Programare declarativă

Combinatori aplicativi pentru analiză sintactică

Ioana Leuștean
Traian Florin Șerbănuță

Departamentul de Informatică, FMI, UB
ioana@fmi.unibuc.ro
traian.serbanuta@unibuc.ro

22 decembrie 2020

Functori aplicativi

Problemă

- Folosind fmap putem transforma o funcție h :: a –> b într-o funcție între colecții/computații fmap h :: m a –> m b
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente E.g., cum trecem de la h :: a –> b –> c la h' :: m a –> m b –> m c
- putem încerca să folosim fmap

Problemă

- Folosind fmap putem transforma o funcție h :: a –> b într-o funcție între colecții/computații fmap h :: m a –> m b
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente
  E.g., cum trecem de la h :: a –> b –> c la h' :: m a –> m b –> m c
- putem încerca să folosim fmap
- Dar, deoarece h :: a –> (b –> c), avem că
  fmap h :: m a –> m (b –> c)
- Putem aplica fmap h la o valoare ca :: m a și obținem
  fmap h ca :: m (b –> c)

Problemă

Cum transformăm un obiect din m (b –> c) într-o funcție m b –> m c?

- (<*>) :: m (b –> c) –> m b –> m c

Merge pentru funcții cu oricâte argumente

Problemă

Dată fiind o funcție f :: a1 –> a2 –> a3 –> ... –> an –> a și computațiile ca1 :: m a1, . . . , can :: m an, vrem să „combinăm” rezultatele computațiilor ca1, . . . , can folosind funcția f pentru a obține o computație finală ca :: m a, fără a pierde efectele laterale specifice argumentelor.

Soluție: Date fiind

- funcția fmap :: (a –> b) –> m a –> m b
- funcția (<*>) :: m (b –> c) –> m b –> m c cu „proprietăți bune”

Atunci

```
fmap f :: m a1 –> m (a2 –> a3 –> ... –> an –> a)
fmap f ca1 :: m (a2 –> (a3 –> ... –> an –> a))
fmap f ca1 <*> ca2 :: m (a3 –> ... –> an –> a)
...
fmap f ca1 <*> ca2 <*> ca3 ... <*> can :: m a
```

Clasa de tipuri Applicative

Definiție

```
class Functor m => Applicative m where
  pure :: a -> m a
(<*>) :: m (a -> b) -> m a -> m b
```

- Orice instanță a lui Applicative trebuie să fie instanță a lui **Functor**
- pure transformă o valoare într-o computație minimală care are acea valoare ca rezultat
- (<*>) ia o computație care produce funcții și o computație care produce argumente pentru funcții și obține o computație care produce rezultatele aplicării funcțiilor asupra argumentelor

Proprietate importantă

- fmap f x == pure f <*> x
- Se definește operatorul (<$>) prin (<$>) = fmap

Tipul opțiune

```
Main> pure "Hey" :: Maybe String
Just "Hey"
Main> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
Main> let mDiv x y = if y == 0 then Nothing else Just (x `
    div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Tipul opțiune

```
Main> pure "Hey" :: Maybe String
Just "Hey"
Main> (++) <$> (Just "Hey ") <*> (Just "You!")
Just "Hey You!"
Main> let mDiv x y = if y == 0 then Nothing else Just (x `
    div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Instanță pentru tipul opțiune

```
instance Applicative Maybe where
  pure = Just
  Nothing <*> _ = Nothing
  Just f  <*> x = fmap f x
```

Tipul eroare (Either)

```
Main> pure "Hey" :: Either a String
Right "Hey"
Main> (++) <$> (Right "Hey ") <*> (Right "You!")
Right "Hey You!"
Main> let mDiv x y = if y == 0 then Left "Division by 0!"
    else Right (x `div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Tipul eroare (Either)

```
Main> pure "Hey" :: Either a String
Right "Hey"
Main> (++) <$> (Right "Hey ") <*> (Right "You!")
Right "Hey You!"
Main> let mDiv x y = if y == 0 then Left "Division by 0!"
    else Right (x `div` y)
Main> let f x = 4 + 10 `div` x
Main> let mF x = (+) <$> pure 4 <*> mDiv 10 x
```

Instanță pentru tipul eroare

```
instance Applicative (Either a) where
  pure = Right
  Left e <*> _ = Left e
  Right f  <*> x = fmap f x
```

Tipul listă (computație nedeterministă)

