UFABC
Paradigmas de Linguagens de Programação
Semigrupos e Monóides

Emílio de Camargo Francesquini
Mario Leston Rey

3 de agosto de 2023

Um *semigrupo* é um par $(a, \diamond)$ em que a é um tipo e $\diamond :: a \to a \to a$ satisfaz:

$$(x \diamond y) \diamond z = x \diamond (y \diamond z)$$

para cada $x, y, z :: a$; neste caso dizemos que $\diamond$ é *associativo*. Um semigrupo é uma estrutura algébrica bem simples. No entanto, o que ela tem de simples também tem de importante (dizer que algo é importante é extremamente subjetivo, mas, afinal de contas, precisamos de uma motivação). Eis alguns exemplos de semigrupos:

(i) para cada a, se `Num` a, então $(a, +)$ é um semigrupo,

(ii) para cada a, se `Num` a, então $(a, *)$ é um semigrupo, e

(iii) para cada a, $([a], ++)$ é um semigrupo.

Há diversos outros exemplos — matrizes, listas ordenadas, aritmética modular, endomorfismos — de semigrupos que vamos explorar nesta aula.

O conjunto das *expressões formais* de um semigrupo $(a, \diamond)$ é definido recursivamente da seguinte forma:

(i) x é uma expressão para cada $x :: a$, e

(ii) se α e β são expressões, então $(\alpha \diamond \beta)$ também é uma expressão.

Assim, por exemplo, se $x_1, x_2, x_3, x_4 :: a$, então $((x_1 \diamond (x_2 \diamond x_3)) \diamond x_4)$ é uma expressão formal. A cada expressão formal está associado um único valor do tipo a. Por exemplo, a expressão formal $((2 + 3) + 4)$ tem valor 9 no semigrupo $(\texttt{Int}, +)$. Dada uma expressão formal α, a lista de *valores* de α é definida recursivamente como:

(i) para cada $x :: a$, $[x]$ é a lista de valores de a,

(ii) se α e β são expressões e xs é a lista de valores de α e ys é a lista de valores de β, então $xs ++ ys$ é a lista de valores de $(\alpha \diamond \beta)$.

Por exemplo, a lista de valores da expressão $((x_1 \diamond x_2) \diamond (x_3 \diamond x_4))$ é $[x_1, x_2, x_3, x_4]$. A associatividade de $\diamond$ permite concluir que se α e β são expressões formais e a lista

de valores de α é igual a lista de valores de β então α e β possuem o mesmo valor. Logo, o valor de uma expressão formal qualquer cuja lista de valores é $[x_1, x_2, \ldots, x_n]$ é igual ao valor da expressão formal $(x_1 \diamond (x_2 \diamond (x_3 \diamond (\cdots \diamond x_n) \ldots)$.

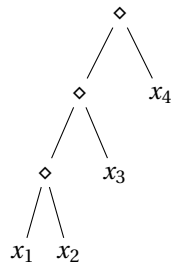
O primeiro problema que vamos resolver, até para apreciar com mais cuidado o papel da associatividade, é obter todas as expressões formais de uma lista de valores. Para isso, vamos precisar primeiro da seguinte definição. Suponha que *xs* é uma lista não-vazia de valores. Uma *bipartição* de *xs* é é um par (*ys*, *zs*) de listas não-vazias tais que $ys ++ zs = xs$. Por exemplo, $([1,2],[3,4])$ é uma bipartição de $[1,2,3,4]$.

Exercício 1. Escreva uma função

```
allBips :: [a] -> [([a], [a])]
```

que recebe uma lista *xs* e devolve todas as bipartições de *xs*.

É claro quodemos representar uma expressão formal como uma árvore binária cujas folhas estão rotuladas com valores e cujos nós internos estão (implicitamente rotulados com $\diamond$. Por exemplo, a seguinte árvore representa a expressão formal $(((x_1 \diamond x_2) \diamond x_3) \diamond x_4))$; neste caso, dizemos que uma tal árvore é uma árvore *para* a lista de valores $[x_1, x_2, x_3, x_4]$.



Observe que a cada expressão formal está associada uma única árvore binária; além disso, a cada árvore binária está associada uma única expressão formal.

Considere, então, a definição

```
data BTPar a = Leaf a | Node (BTPar a) (BTPar a)
```

Assim, a árvore desenhada acima é representada pelo termo

```
Node (Node (Node (Leaf x1) (Leaf x2)) (Leaf x3)) (Leaf x4)
```

O próximo problema consiste em, dada uma lista de valores *vs*, determinar todas as árvores para *vs*.

