PARADIGME DE PROGRAMARE

Curs 18

Raționament ecuațional.

Raționament ecuațional – Cuprins

- Motivație
- Compoziționalitate
- Proprietăți ale funcționalelor
- Optimizarea implementărilor
- Reduceri (folds)

Raționament ecuațional

Programare funcțională = calcul, nu execuție

- Program = set de ecuații matematice
- Transparență referențială
  - Substituția egalilor: Dacă x = y, atunci y poate înlocui x în orice context
  - Demonstrații de corectitudine: Demonstrații matematice prin substituția egalilor
  - Limbaj: Limbajul de programare (demonstrațiile nu necesită notații externe)

Raționament ecuațional = **manipularea codului ca set de ecuații matematice**, in scop de:

- Demonstrare a corectitudinii: inducție structurală
- Optimizare a implementării: rescrieri care conduc la execuție mai eficientă

Raționament ecuațional – Cuprins

- Motivație
- Compoziționalitate
- Proprietăți ale funcționalelor
- Optimizarea implementărilor
- Reduceri (folds)

Transformare compozițională

Transformare T compozițională

- Rezultatul aplicării T asupra unei structuri complexe depinde doar de rezultatele aplicării T asupra substructurilor (nu de alte caracteristici ale substructurilor, nici de informație externă)
  - Liste: T(x:xs) = f(x, T(xs)) T se aplică doar pe xs, iar xs este inspectat doar de T
  - Arbori binari: T(node(x,l,r)) = f(x, T(l), T(r))

Exemple

- Compoziționale:

```
sum (x:xs) = x + sum xs
size (Node x l r) = 1 + size l  + size r
```

- Necompoziționale:

```
isSorted (x:y:xs) = x <= y && isSorted (y:xs)
isBST (Node x l r) = x >= maxVal l && x <= minVal r
                          && isBST l && isBST r
```

Compoziționalitate = predictibilitate

Compoziționalitatea oferă garanții de:

- Design: compoziționalitate = localitate
  rezultat = f (parametrii curenți, rezultatul apelului recursiv)
- Verificare: compoziționalitate = pas inductiv care curge natural
  P(xs) conține toată informația necesară pentru a demonstra P(x:xs)
- Calcul: compoziționalitate = raționament ecuațional valid
  Proprietățile folosite de raționamentul ecuațional sunt valide (demonstrate) doar pentru transformări compoziționale

În absența compoziționalității, șabloanele de design / demonstrație / calcul sunt „murdărite” de soluții particulare de a penetra cutia neagră care este substructura.

- Ex: isSorted: accesează primul element din xs pentru a-l compara cu x
  isBST: accesează elementul maxim din l / minim din r

Raționament ecuațional – Cuprins

- Motivație
- Compoziționalitate
- Proprietăți ale funcționalelor
- Optimizarea implementărilor
- Reduceri (folds)

Proprietățile funcționalei **map**

1. `map id = id` — Identitate
2. `map (f . g) = map f . map g` — Fuziune

Demonstrație fuziune (prin inducție structurală)

```
-- Caz de bază: xs = []
LHS: map (f . g) [] = []
RHS: (map f . map g) [] = map f [] = []          -- LHS = RHS
-- Pas inductiv: xs = (x:xs')
LHS: map (f . g) (x:xs') =
     (f . g) x : map (f . g) xs'
RHS: (map f . map g) (x:xs') =
     map f (g x : map g xs') =
     f (g x) : map f (map g xs') =
     (f . g) x : (map f . map g) xs' =           -- ipoteza inductivă
     (f . g) x : map (f . g) xs' =               -- LHS = RHS
```

Proprietățile funcționalei `filter`

1. `filter (\x -> p x && q x) = filter q . filter p` Fuziune
2. `filter p . map f = map f . filter (p . f)` Comutativitate cu map

Observație: Aceste proprietăți (și, în general, raționamentul ecuațional) sunt valide doar dacă transformările (f, g, p, q, etc.) sunt compoziționale.

