Exercícios sobre *fold*

Prof.: Carlos Camarão

17 de Abril de 2017

1 Fold

Funções de ordem superior como *map* e *filter* permitem a implementação de operações comuns em estruturas de dados: aplicar uma função a todos os elementos da estrutura de dados e filtrar os elementos que satisfazem a um predicado.

No entanto, estes são casos especiais de uma operação mais expressiva, chamada de *fold* (ou *reduce*), que permite obter um resultado pela aplicação de uma função binária aos elementos de uma estrutura de dados para obtenção de um resultado final, a partir de um valor inicial.

Vamos trabalhar nestas notas de aula com *fold*s sobre listas, para as quais podemos ter `foldr` ("fold right", i.e. `fold` sobre os elementos da lista da direita para a esquerda, em outras palavras, do último até o primeiro elemento da lista) e `foldl` ("fold left", i.e. `fold` dos elementos da lista da esquerda para a direita, em outras palavras, do primeiro até o último elemento).

1.1 *foldr*

Consideremos primeiro `foldr`. O tipo dessa função é:

```
(a -> b -> b) -> b -> [a] -> b}
```

`foldr f z` aplicado a uma lista `[`x_1`, ..., `x_n`]` fornece o resultado:

$$x_1 \grave{} f \grave{} (\ldots (x_{n-1} \grave{} f \grave{} (x_n \grave{} f \grave{} z))\ldots) \tag{1}$$

Em outras palavras, o resultado de `foldr f z` aplicado à lista x_1 `:(...(`x_n `:[]))` é obtido substituindo `[]` por `z` e `(:)` por `` `f` ``.

O valor de tipo `b` obtido em aplicações sucessivas de `f` em um *fold* é chamado de *acumulador*.

O resultado é calculado da direita para a esquerda, isto é, a primeira aplicação é `f` x_n `z`, depois `f` x_{n-1} r_1, sendo r_1 o resultado obtido por essa primeira aplicação, até `f` x_1 r_{n-1}, sendo esse o último e r_{n-1} o penúltimo resultado obtido com aplicações de f.

Exemplos

Exemplos básicos de uso de *foldr* são mostrados a seguir. Procure exercitar, definindo você mesmo, antes de ver a definição.

Para isso, pense o seguinte: *foldr* f z obtém um valor final aplicando f a partir do valor inicial z. É preciso definir f e z. Leve em conta que: f recebe um elemento da lista, o resultado (de tipo r) de fazer *foldr* no restante da lista, e retorna um valor (do mesmo tipo de r); z é o valor retornado quando a lista é vazia.

É útil também ter em mente como *foldr* computa seu resultado, como mostrado em (1).

1. soma de todos os elementos:

```
sum = foldr (+) 0
```

2. multiplicação de todos os elementos:

```
prod = foldr (*) 1
```

3. concatenação de todos os elementos (de uma lista de listas):

```
concat = foldr (++) []
```

4. conjunção de todos os elementos:

```
and = foldr (&&) True
```

5. disjunção de todos os elementos:

```
or = foldr (||) False
```

6. máximo dos elementos (de uma lista não vazia):

```
maximum (a:x) = foldr max a x
```

7. mínimo dos elementos (de uma lista não vazia):

```
minimum (a:x) = foldr min a x
```

8. comprimento (número de elementos):

```
len = foldr (\_ -> (+1)) 0
```

9. *map*:

```
map f = foldr (:) []
```

10. *filter*:

```
filter p = foldr (\a -> if p a then (a:) else id) []
```

Mais exemplos são mostrados a seguir.

1. teste de pertinência a lista:

```
elem a = foldr ((||) . (==a)) False
```

2. ordenação por inserção:

```
insertionSort = foldr ins []
  ins a []       = [a]
  ins a (b:x)
    | a <= b     = a:b:x
    | otherwise = b: ins a x
```

3. inserção de valor (`k`) entre dois elementos consecutivos de uma lista; na definição de `intersperse` mostrada abaixo, um valor booleano é usado para não inserir `k` após o último elemento:

```
intersperse k = fst . foldr consIfTrue ([],(k,False))
  where consIfTrue a (x,(k,False)) = (a:x  ,(k,True))
        consIfTrue a (x,(k,b    )) = (a:k:x,(k,b   ))
```

1.2 *foldl*

`foldl f z` aplicado a uma lista `[`x_1`, ..., `x_n`]` fornece o resultado:

$$\texttt{(...((z `f`}\ x_1\texttt{) `f`}\ x_2\texttt{) ...`f`}\ x_n\texttt{)}$$

O tipo de `foldl` é:

```
(b -> a -> b) -> b -> [a] -> b
```

A função `f` em `foldl f`, ao contrário de `foldr`, recebe o "acumulador" primeiro, e depois o elemento da lista.

fold é útil em casos (não muito comuns) em que a função (primeiro argumento) é não-estrito no segundo argumento, ou quando se deseja inverter a ordem dos elementos da lista.

Exemplos de uso de *foldl*:

1. inversão da ordem dos elementos da lista:

```
reverse = foldl (flip (:)) []
```

2. subtração de todos elementos de lista a valor:

```
subtraiTodosDe n = foldl (-) n
```

1.2.1 Ineficiência da complexidade de espaço da avaliação preguiçosa de *foldl*

Infelizmente, ao contrário da complexidade de tempo, a complexidade de espaço da estratégia de avaliação preguiçosa não é ótima. Por exemplo, a quantidade de espaço usada para obter o resultado de:

$$\texttt{foldl}\ f\ z\ x$$

é $O(n)$, onde n é o tamanho da lista, quando f é estrita. A complexidade de espaço usada para obter o resultado de `foldl` f z x é também $O(n)$ quando f é estrita. Se a lista for grande, isso poderá acarretar problema de falta de espaço suficiente para obter o resultado.

