Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2025/26)

Lic. em Ciências da Computação

Lic. em Engenharia Informática

Grupo G31

a107291 Gabriel Pinto Dantas
a106937 José Lourenço Ferreira Fernandes
a107322 Simão Azevedo Oliveira

dezembro 2025

Preâmbulo

Em Cálculo de Programas pretende-se ensinar a programação de computadores como uma disciplina científica. Para isso parte-se de um repertório de *combinadores* que formam uma álgebra da programação 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 abordarem os problemas propostos no trabalho, os grupos devem ler com atenção o anexo A onde encontrarão as instruções relativas ao *software* a instalar, etc.

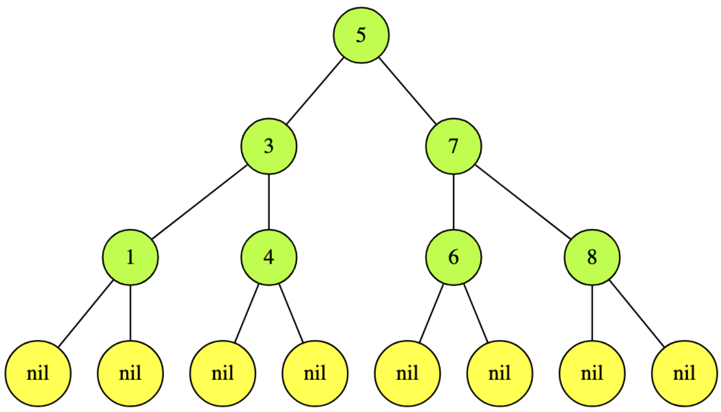
Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

Avaliação. Faz parte da avaliação do trabalho a sua defesa por parte dos elementos de cada grupo. Estes devem estar preparados para responder a perguntas sobre *qualquer* dos problemas deste enunciado. A prestação *individual* de cada aluno nessa defesa oral será uma componente importante e diferenciadora da avaliação.

Problema 1

Uma serialização (ou travessia) de uma árvore é uma sua representação sob a forma de uma lista. Na biblioteca $BTree$ encontram-se as funções de serialização $inordt$, $preordt$ e $postordt$, que fazem as travessias *in-order*, *pre-order* e *post-order*, respectivamente. Todas essas travessias são catamorfismos que percorrem a árvore argumento em regime *depth-first*.

Pretende-se agora uma função $bforder$ que faça a travessia em regime *breadth-first*, isto é, por níveis. Por exemplo, para a árvore t_1 dada em anexo e mostrada na figura a seguir,



a função deverá dar a lista

$$[5, 3, 7, 1, 4, 6, 8]$$

em que se vê como os níveis 5, depois $3, 7$ e finalmente $1, 4, 6, 8$ foram percorridos.

Pretendemos propor duas versões dessa função:

1. Uma delas envolve um catamorfismo de *BTree*s:

$$bfsLevels :: BTree\ a \rightarrow [a]$$
$$bfsLevels = concat \cdot levels$$

   Complete a definição desse catamorfismo:

$$levels :: BTree\ a \rightarrow [[a]]$$
$$levels = (\!|\, glevels \,|\!)$$

2. A segunda proposta,

$$bft :: BTree\ a \rightarrow [a]$$

   deverá basear-se num anamorfismo de listas.

Sugestão: estudar o artigo [2] cujo PDF está incluído no material deste trabalho. Quando fizer testes ao seu código pode, se desejar, usar funções disponíveis na biblioteca *Exp* para visualizar as árvores em GraphViz (formato .dot).

Justifique devidamente a sua resolução, que deverá vir acompanhada de diagramas explicativos. Como já se disse, 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 2

Considere a seguinte função em Haskell:

$$\begin{array}{l} f\ x = wrapper \cdot worker\ \textbf{where} \\ \quad wrapper = head \\ \quad worker\ 0 = start\ x \\ \quad worker\ (n+1) = loop\ x\ (worker\ n) \\ loop\ x\ [s, \quad h, \quad k, j, \quad m\ ] = \\ \qquad\qquad [h\ /\ k + s, x \uparrow 2 * h, k * j, j + m, m + 8] \\ start\ x = [x, x \uparrow 3, 6, 20, 22] \end{array}$$

Pode-se provar pela lei de recursividade mútua que $f\ x\ n$ calcula o seno hiperbólico de x, $sinh\ x$, para n aproximações da sua série de Taylor. Faça a derivação da função dada a partir da referida série de Taylor, apresentando todos os cálculos justificativos, tal como se faz para outras funções no capítulo respectivo do texto base desta UC [3].

Problema 3

Quem em Braga observar, ao fim da tarde, o tráfego onde a Avenida Clairmont Fernand se junta à N101, aproximadamente na coordenada 41°33'46.8"N 8°24'32.4"W — ver as setas da figura que se segue — reparará nas sequências imparáveis (infinitas!) de veículos provenientes dessas vias de circulação.