- Limbaje pure (Haskell): orice funcție unară f :: a -> b este automat compozițională
- Limbaje impure (C++, Java, Python): o funcție unară sintactic poate fi necompozițională
  - Ex:
    - f(x) incrementează o variabilă globală counter și întoarce x + counter
    - g(x) incrementează aceeași variabilă counter și întoarce x – counter
    - Pentru `counter = 0` (inițial), și `xs = [10, 20]` fuziunea eșuează:
      - `map (f . g) xs = [11, 21]`
      - `(map f . map g) xs = [12, 22]`

Proprietățile funcționalei `foldr`

- Universalitate: Orice funcție compozițională pe liste se poate exprima prin foldr
  - Identitate: id `foldr (:) []`
  - Transformare:
    - map `foldr (\x acc -> f x : acc) []`
    - filter `foldr (\x acc -> if p x then x : acc else acc) []`
    - reverse `foldr (\x acc -> acc ++ [x]) []`
  - Reducere:
    - sum `foldr (+) 0`
    - product `foldr (*) 1`
    - any p `foldr ((||) . p) False`
    - all p `foldr ((&&) . p) True`
- Generalizare: `g = foldr f acc` este echivalentă cu orice funcție g definită prin:

```
g [] = acc
g (x:xs) = f x (g xs)
```

Proprietățile funcționalei `foldr`

```
h . foldr f a = foldr g b
```

Fuziune

Condiții:

```
h a = b
h (f x y) = g x (h y)
```

Observație

- Condiția 2 este constructivă – oferă o rețetă simplă, universală, de a determina funcția g, eliminând nevoia de a face inducție structurală de fiecare dată
- Condiția 2 nu este necesară – există cazuri particulare cu alte funcții g pentru care fuziunea funcționează, dar acestea rămân cazuri particulare, nu proprietăți generale utilizabile în raționamentul ecuațional

Utilizarea proprietăților în demonstrații

Demonstrație `(*2) . sum = sum . map (*2)`

```
-- LHS

(*2) . sum =                    -- definiție sum
(*2) . foldr (+) 0 =            -- fuziune foldr: h = (*2), f = (+), a = 0
foldr (\x acc -> x*2 + acc) 0

-- Calcule pentru fuziune

h a = b                         b = h 0 = 0

h (f x y) = g x (h y)           g x acc = g x (y*2) = (x + y)*2 = x*2 + y*2 =
                                x*2 + acc
```

Utilizarea proprietăților în demonstrații

Demonstrație `(*2) . sum = sum . map (*2)`

```
-- RHS

sum . map (*2) =                                  -- map ca foldr
sum . foldr (\x acc -> x*2 : acc) [] =            -- fuziune: h = sum, f = ..., a = []
foldr (\x acc -> x*2 + acc) 0                     -- LHS = RHS

-- Calcule pentru fuziune

h a = b                          b = sum [] = 0

h (f x y) = g x (h y)            g x acc = g x (sum y) = sum (x*2 : y) =
                                 x*2 + sum y = x*2 + acc
```

Raționament ecuațional – Cuprins

- Motivație
- Compoziționalitate
- Proprietăți ale funcționalelor
- Optimizarea implementărilor
- Reduceri (folds)

Demonstrație ca rescriere optimizată

Demonstrație `(*2) . sum = sum . map (*2) = foldr ((+) . (*2)) 0`

Observație: Raționamentul ecuațional este și o metodă de optimizare a implementărilor!

- Demonstrația proprietății a generat rescrierea optimizată a RHS
- RHS: map construiește o listă intermediară, sum consumă lista
  - Evaluare aplicativă: timp O(n) + O(n), spațiu auxiliar O(n) pentru lista intermediară
  - Evaluare leneșă: timp O(n)
    spațiu auxiliar O(1) pentru lista intermediară
    ➔ map construiește celule intermediare una câte una, care se eliberează imediat ce au fost consumate de sum
    spațiu auxiliar O(n) pentru recursivitate pe stivă și însumare leneșă
- foldr: consumă lista inițială
  - tot timp O(n) și spațiu auxiliar O(n), dar mai eficient:
    o singură parcurgere, în care nu se creează nici liste, nici celule intermediare
    ➔ mai puțin garbage collection, O(n) cu constantă mai mică

RE ➔ Recursivitate pe coadă

- Acumulator
  - Tehnică de conversie din recursivitatea pe stivă în recursivitate pe coadă
    - Liste: fStack xs ➔ fTail xs acc
  - Rol: stochează rezultatul parțial al calculelor efectuate la avansul în recursivitate
- Conversia stivă-coadă prin raționament ecuațional
  = dezvoltarea unei proprietăți RE de forma **fTail xs acc = fStack xs ⊕ acc**
  - ⊕ este funcția de combinare dintre:
    - Procesarea restului listei
    - Rezultatul parțial stocat în acc
  - ex: `sumTail xs acc = sumStack xs + acc`

