Aula 1: *folds* sobre listas, árvores e naturais.

Alcino Cunha

21 de Setembro de 2006

1 Introdução

Considere as seguintes funções definidas recursivamente sobre listas. Dada uma lista, a primeira calcula o seu somatório e a segunda inverte-a.

$$
\begin{aligned}
&sum :: [Int] \to Int \\
&sum\ [\,] \quad\ = 0 \\
&sum\ (h : t) = h + sum\ t \\
&reverse :: [a] \to [a] \\
&reverse\ [\,] \quad\ = [\,] \\
&reverse\ (h : t) = reverse\ t +\!\!+ [h]
\end{aligned}
$$

Facilmente se verifica que ambas partilham um *padrão de recursividade* muito típico: quando a lista não é vazia, começam por invocar recursivamente a função sobre a cauda da lista, sendo este resultado posteriormente combinado com a cabeça da lista por forma a produzir o resultado final. Esta semelhança é ainda mais evidente se isolarmos a função que processa o resultado recursivo e a cabeça da lista usando a notação lambda do Haskell.

$$
\begin{aligned}
&sum :: [Int] \to Int \\
&sum\ [\,] \quad\ = 0 \\
&sum\ (h : t) = (\lambda x\ y \to x + y)\ h\ (sum\ t) \\
&reverse :: [a] \to [a] \\
&reverse\ [\,] \quad\ = [\,] \\
&reverse\ (h : t) = (\lambda x\ y \to y +\!\!+ [x])\ h\ (reverse\ t)
\end{aligned}
$$

2 O meu primeiro fold

Agora é evidente que a única diferença entre estas funções, para além do nome, é a função que processa o resultado recursivo e o valor devolvido quando a lista está vazia. Podemos então escrever uma função de ordem superior que recebe estes dois parâmetros e implementa este padrão de recusividade. Esta função chama-se *foldr* e já existe predefinida em Haskell.

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{foldr}\ f\ z\ [\,] \quad\ = z \\
&\mathit{foldr}\ f\ z\ (h : t) = f\ h\ (\mathit{foldr}\ f\ z\ t)
\end{aligned}
$$

Usando o *foldr* podemos agora redefinir as funções anteriores sem mencionar explicitamente a invocação recursiva.

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum}\ l = \mathit{foldr}\ (\lambda x\ y \to x + y)\ 0\ l \\
&\mathit{reverse} :: [a] \to [a] \\
&\mathit{reverse}\ l = \mathit{foldr}\ (\lambda x\ y \to y +\!\!+ [x])\ [\,]\ l
\end{aligned}
$$

Dadas duas funções f e g, se $\forall x.fx = gx$ então $f = g$. Esta lei chama-se *igualdade extensional* e permite simplifcar as definições acima, pois as equações que as definem são obviamente válidas para qualquer l. Por exemplo, a função *sum* pode definir-se simplesmente da seguinte forma.

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum} = \mathit{foldr}\ (\lambda x\ y \to x + y)\ 0
\end{aligned}
$$

Se passarmos o operador binário de soma para a notação prefixa é possível também retirar as variáveis do primeiro argumento deste *foldr*.

$$
\begin{aligned}
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum} = \mathit{foldr}\ (+)\ 0
\end{aligned}
$$

Exercício 1 *Escreva a seguinte função usando o foldr. Neste caso a função que processa o resultado recursivo deverá ignorar a cabeça da lista.*

$$
\begin{aligned}
&\mathit{length} :: [a] \to \mathit{Int} \\
&\mathit{length}\ [\,] \quad\ = 0 \\
&\mathit{length}\ (h : t) = 1 + \mathit{length}\ t
\end{aligned}
$$

Exercício 2 *Escreva a seguinte função que concatena duas listas unsando o foldr. Note que a recursividade é feita sobre o primeiro argumento, enquanto que o segundo se comporta como uma constante. Será possível neste caso escrever uma definição sem argumentos, tal como foi feito acima para a função sum?*

$$
\begin{aligned}
&\mathit{cat} :: [a] \to [a] \to [a] \\
&\mathit{cat}\ [\,] \quad\ l = l \\
&\mathit{cat}\ (h : t)\ l = h : \mathit{cat}\ t\ l
\end{aligned}
$$

Como se pode facilmente verificar, dada uma lista qualquer, a função $\mathit{foldr}\ f\ z$ substitui os contrutores (:) e [ ] pelos argumentos f e z, respectivamente.

$$\mathit{foldr}\ f\ z\ [x1, x2, ..., xn] = \mathit{foldr}\ f\ z\ (x1 : x2 : ... : xn : [\,]) = f\ x1\ (f\ x2\ ...\ (f\ xn\ z))$$

No caso de os argumentos serem outra vez os construtores das listas, o *foldr* resultante comporta-se como a função identidade.

$$foldr\ (:)\ [\,] = id$$

Esta lei, denominada *reflexão*, é o primeiro exemplo de uma lei de cálculo que nos permite raciocinar e optimizar programas funcionais. Neste caso, a elegância da sua formulação foi conseguida à custa da utilização do padrão de recursividade *foldr* e da notação *point-free* (ou seja, sem variáveis). Estes dois princípios irão acompanhar-nos ao longo do semestre: sempre que quisermos calcular com uma dada função recursiva, será necessário convertê-la primeira para uma função equivalente sem recursividade explícita (usando padrões de recusividade) e sem variáveis.

