Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2023/24)

Lic. em Engenharia Informática

Grupo G13

a100764 Duarte Afonso Freitas Ribeiro
a100596 João Henrique da Silva Gomes Peres Coelho
a100692 José Filipe Ribeiro Rodrigues

janeiro 2024

Preâmbulo

Cálculo de Programas tem como objectivo principal ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação corolários) e usam-se esses combinadores para construir programas *composicionalmente*, isto é, agregando programas já existentes.

Na sequência pedagógica dos planos de estudo dos cursos que têm esta disciplina, opta-se pela aplicação deste método à programação em Haskell (sem prejuízo da sua aplicação a outras linguagens funcionais). Assim, o presente trabalho prático coloca os alunos perante problemas concretos que deverão ser implementados em Haskell. Há ainda um outro objectivo: o de ensinar a documentar programas, a validá-los e a produzir textos técnico-científicos de qualidade.

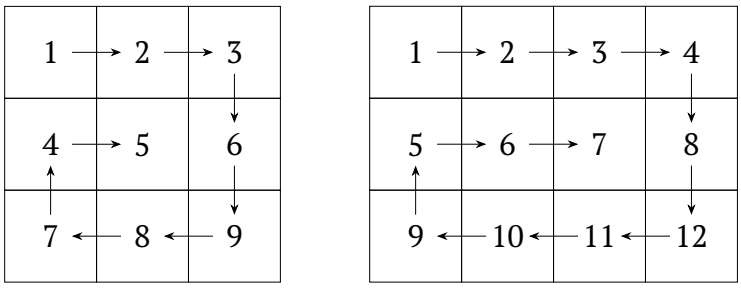
Antes de abodarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao sofware a instalar, etc.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Problema 1

Este problema, retirado de um *site* de exercícios de preparação para entrevistas de emprego, tem uma formulação simples:

> *Dada uma matriz de uma qualquer dimensão, listar todos os seus elementos rodados em espiral.*
>
> *Por exemplo, dadas as seguintes matrizes:*



> *dever-se-á obter, respetivamente,* $[1,2,3,6,9,8,7,4,5]$ *e* $[1,2,3,4,8,12,11,10,9,5,6,7]$.
>
> □

Valorizar-se-ão as soluções *pointfree* que empreguem os combinadores estudados na disciplina, e.g. $f \cdot g$, $\langle f,g\rangle$, $f \times g$, $[f,g]$, $f+g$, bem como catamorfismos e anamorfismos.

Recomenda-se a escrita de *pouco* código e de soluções simples e fáceis de entender. Recomenda-se que o código venha acompanhado de uma descrição de como funciona e foi concebido, apoiado em diagramas explicativos. Para instruções sobre como produzir esses diagramas e exprimir raciocínios de cálculo, ver o anexo D.

Problema 2

Este problema, que de novo foi retirado de um *site* de exercícios de preparação para entrevistas de emprego, tem uma formulação muito simples:

Inverter as vogais de um string.

Esta formulação deverá ser generalizada a:

Inverter os elementos de uma dada lista que satisfazem um dado predicado.

Valorizam-se as soluções tal como no problema anterior e fazem-se as mesmas recomendações.

Problema 3

Sistemas como chatGPT etc baseiam-se em algoritmos de aprendizagem automática que usam determinadas funções matemáticas, designadas *activation functions* (AF), para modelar aspectos não lineares do mundo real. Uma dessas AFs é a tangente hiperbólica, definida como o quociente do seno e coseno hiperbólicos,

$$\tanh x = \frac{\sinh x}{\cosh x} \tag{1}$$

podendo estes ser definidos pelas seguintes séries de Taylor:

$$\begin{aligned} \sum_{k=0}^{\infty} \frac{x^{2k+1}}{(2k+1)!} &= \sinh x \\ \sum_{k=0}^{\infty} \frac{x^{2k}}{(2k)!} &= \cosh x \end{aligned} \tag{2}$$

Interessa que estas funções sejam implementadas de forma muito eficiente, desdobrando-as em operações aritméticas elementares. Isso pode ser conseguido através da chamada programação dinâmica que, em Cálculo de Programas, é feita de forma *correct-by-construction* derivando-se ciclos-**for** via lei de recursividade mútua generalizada a tantas funções quanto necessário — ver o anexo E.

O objectivo desta questão é codificar como um ciclo-for (em Haskell) a função

$$snh\ x\ i = \sum_{k=0}^{i} \frac{x^{2k+1}}{(2k+1)!} \tag{3}$$

que implementa $sinh\ x$, uma das funções de $tanh\ x$ (1), através da soma das i primeiras parcelas da sua série (2).

Deverá ser seguida a regra prática do anexo E e documentada a solução proposta com todos os cálculos que se fizerem.

Problema 4

Uma empresa de transportes urbanos pretende fornecer um serviço de previsão de atrasos dos seus autocarros que esteja sempre actual, com base em *feedback* dos seus paassageiros. Para isso, desenvolveu uma *app* que instala num telemóvel um botão que indica coordenadas GPS a um serviço central, de forma anónima, sugerindo que os passageiros o usem preferencialmente sempre que o autocarro onde vão chega a uma paragem.

Com base nesses dados, outra funcionalidade da *app* informa os utentes do serviço sobre a probabilidade do atraso que possa haver entre duas paragens (partida e chegada) de uma qualquer linha.