Exemple

```
sumTail xs acc = sumStack xs + acc

prodTail xs acc =

lenTail xs acc =

revTail xs acc =

minTail xs acc =

mapTail' f xs acc =
```

Exemple

```
sumTail xs acc = sumStack xs + acc

prodTail xs acc = prodStack xs * acc

lenTail xs acc = lenStack xs + acc

revTail xs acc = revStack xs ++ acc

minTail xs acc = minStack xs `min` acc

mapTail’ f xs acc = reverse (mapStack f xs) ++ acc
```

RE ➔ Recursivitate pe coadă

Proprietatea RE se valorifică prin pattern matching pe constructorii de date:

```
sumTail [] acc =            -- proprietatea RE
sumStack [] + acc =         -- axioma sum1
acc

sumTail (x:xs) acc =        -- proprietatea RE
sumStack (x:xs) + acc =     -- axioma sum2
x + sumStack xs + acc =     -- comutativitate + asociativitate adunare
sumStack xs + (x + acc) =   -- proprietatea RE
sumTail xs (x + acc)
```

Rezultat: implementarea prin recursivitate pe coadă

```
sumTail [] acc = acc
sumTail (x:xs) acc = sumTail xs (x + acc)
```

RE ➔ Recursivitate pe coadă

Proprietatea RE se valorifică prin pattern matching pe constructorii de date:

```
mapTail' f [] acc =                  -- proprietatea RE
reverse (mapStack f []) ++ acc =     -- axioma map1
reverse [] ++ acc                    -- axioma reverse1
[] ++ acc                            -- axioma ++1
acc
```

```
mapTail' f (x:xs) acc =                          -- proprietatea RE
reverse (mapStack f (x:xs)) ++ acc =             -- axioma map2
reverse (f x : mapStack f xs) ++ acc =           -- axioma reverse2
(reverse (mapStack f xs) ++ [f x]) ++ acc =      -- asociativitate ++
reverse (mapStack f xs) ++ ([f x] ++ acc) =      -- axiome ++2, ++1
reverse (mapStack f xs) ++ (f x : acc) =         -- proprietatea RE
mapTail' f xs (f x : acc)
```

RE ➔ Tupling

Exemplu: `mean xs = sum xs / length xs`

- Problema: 2 parcurgeri xs
- Soluția: o singură parcurgere care întoarce perechea (sumă, lungime)
- Proprietatea RE: `sumLen xs = (sum xs, length xs)`

Pattern matching pe constructorii de date:

```
sumLen [] =                    -- proprietatea RE
(sum [], length []) =          -- axiome sum1, length1
(0, 0)

sumLen (x:xs) =                                    -- proprietatea RE
(sum x:xs, length x:xs) =                          -- axiome sum2, length2
(x + sum xs, 1 + length xs) =                      -- proprietatea RE
let (s, len) = sumLen xs in (x + s, 1 + len)
```

Recunoaștere șablon

Exemplu: `mean xs = sum xs / length xs`

- Soluția RE:

```
--sumLen xs = (sum xs, length xs)
sumLen [] = (0, 0)
sumLen (x:xs) = let (s, len) = sumLen xs in (x + s, 1 + len)
```

- Observații:
  - sum, length, sumLen sunt transformări compoziționale pe liste și pot fi rescrise cu foldr:

```
sum = foldr (+) 0
length = foldr (const (+1)) 0
sumLen = foldr (\x (s, len) -> (x + s, 1 + len)) (0, 0)
```

  - Șablon: Combinare a 2 reduceri (foldr) într-una singură
  - Scop: Abstractizare șablon
  - Pași: Recunoaștere șablon -> Definire -> Reutilizare

Raționament ecuațional – Cuprins

- Motivație
- Compoziționalitate
- Proprietăți ale funcționalelor
- Optimizarea implementărilor
- Reduceri (folds)

Reduceri și combinări de reduceri

Tipuri de reduceri

- Reduceri liniare: foldr
  - orice foldr este, în esență, un foldr pe liste:
    lista procesată este liniarizarea valorilor din container, într-o ordine acceptată ca dreapta-stânga
  - ex: transformări compoziționale pe liste, unele transformări compoziționale pe arbori binari (sum, size)
- Reduceri ierarhice: ?
  - reduceri care nu funcționează pe conținutul liniarizat, întrucât depind de structura datelor
  - ex: înălțimea unui arbore binar necesită să comparăm înălțimile celor doi subarbori

Tipuri de combinări

- Reduceri independente: ex: sum și length
- Reduceri semidependente: ex: sum și steep (steep: fiecare element > suma restului)
- Reduceri mutual dependente: ex: evenSum și oddSum (suma pozițiilor pare/impare)

