Erlang lab

0. Warm-up

In this warm-up section, we introduce some basic tools that facilitate programming in Erlang. After reading this section, you will know how to compile Erlang modules and invoke functions in the Erlang shell, use a `Makefile` to do repetitive tasks conveniently, and use `EUnit` and `PropEr` for testing.

Create a file named `hello_world.erl` with the following content:

```
-module(hello_world).

-export([run/0]).

run() ->
  io:format("~s~n", ["Hello World!"]).
```

The Erlang Shell

If you would like to play with the program interactively, a shell is provided after launching `erl`, like this:

```
$ erl
Erlang/OTP 18 [erts-7.1] [source] [64-bit] [smp:4:4] [async-threads:10] [hipe] [kernel-poll:false]

Eshell V7.1  (abort with ^G)
1>
```

Let's compile our `hello_world` module inside the Erlang shell using the `c/1` function. More shell commands could be found in its manual.

```
1> c(hello_world).
{ok,hello_world}
```

Now we can call the exported function:

```
2> hello_world:run().
Hello World!
ok
```

If we wanted to compile the module to native code, we could have used the `c/2` function that takes an option list in its second argument:

```
3> c(hello_world, [native]).
{ok,hello_world}
```

Because only exported functions can be called from other modules or in the shell, sometimes during debugging it is convenient to be able to (temporarily) export all functions. For this, we can use the `export_all` compiler option:

```
3> c(hello_world, [export_all]).
{ok,hello_world}
```

though using this option in such a simple a module (whose single function is exported anyway) has no visible effect.

Working using a Makefile

Create a file named `Makefile` with the following content:

```
default: run

MODULE = hello_world

.PHONY: run
run: compile
	erl -noshell -eval "$(MODULE):run()" -s init stop
```

```
.PHONY: compile
compile:
	erlc -Werror +debug_info $(MODULE).erl

.PHONY: clean
clean:
	$(RM) *.beam erl_crash.dump
```

Note that the whitespace before `erlc`, `erl` and `$(RM)` commands should not be spaces, but a tab character. There are various places on the internet to find out more about Makefiles. One of them is here.

Given this `Makefile` and our `hello_world` module, we could run our program using:

```
$ make run
```

Dialyzer

Let's add another function (`fact/1` which is supposed to implement the factorial function) to our `hello_world` module and use it in `run/0`. It is not necessary to export it.

```
-module(hello_world).

-export([run/0]).

run() ->
  io:format("~s~n", ["Hello World!"]),
  io:format("Factorial of 42 is ~p~n", [fact(42)]).

fact(0) -> 1;
fact(X) when X > 0 ->
  X * fact(X-1).
```

The factorial function (and `fact/1`) is not defined for negative numbers. Let's change `42` to `-17`, recompile the module, and see what happens.

```
$ make compile
```

The Erlang compiler does not complain about this at all. (Nor would compilers of most other languages!) If we wanted to catch this kind of error before running the code, we could use the Dialyzer tool.

In order to use Dialyzer, we first need to build Persistent Lookup Table (PLT). We can build a basic PLT with the following Unix command. (While this command runs, you can read more about Dialyzer here.)

```
$ dialyzer --build_plt --apps erts kernel stdlib
```

Since we would like to run Dialyzer frequently, it is convenient to create a command for it in the `Makefile`:

```
.PHONY: dialyzer
dialyzer: compile
	dialyzer -Wunmatched_returns $(MODULE).beam
```

Now we could be informed about this error like this:

```
$ make dialyzer
  ...
hello_world.erl:5: Function run/0 has no local return
hello_world.erl:7: The call hello_world:fac(-17) will never return since it
differs in the 1st argument from the success typing arguments: (non_neg_integer())
 done in 0m0.35s
done (warnings were emitted)
make: *** [dialyzer] Error 2
```

Dialyzer is smart enough to infer that `fact/1` will only return a value when called with a non-negative integer as its argument, which helps us catch and correct these kinds of errors early in the development process.

EUnit

For pedagogical reasons, let us assume we have made a mistake and used `+` instead of `*` in our factorial function. This is not a type-related error, but a semantics one, and Dialyzer will not give us any warnings in this case, because the program is consistent as far as success typings are concerned. This is the point where unit testing can help us catch such errors.

Let's add some EUnit tests to our `hello_world` module. (Note that it now contains an incorrect implementation of `fact/1`.)

```
-module(hello_world).

-export([run/0]).

-include_lib("eunit/include/eunit.hrl").

run() ->
  io:format("~s~n", ["Hello World!"]),
  io:format("Factorial of 42 is ~p~n", [fact(42)]).

fact(0) -> 1;
fact(X) when X > 0 ->
  X + fact(X-1).

fact_test_() ->
  [?_assertEqual(R, fact(N)) || {N,R} <- [{0,1}, {1,1}, {4,24}]].
```

Let's add the command for EUnit in our `Makefile` as well:

```
default : run

MODULE = hello_world

.PHONY: run
run: compile
	erl -noshell -eval "$(MODULE):run()" -s init stop

.PHONY: compile
compile:
	erlc -Werror +debug_info $(MODULE).erl

.PHONY: dialyzer
dialyzer: compile
	dialyzer -Wunmatched_returns $(MODULE).beam

.PHONY: eunit
eunit: compile
	erl -noshell -eval "eunit:test($(MODULE))" -s init stop

.PHONY: clean
clean:
	$(RM) *.beam erl_crash.dump
```