Pretende-se implementar esta segunda funcionalidade assumindo disponíveis os dados da primeira. No que se segue, ir-se-á trabalhar sobre um modelo intencionalmente *muito simplificado* deste sistema, em que se usará o mónade das distribuições probabilísticas (ver o anexo F). Ter-se-á, então:

- paragens de autocarro

  $$\textbf{data}\ Stop = S0 \mid S1 \mid S2 \mid S3 \mid S4 \mid S5\ \textbf{deriving}\ (Show, Eq, Ord, Enum)$$

  que formam a linha $[S0\,..\,S5]$ assumindo a ordem determinada pela instância de $Stop$ na classe $Enum$;

- segmentos da linha, isto é, percursos entre duas paragens consecutivas:

  $$\textbf{type}\ Segment = (Stop, Stop)$$

- os dados obtidos a partir da *app* dos passageiros que, após algum processamento, ficam disponíveis sob a forma de pares *(segmento, atraso observado)*:

  $$dados :: [(Segment, Delay)]$$

  (Ver no apêndice G, página 9, uma pequena amostra destes dados.)

A partir destes dados, há que:

- gerar a base de dados probabilística

  $$db :: [(Segment, \text{Dist}\ Delay)]$$

  que regista, estatisticamente, a probabilidade dos atrasos ($Delay$) que podem afectar cada segmento da linha. Recomenda-se aqui a definição de uma função genérica

  $$mkdist :: Eq\ a \Rightarrow [a] \to \text{Dist}\ a$$

  que faça o sumário estatístico de uma qualquer lista finita, gerando a distribuição de ocorrência dos seus elementos.

- com base em db, definir a função probabilística

  $$delay :: Segment \to \text{Dist}\ Delay$$

  que dará, para cada segmento, a respectiva distribuição de atrasos.

Finalmente, o objectivo principal é definir a função probabilística:

$$pdelay :: Stop \to Stop \to \text{Dist}\ Delay$$

$pdelay\ a\ b$ deverá informar qualquer utente que queira ir da paragem a até à paragem b de uma dada linha sobre a probabilidade de atraso acumulado no total do percurso $[a\,..\,b]$.

Valorizar-se-ão as soluções que usem funcionalidades monádicas genéricas estudadas na disciplina e que sejam elegantes, isto é, poupem código desnecessário.

Anexos

A Natureza do trabalho a realizar

Este trabalho teórico-prático deve ser realizado por grupos de 3 alunos. Os detalhes da avaliação (datas para submissão do relatório e sua defesa oral) são os que forem publicados na página da disciplina na *internet*.

Recomenda-se uma abordagem participativa dos membros do grupo em **todos** os exercícios do trabalho, para assim poderem responder a qualquer questão colocada na *defesa oral* do relatório.

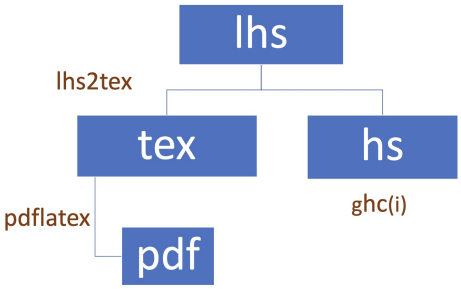
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *Um programa e a sua documentação devem coincidir.*

Por outras palavras, o **código fonte** e a **documentação** de um programa deverão estar no mesmo ficheiro.

O ficheiro `cp2324t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2324t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2324t.zip`.

Como se mostra no esquema abaixo, de um único ficheiro (*lhs*) gera-se um PDF ou faz-se a interpretação do código Haskell que ele inclui:



Vê-se assim que, para além do GHCi, serão necessários os executáveis pdflatex e lhs2TeX. Para facilitar a instalação e evitar problemas de versões e conflitos com sistemas operativos, é recomendado o uso do Docker tal como a seguir se descreve.

B Docker

Recomenda-se o uso do container cuja imagem é gerada pelo Docker a partir do ficheiro `Dockerfile` que se encontra na diretoria que resulta de descompactar `cp2324t.zip`. Este container deverá ser usado na execução do GHCi e dos comandos relativos ao LaTeX. (Ver também a `Makefile` que é disponibilizada.)

[1] O sufixo 'lhs' quer dizer *literate Haskell*.

Após instalar o Docker e descarregar o referido zip com o código fonte do trabalho, basta executar os seguintes comandos:

```
$ docker build -t cp2324t .
$ docker run -v ${PWD}:/cp2324t -it cp2324t
```

NB: O objetivo é que o container seja usado *apenas* para executar o GHCi e os comandos relativos ao LaTeX. Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2324t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2324t` no container sejam partilhadas.

Pretende-se então que visualize/edite os ficheiros na sua máquina local e que os compile no container, executando:

```
$ lhs2TeX cp2324t.lhs > cp2324t.tex
$ pdflatex cp2324t
```

lhs2TeX é o pre-processador que faz "pretty printing" de código Haskell em LaTeX e que faz parte já do container. Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro `cp2324t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2324t.lhs
```

Abra o ficheiro `cp2324t.lhs` no seu editor de texto preferido e verifique que assim é: todo o texto que se encontra dentro do ambiente

```
\begin{code}
...
\end{code}
```

é seleccionado pelo GHCi para ser executado.

C Em que consiste o TP

Em que consiste, então, o *relatório* a que se referiu acima? É a edição do texto que está a ser lido, preenchendo o anexo H com as respostas. O relatório deverá conter ainda a identificação dos membros do grupo de trabalho, no local respectivo da folha de rosto.

Para gerar o PDF integral do relatório deve-se ainda correr os comando seguintes, que actualizam a bibliografia (com BibTeX) e o índice remissivo (com `makeindex`),

