



Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2024/25)

Lic. em Engenharia Informática

Grupo 7

a95485	Miguel Torres Carvalho
a96587	Flávia Alexandra da Silva Araújo
a104001	Frederico Cunha Afonso

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.

Problema 1

Esta questão aborda um problema que é conhecido pela designação '*H-index of a Histogram*' e que se formula facilmente:

O h-index de um histograma é o maior número n de barras do histograma cuja altura é maior ou igual a n .

Por exemplo, o histograma

$$h = [5, 2, 7, 1, 8, 6, 4, 9]$$

que se mostra na figura



tem *hindex* $h = 5$ pois há 5 colunas maiores que 5. (Não é 6 pois maiores ou iguais que seis só há quatro.)

Pretende-se definida como um catamorfismo, anamorfismo ou hilomorfismo uma função em Haskell

$$\text{hindex} :: [Int] \rightarrow (Int, [Int])$$

tal que, para $(i, x) = \text{hindex } h$, i é o H-index de h e x é a lista de colunas de h que para ele contribuem.

A proposta de *hindex* deverá vir acompanhada de um **diagrama** ilustrativo.

Problema 2

Pelo [teorema fundamental da aritmética](#), todo número inteiro positivo tem uma única factorização prima. Por exemplo,

```
primes 455
[5,7,13]
primes 433
[433]
primes 230
[2,5,23]
```

1. Implemente como anamorfismo de listas a função

$primes :: \mathbb{Z} \rightarrow [\mathbb{Z}]$

que deverá, recebendo um número inteiro positivo, devolver a respectiva lista de factores primos.

A proposta de *primes* deverá vir acompanhada de um **diagrama** ilustrativo.

2. A figura mostra a “*árvore dos primos*” dos números [455, 669, 6645, 34, 12, 2].



Com base na alínea anterior, implemente uma função em Haskell que faça a geração de uma tal árvore a partir de uma lista de inteiros:

$prime_tree :: [\mathbb{Z}] \rightarrow Exp \mathbb{Z} \mathbb{Z}$

Sugestão: escreva o mínimo de código possível em *prime_tree* investigando cuidadosamente que funções disponíveis nas bibliotecas que são dadas podem ser reutilizadas.¹

Problema 3

A convolução $a \star b$ de duas listas a e b — uma operação relevante em computação — está muito bem explicada [neste vídeo](#) do canal **3Blue1Brown** do YouTube, a partir de $t = 6 : 30$. Aí se mostra como, por exemplo:

¹ Pense sempre na sua produtividade quando está a programar — essa atitude será valorizada por qualquer empregador que vier a ter.

$$[1, 2, 3] \star [4, 5, 6] = [4, 13, 28, 27, 18]$$

A solução abaixo, proposta pelo chatGPT,

```
convolve :: Num a => [a] -> [a] -> [a]
convolve xs ys = [sum $ zipWith (*) (take n (drop i xs)) ys | i <- [0..(length xs - n)]]
  where n = length ys
```

está manifestamente errada, pois $\text{convolve } [1, 2, 3] [4, 5, 6] = [32]$ (!).

Proponha, explicando-a devidamente, uma solução sua para *convolve*. Valorizar-se-á a economia de código e o recurso aos combinadores *pointfree* estudados na disciplina, em particular a triologia *ana-cata-hilo* de tipos disponíveis nas bibliotecas dadas ou a definir.

Problema 4

Considere-se a seguinte sintaxe (abstrata e simplificada) para **expressões numéricas** (em *b*) com variáveis (em *a*),

```
data Expr b a = V a | N b | T Op [Expr b a] deriving (Show, Eq)
data Op = ITE | Add | Mul | Suc deriving (Show, Eq)
```

possivelmente condicionais (cf. *ITE*, i.e. o operador condicional “if-then-else”). Por exemplo, a árvore mostrada a seguir



representa a expressão

$$\text{ite } (V \text{ "x"}) (N \ 0) (\text{multi } (V \text{ "y"}) (\text{soma } (N \ 3) (V \text{ "y"}))) \quad (1)$$

– i.e. **if** *x* **then** 0 **else** *y* * (3 + *y*) – assumindo as “helper functions”:

```
soma x y = T Add [x, y]
multi x y = T Mul [x, y]
ite x y z = T ITE [x, y, z]
```

No anexo E propõe-se uma base para o tipo *Expr* (*baseExpr*) e a correspondente algebra *inExpr* para construção do tipo *Expr*.

