

# Cálculo de Programas

## Trabalho Prático

### MiEI+LCC — 2020/21

*Departamento de Informática*  
Universidade do Minho

Junho de 2021

Grupo nr.	17
a91683	Alef Pinto Keuffer
a93546	Fernando Maria Bicalho
a88062	Pedro Paulo Costa Pereira
a91693	Tiago André Oliveira Leite

## 1 Preâmbulo

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

Na sequência pedagógica dos planos de estudo dos dois 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.

## 2 Documentação

Para cumprir de forma integrada os objectivos enunciados acima 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 `cp2021t.pdf` que está a ler é já um exemplo de **programação literária**: foi gerado a partir do texto fonte `cp2021t.lhs`<sup>1</sup> que encontrará no **material pedagógico** desta disciplina descompactando o ficheiro `cp2021t.zip` e executando:

```
$ lhs2TeX cp2021t.lhs > cp2021t.tex
$ pdflatex cp2021t
```

em que **lhs2tex** é um pre-processor que faz “pretty printing” de código Haskell em **L<sup>A</sup>T<sub>E</sub>X** e que deve desde já instalar executando

```
$ cabal install lhs2tex --lib
```

Por outro lado, o mesmo ficheiro `cp2021t.lhs` é executável e contém o “kit” básico, escrito em **Haskell**, para realizar o trabalho. Basta executar

```
$ ghci cp2021t.lhs
```

---

<sup>1</sup>O suffixo ‘lhs’ quer dizer *literate Haskell*.

Abra o ficheiro `cp2021t.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.

### 3 Como realizar o trabalho

Este trabalho teórico-prático deve ser realizado por grupos de 3 (ou 4) 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 de trabalho por forma a poderem responder às questões que serão colocadas na *defesa oral* do relatório.

Em que consiste, então, o *relatório* a que se refere o parágrafo anterior? É a edição do texto que está a ser lido, preenchendo o anexo **D** 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 cp2021t.aux
$ makeindex cp2021t.idx
```

e recompilar o texto como acima se indicou. Dever-se-á ainda instalar o utilitário **QuickCheck**, que ajuda a validar programas em **Haskell** e a biblioteca **Gloss** para geração de gráficos 2D:

```
$ cabal install QuickCheck gloss --lib
```

Para testar uma propriedade **QuickCheck** *prop*, basta invocá-la com o comando:

```
> quickCheck prop
+++ OK, passed 100 tests.
```

Pode-se ainda controlar o número de casos de teste e sua complexidade, como o seguinte exemplo mostra:

```
> quickCheckWith stdArgs { maxSuccess = 200, maxSize = 10 } prop
+++ OK, passed 200 tests.
```

Qualquer programador tem, na vida real, de ler e analisar (muito!) código escrito por outros. No anexo **C** disponibiliza-se algum código **Haskell** relativo aos problemas que se seguem. Esse anexo deverá ser consultado e analisado à medida que isso for necessário.

#### 3.1 Stack

O **Stack** é um programa útil para criar, gerir e manter projetos em **Haskell**. Um projeto criado com o **Stack** possui uma estrutura de pastas muito específica:

- Os módulos auxiliares encontram-se na pasta *src*.
- O módulos principal encontra-se na pasta *app*.
- A lista de dependências externas encontra-se no ficheiro *package.yaml*.

Pode aceder ao **GHCi** utilizando o comando:

```
stack ghci
```

Garanta que se encontra na pasta mais externa **do projeto**. A primeira vez que correr este comando as dependências externas serão instaladas automaticamente.

Para gerar o PDF, garanta que se encontra na diretoria *app*.

# Problema 1

Os *tipos de dados algébricos* estudados ao longo desta disciplina oferecem uma grande capacidade expressiva ao programador. Graças à sua flexibilidade, torna-se trivial implementar **DSLs** e até mesmo **linguagens de programação**.

Paralelamente, um tópico bastante estudado no âmbito de **Deep Learning** é a derivação automática de expressões matemáticas, por exemplo, de derivadas. Duas técnicas que podem ser utilizadas para o cálculo de derivadas são:

- *Symbolic differentiation*
- *Automatic differentiation*

*Symbolic differentiation* consiste na aplicação sucessiva de transformações (leia-se: funções) que sejam congruentes com as regras de derivação. O resultado final será a expressão da derivada.

O leitor atento poderá notar um problema desta técnica: a expressão inicial pode crescer de forma descontrolada, levando a um cálculo pouco eficiente. *Automatic differentiation* tenta resolver este problema, calculando o **valor** da derivada da expressão em todos os passos. Para tal, é necessário calcular o valor da expressão e o valor da sua derivada.

Vamos de seguida definir uma linguagem de expressões matemáticas simples e implementar as duas técnicas de derivação automática. Para isso, seja dado o seguinte tipo de dados,

```
data ExpAr a = X
  | N a
  | Bin BinOp (ExpAr a) (ExpAr a)
  | Un UnOp (ExpAr a)
  deriving (Eq, Show)
```

onde *BinOp* e *UnOp* representam operações binárias e unárias, respectivamente:

```
data BinOp = Sum
  | Product
  deriving (Eq, Show)
data UnOp = Negate
  | E
  deriving (Eq, Show)
```

O construtor *E* simboliza o exponencial de base *e*.

Assim, cada expressão pode ser uma variável, um número, uma operação binária aplicada às devidas expressões, ou uma operação unária aplicada a uma expressão. Por exemplo,

```
Bin Sum X (N 10)
```

designa  $x + 10$  na notação matemática habitual.

1. A definição das funções *inExpAr* e *baseExpAr* para este tipo é a seguinte:

```
inExpAr = [X, num_ops] where
  num_ops = [N, ops]
  ops = [bin, Un]
  bin (op, (a, b)) = Bin op a b
baseExpAr f g h j k l z = f + (g + (h × (j × k) + l × z))
```

Defina as funções *outExpAr* e *recExpAr*, e teste as propriedades que se seguem.

**Propriedade [QuickCheck] 1** *inExpAr* e *outExpAr* são testemunhas de um isomorfismo, isto é,  $inExpAr \cdot outExpAr = id$  e  $outExpAr \cdot inExpAr = id$ :

```
prop_in_out_idExpAr :: (Eq a) => ExpAr a -> Bool
prop_in_out_idExpAr = inExpAr . outExpAr == id
prop_out_in_idExpAr :: (Eq a) => OutExpAr a -> Bool
prop_out_in_idExpAr = outExpAr . inExpAr == id
```

2. Dada uma expressão aritmética e um escalar para substituir o  $X$ , a função

$$eval\_exp :: Floating a \Rightarrow a \rightarrow (ExpAr a) \rightarrow a$$

calcula o resultado da expressão. Na página 12 esta função está expressa como um catamorfismo. Defina o respectivo gene e, de seguida, teste as propriedades:

**Propriedade [QuickCheck] 2** A função `eval_exp` respeita os elementos neutros das operações.

$$\begin{aligned} prop\_sum\_idr &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_sum\_idr a exp &= eval\_exp a exp \stackrel{?}{=} sum\_idr \textbf{ where} \\ sum\_idr &= eval\_exp a (Bin Sum exp (N 0)) \\ prop\_sum\_idl &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_sum\_idl a exp &= eval\_exp a exp \stackrel{?}{=} sum\_idl \textbf{ where} \\ sum\_idl &= eval\_exp a (Bin Sum (N 0) exp) \\ prop\_product\_idr &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_product\_idr a exp &= eval\_exp a exp \stackrel{?}{=} prod\_idr \textbf{ where} \\ prod\_idr &= eval\_exp a (Bin Product exp (N 1)) \\ prop\_product\_idl &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_product\_idl a exp &= eval\_exp a exp \stackrel{?}{=} prod\_idl \textbf{ where} \\ prod\_idl &= eval\_exp a (Bin Product (N 1) exp) \\ prop\_e\_id &:: (Floating a, Real a) \Rightarrow a \rightarrow Bool \\ prop\_e\_id a &= eval\_exp a (Un E (N 1)) \equiv expd 1 \\ prop\_negate\_id &:: (Floating a, Real a) \Rightarrow a \rightarrow Bool \\ prop\_negate\_id a &= eval\_exp a (Un Negate (N 0)) \equiv 0 \end{aligned}$$

**Propriedade [QuickCheck] 3** Negar duas vezes uma expressão tem o mesmo valor que não fazer nada.

$$\begin{aligned} prop\_double\_negate &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_double\_negate a exp &= eval\_exp a exp \stackrel{?}{=} eval\_exp a (Un Negate (Un Negate exp)) \end{aligned}$$

3. É possível otimizar o cálculo do valor de uma expressão aritmética tirando proveito dos elementos absorventes de cada operação. Implemente os genes da função

$$optimize\_eval :: (Floating a, Eq a) \Rightarrow a \rightarrow (ExpAr a) \rightarrow a$$

que se encontra na página 12 expressa como um hilomorfismo<sup>2</sup> e teste as propriedades:

**Propriedade [QuickCheck] 4** A função `optimize_eval` respeita a semântica da função `eval`.

$$\begin{aligned} prop\_optimize\_respects\_semantics &:: (Floating a, Real a) \Rightarrow a \rightarrow ExpAr a \rightarrow Bool \\ prop\_optimize\_respects\_semantics a exp &= eval\_exp a exp \stackrel{?}{=} optimize\_eval a exp \end{aligned}$$

4. Para calcular a derivada de uma expressão, é necessário aplicar transformações à expressão original que respeitem as regras das derivadas.<sup>3</sup>

- Regra da soma:

$$\frac{d}{dx}(f(x) + g(x)) = \frac{d}{dx}(f(x)) + \frac{d}{dx}(g(x))$$

<sup>2</sup>Qual é a vantagem de implementar a função `optimize_eval` utilizando um hilomorfismo em vez de utilizar um catamorfismo com um gene "inteligente"?

<sup>3</sup>Apesar da adição e multiplicação gozarem da propriedade comutativa, há que ter em atenção a ordem das operações por causa dos testes.

- Regra do produto:

$$\frac{d}{dx}(f(x)g(x)) = f(x) \cdot \frac{d}{dx}(g(x)) + \frac{d}{dx}(f(x)) \cdot g(x)$$

Defina o gene do catamorfismo que ocorre na função

$$sd :: Floating a \Rightarrow ExpAr a \rightarrow ExpAr a$$

que, dada uma expressão aritmética, calcula a sua derivada. Testes a fazer, de seguida:

**Propriedade [QuickCheck] 5** A função *sd* respeita as regras de derivação.

