Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2025/26)

Lic. em Ciências da Computação

Lic. em Engenharia Informática

Grupo G01

a68243 José Pedro Pinheiro da Silva
a106836 João Pedro Delgado Teixeira
a106928 Simão Pedro Pacheco Mendes

fevereiro 2026

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.

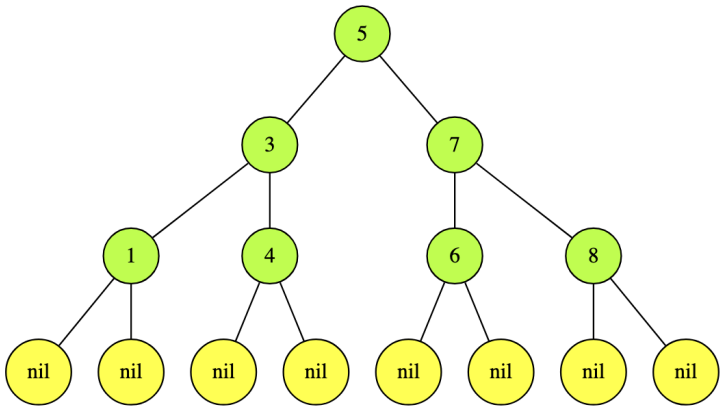
Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes que utilizem os combinadores de ordem superior estudados na disciplina.

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:

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

   Complete a definição desse catamorfismo:

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

2. A segunda proposta,

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

   deverá basear-se num anamorfismo de listas.

Sugestão: estudar o artigo [3] 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:

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

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 [4].

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,

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

para o qual se define também:

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

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

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

Defina *fair_merge* como um **anamorfismo** de *Stream*s, usando o combinador

$$[\!(g)\!] = Cons \cdot (id \times [\!(g)\!]) \cdot g$$

e a seguinte estratégia:

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

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

  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

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

[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

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

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*
>
> $$words\ \texttt{"Vamos atacar hoje"}$$
>
> *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.

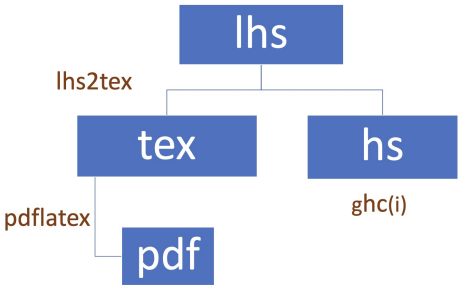
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [2], cujo princípio base é o seguinte:

> *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]

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

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

[2] Exemplos tirados de [4].

$$
\begin{cases} \pi_1 \cdot id = f \\ \pi_2 \cdot id = g \end{cases}
$$

$$
\equiv \quad \{ \text{ identity } \}
$$

$$
\begin{cases} \pi_1 = f \\ \pi_2 = g \end{cases}
$$

$$
\square
$$

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 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

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

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

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

A ▬ 2%
B ▬▬▬ 12%
C ▬▬▬▬▬▬▬▬ 29%
D ▬▬▬▬▬▬▬▬▬▬▬ 35%
E ▬▬▬▬▬▬▬ 22%

será representada pela distribuição

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

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.

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

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

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

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.

F Código fornecido

Problema 1

Árvores exemplo:

```
t1 :: BTree Int
t1 = Node (5,(Node (3,(Node (1,(Empty,Empty)),Node (4,(Empty,Empty)))),
   Node (7,(Node (6,(Empty,Empty)),Node (8,(Empty,Empty))))))
t2 :: BTree Int
t2 =
   node 1
      (node 2 (node 4 Empty Empty) (node 5 Empty Empty))
      (node 3 (node 6 Empty Empty) (node 7 Empty Empty))
t3 :: BTree Char
t3 =
   node 'A'
      (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
      (node 'E' Empty Empty)
t4 :: BTree Char
t4 =
   node 'A'
      (node 'B' (node 'C' (node 'D' Empty Empty) Empty) Empty)
      Empty
t5 :: BTree Int
t5 =
   node 1
```

[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 [1].

$$(node\ 2\ (node\ 4\ Empty\ Empty)\ Empty)$$
$$(node\ 3\ Empty\ (node\ 5\ (node\ 6\ Empty\ Empty)\ Empty))$$

$$node\ a\ b\ c = Node\ (a,(b,c))$$

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

Introdução e Análise do Problema

A travessia em largura (*Breadth-First Search* — BFS) constitui um desafio clássico no paradigma da programação funcional. Enquanto as travessias em profundidade (*Depth-First Search* — DFS) emergem naturalmente da estrutura indutiva dos tipos de dados algébricos, com o processamento de cada ramo na totalidade antes da transição para o seguinte, a BFS exige uma estratégia de processamento transversal de modo a visitar os nós por níveis de profundidade (camada a camada).

Para a definição da estrutura de dados:

$$\textbf{data}\ BTree\ a = Empty \mid Node\ (a,(BTree\ a, BTree\ a))\ \textbf{deriving}\ Show$$

A manipulação desta estrutura é efetuada através dos isomorfismos *inBTree* e *outBTree*, que respetivamente montam e desmontam a árvore de acordo com a sua assinatura functorial:

$$inBTree :: () + (b,(BTree\ b, BTree\ b)) \to BTree\ b$$
$$inBTree = [\underline{Empty}, Node]$$
$$outBTree :: BTree\ a \to () + (a,(BTree\ a, BTree\ a))$$
$$outBTree\ Empty = i_1\ ()$$
$$outBTree\ (Node\ (a,(t_1,t_2))) = i_2\ (a,(t_1,t_2))$$

onde é definido o functor:

$$\begin{cases} FX = 1 + A \times X^2 \\ Ff = id + id \times f^2 \end{cases} \tag{3}$$

Exploramos a dualidade entre o catamorfismo e o anamorfismo na realização de uma travessia BFS. Propomos, assim, duas soluções distintas para o problema:

1. **Catamorfismo:** Consome a estrutura da árvore para gerar uma representação intermédia estratificada por níveis, tipificada por uma lista de listas ($[[a]]$).
2. **Anamorfismo:** Gera a sequência de visita a partir de um estado que simula uma fila de espera (*FIFO*), o que modela o comportamento dinâmico da travessia.

Procurou-se definir as soluções recorrendo ao máximo de primitivas *pointfree* possível, com o uso dos combinadores e operadores fornecidos pela equipa docente no contexto do cálculo de programas.