1. Complete as restantes definições da biblioteca *Expr* pedidas no anexo F.
2. No mesmo anexo, declare *Expr b* como instância da classe *Monad*. **Sugestão:** relembre os exercícios da ficha 12.

3. Defina como um catamorfismo de *Expr* a sua versão monádica, que deverá ter o tipo:

$$mcataExpr :: Monad\ m \Rightarrow (a + (b + (Op, m\ [c]))) \rightarrow m\ c \rightarrow Expr\ b\ a \rightarrow m\ c$$

4. Para se avaliar uma expressão é preciso que todas as suas variáveis estejam instanciadas. Complete a definição da função

$$let_exp :: (Num\ c) \Rightarrow (a \rightarrow Expr\ c\ b) \rightarrow Expr\ c\ a \rightarrow Expr\ c\ b$$

que, dada uma expressão com variáveis em *a* e uma função que a cada uma dessas variáveis atribui uma expressão (*a* \rightarrow *Expr* *c* *b*), faz a correspondente substituição.¹ Por exemplo, dada

$$\begin{aligned} f\ "x" &= N\ 0 \\ f\ "y" &= N\ 5 \\ f\ _ &= N\ 99 \end{aligned}$$

ter-se-á

$$let_exp\ f\ e = T\ ITE\ [N\ 1, N\ 0, T\ Mul\ [N\ 5, T\ Add\ [N\ 3, N\ 1]]]$$

isto é, a árvore da figura a seguir:



5. Finalmente, defina a função de avaliação de uma expressão, com tipo

$$evaluate :: (Num\ a, Ord\ a) \Rightarrow Expr\ a\ b \rightarrow Maybe\ a$$

que deverá ter em conta as seguintes situações de erro:

- (a) *Variáveis* — para ser avaliada, *x* em *evaluate* *x* não pode conter variáveis. Assim, por exemplo,

$$\begin{aligned} evaluate\ e &= Nothing \\ evaluate\ (let_exp\ f\ e) &= Just\ 40 \end{aligned}$$

para *f* e *e* dadas acima.

- (b) *Aridades* — todas as ocorrências dos operadores deverão ter o devido número de sub-expressões, por exemplo:

$$\begin{aligned} evaluate\ (T\ Add\ [N\ 2, N\ 3]) &= Just\ 5 \\ evaluate\ (T\ Mul\ [N\ 2]) &= Nothing \end{aligned}$$

¹ Cf. expressões **let ... in...**

Sugestão: de novo se insiste na escrita do mínimo de código possível, tirando partido da riqueza estrutural do tipo *Expr* que é assunto desta questão. Sugere-se também o recurso a diagramas para explicar as soluções propostas.

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](#)” [?], 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 `cp2425t.pdf` que está a ler é já um exemplo de [programação literária](#): foi gerado a partir do texto fonte `cp2425t.lhs`¹ que encontrará no [material pedagógico](#) desta disciplina descompactando o ficheiro `cp2425t.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 `cp2425t.zip`. Este [container](#) deverá ser usado na execução do [GHCi](#) e dos comandos relativos ao [L^AT_EX](#). (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 cp2425t .  
$ docker run -v ${PWD}:/cp2425t -it cp2425t
```

NB: O objetivo é que o container seja usado *apenas* para executar o [GHCi](#) e os comandos relativos ao [L^AT_EX](#). Deste modo, é criado um *volume* (cf. a opção `-v ${PWD}:/cp2425t`) que permite que a diretoria em que se encontra na sua máquina local e a diretoria `/cp2425t` no [container](#) sejam partilhadas.

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

```
$ lhs2TeX cp2425t.lhs > cp2425t.tex  
$ pdflatex cp2425t
```

[lhs2TeX](#) é o pre-processor que faz “pretty printing” de código Haskell em [L^AT_EX](#) e que faz parte já do [container](#). Alternativamente, basta executar

```
$ make
```

para obter o mesmo efeito que acima.

Por outro lado, o mesmo ficheiro `cp2425t.lhs` é executável e contém o “kit” básico, escrito em [Haskell](#), para realizar o trabalho. Basta executar

```
$ ghci cp2425t.lhs
```

Abra o ficheiro `cp2425t.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 [F](#) 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 [Bib_TE_X](#)) e o índice remissivo (com [makeindex](#)),

