Curs 12

2021-2022

Fundamentele limbajelor de programare

Cuprins

Monade în Haskell (recap)

Funcţii îmbogăţite şi efecte

- Funcţie simplă: $x \mapsto y$

  ştiind x, obţinem direct y

- Funcţie îmbogăţită: $x \mapsto$

  ştiind x, putem să extragem y şi producem un efect

https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/

https://bartoszmilewski.com/2016/11/30/monads-and-effects/

Funcţii îmbogăţite şi efecte

Funcţie îmbogăţită: $x \mapsto$ 

Exemplu

- Folosind tipul Maybe a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x)
```

Funcţii îmbogăţite şi efecte

Funcţie îmbogăţită: $x \mapsto$ 

Exemplu

- Folosind un tip care are ca efect un mesaj

```haskell
newtype Writer log a = Writer {runWriter :: (a, log)}

f :: Int -> Writer String Int
f x = if x < 0 then (Writer (-x, "negativ"))
               else (Writer (x, "pozitiv"))
```

Clasa de tipuri Monad

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a - > m b -> m b
    return :: a -> m a

    ma >> mb = ma >>= \_ -> mb
```

- m a este tipul computațiilor care produc rezultate de tip a (și au efecte laterale)
- a −> m b este tipul continuărilor / a funcțiilor cu efecte laterale
- >>= este operația de „secvențiere” a computațiilor

În Haskell, monada este o clasă de tipuri!

Notația do pentru monade

Notaţia cu operatori	Notaţia do
e >>= \x −> rest	x <− e rest
e >>= _ −> rest	e rest
e >> rest	e rest

De exemplu

```
e1    >>= \x1 ->
e2    >> e3
```

devine

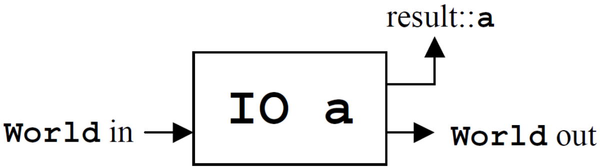
```
do
  x1 <- e1
  e2
  e3
```

Exemple de efecte laterale

I/O	Monada IO
Logging	Monada Writer
Stare	Monada State
Excepții	Monada Either
Parțialitate	Monada Maybe
Nedeterminism	Monada [] (listă)
Memorie read-only	Monada Reader

Monada IO

```haskell
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Monada IO

- IO () corespunde comenzilor care nu produc rezultate

```
putChar   :: Char -> IO ()
putStr    :: String -> IO ()
putStrLn  :: String -> IO ()
```

- IO a corespunde comenzilor care produc rezultate de tip a

```
getChar   :: IO Char
getLine   :: IO String
```

Monada IO

```haskell
sayHello :: String -> IO ()
sayHello name = putStrLn ("Hi " ++ name ++ "!")

main :: IO ()
main = do
  putStr "Please input your name: "
  name <- getLine
  sayHello name
```

Interpretoare monadice

Mini-Haskell - Sintaxă abstractă

```haskell
type Name = String

data Term = Var Name
          | Con Integer
          | Term :+: Term
          | Lam Name Term
          | App Term Term
```

Mini-Haskell - Valori și medii de evaluare

```
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

instance Show Value where
 show (Num x) = show x
 show (Fun _) = "<function>"
 show (Wrong) = "<wrong>"
```

Observații

- Vom interpreta termenii în valori ‘M Value‘, unde ‘M‘ este o monadă; variind monada, se obțin comportamente diferite;
- ‘Wrong‘ reprezintă o eroare, de exemplu adunarea unor valori care nu sunt numere sau aplicarea unui termen care nu e funcție.

Evaluare - variabile și valori

```
type Environment = [(Name, Value)]
```

Interpretarea termenilor în monada M

```
interp :: Term -> Environment -> M Value
interp (Var x) env    = lookupM x env
interp (Con i) _      = return $ Num i
interp (Lam x e) env = return $
                       Fun $ \ v -> interp e ((x,v):env)

