Map, Filter, Fold

Inteligência na Web e Big Data

Fabricio Olivetti de França e Thiago Ferreira Covões
folivetti@ufabc.edu.br, thiago.covoes@ufabc.edu.br

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

Dados Particionados

Classe `Particao`

Para começarmos a entender a relação entre as funções que apresentaremos hoje com computação distribuída, vamos definir uma estrutura de dados que recebe uma lista de dados e particiona ela em diversos pedaços.

Classe Particao

```
1  class Particao:
2    def __init__(self, xs, parts):
3      self.parts = parts
4      n = math.ceil(len(xs)/parts)
5      self.chunks = [
6        [
7          xs[j] for j in range(i*n, i*n + n)
8                if j < len(xs)
9        ] for i in range(parts)
10     ]
```

Classe Particao

```
1  xs = [i for i in range(19)]
2  ps = Particao(xs, 4)
3  print(ps.chunks)
```

E o resultado será:

```
1  [[0, 1, 2, 3, 4], [5, 6, 7, 8, 9],
2   [10, 11, 12, 13, 14], [15, 16, 17, 18]]
```

Opoerações em Partições

Operações: transformar, filtrar, reduzir

Conforme mencionado na aula passada, três operações fundamentais que queremos fazer em nossos dados são:

- Transformação: aplica uma função $f : X \to Y$ em cada elemento.
- Filtragem: filtra os elementos utilizando uma função $f : X \to Bool$.
- Redução: aplica uma operação binária cumulativamente nos elementos de forma a gerar um resultado final do tipo X.

Transformação

A implementação da função de transformação, denominada `map`, para nossa estrutura de partição é a simples aplicação de uma função f em todos os elementos de nossa estrutura.

Transformação: map

```
def map(self, f):
    xs = [
      f(c)
      for cs in self.chunks
      for c in cs
    ]
    return Particao(xs, self.parts)
```

Transformação: `map`

Reparem que essa função gera uma cópia de nossa estrutura, só que transformada, mantendo a ideia de imutabilidade.

Transformação: `map`

Lembrando que essa função define um *Functor*, ela deve obedecer as duas leis:

```
1 particao = Particao([1,2,3,4], 4)
2 particao.map(lambda x: x)
3 > [[1],[2],[3],[4]]
```

Transformação: `map`

Fazer a sequência de dois `map` é equivalente a fazer um único `map` da composição das funções:

```
1  dobra = lambda x: 2*x
2  somaUm = lambda x: x+1
3
4  particao.map(dobra)
5          .map(somaUm)
6  > [[3], [5], [7], [9]]
7
8  particao.map(comp(dobra, somaUm)
9  > [[3], [5], [7], [9]]
```

Filtragem

A função de filtragem deve criar uma nova `Particao` de tal forma que contenha apenas os elementos que retornem verdadeiros para uma função de predicaod $p : X \to Bool$.

Filtragem: `filter`

Isso será implementado através da função `filter`:

```
1  def filter(self, f):
2      xs = [
3        c
4        for cs in self.chunks
5        for c in cs
6        if f(c)
7      ]
8      return Particao(xs, self.parts)
```

Da mesma forma, essa função retorna uma nova `Particao`, garantindo a imutabilidade.

Filtragem: leis

Podemos pensar em algumas leis para a filtragem, assim como na transformação:

- Ela deve retornar exatamente a mesma estrutura quando aplicado em um predicado que sempre retorna verdadeiro.
- A sequência de duas ou mais chamadas de `filter` é equivalente a aplicar o operador de conjunção (e-lógico) sequencialmente na saída das funções.

Filtragem: otimizações

Além disso podemos fazer funções auxiliares `filterMap` e `mapFilter` que recebem um predicado e uma função (ou o contrário) e aplica a sequência de `filter` e `map` em uma única passagem por nossa estrutura.

Filtragem: `filter`

```
1  ps = Particao([i for i in range(19)])
2  print(ps.filter(lambda x: x%2==0).chunks)
3  > [0, 2, 4], [6, 8, 10], [12, 14, 16], [18]]
```

Redução

O processo de redução consiste em uma função de dois argumentos (um operador) e um valor inicial (valor neutro).

Redução

Essa função é aplicada inicialmente entre o valor inicial e o primeiro elemento de nossa estrutura.

Em seguida, ela é aplicada repetidamente entre o resultado e o próximo elemento, até não haver mais elementos.

Redução: `fold`

Essa função é conhecida como `fold`, ou *dobra*, pois, pensando em listas, ela aplica o operador incrementalmente enquanto dobra essa lista, até chegar a um único elemento.

