CSE216 Programming Abstractions

Lazy Evaluation

YoungMin Kwon

SPL: Syntax

```
type expr = B of bool   (*Boolean*)
          | N of int    (*number*)
          | V of string (*variable*)
          (*arithmetic exprs*)
          | Add of expr * expr | Sub of expr * expr
          (*predicates*)
          | Equ of expr * expr | Leq of expr * expr
          (*logical exprs*)
          | And of expr * expr | Or  of expr * expr | Not of expr
          (*conditional expr*)
          | If  of expr * expr * expr
          (*function definition: parameter, body*)
          | Fun of string * expr
          (*closure: parameter, environment, body*)
          | Clo of string * (string * expr) list * expr
          (*lazy* thunk: expr, env*)
          | Thk of expr * (string * expr) list
          (*function application: operator, operand*)
          | App of expr * expr
          (*TODO add a definition for let binding:
            let variable = expr in expr*)
```

Lazy Evaluation

```
let eval expr env =
    let rec force = function Thk (e, ev) -> (*TODO*)
                            | x -> x (*lazy*)
    and eval' expr env =
        let getNum  e = (*TODO*) | e -> print e; assert false in
        let getBool e = (*TODO*) | e -> print e; assert false in
        let getClo  e = (*TODO*) | e -> print e; assert false in

        match expr with
        | B e -> B e
        | N n -> N n
        | V v -> lookup v env
        | Add (a, b)   -> eval' a env |> fun x ->
                          eval' b env |> fun y ->
                          N (getNum x + getNum y)
        | Sub (a, b)   -> (*TODO*)
…
```

Lazy Evaluation

```
| If (c, t, f) -> (*TODO: eval c, if true eval t;
                          otherwise eval f*)

| Fun (v, e)   -> Clo (v, env, e)
| App (f, a)   -> (*TODO: eval the body of f in the
                          extended env with a binding of
                          param of f and thunk of a*)

(*TODO: add a case for Let*)
| _ -> assert false
```

Lazy Evaluation

```
(* Test let
 *)
let test1 () =
    let sum = Fun ("self", Fun ("x",
                    Let ("sum", App (V "self", V "self"),
                    Let ("dec", Fun ("x", Sub (V "x", N 1)),
                    If (Equ (V "x", N 0),
                        N 0,
                        Add (V "x",
                             App (V "sum",
                                  App (V "dec", V "x")))))))) in
    print (eval (App (App (sum, sum), N 10)) [])

let _ = test1 ()
```

Lazy Evaluation

```
(* Test lazy eval
 *)
let test2 () =
    let open SyntaticSugar in

    (*y combinator*)
    let y = "f" @ ("k" @ !"k" ->> !"k") ->>
                  ("g" @ !"f" ->> (!"g" ->> !"g")) in

    (*cons/car/cdr*)
    let cons = "x" @ "y" @ "b" @ If (!"b", !"x", !"y") in
    let car  = "p" @ !"p" ->> B true in
    let cdr  = "p" @ !"p" ->> B false in

    (*natural numbers from n*)
    let nat = y ->> ("nat" @ "n" @
        cons ->> !"n" ->> (!"nat" ->> (!"n" + N 1))) in
…
```

Lazy Evaluation

```
…
    (*pick n-th number in l*)
    let num = y ->> ("num" @ "l" @ "n" @
        If (!"n" = N 0,
            car ->> !"l",
            !"num" ->> (cdr ->> !"l") ->> (!"n" - N 1))) in

    (*infinite loop*)
    let sink = y ->> ("sink" @ "x" @ !"sink" ->> N 0) in

    (*check if eval works with sink*)
    print (eval (car ->> (cons ->> N 1 ->> (sink ->> N 0))) []);

    (*get the fifth number from the stream of natural numbers*)
    print (eval (num ->> (nat ->> N 0) ->> N 5) [])

let _ = test2()
```

Lazy Evaluation

- Expected result

```
val test1 : unit -> unit = <fun>
55
- : unit = ()

…

val test2 : unit -> unit = <fun>
1
5
- : unit = ()
```