```
$ bibtex cp2324t.aux
$ makeindex cp2324t.idx
```

e recompilar o texto como acima se indicou. (Como já se disse, pode fazê-lo correndo simplesmente `make` no container.)

No anexo G disponibiliza-se algum código Haskell relativo aos problemas que são colocados. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

Deve ser feito uso da programação literária para documentar bem o código que se desenvolver, em particular fazendo diagramas explicativos do que foi feito e tal como se explica no anexo D que se segue.

D Como exprimir cálculos e diagramas em LaTeX/lhs2TeX

Como primeiro exemplo, estudar o texto fonte (lhs) do que está a ler[1] onde se obtém o efeito seguinte:[2]

$$
\begin{array}{ll}
 & id = \langle f, g\rangle \\
\equiv & \quad \{\ \text{universal property}\ \} \\
 & \left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right. \\
\equiv & \quad \{\ \text{identity}\ \} \\
 & \left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right. \\
\Box &
\end{array}
$$

Os diagramas podem ser produzidos recorrendo à *package* xymatrix, por exemplo:

$$
\begin{array}{ccc}
\mathbb{N}_0 & \xleftarrow{\ \text{in}\ } & 1 + \mathbb{N}_0 \\
{\scriptstyle (\!|g|\!)}\downarrow & & \downarrow{\scriptstyle id + (\!|g|\!)} \\
B & \xleftarrow[\ g\ ]{} & 1 + B
\end{array}
$$

E Regra prática para a recursividade mútua em $\mathbb{N}_0$

Nesta disciplina estudou-se como fazer programação dinâmica por cálculo, recorrendo à lei de recursividade mútua.[3]

Para o caso de funções sobre os números naturais ($\mathbb{N}_0$, com functor F $X = 1 + X$) é fácil derivar-se da lei que foi estudada uma *regra de algibeira* que se pode ensinar a programadores que não tenham estudado Cálculo de Programas. Apresenta-se de seguida essa regra, tomando como exemplo o cálculo do ciclo-for que implementa a função de Fibonacci, recordar o sistema:

$$
\begin{array}{l}
fib\ 0 = 1 \\
fib\ (n+1) = f\ n \\
f\ 0 = 1 \\
f\ (n+1) = fib\ n + f\ n
\end{array}
$$

Obter-se-á de imediato

$$
\begin{array}{l}
fib' = \pi_1 \cdot \text{for}\ loop\ init\ \textbf{where} \\
\quad loop\ (fib, f) = (f, fib + f) \\
\quad init = (1, 1)
\end{array}
$$

usando as regras seguintes:

[1] Procure e.g. por `"sec:diagramas"`.

[2] Exemplos tirados de [2].

[3] Lei (3.95) em [2], página 110.

- O corpo do ciclo *loop* terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.[1]
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável n.
- Em *init* coleccionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios do segundo grau $ax^2 + bx + c$ em $\mathbb{N}_0$. Seguindo o método estudado nas aulas[2], de $f\ x = ax^2 + bx + c$ derivam-se duas funções mutuamente recursivas:

$$\begin{aligned} &f\ 0 = c \\ &f\ (n+1) = f\ n + k\ n \\ &k\ 0 = a + b \\ &k\ (n+1) = k\ n + 2\ a \end{aligned}$$

Seguindo a regra acima, calcula-se de imediato a seguinte implementação, em Haskell:

$$\begin{aligned} &f'\ a\ b\ c = \pi_1 \cdot \text{for}\ loop\ init\ \textbf{where} \\ &\quad loop\ (f,k) = (f + k, k + 2 * a) \\ &\quad init = (c, a + b) \end{aligned}$$

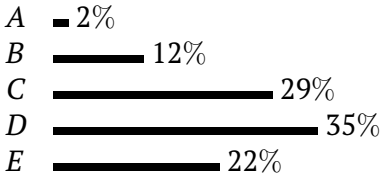
F O mónade das distribuições probabilísticas

Mónades são functores com propriedades adicionais que nos permitem obter efeitos especiais em programação. Por exemplo, a biblioteca Probability oferece um mónade para abordar problemas de probabilidades. Nesta biblioteca, o conceito de distribuição estatística é captado pelo tipo

$$\textbf{newtype}\ \text{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{4}$$

em que *ProbRep* é um real de 0 a 1, equivalente a uma escala de 0 a 100%.

Cada par (a, p) numa distribuição $d :: \text{Dist}\ a$ indica que a probabilidade de a é p, devendo ser garantida a propriedade de que todas as probabilidades de d somam 100%. Por exemplo, a seguinte distribuição de classificações por escalões de A a E,



será representada pela distribuição

$$\begin{aligned} &d1 :: \text{Dist}\ Char \\ &d1 = D\ [('\texttt{A}', 0.02), ('\texttt{B}', 0.12), ('\texttt{C}', 0.29), ('\texttt{D}', 0.35), ('\texttt{E}', 0.22)] \end{aligned}$$

que o GHCi mostrará assim:

[1] Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.

[2] Secção 3.17 de [2] e tópico Recursividade mútua nos vídeos de apoio às aulas teóricas.

