Universidade do Minho
Escola de Engenharia

Cálculo de Programas

Trabalho Prático (2024/25)

Lic. em Engenharia Informática

Grupo G2

a104356	João d'Araújo Dias Lobo
a90817	Mariana Rocha Cristino
a104439	Rita da Cunha Camacho

janeiro 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.

Problema 1

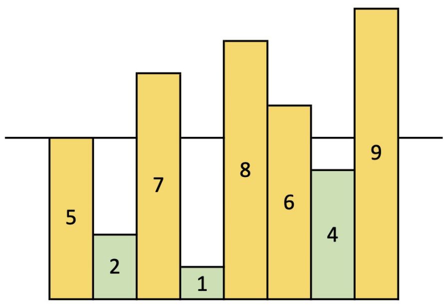
Esta questão aborda um problema que é conhecido pela designação '*H-index of a Histogram*' e que se formula facilmente:

> *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

$$hindex :: [Int] \to (Int, [Int])$$

tal que, para $(i, x) = hindex\ h$, 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. For exemplo,

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

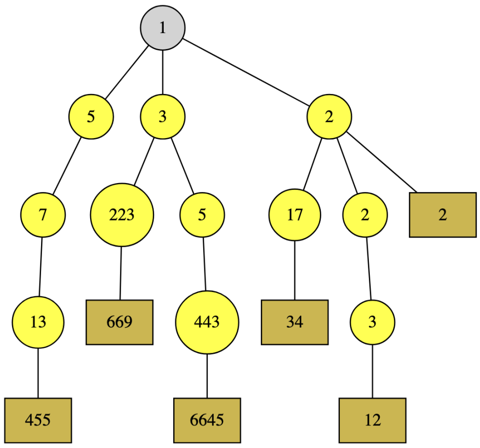
1. Implemente como anamorfismo de listas a função

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

   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:

   $$prime\_tree :: [\mathbb{Z}] \to Exp\ \mathbb{Z}\ \mathbb{Z}$$

   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,

$$\begin{array}{l} \mathit{convolve} :: \mathit{Num}\ a \Rightarrow [a] \to [a] \to [a] \\ \mathit{convolve}\ \mathit{xs}\ \mathit{ys} = [\mathit{sum}\ \$\ \mathit{zipWith}\ (*)\ (\mathit{take}\ n\ (\mathit{drop}\ i\ \mathit{xs}))\ \mathit{ys} \mid i \leftarrow [0\,..\,(\mathit{length}\ \mathit{xs} - n)]] \\ \quad \textbf{where}\ n = \mathit{length}\ \mathit{ys} \end{array}$$

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

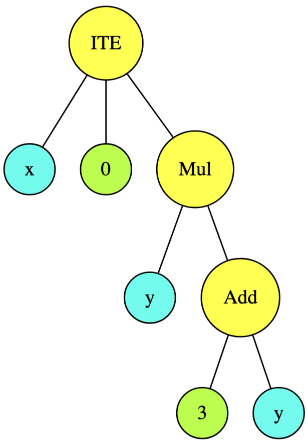
Proponha, explicando-a devidamente, uma solução sua para *convolve*. Valorizar-se-á a economia de código e o recurso aos combinadores *pointfree* estudados na disciplina, em particular a triologia *ana-cata-hilo* de tipos disponíveis nas bibliotecas dadas ou a definir.

Problema 4

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

$$\begin{array}{l} \textbf{data}\ \mathit{Expr}\ b\ a = V\ a \mid N\ b \mid T\ \mathit{Op}\ [\mathit{Expr}\ b\ a]\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \\ \textbf{data}\ \mathit{Op} = \mathit{ITE} \mid \mathit{Add} \mid \mathit{Mul} \mid \mathit{Suc}\ \textbf{deriving}\ (\mathit{Show}, \mathit{Eq}) \end{array}$$

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



representa a expressão

$$\mathit{ite}\ (V\ \texttt{"x"})\ (N\ 0)\ (\mathit{multi}\ (V\ \texttt{"y"})\ (\mathit{soma}\ (N\ 3)\ (V\ \texttt{"y"}))) \tag{1}$$

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

$$\begin{array}{l} \mathit{soma}\ x\ y = T\ \mathit{Add}\ [x, y] \\ \mathit{multi}\ x\ y = T\ \mathit{Mul}\ [x, y] \\ \mathit{ite}\ x\ y\ z = T\ \mathit{ITE}\ [x, y, z] \end{array}$$

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 $\mathit{Expr}\ b$ como instância da classe Monad. **Sugestão**: relembre os exercícios da ficha 12.

3. Defina como um catamorfismo de *Expr* a sua versão monádia, 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

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

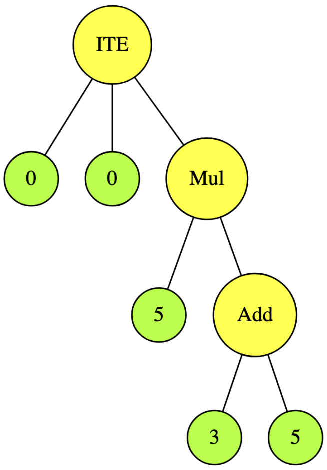
que, dada uma expressão com variáveis em a e uma função que a cada uma dessas variáveis atribui uma expressão ($a \rightarrow Expr\ c\ b$), faz a correspondente substituição.[1] Por exemplo, dada

$$f\ \texttt{"x"} = N\ 0$$
$$f\ \texttt{"y"} = N\ 5$$
$$f\ \_ = N\ 99$$

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,

$$evaluate\ e = Nothing$$
$$evaluate\ (let\_exp\ f\ e) = Just\ 40$$

para f e e dadas acima.

