unicaml

Operational interpreter of a functional language

In this lesson we will develop the operational semantics of a quite simple *functional* language. The language has integer of boolean expression, the possibility of building complex expressions with operators like `Sum`, `Or`, has an `if then else`, and variables (with the usual functional memaning). We will add later what is now left to make the language Turing powerful.

Syntax and environment

We start recalling that a functional language returns *expressible values*, hence we have to introduce a type for the result of the evaluation of an expression written in the functional language:

```
type eval = None
          | Int of int
          | Bool of bool
```

The syntax of the language is the following:

```
type exp =
        Eint of int
      | Ebool of bool
      | Den of ide
      | Prod of exp * exp
      | Sum of exp * exp
      | Diff of exp * exp
      | Eq of exp * exp
      | Minus of exp
      | Iszero of exp
      | Or of exp * exp
      | And of exp * exp
      | Not of exp
      | Ifthenelse of exp * exp * exp
```

where the type `ide` is the one for variables:

```
type ide = string
```

Note. Recall to write everything in the proper order or to embrace them with the keyword `and`.

Recall the implementation of an environment introduced in the last lesson of the crash course, we adapt it to our purpose.

Suppose we want to define a *partial* function `f` with domain `'b` and codomain `'a`. Then, we can model `f` as a *total* function from `'b` to the "lifted" codomain `'a partial` (lifted means that we add an element to explicitly say that the function on an argument is not defined).

We do not have to introduce a new "lifted" (co)domain, as we have already it: `eval` is a lifted domain (the elements are *integers* and *boolean*, with the proper constructor, and the value `None` represents no result).

The following function, given a predicate `p: 'a → bool`, a function `f : 'a → eval` and a list, finds the first element of the list (if any) that satisfies the predicate.

```
# let rec find p f = function
    [] -> None
  | x::l -> if p x then f x else find p f l;;

val find : ('a -> bool) -> ('a -> eval) -> 'a list -> eval = <fun>
```

(we added the function `f` as we have to implement a projection from a pair…)

This function can be used to implement an **environment**, i.e. a partial mapping from identifiers to values, as a list of pairs (id,value).

First, we define a function `bind`, that binds an identifier `x` to a value `v` in the given environment.

```
# let rec bind x v = function
    [] -> [(x,v)]
  | (y,v')::l when x=y -> (x,v)::l
  | (y,v')::l -> (y,v')::(bind x v l);;

val bind : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

Then, we define the function `applyenv`, that searches the environment for the value bound to the identifier `x`. We model it as a partial function that produces `None` if `x` is not bound in the environment, or `v` if `x` is bound to `v`.

```
# let applyenv env x = (find (fun (y,v) -> x=y) (fun (y,v) -> v)) env;;

val applyenv : ('a * eval) list -> 'a -> eval = <fun>
```

Here are some examples:

```
# let rho0 = [("x",Int 3);("y",Int 5)];;
val rho0 : (string * eval) list = [("x", Int 3); ("y", Int 5)]

# applyenv rho0 "z";;
- : eval = None

# applyenv rho0 "x";;
- : eval = Int 3
```

The behaviour of the function `bind` does not change:

```
# let rho1 = bind "x" (Int 7) rho0;;
val rho1 : (string * eval) list = [("x", Int 7); ("y", Int 5)]

# applyenv rho1 "x";;
- : eval = Int 7

# let rho2 = bind "z" (Int 9) rho1;;
val rho2 : (string * eval) list = [("x", Int 7); ("y", Int 5); ("z", Int 9)]

# applyenv rho2 "z";;
- : eval = Int 9
```

We have now almost all the ingredients to write the *interpreter* for `exp`, namely a function `sem : exp → env → eval`, where

```
# type env = (ide*eval) list;;
```

Intermezzo

The run time support is the collections of data and functions that allow the execution of the code written in the programming language (and in our case interpreted). Thus the environment, implemented as we have suggested, is a part of the run time support (we will point out later that there are alternative implementations of the environment).

Recall that an expression returns an `eval` (integer and boolean of our language) whereas the machine code of our abstract machine (Ocaml) gives `int` and `bool`. We have to implement part of the rts to cope with the values of our language.

For instance, if we want to sum two integers (`Int 3` and `Int 4`) we have to sum `3` and `4` to obtain `Int 7`. Clearly we can sum numbers, so we have to be sure that the two `eval` we have to *combine* represent integers:

Let us start to implement a *type-checker* (we can do at top level, but in this way it can be more readable):

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

We can use this to implement operations at machine level:

```
let plus (x,y) = if typecheck("int",x) & typecheck("int",y)
      then
        (match (x,y) with
        | (Int(u), Int(w)) -> Int(u+w))
      else failwith ("type error")
```

Exercise. Complete by providing the low level implementation for the various operations on data types.

A solution to the exercise:

```
(* Operations on Eval *)

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

    let minus x =
      if typecheck("int",x)
      then
        (match x with
        | Int(y) -> Int(-y) )
      else
        failwith ("type error")

    let iszero x =
      if typecheck("int",x)
      then
        (match x with
        | Int(y) -> Bool(y=0) )
      else
        failwith ("type error")

    let equ (x,y) =
      if typecheck("int",x) & typecheck("int",y)
      then
        (match (x,y) with
        | (Int(u), Int(w)) -> Bool(u = w))
      else failwith ("type error")

    let plus (x,y) =
      if typecheck("int",x) & typecheck("int",y)
      then
        (match (x,y) with
        | (Int(u), Int(w)) -> Int(u+w))
      else failwith ("type error")

    let diff (x,y) =
      if typecheck("int",x) & typecheck("int",y)
      then
        (match (x,y) with
        | (Int(u), Int(w)) -> Int(u-w))
      else failwith ("type error")

    let mult (x,y) =
      if typecheck("int",x) & typecheck("int",y)
      then
        (match (x,y) with
        | (Int(u), Int(w)) -> Int(u*w))
      else failwith ("type error")

    let et (x,y) =
      if typecheck("bool",x) & typecheck("bool",y)
      then
        (match (x,y) with
        | (Bool(u), Bool(w)) -> Bool(u & w))
      else failwith ("type error")

    let vel (x,y) =
      if typecheck("bool",x) & typecheck("bool",y)
      then
        (match (x,y) with
        | (Bool(u), Bool(w)) -> Bool(u or w))
      else failwith ("type error")

    let non x =
      if typecheck("bool",x)
      then
        (match x with
        | Bool(y) -> Bool(not y) )
      else failwith ("type error");;
```

Interpreter

As we said, our target is to write an interpreter for `exp`.

Complete the following:

```
let rec sem (e:exp) (r:env) = match e with
      | Eint(n) -> Int(n)
      | Ebool(b) -> Bool(b)
      | ...
```

Exercise. Complete the implementation of the interpreter.

The **solution** of the exercise, as we have seen in class, can be the following:

```
(*SEMANTIC EVALUATION FUNCTIONS  *)
let rec sem (e:exp) (r:env) =
      match e with
      | Eint(n) -> Int(n)
      | Ebool(b) -> Bool(b)
      | Den(i) -> (applyenv r i)
      | Iszero(a) -> iszero((sem a r) )
      | Eq(a,b) -> equ((sem a r) ,(sem b r) )
      | Prod(a,b) ->  mult ( (sem a r), (sem b r))
      | Sum(a,b) ->  plus ( (sem a r), (sem b r))
      | Diff(a,b)  ->  diff ( (sem a r), (sem b r))
      | Minus(a) ->  minus( (sem a r))
      | And(a,b) ->  et ( (sem a r), (sem b r))
      | Or(a,b) ->  vel ( (sem a r), (sem b r))
      | Not(a) -> non( (sem a r))
      | Ifthenelse(a,b,c) ->
            let g = sem a r in
            if typecheck("bool",g) then
               (if g = Bool(true)
               then sem b r
               else sem c r)
            else failwith ("nonboolean guard")
```

Just one comment: any other different but correct implemetation would do.

Local declarations

Once you have completed the interpreter (and it runs), add to the language the *local declarations*. The complete syntax of the language is now

```
type ide = string
type exp = Eint of int
      | Ebool of bool
      | Den of ide
      | Prod of exp * exp
      | Sum of exp * exp
      | Diff of exp * exp
      | Eq of exp * exp
      | Minus of exp
      | Iszero of exp
      | Or of exp * exp
      | And of exp * exp
      | Not of exp
      | Ifthenelse of exp * exp * exp
      | Let of ide * exp * exp
```

The `Let(“x”,e1,e2)` construct binds to the identifier “x” the evaluation of the expression `e1` and the identifier may appear in the expression `e2`.

Exercise. Add to the implementation of the interpreter the interpretation of the let

Solution of the exercise: just add to the interpreter the following line: `Let(x,a,b) → sem b (bind x (sem a r) r)`

Adding other data types

The language supports integer and boolean. First let us identify what you have to change in order to add characters to the language, and second let us modify accordingly what has to be modified.

First of all, as we have added a new data type, we have to decide whether or not this type would be a value that the evaluation of an expression can give as result, and it is trivial to decide *positively*. The consequence is that we have to add this to out `eval`:

```
type eval = None
          | Int of int
          | Bool of bool
          | Char of char
```

Once we have done this, we can introduce the syntax of the language (on characters we can define an equality and an operation that converts a charcter in an integer (its coding). We may use the ocaml function `int_of_char : char → int`:

```
type ide = string
type exp = Eint of int
      | Ebool of bool
      | Echar of char
      | Den of ide
      | Prod of exp * exp
      | Sum of exp * exp
      | Diff of exp * exp
      | Eq of exp * exp
      | Eqchar of exp * exp
      | Minus of exp
      | Iszero of exp
      | Or of exp * exp
      | And of exp * exp
      | Not of exp
      | Casttoint of exp
      | Ifthenelse of exp * exp * exp
      | Let of ide * exp * exp
```

where we have added the possibility of having constants of the type `char`, and equality on character and the required cast operator.

We then modify the function checking whether the evaluated expression is of the proper type:

```
   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
          | Char(u) -> true
          | _ -> false)
      | _ -> failwith ("not a valid type")
```

and the implementation of the `Eqchar` and `Casttoint`:

```
let equchar(x,y) = if typecheck("char",x) & typecheck("char",y)
      then
        (match (x,y) with
        | (Char(u), Char(w)) -> Bool(u = w))
      else failwith ("type error")

let casttoint(x) = if typecheck("char",x)
      then
        (match c with
        | Char(u) -> Int(int_of_char u))
      else failwith ("type error")
```

Now the modification of the interpreter is trivial, as we have to add just some lines:

```
let rec sem (e:exp) (r:env) =
      match e with
      | Eint(n) -> Int(n)
      | Ebool(b) -> Bool(b)
      | Echar(c) -> Char c
      | Den(i) -> applyenv(r,i)
      | Iszero(a) -> iszero((sem a r) )
      | Eq(a,b) -> equ((sem a r) ,(sem b r) )
      | Eqchar(a,b) -> equchar((sem a r) ,(sem b r) )
      | Prod(a,b) ->  mult ( (sem a r), (sem b r))
      | Sum(a,b) ->  plus ( (sem a r), (sem b r))
      | Diff(a,b)  ->  diff ( (sem a r), (sem b r))
      | Minus(a) ->  minus( (sem a r))
      | And(a,b) ->  et ( (sem a r), (sem b r))
      | Or(a,b) ->  vel ( (sem a r), (sem b r))
      | Not(a) -> non( (sem a r))
      | Casttoint(a) -> casttoint (sem a r)
      | Ifthenelse(a,b,c) ->
            let g = sem a r in
            if typecheck("bool",g) then
               (if g = Bool(true)
               then sem b r
               else sem c r)
            else failwith ("nonboolean guard")
      | Let(x,a,b) -> sem b (bind x (sem a r) r)
```

A different implementation of the environment

Ocaml allows to define function, so why bother with lists? Here is an alternative implementation. It has the operation `emptyenv` that create the partial function always undefined.

```
# type 't env = ide -> 't;;

# let emptyenv x = function (y:ide) -> x;;
val emptyenv : 'a -> 'b -> 'a = <fun>

# let applyenv x y = x y;;
val applyenv : ('a -> 'b) -> 'a -> 'b = <fun>

# let bind l e r =
      function lu -> if lu = l then e else applyenv r lu;;
val bind :  'a -> 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
```

and we can write

```
# let rho0 = emptyenv None;;
val rho0 : '_a -> eval = <fun>

# let rho1 = bind "x" (Int 3) rho0;;
val rho1 : string -> eval = <fun>

# applyenv rho1 "z";;
- : eval = None

# applyenv rho1 "x";;
- : eval = Int 3
```

Both implementations can be used, provided we force the type of variables in ocaml:

```
# type env = ide -> eval;;

# let emptyenv (x:eval) = function (y:ide) -> x;;
val emptyenv : eval -> ide -> eval = <fun>

# let applyenv (x:env) (y:ide) = x y;;
val applyenv : env -> ide -> eval = <fun>

# let bind (l:ide) (e:eval) (r:env) =
      ((function lu -> if lu = l then e else applyenv r lu):env);;
val bind :  ide -> eval -> env -> env = <fun>
```

Exercise. Modify accordingly the other implementation of the environment.

Functions, static and dynamic scope

We want extend the language with functions: like in Ocaml we add functions and the recursive functions. Clearly functions should be used and can be the result of an evaluation of an expression, hence functions should belong to the type `eval`. Here is possibility:

```
type eval =     None
     | Int of int
     | Bool of bool
     | Funval of efun
and efun = exp * eval env
```

A function has a name (which will be put in the environment) and, as we are dealing with statically scoped language, it is a *closure*, i.e., a pair `exp` and `eval env`, which is the environment in which the function has to be evaluated.

A function receive a list of arguments that should match the formal parameter in the definition of the function. Here is the implementation a new operation on the environment that can ease the actual write of the interpreter:

```
#   exception WrongBindlist;;
exception WrongBindlist

# let rec bindlist (r,il,el) = match (il,el) with
        | ([],[]) -> r
        | i::il1, e::el1 -> bindlist ((bind  i e r), il1, el1)
        | _ -> raise WrongBindlist;;
val bindlist : ('a -> 'b) * 'a list * 'b list -> 'a -> 'b = <fun>
```

The syntax of the language is

```
type exp = Eint of int
      | Ebool of bool
      | Den of ide
      | Prod of exp * exp
      | Sum of exp * exp
      | Diff of exp * exp
      | Mod of exp * exp
      | Div of exp * exp
      | Less of exp * exp
      | Eq of exp * exp
      | Minus of exp
      | Iszero of exp
      | Or of exp * exp
      | And of exp * exp
      | Not of exp
      | Ifthenelse of exp * exp * exp
      | Let of ide * exp * exp
      | Fun of ide list * exp
      | Appl of exp * exp list
```

the function `bindlist` receives two lists: one of identifiers anda the other of eval values, and it inserts all of them in the environment. In case they do not match in length, an exception is raised.

Exercise Write the interpreter keeping in mind that you have to indicate in some way that you have functions

Solution of the exercise, as we have seen it during the lecture.

```
let rec sem (e:exp) (r:eval env) =
      match e with
      | Eint(n) -> Int(n)
      | Ebool(b) -> Bool(b)
      | Den(i) -> applyenv r i
      | Iszero(a) -> iszero((sem a r) )
      | Eq(a,b) -> equ((sem a r) ,(sem b r) )
      | Prod(a,b) ->  mult ( (sem a r), (sem b r))
      | Sum(a,b) ->  plus ( (sem a r), (sem b r))
      | Diff(a,b)  ->  diff ( (sem a r), (sem b r))
      | Minus(a) ->  minus( (sem a r))
      | And(a,b) ->  et ( (sem a r), (sem b r))
      | Or(a,b) ->  vel ( (sem a r), (sem b r))
      | Not(a) -> non( (sem a r))
      | Ifthenelse(a,b,c) ->
            let g = sem a r in
            if typecheck("bool",g) then
               (if g = Bool(true)
              then sem b r
              else sem c r)
           else failwith ("nonboolean guard")
      | Let(i,e1,e2) -> sem e2 (bind i (sem e1 r) r)
      | Fun(i,a) ->  makefun(Fun(i,a), r)
      | Appl(a,b) -> applyfun(sem a r, semlist b r)

and semlist el r = match el with
          | [] -> []
          | e::el1 -> (sem e r) :: (semlist el1 r)

and makefun ((a:exp),(x:eval env)) =
      (match a with
      | Fun(ii,aa) -> Funval(a,x)
      | _ -> failwith ("Non-functional object"))

and applyfun ((ev1:eval),(ev2:eval list)) =
      ( match ev1 with
      | Funval(Fun(ii,aa),r) -> sem aa (bindlist(r,ii,ev2))
      | _ -> failwith ("attempt to apply a non-functional object"));;
```

Here are some example of how to use it:

```
# let fesempio = Fun(["x"],Fun(["y"],Sum(Den("x"),Den("y"))));;
val fesempio : exp = Fun (["x"], Fun (["y"], Sum (Den "x", Den "y")))
```

This is the definition of a function that takes an argument `x` and return a function that will sum the argument to the argument of the returned function. To use it we *apply* it to some arguments:

```
# let aexp1 = Appl(fesempio,[Eint 3]);;
val aexp1 : exp =
  Appl (Fun (["x"], Fun (["y"], Sum (Den "x", Den "y"))), [Eint 3])
# let ris = sem aexp1 rho;;
val ris : eval = Funval (Fun (["y"], Sum (Den "x", Den "y")), <fun>)
```

Observe that `ris` is a function in our language.