Mas também irá observar um comportamento interessante por parte dos condutores desses veículos: por regra, *cada carro numa via deixa passar, à sua frente, exactamente outro carro da outra via.*

Este comportamento *civilizado* chama-se *fair-merge* (ou *fair-interleaving*) de duas sequências infinitas, também designadas *streams* em ciência da computação. Seja dado o tipo dessas sequências em Haskell,

$$\mathbf{data}\ \mathit{Stream}\ a = \mathit{Cons}\ (a, \mathit{Stream}\ a)\ \mathbf{deriving}\ \mathit{Show}$$

para o qual se define também:

$$\mathsf{out}\ (\mathit{Cons}\ (x, xs)) = (x, xs)$$

O referido comportamento civilizado pode definir-se, em Haskell, da forma seguinte:[1]

$$\begin{array}{l}
\mathit{fair_merge} :: (\mathit{Stream}\ a, \mathit{Stream}\ a) + (\mathit{Stream}\ a, \mathit{Stream}\ a) \to \mathit{Stream}\ a \\
\mathit{fair_merge} = [h, k]\ \mathbf{where} \\
\quad h\ (\mathit{Cons}\ (x, xs), y) = \mathit{Cons}\ (x, k\ (xs, y)) \\
\quad k\ (x, \mathit{Cons}\ (y, ys)) = \mathit{Cons}\ (y, h\ (x, ys))
\end{array}$$

Defina *fair_merge* como um **anamorfismo** de *Streams*, usando o combinador

$$[\!(\, g \,)\!] = \mathit{Cons} \cdot (id \times [\!(\, g \,)\!]) \cdot g$$

e a seguinte estratégia:

- Derivar a lei **dual** da recursividade mútua,

$$[f, g] = [\!(\, [h, k] \,)\!] \quad \equiv \quad \left\{ \begin{array}{l} out \cdot f = \mathsf{F}\ [f, g] \cdot h \\ out \cdot g = \mathsf{F}\ [f, g] \cdot k \end{array} \right. \tag{1}$$

  tal como se fez, nas aulas, para a que está no formulário.

- Usar (1) na resolução do problema proposto.

Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

Problema 4

Como se sabe, é possível pensarmos em catamorfismos, anamorfismos etc *probabilísticos*, quer dizer, programas recursivos que dão distribuições como resultados. Por exemplo, podemos pensar num combinador

$$\mathit{pcataList} :: (() + (a, b) \to \mathsf{Dist}\ b) \to [a] \to \mathsf{Dist}\ b$$

[1] O facto das sequências serem infinitas não nos deve preocupar, pois em Haskell isso é lidado de forma transparente por lazy evaluation.

que é muito parecido com

$$(\!| \cdot |\!) :: (() + (a,b) \to b) \to [a] \to b$$

da biblioteca List. A principal diferença é que o gene de *pcataList* é uma função probabilística.

Como exemplo de utilização, recorde-se que $(\!| [zero, add] |\!)$ soma todos os elementos da lista argumento, por exemplo:

$$(\!| [zero, add] |\!)\ [20, 10, 5] = 35.$$

Considere-se agora a função *padd* (adição probabilística) que, com probabilidade 90% soma dois números e com probabilidade 10% os subtrai:

$$padd\ (a, b) = D\ [(a + b, 0.9), (a - b, 0.1)]$$

Se se correr

$$d4 = pcataList\ [pzero, padd]\ [20, 10, 5]\ \mathbf{where}\ pzero = return \cdot zero$$

obter-se-á:

```
35  81.0%
25   9.0%
 5   9.0%
15   1.0%
```

Com base neste exemplo, resolva o seguinte

> ***Problema**: Uma unidade militar pretende enviar uma mensagem urgente a outra, mas tem o aparelho de telegrafia meio avariado. Por experiência, o telegrafista sabe que a probabilidade de uma palavra se perder (não ser transmitida) é 5%; e que, no final de cada mensagem, o aparelho envia o código* `"stop"`*, mas (por estar meio avariado), falha 10% das vezes.*
>
> *Qual a probabilidade de a palavra* `"atacar"` *da mensagem*
>
> *words* `"Vamos atacar hoje"`
>
> *se perder, isto é, o resultado da transmissão ser* `["Vamos","hoje","stop"]`*? E a de seguirem todas as palavras, mas faltar o* `"stop"` *no fim? E a da transmissão ser perfeita?*

Responda a estas perguntas encontrando *gene* tal que

$$transmitir = pcataList\ gene$$

descreve o comportamento do aparelho. Justificar devidamente a resolução, que deverá vir acompanhada de diagramas explicativos.

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.

