Interprete del Lambda-Calcolo

Sintassi

```
type id = string
type exp =
    Var of id
     | Lam of id * exp
     | App of exp * exp
```

```
type id = string
type exp = Var of id | Lam of id * exp | App of exp * exp
```

$$e ::= x \mid \lambda x.e \mid e\,e$$

Semantica

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e_1'}{e_1\,e_2 \to e_1'\,e_2} \qquad \frac{e_2 \to e_2'}{e_1\,e_2 \to e_1\,e_2'}$$

$$\frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

Implementazione Capture-Avoiding Substitution

$$
\begin{array}{rcl}
x\{x := e\} & \equiv & e \\
y\{x := e\} & \equiv & y \text{ se } y \neq x \\
(e_1 e_2)\{x := e\} & \equiv & (e_1\{x := e\})(e_2\{x := e\}) \\
(\lambda y.e_1)\{x := e\} & \equiv & \lambda y.(e_1\{x := e\}) \\
 & & \text{se } y \neq x \text{ e } y \notin FV(e) \\
(\lambda y.e_1)\{x := e\} & \equiv & \lambda z.((e_1\{y := z\})\{x := e\}) \\
 & & \text{se } y \neq x \text{ e } y \in FV(e) \text{ e } z \text{ fresca}
\end{array}
$$

Implementazione di `FV(e)`

$$FV(x) = x \qquad FV(\lambda x.e) = FV(e) \setminus \{x\}$$

$$FV(e_1\, e_2) = FV(e_1) \cup FV(e_2)$$

Funzione che calcola ricorsivamente l'insieme delle variabili libere dell'espressione e rappresentato come lista di identificatori:

```
(* computes the free ( non-bound ) variables in e *)
let rec fvs e =
    match e with
     Var x -> [x]
     | Lam (x,e) -> List.filter (fun y -> x <> y) (fvs e)
     | App (e1,e2) -> (fvs e1) @ (fvs e2)
;;
```

```
val fvs : exp -> id list = <fun>
```

esempi di utilizzo:

```
(* TESTS *)
fvs (Var "x") = ["x"];;
fvs (Lam ("x",Var "y")) = ["y"];;
```

```
fvs (Lam ("x",Var "x")) = [];;
fvs (App (Lam ("x", Var "z"), Var "y")) = ["z"; "y"];;
```

```
- : bool = true
- : bool = true
- : bool = true
- : bool = true
```

Generatore di identificatori “freschi”

```
(* generates a fresh variable *)
let newvar =
let x = ref 0 in
    fun () ->
     let c = !x in
     incr x;
     "v"^(string_of_int c)
```

```
val newvar : unit -> string = <fun>
```

Capture-avoiding substitution

```
(* substitution: subst e y m means "substitute occurrences of variable y wi
let rec subst e y m =
  match e with
  | Var x ->
    if y = x then m (* replace x with m *)
    else e (* variables don't match: leave x unchanged *)
  | App (e1,e2) -> App (subst e1 y m, subst e2 y m)
  | Lam (x,e) ->
    if y = x then (* don't substitute under the variable binder *)
      Lam(x,e)
    else if not (List.mem x (fvs m)) then (* no need to alpha convert *)
      Lam (x, subst e y m)
    else (* need to alpha convert *)
      let z = newvar() in (* assumed to be "fresh" *)
```

```
      let e' = subst e x (Var z) in (* replace x with z in e *)
      Lam (z,subst e' y m) (* substitute for y in the adjusted term, e' *)
```

```
val subst : exp -> id -> exp -> exp = <fun>
```

esempi di utilizzo:

```
(* TESTS *)
let m1 = (App (Var "x", Var "y"));; (* x y *)
let m2 = (App (Lam ("z",Var "z"), Var "w"));; (* ( lambda z . z ) w *)
let m3 = (App (Lam ("z",Var "x"), Var "w"));; (* ( lambda z . x ) w *)
let m4 = (App (App (Lam ("z",Var "z"), Lam ("x", Var "x")), Var "w"))
(* ( lambda z . z ) ( lambda x . x ) w *)
let m1_zforx = subst m1 "x" (Var "z");;
let m1_m2fory = subst m1 "y" m2
let m2_ughforz = subst m2 "z" (Var "ugh")
let m3_zforx = subst m3 "x" (Var "z")
let m1_m3fory = subst m1 "y" m3
```

```
val m1 : exp = App (Var "x", Var "y")
val m2 : exp = App (Lam ("z", Var "z"), Var "w")
val m3 : exp = App (Lam ("z", Var "x"), Var "w")
val m4 : exp = App (App (Lam ("z", Var "z"), Lam ("x", Var "x")), Var "w")
val m1_zforx : exp = App (Var "z", Var "y")
val m1_m2fory : exp = App (Var "x", App (Lam ("z", Var "z"), Var "w"))
val m2_ughforz : exp = App (Lam ("z", Var "z"), Var "w")
val m3_zforx : exp = App (Lam ("v2", Var "z"), Var "w")
[7]:val m1_m3fory : exp = App (Var "x", App (Lam ("z", Var "x"), Var "w"))
```

Implementazione della semantica in OCaml

$$(\lambda x.e_1)e_2 \to e_1\{x := e_2\}$$

$$\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{e_1\, e_2 \to e_1\, e_2'} \qquad \frac{e \to e'}{\lambda x.e \to \lambda x.e'}$$

```
(* beta reduction. *)
let rec reduce e =
  match e with
  | App (Lam (x, e1), e2) -> subst e1 x e2 (* direct beta rule *)
  | App (e1, e2) ->
    let e1' = reduce e1 in (* try to reduce a term in the lhs *)
    if e1' <> e1 then App(e1', e2)
    else App (e1, reduce e2) (* didn't work; try rhs *)
  | Lam (x, e) -> Lam (x, reduce e) (* reduce under the lambda ( !) *)
  | _ -> e (* no opportunity to reduce *)
```

```
val reduce : exp -> exp = <fun>
```

esempi di utilizzo:

```
(* TESTS *)
let m2red = reduce m2 ;; (* ( lambda z . z ) w *)
let m3red = reduce m3 ;; (* ( lambda z . x ) w *)
let m4red1 = reduce m4 ;; (* ( lambda z . z ) ( lambda x . x ) w *)
let m4red2 = reduce m4red1 ;; (* vedi sopra *)
let m13sred = reduce m1_m3fory ;; (* x ( ( lambda z . x ) w ) *)
```

```
val m2red : exp = Var "w"
val m3red : exp = Var "x"
val m4red1 : exp = App (Lam ("x", Var "x"), Var "w")
val m4red2 : exp = Var "w"
val m13sred : exp = App (Var "x", Var "x")
```