```
$ bibtex cp2425t.aux  
$ makeindex cp2425t.idx
```

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

No anexo [E](#) 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 ler¹ onde se obtém o efeito seguinte:²

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

Os diagramas podem ser produzidos recorrendo à *package* [xymatrix](#), por exemplo:

$$\begin{array}{ccc}
 \mathbb{N}_0 & \xleftarrow{\text{in}} & 1 + \mathbb{N}_0 \\
 \text{\scriptsize $\langle g \rangle$} \downarrow & & \downarrow \text{\scriptsize $id + \langle g \rangle$} \\
 B & \xleftarrow{g} & 1 + B
 \end{array}$$

E Código fornecido

Problema 1

$h :: [Int]$

Problema 4

Definição do tipo:

$inExpr = [V, [N, \widehat{T}]]$
 $baseExpr\ g\ h\ f = g + (h + id \times \text{map}\ f)$

Exemplos de expressões:

$e = \text{ite}\ (V\ "x")\ (N\ 0)\ (\text{multi}\ (V\ "y")\ (\text{soma}\ (N\ 3)\ (V\ "y")))$
 $i = \text{ite}\ (V\ "x")\ (N\ 1)\ (\text{multi}\ (V\ "y")\ (\text{soma}\ (N\ (3 / 5))\ (V\ "y")))$

Exemplo de teste:

$\text{teste} = \text{evaluate}\ (\text{let_exp}\ f\ i) \equiv \text{Just}\ (26 / 245)$
 $\textbf{where}\ f\ "x" = N\ 0; f\ "y" = N\ (1 / 7)$

¹ Procure e.g. por "sec:diagramas".

² Exemplos tirados de [?].

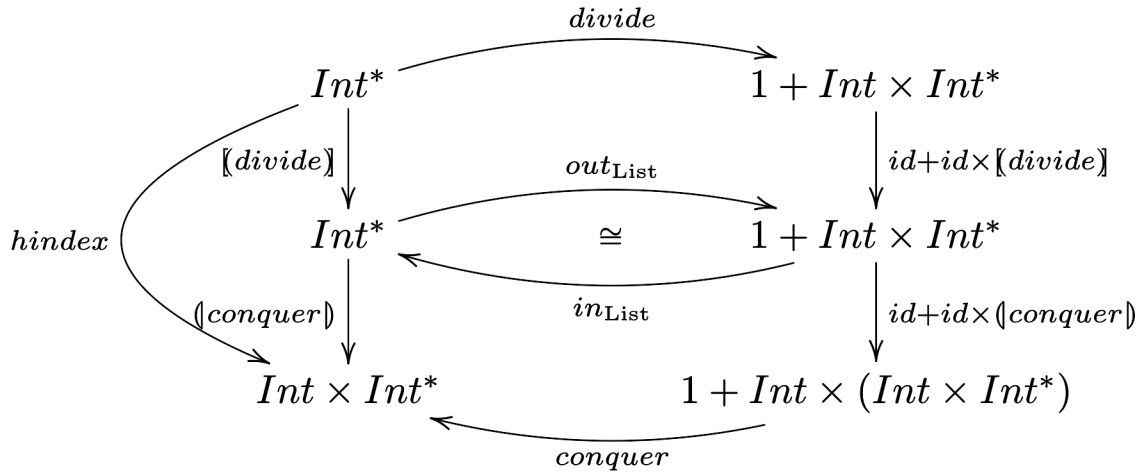
F Soluções dos alunos

Problema 1

O *h-index* de um histograma é o maior número n de barras do histograma cuja altura é maior ou igual a n .

Para a resolução deste problema, recorreu-se a uma estratégia de divisão e conquista com recurso a um hilomorfismo de listas, definido na biblioteca [List](#), com o anamorfismo responsável por ordenar a lista de *input* de forma crescente e o catamorfismo que, iterativamente, seleciona os elementos, incrementando o valor de *h-index*, em que n corresponde aos n elementos maiores ou iguais a n , sendo esta condição um invariante ao longo das iterações do catamorfismo. Graças à forma recursiva do catamorfismo, dada pelo functor do catamorfismo *conquer*, é possível iterar sobre a lista de forma decrescente, garantindo que o valor de *h-index* é o maior possível.

A estrutura de dados intermédia deste hilomorfismo é uma lista de inteiros, do tipo $[Int]$, esta correspondendo à saída do anamorfismo *divide* e à respetiva entrada do catamorfismo *conquer*.



$hindex = \llbracket conquer, divide \rrbracket$

where

$conquer = [g1, g2]$

$g1 = 0, []$

$g2 (h, t@(i, x))$

| $h > i = (i + 1, h : x)$

| $otherwise = t$

$divide [] = i_1 ()$

$divide xs = \mathbf{let} m = \mathbf{minimum} xs$

$\mathbf{in} i_2 (m, \mathbf{delete} m xs)$