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

$$evaluate\ (T\ Add\ [N\ 2, N\ 3]) = Just\ 5$$
$$evaluate\ (T\ Mul\ [N\ 2]) = Nothing$$

[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.

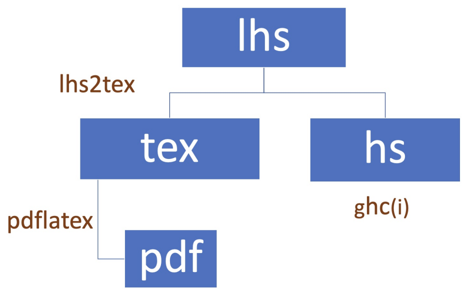
Para cumprir de forma integrada os objectivos do trabalho vamos recorrer a uma técnica de programação dita "literária" [1], cujo princípio base é o seguinte:

> *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 `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 LaTeX. (Ver também a `Makefile` que é disponibilizada.)

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

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

```
$ docker build -t 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 LaTeX. 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-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 `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 BibTeX) 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 ler[1] onde se obtém o efeito seguinte:[2]

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

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

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

E Código fornecido

Problema 1

$$h :: [Int]$$

Problema 4

Definição do tipo:

$$inExpr = [V, [N, \widehat{T}]]$$

$$baseExpr\ g\ h\ f = g + (h + id \times \mathsf{map}\ \ f)$$

Exemplos de expressões:

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

Exemplo de teste:

$$\begin{array}{l} teste = evaluate\ (let\_exp\ f\ i) \equiv Just\ (26\ /\ 245) \\ \quad \mathbf{where}\ f\ \texttt{"x"} = N\ 0; f\ \texttt{"y"} = N\ (1\ /\ 7) \end{array}$$

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

[2] Exemplos tirados de [?].

F 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

A função *hindex* foi implementada como um hilomorfismo de *BTree* (*hyloBTree*), visto que o problema assemelha-se ao processo de ordenação *qsort*, que também utiliza um hilomorfismo. A ideia principal foi usar a partição de elementos como o *qSort* usa e adaptar o restante processo para calcular o h-index.

A função *hindex* é representada pelo seguinte diagrama:

$$\begin{tikzcd}
\mathbb{Z}^* \arrow[r, "qsep"] \arrow[d, "{[\![qsep]\!]}"'] & 1 + (\mathbb{Z}, (\mathbb{Z}^*, \mathbb{Z}^*)) \arrow[d, "id + id \times ([\![qsep]\!] \times [\![qsep]\!])"] \\
BTree\ \mathbb{Z} \arrow[r, shift right, "out"'] \arrow[d, "(\!| f |\!)"'] & 1 + (\mathbb{Z}, (BTree\ \mathbb{Z}, BTree\ \mathbb{Z})) \arrow[l, shift right, "\mathbf{in}"'] \arrow[d, "id + id \times ((\!| f |\!) \times (\!| f |\!))"] \\
(\mathbb{Z}, \mathbb{Z}^*) & 1 + (\mathbb{Z}, ((\mathbb{Z}, \mathbb{Z}^*), (\mathbb{Z}, \mathbb{Z}^*))) \arrow[l, "{f = [\underline{0, [\,]}, hI]}"]
\end{tikzcd}$$

Esta função é composta por um anamorfismo (*anaBTree qsep*) e por um catamorfismo (*cataBTree* $[\underline{0, [\,]}, hI]$).

1. Anamorfismo

A função *qsep* é responsável por dividir a lista de alturas do histograma e construir recursivamente a árvore binária. Assim, caso a lista esteja vazia, retorna $i_1\ ()$.

Caso contrário, o primeiro elemento da lista é escolhido como pivô e os elementos restantes são divididos em dois subconjuntos: s contém os elementos menores que o pivô e l contém os elementos maiores ou iguais ao pivô. Esta divisão é realizada pela função *part* que percorre a lista e verifica, para cada elemento, se este satisfaz o predicado p, no caso da função *qsep*, se é menor que o pivô.

Então, o resultado da função *qsep* é uma árvore binária onde cada nodo contém um pivô e as suas subárvores representam os valores menores e maiores, respetivamente.

2. Catamorfismo

O catamorfismo *cataBTree* $[\underline{0, [\,]}, hI]$ verifica se o nodo é vazio e retorna $(0, [\,])$. Caso contrário, aplica a função *hI* que calcula o h-index e os contribuidores para o nodo atual.

A função *hI* segue os seguintes passos:

2.1. Combinação dos valores das subárvores: junta os valores das subárvores esquerda e direita numa lista, adicionando o valor do nodo atual.

2.2. Cálculo do h-index: cada elemento da lista é emparelhado com a sua posição usando $zip\ [1\,..]\ list$, a função *myfoldr* percorre esses pares para calcular o maior índice k tal que o valor associado seja maior ou igual a k. Ou seja, a lista *list* é transformada em pares $(k, height)$, onde k representa a posição e *height* é o alor da altura correspondente. A função *process* verifica:

- Se $height \geqslant k$, então o h-index é atualizado para o máximo entre o valor atual e k.
- Caso contrário, o h-index mantém-se inalterado.

A função *process*:

- verifica se a altura é maior ou igual ao índice: $\widehat{(\geqslant)} \cdot swap \cdot \pi_1$;
- se a condição for satisfeita, atualiza o h-index: $\widehat{max} \cdot swap \cdot (\pi_1 \times id)$;
- caso contrário, mantém o valor atual: π_2.

2.3. Identificação dos contribuidores: a lista é filtrada para conter apenas os valores maiores ou iguais ao h-index (*filter* $(\geqslant hIndex)$ *list*).

Segue a implementação da função *hindex*:

$$hindex = hyloBTree\ [\underline{0, [\,]}, hI]\ qsep$$

$$\begin{array}{l}
hI :: (Int, ((Int, [Int]), (Int, [Int]))) \rightarrow (Int, [Int]) \\
hI\ (n, ((_, ll), (_, lr))) = (hIndex, contributors) \\
\quad \textbf{where} \\
\qquad list = lr + [n] + ll \\
\qquad hIndex = myfoldr\ \overline{process}\ 0\ (zip\ [1\,..]\ list) \\
\qquad process :: (Ord\ a) \Rightarrow ((a, a), a) \rightarrow a \\
\qquad process = cond\ (\widehat{(\geqslant)} \cdot swap \cdot \pi_1)\ (\widehat{max} \cdot swap \cdot (\pi_1 \times id))\ \pi_2 \\
\qquad contributors = filter\ (\geqslant hIndex)\ list
\end{array}$$

Problema 2

Primeira parte:

A função *primes* é responsável por criar a lista de fatores primos de um dado número. De modo que, esta função pode ser definida como um anamorfismo de listas (*List*). Assim, o diagrama que representa a operação é o seguinte:

$$\begin{tikzcd}
\mathbb{Z} \arrow[r, "g"] \arrow[d, "{[\![g]\!]}"'] & 1 + \mathbb{Z} \times \mathbb{Z} \arrow[d, "id + (id \times [\![g]\!])"] \\
\mathbb{Z}^* \arrow[r, bend right, "out_{List}"'] & 1 + \mathbb{Z} \times \mathbb{Z}^* \arrow[l, bend right, "\mathbf{in}_{List}"']
\end{tikzcd}$$

A implementação baseia-se em decompor o número repetidamente no seu menor fator primo, este processo repete-se até que o quociente resultante seja 1.

O processo pode ser representado graficamente como se segue para o número 455:

$$
\begin{tikzcd}
455 \arrow[d] \\
(5, 91) \arrow[d] \\
(7, 13) \arrow[d] \\
(13, 1) \arrow[d] \\
[\,]
\end{tikzcd}
$$

Assim, $primes\ 455 = [5, 7, 13]$.

A definição de *primes* como $[\!(g)\!]$ tira partido de que um anamorfismo constrói uma estrutura recursiva ao aplicar sucessivamente o gene g a um valor inicial. O gene g determina como cada passo da construção ocorre, neste caso g divide o número n no seu menor fator primo (calculado pela função *smallestPrimeFactor*) e no quociente resultante após a divisão. O processo termina quando $n = 1$, porque não existem mais fatores primos para serem determinados.

A função *smallestPrimeFactor* é responsável por determinar o menor fator primo de um número n, e é definida como um catamorfismo de naturais (*catNat*). Esta função aplica sucessivamente a lógica de "testar se um divisor d divide n" para valores d crescentes, assim inicia com o menor número primo (2).

O ciclo-for contém uma estrutura recursiva que verifica duas condições:

1. **Teste de primalidade:** Se $d^2 > n$: Nesse caso, n é primo e o seu menor fator primo é ele mesmo (o processo termina).

2. **Encontrar o menor fator primo:** Se $n \mod d = 0$: Nesse caso, d é o menor fator primo de n.

Caso contrário: Incrementámos d e continuámos o processo.

Fundamentação matemática: A implementação baseia-se no Teorema Fundamental da Aritmética, que garante que todo o número inteiro positivo maior que 1 pode ser decomposto de forma única como um produto de fatores primos. O processo descrito no gene g utiliza esta propriedade para decompor iterativamente o n nos seus fatores primos, onde a divisibilidade é verificada e avançamos na procura do menor fator primo.

$$
\begin{array}{l}
smallestPrimeFactor\ x = \mathsf{for}\ \lambda n \rightarrow cond\ (\widehat{(>)} \cdot ((\uparrow 2) \times id))\ \pi_2 \\
\quad (cond\ ((\equiv 0) \cdot \widehat{mod} \cdot swap)\ \pi_1\ (\mathsf{succ}\ \cdot \pi_1))\ (n, x)\ 2\ x \\
g\ 1 = i_1\ () \\
g\ n = i_2\ (smallestPrimeFactor\ n, n \div smallestPrimeFactor\ n) \\
primes = [\!(g)\!]
\end{array}
$$

Segunda parte:

A função *prime_tree* é responsável por criar a árvore dos primos de uma lista de inteiros, como se encontra ilustrado no enunciado. De modo que, esta função pode ser definida da seguinte forma:

$$prime\_tree = Term\ 1 \cdot untar \cdot \mathsf{map}\ (\lambda n \rightarrow (primes\ n, n))$$