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v   -> return v
  Nothing -> return Wrong
```

Evaluare - adunare

```haskell
interp (t1 :+: t2) env = do
  v1 <- interp t1 env
  v2 <- interp t2 env
  add v1 v2
```

Interpretarea adunării în monada M

```haskell
add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add  _ _            = return Wrong
```

Evaluare - aplicarea funcțiilor

```haskell
interp (App t1 t2) env = do
  f <- interp t1 env
  v <- interp t2 env
  apply f v
```

Interpretarea aplicării funcțiilor în monada M

```haskell
apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = return Wrong

-- k :: Value -> M Value
```

Testarea interpretorului

```haskell
test :: Term -> String
test t = showM $ interp t []
```

unde

```haskell
showM :: Show a => M a -> String
```

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Testarea interpretorului

```
test :: Term -> String
test t = showM $ interp t []
```

unde

```
showM :: Show a => M a -> String
```

este o funcție definită special pentru fiecare tip de efecte laterale dorit.

Exemplu de program

```
pgm :: Term
pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

test pgm  -- apelul pentru testare
```

Interpretor monadic

```haskell
data Value = Num Integer
           | Fun (Value -> M Value)
           | Wrong

interp :: Term -> Environment -> M Value
```

În continuare vom înlocui monada M cu:

- Identity
- Maybe
- Either String
- Writer

Interpretare în monada ‘Identity‘

Monada ‘Identity‘ este "efectul identitate".

```
newtype Identity a = Identity { runIdentity :: a }

instance Monad Identity where
    return a   = Identity a
    ma >>= k   = k (runIndentity ma)

instance Applicative Identity where
  pure = return
  mf <*> ma = do
    f <- mf
    a <- ma
    return (f a)

instance Functor Identity where
  fmap f ma = pure f <*> ma
```

Interpretare în monada ‘Identity‘

Pentru a particulariza interpretorul definim

```haskell
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity
```

Interpretare în monada ‘Identity‘

Pentru a particulariza interpretorul definim

```haskell
type M a = Identity a
```

```haskell
showM :: Show a => M a -> String
showM = show . runIdentity
```

Obținem interpretorul standard, asemănător celui discutat pentru limbajul Mini-Haskell.

Interpretare folosind monada ‘Identity‘

```haskell
type M a = Identity a

showM :: Show a => M a -> String
showM = show . runIdentity

pgm = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Con 10) :+:  (Con 11))

*Var0> test pgm
"42"
```

Interpretare în monada ‘Maybe‘ (opțiune)

```haskell
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just a  >>= k   = k a
  Nothing >>= _   = Nothing
```

Plus instante pentru Applicative si Functor.

Interpretare în monada ‘Maybe‘ (opțiune)

```haskell
data Maybe a = Nothing | Just a

instance Monad Maybe where
  return = Just
  Just a  >>= k    = k a
  Nothing >>= _    = Nothing
```

Plus instante pentru Applicative si Functor.

Putem renunța la valoarea ‘Wrong‘, folosind monada ‘Maybe‘

```haskell
type M a = Maybe a

showM :: Show a => M a -> String
showM (Just a) = show a
showM Nothing  = "<wrong>"
```

Interpretare în monada ‘Maybe‘

Putem acum înlocui rezultatele ‘Wrong‘ cu ‘Nothing‘

```
type M a = Maybe a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v   -> return v
  Nothing -> Nothing

add :: Value -> Value -> M Value
add (Num i) (Num j) = return (Num $ i + j)
add _ _             = Nothing

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply _ _       = Nothing
```

Interpretare în monada ‘Either String‘

```haskell
data Either a b = Left a | Right b

instance Monad (Either err) where
  return = Right
  Right a >>= k   = k a
  err     >>= _   = err
```

Plus instante pentru Applicative si Functor.

Interpretare în monada ‘Either String‘

```
data Either a b = Left a | Right b

instance Monad (Either err) where
  return = Right
  Right a >>= k   = k a
  err     >>= _   = err
```

Plus instante pentru Applicative si Functor.

Putem nuanța erorile folosind monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a
```