Para além da solução apresentada, foi também desenvolvida uma solução alternativa que recorre somente a um catamorfismo de listas, modificado de forma a que a função *out* seja capaz, aquando a lista de entrada não é vazia, de dividir a lista em duas partes, uma com o valor mínimo e outra com o restante da lista. Deste modo, a funcionalidade do anamorfismo *divide* é incorporada na função *out*.

A seguinte solução alternativa é 100% *pointfree*, com a função *g2* do gene do catamorfismo redesenhada, assim como a função *out*, esta, como supramencionado, equivalente à função *divide* da solução principal.

$$\begin{array}{ccc}
 Int^* & \xrightarrow{outSortList} & 1 + Int \times Int^* \\
 \downarrow \scriptstyle hindex' = \llbracket conquer \rrbracket & & \downarrow \scriptstyle id + id \times hindex' \\
 Int \times Int^* & \xleftarrow{conquer} & 1 + Int \times (Int \times Int^*)
 \end{array}$$

$hindex' :: [Int] \rightarrow (Int, [Int])$

$hindex' = \llbracket conquer \rrbracket$

where

$conquer = [g1, g2]$

$g1 = \underline{0}, []$

$g2 = \underline{cond} \left((\widehat{(>)}) \cdot (id \times \pi_1) \right)$
 $\langle succ \cdot \pi_1 \cdot \pi_2, cons \cdot (id \times \pi_2) \rangle$
 π_2

$outSortList = cond ([] \equiv)$

$(i_1 \cdot (!))$

$(i_2 \cdot \langle \pi_1, \widehat{delete} \rangle \cdot \langle minimum, id \rangle)$

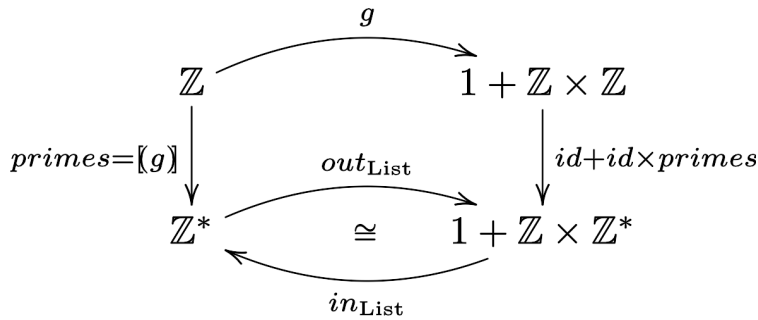
$\llbracket g \rrbracket = g \cdot recList \llbracket g \rrbracket \cdot outSortList$

Problema 2

Para a descoberta dos fatores primos de um número, não negativo, foi implementado um anamorfismo de listas, *primes*, que, para um dado número, devolve a sua lista única de fatores primos.

Este anamorfismo é definido por um gene que, para um número n , verifica se este é menor que 2, caso em que devolve um valor à esquerda do tipo $1 + \text{Int} \times \text{Int}$ que indica o fim da iteração, ou, caso contrário, devolve um valor à direita do tipo $1 + \text{Int} \times \text{Int}$ cujo a primeira componente do par é o menor fator primo de n e a segunda componente é o quociente da divisão de n pelo seu menor fator primo, para que a iteração continue.

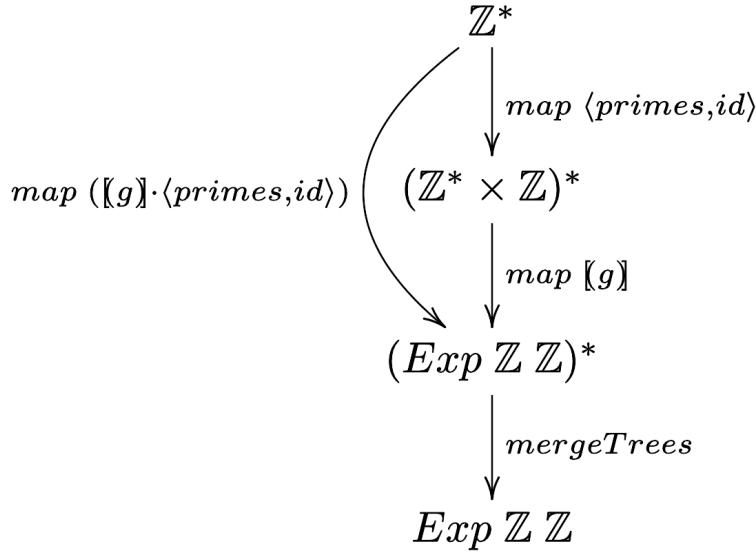
Para obtenção do menor fator primo de um número n , recorre-se a uma lista de inteiros por compreensão, $x \leftarrow [2..n]$, que, para cada x , verifica se $n \text{ 'mod' } x \equiv 0$, ou seja, se x é um fator de n . Caso seja, x é o menor fator primo de n e devolve-se o par $(x, n \text{ 'div' } x)$.