No entanto, no caso de *foldl*, a complexidade de espaço usada pela estratégia de avaliação estrita é $O(1)$. Isso ocorre, no cálculo de `foldl` f z x, sendo $x = \texttt{[}x_1\texttt{,...,}x_n\texttt{]}$, porque o resultado r_i de $f\ z\ x_i$ é calculado imediatamente, em vez de ser salvo para ser calculado posteriormente, no cálculo de $f\ r_1\ x_{i+1}$, para $i = 1, \ldots, n$, e isso evita que espaço de memória tenha que ser alocado a cada i de 1 até n.

No caso de *foldr*, a complexidade de espaço da avaliação preguiçosa não pode ser melhorada (de $O(n)$ para uma complexidade menor) porque a construção dos resultados tem que ser realizada a partir o primeiro elemento da lista e terminar com o último elemento da lista (e isso envolve um custo $O(n)$), de modo a permitir que os resultados sejam calculados do último elemento da lista até o primeiro.

1.3 *foldr* versus *foldl*

Quando analisamos a eficiência, em termos de tempo e espaço, de *foldr* e *foldl*, por exemplo para decidir quando usar uma ou outra função, podemos chegar a conclusões interessantes.

Para isso, consideremos *foldr* primeiro:

$$\begin{array}{lccccccccc} \texttt{foldr}\ f\ z\ (\ x_1 & : & (\ldots & : & (x_n & : & [\,]) & \ldots)) = \\ \quad x_1 & `f` & (\ldots & `f` & (x_n & `f` & z) & \ldots) \end{array}$$

Note:

1. Se $f\ e$ não for estrita (isto é, se o resultado da avaliação de e_1 for suficiente para estabelecer o resultado de $f\ e_1\ e_2$, ou seja, se a avaliação de e_2 puder não ser necessária para estabelecer o resultado de e_1 `` `f` `` e_2), a complexidade de tempo do cálculo de `(foldr f z x)` é sub-linear, pois o número de valores de x, a partir de x_1, necessários para que f estabeleça um resultado, será menor ou igual a n.

   Por exemplo, `foldr (&&) True [False..]` é igual a `False` (apesar de `[False..]` ser infinita), e o tempo necessário para avaliação é constante (esse é o tempo necessário para o cálculo de `False && l`, podendo `l` ser qualquer lista, finita ou infinita).

   O mesmo acontece para a complexidade de espaço.

2. Se, no entanto, se f e $f\ e$ forem estritas (isto é, se o resultado da avaliação de e_1 e de e_2 forem necessários para a avaliação de e_1 `` `f` `` e_2), a complexidade de tempo do cálculo de `foldr` $f\ z\ x$ (onde x é igual a `(`x_1 `:` `(...:` `(`x_n `:` `[]) ...)`) é $O(n)$, onde n é o tamanho de x.

   O mesmo acontece para a complexidade de espaço.

3. No caso de *foldl*, a função $f\ z\ e$ deve ser não-estrita no segundo argumento (e): i.e. o resultado de $f\ z\ e$ deve poder ser obtido sem necessidade de avalizar z). Lembre-se:

$$foldl\ f\ z\ (x_1 : (\ldots : (x_n : [\,]) \ldots)) = f(\ldots(f(f\ z\ x_1)\ x_2)\ \ldots\ x_n)$$

   Por exemplo, a avaliação de `foldl (&&) True [False..]` não termina, porque `(&&)` é não-estrita no primeiro argumento (faz casamento de padrão no primeiro argumento) e não no segundo. Ou seja, é preciso o resultado de cada conjunção precisa ser obtido para obtençao da próxima conjunção (o que requer o processamento de uma lista infinita de valores iguais a *False*.

2 Exercícios

1. Escreva, usando *foldl* ou *foldr* uma função que recebe uma lista de cadeias de caracteres (valores do tipo *String*) e retorna uma cadeia de caracteres que contém os 3 primeiros caracteres de cada cadeia.

   Por exemplo, ao receber `["Abcde", "1Abcde", "12Abcde", "123Abcde"]` deve retornar `"Abc1Ab12A123"`.

2. Escreva, usando *foldr* ou *foldl*, uma função que recebe uma lista de itens (valores de tipo *Item*, veja definição a seguir) e retorna a soma das idades (valores do campo Idade) de todos os elementos da lista.

```
data Item  = Pessoa Nome Idade RG
type Nome  = String
type Idade = Integer
type RG    = String
```

3. Escreva, usando *foldr* ou *foldl*, uma função que recebe uma lista de valores de tipo Item, da definição acima, e retorna o nome da pessoa mais nova da lista.

4. Escreva, usando *foldl* ou *foldr*, uma função que recebe uma lista de cadeias de caracteres (valores do tipo *String*) e retorna uma cadeia de caracteres que contém os 3 primeiros caracteres de cada cadeia removidos se não forem letras, ou com as letras em caixa alta se forem letras, e com os demais caracteres depois dos 3 primeiros sem alteração.

   Por exemplo, ao receber `["Abcde", "1Abcde", "12Abcde", "123Abcde"]` deve retornar `"ABCdeABcdeAbcdeAbcde"`.

5. Explique porque *foldr* $f\ x$ pode não percorrer toda a lista x, ao passo que toda a lista x é sempre percorrida, no caso de foldl.

6. . A função *remdups* remove elementos iguais adjacentes de uma lista, conservando só um dos elementos.

   Por exemplo, `remdups [1,2,2,3,3,3,1,1] = [1,2,3,1]`.

   Defina *remdups* usando *foldr* ou *foldl*.