Interpreter Ocaml

Project for the subject “Linguaggi”

University of Verona, 2017

Nicola Gemo

Vr386790

The project

This project is based on the interpreter that we saw in the lessons extended with support to string type, function to manipulate them and a reflect function that apply the interpreter to a string that represent a list of commands.
This project is write in Ocaml using Denotational Semantic and it's about 700 line of code.

Considerations:
Not all the expressions are implemented into the reflect.
The String that the reflect analyze must be without spaces.

The String type

Following all the changes to do to the types:

```
| Estring of string                    (*Contructor for Estring*)
| Len of exp                           (*Constructor of Length*)
| Concat of exp * exp                  (*Constructor of Concatenation*)
| Substr of exp * exp * exp            (*Constructor of Substr*)
| Trim of exp                          (*Constructor of Trim*)
| Uppercase of exp                     (*Constructor of Uppercase*)
| Lowercase of exp                     (*Constructor of Lowercase*)
```

These are the new expression added to the **exp** type:

Estring: Expression that represent the string type

Len: Expression that handle the request of the string length

Concat: Expression for the concatenation of two Strings

Substr: Expression for the resize the String

Trim: Expression for delete space before and after the text of a String

Uppercase: Expression for transform the String in uppercase

Lowercase: Expression for transform the String in lowercase

```
| Mstring of string                    (*Constructor of string type for memory*)
```

The representation of the String in the **mval** type, used for the memory

```
| String of string                     (*Constructor of string type*)
```

The representation of the String in the **eval** type, used for the evaluated expressions.

```
| Dstring of string                    (*Constructor of string type for enviroment*)
```

The representation of the String in the **dval** type, used in the environment.

```
let evaltomval e = match e with
    | Int n -> Mint n
    | Bool n -> Mbool n
    | String n -> Mstring n                (*From eval to mval*)
    | _ -> raise Nonstorable
let mvaltoeval m = match m with
    | Mint n -> Int n
    | Mbool n -> Bool n
    | Mstring n -> String n                (*From mval to eval*)
    | _ -> Novalue
let evaltodval e = match e with
    | Int n -> Dint n
    | Bool n -> Dbool n
    | String n -> Dstring n                (*From eval to mval*)
    | Novalue -> Unbound
    | Funval n -> Dfunval n
let dvaltoeval e = match e with
    | Dint n -> Int n
    | Dbool n -> Bool n
    | Dstring n -> String n                (*From dval to eval*)
    | Dloc n -> raise Nonexpressible
    | Dfunval n -> Funval n
    | Dprocval n -> raise Nonexpressible
    | Unbound -> Novalue
```

New transformation between types for String.

```
let typecheck (x, y) = match x with
    | "int" -> (match y with
        | Int u -> true
        | _ -> false)
    | "bool" -> (match y with
        | Bool u -> true
        | _ -> false)
    | "string" -> (match y with         (*return true only if is a string*)
        | String(u) -> true
        | _ -> false)
    | _ -> failwith ("not a valid type")
```

Modified type check for the String type

The String functions

len:

```
let len x =                 (*input: string, output: length of the string*)
    if typecheck("string", x)
        then (match x with String(u) -> Int(String.length u))
        else failwith("type error")
```

```
val len : eval -> eval = <fun>
```

This function takes in input a String and return the length of it.

concat:

```
let concat (x,y)=           (*input: (string,string)  , output: the concatenation of the two string *)
    if typecheck("string",x) & typecheck("string",y)
        then (match (x,y) with (String(u),String(w)) -> String(String.concat "" [u;w]) )
        else failwith("type error")
```

```
val concat : eval * eval -> eval = <fun>
```

This function takes in input a String and return the merge between them.

subsrt:

```
let substr (x,y,z)=        (*input: (string, inizio , fine)  , output: the substring starting from inizio to fine *)
    if typecheck("string",x) & typecheck("int",y) & typecheck("int",z)

        then (match (x,y,z) with (String(s),Int(i),Int(j)) -> String(
            if (j-i)>0
                then    String.sub s i (j-i)
                else    ""
        ))
        else failwith("type error")
```

```
val substr : eval * eval * eval -> eval = <fun>
```

This function returns the substring of the string x from the position y to z.

trim:

```
let trim x =               (*input: string  , output: the string without space in the beginning and end *)
    if typecheck("string", x)
        then (match x with String(u) -> String(String.trim u))
        else failwith("type error")
```

```
val trim : eval -> eval = <fun>
```