Exercício 2. Escreva, então, uma função

```
allTrees :: [a] -> [BTPar a]
```

que recebe uma lista *vs* de valores e devolve uma lista com as árvores para *vs*. Por exemplo, a chamada

```
allTrees [1,2,3,4]
```

deve devolver

```
[Node (Leaf 1) (Node (Leaf 2) (Node (Leaf 3) (Leaf 4))),
 Node (Leaf 1) (Node (Node (Leaf 2) (Leaf 3)) (Leaf 4)),
 Node (Node (Leaf 1) (Leaf 2)) (Node (Leaf 3) (Leaf 4)),
 Node (Node (Leaf 1) (Node (Leaf 2) (Leaf 3))) (Leaf 4),
 Node (Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)) (Leaf 4)]
```

Agora, vamos definir uma `type class` para um semigrupo:

```
class Semigroup a where
    (<>) :: a -> a -> a
    stimes :: Integral b => b -> a -> a
    sconcat :: NonEmpty a -> a
```

Há um novo elemento na definição acima, o tipo `NonEmpty a`. Antes de lidar com ele, vamos primeiro entender o papel da função `stimes`. Considere um semigrupo $(a : \diamond)$. É natural considerar uma expressão da forma

$$\underbrace{x \diamond x \diamond \cdots \diamond x}_{n \text{ vezes}},$$

onde $x :: a$ e $n \geqslant 1$. Uma definição recursiva é a seguinte:

$$x^n = \begin{cases} x, & \text{se } n = 1 \\ x \diamond x^{n-1}, & \text{caso contrário} \end{cases}$$

para cada n inteiro tal que $n \geqslant 1$. O papel da função `stimes` é justamente esse: dado $n :: b$ tal que `Integral` b e $x :: a$, então a chamada

```
stimes n x
```

devolve x^n.

Exercício 3. Escreva a função `stimes`.

Vamos, agora, implementar uma versão alternativa da função `stimes` baseada na seguinte ideia:

$$x^n = \begin{cases} x, & \text{se n} = 1 \\ x^{n/2} \diamond x^{n/2}, & \text{se } n \text{ é par} \\ x \diamond x^{n-1}, & \text{caso contrário} \end{cases}$$

para cada n inteiro tal que $n \geqslant 1$. Se implementada da maneira correta, em alguns casos, esta função, dependendo da versão concreta do operador $\diamond$, permite calcular x^n de uma forma mais eficiente (em $O(\lg n)$, lembre-se das aulas de Análise de Algoritmos).

Exercício 4. Implemente `stimes` de acordo com a descrição acima.

Vamos, agora, lidar com a função `sconcat`. A função `sconcat` recebe uma lista de valores $[x_1, x_2, \ldots, x_n]$ com $n \geqslant 1$ e devolve o valor da expressão $x_1 \diamond x_2 \diamond \cdots \diamond x_n$. Note que, em virtude da associatividade de $\diamond$ é desnecessário especificar uma expressão formal. Observe também que uma tal operação só está definida se $n \geqslant 1$ (ou seja, somente para listas não-vazias). Isto motiva a seguinte definição:

```haskell
data NonEmpty a = a :| [a]
infixr 5 :|
```

cujo propósito é representar listas não vazias. Vamos, em primeiro lugar, fazer com que `NonEmpty` seja uma instância de um `Functor`.

```haskell
class Functor f where
    (<$>), fmap :: (a -> b) -> f a -> f b
    (<$>) = fmap
```

Exercício 5. Defina, então, `NonEmpty` como uma instância de `Functor`.

Exercício 6. Escreva a função `sconcat`.

Podemos, de forma similar, ao que fizemos com a função `stimes` tentar melhorar o desempenho de `sconcat` para algumas instâncias de `Semigroup`. Para isso, considere uma lista $[x_1, x_2, \ldots, x_n]$ (com $n \geqslant 1$). Vamos transformar esta lista numa lista com $\lceil n/2 \rceil$ elementos aplicando a operação $\diamond$ a pares de elementos consecutivos:[1]

$$[x_1 \diamond x_2, x_3 \diamond x_4, \ldots, x_{n-1} \diamond x_n]$$

Exercício 7. Escreva a função

```haskell
pairing :: (a -> a -> a) -> [a] -> [a]
```

que realiza a operação acima descrita. Assim, se a é uma instância de um `Semigroup` e $xs :: [a]$, então a chamada

```haskell
pairing (<>) xs
```

produz uma lista como a descrita acima.

Exercício 8. Escreva, agora, uma versão alternativa da função `sconcat` usando a função `pairing`. Quando a função `sconcat` é submetida a uma lista com um único elemento, então ela devolve tal elemento. Se a lista tem mais de um elemento, então reduz a lista original usando a função `pairing` e realiza, recursivamente, o `sconcat` da lista reduzida.

Vamos implementar algumas instâncias de um `Semigroup`. Considere os seguintes tipos:

```haskell
newtype Sum a = Sum a deriving Show
newtype Prod a = Prod a deriving Show
```

Vamos usar o primeiro para definir o semigrupo $(a, +)$ e, o segundo, para definir o semigrupo $(a, *)$ tal que `Num` a.