1. Solução via Catamorfismo

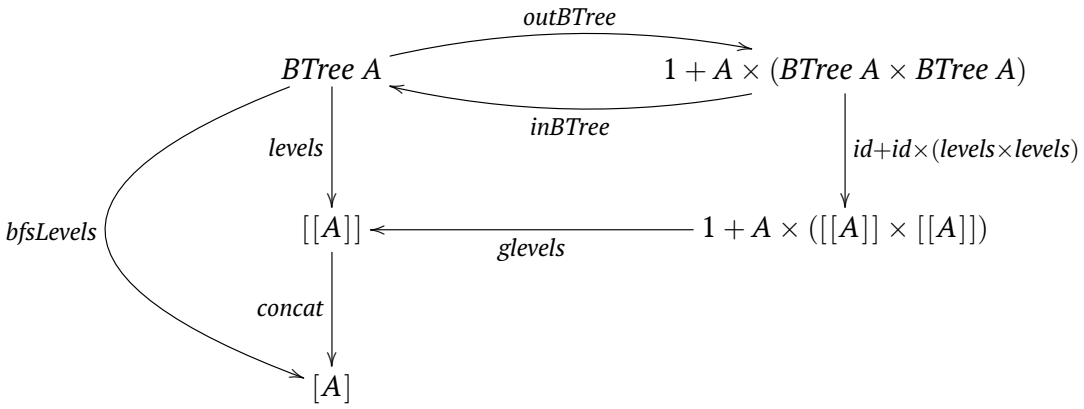
A análise do problema iniciou-se com o estudo da base de código fornecida pela equipa docente:

$$
\begin{aligned}
&bfsLevels :: BTree\ a \to [a] \\
&bfsLevels = concat \cdot levels \\
&levels :: BTree\ a \to [[a]] \\
&levels = (\!|glevels|\!)
\end{aligned}
$$

A função *bfsLevels* constitui o ponto de entrada e saída para o processamento da $BTree\ a$. Esta utiliza a função predefinida *concat*, que recebe a estrutura estratificada produzida pelo catamorfismo e aglutina os vários níveis numa lista única.

A função *levels* tem como objetivo processar uma *BTree* para gerar uma lista de listas ($[[a]]$), onde cada sub-lista de índice k contém exclusivamente os elementos que residem na profundidade k da árvore original, ordenados do topo para a base.

Formalmente, *bfslevels* define-se como a composição de uma concatenação e um catamorfismo de árvores binárias, cuja tipagem e estrutura recursiva são descritas pelo seguinte diagrama:



Análise do gene *glevels*: O gene deste catamorfismo especifica a reconstrução dos níveis da árvore a partir das sub-estruturas já processadas:

- **Caso Base ($i_1\ ()$):** Uma árvore vazia resulta numa lista de níveis vazia (*nil*), representada por uma lista de listas sem elementos ($[]$).
- **Caso Recursivo ($i_2\ (a,(ls,rs))$):** A raiz a é isolada como o nível inicial através de $singl\ a$. Este nível é então colocado à cabeça (via *cons*) da estrutura resultante da fusão das sub-árvores esquerda (*ls*) e direita (*rs*), as quais são combinadas através da função auxiliar *mergeLevels*.

O resultado final do gene é a aglutinação destes componentes numa estrutura de lista de listas ($[[a]]$), o que preserva a hierarquia por níveis necessária para o achatamento final pela função *concat*.

A função *mergeLevels* é o componente crítico desta abordagem. Esta opera como uma generalização do combinador $zipWith\ (++)$; contudo, ao contrário do *zipWith* definido na linguagem Haskell, esta preserva a cauda da estrutura mais profunda caso a árvore não seja balanceada. Desta forma, garante-se que nenhum nível é descartado durante a fusão dos ramos.

A implementação em Haskell de *glevels* e da respetiva auxiliar é a seguinte:

$$
\begin{aligned}
&glevels :: () + (a,([[a]],[[a]])) \to [[a]] \\
&glevels = [nil, cons \cdot (singl \times mergeLevels)]
\end{aligned}
$$

where
$mergeLevels :: ([[a]],[[a]]) \to [[a]]$
$mergeLevels\ ([],ys) = ys$
$mergeLevels\ (xs,[]) = xs$
$mergeLevels\ ((x:xs),(y:ys)) = cons\ (\text{conc}\ (x,y), mergeLevels\ (xs,ys))$

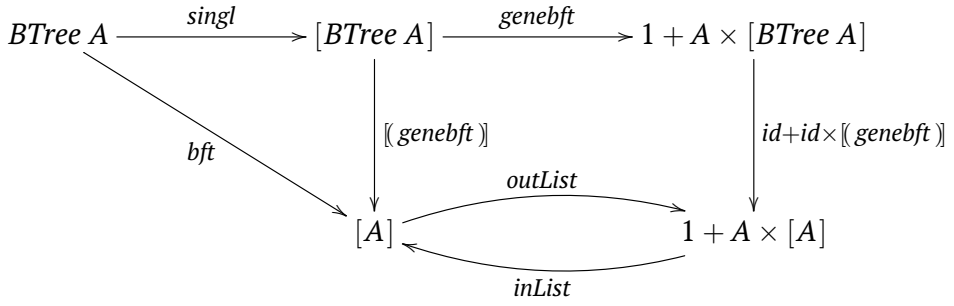
2. Solução via Anamorfismo: *bft*

A segunda solução aborda a BFS como um processo de produção de uma lista a partir de um estado dinâmico. O estado interno é definido como uma **floresta** ($State = [BTree\ a]$), que assume o papel de uma fila de espera (*FIFO*).

A função *bft* define-se como um anamorfismo de listas, cujo comportamento é regido pelo gene *genebft*:

$bft :: BTree\ a \to [a]$
$bft = [\!(\ genebft\ )\!]\cdot singl$

O diagrama seguinte ilustra a transição de estados e a geração da sequência de visita:



Análise do gene *genebft*: O gene estabelece a gestão da fila de modo a assegurar a ordem de visita transversal:

1. **Condição de Paragem:** Perante uma fila vazia, o anamorfismo termina ($i_1\ ()$).
2. **Processamento da Cabeça:** A análise da primeira árvore da fila determina o passo seguinte:
   - Caso se trate de uma árvore vazia (*Empty*), o processo ignora este elemento e prossegue de forma recursiva com o restante conteúdo da fila.
   - Caso se trate de um nó ($Node\ (a,(l,r))$), o valor a é emitido para a estrutura de saída. As sub-árvores l e r são obrigatoriamente adicionadas ao **fim** da fila.

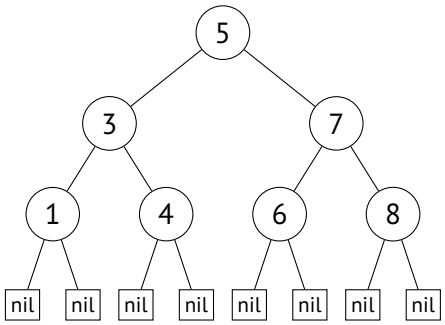
Esta inserção no final da estrutura (concatenação) garante a semântica de uma fila de espera, o que assegura a visita de todos os nós de um nível k antes do início do processamento de qualquer nó pertencente ao nível $k+1$.