Inicialmente, adotámos uma abordagem extensiva para resolver o problema, com a definição de um hilomorfismo e todas as operações necessárias para construir a árvore. No entanto, durante este processo, reparámos na função *untar* da biblioteca *Exp.hs*, que efetua a operação necessária para transformar uma lista de pares numa estrutura do tipo $[Exp\ v\ o]$. Após compreendermos o comportamento e a definição da função *untar*, percebemos que era possível utilizá-la na construção da função *prime_tree*, o que simplificou a implementação.

Explicação da função *prime_tree*:

1. A função *primes* é aplicada a cada elemento da lista de inteiros e com o uso da expressão $\mathsf{map}\ (\lambda n \rightarrow (primes\ n, n))$, obtemos uma lista de pares, onde o primeiro elemento é a lista de fatores primos de um número e o segundo elemento é o próprio número. Assim, no final da execução desta expressão, obtemos uma lista de pares do tipo $[([\mathbb{Z}], \mathbb{Z})]$.

2. Neste contexto, a função *untar* converte os fatores primos de um número e o próprio número numa representação de árvore onde os nodos intermediários são os fatores e as folhas são os números originais, $[Exp\ \mathbb{Z}\ \mathbb{Z}]$. Esta conversão é realizada em três partes principais: a coalgebra, a base e a álgebra.

2.1. A coalgebra, representada pela função c, é responsável por decompor os dados, ou seja, separa os pares da lista inicial - $[([\mathbb{Z}], \mathbb{Z})]$ - e transforma cada elemento para o formato $\mathbb{Z} + (\mathbb{Z}, [([\mathbb{Z}], \mathbb{Z})])$.

2.1.1. O $\mathsf{map}\ ((\pi_2 + assocr) \cdot distl \cdot (outList \times id))$ é aplicado a cada par da lista, onde:

- $outList \times id$ transforma a lista de fatores primos num tipo $\cdot + \cdot$ e retorna o número original. Permitindo tratar em separado os fatores primos e os números.
- $distl$ distribui os elementos $(a, b) + \cdot$ para o tipo $(a + b, b)$, separa os dados para facilitar o processamento posterior.
- $\pi_2 + assocr$ reorganiza os pares para agrupar corretamente os fatores primos associados a um número.

2.1.2. *sep* percorre a lista de elementos $\cdot + \cdot$, e separa os elementos i_1 para o primeiro grupo e os i_2 para o segundo grupo.

2.1.3. $id \times collect$ aplica a função *id* ao primeiro valor do tuplo e *collect* ao segundo, de modo que a função *collect* é responsável por agrupar os fatores primos em listas separadas para cada número. Então, os números que partilham o mesmo fator primo são agrupados juntos.

2.1.4. *join* junta os valores numa lista única, recriando a estrutura necessária para representare os dados.

2.2. Após a coalgebra, avançamos para a base, esta aplica recursivamente a função *untar* a cada sublista e cria subárvores para cada conjunto de fatores. O tipo da função *base* é definido como:

$base :: (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow (\mathbb{Z} \rightarrow \mathbb{Z}) \rightarrow ([([\mathbb{Z}], \mathbb{Z})] \rightarrow [Exp\ \mathbb{Z}\ \mathbb{Z}]) \rightarrow [\mathbb{Z} + (\mathbb{Z}, [([\mathbb{Z}], \mathbb{Z})])] \rightarrow [\mathbb{Z} + (\mathbb{Z}, [Exp\ \mathbb{Z}\ \mathbb{Z}])]$.

2.3. Para finalizar, a álgebra a organiza os dados processados na estrutura final, a operação *sort* organiza os nodos, enquanto que o $\mathsf{map}\ inExp$ converte os elementos numa estrutura do tipo $Exp\ \mathbb{Z}\ \mathbb{Z}$. O seu tipo, neste contexto, é definido como: $a :: [\mathbb{Z} + (\mathbb{Z}, [Exp\ \mathbb{Z}\ \mathbb{Z}])] \rightarrow [Exp\ \mathbb{Z}\ \mathbb{Z}]$.

3. Por fim, a função $Term\ 1$ é aplicada para adicionar a raíz da árvore com o valor 1, isto conecta todas as subárvores criadas pela função *untar*, construindo uma única árvore que representa a decomposição de todos os números da lista.

Problema 3

A função *convolve* foi implementada como um anamorfismo de listas (*List*), já que a função é responsável por criar a lista resultante da convolução de 2 listas fornecidas, $l1$ e l_2. Esta vai construindo recursivamente a lista resultado recorrendo à função $[\!(\cdot)\!]$, aplicando sucessivamente às duas listas o gene g que, neste caso, está realmente construído na função *anaGene*.

A função *anaGene* começa por verificar se o índice i (inicialmente 0) ainda se encontra dentro dos limites da convolução, isto é, se $i \geqslant m + n - 1$. Se isto se verificar, será retornada uma lista vazia, visto que a convolução terminou, não sendo adicionado nenhum valor novo à lista final. Caso contrário, será calculado o elemento seguinte da lista resultado, realizando efetivamente o próximo passo da convolução: $sum\ \$\ zipWith\ (*)\ l1\ (\mathsf{map}\ \ (\lambda j \rightarrow access\ (l_2, (i, j)))\ [0 \mathinner{..} (m-1)])$