This function returns the String without space before and after the text.

uppercase:

```
let uppercase x =           (*input: string , output: the string in uppercase *)
    if typecheck("string", x)
        then (match x with String(u) -> String(String.uppercase u))
        else failwith("type error")
```

```
val uppercase : eval -> eval = <fun>
```

This function returns the String x in uppercase.

lowercase:

```
let lowercase x =           (*input: string , output: the string in lowercase *)
    if typecheck("string", x)
        then (match x with String(u) -> String(String.lowercase u))
        else failwith("type error")
```

```
val lowercase : eval -> eval = <fun>
```

This function returns the String x in lowercase.

```
| Len(a) -> len( (sem a r s) )                                (*return the string lenght*)
| Concat(a,b) -> concat((sem a r s), (sem b r s))             (*return the concatenated string*)
| Substr(a,b,c) -> substr((sem a r s),(sem b r s),(sem c r s))  (*return the cutted string*)
| Trim(a) -> trim(((sem a r s)))                              (*return the string without space in the beginning and end*)
| Lowercase(a) -> lowercase (((sem a r s)))                   (*return the string in lowercase*)
| Uppercase(a) -> uppercase (((sem a r s)))                   (*return the string in uppercase*)
```

Update the **sem** function with the new **exp** to match.

Reflect

This function takes in input a String and return the command list represented by that String

Following the code for implement this function:

```
| Reflect of string                          (*Constructor od Reflect*)
```

The new command for the Reflect to add in com type

```
| Reflect(a) -> semcl (reflect(a)) r s          (*return the command list rappresented by the string*)
```

Update the **semc** function with the new **com** to match.

explode:

```
let explode s =             (*input: string , output: convert the string into a list of char *)
    let rec expl i l =
        if i < 0 then l else
            expl (i - 1) (s.[i] :: l) in
    expl (String.length s - 1) [];;
```

```
val explode : string -> char list = <fun>
```

This function takes in input a String and return the char list.

implode:

```
let implode l =            (*input: list of char, output: convert the list of char into a string *)
    let result = String.create (List.length l) in
    let rec imp i = function
    | [] -> result
    | c :: l -> result.[i] <- c; imp (i + 1) l in
    imp 0 l;;
```

```
val implode : char list -> string = <fun>
```

This function takes in input a char list and return the String.

stringToStringList:

```
let stringToStringList c s= (*input: char string, output: divide the string using the given character , ignoring the character inside brackets*)
    let sl=String.length s in
    let rec loop fine parentesi parentesiG r i=
        if i<0 then (String.sub s (i+1) (fine-(i)) :: r)
        else if s.[i]==c && parentesi==0 && parentesiG==0 then loop (i-1) (parentesi)  (parentesiG) (String.sub s (i+1) (fine-(i)) :: r) (i-1)
        else if s.[i]=='(' then loop (fine) (parentesi-1) (parentesiG) r (i-1)
        else if s.[i]==')' then loop (fine) (parentesi+1) (parentesiG) r (i-1)
        else if s.[i]=='[' then loop (fine) (parentesi) (parentesiG-1) r (i-1)
        else if s.[i]==']' then loop (fine) (parentesi) (parentesiG+1) r (i-1)
        else loop (fine) (parentesi) (parentesiG) r (i-1)

    in
    loop (String.length s - 1) 0 0 [] (String.length s -1)
```

```
val stringToStringList : char -> string -> string list = <fun>
```

This function return the string List represented by the String s divided using the character c, ignoring the character if is inside brackets.

stringToExp:

```
let rec stringToExp x=      (*input: String, output: return the expression rappresented by the string *)
    let s =explode x in
    match s with
    |   'E'::'i'::'n'::'t'::'('::r ->   Eint(int_of_string (String.sub (implode r) 0 (String.length (implode r) - 1)) )
    |   'E'::'s'::'t'::'r'::'i'::'n'::'g'::'('::r ->  Estring(String.sub (implode r) 1 ((String.length (implode r)) - 3))
    |   'E'::'b'::'o'::'o'::'l'::'('::r ->  Ebool(bool_of_string (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'D'::'e'::'n'::'('::r ->    Den(String.sub (implode r) 1 ((String.length (implode r)) - 3))
    |   'V'::'a'::'l'::'('::r ->    Val(stringToExp (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'N'::'o'::'t'::'('::r ->    Not(stringToExp (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'E'::'q'::'('::r ->
                let l = stringToStringList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                    if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                    else Eq(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))
    |   'I'::'s'::'z'::'e'::'r'::'o'::'('::r ->    Iszero(stringToExp (String.sub (implode r) 0 (String.length (implode r) - 1)))
    |   'P'::'r'::'o'::'d'::'('::r ->
                let l = stringToStringList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                    if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                    else Prod(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))
    |   'D'::'i'::'f'::'f'::'('::r ->
                let l = stringToStringList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                    if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                    else Diff(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))
    | _ -> failwith ("Errore sintassi, stringToExp, " ^ (implode s))
```

```
val stringToExp : string -> exp = <fun>
```