A implementação do gene *genebft* em Haskell reflete esta lógica de manipulação de estados:

$genebft :: [BTree\ a] \to () + (a,[BTree\ a])$
$genebft\ [] = i_1\ ()$
$genebft\ (h:t) = \textbf{case}\ outBTree\ h\ \textbf{of}$
$\quad i_1\ () \to genebft\ t$
$\quad i_2\ (a,(l,r)) \to i_2\ (a, t +\!\!+ [l,r])$

Exemplo de Execução e Verificação

Para a validação da equivalência funcional entre o catamorfismo (*levels*) e o anamorfismo ($[\!(\mathit{genebft})\!]$), procede-se à análise da execução sobre a árvore de teste t_1, representada por:



Catamorfismo (*levels*): A abordagem catamórfica processa a árvore de forma *bottom-up*. O gene *glevels* combina a raiz de cada sub-árvore com o resultado da fusão dos níveis dos seus ramos. Esta fusão é efetuada pela função *mergeLevels*, que aglutina listas de elementos ao mesmo nível de profundidade:

1. **Processamento das Folhas (ex: nó 1):** O catamorfismo recebe *Empty* em ambos os ramos, o que resulta em $([\,],[\,])$. A aplicação de *mergeLevels* produz $[\,]$. O gene acrescenta a raiz, o que resulta em $[1] : [\,] = [[1]]$.
2. **Nível Intermédio (ex: nó 3):** As sub-árvores esquerda e direita já foram processadas para $[[1]]$ e $[[4]]$, respetivamente.
   - *mergeLevels* $([[1]],[[4]]) \rightarrow [[1,4]]$.
   - O gene prefixa a raiz 3: $[3] : [[1,4]] \rightarrow [[3],[1,4]]$.
3. **Nível Intermédio (ex: nó 7):** De forma análoga, com os resultados $[[6]]$ e $[[8]]$:
   - *mergeLevels* $([[6]],[[8]]) \rightarrow [[6,8]]$.
   - O gene prefixa a raiz 7: $[7] : [[6,8]] \rightarrow [[7],[6,8]]$.
4. **Resultado Final (*levels* t_1):** A raiz principal 5 combina os resultados dos dois ramos anteriores:
   - *mergeLevels* $([[3],[1,4]],[[7],[6,8]]) \rightarrow [[3,7],[1,4,6,8]]$.
   - O gene finaliza com a raiz 5: $[5] : [[3,7],[1,4,6,8]] \rightarrow [[5],[3,7],[1,4,6,8]]$.

A aplicação final de *concat* lineariza esta estrutura estratificada e produz a sequência: $[5,3,7,1,4,6,8]$.

Anamorfismo ($[\!(\mathit{bft})\!]$): O anamorfismo opera de forma iterativa sobre uma fila de espera (*FIFO*), onde o estado é representado por uma floresta ($\mathit{State} = [\mathit{BTree}\ a]$). O processo desenrola-se através do consumo sucessivo do primeiro elemento da fila e da injeção dos seus descendentes no final da mesma:

1. **Estado Inicial (Fila):** $[t_1]$
   - O gene analisa a raiz de t_1 (valor 5) e emite-a para a lista de saída.
   - As sub-árvores t_3 (esquerda) e *t7* (direita) são colocadas no fim da fila (vazia nesta fase).
2. **Estado 1 (Fila):** $[t_3, t7]$
   - Consumo de t_3 (valor 3).
   - Os descendentes t_1 e t_4 são concatenados ao final da fila existente (*t7*).

- **Nova Fila:** $[t7, t_1, t_4]$.

3. **Estado 2 (Fila):** $[t7, t_1, t_4]$
   - Consumo de *t7* (valor 7).
   - Os descendentes *t6* e *t8* são colocados após t_4.
   - **Nova Fila:** $[t_1, t_4, t6, t8]$.
4. **Estados Seguintes (Folhas):** A fila contém agora apenas nós cujos descendentes são *Empty*.
   - Para cada nó $(1, 4, 6, 8)$, o valor é emitido e dois elementos *Empty* são adicionados à fila.
   - Perante um elemento *Empty*, o gene descarta-o e prossegue de imediato para o próximo nó, sem emissão de valor, até à exaustão total da fila.

O resultado final é a sequência linear $[5, 3, 7, 1, 4, 6, 8]$, onde a ordem de emissão corresponde estritamente à profundidade dos nós na árvore original.

Conclusão da Verificação: A análise demonstra que ambos os processos convergem para o mesmo resultado linear. Enquanto o catamorfismo organiza a informação por níveis de profundidade antes da junção, o anamorfismo explora a árvore através da gestão dinâmica de uma fila. Esta equivalência valida a complementaridade entre a desconstrução estrutural e a geração comportamental.

Validação exaustiva

Para validar a robustez da implementação, comparamos os resultados produzidos por *bfsLevels* e *bft* com os valores esperados para as árvores de teste t_1 a t_5:

$$
\begin{aligned}
&exp\_t1 :: [Int] \\
&exp\_t1 = [5, 3, 7, 1, 4, 6, 8] \\
&exp\_t2 :: [Int] \\
&exp\_t2 = [1, 2, 3, 4, 5, 6, 7] \\
&exp\_t3 :: [Char] \\
&exp\_t3 = [\texttt{'A'}, \texttt{'B'}, \texttt{'E'}, \texttt{'C'}, \texttt{'D'}] \\
&exp\_t4 :: [Char] \\
&exp\_t4 = [\texttt{'A'}, \texttt{'B'}, \texttt{'C'}, \texttt{'D'}] \\
&exp\_t5 :: [Int] \\
&exp\_t5 = [1, 2, 3, 4, 5, 6] \\
&test\_t1 = bfsLevels\ t_1 \equiv exp\_t1 \wedge bft\ t_1 \equiv exp\_t1 \\
&test\_t2 = bfsLevels\ t_2 \equiv exp\_t2 \wedge bft\ t_2 \equiv exp\_t2 \\
&test\_t3 = bfsLevels\ t_3 \equiv exp\_t3 \wedge bft\ t_3 \equiv exp\_t3 \\
&test\_t4 = bfsLevels\ t_4 \equiv exp\_t4 \wedge bft\ t_4 \equiv exp\_t4 \\
&test\_t5 = bfsLevels\ t_5 \equiv exp\_t5 \wedge bft\ t_5 \equiv exp\_t5 \\
&tests\_P1 :: [(String, Bool)] \\
&tests\_P1 = \\
&\quad [(\texttt{"t1"}, test\_t1) \\
&\quad , (\texttt{"t2"}, test\_t2) \\
&\quad , (\texttt{"t3"}, test\_t3) \\
&\quad , (\texttt{"t4"}, test\_t4) \\
&\quad , (\texttt{"t5"}, test\_t5) \\
&\quad]
\end{aligned}
$$

$$all\_tests\_P1 :: Bool$$
$$all\_tests\_P1 = and\ (\text{map}\ \ \pi_2\ tests\_P1)$$

A execução em GHCi confirma o sucesso de todos os testes, resultando em True para a função *all_tests_P1*. Desta forma, verifica-se que ambas as definições produzem a travessia breadth-first correta.

