LANGUAGES

OCAML: INTERPRETER

University of Verona

COMPUTER SCIENCE DEPARTMENT

Documentation

Giovanni Bellorio

SUMMARY

OCAML

PROJECT ANALYSIS

The whole project can be summarized in these steps:

Extend the OCAML interpreter learnt during languages choosing one of the three ways studied: operational, denotational or iterative. The solution should contain the base code offered by the lecturer, which already has procedures, functions and imperative base code, excluding the implementation of objects.

To extend the language, the student must add:

- String type s
- The string constant in the semantics like a sequence of alphanumeric characters
- The calculation of the length of a string: len(s)
- The concatenation of two strings: si@s2
- The substring operator: substring (s, idx1, idx2)

The possibility of introducing of extensions or operations taking note from other scripting languages in left to the student to decide.

The extension of the project requires the implementation of the reflect(s) command which takes a string s as an input and it calls the same interpreter on s as a sequence of commands.

OCAML

REPRESENTATION

The project was grouped in this manner:

We used the denotational semantics to write the interpreter.
A script that loads all the various files containing the ml code was used to simplify the use of the interpreter. See the image below to see how to run the interpreter:

```
lMacBook-Pro-di-Giovanni:progetto Giovanni$ ocaml
        OCaml version 4.03.0

# #use "script.ml";;
```

The project was divided in syntax, environment interface, memory interface, stack interface, semantic domain eval, operations and semantics. Here is a brief description of the various files in the project:

```
#use "1_Sintassi.ml";;

#use "2_Env_Interfaccia.ml";;
#use "2_Env_Semantica.ml";;
open Funenv;;

#use "3_Store_Interfaccia.ml";;
#use "3_Store_Semantica.ml";;
open Funstore;;

#use "4_Stack_Interfaccia.ml";;
#use "4_Stack_Semantica.ml";;
open SemPila;;

#use "4_StackM_Interfaccia.ml";;
#use "4_StackM_Semantica.ml";;
open SemMPila;;

#use "5_Eval.ml";;
#use "6_Operazioni.ml";;
#use "7_Semantica.ml";;
```

- **sintassi**, it contains the syntax domain and the definition of the syntax of the denotational semantics we chose to work with.
- **env_interfaccia**, it contains the environment signature.
- **env_semantica**, the implementation of the environment interface is done here.
- **store_interfaccia**, it contains the memory's signature
- **store_semantica**, it contains the implementation of the interface of store.
- **stack_interfaccia**, it contains the signature of the non-modifiable stack interface.
- **stack_semantica**, it contains the implementation of the interface of the non-modifiable stack.
- **stackM_interfaccia**, it contains the signature of the modifiable stack.
- **stackM_semantica**, it contains the implementation of the modifiable stack.
- **eval**, it contains the semantic domain. (expressible eval, declarable dval, storable mval) and type conversion.
- **operazioni**, it contains the type check which is used to test, from time to time, the parameters of operations and the implementation of all the operations.
- **semantica**, it contains the definition of the semantics of all the syntactic parts needed by the interpreter.

OCAML

OPERATIONS

The project has been completed like this:

After a rapid analysis of the project and all the requirements, we decided to develop a type of code that is similar to an imperative language which also covers a functional language requiring an environment, a memory and an expression to evaluate. Another technical choice was to use a denotational interpreter.

To support string input, the string type was added to the syntax of the language (type exp) and also to the semantics (type eval expressible, type eval denotable and type mval storable).

```
type exp =

  (* costanti *)
  | Eint of int
  | Ebool of bool
  | Estring of string
```

```
type eval =
    | Int of int
    | Bool of bool
    | Funval of efun
    | String of string
    | Novalue
```

The operations on expressions were implemented in operazioni.ml. This file contains the type check declared in the syntax with the number of parameters. In the extended language there are three types of data: Int, Bool, and String. To run an operation, we have to be sure that the arguments are correct otherwise the interpreter won't run. Here is an example of a sum of two strings that can't be computed.

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

The len operation returns an integer with the length of the strings passed as an expression. To implement the len operation, we used the string libraries in Ocaml (#use string.ml). Firstly, we put a type check in len() and we converted the function len() of the library in the String type needed for the project.

```
let len x =
  if typecheck("char",x) then
    (match x with String(x) -> Int(String.length x))
  else
    failwith ("type error")
```

The same implementation was done for the concat operation. Given two strings, we have to concatenate them and return the result string. In the library, the concat operation is peculiar because it has been extended to concatenate a list of strings. The trick was to transform the string to be concatenated into a list and then joining it to a list of empty strings, so the concatenation would be with a string and a list that contains a valid, non-empty string.

```
let concat (x, y) =
  if typecheck("char",x) && typecheck("char",y) then
    (match (x,y) with (String(u), String(w)) -> String(String.concat (u) [""];w]))
  else
    failwith ("type error")
```

The substring operation requires various checks other than a type check. The length of the string is checked using the compare function which verifies if its numerical argument (the length of s) is greater than the other argument (in this case 0). After that, the two indexes have to be greater than 0, the first index should be less than the second one and both indexes should be less than the length of the string.

It is a requirement of the project that a string should be extracted from position i to position j, however, the substring function from the Ocaml library works in a different manner. It extracts the string from position i for j characters. When we pass the index i as the starting position and we calculate j as the difference between the two indexes plus one everything seems to work appropriately, and the substring required is returned only if the conditions mentioned earlier are satisfied.