This function takes in input the String s that represent an expression.

It transforms the String in a char list and perform a match operation for select the right expression to return.

stringToCom:

```
let rec stringToCom x=      (*input: String, output: return the command rappresented by the string *)

    let s = explode x in
    match s with
    |   'A'::'s'::'s'::'i'::'g'::'n'::'('::r ->
            let l = stringToStringList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s) )
                else Assign(stringToExp(List.hd l) , stringToExp(List.hd (List.tl l)))

    |   'C'::'i'::'f'::'t'::'h'::'e'::'n'::'e'::'l'::'s'::'e'::'('::r ->
            let l = stringToStringList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                if (List.length l) != 3  then failwith ("Errore sintassi, " ^ (implode s))
                else Cifthenelse(stringToExp(List.hd l) ,
                                reflect(String.sub (List.hd (List.tl l)) 1 (String.length (List.hd (List.tl l)) - 2)),
                                reflect(String.sub (List.hd (List.tl (List.tl l))) 1 (String.length (List.hd (List.tl (List.tl l))) - 2))
                                )

    |   'W'::'h'::'i'::'l'::'e'::'('::r ->
            let l = stringToStringList (',') (String.sub (implode r) 0 (String.length (implode r) - 1)) in
                if (List.length l) != 2  then failwith ("Errore sintassi, " ^ (implode s))
                else While(stringToExp(List.hd l) ,
                                reflect(String.sub (List.hd (List.tl l)) 1 (String.length (List.hd (List.tl l)) - 2))
                                )


    | _ -> failwith ("Errore sintassi, stringToExp, " ^ (implode s))
```

```
val stringToCom : string -> com = <fun>
```

This function takes in input the String s that represent a command.

It transforms the String in a char list and perform a match operation for select the right command to return.

reflect:

```
        (*input: string, output: the command list rappresented by the string *)
and reflect (s:string) = List.map stringToCom (stringToStringList ';' s)
```

```
val reflect : string -> com list = <fun>
```

The reflect function, it returns the list of commands created by running the function “stringToCom” to each element of the input String s divided by the function “stringToStringList” into a list.