Exercício 9. Derive estas duas instâncias de `Semigroup`.

É evidente que a operação de concatenação de listas é associativa. Logo, $([a], ++)$ é também um semigrupo para qualquer tipo a. É claro também que o tipo das listas não vazias também forma um semigrupo em relação à operação de concatenação de listas não-vazias.

[1] É claro, n ímpar implica que o último elemento é x_n.

Exercício 10. Derive uma instância de `Semigroup` para `NonEmpty` a.

Há diversas outras instãncias interessantes de `Semigroup`. Um dos hábitos que você deve adquirir sempre que definir um novo ADT é verificar se ele possui propriedades que permitam derivá-lo como uma instância de alguma `type class`.

Vamos, agora, lidar com *monóides*, que nada mais são que um tipo especial de semigrupo. Para um tipo a e $\diamond : a \to a \to a$, dizemos que $(a, \diamond)$ é um *monóide* se $(a, \diamond)$ é um semigrupo e existe $e :: a$ tal que

$$e \diamond x = x \diamond e = x$$

para cada $x :: a$; um tal e é chamdo de *identidade* do monóide. Eis a definição de uma `type class` para um monóide:[2]

```
class Semigroup a => Monoid a where
    mempty :: a
    mappend :: a -> a -> a
    mappend = (<>)
    mtimes :: Integral b => b -> a -> a
    mconcat :: [a] -> a
```

O papel de e em uma instância de um `Monoid` é feito por `mempty`.[3]

Exercício 11. Implemente a função `mtimes`, que agora pode receber 0, e a função `mconcat`, que agora pode receber a lista vazia. Use, para isso, as funções `stimes` e `sconcat`.

Exercício 12. Derive as respectivas instâncias de um monóide para `Sum` a e `Prod` a; é claro, a deve satisfazer `Num` a.

Exercício 13. Derive uma instância de um monoíde para listas.

Mergesort. Considere o conjunto das listas finitas e ordenadas, representado pelo seguinte ADT:

```
newtype Sorted a = Sorted [a]
```

A intercalação de duas listas ordenadas produz uma lista que é um rearranjo ordenado da concatenação das duas listas. Não é difícil se convencer que esta operação é associativa. Ademais, possui uma identidade (que é a lista vazia). Eis, para quem não se lembra, a função `merge` que intercala duas listas ordenadas:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) | x <= y    = x : merge xs (y:ys)
                    | otherwise = y : merge (x:xs) ys
```

Exercício 14. Derive `Sorted` a como uma instância de `Semigroup` e, a seguir, como uma instância de `Monoid`.

[2] A função `mtimes` não está presente na definição de um monóide do `Prelude`.

[3] O nome `mempty` não é muito sugestivo, assim como `mappend` e `mconcat`. A motivação para esses nomes vem do fato de que o conjunto das listas finitas munidas da operação de concatenação forma um monóide.

Exercício 15. Escreva uma versão do *mergesort* usando as definições anteriores. É verdade que esta versão tem consumo de tempo em $O(n \lg n)$ para uma lista de comprimento n?

Números de Fibonacci. Vamos lidar agora com números de Fibonacci e certificar que a implementação sugerida permite obter um algoritmo, cujo consumo de tempo está em $O(\lg n)$, que determina o n-ésimo número de Fibonacci. Eis a definição, caso você não se lembre,

$$f(n) = \begin{cases} n, & \text{se } n \leqslant 1 \\ f(n-1) + f(n-2), & \text{caso contrário} \end{cases}$$

para cada natural n. Para ver como isso é feito, suponha que para algum natural $n \geqslant 1$, temos a matriz

$$\begin{bmatrix} f(n+1) & f(n) \\ f(n) & f(n-1) \end{bmatrix}.$$

Então

$$\begin{bmatrix} f(n+1) & f(n) \\ f(n) & f(n-1) \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} f(n+2) & f(n+1) \\ f(n+1) & f(n) \end{bmatrix}.$$

Note também que

$$\begin{bmatrix} f(2) & f(1) \\ f(1) & f(0) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Logo,

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n+1} = \begin{bmatrix} f(n+2) & f(n+1) \\ f(n+1) & f(n) \end{bmatrix}$$

para cada natural n.

O seguinte ADT será usado para representar as matrizes 2×2 necessárias para implementar a ideia destacada acima:

```
newtype Fib = Fib [[Integer]]
```

Exercício 16. Derive `Fib` como uma instância de `Semigroup` e `Monid`. Note que o seu código deve garantir que as matrizes envolvidadas são 2×2. Finalmente, escreva uma função

```
fib :: Integral a => Int -> a
```

que devolve o n-ésimo número de Fibonacci.