```
primes = [g]
where
  g n
    | n < 2 = i1 ()
    | otherwise = let p = head [x | x ← [2..n], n 'mod' x ≡ 0]
                  in i2 (p, n 'div' p)
```

Para a geração da árvore de primos de um conjunto de inteiros, optou-se por uma estratégia composta por dois passos principais:

1. Geração de uma de árvore de expressões após a obtenção da lista de fatores primos para cada número do conjunto de entrada, recorrendo ao anamorfismo de $\text{Exp}[[g]]$ e a função $\langle \text{primes}, \text{id} \rangle$, respetivamente.
2. Construção da árvore de primos, com recurso à função mergeTrees , que recebe uma lista de árvores de expressões e as funde numa única árvore.



```

prime_tree = mergeTrees · map (anaExp g · <primes, id>)
  where
    g ([], x) = i1 x
    g (x : xs, y) = i2 (x, [(xs, y)])
mergeTrees [tree] = tree
mergeTrees trees = Term 1 (mergeSubtrees trees)
mergeSubtrees [] = []
mergeSubtrees [tree] = [tree]
mergeSubtrees (Term o xs : Term o' ys : trees)
  | o ≡ o' = mergeSubtrees (Term o (xs ++ ys) : trees)
  | otherwise = Term o xs : mergeSubtrees (Term o' ys : trees)
mergeSubtrees (tree : trees) = tree : mergeSubtrees trees

```

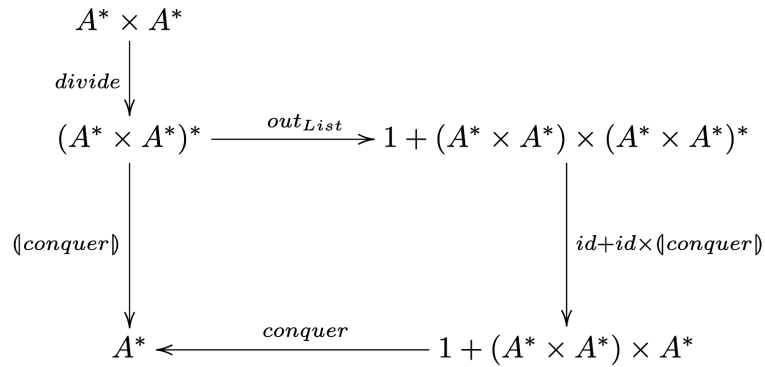
Problema 3

A convolução é uma operação muito utilizada em áreas como processamento de sinais, redes neurais e outras relacionadas à computação. Assim, consistindo em, partindo de duas listas a e b , combinar a informação de ambas as listas, através de multiplicações e somas, resultando na criação de uma nova lista.

De maneira a alcançar uma função que tivesse este comportamento, definimos a *convolve*.

Após verificar qual de ambas listas recebidas é a menor, esta irá criar uma lista de $N + M - 1$ tuplos (onde N e M representam o tamanho da maior e menor lista recebida). Os tuplos vão ser constituídos pela maior lista recebida original à esquerda, e por uma versão alterada da menor lista recebida à direita (sendo cada versão criada única ao tuplo a que pertence).

Finalmente, com auxílio de um catamorfismo de listas, a lista será iterada, aplicando a cada tuplo uma função que multiplique todos os membros de ambas listas com um mesmo índice, e, mais tarde, faça o somatório da lista resultante, adicionando o resultado obtido a uma lista final.



$\text{convolve} :: \text{Num } a \Rightarrow [a] \rightarrow [a] \rightarrow [a]$

$\text{convolve} = \langle \text{conquer} \rangle . : \overline{\text{divide}}$