Interpretare în monada ‘Either String‘

Putem acum înlocui rezultatele ‘Wrong‘ cu valori ‘Left‘

```
type M a = Either String a

lookupM :: Name -> Environment -> M Value
lookupM x env = case lookup x env of
  Just v   -> return v
  Nothing -> Left ("unbound variable " ++ x)

add :: Value -> Value -> M Value
add (Num i) (Num j) = return $ Num $ i + j
add v1 v2           = Left $
  "Expected numbers: " ++ show v1 ++ ", " ++ show v2

apply :: Value -> Value -> M Value
apply (Fun k) v = k v
apply v _       = Left $
  "Expected function: " ++ show v
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
        (Lam "x" ((Var "x") :+: (Var "x")))
        ((Con 10) :+:  (Con 11))

*Var2> test pgm
"Success: 42"
```

Interpretare în monada ‘Either String‘

```
type M a = Either String a

showM :: Show a => M a -> String
showM (Left s)  = "Error: " ++ s
showM (Right a) = "Success: " ++ show a

pgm = App
        (Lam "x" ((Var "x") :+: (Var "x")))
        ((Con 10) :+:  (Con 11))

*Var2> test pgm
"Success: 42"

pgmE = App (Var "x") ((Con 10) :+:  (Con 11))

*Var2> test pgmE
"Error: unbound variable x"
```

Monada ‘Writer‘

Este folosită pentru a acumula (logging) informație produsă în timpul execuției.

```
newtype Writer log a = Writer { runWriter :: (a, log) }

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
              in Writer (b, log1 `mappend` log2)
```

Plus instante pentru Applicative si Functor.

Monada ‘Writer‘

Este folosită pentru a acumula (logging) informație produsă în timpul execuției.

```haskell
newtype Writer log a = Writer { runWriter :: (a, log) }

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (a, log1) = runWriter ma
                 (b, log2) = runWriter (k a)
              in Writer (b, log1 `mappend` log2)
```

Plus instante pentru Applicative si Functor.

Funcție ajutătoare

```haskell
tell :: log -> Writer log ()
tell log = Writer ((), log)  -- produce mesajul
```

Interpretare în monada ‘Writer‘

Adăugarea unei instrucțiuni de afișare
data Term = ... | Out Term

```haskell
type M a = Writer String a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ "  Value: " ++ show a
  where (a, w) = runWriter ma


interp (Out t) env = do
  v <- interp t env
  tell (show v ++ "; ")
  return v
```

- Out t se evaluează la valoarea lui t, cu efectul lateral de a adăuga valoarea la șirul de ieșire.

Interpretare în monada ‘Writer‘

```
data Term = ... | Out Term

type M a = Writer String a

showM :: Show a => M a -> String
showM ma = "Output: " ++ w ++ "  Value: " ++ show a
  where (a, w) = runWriter ma

pgmW = App
          (Lam "x" ((Var "x") :+: (Var "x")))
          ((Out (Con 10)) :+: (Out (Con 11)))

> test pgm
"Output: 10; 11; Value: 42"
```

Introducere in λ-calcul

λ-calcul

- În 1929-1932 Church a propus λ-calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată in λ-calcul.
- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi Mașina Turing. În 1936 și el a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing. De asemenea, a arătat echivalența celor două modele de calcul. Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele, conducând la ceea ce numim astăzi "Teza Church-Turing".

Referinţe

- Benjamin C. Pierce, Types and Programming Languages, The MIT Press 2002
- J.R. Hindley, J.P. Seldin, Lambda-Calculus and Combinators, an Introduction, Cambridge University Press, 2008
- R. Nederpelt, H. Geuvers, Type Theory and Formal Proof, an Introduction, Cambridge University Press 2014

λ-calcul: sintaxa

Lambda Calcul - sintaxă

$$
\begin{array}{lll}
t = & x & \text{(variabilă)} \\
 & \mid \lambda x.\, t & \text{(abstractizare)} \\
 & \mid t\ t & \text{(aplicare)}
\end{array}
$$