Problema 2

Introdução e objetivo

Pretende-se justificar (derivar) a função Haskell fornecida no enunciado a partir da série de Taylor de $\sinh x$, de modo a explicar o significado de cada componente do estado e provar, por indução, que a função calculada coincide com as somas parciais dessa série.

A função dada é:

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

Como $wrapper = head$, temos imediatamente:

$$f\ x\ n\ =\ head(worker\ n).$$

Logo, basta compreender e justificar o primeiro componente do estado $[s,h,k,j,m]$.

1. Série de Taylor de $\sinh x$ e somas parciais

Recorde-se a expansão de Taylor em torno de 0:

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

A aproximação com n iterações (ou n passos de refinamento) é a soma parcial

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

O objetivo passa a ser provar que:

$$f\ x\ n\ =\ S_n(x).$$

2. Ideia do algoritmo: acumular a soma e gerar o próximo termo

O cálculo de $S_n(x)$ pode ser visto como um processo iterativo:

- manter um acumulador s com a soma já obtida;

- em cada passo adicionar o próximo termo da série;
- atualizar esse "próximo termo" para o passo seguinte.

No código, essa lógica está explícita na primeira componente de `loop`:

$$s' \;=\; s + \frac{h}{k}.$$

Isto sugere a interpretação:

$\frac{h}{k}$ é o próximo termo a acrescentar à soma s.

Assim, o estado $[s, h, k, j, m]$ foi escolhido para que:

1. s seja a soma parcial já computada;
2. $\frac{h}{k}$ seja o termo seguinte da série;
3. os restantes componentes (j, m) permitam atualizar k (o fatorial) de forma barata, sem recalcular fatoriais do início.

3. Derivação das Leis de Atualização

Para justificar formalmente os valores do estado e as constantes, derivam-se as recorrências matematicamente.

Seja $t_i = \frac{x^{2i+1}}{(2i+1)!}$ o termo geral. Calcula-se a razão entre o termo seguinte (t_{i+1}) e o atual (t_i) dividindo as frações correspondentes:

$$\frac{t_{i+1}}{t_i} = \frac{\frac{x^{2i+3}}{(2i+3)!}}{\frac{x^{2i+1}}{(2i+1)!}} = \frac{x^{2i+3}}{(2i+3)!} \times \frac{(2i+1)!}{x^{2i+1}} = \frac{x^{2i+3}}{x^{2i+1}} \times \frac{(2i+1)!}{(2i+3)(2i+2)(2i+1)!}$$
$$= x^{(2i+3)-(2i+1)} \times \frac{1}{(2i+3)(2i+2)} = \frac{x^2}{(2i+3)(2i+2)}.$$

Desta razão resulta a relação de recorrência utilizada no algoritmo:

$$t_{i+1} \;=\; t_i \times \frac{x^2}{(2i+3)(2i+2)}.$$

Daqui resulta a necessidade de manter o numerador h e o denominador k separadamente. Aplicando a Lei da Recursividade Mútua (Lei de Fokkinga), decompõe-se a recorrência nas atualizações do estado:

- $h_{next} = h \cdot x^2$ (o numerador multiplica-se por x^2);
- $k_{next} = k \cdot (2i+3)(2i+2)$ (o denominador multiplica-se pelo polinómio quadrático).

Otimização por Diferenças Finitas (Cálculo de j e m) Para evitar multiplicações complexas dependentes do índice i no cálculo de $(2i+3)(2i+2)$, aplica-se o Cálculo de Diferenças Finitas. O fator de atualização do fatorial para o passo seguinte implica o polinómio:

$$P(n) = (2n+5)(2n+4) = 4n^2 + 18n + 20$$

(Nota: Substitui-se i por $n+1$, logo os fatores tornam-se $2(n+1)+3 = 2n+5$ e $2(n+1)+2 = 2n+4$).

Calculam-se as diferenças sucessivas para reduzir a avaliação polinomial a somas simples:

1. **Primeira Diferença (m_n):**

$$m_n = P(n+1) - P(n) = (4(n+1)^2 + 18(n+1) + 20) - (4n^2 + 18n + 20) = 8n + 22.$$

2. **Segunda Diferença (Δm):**

$$\Delta m = m_{n+1} - m_n = (8(n+1) + 22) - (8n + 22) = \mathbf{8}.$$

Visto que a segunda diferença é constante, justifica-se a introdução das variáveis auxiliares *j* (o valor acumulado do polinómio) e m (a primeira diferença), resultando nas atualizações lineares presentes no código:

$$j' = j + m \quad \text{e} \quad m' = m + 8.$$

Derivação dos Valores Iniciais (*start*): Para a iteração $n = 0$, o estado deve conter a soma atual e preparar o termo seguinte (devido à natureza *look-ahead* do ciclo, que calcula valores para a iteração $n + 1$):

- **Soma atual (s_0):** Corresponde ao primeiro termo da série ($i = 0$):

$$s_0 = \frac{x^{2(0)+1}}{(2(0)+1)!} = \frac{x^1}{1!} = x.$$

- **Próximo Numerador (h_0) e Denominador (k_0):** Correspondem ao termo para $i = 1$ (o termo que será somado na próxima execução do corpo do ciclo):

$$t_1 = \frac{x^{2(1)+1}}{(2(1)+1)!} = \frac{x^3}{3!} = \frac{x^3}{6}.$$

  Logo, $h_0 = x^3$ e $k_0 = 6$.

- **Variáveis Auxiliares (j_0, m_0):** Calculadas pelas fórmulas de diferenças finitas deduzidas acima, para $n = 0$:

$$j_0 = P(0) = 4(0)^2 + 18(0) + 20 = \mathbf{20}.$$

$$m_0 = 8(0) + 22 = \mathbf{22}.$$

Este cálculo fundamenta a origem exata de todos os componentes do vetor inicial $[x, x\uparrow 3, 6, 20, 22]$.

4. Invariante de correção

Define-se o invariante central que justifica a devolução das somas parciais corretas por parte da função *head*.