where

$(. :) :: (c \rightarrow d) \rightarrow (a \rightarrow b \rightarrow c) \rightarrow a \rightarrow b \rightarrow d$

$(. :) = (\cdot) \cdot (\cdot)$

$\text{divide} = \widehat{\text{zipper}} \cdot \text{cond } ((\widehat{>}) \cdot (\text{length} \times \text{length})) \text{ swap id}$

$\text{zipper } n \ m = \text{zip } (\text{replicate } l \ n) \ (\text{map } (h \ m) [1..l])$

where

$l = \text{length } n + \text{length } m - 1$

$h \ xs \ i = (++) \ r \cdot \text{reverse} \cdot \text{take } (i - l) \ \$ \ \text{drop } l \ xs$

where

$l = \max 0 \ (i - \text{length } xs)$

$r = \text{replicate } l \ 0$

$\text{conquer} = [g1, g2]$

$g1 = \text{nil}$

$g2 = \text{cons} \cdot (f \times \text{id})$

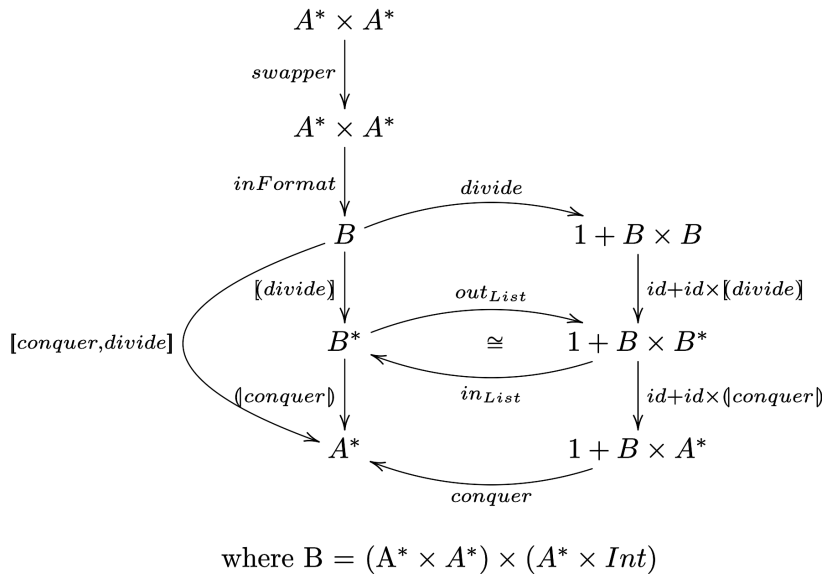
$f = \text{sum} \cdot (\widehat{\text{zipWith}} \ (*))$

Como alternativa, foi desenvolvida uma versão desta função que é completamente *pointfree* e aproveita a inferência de tipos do Haskell.

A função *convolve'*, ao invés de usar somente um catamorfismo de listas, baseia-se num hylomorfismo, este também de listas.

O gene do anamorfismo vai recursivamente desenvolver o tuplo recebido, guardando neste um tuplo com a maior lista recebida e uma versão alterada da menor, e num segundo tuplo, a menor lista (original) e o número da iteração efetuada.

Por outro lado, o gene do catamorfismo será aquele usado na versão anterior de *convolve*, aplicando as anteriormente mencionadas multiplicações seguidas de um somatório da lista resultante a todos os tuplos ali presentes.



$$\begin{array}{c}
 A^* \times A^* \\
 \downarrow \langle \widehat{sizeDiff}, [conquer, divide] \cdot inFormat \cdot swapper \rangle \\
 Int \times A^* \\
 \downarrow \widehat{drop} \\
 A^*
 \end{array}$$

$$convolve' = \widehat{drop} \cdot \langle \widehat{sizeDiff}, [conquer, divide] \cdot inFormat \cdot swapper \rangle$$

where

$$sizeDiff = abs \cdot \widehat{(-)} \cdot (length \times length)$$

$$swapper = cond ((\widehat{>}) \cdot (length \times length)) id swap$$

$$inFormat = \langle id, swap \cdot \widehat{flip} (-) 1 \cdot (2*) \cdot length \cdot \pi_1, \pi_2 \rangle$$

$$conquer = [nil, cons \cdot (applyMaths \times id)]$$

$$\textbf{where } applyMaths = sum \cdot \widehat{zipWith} (*) \cdot swap \cdot \pi_1$$

$$divide = cond ((0 \equiv) \cdot \pi_2 \cdot \pi_2) (i_1 \cdot (!)) (i_2 \cdot dup \cdot \langle \langle \pi_1 \cdot \pi_1, makeList \rangle, (id \times \widehat{flip} (-) 1) \cdot \pi_2 \rangle)$$