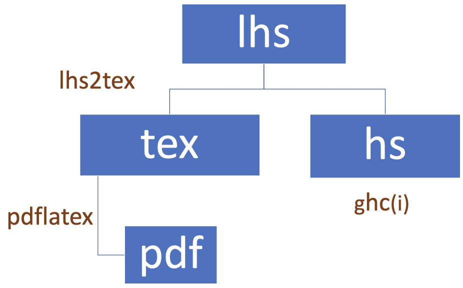
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *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 `cp2526t.pdf` que está a ler é já um exemplo de programação literária: foi gerado a partir do texto fonte `cp2526t.lhs`[1] que encontrará no material pedagógico desta disciplina descompactando o ficheiro `cp2526t.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 `cp2526t.zip`. Este container deverá ser usado na execução do GHCi e dos comandos relativos ao LaTeX. (Ver também a `Makefile` que é disponibilizada.)

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

```
$ docker build -t cp2526t .
$ docker run -v ${PWD}:/cp2526t -it cp2526t
```

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}:/cp2526t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2526t` no container sejam partilhadas.

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

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

```
$ lhs2TeX cp2526t.lhs > cp2526t.tex
$ pdflatex cp2526t
```

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 `cp2526t.lhs` é executável e contém o "kit" básico, escrito em Haskell, para realizar o trabalho. Basta executar

```
$ ghci cp2526t.lhs
```

Abra o ficheiro `cp2526t.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 G 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 cp2526t.aux
$ makeindex cp2526t.idx
```

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

No anexo F 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]

$$
\begin{aligned}
& id = \langle f, g \rangle \\
\equiv \quad & \{ \text{ universal property } \}
\end{aligned}
$$

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

[2] Exemplos tirados de [3].

$$\left\{ \begin{array}{l} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{array} \right.$$

$\equiv \qquad \{$ identity $\}$

$$\left\{ \begin{array}{l} \pi_1 = f \\ \pi_2 = g \end{array} \right.$$

$\square$

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

$$\begin{tikzcd} \mathbb{N}_0 \arrow[d, "(\!|g|\!)"'] & 1 + \mathbb{N}_0 \arrow[l, "\mathsf{in}"'] \arrow[d, "id + (\!|g|\!)"] \\ B & 1 + B \arrow[l, "g"] \end{tikzcd}$$

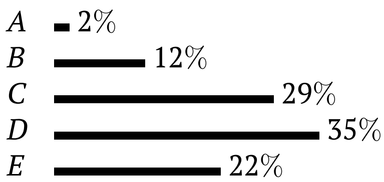
E O mónade das distribuições probabilísticas

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

$$\mathbf{newtype}\ \mathsf{Dist}\ a = D\ \{unD :: [(a, ProbRep)]\} \tag{2}$$

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

Cada par (a,p) numa distribuição $d :: \mathsf{Dist}\ a$ 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

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

que o GHCi mostrará assim:

```
'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.

$$d3 = normal\ [10\mathinner{..}20]$$

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 $g : A \to \text{Dist } B$ e $f : B \to \text{Dist } C$ 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.

F Código fornecido

Problema 1

Árvores exemplo:

$$
\begin{array}{l}
t_1 :: BTree\ Int \\
t_1 = Node\ (5, (Node\ (3, (Node\ (1, (Empty, Empty)), Node\ (4, (Empty, Empty)))), \\
\quad Node\ (7, (Node\ (6, (Empty, Empty)), Node\ (8, (Empty, Empty)))))) \\
t_2 :: BTree\ Int \\
t_2 = \\
\quad node\ 1 \\
\qquad (node\ 2\ (node\ 4\ Empty\ Empty)\ (node\ 5\ Empty\ Empty)) \\
\qquad (node\ 3\ (node\ 6\ Empty\ Empty)\ (node\ 7\ Empty\ Empty)) \\
t_3 :: BTree\ Char \\
t_3 = \\
\quad node\ \texttt{'A'} \\
\qquad (node\ \texttt{'B'}\ (node\ \texttt{'C'}\ (node\ \texttt{'D'}\ Empty\ Empty)\ Empty)\ Empty) \\
\qquad (node\ \texttt{'E'}\ Empty\ Empty) \\
t_4 :: BTree\ Char \\
t_4 = \\
\quad node\ \texttt{'A'} \\
\qquad (node\ \texttt{'B'}\ (node\ \texttt{'C'}\ (node\ \texttt{'D'}\ Empty\ Empty)\ Empty)\ Empty) \\
\qquad Empty \\
t_5 :: BTree\ Int \\
t_5 = \\
\quad node\ 1
\end{array}
$$

[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 [?].

$$
\begin{array}{l}
\quad (node\ 2\ (node\ 4\ Empty\ Empty)\ Empty) \\
\quad (node\ 3\ Empty\ (node\ 5\ (node\ 6\ Empty\ Empty)\ Empty)) \\
node\ a\ b\ c = Node\ (a, (b, c))
\end{array}
$$

G 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

Solução 1: catamorfismo

Análise intuitiva do problema

Como primeira proposta é desejado que a travessia se baseie num catamorfismo de árvores binárias.

