

# Lambda-calcul netipizat.

## Laborator 11

În acest laborator, vom crea un interpretor pentru lambda-calcul (prezentat în cursul precedent).

În primul rând, vom reprezenta identificatorii (variabilele) prin intermediul unui `String`. Pentru a marca acest lucru, vom crea un sinonim de tip, numit `Id`, pentru tipul predefinit `String`.

```
type Id = String
```

Lambda-termenii vor fi reprezentați ca valori de tip `Term`:

```
data Term = Var Id
          | App Term Term
          | Lambda Id Term deriving (Show, Eq)
```

**Exercițiul 0.1.** Scrieți o valoare de tip `Term` care reprezintă lambda-termenul  $\lambda x.\lambda y.x$ .

În continuare, vom scrie o funcție `subst` care implementează substituția, notată în curs cu  $t[x/t']$ . Funcția va primi trei parametri:

1. variabila `id` care trebuie înlocuită;
2. termenul `term` care o va înlocui;
3. termenul în care se face înlocuirea.

Astfel, pentru a calcula  $t[x/t']$ , vom apela `subst x t' t`.

**Exercițiul 0.2.** Completați funcția de mai jos (înlocuind ... cu codul dumneavoastră):

```
subst :: Id -> Term -> Term -> Term
subst id term (Var id') | id == id' = term
                        | True      = ...
subst id term (App term1 term2) = App ... (subst id term term2)
subst id term (Lambda id' term') | id == id' = ...
                                | True      = (Lambda id' (subst id term term'))
```

Asigurați-vă că funcția calculează corect rezultatul pe câteva exemple:

1.  $x[x/y] = y$ ;
2.  $x[y/z] = x$ ;
3.  $(x\ y)[y/z] = x\ z$ ;

4.  $(y\ x)[y/z] = z\ x;$
5.  $(\lambda x.(y\ x))[x/(\lambda z.z)] = \lambda x.(y\ x);$
6.  $(\lambda y.(y\ x))[x/(\lambda z.z)] = \lambda y.(y\ (\lambda z.z)).$

Substituția, așa cum este implementată mai sus, este de tip *non-capture-avoiding substitution*, deoarece:

1.  $\lambda x.y[y/x] = \lambda x.x$  (verificați).

În continuare, vom scrie o funcție care implementează o substituție *capture-avoiding*. Întâi, avem nevoie de câteva funcții ajutoare:

**Exercițiul 0.3.** Completați funcția `remove`, care elimină toate aparițiile unui element dat ca argument dintr-o listă:

```
remove :: Id -> [Id] -> [Id]
remove _ [] = []
remove id (hd:tl) | id == hd = remove id tl
                  | True      = ...
```

Testați funcția pe câteva exemple.

**Exercițiul 0.4.** Completați funcția `free`, care calculează toate variabilele libere ale unui lambda-term (rezultatul poate conține duplicate):

```
free :: Term -> [Id]
free (Var id) = [id]
free (App term1 term2) = ...
free (Lambda id term) = ...
```

**Exercițiul 0.5.** Completați funcția `vars`, care calculează toate variabilele unui lambda-term, indiferent dacă sunt libere sau nu (rezultatul poate conține duplicate):

```
vars :: Term -> [Id]
vars (Var id) = ...
vars (App term1 term2) = ...
vars (Lambda id term) = ...
```

**Exercițiul 0.6.** Completați funcția `fresh` (puteți folosi funcția ajutoare `fresh'`), care este folosită pentru calcularea unui identificator *proaspăt* (identificator ce nu apare deja în lista de identificatori dată ca argument):

```
fresh' :: [Id] -> Int -> Id
fresh' ids index = if ("n" ++ (show index)) `elem` ids then fresh' ids (index + 1)
                  else "n" ++ (show index)

fresh :: [Id] -> Id
fresh ids = ...
```

**Exercițiul 0.7.** Completați funcția `casubst`, care implementează o substituție de tip *capture-avoiding*:

```

casubst :: Id -> Term -> Term -> [Id] -> Term
casubst id term (Var id') _ | id == id' = ...
                                | True      = (Var id')
casubst id term (App term1 term2) avoid = App ...
casubst id term (Lambda id' term') avoid | id == id' = ...
                                                | id' 'elem' (free term) =
    let id'' = fresh avoid in
        ...
                                | True      = Lambda id' ...