where

$$makeList = buildPair \cdot \langle \pi_2, length \cdot \pi_1 \cdot \pi_1 \rangle$$

$$buildPair = \widehat{drop} \cdot \langle \pi_2 \cdot \pi_1, \widehat{++} \cdot \widehat{flip} replicate 0 \cdot \pi_2, reverse \cdot \pi_1 \cdot \pi_1 \rangle$$

Problema 4

Definição do tipo:

Para o cálculo de *outExpr*, partiu-se do isomorfismo de *outExpr* e *inExpr* e da definição de *inExpr*, concluindo-se a seguinte definição de *outExpr*:

$$\begin{aligned}
 & outExpr \cdot inExpr = id \\
 \equiv & \quad \{ \text{Def. inExpr} \} \\
 & outExpr \cdot [V, [N, \hat{T}]] = id \\
 \equiv & \quad \{ 20: \text{Fusão-+ (2x)} \} \\
 & [outExpr \cdot V, [outExpr \cdot N, outExpr \cdot \hat{T}]] = id \\
 \equiv & \quad \{ 17: \text{Universal-+ (2x)}, 1: \text{Natural-id (2x)} \} \\
 & \begin{cases} outExpr \cdot V = i_1 \\ outExpr \cdot N = i_2 \cdot i_1 \\ outExpr \cdot \hat{T} = i_2 \cdot i_2 \end{cases} \\
 \equiv & \quad \{ 72: \text{Igualdade Extencional}, 73: \text{Def-comp}, 86: \text{Uncurry} \} \\
 & \begin{cases} outExpr (V v) = i_1 v \\ outExpr (N n) = (i_2 \cdot i_1) n \\ outExpr (T o exprs) = (i_2 \cdot i_2) (o, exprs) \end{cases} \\
 \square
 \end{aligned}$$

$$\begin{aligned}
 outExpr (V v) &= i_1 v \\
 outExpr (N n) &= (i_2 \cdot i_1) n \\
 outExpr (T o exprs) &= (i_2 \cdot i_2) (o, exprs)
 \end{aligned}$$

Para a definição de *recExpr*, recorreu-se ao functor base *baseExpr* assim como tomou-se inspiração na definição de *recExp*, definida na biblioteca [Exp](#). Deste modo, obteu-se que:

$$recExpr f = baseExpr id id f = id + (id + id \times \text{map } f)$$

$$recExpr f = id + (id + id \times \text{map } f)$$

Ana + cata + hylo:

Através do isomorfismo entre *inExpr* e *outExpr*, as respetivas leis de isomorfismos, ‘Shunt-left’ (33) e ‘Shunt-right’ (34), assim como da lei de indução, Universal-cata (46), da lei de coindução, Universal-ana (55), e da definição de hilomorfismos, obteve-se as seguintes definições:

$$\begin{aligned}
 cataExpr g &= g \cdot recExpr (cataExpr g) \cdot outExpr \\
 anaExpr g &= inExpr \cdot recExpr (anaExpr g) \cdot g \\
 hyloExpr h g &= cataExpr h \cdot anaExpr g
 \end{aligned}$$

Maps:

Para o tipo *Expr b* também foram desenvolvidas as instâncias de *Functor* e *BiFunctor*, com as definições de *fmap* e *bmap* respectivamente, estas derivadas da lei *Def-map-cata* (51).