Além disso, neste cálculo, é verificado se o índice $i - j$ de l_2 é válido, através da função *access*, onde duas condições são confirmadas: $i - j$ tem de ser não negativo $(\widehat{(>)} \cdot (\underline{0} \times \widehat{(-)}))$ e menor que o comprimento de l_2 $(\widehat{(\leqslant)} \cdot (length \times \widehat{(-)}))$.

A função *convolve* é representada pelo seguinte diagrama:

$$
\begin{tikzcd}
\mathbb{Z}^* \times \mathbb{Z}^* \arrow[r, "g"] \arrow[d, "[\!(g)\!]"'] & 1 + \mathbb{Z} \times (\mathbb{Z}^* \times \mathbb{Z}^*) \arrow[d, "id + (id \times [\!(g)\!])"] \\
\mathbb{Z}^* \arrow[r, bend right, "out_{List}"'] & 1 + \mathbb{Z} \times \mathbb{Z}^* \arrow[l, bend right, "\mathbf{in}_{List}"']
\end{tikzcd}
$$

A implementação da função *convolve* realizada foi a seguinte:

$$
\begin{array}{l}
convolve :: Num\ a \Rightarrow [a] \rightarrow [a] \rightarrow [a] \\
convolve\ l1\ l_2 = [\!(\ anaGene\ l1\ l_2\ )\!]\ 0 \\
anaGene :: Num\ a \Rightarrow [a] \rightarrow [a] \rightarrow Int \rightarrow () + (a, Int) \\
anaGene\ l1\ l_2 = cond\ (\geqslant m + n - 1)\ (\underline{\ }\ \$\ i_1\ ()) \\
\quad (\lambda i \rightarrow i_2\ (sum\ \$\ zipWith\ (*)\ l1\ (\mathsf{map}\ \ (\lambda j \rightarrow access\ (l_2, (i, j)))\ [0 \mathinner{..} (m-1)]), i + 1)) \\
\quad \mathbf{where} \\
\qquad m = length\ l1; n = length\ l_2 \\
\qquad access = cond\ ((\vee)\ \langle \$ \rangle\ cond1 < * > cond2)\ \underline{0}\ (\widehat{(!!)} \cdot (id \times \widehat{(-)})) \\
\qquad cond1 = \widehat{(>)} \cdot (\underline{0} \times \widehat{(-)}) \\
\qquad cond2 = \widehat{(\leqslant)} \cdot (length \times \widehat{(-)})
\end{array}
$$

Problema 4

1. Nesta pergunta, pretende-se definir por completo, a biblioteca *Expr*, à semelhança das outras bibliotecas de estruturas fornecidas.

Definição do tipo de *Expr*:

Para o cálulo de *outExpr*, partimos da afirmação $outExpr \cdot inExpr = id$,

$$
\begin{array}{ll}
 & outExpr \cdot inExpr = id \\
\equiv & \quad \{\ \text{def inExpr}\ \}
\end{array}
$$

$$
\begin{array}{ll}
& outExpr \cdot [V, [N, \widehat{T}]] = id \\
\equiv & \quad \{ \text{ fusão + } \} \\
& [outExpr \cdot V, [outExpr \cdot N, outExpr \cdot \widehat{T}]] = id \\
\equiv & \quad \{ \text{ universal +, natural id } \} \\
& \left\{ \begin{array}{l} outExpr \cdot V = i_1 \\ [outExpr \cdot N, outExpr \cdot \widehat{T}] = i_2 \end{array} \right. \\
\equiv & \quad \{ \text{ universal + } \} \\
& \left\{ \begin{array}{l} outExpr \cdot V = i_1 \\ outExpr \cdot N = i_2 \cdot i_1 \\ outExpr \cdot \widehat{T} = i_2 \cdot i_2 \end{array} \right. \\
\equiv & \quad \{ \text{ igualdade extensional, def-comp, uncurry } \} \\
& \left\{ \begin{array}{l} outExpr\ (V\ n) = i_1\ n \\ outExpr\ (N\ n) = (i_2 \cdot i_1)\ n \\ outExpr\ (T\ op\ exprs) = (i_2 \cdot i_2)\ (op, exprs) \end{array} \right.
\end{array}
$$

Ficando assim, em Haskell, com:

$outExpr :: Expr\ b\ a \to a + (b + (Op, [Expr\ b\ a]))$
$outExpr\ (V\ n) = i_1\ n$
$outExpr\ (N\ n) = (i_2 \cdot i_1)\ n$
$outExpr\ (T\ op\ exprs) = (i_2 \cdot i_2)\ (op, exprs)$

Cálculo do functor de *Expr*:

Sabendo que $F\ f = B\ (id, f)$, temos que:

$$
\begin{array}{ll}
& F\ f = B\ (id, id, f) \\
\equiv & \quad \{ \text{ def B } \} \\
& F\ f = id + (id + id \times \mathsf{map}\ \ f)
\end{array}
$$

Definindo, então, *recExpr* como:

$recExpr :: (a \to b1) \to b2 + (b3 + (b4, [a])) \to b2 + (b3 + (b4, [b1]))$
$recExpr = baseExpr\ id\ id$

Definição da triologia ana-cata-hylo:

Começando pelo catamorfismo de *Expr*, temos:

$$
\begin{array}{ll}
\equiv & \quad \{ \text{ cancelamento-cata } \} \\
& (\!| g |\!) \cdot \mathsf{in} = g \cdot \mathsf{F}\ (\!| g |\!) \\
\equiv & \quad \{ \text{ shunt-left } \} \\
& (\!| g |\!) = g \cdot \mathsf{F}\ (\!| g |\!) \cdot \mathsf{out}
\end{array}
$$

Representado pelo seguinte diagrama:

$$\begin{tikzcd}
Expr\ C\ A \arrow[d, "(\!|g|\!)"'] \arrow[rr, bend right, "out"'] & & A + (C + (Op \times (Expr\ C\ A)^*)) \arrow[ll, bend right, "\mathsf{in}"'] \arrow[d, "id+(id+(id\times \mathsf{map}\ (\!|g|\!)))"] \\
Expr\ C\ B & & A + (C + (Op \times (Expr\ C\ B)^*)) \arrow[ll, "g"]
\end{tikzcd}$$

Utilizando as funções previamente definidas, temos então:

$$cataExpr\ g = g \cdot recExpr\ (cataExpr\ g) \cdot outExpr$$

Para o anamorfismo de *Expr*, temos:

$$\begin{aligned}
\equiv & \quad \{ \text{ cancelamento-ana } \} \\
