Sequential Erlang – Workshop

Hans Svensson <hanssv@ituniv.se>

Example: Geometry

- We will look at some code handling geometrical shapes
  - **`{square, X}`**
  - **`{circle, R}`**
  - **`{rectangle, W, H}`**

- We want to compute area, circumference, total area, etc.

area/1

- First we implement the function area/1
  - Circle: Radius * Radius * π
  - Square: Side * Side
  - Rectangle: Width * Height

area/1

- First we implement the function area/1

```
%% Area of a geometrical figure
area({square, X}) ->
    X * X;
area({circle, R}) ->
    R * R * ?PI;
area({rectangle, H, W}) ->
    H * W.
```

Function clauses

Clause separator

Function separator

area/1

- Remember how pattern matching works

```
%% Area of a geometrical figure
area({square, X}) ->
    X * X;
area({circle, R}) ->
    R * R * ?PI;
area({rectangle, H, W}) ->
    H * W.
```

1st Pattern

2nd Pattern

3rd Pattern

area/1

- Let’s try it

```
…
11> c(geometry).
{ok,geometry}
12> geometry:area({circle,5}).
78.53975
13> geometry:area({square,5}).
25
14>
```

In the right directory!!

circumference/1

- circumference/1 is very similar to area/1
  - Circle: 2 * Radius * π
  - Square: 4 * Side
  - Rectangle: 2* Width + 2*Height

circumference/1

- circumference/1 is very similar to area/1

```
%% Circumference of geometrical figures
circumference({square, X}) ->
    4 * X;
circumference({circle, R}) ->
    2 * R * ?PI;
circumference({rectangle, H, W}) ->
    2 * H + 2 * W.
```

circumference/1

- Let’s try it

```
…
17> c(geometry).
{ok,geometry}
18> geometry:circumference({rectangle,3,7}).
20
19> geometry:circumference({circle,3.5}).
21.99113
20>
```

larger/2

- Comparing the area of two shapes

- Which shape has the bigger area, return the larger shape.

larger/2

- Comparing the area of two shapes

```
%% Comparison function, which shape
%% is larger
larger(Shape1, Shape2) ->
    case area(Shape1) > area(Shape2) of
     true  -> Shape1;
     false -> Shape2
    end.
```

Case statement

larger/2

- Comparing the area of two shapes

```
%% Comparison function, which shape
%% is larger
larger(Shape1, Shape2) ->
    case area(Shape1) > area(Shape2) of
     true  -> Shape1;
     false -> Shape2
    end.
```

Case expression

larger/2

- Comparing the area of two shapes

```erlang
%% Comparison function, which shape
%% is larger
larger(Shape1, Shape2) ->
    case area(Shape1) > area(Shape2) of
     true  -> Shape1;
     false -> Shape2
    end.
```

Case clauses

Pattern matching, again!

larger/2

- Let’s try it

```
…
20> c(geometry).
{ok,geometry}
21> geometry:larger({circle,4},{square,3}).
{circle,4}
22>
geometry:larger({rectangle,2,8},{square,4}).
{square,4}
23>
```

Is that really right!?

Recursion

- The basic pattern

```
double([]) -> [];
double([H | T]) -> [2*H | double(T)].
```

- Recursive case: Do something with the **head** of the list, apply recursive function to **tail**

sum_areas/1

- Computing the sum of areas of a list of shapes
- Let’s use basic recursion

sum_areas/1

- Computing the sum of areas of a list of shapes
- Let’s use basic recursion

```
%% Total area for a list of shapes
sum_areas([]) ->
    0;
sum_areas([Shape | Shapes]) ->
    area(Shape) + sum_areas(Shapes).
```

sum_areas/1

- Let’s try it

```
…
26> c(geometry).
{ok,geometry}
27> geometry:sum_areas([{square,4},
        {circle,5},{rectangle,3,5}]).
109.53975
28>
```

Recursion

- With an accumulator

```
average([]) -> 0;
average(X) -> avg(X, 0, 0).

avg([H | T], Len, Sum) ->
     avg(T, Len + 1, Sum + H);
avg([], Len, Sum) ->
     Sum / Len.
```

- **Accumulate result, when the whole list is traversed return (or compute) final result.**

largest/1

- Finding the largest shape in a list of shapes
  - Crash for the empty list (none is largest!)
  - Use recursion, with an accumulator argument

largest/1

- Finding the largest shape in a list of shapes
- Let's use recursion, with an accumulator

```
largest([Shape | Shapes]) ->
    largest(Shape, Shapes).

largest(Largest, []) ->
    Largest;
largest(Largest, [Shape | Shapes]) ->
    largest(larger(Shape, Largest), Shapes).
```

Crashes for largest([]).