```
prop_const_rule :: (Real a, Floating a) => a -> Bool
prop_const_rule a = sd (N a) == N 0

prop_var_rule :: Bool
prop_var_rule = sd X == N 1

prop_sum_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool
prop_sum_rule exp1 exp2 = sd (Bin Sum exp1 exp2) == sum_rule where
  sum_rule = Bin Sum (sd exp1) (sd exp2)

prop_product_rule :: (Real a, Floating a) => ExpAr a -> ExpAr a -> Bool
prop_product_rule exp1 exp2 = sd (Bin Product exp1 exp2) == prod_rule where
  prod_rule = Bin Sum (Bin Product exp1 (sd exp2)) (Bin Product (sd exp1) exp2)

prop_e_rule :: (Real a, Floating a) => ExpAr a -> Bool
prop_e_rule exp = sd (Un E exp) == Bin Product (Un E exp) (sd exp)

prop_negate_rule :: (Real a, Floating a) => ExpAr a -> Bool
prop_negate_rule exp = sd (Un Negate exp) == Un Negate (sd exp)
```

5. Como foi visto, *Symbolic differentiation* não é a técnica mais eficaz para o cálculo do valor da derivada de uma expressão. *Automatic differentiation* resolve este problema calculando o valor da derivada em vez de manipular a expressão original.

Defina o gene do catamorfismo que ocorre na função

$$ad :: Floating a \Rightarrow a \rightarrow ExpAr a \rightarrow a$$

que, dada uma expressão aritmética e um ponto, calcula o valor da sua derivada nesse ponto, sem transformar manipular a expressão original. Testes a fazer, de seguida:

**Propriedade [QuickCheck] 6** Calcular o valor da derivada num ponto *r* via *ad* é equivalente a calcular a derivada da expressão e avalia-la no ponto *r*.

```
prop_congruent :: (Floating a, Real a) => a -> ExpAr a -> Bool
prop_congruent a exp = ad a exp == eval_exp a (sd exp)
```

## Problema 2

Nesta disciplina estudou-se como fazer **programação dinâmica** por cálculo, recorrendo à lei de recursividade mútua.<sup>4</sup>

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

$$\begin{aligned} fib\ 0 &= 1 \\ fib\ (n + 1) &= fib\ n \end{aligned}$$

---

<sup>4</sup>Lei (3.94) em [2], página 98.

$$\begin{aligned} f\,0 &= 1 \\ f\,(n+1) &= \text{fib}\,n + f\,n \end{aligned}$$

Obter-se-á de imediato

$$\begin{aligned} \text{fib}' &= \pi_1 \cdot \text{for loop init where} \\ \text{loop}\,(f, f) &= (f, \text{fib} + f) \\ \text{init} &= (1, 1) \end{aligned}$$

usando as regras seguintes:

- O corpo do ciclo *loop* terá tantos argumentos quanto o número de funções mutuamente recursivas.
- Para as variáveis escolhem-se os próprios nomes das funções, pela ordem que se achar conveniente.<sup>5</sup>
- Para os resultados vão-se buscar as expressões respectivas, retirando a variável  $n$ .
- Em *init* colecionam-se os resultados dos casos de base das funções, pela mesma ordem.

Mais um exemplo, envolvendo polinómios do segundo grau  $ax^2 + bx + c$  em  $\mathbb{N}_0$ . Seguindo o método estudado nas aulas<sup>6</sup>, de  $f\,x = ax^2 + bx + c$  derivam-se duas funções mutuamente recursivas:

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

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

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

O que se pede então, nesta pergunta? Dada a fórmula que dá o  $n$ -ésimo **número de Catalan**,

$$C_n = \frac{(2n)!}{(n+1)!(n!)} \quad (1)$$

derivar uma implementação de  $C_n$  que não calcule factoriais nenhuns. Isto é, derivar um ciclo-for

$$\text{cat} = \dots \cdot \text{for loop init where } \dots$$

que implemente esta função.

**Propriedade [QuickCheck] 7** A função proposta coincide com a definição dada:

$$\text{prop\_cat} = (\geq 0) \Rightarrow (\text{catdef} \equiv \text{cat})$$

**Sugestão:** Começar por estudar muito bem o processo de cálculo dado no anexo B para o problema (semelhante) da função exponencial.

## Problema 3

As **curvas de Bézier**, designação dada em honra ao engenheiro **Pierre Bézier**, são curvas ubíquas na área de computação gráfica, animação e modelação. Uma curva de Bézier é uma curva paramétrica, definida por um conjunto  $\{P_0, \dots, P_N\}$  de pontos de controlo, onde  $N$  é a ordem da curva.

O algoritmo de *De Casteljau* é um método recursivo capaz de calcular curvas de Bézier num ponto. Apesar de ser mais lento do que outras abordagens, este algoritmo é numericamente mais estável, trocando velocidade por correção.

<sup>5</sup>Podem obviamente usar-se outros símbolos, mas numa primeira leitura dá jeito usarem-se tais nomes.

<sup>6</sup>Secção 3.17 de [2] e tópico **Recursividade mútua** nos vídeos das aulas teóricas.

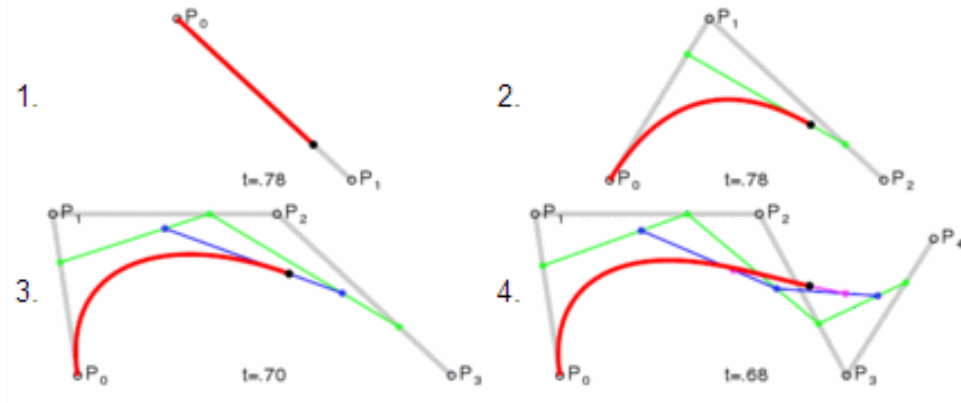


Figura 1: Exemplos de curvas de Bézier retirados da [Wikipedia](#).

De forma sucinta, o valor de uma curva de Bézier de um só ponto  $\{P_0\}$  (ordem 0) é o próprio ponto  $P_0$ . O valor de uma curva de Bézier de ordem  $N$  é calculado através da interpolação linear da curva de Bézier dos primeiros  $N - 1$  pontos e da curva de Bézier dos últimos  $N - 1$  pontos.

A interpolação linear entre 2 números, no intervalo  $[0, 1]$ , é dada pela seguinte função:

```
linear1d :: Q → Q → OverTime Q
linear1d a b = formula a b where
  formula :: Q → Q → Float → Q
  formula x y t = ((1.0 :: Q) - (to_Q t)) * x + (to_Q t) * y
```

A interpolação linear entre 2 pontos de dimensão  $N$  é calculada através da interpolação linear de cada dimensão.

O tipo de dados *NPoint* representa um ponto com  $N$  dimensões.

```
type NPoint = [Q]
```

Por exemplo, um ponto de 2 dimensões e um ponto de 3 dimensões podem ser representados, respetivamente, por:

```
p2d = [1.2, 3.4]
p3d = [0.2, 10.3, 2.4]
```

O tipo de dados *OverTime a* representa um termo do tipo  $a$  num dado instante (dado por um *Float*).