```
Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"]["Hello world","Hello happiness","Goodbye
    world","Goodbye happiness"]
Main> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Tipul listă (computație nedeterministă)

```
Main> pure "Hey" :: [String]
["Hey"]
Main> (++) <$> ["Hello ", "Goodbye "] <*> ["world", "
    happiness"]["Hello world","Hello happiness","Goodbye
    world","Goodbye happiness"]
Main> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
Main> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

Instanță pentru tipul computațiilor nedeterministe (liste)

```
instance Applicative [] where
  pure x = [x]
  fs  <*> xs = [f x | f <- fs, x <- xs]
```

Tipul funcțiilor de sursă dată

```
data Exp = Lit Int | Var String | Exp :+: Exp
type Env = [(String, Int)]

find :: String -> (Env -> Int)
find x env = head [i | (y,i) <- env, y == x]

eval :: Exp -> (Env -> Int)
eval (Lit i) = pure i
eval (Var x) = find x
eval (e1 :+: e2) = (+) <$> eval e1 <*> eval e2
```

Tipul funcțiilor de sursă dată

```
data Exp = Lit Int | Var String | Exp :+: Exp
type Env = [(String, Int)]

find :: String -> (Env -> Int)
find x env = head [i | (y,i) <- env, y == x]

eval :: Exp -> (Env -> Int)
eval (Lit i) = pure i
eval (Var x) = find x
eval (e1 :+: e2) = (+) <$> eval e1 <*> eval e2
```

Instanță pentru tipul funcțiilor de sursă dată

```
instance Applicative ((->) t) where
  pure :: a -> (t -> a)
  pure x = \ _ -> x
  (<*>) :: (t -> (a -> b)) -> (t -> a) -> (t -> b)
  f  <*> g = \ x -> f x (g x)
```

Liste ca fluxuri de date.

```
newtype ZipList a = ZipList { get :: [a]}

> get $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
> get $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
> get $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList
    "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Liste ca fluxuri de date.

```
newtype ZipList a = ZipList { get :: [a]}

> get $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
> get $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
> get $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList
    "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

Instanță pentru ZipList

```
instance Functor ZipList where
  fmap f (ZipList xs) = ZipList (fmap f xs)

instance Applicative ZipList where
  pure x = repeat x
  ZipList fs <*> ZipList xs =
    ZipList (zipWith (\ f x -> f x) fs xs)
```

Proprietăți ale functorilor aplicativi

identitate `pure id <*> v = v`

compoziție `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`

homomorfism `pure f <*> pure x = pure (f x)`

interschimbare `u <*> pure y = pure ($ y) <*> u`

Consecință: `fmap f x == f <$> x == pure f <*> x`

Operații "ciudate" pe functori aplicativi

Păstrăm primul rezultat și îl ignorăm pe al doilea

```
(<*) :: Applicative m => m a => m b => m a
ma <* mb = pure const <*> ma <*> mb
```

Păstrăm al doilea rezultat și îl ignorăm pe primul

```
(*>) :: Applicative m => m a => m b => m b
ma *> mb = pure (flip const) <*> ma <*> mb
```

- Nu par să aibă foarte mult sens
  - De ce să ignorăm unul din argumente?

Operații "ciudate" pe functori aplicativi

Păstrăm primul rezultat și îl ignorăm pe al doilea

```
(<*) :: Applicative m => m a => m b => m a
ma <* mb = pure const <*> ma <*> mb
```

Păstrăm al doilea rezultat și îl ignorăm pe primul

```
(*>) :: Applicative m => m a => m b => m b
ma *> mb = pure (flip const) <*> ma <*> mb
```

- Nu par să aibă foarte mult sens
  - De ce să ignorăm unul din argumente?
- Capătă sens dacă ne gândim la ma, mb ca acțiuni cu efecte laterale
  - ignorăm rezultatul dar propagăm efectele laterale

Alternative: Functori aplicativi cu structură de monoid

Alternative specifică... alternative și colectează rezultatele

```
class Applicative f => Alternative f where
  empty :: f a
  (<|>) :: f a -> f a -> f a

  many :: f a -> f [a] -- Zero or more
  many v = some v <|> pure []

  some :: f a -> f [a] -- One or more
  some v = pure (:) <*> v <*> many v
```

- many repetă aceeași computație de zero sau mai multe ori colectând rezultatele într-o listă
- some repetă aceeași computație de una sau mai multe ori colectând rezultatele într-o listă
- Observație: computația repetată ar trebui să eșueze la un moment dat

Instanțe pentru Alternative

Liste