Abstractizarea reducerilor liniare

```
data Folder a b c = Folder
  { foldNull :: c                    -- acc inițial (rezultatul pentru [])
  , foldCons :: a -> b -> c          -- funcția binară (care reduce liste nevide)
  }
```

Explicații

- Cele 2 câmpuri (acumulatorul inițial, funcția binară) diferențiază un foldr de altul
- Tradițional: `foldr :: Foldable t => (a -> c -> c) -> c -> t a -> c`
  - `(a -> c -> c)` — foldCons
  - `c` — foldNull
- Generalizarea `foldCons :: a -> b -> c` servește combinării reducerilor dependente
  - În calculul propriului acumulator următor, se ține cont de:
    - Propriul acumulator anterior
    - Acumulatorul anterior al celuilalt fold

Utilizare

```
fold :: Foldable t => Folder a b b -> t a -> b

fold folder = foldr (foldCons folder) (foldNull folder)
```

Exemplu

```
sum :: (Foldable t, Num a) => t a -> a
sum = fold sumFolder

sumFolder :: Num a => Folder a a a
sumFolder = Folder
  { foldNull = 0
  , foldCons = (+)
  }
```

Combinare: Reduceri independente

```
(<+>) :: Folder a b b
      -> Folder a c c
      -> Folder a (b, c) (b, c)
f <+> g = Folder
  { foldNull = (foldNull f, foldNull g)
  , foldCons = \a (b, c) -> (foldCons f a b, foldCons g a c)
  }
```

Explicație

- Tupling: <+> combină două reduceri independente într-una singură
- Utilizare: `sumLen = fold (sumFolder <+> lenFolder)`
  - Modularitate prin reutilizarea sumFolder și lenFolder, în contrast cu asamblarea lor prin RE

Combinare: Reduceri semidependente

```
(>.>) :: Folder a b b
      -> Folder a (b, c) c
      -> Folder a (b, c) (b, c)
f >.> g = Folder
  { foldNull = (foldNull f, foldNull g)
  , foldCons = \a (b, c) -> (foldCons f a b, foldCons g a (b, c))
  }
```

Explicație

- Tupling: >.> combină două reduceri semidependente, cu flux de informație stânga-dreapta
- Utilizare: `steep = snd . fold (sumFolder >.> steepFolder)`

Combinare: Reduceri mutual dependente

```
(<.>) :: Folder a (b, c) b
      -> Folder a (b, c) c
      -> Folder a (b, c) (b, c)
f <.> g = Folder
  { foldNull = (foldNull f, foldNull g)
  , foldCons = \a (b, c) -> (foldCons f a (b, c), foldCons g a (b, c))
  }
```

Explicație

- Tupling: <.> combină două reduceri mutual dependente, cu flux bidirecțional de informație
- Utilizare: `evenOddSums = fold (evenFolder <.> oddFolder)`

Abstractizarea reducerilor ierarhice – BST

```haskell
data BSTFolder a b c = BSTFolder
  { foldNil :: c                    -- acc inițial (rezultatul pentru BSTNil)
  , foldNod :: a -> b -> b -> c     -- funcția binară (reduce arbori nevizi)
  }
```

Utilizare

```haskell
foldBST :: Foldable t => BSTFolder a b b -> t a -> b
foldBST folder = go
  where
  go BSTNil = foldNil folder
  go (BSTNod elem left right) = foldNod folder elem (go left) (go right)
```

Combinare polimorfică

```
1.   -- Clasa Aggregatable
2.   -- Grupează operatorii de combinare pentru reduceri
3.   class Agg t where
4.     --Combinare independentă: fuzionează două reduceri paralele
5.     (<+>) :: t a b b -> t a c c -> t a (b, c) (b, c)
6.     -- Combinare semidependentă: a doua folosește rezultatul primeia
7.     (>.>) :: t a b b -> t a (b, c) c -> t a (b, c) (b, c)
8.     -- Combinare mutual dependentă: ambele au acces la rezultatul comun
9.     (<.>) :: t a (b, c) b -> t a (b, c) c -> t a (b, c) (b, c)
```

Explicație

- Polimorfism: atât reducerile liniare, cât și cele ierarhice se pot combina în cele 3 moduri
- Instanțiere: implementări diferite pentru Folder și BSTFolder