```
instance Functor (Expr b) where
  fmap g = cataExpr (inExpr · baseExpr g id id)
instance BiFunctor Expr where
  bmap h g = cataExpr (inExpr · baseExpr g h id)
```

Monad:

```
instance Monad (Expr b) where
  return = V
  t >>= g = (muExpr · fmap g) t
  muExpr :: Expr b (Expr b a) → Expr b a
  muExpr = cataExpr [id, [N,  $\hat{T}$ ]]
instance Applicative (Expr b) where
  pure = return
  (< * >) = aap
```

Let expressions:

Foram desenvolvidas duas versões da função *let_exp*, uma com recurso ao catamorfismo *cataExpr* com um gene similar a *inExpr* com exceção do uso da função *f* para as variáveis de *Expr*.

```
let_exp f = cataExpr [f, [N,  $\hat{T}$ ]]
let_exp' = muExpr . : fmap -- equivalent to: let_exp f = muExpr . fmap f
where (. :) = (·) · (·)
```

$$\begin{array}{ccc}
Expr\ C\ A & \xrightarrow{out_{Expr}} & A + (C + Op \times Expr\ C\ A) \\
\downarrow let_exp\ f = \llbracket [f, [N, \hat{T}]] \rrbracket & & \downarrow id + (id + id \times map\ (let_exp\ f)) \\
Expr\ C\ B & \xleftarrow{[f, [N, \hat{T}]]} & A + (C + Op \times Expr\ C\ B)
\end{array}$$

where $f :: A \rightarrow Expr\ C\ B$

A outra versão é equivalente à principal, como demonstrado seguidamente, sendo esta uma versão *pointfree*, que recorre às funções *muExpr* e *fmap*.

$$\begin{aligned}
& \text{let_exp } f = \text{muExpr} \cdot \text{fmap } f \\
\equiv & \quad \{ \text{Def. muExpr, fmap } f = T \, f \} \\
& \text{let_exp } f = \llbracket [id, [N, \widehat{T}]] \rrbracket \cdot T \, f \\
\equiv & \quad \{ 52: \text{Absorção-cata} \} \\
& \text{let_exp } f = \llbracket [id, [N, \widehat{T}]] \cdot B \, (f, id) \rrbracket \\
\equiv & \quad \{ B \, (f, id) = \text{baseExpr } f \, id \, id = f + (id + id \times \text{map } id) \} \\
& \text{let_exp } f = \llbracket [id, [N, \widehat{T}]] \cdot (f + (id + id \times \text{map } id)) \rrbracket \\
\equiv & \quad \{ \text{map } id = id, 22: \text{Absorção-+}, 1: \text{Natural-id}, 15: \text{Functor-id-}\times \} \\
& \text{let_exp } f = \llbracket [f, [N, \widehat{T}]] \rrbracket \\
& \square
\end{aligned}$$

Catamorfismo monádico:

O catamorfismo monádico é uma extensão do conceito clássico de catamorfismo, integrando efeitos monádicos no processo de redução de uma estrutura. No contexto do tipo *Expr*, ele permite combinar a manipulação de subestruturas com operações que podem falhar ou envolver efeitos computacionais.

A definição do *mcataExpr* recorre a função *traverseExpr* para percorrer a estrutura exterior de forma monádica e recursiva, após a travessia, aplica a função *phi* ao resultado, que encapsula a lógica final de transformação.

$$\begin{aligned}
\text{mcataExpr } \phi \, t &= \mathbf{do} \{ b \leftarrow \text{traverseExpr } (\text{mcataExpr } \phi) \, (\text{outExpr } t); \phi \, b \} \\
\text{traverseExpr } f &= [\text{return} \cdot i_1, [\text{return} \cdot i_2 \cdot i_1, m]] \\
&\mathbf{where} \, m \, (op, es) = \mathbf{do} \{ cs \leftarrow \text{mapM } f \, es; \text{return } \$ i_2 \, \$ i_2 \, (op, \text{return } cs) \}
\end{aligned}$$

Avaliação de expressões:

A função *evaluate* utiliza o catamorfismo monádico para calcular o valor de uma expressão, considerando:

1. **Variáveis** — não avaliadas, retornam *Nothing*.
2. **Aridades** — *Just* com o valor.
3. **Termos compostos** — avaliados de acordo com operadores pré-definidos.

Os operadores suportados incluem:

- *Add* — adição de dois valores,
- *Mul* — multiplicação de dois valores,
- *Suc* — sucessor de um valor, equivalente a $+1$,
- *ITE* — *if-then-else*, controlado por um valor condicional.

Cada operador é aplicado a uma lista de operandos previamente avaliados. Se os operandos não forem válidos (exemplo: número insuficiente de argumentos), a avaliação falha com *Nothing*.

Esta definição segue um estilo declarativo, aproveitando o poder combinatório de *monads* para simplificar a lógica de processamento, abstraindo os detalhes da manipulação estrutural.

```
evaluate = mcataExpr [Nothing, Just, evalTerm]]  
where  
  evalTerm op = case op of  
    (Add, Just [x, y]) → Just (x + y)  
    (Mul, Just [x, y]) → Just (x * y)  
    (Suc, Just [x])     → Just (x + 1)  
    (ITE, Just [x, y, z]) → if x ≠ 0 then Just y else Just z  
    _ → Nothing
```