λ-calcul: sintaxa

Lambda Calcul - sintaxă

$$
\begin{array}{lll}
t = & x & \text{(variabilă)} \\
 & \mid \lambda x.\, t & \text{(abstractizare)} \\
 & \mid t \ \ t & \text{(aplicare)}
\end{array}
$$

λ-termeni

Fie $Var = \{x, y, z, \ldots\}$ o mulţime infinită de variabile.
Mulţimea λ-termenilor ΛT este definită inductiv astfel:

[Variabilă] $Var \subseteq \Lambda T$

[Aplicare] dacă t_1, $t_2 \in \Lambda T$ atunci $(t_1 t_2) \in \Lambda T$

[Abstractizare] dacă $x \in Var$ ş i $t \in \Lambda T$ atunci $(\lambda x.t) \in \Lambda T$

Lambda termeni

λ-termeni: exemple

- x, y, z

Lambda termeni

λ-termeni: exemple

- x, y, z
- (xy), (yx), $(x(yx))$

Lambda termeni

λ-termeni: exemple

- x, y, z
- (xy), (yx), $(x(yx))$
- $(\lambda x.x)$, $(\lambda x.(xy))$, $(\lambda z.(xy))$, $(\lambda x.(\lambda z.(xy)))$

Lambda termeni

λ-termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

Lambda termeni

λ-termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

Convenţii:

- se elimină parantezele exterioare
- aplicarea este asociativă la stânga: $t_1 t_2 t_3$ este $(t_1 t_2) t_3$
- corpul abstractizării este extins la dreapta:
  $\lambda x.t_1 t_2$ este $\lambda x.(t_1 t_2)$ (nu $(\lambda x.t_1)t_2$)
- scriem $\lambda xyz.t$ în loc de $\lambda x.\lambda y.\lambda z.t$

Lambda termeni. Funcții anonime

λ-termeni: exemple

- x, y, z
- $(xy), (yx), (x(yx))$
- $(\lambda x.x), (\lambda x.(xy)), (\lambda z.(xy)), (\lambda x.(\lambda z.(xy)))$
- $((\lambda x.x)y), ((\lambda x.(xz))y), ((\lambda x.x)(\lambda y.y))$

Funcţii anonime în Haskell

În Haskell, \ e folosit în locul simbolului λ și -> în locul punctului.

$\lambda x.x * x$ este `\x -> x * x`

$\lambda x.x > 0$ este `\x -> x > 0`

Variabile libere şi legate

Apariţii libere şi legate

Pentru un termen $\lambda x.t$ spunem că:

- apariţiile variabilei x în t sunt legate (*bound*)
- λx este legătura (*binder*), iar t este domeniul (*scope*) legării
- o apariţie a unei variabile este liberă (*free*) dacă apare într-o poziţie în care nu e legată.

Variabile libere şi legate

Apariţii libere şi legate

Pentru un termen $\lambda x.t$ spunem că:

- apariţiile variabilei x în t sunt legate (*bound*)
- λx este legătura (*binder*), iar t este domeniul (*scope*) legării
- o apariţie a unei variabile este liberă (*free*) dacă apare într-o poziţie în care nu e legată.

Un termen fără variable libere se numeşte închis (*closed*).

Variabile libere şi legate

Apariţii libere şi legate

Pentru un termen $\lambda x.t$ spunem că:

- apariţiile variabilei x în t sunt legate (*bound*)
- λx este legătura (*binder*), iar t este domeniul (*scope*) legării
- o apariţie a unei variabile este liberă (*free*) dacă apare într-o poziţie în care nu e legată.

Un termen fără variable libere se numeşte închis (*closed*).

Exemplu:

- $\lambda x.x$ este un termen închis
- $\lambda x.xy$ nu este termen închis, x este legată, y este liberă
- în termenul $x(\lambda x.xy)$ prima apariţie a lui x este liberă, a doua este legată.