```
'D'  35.0%
'C'  29.0%
'E'  22.0%
'B'  12.0%
'A'   2.0%
```

É possível definir geradores de distribuições, por exemplo distribuições *uniformes*,

$$d2 = uniform\ (words\ \texttt{"Uma frase de cinco palavras"})$$

isto é

```
   "Uma"  20.0%
 "cinco"  20.0%
    "de"  20.0%
 "frase"  20.0%
"palavras"  20.0%
```

distribuição *normais*, eg.

$$d3 = normal\ [10\,..\,20]$$

etc.[1] Dist forma um **mónade** cuja unidade é $return\ a = D\ [(a,1)]$ e cuja composição de Kleisli é (simplificando a notação)

$$(f \bullet g)\ a = [(y, q*p)\mid (x,p) \leftarrow g\ a, (y,q) \leftarrow f\ x]$$

em que $g : A \to$ Dist B e $f : B \to$ Dist C são funções **monádicas** que representam *computações probabilísticas*.

Este mónade é adequado à resolução de problemas de *probabilidades e estatística* usando programação funcional, de forma elegante e como caso particular da programação monádica.

G Código fornecido

Problema 1

$$m1 = [[1,2,3],[4,5,6],[7,8,9]]$$
$$m2 = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]$$
$$m3 = words\ \texttt{"Cristina Monteiro Carvalho Sequeira"}$$
$$test1 = matrot\ m1 \equiv [1,2,3,6,9,8,7,4,5]$$
$$test2 = matrot\ m2 \equiv [1,2,3,4,8,12,11,10,9,5,6,7]$$
$$test3 = matrot\ m3 \equiv \texttt{"CristinaooarieuqeSCMonteirhlavra"}$$

Problema 2

$$test4 = reverseVowels\ \texttt{""} \equiv \texttt{""}$$
$$test5 = reverseVowels\ \texttt{"ácidos"} \equiv \texttt{"ocidás"}$$
$$test6 = reverseByPredicate\ even\ [1\,..\,20] \equiv [1,20,3,18,5,16,7,14,9,12,11,10,13,8,15,6,17,4,19,2]$$

[1] Para mais detalhes ver o código fonte de Probability, que é uma adaptação da biblioteca PFP ("Probabilistic Functional Programming"). Para quem quiser saber mais recomenda-se a leitura do artigo [**?**].

Problema 3

Nenhum código é fornecido neste problema.

Problema 4

Os atrasos, medidos em minutos, são inteiros:

$$\textbf{type}\ Delay = \mathbb{Z}$$

Amostra de dados apurados por passageiros:

$$\begin{aligned} dados = [&((S0,S1),0),((S0,S1),2),((S0,S1),0),((S0,S1),3),((S0,S1),3),\\ &((S1,S2),0),((S1,S2),2),((S1,S2),1),((S1,S2),1),((S1,S2),4),\\ &((S2,S3),2),((S2,S3),2),((S2,S3),4),((S2,S3),0),((S2,S3),5),\\ &((S3,S4),2),((S3,S4),3),((S3,S4),5),((S3,S4),2),((S3,S4),0),\\ &((S4,S5),0),((S4,S5),5),((S4,S5),0),((S4,S5),7),((S4,S5),-1)] \end{aligned}$$

"Funcionalização" de listas:

$$\begin{aligned} &mkf :: Eq\ a \Rightarrow [(a,b)] \to a \to Maybe\ b\\ &mkf = flip\ Prelude.lookup \end{aligned}$$

Ausência de qualquer atraso:

$$\begin{aligned} &instantaneous :: \text{Dist}\ Delay\\ &instantaneous = D\ [(0,1)] \end{aligned}$$

H Soluções dos alunos

Os alunos devem colocar neste anexo as suas soluções para os exercícios propostos, de acordo com o "layout" que se fornece. Não podem ser alterados os nomes ou tipos das funções dadas, mas pode ser adicionado texto ao anexo, bem como diagramas e/ou outras funções auxiliares que sejam necessárias.

Importante: Não pode ser alterado o texto deste ficheiro fora deste anexo.

Problema 1

Depois de analisar o problema dado, o grupo concluiu que o mesmo se resumia a um simples catamorfismo.

A estratégia desenvolvida para a resolução do exercício foi a seguinte:

1. Pegar na cabeça da lista, visto que já está na ordem pretendida;
2. Transpor o resto da matriz de forma a que a próxima cabeça da lista fique na ordem pretendida;
3. Repetir o processo até que a lista esteja vazia, usando o catamorfismo.

Esta estratégia é representada pelo seguinte diagrama:

$$\begin{array}{ccc} A^{**} & \xrightarrow{myOutList} & 1 + A^* \times A^{**} \\ {\scriptstyle (\!|\, matrot \,|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|\, matrot \,|\!)} \\ A^* & \xleftarrow[g=[nil,\mathsf{conc}]]{} & 1 + A^* \times A^* \end{array}$$

A função *myOutList* aplica desde já a transposição à matriz para que esta preocupação não seja da função *matrot*.

Esta função, chamada de f, é representada pelo seguinte diagrama, que também é um catamorfismo:

$$\begin{array}{ccc} A^{**} & \xrightarrow{outMatrix} & 1 + A^* \times A^{**} \\ {\scriptstyle (\!|f|\!)}\downarrow & & \downarrow{\scriptstyle id+id\times(\!|f|\!)} \\ A^{**} & \xleftarrow[g=[nil,cons]]{} & 1 + A^* \times A^{**} \end{array}$$

