Programação Funcional 17ª Aula — Um resolutor de Sudoku

Sandra Alves
DCC/FCUP

2018/19

O jogo Sudoku

Um puzzle Sudoku é uma grelha de 9×9, formado por 9 caixas de 3×3.

O objectivo do jogo é preencher as células com dígitos de 1 a 9, sem repetir valores nas linhas colunas e caixas.

Um mesmo puzzle poderá ter várias soluções, embora um puzzle mais desafiante terá apenas uma.

Vamos construir um programa para resolver puzzles Sudoku.

Baseado na solução de R. Bird do livro "Thinking Functionally with Haskell", Cambridge University Press, 2015.

Representação

Começamos por definir um tipo de dados apropriado para representar um puzzle Sudoku

```
type Matrix a = [Row a]
type Row a = [a]
```

Esta declaração de tipo enfatiza o facto de uma matriz $n \times m$ ser constituída por n linhas.

Uma grelha Sudoku é uma matriz 9×9 de dígitos:

```
type Grid = Matrix Digit
type Digit = Char
```

Os dígitos válidos vão de '1' a '9':

```
digits :: [Char]
digits = ['1'..'9']
```

O tipo `Char` faz parte da classe `Enum`, logo `['1'..'9']` define correctamente a lista de dígitos.

Vamos usar o dígito '0' para representar células vazias na grelha.

```
blank :: Digit -> Bool
blank = (== '0')
```

Exemplo

		4			5	7		
					9	4		
3	6							8
7	2			6				
			4		2			
				8			9	3
4							5	6
		5	3					
		6	1			9		

O puzzle da grelha acima corresponde à seguinte matriz:

```
["004005700",
 "000009400",
 "360000008",
 "720060000",
 "000402000",
 "000080093",
 "400000056",
 "005300000",
 "006100900"]
```

A função `solve`

Começamos por considerar uma solução para o problema em que dada uma grelha inicial consideramos todas as hipóteses para preencher as posições vazias. O resultado será uma lista de grelhas preenchidas de onde filtramos as que não satisfazem as condições de validade de um puzzle (dígitos repetidos em linhas, colunas ou caixas). A nossa função principal será:

```
solve :: Grid -> [Grid]
solve = filter valid . completions
```

Onde os tipos das funções secundárias são:

```
completions :: Grid -> [Grid]
valid :: Grid -> Bool
```

A função `completions`

Vamos definir a função `completions` considerando uma lista possível de escolhas e aplicando essa lista à matriz.

```
completions = expand . choices
```

onde

```
expand ::  Matrix [Digit] -> [Grid]
choices :: Grid -> Matrix [Digit]
```

A função `choices` calcula a lista de possíveis dígitos para cada célula:

```
choices = map (map choice)
    where choice d = if blank d then digits else [d]
```

Tendo a matriz de todas as escolhas possíveis, vamos expandir essa matriz para obtermos a lista de todas as matrizes possíveis. Por exemplo, se pensarmos no problema para uma lista de tamanho 3:

$$[[1,2,3],[2],[1,3]]$$

queremos obter:

$$[[1,2,1],[1,2,3],[2,2,1],[2,2,3],[3,2,1],[3,2,3]]$$

Ou seja, o produto cartesiano. Assumindo que temos:

$$\mathtt{cp}[[2],[1,3]] = [[2,1],[2,3]]$$

Como expandimos esta definição para:

$$\mathtt{cp}[1,2,3] : [[2],[1,3]]?$$

```
cp :: [[a]] -> [[a]]
cp []       = [[]]
cp (xs:xss) = [x:ys | x <- xs, ys <- yss]
   where yss = cp xss
```

Semelhante à definição da função `bits` que definimos para o verificador de tautologias.

Podemos agora definir a função `expand`:

```
expand :: Matrix [Digit] -> Grid
expand = cp . map cp
```

Note que

```
cp . map cp :: [[[a]]] -> [[[a]]]
```

A função `valid`

Vamos agora definir a função `valid`

```
valid :: Grid -> Bool
valid g = all nodups (rows g) &&
          all nodups (cols g) &&
          all nodups (boxs g)
```

A função `all` está definida no Prelude (`all p = and .  map p`). Definimos agora a função `nodups`

```
nodups :: Eq a => [a] -> Bool
nodups []     = True
nodups (x:xs) = all (/=x) xs && nodups xs
```

Uma versão mais eficiente poderia ordenar a lista antes de remover os duplicados, mas para listas de tamanho 9 o ganho em eficiência não é claro.

Precisamos agora de funções apropriadas para calcular as linhas, colunas e caixas

```
rows:: Matrix a -> Matrix a
rows = id
```

As colunas correspondem à matriz transporta. Vamos assumir que temos números positivos de linhas e colunas.

```
cols :: Matrix a -> Matrix a
cols [xs]     = [[x] | x <- xs]
cols (xs:xss) = zipWith (:) xs (cols xss)
```

A função `boxs` é mais complicada.

Vamos começar por considerar uma função `group` que agrupa uma lista em grupos de 3:

```
group :: [a] -> [[a]]
group [] = []
group xs = take 3 xs: group (drop 3 xs)
```

Consideramos uma função `ungroup` the dada uma lista agrupada, desagrupa a lista:

```
ungroup :: [[a]] -> [a]
ungroup = concat
```

Agora definimos `boxs` da seguinte forma:

```
boxs :: Matrix a -> Matrix a
boxs = map ungroup . ungroup .
       map cols .
       group . map group
```

Propriedades dos puzzles Sudoku

```
rows . rows = id
cols . cols = id
boxs . boxs = id
```

Duas destas propriedades são demonstradas tivialmente usando raciocínio equacional. Quais? A propriedade acerca das caixas é demonstrada facilmente considerando que:

```
group . ungroup = id
ungroup . group = id
```

E considerando também as leis da função `map` e o facto de `id` ser o elemento neutro da composição. É só fazer as contas :-)

Mais algumas propriedades válidas em puzzles Sudoku:

```
map rows . expand = expand . rows
map cols . expand = expand . cols
map boxs . expand = expand . boxs
```

Estas propriedades vão ser úteis mais tarde. E consideramos ainda duas propriedades envolvendo a função `cp`:

```
map (map f) . cp = cp . map (map f)
filter (all p) . cp = cp . map (filter p)
```

A segunda equação sugere uma implementação alternativa da função `cp`.