Redução: `fold`

A implementação natural dessa função é recursiva:

```haskell
fold :: (a -> b -> b) -> b -> [a] -> b
fold f z []     = z
fold f z (x:xs) = fold f (f x z) xs
```

Redução: `fold`

Por exemplo, dado o operador `(+)` e o valor neutro `0`, temos:

```
1  fold (+) 0 [1,2,3,4]
2  = fold (+) (1+0) [2,3,4]
3  = fold (+) (1+2) [3,4]
4  = fold (+) (3+3) [4]
5  = fold (+) (6+4) []
6  = 10
```

Redução: `fold`

Para linguagens que não otimizam a recursão, pode ser mais interessante a implementação iterativa:

```
1  def fold(self, f, z):
2      for cs in self.chunks:
3        for c in cs:
4          z = f(z, c)
5      return z
```

Redução: `fold`

```
1  ps = Partition(range(19), 4)
2  print(ps.fold(lambda x, y: x+y, 0))
3  > 171
```

Monoids

Monoid

Um tipo que possui um operador binário $\otimes$ e um elemento neutro ϵ tal que:

- $a \otimes \epsilon = \epsilon \otimes a = a$
- $a \otimes (b \otimes c) = (a \otimes b) \otimes c$

é denominado de Monoid.

Monoid e computação distribuída

As leis estabelecidas para um *Monoid* permite processarmos a função fold de forma distribuída:

```
1  def fold(self, f, z):
2      def foldlista(xs):
3          x0 = z
4          for x in xs:
5              x0 = f(x0, x)
6          return x0
7      return foldlista(
8                parmap(foldlista, self.chunks)
9              )
```

Monoid e computação distribuída

Note que para isso dar certo, devemos garantir que a sequência dos valores é mantida como da forma original.

Monoid Comutativo

Se além das leis anteriores também tivermos que $a \otimes b = b \otimes a$ dizemos que ele é um Monoid Comutativo e a ordem em que os dados são armazenados não importa para o processamento correto.

Exercícios

Quais tipos formam um Monoid? Mostre que eles obedecem as leis, verifique também a comutatividade.

Monoid: Lista Ordenada

Vamos criar um Monoid de uma lista ordenada:

```
z = []

def append(xs, ys):
  if len(xs) == 0 or len(ys)==0:
    return xs+ys
  if xs[0] < ys[0]:
    return [xs[0]] + append(xs[1:], ys)
  return [ys[0]] + append(xs, ys[1:])
```

Monoid: Insertion Sort

Se tivermos uma `Particao` de listas, podemos gerar uma lista ordenada fazendo:

```
1 listas.fold(append, [])
```

E cada partição faz parte do trabalho em paralelo.

Exercícios

Exercícios sobre `fold`

Utilize `fold` para determinar a quantidade de elementos em uma `Particao`.

Exercícios sobre `fold`

```
1  def count(x, y):
2    return x+1
3  print("Contagem: ", particao.fold(count,0))
4
5  > Contagem: 4
```

mas nossa partição tinha 19 elementos…

Exercícios sobre `fold`

Quando fazemos `foldlista(map(foldlista, self.chunks))`, o `fold` externo deveria fazer outra operação...

Combine

Vamos passar uma segunda função para `fold`, denominada `combine` que combinará os resultados do `fold`:

```
def combine(self, g, z):
    def combine(xs):
      x0 = z
      for x in xs:
        x0 = g(x0, x)
      return x0
    return combine([c for cs in self.chunks for c in
↪   cs])
```

Combine

E agora nosso `fold` fica:

```
1  def fold(self, f, z):
2      def foldlista(xs):
3          x0 = z
4          for x in xs:
5              x0 = f(x0, x)
6          return x0
7
8      return Particao([foldlista(c) for c in self.chunks],
   ↪   self.parts)
```

Combine

Agora podemos fazer:

```
1  def count(x, y):
2    return x+1
3  print("Contagem: ", particao.fold(count,0)
4                                .combine(soma,0))
5
6  > Contagem: 19
```

Exercícios sobre `fold`

Utilize `fold` para determinar o maior elemento em uma `Particao`. Em seguida, modifique para encontrar o menor.

Exercícios sobre `fold`

Utilize `fold` para retornar o maior e o menor elemento em uma `Particao`.

Exercícios sobre `fold`

Utilize `fold` para determinar se determinado elemento existe em uma `Particao`.

Exercícios sobre `fold`

Utilize `fold` para determinar se todos os elementos de uma `Particao` obedecem um certo predicado p.

Exercícios sobre `fold`

Utilize `fold` para determinar se pelo menos um elemento de uma `Particao` obedece um predicado p.

Exercícios sobre `fold`

Utilize `fold` para implementar `map`.

Exercícios sobre `fold`

Utilize `fold` para implementar `filter`.

Considerações Finais

Considerações Finais

Dada uma estrutura particionada temos três funções importantes:

- Transformação
- Filtragem
- Redução

Considerações Finais

Essas três funções possuem propriedades que permitem otimizações e paralelização do código.

Vimos também que podemos generalizar todas as funções apenas utilizando a função de redução.

Leitura Sugerida

An Algebra for Distributed Big Data Analytics