Intuitivamente, dado que desejamos uma travessia por níveis (*breadth-first*), precisamos de uma recursão que percorra a árvore e devolva os elementos ordenados por níveis. Ora, uma recursão implicitamente faz uma travessia em profundidade (*depth-first*), pelo que teremos de manipular o resultado dessa recursão para obter o resultado desejado.

Inicie-se o processo por analisar ambas as funções dadas, *bfsLevels* e *bft*, de modo a a perceber o que cada uma delas deve fazer.

A função *bfsLevels*, dado uma árvore binária de elementos de tipo A, deve devolver uma lista de elementos de A, ordenados por níveis, ou seja, primeiro os elementos do nível 0 (a raiz), depois os do nível 1 (os filhos da raiz), depois os do nível 2 (os netos da raiz), e assim sucessivamente. Esta função é definida à custa de um *concat* após um catamorfismo *levels*.

O catamorfismo *levels*, por sua vez, deve devolver uma lista de listas de elementos de A, onde cada lista interna corresponde a um nível da árvore.

Exemplifique-se o funcionamento deste algoritmo para um entendimento claro do que se pretende alcançar. Dada a árvore t_1, o catamorfismo $levels\ \ t_1$ deve devolver a lista de listas $[[5], [3, 7], [1, 4, 6, 8]]$. A função $bfsLevels\ \ t_1$, por sua vez, deve devolver a lista resultante da concatenação das listas internas, ou seja, $[5, 3, 7, 1, 4, 6, 8]$.

Definição do catamorfismo

O objetivo é definir o gene *glevels* de *levels*, de modo a que este catamorfismo devolva a lista de listas de elementos de A ordenados por níveis.

Represente-se esquematicamente este catamorfismo:

$$
\begin{tikzcd}
BTree\ A \arrow[r, "\mathsf{out}"] \arrow[d, "(\!|glevels|\!)"'] & 1 + A \times (BTree\ A \times BTree\ A) \arrow[d, "id + id \times ((\!|glevels|\!) \times (\!|glevels|\!))"] \\
{[[A]]} & 1 + A \times ([[A]] \times [[A]]) \arrow[l, "g"]
\end{tikzcd}
$$

Devido à existência do coproduto, sabe-se que o gene do catamorfismo será uma função definida por:

$$g = [g1, g2] \tag{3}$$

Para definir o gene, considere-se o que deve acontecer nos dois casos possíveis da árvore dada como argumento:

- No caso base, em que a árvore é vazia ($Empty$), a lista de níveis deve ser a lista vazia ($[\,]$).
- No caso recursivo, em que a árvore é um nó $Node\ (a, (l, r))$, com raiz a e subárvores l e r, devem ser combinadas as listas de níveis das subárvores l e r, adicionando-se o elemento a a essa combinação de forma pertinente.

O caso base é simples de definir:

$$g1 = nil \tag{4}$$

Para o caso recursivo, defina-se uma função auxiliar $concatPointWise$:

$$\begin{array}{l}
concatPointWise :: [[a]] \rightarrow [[a]] \rightarrow [[a]] \\
concatPointWise\ [\,]\ ys = ys \\
concatPointWise\ xs\ [\,] = xs \\
concatPointWise\ (x : xs)\ (y : ys) = (x +\!\!+\ y) : concatPointWise\ xs\ ys
\end{array}$$

Esta função trata de concatenar duas listas de listas de elementos do tipo A. Isto trata de juntar os níveis das duas listas, ponto a ponto. A primeira claúsula trata do caso em que a primeira lista é vazia, retornando a segunda lista. A segunda claúsula trata do caso em que a segunda lista é vazia, retornando a primeira lista. A terceira claúsula trata do caso em que ambas as listas são não vazias, concatenando as cabeças das listas e chamando-se recursivamente para as caudas. Por exemplo,

$$concatPointWise\ [[3], [1, 4]]\ [[7], [6, 8]] = [[3, 7], [1, 4, 6, 8]]$$

Assim, o gene $g2$ pode ser definido como:

$$g2\ (a, (ls, rs)) = singl\ a +\!\!+\ concatPointWise\ ls\ rs \tag{5}$$

Em que ls e rs são as listas de níveis das subárvores l e r, respetivamente. O resultado é a lista de níveis das subárvores, combinadas ponto a ponto, com o elemento a adicionado como novo nível no início da lista.

Que, transformando em uma versão pointfree, resulta em:

$$g2 = cons \cdot (singl \times \widehat{concatPointWise}) \tag{6}$$

Definição de *glevels*

Com base nas equações (3), (4) e (6), A definição final de $glevels$ é:

$$glevels = [nil, cons \cdot (singl \times \widehat{concatPointWise})]$$

Definição pointfree da função auxiliar

Com o intuito de embelezar a nossa solução, optámos por desafiar-nos a definir a função de concatenação ponto a ponto $concatPointWise$ de forma pointfree através de um anamorfismo de listas.