É aplicado aqui um novo *out*, o *outMatrix*, que cria as linhas da nova matriz, pegando nos ultimos elementos de cada linha, e manda o resto da matriz no segundo elemento do either para ser processado pelo catamorfismo.

Para além disso, ainda desenvolvemos uma outra versão da função *matrot*, $matrotV2$, que usa funções pré-definidas do Haskell para a resolução do problema, mais precisamente a função *transpose* e a função *reverse*.

No entanto, ao contrário da estratégia apresentada anteriormente, esta função não é um verdadeiro catamorfismo, apesar de ter um comportamento semelhante.

Podemos então ver o código desenvolvido pelo grupo que resolve o problema proposto:

$$\begin{aligned} &myOutList\ [\,] = i_1\ () \\ &myOutList\ (h:t) = i_2\ (h, f\ t) \end{aligned}$$

$$\begin{aligned} &outMatrix\ ([\,]:xs) = i_1\ () \\ &outMatrix\ l = i_2\ (\mathsf{map}\ \ last\ l, \mathsf{map}\ \ init\ l) \end{aligned}$$

$$\begin{aligned} &matrot :: Eq\ a \Rightarrow [[a]] \to [a] \\ &matrot = [nil, \mathsf{conc}] \cdot recList\ matrot \cdot myOutList \end{aligned}$$

$$f = [nil, cons] \cdot recList\ f \cdot outMatrix$$

$$matrotV2 = [nil, \mathsf{conc}] \cdot recList\ (matrotV2 \cdot reverse \cdot transpose) \cdot outList$$

Problema 2

Neste problema, o grupo desenvolveu duas funções principais, a *reverseVowels* e a *reverseByPredicate*.

Começando pela *reverseVowels*, primeiramente foi desenvolvida uma função auxiliar, a *trocaVowels*, que troca as vogais de uma string por outras vogais, dadas numa outra string. Esta função pode ser representada por um anamorfismo, cujo gene é trivial observando a própria função. Assim, foi desenvolvido o *geneTwoLists*, que recebe duas strings e devolve um either com a primeira string, caso a segunda seja vazia, ou com um par. O valor do par depende do primeiro elemento da primeira string:

1. Se o primeiro elemento da primeira string for uma vogal:
   (a) 1º elemento do par: primeiro elemento da segunda string;
   (b) 2º elemento do par: par com o resto da primeira string e o resto da segunda string.
2. Se o primeiro elemento da primeira string não for uma vogal:
   (a) 1º elemento do par: primeiro elemento da primeira string;
   (b) 2º elemento do par: par com o resto da primeira string e a segunda string inteira.

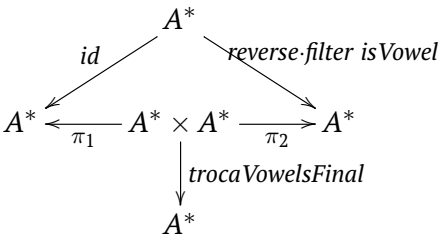
Com estas funções, é possível criar o seguinte diagrama para a *trocaVowels*:

$$\begin{array}{ccc} A^* & \xleftarrow{\mathbf{in}=[id,cons]} & A^* + (A \times A^*) \\ {\scriptstyle [\![trocaVowelsFinal]\!]}\uparrow & & \uparrow{\scriptstyle id+id\times[\![trocaVowelsFinal]\!]} \\ A^* \times A^* & \xrightarrow[geneTwoLists]{} & A^* + (A \times (A^* \times A^*)) \end{array}$$

De seguida, foi então desenvolvida a função *reverseVowels*, que recebe uma string e devolve-a com as vogais invertidas. A estratégia utilizada para a resolução deste problema foi a seguinte:

1. Criar um par de *strings* a partir da original utilizando um split;
2. O primeiro elemento do par é a *string* original, enquanto o segundo elemento é uma *string* constituída pelas vogais presentes na *string* original, só que invertidas;
3. Trocar as vogais da primeira string pelas vogais da segunda utilizando a função *trocaVowels*.

Assim, o diagrama da função *reverseVowels* é o seguinte:



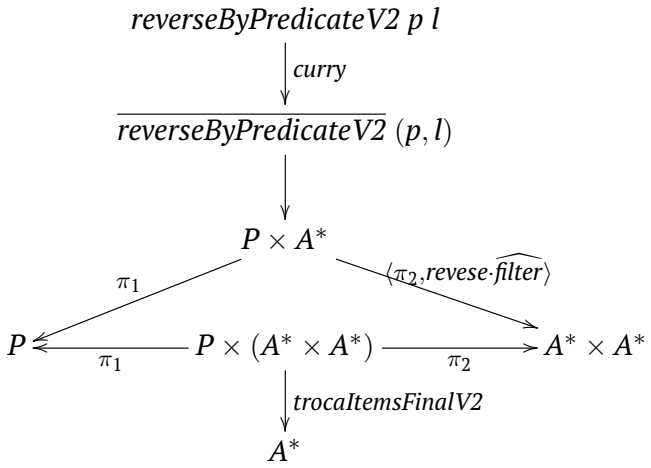
Finalmente, foi desenvolvida a função *reverseByPredicate* que recebe uma função e uma lista e devolve-a com os elementos que satisfazem a função passada como argumento com ordem invertida. A estratégia utilizada para a resolução deste problema foi a seguinte:

1. Tal como na função *reverseVowels*, é criado um par de *strings* a partir da original utilizando um split, os elementos do par também são idênticos;
2. É aplicada a função *trocaItemsFinal* ao par
3. A única diferença entre a função *reverseVowels* e a função *reverseByPredicate* é que a função utilizada no *filter* é a função passada como argumento, em vez da *isVowel*.

Com isto, o diagrama da função *reverseByPredicate* é semelhante ao da função *reverseVowels*. As únicas diferenças são a presença da função p, passada como argumento, em vez da *isVowel*, e o uso da função *trocaItemsFinal* em vez da *trocaVowelsFinal*:

Para além disso, desenvolvêmos também uma segunda versão da função *reverseByPredicate*, a *reverseByPredicateV2*, que permite realizar a tarefa desejada sem ser necessária a presença de uma

função nos argumentos. Por esta razão é que o diagrama da função *reverseByPredicateV2* é diferente do da função *reverseByPredicate*:



De seguida, apresentamos o código desenvolvido com todas as funções referidas anteriormente:

$$isVowel = (\in \texttt{"áàâãaeéèêiíìîóòôõoúùûuAÀÁÂÃEÉÈÊIÍÌÎÓÒÕÔOÚÙÛU"})$$

$$\begin{aligned}
&trocaVowels :: (String, String) \to String\\
&trocaVowels\ (l, [\,]) = l\\
&trocaVowels\ ((x:xs), (y:ys)) =\\
&\quad \textbf{case}\ isVowel\ x\ \textbf{of}\\
&\qquad True \to y : trocaVowels\ (xs, ys)\\
&\qquad False \to x : trocaVowels\ (xs, (y:ys))
\end{aligned}$$

$$\begin{aligned}
&geneTwoLists\ _\ (xs, [\,]) = i_1\ xs\\
&geneTwoLists\ f\ (x:xs, y:ys) = \textbf{if}\ f\ x\ \textbf{then}\ i_2\ (y, (xs, ys))\ \textbf{else}\ i_2\ (x, (xs, y:ys))
\end{aligned}$$

$$trocaVowelsFinal = [id, cons] \cdot recList\ trocaVowelsFinal \cdot geneTwoLists\ isVowel$$

$$\begin{aligned}
&reverseVowels :: String \to String\\
&reverseVowels = trocaVowelsFinal \cdot \langle id, reverse \cdot filter\ isVowel\rangle
\end{aligned}$$

$$\begin{aligned}
&reverseByPredicate :: (a \to Bool) \to [a] \to [a]\\
&reverseByPredicate\ p = trocaItemsFinal \cdot \langle id, reverse \cdot filter\ p\rangle\ \textbf{where}\\
&\quad trocaItemsFinal = [id, cons] \cdot recList\ trocaItemsFinal \cdot geneTwoLists\ p
\end{aligned}$$

$$\begin{aligned}
&reverseByPredicateV2 = \overline{trocaItemsFinalV2 \cdot \langle \pi_1, \langle \pi_2, reverse \cdot \widehat{(filter)}\rangle\rangle}\ \textbf{where}\\
&\quad trocaItemsFinalV2 = [id, cons] \cdot recList\ trocaItemsFinalV2 \cdot geneTwoListsV2\\
&\quad geneTwoListsV2\ (_, (xs, [\,])) = i_1\ xs\\
&\quad geneTwoListsV2\ (f, (x:xs, y:ys)) = \textbf{if}\ f\ x\ \textbf{then}\ i_2\ (y, (f, (xs, ys)))\ \textbf{else}\ i_2\ (x, (f, (xs, y:ys)))
\end{aligned}$$

Problema 3

Como indicado no enunciado, este problema pode ser otimizado utilizando o conceito de programação dinâmica para derivar um ciclo *for* via lei de recursividade mútua.

Uma primeira análise a este problema indica que uma abordagem *Bottom Up* seria a mais indicada para este problema específico, visto que a decomposição da exponenciação e da fatorização presentes na equação poderão permitir definir o valor de uma iteração do somatório em função da imediatamente anterior a essa, permitindo assim calcular cada nova iteração *"guardando"* o valor calculado em iterações anteriores e multiplicando-o por um determinado valor.

Esta abordagem torna o cálculo mais eficiente do que calcular todos os valores do somatório de forma independente, pois existiriam muitos cálculos repetidos desnecessários, visto que o valor da iteração seguinte pode ser calculado usando o valor da iteração anterior[1]. Para além disso, é mais eficiente que a abordagem *Top Down* pois esta perderia tempo a calcular imediatamente a última iteração (a mais complexa) e depois calcularia as iterações mais pequenas a partir dessa, perdendo assim a eficiência de parte do cálculo da próxima iteração ser o resultado da iteração anterior.

Com a abordagem definida, a resolução do problema começa por perguntar a seguinte questão: *Existe uma forma de calcular o valor em certa iteração em função da iteração anterior? Se sim, qual é?*

Começamos por avaliar a equação inicial:

$$snh\ x\ i = \sum_{k=0}^{i} \frac{x^{2k+1}}{(2k+1)!} \tag{5}$$

Podemos separá-la em dois casos, quando o número de iterações é 0 e quando o número de iterações é maior que 0, definindo este segundo recursivamente ao invés de com um somatório:

$$\begin{cases} snh\ x\ 0 = x \\ snh\ x\ (i+1) = \frac{x^{2\times(i+1)+1}}{(2(i+1)+1)!} + snh\ x\ i \end{cases} \tag{6}$$

Podemos agora mover a primeira componente da adição de cada iteração para uma função auxiliar e também tentar defini-la de forma recursiva:

$$\begin{cases} snh\ x\ 0 = f\ x\ 0 \\ snh\ x\ (n+1) = f\ x\ (n+1) + snh\ x\ n \end{cases} \tag{7}$$

$$\begin{cases} f\ x\ 0 = \frac{x^{2\times0+1}\times x^{2\times0+1}}{(2\times0+1)!} \\ f\ x\ (n+1) = \frac{x^2\times x^{2n+1}}{(2+2n+1)!} \end{cases} \tag{8}$$

