Lecture13-30: Exercises

prof. Marco Bellia, Dip. Informatica, Università di Pisa

May 27, 2014

Excercise1

Exercise1.
We decide to add an iterator *for*, to the language Ocaml. Ocaml already has an iterator *for* but we want to add an iterator having the same structure and behavior of *for* of Ansi-C :

(a) Explain:
   1. What is the difference of the two *for* and
   2. How the structure and the behaviour of the new one should be;

(b) Give an abstract syntax and a denotational semantics of the new construct;

(c) Show the implementation, in Ocaml, of the new construct ;

(d) Discuss the mechanisms that have been used to do previous point;

(e) Apply the new construct in rephrasing the code below and comment about its running:

```
int x=0;
int y=0;
for(x=y=1; x+y<100;x++){x=x-1; y=y+2}
```

Excercise1 - sol/1

Exercise1.
We decide to add an iterator *for*, to the language Ocaml. Ocaml already has an iterator *for* but we want to add an iterator having the same structure and behavior of *for* of Ansi-C:

(a) Explain:
   1. What is the difference of the two *for* and
   2. How the structure and the behaviour of the new one should be;

Example (Solution)

a1. The for of Ocaml is a determined iterator whilst the C's one is a non-determined iterator;

a2. **About Syntax.** We use the following syntactic structure:
$E ::= \text{For } (E_1,E_2,E_3,E_C);$
where, E_1,E_2,E_3 are expressions for initialization, limit, increment, and E_C is the command-like expression to be iterated.
About Semantic. We assume the following semantic constraints:
(i) Expressions E_1,E_2,E_3,E_C are delayed expressions, i.e. they are functions of type unit−>'a for 'a which is bool for E_2 and unit for E_C ;
(ii) all the mutable values that should be shared from the expressions E_1,E_2,E_3,E_C, have been introduced in an environment having the for-expression in its scope.

Excercise1 - sol/2

Exercise1.
We decide to add an iterator *for*, to the language Ocaml. Ocaml already has an iterator *for* but we want to add an iterator having the same structure and behavior of *for* of Ansi-C :

(b) Give an abstract syntax and a denotational semantics of the new construct;

Example (Solution in a ordinary Imperative context (first table) and then, in an Applicative one (2nd table))

Syntactic Domains

$\mathtt{C} ::= \ldots \mid \mathtt{For}\ \mathtt{C}_1\ \mathtt{E}_2\ \mathtt{C}_3\ \mathtt{C} \mid \ldots$

Semantic Functions

$$\mathcal{M}[\![\mathtt{C}]\!]_\rho : \mathtt{Store} \longrightarrow \mathtt{Store}_\perp$$

$$\begin{array}{l} \mathcal{M}[\![\mathtt{For}\ \mathtt{C}_1\ \mathtt{E}_2\ \mathtt{C}_3\ \mathtt{C}]\!]_\rho(\mathtt{s}) = \\ \quad \mathtt{Let}\{\mathtt{s}_1 = \mathcal{M}[\![\mathtt{C}_1]\!]_\rho(\mathtt{s})\} \\ \quad\quad \{\mathrm{Y}\,\mathtt{g}.\lambda\,\mathtt{g}.\lambda\,\mathtt{s_w}.\mathtt{Let}\{(\mathtt{v},\mathtt{s}_2) = \mathcal{E}[\![\mathtt{E}_2]\!]_\rho(\mathtt{s_w})\} \\ \quad\quad\quad\quad \mathtt{if}(\mathtt{false}(\mathtt{v}),\mathtt{s}_2,(\mathcal{M}[\![\mathtt{C}]\!]_\rho \circ \mathcal{M}[\![\mathtt{C}_3]\!]_\rho \circ \mathtt{g})(\mathtt{s}_2))\} \\ \quad\quad \mathtt{g}(\mathtt{s}_1) \end{array}$$