All these operations were put in the semantics of the language and the can be called dynamically by passing the correct number of parameters already having a type eval (no more exp) of the semantics.

```
val sem : exp -> dval Funenv.env -> mval Funstore.store -> eval = <fun>
```

```
let rec sem (e:exp) (r:dval env) (s:mval store) = match e with
                | Estring(c)      -> String(c)
                | Len(c)          -> len(sem c r s)
                | Concat(c1, c2)  -> concat(sem c1 r s, sem c2 r s)
                | Substr(c, a, b) -> substr(sem c r s, sem a r s, sem b r s)
```

OCAML

REFLECT

The reflect command takes a string as an input and it converts it in expressions recalling on itself the command interpreter. The command itself was implemented in the semantics of the commands of our extended language. The semantics of the commands returns a memory (store) that was defined beforehand. To view the results, we must allocate and declare an output variable of parse in the memory and then switch to the semantics of the command, the environment and the memory in which we are working. The variable named result will be the variable to be initialized and it will be used to store the result of all our operations. We have to implement a parsing mechanism from string to exp.

```
val parseric : eval * exp SemMPila.stack * eval SemMPila.stack -> exp = <fun>
```

The idea implemented for the solution of the Reflect problem can be summarized in these examples:

Example 1:

```
Sum(Eint a,Eint b)
Eint a,Eint b)
Eint b
)
```

Example 2:

```
Sum(Sum(Eint a,Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
Sum(Eint a,Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
Eint a,Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
Sum(Eint a,Eint b)),Diff(Eint a,Eint b))
,Diff(Eint a,Eint b))
)
```

The parse implemented in our extended version of the language requires three parameters: the expression (string), a stack for intermediate results (of type exp), and a stack that contains the remaining expression to be converted of type eval. The parse function uses a dynamic recursion and our solution interprets the string sequentially. There is no initial checking of the formatting and content. The process recognizes various internal commands using the substring function that checks the characters of the string with real expressions. Once the expressions are recognized, the interpreter searches for their arguments (every operation has a different number of arguments) which could be operations that returns a base result. To do such operations, the checking of the number of commas in the operation is limited, so we had to use a stack. The result of the parsing of the remainder of the string is pushed onto the stack until we reach the base case of the string (until we find an integer, a Boolean). The conversion happens by writing the operation as an exp. To obtain the content of the stack, the topandpop function was used. The topandpop function automatically tops and pops the stack.

```
val topandpop : 'a SemMPila.stack -> 'a = <fun>
```

The base cases are implemented in a way that concludes partially the operation, saving its (partial) results onto the stack and checking the position of comma and saving onto the auxiliary stack the remaining string to scan. In case we scan an integer or a Boolean, the default libraries were used to convert them from string to integer or Boolean.

```
val convert : eval -> int = <fun>
val convert_bool : eval -> bool = <fun>
```

Analysing the problem, its necessary to put some conditions so that the operation can continue: in case we find a closing parenthesis “)” out of place due to multiple nested operations or when find too many commas that divide the arguments of operations.

```
else if equals(substr(String(n),Int(0),Int(0)),String(",")) then
  parseric(String(String.sub (n) 1 (((String.length) n)-1)),stack,stackstr)

else if equals(substr(String(n),Int(0),Int(0)),String(")")) && (String.length(n)!=1) then
  parseric(String(String.sub (n) 1 (((String.length) n)-1)),stack,stackstr)
```

OCAML

LIMITATIONS

The implemented solution implies the use of the stack. To initialize a modifiable stack, its necessary that we specify what type of stack and also the number of cells to reserve in memory to save data.

```
val emptystack : int * 't -> 't stack
```

For the result of the result command, we can be sure that it is only one but we can’t know the number of operations that can be evaluated so we imposed a maximum number of 100 operations. Ideally it could resolve an infinite number of recursions but we imposed the limit.

The warnings, in this project, are because of non-used variables declared with let.

It’s possible to print the results of reflect only in the result variable only if it has been initialized.

```
let com = Reflect(Estring s);;

let (l,sigma) = allocate(emptystore(Undefined), Undefined);;
let rho = bind (emptyenv(Unbound), "result", Dloc l);;
let sigma1 = semc com rho sigma;;

sem (Val(Den "result")) rho sigma1;;
```

OCAML
EXAMPLES

```
################ESEMPI SEMANTICA#############################################################

let l = "CIAO";;
let l1 = "COME";;
let l2 = "VA";;

sem (Estring l) (emptyenv Unbound) (emptystore Undefined);;
sem (Len(Estring l)) (emptyenv Unbound) (emptystore Undefined);;
sem (Len(Estring l1)) (emptyenv Unbound) (emptystore Undefined);;
sem (Len(Estring l2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Concat(Estring l1, Estring l2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Concat(Estring l, Estring l2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 1, Eint 3)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 1, Eint 2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 0, Eint 1)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 4, Eint 10)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 10, Eint 12)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 2)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 3)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 0)) (emptyenv Unbound) (emptystore Undefined);;
sem (Substr(Estring l, Eint 2, Eint 0)) (emptyenv Unbound) (emptystore Undefined);;

################ESEMPI REFLECT#########################################

let s = "Sum(Eint 1,Sum(Eint 5,Eint 4))";;
let s = "Sum(Sum(Eint 10,Eint 1),Diff(Eint 2,Eint 1))";;
let s = "Sum(Len(Estring CIAO),Eint 1)";;
let s = "Len(Estring CIAO)";;
let s = "Concat(Estring CIAO,Estring CIAO)";;
let s = "Substr(Estring CIAO,Eint 0,Eint 1)";;
let s = "Substr(Estring CIAO,Sum(Eint 0,Eint 1),Sum(Eint 1,Eint 1))";;

let com = Reflect(Estring s);;

let (l,sigma) = allocate(emptystore(Undefined), Undefined);;
let rho = bind (emptyenv(Unbound), "result", Dloc l);;
let sigma1 = semc com rho sigma;;

sem (Val(Den "result")) rho sigma1;;
```