Walter Cazzola

Home Page
ADAPT Lab.
Curriculum Vitae
Research Topic

Didactics

Publications

Funded Projects

Research Projects

Related Events

Exam of Programming Languages

15 June 2016

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 ML/OCaML: A «Continued» QuickSort.

As you probably remember from the lectures, functional programming permits to write a really elegant implementation for the quicksort algorithm:

```
let qsort (>:) l =
  let rec qsort = function
    [] -> []
  | h::tl -> (qsort (List.filter (fun x -> (x >: h)) tl) )
             @ [h] @
             (qsort (List.filter (fun x -> (h >: x)) tl) )
  in qsort l
```

Unfortunately, this implementation is not efficient as it could be and the call stack could explode with very large lists. Mainly this is due to the fact that it is not tail recursive. The quicksort algorithm can't be easily transformed into a recursive version. The only way is to use the *continuation passing style*.

For those not remembering the lectures (look at the one on functors), a continuation reifies the program control state, i.e., the continuation is a data structure that represents the computational process at a given point in the process's execution; the created data structure can be accessed by the programming language, instead of being hidden in the runtime environment. The term continuation is also be used to refer to first-class continuations, which are constructs that give a programming language the ability to save the execution state at any point and return to that point at a later point in the program, possibly multiple times. This is illustrated by the *continuation sandwich* description:

> «Say you're in the kitchen in front of the refrigerator, thinking about a sandwich. You take a continuation right there and stick it in your pocket. Then you get some turkey and bread out of the refrigerator and make yourself a sandwich, which is now sitting on the counter. You invoke the continuation in your pocket, and you find yourself standing in front of the refrigerator again, thinking about a sandwich. But fortunately, there's a sandwich on the counter, and all the materials used to make it are gone. So you eat it. :-)» [Palmer'04]

In this description, the sandwich is part of the program data (e.g., an object on the heap), and rather than calling a *make sandwich* routine and then returning, the person called a *make sandwich with current continuation* routine, which creates the sandwich and then continues where execution left off.

This exercise consists in using continuations in order to write a tail recursive implementation of the quicksort algorithm, named `cqsort`. Please note that there will be only one possible solution that can be considered corrected, that is, the one that use continuation-passing style.

This is a possible main, that will be used to test your assignments. Please note that `Qsort` is the module with the old `qsort` implementation, `Cqsort` is the module with your assignment and the reported figures depends on the machine you are using.

```
open Qsort ;;
open Cqsort ;;

let make_list x =
   let rec make_list x acc =
      if x = 0 then acc
      else make_list (x-1) (x::acc)
   in make_list x [];;

let profile f =
   let s = Sys.time() in
      let _ = f() in (Sys.time()) -. s;;

let main () =
  let l = make_list 10000 in
    Printf.printf
       "Sorting 10000 elements with the functional qsort takes %3.2f µs\n"
       (profile (fun () -> qsort (>) l));
    Printf.printf
```

```
        "Sorting 10000 elements with the continuation qsort takes %3.2f µs\n"
        (profile (fun () -> cqsort (>) l));;

let() = main() ;;
```

```
[16:16]cazzola@hymir:~/lp/ocaml> ./main
Sorting 10000 elements with the functional qsort takes 7.34 µs
Sorting 10000 elements with the continuation qsort takes 6.36 µs
```

Exercise Erlang: Joseph's Problem.

Flavius Josephus was a roman historian of Jewish origin. During the Jewish-Roman wars of the first century AD, he was in a cave with fellow soldiers, 40 men in all, surrounded by enemy Roman troops. They decided to commit suicide by standing in a ring and counting off each third man. Each man so designated was to commit suicide... Josephus, not wanting to die, managed to place himself in the position of the last survivor.

In the general version of the problem, there are n Hebrews numbered from 1 to n and each k-th Hebrew will be eliminated. The count starts from the first Hebrew. What is the number of the last survivor?

The exercise consists of implementing a solution for the general version of the «Joseph's Problem». The Hebrews are represented by actors connected in a ring and every time one Hebrew is eliminated the corresponding actor is removed from the ring. The count is done by letting a message circulate on the ring and after k hops the actor which receives it is the one appointed to commit suicide. After that the count restarts from the actor next to the one eliminated. When only one actor remains alive this communicate master process that he is the survivor.

To recap, two modules are needed, `hebrew` for the actors describing the Hebrews and `joseph` implementing the homonymous function `joseph` that creates the ring according to the n and k passed as arguments and coordinates the initial and final messages. The following is a possible test case to verify the correct behavior of your assignment (run it with `erl -noshell -eval 'test:test()' -s init stop`.)

```
-module(test).
-export([test/0]).

test() ->
  joseph:joseph(30,3),
  joseph:joseph(300,1001),
  joseph:joseph(3000,37),
  joseph:joseph(26212,2025),
  joseph:joseph(1000,1000),
  joseph:joseph(2345,26212),
  joseph:joseph(100000,7).
```

```
[0:37]cazzola@hymir:~/lp/erlang>erl -noshell -eval 'test:test()' -s init stop
In a circle of 30 people, killing number 3
Joseph is the Hebrew in position 29
In a circle of 300 people, killing number 1001
Joseph is the Hebrew in position 226
In a circle of 3000 people, killing number 37
Joseph is the Hebrew in position 1182
In a circle of 26212 people, killing number 2025
Joseph is the Hebrew in position 20593
In a circle of 1000 people, killing number 1000
Joseph is the Hebrew in position 609
In a circle of 2345 people, killing number 26212
Joseph is the Hebrew in position 1896
In a circle of 100000 people, killing number 7
Joseph is the Hebrew in position 27152
```

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

Exercise Scala: ArnoldC.