& \mathsf{out} \cdot [\!(\, g \,)\!] = \mathsf{F}\ [\!(\, g \,)\!] \cdot g \\
\equiv & \quad \{ \text{ shunt-right } \} \\
& [\!(\, g \,)\!] = \mathsf{in} \cdot \mathsf{F}\ [\!(\, g \,)\!] \cdot g
\end{aligned}$$

Representado pelo seguinte diagrama:

$$\begin{tikzcd}
Expr\ C\ A \arrow[rr, bend right, "out"'] & & A + (C + (Op \times (Expr\ C\ A)^*)) \arrow[ll, bend right, "\mathsf{in}"'] \\
Expr\ C\ D \arrow[u, "[\!(\, g \,)\!]"] \arrow[rr, "g"'] & & A + (C + (Op \times (Expr\ C\ D)^*)) \arrow[u, "id+(id+(id\times \mathsf{map}\ [\!(\, g \,)\!]))"']
\end{tikzcd}$$

Utilizando as funções previamente definidas, temos então:

$$anaExpr\ g = inExpr \cdot recExpr\ (anaExpr\ g) \cdot g$$

Sendo o hilomorfismo, a composição do catamorfismo e do anamorfismo, representado pelo seguinte diagrama:

$$\begin{tikzcd}
Expr\ C\ D \arrow[d, "[\!(\, g \,)\!]"'] \arrow[rr, "g"'] & & A + (C + (Op \times (Expr\ C\ D)^*)) \arrow[d, "id+(id+(id\times \mathsf{map}\ [\!(\, g \,)\!]))"] \\
Expr\ C\ A \arrow[d, "(\!|h|\!)"'] \arrow[rr, bend right, "out"'] & & A + (C + (Op \times (Expr\ C\ A)^*)) \arrow[ll, bend right, "\mathsf{in}"'] \arrow[d, "id+(id+(id\times \mathsf{map}\ (\!|h|\!)))"] \\
Expr\ C\ B & & A + (C + (Op \times (Expr\ C\ B)^*)) \arrow[ll, "h"]
\end{tikzcd}$$

ou seja,

$$[\![h, g]\!] = (\!| h |\!) \cdot [\!(g)\!]$$

Este é definido em Haskell usando as funções *cataExpr* e *anaExpr* previamente definidas:

$$hyloExpr\ h\ g = cataExpr\ h \cdot anaExpr\ g$$

2. *Monad*:

Para declarar *Expr b* como instância da classe *Monad*, foram implementadas as intâncias de *Functor*, *Applicative* e *Monad* do tipo *Expr b*. A abordagem utilizada foi guiada pelo exercício 4 da ficha 12, adaptando ao contexto específico de *Expr b*.

Começamos pelo *Functor*, onde a função fmap aplica uma transformação às variáveis da expressão, mantendo as restantes estruturas (N e T) inalteradas. Esta operação é realizada de forma recursiva usando o catamorfismo - *cataExpr* -, que reconstrói a expressão após aplicar a f às variáveis. A composição com a função *inExpr* e a base do catamorfismo (*baseExpr*) assegura que a estrutura é processada corretamente.

$$\begin{array}{l} \textbf{instance}\ Functor\ (Expr\ b) \\ \quad \textbf{where}\ \mathsf{fmap}\ f = cataExpr\ (inExpr \cdot baseExpr\ f\ id\ id) \end{array}$$

De seguida, definimos a instância *Applicative*, onde a função *pure* cria uma expressão com uma variável (V) com o valor fornecido, a função $< * >$ considera três casos:

- se a expressão é uma variável ($V\ f$), aplica fmap para mapear função sobre a outra expressão;
- se a expressão é um número ($N\ b$), mantém o número inalterado;
- se a expressão é uma operação ($T\ op\ fs$), aplica a função a cada subexpressão da operação.

$$\begin{array}{l} \textbf{instance}\ Applicative\ (Expr\ b)\ \textbf{where} \\ \quad pure :: a \rightarrow Expr\ b\ a \\ \quad pure = V \\ \quad (V\ f) < * > x = \mathsf{fmap}\ f\ x \\ \quad (N\ b) < * > \_ = N\ b \\ \quad (T\ op\ fs) < * > x = T\ op\ (\mathsf{map}\ \ (< * > x)\ fs) \end{array}$$

Por fim, a instância *Monad* foi definida, a definição *return* é equivalente a *pure*, cria uma variável. A operação $\gg=$ aplica a função g a cada variável da expressão, usando a função auxiliar *muExpr* para processar a expressão resultante.

$$\begin{array}{l} \textbf{instance}\ Monad\ (Expr\ b)\ \textbf{where} \\ \quad return :: a \rightarrow Expr\ b\ a \\ \quad return = pure \\ \quad (\gg=) :: Expr\ b\ a \rightarrow (a \rightarrow Expr\ b\ b1) \rightarrow Expr\ b\ b1 \\ \quad t \gg= g = muExpr\ (\mathsf{fmap}\ g\ t) \\ muExpr :: Expr\ b\ (Expr\ b\ a) \rightarrow Expr\ b\ a \\ muExpr = cataExpr\ [id, inExpr \cdot i_2] \\ u :: a \rightarrow Expr\ b\ a \\ u = V \end{array}$$

Provemos que *Expr b* é uma instância de *Monad*:

- $u = V$ e $V = inExpr \cdot i_1$, logo $u = inExpr \cdot i_1$;
- $muExpr = cataExpr\ [id, inExpr \cdot i_2]$.

Provar a lei monádica Unidade (63):

$$
\begin{array}{ll}
& mu \cdot u = mu \cdot T\ u \\
\equiv & \quad \{ \text{ definição de mu; definição de u } \} \\
& (\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!) \cdot \mathbf{in} \cdot i_1 = (\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!) \cdot T\ u \\
\equiv & \quad \{ \text{ absorção-cata } \} \\
& (\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!) \cdot \mathbf{in} \cdot i_1 = (\!|\,[id,\ \mathbf{in} \cdot i_2] \cdot B\ (u, id)\,|\!) \\
\equiv & \quad \{ \text{ B(f,g) = f + G g, absorção-+, natural-id, functor-id-F } \} \\