Recordemos que a função *concatPointWise* recebe duas listas de listas e combina-as ponto a ponto. Podemos pensá-la como um processo iterativo que constrói progressivamente a lista resultado, o que sugere o uso de um **anamorfismo**.

O anamorfismo de listas tem a forma:

$$[\!(\, g \,)\!] = \mathsf{in}_{List} \cdot (id + id \times [\!(\, g \,)\!]) \cdot g$$

O gene g recebe um estado (neste caso, o par de listas $([[a]],[[a]])$) e decide:

- Se deve parar (devolvendo $i_1\ ()$)
- Ou produzir o próximo elemento e o novo estado (devolvendo $i_2\ (elemento, novo\_estado)$)

Definamos então *concatCondicional* como um anamorfismo:

$$
\begin{array}{l}
concatCondicional :: [[a]] \to [[a]] \to [[a]] \\
concatCondicional = \overline{[\!(\, gene \,)\!]} \\
\quad \mathbf{where} \\
\qquad gene = cond\ ambasVazias\ stop \\
\qquad\quad (cond\ listaEsquerdaVazia\ doListaDireita \\
\qquad\quad (cond\ listaDireitaVazia\ doListaEsquerda\ doAmbas)) \\
\qquad\quad \text{-- Predicados} \\
\qquad ambasVazias = \widehat{(\wedge)} \cdot (null \times null) \\
\qquad listaEsquerdaVazia = null \cdot \pi_1 \\
\qquad listaDireitaVazia = null \cdot \pi_2 \\
\qquad\quad \text{-- Ações} \\
\qquad stop = i_1 \cdot \\
\qquad doListaDireita = i_2 \cdot \langle head \cdot \pi_2, \langle nil, tail \cdot \pi_2 \rangle \rangle \\
\qquad doListaEsquerda = i_2 \cdot \langle head \cdot \pi_1, \langle tail \cdot \pi_1, nil \rangle \rangle \\
\qquad doAmbas = i_2 \cdot \langle \mathsf{conc} \cdot (head \times head), tail \times tail \rangle
\end{array}
$$

Explicação detalhada do gene:

O gene implementa uma máquina de estados que analisa o par de listas e decide como proceder. Usa uma cascata de condicionais ($cond\ p\ f\ g$) que funciona como $\mathbf{if}\ p\ \mathbf{then}\ f\ \mathbf{else}\ g$:

1. **Caso base - ambas as listas vazias** $([\,],[\,])$:

   Quando não há mais elementos em nenhuma das listas, o processo termina. O gene devolve $i_1\ ()$ para sinalizar paragem ao anamorfismo.

   $$stop = i_1 \cdot$$

   Este é o critério de terminação que garante que a recursão não é infinita.

2. **Caso degenerado - só a lista esquerda está vazia** $([\,], y : ys)$:

   Neste caso, não há mais elementos da lista esquerda para combinar, mas ainda existem elementos em *ys* que devem ser incluídos no resultado final.

   A ação *doListaDireita* constrói o próximo elemento da lista resultado e o novo estado:

   - **Elemento emitido**: $head \cdot \pi_2$ extrai y (a primeira sublista de *ys*)
   - **Novo estado**: $([\,], tail \cdot \pi_2)$ resulta em $([\,], ys')$ onde ys' são as sublistas restantes

$$doListaDireita = i_2 \cdot \langle head \cdot \pi_2, \langle nil, tail \cdot \pi_2 \rangle \rangle$$

Em termos práticos: como não há nada à esquerda, simplesmente copiamos y para o resultado e continuamos com $([\,], ys')$.

3. **Caso simétrico - só a lista direita está vazia** $(x : xs, [\,])$:

   Situação simétrica ao caso anterior: esgotaram-se os elementos de ys mas ainda há elementos em xs para processar.

   A ação *doListaEsquerda* funciona analogamente:

   - **Elemento emitido**: $head \cdot \pi_1$ extrai x (a primeira sublista de xs)
   - **Novo estado**: $(tail \cdot \pi_1, [\,])$ resulta em $(xs', [\,])$ onde xs' são as sublistas restantes

   $$doListaEsquerda = i_2 \cdot \langle head \cdot \pi_1, \langle tail \cdot \pi_1, nil \rangle \rangle$$

   Interpretação: copiamos x para o resultado e prosseguimos com $(xs', [\,])$.

4. **Caso geral - ambas as listas não-vazias** $(x : xs, y : ys)$:

   Este é o caso principal onde efetivamente combinamos elementos de ambas as listas. Queremos concatenar x com y (ambas são sublistas) e continuar o processo com os elementos restantes.