3 Folds sobre árvores

Considere agora o seguinte tipo de dados para codificar árvores binárias e duas funções para determinar, respectivamente, a altura e a travessia inorder de uma árvore.

$$\begin{aligned}
&\textbf{data}\ Tree\ a = Empty \mid Node\ a\ (Tree\ a)\ (Tree\ a)\\
&height :: Tree\ a \to Int\\
&height\ Empty\ \ \ \ \ \ \ \ = 0\\
&height\ (Node\ _\ l\ r) = 1 + max\ (height\ l)\ (height\ r)\\
&inorder :: Tree\ a \to [\,a\,]\\
&inorder\ Empty\ \ \ \ \ \ \ \ = [\,]\\
&inorder\ (Node\ x\ l\ r) = (inorder\ l) +\!\!+ [\,x\,] +\!\!+ (inorder\ r)
\end{aligned}$$

Mais uma vez se verifica que ambas as funções partilham um padrão de recursividade semelhante ao do *foldr*: quando a árvore não é vazia, a função é invocada recursivamente sobre ambas as sub-árvores, sendo os dois resultados posteriormente combinados com o conteúdo do nodo por forma a produzir o resultado final. Podemos então escrever uma função de ordem superior que codifica este padrão de recursividade.

$$\begin{aligned}
&foldt :: (a \to b \to b \to b) \to b \to Tree\ a \to b\\
&foldt\ f\ z\ Empty\ \ \ \ \ \ \ \ = z\\
&foldt\ f\ z\ (Node\ x\ l\ r) = f\ x\ (foldt\ f\ z\ l)\ (foldt\ f\ z\ r)
\end{aligned}$$

Exercício 3 *Qual será a lei de reflexão para o foldt?*

Usando o *foldt* podemos então reescrever a função *height* sem usar recursividade explícita.

$$\begin{aligned}
&height :: Tree\ a \to Int\\
&height = foldt\ (\lambda_\ l\ r \to 1 + max\ l\ r)\ 0
\end{aligned}$$

Exercício 4 *Codifique a função inorder usando o foldt.*

Exercício 5 *Considere o seguinte tipo de dados para codificar números naturais.*

$$\textbf{data}\ Nat = Zero \mid Succ\ Nat$$

Implemente a função foldn que codifica um padrão de recursividade típico para este tipo semelhante ao foldr para as listas. Escreva a sua lei de reflexão.

Exercício 6 *Implemente as seguintes funções usando o foldn.*

$$\begin{aligned}
&toint :: Nat \to Int\\
&toint\ Zero = 0\\
&toint\ (Succ\ n) = 1 + toint\ n\\
&soma :: Nat \to Nat \to Nat\\
&soma\ n\ Zero = n\\
&soma\ n\ (Succ\ m) = Succ\ (soma\ n\ m)
\end{aligned}$$

4 Infelizmente nem tudo são folds

Nem todas as funções recursivas podem ser implementadas usando folds. No caso das listas, para que uma função possa ser definida usando o *foldr* é necessário que se verifiquem as seguintes condições:

1. O caso de paragem deve ser na lista vazia.
2. A invocação recursiva tem que ser feita sobre a cauda da lista.
3. O resultado final só pode depender do resultado recursivo e da cabeça da lista.

Exercício 7 *Será possível implementar a seguinte função com um foldr? Justifique a sua resposta.*

$$\begin{aligned}
&qsort :: Ord\ a \Rightarrow [a] \to [a]\\
&qsort\ [\,] \quad\ = [\,]\\
&qsort\ (h : t) = (qsort\ menores) +\!\!+ [h] +\!\!+ (qsort\ maiores)\\
&\quad \textbf{where} \quad menores = filter\ (<h)\ t\\
&\qquad\qquad\ \ maiores = filter\ (\geqslant h)\ t
\end{aligned}$$

Exercício 8 *Será possível implementar a seguinte função com um foldn? Justifique a sua resposta.*

$$\begin{aligned}
&downto :: Nat \to [Nat]\\
&downto\ Zero \quad\ = [\,]\\
&downto\ (Succ\ n) = Succ\ n : downto\ n
\end{aligned}$$