Invariante $I(n)$. Se *worker* $n = [s, h, k, j, m]$, então:

$$\text{(I1)} \quad s = \sum_{i=0}^{n} \frac{x^{2i+1}}{(2i+1)!} = S_n(x),$$

$$\text{(I2)} \quad \frac{h}{k} = \frac{x^{2n+3}}{(2n+3)!} \quad \text{(o termo seguinte a acrescentar).}$$

Note-se que (I1) é suficiente para a prova, uma vez que:

$$f\ x\ n = head(worker\ n) = s = S_n(x).$$

5. Prova por indução sobre n

Base ($n = 0$). Temos *worker* $0 = start\ x = [x, x^3, 6, 20, 22]$. Logo:

$$s = x = \frac{x^1}{1!} = S_0(x),$$

pelo que (I1) se verifica.

Além disso:

$$\frac{h}{k} = \frac{x^3}{6} = \frac{x^3}{3!},$$

valor que coincide com o termo seguinte a S_0. Logo, (I2) também se verifica.

Passo indutivo ($n \to n+1$). Sob a hipótese de que $I(n)$ é verdadeiro para *worker* $n = [s, h, k, j, m]$, tem-se:

$$worker(n+1) = loop\ x\ [s, h, k, j, m] = [s', h', k', j', m'].$$

Pelo código:

$$s' = s + \frac{h}{k}.$$

Com recurso a (I1) e (I2), obtém-se:

$$s' = S_n(x) + \frac{x^{2n+3}}{(2n+3)!} = S_{n+1}(x),$$

logo (I1) é válido para $n+1$.

Para (I2), o programa atualiza $h' = x^2 h$, o que faz avançar o numerador do termo seguinte de x^{2n+3} para x^{2n+5}. Por construção dos componentes (k, j, m), o valor k' passa do fatorial ímpar $(2n+3)!$ para $(2n+5)!$. Assim:

$$\frac{h'}{k'} = \frac{x^{2n+5}}{(2n+5)!},$$

ou seja, exatamente o termo seguinte a $S_{n+1}(x)$. Portanto, $I(n+1)$ verifica-se.

Conclui-se por indução que $I(n)$ é válido para todo n. Consequentemente:

$$f\ x\ n = head(worker\ n) = S_n(x),$$

isto é, $f\ x\ n$ calcula a aproximação de $\sinh x$ por n iterações (somas parciais da série).

6. Mini-exemplo numérico: $x = 1$ (primeiros estados)

Para $x = 1$, tem-se $1^2 = 1$. Logo, h mantém-se em 1 com o arranque em $h = 1^3 = 1$. O denominador k (fatoriais ímpares) sofre as alterações principais.

Estado 0:

$$worker\ 0 = [1,\ 1,\ 6,\ 20,\ 22]$$

$f\ 1\ 0 = s = 1$ (ou seja, $1 = 1/1!$).

Estado 1:

$$s' = 1 + 1/6 = 7/6 \approx 1.166666...$$

$$worker\ 1 = [7/6,\ 1,\ 120,\ 42,\ 30]$$

Logo:

$$f\ 1\ 1 = 7/6 = 1 + 1/3!$$

Estado 2:

$$s'' = 7/6 + 1/120 = 141/120 = 47/40 = 1.175$$

$$worker\ 2 = [47/40,\ 1,\ 5040,\ 72,\ 38]$$

Logo:

$$f\ 1\ 2 = 47/40 = 1 + 1/3! + 1/5!$$

Conclusão

A função `worker` mantém um estado $[s, h, k, j, m]$ onde:

- s é a soma parcial já acumulada;
- $\frac{h}{k}$ é o próximo termo da série a adicionar;
- as atualizações de h e de k garantem a progressão para o termo seguinte sem recomputações dispendiosas.

Como `f x n = head (worker n)`, conclui-se que `f x n` devolve a aproximação de $\sinh x$ por n iterações da série de Taylor.

Problema 3

Introdução: Streams e Processos Infinitos

Ao contrário das estruturas de dados finitas, os *Streams* modelam sequências infinitas de dados, fundamentais na modelação de processos reativos e fluxos contínuos. No contexto de Haskell, esta estrutura define-se pelo tipo indutivo:

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

A semântica desta estrutura é ditada pela assinatura do functor $FX = A \times X$. O correspondente destrutor, ou estrutura, define-se por:

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

A construção de instâncias desta estrutura realiza-se via anamorfismos ($[\![\cdot]\!]$), que permitem a expansão de um estado inicial num fluxo infinito através de um gene gerador g:

$$[\![g]\!] = Cons \cdot (id \times [\![g]\!]) \cdot g$$

O presente problema foca-se na implementação de uma fusão justa (*fair_merge*). O objetivo é a garantia de que os elementos de dois fluxos de entrada sejam intercalados de forma estrita, o que previne fenómenos de inanição (*starvation*) e assegura a vivacidade do sistema.

Dedução da Lei de Recursividade Mútua (Dual)

A modelação da alternância estrita entre dois fluxos independentes sugere uma dependência mútua. Para a formalização desta dinâmica sem recurso a recursividade explícita, aplica-se a **Lei de Fokkinga (Dual)**, ou Lei da Recursividade Mútua para Anamorfismos.

O objetivo desta derivação consiste em determinar o gene g que satisfaz a definição de $fair\_merge'$ como um anamorfismo. Partimos da premissa:

$$fair\_merge' = [\!(\, g \,)\!]$$

Considera-se a definição recursiva original fornecida no enunciado, onde se observa que as funções h e k dependem ciclicamente uma da outra para o processamento do elemento seguinte:

$$\begin{aligned}
&fair_merge :: (Stream\ a, Stream\ a) + (Stream\ a, Stream\ a) \to Stream\ a \\
&fair_merge = [h, k] \ \textbf{where} \\
&\quad h\ (Cons\ (x, xs), y) = Cons\ (x, k\ (xs, y)) \\
&\quad k\ (x, Cons\ (y, ys)) = Cons\ (y, h\ (x, ys))
\end{aligned}$$