Examples of expressions and commands

```
(* 1 TEST Sum*)
(*5+6*)
let ex1=Sum(Eint 5,Eint 6);;
(*configuro memoria e ambiente*)
let rho1=Funenv.emptyenv(Unbound);;
let sigma1=Funstore.emptystore(Undefined);;
(*interprete*)
let result1=sem ex1 rho1 sigma1
```

```
val result1 : eval = Int 11
```

```
(* 2 TEST Ifthenelse,Eq,Minus,Eint,Diff*)
(*
    if(0==(5-5)
        then 1
        else 0
*)
let ex2=Ifthenelse(Eq(Eint(0),Diff(Eint(5),Minus(Eint(-5)))),Eint 1,Eint 0)
(*configuro memoria e ambiente*)
let rho2=Funenv.emptyenv(Unbound);;
let sigma2=Funstore.emptystore(Undefined);;
(*interprete*)
let result2=sem ex2 rho2 sigma2
```

```
val result2 : eval = Int 1
```

```
(* 3 TEST Newloc, While,Assign,Val *)
(*
    z=4
    w=1
    while(!(z==0)){
        w=w*z
        z=z-1
    }
*)
let d3 = [("z",Newloc(Eint 4));("w",Newloc(Eint 1))];;
let ex3 = [While(Not(Eq(Val(Den "z"), Eint 0)),
    [Assign(Den "w", Prod(Val(Den "w"),Val(Den "z")));
    Assign(Den "z", Diff(Val(Den "z"), Eint 1))])];;
(*configuro memoria e ambiente*)
let (rho3, sigma3) = semdv d3 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
(*interprete*)
let sigma3final=semcl ex3 rho3 sigma3
let result3Z= sem (Val(Den "z")) rho3 sigma3final;;
let result3W= sem (Val(Den "w")) rho3 sigma3final;;
```

```
val result3Z : eval = Int 0
val result3W : eval = Int 24
```

```
(* 4 TEST Let, Fun,Rec *)
(*
    {
        fact(x)
            if(x==0)
                then 1
                else x*fact(x-1)
        result4=fact(4)
    }
*)
let ex4=Let(
    "fact",
    Rec("fact",
        Fun(
            ["x"],
            Ifthenelse(
                Eq(Den "x", Eint 0),
                Eint 1,
                Prod(Den "x", Appl (Den "fact", [Diff(Den "x", Eint 1)]))
            )
        )
    ),
    Appl(Den "fact",[Eint 4])
 )
 (*configuro memoria e ambiente*)
let rho4=Funenv.emptyenv(Unbound);;
let sigma4=Funstore.emptystore(Undefined);;
(*interprete*)
let result4= sem ex4 rho4 sigma4;;
```

```
val result4 : eval = Int 24
```

```
(* 5 TEST Block *)
(*
    z=4
    w=1
    while(!(z==0)){
        w=w*z
        z=z-1
    }
*)
let d5 = ([("z",Newloc(Eint 4));("w",Newloc(Eint 1))],[]);;
let ex5 = [While(Not(Eq(Val(Den "z"), Eint 0)),
    [Assign(Den "w", Prod(Val(Den "w"),Val(Den "z")));
    Assign(Den "z", Diff(Val(Den "z"), Eint 1))]);
    Assign(Den "y", Val(Den "w"))
    ];;
let (ex5: block) =(d5,ex5)
(*configuro memoria e ambiente*)
let dr = [("y",Newloc(Eint 0))];;
let (rho5, sigma5) = semdv dr (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
 (*interprete*)
let result5 = semb ex5 rho5 sigma5 ;;
let result5y= sem (Val(Den "y")) rho5 result5;;
(*let result5Z= sem (Val(Den "z")) rho5 result5;;*)
```

```
val result5y : eval = Int 24
```

```
(* 5.5 TEST Block,procedure funzioni *)
(*
    int r=0

    mul2(int n){
        return n*2
    }

    testproc(int p){
        w=1
        w=p+1
        r=mul(w)
    }

    testproc(4)
*)
let(ex55: block) =
 (  ([],
    [
        ("mul2", Fun(["x"],
                Prod(Eint 2,Den "x"))
        );
        ("testproc", Proc(
                            ["x"],
                                ((([("z", Newloc(Den "x"));("w", Newloc(Eint 1))],
                                []),
                                [
                                    Assign(Den "w",Sum(Val(Den "z"),Val(Den "w")));
                                    Assign (Den "r", Appl (Den "mul2", [Val(Den "w")]))
                                ])
                        )
        )
    ]),
    [ Call(Den "testproc", [Eint 4])]) ;;

(*configuro memoria e ambiente*)
let dr55 = [("r",Newloc(Eint 0))];;
let (rho55, sigma55) = semdv dr55 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
 (*interprete*)
let result55 = semb ex55 rho55 sigma55 ;;
let result55y= sem (Val(Den "r")) rho55 result55;;
```

```
val result55y : eval = Int 10
```