```

Testați funcția `casubst` pe toate exemplele de la `subst`, plus exemplul:

$$1. \lambda x.y[y/x] = \lambda n_0.x.$$

**Exercițiul 0.8.** Completați funcția `reduce1`, care aplică o beta-reducere în termenul dat ca parametru și în acest caz întoarce termenul rezultat (protejat cu constructorul `Just`), sau întoarce `Nothing`, dacă nu există nicio beta-reducere care se poate aplica.

```

reduce1' :: Term -> [Id] -> Maybe Term
reduce1' (Var id') _ = Nothing
reduce1' (App (Lambda id term) term') avoid =
    Just (casubst id term' term avoid) -- beta-reducerea propriu-zisa
reduce1' (App term1 term2) avoid = case reduce1' term1 avoid of
    Nothing -> case reduce1' term2 avoid of
        Nothing -> ...
        Just term2' -> ...
    Just term1' -> Just (App term1' term2)
reduce1' (Lambda id term) avoid = case reduce1' term avoid of
    Nothing -> ...
    Just term' -> ...

reduce1 :: Term -> Maybe Term
reduce1 t = reduce1' t (vars t)

```

Testați funcția `reduce1` pe următoarele exemple:

```

x = Var "x"
y = Var "y"
z = Var "z"

term1 = Lambda "x" x
term2 = App term1 term1
term3 = Lambda "y" (Lambda "x" term2)
term4 = App term3 term1

ex1 = reduce1 term1 -- Nothing
ex2 = reduce1 term2 -- Just (\x.x)
ex3 = reduce1 term3 -- Just \y.\x.(\x.x)
ex4 = reduce1 term4 -- Care este rezultatul?

```

**Exercițiul 0.9.** Completați următoarea funcție recursivă, care calculează o formă normală a lambda-termenului dat ca parametru (aplică toate beta-reducerile posibile, cât timp există):

```
reduce :: Term -> Term
reduce term = case reduce1 term of
  Nothing -> ...
  Just term' -> ...
```

Exemple: `reduce` aplicat pe  $(\lambda x.x)((\lambda z.z)y)$  ar trebui să întoarcă  $y$ . Care sunt beta-reducerile efectuate de `reduce`?

**Exercițiul 0.10.** Dați exemplu de un lambda termen pentru care `reduce` intră în buclă infinită. Completați următoarea funcție, care aplică cel mult  $n$  pași de beta-reducere:

```
reduceFor :: Int -> Term -> Term
reduceFor 0 term = term
reduceFor n term = case reduce1 term of
  Nothing -> ...
  Just term' -> ...
```

**Exercițiul 0.11.** Scrieți ca valori de tip `Term` lambda-termenii `TRUE`, `FALSE`, `AND`, `OR`, `NOT`, `ITE`:

```
tTRUE = Lambda "x" (Lambda "y" x) -- folosim x = Var "x" declarat mai sus
tFALSE = ...
tAND = ...
tOR = ...
tNOT = ...
```

Calculați `reduce (App (App tAND tTRUE) tFALSE)`. Calculați formele normale ale altor asemenea lambda termeni pentru a testa faptul că `tAND`, `tOR` etc funcționează conforma așteptărilor.

**Exercițiul 0.12.** Folosiți codificarea numerelor naturale ca lambda-termeni discutată la curs, implementați funcțiile `PLUS` și `MULT`, și testați-le pe câteva exemple.

**Exercițiul 0.13.** Scrieți o funcție care testează dacă doi termeni sunt  $\alpha$ -echivalenți.

**Exercițiul 0.14.** Folosiți `QuickCheck` pentru a testa că substituția care evită capturarea transformă orice doi termeni  $\alpha$ -echivalenți în doi termeni  $\alpha$ -echivalenți.