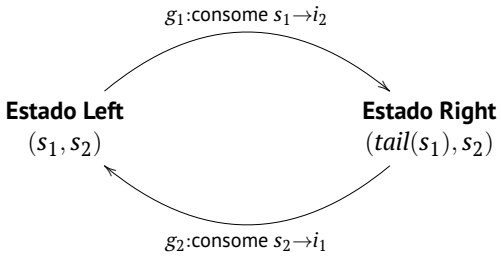
A derivação formal, baseada na propriedade universal dos anamorfismos e nas leis de fusão do coproduto do cálculo de programas, segue abaixo para provar que $[h, k] = [\!(\, [g1, g2] \,)\!]$:

$$\begin{aligned}
& r = [\!(\, gene \,)\!] \\
\equiv & \quad \{\ \text{r = [h,k] e gene = [g1, g2]}\ \} \\
& [h, k] = [\!(\, [g1, g2] \,)\!] \\
\equiv & \quad \{\ \text{Universal - Ana (56)}\ \} \\
& out \cdot [h, k] = F\ [h, k] \cdot [g1, g2] \\
\equiv & \quad \{\ \text{Fusão-+ (21)}\ \} \\
& [out \cdot h, out \cdot k] = [F\ [h, k] \cdot g1, F\ [h, k] \cdot g2] \\
\equiv & \quad \{\ \text{Eq-+ (28)}\ \} \\
& \begin{cases} out \cdot h = F\ [h, k] \cdot g1 \\ out \cdot k = F\ [h, k] \cdot g2 \end{cases} \\
\equiv & \quad \{\ \text{F (either h k) = id x (either h k)}\ \} \\
& \begin{cases} out \cdot h = (id \times [h, k]) \cdot g1 \\ out \cdot k = (id \times [h, k]) \cdot g2 \end{cases} \\
\square &
\end{aligned}$$

Análise do Sistema e Estratégia de Implementação

O sistema de equações resultante estabelece as condições de correção para o gene do anamorfismo. As igualdades obtidas indicam que cada componente do gene (g_1 e g_2) deve produzir um par contendo o valor de saída imediata e o estado necessário para a iteração seguinte.

Fundamentalmente, este resultado demonstra que, para manter a alternância estrita, o gene deve realizar uma "troca de prioridade": sempre que a componente g_1 (vinda de h) consome um elemento, o estado sucessor deve ser injetado de forma a que a componente g_2 (vinda de k) seja a próxima a ser executada. Este comportamento é modelável através de uma máquina de estados finita:



Análise do Fluxo de Dados: Nesta estrutura, as variáveis representam o estado dos fluxos em cada passo:

- No **Estado Left**, a função g_1 extrai a cabeça de s_1 e preserva s_2 intacto, passando o par resultante para o lado direito através da injeção i_2.
- No **Estado Right**, a prioridade inverte-se: g_2 consome a cabeça de s_2 e devolve o controlo ao lado esquerdo via i_1.

Esta alternância contínua impede a inanição (*starvation*) de qualquer um dos fluxos, resultando numa fusão perfeitamente equilibrada.

Desta forma, os diagramas categoriais seguintes detalham a composição interna de cada componente do gene. Estes diagramas seguem rigorosamente as definições de g_1 e g_2 e evidenciam a desconstrução do estado para a alternância de prioridade.

Gene g_1 (Processamento de s_1): Este componente extrai a cabeça do primeiro fluxo e encapsula o estado sucessor com a injeção i_2. Este passo transfere a prioridade para o segundo fluxo na iteração seguinte.

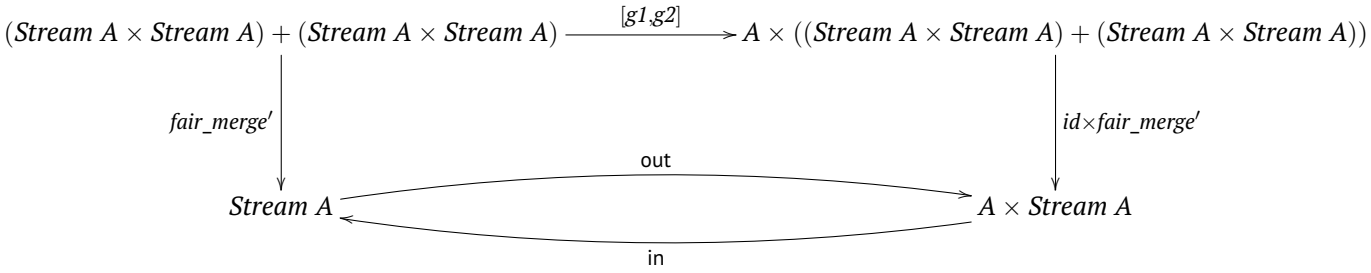
$$\begin{array}{ccc} \mathit{Stream}\ A \times \mathit{Stream}\ A & \xrightarrow{\langle \pi_1 \cdot out \cdot \pi_1, \langle \pi_2 \cdot out \cdot \pi_1, \pi_2 \rangle \rangle} & A \times (\mathit{Stream}\ A \times \mathit{Stream}\ A) \\ & \searrow^{g_1} & \downarrow{\scriptstyle id \times i_2} \\ & & A \times ((\mathit{Stream}\ A \times \mathit{Stream}\ A) + (\mathit{Stream}\ A \times \mathit{Stream}\ A)) \end{array}$$

Gene g_2 (Processamento de s_2): Simetricamente, este componente consome o elemento do segundo fluxo e utiliza a injeção i_1. Desta forma, devolve a prioridade ao primeiro fluxo.

$$\begin{array}{ccc} \mathit{Stream}\ A \times \mathit{Stream}\ A & \xrightarrow{\langle \pi_1 \cdot out \cdot \pi_2, \langle \pi_1, \pi_2 \cdot out \cdot \pi_2 \rangle \rangle} & A \times (\mathit{Stream}\ A \times \mathit{Stream}\ A) \\ & \searrow^{g_2} & \downarrow{\scriptstyle id \times i_1} \\ & & A \times ((\mathit{Stream}\ A \times \mathit{Stream}\ A) + (\mathit{Stream}\ A \times \mathit{Stream}\ A)) \end{array}$$

O Diagrama do Anamorfismo

O processo pode ser visualizado através do seguinte diagrama comutativo. O gene $g = [g_1, g_2]$ mapeia o estado atual (uma soma de pares de streams) no par $(\text{valor}, \text{próximo estado})$, onde o functor $Ff = id \times f$ assegura a continuidade da expansão infinita:



Este diagrama explicita a semântica estrutura: o anamorfismo "desenrola" o estado inicial aplicando o gene sucessivamente. A "justiça" da fusão não reside na estrutura do anamorfismo em si, mas na definição interna de g_1 e g_2, que forçam a alternância entre as injeções i_1 e i_2 do tipo soma.