Instanțiere cu BSTFolder

```
1.   instance Agg BSTFolder where
2.    f <+> g = BSTFolder (foldNil f, foldNil g)
3.                        (\a (b1, c1) (b2, c2) ->
4.                        (foldNod f a b1 b2,
                           foldNod g a c1 c2))
5.    f >.> g = BSTFolder (foldNil f, foldNil g)
6.                        (\a (b1, c1) (b2, c2) ->
7.                        (foldNod f a b1 b2,
                           foldNod g a (b1, c1) (b2, c2)))
8.    f <.> g = BSTFolder (foldNil f, foldNil g)
9.                        (\a (b1, c1) (b2, c2) ->
10.                       (foldNod f a (b1, c1) (b2, c2),
                           foldNod g a (b1, c1) (b2, c2)))
```

Rezumat

Raționament ecuațional

Transformare compozițională

Proprietăți map

Proprietăți filter

Proprietăți foldr

Conversia stivă-coadă

Combinare independentă

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional: manipularea codului ca set de ecuații matematice

Transformare compozițională

Proprietăți map

Proprietăți filter

Proprietăți foldr

Conversia stivă-coadă

Combinare independentă

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional: manipularea codului ca set de ecuații matematice

Transformare compozițională: T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))

Proprietăți map

Proprietăți filter

Proprietăți foldr

Conversia stivă-coadă

Combinare independentă

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional: manipularea codului ca set de ecuații matematice

Transformare compozițională: T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))

Proprietăți map: identitate, fuziune

Proprietăți filter

Proprietăți foldr

Conversia stivă-coadă

Combinare independentă

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional:	manipularea codului ca set de ecuații matematice
Transformare compozițională:	T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))
Proprietăți map:	identitate, fuziune
Proprietăți filter:	fuziune, comutativitate cu map

Proprietăți foldr

Conversia stivă-coadă

Combinare independentă

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional:	manipularea codului ca set de ecuații matematice
Transformare compozițională:	T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))
Proprietăți map:	identitate, fuziune
Proprietăți filter:	fuziune, comutativitate cu map
Proprietăți foldr:	universalitate, fuziune

Conversia stivă-coadă

Combinare independentă

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional:	manipularea codului ca set de ecuații matematice
Transformare compozițională:	T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))
Proprietăți map:	identitate, fuziune
Proprietăți filter:	fuziune, comutativitate cu map
Proprietăți foldr:	universalitate, fuziune
Conversia stivă-coadă:	fTail xs acc = fStack xs $\oplus$ acc, particularizat pe constructori

Combinare independentă

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional:	manipularea codului ca set de ecuații matematice
Transformare compozițională:	T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))
Proprietăți map:	identitate, fuziune
Proprietăți filter:	fuziune, comutativitate cu map
Proprietăți foldr:	universalitate, fuziune
Conversia stivă-coadă:	fTail xs acc = fStack xs ⊕ acc, particularizat pe constructori
Combinare independentă:	(<+>) :: t a b b -> t a c c -> t a (b, c) (b, c)

Combinare semidependentă

Combinare mutual dependentă

Rezumat

Raționament ecuațional:	manipularea codului ca set de ecuații matematice
Transformare compozițională:	T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))
Proprietăți map:	identitate, fuziune
Proprietăți filter:	fuziune, comutativitate cu map
Proprietăți foldr:	universalitate, fuziune
Conversia stivă-coadă:	fTail xs acc = fStack xs $\oplus$ acc, particularizat pe constructori
Combinare independentă:	(<+>) :: t a b b -> t a c c -> t a (b, c) (b, c)
Combinare semidependentă:	(>.>) :: t a b b -> t a (b, c) c -> t a (b, c) (b, c)

Combinare mutual dependentă

Rezumat

Raționament ecuațional:	manipularea codului ca set de ecuații matematice
Transformare compozițională:	T(x:xs) = f(x, T(xs)), T(node(x, l, r)) = f(x, T(l), T(r))
Proprietăți map:	identitate, fuziune
Proprietăți filter:	fuziune, comutativitate cu map
Proprietăți foldr:	universalitate, fuziune
Conversia stivă-coadă:	fTail xs acc = fStack xs $\oplus$ acc, particularizat pe constructori
Combinare independentă:	(<+>) :: t a b b -> t a c c -> t a (b, c) (b, c)
Combinare semidependentă:	(>.>) :: t a b b -> t a (b, c) c -> t a (b, c) (b, c)
Combinare mutual dependentă:	(<.>) :: t a (b, c) b -> t a (b, c) c -> t a (b, c) (b, c)