   A ação *doAmbas* realiza esta operação:

   - **Elemento emitido**: $\mathsf{conc} \cdot (head \times head)$
     - $(head \times head)$ extrai (x, y) — as primeiras sublistas de cada lado
     - conc concatena-as, produzindo $x +\!\!+\, y$
   - **Novo estado**: $(tail \times tail)$ produz (xs, ys) — os restantes elementos de ambas as listas

   $$doAmbas = i_2 \cdot \langle \mathsf{conc} \cdot (head \times head), tail \times tail \rangle$$

   Esta é a operação central da concatenação ponto a ponto: combinar x e y em $x +\!\!+\, y$ e avançar para o próximo par de sublistas.

Fluxo de controlo:

A cascata de condicionais funciona como uma árvore de decisão:

if *ambasVazias* **then**
  stop
else if *listaEsquerdaVazia* **then**
  doListaDireita
else if *listaDireitaVazia* **then**
  doListaEsquerda
else
  doAmbas

Esta estrutura garante que tratamos todos os casos possíveis de forma exaustiva e mutuamente exclusiva.

Exemplo de execução:

Para $[[3], [1, 4]]$ e $[[7], [6, 8]]$:

$$
\begin{aligned}
& concatCondicional\ [[3],[1,4]]\ [[7],[6,8]] \\
&= (\!|\, gene\,|\!)\ ([[3],[1,4]],[[7],[6,8]]) \\
&= \quad \text{-- gene devolve i2 ([3,7], ([[1,4]], [[6,8]]))} \\
&= [3,7] : (\!|\, gene\,|\!)\ ([[1,4]],[[6,8]]) \\
&= \quad \text{-- gene devolve i2 ([1,4,6,8], ([], []))} \\
&= [3,7] : [1,4,6,8] : (\!|\, gene\,|\!)\ ([\,],[\,]) \\
&= \quad \text{-- gene devolve i1 ()} \\
&= [[3,7],[1,4,6,8]]
\end{aligned}
$$

E assim podemos definir a função glevels de forma mais elegante:

$$glevels1 = [nil, cons \cdot (singl \times \widehat{concatCondicional})]$$

Solução 2: anamorfismo

Motivação: limitações da solução por catamorfismo

A solução anterior baseada em *levels* funciona corretamente, mas tem uma ineficiência: constrói toda a estrutura de níveis $[[A]]$ para depois a concatenar. Para árvores grandes, isto consome memória desnecessária.

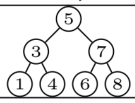
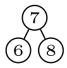
Idealmente, gostaríamos de gerar diretamente a lista final $[A]$ sem construir a estrutura intermédia. Isto sugere o uso de um **anamorfismo**.

Intuição: processamento por níveis com uma fila

Para fazer uma travessia *breadth-first*, o algoritmo clássico usa uma fila (queue), tal como sugerido no artigo [2]. Pensámos inicialmente num algoritmo iterativo que funciona da seguinte forma:

1. Inicializamos a fila com a árvore argumento
2. Repetidamente:
   - Retiramos uma árvore da frente da fila
   - Se for $Empty$, ignoramos
   - Se for $Node\ (a,(l,r))$, emitimos a e adicionamos l e r ao fim da fila
3. Paramos quando a fila fica vazia

Exemplo ilustrativo: Para a árvore t_1 o processamento da fila procede da seguinte forma:

Fila (Queue)	Saída	Ação
	[]	Inicia
	[5]	Processa 5
 1 4	[5,3]	Processa 3
1 4 6 8	[5,3,7]	Processa 7
4 6 8	[5,3,7,1]	Processa 1
6 8	[5,3,7,1,4]	Processa 4
8	[5,3,7,1,4,6]	Processa 6
	[5,3,7,1,4,6,8]	Termina

Nota: Ocultam-se os *Empty*s para simplificar a ilustração.

Do algoritmo iterativo ao anamorfismo

O algoritmo acima é iterativo e gera uma lista progressivamente, processando primeiro os nós de nível n antes dos de nível $n + 1$. Esta estrutura corresponde precisamente a um anamorfismo de listas, que constrói uma lista a partir de um estado através de um gene $g :: B \to 1 + A \times B$.

O gene recebe um estado B e devolve $i_1\ ()$ quando não há mais elementos (terminação), ou $i_2\ (a, b')$ onde a é o próximo elemento a emitir e b' é o novo estado. No nosso caso, o estado será a fila $[BTree\ A]$ que contém as árvores ainda por processar.

Representemos o anamorfismo através do seguinte diagrama:

$$\begin{tikzcd}
{[BTree\ A]} \arrow[r, "g"] \arrow[d, "[\![g]\!]"'] & 1 + A \times [BTree\ A] \arrow[d, "id + id \times [\![g]\!]"] \\
{[A]} & 1 + A \times [A] \arrow[l, "\mathsf{in}_{List}"]
\end{tikzcd}$$

O gene $g :: [BTree\ A] \to 1 + A \times [BTree\ A]$ deve inspecionar a fila e decidir o que fazer. Caso esta esteja vazia, termina devolvendo $i_1\ ()$; caso contrário, retira a primeira árvore, emite a sua raiz e adiciona os filhos ao fim da fila. No entanto, se a árvore for *Empty*, deve ser ignorada e o gene deve processar o restante da fila recursivamente.