Running `make dialyzer` doesn't reveal any inconsistencies, but `make eunit` tells us that `fact/1` is behaving weirdly. An excerpt of the terminal output appears below:

```
hello_world:18: fact_test_...*failed*
in function hello_world:'-fact_test_/0-fun-2-'/1 (hello_world.erl, line 18)
**error:{assertEqual,[{module,hello_world},
              {line,18},
              {expression,"fact ( 1 )"},
              {expected,1},
              {value,2}]}
  output:<<"">>
```

```
hello_world:19: fact_test_...*failed*
in function hello_world:'-fact_test_/0-fun-4-'/1 (hello_world.erl, line 19)
**error:{assertEqual,[{module,hello_world},
              {line,19},
              {expression,"fact ( 4 )"},
              {expected,24},
              {value,11}]}
  output:<<"">>

=======================================================
  Failed: 2.  Skipped: 0.  Passed: 1.
```

PropEr

As mentioned in the lectures, an alternative to writing tests manually is to employ property-based testing and generate tests automatically. Let us consider the factorial function; a natural property to ensure is that the result of calling `fact(N)` for all valid inputs `N` should be divisible by all integers from 1 up to `N`. (If `N` is zero, there are no such numbers, so the property trivially holds in this case also.)

Let's see how to express that property using PropEr:

```
-module(hello_world).

-export([run/0]).

-include_lib("proper/include/proper.hrl").  % should be above EUnit
-include_lib("eunit/include/eunit.hrl").

...

prop_divisible_by_all_up_to() ->
  ?FORALL(N, non_neg_integer(), divisible_by_all_up_to(N, fact(N))).

divisible_by_all_up_to(N, F) ->
  lists:all(fun(X) -> F rem X =:= 0 end, lists:seq(1, N)).
```

Unfortunately this is the point when we need to take a small break. In contrast to Dialyzer and EUnit which are part of Erlang/OTP, PropEr is not a tool which is part of the Erlang distribution. To use it, you first need to install it in your file system somewhere. Pick a suitable place (e.g., your home directory) and follow the instructions on PropEr's quickstart guide to install PropEr there.

After this is done, let us also edit the `Makefile` and modify the `PROPER_EBIN` variable appropriately. (If you chose your home directory to install PropEr, this should point to `$(HOME)/proper/ebin`.) You should also remember to add a `-pa $(PROPER_EBIN)` option to the `erlc` command of the `compile` target. Subsequently, we can add one more target to our Makefile, called `proper`:

```
.PHONY: proper
proper: compile
	erl -noshell -pa $(PROPER_EBIN) -eval "proper:module($(MODULE))" -s init stop
```

Now, let's invoke it:

```
$ make proper
Testing hello_world:prop_divisible_by_all_up_to/0
.....................................................................
OK: Passed 100 test(s).
```

1. Power Set

Time for the first task of this lab and time for you to exercise your knowledge by writing some Erlang code yourselves. The task is simple: write a program that computes the power set of any given set (of some Erlang terms). The definition of *power set* can be found in Wikipedia.

Start by creating new file named with `powerset.erl`, and populate it with basic structures (module declaration, exported functions, etc.) like we did for the `hello_world` example.

You could name the main exported function `powerset/1`, or something you think is more appropriate. According to the definition of power set, the function should accept a set and return a set. Now, we need to look at how `set`, as a data type, is defined/used in Erlang. The documentation for a set library in Erlang could be found by googling this way.

Some hints for this task:

- Since Erlang does not have native syntax for sets, you could use `sets:from_list/1` and `sets:to_list/1` to convert between lists and sets (if you need something like that).
- To test your code, start with a few simple EUnit test cases, such as the empty set, or sets with one or two elements.
- Erlang does not provide any functions for set equality checking. What should you write in `?_assertEqual`?
- After your code passes some simple EUnit teat cases, continue with writing property-based tests.
- What property should power set satisfy? Is there any relation between the size of the power set and the size of the input set? What is the relation between the elements of the power set and the input set?

Self-assessment

In the end of this task, you should be able to call `make eunit` to run all the `EUnit` tests and `make proper` to run the property-based tests, and see no failures.

2. Depth-First Search

The second task to do is to implement Depth-First Search.

In order to keep the problem simple and the solution easy to check, we can confine ourselves to complete Binary Trees with integer labels.

Call your module `dfs` and write and export one function to construct a complete binary tree with Breadth-First Traversal visiting nodes in the order corresponding to their labels. For example, the constructed 3-levels complete binary tree would look like this. The root node is always labeled with `1`.

The `search/2` function would accept two arguments, a complete binary tree constructed using the above function, and an integer (the goal) that's guaranteed to exist within the tree, and returns a list contains the nodes we could encounter starting from the root to the goal. For example, if we use the 3-levels complete binary tree above, `search(Tree, 7)` would return `[1,3,7]`.

Since we construct our trees in such a special way (complete binary tree, root node is 1, Breadth-First Traversal returns natural number sequence [1,2,3,4...]), we could deduce the solution without traversing the tree at all. You would implement this short-cut approach in another function (say `solution/1`), and use it to check your implementation of Depth-First Search using property-based tests.

Some hints for this task

- Since we explicitly stated that the goal we are passing to the `search/2` function exits in the tree, what is the valid range for its value, given a `D`-level(s) complete binary tree? (There is a module called `math` in Erlang, which might be useful.)
- Arithmetic operators could be found here, because some operators are different from other languages.

Self-assessment

In the end, you should be able to call `make proper` to run the property-based tests and see no failures.