$$\Leftrightarrow \begin{cases} f\ x\ 0 = x \\ f\ x\ (n+1) = \frac{x^2}{(2n+1)\times(2n)} \times f\ x\ n \end{cases} \tag{9}$$

Conseguimos assim duas funções mutuamente recursivas (snh depende de si mesma e de f, f apenas de si própria). Podemos deste modo inferir um *for loop* que implemente essa funcionalidade,onde:

[1] Por exemplo: calcular fatoriais de 1 a 10 independentemente quando podemos apenas definir o fatorial de certo valor em função do anterior vezes o número que estamos a fatorizar

- *loop* recebe um par (snh,f) e realiza as transformações definidas acima
- A condição inicial (*start*) seria um par com valor igual aos casos de paragem
- O resultado final é determinado pela soma do valor das duas funções na iteração final (Como definido em snh, é a soma da iteração de f atual com o snh da iteração anterior, devido à natureza das funções mutuamente recursivas snh no fim do loop apenas tem o resultado da iteração anterior)

Sendo assim ficaríamos com uma função deste género:

$$
\begin{aligned}
&snh\ x = wrapper \cdot worker\ \textbf{where}\\
&\quad worker = \text{for}\ loop\ start\\
&\quad wrapper\ (res, f) = res + f\\
&\quad loop\ (snh, f) = (snh + f, (f * x ** 2)\ /\ (4 * k ** 2 + 2 * k))\\
&\quad start = (0, x)
\end{aligned}
$$

No entanto, existe um problema. Onde é que o programa encontra esse k (número da iteração atual)? É verdade que é um argumento (implícito) da função snh, mas analisando a função for da biblioteca Nat, percebemos que esta não torna disponível o número da iteração de forma explícita a *loop*. Sendo assim precisaremos de uma terceira função que conte e disponibilize o número da iteração atual a f. Deste modo, obtemos:

$$
\begin{aligned}
&snh\ x = wrapper \cdot worker\ \textbf{where}\\
&\quad worker = \text{for}\ loop\ start\\
&\quad wrapper\ (res, f, \_) = res + f\\
&\quad loop\ (snh, f, k) = (snh + f, (f * x ** 2)\ /\ (4 * k ** 2 + 2 * k), \text{succ}\ \ k)\\
&\quad start = (0, x, 1)
\end{aligned}
$$

Chegamos assim a uma solução satisfatória de todos os requisitos pedidos. No entanto, não nos agradou o facto da terceira função, que pouco passa de um contador, estar assim tão "explícita", e optamos por colocá-la junto de f num par dentro do par principal, pois f depende da função contadora. Chegamos assim à seguinte solução:

$$
\begin{aligned}
&snh\ x = wrapper \cdot worker\ \textbf{where}\\
&\quad worker = \text{for}\ loop\ start\\
&\quad wrapper = \widehat{(+)} \cdot (id \times \pi_1)\\
&\quad loop\ (snh, f) = (snh + \pi_1\ f, (nextIter\ (\pi_2\ f) \times \text{succ}\ )\ f)\\
&\quad nextIter\ k\ v = (v * x ** 2)\ /\ (4 * k ** 2 + 2 * k)\\
&\quad start = (0, (x, 1))
\end{aligned}
$$

No entanto, esta função ainda não está completamente de acordo com os requisitos estabelecidos. No enunciado é-nos pedido que a função *worker* tenha a sintaxe *worker = for (loop x) (start x)*, o que implica que *loop* e *start* tenham de estar fora do corpo de *snh* e passem a ser funções independentes. Para cumprir este requisito, desevolvemos a seguinte solução:

$$
\begin{aligned}
&snh\ x = wrapper \cdot worker\ \textbf{where}\\
&\quad worker = \text{for}\ loop\ x\ start\ x\\
&\quad wrapper = \widehat{(+)} \cdot (id \times \pi_1)\\
&loop\ x\ (snh, f) = (snh + \pi_1\ f, (nextIter\ (\pi_2\ f) \times \text{succ}\ )\ f)\ \textbf{where}\\
&\quad nextIter\ k\ v = (v * x ** 2)\ /\ (4 * k ** 2 + 2 * k)\\
&start\ x = (0, (x, 1))
\end{aligned}
$$

Esta solução preenche realmente todos os requisitos necessários. No entanto achamos interessante desenvolver uma versão onde as funções mutuamente recursivas em *loop* não estão separadas nos

argumentos, e que depois são utilizadas de acordo com uma sintaxe mais *point-wise*. Aprensentamos aqui essa solução:

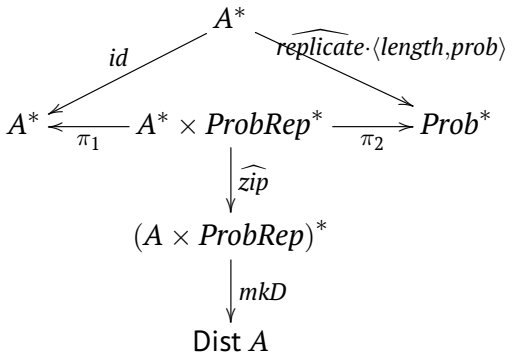
```
snh x = wrapper · worker where
  worker = for loop x start x
  wrapper = (+)^ · (id × π1)
loop x res = ((π1 (π2 res)+) × (nextIter (π2 (π2 res)) × succ )) res where
  nextIter k v = (v ∗ x ∗∗ 2) / (4 ∗ k ∗∗ 2 + 2 ∗ k)
start x = (0, (x, 1))
```

Problema 4

Passando agora para o exercicío 4, foi-nos pedido que desenvolvêssemos 4 funções que manipulam os dados fornecidos no exercício. Começando pela função *mkdist*, a mesma recebe uma lista finita de valores e devolve a distribuição de ocorrência dos seus elementos. A estratégia utilizada para a resolução deste problema foi a seguinte:

1. criar um par em que o primeiro elemento é a lista de dados original e o segundo elemento é a lista de probabilidades;
2. aplicar a função *zip* ao par criado anteriormente, de forma a que cada elemento da lista de dados original fique associado à sua probabilidade;
3. chamar a função *mkD* com a lista de pares criada anteriormente como argumento, retornando assim a distribuição de ocorrência dos elementos.

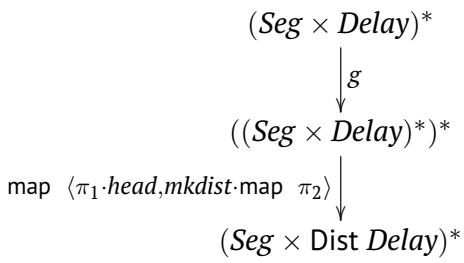
Assim, é possível criar o seguinte diagrama para a função *mkdist*:

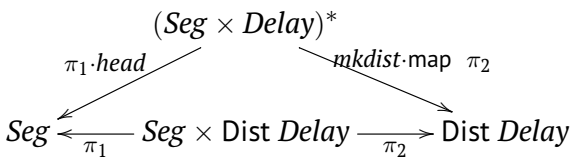


De seguida, foi desenvolvida a função *db* que recebe e devolve uma lista de pares, em que o primeiro elemento é o elemento do par original e o segundo elemento é a distribuição dos atrasos do mesmo. A estratégia utilizada para a resolução deste problema foi a seguinte:

1. Agrupar os pares da lista original pelos segmentos de partida e chegada;
2. Para cada segmento, criar um par em que o primeiro elemento é o segmento e o segundo elemento é a distribuição de ocorrência dos atrasos;
3. É utilizado um *map* para aplicar a função *split* a cada par da lista original.

Com isto, é possível criar o seguinte diagrama para a função *db*, à esquerda, e para o split usado no *map*, à direita:





Continuando, foi desenvolvida a função *delay* que recebe um segmento e devolve a distribuição de ocorrência dos atrasos desse segmento. A estratégia utilizada para a resolução deste problema foi a seguinte:

1. É chamada a função *mkf* com a lista de pares e o segmento como argumentos;
2. Se a função *mkf* devolver *Nothing*, então é devolvida a distribuição *instantaneous*;
3. Se a função *mkf* devolver *Just*, então é devolvida a distribuição devolvida pela função *mkf*.

Devido à complexidade reduzida da função não foi criado um diagrama para a mesma.

Por fim, foi desenvolvida a função *pdelay*, que recebe duas paragens e devolve a distribuição de atrasos acumulados entre as duas paragens. Antes de calcular a distribução é necessário calcular os segmentos entre as duas paragens, para isso é gerada uma lista de paragens entre as duas, usando a função *enumFromTo*, cuja lista é passada para a expressão $zip <*> tail$. Esta expressão irá calcular os segmentos entre as paragens, usando o operador de aplicação sequencial. Este operador irá passar a lista resultante de aplicar *tail* à lista de paragens para a função *zip*, que irá manter a lista original, juntando assim as duas listas originandos os segmentos. Desta forma, tendo obtido os segmentos, é possível calcular a distribuição de atrasos acumulados. Para isso é usado a função *foldl* que irá aplicar a função g a cada segmento, acumulando os atrasos, em que o caso inicial é a distribuição *instantaneous*, que funciona como elemento neutro. A função g tira proveito do monad desenvolvido para a resolução do problema, usando o operador de composição de Kleisli, que irá aplicar a função p ao atraso acumulado e à distribuição de atrasos do segmento, obtendo assim a distribuição de atrasos acumulados do segmento seguinte. É fácil de se notar que a função p irá percorrer a distribuição do segmento e irá somar o atraso acumulado ao atraso de cada elemento da distribuição. Desta forma, o próprio monad irá calcular as probablidades de cada atraso acumulado.

$$db = \text{map}\ \ \langle \pi_1 \cdot head, mkdist \cdot \text{map}\ \ \pi_2 \rangle\ (g\ dados)\ \textbf{where}$$
$$g = groupBy\ \widehat{\overline{(\equiv)}} \cdot (\pi_1 \times \pi_1)$$

$$mkdist = mkD \cdot \widehat{zip} \cdot \langle id, \widehat{replicate} \cdot \langle length, prob \rangle \rangle\ \textbf{where}$$
$$prob = (1/) \cdot fromIntegral \cdot length$$

$$delay = maybe\ instantaneous\ id \cdot mkf\ \ db$$

$$pdelay\ a\ b = foldl\ g\ instantaneous\ \$\ ((zip <*> tail)\ \$\ enumFromTo\ a\ b)\ \textbf{where}$$
$$g = \widehat{\overline{(\gg\!=)}} \cdot (id \times p)$$
$$p\ s\ x = \text{fmap}\ (x+)\ (delay\ s)$$

Index

References

[1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[2] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.