```
instance Alternative [] where
  empty = []
  (<|>) = (++)
```

Maybe

```
instance Alternative Maybe where
  empty = Nothing

  Nothing <|> x = x
  x <|> _ = x
```

Instanțele sunt structurile naturale de monoid peste tipurile respective.

Analiză sintactică

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

A doua încercare

```
type Parser a = String -> (a, String)
```

Tipul unui analizor sintactic

Prima încercare

```
type Parser a = String -> a
```

- Dar cel puțin pentru rezultate parțiale, va mai rămâne ceva de analizat

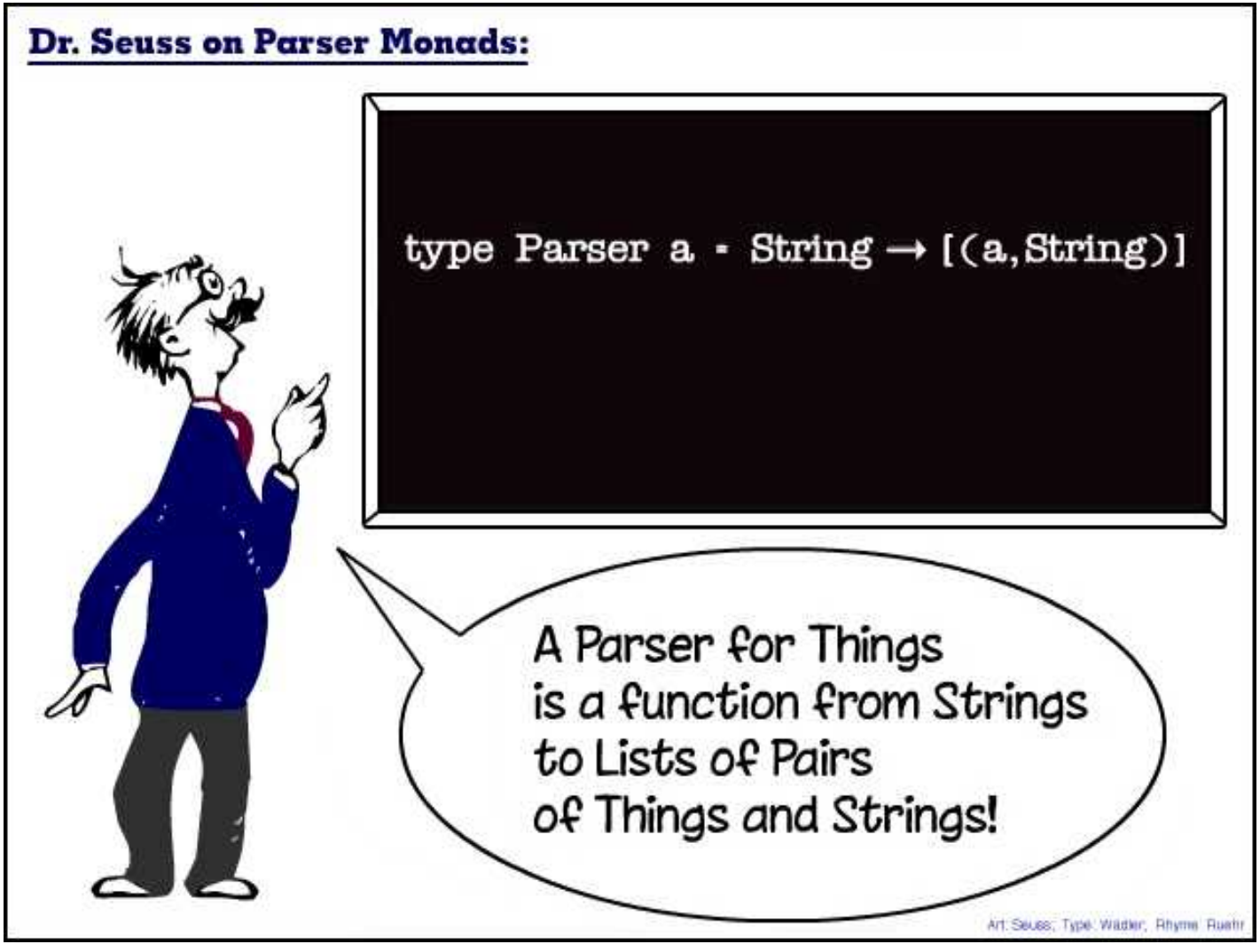
A doua încercare