```
type OverTime a = Float → a
```

O anexo C tem definida a função

```
calcLine :: NPoint → (NPoint → OverTime NPoint)
```

que calcula a interpolação linear entre 2 pontos, e a função

```
deCasteljau :: [NPoint] → OverTime NPoint
```

que implementa o algoritmo respectivo.

1. Implemente *calcLine* como um catamorfismo de listas, testando a sua definição com a propriedade:

**Propriedade [QuickCheck] 8** *Definição alternativa.*

```
prop_calcLine_def :: NPoint → NPoint → Float → Bool
prop_calcLine_def p q d = calcLine p q d ≡ zipWithM linear1d p q d
```

2. Implemente a função *deCasteljau* como um hilomorfismo, testando agora a propriedade:

**Propriedade [QuickCheck] 9** *Curvas de Bézier são simétricas.*

```
prop_bezier_sym :: [[Q]] → Gen Bool
prop_bezier_sym l = all (<Δ) · calc_difs · bezs ($) elements ps where
  calc_difs = (λ(x, y) → zipWith (λw v → if w ≥ v then w - v else v - w) x y)
  bezs t = (deCasteljau l t, deCasteljau (reverse l) (from_Q (1 - (to_Q t))))
  Δ = 1e-2
```

- Corra a função `runBezier` e aprecie o seu trabalho<sup>7</sup> clicando na janela que é aberta (que contém, a verde, um ponto inicial) com o botão esquerdo do rato para adicionar mais pontos. A tecla `Delete` apaga o ponto mais recente.

## Problema 4

Seja dada a fórmula que calcula a média de uma lista não vazia  $x$ ,

$$\text{avg } x = \frac{1}{k} \sum_{i=1}^k x_i \quad (2)$$

onde  $k = \text{length } x$ . Isto é, para sabermos a média de uma lista precisamos de dois catamorfismos: o que faz o somatório e o que calcula o comprimento a lista. Contudo, é fácil de ver que

$$\begin{aligned} \text{avg } [a] &= a \\ \text{avg}(a : x) &= \frac{1}{k+1} (a + \sum_{i=1}^k x_i) = \frac{a + k(\text{avg } x)}{k+1} \text{ para } k = \text{length } x \end{aligned}$$

Logo `avg` está em recursividade mútua com `length` e o par de funções pode ser expresso por um único catamorfismo, significando que a lista apenas é percorrida uma vez.

- Recorra à lei de recursividade mútua para derivar a função `avg_aux = ([b, q])` tal que `avg_aux = (avg, length)` em listas não vazias.
- Generalize o raciocínio anterior para o cálculo da média de todos os elementos de uma `LTree` recorrendo a uma única travessia da árvore (i.e. catamorfismo).

Verifique as suas funções testando a propriedade seguinte:

**Propriedade [QuickCheck] 10** *A média de uma lista não vazia e de uma `LTree` com os mesmos elementos coincide, a menos de um erro de 0.1 milésimas:*

```
prop_avg = nonempty ⇒ diff ≤ 0.000001 where
  diff l = avg l - (avgLTree · genLTree) l
  genLTree = ([lsplit])
  nonempty = (>[])
```

## Problema 5

(NB: Esta questão é **opcional** e funciona como **valorização** apenas para os alunos que desejarem fazê-la.)

Existem muitas linguagens funcionais para além do `Haskell`, que é a linguagem usada neste trabalho prático. Uma delas é o `F#` da Microsoft. Na directoria `fsharp` encontram-se os módulos `Cp`, `Nat` e `LTree` codificados em `F#`. O que se pede é a biblioteca `BTree` escrita na mesma linguagem.

Modo de execução: o código que tiverem produzido nesta pergunta deve ser colocado entre o `\begin{verbatim}` e o `\end{verbatim}` da correspondente parte do anexo D. Para além disso, os grupos podem demonstrar o código na oral.

<sup>7</sup>A representação em Gloss é uma adaptação de um `projeto` de Harold Cooper.



# Anexos

## A Como exprimir cálculos e diagramas em LaTeX/lhs2tex

Como primeiro exemplo, estudar o texto fonte deste trabalho para obter o efeito:<sup>8</sup>

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

Os diagramas podem ser produzidos recorrendo à *package* L<sup>A</sup>T<sub>E</sub>X **xymatrix**, por exemplo:

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

## B Programação dinâmica por recursividade múltipla

Neste anexo dão-se os detalhes da resolução do Exercício 3.30 dos apontamentos da disciplina<sup>9</sup>, onde se pretende implementar um ciclo que implemente o cálculo da aproximação até  $i = n$  da função exponencial  $exp\ x = e^x$ , via série de Taylor:

$$exp\ x = \sum_{i=0}^{\infty} \frac{x^i}{i!} \quad (3)$$

Seja  $e\ x\ n = \sum_{i=0}^n \frac{x^i}{i!}$  a função que dá essa aproximação. É fácil de ver que  $e\ x\ 0 = 1$  e que  $e\ x\ (n+1) = e\ x\ n + \frac{x^{n+1}}{(n+1)!}$ . Se definirmos  $h\ x\ n = \frac{x^{n+1}}{(n+1)!}$  teremos  $e\ x$  e  $h\ x$  em recursividade mútua. Se repetirmos o processo para  $h\ x\ n$  etc obteremos no total três funções nessa mesma situação:

$$\begin{aligned}
 e\ x\ 0 &= 1 \\
 e\ x\ (n+1) &= h\ x\ n + e\ x\ n \\
 h\ x\ 0 &= x \\
 h\ x\ (n+1) &= x / (s\ n) * h\ x\ n \\
 s\ 0 &= 2 \\
 s\ (n+1) &= 1 + s\ n
 \end{aligned}$$

Segundo a *regra de algibeira* descrita na página 3.1 deste enunciado, ter-se-á, de imediato:

$$\begin{aligned}
 e'\ x &= prj \cdot \text{for loop } init \textbf{ where} \\
 init &= (1, x, 2) \\
 loop\ (e, h, s) &= (h + e, x / s * h, 1 + s) \\
 prj\ (e, h, s) &= e
 \end{aligned}$$

---

<sup>8</sup>Exemplos tirados de [2].

<sup>9</sup>Cf. [2], página 102.

## C Código fornecido

### Problema 1

```
expd :: Floating a => a -> a
expd = Prelude.exp
type OutExpAr a = () + (a + ((BinOp, (ExpAr a, ExpAr a)) + (UnOp, ExpAr a)))
```

### Problema 2

Definição da série de Catalan usando factoriais (4):

$$catdef\ n = (2 * n)! \div ((n + 1)! * n!)$$

Oráculo para inspeção dos primeiros 26 números de Catalan<sup>10</sup>:

```
oracle = [
  1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845,
  35357670, 129644790, 477638700, 1767263190, 6564120420, 24466267020,
  91482563640, 343059613650, 1289904147324, 4861946401452
]
```

### Problema 3

Algoritmo:

```
deCasteljau :: [NPoint] -> OverTime NPoint
deCasteljau [] = nil
deCasteljau [p] = p
deCasteljau l = λpt -> (calcLine (p pt) (q pt)) pt where
  p = deCasteljau (init l)
  q = deCasteljau (tail l)
```

Função auxiliar:

```
calcLine :: NPoint -> (NPoint -> OverTime NPoint)
calcLine [] = nil
calcLine (p : x) = λp -> (calcLine x) where
  g :: (Q, NPoint -> OverTime NPoint) -> (NPoint -> OverTime NPoint)
  g (d, f) l = case l of
    [] -> nil
    (x : xs) -> λz -> concat $ (sequenceA [singl · linear1d d x, f xs]) z
```

2D:

```
bezier2d :: [NPoint] -> OverTime (Float, Float)
bezier2d [] = (0, 0)
bezier2d l = λz -> (fromQ × fromQ) · (λ[x, y] -> (x, y)) $ ((deCasteljau l) z)
```

Modelo:

```
data World = World { points :: [NPoint]
  , time :: Float
  }
initW :: World
initW = World [] 0
tick :: Float -> World -> World
```

---

<sup>10</sup>Fonte: [Wikipedia](#).

```

tick dt world = world { time = (time world) + dt }
actions :: Event → World → World
actions (EventKey (MouseButton LeftButton) Down _ p) world =
  world { points = (points world) ++ [(λ(x, y) → map toQ [x, y]) p] }
actions (EventKey (SpecialKey KeyDelete) Down _ _) world =
  world { points = cond (≡ []) id init (points world) }
actions _ world = world
scaleTime :: World → Float
scaleTime w = (1 + cos (time w)) / 2
bezier2dAtTime :: World → (Float, Float)
bezier2dAtTime w = (bezier2dAt w) (scaleTime w)
bezier2dAt :: World → OverTime (Float, Float)
bezier2dAt w = bezier2d (points w)
thicCirc :: Picture
thicCirc = ThickCircle 4 10
ps :: [Float]
ps = map fromQ ps' where
  ps' :: [Q]
  ps' = [0, 0.01 .. 1] -- interval

```

Gloss:

```

picture :: World → Picture
picture world = Pictures
  [ animateBezier (scaleTime world) (points world)
  , Color white · Line · map (bezier2dAt world) $ ps
  , Color blue · Pictures $ [ Translate (fromQ x) (fromQ y) thicCirc | [x, y] ← points world ]
  , Color green $ Translate cx cy thicCirc
  ] where
  (cx, cy) = bezier2dAtTime world

```

Animação:

```

animateBezier :: Float → [NPoint] → Picture
animateBezier _ [] = Blank
animateBezier _ [_] = Blank
animateBezier t l = Pictures
  [ animateBezier t (init l)
  , animateBezier t (tail l)
  , Color red · Line $ [a, b]
  , Color orange $ Translate ax ay thicCirc
  , Color orange $ Translate bx by thicCirc
  ] where
  a@(ax, ay) = bezier2d (init l) t
  b@(bx, by) = bezier2d (tail l) t

```

Propriedades e main:

```

runBezier :: IO ()
runBezier = play (InWindow "Bézier" (600, 600) (0, 0))
  black 50 initW picture actions tick
runBezierSym :: IO ()
runBezierSym = quickCheckWith (stdArgs { maxSize = 20, maxSuccess = 200 }) prop_bezier_sym

```

Compilação e execução dentro do interpretador:<sup>11</sup>

```

main = runBezier
run = do { system "ghc cp2021t"; system "./cp2021t" }

```

---

<sup>11</sup>Pode ser útil em testes envolvendo **Gloss**. Nesse caso, o teste em causa deve fazer parte de uma função *main*.

## QuickCheck

Código para geração de testes:

```
instance Arbitrary UnOp where
  arbitrary = elements [Negate, E]
instance Arbitrary BinOp where
  arbitrary = elements [Sum, Product]
instance (Arbitrary a) => Arbitrary (ExpAr a) where
  arbitrary = do
    binop <- arbitrary
    unop <- arbitrary
    exp1 <- arbitrary
    exp2 <- arbitrary
    a <- arbitrary
    frequency · map (id × pure) $ [(20, X), (15, N a), (35, Bin binop exp1 exp2), (30, Un unop exp1)]

infixr 5 ?
(?) :: Real a => a → a → Bool
(?) x y = (toℚ x) ≡ (toℚ y)
```

## Outras funções auxiliares

Lógicas:

```
infixr 0 =>
(=>) :: (Testable prop) => (a → Bool) → (a → prop) → a → Property
p => f = λa → p a => f a

infixr 0 <=>
(<=>) :: (a → Bool) → (a → Bool) → a → Property
p <=> f = λa → (p a => property (f a)) .&&. (f a => property (p a))

infixr 4 ≡
(≡) :: Eq b => (a → b) → (a → b) → (a → Bool)
f ≡ g = λa → f a ≡ g a

infixr 4 ≤
(≤) :: Ord b => (a → b) → (a → b) → (a → Bool)
f ≤ g = λa → f a ≤ g a

infixr 4 ∧
(∧) :: (a → Bool) → (a → Bool) → (a → Bool)
f ∧ g = λa → (f a) ∧ (g a)
```

## D 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, diagramas e/ou outras funções auxiliares que sejam necessárias.

Valoriza-se a escrita de *pouco* código que corresponda a soluções simples e elegantes.

### Problema 1

São dadas:

```
cataExpAr g = g · recExpAr (cataExpAr g) · outExpAr
anaExpAr g = inExpAr · recExpAr (anaExpAr g) · g
hyloExpAr h g = cataExpAr h · anaExpAr g
eval_exp :: Floating a => a → (ExpAr a) → a
```

$eval\_exp\ a = cataExpAr\ (g\_eval\_exp\ a)$   
 $optimize\_eval :: (Floating\ a, Eq\ a) \Rightarrow a \rightarrow (ExpAr\ a) \rightarrow a$   
 $optimize\_eval\ a = hyloExpAr\ (gopt\ a)\ clean$   
 $sd :: Floating\ a \Rightarrow ExpAr\ a \rightarrow ExpAr\ a$   
 $sd = \pi_2 \cdot cataExpAr\ sd\_gen$   
 $ad :: Floating\ a \Rightarrow a \rightarrow ExpAr\ a \rightarrow a$   
 $ad\ v = \pi_2 \cdot cataExpAr\ (ad\_gen\ v)$

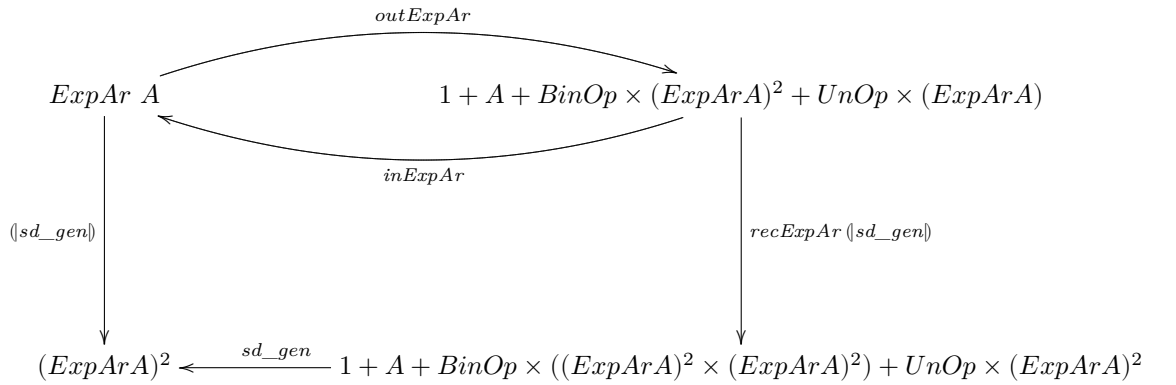
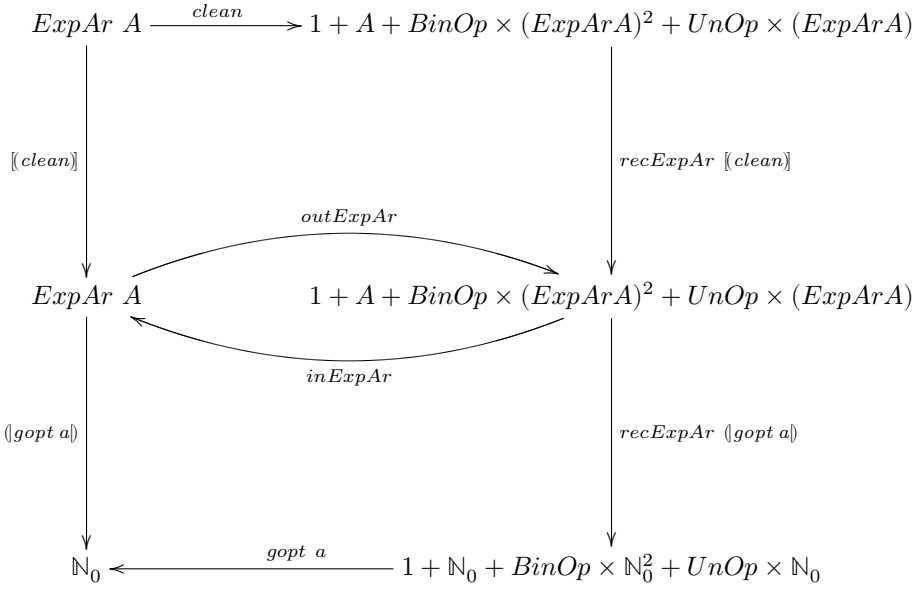
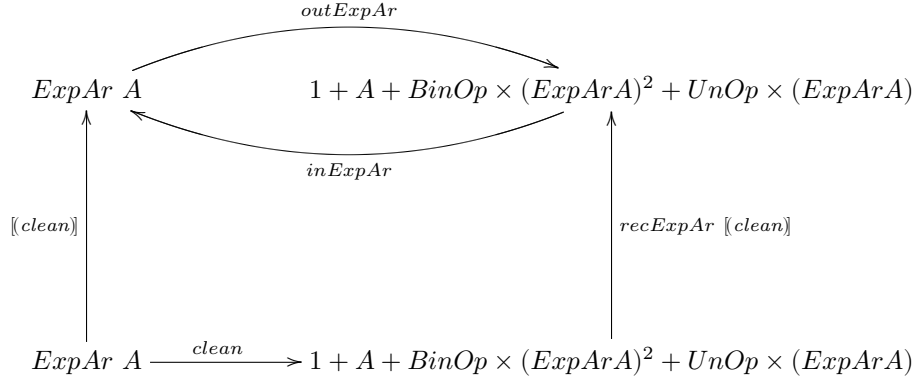
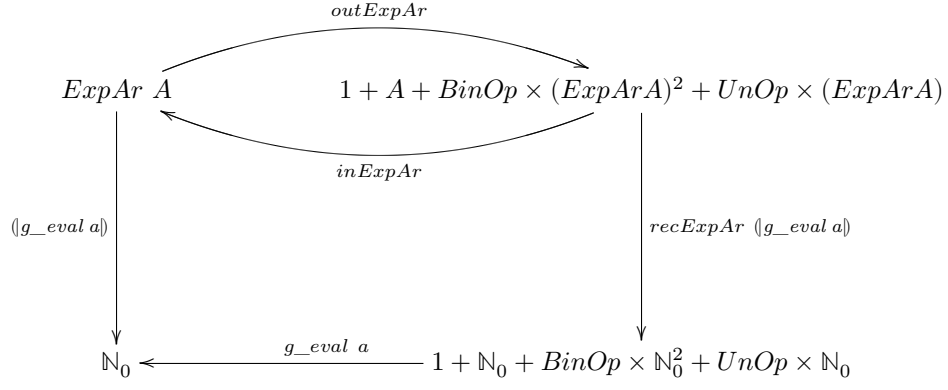
$$\begin{aligned}
& outExpAr \cdot inExpAr = id \\
\equiv & \quad \{ \text{Definição de } inExpAr; \text{ Fusão-+; Cancelamento-+} \} \\
& \left\{ \begin{array}{l} outExpAr \cdot \underline{X} = id \cdot i_1 \\ outExpAr \cdot N = id \cdot i_2 \cdot i_1 \\ \left\{ \begin{array}{l} outExpAr \cdot bin = id \cdot i_2 \cdot i_2 \cdot i_1 \\ outExpAr \cdot \widehat{Un} = id \cdot i_2 \cdot i_2 \cdot i_2 \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \text{Igualdade extensional, Natural-id} \} \\
& \left\{ \begin{array}{l} (outExpAr \cdot \underline{X})\ () = i_1\ () \\ \left\{ \begin{array}{l} (outExpAr \cdot N)\ a = (i_2 \cdot i_1)\ a \\ \left\{ \begin{array}{l} (outExpAr \cdot bin)\ (op, (l, r)) = (i_2 \cdot i_2 \cdot i_1)\ (op, (l, r)) \\ (outExpAr \cdot \widehat{Un})\ (op, a) = (i_2 \cdot i_2 \cdot i_2)\ (op, a) \end{array} \right. \end{array} \right. \end{array} \right. \\
\equiv & \quad \{ \text{Def-comp, Def-const, Def-N, Def-bin, Def-Uncurry, Def-Un} \} \\
& \left\{ \begin{array}{l} outExpAr\ X = i_1\ () \\ \left\{ \begin{array}{l} outExpAr\ (N\ a) = i_2\ \$\ i_1\ a \\ \left\{ \begin{array}{l} outExpAr\ (Bin\ op\ l\ r) = i_2\ \$\ i_2\ \$\ i_1\ (op, (l, r)) \\ outExpAr\ (Un\ op\ a) = i_2\ \$\ i_2\ \$\ i_2\ (op, a) \end{array} \right. \end{array} \right. \end{array} \right.
\end{aligned}$$

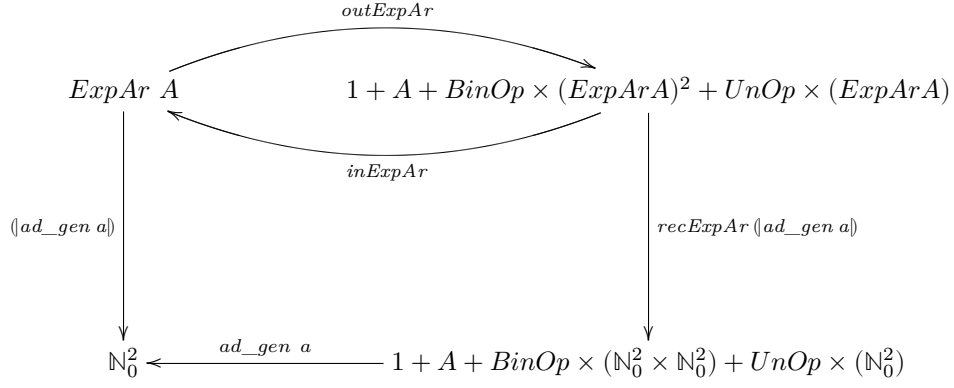
□

$$\begin{array}{ccc}
& outExpAr & \\
& \curvearrowright & \\
ExpAr\ A & \xrightarrow{\cong} & 1 + A + BinOp \times (ExpAr\ A)^2 + UnOp \times (ExpAr\ A) \\
& \curvearrowleft & \\
& inExpAr &
\end{array}$$

$$\begin{aligned}
& recExprAr\ f = id + (id + (id \times (f \times f) + id \times f)) \\
\equiv & \quad \{ \text{Def-baseExpAr} \} \\
& recExprAr\ f = baseExpAr\ id\ id\ id\ ff\ id\ f
\end{aligned}$$

□





Definir:

$\text{outExpAr } X = i_1 ()$   
 $\text{outExpAr } (N a) = i_2 \$ i_1 a$   
 $\text{outExpAr } (\text{Bin op } l r) = i_2 \$ i_2 \$ i_1 (\text{op}, (l, r))$   
 $\text{outExpAr } (\text{Un op } a) = i_2 \$ i_2 \$ i_2 (\text{op}, a)$   
 $\text{recExpAr } f = \text{baseExpAr id id id ff id f}$   
 $g\_eval\_exp x (i_1 ()) = x$   
 $g\_eval\_exp x (i_2 (i_1 a)) = a$   
 $g\_eval\_exp x (i_2 (i_2 (i_1 (\text{Sum}, (e, d))))) = e + d$   
 $g\_eval\_exp x (i_2 (i_2 (i_1 (\text{Product}, (e, d))))) = e * d$   
 $g\_eval\_exp x (i_2 (i_2 (i_2 (\text{Negate}, a)))) = \text{negate } a$   
 $g\_eval\_exp x (i_2 (i_2 (i_2 (E, a)))) = \text{expd } a$   
 $\text{clean } a = (\text{outExpAr} \cdot h) a \text{ where}$   
 $h (\text{Bin Product } (N 0) r) = N 0$   
 $h (\text{Bin Product } r (N 0)) = N 0$   
 $h (\text{Un } E (N 0)) = N 1$   
 $h (\text{Un Negate } (N 0)) = N 0$   
 $h x = x$   
 $\text{gopt } a = g\_eval\_exp a$

$\text{sd\_gen} :: \text{Floating } a \Rightarrow$   
 $() + (a + ((\text{BinOp}, ((\text{ExpAr } a, \text{ExpAr } a),$   
 $(\text{ExpAr } a, \text{ExpAr } a))) + \cdot (\text{UnOp}, (\text{ExpAr } a, \text{ExpAr } a)))) \rightarrow (\text{ExpAr } a, \text{ExpAr } a)$   
 $\text{sd\_gen } (i_1 ()) = (X, N 1)$   
 $\text{sd\_gen } (i_2 (i_1 a)) = (N a, N 0)$   
 $\text{sd\_gen } (i_2 (i_2 (i_1 (\text{Sum}, ((e_1, d1), (e_2, d2)))))) = (\text{Bin Sum } e_1 e_2, \text{Bin Sum } d1 d2)$   
 $\text{sd\_gen } (i_2 (i_2 (i_1 (\text{Product}, ((e_1, d1), (e_2, d2))))))$   
 $= (\text{Bin Product } e_1 e_2, \text{Bin Sum } (\text{Bin Product } e_1 d2) (\text{Bin Product } d1 e_2))$   
 $\text{sd\_gen } (i_2 (i_2 (i_2 (\text{Negate}, (e, d)))) = (\text{Un Negate } e, \text{Un Negate } d)$   
 $\text{sd\_gen } (i_2 (i_2 (i_2 (E, (e, d)))) = (\text{Un } E e, \text{Bin Product } (\text{Un } E e) d)$

$\text{ad\_gen } x (i_1 ()) = (x, 1)$   
 $\text{ad\_gen } x (i_2 (i_1 a)) = (a, 0)$   
 $\text{ad\_gen } x (i_2 (i_2 (i_1 (\text{Sum}, ((e_1, d1), (e_2, d2)))))) = (e_1 + e_2, d1 + d2)$   
 $\text{ad\_gen } x (i_2 (i_2 (i_1 (\text{Product}, ((e_1, d1), (e_2, d2)))))) = (e_1 * e_2, e_1 * d2 + e_2 * d1)$   
 $\text{ad\_gen } x (i_2 (i_2 (i_2 (\text{Negate}, (e, d)))) = (\text{negate } e, \text{negate } d)$   
 $\text{ad\_gen } x (i_2 (i_2 (i_2 (E, (e, d)))) = (\text{expd } e, d * (\text{expd } e))$

## Problema 2

Definir

$loop = g$  **where**  $g(c, a, b) = (c * a \div b, a + 4, b + 1)$   
 $inic = (1, 2, 2)$   
 $prj = p$  **where**  $p(c, -, -) = c$

por forma a que

$$cat = prj \cdot \text{for } loop \text{ inic}$$

seja a função pretendida. **NB:** usar divisão inteira. Apresentar de seguida a justificação da solução encontrada.

$$C_n = \frac{(2n)!}{(n+1)!(n!)} \quad (4)$$

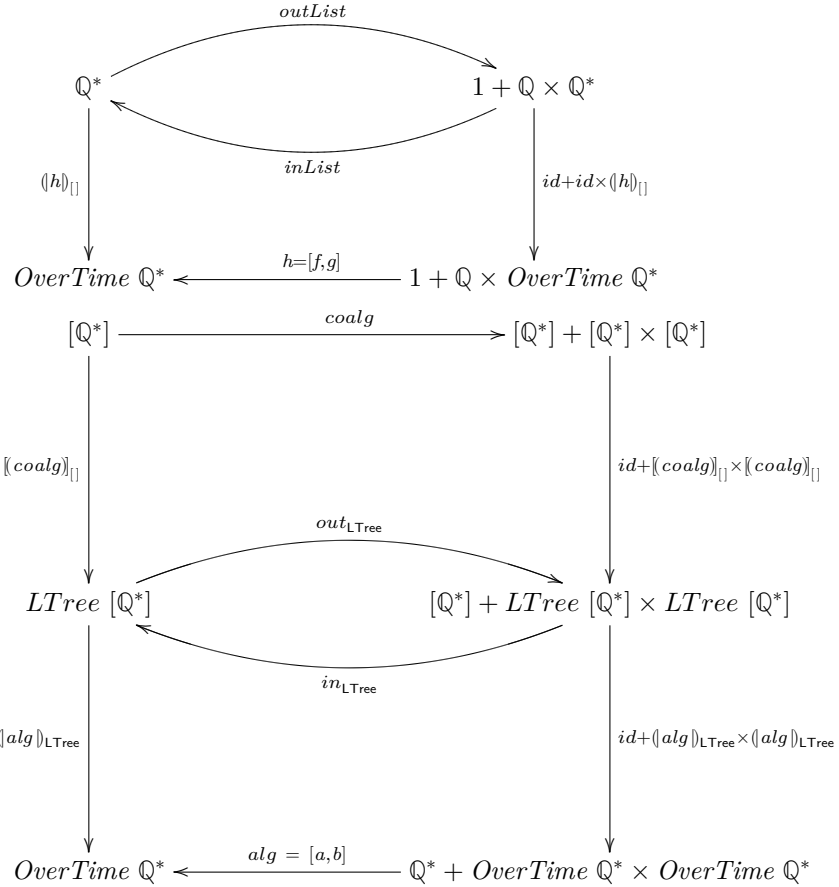
$$\begin{aligned} C_0 &= 1 \\ C_{n+1} &= \frac{C_n a_n}{b_n} \end{aligned}$$

$$\begin{aligned} a_n &= 4n + 2 \\ b_n &= n + 2 \end{aligned}$$

$$\begin{aligned} a_0 &= 2 \\ a_{n+1} &= a_n + 4 \end{aligned}$$

$$\begin{aligned} b_0 &= 2 \\ b_{n+1} &= b_n + 1 \end{aligned}$$

### Problema 3



$calcLine :: NPoint \rightarrow (NPoint \rightarrow OverTime\ NPoint)$   
 $calcLine = (h)_[]$  **where**



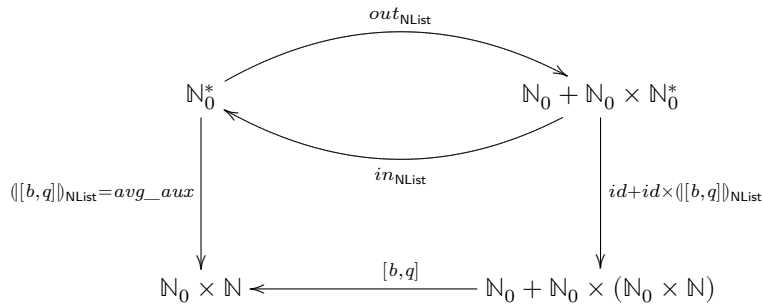
$h = [f, g]$  **where**  
 $f \_ = nil$   
 $g \_ [] = nil$   
 $g (d, f) (x : xs) = \lambda z \rightarrow concat \$ (sequenceA [singl \cdot linear1d d x, f xs]) z$

$deCasteljau :: [NPoint] \rightarrow OverTime NPoint$   
 $deCasteljau = hyloAlgForm \text{ alg } coalg$  **where**  
 $coalg = c$  **where**  
 $c [] = i_1 []$   
 $c [a] = i_1 [a]$   
 $c l = i_2 (init l, tail l)$   
 $alg = [a, b]$  **where**  
 $a [] = nil$   
 $a [x] = \underline{x}$   
 $b (e, d) = \lambda pt \rightarrow (calcLine (e pt) (d pt)) pt$   
 $hyloAlgForm = hyloLTree$

Uma outra soluo para o deCasteljau, criando um novo tipo de dados intermedio.

$deCasteljau' :: [NPoint] \rightarrow OverTime NPoint$   
 $deCasteljau' = hyloAlgForm' \text{ alg } coalg$  **where**  
 $coalg = (id + (id + \langle init, tail \rangle)) \cdot outSL$   
 $alg = [\underline{nil}, a]$   
 $a = [\_, b]$   
 $b (e, d) = \lambda pt \rightarrow (calcLine (e pt) (d pt)) pt$   
 $outSL [] = i_1 ()$   
 $outSL [a] = i_2 (i_1 a)$   
 $outSL l = i_2 (i_2 l)$   
 $hyloAlgForm' = h$  **where**  
 $h a b = \langle a \rangle_{Castel} \cdot \langle b \rangle_{Castel}$   
**data** Castel  $a = Empty \mid Single a \mid InitTail (Castel a, Castel a)$  **deriving** Show  
 $in_{Castel} = [Empty, [Single, InitTail]]$   
 $out_{Castel} Empty = i_1 ()$   
 $out_{Castel} (Single a) = i_2 (i_1 a)$   
 $out_{Castel} (InitTail (e, d)) = i_2 (i_2 (e, d))$   
 $rec_{Castel} f = id + (id + f \times f)$   
 $\langle f \rangle_{Castel} = f \cdot rec_{Castel} \langle f \rangle_{Castel} \cdot out_{Castel}$   
 $\langle g \rangle_{Castel} = in_{Castel} \cdot rec_{Castel} \langle g \rangle_{Castel} \cdot g$

## Problema 4

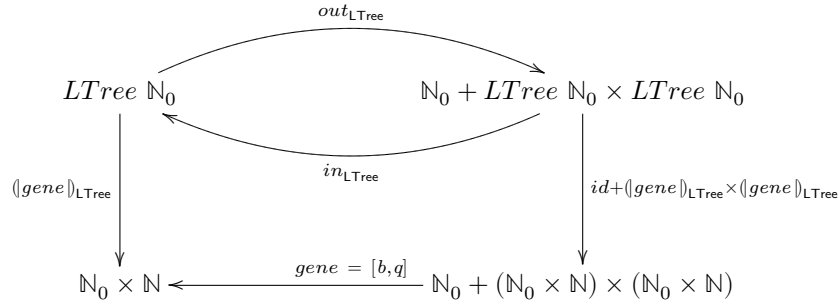


$$\begin{aligned}
& \langle avg, length \rangle = ([b, q])_{\mathbf{NList}} \\
\equiv & \quad \{ \text{Univeral-cata} \} \\
& \langle avg, length \rangle \cdot in_{\mathbf{NList}} = ([b, q])_{\mathbf{NList}} \cdot rec_{\mathbf{NList}} \langle avg, length \rangle \\
\equiv & \quad \{ \text{Fusão-+}, \text{Absorção-+}, \text{Eq-+}, \text{Definição de } in_{\mathbf{NList}}, \text{Definição de } rec_{\mathbf{NList}} \} \\
& \begin{cases} \langle avg, length \rangle \cdot singl = b \cdot id \\ \langle avg, length \rangle \cdot cons = q \cdot id \times \langle avg, length \rangle \end{cases} \\
\equiv & \quad \{ \text{Igualdade extensional}, \text{Natural-id} \} \\
& \begin{cases} (\langle avg, length \rangle \cdot singl) x = b x \\ (\langle avg, length \rangle \cdot cons) (x, xs) = (q \cdot id \times \langle avg, length \rangle) (x, xs) \end{cases} \\
\equiv & \quad \{ \text{Def-comp}, \text{Natural-id}, \text{Def-}\times, \text{Def-split}, \text{Definição de singl}, \text{Definição de cons} \} \\
& \begin{cases} \langle avg, length \rangle [x] = b x \\ \langle avg, length \rangle (x : xs) = q (x, (avg xs, length xs)) \end{cases} \\
& \square
\end{aligned}$$

Solução para listas não vazias:

$$avg = \pi_1 \cdot avg\_aux$$

$$\begin{aligned}
in_{\mathbf{NList}} &= [singl, cons] \\
out_{\mathbf{NList}} [a] &= i_1 a \\
out_{\mathbf{NList}} (a : x) &= i_2 (a, x) \\
rec_{\mathbf{NList}} f &= id + id \times f \\
([g])_{\mathbf{NList}} &= g \cdot rec_{\mathbf{NList}} ([g])_{\mathbf{NList}} \cdot out_{\mathbf{NList}} \\
avg\_aux &= ([b, q])_{\mathbf{NList}} \text{ where} \\
b x &= (x, 1) \\
q (x, (a, l)) &= ((x + (a * l)) / (l + 1), l + 1)
\end{aligned}$$



$$\begin{aligned}
& \langle avg, length \rangle = ([gene])_{\mathbf{LTree}} \\
\equiv & \quad \{ \text{Univeral-cata}, \text{gene} = [b, q] \} \\
& \langle avg, length \rangle \cdot in_{\mathbf{LTree}} = ([b, q])_{\mathbf{LTree}} \cdot rec_{\mathbf{LTree}} \langle avg, length \rangle \\
\equiv & \quad \{ \text{Fusão-+}, \text{Absorção-+}, \text{Eq-+}, \text{Definição de } in_{\mathbf{L}}, \text{Definição de } rec_{\mathbf{LTree}} \} \\
& \begin{cases} \langle avg, length \rangle \cdot Leaf = b \cdot id \\ \langle avg, length \rangle \cdot Fork = q \cdot \langle avg, length \rangle \times \langle avg, length \rangle \end{cases} \\
\equiv & \quad \{ \text{Igualdade extensional}, \text{Natural-id} \} \\
& \begin{cases} (\langle avg, length \rangle \cdot Leaf) a = b a \\ (\langle avg, length \rangle \cdot Fork) (\mathbf{LTree} a, \mathbf{LTree} a) = (q \cdot \langle avg, length \rangle \times \langle avg, length \rangle) (\mathbf{LTree} a, \mathbf{LTree} a) \end{cases} \\
\equiv & \quad \{ \text{Def-comp}, \text{Natural-id}, \text{Def-}\times, \text{Def-split}, \text{Definição de Leaf}, \text{Definição de Fork} \}
\end{aligned}$$

$$\left\{ \begin{array}{l} \langle \text{avg}, \text{length} \rangle (\text{Leaf } a) = b \ a \\ \langle \text{avg}, \text{length} \rangle (\text{Fork } (\text{LTree } a, \text{LTree } a)) = q ((\text{avg } (\text{LTree } a), \text{length } (\text{LTree } a)), (\text{avg } (\text{LTree } a), \text{length } (\text{LTree } a))) \end{array} \right.$$

□ Solução para árvores de tipo **LTree**:

```
avgLTree = π1 · (|gene|)LTree where
  gene = [b, q]
  b a = (a, 1)
  q ((a1, l1), (a2, l2)) = (((a1 * l1) + (a2 * l2)) / (l1 + l2), l1 + l2)
```

## Problema 5

Inserir em baixo o código **F#** desenvolvido, entre `\begin{verbatim}` e `\end{verbatim}`:

## E Resoluções Alternativas e Simplificações sugeridas por Alef

Nessa seção mostro uma forma alternativa que percebi de resolver alguns problemas que não poderiam ser colocadas na seção principal por alguma razão, entre elas por usarem extensões que não estavam já no trabalho.

Também tem simplificações triviais que acho que, para alguns pode facilitar o entendimento. São simplesmente funções que já estão no trabalho reescritas com uma sintaxe mais simples em Haskell (muitas vezes usando lambda case). Conversando com o Tiago, ele achou que seria mais complicado explicar dessa forma. Como no final são equivalentes, decidi deixar aqui caso alguém ache mais fácil ou simplesmente esteja interessado em ver um pouco mais da sintaxe de Haskell, já que a última vez que tivemos uma cadeira que usasse a linguagem diretamente foi no 1º ano.

### E.1 Problema 1

Para criar uma interpretação de um tipo  $A$  como um tipo  $B$ . Assim, por exemplo, posso definir que uma expressão  $x$  do tipo **ExpAr**  $a$  pode ser interpretada como  $i_1()$  do tipo **OutExpAr**  $a$ .

