Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Didactics

Publications

Funded Projects

Research Projects

Related Events

Exam of Programming Languages

26 February 2015

Disclaimer. Note that to have a running solution for an exercise is not enough: you need a well-cooked solution that proves your ability to use what explained during the classes. All the exercises have the same value: 11; Exercise are valued separately and it is necessary to get at least 6 points each exercise to pass the exams, i.e., 18 achieved with only 2 exercises means to fail the exam.

Exercise OCaML/ML: Playing with Intervals.

An interval represents a sequence of computable and subsequent values delimited by two of them: the minimum and maximum of the sequence. Such a minimum and maximum are enough to characterize the whole interval.

The interval data type that we are going to consider should be compliant to the following signature

```
module type IntervalI =
  sig
    type interval
    type endpoint
    val create : endpoint -> endpoint -> interval
    val is_empty : interval -> bool
    val contains : interval -> endpoint -> bool
    val intersect : interval -> interval -> interval
    val tostring : interval -> string
    exception WrongInterval
  end;;
```

where `interval` is the abstract type for the interval and the `endpoint` is the abstract type for the set used to define the interval. All the operations have the obvious meaning.

Note that an interval can be defined on any ordered set not only on numeric sets, e.g., you can have an interval of words lexicographically ordered. To be compliant with this constraint any data sort used in combination with the described data type must be compliant to the `Comparable` signature:

```
module type Comparable = sig
    type t
    val compare : t -> t -> int
    val tostring : t -> string
  end ;;
```

where `t` is the abstract type for the comparable elements—that is, those elements that will be part of the interval—and the operations have the obvious meaning.

The exercise consists in providing a **polymorphic** implementation for the `Interval` abstract data type based on the `Comparable` and the `IntervalI` signatures and an instantiation for the integer (`IntInterval`) and string (`StringInterval`) intervals that permits to run this example:

```
open Interval ;;

let main () =
  let i1 = IntInterval.create 3 8  and
      i2 = IntInterval.create 4 10 and
      s1 = StringInterval.create "abacus" "zyxt" and
      s2 = StringInterval.create "dog" "wax"
  in
    Printf.printf "%s\n" (IntInterval.tostring (IntInterval.intersect i1 i2));
    try
      Printf.printf "%s\n"
        (StringInterval.tostring (StringInterval.create "wax" "fog"))
    with StringInterval.WrongInterval->Printf.printf "Exception: WrongInterval\n";
    Printf.printf "%s\n"
        (StringInterval.tostring (StringInterval.intersect s1 s2)) ;
    Printf.printf
        "Does \"%s\" belong to %s? %B\nDoes it belong to %s? %B\n" "asylum"
        (StringInterval.tostring s2) (StringInterval.contains s2 "asylum")
        (StringInterval.tostring s1) (StringInterval.contains s1 "asylum") ;;
let() = main() ;;
```

The following is the expected behavior:

```
[13:33]cazzola@surtur:~/lp/ocaml>main
```

```
[4, 8]
Exception: WrongInterval
[dog, wax]
Does "asylum" belong to [dog, wax]? false
Does it belong to [abacus, zyxt]? true
```

If you have any doubt, yes, what you have to implement is a **functor** and it must be compilable as follows (the given code **CAN'T BE CHANGED**):

```
[10:37]cazzola@surtur:~/lp/ocaml>ocamlc -c *.mli
[10:37]cazzola@surtur:~/lp/ocaml>ocamlc -c «file».ml (* This is your file *)
[10:37]cazzola@surtur:~/lp/ocaml>ocamlc -o main «file».cmo main.ml
```

Exercise Erlang: Distributed Tasks.

One of the basics of distributed computation consists of splitting the task to do in several subtasks that can be distributed among several computational units and then the partial results will be composed together to form the expected result. This has several advantages; the most evident are:

1. The whole process can be sped up and
2. The subtasks are easier than the original task

Neglecting the first point, mathematics can help a lot on the second point. If the original task can be expressed as a function ψ this can decomposed in several functions φ^1, ..., φ^n such that holds

$$\psi \equiv \varphi^1 \circ ... \circ \varphi^n$$

This means that each subtask is implemented by an easier function and the final result depends on the mathematical composition of such functions in the right order.

Given this, the whole approach is realized by an actors ring where the number of actors in the ring depends on the number of functions the original function is decomposed into . Each actor runs the function corresponding to its position in the sequence—i.e.,the actor position in the ring corresponds to the function position in the original function decomposition. The first actor applies its function to an external input; all the other actors will apply their function to the result of the computation of the actor that precedes them and the last actor will print out the final result.

Basically, all the actors have the same behavior but the last one. All data are exchanged through message passing. You interact with the first actor in the ring through three functions: `send_message/1`, `send_message/2` and `stop/0`. The `stop` function has to nicely shut down the actors ring; the `send_message/1` function passes the initial value to the first actor and initiates the whole computations that ends when the message reaches the last actor in the ring; whereas the `send_message/2` function does the same as the `send_message/1` but the second argument represents the number of times the message should run through the whole ring—i.e., the number of times the whole algorithm should be applied. The actors ring is started by the `start/2` functions whose arguments are the number of actors in the ring and a list of functions (the φ in the above formula).

This exercise consists in implementing the module `ring` that implements the described situation, To simplify the exercise the only admissible functions are unary on integers—i.e., from int to int.

The following is an example of the expected behavior:

```
1> L1 = [fun(X)-> X*N end||N<-lists:seq(1,7)].
[#Fun<erl_eval.6.80484245>,#Fun<erl_eval.6.80484245>,
 #Fun<erl_eval.6.80484245>,#Fun<erl_eval.6.80484245>,
 #Fun<erl_eval.6.80484245>,#Fun<erl_eval.6.80484245>,
 #Fun<erl_eval.6.80484245>]
2> ring:start(7,L1).
3> ring:send_message(1). % this is 7!
5040
4> ring:send_message(2). % this is 2×7!
10080
5> ring:send_message(1,10). % this is 7!^10
1057560848118006498591768576000000000
6> ring:stop().
7> L2 = [fun(X)-> X+1 end||N<-lists:seq(1,1000)].
[#Fun<erl_eval.6.80484245>,#Fun<erl_eval.6.80484245>,
 #Fun<erl_eval.6.80484245>,#Fun<erl_eval.6.80484245>,
 #Fun<erl_eval.6.80484245>,#Fun<erl_eval.6.80484245>,
 #Fun<erl_eval.6.80484245>,#Fun<erl_eval.6.80484245>|...]
8> ring:start(1000,L2).
9> ring:send_message(969). % this is 1000+969
1969
10> ring:send_message(0,1000). % this is 1000^2
1000000
10> ring:send_message(1000,1000). % this is 1000+1000^2
1001000
```

All the solutions not conform to the given specification will be considered wrong.

On Your Desk.

Desk is a pretty easy language thought to stress some parsers capabilities sine it relies on absolutely non circular attribute grammars. But doesn't matter if you don't know what this implies since the language is really simple. It is an evaluator of expressions with literals and variables whose only available operation is the sum on integers.

Any Desk program has the form:

> **PRINT** «expression» **WHERE** «variable initialization»

Please note that the variable initialization is done through the = symbol and can be related to several variables, each assigment is comma-separated.

The exercise consists of implementing a DSL by using parser combinator techniques that can parse and execute Desk programs.

The following is an example of expected behavior

```
print x+y+z+1+x+-3 where x = 25, y = 1, z=-7
```

```
[10:29]cazzola@surtur:~/lp/scala>scala DeskEvaluator test.desk
42
Map(z -> -7, y -> 1, x -> 25)
```