Variabile libere

Mulţimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulţimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Variabile libere

Mulţimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulţimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Exemplu:

$$
\begin{aligned}
FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\
& (FV(x) \cup FV(y)) \setminus \{x\} \\
& (\{x\} \cup \{y\}) \setminus \{x\} \\
& \{y\}
\end{aligned}
$$

Variabile libere

Mulţimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulţimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Exemplu:

$$\begin{aligned} FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\ & (FV(x) \cup FV(y)) \setminus \{x\} \\ & (\{x\} \cup \{y\}) \setminus \{x\} \\ & \{y\} \end{aligned}$$

$FV(x\lambda x.xy) =$

Variabile libere

Mulţimea variabilelor libere $FV(t)$

Pentru un λ-termen t mulţimea variabilelor libere este definită astfel:

[Variabilă] $FV(x) = \{x\}$

[Aplicare] $FV(t_1 t_2) = FV(t_1) \cup FV(t_2)$

[Abstractizare] $FV(\lambda x.t) = FV(t) \setminus \{x\}$

Exemplu:

$$
\begin{aligned}
FV(\lambda x.xy) = \quad & FV(xy) \setminus \{x\} \\
& (FV(x) \cup FV(y)) \setminus \{x\} \\
& (\{x\} \cup \{y\}) \setminus \{x\} \\
& \{y\} \\
FV(x\lambda x.xy) = \quad & \{x, y\}
\end{aligned}
$$

Substituţii

Fie t un λ-termen si $x \in Var$.

Definiţie intuitivă

Pentru un λ-termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor apariţiilor libere ale lui x cu u în t.

Substituţii

Fie t un λ-termen si $x \in Var$.

Definiţie intuitivă

Pentru un λ-termen u vom nota prin $[u/x]t$ rezultatul înlocuirii tuturor apariţiilor libere ale lui x cu u în t.

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- $[(\lambda z.zw)/x](\lambda y.x) = \lambda y.\lambda z.zw$

Substituţii

Definirea substituţiei

Rezultatul substituirii lui x cu u în t este definit astfel:

[Variabilă] $[u/x]x = u$

[Variabilă] $[u/x]y = y$ dacă $x \neq y$

[Aplicare] $[u/x](t_1 t_2) = [u/x]t_1 [u/x]t_2$

[Abstractizare] $[u/x]\lambda y.t = \lambda y.[u/x]t$ unde
$y \neq x$ şi $y \notin FV(u)$

Substituţii

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$

Substituţii

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?

Substituţii

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$

- Cine este $[y/x]\lambda y.x$?
  Dacă folosim definiţia intuitivă obţinem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greşit!

Substituţii

Exemple: Dacă x, y, z sunt variabile distincte atunci

- $[y/x]\lambda z.x = \lambda z.y$
- Cine este $[y/x]\lambda y.x$?
  Dacă folosim definiţia intuitivă obţinem
  $[y/x]\lambda y.x = \lambda y.y$ ceea ce este greşit!

  Cum procedăm pentru a repara greşeala? Observăm că $\lambda y.x$ desemneaza o funcţie constantă, aceeaşi funcţie putând fi reprezentată prin $\lambda z.x$. Aplicarea corectă a substituţiei este:
  $[y/x]\lambda y.x = [y/x]\lambda z.x = \lambda z.y$

  Avem libertatea de a redenumi variabilele legate!

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_\alpha t_2$ implică
$t\, t_1 =_\alpha t\, t_2$, $t_1 t =_\alpha t_2 t$ şi $\lambda x.t_1 =_\alpha \lambda x.t_2$

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_\alpha t_2$ implică
$t\, t_1 =_\alpha t\, t_2$, $t_1 t =_\alpha t_2 t$ şi $\lambda x.t_1 =_\alpha \lambda x.t_2$

Exemplu:
$[xy/x](\lambda y.yx) =_\alpha [xy/x](\lambda z.zx) =_\alpha \lambda z.z(xy)$

α-conversie (α-echivalenţă)

Definim următoarea relaţie peste termeni:

α-conversia $=_\alpha$

[Reflexivitate] $t =_\alpha t$

[Simetrie] $t_1 =_\alpha t_2$ implică $t_2 =_\alpha t_1$

[Tranzitivitate] $t_1 =_\alpha t_2$ şi $t_2 =_\alpha t_3$ implică $t_1 =_\alpha t_3$

[Redenumire] $\lambda x.t =_\alpha \lambda y.[y/x]t$ dacă $y \notin FV(t)$

[Compatibilitate] $t_1 =_\alpha t_2$ implică
$t\, t_1 =_\alpha t\, t_2$, $t_1 t =_\alpha t_2 t$ şi $\lambda x.t_1 =_\alpha \lambda x.t_2$

Exemplu:
$[xy/x](\lambda y.yx) =_\alpha [xy/x](\lambda z.zx) =_\alpha \lambda z.z(xy)$

Vom lucra modulo α-conversie, doi termeni α-echivalenţi vor fi consideraţi "egali".

α-conversie

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

α-conversie

Exemplu:

- $[xy/x]\lambda x.yx =_\alpha [xy/x]\lambda z.yz = \lambda z.[xy/x](yz) = \lambda z.yz$

  Observăm că $\lambda z.yz =_\alpha \lambda x.yx$

- $[y/z]\lambda xy.zzx = \lambda x.[y/z]\lambda y.zzx =_\alpha \lambda x.[y/z]\lambda v.zzx = \lambda x.\lambda v.[y/z](zzx) = \lambda xv.yyx$

β-reducţie

β-reducţia este o relaţie pe mulţimea α-termenilor.

β-reducţia $\to_\beta$, $\xrightarrow{*}_\beta$

- un singur pas $\to_\beta \subseteq \Lambda T \times \Lambda T$

  [Aplicarea] $(\lambda x.t)u \to_\beta [u/x]t$
  [Compatibilitatea] $t_1 \to_\beta t_2$ implică
  $t\, t_1 \to_\beta t\, t_2$, $t_1 t \to_\beta t_2 t$ şi $\lambda x.t_1 \to_\beta \lambda x.t_2$

β-reducţie

β-reducţia este o relaţie pe mulţimea α-termenilor.

β-reducţia $\to_\beta$, $\stackrel{*}{\to}_\beta$

- un singur pas $\to_\beta \subseteq \Lambda T \times \Lambda T$

  [Aplicarea] $(\lambda x.t)u \to_\beta [u/x]t$

  [Compatibilitatea] $t_1 \to_\beta t_2$ implică

  $t\, t_1 \to_\beta t\, t_2$, $t_1 t \to_\beta t_2 t$ şi $\lambda x.t_1 \to_\beta \lambda x.t_2$

- zero sau mai mulţi paşi $\stackrel{*}{\to}_\beta \subseteq \Lambda T \times \Lambda T$

  $t_1 \stackrel{*}{\to}_\beta t_2$ dacă există $n \geq 0$ şi $u_0, \ldots, u_n$ astfel încât

  $t_1 =_\alpha u_0 \to_\beta u_1 \to_\beta \cdots \to_\beta u_n =_\alpha t_2$

β-reducţie

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$

β-reducţie

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v \to_\beta zv$

β-reducţie

Să considerăm termenul $(\lambda x.(\lambda y.yx)z)v$

- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v \to_\beta zv$

Observăm că un termen poate fi β-redus în mai multe moduri.

Proprietatea de confluenţă ne asigură că vom ajunge întotdeauna la acelaşi rezultat.

Confluenţa β-reducţiei

Teorema Church-Rosser

Dacă $t \stackrel{*}{\rightarrow}_\beta t_1$ şi $t \stackrel{*}{\rightarrow}_\beta t_2$

$$\begin{tikzcd} & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\ t_1 & & t_2 \end{tikzcd}$$

Confluenţa β-reducţiei

Teorema Church-Rosser

Dacă $t \stackrel{*}{\rightarrow}_\beta t_1$ şi $t \stackrel{*}{\rightarrow}_\beta t_2$

$$
\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 & & t_2
\end{tikzcd}
$$

atunci există u astfel încât $t_1 \stackrel{*}{\rightarrow}_\beta u$ şi $t_2 \stackrel{*}{\rightarrow}_\beta u$.