Syntactic Domains

$\mathtt{E} ::= \ldots \mid \mathtt{For}\ \mathtt{E}_1\ \mathtt{E}_2\ \mathtt{E}_3\ \mathtt{E}_\mathtt{C} \mid \ldots$

Semantic Functions

$$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \longrightarrow \mathtt{Store}_\perp$$

$$\begin{array}{l} \mathcal{E}[\![\mathtt{For}\ \mathtt{E}_1\ \mathtt{E}_2\ \mathtt{E}_3\ \mathtt{E}_\mathtt{C}]\!]_\rho(\mathtt{s}) = \\ \quad \mathtt{Let}\{(\mathtt{unit},\mathtt{s}_1) = \mathcal{E}[\![\mathtt{E}_1]\!]_\rho(\mathtt{s})\} \\ \quad\quad \{\mathrm{Y}\,\mathtt{g}.\lambda\,\mathtt{g}.\lambda\,\mathtt{s_w}.\mathtt{Let}\{(\mathtt{v},\mathtt{s}_2) = \mathcal{E}[\![\mathtt{E}_2]\!]_\rho(\mathtt{s_w})\} \\ \quad\quad\quad\quad \mathtt{if}(\mathtt{false}(\mathtt{v}),\mathtt{s}_2,(\mathcal{E}[\![\mathtt{E_C}]\!]_\rho \circ_\mathtt{u} \mathcal{E}[\![\mathtt{E}_3]\!]_\rho \circ_\mathtt{u} \mathtt{g})(\mathtt{s}_2))\} \\ \quad\quad (\mathtt{unit},\mathtt{g}(\mathtt{s}_1)) \end{array}$$

where: $\mathtt{g1} \circ_\mathtt{u} \mathtt{g2(s)}$ = g2(s2) where (unit,s2)=g(s1) for all states s and functions g1,g2 of type $\mathtt{Store} \longrightarrow (\mathtt{unit} \times \mathtt{Store})$

Excercise1 - sol/3

Exercise1.
We decide to add an iterator *for*, to the language Ocaml. Ocaml already has an iterator *for* but we want to add an iterator having the same structure and behavior of *for* of Ansi-C :

(c) Show the implementation, in Ocaml, of the new construct;

Example (Solution the semantics in an Applicative context (1st table), its impementation in Ocaml (2nd table))

Syntactic Domains

$\mathtt{E} ::= \ldots \mid \mathtt{For}\ \mathtt{E_1}\ \mathtt{E_2}\ \mathtt{E_3}\ \mathtt{E_C} \mid \ldots$

Semantic Functions

$$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \longrightarrow \mathtt{Store}_\perp$$

$$\begin{aligned}&\mathcal{E}[\![\mathtt{For}\ \mathtt{E_1}\ \mathtt{E_2}\ \mathtt{E_3}\ \mathtt{E_C}]\!]_\rho(\mathtt{s}) = \\ &\quad \mathtt{Let}\{(\mathtt{unit}, \mathtt{s_1}) = \mathcal{E}[\![\mathtt{E_1}]\!]_\rho(\mathtt{s})\} \\ &\quad\quad \{\mathtt{Y}\,\mathtt{g}.\lambda\,\mathtt{g}.\lambda\,\mathtt{s_w}.\mathtt{Let}\{(\mathtt{v}, \mathtt{s_2}) = \mathcal{E}[\![\mathtt{E_2}]\!]_\rho(\mathtt{s_w})\} \\ &\quad\quad\quad \mathtt{if}(\mathtt{false}(\mathtt{v}), \mathtt{s_2}, (\mathcal{E}[\![\mathtt{E_C}]\!]_\rho \circ_{\mathtt{u}} \mathcal{E}[\![\mathtt{E_3}]\!]_\rho \circ_{\mathtt{u}}\ \mathtt{g})(\mathtt{s_2}))\} \\ &\quad\quad (\mathtt{unit}, \mathtt{g}(\mathtt{s_1}))\end{aligned}$$