```
class Interpretation a b where
  to :: a → b
```

A fim de diminuir o número de parêntese e facilitar a legibilidade defini as funções: **bimap** de tuplos  $((,))$ :

```
infixr 6 ×
type a × b = (a, b)
(×) :: (a → b) → (c → d) → (a, c) → (b, d)
(×) = (×)
```

**bimap** de  $\cdot + \cdot$ :

```
infixr 4 +
(+) :: (a → b) → (c → d) → a ∥ c → b ∥ d
(+) = (+)
```

```
infixr 4 ∥
type (∥) = · + ·
(∥) :: (a → c) → (b → c) → a ∥ b → c
(∥) = [·, ·]
```

Novamente, para simplificar a tipagem:

```
type BinExp d = BinOp × ExpAr d × ExpAr d
```

Note que por conta de precedência  $BinExp\ d \equiv BinOp \times (ExpAr\ d \times ExpAr\ d)$ .

**type**  $UnExp\ d = UnOp \times ExpAr\ d$

Isso é uma redefinição do que o Professor definiu. É igual excepto os símbolos mais fáceis de ler.

**type**  $OutExpAr\ a = () \coprod a \coprod BinExp\ a \coprod UnExp\ a$

Vamos criar uma interpretação de  $ExpAr\ a$  como  $OutExpAr\ a$ . Ou seja, essa interpretação é  $out_{ExpAr}$ .

**instance**  $Interpretation\ (ExpAr\ a)\ (OutExpAr\ a)\ \mathbf{where}$

$to\ X = i_1\ ()$   
 $to\ (N\ a\ \_) = i_2\ \$\ i_1\ a$   
 $to\ (Bin\ op\ l\ r) = i_2\ \$\ i_2\ \$\ i_1\ (op, (l, r))$   
 $to\ (Un\ op\ a) = i_2\ \$\ i_2\ \$\ i_2\ (op, a)$

Como dito, temos

$out_{ExpAr} :: ExpAr\ a \rightarrow OutExpAr\ a$   
 $out_{ExpAr} = to :: ExpAr\ a \rightarrow OutExpAr\ a$

Agora vou interpretar os símbolos que representam as operações como as funções que representam essas operações.

Interpretamos cada símbolo  $BinOp$  como uma função  $(c, c) \rightarrow c$ . Por exemplo, o símbolo  $Sum$  é interpretado como a função  $add$ .

**instance**  $(Num\ c) \Rightarrow Interpretation\ BinOp\ ((c, c) \rightarrow c)\ \mathbf{where}$

$to\ Sum = add$   
 $to\ Product = mul$

Na nossa linguagem mais usual:

$out_{BinOp} :: Num\ c \Rightarrow BinOp \rightarrow (c \times c) \rightarrow c$   
 $out_{BinOp} = to :: (Num\ c) \Rightarrow BinOp \rightarrow (c \times c \rightarrow c)$

Interpretamos cada símbolo  $UnOp$  como uma função  $c \rightarrow c$  onde  $c$  é da classe  $Floating$ .

**instance**  $(Floating\ c) \Rightarrow Interpretation\ UnOp\ (c \rightarrow c)\ \mathbf{where}$

$to\ Negate = negate$   
 $to\ E = Prelude.exp$

Na nossa linguagem mais usual

$out_{UnOp} :: Floating\ c \Rightarrow UnOp \rightarrow (c \rightarrow c)$   
 $out_{UnOp} = to :: (Floating\ c) \Rightarrow UnOp \rightarrow (c \rightarrow c)$

Graças a essas funções auxiliares (que acho intuitivas), podemos simplificar a escrita de  $g\_eval\_exp$ :

$g\_eval\_exp :: Floating\ c \Rightarrow c \rightarrow b \coprod c \coprod BinOp \times c \times c \coprod UnOp \times c \rightarrow c$   
 $g\_eval\_exp\ a = \underline{a} \coprod id \coprod ap \cdot (out_{BinOp} \times id) \coprod ap \cdot (out_{UnOp} \times id)$

Sabemos que  $e^0 = 1$ ,  $-0 = 0$  e  $a = 0 \vee b = 0 \implies ab = 0$ . Nós optimizamos esses 4 casos:

$clean :: (Eq\ a, Num\ a) \Rightarrow ExpAr\ a \rightarrow OutExpAr\ a$

$clean = \lambda \mathbf{case}$

$(Un\ E\ \_)\ (N\ 0)\ \_ \rightarrow tag\ 1$   
 $(Un\ Negate\ \_)\ (N\ 0)\ \_ \rightarrow tag\ 0$   
 $(Bin\ Product\ \_)\ (N\ 0)\ \_ \rightarrow tag\ 0$   
 $(Bin\ Product\ \_)\ \_ (N\ 0) \rightarrow tag\ 0$   
 $a \rightarrow out_{ExpAr}\ a$   
**where**  $tag = i_2 \cdot i_1$

Baseado na função *dup* definida em **Cp.hs**:

```
type Dup d = d × d
```

Mais dois sinônimos:

```
type Bin d = BinOp × Dup d
type Un d = UnOp × d
```

A fim de criar código mais sucinto extrai o que se repetia nas funções *sd\_gen* e *ad\_gen* do Tiago.

```
binaux :: (t → t → t) → (t → t → t) → (BinOp, Dup (Dup t)) → Dup t
binaux ◇ □ (op, ((e1, d1), (e2, d2))) = case op of
  Sum      → (e1 ◇ e2, d1 ◇ d2)
  Product  → (e1 □ e2, (e1 □ d2) ◇ (d1 □ e2))
unaux :: (a → b) → (b → a → b) → (a → b) → (UnOp, Dup a) → Dup b
unaux ⊖ ⊗ ⊔ (op, (e, d)) = case op of
  Negate → (⊖ e, ⊖ d)
  E      → (⊔ e, (⊔ e) ⊗ d)
```

Agora podemos escrever:

```
sd_gen :: Floating a ⇒ () ∥ a ∥ Bin (Dup (ExpAr a)) ∥ Un (Dup (ExpAr a)) → Dup (ExpAr a)
sd_gen = f ∥ g ∥ h ∥ k where
  f = (X, N 1)
  g a = (N a, N 0)
  h = binaux (Bin Sum) (Bin Product)
  k = unaux (Un Negate) (Bin Product) (Un E)
ad_gen :: Floating a ⇒ a → () ∥ a ∥ (BinOp, Dup (Dup a)) ∥ (UnOp, Dup a) → Dup a
ad_gen x = f ∥ g ∥ h ∥ k where
  f = (x, 1)
  g a = (a, 0)
  h = binaux (+) (*)
  k = unaux negate (*) expd
```

## E.2 Problema 3

É interessante ver que podemos ver *calcLine* como um hilomorfismo.

A ideia que levou a isso parte da definição alternativa *calcLine* = *zipWithM linear1d*.

Sabemos que:

```
zipWithM :: (Applicative m) ⇒ (a → b → m c) → [a] → [b] → m [c]
zipWithM f xs ys = sequenceA (zipWith f xs ys)
```

Percebi que podia escrever uma função (*zip*):

```
zip' :: [a] × [b] → [a × b]
zip' = [(outA* × B*)][]
```

Desde que transforme os pares de lista de uma forma que respeite o funcionamento de *zipWith* que será descrito em seguida dessa definição:

```
outA* × B* :: [a] × [b] → () + ((a × b) × ([a] × [b]))
outA* × B* = λ case
  ([], -)      → i1 ()
  (-, [])      → i1 ()
  (a : as, b : bs) → i2 ((a, b), (as, bs))
```

*zipWith* pega uma função (de aridade 2), por exemplo, *f*, e duas listas (digamos *a* e *b*) e devolve uma lista (digamos *c*) onde *c*[*i*] = *f*(*a*[*i*], *b*[*i*]) para todo  $0 \leq i \leq \min(\text{length } a, \text{length } b)$ .

Ora, então posso pegar uma função curried e pegar uma par de listas. Transformo o par de listas numa lista de pares com  $out_{A^* \times B^*}$  e aplico a função argumento em cada um dos pares. Logo tenho a seguinte definição:

$$\begin{aligned} zipWith' &:: ((a \times b) \rightarrow c) \rightarrow ([a] \times [b]) \rightarrow [c] \\ zipWith' f &= T_{[]} f \cdot zip' \end{aligned}$$

A próxima etapa é baseada nas seguintes definições

$$\begin{aligned} sequenceA &:: Applicative f \Rightarrow t (f a) \rightarrow f (t a) \\ sequenceA &= traverse id \\ traverse &:: Applicative f \Rightarrow (a \rightarrow f b) \rightarrow t a \rightarrow f (t b) \\ traverse f &= sequenceA \cdot T_{[]} f \end{aligned}$$

Ora, vamos ver como *traverse* é definido para listas

$$\begin{aligned} \text{instance Traversable [] where} \\ traverse f &= foldr cons\_f (pure []) \\ \text{where } cons\_f x ys &= liftA2 (\cdot) (f x) ys \end{aligned}$$

Vou criar um *sequenceA'* (será uma versão menos genérica de *sequenceA* uma vez que estamos sendo específicos no trabalho com listas).

$$\begin{aligned} sequenceA &= traverse id = foldr cons\_f (pure []) \text{ where} \\ cons\_f x ys &= liftA2 (\cdot) x ys \end{aligned}$$

Já fizemos o catamorfismo para *foldr* nas aulas:

$$\begin{aligned} foldrC &:: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b \\ foldrC f u &= ([\hat{u}, \hat{f}])_{[]} \end{aligned}$$

Então temos:

$$\begin{aligned} sequenceA' &:: Applicative f \Rightarrow [f a] \rightarrow f [a] \\ sequenceA' &= ([\hat{b}, \hat{g}])_{[]} \text{ where} \\ b &= (pure []) \\ g x ys &= liftA2 (\cdot) x ys \end{aligned}$$

Sabemos que, em *Applicative*  $((\rightarrow) r)$ ,  $\text{pure} = \text{const}$  e  $\text{liftA2 } q f g x = q (f x) (g x)$ . Logo:

$$\begin{aligned} sequenceA' &:: [a \rightarrow b] \rightarrow a \rightarrow [b] \\ sequenceA' &= ([\hat{b}, \hat{g}])_{[]} \text{ where} \\ b &= [] \\ g x ys &= (\lambda z \rightarrow x z : ys z) \end{aligned}$$

Lembre que *zipWithM'* como vimos recebia duas listas. No nosso caso essas duas listas (digamos *xs* e *ys*) estão em um só argumento  $t = (xs, ys)$ . A seguir seguem uma série de equivalências. Dentro de cada  $\{\}$  está uma explicação/justificativa do que foi feito de um passo para outro.

$$\begin{aligned} zipWithM' f t &= sequenceA' (zipWith' f t) \\ \{ (\cdot) f g &= \lambda x \rightarrow f (g x) \} \\ zipWithM' f &= sequenceA' \cdot zipWith' f \\ \{ \text{Def-}zipWith' \} \\ zipWithM' f &= sequenceA' \cdot (T_{[]} f \cdot zip') \\ \{ \text{Assoc-comp} \} \\ zipWithM' f &= (sequenceA' \cdot T_{[]} f) \cdot zip' \\ \{ \text{Def-}sequenceA' \} \\ zipWithM' f &= ([\hat{[]}, \hat{g}])_{[]} \cdot T_{[]} f \cdot zip' \text{ where } g x ys = (\lambda z \rightarrow x z : ys z) \end{aligned}$$

$\{ \text{Absorção-cata} \}$   
 $zipWithM' f = (\llbracket \llbracket \_, \hat{g} \rrbracket \cdot B_{\square}(f, id) \rrbracket_{\square} \cdot zip' \textbf{ where } g \ x \ ys = (\lambda z \rightarrow x \ z : ys \ z)$   
 $\{ \text{Def-baseList} \}$   
 $zipWithM' f = (\llbracket \llbracket \_, \hat{g} \rrbracket \cdot (id + f \times id) \rrbracket_{\square} \cdot zip' \textbf{ where } g \ x \ ys = (\lambda z \rightarrow x \ z : ys \ z)$   
 $\{ \text{Absorção-+; Natural-const} \}$   
 $zipWithM' f = (\llbracket \llbracket \_, \hat{g} \cdot (f \times id) \rrbracket \rrbracket_{\square} \cdot zip' \textbf{ where } g \ x \ ys = (\lambda z \rightarrow x \ z : ys \ z)$   
 $\{ (\cdot) \ f \ g = \lambda x \rightarrow f \ (g \ x) \}$   
 $zipWithM' f = (\llbracket \llbracket \_, \lambda(a, b) \rightarrow \hat{g} \ ((f \times id) \ (a, b)) \rrbracket \rrbracket_{\square} \cdot zip' \textbf{ where } g \ x \ ys = (\lambda z \rightarrow x \ z : ys \ z)$   
 $\{ \text{Def-} \times \}$   
 $zipWithM' f = (\llbracket \llbracket \_, \lambda(a, b) \rightarrow g \ (f \ a, b) \rrbracket \rrbracket_{\square} \cdot zip' \textbf{ where } g \ (x, ys) = (\lambda z \rightarrow x \ z : ys \ z)$   
 $\{ \text{Deixe que } h = (\lambda(a, b) \rightarrow g \ (f \ a, b)); \text{Notação-}\lambda \}$   
 $zipWithM' f = (\llbracket \llbracket \_, h \rrbracket \rrbracket_{\square} \cdot zip' \textbf{ where } h \ (a, b) = (\lambda z \rightarrow (f \ a) \ z : b \ z)$   
 $\{ \text{Def-} zip'; \text{Notação-}\lambda \}$   
 $zipWithM' f = (\llbracket \llbracket \_, h \rrbracket \rrbracket_{\square} \cdot \llbracket out_{A^* \times B^*} \rrbracket_{\square} \textbf{ where } h \ (a, b) \ z = (f \ a) \ z : b \ z$   
 $\{ \text{catamorfismo após anamorfismo é um hilomorfismo} \}$   
 $zipWithM' f = \llbracket \llbracket \llbracket \_, h \rrbracket, out_{A^* \times B^*} \rrbracket_{\square} \textbf{ where } h \ (a, b) \ z = (f \ a) \ z : b \ z$   
 $zipWithM' f = \llbracket \llbracket \llbracket \_, h \rrbracket, out_{A^* \times B^*} \rrbracket_{\square} \textbf{ where } h \ (a, b) = cons \cdot \langle f \ a, b \rangle$

Portanto, lembrando que  $calcLine = zipWithM \ linear1d$  e tendo em mente que  $calcLine :: [\mathbb{Q}] \rightarrow [\mathbb{Q}] \rightarrow Float \rightarrow [\mathbb{Q}]$ , mas  $zipWithM' :: ((a \times b) \rightarrow c \rightarrow d) \rightarrow ([a] \times [b]) \rightarrow c \rightarrow [d]$

$calcLine = \widehat{zipWithM' \ linear1d}$   
 $\{ \text{Def-} zipWithM' \}$   
 $calcLine = \llbracket \llbracket \llbracket \_, h \rrbracket, out_{A^* \times B^*} \rrbracket_{\square} \textbf{ where } h \ (a, b) = cons \cdot \langle \widehat{linear1d} \ a, b \rangle$

## E.2.1 Notação case

Notação lambda simplifica expressão das funções:

$outSL :: [a] \rightarrow () \coprod a \coprod [a]$   
 $outSL = \lambda \textbf{case}$   
 $\quad [] \rightarrow i_1 \ ()$   
 $\quad [a] \rightarrow i_2 \ (i_1 \ a)$   
 $\quad l \rightarrow i_2 \ (i_2 \ l)$

A notação case que acho mais simples mas requer uma extensão não usada no trabalho.

$out_{\text{Castel}} :: \text{Castel } a \rightarrow () \coprod a \coprod \text{Castel } a \times \text{Castel } a$   
 $out_{\text{Castel}} = \lambda \textbf{case}$   
 $\quad Empty \rightarrow i_1 \ ()$   
 $\quad Single \ a \rightarrow i_2 \ (i_1 \ a)$   
 $\quad InitTail \ (e, d) \rightarrow i_2 \ (i_2 \ (e, d))$

## E.3 Problema 4

### E.3.1 Notação case

Notação lambda facilita legibilidade:

$out_{\text{NList}} :: [a] \rightarrow a \coprod a \times [a]$   
 $out_{\text{NList}} = \lambda \textbf{case}$   
 $\quad [a] \rightarrow i_1 \ a$   
 $\quad (a : x) \rightarrow i_2 \ (a, x)$

# Índice

L<sup>A</sup>T<sub>E</sub>X, [1](#)

**bibtex**, [2](#)

**lhs2TeX**, [1](#)

**makeindex**, [2](#)

Combinador “pointfree”

*cata*, [8](#), [9](#), [14–19](#), [22](#), [23](#)

*either*, [3](#), [8](#), [16–19](#), [22](#), [23](#)

Curvas de Bézier, [6](#), [7](#)

Cálculo de Programas, [1](#), [2](#), [5](#)

    Material Pedagógico, [1](#)

        BTree.hs, [8](#)

        Cp.hs, [8](#)

        LTree.hs, [8](#), [19](#)

        Nat.hs, [8](#)

Deep Learning), [3](#)

DSL (linguagem específica para domínio), [3](#)

F#, [8](#), [19](#)

Functor, [5](#), [11](#)

Função

$\pi_1$ , [6](#), [9](#), [18](#), [19](#)

$\pi_2$ , [9](#), [13](#)

*for*, [6](#), [9](#), [16](#)

*length*, [8](#), [18](#), [19](#), [21](#)

*map*, [11](#), [12](#)

*uncurry*, [3](#), [13](#), [22](#), [23](#)

Haskell, [1](#), [2](#), [8](#)

    Gloss, [2](#), [11](#)

    interpretador

        GHCi, [2](#)

    Literate Haskell, [1](#)

    QuickCheck, [2](#)

    Stack, [2](#)

Números de Catalan, [6](#), [10](#)

Números naturais (*I*

*N*), [5](#), [6](#), [9](#), [14](#), [15](#), [17](#), [18](#)

Programação

    dinâmica, [5](#)

    literária, [1](#)

Racionais, [7](#), [8](#), [10–12](#), [16](#)

U.Minho

    Departamento de Informática, [1](#)



## Referências

- [1] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes Number 27. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.
- [2] J.N. Oliveira. *Program Design by Calculation*, 2018. Draft of textbook in preparation. viii+297 pages. Informatics Department, University of Minho.