Implementação em Haskell

A tradução deste formalismo para código Haskell resulta na seguinte definição *point-free*, onde a separação de responsabilidades é total: o combinador $[\![\cdot]\!]$ gere a recursividade infinita, enquanto o gene g gere a lógica de intercalação.

$$
\begin{aligned}
&\mathit{fair\_merge'} :: (\mathit{Stream}\ A, \mathit{Stream}\ A) + (\mathit{Stream}\ A, \mathit{Stream}\ A) \to \mathit{Stream}\ A \\
&\mathit{fair\_merge'} = [\![g]\!] \\
&\quad \textbf{where} \\
&\qquad g = [g1, g2] \\
&\qquad g1 = (id \times i_2) \cdot \langle \pi_1 \cdot \mathsf{out} \cdot \pi_1, \langle \pi_2 \cdot \mathsf{out} \cdot \pi_1, \pi_2 \rangle \rangle \\
&\qquad g2 = (id \times i_1) \cdot \langle \pi_1 \cdot \mathsf{out} \cdot \pi_2, \langle \pi_1, \pi_2 \cdot \mathsf{out} \cdot \pi_2 \rangle \rangle
\end{aligned}
$$

Validação de Resultados e Simulação

Para validar a implementação do *fair_merge′*, utilizam-se fluxos aritméticos gerados via anamorfismo. Define-se a função auxiliar *nextS* para calcular o par (valor, próximo estado), o que evita conflitos de formatação:

$$
\begin{aligned}
&\mathit{nextS}\ n = (n, n+4) \\
&s1 = [\![\mathit{nextS}]\!]\ 1 \quad \text{-- [1, 5, 9, 13, ...]} \\
&s2 = [\![\mathit{nextS}]\!]\ 2 \quad \text{-- [2, 6, 10, 14, ...]} \\
&s3 = [\![\mathit{nextS}]\!]\ 3 \quad \text{-- [3, 7, 11, 15, ...]} \\
&s4 = [\![\mathit{nextS}]\!]\ 4 \quad \text{-- [4, 8, 12, 16, ...]} \\
&\mathit{takeS}\ 0\ \_ = [\,] \\
&\mathit{takeS}\ n\ (\mathit{Cons}\ (x, xs)) = x : \mathit{takeS}\ (n-1)\ xs
\end{aligned}
$$

A função *takeS* extrai os primeiros n elementos de um *Stream*, permitindo a observação finita dos resultados.

A simulação adota uma estrutura em cascata: fundem-se dois pares de fluxos e, posteriormente, combinam-se os resultados dessas operações. Esta composição verifica a robustez da máquina de estados perante fluxos aninhados:

$$
\begin{aligned}
&f1 = \mathit{takeS}\ 10\ (\mathit{fair\_merge}\ (i_1\ (\mathit{fair\_merge}\ (i_1\ (s1, s3)), \mathit{fair\_merge}\ (i_1\ (s2, s4))))) \\
&f2 = \mathit{takeS}\ 10\ (\mathit{fair\_merge'}\ (i_1\ (\mathit{fair\_merge'}\ (i_1\ (s1, s3)), \mathit{fair\_merge'}\ (i_1\ (s2, s4)))))
\end{aligned}
$$

Análise do Resultado da Simulação:

Ao executar *f1* ou *f2*, obtém-se, em ambos os casos, a sequência: $[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$

Este resultado comprova a correção do operador através da seguinte lógica de junção:

- A fusão da esquerda combina *s1* e *s3*, intercalando os seus elementos para formar a sequência de ímpares: $[1, 3, 5, 7, 9, \ldots]$.
- A fusão da direita combina *s2* e *s4*, intercalando os seus elementos para formar a sequência de pares: $[2, 4, 6, 8, 10, \ldots]$.
- A fusão final intercala estas duas sequências (ímpares e pares), o que resulta na sequência numérica contínua.

A igualdade verificada entre *f1* e *f2* confirma que a implementação baseada no gene respeita a semântica de fusão justa, garantindo que nenhum fluxo é preferido na estrutura de cascata.

Problema 4

Introdução e Modelação

O problema proposto consiste na modelação de um sistema de comunicação falível, um telegrafista com uma avaria intermitente, através de um catamorfismo probabilístico. O objetivo é calcular a distribuição de probabilidades das mensagens recebidas dada a frase original "Vamos atacar hoje".

O comportamento do sistema caracteriza-se por dois tipos de falha independente:

1. **Falha na transmissão de palavras:** Cada palavra tem uma probabilidade de 5% de se perder durante a transmissão.
2. **Falha na terminação:** O código de fim de mensagem (*stop*) deve ser enviado no final, mas falha em 10% das vezes.

Para modelar este comportamento, recorre-se a uma função de ordem superior, *pcataList*, que generaliza o conceito de *fold* para o contexto do mónade de probabilidades Dist.

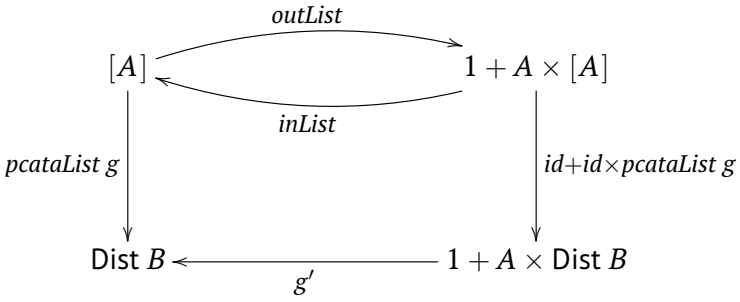
1. O Catamorfismo Probabilístico (*pcataList*)

O combinador *pcataList* define-se como um catamorfismo sobre listas cujo gene produz resultados probabilísticos. Matematicamente, enquanto um catamorfismo de listas tradicional tem o tipo $(1 + A \times B \to B) \to [A] \to B$, a versão probabilística opera no mónade Dist, tendo o tipo $(1 + A \times B \to \text{Dist } B) \to [A] \to \text{Dist } B$.

A implementação reflete a estrutura algébrica da lista (soma de produtos), tratando explicitamente o caso vazio (injeção i_1) e o caso não-vazio (injeção i_2):

$$
\begin{aligned}
&\mathit{pcataList} :: ((\,) + (a, b) \to \text{Dist } b) \to [a] \to \text{Dist } b \\
&\mathit{pcataList}\ g\ [\,] = g\ (i_1\ (\,)) \\
&\mathit{pcataList}\ g\ (a : as) = \mathbf{do} \\
&\quad b \leftarrow \mathit{pcataList}\ g\ as \\
&\quad g\ (i_2\ (a, b))
\end{aligned}
$$

O funcionamento estrutural deste combinador visualiza-se através do seguinte diagrama comutativo:



Neste esquema, a seta inferior g' denota a aplicação monádica do gene. Ao contrário de um catamorfismo determinístico, a recursão devolve uma distribuição (Dist B). Consequentemente, g' incorpora a lógica de sequenciação (o operador *bind* implícito na notação *do*), extraindo o resultado da cauda antes da aplicação efetiva de g.

O processo de avaliação detalhado ocorre da seguinte forma:

- **Caso Base ($[\,]$):** O catamorfismo atinge o fim da lista e invoca o gene com $i_1()$, permitindo ao gene decidir como terminar a estrutura (neste caso, decidindo se envia "stop").
- **Passo Recursivo ($a : as$):**
  1. Avalia-se primeiramente a cauda da lista (as), o que resulta numa distribuição de possíveis caudas processadas (b).
  2. Para cada resultado possível dessa distribuição, aplica-se o gene à cabeça atual (a) emparelhada com esse resultado ($i_2(a,b)$).
  3. O mónade Dist encarrega-se de combinar as probabilidades resultantes (multiplicação de probabilidades para eventos independentes).