$$
\begin{tikzcd}
 & t \arrow[dl, "*"'] \arrow[dr, "*"] & \\
t_1 \arrow[dr, "*"'] & & t_2 \arrow[dl, "*"] \\
 & u &
\end{tikzcd}
$$

β-forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normală

- un λ-termen căruia nu i se mai poate aplica reducerea într-un pas $\rightarrow_\beta$ se numeş te *β-formă normală*
- dacă $t \stackrel{*}{\rightarrow}_\beta u_1$, $t \stackrel{*}{\rightarrow}_\beta u_2$ ş i u_1, u_2 sunt β-forme normale atunci, datorită confluenţei, $u_1 =_\alpha u_2$
- un λ-termen poate avea cel mult o β-formă normală (modulo α-echivalenţă)

β-forma normală

Intuitiv, o formă normală este un termen care nu mai poate fi redus (sau punctul final al unui calcul).

Formă normală

- un λ-termen căruia nu i se mai poate aplica reducerea într-un pas $\to_\beta$ se numeş te *β-formă normală*
- dacă $t \stackrel{*}{\to}_\beta u_1$, $t \stackrel{*}{\to}_\beta u_2$ ş i u_1, u_2 sunt β-forme normale atunci, datorită confluenţei, $u_1 =_\alpha u_2$
- un λ-termen poate avea cel mult o β-formă normală (modulo α-echivalenţă)

Exemplu:

- zv este β-formă normală pentru $(\lambda x.(\lambda y.yx)z)v$
  $(\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda y.yv)z \to_\beta zv$
- există termeni care **nu** pot fi reduşi la o β-formă normală, de exemplu $(\lambda x.xx)(\lambda x.xx)$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \rightarrow_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \rightarrow_\beta (\lambda x.zx)v$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ şi $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ şi, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

β-conversia

Intuitiv, β-conversia extinde β-reducţia în ambele direcţii.

- $(\lambda y.yv)z \to_\beta zv \leftarrow_\beta (\lambda x.zx)v$
- $(\lambda y.yv)z \leftarrow_\beta (\lambda x.(\lambda y.yx)z)v \to_\beta (\lambda x.zx)v$

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ şi $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ şi, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

Exemplu: $(\lambda y.yv)z =_\beta (\lambda x.zx)v$

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

Observaţii

- $=_\beta$ este o relaţie de echivalenţă

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \to_\beta u_{i+1}$ sau $u_{i+1} \to_\beta u_i$

Observaţii

- $=_\beta$ este o relaţie de echivalenţă
- pentru t_1, t_2 λ-termeni şi u_1, u_2 β-forme normale
  dacă $t_1 \stackrel{*}{\to}_\beta u_1$, $t_2 \stackrel{*}{\to}_\beta u_2$ şi $u_1 =_\alpha u_2$ atunci $t_1 =_\beta t_2$

β-conversia

β-conversia $=_\beta$

- $=_\beta \subseteq \Lambda T \times \Lambda T$
  $t_1 =_\beta t_2$ dacă există $n \geq 0$ ş i $u_0, \ldots, u_n$ astfel încât
  $t_1 =_\alpha u_0$, $u_n =_\alpha t_2$ ş i, pentru orice i, $u_i \rightarrow_\beta u_{i+1}$ sau $u_{i+1} \rightarrow_\beta u_i$

Observaţii

- $=_\beta$ este o relaţie de echivalenţă
- pentru t_1, t_2 λ-termeni şi u_1, u_2 β-forme normale
  dacă $t_1 \stackrel{*}{\rightarrow}_\beta u_1$, $t_2 \stackrel{*}{\rightarrow}_\beta u_2$ şi $u_1 =_\alpha u_2$ atunci $t_1 =_\beta t_2$

β-conversia reprezintă "egalitatea prin calcul", iar β-reducţia (modulo α-conversie) oferă o procedură de decizie pentru aceasta.

Pe săptămâna viitoare!