Podemos expressar isto de forma simples:

$$\begin{array}{l}
g :: [BTree\ a] \to () + (a, [BTree\ a]) \\
g\ [\,] = i_1\ () \\
g\ (Empty : queue) = g\ queue \\
g\ (Node\ (a, (l, r)) : queue) = i_2\ (a, queue +\!\!+ [l, r])
\end{array}$$

A função *bft* inicia o processo com uma fila contendo apenas a árvore argumento, aplicando depois o anamorfismo com o gene g:

$$bft = [\![g]\!] \cdot singl$$

Problema 2

Parta-se da definição matemática da série de Taylor do seno hiperbólico:

$$sinh\ x = \sum_{n=0}^{\infty} \frac{x^{2n+1}}{(2n+1)!} = x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \dots$$

A aproximação de $sinh\ x$ com n termos da série será dada por:

$$sinh\ x\ n = \sum_{i=0}^{n-1} \frac{x^{2i+1}}{(2i+1)!}$$

Definição recursiva direta

Podemos definir $sinh\ x\ n$ recursivamente como:

$$sinh\ x\ n = \begin{cases} x & \text{se } n = 0 \\ \frac{x^{2(n-1)+1}}{(2(n-1)+1)!} + sinh\ x\ (n-1) & \text{se } n > 0 \end{cases} \tag{7}$$

No entanto, esta definição é ineficiente, pois cada termo envolve o cálculo de potências e fatoriais.

Descoberta de uma definição eficiente por recursividade mútua

Podemos escrever a soma como uma recursão simples:

$$s\ x\ n = \begin{cases} x & \text{se } n = 0 \\ s\ x\ (n-1) + \frac{h\ x\ (n-1)}{k\ x\ (n-1)} & \text{se } n > 0 \end{cases} \tag{8}$$

onde $s\ x\ n$ é a soma dos primeiros $n+1$ termos, $h\ x\ n$ é o numerador do termo n e $k\ x\ n$ é o denominador do termo n.

Descoberta de h por recursividade mútua

Observando que cada termo tem a forma $\frac{x^{2i+1}}{(2i+1)!}$, procuramos uma relação de recorrência para o numerador h.

Para o termo na posição n, o numerador é x^{2n+1}, e para o termo na posição $n+1$, o numerador é $x^{2(n+1)+1} = x^{2n+3}$.

A relação entre numeradores consecutivos é:

$$h_{n+1} = x^{2n+3} = x^{2n+1} \cdot x^2 = h_n \cdot x^2$$

Portanto, o numerador do próximo termo é x^2 vezes o numerador do termo atual. Assim:

$$h\ x\ n = \begin{cases} x^3 & \text{se } n = 1 \\ x^2 \cdot h\ x\ (n-1) & \text{se } n > 1 \end{cases} \tag{9}$$

Descoberta de k por recursividade mútua (2ª iteração)

Calculemos a relação de recorrência para o denominador k. O denominador do termo na posição n considerando que este começa em $n = 1$ é $(2n+3)!$. Para o termo na posição $n+1$, o denominador é $(2(n+1)+3)! = (2n+5)!$. A relação entre denominadores consecutivos é:

$$\frac{k_{n+1}}{k_n} = \frac{(2n+5)!}{(2n+3)!} = (2n+5)(2n+4)$$

Portanto, o denominador do próximo termo é $(2n+5)(2n+4)$ vezes o denominador do termo atual.

Para o caso base $n = 1$, temos $k\ x\ 1 = 3! = 6$. Assim:

$$k\ x\ n = \begin{cases} 6 & \text{se } n = 1 \\ k\ x\ n \cdot (2n+5)(2n+4) & \text{se } n > 1 \end{cases} \tag{10}$$

Aplicar-se-á a lei de recursividade mútua novamente para obter uma definição de k que não dependa de n diretamente.

Descoberta de j por recursividade mútua (3ª iteração)

Introduzamos uma nova função $j\ x\ n = (2\ n+5)\ (2\ n+4)$ para simplificar a expressão de k.

Teremos uma relação entre termos consecutivos de j de:

$$j_{n+1} - j_n = (2(n+1)+5)(2(n+1)+4) - (2n+5)(2n+4) = 4n^2+26n+42-4n^2-18n-20 = 8n+22$$

No caso base $n = 0$, temos $j\ x\ 0 = (2*0+5)\ (2*0+4) = 20$.

Portanto, podemos definir j como:

$$j\ x\ n = \begin{cases} 20 & \text{se } n = 0 \\ j\ x\ n + 8n + 22 & \text{se } n > 0 \end{cases} \tag{11}$$

Descoberta de m por recursividade mútua (4ª iteração)

Para simplificar ainda mais, introduzimos $m\ x\ n = 8\ n + 22$.