& (\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!) \cdot \mathbf{in} \cdot i_1 = (\!|\,[u,\ \mathbf{in} \cdot i_2]\,|\!) \\
\equiv & \quad \{ \text{ definição de u, cancelamento-cata } \} \\
& [id,\ \mathbf{in} \cdot i_2] \cdot F\ mu \cdot i_1 = (\!|\,[\ \mathbf{in} \cdot i_1,\ \mathbf{in} \cdot i_2]\,|\!) \\
\equiv & \quad \{ \text{ fusão-+, reflexão-+, reflexão-cata, base-cata, B(id, mu) = id + G mu } \} \\
& [id,\ \mathbf{in} \cdot i_2] \cdot (id + G\ mu) \cdot i_1 = id \\
\equiv & \quad \{ \text{ natural-i1, natural-id } \} \\
& [id,\ \mathbf{in} \cdot i_2] \cdot i_1 = id \\
\equiv & \quad \{ \text{ cancelamento-+ } \} \\
& id = id \\
\equiv & \quad \{ \text{ P.R.I. } \} \\
& True \\
\Box &
\end{array}
$$

Provar a lei monádica Multiplicação (62):

$$
\begin{array}{ll}
& mu \cdot mu = mu \cdot T\ mu \\
\equiv & \quad \{ \text{ definição de mu } \} \\
& mu \cdot mu = (\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!) \cdot T\ (\!|\cdot|\!)\ [id,\ \mathbf{in} \cdot i_2] \\
\equiv & \quad \{ \text{ absorção-cata } \} \\
& mu \cdot mu = (\!|\,[id,\ \mathbf{in} \cdot i_2] \cdot ((\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!) + G\ id)\,|\!) \\
\equiv & \quad \{ \text{ Functor-id-F, natural-id, absorção-+ } \} \\
& mu \cdot mu = (\!|\,[(\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!),\ \mathbf{in} \cdot i_2]\,|\!) \\
\equiv & \quad \{ \text{ definição de mu } \} \\
& mu \cdot (\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!) = (\!|\,[(\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!),\ \mathbf{in} \cdot i_2]\,|\!) \\
\Leftarrow & \quad \{ \text{ fusão-cata } \} \\
& mu \cdot [id,\ \mathbf{in} \cdot i_2] = [(\!|\,[id,\ \mathbf{in} \cdot i_2]\,|\!),\ \mathbf{in} \cdot i_2] \cdot (id + G\ mu) \\
\equiv & \quad \{ \text{ fusão-+, absorção-+, natural-id, eq-+ } \} \\
& [id,\ \mathbf{in} \cdot i_2] \cdot i_1 = id \\
\equiv & \quad \{ \text{ cancelamento-+ } \}
\end{array}
$$

$$\left\{\begin{array}{l} mu = mu \\ mu \cdot \mathbf{in} \cdot i_2 = \mathbf{in} \cdot i_2 \cdot G\ mu \end{array}\right.$$

$\equiv \qquad \{$ p & True = True, definição de mu, cancelamento-cata, base-cata $\}$

$[id,\ \mathbf{in} \cdot i_2] \cdot (id + G\ mu) \cdot i_2 = \mathbf{in} \cdot i_2 \cdot G\ mu$

$\equiv \qquad \{$ natural-i2, cancelamento-+ $\}$

$\mathbf{in} \cdot i_2 \cdot G\ mu = \mathbf{in} \cdot i_2 \cdot G\ mu$

$\equiv \qquad \{$ P.R.I $\}$

$True$

$\square$

3. *Catamorifsmo monádico*

$$\begin{tikzcd}
Expr\ C\ A \arrow[d, "\llparenthesis g \rrparenthesis"'] \arrow[rr, bend right, "out"'] & & A + (C + (Op \times (Expr\ C\ A)^*)) \arrow[ll, bend right, "\mathsf{in}"'] \arrow[d, "id+(id+(id\times \mathsf{map}\ \llparenthesis g \rrparenthesis))"] \\
Expr\ C\ B \arrow[d, "u"'] & & A + (C + (Op \times (Expr\ C\ B)^*)) \arrow[ll, "g"] \arrow[d, "id+(id+(id\times \mathsf{map}\ u))"] \\
m\ (Expr\ C\ B) & & A + (C + (Op \times (m\ (Expr\ C\ A))^*)) \arrow[ll, "h"]
\end{tikzcd}$$

$$\begin{tikzcd}
Expr\ C\ A \arrow[d, "\llparenthesis g \rrparenthesis"'] \arrow[rr, bend right, "out"'] & & A + (C + (Op \times (Expr\ C\ A)^*)) \arrow[ll, bend right, "\mathsf{in}"'] \arrow[d, "id+(id+(id\times \mathsf{map}\ \llparenthesis g \rrparenthesis))"] \\
m\ (Expr\ C\ B) & & A + (C + (Op \times (m\ (Expr\ C\ A))^*)) \arrow[ll, "{[h_1,h_2]}"]
\end{tikzcd}$$

Para definir o catamorfismo monádico de *Expr*, a função *mcataExpr*, começamos por entender o papel fundamental da função *dl*, que é responsável por transformar a estrutura de *Expr* num contexto monádico. Esta transformação permite que o processamento das expressões ocorra dentro de um monad.

A função *dl* é definida como:

$$\begin{array}{l}
dl :: Monad\ m \Rightarrow a + (b + (Op, [m\ c])) \rightarrow m\ (a + (b + (Op, m\ [c]))) \\
dl = [return \cdot i_1, [return \cdot i_2 \cdot i_1, aux]] \\
\quad \mathbf{where}\ aux\ (op, ms) = \mathbf{do}\ m \leftarrow lamb\ ms; (return \cdot i_2 \cdot i_2)\ (op, return\ m)
\end{array}$$

- No caso de um valor do tipo a, a função simplesmente o encapsula no monad utilizando $return \cdot i_1$.

- No caso de um valor do tipo b, o mesmo ocorre, utilizando $return \cdot i_2 \cdot i_1$.

- No caso de uma operação (*Op*) acompanhada por uma lista de valores monádicos ($[m\ c]$), é usada a função auxiliar *aux*.

Dentro de *aux*, a função *lamb* é utilizada para distribuir o monad pela lista. Em seguida, o resultado é encapsulado novamente na estrutura esperada pelo tipo $m\ (a + (b + (Op, m\ [c])))$.