```
# let aexp = Appl(Appl(fesempio,[Eint 3]),[Eint 5]);;
val aexp : exp =
  Appl (Appl (Fun (["x"], Fun (["y"], Sum (Den "x", Den "y"))), [Eint 3]),
   [Eint 5])

# let execution = sem aexp rho;;
val execution : eval = Int 8
```

Exercise The language with the `eval` we have introduced is obviously statically scoped. Keeping the same syntax for the language, describe (and realize) what has to be done to have the language dynamically scoped

Solution to the exercise on dynamic scope: as in dynamic scope the function has to be evaluated in the environment at the moment of its use, we do not need to have as functions result a closure as before, but we need simply an expression. Clearly, as we have not yet implemented any binding policy (except the one forced by the dynamic/static rules), the previous example with the function returning a *function* fails (why?)

Here we list the relevant changes:

```
type eval =     None
     | Int of int
     | Bool of bool
     | Funval of efun
and efun = exp
```

and

```
let rec sem (e:exp) (r:eval env) =
      match e with
      ...
      | Fun(i,a) ->  makefun(Fun(i,a))
      | Appl(a,b) -> applyfun(sem a r, semlist b r, r)

and semlist ...

and makefun (a:exp) =
      (match a with
      | Fun(ii,aa) -> Funval(a)
      | _ -> failwith ("Non-functional object"))

and applyfun ((ev1:eval),(ev2:eval list),(r:eval env)) =
      ( match ev1 with
      | Funval(Fun(ii,aa)) -> sem aa (bindlist(r,ii,ev2))
      | _ -> failwith ("attempt to apply a non-functional object"))
```

We have changed the `makefun` that now returns an expression and we have added to the `applyfun` the environment where the function has to be evaluated.

The usual test: the *factorial* function (observe that as now we use the *name* of the function to apply to it an argument, we have to use also the `Let` construct:

```
# let fact = Let("fatt",
                Fun(["x"],
                Ifthenelse(Eq(Den("x"),Eint 1), Eint 1,
                   Prod(Appl(Den("fatt"),[Diff(Den("x"),Eint 1)]),Den("x")))),
                Appl(Den("fatt"),[Eint 5]));;
val fact : exp =
  Let ("fatt",
   Fun (["x"],
    Ifthenelse (Eq (Den "x", Eint 1), Eint 1,
     Prod (Appl (Den "fatt", [Diff (Den "x", Eint 1)]), Den "x"))),
   Appl (Den "fatt", [Eint 5]))

# sem fact (emptyenv None);;
- : eval = Int 120
```

Exercise Add the possibility of having recursion in the language statically scoped. In this case to the function we cannot simply associate the environment it receives when the function is declared: if you try to execute the `fatt =` in the solution to the previous exercise in the interpreter for the statically scoped language we have developed, it will fail as the identifier "`fatt`" is not in the environment, hence we have to add it. However, as we have to *calculate* the proper one, we do it only when needed, i.e. when it is really necessary and we indicate it with `Rec`

Solution:

The syntax changes in this way, conveying the idea that a recursive function must have the keyword **rec** somewhere (and we expect that `exp` is a function where the `ide` appears):

```
type exp =
       ....
      | Rec of ide * exp;;
```

and the interpreter is modified as follows:

```
let rec sem (e:exp) (r:eval env) =
      match e with
      ...
      | Rec(f,e) -> makefunrec (f,e,r)


and semlist el r = ...


and makefun ((a:exp),(x:eval env)) = ...

and applyfun ((ev1:eval),(ev2:eval list)) = ...

and makefunrec (i, e1, (r:eval env)) =
    let calculateenv (rr: eval env)  =
       (bind i (makefun(e1,rr)) r) in
         let rec rfix =
          function x -> calculateenv rfix x in makefun(e1,rfix);;
```

The `makefunrec` calculate the environment `rfix` that have to be associated to the recursive function. It uses a function `calculateenv` that receives an environment `rr` and returns an environment where to the identifier of the recursive function is associated the function and the proper environment. What it really does is to calculate the smallest environment where this works.

Here the usal example of how it works:

```
# let fact = Rec("fatt",
      Fun(["x"],
       Ifthenelse(Eq(Den("x"),Eint 1),
          Eint 1,
          Prod(Appl(Den("fatt"),[Diff(Den("x"),Eint 1)]),Den("x")))));;
val fact : exp =
  Rec ("fatt",
   Fun (["x"],
    Ifthenelse (Eq (Den "x", Eint 1), Eint 1,
     Prod (Appl (Den "fatt", [Diff (Den "x", Eint 1)]), Den "x"))))
```

```
# let exec = Appl(fact,[Eint 5]);;
val exec : exp =
  Appl
   (Rec ("fatt",
     Fun (["x"],
      Ifthenelse (Eq (Den "x", Eint 1), Eint 1,
       Prod (Appl (Den "fatt", [Diff (Den "x", Eint 1)]), Den "x")))),
   [Eint 5])

# sem exec (emptyenv None);;
- : eval = Int 120
```