Examples of String function

```
(* 6 TEST stringa lenght*)
(*   length("ciao")     *)
let ex6= Len(Estring("ciao"));;
(*configuro memoria e ambiente*)
let rho6=Funenv.emptyenv(Unbound);;
let sigma6=Funstore.emptystore(Undefined);;
(*interprete*)
let result6=sem ex6 rho6 sigma6
```

```
val result6 : eval = Int 4
```

```
(* 7 TEST stringa concat*)
(* concat("ciao"," come va")*)
let ex7=Concat(Estring("Ciao") ,Estring(" come va?"));;
(*configuro memoria e ambiente*)
let rho7=Funenv.emptyenv(Unbound);;
let sigma7=Funstore.emptystore(Undefined);;
(*interprete*)
let result7=sem ex7 rho7 sigma7
```

```
val result7 : eval = String "Ciao come va?"
```

```
(* 8 TEST stringa substr*)
(* substr("Meraviglioso",5,6)*)
let ex8=Substr(Estring("Meraviglioso"),Eint(5),Eint(6))
(*configuro memoria e ambiente*)
let rho8=Funenv.emptyenv(Unbound);;
let sigma8=Funstore.emptystore(Undefined);;
(*interprete*)
let result8=sem ex8 rho8 sigma8
```

```
val result8 : eval = String "i"
```

```
(* 9 TEST stringa trim*)
(*   trim("     Ciao  ")     *)
let ex9= Trim(Estring("     Ciao  "));;
(*configuro memoria e ambiente*)
let rho9=Funenv.emptyenv(Unbound);;
let sigma9=Funstore.emptystore(Undefined);;
(*interprete*)
let result9=sem ex9 rho9 sigma9
```

```
val result9 : eval = String "Ciao"
```

```
(* 10 TEST stringa lower/uppercase*)
(*    concat(uppercase("Ciao"),lowercase"Come Va"))   *)
let ex10= Concat(Uppercase(Estring("Ciao")) ,Lowercase(Estring(" Come Va?")));;
(*configuro memoria e ambiente*)
let rho10=Funenv.emptyenv(Unbound);;
let sigma10=Funstore.emptystore(Undefined);;
(*interprete*)
let result10=sem ex10 rho10 sigma10
```

```
val result10 : eval = String "CIAO come va?"
```

```
(* 11 TEST memoria con stringhe*)
(*
    z="TEST"
    z=lowercase(z)
*)
let d11 = [("z",Newloc(Estring("TEST")))]
let ex11 = [
    (Assign(Den "z",Lowercase(Val(Den "z"))))
];;
(*configuro memoria e ambiente*)
let (rho11, sigma11) = semdv d11 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
(*interprete*)
let sigma11final=semcl ex11 rho11 sigma11
let result11Z= sem (Val(Den "z")) rho11 sigma11final;;
```

```
val result11Z : eval = String "test"
```

Examples of Reflect

```
(* 12 TEST reflex con if e memoria *)
(*
    z=1
    z=2
    if(false)
        then z=5
        else  z=10
*)
let d12 = [("z",Newloc(Eint 1))];;
let ex12 = [
    Reflect("Assign(Den(\"z\"),Eint(2));Cifthenelse(Ebool(false),[Assign(Den(\"z\"),Eint(5))],[Assign(Den(\"z\"),Eint(10))])")
];;
(* creo com list per debug*)
let cl12= reflect("Assign(Den(\"z\"),Eint(5));Cifthenelse(Ebool(true),[Assign(Den(\"z\"),Eint(5))],[Assign(Den(\"z\"),Eint(5))])")
(*configuro memoria e ambiente*)
let (rho12, sigma12) = semdv d12 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
(*interprete*)
let sigma12final=semcl ex12 rho12 sigma12
let result12Z= sem (Val(Den "z")) rho12 sigma12final;;
```

```
val result12Z : eval = Int 10
```

```
(* 13 TEST reflex while espressioni memoria... *)
(*
    z=4
    w=1
    while(!(z==0)){
        w=w*z
        z=z-1
    }
*)
let d13 = [("z",Newloc(Eint 4));("w",Newloc(Eint 1))];;
let ex13 = [
    Reflect("While(Not(Eq(Val(Den(\"z\")),Eint(0))),[Assign(Den(\"w\"),Prod(Val(Den(\"w\")),Val(Den(\"z\"))));Assign(Den(\"z\"),Diff(Val(Den(\"z\")),Eint(1)))])")
];;
(* creo com list per debug*)
let cl13= reflect("While(Not(Eq(Val(Den(\"z\")),Eint(0))),[Assign(Den(\"w\"),Prod(Val(Den(\"w\")),Val(Den(\"z\"))));Assign(Den(\"z\"),Diff(Val(Den(\"z\")),Eint(1)))])")
(*configuro memoria e ambiente*)
let (rho13, sigma13) = semdv d13 (Funenv.emptyenv Unbound) (Funstore.emptystore Undefined);;
(*interprete*)
let sigma13final=semcl ex13 rho13 sigma13
let result13Z= sem (Val(Den "z")) rho13 sigma13final;;
let result13W= sem (Val(Den "w")) rho13 sigma13final;;
```

```
val result13Z : eval = Int 0
val result13W : eval = Int 24
```