A função *mcataExpr*, que define o catamorfismo monádico propriamente dito, é então definida da seguinte forma:

$$\begin{array}{l} mcataExpr :: Monad\ m \Rightarrow (a + (b + (Op, m\ [c])) \rightarrow m\ c) \rightarrow Expr\ b\ a \rightarrow m\ c \\ mcataExpr\ g = g\ .!\ (dl \cdot recExpr\ (mcataExpr\ g) \cdot outExpr) \end{array}$$

Esta definição, assemelha-se à definição do catamorfismo simples, com a diferença de que a função *dl* é usada para distribuir o monad sobre a lista de subexpressões.

4. *Maps*: *Monad*: *Let expressions*:

A função *let_exp* é responsável por substituir todas as variáveis numa expressão *Expr* pelas suas correspondentes expressões atribuídas por uma função fornecida como argumento.

A definição da função *let_exp* utiliza o catamorfismo *cataExpr*. No caso de encontrar uma variável (V), a função f é usada para substituir essa variável pela expressão correspondente. Para valores numéricos (N), a função mantém o valor inalterado. Para operadores (caso T), a função constrói uma nova expressão que combina os resultados das subexpressões recursivamente.

Em suma, *let_exp* avalia uma expressão ao substituir todas as variáveis pelas expressões correspondentes, garantindo que a estrutura da expressão original é mantida. De seguida, o diagrama mostra como a operação do catamorfismo percorre e transforma a expressão.

$$\begin{tikzcd} Expr\ C\ A \arrow[r, bend right, "out_{Expr}"'] \arrow[d, "let\_exp\ f"'] & A + (C + (Op \times (Expr\ C\ A)^*)) \arrow[l, bend right, "\mathbf{in}_{Expr}"'] \arrow[d, "id + (id + (id \times \mathsf{map}\ \ (let\_exp\ f)))"] \\ Expr\ C\ B & A + (C + (Op \times (Expr\ C\ B)^*)) \arrow[l, "{[f, [N, \widehat{T}]]}"] \end{tikzcd}$$

Segue a implementação da função *let_exp*:

$$let\_exp\ f = cataExpr\ [f, [N, \widehat{T}]]$$

Catamorfismo monádico:

$$\begin{array}{l} mcataExpr\ g = g\ .!\ (dl' \cdot recExpr\ (mcataExpr\ g) \cdot outExpr) \\ dl' :: Monad\ m \Rightarrow a + (b + (Op, [m\ c])) \rightarrow m\ (a + (b + (Op, m\ [c]))) \\ dl' = [return \cdot i_1, [return \cdot i_2 \cdot i_1, aux]] \\ \quad \mathbf{where}\ aux\ (op, ms) = \mathbf{do}\ m \leftarrow lamb\ ms; (return \cdot i_2 \cdot i_2)\ (op, return\ m) \end{array}$$

Avaliação de expressões:

A função *evaluate* avalia expressões do tipo *Expr* e retorna o resultado da avaliação com o tipo *Maybe*. Esta função tem em conta dois cenários de erro: variáveis não instanciadas e operadores aplicados a um número incorreto de argumentos.

A função *evaluate* utiliza o catamorfismo *mcataExpr* para avaliar a expressão. Este conceito generaliza o conceito de catamorfismo simples para permitir trabalhar em contextos monádicos. O ponto central deste conceito é que combina a lógica de transformação (g) com a recursão da estrutura, enquanto lida automaticamente com contextos monádicos. Assim, evitámos tratar manualmente de cada contexto monádico *Maybe* em cada passo.

No caso do *evaluate*, o gene *gene* define como é que se processa cada nodo da estrutura *Expr*. O gene *gene* lida com três casos principais:

- V : Para uma variável, retornámos *Nothing*, porque as variávis não podem ser avaliadas.
- N : Para um número, retornámos o próprio número com *Just*.
- T : Para um operador, utilizámos a função auxiliar *aux* que:
  - avalia todos os argumentos no contexto *Maybe*, isto é, verifica se todos os argumentos são válidos;
  - aplica a função *result* para calcular o resultado final, caso todos os argumentos sejam válidos.

A função *result* define o comportamento esperado para cada operados e valida a aridade, assim garante que apenas os operadores com a aridade correta sejam avaliados. Caso contrário, a avaliação falha e retorna *Nothing*.

Segue a implementação da função *evaluate*:

$$
\begin{array}{l}
evaluate = mcataExpr\ gene\\
gene :: (Num\ a, Ord\ a) \Rightarrow b + (a + (Op, Maybe\ [a])) \rightarrow Maybe\ a\\
gene = [\underline{Nothing}, [Just, aux]]\\
\quad \mathbf{where}\ aux\ (op, args) = \mathbf{do}\ argsR \leftarrow args; result\ op\ argsR\\
result :: (Num\ a, Ord\ a) \Rightarrow Op \rightarrow [a] \rightarrow Maybe\ a\\
result\ Add\ [x, y] = Just\ (x + y)\\
result\ Mul\ [x, y] = Just\ (x * y)\\
result\ Suc\ [x] = Just\ (x + 1)\\
result\ ITE\ [cond, t, f] = \mathbf{if}\ cond \not\equiv 0\ \mathbf{then}\ Just\ t\ \mathbf{else}\ Just\ f\\
result\ \_\ \_ = Nothing
\end{array}
$$

Index

References

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

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