A >= 1, A =< 10 -> number;
kind(_) -> face.

main() ->
number = kind({spades, 7}),
face   = kind({hearts, k}),
number = kind({rubies, 4}),
face   = kind({clubs, q}).
```

```
1> c1:main().
face
```

Somewhat unexpected...

An Example

```
$ dialyzer c1.erl
  Checking whether the PLT /Users/ebonelli/.dialyzer_plt is
      up-to-date... yes
  Proceeding with analysis...
 done in 0m1.33s
done (warnings were emitted)
```

According to Dialyzer, everything is ok.

An Example

```
-module(cards).
-export([kind/1, main/0]).

-type suit() :: spades | clubs | hearts | diamonds.
-type value() :: 1..10 | j | q | k.
-type card() :: {suit(), value()}.

-spec kind(card()) -> face | number.
kind({_, A}) when A >= 1, A =< 10 -> number;
kind(_) -> face.

main() ->
number = kind({spades, 7}),
face   = kind({hearts, k}),
number = kind({rubies, 4}),
face   = kind({clubs, q}).
```

An Example

```
$ dialyzer c1.erl
  Checking whether the PLT /Users/ebonelli/.dialyzer_plt is
      up-to-date... yes
  Proceeding with analysis...
c1.erl:34: Function main/0 has no local return
c1.erl:37: The call c1:kind({'rubies',4}) breaks the
    contract (card()) -> 'face' | 'number'
 done in 0m1.02s
done (warnings were emitted)
```

(Lack of) Types

Documenting Types using `spec`

Tail Recursion

Exceptions

Control Structures

List Examples

```
> c(list_examples).
{ok,list_examples}
> list_examples:sum([1,2,3,4]).
10
> list_examples:len([0,1,0,1]).
4
> list_examples:append([5,4],[1,2,3]).
[5,4,1,2,3]
```

- We will define them recursively (inductively)
  - Base case: empty list (`[]`)
  - Recursive case: a list with at least one element (`[X | XS]`)

Tail Recursion

- Programming pattern to increase performance
- It helps compilers when optimizing code
- Inefficient recursive definition

```
len([_|XS]) -> 1 + len(XS) ;
len([])     -> 0.
```

Observe the evaluation of `len([1,2,3])`

```
len([1,2,3]) == 1 + len([2,3])
len([1,2,3]) == 1 + (1 + len([3]))
len([1,2,3]) == 1 + (1 + (1 + len([]))) %%
len([1,2,3]) == 1 + (1 + (1 + 0))
len([1,2,3]) == 1 + (1 + 1)
len([1,2,3]) == 1 + 2
len([1,2,3]) == 3
```

- At the time of reaching the marked line, Erlang needs to keep in memory a long expression
- After that line, it starts shrinking the expression
- Imaging how it will work for a very big list!

Tail Recursion

- More efficiency by tail recursion
- Space (constant if we assume elements of the list have the same size)
- Efficiency (No returns from recursive calls)
- What is the trick?
  - Use of accumulators (partial results)
  - There are no more computations after the recursive call

Tail Recursion

- We define `len_a`, the tail recursive version of len
- Function `len_a` has an extra parameters capturing the partial result of the function, i.e., how many elements `len_a` has seen so far

```
len_a([_|XS], Acc) -> len_a(XS, Acc+1);
len_a([], Acc) -> Acc.
```

We define len based on `len_a` as follows

```
len(XS) -> len_a(XS, 0).
```

What about the tail recursive version of sum and append?

Exceptions

Three kinds:

- errors: run-time errors such as `1+a`; can be emulated with `error(Reason)`
- exits: generated error; generated by a process using `exit/1`
  - Studied next class
- throws: generated error; generated by a process using `throw/1`
  - Brief overview next

Throw Exceptions

- Used for cases that the programmer can be expected to handle
- In comparison with exits and errors, they don't really carry any 'crash that process!' intent behind them, but rather control flow
- Good idea to document their use within a module using them

```
1> throw(permission_denied).
** exception throw: permission_denied
```

Try...Catch

```
-module(exceptions).
-compile(export_all).

throws(F) ->
    try F() of
      _ -> ok
    catch
      Throw -> {throw, caught, Throw}
    end.
```

```
1> c(exceptions).
{ok,exceptions}
2> exceptions:throws(fun() -> throw(thrown) end).
{throw,caught,thrown}
3> exceptions:throws(fun() -> erlang:error(pang) end).
** exception error: pang
```

Try..Catch

```
talk() -> "blah blah".

sword(1) -> throw(slice);
sword(2) -> erlang:error(cut_arm);
sword(3) -> exit(cut_leg);
sword(4) -> throw(punch);
sword(5) -> exit(cross_bridge).

black_knight(Attack) when is_function(Attack, 0) ->
   try Attack() of
     _ -> "None shall pass."
   catch
     throw:slice -> "It is but a scratch.";
     error:cut_arm -> "I've had worse.";
     exit:cut_leg -> "Come on you pansy!";
     _:_ -> "Just a flesh wound."
end.
```

Try–Catch

```
7> c(exceptions).
{ok,exceptions}
8> exceptions:talk().
"blah blah"
9> exceptions:black_knight(fun exceptions:talk/0).
"None shall pass."
10> exceptions:black_knight(fun() -> exceptions:sword(1) end).
"It is but a scratch."
11> exceptions:black_knight(fun() -> exceptions:sword(2) end).
"I've had worse."
12> exceptions:black_knight(fun() -> exceptions:sword(3) end).
"Come on you pansy!"
13> exceptions:black_knight(fun() -> exceptions:sword(4) end).
"Just a flesh wound."
14> exceptions:black_knight(fun() -> exceptions:sword(5) end).
"Just a flesh wound."
```

Additional Constructs

```
try Expr of
    Pattern -> Expr1
catch
    Type:Exception -> Expr2
after % this always gets executed
    Expr3
end
```

- `Expr3` is always run, be there an exception or not

Additional Constructs

```
1> catch throw(whoa).
whoa
2> catch exit(die).
{'EXIT',die}
3> catch 1/0.
{'EXIT',{badarith,[{erlang,'/',[1,0]},
{erl_eval,do_apply,5},
{erl_eval,expr,5},
{shell,exprs,6},
{shell,eval_exprs,6},
{shell,eval_loop,3}]}}
4> catch 2+2.
4
```

(Lack of) Types

Documenting Types using `spec`

Tail Recursion

Exceptions

Control Structures

Control Structures

```
is_greater_than(X, Y) ->
    if
        X>Y ->
            true;
        true -> % works as an 'else' branch
            false
    end
```

Control Structures

```
is_valid_signal(Signal) ->
    case Signal of
        {signal, _What, _From, _To} ->
            true;
        {signal, _What, _To} ->
            true;
        _Else ->
            false
    end.
```