Uma implementação muito pouco eficiente

Recordemos a definição de `solve`:

```
solve :: Grid -> [Grid]
solve =  filter valid . expand . choices
```

Esta implementação é impraticável. Se consideramos que temos apenas 40 das 81 posições inicialmente preenchidas significa que teremos que testar 9^{41} grelhas, ou seja:

$$133027946472911330984474889185744967840 9$$

Vamos considerar uma versão mais eficiente, onde removemos escolhas para c que já ocorrem na linha, coluna ou caixa de c.

```
prune :: Matrix [Digit] -> Matrix [Digit]
```

que satisfaça a seguinte equação:

```
filter valid . expand = filter valid . expand . prune
```

A função `prune`

Vamos começar por eliminar alternativas ao nível das linhas, considerando as escolhas fixas.

```
pruneRow :: Row [Digit] -> Row [Digit]
pruneRow row = map (remove fixed) row
   where fixed = [d | [d] <- row]
```

A função `remove`, remove as escolhas fixas de qualquer escolha não fixa:

```
remove :: [Digit] -> [Digit] -> [Digit]
remove ds [x] = [x]
remove ds xs  = filter (`notElem` ds) xs
```

Usamos a função do Prelude `notElem` de forma infixa para definirmos a função `\x -> notElem x ds`.

A função `pruneRow` satisfaz a seguinte equação:

```
filter nodups . cp = filter nodups . cp . pruneRow
```

Ou seja, `pruneRow` não elimina alternativas que não contenham duplicados. Antes de tentarmos sintetizar a definição de prune, vamos considerar as seguintes leis, para funções `f` tais que `f . f = id`:

```
filter (p . f) = map f . filter p . map f
filter (p . f) . map f = map f . filter p
```

A segunda é uma consequência da primeira e a primeira sucede de `filter p . map f = map f . filter (p . f))` Demonstrar!!!

```
filter valid . expand
 = filter (all nodups . rows) .
   filter (all nodups . cols) .
   filter (all nodups . boxs) . expand
```

Vamos considerar a interação entre as diferentes aplicações de `filter` e `expand`:

```
    filter (all nodups.boxs).expand
=   considerando que boxs.boxs = id
    map boxs.filter (all nodups).map boxs.expand
=   considerando que map boxs.expand = expand.boxs
    map boxs.filter (all nodups).expand.boxs
=   definição de expand
    map boxs.filter (all nodups).cp.map cp.boxs
=   uma vez que filter (all p).cp = cp.map(filter p)
    map boxs.cp.map filter (nodups).map cp.boxs
=   lei do functor map
    map boxs.cp.map (filter nodups.cp).boxs
```

Usamos agora a propriedade:

```
filter nodups . cp = filter  nodups . cp . pruneRow
```

e obtermos a expressão final

```
map boxs . cp . map (filter nodups . cp . pruneRow) . boxs
```

Prosseguimos agora na direção contrária

```
    map boxs.cp.map (filter nodups.cp.pruneRow).boxs
=   map boxs.cp.map (filter nodups).map (cp.pruneRow).boxs
=   usando cp. map (filter p) = filter (all p).cp
    map boxs.filter (all nodups).cp.map (cp.pruneRow).boxs
=   map boxs.filter (all nodups).cp.map cp.map pruneRow.boxs
=   pela definição de expand
    map boxs.filter (all nodups).expand.map pruneRow.boxs
=   considerando que boxs.boxs = id
    filter (all nodups.boxs).map boxs.expand.map pruneRow.boxs
=   considerando que map boxs.expand = expand.boxs
    filter (all nodups.boxs).expand.boxs.map pruneRow.boxs
=   introduzindo pruneBy f = f.map pruneRow.f
    filter (all nodups.boxs).expand.pruneby boxs
```

Demonstramos então que:

```
filter (all nodups . boxs) . expand
  = filter (all nodups . boxs) . expand . pruneBy boxs
```

onde

```
pruneBy f = f . map pruneRow . f
```

Repetindo o cálculo para linhas e colunas obtemos:

```
filter valid . expand = filter valid . expand . prune
```

onde

```
prune = pruneBy boxs . pruneBy cols . pruneBy rows
```

Definimos então a função `solve` como:

```
solve = filter valid . expand . prune . choices
```

Uma vez que ao aplicarmos `prune` algumas escolhas passam a ser únicas, podemos repetir o processo enquanto for possível.

```
many :: Eq a => (a -> a) -> a -> a
many f x = if x == y then x else many f y
     where y = f x
```

Redefining a função `solve`:

```
solve = filter valid . expand . many prune . choices
```

Um resolutor de puzzles Sudoku

Definimos uma função solve para encontrar soluções para um determinado puzzle Sudoku.

A eficiência do algoritmo aumenta significativamente quando eliminamos alternativas que nunca serão válidas.

Podemos melhorar ainda mais a eficiência do algoritmo se combinarmos o corte de alternativas com a expansão de uma única célula.

Ver "Thinking Functionally with Haskell", R. Bird, Cambridge University Press, 2015.