Ocaml Implementation

$$\begin{aligned}&\mathtt{let\ forExp} = \mathtt{fun}\,(\mathtt{e_1}, \mathtt{e_2}, \mathtt{e_3}) \longrightarrow \mathtt{fun}\,\mathtt{e_c} \longrightarrow \\ &\quad \mathtt{let\ rec\ forLoop} = \mathtt{fun}() \longrightarrow \\ &\quad\quad \mathtt{if}\ \mathtt{e_2}()\ \mathtt{then}\,(\mathtt{e_c}(); \mathtt{e_3}(); \mathtt{forLoop}()) \\ &\quad\quad \mathtt{else}\ () \\ &\quad \mathtt{in}\ (\mathtt{e_1}(); \mathtt{forLoop}());;\end{aligned}$$

Noting the type of:
$\mathtt{forExp}: (\mathtt{unit} \longrightarrow \mathtt{'a}) \times (\mathtt{unit} \longrightarrow \mathtt{bool}) \times (\mathtt{unit} \longrightarrow \mathtt{'b}) \longrightarrow (\mathtt{unit} \longrightarrow \mathtt{'c}) \longrightarrow \mathtt{unit}$

Excercise1 - sol/4

Exercise1.
We decide to add an iterator *for*, to the language Ocaml. Ocaml already has an iterator *for* but we want to add an iterator having the same structure and behavior of *for* of Ansi-C:

(d) Discuss the mechanisms that have been used to do previous point;

Example (Solution. Impementation in Ocaml)

Ocaml Implementation

```
let forExp = fun (e1, e2, e3) → fun ec →
   let rec forLoop = fun() →
            if e2() then (ec(); e3(); forLoop())
            else ()
   in (e1(); forLoop());;
```

The mechanisms are listed below with considerations on the role:

- Expressions instead of Expressions and Commands. Command are viewed as "unit" expressions with side effects. Hence, ordinary expressions (with transparency) are replaced by expressions computing "unit" and producing side-effects;
- Delay Expressions for by Name/Function parameter passing. Delay Expressions have type (unit → 'a) and when used as commands, it becomes (unit → unit);
- Finally, noting:
  - Composition operators: Ocaml's ";" implements $\circ_u$
  - The type, forExp:(unit → 'a) × (unit → bool) × (unit → 'b) → (unit → 'c) → unit

Excercise1 - sol/5

Exercise1.
We decide to add an iterator *for*, to the language Ocaml. Ocaml already has an iterator *for* but we want to add an iterator having the same structure and behavior of *for* of Ansi-C:

(e) Apply the new construct in rephrasing the code below and ...

```
int x=0;
int y=0;
for(x=y=1; x+y<100;x++){x=x-1; y=y+2}
```

Example (Solution. Rephrasing: The session in Ocaml)

```
# let x = ref 0;;                                   comments ...
val x : int ref = {contents = 0}
# let y = ref 0;;
val y : int ref = {contents = 0}
# let init = fun() -> y:=1;x:=1;;                   comments ...
val init : unit -> unit = <fun>
# let test = fun() -> (!x + !y)<100;;               comments ...
val test : unit -> bool = <fun>
# let inc = fun() -> x:= !x + 1;;                   comments ...
val inc : unit -> unit = <fun>
# let cmd = fun() -> x:=!x - 1; y:= !y + 2;;        comments ...
val cmd : unit -> unit = <fun>
# forExp(init,test,inc)cmd;;                        comments ...
- : unit = ()
# !x;;                                              Store state.
- : int = 1
# !y;;
- : int = 99
```

Excercises 2-6

Exercise2.
Complete in Ocaml, the definition of the memoized factorial, discussed in the slides on the memoization. The definition must use a local (hash) table. The (hash) table could be reduced to a simple list of pairs or to a suitable function.
Solution has been given in slide 5 of Lecture20.