ArnoldC is an imperative programming language where the basic keywords are replaced with quotes from different Schwarzenegger movies. We are now going to implement **ArnoldC** through Scala parser combinators. Here follows **ArnoldC** syntax and semantics description plus a few examples.

Every **ArnoldC** program must always have a main method enclosed between `IT'S SHOW TIME` and `YOU HAVE BEEN TERMINATED`. A program with only these two keywords is also the minimal **ArnoldC** admissible program.

Printing

The statement `TALK TO THE HAND` is used to print strings or variables. E.g., `TALK TO THE HAND "Hello World"` prints the string "Hello World". All print commands go to a new line.

Declaring variables

The only variable type in ArnoldC is integer. A value must be given to the variable when it is declared.

```
HEY CHRISTMAS TREE variable
YOU SET US UP initial_value
```

Assigning variables

Variable assignment is done using the pattern:

```
GET TO THE CHOPPER myvar
HERE IS MY INVITATION first_operand
«operations»
ENOUGH TALK
```

The `HERE IS MY INVITATION` sets a value on the top of the stack. The rest of the operations always apply to the current value of the stack which is finally assigned to the `myvar` variable.

Arithmetic operations

Only the four basic arithmetic operations are available in ArnoldC: + is `GET UP operand`, - is `GET DOWN operand`, * is `YOU'RE FIRED operand` and / is `HE HAD TO SPLIT operand`. All operations works on the stack and have the same precedence.

Example a = (4 + b) * 2 is

```
GET TO THE CHOPPER a
HERE IS MY INVITATION 4
GET UP b
YOU'RE FIRED 2
ENOUGH TALK
```

Logical operations

True statements result the value of 1 and false statements the value of 0. Four logic operators are available: == is `YOU ARE NOT YOU YOU ARE ME operand`, > is `LET OFF SOME STEAM BENNET operand`, ∨ is `CONSIDER THAT A DIVORCE operand` and ∧ is `KNOCK KNOCK operand`. All operators work on the stack and have the same precedence.

Note that «operations» in both logical and arithmetic operations represent other logical or arithmetic operations. These kind of operations can be used only in conjunction with an assignment statement.

Example a = (b ∨ c) ∧ d

```
GET TO THE CHOPPER a
HERE IS MY INVITATION b
CONSIDER THAT A DIVORCE c
KNOCK KNOCK d
ENOUGH TALK
```

Conditional statements

The conditional statement (if-the-else) follows the pattern:

```
BECAUSE I'M GOING TO SAY PLEASE value [
«statements»
] BULLSHIT [
«statements»
] YOU HAVE NO RESPECT FOR LOGIC
```

The first branch is executed when `value` is anything other than 0.

Example if(a) print "a is true" else print "a is not true"

```
BECAUSE I'M GOING TO SAY PLEASE a
TALK TO THE HAND "a is true"
BULLSHIT
TALK TO THE HAND "a is not true"
YOU HAVE NO RESPECT FOR LOGIC
```

Note that the `value` can only be an identifier.

Loop Statement

There is just one kind of loop statement in ArnoldC that works as a while statement it follows the following pattern:

```
STICK AROUND value [
«statements»
] CHILL
```

note `value` compare against zero and it can be only an identifier.

The following are some **ArnoldC** programs with the expected execution.

Printing the first ten integers:

```
IT'S SHOWTIME
HEY CHRISTMAS TREE isLessThan10
YOU SET US UP 1
HEY CHRISTMAS TREE n
YOU SET US UP 0
STICK AROUND isLessThan10 [
GET TO THE CHOPPER n
HERE IS MY INVITATION n
GET UP 1
ENOUGH TALK
TALK TO THE HAND n
GET TO THE CHOPPER isLessThan10
HERE IS MY INVITATION 10
LET OFF SOME STEAM BENNET n
ENOUGH TALK
] CHILL
YOU HAVE BEEN TERMINATED
```

```
[18:50]cazzola@hymir:~/lp/scala>scala ArnoldCEvaluator print-10.arnoldc
1
2
3
4
5
6
7
8
9
10
Stack()
Symbol Table :-
        (n,10)
        (isLessThan10,0)
```

Printing the square of the first ten numbers:

```
IT'S SHOWTIME
HEY CHRISTMAS TREE limit
YOU SET US UP 10
HEY CHRISTMAS TREE index
YOU SET US UP 1
HEY CHRISTMAS TREE squared
YOU SET US UP 1
HEY CHRISTMAS TREE loop
YOU SET US UP 1
STICK AROUND loop [
GET TO THE CHOPPER squared
HERE IS MY INVITATION index
YOU'RE FIRED index
ENOUGH TALK
TALK TO THE HAND squared
GET TO THE CHOPPER loop
HERE IS MY INVITATION limit
LET OFF SOME STEAM BENNET index
ENOUGH TALK
GET TO THE CHOPPER index
HERE IS MY INVITATION index
GET UP 1
ENOUGH TALK
] CHILL
YOU HAVE BEEN TERMINATED
```

```
[18:53]cazzola@hymir:~/lp/scala>scala ArnoldCEvaluator square.arnoldc
1
4
9
16
25
36
49
64
81
100
Stack()
Symbol Table :-
        (limit,10)
        (loop,0)
        (index,11)
        (squared,100)
```