Temos então a relação entre termos consecutivos de m de:

$$m_{n+1} - m_n = 8(n+1) + 22 - (8n+22) = 8$$

No caso base $n = 0$, temos $m\ x\ 0 = 8*0+22 = 22$.

Portanto, podemos definir m como:

$$m\ x\ n = \begin{cases} 22 & \text{se } n = 0 \\ m\ x\ n + 8 & \text{se } n > 0 \end{cases} \tag{12}$$

Tendo chegado a um ponto onde m é uma função linear simples, podemos agora expressar todas as funções em termos de recursividade mútua.

Sistema final de recursividade mútua

Reunindo todas as definições recursivas obtidas:

$$\begin{aligned}
s\ x\ n &= \begin{cases} x & \text{se } n = 0 \\ s\ x\ (n-1) + \frac{h\ x\ (n-1)}{k\ x\ (n-1)} & \text{se } n > 0 \end{cases} \\
h\ x\ n &= \begin{cases} x^3 & \text{se } n = 1 \\ x^2 \cdot h\ x\ (n-1) & \text{se } n > 1 \end{cases} \\
k\ x\ n &= \begin{cases} 6 & \text{se } n = 1 \\ k\ x\ (n-1) \cdot j\ x\ (n-1) & \text{se } n > 1 \end{cases} \\
j\ x\ n &= \begin{cases} 20 & \text{se } n = 0 \\ j\ x\ (n-1) + m\ x\ (n-1) & \text{se } n > 0 \end{cases} \\
m\ x\ n &= \begin{cases} 22 & \text{se } n = 0 \\ m\ x\ (n-1) + 8 & \text{se } n > 0 \end{cases}
\end{aligned}$$

Pela lei de recursividade mútua, este sistema pode ser expresso como:

$$(s, h, k, j, m) = \text{for}\ (\text{loop}\ x)\ (\text{start}\ x)$$

onde *start x* representa os valores iniciais e *loop x* representa a função de transição.

Para determinar *start x* e *loop x*, observemos que a função for $\cdot$ $\cdot$ itera n vezes, começando do estado inicial e aplicando repetidamente a função *loop*.

Os valores iniciais correspondem a $n = 0$:

$$\text{start}\ x = (x, x^3, 6, 20, 22)$$

A função *loop x* transforma o estado (s, h, k, j, m) no próximo estado, incrementando implicitamente n:

$$\text{loop}\ x\ (s, h, k, j, m) = (s + \frac{h}{k}, x^2 \cdot h, k \cdot j, j + m, m + 8)$$

Finalmente, a função *sinh x n* corresponde à primeira componente do estado após n iterações:

$$\text{sinh}\ x\ n = \pi_1 \cdot \text{for}\ (\text{loop}\ x)\ (\text{start}\ x)\ n$$

ou, usando *head* para extrair a primeira componente:

$$\text{sinh}\ x\ n = \text{head} \cdot \text{for}\ (\text{loop}\ x)\ (\text{start}\ x)\ n$$

Esta é precisamente a estrutura da função $f\ x\ n$ fornecida no enunciado, provando que $f\ x\ n$ calcula o seno hiperbólico de x com n aproximações da série de Taylor.

Transpondo para o código Haskell obtemos:

```
sinh :: Floating a => a -> Int -> a
sinh x n = p1 . for (loop x) (start x) $ n
     where
          start x = (x, x^3, 6, 20, 22)
          loop x (s, h, k, j, m) =
               (s + h / k, x^2 * h, k * j, j + m, m + 8)
```

Com pequenos ajustes, modificando os tuplos para listas e usando funções auxiliares, obtemos a definição final de *sinh* igual à fornecida no enunciado.

$$\begin{aligned}
&s :: \mathit{Floating}\ a \Rightarrow a \rightarrow \mathit{Int} \rightarrow a \\
&s\ x\ n = \mathit{head} \cdot \text{for}\ \mathit{loop}\ x\ \mathit{start}\ x\ \$\ n \\
&\quad \textbf{where} \\
&\qquad \mathit{start}\ x = [x, x \uparrow 3, 6, 20, 22] \\
&\qquad \mathit{loop}\ x\ [s, h, k, j, m] = [s + h\ /\ k, x \uparrow 2 * h, k * j, j + m, m + 8]
\end{aligned}$$

Problema 3

$$\mathit{fair\_merge}' = (\!|\ \bot\ |\!)$$

Problema 4

$$\begin{aligned}
&\mathit{pcataList} = \bot \\
&\mathit{gene} = \bot
\end{aligned}$$

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] Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 131–136. ACM, 2000.

[3] J.N. Oliveira. Program Design by Calculation, 2024. Draft of textbook in preparation. First version: 1998. Current version: Sep. 2024. Informatics Department, University of Minho (pdf).