Exercise3.
Write, in Ocaml, a definition of QuickSort that must be developed according to the following methodologies: Divide and Conquer, Higher Order, Generic Types
Solution has been given in slide 7 of Lecture20.

Exercise4.
Give, in Ocaml, a tail recursive definition of a function that computes the n-th of the Fibonacci series

Exercise5.
Use iterative HOP for defining, in Ocaml, the size of lists.

Exercise6.
Use Data Extensions Through Functional Abstractions for defining, in Ocaml, data behaving as array of declared size. The new data have the following operations:
array(k) that returns an array with the index ranging over [0,k-1] and with undefined elements;
set(i,u,g) that returns an array that differs from g for the setting to u, of the i-indexed element of g, if any;
get(i,g) that returns the i-indexed element of g, if any;
Remind that You can't use structured types of any sort.

Excercises 2-6 sol/1

Exercise5.
Use iterative HOP for getting a program, in Ocaml, that defines the size of generic lists.
Solution
```
let size n = List.fold_right (fun x → ((+)1)) n 1;;
```

Exercise4.
Give, in Ocaml, a tail recursive definition of a function that computes the n-th of the Fibonacci series

Example (Solution. Ordinary and Tail Recursive Fibonacci n-th)

```
(* ordinary recursive definition *)
let rec fibN = fun n ->
        if n=0 then 1
        else if n=1 then 1
                 else (fibN (n-1)) + (fibN (n-2));;

(* tail recursive definition *)
let fibN = fun n ->
                 let rec innerFibNTl = fun n pred1 pred2 ->
                         if (n=0) then pred2
                         else if (n=1) then pred1
                     else innerFibNTl (n-1) (pred1+pred2) pred1
                  in innerFibNTl n 1 1;;
```

Exercise6.
Use Data Extensions through Functional Abstractions for defining, in Ocaml, data behaving as array of declared size. The new data have the following operations:
array(k,w) that returns an array with the index ranging over [0,k-1] and with all the elements initialized to the (default) value w.
set(i,u,g) that returns an array that differs from g for the setting to u, of the i-indexed element of g, if any;
get(i,g) that returns the i-indexed element of g, if any;
Remind that You can't use structured types of any sort.

Example (Solution. Data Extension through Functional Abstractions of Array)

```
exception ArrayOutOfBoundsException;;
array(k,w) = fun i →
   if i=-1 then k
   else if (i>-1 & i<k) then w else raise ArrayOutOfBoundsException;;
set(i,u,g) = if (i<0 or i>g(-1)) then g
             else fun n → if n=-1 then g(-1)
                          else if n=i then u else g(n)
get(i,g) = g(i)
```

Noting that such a definition could be encapsulated into an abstract data type.

Exercise7.
Though the definition of array in exercise6 uses a representation which is quite protected, it is not completely safe against illegal or inappropriate use.
a. Give a concrete example of this fact;
b. Provide a solution that guarantees complete protection.

Excercise7 sol

Exercise7.
Though the definition of array in exercise6 uses a representation which is quite protected, it is not completely safe against illegal or inappropriate use.
a. Give a concrete example of this fact;
b. Provide a solution that guarantees complete protection.

Example (Solution. Part a: Operations are not protected against fake values)

```
# let anarray = array(3,0);;
val anarray : int → int = <fun>
# get(0,anarray);;
- : int = 0
# get(5,anarray);;
Exception: ArrayOutOfBoundsException.
# let aFake = fun n → if n = -1 then 3 else 5;;
val aFake : int → int = <fun>
# get(5,aFake);;
- : int = 5
# let aFake1 = set(5,12,aFake);;
val aFake1 : int → int = <fun>
# get(5,aFake1);;
- : int = 5
```

The use of aFake1 as value for the operations on "array" result in wrong behaviours and can lead the program into a stuck

Example (Solution. Part b: Use of ADT against fake values)

Complete with an API and one ADT for the API