```
type Parser a = String -> (a, String)
```

- Dar dacă gramatica e ambiguă?
- Dar dacă intrarea nu corespunde nici unui element din a?

Tipul unui analizor sintactic

A treia încercare



Tipul Parser

```
-- Tipul (incapsulat) Parser
newtype Parser a =
  Parser { apply :: String -> [(a, String)] }

-- Daca exista parsare, da prima varianta
parse :: Parser a -> String -> a
parse m s = head [ x | (x,t) <- apply m s, t == "" ]
```

Parser e functor

```
--    class Functor m where
--      fmap :: (a -> b) -> m a -> m b

instance Functor Parser where
  fmap f p =
    Parser
      (\s -> [(f a, r) | (a, r) <- apply p s])
```

- fmap ne ajută să transformăm un parser

```
parseString :: Parser String
newtype Name = Name { getName :: String}
parseName :: Parser Name
parseName = fmap Name parseString
-- sau Name <$> parseString
```

Parser e aplicativ

```
--    class Applicative m where
--      pure :: a -> m a
--      (<*>) :: m (a -> b) -> m a -> m b

instance Applicative Parser where
  pure a = Parser (\s -> [(a, s)])
  pf <*> pa =
      Parser
          (\s -> [(f a, r) | (f, rf) <- apply pf s
                           , (a, r) <- apply pa rf ])
```

- pure și <*> ne ajută să combinăm

```
data Point = Point Int Int
parsePoint = pure Point <*> parseInt <*> parseInt
```

- (<*) :: Parser a -> Parser b -> Parser a și
  (*>) :: Parser a -> Parser b -> Parser b
  ajută la ignorarea terminalelor

Parser e alternativ

```
-- class Alternative m where
--   empty :: m a
--   (<|>) :: m a -> m a -> m a

instance Alternative Parser where
  empty = Parser (\s -> [])
  pa <|> pb = Parser (\s -> apply pa s ++ apply pb s)
```

- empty reprezintă analizorul sintactic care eșuează tot timpul
- <|> reprezintă combinarea alternativelor — ajută la combinarea producțiilor pentru același non-terminal
- some, many :: Parser a -> Parser [a] folosesc pentru expresii regulate
  - some pa — una sau mai multe copii ale lui pa
  - many pa — zero sau mai multe copii ale lui pa

Parsare pentru caractere

```
-- Recunoasterea unui caracter cu o proprietate
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = Parser f
  where
  f []                    = []
  f (c:s) | p c           = [(c, s)]
          | otherwise = []

-- Recunoasterea unui anumit caracter
char :: Char -> Parser Char
char c = satisfy (== c)
```

Eliminarea spațiilor

WhiteSpace

```
skipSpace :: Parser ()
skipSpace = many (satisfy isSpace) *> pure ()
```

Eliminarea spațiilor de după

```
token :: Parser a -> Parser a
token p = skipSpace *> p <* skipSpace
```

Recunoașterea unui numar întreg

```
-- Recunoasterea unui numar natural
parseNat :: Parser Int
parseNat = read <$> some (satisfy isDigit)

-- Recunoasterea unui numar negativ
parseNeg :: Parser Int
parseNeg =  char '-' *> (negate <$> parseNat)

-- Recunoasterea unui numar intreg
parseInt :: Parser Int
parseInt = parseNat <|> parseNeg
```

Expresii aritmetice

```
data Exp = Lit Int
         | Add Exp Exp
         | Mul Exp Exp
         deriving (Eq,Show)

evalExp    :: Exp -> Int
evalExp    (Lit n)     = n
evalExp    (Add e f) = evalExp e + evalExp f
evalExp    (Mul e f) = evalExp e * evalExp f
```

Recunoașterea unei expresii

```
parseExp :: Parser Exp
parseExp = parseTerm
    <|> pure Add <*> parseTerm <* token (char '+') <*>
        parseExp
  where
    parseTerm = parseFactor
        <|> pure Mul <*> parseFactor <* token (char '*')
            <*> parseTerm
    parseFactor = Lit <$> parseInt
        <|> char '(' *> skipSpace *> parseExp <* skipSpace
             <* char ')'
```

Recunoașterea unei expresii

Test

```
*Exp> parse parseExp "1+2*3"
Lit 1 :+: (Lit 2 :*: Lit 3)
*Exp> evalExp (parse parseExp "1+2*3")
7
*Exp> parse parseExp "(1+2)*3"
(Lit 1 :+: Lit 2) :*: Lit 3
*Exp> evalExp (parse parseExp "(1+2)*3")
9
```