- The accumulator is the largest shape seen so far… When the list is traversed it is the result.

largest/1

- Let’s try it

```
…
23> c(geometry).
{ok,geometry}
24> geometry:largest([{rectangle,2,5},
       {square,5},{circle,7}]).
{circle,7}
25> geometry:largest([]).
** exception error: no function clause
matching geometry:largest([])
26>
```

Dialyzer

- So why are there –spec annotations in my code?
- The reasons are twofold
  - **Documentation** - the expected type(s) says a lot about a function in few keystrokes
  - **Confidence** - using the dialyzer tool, we can check whether the type annotations are respected by the code

```
dialyzer --src –Wunderspecs geometry.erl
```

Edoc

- Writing comments in edoc style has its benefits, one can generate nice API documentation automatically (with the not very user friendly):

```
erl –noshell –run edoc_run files \
     [geometry.erl] [{dir,’./doc’}] \
     -s init stop
```

More copy friendly:

```
erl -noshell -run edoc_run files [geometry.erl] [{dir,'./doc'}] -s init stop
```

Example: Sorting

- As a second example we will look at a couple of different implementations of Quicksort
- Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.
    1. Pick an element, called a *pivot* from the list.
    2. Reorder the list so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position.
    3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

Standard Quicksort

```
%% Normal verbose quicksort
qsort([]) -> [];
qsort([Elem]) -> [Elem];
qsort([Pivot | List]) ->
  Larger  = larger_than(Pivot, List),
  Smaller = less_than_or_equal(Pivot, List),
  qsort(Smaller) ++ [Pivot] ++ qsort(Larger).
```

Standard Quicksort

```
larger_than(_Pivot,[]) -> [];
larger_than(Pivot,[H | T])
  when H > Pivot ->
     [H | larger_than(Pivot, T)];
larger_than(Pivot,[_ | T]) ->
    larger_than(Pivot, T).

less_than_or_equal(_Pivot, []) -> [];
less_than_or_equal(Pivot, [H | T])
  when H =< Pivot ->
     [H | less_than_or_equal(Pivot, T)];
less_than_or_equal(Pivot,[_ | T]) ->
    less_than_or_equal(Pivot, T).
```

qsort/1

- Let’s try it

```
…
37> c(sorting).
{ok,sorting}
38> sorting:qsort([1,2,45,623,3,1,65,
                   3,5,87,21,3]).
[1,1,2,3,3,3,5,21,45,65,87,623]
39>
```

More compact Quicksort

```
%% A more compact version
qsort1([])      -> [];
qsort1([Elem]) -> [Elem];
qsort1([Pivot | List]) ->
    qsort1([E || E <- List, E =< Pivot]) ++
     [Pivot] ++
     qsort1([E || E <- List, E > Pivot]).
```

qsort1/1

- Let’s try it

```
…
40> c(sorting).
{ok,sorting}
41> sorting:qsort1([1,2,45,623,3,1,65,
                    3,5,87,21,3]).
[1,1,2,3,3,3,5,21,45,65,87,623]
42>
```

Generic Quicksort

```
%% A generic quicksort, that accepts the
%% comparison function as an argument
gen_qsort(_CmpFun,[])      -> [];
gen_qsort(_CmpFun,[Elem]) -> [Elem];
gen_qsort(CmpFun,[Pivot | List]) ->
  gen_qsort(CmpFun,[E || E <- List,
                         CmpFun(Pivot,E)]) ++
  [Pivot] ++
  gen_qsort(CmpFun,[E || E <- List,
                         not CmpFun(Pivot,E)]).

cmp_fun(A,B) -> A > B.
```

gen_qsort/1

```
…
43> c(sorting).
{ok,sorting}
44> sorting:gen_qsort(
      fun(A,B) -> A > B end,
      [1,32,6,13,6,51,4,67,3]).
[1,3,4,6,6,13,32,51,67]
45> sorting:gen_qsort(
      fun sorting:cmp_fun/2,
      [1,32,6,13,6,51,4,67,3]).
[1,3,4,6,6,13,32,51,67]
46>
```

Assignment – Seq. Erlang

- The Erlang shell will automatically convert [8] into “\b”, that is not an error! (But “8” is not equivalent to [8] or “\b”!!)
- You are **expected** to change **`sum(_List)`** into for example **`sum([X | Xs])`**.
- **TEST** before you submit, so far I have seen 4 submissions, all of them fail on trivial input.
- The principle of *least surprise* should hold.

Sample exam

- A sample exam with solutions is published in GUL (under Documents)
- 5-6 questions, 100 points in total
- 50 points for G, 75 for VG.

Assignment – Seq. Erlang

QUESTIONS?