2. Definição do Gene Probabilístico

O gene captura o comportamento específico da falha. Este recebe um co-produto que representa os dois estados possíveis da travessia da lista. A função *gene* define-se pelo combinador $[\cdot,\cdot]$, separando a lógica de terminação da lógica de transmissão:

$$
\begin{aligned}
&\mathit{gene} :: () + (\mathit{String}, [\mathit{String}]) \to \text{Dist } [\mathit{String}] \\
&\mathit{gene} = [\mathit{base}, \mathit{step}] \\
&\quad \textbf{where} \\
&\qquad \mathit{base} :: () \to \text{Dist } [\mathit{String}] \\
&\qquad \mathit{base}\ () = D\ [([\texttt{"stop"}], 0.90), ([\,], 0.10)] \\
&\qquad \mathit{step} :: (\mathit{String}, [\mathit{String}]) \to \text{Dist } [\mathit{String}] \\
&\qquad \mathit{step}\ (w, ws) = D\ [(ws, 0.05), (w : ws, 0.95)]
\end{aligned}
$$

A lógica de decisão pode ser visualizada na árvore de decisão probabilística (Figura 1).

Assim, a função de transmissão final obtém-se pela aplicação do catamorfismo com este gene:

$$
\begin{aligned}
&\mathit{transmitir} :: [\mathit{String}] \to \text{Dist } [\mathit{String}] \\
&\mathit{transmitir} = \mathit{pcataList}\ \mathit{gene}
\end{aligned}
$$

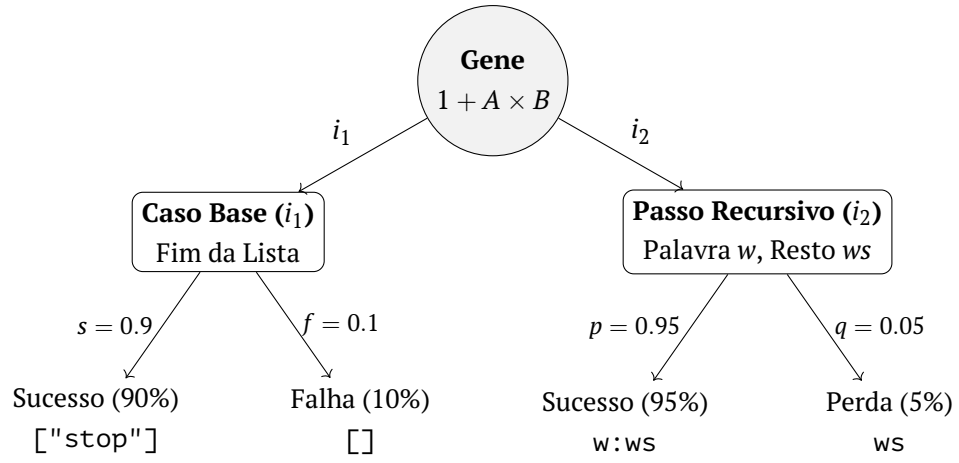


Figure 1: Diagrama de decisão do Gene: bifurcação entre terminação (i_1) e processamento (i_2).

3. Cálculo de Probabilidades

Considere-se a mensagem de entrada $M = [\texttt{"Vamos"}, \texttt{"atacar"}, \texttt{"hoje"}]$. Definem-se as probabilidades elementares: $p = 0.95$ (manter), $q = 0.05$ (perder), $s = 0.90$ (stop), $f = 0.10$ (sem stop).

Dado que o mónade assegura a independência estatística, calculam-se analiticamente os cenários:

Cenário 1: Perder "atacar" ($[\texttt{"Vamos"}, \texttt{"hoje"}, \texttt{"stop"}]$). Eventos: "Vamos" segue (p), "atacar" falha (q), "hoje" segue (p), "stop" segue (s).

$$P_1 = p \cdot q \cdot p \cdot s = 0.95^2 \cdot 0.05 \cdot 0.90 \approx \mathbf{4.06}\%$$

Cenário 2: Faltar o "stop" ($[\texttt{"Vamos"}, \texttt{"atacar"}, \texttt{"hoje"}]$). Eventos: Três palavras seguem (p^3), "stop" falha (f).

$$P_2 = p^3 \cdot f = 0.95^3 \cdot 0.10 \approx \mathbf{8.57}\%$$

Cenário 3: Perfeita ($[\texttt{"Vamos"}, \texttt{"atacar"}, \texttt{"hoje"}, \texttt{"stop"}]$). Eventos: Tudo segue com sucesso.

$$P_3 = p^3 \cdot s = 0.95^3 \cdot 0.90 \approx \mathbf{77.16}\%$$

4. Validação Experimental

A validação dos cálculos analíticos é realizada através da execução direta da função *transmitir* com a frase do enunciado.

```
msg :: [String]
msg = words "Vamos atacar hoje"
```

A invocação de *transmitir msg* no interpretador gera a distribuição de todas as mensagens possíveis. O resultado abaixo (formatado automaticamente pela biblioteca *Probability*) apresenta os cenários por ordem decrescente de probabilidade:

```
*Problema4> transmitir msg
["Vamos","atacar","hoje","stop"]  77.2%
       ["Vamos","atacar","hoje"]   8.6%
        ["atacar","hoje","stop"]   4.1%
```

```
["Vamos","atacar","stop"]   4.1%
 ["Vamos","hoje","stop"]   4.1%
      ["Vamos","atacar"]   0.5%
...
```

Análise dos Resultados: Ao comparar este *output* com os valores teóricos calculados na secção anterior, confirma-se a correção do modelo:

1. O cenário de **transmissão perfeita** (`["Vamos","atacar","hoje","stop"]`) surge com 77.2%, correspondendo ao valor calculado $P_3 \approx 77.16\%$.
2. O cenário onde **falta o "stop"** (`["Vamos","atacar","hoje"]`) surge com 8.6%, correspondendo a $P_2 \approx 8.57\%$.
3. O cenário onde se **perde a palavra "atacar"** (`["Vamos","hoje","stop"]`) surge com 4.1%, correspondendo a $P_1 \approx 4.06\%$.

Nota: A ligeira diferença deve-se apenas ao arredondamento para uma casa decimal na visualização do interpretador.

Index

References

[1] M. Erwig and S. Kollmansberger. Functional pearls: Probabilistic functional programming in haskell. *J. Funct. Program.*, 16:21–34, January 2006.

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

[3] 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.

[4] 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).